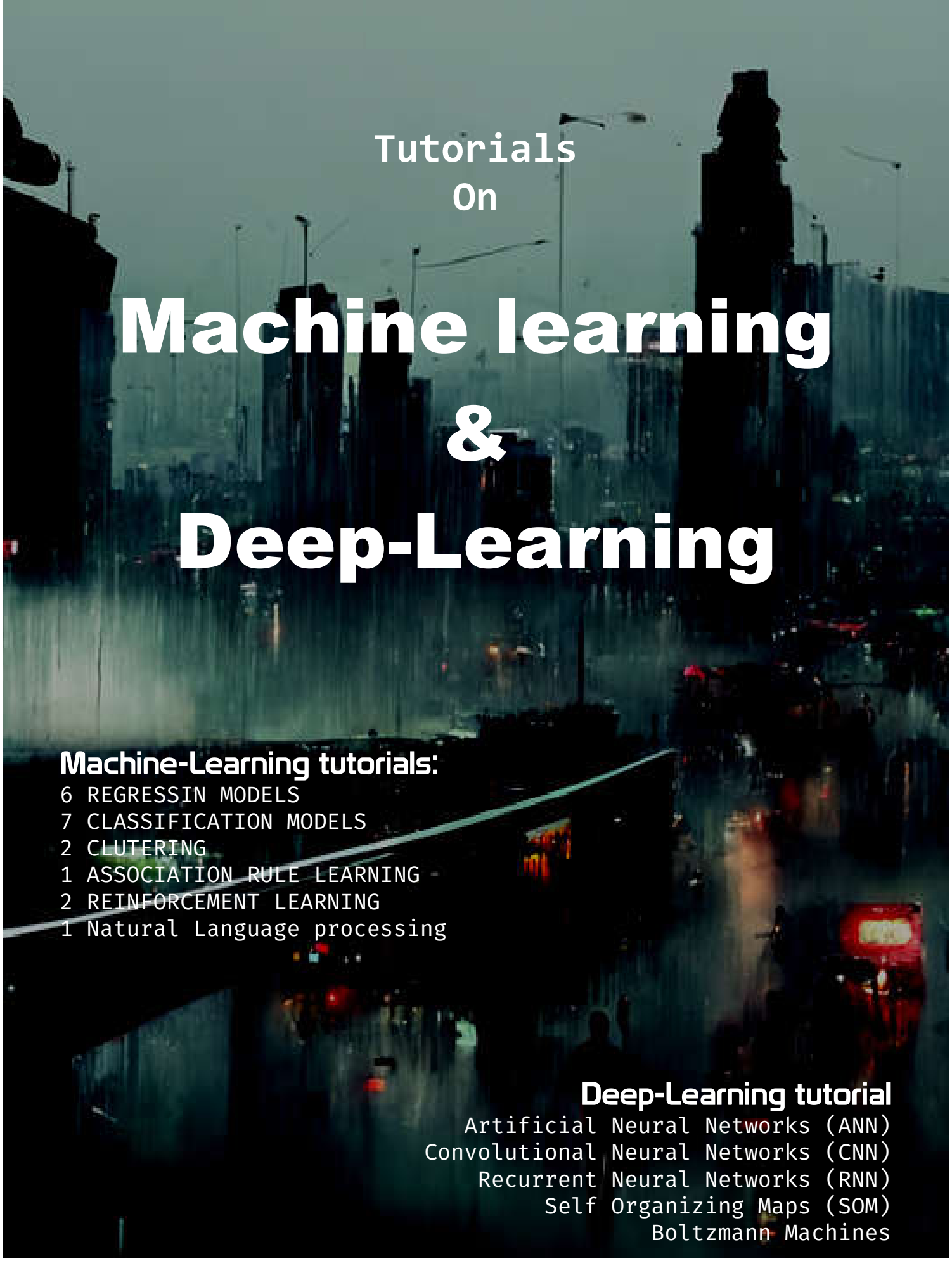


Tutorials
On

Machine learning
&
Deep-Learning

Machine-Learning tutorials:

6 REGRESSIN MODELS
7 CLASSIFICATION MODELS
2 CLUTERING
1 ASSOCIATION RULE LEARNING
2 REINFORCEMENT LEARNING
1 Natural Language processing

Deep-Learning tutorial

Artificial Neural Networks (ANN)
Convolutional Neural Networks (CNN)
Recurrent Neural Networks (RNN)
Self Organizing Maps (SOM)
Boltzmann Machines

Tutorials
on

Machine learning
&
Deep-Learning

Machine-Learning tutorials:

6 REGRESSIN MODELS
7 CLASSIFICATION MODELS
2 CLUTERING
1 ASSOCIATION RULE LEARNING
2 REINFORCEMENT LEARNING
1 Natural Language processing

Deep-Learning tutorial

Artificial Neural Networks (ANN)
Convolutional Neural Networks (CNN)
Recurrent Neural Networks (RNN)
Self Organizing Maps (SOM)
Boltzmann Machines
AutoEncoders

Tutorials on

Machine learning & Deep-Learning

Description

This Book is form the courses of ***two professional Data Scientists*** Kirill Eremenko from **SuperDataScience** and Hadelin de Ponteves from **BlueLife AI**.

You can get a quick overview on Machine Learning & Deep Learning from this book.
Also this book will be the best guide for the ***Courses*** of ***Kirill Eremenko*** and ***Hadelin de Ponteves***. It can also helpful for ***Andrew Ng's*** Machine learning course.

This book will help you learn complex ***theory***, ***algorithms*** and ***coding libraries*** in a simple way (as a beginner).
It will walk you step-by-step into the World of Machine Learning. With every tutorial you will develop new skills and improve your understanding of this challenging yet lucrative sub-field of Data Science.

Requirements?

Just some high school mathematics level.

What you can get from this Book?

- Master Machine Learning on ***Python***
- Have a great intuition of ***many Machine Learning models***
- Make ***accurate predictions***
- Make ***powerful analysis***
- Make ***robust Machine Learning models***
- Create strong added value to your business
- Use Machine Learning for personal purpose
- Handle specific topics like ***Reinforcement*** Learning, ***NLP*** and ***Deep Learning***
- Handle ***advanced*** techniques like ***Dimensionality Reduction***
- Know which ***Machine Learning model*** to choose for each ***type*** of ***problem***
- Build an ***army of powerful Machine Learning models*** and know how to combine them to solve any problem

What is the target audience?

- Anyone ***interested*** in ***Machine Learning***.
- Students who have at least ***high school knowledge*** in ***math*** and who want to ***start learning Machine Learning***.
- Any ***intermediate level*** people who know the ***basics*** of ***machine learning***, including the ***classical algorithms*** like ***linear regression*** or ***logistic regression***, but who want to learn more about it and explore all the different fields of Machine Learning.
- Any people who are not that ***comfortable*** with ***coding*** but who are interested in ***Machine Learning*** and want to ***apply it easily*** on ***datasets***.
  - Any students in *college* who want to start a *career* in *Data Science*.
  - Any *data analysts* who want to *level* up in *Machine Learning*.
  - Any people who are *not satisfied* with their *job* and who want to become a *Data Scientist*.
  - Any people who want to *create* added value to their *business* by using *powerful Machine Learning tools*.

After reading/using this Book you can dive deeper in Machine-Learning or Deep-Learning.

0.1 Machine Learning TOPICS

[0] Data Preprocessing

Data Preprocessing in Python

[1] Regression

i. Simple Linear Regression
ii. Multiple Linear Regression
iii. Polynomial Regression
iv. Support Vector Regression (SVR)
v. Decision Tree Regression
vi. Random Forest Regression
vii. Evaluating Regression Models Performance

[2] Classification

i. Logistic Regression
ii. K-Nearest Neighbors (K-NN)
iii. Support Vector Machine (SVM)
iv. Kernel SVM
v. Naive Bayes
vi. Decision Tree Classification
vii. Random Forest Classification
viii. Evaluating Classification Models Performance

[3] Clustering

i. K-Means Clustering
ii. Hierarchical Clustering

[4] Association Rule Learning

i. Apriori
ii. Eclat

[5] Reinforcement Learning -

i. Upper Confidence Bound (UCB)
ii. Thompson Sampling

[6] Natural Language Processing

0.2 Deep Learning TOPICS

[7] Artificial Neural Networks (ANN)

i. ANN Intuition
ii. Building an ANN

[8] Convolutional Neural Networks (CNN)

i. CNN Intuition
ii. Building a CNN

[9] Dimensionality Reduction

i. Principal Component Analysis (PCA)
ii. Linear Discriminant Analysis (LDA)
iii. Kernel PCA

[10] Model Selection & Boosting

i. Model Selection
ii. XG Boost
iii. Bonus Lectures

[11] Recurrent Neural Networks (RNN)

i. RNN Intuition Building a RNN
ii. Evaluating and Improving the RNN

[12] Self Organizing Maps (SOM)

i. SOMs Intuition Building a SOM
ii. Mega Case Study

[13] Boltzmann Machines

i. Boltzmann Machine Intuition
ii. Building a Boltzmann Machine

[14] AutoEncoders

i. AutoEncoders Intuition
ii. Building an AutoEncoder

Chapter 0 : Part 1

Usage of ML

Get exited, Inspired, Amount of data and how to access them.

0.1 Applications of Machine Learning

[1] Facial Recognition: Eg: facebook.

[2] Games without joystick. Recognize action, movement for game control.

☞ ***Random forest*** is udsed.

[3] In VR movement. ML algorithm detecting your movement.

[4] Text to speech, voice recognition, voice typing

[5] Robot companion.

☞ ***Reinforcement learning*** used.

[6] ***Recommender system, or a recommendation system:*** ***Amazon***, ***Netflix***, ***audible***. ***Facebook*** ads, ***Youtube*** video suggestion.

[7] Medicines

[8] In space-satellite for recognize certain aria of earth. Image processing.

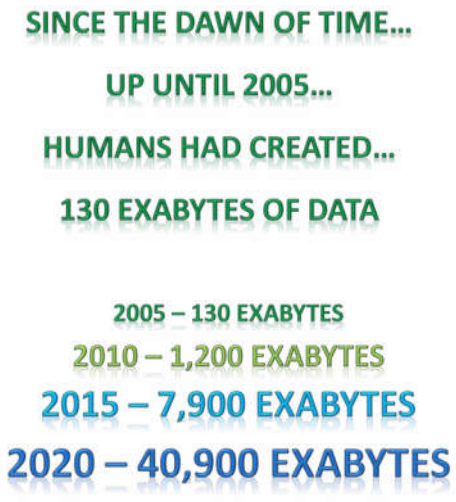
[9] Explore other planet, Space-Robot.

0.2 ML is the Future

❒ ***Data is everywhere:***

Data exhaust.

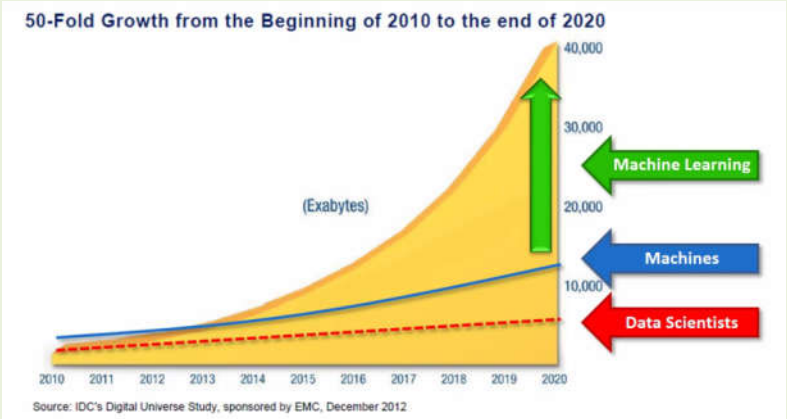




1 letter = 1byte
1 page = 1 kilo byte, 1000 letter
1 book = 1000 page, 1 mega byte
1 DNA, Gnome = 1000 books, 1 giga byte
1 persons life (80 year) as video(hd) = 1 tb
1 Amazon Rainforest 700, 000 000 000 trees, as paper and fill those page = 1 petabyte
1 Exabyte = 1000 petabyte as whole world.

❒ ***Exponential growth of data:***

We need ML to explore **that** ***high amount of data***. Only **ML-Algorithms** can explore these data.



Chapter 0 : Part 2

ML : What? Why? How?

Understanding the concepts.

0.3 What Is Machine Learning?

Machine learning is a **data analytics technique** that teaches computers to do what comes naturally to humans and animals: ***learn from experience***.

❒ ***Machine Learning:*** **Machine learning algorithms** use ***computational methods*** to ***"learn" information directly from data*** without relying on a ***predetermined equation*** as a model. The algorithms adaptively ***improve their performance*** as the ***number of samples*** available for learning ***increases***. ***Deep learning*** is a ***specialized*** form of ***machine learning***.

0.4 Why Machine Learning Matters

With the rise in big data, machine learning has become a key technique for solving problems in areas, such as:

i). ***Computational finance***, for credit scoring and algorithmic trading
ii). ***Image processing*** and ***computer vision***, for face recognition, motion detection, and object detection
iii). ***Computational biology***, for tumor detection, drug discovery, and DNA sequencing
iv). ***Energy production***, for price and load forecasting
v). ***Automotive***, aerospace, and manufacturing, for predictive maintenance
vi). ***Natural language*** processing, for ***voice recognition*** applications

☞ ***Machine learning algorithms*** find ***natural patterns*** in **data** that ***generate insight*** and help you make ***better decisions*** and ***predictions***. They are used every day to make ***critical decisions*** in *medical diagnosis, stock trading, energy load forecasting*, and more.
For example, media sites rely on machine learning to sift through millions of options to give you song or movie ***recommendations***. Retailers use it to gain insight into their customers' purchasing behavior.

❒ ***When Should You Use ML:*** Consider using machine learning when you have a complex task or problem ***involving a large amount of data*** and ***lots of variables***, but ***no existing formula or equation***. For example, machine learning is a good option if you need to handle situations like these:

☞ ***Hand-written rules*** and ***equations*** are ***too complex***—as in face recognition and speech recognition.
☞ The ***rules*** of a task are ***constantly changing***—as in fraud detection from transaction records.
☞ The ***nature*** of the ***data*** keeps ***changing***, and the program needs to adapt—as in ***automated trading***, energy demand forecasting, and predicting shopping trends.

0.5 How Machine Learning Works

Walk through the three types of machine learning (***clustering***, ***classification***, and ***regression***). Machine learning uses two types of techniques:

[1] ***Supervised learning:*** which **trains** a model on **known input** and output data so that it can **predict future** outputs,
[2] ***Unsupervised learning:*** which **finds hidden patterns** or **intrinsic structures** in input data.

Machine learning		
SUPERVISED LEARNING Develop predictive model based on both input and output data		***UNSUPERVISED*** LEARNING Group and interpret data based only on input data
CLASSIFICATION	***REGRESSION***	***CLUSTERING***

0.6 Supervised Learning

❒ ***Supervised Learning:*** Supervised machine learning builds a model that makes ***predictions*** based on ***evidence*** in the ***presence of uncertainty***.

☞ A ***supervised learning algorithm*** takes a ***known set of input data*** and ***known responses*** to the data (output) and trains a model to generate ***reasonable predictions*** for the response to ***new data***.
☞ Use ***supervised learning*** if you have known data ***for*** the ***output*** you are trying to ***predict***.

❒ ***Techniques for SUPERVISED LEARNING:*** Supervised learning uses ***CLASSIFICATION*** and ***REGRESSION*** techniques to develop machine learning models.

0.6.1 Classification

❒ ***Classification:*** Classification techniques *predict discrete responses*—for example, whether an *email is genuine or spam*, or *whether a tumor is cancerous or benign*.

☞ Classification models ***classify input data*** into ***categories***. Typical applications include *medical imaging*, *speech recognition*, and *credit scoring*.

☞ Use classification if your data can be **tagged**, **categorized**, or **separated** into ***specific groups*** or ***classes***. For example, applications for ***hand-writing recognition*** use ***classification*** to recognize letters and numbers.

☯ In image processing and computer vision, ***Unsupervised Pattern Recognition*** techniques are used for **object detection** and **image segmentation**.

🛠 ***Common CLASSIFICATION Algorithms:*** Common algorithms for performing classification include

[1] ***Support Vector Machine*** (svm),

[2] Boosted and Bagged ***DECISION TREES***,

[3] ***K-Nearest Neighbor***,

[4] ***Naïve bayes***,

[5] ***Discriminant analysis***,

[6] ***Logistic Regression***, and

[7] ***Neural Networks***.

0.6.2 Regression

Regression: Regression techniques *predict continuous responses*—for example, ***changes in temperature*** or ***fluctuations in power demand***. Typical applications include electricity load forecasting and algorithmic trading.

Use regression techniques if you are working with a ***data range*** or if the nature of your ***response is a real number***, such as ***temperature*** or the ***time*** until failure for a piece of equipment.

🛠 ***Common REGRESSION Algorithms:*** Common regression algorithms include

[1] ***Linear*** model,

[2] ***Nonlinear*** model,

[3] ***Regularization***,

[4] ***Stepwise Regression***,

[5] ***Boosted*** and ***Bagged Decision Trees***,

[6] ***Neural Networks***, and

[7] ***Adaptive Neuro-Fuzzy Learning***.

☯ ***Example of supervised learning:***

☯ ***Using Supervised Learning to Predict Heart Attacks:*** Suppose clinicians want to predict whether someone will have a heart attack within a year. They have data on ***previous patients***, including ***age***, ***weight***, ***height***, and ***blood pressure***. They know whether the previous patients had heart attacks within a year. So the problem is combining the existing data into a model that can predict whether a new person will have a heart attack within a year.

0.7 Unsupervised Learning

Unsupervised learning finds ***hidden patterns*** or ***intrinsic structures*** in data. It is used to ***draw inferences from datasets*** consisting of input data without labeled responses.

Clustering

Clustering is the most common **unsupervised learning technique**. It is used for exploratory data analysis to find ***hidden patterns*** or ***groupings*** in data. Applications for cluster analysis include ***Gene Sequence*** analysis, ***Market Research***, and ***Object Recognition***.

☯ ***Example:***

☯ For example, if a cell phone company wants ***optimize*** the ***locations*** where they build ***cell phone towers***, they can use machine learning to ***estimate*** the ***number of clusters of people*** relying on their towers. A phone can only talk to one tower at a time, so the team uses ***Clustering Algorithms*** to design the ***best placement of cell towers*** to optimize ***Signal Reception*** for ***groups***, or ***clusters***, of their ***customers***.

🛠 ***Common CLUSTERING algorithms:***

[1] ***K-Means*** and ***K-Medoids***,

[2] ***Hierarchical Clustering***,

[3] ***Gaussian Mixture*** models,

[4] ***Hidden Markov Models***,

[5] ***Self-Organizing Maps***,

[6] ***Fuzzy C-Means Clustering***, and

[7] ***Subtractive Clustering***.

0.8 Which Machine Learning Algorithm to Use?

Choosing the right algorithm can seem overwhelming—there are dozens of supervised and unsupervised machine learning algorithms, and each takes a different approach to learning.

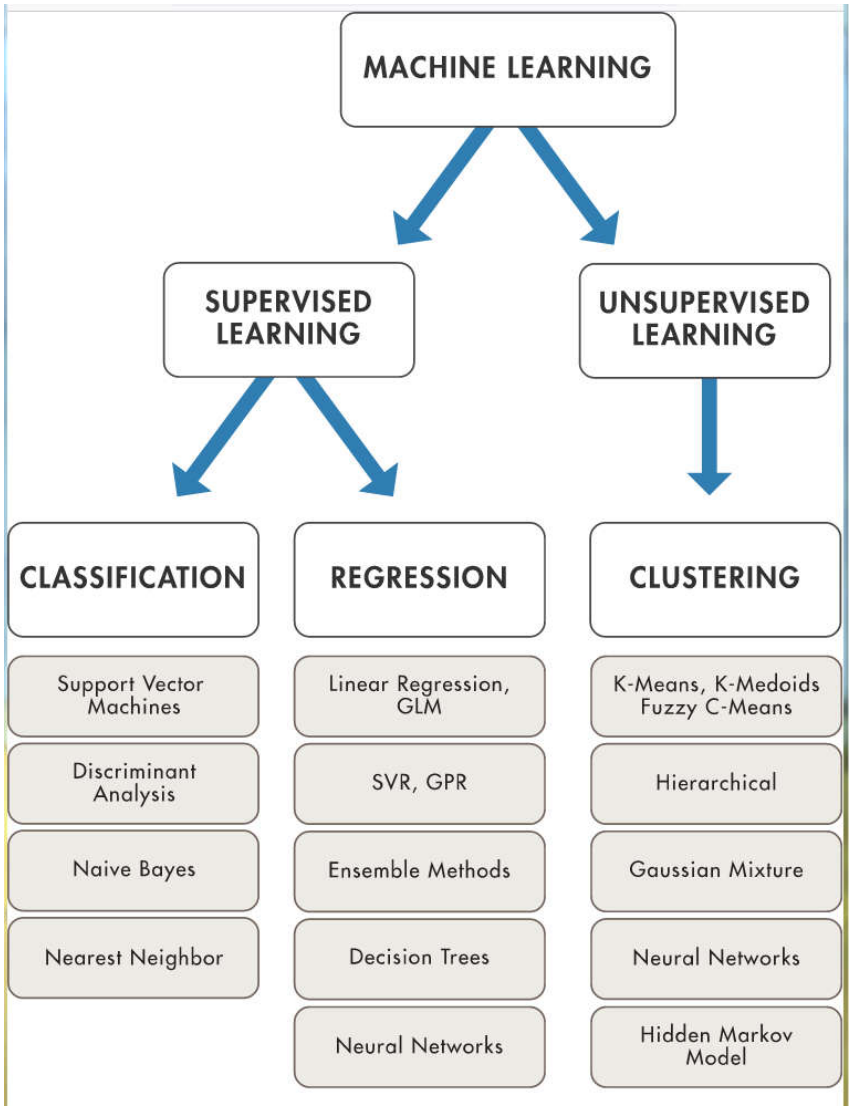
☞ There is no best method or one size fits all. Finding the right algorithm is partly just ***trial*** and ***error***—even highly experienced data scientists can't tell whether an algorithm will work without trying it out.

☞ Algorithm selection also ***depends*** on the ***size and type of data*** you're working with, the insights you want to get from the data, and how those insights will be used.

❒ Here are some ***guidelines*** on choosing between ***Supervised*** and ***Unsupervised*** machine learning:

☞ Choose ***Supervised Learning*** if you need to train a model to make a ***Prediction***—for example, the future value of a continuous variable, such as temperature or a stock price, or a classification—for example, identify makes of cars from webcam video footage.

☞ Choose ***unsupervised*** learning if you need to ***explore your data*** and want to ***train*** a ***model*** to find a good ***internal representation***, such as ***splitting*** *data up into* ***clusters***.



Chapter 1

Data Preprocessing

1.1 Python and R installation

[1] Install python with Anaconda(IDE).
[2] Install R with R-studio(IDE).

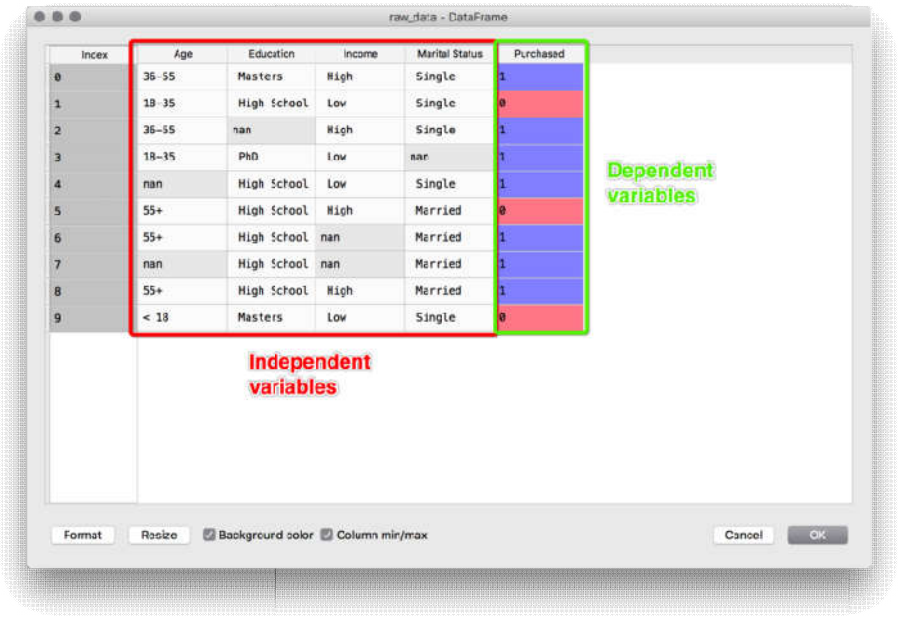
1.2 Extract Given data sets and Codes and file tree

[1] Extract All examples.
[2] Extract All codes and Datasets.
[3] Extract the given file tree.

1.3 INDEPENDENT and DEPENDENT Variables in ML

Independent variables (also referred to as Features) are the input for a process that is being analyzes. ***Dependent*** variables are the output of the process.

- For example, in the below data set, the ***independent*** **variables** are the **input** of the **purchasing process** being analyzed. The result (whether a user ***purchased*** or ***not***) is the ***dependent*** **variable**.
- Using ***Independent*** variables, we PREDICT ***Dependent*** variables.



1.4 Importing the Libraries

❒ ***Three essential libraries:*** Use this kind of command to install the libraries: Eh: to install ***matplotlib***,

```
pip install matplotlib
```

```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
```

[1] ***NumPy:*** Contains mathematical tools. Since of course ***machine learning models*** are based on ***mathematics*** then we will absolutely need ***NumPy*** from time to time to do them but ***not all the time*** you'll see.

[2] ***matplotlib:*** *Matplotlib* is a comprehensive library for creating static, animated, and interactive visualizations in Python. We're gonna use the module ***pyplot*** of ***matplotlib*** library.

- ***module matplotlib.pyplot:*** ***matplotlib.pyplot*** is a state-based interface to ***matplotlib***. It provides a ***MATLAB-like*** way of plotting.
  - ✓ *pyplot* is mainly intended for *interactive plots* and simple cases of *programmatic plot generation*.
  - ✓ The object-oriented API is recommended for more complex plots.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 5, 0.1)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```

[3] ***pandas:*** Used for import and manage datasets. pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

1.5 Importing the Dataset

❒ ***Use the ide Spider:*** ***Spider*** in ***Anaconda***. Enable ***panes***: Editor, ipython console, Variable explorer, object inspector (aka Help).

☞ ***Ctrl+enter*** to execute.

❒ ***Pandas intro:*** Recall **100-days-of-code**.

☞ ***DataFrame.iloc***

Purely integer-location based indexing for selection by position. ***.iloc[]*** is primarily integer position based (from ***0*** to ***length-1*** of the axis), but may also be used with a boolean array.

Indexing just the rows:

With a scalar integer:	`df.iloc[0]`
With a list of integers:	`df.iloc[[0]]`
	`df.iloc[[0, 1]]`
With a slice object:	`df.iloc[:3]`

Indexing both axes: You can mix the indexer types for the index and columns. Use ***:*** to select the ***entire axis***.

With scalar integers:	`df.iloc[0, 1]`
With lists of integers.	`df.iloc[[0, 2], [1, 3]]`
With slice objects:	`df.iloc[1:3, 0:3]`

❒ ***Matrix of features and dependent variable vector:***

☞ ***Matrix of features:*** ***Matrix*** of independent variables. "**:**" is for slicing.

```
x = dataset.iloc[:, :-1]
```

- ➢ "**:-1**" is to all columns excluding last column
- ➢ "**:**" to select all rows

☞ ***Dependent variable vector:*** ***Vector*** of last column as dependent variable.

```
y = dataset.iloc[:, 3]
```

```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# x = np.arange(0, 5, 0.1)
# y = np.sin(x)
# plt.plot(x, y)
# plt.show()
```

```
# importing the dataset
dataset = pd.read_csv("Data.csv")
print(dataset)
# ":-1" is to all columns excluding last column
# ":" to select all rows
x = dataset.iloc[:, :-1]
print(x)
y = dataset.iloc[:, 3]
print(y)

# python prcts_dt_prep.py
```

1.6 Missing Data

Install library ***sklearn***. Import ***imputer*** from ***preprocessing*** module.

```
pip install -U scikit-learn
```

❒ ***Use Mean:*** Eliminate ***NaN***.

```
class sklearn.preprocessing.Imputer(missing_values='NaN', strategy='mean', axis=0, verbose=0, copy=True)
```

```
# ----------- For older version -------------
# from sklearn.preprocessing import Imputer
# imptr = Imputer()
```

☞ `***SimpleImputer***` replaces the previous `***sklearn.preprocessing.Imputer***` estimator which is now removed.

```
class SimpleImputer(self, missing_values=np.nan, strategy="mean", fill_value=None, verbose=0, copy
=True, add_indicator=False)
```

Parameters

missing_values : number, string, np.nan (default) or None

The placeholder for the missing values. All occurrences of `missing_values` will be imputed. For pandas' dataf
rames with nullable integer dtypes with missing values, `missing_values` should be set to `np.nan`, since `pd
.NA` will be converted to `np.nan`.

strategy : string, default='mean'

The imputation strategy.

- If "mean", then replace missing values using the mean along each column. Can only be used with numeric data.

- If "median", then replace missing values using the median along each column. Can only be used with numeric d
ata.
- If "most_frequent", then replace missing using the most frequent value along each column. Can be used with s
trings or numeric data.
- If "constant", then replace missing values with fill_value. Can be used with strings or numeric data.

strategy="constant" for fixed value imputation.

fill_value : string or numerical value, default=None

When strategy == "constant", fill_value is used to replace all occurrences of missing_values.
If left to the default, fill_value will be 0 when imputing numerical data and "missing_value" for strings or o
bject data types.

```
# ------------------ Data Preprocessing --------------------

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# x = np.arange(0, 5, 0.1)
# y = np.sin(x)
# plt.plot(x, y)
# plt.show()


#importing the dataset
dataset = pd.read_csv("Data.csv")
print(dataset)
```

```
# ":-1" is to all columns exluding last column
# ":" to select all rows
old_x = dataset.iloc[:, :-1]
old_y = dataset.iloc[:, 3]
print("Before\n",old_x)


# Taking care of missing data
# import sklearn
# print(sklearn.__version__)

from sklearn.impute import SimpleImputer
imptr = SimpleImputer(missing_values= np.nan, strategy="mean") # "NaN" connot be used. Use np.nan
# fix the 2nd and 3rd column
impt = imptr.fit(old_x.iloc[:, 1:3])
print(impt) # it is a method, acts on feature matrix
old_x.iloc[:, 1:3] = impt.transform(old_x.iloc[:, 1:3])
print("After\n",old_x)

# ----------- For older version -------------
# from sklearn.preprocessing import Imputer
# imptr = Imputer(missing_values='NaN', strategy='mean', axis=0, verbose=0, copy=True)
```

NOTE: Slice list in python: Index ***-1*** represents the ***last*** element and ***-n*** represents the ***first*** element of the list(considering n as the length of the list). Lists can also be manipulated using negative indexes also.

```
Lst[ Initial : End : IndexJump ]
```

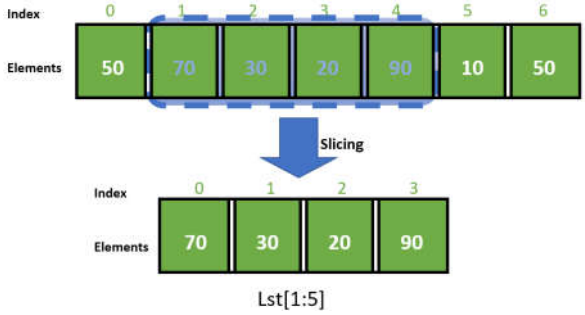
```
# List slicing

# Initialize list
Lst = [50, 70, 30, 20, 90, 10, 50]

# Display list
print(Lst[::])

# Negative indexing, reverse
print(Lst[-7::1])

# Display list portion
print(Lst[1:5])
```



NOTE: Use ***power shell*** **cd** with " " to consider it as a ***string*** with ***spaces***.

```
CD "L:\2_Code_source_1\ML_a2z_cods_dtsts\Part 1 - Data Preprocessing\Section 2 -----------
--------- Part 1 - Data Preprocessing --------------------\Python"
```

1.7 Categorical Data : Encode texts into numbers

Country	Age	Salary	Purchased
France		72000	No
Spain	27	48000	Yes
Germany	30	54000	No
Spain	38	61000	No
Germany	40		Yes
France	35	58000	Yes
Spain		52000	No
France	48	79000	Yes
Germany	50	83000	No
France	37	67000	Yes

- In this dataset we can see that we have two categorical variables, ***country*** and ***purchase***. These two variables are categorical variables because they contain categories.
  - Here the ***country*** contains three categories. It's France Spain and Germany. And the ***purchase*** variable contains two categories. ***Yes*** and ***No*** that's why they're called categorical variables.
  - But ***ML models*** works with numbers, so we need to encode these categories into ***numerical values***. Since machine learning models are based on mathematical equations you can intuitively understand that it would cause some problem if we keep the ***text*** here and the ***categorical variables*** in the equations because we would only want numbers in the equations.
  - So that's why we need to encode the ***categorical variables***. That is to encode the text that we have here into numbers.

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
label_encode_x = LabelEncoder()
old_x.iloc[:, 0] = label_encode_x.fit_transform(old_x.iloc[:, 0])
# one_hot_encode = OneHotEncoder(categories= old_x[0])
# old_x = one_hot_encode.fit_transform(old_x).toarray()
print(old_x)
```

	Country	Age	Salary
0	0	44.000000	72000.000000
1	2	27.000000	48000.000000
2	1	30.000000	54000.000000
3	2	38.000000	61000.000000
4	1	40.000000	63777.777778
5	0	35.000000	58000.000000
6	2	38.777778	52000.000000
7	0	48.000000	79000.000000
8	1	50.000000	83000.000000
9	0	37.000000	67000.000000

🛠 ***Labelencoder:*** If only two categories occurs, like yes-no or on-off we then use "labelencoder" give 0 and 1.

```
#Encode dependent vector Column
from sklearn.preprocessing import LabelEncoder
label_encode = LabelEncoder()
old_y = label_encode.fit_transform(old_y)
print(old_y)
```

- But for more than two categories, " `LabelEncoder` " returns 0, 1, 2, 3, ... This can cause error. For example ***country*** is conceded to 0, 1, 2.
- ***Problem:*** The problem is ***there is unwanted numerical-orders***. France is 0, Spain is 2 and Germany is 1. But there is no relationship order between these three countries France Germany and Spain.
  - So we want to avoid the model to have such an interpretation because that could cause some misinterpreted correlations between the ***features*** and the ***outcome*** which we want to ***predict***.

```
array([[0, 44.0, 72000.0],
       [2, 27.0, 48000.0],
       [1, 30.0, 54000.0],
       [2, 38.0, 61000.0],
       [1, 40.0, 63777.77777777778],
       [0, 35.0, 58000.0],
       [2, 38.77777777777778, 52000.0],
       [0, 48.0, 79000.0],
       [1, 50.0, 83000.0],
       [0, 37.0, 67000.0]], dtype=object)
```

🛠 ***DUMMY-Encoding (use `OneHotEncoder`) Dummy variables:*** Don't use LabelEncoder to transform categorical into numerical, just use ***`OneHotEncoder`***.

- Using ***LabelEncoder*** introduces a problem of ***data ordering***, when your categorical data gets encoded into let's say 1,2,3 etc. and for model this is a clear ordering 1 < 2 < 3, however in reality with **categorical features this is not true**. For this reason we introduce ***three dummy variables*** for these ***counties***, 1 represents the occurrence of the county in a row.
- So ***Country*** column into three columns, because there are actually three different classes in this country column you know three different categories (countries).
- If there were for example ***five countries*** here we would turn this column into ***five columns*** and ***`OneHotEncoder`*** consists of creating binary vectors for each of the countries.

☠ ***Legacy codes not supported anymore:***

Legacy Code

```
# Encoding the Independent Variable
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X = LabelEncoder()
X[:, 0] = labelencoder_X.fit_transform(X[:, 0])
onehotencoder = OneHotEncoder(categorical_features = [0])
X = onehotencoder.fit_transform(X).toarray()
```

For Newer Version

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

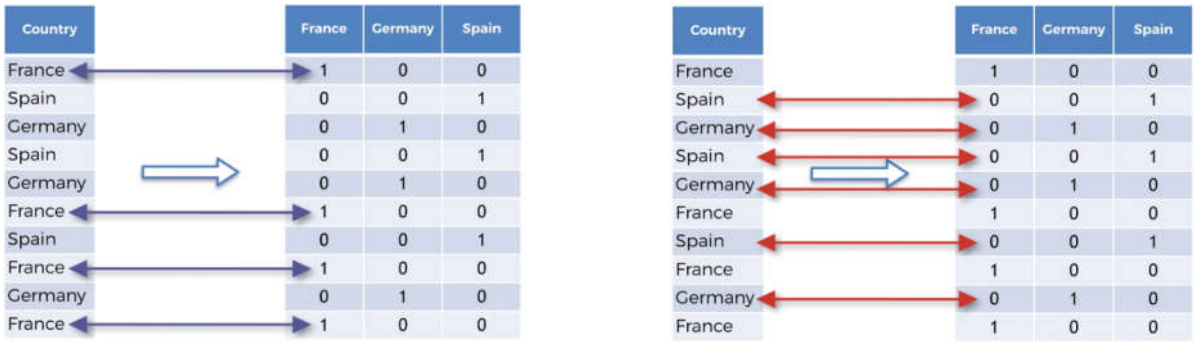
#Encode Country Column
ct = ColumnTransformer(transformers = [("encoding", OneHotEncoder(), [0])], remainder = 'passthrough')
# remainder = 'passthrough' for remaining columns to be unchanged
old_x = ct.fit_transform(old_x) # convert this output to NumPy array
print(old_x)
old_x = np.array(old_x)
print(old_x)
```

Dummy Encoding

Country
France
Spain
Germany
Spain
Germany
France
Spain
France
Germany
France

⇒

France	Germany	Spain
1	0	0
0	0	1
0	1	0
0	0	1
0	1	0
1	0	0
0	0	1
1	0	0
0	1	0
1	0	0

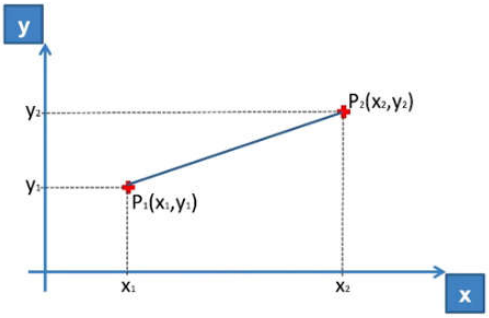


1.8 Feature Scaling

Why Feature Scaling ?

Let's just focus on the age and the salary. You notice that the variables are not on the same scale because the age are going from 27 to 50. And the salaries going from 40 K to like 90 K. Hence this age variable in the salary variable don't have the same scale.

☞ This will cause some issues in your machinery models. Because a lot of ***ML model*** are based on the Euclidean distance. The Euclidean distance between two points is the square root of the sum of the square coordinates.



Euclidean Distance between P_1 and $P_2 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

	A	B	C	D	E	F	G
	Country	**Age**	**Salary**	**Purchased**			
	France	44	72000	No			
	Spain	27	48000	Yes			
	Germany	30	54000	No			
	Spain	38	61000	No		31000	961000000
	Germany	40	63777.77778	Yes			
	France	35	58000	Yes		21	441
	Spain	38.77777778	52000	No			
	France	48	79000	Yes			
	Germany	50	83000	No			
	France	37	67000	Yes			

- Now in our dataset consider the 9th and the third rows. Salary difference is 31000 and age difference is 21. Their squares are 96100000 and 441.
  - ✓ So you can see very clearly how this square difference 96100000 dominates this square difference 441.
  - ✓ And that's because these two variables are not on the same scale.
  - ✓ So that's why we absolutely need to put the variables in the same scale. And going to have values in the same range.

- ***fit and transform for train-data only transform test-data:*** We're calling the method fit and transform for train-data but we will only transform the test set because it will automatically fitted during train-data (it's already fitted to the training set).

```
# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
```

- ***Do we need to fit and transform the DUMMY VARIABLES:*** It depends on the context. It depends on how much you want to keep interpretation in your models. Because if we scale this it will be good because everything will be on the same scale we will be happy with that and it will be good for our predictions but who will lose the interpretation of knowing which observations belongs to which country etc..
  - ☞ So as you want it won't break your model if you don't scale the dummy variables because there will be actually on the same scale as the future scales.

- ***Do we need to apply features scaling to the DEPENDENT VARIABLE VECTOR:*** In our case it is a categorical variable because it's taking only two values 0 and one.

We don't need to do it because this is a classification problem with a category called dependent variable.
But you will see that for regression when the **dependent variable will take a huge range of values**. We will **need to apply feature scaling t**o the **dependent variable** as well.

- ***Feature Scaling for other models:*** Even if sometimes machine models are not based on ***Euclidean distances*** we will still need to do ***features scaling*** because the ***ALGORITHM*** will ***converge much faster***. That will be the case for ***decision trees***. Decision trees are ***not based on Euclidean distances*** but you will see that we will need to do ***feature scaling*** because if we don't do it they will run for a very long time.

```
# ---------------------- Feature Scaling -----------------------
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
old_x = sc.fit_transform(old_x)
print(old_x)
```

■ ***Feature Scaling:*** Feature Scaling is a technique to ***standardize*** the ***independent features*** present in the data in a fixed range. It is performed during the ***data pre-processing*** to handle ***highly varying magnitudes*** or values or units. If feature scaling is not done, then a machine learning algorithm tends to ***weigh greater*** values, ***higher*** and consider ***smaller values*** as the ***lower*** values, regardless of the unit of the values.

☞ ***Example:*** If an algorithm is not using the feature scaling method then it can consider the value 3000 meters to be greater than 5 km but that's actually not true and in this case, the algorithm will give wrong predictions. So, we use Feature Scaling to bring all values to the same magnitudes and thus, tackle this issue.

	Country	Age	Salary
0	France	44.000000	72000.000000
1	Spain	27.000000	48000.000000
2	Germany	30.000000	54000.000000
3	Spain	38.000000	61000.000000
4	Germany	40.000000	63777.777778
5	France	35.000000	58000.000000
6	Spain	38.777778	52000.000000
7	France	48.000000	79000.000000
8	Germany	50.000000	83000.000000
9	France	37.000000	67000.000000

```
[[ 1.22474487e+00 -6.54653671e-01 -6.54653671e-01  7.58874362e-01
   7.49473254e-01]
 [-8.16496581e-01 -6.54653671e-01  1.52752523e+00 -1.71150388e+00
  -1.43817841e+00]
 [-8.16496581e-01  1.52752523e+00 -6.54653671e-01 -1.27555478e+00
  -8.91265492e-01]
 [-8.16496581e-01 -6.54653671e-01  1.52752523e+00 -1.13023841e-01
  -2.53200424e-01]
 [-8.16496581e-01  1.52752523e+00 -6.54653671e-01  1.77608893e-01
   6.63219199e-16]
 [ 1.22474487e+00 -6.54653671e-01 -6.54653671e-01 -5.48972942e-01
  -5.26656882e-01]
 [-8.16496581e-01 -6.54653671e-01  1.52752523e+00  0.00000000e+00
  -1.07356980e+00]
 [ 1.22474487e+00 -6.54653671e-01 -6.54653671e-01  1.34013983e+00
   1.38753832e+00]
 [-8.16496581e-01  1.52752523e+00 -6.54653671e-01  1.63077256e+00
   1.75214693e+00]
 [ 1.22474487e+00 -6.54653671e-01 -6.54653671e-01 -2.58340208e-01
   2.93712492e-01]]
```

☞ ***What is Feature Scaling and why do we have to use it for:*** Feature Scaling is a technique that will put all your features in the same range.

- If we have a look again at our data well we can clearly see that the values of the ***age*** feature are not in the same range as the values of the ***salary*** feature.
- The ***ages*** from like ***0*** to ***100*** and ***salary*** is from ***0*** to ***100,000***. So these two features are not in the same range and feature scaling consist of putting the values of these two features in the same range.

Country	*Age*	*Salary*	*Purchased*
France		72000	No
Spain	27	48000	Yes
Germany	30	54000	No
Spain	38	61000	No
Germany	40		Yes
France	35	58000	Yes
Spain		52000	No
France	48	79000	Yes
Germany	50	83000	No
France	37	67000	Yes

☞ ***Why we have to apply FEATURE SCALING:*** Well that's because when training our ML models (some of them not all of them), if you have ***some features*** that are taking ***much higher values*** than ***other features*** this can create a ***bias*** in the ***correlations computations***.

- In other words the ***features*** that have ***high values*** compared to the other ones will ***dominate*** the other ***features*** so that the ***other features*** might not be ***considered***.
- However the other features might be the ones having the largest impact on the dependent variable.
- You know the ones that actually contribute the most to predicting the outcome.
- The dependent variable and that's why it's very very important to put all the features in the same scale.

Feature Scaling

Standardisation	Normalisation
$x_{stand} = \frac{x - \text{mean}(x)}{\text{standard deviation}\ (x)}$	$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)}$

$$X_{stand} = \frac{x - \mu}{\sigma}$$

μ is the mean and

σ is the standard deviation.

- ***Not all ML models needs feature scaling:*** You will see when we start building ***ML models***, sometimes we will apply ***feature scaling*** and sometimes not. That's because some ***ML models*** models actually ***automatically understand*** that some ***features*** have indeed values ***lower*** than the ***others*** and they will actually ***fix*** that ***automatically*** by playing with the ***coefficients***.
  - For example consider linear regression. ***`Linear regression`*** has some ***`coefficients`*** for each of the ***`features`*** and for the features having ***`super high values`*** it will just compensate with a ***`very low coefficient`***.
  - So it's fine that we won't have to apply ***Features Scaling*** for ***`linear regression`***
  - But you will see that for other models like ***Logistic Regression*** and ***SVR*** we have to apply ***Feature Scaling***.

  Bounding Range: Normalization has a bounding range in (0, 1). Unlike normalization, standardization does not have a bounding range, however in many cases it may be in between (-3, 3).

- ***The Big Question – NORMALIZE or STANDARDIZE?***
  Normalization vs. standardization is an eternal question among machine learning newcomers. Let me elaborate on the answer in this section.

  `Normalization` is good to use when you know that the distribution of your data **does not follow** a ***Gaussian distribution***. This can be useful in ***algorithms*** that ***do not assume any distribution*** of the data like ***`K-Nearest Neighbors`*** and ***`Neural Networks`***.

  `Standardization`, on the other hand, can be helpful in cases where the data **follows** a ***Gaussian distribution***. However, this does not have to be ***`necessarily true`***. Also, unlike ***normalization***, ***standardization*** does not have a ***Bounding Range***. So, even if you have outliers in your data, they will ***not be `affected`*** by ***standardization***.

  However, at the end of the day, the choice of using normalization or standardization will depend on your problem and the machine learning algorithm you are using. There is no hard and fast rule to tell you when to normalize or standardize your data. You can always start by fitting your model to raw, normalized and standardized data and compare the performance for best results.

- ***How to apply:***

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
```

  Will automatically apply $X_{stand} = \frac{x-\mu}{\sigma}$.

```
old_x = sc.fit_transform(old_x)
```

  Will scale our features.

1.9 Splitting the Dataset into the Training-set and Test-set

In machine your algorithm or model is going to learn from your data to make predictions or other machine learning goals. So ***Machine Learning Model*** is going to learn to do something on your data set by *understanding* some *correlations* that there is in your ***dataset*** and *imagine* your *machine learning model* is learning too much on the data set like it's learning *too much to correlations* then I'm not sure it's performance would be great.

We are going to build our *machine learning models* on a ***data set*** but then we have to test it on a ***new set*** which is going to be ***slightly different*** from the ***dataset*** on which we build the ***machine model***.

So we have to make ***two different sets of Training Sets*** on which we build the ***`ML model`*** and a ***Test Set*** on which we ***test*** the *performance* of this ***`ML model`*** and the performance on the **test set** *shouldn't be much different* from the performance on the **training sets** because this would mean that the ***`ML model`*** understood well the correlations and didn't learn them by heart so that you can ***adapt*** to *new sets* and *new situations*.
So that's the idea about splitting the dataset into a ***`training set`*** and a ***`test set`***.

Learn from train_data (80%) and apply to test_data (20%): We are building our ***`ML model`*** on this ***Test Set*** by establishing some correlations between the *independent variables* here and the *dependent variable* of the ***Test Set***.
And then once the machine learning model understands the correlations between *independent variables* and the *dependent variable* we will test ***`ML model`*** that it can apply the correlations based on the *training set* on the *test set*.
That means that we will see that ***`ML model`*** can predict about the *dependent variable* on the *teat dataset* (which is 20% in our case).

If ML model learn from 100% from the data it didn't understand quite well the logic and won't be able to make good predictions. That's called ***overfitting*** we will be talking about that in further details in the ***regression section*** and we will learn how to use ***REGULARISATION*** techniques to prevent this.

We need to make two different datasets. The ***training sets*** on which the machine learning model ***learns*** and the ***test set*** on which we test if the ***ML model learned correctly*** the ***correlations***.

```
# ------------ Splitting the Dataset: Train and Test ----------------
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(old_x, old_y, test_size= 0.2, random_state = 0)
print(x_train)
print(y_train)
print(x_test)
print(y_test)
```

❒ ***What is Splitting Dataset:*** We will split our dataset into two separate dataset, Training-set and Test-set.

- ▶ Training-set will contain 70% or 80% data to train our ML models.
- ▶ Other 30% or 20% data is the Test-set, which will be used to test our models how well it will predict.

❒ ***Why Splitting Dataset:*** We will always keep test-set to avoid the ***Overfitting***. ***Overfitting*** is when ML model learn too much from ***train-set*** but ***bad*** at ***prediction***

☯ ***Examples of Overfitting:*** Let's say we want to predict if a student will land a job interview based on her resume. Now, assume we train a model from a dataset of 10,000 resumes and their outcomes. Next, we try the model out on the original dataset, and it predicts outcomes with 99% accuracy... wow!

- ☹ But now comes the bad news. When we run the model on a ***new ("unseen")*** dataset of resumes, we only get 50% accuracy... uh-oh!
- ☹ Our model doesn't generalize well from our ***training data*** to ***unseen data***.
- ☹ This is known as ***overfitting***, and it's a common problem in ***machine learning*** and ***data science***.

❒ ***How to apply:***

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(old_x, old_y, test_size= 0.2, random_state = 0)
```

Parameters:

arrays : sequence of indexables with same length / shape[0]
Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.

test_size : *float or int, default=None*
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If `train_size` is also None, it will be set to 0.25.

train_size : *float or int, default=None*
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state : *int, RandomState instance or None, default=None*
Controls the shuffling applied to the data before applying the split. Pass an int for reproducible output across multiple function calls. See Glossary.

shuffle : *bool, default=True*
Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None.

stratify : *array-like, default=None*
If not None, data is split in a stratified fashion, using this as the class labels. Read more in the User Guide.

➲ ***x_train, x_test, y_train, y_test*** is used because `train_test_split()` returns four matrices. And we grab them in these four variables ***x_train, x_test, y_train, y_test***. The order must be maintained.

➲ `test_size= 0.2` means ***test size*** is ***20%***. It should be 20% or 25% or 30-33% and less than 40% (very rare). `train_size` Not needed, it would automatically set to ***1- test_size***.

➲ `random_state = 0` Means ***no shuffling***. Actually ***shuffling*** is ***good*** option 0 is used here to generate same results. Use 42 generally.

1.10 Data Preprocessing Template

```
# Data Preprocessing Template

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Data.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

```
# Feature Scaling
"""from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
sc_y = StandardScaler()
y_train = sc_y.fit_transform(y_train.reshape(-1,1))"""
```

☝ ***Missing data***, ***Categorical data***, ***Feature scaling*** are optional. Hence not included here.

1.11 Standard Deviation

In statistics, the **standard deviation** is a measure of the amount of variation or dispersion of a set of values.[1] A low standard deviation indicates that the values tend to be close to the mean (also called the expected value) of the set, while a high standard deviation indicates that the values are spread out over a wider range.

Standard deviation may be abbreviated **SD**, and is most commonly represented in mathematical texts and equations by the lower case Greek letter sigma **σ** or **σ^2**, for the population standard deviation, or the Latin letter s, for the sample standard deviation.

The standard deviation of a random variable, sample, statistical population, data set, or probability distribution is the square root of its variance. It is algebraically simpler, though in practice less robust, than the average absolute deviation.[2][3] A useful property of the standard deviation is that, unlike the variance, it is expressed in the same unit as the data.

Example:

Suppose that the entire population of interest is eight students in a particular class. For a finite set of numbers, the population standard deviation is found by taking the square root of the average of the squared deviations of the values subtracted from their average value. The marks of a class of eight students (that is, a statistical population) are the following eight values:

$$2,\ 4,\ 4,\ 4,\ 5,\ 5,\ 7,\ 9.$$

These eight data points have the mean (average) of 5:

$$\mu = \frac{2+4+4+4+5+5+7+9}{8} = \frac{40}{8} = 5.$$

First, calculate the deviations of each data point from the mean, and square the result of each:

$$\begin{aligned} (2-5)^2 &= (-3)^2 = 9 & (5-5)^2 &= 0^2 = 0 \\ (4-5)^2 &= (-1)^2 = 1 & (5-5)^2 &= 0^2 = 0 \\ (4-5)^2 &= (-1)^2 = 1 & (7-5)^2 &= 2^2 = 4 \\ (4-5)^2 &= (-1)^2 = 1 & (9-5)^2 &= 4^2 = 16. \end{aligned}$$

The variance is the mean of these values:

$$\sigma^2 = \frac{9+1+1+1+0+0+4+16}{8} = \frac{32}{8} = 4.$$

and the *population* standard deviation is equal to the square root of the variance:

$$\sigma = \sqrt{4} = 2.$$

☞ In the case where X takes random values from a finite data set x_1, x_2, …, x_N, with each value having the same probability, the standard deviation is

$$\sigma = \sqrt{\frac{1}{N}\left[(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_N - \mu)^2\right]}, \text{ where } \mu = \frac{1}{N}(x_1 + \cdots + x_N),$$

or, by using summation notation,

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}, \text{ where } \mu = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

☞ If, instead of having equal probabilities, the values have different probabilities, let x_1 have probability p_1, x_2 have probability p_2, …, x_N have probability p_N. In this case, the standard deviation will be

$$\sigma = \sqrt{\int_{\mathbf{X}} (x - \mu)^2\, p(x)\, dx}, \text{ where } \mu = \int_{\mathbf{X}} x\, p(x)\, dx,$$

Practiced version code

```
# ------------------ Data Preprocessing --------------------



# ------------------- Importing the libraries ----------------------
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# x = np.arange(0, 5, 0.1)
# y = np.sin(x)
# plt.plot(x, y)
# plt.show()



#-------------------- importing the dataset --------------------------
dataset = pd.read_csv("Data.csv")
print(dataset)
# ":-1" is to all columns exluding last column
# ":" to select all rows
old_x = dataset.iloc[:, :-1]
old_y = dataset.iloc[:, 3]
print("Before\n",old_x)



# ----------------------  Taking care of missing data ----------------------
# import sklearn
# print(sklearn.__version__)

from sklearn.impute import SimpleImputer
imptr = SimpleImputer(missing_values= np.nan, strategy="mean") # "NaN" connot be used. Use np.nan
# fix the 2nd and 3rd column
impt = imptr.fit(old_x.iloc[:, 1:3])
print(impt) # it is a method, acts on feature matrix
old_x.iloc[:, 1:3] = impt.transform(old_x.iloc[:, 1:3])
print("After\n",old_x)

# ooooooooo For older version ooooooooo
# from sklearn.preprocessing import Imputer
# imptr = Imputer(missing_values='NaN', strategy='mean', axis=0, verbose=0, copy=True)



# -------------- Categorical data --------------
# from sklearn.preprocessing import LabelEncoder, OneHotEncoder
# label_encode_x = LabelEncoder()
# old_x.iloc[:, 0] = label_encode_x.fit_transform(old_x.iloc[:, 0])
```

```
# print(old_x)

from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

#Encode Country Column
ct = ColumnTransformer(transformers = [("encoding", OneHotEncoder(), [0])], remainder = 'passthrough')
# remainder = 'passthrough' for remaining columns to be unchanged
old_x = ct.fit_transform(old_x) # convert this output to NumPy array
print(old_x)
old_x = np.array(old_x)
print(old_x)

#Encode dependent vector Column
label_encode = LabelEncoder()
old_y = label_encode.fit_transform(old_y)
print(old_y)




# ---------------------- Feature Scaling ------------------------
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
old_x = sc.fit_transform(old_x)
print(old_x)



# ------------ Splitting the Dataset: Train and Test ----------------
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(old_x, old_y, test_size= 0.2, random_state = 0)
print(x_train)
print(y_train)
print(x_test)
print(y_test)




# # ^^^^^^^^^^ List slicing ^^^^^^^^^^

# # Initialize list
# Lst = [50, 70, 30, 20, 90, 10, 50]

# # Display list
# print(Lst[::])

# # Index -1 represents the last element and -
n represents the first element of the list(considering n as the length of the list). Lists can also be ma
nipulated using negative indexes also.


# # Initialize list
# Lst = [50, 70, 30, 20, 90, 10, 50]

# # Display list
# print(Lst[-7::1])

# # Display list
# print(Lst[1:5])


# python prcts_dt_prep.py
```

Current codes

```
# Data Preprocessing Tools

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
```

```
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Data.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
print(X)
print(y)

# Taking care of missing data
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
imputer.fit(X[:, 1:3])
X[:, 1:3] = imputer.transform(X[:, 1:3])
print(X)

# Encoding categorical data
# Encoding the Independent Variable
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [0])], remainder='passthrough')
X = np.array(ct.fit_transform(X))
print(X)
# Encoding the Dependent Variable
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
print(y)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)
print(X)

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
print(X_train)
print(X_test)
print(y_train)
print(y_test)
```

Legacy (older) codes. Cause ERR.

missing_data.py and ***categorical_data.py***

```
# Data Preprocessing

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Data.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 3].values

# Taking care of missing data
from sklearn.preprocessing import Imputer
imputer = Imputer(missing_values = 'NaN', strategy = 'mean', axis = 0)
imputer = imputer.fit(X[:, 1:3])
X[:, 1:3] = imputer.transform(X[:, 1:3])

# Encoding categorical data
# Encoding the Independent Variable
```

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X = LabelEncoder()
X[:, 0] = labelencoder_X.fit_transform(X[:, 0])
onehotencoder = OneHotEncoder(categorical_features = [0])
X = onehotencoder.fit_transform(X).toarray()
# Encoding the Dependent Variable
labelencoder_y = LabelEncoder()
y = labelencoder_y.fit_transform(y)
```

FAQ: Normalize data before or after split of training and testing data?

I want to separate my data into ***train*** and ***test*** set, should I apply ***normalization over*** data ***before*** or ***after*** the ***split***? Does it make any difference while building predictive model?

- First need to ***split*** the data into ***training*** and ***test set*** (validation set could be useful too).
  - Don't forget that ***testing*** data points represent ***real-world data***. Feature normalization (or data standardization) of the explanatory (or predictor) variables is a technique used to center and normalise the data by subtracting the mean and dividing by the variance. If you take the mean and variance of the whole dataset you'll be introducing future information into the training explanatory variables (i.e. the mean and variance).

  - Therefore, you should perform feature normalisation over the training data. Then perform normalisation on testing instances as well, but this time using the mean and variance of training explanatory variables. In this way, we can test and evaluate whether our model can generalize well to new, unseen data points.

For a more comprehensive read, you can read my article

Feature Scaling and Normalisation in a nutshell

Simple Linear Regression

Algorithms and codes for Simple Linear Regression

2.1.1 What is Simple Linear Regression

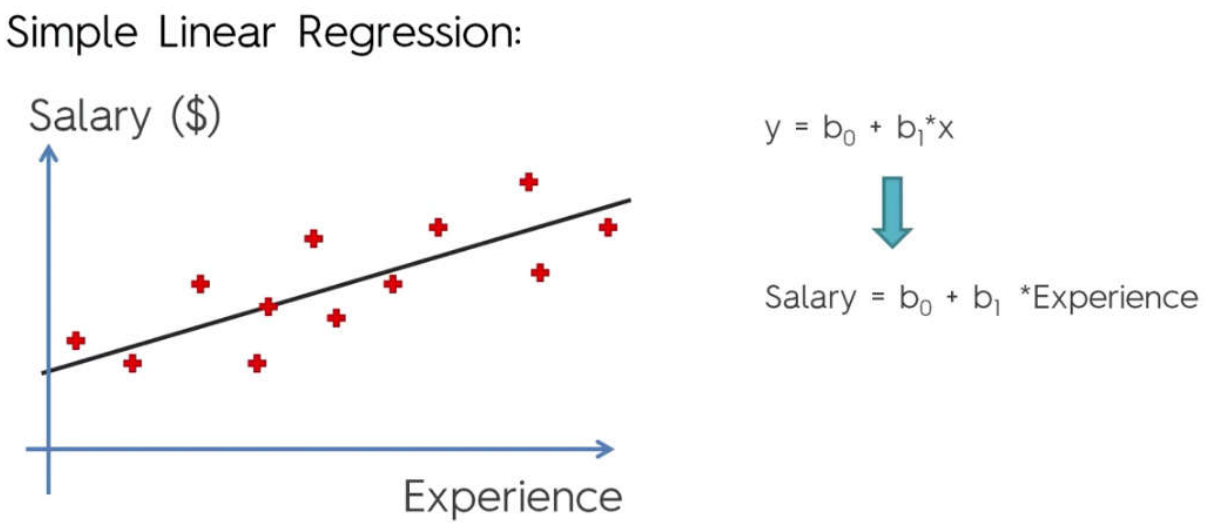
Simple Linear Regression is a type of Regression algorithms that models the ***relationship between*** a ***dependent variable*** and a ***single independent variable***. The relationship shown by a Simple Linear Regression model is linear or a **sloped straight line**, hence it is called Simple Linear Regression.

$$y = b_0 + b_1 x + \epsilon$$

x = independent variable
y = dependent variable

b_0 = It is the ***intercept*** of the Regression line (can be obtained putting x=0) with y-axis. Eg: *Minimum salary for Fresher*.
b_1 = It is the ***slope*** of the regression line, which tells whether the line is increasing or decreasing.
ϵ = The ***error*** term. (For a good model it will be negligible)

☞ The key point in Simple Linear Regression is that the ***dependent variable*** must be a ***continuous/real*** value. However, the ***independent variable*** can be measured on ***continuous*** or ***categorical*** values.



❒ ***Simple Linear regression algorithm has mainly two objectives:***

☞ ***Model the relationship between the two variables:*** Such as the relationship between ***Income*** and ***expenditure***, ***experience*** and ***Salary***, etc.

☞ ***Forecasting new observations:*** Such as ***Weather forecasting*** according to temperature, ***Revenue*** of a company according to the investments in a year, etc.

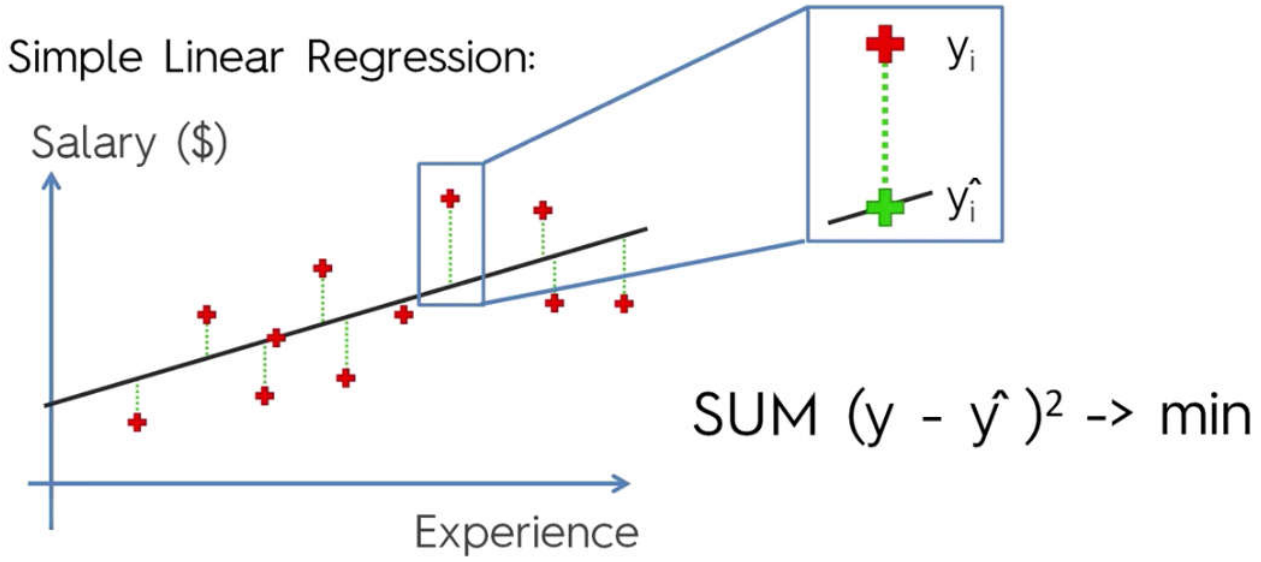
2.1.2 Finding the Best fitting line

The algorithm finds a line for which

$$minimize \sum_{i=0}^{n} (y_i - \hat{y}_i)^2$$

Where y_i are the given points $(x_i, y_i), i = 0, 1, 2, 3, \dots, n$.
And $\hat{y}_i$ are $\hat{y}_i = b_0 + b_1 x_i$, where $i = 0, 1, 2, 3, \dots, n$

Ordinary Least Squares



2.1.3 Implementation of Simple Linear Regression

❒ ***Problem Statement example for Simple Linear Regression:***

- ☞ Here we are taking a dataset that has two variables: ***salary*** (dependent variable) and ***experience*** (Independent variable).

❒ ***The goals of this problem is:***

- ☞ We want to find out if there is any *correlation* between these *two variables*
- ☞ We will find the *best fit line* for the dataset.
- ☞ How the *dependent variable* is changing by changing the *independent variable*.

❒ ***Create a Simple Linear Regression model:*** Now we will create a Simple Linear Regression model to find out the best fitting line for representing the relationship between these two variables.

- ☞ To implement the Simple Linear regression model in machine learning using Python, we need to follow the below steps:

[1] ***Step-1: Data Pre-processing***

The first step for creating the Simple Linear Regression model is data pre-processing. We have already done it earlier in this tutorial. But there will be some changes, which are given in the below steps:

- ➢ First, we will import the three important libraries, which will help us for loading the dataset, plotting the graphs, and creating the Simple Linear Regression model.

```
# ------------------- Importing the libraries -------------------
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# ------------------- Importing the dataset -------------------
dataset = pd.read_csv('Salary_Data.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 1].values

# -------------- Splitting the dataset into the Training set and Test set ----------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/3, random_state = 0)
```

- ➢ Next, we will load the dataset into our code:

```
dataset = pd.read_csv('Salary_Data.csv')
```

By executing the above line of code (ctrl+ENTER), we can read the dataset on our Spyder IDE screen by clicking on the variable explorer option.

- After that, we need to extract the dependent and independent variables from the given dataset. The independent variable is years of experience, and the dependent variable is salary. Below is code for it:

```
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 1].values
```

In the above lines of code, for **X** variable, we have taken **-1** value since we want to **remove the last column from the dataset**. For ***y*** variable, we have taken ***1*** value as a parameter, since we want to extract the second column and indexing starts from the zero.

- Next, we will split both variables into the test set and training set. We have ***30*** observations, so we will take ***20*** observations for the ***training*** set and ***10*** observations for the ***test*** set.
  - We are splitting our dataset so that we can train our model using a training dataset and then test the model using a test dataset. The code for this is given below:

```
# -------------- Splitting the dataset into the Training set and Test set ----------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/3, random_state = 0)
```

  - ***Feature Scaling:*** For simple linear Regression, we will not use Feature Scaling. Because Python libraries take care of it for some cases, so we don't need to perform it here. Now, our dataset is well prepared to work on it and we are going to start building a Simple Linear Regression model for the given problem.

[2] ***Step-2: Fitting the Simple Linear Regression to the Training Set:***

- Now the second step is to fit our model to the training dataset. To do so, we will import the ***LinearRegression*** **class** of the ***linear_model*** **library** from the ***scikit learn***.
- After importing the ***class***, we are going to create an ***object*** of the class named as a ***regressor***.
- And then we fit our data by using ***fit()*** method.

The code for this is given below:

```
# ---------  Fitting the Simple Linear Regression model to the Training dataset -----------
from sklearn.linear_model import LinearRegression   # import library
s_l_regressor = LinearRegression()                  # regressor object
s_l_regressor.fit(X_train, y_train)                 # fit train data
```

In the above code, we have used a ***fit()*** method to fit our Simple Linear Regression object to the training set. In the ***fit()*** function, we have passed the ***x_train*** and ***y_train***, which is our training dataset for the ***dependent*** and an ***independent*** variable. We have fitted our ***regressor*** object to the training set so that the model can **easily learn the correlations** between the ***predictor*** and ***target variables***. After executing the above lines of code, we will get the below output.

[3] ***Step: 3. Prediction of test set result:***

In this step, we will provide the test dataset (new observations) to the model to check whether it can predict the correct output or not. We will create a prediction vector y_pred,

```
# Predicting the test set results. Test set will be used
y_pred = s_l_regressor.predict(X_test)
```

- You can check the variable by clicking on the variable explorer option in the IDE, and also compare the result by comparing values from ***y_pred*** and ***y_test***. By comparing these values, we can check how good our model is performing.

- We will create a vector of predictive values called ***y_pred*** that will contain the predictions of the ***test set salaries*** and this is going to be the same for all machinery models that we will create. ***y_pred*** is going to be the ***vector*** of ***predictions*** of the ***dependent variable***.

[4] ***Step: 4. visualizing the Training set results:***

Now in this step, we will visualize the training set result. To do so, we will use the ***scatter()*** function of the ***pyplot*** library, which we have already imported in the pre-processing step. The ***scatter()*** function will create a scatter plot of observations.

```
# visualising the Training-set result
plt.scatter(X_train, y_train, color = "red")
plt.plot(X_train, s_l_regressor.predict(X_train), color = "blue")   # notice the Train set is used
plt.title("Salary vs Experience 'Trainig Set'")
plt.xlabel("Years of Experience")
plt.ylabel("Salary")
plt.show()
```

➢ The good fit of the line can be observed by calculating the ***difference between actual values and predicted values (difference of the corresponding y-values)***. But as we can see in the above plot, most of the observations are close to the regression line, hence our model is good for the training set.

[5] ***Step: 5. visualizing the Test set results:***

In the previous step, we have visualized the performance of our model on the training set. Now, we will do the same for the Test set. The complete code will remain the same as the above code, except in this, we will use ***x_test***, and ***y_test*** instead of ***x_train*** and ***y_train***.

➢ Here we are also changing the color of observations and regression line to differentiate between the two plots, but it is optional.

```
# visualising the Test-set result
plt.scatter(X_test, y_test, color = "red")
plt.plot(X_train, s_l_regressor.predict(X_train), color = "blue")   # notice the Train set is used again
plt.title("Salary vs Experience 'Trainig Set'")
plt.xlabel("Years of Experience")
plt.ylabel("Salary")
plt.show()
```

NOTICE:

☝ `plt.plot(X_train, s_l_regressor.predict(X_train), color = "blue")`remain same for both plot. Because it is the same regression-line.

☝ Here ***s_l_regressor.predict(x)*** is the equation (function) of regression line. As we can see, most of the observations are close to the regression line.

2.1.4 Idea behind making a ML model

Our model will ***learn the correlations between*** the *x_train* and *y_train* so that it can later predict the results of the ***dependent variable*** the ***salaries*** based on these information here.

☞ Once our model is trained, we will test its performance (its power of prediction) on the ***test-set***.

☯ For example, here we will predict the corresponding ***salary*** and then we will compare the ***predicted salary*** to the ***real salary*** which are the salaries on the ***test-set***.

☝ So here the ***machine*** is the *simple linear regression model* and ***learning*** is the fact that this *model learns on the training set* here composed of ***x_train*** and ***y_train***.

☝ We made this machine to learn on the ***training set*** to understand the ***correlations*** between the ***experience*** and the ***salary*** so that this machine based on its learning experience (on train-dataset) can then predict the ***salary*** with respect to the ***experience*** on ***test-dataset***.

That's what machine learning is about.

2.1.5 FAQ

👽 ***Why do we take the squared differences and simply not the absolute differences?***

Because the ***squared differences*** makes it easier to ***derive a regression line***. Indeed, *to find that line we need to compute the* ***first derivative*** *of the* ***loss error function***, and it is much harder to compute the **derivative** of **absolute values** than **squared values**.

👽 ***Why didn't we apply Feature Scaling in our Simple Linear Regression model?***

It's simply because since ***y*** is a ***linear combination*** of the ***independent variables***, the coefficients can ***adapt their scale*** to put everything on the same scale. For example if you have two independent variables ***x1*** and ***x2*** and if ***y*** takes values between ***0*** and ***1***, ***x1*** takes values between ***1*** and ***10*** and ***x2*** takes values between ***10*** and ***100***, then ***b1*** can be multiplied by ***0.1*** and ***b2*** can be multiplied by ***0.01*** so that ***y***, ***b1x1*** and ***b2x2*** are all on the same scale.

👽 ***What does 'regressor.fit(X_train, y_train)' do exactly?***

The **fit** method will take the values of ***X_train*** and ***y_train*** and then will compute the coefficients b_0 and b_1 of the Simple Linear Regression equation ($y = b_0 + b_1 x$). That's the whole purpose of this fit method here.

Chapter 2 : Section 2

Multiple Linear Regression

Introduction to Multiple Linear Regressions

2.2.1 Multiple Linear Regression

There many cases in which the ***response/dependent variable*** is affected by more than one ***predictor/independent variable***, for such cases, the Multiple Linear Regression algorithm is used.

- Multiple Linear Regression is an extension of Simple Linear regression as it takes ***more*** than one ***predictor variable*** to predict the ***response variable***.
- Multiple Linear Regression is one of the important regression algorithms which models the ***linear relationship*** between a ***single dependent continuous variable*** and ***more than one independent variable***.

☯ ***Example:*** Prediction of CO2 emission based on engine size and number of cylinders in a car.

❒ ***Some key points about MLR:***

- For ***MLR***, the ***dependent or target variable*** Y must be the ***continuous/real***, but the ***predictor or independent variable*** may be of ***continuous*** or ***categorical*** form.
- Each ***feature variable*** must model the ***linear relationship*** with the ***dependent variable***.
- ***MLR*** tries to ***fit*** a ***regression line*** through a ***multidimensional space of data-points***.

2.2.2 Equation of MLR|

❒ ***More than two independent variables:*** In multi linear regression, we deal with more than one independent variables.

☯ In terms of a student and his ***Grades*** is ***depend variable***. What grade does a student get? Then in the ***independent variables*** could be how much the student has studied for the exam (***study time***), how much he has slept before the exam (***sleep time***), how many lectures he has attended throughout the course (***attendance***) and the things like that.

❒ ***The equation:***

$$y = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \ldots\ldots + b_nx_n$$

- y is the dependent variable, b_0 is the constant, b_i are coefficients and x_i are independent variables .

☯ Consider the following dataset.

fx R&D Spend

	A	B	C	D	E	F
1	R&D Spend	Administration	Marketing Spend	State	Profit	
2	165349.2	136897.8	471784.1	New York	192261.83	
3	162597.7	151377.59	443898.53	California	191792.06	
4	153441.51	101145.55	407934.54	Florida	191050.39	
5	144372.41	118671.85	383199.62	New York	182901.99	
6	142107.34	91391.77	366168.42	Florida	166187.94	
7	131876.9	99814.71	362861.36	New York	156991.12	
8	134615.46	147198.87	127716.82	California	156122.51	
9	130298.13	145530.06	323876.68	Florida	155752.6	
10	120542.52	148718.95	311613.29	New York	152211.77	
11	123334.88	108679.17	304981.62	California	149759.96	
12	101913.08	110594.11	229160.95	Florida	146121.95	
13	100671.96	91790.61	249744.55	California	144259.4	

Simple Linear Regression

$$y = b_0 + b_1 * x_1$$

Multiple Linear Regression

Dependent variable (DV) — Independent variables (IVs)

$$y = b_0 + b_1 * x_1 + b_2 * x_2 + \ldots + b_n * x_n$$

2.2.3 Multiple Linear Regression assumptions

❒ ***ASSUMPTIONS:*** Before building a linear regression model you need to check that these assumptions are true.

- ☝ If you ever do need to build a linear regression then make sure that you don't just blindly follow the steps presented here.
- ☝ First you need to check the assumptions of the linear regression and research them and make sure that they are correct when you're building your regression model.
- ☝ And then you can only proceed and be sure that you're building a good linear regression model.

Assumptions of a Linear Regression:

1. Linearity
2. Homoscedasticity
3. Multivariate normality
4. Independence of errors
5. Lack of multicollinearity

[1] ***Linearity:*** There must be a linear relationship between the ***dependent*** variable and the ***independent*** variables.

☞ ***`Scatterplots`*** can show whether there is a linear or curvilinear relationship.

[2] ***Homoscedasticity:*** This assumption states that the variance of error terms is similar across the values of the independent variables.

☞ A ***`plot`*** of **standardized residuals** versus **predicted values** can show whether points are equally distributed across all values of the independent variables.

[3] ***Multivariate Normality:*** Multiple Linear Regression assumes that the ***residuals*** (the **`differences between`** the observed value of the **dependent variable** y and the **predicted value** $\hat{y}$) are ***`normally distributed`***.

[4] ***Independence of errors:*** Multiple Linear Regression assumes that the ***residuals*** (the **`differences between`** the observed value of the **dependent variable** y and the **predicted value** $\hat{y}$) are ***`independent`***.

[5] ***Lack of multi-collinearity:*** Multiple Linear Regression assumes that the ***independent variables*** are not ***highly correlated*** with each other. This assumption is tested using ***Variance Inflation Factor (VIF)*** values.

2.2.4 Dummy Variables

In following table ***profit*** is our dependent variable and the rest the blue ones are all ***independent*** variables.

❒ What should we place in our equation for the ***`State`*** column because we don't actually have numerical values here, the ***`State`*** is actually a ***`categorical variable`*** (there is categorical variables and there's numeric variables).

☞ First you need to **go through** your **column** and find all the **different categories** you have. For every ***single category*** that you find, you need to create a new column. Then in each category-column put **1** where corresponding matching category appears and **0** for unmatched.

Profit	R&D Spend	Admin	Marketing	State
192,261.83	165,349.20	136,897.80	471,784.10	New York
191,792.06	162,597.70	151,377.59	443,898.53	California
191,050.39	153,441.51	101,145.55	407,934.54	California
182,901.99	144,372.41	118,671.85	383,199.62	New York
166,187.94	142,107.34	91,391.77	366,168.42	California

$$y = b_0 + b_1*x_1 + b_2*x_2 + b_3*x_3 + ???$$

🛠 The approach that you need to take when you face ***categorical variables*** in ***regression models*** is you need to create ***dummy variables***.

❒ So in this case we have two categories. Hence, we need to create a new column for New York and one for California. So we're kind of expanding our data set and adding some additional columns into it.

☞ Then we find all of rows where the state actually says New York and put a **1**.

Dummy Variables

Profit	R&D Spend	Admin	Marketing	State	New York	California
192,261.83	165,349.20	136,897.80	471,784.10	New York	1	0
191,792.06	162,597.70	151,377.59	443,898.53	California	0	1
191,050.39	153,441.51	101,145.55	407,934.54	California	0	1
182,901.99	144,372.41	118,671.85	383,199.62	New York	1	0
166,187.94	142,107.34	91,391.77	366,168.42	California	0	1

$$y = b_0 + b_1*x_1 + b_2*x_2 + b_3*x_3 + ???$$

State	New York
New York	1
California	0
California	0
New York	1
California	0

State	New York	California
New York	1	0
California	0	1
California	0	1
New York	1	0
California	0	1

❒ ***Do not use all the Dummy variables:*** This can leads us to collinearity and we eventually fall into dummy variable trap.

☞ All you have to do is use the ***New York*** column instead of ***States***.

☞ Don't use the California column (leads to dummy variable trap).

❒ ***Here all the information in our data is preserved:*** If we just stick to the one ***New York***, **1** means ***New York*** and **0** means ***California***. So we didn't lose any information by including only the New York. work as switches.

☞ ***Dummy variables act as switches:*** Actually all of the dummy variables they work as switches.

☞ ***Omit one dummy?:*** When you look at this approach it might seem biased. There is a **coefficient** b_4 for ***New York*** but for ***California*** there's **no coefficient**.

☝ In reality that's not the case because the way ***regression models*** work is that the coefficient of the dummy variable that you have not included will become the ***default situation*** for this ***regression model***.

What that means is that the **coefficient** for **California** is going to be included in the **constant** b_0 by default. When D_1 is equal to zero this whole equation will turn into an equation for **California**.

When D_1 becomes 1 you're adding b_4 which is the difference between ***New York*** and ***California coefficient***.

So basically you're altering from **California** to **New York** by flipping this **light switch** if it's on off. And the default state or the whole equation is working for **California**.

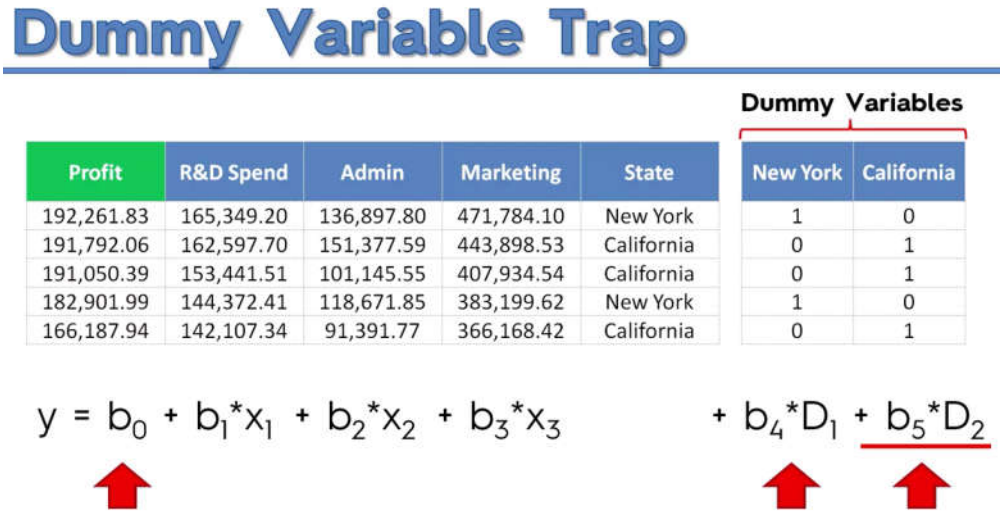
2.2.5 Dummy variable trap

Why you should never include all of your dummy variable columns

❒ ***Multicollinearity:*** Intuition here is that you're basically ***duplicating a variable***. This is because $D_2 = 1 - D_1$.

☞ The phenomenon where one or several **independent variables** in a linear regression predict another is called ***MULTICOLLINEARITY***. As a result of this effect the model cannot distinguish between the effects of D_1 from the effects from of D_2. Therefore it won't work properly. And this is the ***Dummy Variable Trap***.

☞ If you do the math behind this scenario you will see that the real problem is that you cannot have these three elements in your model at the same time the constant b_0 and both the dummy variables b_4D_1, b_5D_2.



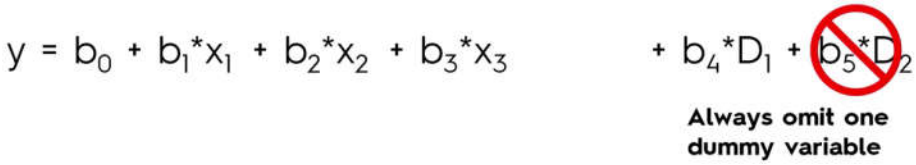
❒ ***How is the coefficient b_0 related to the dummy variable trap?***

Since $D_2 = 1 - D_1$ then if you include both D_1 and D_2 you get:

$$\begin{aligned} y &= b_0 + b_1x_1 + b_2x_2 + b_3x_3 + b_4D_1 + b_5D_2 \\ &= b_0 + b_1x_1 + b_2x_2 + b_3x_3 + b_4D_1 + b_5(1 - D_1) \\ &= (b_0 + b_5) + b_1x_1 + b_2x_2 + b_3x_3 + (b_4 - b_5)\, D_1 \\ &= b_0^* + b_1x_1 + b_2x_2 + b_3x_3 + b_4^*D_1 \end{aligned}$$

Where $b_0^* = (b_0 + b_5)$ and $b_4^* = (b_4 - b_5)$

Therefore the information of the redundant dummy variable D_2 is going into the constant b_0.



🛠 Whenever you're building a model always ***omit one dummy variable*** and this applies to the number of dummy variables in that specific dummy set. If you have **9** then you should only include **8**, if you have **100** then you should only include **99** of them.

🛠 Also note that if you have two sets of dummy variables then you need to apply the same rule to each set.

2.2.6 p-value

Before we get into ***Backward Elimination***, make sure to be introduced to the ***p-value*** and have a basic understanding of how it works. By looking at almost all the explanations of the p-value on the internet

❒ ***What is the p-value?***

☞ ***Null Hypothesis:*** To understand the P-value, we need to start by understanding the ***null hypothesis***: the **null hypothesis** is the assumption that the **parameters associated** to your **independent** variables are **equal** to **zero**. Therefore under this hypothesis, your ***observations are totally random***, and ***don't follow a certain pattern***. For example:

☯ ***Does the size of a state affect population density?*** The ***null hypothesis*** is **"all states have the same population density."**

☯ ***Do cats prefer fish or milk?*** The ***null hypothesis*** is **"cats have no preference; they like them the same."**

☞ ***P-value:*** The ***P-value*** is the probability that the parameters associated to your independent variables have ***certain nonzero values***, given that the ***null hypothesis is True***. The most important thing to keep in mind about the P-Value is that it is a statistical metric: the ***lower is the P-Value, the more statistically significant is an independent variable,*** that is the ***better predictor*** it will be.

❒ The ***smaller*** the ***p-value***, the stronger the evidence that you should ***reject*** the ***null hypothesis***. A ***p-value*** less than ***0.05*** (typically ≤ 0.05) is ***statistically significant***. It indicates strong evidence against the **null hypothesis**, as there is less than a **5% probability** the ***null is correct (and the results are random)***.

☞ The first step in backward elimination is pretty simple, you just select a ***significance level***, or select the ***P-value***. Usually, in most cases, a ***5%*** significance ***level is selected***. This means the ***P-value*** will be ***0.05***.

☞ ***P value*** is a ***statistical measure*** that helps scientists determine whether or not their ***hypotheses are correct***. ***P values*** are used to ***determine*** whether the ***results of their experiment*** are within the ***normal range of values*** for the ***events*** being ***observed***. Usually, if the P value of a data set is below a certain pre-determined amount (like, for instance, 0.05), scientists will *reject* the *"null hypothesis"* of their experiment.

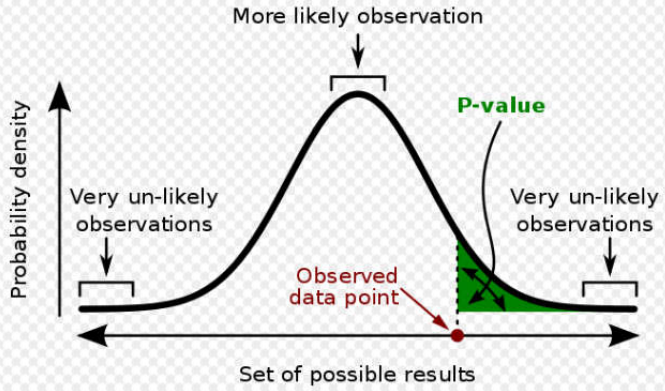
☞ The ***p-value*** is actually the **probability of getting a sample like ours**, or *more* ***extreme*** *than ours IF the* ***null hypothesis*** *is true*. So, we ***assume*** the ***null hypothesis is true*** and then ***determine how "strange" our sample really is***. If it is `not that strange` (a `large p-value`) then we *don't change our mind about the* ***null hypothesis***. As the **p-value** gets **smaller**, we start wondering if the **null really is true** and well maybe we should change our minds (and **reject** the **null hypothesis**).

Important:

Pr (observation | hypothesis) ≠ Pr (hypothesis | observation)

The probability of observing a result given that some hypothesis is true is *not equivalent* to the probability that a hypothesis is true given that some result has been observed.

Using the p-value as a "score" is committing an egregious logical error: **the transposed conditional fallacy.**



A **p-value** (shaded green area) is the probability of an observed (or more extreme) result assuming that the null hypothesis is true.

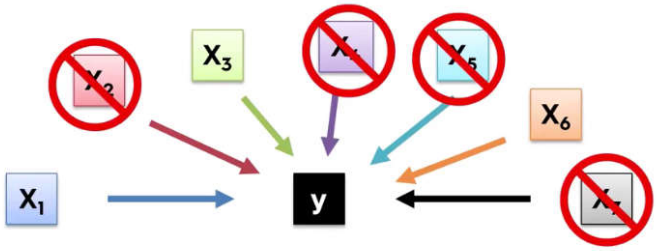
A little more detail: A small p-value indicates that by pure luck alone, it would be unlikely to get a sample like the one we have if the null hypothesis is true. If this is small enough we start thinking that maybe we aren't super lucky and instead our assumption about the null being true is wrong. Thats why we reject with a small p-value.

A large p-value indicates that it would be pretty normal to get a sample like ours if the null hypothesis is true. So you can see, there is no reason here to change our minds like we did with a small p-value.

In inferential statistics, the null hypothesis (often denoted H0)[1] is that two possibilities are the same. The null hypothesis is that the observed difference is due to chance alone. Using statistical tests, it is possible to calculate the likelihood that the null hypothesis is true.

2.2.7 Feature Selection methods

Different methods to select significant features



Among the multiple features we need to decide which ones we want to keep and which ones we want to throw out (throw out columns).

❒ ***Two common reasons:***

☞ ***Garbage in garbage out:*** If you throw lots of stuff into your model then your model is going to be a ***garbage model***.

- Unnecessary features increase the complexity of the model. Hence it is good to have only the most significant features and keep our model simple to get the better result.

☞ ***Don't make it too complex to explain to Someone:*** At the end of the day you're going to have to explain these variables and understand the not just the math behind them but actually what it means that certain variables predict the behavior of your dependent variable and you will have to explain that to people you're presenting to.

❒ ***5 methods of building models:***

i. All-in
ii. Backward Elimination
iii. Forward Selection
iv. Bidirectional Elimination
v. Score Comparison

5 methods of building models:

1. All-in
2. Backward Elimination
3. Forward Selection
4. Bidirectional Elimination
5. Score Comparison

} Stepwise Regression

☞ ***Stepwise regression:*** Sometimes you'll hear ***stepwise regression***. It's actually refers to ***2***, ***3*** and ***4*** because they are true ***step by step*** methods.

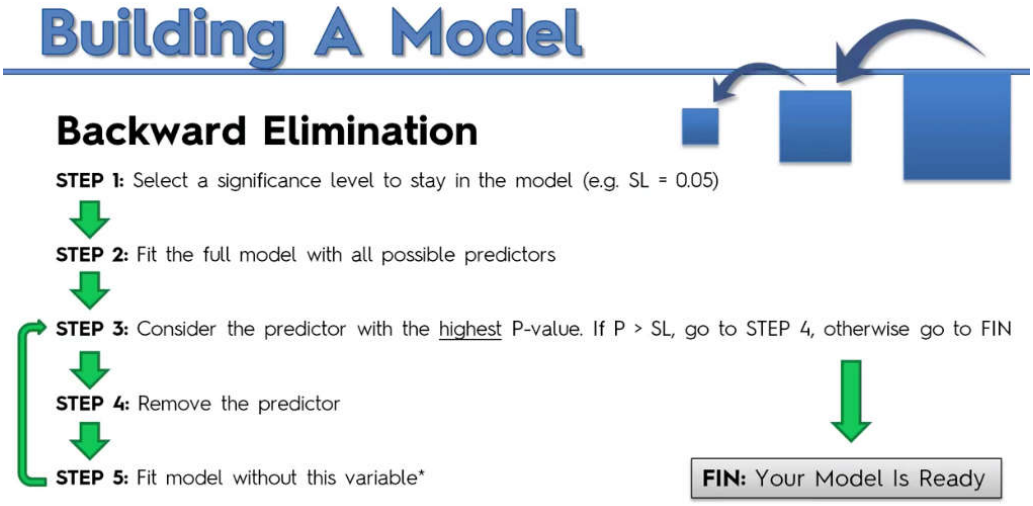
- More generally sometimes ***Bidirectional Elimination*** is called the ***Stepwise Regression*** because it is combination of ***Backward Elimination*** and ***Forward Selection***.

[1] ***All-in:*** All features are used. No specific feature is selected. When we use it:

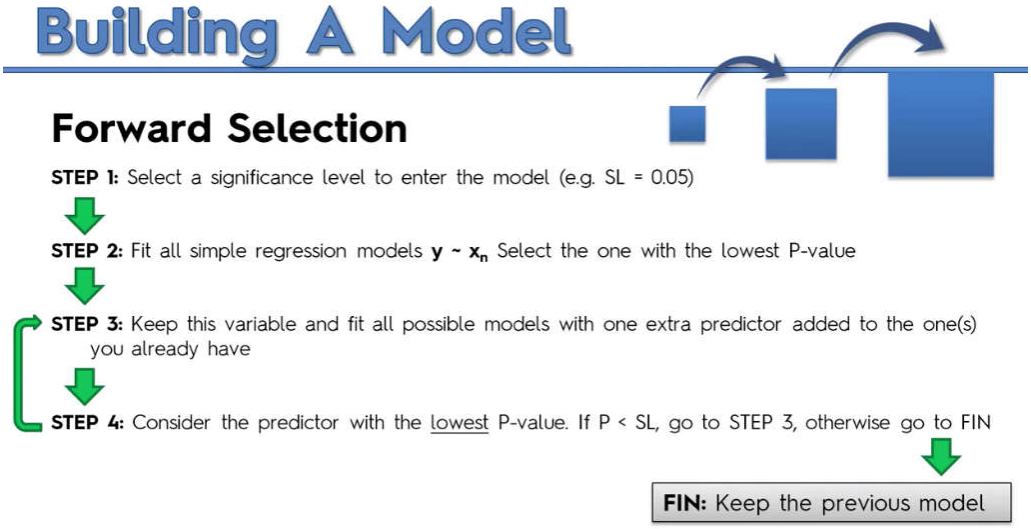
- ***Prior knowledge:*** If you have prior knowledge about those exact variables are going to be true predictors of your model (in case you build that model before)
- ***You have to:*** Somebody just gave you these variables and said please build a model. Well then you don't really have a choice. You just build the model (In case of framework or a company says that you have to use these variables.).
- ***Preparing for Backward Elimination:*** You need to use this method if you're preparing for a backward elimination.

[2] ***Backward Elimination:*** Start with ***All Independent Variables*** and `eliminate` the rest variables one by one by checking the **p-values**. We keep doing the procedure until we come to a point where the highest ***P values*** of all the variable are still less than your ***significance level***.

- ***STEP 1:*** Select a ***Significance Level*** **SL** to ***stay*** in the model (e.g. **`SL = 0.05`**). We check ***P-value*** against this ***SL***.
- ***STEP 2:*** **`Fit the full model`** with ***all possible predictors/Independent variable***
- ***STEP 3:*** Consider the `predictor/Independent variable` with the ***highest P-value***. If ***P > SL***, go to ***STEP 4***, otherwise go to ***FIN***
- ***STEP 4:*** **`Remove`** the predictor
- ***STEP 5:*** **`Fit model again`** without this variable. i.e. Rebuild and fit the model with the ***remaining variables***.
- ***FIN:*** Your Model Is **`Ready`**



[3] ***Forward Selection:*** Start with ***One Independent Variable*** and add the rest variables one by one by checking the p-values. Note that we do not keep the current model where P>SL, instead we keep the previous model.



- ***STEP 1:*** Select a ***Significance Level*** SL to *enter* the model (e.g. **SL = 0.05**). We check ***P-value*** against this ***SL***.
- ***STEP 2:*** Fit all ***Simple Linear Regression*** models $\sim x_n$. I.e for ***n*** predictors there will be ***n Simple Linear Regression*** models. Select the **one** with the **lowest P-value.**
- ***STEP 3:*** Keep this variable and fit all possible models again with one extra predictor (increase one independent/predictor variable added to the one(s) you already have).
- ***STEP 4:*** Consider the predictor with the **lowest P-value**. If $P < SL$, go to ***STEP 3***, otherwise go to ***FIN***
- ***FIN:*** Keep the previous model

☞ ***Actually what are we doing here is:***

- We create ***Multiple Simple Regression*** model with *every single independent variable*. Test their ***P-value*** against ***SL*** and increase predictors one by one. And select out of all those models that has the lowest p value for the independent variable
- We keep this selected variable and we fit all other possible models with one extra predictor. That means we've selected a ***Simple Linear Regression*** with **one variable**. Then construct all possible ***Linear Regressions with two variables*** from other predictors.
- Now we have all possible ***2 variable linear regressions.*** Out of all of these possible ***Two Variable Regressions*** we consider the one where the *new variable* that had the lowest ***p-value*** with $P < SL$ **(**means that variables a good one and it's a significant variable**)**.
- Then we moved back to ***Step 3***.

- Means that now we have a ***Regression With Two Variables*** and now we will add a third variable. We'll try all possible variables that we have left as our ***third variable*** and then out of all of those models with three variables we will go to ***Step 4*** and we'll select again the one of the **lowest p value** for that third variable. And so on.

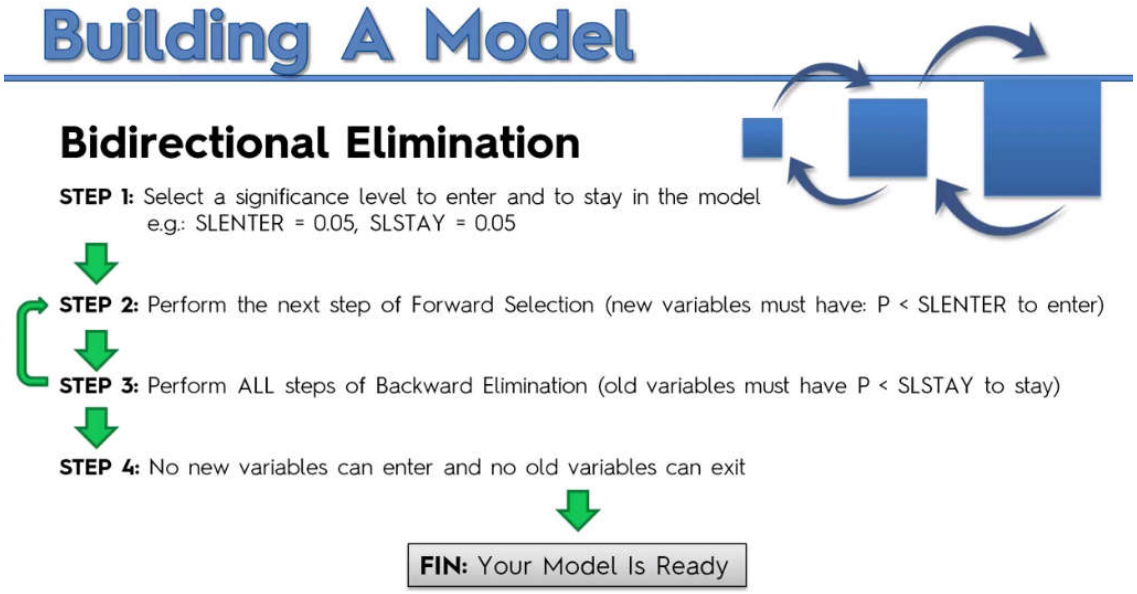
- So basically we'll be keep ***growing the regression model*** out of the all of the possible combinations every single time and ***growing at one variable at a time***.

- We will only stop when the variable that we've added that has a ***p-value*** that is greater than our *significance level* ***SL***. That is $P > SL$ is ***false***. Means that variable we just added is ***no longer significant***.

NOTE

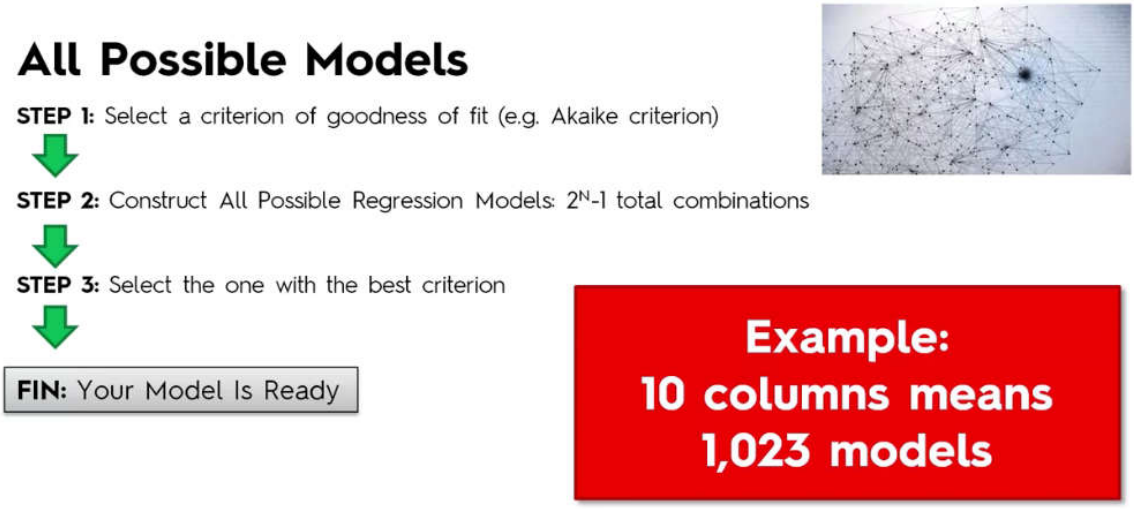
The trick is you not keep the current model but the previous one: We **reject the current model** with ***non-significant variable*** and **pick the previous model** (where all variables are significant).

[4] ***Bidirectional Elimination:***



- ***STEP 1:*** Select a significance level ***SL*** to ***Enter*** and to ***Stay*** in the model e.g.: $SLENTER = 0.05$, $SLSTAY = 0.05$
- ***STEP 2:*** Perform the next step of ***Forward Selection*** (new variables must have: $P < SLENTER$ to ***Enter***)
- ***STEP 3:*** Perform ALL steps of Backward Elimination (old variables must have $P < SLSTAY$ to ***Stay***)
- ***STEP 4:*** No new variables can enter and no old variables can exit
- ***FIN:*** Your Model Is Ready

[5] ***Score Comparison:*** Most resource consuming process.



2.2.8 Implement the MLR

Implementation of Multiple Linear Regression (MLR) model using Python. To implement MLR using Python, we have below problem:

Problem Description:

We have a dataset of 50 start-up companies. This dataset contains five main information: ***R&D*** Spend (Research and development), ***Administration*** Spend, ***Marketing*** Spend, ***State***, and ***Profit*** for a financial year.

Goal: To create a model that can easily determine which company has a ***maximum profit***, and which is the **most affecting factor** for the ***profit*** of a company.

Since we need to find the ***Profit***, so it is the ***dependent/target variable***, and the other ***four variables*** are ***independent variables***. Below are the main steps of deploying the MLR model:

1. ***Data Pre-processing*** Steps
2. ***Fitting the MLR*** model to the training set
3. ***Predicting*** the result of the test set

We analyze these 50 companies over this data set and create a model that will tell us which types of companies we should invest and our main criteria is the profit.

- ***What we are looking for is:*** we want to understand for instance where companies perform better in ***New York*** or ***California*** all other things held equal or which companies perform better if you hold this "State" column equal.
- ***We want to know:***
  - Will a company that ***spends more*** on ***marketing*** perform ***better*** or a company spends less on marketing.
  - Also we want to understand how a company spend more on ***R&D*** spend or to spend more on ***marketing***.

data_set - DataFrame

Index	R&D Spend	Administration	Marketing Spend	State	Profit
0	165349	136898	471784	New York	192262
1	162598	151378	443899	California	191792
2	153442	101146	407935	Florida	191050
3	144372	118672	383200	New York	182902
4	142107	91391.8	366168	Florida	166188
5	131877	99814.7	362861	New York	156991
6	134615	147199	127717	California	156123
7	130298	145530	323877	Florida	155753
8	120543	148719	311613	New York	152212
9	123335	108679	304982	California	149760
10	101913	110594	229161	Florida	146122
11	100672	91790.6	249745	California	144259
12	93863.8	127320	249839	Florida	141586
13	91992.4	135495	252665	California	134307

Format | Resize | Background color | Column min/max | Save and Close | Close

☺ It will help us to set up a **set of guidelines** for our own venture capitalist fund . For example, we are more interested in companies that work in ***New York*** and that have a very ***low administration*** spend and a very ***high R&D*** spend which is much higher than ***Administration*** or ***Marketing*** spend.

☺ So basically we are creating a model based off of this sample that will allow us to assess where and in which into which companies we want to invest to achieve their goal of maximizing profit.

NOTE:

iloc[:, 3] means only 4th column.
iloc[:, 3:] means all columns from 4th column

[1] ***Data preparation:***

```
# Funamental Libraries
import matplotlib as plt
import pandas as pd
import numpy as np

# import dataset
dataSet = pd.read_csv("50_Startups.csv")
X = dataSet.iloc[:, :-1] #all rows except last
y = dataSet.iloc[:, 4] # 5th row

# categorical to numerical
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
colTfrm = ColumnTransformer(transformers = [("encoder", OneHotEncoder(), [3])], remainder="passthr
ough" )
X_encoded = np.array(colTfrm.fit_transform(X))
        #  last column is now replaced with dummy colums (1st 3 colmns)
```

[2] ***Dummy Variable Trap:*** However, of course the ***Python library*** for ***`linear regression`*** is taking care of the ***Dummy Variable Trap***, so we wouldn't need to do it manually like we do it here.

☞ This line just remind us about the ***dummy variable trap*** because for some software/libraries you need to do it manually.

```
# Avoiding dummy-var trap: omit one dummy varable
X_go = X_encoded[:, 1:]         # select all columns starting from 2nd column
y_go = np.array(y) #converting Dataframe to Vector/Array
```

[3] ***Test size:*** `40` `train` and `10m` `test`.

```
# split dataset to Train and Test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_go, y_go, test_size = 0.2, random_state = 0)
```

[4] ***Create the model:*** Fitting multiple linear regression on ***Training set***

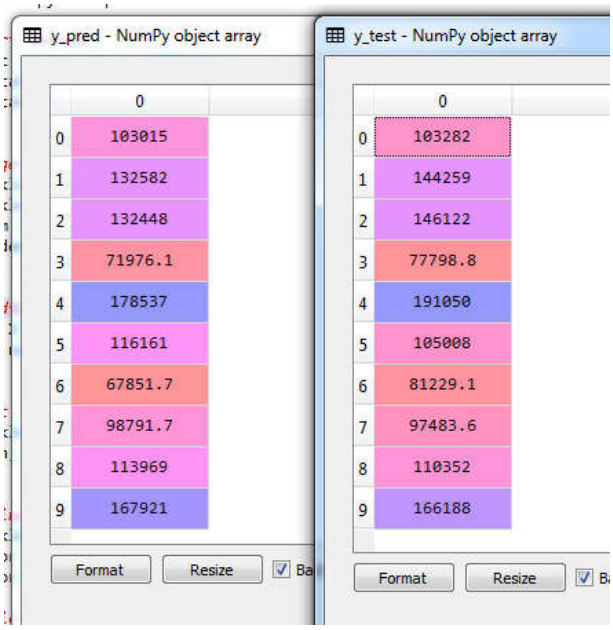
```
# Fitting multiple linear regression on traing set
from sklearn.linear_model import LinearRegression
regResor = LinearRegression()
regResor.fit(X_train, y_train)
```

[5] ***Predicting:*** Cannot plot the graph for this model because it is multidimensional.

```
# predict on the test-set X_test
y_pred = regResor.predict(X_test)
```

NOTE: We can convert a dataframe to Array obj.





Practiced version

```
# Funamental Libraries
import matplotlib as plt
import pandas as pd
import numpy as np

# import dataset
dataSet = pd.read_csv("50_Startups.csv")
X = dataSet.iloc[:, :-1] #all rows except last
y = dataSet.iloc[:, 4] # 5th row
```

```
# categorical to numerical
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
colTfrm = ColumnTransformer(transformers = [("encoder", OneHotEncoder(), [3])], remainder="passthrough" )
X_encoded = np.array(colTfrm.fit_transform(X))
        #  last column is now replaced with dummy colums (1st 3 colmns)

# Avoiding dummy-var trap: omit one dummy varable
X_go = X_encoded[:, 1:]         # select all columns starting from 2nd column
y_go = np.array(y) #converting Dataframe to Vector/Array


# split dataset to Train and Test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_go, y_go, test_size = 0.2, random_state = 0)


# Fitting multiple linear regression on traing set
from sklearn.linear_model import LinearRegression
regResor = LinearRegression()
regResor.fit(X_train, y_train)

# predict on the test-set X_test
y_pred = regResor.predict(X_test)


# python prctc_mul_lin_rgsn.py
```

Solution

```
# Multiple Linear Regression

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('50_Startups.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values      # last column
print(X)

# Encoding categorical data
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [3])], remainder='passthrough')
X = np.array(ct.fit_transform(X))
print(X)

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

# Training the Multiple Linear Regression model on the Training set
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Predicting the Test set results
y_pred = regressor.predict(X_test)
np.set_printoptions(precision=2)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
```

🛠 To print in the cmd for comparison

```
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
```

2.2.9 Backward Elimination

When we built this model we actually used all the **independent variables**. But among these **independent variables** there are some that are ***highly statistically significant***. That means that if we removed this ***non statistically significant variables*** from the model we would still get some amazing predictions.

❒ ***Need for Backward Elimination: An optimal Multiple Linear Regression model:*** In the previous section, we discussed and successfully created our ***Multiple Linear Regression*** model, where we took ***4 independent variables*** (R&D spend, Administration spend, Marketing spend, and state (dummy variables)) and one dependent variable (Profit).

☝ But that model is ***not optimal***, as we have included all the ***independent variables*** and do not know ***which independent variable*** is most affecting and which one is the ***least affecting*** for the prediction.

☝ **Unnecessary features increase the complexity** of the model. Hence it is good to have only the ***most significant features*** and keep our model simple to get the better result.

☞ So, in order to optimize the performance of the model, we will use the ***Backward Elimination method***. This process is used to optimize the performance of the MLR model as it will only include the ***most affecting feature*** and **remove** the ***least affecting feature***. Let's start to apply it to our MLR model.

```
#Checking the score
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
print('\n\n------------ Train Score: ', regResor.score(X_train, y_train))
print('\n\n------------ Test Score: ', regResor.score(X_test, y_test))

#building the optimal model using Backward elimination
import statsmodels.formula.api as smf
# X_opt = np.append(arr = X_go, values = np.ones(shape = (50, 1)).astype(int), axis =1)
# 50 for row and 1 for column (row, column)
# convert to int type
# set axis = 1: column 0:row
X_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1) # interchange the columns
```

➲ ***Step 0: Install statsmodels library***

```
pip install statsmodels
```

Check the score:

```
[[103015.20159796 103282.38      ]
 [132582.27760815 144259.4       ]
 [132447.73845175 146121.95      ]
 [ 71976.09851258  77798.83      ]
 [178537.48221056 191050.39      ]
 [116161.24230166 105008.31      ]
 [ 67851.69209676  81229.06      ]
 [ 98791.73374687  97483.56      ]
 [113969.43533013 110352.25      ]
 [167921.06569551 166187.94      ]]


------------ Train Score:  0.9501847627493607

------------ Test Score:  0.9347068473282436
```

☝ NOTE: The difference between both scores is 0.0154. On the basis of this score, we will estimate the effect of features on our model after using the Backward elimination process.

➲ ***Step: 1 Preparation of Backward Elimination:***

i. ***Importing the library:*** Firstly, we need to import the statsmodels.formula.api library, which is used for the estimation of various statistical models such as OLS(Ordinary Least Square). Below is the code for it:

```
import statsmodels.formula.api as smf
```

ii. ***Adding a column in matrix of features:*** As we can check in our MLR equation (a), there is one constant term b_0, but this term is not present in our matrix of features, so we need to add it manually. We will add a column having values $x_0 = 1$ associated with the constant term b_0.

```
X_pre_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1)
```

☞ Library that we use here to build a linear regression models is ***LinearRegression***, its definitely included the fact (autometically) that ***there is a constant zero*** in ***MLR equation*** but that's actually not the case for this ***statsmodels*** **library** which we will use to compute the values and evaluate the ***Statistical Significance*** of our **independent variables**.

- So that's why we need to add a ***constant column*** (Colum of ***1's***). It will correspond to our constant b_0. That's how our ***statsmodels*** **library** will understand the correct **MLR equn**.

- To add this, we will use **append()** function of **Numpy library** (***np*** which we have already imported into our code), and will assign a value of 1. Below is the code for it.

```
X_pre_opt = np.append(arr = X_go, values = np.ones(shape = (50, 1)).astype(int), axis =1)
```

⇨ ***"values = ":*** So the array that we're going to import here will be a ***matrix of 50*** 1's. So it will be an array of 50 1's `[1, 1, 1, . . . , 1]`. There is actually a trick for that. And it is

```
np.ones(shape = (50, 1)).astype(int)
```

⇨ ***shape():*** is the ***shape of the matrix*** of what we want to create as (row, column). ***astype(int)*** is needed otherwise we will get a ***data type error***.

⇨ ***axis =1:*** The last argument here which is axis because you can use the ***append()*** function here to add either a ***line*** of these values or ***column*** of these values to the matrix. We need to specify if we want to add a column or line.

- If we want to add a ***line/row*** then its **axis =0**.
- If you want to ***column*** then its **axis =1**.

Notice the last column is our Constant column.

	0	1	2	3	4	5
0	0	1	165349	136898	471784	1
1	0	0	162598	151378	443899	1
2	1	0	153442	101146	407935	1
3	0	1	144372	118672	383200	1
4	1	0	142107	91391.8	366168	1
5	0	1	131877	99814.7	362861	1
6	0	0	134615	147199	127717	1
7	1	0	130298	145530	323877	1
8	0	1	120543	148719	311613	1
9	0	0	123335	108679	304982	1

Format Resize Background color

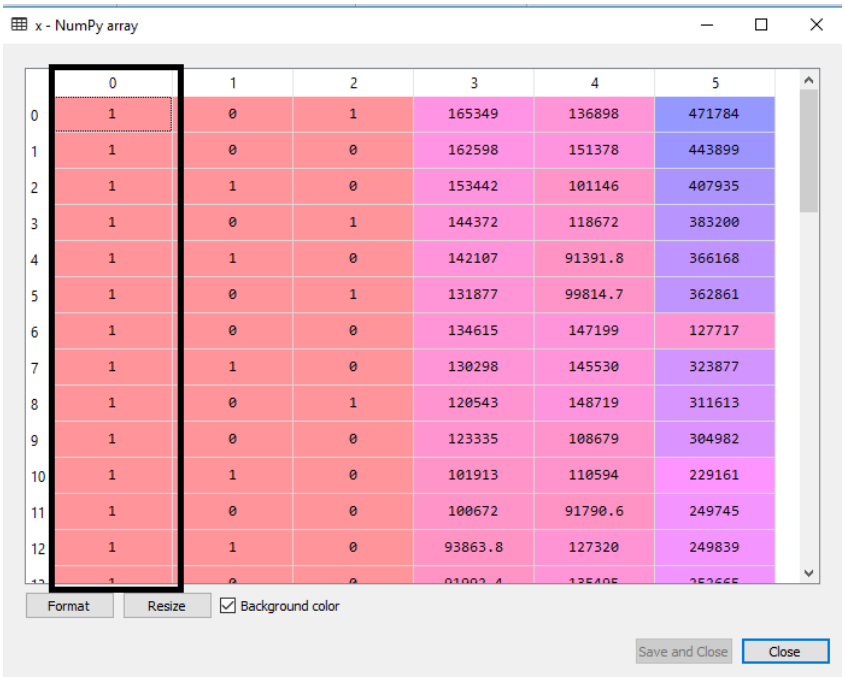
⇨ Now we appended a column to our ***feature matrix***. But the constant column is added to the last. We want constant column at the first column.

⇨ So we need to interchange between ***arr*** and ***values*** attributes of **append()**: Then finally,

```
# interchange the columns
X_pre_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1)
```

```
#building the optimal model using Backward elimination
import statsmodels.formula.api as smf
# X_opt = np.append(arr = X_go, values = np.ones(shape = (50, 1)).astype(int), axis =1)
# 50 for row and 1 for column (row, column)
# convert to int type
# set axis = 1: column 0:row
X_pre_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1) # interc
hange the columns
```

☞ ***Output:*** By executing the above line of code, a new column will be added into our matrix of features, which will have all values equal to 1. We can check it by clicking on the x dataset under the variable explorer option.



	0	1	2	3	4	5
0	1	0	1	165349	136898	471784
1	1	0	0	162598	151378	443899
2	1	1	0	153442	101146	407935
3	1	0	1	144372	118672	383200
4	1	1	0	142107	91391.8	366168
5	1	0	1	131877	99814.7	362861
6	1	0	0	134615	147199	127717
7	1	1	0	130298	145530	323877
8	1	0	1	120543	148719	311613
9	1	0	0	123335	108679	304982
10	1	1	0	101913	110594	229161
11	1	0	0	100672	91790.6	249745
12	1	1	0	93863.8	127320	249839

☞ As we can see in the above output image, the first column is added successfully, which corresponds to the constant term of the MLR equation.

➲ ***Step 2 & 3:*** In this new ***X_opt*** we specify all the indexes of the columns (all 6 columns) ***Explicitly***. And then we eliminate one by one.

```
# --------- iteration 1 ------------

# ------------ step 2 : fit with OLS -------------------
X_opt = X_pre_opt[:, [0, 1, 2, 3, 4, 5]]  # new vector whiuch will  be optimized
# Lets we set SL = 0.05 explicitly. For our learning pupose
# Re-fit  with new regressor. Used OLS "Ordinary Least Squares"
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # fitting with OLS

# ------------ step 3 : inspect p-values -------------------
print(regressor_OLS.summary()) # to inspect the p-valuse
```

i. Firstly we will create a **new feature vector *X_opt***, which will only contain a set of independent features that are significantly affecting the dependent variable.

ii. Next, as per the **Backward Elimination process**, we need to choose a **significant level(0.5)**, and then need to fit the model with all possible predictors. So for fitting the model, we will create a ***regressor_OLS*** object of new ***class OLS*** of ***statsmodels*** library. Then we will fit it by using the ***fit()*** method.

iii. Next we need ***p-value*** to compare with ***SL*** value, so for this we will use ***summary()*** method to get the summary table of all the values.

```
#================== building the optimal model using Backward elimination ====================

# ----------- step 1 : Preprosecc for OLS ------------
# import statsmodels.formula.api as smf --------------------- LEGACY CODE
import statsmodels.api as smf
X_pre_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1)

# ------------ step 2 : fit with OLS -------------------
X_opt = X_pre_opt[:, [0, 1, 2, 3, 4, 5]]  # new vector whiuch will  be optimized
# Lets we set SL = 0.05 explicitly. For our learning pupose
# Re-fit  with new regressor. Used OLS "Ordinary Least Squares"
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # fitting with OLS

# ------------ step 3 : inspect p-values -------------------
print(regressor_OLS.summary()) # to inspect the p-valuse
```

```
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const       5.013e+04   6884.820      7.281      0.000    3.62e+04     6.4e+04
x1           198.7888   3371.007      0.059      0.953   -6595.030    6992.607
x2           -41.8870   3256.039     -0.013      0.990   -6604.003    6520.229
x3             0.8060      0.046     17.369      0.000       0.712       0.900
x4            -0.0270      0.052     -0.517      0.608      -0.132       0.078
x5             0.0270      0.017      1.574      0.123      -0.008       0.062
==============================================================================
```

The successive iterations are given below:

```python
#================= building the optimal model using Backward elimination ====================

# ---------- step 1 : Preprosecc for OLS ------------
# import statsmodels.formula.api as smf --------------------- LEGACY CODE
import statsmodels.api as smf
# X_opt = np.append(arr = X_go, values = np.ones(shape = (50, 1)).astype(int), axis =1)
# 50 for row and 1 for column (row, column)
# convert to int type
# set axis = 1: column 0:row
X_pre_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1) # interc
hange the columns


# --------- iteration 1 ------------

# ------------ step 2 : fit with OLS -------------------
X_opt = X_pre_opt[:, [0, 1, 2, 3, 4, 5]]  # new vector whiuch will  be optimized
# Lets we set SL = 0.05 explicitly. For our learning pupose
# Re-fit  with new regressor. Used OLS "Ordinary Least Squares"
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # fitting with OLS

# ------------ step 3 : inspect p-values -------------------
print(regressor_OLS.summary()) # to inspect the p-valuse

# --------- iteration 2 ------------
X_opt = X_pre_opt[:, [0, 1, 3, 4, 5]]  # removed 3rd column (x2 of iteration 1's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-valuse

# --------- iteration 3 ------------
X_opt = X_pre_opt[:, [0, 3, 4, 5]]  # removed 2nd column (x1 of iteration 2's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-value


# --------- iteration 4 (can be final iteartion) ------------
X_opt = X_pre_opt[:, [0, 3, 5]]  # removed 3rd column (x2 of iteration 3's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-value


# --------- iteration 5 (final iteartion) ------------
X_opt = X_pre_opt[:, [0, 3]]  # removed 3rd column (x2 of iteration 4's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-value

# python prctc_mul_lin_rgsn.py
```

Iteration 4

```
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const       4.698e+04   2689.933     17.464      0.000    4.16e+04    5.24e+04
x1             0.7966      0.041     19.266      0.000       0.713       0.880
x2             0.0299      0.016      1.927      0.060      -0.001       0.061
==============================================================================
Omnibus:                       14.677   Durbin-Watson:                   1.257
Prob(Omnibus):                  0.001   Jarque-Bera (JB):               21.161
Skew:                          -0.939   Prob(JB):                     2.54e-05
Kurtosis:                       5.575   Cond. No.                     5.32e+05
==============================================================================
```

Iteration 5

```
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const       4.903e+04   2537.897     19.320      0.000    4.39e+04    5.41e+04
x1             0.8543      0.029     29.151      0.000       0.795       0.913
==============================================================================
Omnibus:                       13.727   Durbin-Watson:                   1.116
Prob(Omnibus):                  0.001   Jarque-Bera (JB):               18.536
Skew:                          -0.911   Prob(JB):                     9.44e-05
Kurtosis:                       5.361   Cond. No.                     1.65e+05
==============================================================================
```

➲ ***Final conclusion:*** After **4th** and **5th** iteration, we inspected the p-values. At **4th** iteration, we end-up with ***R&D spend*** is definitely a very powerful predictor of the profit and definitely has a high statistical effect impact on the dependent variable profit.

- ➢ And as the ***second independent variable*** is with enough ***statistical effect impact*** is ***Marketing Spend***, we can see that the p-value is **0.06** that is **6%**.
- ➢ But ***Marketing Spend*** actually slightly above the ***5% significance level*** that we set. If we set another SL of 10% for example, we would have kept this independent viable.
- ➢ If you want to follow strictly the framework like the ***Backward Elimination Algorithm*** we need to remove this independent variable.

➧ We will use later some other metrics to make a better decision about that, when we will tack about ***Improving The Models Performance***. In future we'll use ***R-squared*** and ***Adj. R-squared*** to calculate this significance. *[Section 7: Evaluating Regression Models Performance].*

NOTE:

We cannot have a ***0 p-value*** but it's just so small like `0.000001` type number.

All code at Once

```
# Funamental Libraries
import matplotlib as plt
import pandas as pd
import numpy as np

# import dataset
dataSet = pd.read_csv("50_Startups.csv")
X = dataSet.iloc[:, :-1] #all rows except last
y = dataSet.iloc[:, 4] # 5th row


# categorical to numerical
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
colTfrm = ColumnTransformer(transformers = [("encoder", OneHotEncoder(), [3])], remainder="passthrough" )
X_encoded = np.array(colTfrm.fit_transform(X))
        #  last column is now replaced with dummy colums (1st 3 colmns)

# Avoiding dummy-var trap: omit one dummy varable
X_go = X_encoded[:, 1:]         # select all columns starting from 2nd column
y_go = np.array(y) #converting Dataframe to Vector/Array


# split dataset to Train and Test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_go, y_go, test_size = 0.2, random_state = 0)


# Fitting multiple linear regression on traing set
from sklearn.linear_model import LinearRegression
regResor = LinearRegression()
regResor.fit(X_train, y_train)

# predict on the test-set X_test
y_pred = regResor.predict(X_test)


#Checking the score
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
print('\n\n------------ Train Score: ', regResor.score(X_train, y_train))
print('\n\n------------ Test Score: ', regResor.score(X_test, y_test))

#================= building the optimal model using Backward elimination ====================

# ----------- step 1 : Preprosecc for OLS ------------
# import statsmodels.formula.api as smf ---------------------- LEGACY CODE
import statsmodels.api as smf
# X_opt = np.append(arr = X_go, values = np.ones(shape = (50, 1)).astype(int), axis =1)
# 50 for row and 1 for column (row, column)
# convert to int type
# set axis = 1: column 0:row
X_pre_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1) # interchange the columns


# --------- iteration 1 ------------

# ------------ step 2 : fit with OLS -------------------
X_opt = X_pre_opt[:, [0, 1, 2, 3, 4, 5]]  # new vector whiuch will  be optimized
```

```
# Lets we set SL = 0.05 explicitly. For our learning pupose
# Re-fit  with new regressor. Used OLS "Ordinary Least Squares"
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # fitting with OLS

# ------------ step 3 : inspect p-values -------------------
print(regressor_OLS.summary()) # to inspect the p-valuse

# --------- iteration 2 ------------
X_opt = X_pre_opt[:, [0, 1, 3, 4, 5]]  # removed 3rd column (x2 of iteration 1's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-valuse

# --------- iteration 3 ------------
X_opt = X_pre_opt[:, [0, 3, 4, 5]]  # removed 2nd column (x1 of iteration 2's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-valuse


# --------- iteration 4 (can be final iteartion) ------------
X_opt = X_pre_opt[:, [0, 3, 5]]  # removed 3rd column (x2 of iteration 3's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-valuse


# --------- iteration 5 (final iteartion) ------------
X_opt = X_pre_opt[:, [0, 3]]  # removed 3rd column (x2 of iteration 4's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-valuse

# python prctc_mul_lin_rgsn.py
```

Implement Automatic Backward Elimination

```
# Fundamental Libraries
import matplotlib as plt
import pandas as pd
import numpy as np

# import dataset
dataSet = pd.read_csv("50_Startups.csv")
X = dataSet.iloc[:, :-1] #all rows except last
y = dataSet.iloc[:, 4] # 5th row


# categorical to numerical
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
colTfrm = ColumnTransformer(transformers = [("encoder", OneHotEncoder(), [3])], remainder="passthrough" )
X_encoded = np.array(colTfrm.fit_transform(X))
        #  last column is now replaced with dummy colums (1st 3 colmns)

# Avoiding dummy-var trap: omit one dummy varable
X_go = X_encoded[:, 1:]         # select all columns starting from 2nd column
y_go = np.array(y) #converting Dataframe to Vector/Array


# split dataset to Train and Test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_go, y_go, test_size = 0.2, random_state = 0)


# Fitting multiple linear regression on traing set
from sklearn.linear_model import LinearRegression
regResor = LinearRegression()
regResor.fit(X_train, y_train)

# predict on the test-set X_test
y_pred = regResor.predict(X_test)


#Checking the score
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
print('\n\n------------ Train Score: ', regResor.score(X_train, y_train))
print('\n\n------------ Test Score: ', regResor.score(X_test, y_test))

#================= building the optimal model using Backward elimination ====================
"""
# ----------- step 1 : Preprosecc for OLS ------------
# import statsmodels.formula.api as smf ---------------------- LEGACY CODE
import statsmodels.api as smf
```

```
# X_opt = np.append(arr = X_go, values = np.ones(shape = (50, 1)).astype(int), axis =1)
# 50 for row and 1 for column (row, column)
# convert to int type
# set axis = 1: column 0:row
X_pre_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1) # interchange the columns


# --------- iteration 1 ------------

# ------------ step 2 : fit with OLS -------------------
X_opt = X_pre_opt[:, [0, 1, 2, 3, 4, 5]]  # new vector whiuch will  be optimized
# Lets we set SL = 0.05 explicitly. For our learning pupose
# Re-fit  with new regressor. Used OLS "Ordinary Least Squares"
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # fitting with OLS

# ------------ step 3 : inspect p-values -------------------
print(regressor_OLS.summary()) # to inspect the p-valuse

# --------- iteration 2 ------------
X_opt = X_pre_opt[:, [0, 1, 3, 4, 5]]  # removed 3rd column (x2 of iteration 1's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-valuse

# --------- iteration 3 ------------
X_opt = X_pre_opt[:, [0, 3, 4, 5]]  # removed 2nd column (x1 of iteration 2's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-valuse


# --------- iteration 4 (can be final iteartion) ------------
X_opt = X_pre_opt[:, [0, 3, 5]]  # removed 3rd column (x2 of iteration 3's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-valuse


# --------- iteration 5 (final iteartion) ------------
X_opt = X_pre_opt[:, [0, 3]]  # removed 3rd column (x2 of iteration 4's X_opt)
regressor_OLS = smf.OLS(endog = y_go, exog=X_opt).fit() # Re-fit with OLS
print(regressor_OLS.summary()) # to inspect the p-valuse

"""

# ---------- implement automatic Backward Elimination: No manual iteration is needed ----------------
import statsmodels.api as sm
def backwardElimination(x, sl):
    numVars = len(x[0])
    for i in range(0, numVars):
        regressor_OLS = sm.OLS(endog = y_go, exog=x).fit()
        # picking the max p-value
        maxPval = max(regressor_OLS.pvalues).astype(float)
        if maxPval >= sl:
            for j in range(0, numVars - i):
                # deleting the clumn
                if (regressor_OLS.pvalues[j].astype(float) == maxPval):
                    x = np.delete(x, j, 1) # deletes j-th column. "1" is used for "column". To delet "row" use "0"
    print(regressor_OLS.summary())
    return x


SL = 0.05
X_pre_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1)
X_opt = X_pre_opt[:, [0, 1, 2, 3, 4, 5]]
X_Modeled = backwardElimination(X_opt, SL)


# python prctc_mul_lin_rgsn.py
```

☞ Here ***len(x[0])*** is the length of the **first row** , which is actually **No. of columns**. (No. of rows is ***len(x)***)

Numpy Delete

```
# Python Program illustrating
# numpy.delete()

import numpy as geek

#Working on 1D
```

```
arr = geek.arange(12).reshape(3, 4)
print("arr : \n", arr)
print("Shape : ", arr.shape)

# deletion row from 2D array
a = geek.delete(arr, 1, 0)
'''
       [[ 0  1  2  3]
        [ 4  5  6  7] -> deleted
        [ 8  9 10 11]]
'''
print("\ndeleteing arr 2 times : \n", a)
print("Shape : ", a.shape)

# deletion column from 2D array
a = geek.delete(arr, 1, 1)
'''
       [[ 0  1*  2  3]
        [ 4  5*  6  7]
        [ 8  9* 10 11]]
             ^
             Deletion
'''
print("\ndeleteing arr 2 times : \n", a)
print("Shape : ", a.shape)
```

New model After Backward Elimination Feature selection

```
# After Backward Elimination Feature selection
# Building Multiple Linear Regression model by only using R&D spend
import matplotlib as plt
import pandas as pd
import numpy as np

# import dataset
dataSet = pd.read_csv("50_Startups.csv")
X_bak_eli = dataSet.iloc[:, 0].values # R&D spend
y_bak_eli = dataSet.iloc[:, 4].values # 5th row


#converting Dataframe to Vector/Array
X_go = np.array(X_bak_eli)
y_go = np.array(y_bak_eli)


# split dataset to Train and Test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_go, y_go, test_size = 0.2, random_state = 0)


# Fitting multiple linear regression on traing set
from sklearn.linear_model import LinearRegression
regResor = LinearRegression()
# Reshape your data either using array.reshape(-1, 1) if your data has a single feature
regResor.fit(X_train.reshape(-1, 1), y_train)

# predict on the test-set X_test
y_pred = regResor.predict(X_test.reshape(-1, 1))


#Checking the score
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
print('\n\n------------ Train Score: ', regResor.score(X_train.reshape(-1, 1), y_train))
```

```
print('\n\n------------ Test Score: ', regResor.score(X_test.reshape(-1, 1), y_test))

# python prctc_optimized_mul_lin_rgsn.py
```

Comparison between two models Before and after feature selection

[[**103015.20159796** 103282.38] [**132582.27760816** 144259.4] [**132447.73845175** 146121.95] [**71976.09851259** 77798.83] [**178537.48221054** 191050.39] [**116161.24230163** 105008.31] [**67851.69209676** 81229.06] [**98791.73374688** 97483.56] [**113969.43533012** 110352.25] [**167921.0656955** 166187.94]] ------------ Train Score: 0.9501847627493607 ------------ Test Score: 0.9347068473282949	[[**104667.27805998** 103282.38] [**134150.83410578** 144259.4] [**135207.80019517** 146121.95] [**72170.54428856** 77798.83] [**179090.58602508** 191050.39] [**109824.77386586** 105008.31] [**65644.27773757** 81229.06] [**100481.43277139** 97483.56] [**111431.75202432** 110352.25] [**169438.14843539** 166187.94]] ------------ Train Score: 0.9449589778363044 ------------ Test Score: 0.9464587607787219
➲ Difference between both scores is 0.0154	➲ Difference between both scores is **.00149**.

Chapter 2 : Section 3

Polynomial Regression

Introduction to Polynomial Regressions

From this section we are making some new kind of ***regressors*** which are slightly more ***advanced regressors*** especially for ***SVR*** or ***random Forrest***. This polynomial regression model that we're about to build right now is **not that much advanced** compared to ***simple regression*** and ***multiple regression*** because we will just add a *polynomial term* in the *multiple linear regression* equation. But ***SVR*** or ***Random Forrest*** will be based on more complex theory.

2.3.1 Polynomial Regression

Polynomial Regression is a regression algorithm that models the relationship between a ***dependent(y)*** and ***independent variable(x)*** as **nth degree polynomial**. The Polynomial Regression equation is given below:

$$y = b_0 + b_1x + b_2x^2 + b_2x^3 + \ldots\ldots b_nx^n$$

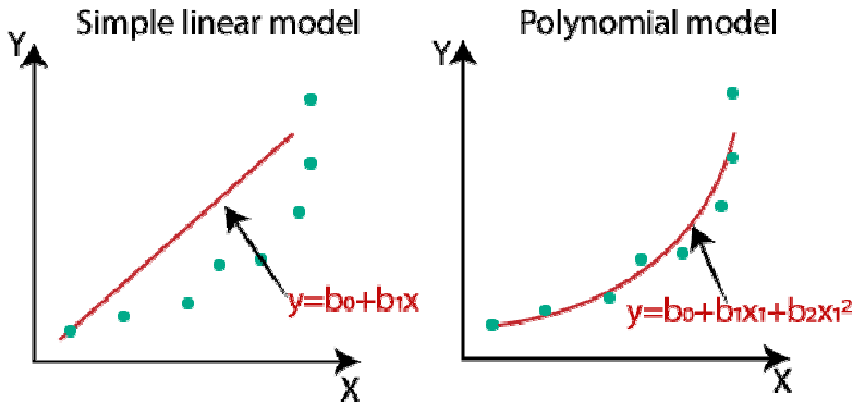
Notice there is only one feature x is present.

- ☞ It is also called the **special case of Multiple Linear Regression** in ML. Because we add some polynomial terms to the ***Multiple Linear regression*** equation to convert it into ***Polynomial Regression***.

- ☝ It is a *linear model* with some modification in order to *increase* the *accuracy*. ***Linearity*** refers to the ***coefficients*** (because they are the unknown here) ***rather than feature variables***.
- ☝ The dataset used in *Polynomial regression* for training is of *non-linear nature*. Eg: Population growth.
- ☝ It makes use of a *linear regression* model to fit the *complicated* and *non-linear functions* and datasets.

- ❒ In Polynomial regression, the original features are converted into Polynomial features of required degree (2,3,..,n) and then modeled using a linear model.

- ❒ ***Need for Polynomial Regression:*** If we apply a ***linear model*** on a ***linear dataset***, then it provides us a good result as we have seen in ***Simple Linear Regression***, but if we apply the same model without any modification on a ***non-linear dataset***, then it will produce a ***drastic*** output. Due to which ***loss function*** will ***increase***, the ***error rate*** will be ***high***, and ***accuracy*** will be ***decreased***.
  - ☞ So for such cases, where ***data points are arranged in a non-linear fashion***, we need the ***Polynomial Regression model***. We can understand it in a better way using the below comparison diagram of the linear dataset and non-linear dataset.



In the above image, we have taken a dataset which is arranged non-linearly. So if we try to cover it with a linear model, then we can clearly see that it hardly covers any data point. On the other hand, a curve is suitable to cover most of the data points, which is of the Polynomial model.

- ❒ ***Why Linear:*** A **Polynomial Regression algorithm** is also called ***Polynomial LINEAR* Regression** because it *does not depend on the variables*, instead, it *depends on the coefficients*, which are arranged in a linear fashion (Generally polynomial equations are Non-

❒ ***Equation of the Polynomial Regression Model:***

- ✓ ***Simple Linear Regression equation:*** $y = b_0 + b_1 x$
- ✓ ***Multiple Linear Regression equation:*** $y = b_0 + b_1 x + b_2 x_2 + b_3 x_3 + \ldots + b_n x_n$
- ✓ ***Polynomial Regression equation:*** $y = b_0 + b_1 x + b_2 x^2 + b_3 x^3 + \ldots + b_n x^n$

☞ We can clearly see that ***all three equations are Polynomial equations*** but differ by the degree of variables. The ***Simple*** and ***Multiple Linear equations are also Polynomial equations*** with a ***single degree***, and the ***Polynomial regression equation*** is ***Linear equation*** with the ***nth degree***. (In mathematics Polynomial equations are nonlinear equations. Here in **Regression** it ***does not depend on the variables***). So if we add a degree to our linear equations, then it will be converted into Polynomial Linear equations.

2.3.2 Implementation of Polynomial Regression using Python

Here we will implement the Polynomial Regression using Python. We will understand it by comparing ***Polynomial Regression*** model with the ***Simple Linear Regression*** model. So first, let's understand the problem for which we are going to build the model.

❒ ***Problem Description:*** There is a Human Resource company, which is going to hire a new candidate. The candidate has told his previous salary **160K** per **annum**, and the HR have to check whether he is telling the ***truth*** or ***bluff***.

❒ So to identify this, they only have a dataset of his previous company in which the salaries of the **top 10 positions** are mentioned with their levels. By checking the dataset available, we have found that there is a ***non-linear relationship*** between the ***Position levels*** and the ***salaries***. Our goal is to build a ***Bluffing detector regression model***, so HR can hire an honest candidate. Below are the steps to build such a model.

Position	Level(X-variable)	Salary(Y-Variable)
Business Analyst	1	45000
Junior Consultant	2	50000
Senior Consultant	3	60000
Manager	4	80000
Country Manager	5	110000
Region Manager	6	150000
Partner	7	200000
Senior Partner	8	300000
C-level	9	500000
CEO	10	1000000

❒ ***Steps for Polynomial Regression:*** The main steps involved in ***Polynomial Regression*** are given below:

i. Data ***Pre-processing***
ii. ***Build*** a ***Linear Regression*** model and ***fit*** it to the dataset
iii. ***Build*** a ***Polynomial Regression*** model and ***fit*** it to the dataset
iv. ***Visualize*** the result for ***Linear Regression*** and ***Polynomial Regression*** model.
v. ***Predicting*** the output.

☝ ***Why Linear & Polynomial both:*** Here, we will build the ***Linear regression*** model as well as ***Polynomial Regression*** to see the ***results between the predictions***. And Linear regression model is for ***reference***.

i. ***Step1 - Data Pre-processing:*** The data pre-processing step will remain the same as in previous regression models, except for some changes. In the Polynomial Regression model, we will ***not use feature scaling,*** and also we will ***not split*** our ***dataset*** into training and test set. It has two reasons:

- ✓ The dataset contains very less information which is not suitable to divide it into a test and training set, else our model will not be able to find the correlations between the salaries and levels.
- ✓ In this model, we want very accurate predictions for salary, so the model should have enough information.

Pre-processing step

```
import pandas as pNd
import matplotlib as pLt
import numpy as nPy

dataSet = pNd.read_csv("Position_Salaries.csv")
X = dataSet.iloc[:, 1:2].values
# X = dataSet.iloc[:, [1]].values # same as above
y = dataSet.iloc[:, 2].values
```

- We will consider only two columns (**Salary** and **Levels**). Because " **Levels** " indicate " **Positions**".
- ***Trick to make feature-matrix:*** For x-variable, we have taken parameters as **`[:,1:2]`**, because we want 1 index(levels), and included :2 to make it as a matrix (we could also use **`dataSet.iloc[:, [1]]`**).

data_set - DataFrame

Index	Position	Level	Salary
0	Business Analyst	1	45000
1	Junior Consultant	2	50000
2	Senior Consultant	3	60000
3	Manager	4	80000
4	Country Manager	5	110000
5	Region Manager	6	150000
6	Partner	7	200000
7	Senior Partner	8	300000
8	C-level	9	500000
9	CEO	10	1000000

Format | Resize | Background color | Column min/max | Save and Close | Close

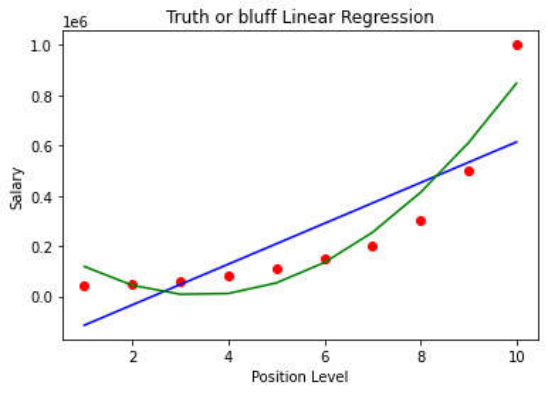
☞ Here we will ***`predict`*** the output for ***`level 6.5`*** because the candidate has ***4+ years'*** experience as a ***regional manager***, so he must be somewhere between ***levels 7 and 6***.

```
# ---------- Simple Linear model for Referace -------------
from sklearn.linear_model import LinearRegression
lin_regressor_1 = LinearRegression()    # object for Simple Linear Regression
lin_regressor_1.fit(X, y)


# ----------- Polynomial regressor --------------------
from sklearn.preprocessing import PolynomialFeatures
# Set the polynomial for X
poly_regressor = PolynomialFeatures(degree= 2)
X_poly = poly_regressor.fit_transform(X) # Generates Polynomial Feature-Matrix
lin_regressor_2 = LinearRegression()    # object for Polinomial Regression
lin_regressor_2.fit(X_poly, y)


# ------------  Visualize the result  --------------

y_lin_pred = lin_regressor_1.predict(X)
# y_poly_pred = lin_regressor_2.predict(X): Does not work- Have to use Polynomial Feature-Matrix
y_poly_pred = lin_regressor_2.predict(X_poly)
pLt.scatter(X, y, color = "red")
pLt.plot(X, y_lin_pred, color = "blue")
pLt.title('Truth or bluff Linear Regression')
pLt.xlabel('Position Level')
pLt.ylabel("Salary")
pLt.plot(X, y_poly_pred, color = "green")
pLt.show()
```



ii. ***Step 2 - Building the Linear regression model:***

In building polynomial regression, we will take the ***Linear regression*** model as reference and compare both the ***results***. The code is given below:

```
# ---------- Simple Linear model for Referace -------------
from sklearn.linear_model import LinearRegression
lin_regressor_1 = LinearRegression()    # object for Simple Linear Regression
lin_regressor_1.fit(X, y)
```

In the above code, we have created the **Simple Linear model** using ***lin_regressor_1*** object of ***LinearRegression*** class and fitted it to the dataset variables (X and y).

iii. ***Step 3 - Building the Polynomial regression model:***

Now we will build the Polynomial Regression model, but it will be a little different from the Simple Linear model. Because here we will use ***PolynomialFeatures*** class of ***preprocessing*** library. We are using this class to add some extra features to our dataset.

```
# ----------- Polynomial regressor --------------------
from sklearn.preprocessing import PolynomialFeatures
# Set the polynomial for X
poly_regressor = PolynomialFeatures(degree= 2)
X_poly = poly_regressor.fit_transform(X) # Generates Polynomial Feature-Matrix
lin_regressor_2 = LinearRegression()    # object for Polinomial Regression
lin_regressor_2.fit(X_poly, y)
```

- ***poly_regressor.fit_transform(X)*** used converting our feature matrix into ***Polynomial Feature Matrix***,
- The parameter value ***(degree= 2)*** depends on our choice. We can choose it according to our **Polynomial features**.
- After executing the code, we will get another matrix ***X_poly***, which can be seen under the **variable explorer** option:
- Next, we have used another ***LinearRegression*** object, namely ***lin_regressor_2***, to fit our x_poly vector to the linear model.
- The ***1st column*** is the "***constant-Column***" Recall Multiple-Linear-regression (this column is automatically created). ***2nd column*** is for ***1st-degrre*** term and 3rd column is the ***2nd –degree*** term.

	0	1	2
0	1	1	1
1	1	2	4
2	1	3	9
3	1	4	16
4	1	5	25
5	1	6	36
6	1	7	49
7	1	8	64
8	1	9	81
9	1	10	100

Polynomial Feature-Matrix: This ***poly_regressor*** object is going to be a ***transformer tool*** that will transform our **matrix of feature *X*** into a ***new matrix of features*** that we're going to call ***X_poly*** which will be a new matrix of features containing not only this independent variable ***x*** but also x^2 **i.e. it actually adds the polynomial terms**. The can be set to any integer (2, 3, 4, ..) using *PolynomialFeatures(**degree= 2**)*.

```
from sklearn.preprocessing import PolynomialFeatures
poly_regressor = PolynomialFeatures(degree= 2) # Set the degree of polynomial for X
X_poly = poly_regressor.fit_transform(X) # Generates Polynomial Feature-Matrix
```

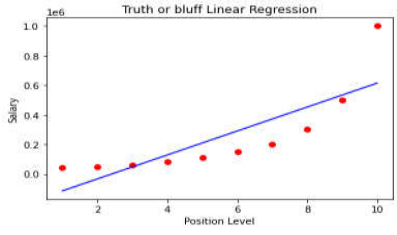
These three lines actually transformed our original ***matrix of features*** **X** into our ***new matrix of features*** containing the original ***independent variable*** **X** and its associated ***polynomial terms***.

Include this X_poly fit into a multiple regression model: In the following two lines we created a new linear regression object ***lin_regressor_2*** that we fitted to this **new matrix *X_poly*** and now **original dependent variable** vector ***y***.

```
lin_regressor_2 = LinearRegression()    # object for Polinomial Regression
lin_regressor_2.fit(X_poly, y)
```

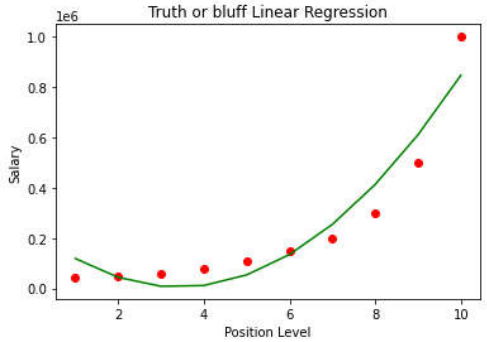
iv. ***Step 4 - Visualizing the result for Linear & Polynomial regression:***

```
    # ======= Visualizing
Simple  Linear ============
y_lin_pred = lin_regressor_1.predict(X)
plt.scatter(X, y, color = "red")
plt.plot(X, y_lin_pred, color = "blue")
plt.title('Truth or bluff Linear Regression')
plt.xlabel('Position Level')
plt.ylabel("Salary")
plt.show()
```



☞ If we consider this output to predict the value of CEO, it will give a salary of approx. 600000$, which is far away from the real value. So we need a curved model to fit the dataset other than a straight line.

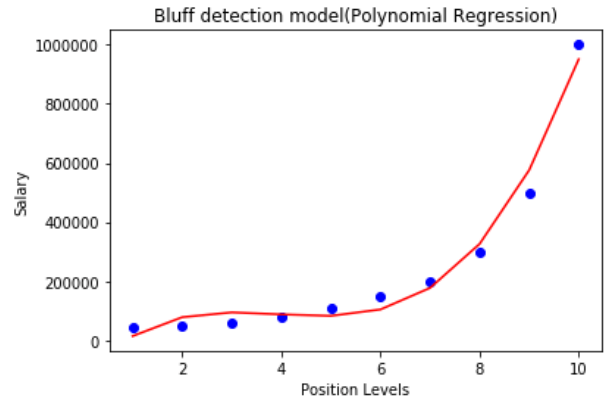
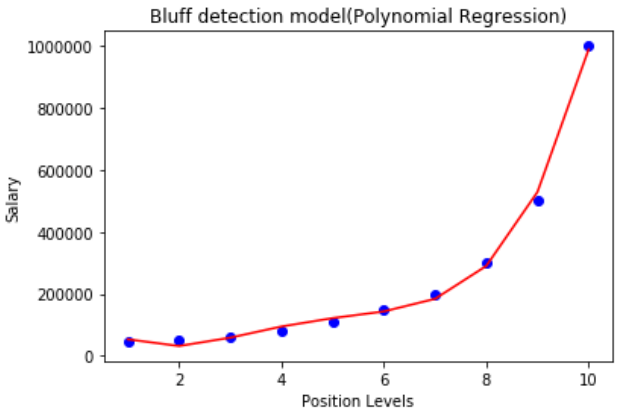
```
    # ======= Polynomial ============
# y_poly_pred = lin_regressor_2.predict(X): Does not work-
 Have to use Polynomial Feature-Matrix
y_poly_pred = lin_regressor_2.predict(X_poly)
pLt.scatter(X, y, color = "red")
pLt.title('Truth or bluff Linear Regression')
pLt.xlabel('Position Level')
pLt.ylabel("Salary")
pLt.plot(X, y_poly_pred, color = "green")
```



🛠 NOTICE: ***X_poly*** is used with ***lin_regressor_2*** and we cannot use ***X***. The reason is ***lin_regressor_1*** is the model built upon ***Single feature X*** but ***lin_regressor_2*** is the model built upon ***extra polynomial terms*** of ***feature*** variable X^2.

```
y_poly_pred = lin_regressor_2.predict(X_poly)
```

Fitting the model by Increasing degree.

For degree= 3: If we change the degree=3, then we will give a more accurate plot, as shown in the below image.	***Degree= 4:*** Hence we can get more accurate results by increasing the degree of Polynomial.
	

V. ***Step – 5: Predicting the final result with the Linear Regression model:***

Now, we will predict the final output using the ***Linear regression*** model to see whether an employee is saying truth or bluff. So, for this, we will use the ***predict()*** method and will pass the value ***6.5***. Below is the code for it:

```
# ----------------- Prediction -------------------
lin_pred = lin_regressor_1.predict([[6.5]])  # Simple linear Regerssion
print(lin_pred)
poly_pred = lin_regressor_2.predict(poly_regressor.fit_transform([[6.5]]))  # Polynl linear Regerssion
print(poly_pred)
```

As we can see, the predicted output for the Polynomial Regression is [158862.45265153], which is much closer to real value hence, we can say that future employee is saying true.

Practiced version

```
# --------------- Import Libraries --------------
import pandas as pNd
import matplotlib.pyplot as pLt
import numpy as nPy



# --------------- Data Preprocessing  -------------
dataSet = pNd.read_csv("Position_Salaries.csv")
X = dataSet.iloc[:, 1:2].values
# X = dataSet.iloc[:, [1]].values # same as above
y = dataSet.iloc[:, 2].values



# ---------- Simple Linear model for Referace -------------
from sklearn.linear_model import LinearRegression
lin_regressor_1 = LinearRegression()    # object for Simple Linear Regression
lin_regressor_1.fit(X, y)



# ---------- Polynomial regressor --------------------
from sklearn.preprocessing import PolynomialFeatures
# Set the polynomial for X
poly_regressor = PolynomialFeatures(degree= 4)
X_poly = poly_regressor.fit_transform(X) # Generates Polynomial Feature-Matrix
lin_regressor_2 = LinearRegression()    # object for Polinomial Regression
lin_regressor_2.fit(X_poly, y)



# ------------  Visualize the result  -------------
    # ======= Simple  Linear ============
y_lin_pred = lin_regressor_1.predict(X)
pLt.scatter(X, y, color = "red")
pLt.plot(X, y_lin_pred, color = "blue")
pLt.title('Truth or bluff ||| Linear Regression')
pLt.xlabel('Position Level')
pLt.ylabel("Salary")
pLt.show()



    # ======= Polynomial ============

# y_poly_pred = lin_regressor_2.predict(X): Does not work- Have to use Polynomial Feature-Matrix
y_poly_pred = lin_regressor_2.predict(X_poly)
pLt.scatter(X, y, color = "red")
pLt.title('Truth or bluff Linear Regression')
pLt.xlabel('Position Level')
pLt.ylabel("Salary")
pLt.plot(X, y_poly_pred, color = "green")
pLt.show()


# ----------------- Prediction -------------------
lin_pred = lin_regressor_1.predict([[6.5]])  # Simple linear Regerssion
print(lin_pred)
poly_pred = lin_regressor_2.predict(poly_regressor.fit_transform([[6.5]]))  # Polynl linear Regerssion
print(poly_pred)



# python prctc_polnm_rgsn.py
```

Instructor version

```
# Polynomial Regression

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Position_Salaries.csv')
X = dataset.iloc[:, 1:-1].values
y = dataset.iloc[:, -1].values

# Training the Linear Regression model on the whole dataset
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)

# Training the Polynomial Regression model on the whole dataset
from sklearn.preprocessing import PolynomialFeatures
poly_reg = PolynomialFeatures(degree = 4)
X_poly = poly_reg.fit_transform(X)
poly_reg.fit(X_poly, y)
lin_reg_2 = LinearRegression()
lin_reg_2.fit(X_poly, y)

# Visualising the Linear Regression results
plt.scatter(X, y, color = 'red')
plt.plot(X, lin_reg.predict(X), color = 'blue')
plt.title('Truth or Bluff (Linear Regression)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()

# Visualising the Polynomial Regression results
plt.scatter(X, y, color = 'red')
plt.plot(X, lin_reg_2.predict(poly_reg.fit_transform(X)), color = 'blue')
plt.title('Truth or Bluff (Polynomial Regression)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()

# Visualising the Polynomial Regression results (for higher resolution and smoother curve)
X_grid = np.arange(min(X), max(X), 0.1)
X_grid = X_grid.reshape((len(X_grid), 1))
plt.scatter(X, y, color = 'red')
plt.plot(X_grid, lin_reg_2.predict(poly_reg.fit_transform(X_grid)), color = 'blue')
plt.title('Truth or Bluff (Polynomial Regression)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()

# Predicting a new result with Linear Regression
lin_reg.predict([[6.5]])

# Predicting a new result with Polynomial Regression
lin_reg_2.predict(poly_reg.fit_transform([[6.5]]))
```

New Template for Non-Linear Regression

Follow the Instructor version

```
# Regression Template

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Position_Salaries.csv')
X = dataset.iloc[:, 1:2].values
y = dataset.iloc[:, 2].values

# Splitting the dataset into the Training set and Test set
"""from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)"""

# Feature Scaling
"""from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
sc_y = StandardScaler()
y_train = sc_y.fit_transform(y_train.reshape(-1,1))"""

# Fitting the Regression Model to the dataset
# Create your regressor here

# Predicting a new result
y_pred = regressor.predict(6.5)

# Visualising the Regression results
plt.scatter(X, y, color = 'red')
plt.plot(X, regressor.predict(X), color = 'blue')
plt.title('Truth or Bluff (Regression Model)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()

# Visualising the Regression results (for higher resolution and smoother curve)
X_grid = np.arange(min(X), max(X), 0.1)
X_grid = X_grid.reshape((len(X_grid), 1))
plt.scatter(X, y, color = 'red')
plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
plt.title('Truth or Bluff (Regression Model)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()
```

Support Vector Regression (SVR)

2.4.1 Implementation of SVR in Python

❒ ***Data Preparation:*** We continue with our previous problem of "Bluff Detection"

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# ----------- data preprocessing -----------------
# Importing the dataset. previous problem of "Bluff Detection"
datASet = pd.read_csv("Position_Salaries.csv")
X = datASet.iloc[:, 1:2].values
y = datASet.iloc[:, 2].values
```

❒ ***Fitting SVR to the dataset:***

We'll just import ***SVR*** class from the sikatLearn ***svm*** library because ***SVR*** is actually a **support vector machine SVM** for ***regression***.

```
# Fitting SVR to the dataset
from sklearn.svm import SVR
regressor = SVR(kernel='rbf')
regressor.fit(X, y)
```

☞ ***Choosing kernel:***

➲ We have many parameters for many ML models. But the most important parameter that we need to focus on is the ***kernel***. The kernel is whether you want a ***linear SVR*** or a ***polynomial SVR*** or ***Gaussian SVR***. ***'linear'***, ***'poly'***, ***'rbf'***, ***'sigmoid'***, ***'precomputed'*** are the most common ***kernels***.

➲ The one we want right now is the **'rbf'** kernel. And why is that? Because we know our problem is non-linear. The *linear kernel* would make a *linear machine model* that would *not* therefore be *appropriate* for a *nonlinear problem*.

➲ And then we have the choice between ***'poly'*** and ***'rbf'***, *both* these kernels *could work* for our problem. But we're going to take the most common one which is the ***Gaussian kernel*** and therefore ***'rbf'*** here.

❒ ***Prediction without scaling:***

In ***y_pred***, we used ***[[6.5]]***, since Parameter must be ***2-D array***.

```
y_prd = regressor.predict([[6.5]])  # [[6.5]], since Parmeter must be 2-D array
print(y_prd)
```

Before scaling is applied

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# ----------- data preprocessing -----------------
# Importing the dataset. previous problem of "Bluff Detection"
datASet = pd.read_csv("Position_Salaries.csv")
X = datASet.iloc[:, 1:2].values
y = datASet.iloc[:, 2].values


# Splitting the dataset into the Training set and Test set: No need here


# Feature Scaling


# Fitting SVR to the dataset
```

```
from sklearn.svm import SVR
regressor = SVR(kernel='rbf')
regressor.fit(X, y)


# ---------------- Visualising the SVR results ----------------
plt.scatter(X, y, color = "red")
plt.plot(X, regressor.predict(X), color = "blue")


# ------------ prediction -----------
"""
Here we will predict the output for level 6.5
because the candidate has 4+ years' experience as a regional manager,
so he must be somewhere between levels 7 and 6.
"""
y_prd = regressor.predict([[6.5]])  # [[6.5]], since Parmeter must be 2-D array
print(y_prd)

# python prctc_SVR.py
```

- ***Feature scaling needed:*** SVR does not apply automatic Feature-Scaling as Simple linear and Multiple linear Model. ***SVR*** is not a common class as ***Linearregression***.

Feature Scaling applied:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ---------- data preprocessing -----------------
# Importing the dataset. previous problem of "Bluff Detection"
datASet = pd.read_csv("Position_Salaries.csv")
X = datASet.iloc[:, 1:2].values
y = datASet.iloc[:, 2].values

# Splitting the dataset into the Training set and Test set: No need here

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
sc_y = StandardScaler()
X_scaled = sc_X.fit_transform(X)
y_scaled = sc_y.fit_transform(y.reshape(-1, 1))
```

- ***New prediction:***
  - ***No fit_transform(6.5) only transform():*** We don't use ***`fit_transfrom()`*** because model is already fitted. We only use ***`transform()`***.

    ```
    sc_X.transform(np.array([[6.5]]))
    ```

  - ***Transform 6.5 to array:*** Since expecting ***2-d-array*** we need to pass ***6.5*** as array the trick is given below:

    ```
    np.array([[6.5]])
    ```

  - ***Reverse Scaling (inverse scale transformation):*** Since is applied and we passed 6.5 as scaled array. The result y_pred is also is scaled output. So we need to ***Reverse scale*** `y_pred`.

    ```
    y_inv_sc = sc_y.inverse_transform(y_prd)
    ```

```
y_prd = regressor.predict(sc_X.transform(np.array([[6.5]])))
print("prediction under scaled data : ", y_prd)
y_inv_sc = sc_y.inverse_transform(y_prd)
print("Reverse scaled prediction : ", y_inv_sc)
```

- Notice in "Visualizing the SVR results" all scaled-dada is used to plot the model.
- SVR fitted to the place where most observations appear.

Practiced version

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ---------- data preprocessing ------------------
# Importing the dataset. previous problem of "Bluff Detection"
datASet = pd.read_csv("Position_Salaries.csv")
X = datASet.iloc[:, 1:2].values
y = datASet.iloc[:, 2].values

# Splitting the dataset into the Training set and Test set: No need here

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
sc_y = StandardScaler()
X_scaled = sc_X.fit_transform(X)
y_scaled = sc_y.fit_transform(y.reshape(-1, 1))

# Fitting SVR to the dataset
from sklearn.svm import SVR
regressor = SVR(kernel='rbf')
# regressor.fit(X, y)
regressor.fit(X_scaled, y_scaled)

# ---------------- Visualising the SVR results ----------------
# Feature scaling is needed
# plt.scatter(X, y, color = 'red')
# plt.plot(X, regressor.predict(X), color = 'blue')

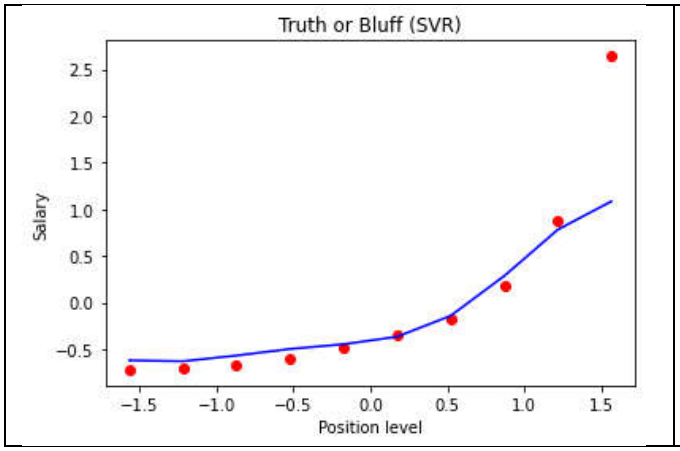
plt.scatter(X_scaled, y_scaled, color = "red")
plt.plot(X_scaled, regressor.predict(X_scaled), color = "blue")
plt.title('Truth or Bluff (SVR)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()

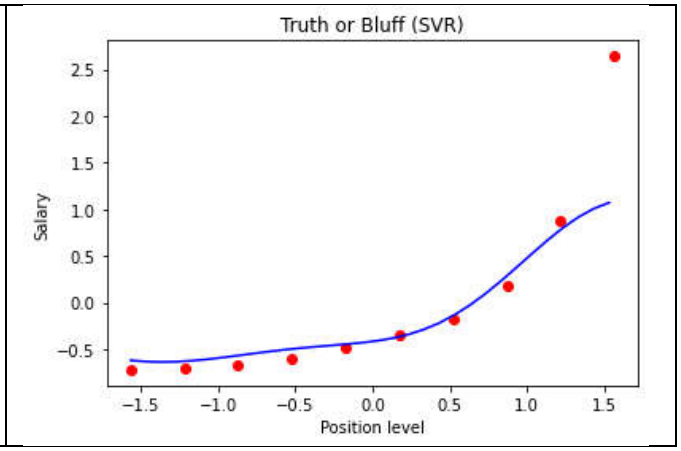
# Visualizing the SVR results (for higher resolution and smoother curve)
X_grid = np.arange(min(X_scaled), max(X_scaled), 0.1)
X_grid = X_grid.reshape((len(X_grid), 1))
plt.scatter(X_scaled, y_scaled, color = "red")
plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
plt.title('Truth or Bluff (SVR)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()



# ------------ prediction -----------
"""
Here we will predict the output for level 6.5
because the candidate has 4+ years' experience as a regional manager,
so he must be somewhere between levels 7 and 6.
"""
# y_prd = regressor.predict([[6.5]])  # [[6.5]], since Parmeter must be 2-D array

# We need to transfom 6.5 in our scaling
# y_prd = regressor.predict(sc_X.transform([[6.5]])) # alternative
y_prd = regressor.predict(sc_X.transform(np.array([[6.5]])))
print("prediction under scaled data : ", y_prd)
y_inv_sc = sc_y.inverse_transform(y_prd)
print("Reverse scaled prediction : ", y_inv_sc)
```





Instructor version: 6.5 is not transformed

```
# Support Vector Regression (SVR)

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Position_Salaries.csv')
X = dataset.iloc[:, 1:-1].values
y = dataset.iloc[:, -1].values

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
sc_y = StandardScaler()
X = sc_X.fit_transform(X)
y = sc_y.fit_transform(y.reshape(-1,1))

# Training the SVR model on the whole dataset
from sklearn.svm import SVR
regressor = SVR(kernel = 'rbf')
regressor.fit(X, y)

# Predicting a new result
y_pred = regressor.predict([[6.5]])
y_pred = sc_y.inverse_transform(y_pred)

# Visualising the SVR results
plt.scatter(X, y, color = 'red')
plt.plot(X, regressor.predict(X), color = 'blue')
plt.title('Truth or Bluff (SVR)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()

# Visualising the SVR results (for higher resolution and smoother curve)
X_grid = np.arange(min(X), max(X), 0.01) # choice of 0.01 instead of 0.1 step because the data is feature scaled
X_grid = X_grid.reshape((len(X_grid), 1))
plt.scatter(X, y, color = 'red')
plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
plt.title('Truth or Bluff (SVR)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()
```

2.4.2 When should I use SVR?

You should use ***SVR*** if a ***linear model like Linear Regression doesn't fit very well*** to your data. This would mean you are dealing with a ***Non Linear Problem***, where your data is not linearly distributed. Therefore in that case ***SVR*** could be a much better ***solution***.

2.4.3 what the heck is SVR?

SVR is a pretty abstract model and besides it is not that commonly used. What you must rather understand is the ***SVM model***, which you will see in ***Chapter-3: Classification***. Then once you understand the ***SVM model***, you will get a better grasp of the ***SVR model***, since the ***SVR*** is simply the ***SVM for Regression***. However we wanted to include SVR in this chapter to give you an extra option in your Machine Learning toolkit.

2.4.4 Why do we need to '`sc_y.inverse_transform`'?

We need the ***inverse_transform()*** method to go back to the ***original scale***. Indeed we applied ***feature scaling*** so we get this scale around ***0*** and if we make a prediction ***without inversing*** the ***scale*** we will get the ***scaled predicted salary***. And of course we want the ***real salary***, not the ***scaled*** one, so we have to use '***sc_Y.inverse_transform***'. Also what is important to understand is that '***transform***' and '***inverse_transform***' are paired methods.

2.4.5 In –R scaling not needed

Because in ***svm()*** function of ***R***, the values are ***automatically scaled***.

2.4.6 No p-values in SVR !!

You couldn't use ***p-value*** because ***SVR*** is ***not*** a ***linear model***, and ***p-values apply only to Linear Models***. Therefore ***feature selection is out of the question***.

- But you could do ***feature extraction***, which you will see in ***Chapter 9 - Dimensionality Reduction***. That you can apply to ***Decision Trees***, and it will reduce the number of your features.

Decision Tree Regression

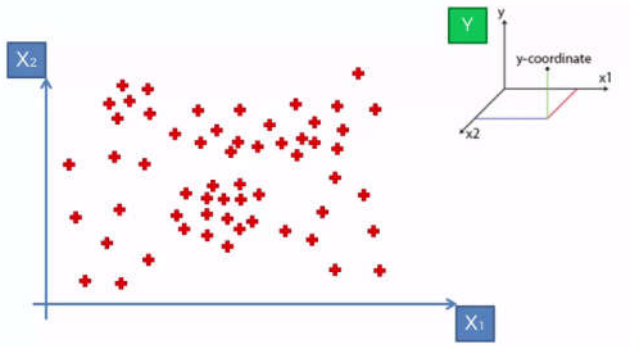
2.5.1 Decision Tree

❒ ***CART (Classification and Regression trees):*** Decision Trees are divided into ***Classification*** and ***Regression*** Trees.

- ☞ ***Regression trees*** are needed when the ***response variable*** is *numeric* or *continuous*.
- ☞ ***Classification trees***, as the name implies are used to ***separate*** the ***dataset*** into classes belonging to the ***response variable***.

❒ ***In this section, we discuss about:*** Decision Tree Regression Model

- ☞ Assume that we've got a ***scatterplot*** which represent our Dataset. Here we've got two independent variables X_1 and X_2. And we're ***predicting*** y as the ***third variable*** which is ***dependent variable***. It is in the 3rd axis and cannot be seen in 2-d.

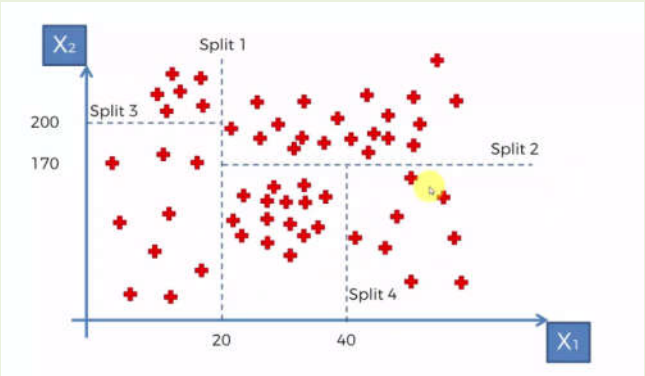


- ☞ We don't actually need to see "**y**" because we need to work with this ***scatterplot*** first, to build our decision tree. And then once we've built it will return to **y**.
- ☞ In X_1X_2 plane all points are the projection of the data points.
- ☞ We need to work with this scatterplot to see how our decision tree is going to be created.

❒ So once you run the ***Regression Tree*** or ***Decision Tree Algorithm*** in ***Regression Sense*** of it, what will happen is your ***scatterplot*** will be ***Split Up*** into ***Segments***. The spitted segments are called ***leaves***. Let's have a look at how an algorithm do that:

☯ Assume that the algorithm would create a split at somewhere around 20 (i.e. your diagram or your ***scatterplot*** now divided into two parts).

i. Everything $X_1 < 20$ and everything else that's $X_1 > 20$.
ii. For $X_1 < 20$, then another split for X_2 is: $X_2 < 200$ and $X_2 > 200$.
iii. For $X_1 > 20$, then other split for X_2 is $X_2 < 170$ and $X_2 > 170$.
iv. Lets again assume for $X_2 < 170$ and for $X_1 > 20$ there is a split for X_1, which is $X_1 < 40$ and $X_1 > 40$



➲ The Scatterplot is given in the Right:

☝ **How** and **where** these ***splits are conducted*** is determined by the ***algorithm***. And it is related to the ***Information Entropy***. When we perform split into our data (create a leaf), the ***amount of information increases*** (it actually adding some value when we group our points). The ***algorithm*** is finding the ***optimal splits*** of our data set into these ***leaves***.

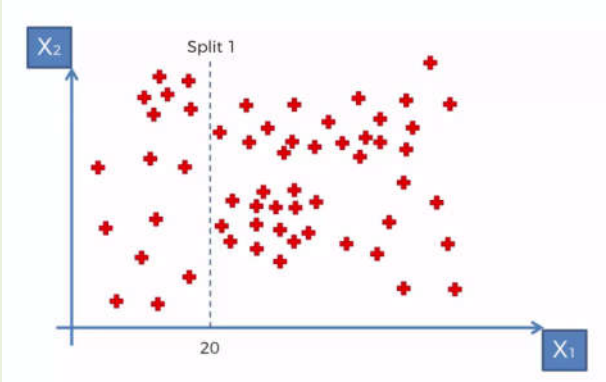
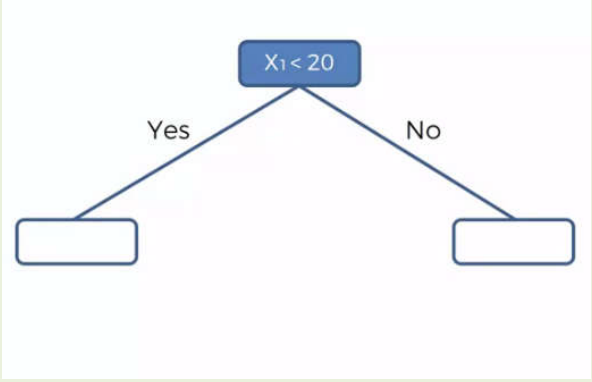
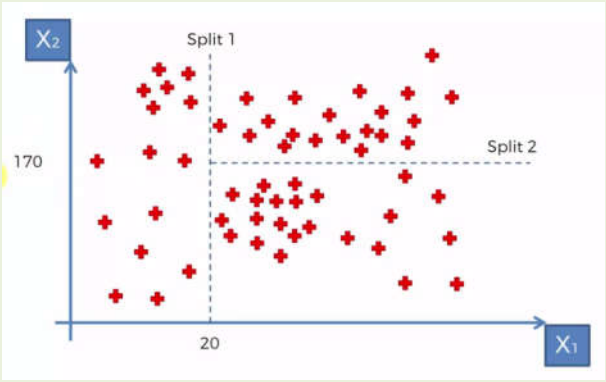
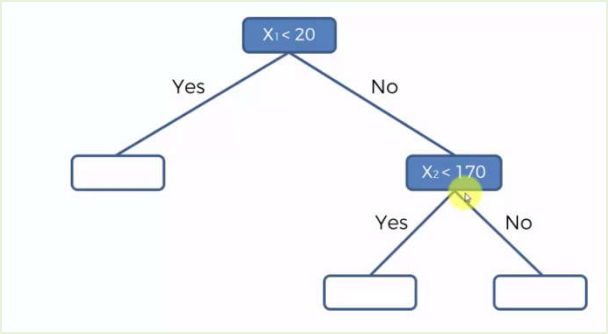
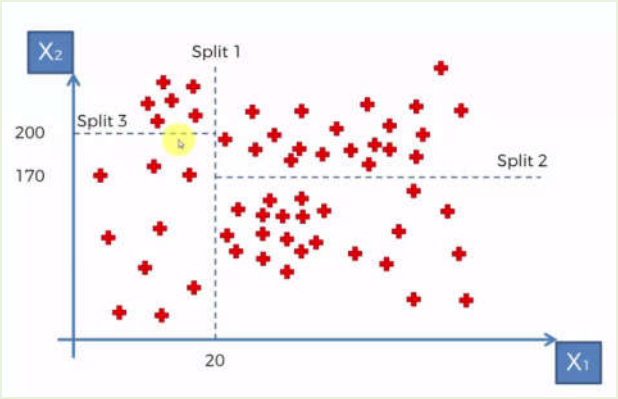
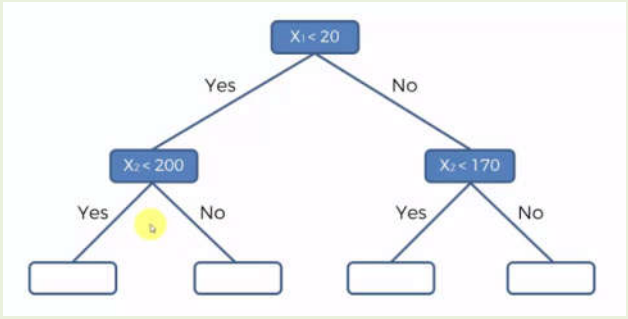
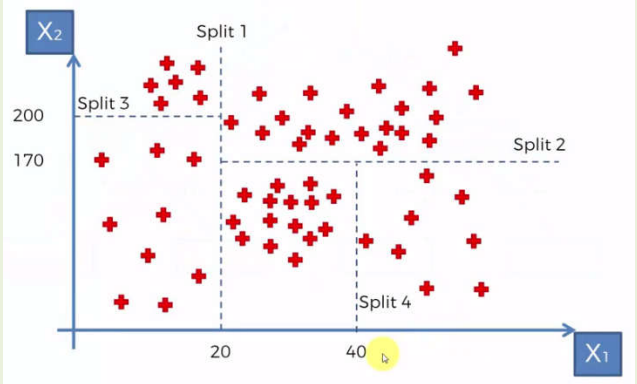
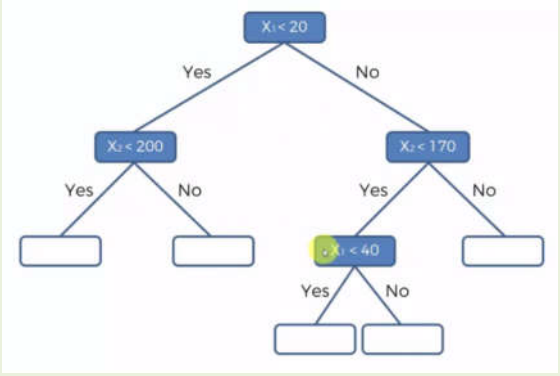
➲ The algorithm knows when to stop and when there is a certain minimum information that needs to be added.

➲ And at certain stage the algorithm cannot add any more information to our ***set-up*** by ***splitting*** these ***leaves***, then it stops. A value is set which determines when to stop (for example `less` than 5% of your ***total points*** in that ***leaf*** and then that leaf wouldn't be created).

So there are different variations and different options. The most important thing is where the splits are happening.

☝ The ***final leaves*** are called *terminal leaves*.

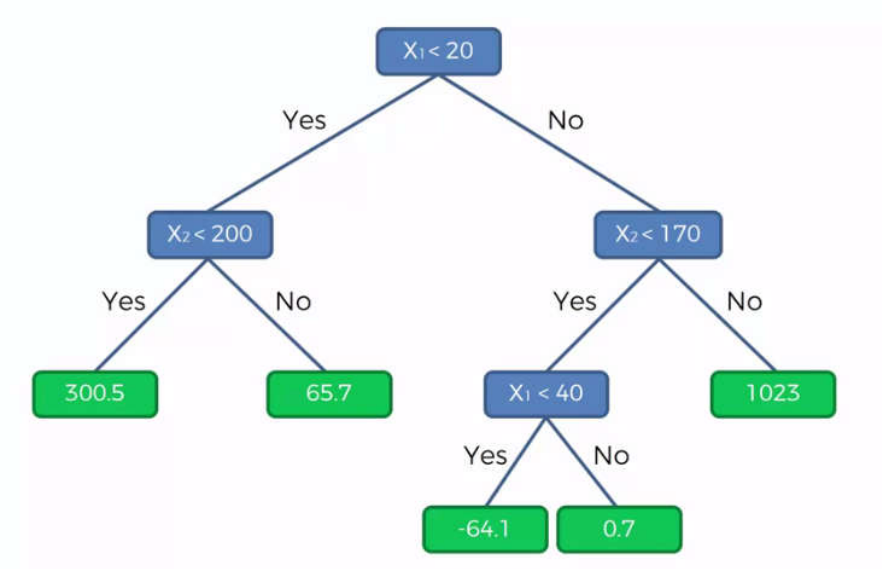
❒ ***The decision tree:*** Now we're going to create these ***splits*** one by one and alongside we're going to actually start drawing our ***decision tree***.

Splitting Data-set	***Decision Tree***
	
	
	
	 Now that's our decision tree.

❒ ***How to predict:*** Next we determine y, the value is determined by taking the average of the values in a terminal-leafs (the last iteration) . i.e. just take the average of ***y*** for all of points of the leaf and that'll be the value that will be assigned to any new point that falls in that terminal.

- ☞ The whole point of this exercise is to `add more information` into our ***chart*** into our ***system*** to better predict ***y***.
- ☞ Our machine learning algorithm just take the average across all of the points and whatever that is wherever you point the new element of data, that is added to our ***data set*** wherever it falls.

☝ So, we've split our diagram up into terminal leaves and the machine learning algorithm has added information nurture into our system. So that we can more accurately predict the value or assign the value of ***y*** to a new coming element. Following is our ***completed decision tree***.



2.5.2 Decision Tree Regression Model practice with Python

Practiced version

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np


# Data Extract
dataSet = pd.read_csv("Position_Salaries.csv")
X = dataSet.iloc[:, 1:2].values
y = dataSet.iloc[:, 2].values


# # Feature-Scaling
# from sklearn.preprocessing import StandardScaler
# sc_x = StandardScaler()
# sc_y = StandardScaler()
# X_scaled = sc_x.fit_transform(X)
# y_scaled = sc_y.fit_transform(y.reshape(-1, 1))


# Data Split : No need for this example

# Fit dataset to model
from sklearn.tree import DecisionTreeRegressor
regressor = DecisionTreeRegressor(random_state= 0)
regressor.fit(X, y)


# Predict
```

```
y_pred = regressor.predict([[6.5]])
print("The predicte value ffor 6.5 is : ", y_pred)

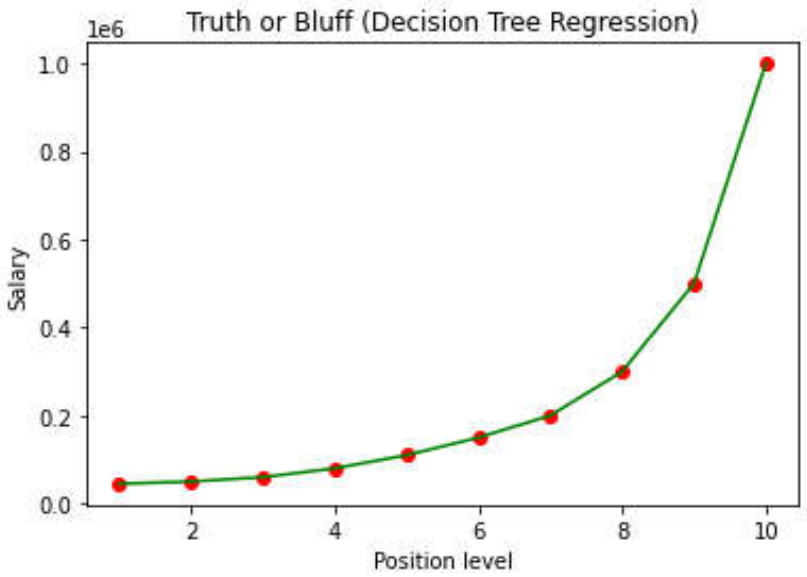
# plot the model
plt.scatter(X, y, color = "red")
plt.plot(X, regressor.predict(X), color = "green")
plt.title("Truth or Bluff (Decision Tree Regression)")
plt.xlabel("Position level")
plt.ylabel("Salary")
```

❒ ***Prediction:*** Our prediction is calculated by

```
y_pred = regressor.predict([[6.5]])
```

➲ Which gives 150,000. This prediction is below 160,000 (the new employee asks). This is not a good prediction, actually its calculated from the mean-value.

The new trap: We get the following figure: In the algorithm of decision tree regression, the entropy in the information increases by ***splitting the independent variables into several intervals***. In the intuition we had two independent variables X_1 and X_2. In this example we have ***only one independent variable***.



So according to algorithm it's taking the average in each interval. So either there are ***infinite intervals*** so that the above graph is so ***smooth*** or we have problem with our ***matplot*** settings. Of course ***Decision Tree algorithm*** not considering infinite intervals, so it's the issue with our plotting.

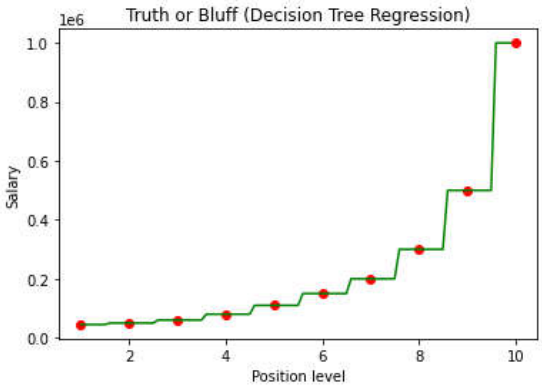
In our previous template, it's only plotting the predictions of the 10 salary's corresponding to the 10 levels and then its joining the predictions by a straight line here because it had no predictions to plot in this interval.

❒ ***Nonlinear and non-continuous regression model:*** Now we're facing a new kind of regression model. It's the nonlinear and non-continuous regression model.

☞ Indeed all the previous regressions were Linear and they were all continuous. But here the decision tree regression model is not continuous and this is the first non-continuous machine model for us.

❒ ***Visualize the non-continuous regression model:*** When we visualize the results for ***Higher Resolution*** and smoother curve we can observe the mean-values and non-continuous regression model. Following is the ***Decision Tree algorithm*** in One-dimension.

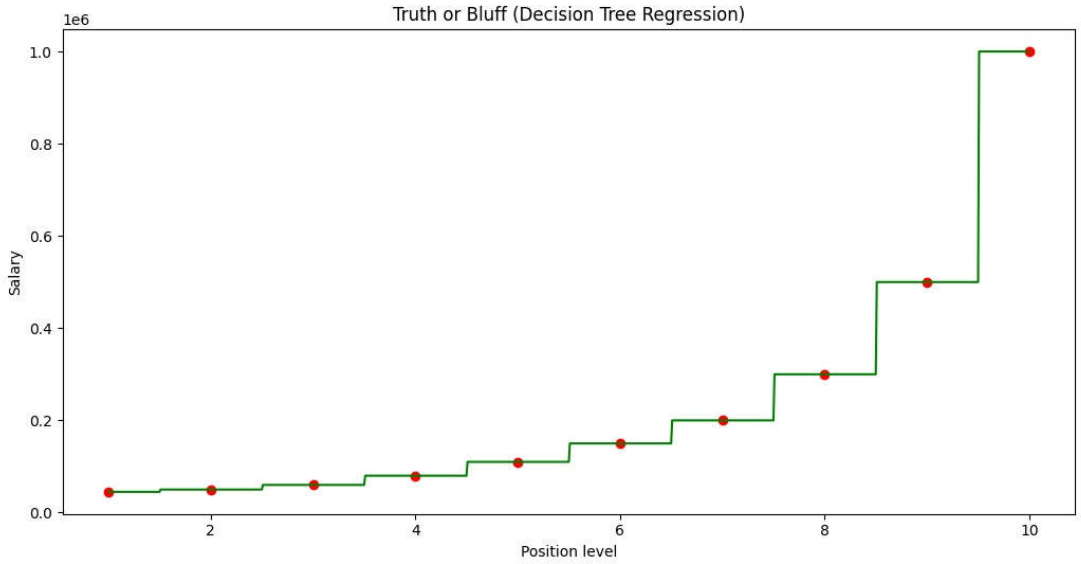
```
X_grid = np.arange(min(X), max(X), 0.01)
X_grid = X_grid.reshape(len(X_grid), 1) # reshape matrix/arr
ay
plt.scatter(X, y, color = "red")
plt.plot(X_grid, regressor.predict(X_grid), color = "green")
plt.title("Truth or Bluff (Decision Tree Regression)")
plt.xlabel("Position level")
plt.ylabel("Salary")
plt.show()
```



Use The high resolution

```
# plot the model
# plt.scatter(X, y, color = "red")
# plt.plot(X, regressor.predict(X), color = "green")
X_grid = np.arange(min(X), max(X), 0.01)
X_grid = X_grid.reshape(len(X_grid), 1) # reshape matrix/array
plt.scatter(X, y, color = "red")
plt.plot(X_grid, regressor.predict(X_grid), color = "green")
plt.title("Truth or Bluff (Decision Tree Regression)")
plt.xlabel("Position level")
plt.ylabel("Salary")
plt.show()
```

❒ We can see the vertical line below as we set resolution set to ***0.01*** so the vertical lines ***makes it Non-Continuous***.



☝ ***Conclusion:*** the decision tree regression model is not an *interesting* model in *one-dimension* but it can be a very interesting and very powerful model in ***more dimensions***.

☝ ***About Random Forrest:*** Random Forrest is actually a team of several decision trees. Previous example is the result of one tree what do you think we'll get with a team of 10 trees or even a hundred trees or 500 trees?

2.5.3 FAQ

👽 ***How does the algorithm Split the Data points?***

☞ It uses **reduction** of **standard deviation** of the ***predictions***. In other words, the *standard deviation* is *decreased* right after a ***split***. Hence, building a decision tree is all about **finding** the attribute that returns the ***highest standard deviation reduction*** (i.e., the most homogeneous branches).

👽 ***What is the Information Gain and how does it work in Decision Trees?***

☞ The **Information Gain** in ***Decision Tree Regression*** is exactly the **Standard Deviation Reduction** we are looking to reach. We calculate by how much the ***Standard Deviation decreases*** after each ***split***. Because the more the *Standard Deviation* is *decreased* after a *split*, the more ***homogeneous*** the ***child nodes*** will be.

👽 ***What is the Entropy and how does it work in Decision Trees?***

☞ The **Entropy** measures the **disorder in a set**, here in a part resulting from a ***split***. So the ***more homogeneous*** is your data in a part, the ***lower*** will be the ***entropy***. The more you have splits, the more you have chance to find *parts* in which your *data* is *homogeneous*, and therefore the *lower* will be the *entropy* (close to 0) in these parts. However you might still find some nodes where the *data is not homogeneous*, and therefore the *entropy* would *not be that small*.

Does a Decision Tree make much sense in 1D?

☞ Not really, as we saw in the practical part of this section. In 1D (meaning ***one independent variable***), the ***Decision Tree*** clearly tends to ***overfit*** the data. The ***Decision Tree*** would be much more relevant in ***higher dimension***, but keep in mind that the implementation we made here in 1D would be exactly the same in ***higher dimension***. Therefore you might want to keep that model in your ***toolkit*** in case you are dealing with a ***higher dimensional space***. This will actually be the case in ***Chapter 3 - Classification***, where we will use Decision Tree for Classification in 2D, which you will see turns out to be more relevant.

[Decision Tree Regression in R
Why do we get different results between Python and R?
The difference is likely due to the random split of data. If we did a cross-validation (see Part 10) on all the models in both languages, then you would likely get a similar mean accuracy. That being said, we would recommend more using Python for Decision Trees since the model is slightly better implemented in Python.]

Is the Decision Tree appropriate here?

☞ Here in this example, we can clearly see that the fitting curve is a ***stair*** with ***large gaps*** in the ***discontinuities***. That decision tree regression model is therefore ***not the most appropriate***, and that is because we have only ***one independent variables*** taking ***discrete values***. So what happened is that the prediction was made in the lower part of the gap in Python, and made in the upper part of the gap in R. And since the gap is large, that makes a big difference. If we had much more observations, taking values with more continuity (like with a 0.1 step), the gaps would be smaller and therefore the predictions in Python and R far from each other.

No p-value for Decision Tree ?

☞ You ***couldn't use p-value*** because ***`Decision Tree`*** is not a ***`linear model`***, and ***p-values*** apply only to ***linear models***. Therefore feature selection is out of the question. But you could do ***feature extraction***, which you will see in ***Chapter 9 - Dimensionality Reduction***. That you can apply to ***Decision Trees***, and it will ***reduce the number of your features***.

Random Forest Regression

2.6.1 Ensemble Learning

❒ ***Ensemble Learning:*** Ensemble learning is when you take *multiple algorithms* or the *same algorithm multiple times* and you put them together to make something much more powerful than the original.

❒ ***Ensemble Methods:*** In *statistics* and *machine learning*, ensemble methods use *multiple learning algorithms* to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone.

[Unlike a *statistical ensemble* in ***statistical mechanics***, which is usually infinite, a machine learning ensemble consists of only a concrete *finite set of alternative models*, but typically allows for much more flexible structure to exist among those alternatives.]

☝ ***Bootstrapping:*** In general, bootstrapping usually refers to a self-starting process that is supposed to continue or grow without external input.

2.6.2 Types of Ensemble Methods

Main Types of Ensemble Methods

[1] ***Bagging:*** Bagging, the short form for ***bootstrap aggregating***, is mainly applied in ***classification*** and ***regression***. It ***increases the accuracy*** of models through **Decision Trees**, which ***reduces variance*** to a large extent. The ***reduction of variance*** increases *accuracy*, eliminating *overfitting*, which is a challenge to many predictive models.

❒ ***Bagging*** is classified into two types, i.e., *bootstrapping* and *aggregation*.

i. ***Bootstrapping*** is a *sampling technique* where samples are derived from the ***whole population*** (set) using the *replacement procedure*. The ***sampling with replacement*** method helps make the ***selection procedure randomized***. The base learning algorithm is run on the samples to complete the procedure.

ii. ***Aggregation*** in **bagging** is done to incorporate *all possible outcomes of the prediction* and *randomize the outcome*. ***Without aggregation, predictions will not be accurate*** because all outcomes are not put into consideration. Therefore, the ***aggregation*** is based on the ***probability bootstrapping procedures*** or on the basis of all outcomes of the predictive models.

Bagging is advantageous since ***weak base learners*** are combined to form a single strong learner that is more stable than single learners. It also eliminates any ***variance***, thereby reducing the ***overfitting*** of models. One limitation of bagging is that it is ***computationally expensive***. Thus, it can *lead to more bias* in models when the *proper procedure of bagging is* ***ignored***.

[2] ***Boosting:*** Boosting is an *ensemble technique* that learns from ***previous predictor mistakes to make better predictions in the future***. The technique ***combines several weak base learners*** to form one strong learner, thus significantly improving the predictability of models. *Boosting works* by *arranging weak learners* in a sequence, such that *weak learners learn from the next learner* in the sequence to *create better predictive models*.

❒ *Boosting takes many forms*, including ***GRADIENT BOOSTING***, ***ADAPTIVE BOOSTING (ADABOOST)***, and ***XGBOOST (EXTREME GRADIENT BOOSTING)***.

i. ***AdaBoost*** uses weak learners in the **form** of **Decision Trees**, which mostly include **one split** that is popularly known as *decision stumps*. AdaBoost's main **decision stump** comprises observations carrying **similar weights**.

ii. ***Gradient boosting*** adds *predictors sequentially* to the *ensemble*, where ***preceding predictors*** *correct their* ***successors***, thereby increasing the model's accuracy. New predictors are fit to counter the effects of errors in the previous predictors. The gradient of descent helps the gradient booster identify problems in learners' predictions and counter them accordingly.

iii. ***XGBoost*** makes use of *Decision Trees* with *Boosted Gradient*, providing improved speed and performance. It relies heavily on the computational speed and the performance of the target model. Model training should follow a sequence, thus making the implementation of gradient boosted machines slow.

[3] ***Stacking:*** Stacking, another ensemble method, is often referred to as ***Stacked Generalization***. This technique works by allowing a ***training algorithm*** to ***ensemble several other similar learning algorithm predictions***. ***Stacking*** has been successfully implemented in ***Regression***, ***Density Estimations***, ***Distance Learning***, and ***Classifications***. It can also be used to **measure the error rate** involved during ***BAGGING***.

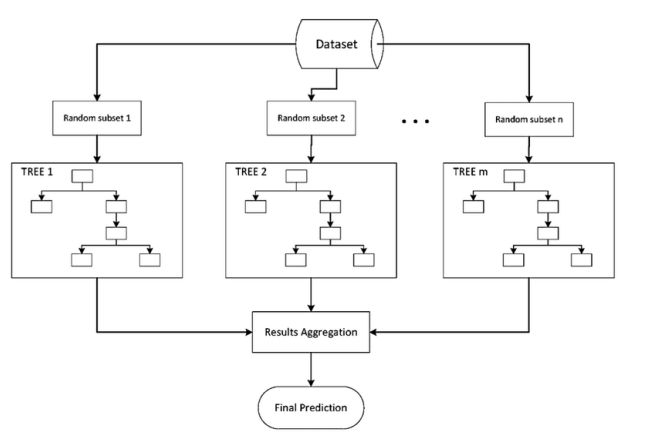
☝ ***Variance Reduction by Ensemble methods:*** Ensemble methods are ideal for **reducing** the **variance** in models, thereby increasing the accuracy of predictions. The ***variance is eliminated*** when ***multiple models*** are combined to form a ***single prediction*** that is chosen from all other possible predictions from the combined models. An ***ensemble of models*** combines various models to ensure that the resulting prediction is the best possible, based on the consideration of all predictions.

2.6.3 BAGGing and Random Forest

❒ ***BAGGing*** gets its name because it combines ***Bootstrapping*** and ***Aggregation*** to form one ***ensemble model***.

- Given a sample of data, ***multiple bootstrapped subsamples*** are pulled.
- A ***Decision Tree*** is ***formed*** on each of the ***bootstrapped subsamples***.
- After each subsample Decision Tree has been formed, an algorithm is used to aggregate over the Decision Trees to form the most efficient predictor.

Image on the right will help explain:

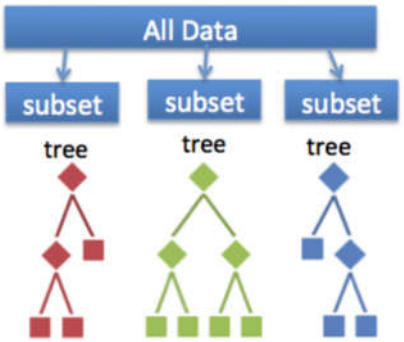


Given a Dataset, bootstrapped subsamples are pulled. A Decision Tree is formed on each bootstrapped sample. The results of each tree are aggregated to yield the strongest, most accurate predictor.

❒ ***Random Forest Models in BAGGING:*** The random forest algorithm is actually a ***bagging algorithm***. ***Random Forest Models*** can be thought of as ***BAGGing***, with a slight tweak.

☞ When deciding where to split and how to make decisions, ***BAGGed Decision Trees*** have the full **disposal of features** to choose from. Therefore, although the ***bootstrapped samples*** may be ***slightly different***, the *data is largely going to break off at the same features throughout each model*.

☞ In contrary, ***Random Forest models decide where to split based on a random selection of features***. *Rather* than *splitting at similar features at each node throughout*, ***Random Forest models implement a level of differentiation*** because `each tree will split based on different features`. This **level of differentiation provides a greater ensemble to aggregate over**, ergo ***producing a more accurate predictor***. Refer to the image for a better understanding.



A random forest takes a random subset of features from the data, and creates n random trees from each subset. Trees are aggregated together at end.

☞ Similar to ***BAGGing***, ***bootstrapped subsamples*** are pulled from a ***larger dataset***. A ***Decision Tree*** is formed on each **subsample**. HOWEVER, the decision tree is split on ***different features*** (in this diagram the ***features*** are represented by ***shapes***).

2.6.4 Random Forest

We will discuss about ***Random Forest*** applied to ***Regression Trees***. We first create N decision trees on N subset of data (picking data points randomly) and then we predict a new data-point using N- N decision trees. Following are the steps for Random Forrest:

- ***STEP 1:*** Pick at ***`random K data points`*** from the Training set. i.e. make a subset of data point.
- ***STEP 2:*** Build the Decision Tree associated to these K data points. i.e. create a Decision Tree on the selected subset.
- ***STEP 3:*** Choose the number N-tree of trees you want to build and repeat STEPS 1 & 2
- ***STEP 4:*** For a new data point, make each one of your Ntree trees predict the value of Y to for the data point in question, and assign the new data point the average across all of the predicted Y values.

❒ In that way you're ***not just predicting one tree but on a forest of trees***. And that improves the accuracy of your prediction because it is you're taking the ***average of many predictions***.

☞ Therefore even if some tree is too perfect (**overfitting**) or too bad at prediction those extreme case are ignored i.e. impact on a forest of trees is negligible. So you're going to get a more accurate prediction.

☝ This ***ensemble algorithms*** are ***more stable*** because any changes in your data set could really impact one tree but it is hard to impact on a forest of trees.

2.6.5 Random Forest in Python

Random forest is just a **team of Decision Trees** each one making some ***prediction*** of your ***dependent variable*** and the ***ultimate prediction*** of the ***Random forest itself*** is simply the **average** of the **different predictions** of ***all*** the ***different trees*** in the ***forest***.

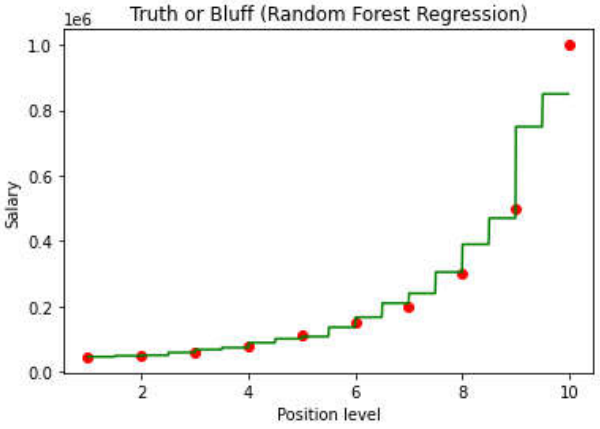
☞ ***Always 3 steps:*** **[1]** import class **[2]** create object **[3]** fit the dataset

```
# Fit dataset to Random Forest Regression
from sklearn.ensemble import RandomForestRegressor # import class
regressor = RandomForestRegressor(n_estimators= 10 ,random_state= 0) # create object
regressor.fit(X, y) # fit the dataset
```

☞ Parameters for `RandomForestRegressor()`: ***`n_estimator`***: no of trees, ***`random_state`*** = 0, ***`criterion`*** = "mse" (mse = mean square method), ***`max_features`*** = "auto" (improve your model), (other max-min parameters).

❒ ***Finding best team of trees:*** Now all code works as the previous model, we just need to select the ***no. of trees `n_estimator`*** to make a better prediction.

10 trees :



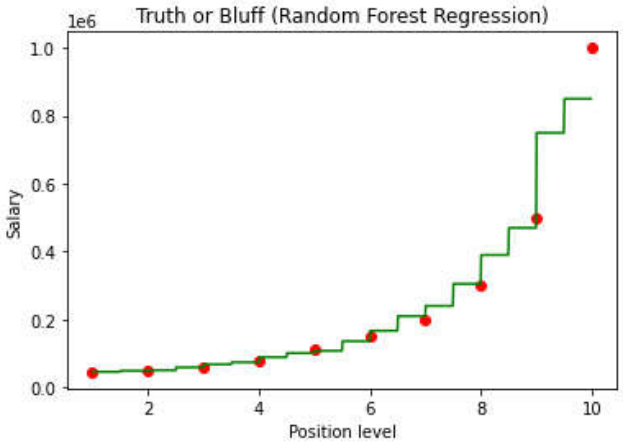
- Here we have ***several trees*** hence we have ***several stairs*** . This is expected for non-continuous models.
- So we have a **lot more** of **splits** of the **whole range** of ***levels*** and therefore a lot more ***intervals*** of the different levels. So each straight ***horizontal line*** here separate by the ***vertical lines***.
- Now if we get prediction for the ***`6.5`*** level (which is ***`167000`***), what happened with this prediction is that- ***we had 10 trees voting on which step the salary of the 6.5 level position would be*** and then the ***Random forest*** takes the ***average of all the different predictions*** of the salary of the 6.5 level made by all the different trees in the forest. And the average of the predictions is ***`167000`***.

❒ ***Stairs get into certain shape:*** If we add a ***lot more trees*** in our random forest, it ***doesn't mean we'll get a lot more steps*** on the ***stairs*** because the *more you add some trees the more the average of the different predictions made by the trees is converging to the same average* this is based on the same technique ***Entropy and Information Gain***.

☞ So the more you add trees the more the average of these votes will converge to the same ***Ultimate Average*** and therefore it will converge to some Certain Shape of stairs here.

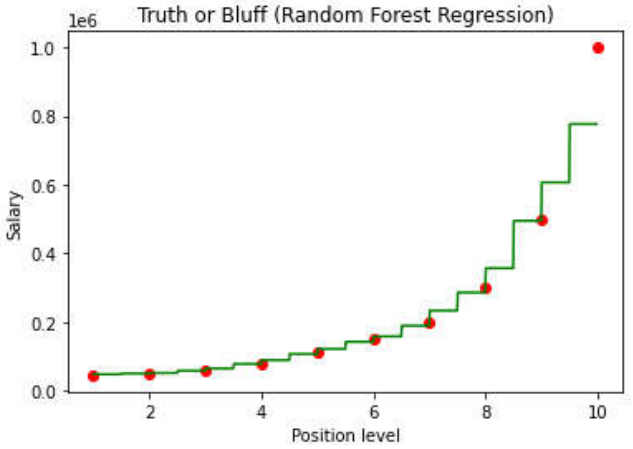
10 trees, predicted salary for 6.5 level is ***167000***.

```
RandomForestRegressor(n_estimators= 10 ,random_state= 0)
```



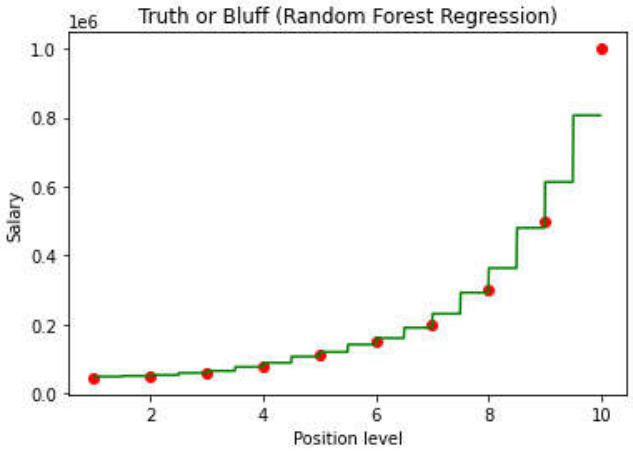
100 trees, predicted salary for 6.5 level is ***158300***.

```
RandomForestRegressor(n_estimators= 100 ,random_state= 0)
```



300 trees, predicted salary for 6.5 level is ***160333.33333333***.

```
RandomForestRegressor(n_estimators= 300 ,random_state= 0)
```



☞ From above we can see that the ***no. of steps in the stairs is not changing***. But the stair it taking a ***unique shape***

❒ ***Conclusion:***

- So as a ***comparison*** this ***Random forest model*** is even ***better*** than the ***Polynomial model***.
- In Chapter-10 we will build some "Ensemble ML models". Some models that are a combination of several ML models and these ***Ensemble ML models*** are actually the best models. When you have a team of several ML models they can actually make an awesome prediction.
- In this section we had a ***team of same ML models*** which were ***Decision Tree regression*** models. But in the future we'll make a ***team of different ML models***.

NOTE: ***Continuous model*** doesn't mean the **dataset** is **continuous**, but the ***"Model/Mathematical-Model"*** is ***continuous***.

Practiced version

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Position_Salaries.csv")
X = dataSet.iloc[:, 1:2].values
y = dataSet.iloc[:, 2].values

# # Feature-Scaling
# from sklearn.preprocessing import StandardScaler
# sc_x = StandardScaler()
# sc_y = StandardScaler()
# X_scaled = sc_x.fit_transform(X)
# y_scaled = sc_y.fit_transform(y.reshape(-1, 1))

# Data Split : No need for this example

# Fit dataset to Random Forest Regression
from sklearn.ensemble import RandomForestRegressor # import class
regressor = RandomForestRegressor(n_estimators= 300 ,random_state= 0) # create object
regressor.fit(X, y) # fit the dataset

# Predict
y_pred = regressor.predict([[6.5]])
print("The predicte value ffor 6.5 is : ", y_pred)

# plot the model
X_grid = np.arange(min(X), max(X), 0.01)
X_grid = X_grid.reshape(len(X_grid), 1) # reshape matrix/array
pLt.scatter(X, y, color = "red")
pLt.plot(X_grid, regressor.predict(X_grid), color = "green")
pLt.title("Truth or Bluff (Random Forest Regression)")
pLt.xlabel("Position level")
pLt.ylabel("Salary")
pLt.show()
```

2.6.6 FAQ

- ***What is the advantage and drawback of Random Forests compared to Decision Trees?***
  - ***Advantage:*** *Random Forests* can give you a *better predictive* power than *Decision Trees*.
  - ***Drawback:*** *Decision Tree* will give you more *interpretability* than *Random Forests*, because you can plot the graph of a *Decision Tree* to see the different splits leading to the prediction, as seen in the Intuition Lecture. That's something you can't do with *Random Forests*.

- ***When to use Random Forest and when to use the other models?***
  - The best answer to that question is: try them all!
    Indeed, thanks to the templates it will only take you 10 minutes to try all the models, which is very little compared to the time dedicated to the other parts of a data science project (like Data Preprocessing for example). So just don't be scared to try all the regression models and compare the results (through cross validation which we will see in Chapter 10). That's we gave you the maximum models in this course for you to have in your toolkit and increase your chance of getting better results.

  - However then, if you want some shortcuts, here are some rules of thumbs to help you decide which model to use: First, you need to figure out whether your problem is ***linear*** or ***non linear***. You will learn how to do that in ***Chapter 10 - Model Selection***.
    - Then: If your problem is *linear*, you should go for *Simple Linear Regression* if you only have *one* feature, and *Multiple Linear Regression* if you have *several features*.
    - If your problem is ***non linear***, you should go for ***Polynomial Regression***, ***SVR***, ***Decision Tree*** or ***Random Forest***.

☹ Then which one should you choose among these four? That you will learn in ***Chapter - 10 – Model Selection***. The method consists of using a very relevant technique that evaluates your models performance, called ***k-Fold Cross Validation***, and then picking the model that shows the best results. Feel free to jump directly to Part 10 if you already want to learn how to do that.

How do I know how many trees I should use?

☞ First, I would recommend to **choose** the number of trees by **experimenting**. It usually takes less time than we think to figure out a best value by *tweaking* and *tuning* your model *manually*. That's actually what we do in *general* when we build a *Machine Learning model:* we do it in *several shots*, by *experimenting several* values of *hyperparameters* like the ***number of trees***. However, also know that in ***Chapter 10*** we will cover ***k-Fold Cross Validation*** and ***Grid Search***, which are powerful techniques that you can use to find the optimal value of a ***hyperparameter***, like here the ***number of trees***.

☞ You should use ***enough trees*** to get a ***good accuracy***, but you shouldn't use ***too many trees*** because that could cause ***overfitting***. You will learn how to find the optimal number of trees in the first section of Chapter 10 - Model Selection. It's done with a technique called ***Parameter Tuning (Grid Search with k-Fold Cross Validation)***.

Why do we get different results between Python and R?

☞ The difference is likely due to the *random split* of *data*. If we did a *cross-validation* (see Chapter 10) on all the models in *both languages*, then you would likely get a *similar mean accuracy*.

No p-valuefor Random Forest

☞ You couldn't use *p-value* because *Random Forests* are *not linear models*, and *p-values apply only to linear models*. Therefore feature selection is out of the question. But you could do feature extraction, which you will see in ***Chapter 9 - Dimensionality Reduction***. That you can apply to ***Random Forests***, and it will reduce the number of your features.

Evaluating Regression Models Performance

2.7.1 R Squared

[*Compare models w.r.t* R^2]

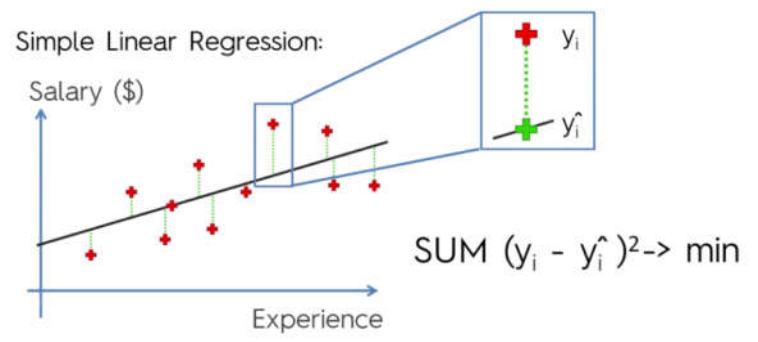
We talked about the ***simple linear regression*** being constructed through the ***ordinary least squares method*** where we are minimizing

$$\Sigma(y_i - \hat{y}_i)^2$$

❒ We were counting that sum and then the ***line that has them smallest sum*** will be the ***best fitting lin***e or will be this ***simple linear regression*** model.

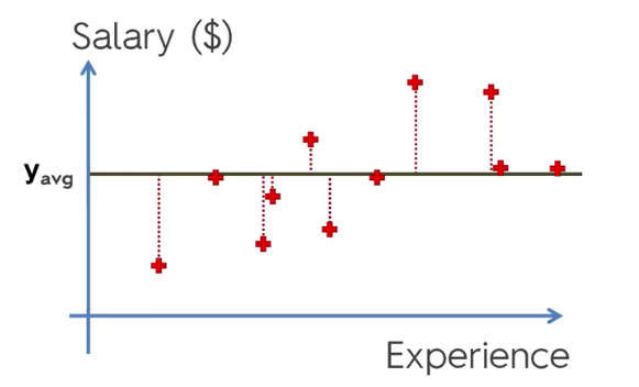
🛠 ***Residual sum of squares:*** For out fitted-line (model line) the residual sum of squares SS_{res} or sum of squared error (SSE) is :

$$SS_{res} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$



🛠 ***Total sum of squares:*** Using the average line we get the ***total sum of squares:***

$$SS_{tot} = \sum_{i=1}^{n} (y_i - y_{avg})^2$$



🛠 ***R-square:*** The following equation is the R-squared

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

- ☞ ***R-square*** is says that *"How good the* ***best fitting line (or model)*** *is with respect to the* ***average line***". Minimizing SS_{res} is maximize the R^2. $R^2 = 1$ is the best scenario (and will never happen yo!!).
- ☞ So R^2 near 1 is better model and near 0 is the bad model. And R^2 also can be negative (model is worse than the average line).

2.7.2 Adjusted R^2

[*Compare models w.r.t* R^2 *and Increase/Decrease Feature variable*]

We talked about R^2 for a *Simple Linear Regression* while the same concepts apply for a multiple linear regression. We use R-squared as a ***goodness of fit parameters***, the bigger it is (close to 1) the better model.

❒ ***Impact on*** R^2 ***by Increase/ Decrease of feature variables:*** R^2 is ***biased***. Reason is: when we add ***new feature variable*** the SS_{res} always ***minimize*** or stays same. The SS_{res} ***does'nt increase*** because of the model, if new variable effect the model in negative way, the co-efficient of the new variable just get smaller, so that the effect to be negligible.

- By adding ***new feature variable***, $\frac{SS_{res}}{SS_{tot}}$ may decrease but never increase. Hence is never decrease by adding new feature variable.
- Then how to determine the impact of new feature variable? The solution is to use Adjusted R^2.

❒ ***Adjusted R^2***: Folowing is the equation for Adjusted R^2.

$$adj\ R^2 = 1 - (1 - R^2)\frac{n-1}{1-p-n}$$

Where n = Sample size,
p = no. of regressor/feature variable.

- ***Penalization factor:*** Adjusted R-squared has a penalization factor. It *penalizes you for adding independent variables that don't help your model.*
  - As we can see when we increase variable, $\boldsymbol{p}$ increase so $\boldsymbol{1-p-n}$ decreases i.e. $\frac{n-1}{1-p-n}$ increases.
  - Now if the new variable doesn't help, then R^2 doesn't decrease so $(1-R^2)\frac{n-1}{1-p-n}$ increase and $adj\ R^2$ decrease.
  - Here actually $(1-R^2)$ and $\frac{n-1}{1-p-n}$ balance each other. If the new variable helps, then $(1-R^2)\frac{n-1}{1-p-n}$ decrease and $adj\ R^2$ will increase.

☝ In statistics, a ***regressor*** is the name given to any variable in a ***regression model*** that is used to predict a ***response variable***. A ***regressor*** is also referred to as: An ***explanatory variable***. An ***independent variable***. A ***manipulated variable***. Feature, independent variable, explanatory variable, regressor, covariate, or predictor are all names of the variables that are used to predict the target, outcome, dependent variable, regressand, or response.

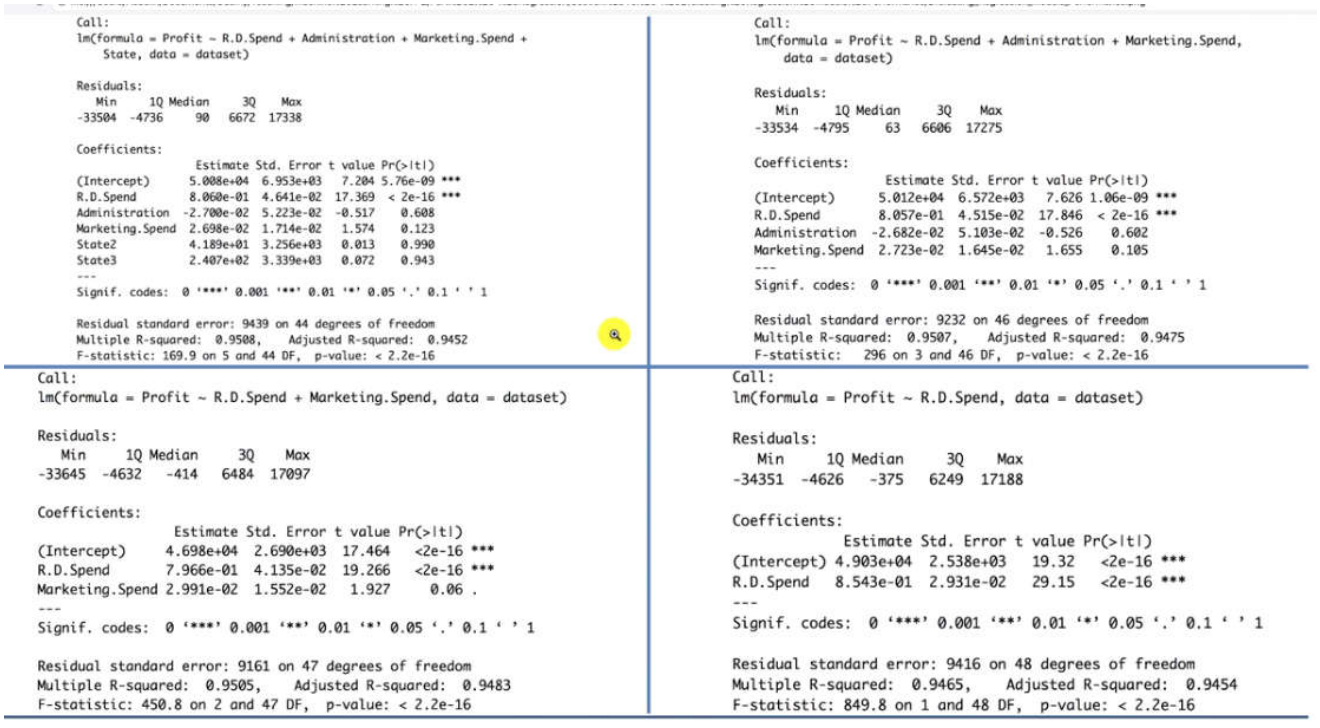
2.7.3 Feature selection using Adjusted R-squared

Recall the Multiple Linear Regression's Backward elimination, where we rejected variables according to p-values. Now at final step we end up with only one variable ***R&D spend.***

➲ After **4th** and **5th** iteration, we inspected the ***p-values***. At **4th** iteration, we end-up with ***R&D spend*** is definitely a very powerful predictor of the profit and definitely has a high statistical effect impact on the dependent variable profit.
- And as the ***second independent variable*** is with enough ***statistical effect impact*** is ***Marketing Spend***, we can see that the p-value is **0.06** that is **6%**.

☝ Now we decide that we can keep this variable or not according to Adjusted R-squared. If we obseve fropm following diagram, that the Adj. R-square is actually greter with ***R&D spend*** and ***Marketing Spend*** both. But is get smaller when we reject ***Marketing Spend***. Hence with ***R&D spend*** and ***Marketing Spend*** both we get the better prediction. And hence we are kipping ***Marketing Spend*** even though it has 6% p-value.

☝ So just inspect the ***R-squared*** and ***Adj. R-square***. Give ***Adj. R-square*** top priority during feature selection.



```
Call:
lm(formula = Profit ~ R.D.Spend + Administration + Marketing.Spend +
    State, data = dataset)

Residuals:
   Min     1Q Median     3Q    Max
-33504  -4736     90   6672  17338

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)      5.008e+04  6.953e+03   7.204 5.76e-09 ***
R.D.Spend        8.060e-01  4.641e-02  17.369  < 2e-16 ***
Administration  -2.700e-02  5.223e-02  -0.517    0.608
Marketing.Spend  2.698e-02  1.714e-02   1.574    0.123
State2           4.189e+01  3.256e+03   0.013    0.990
State3           2.407e+02  3.339e+03   0.072    0.943
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9439 on 44 degrees of freedom
Multiple R-squared:  0.9508,    Adjusted R-squared:  0.9452
F-statistic: 169.9 on 5 and 44 DF,  p-value: < 2.2e-16
```

```
Call:
lm(formula = Profit ~ R.D.Spend + Administration + Marketing.Spend,
    data = dataset)

Residuals:
   Min     1Q Median     3Q    Max
-33534  -4795     63   6606  17275

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)      5.012e+04  6.572e+03   7.626 1.06e-09 ***
R.D.Spend        8.057e-01  4.515e-02  17.846  < 2e-16 ***
Administration  -2.682e-02  5.103e-02  -0.526    0.602
Marketing.Spend  2.723e-02  1.645e-02   1.655    0.105
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9232 on 46 degrees of freedom
Multiple R-squared:  0.9507,    Adjusted R-squared:  0.9475
F-statistic:   296 on 3 and 46 DF,  p-value: < 2.2e-16
```

```
Call:
lm(formula = Profit ~ R.D.Spend + Marketing.Spend, data = dataset)

Residuals:
   Min     1Q Median     3Q    Max
-33645  -4632   -414   6484  17097

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)      4.698e+04  2.690e+03  17.464   <2e-16 ***
R.D.Spend        7.966e-01  4.135e-02  19.266   <2e-16 ***
Marketing.Spend 2.991e-02  1.552e-02   1.927     0.06 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9161 on 47 degrees of freedom
Multiple R-squared:  0.9505,    Adjusted R-squared:  0.9483
F-statistic: 450.8 on 2 and 47 DF,  p-value: < 2.2e-16
```

```
Call:
lm(formula = Profit ~ R.D.Spend, data = dataset)

Residuals:
   Min     1Q Median     3Q    Max
-34351  -4626   -375   6249  17188

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 4.903e+04  2.538e+03   19.32   <2e-16 ***
R.D.Spend   8.543e-01  2.931e-02   29.15   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9416 on 48 degrees of freedom
Multiple R-squared:  0.9465,    Adjusted R-squared:  0.9454
F-statistic: 849.8 on 1 and 48 DF,  p-value: < 2.2e-16
```

2.7.4 Implement automated Backward Elimination with Adjusted R-Squared In Python:

Let's take again the problem of the Multiple Linear Regression, with 5 independent variables. Automated Backward Elimination including Adjusted R Squared can be implemented this way:

```
# Fundamental  Libraries
import matplotlib as plt
import pandas as pd
import numpy as np

# import dataset
dataSet = pd.read_csv("50_Startups.csv")
X = dataSet.iloc[:, :-1] #all rows except last
y = dataSet.iloc[:, 4] # 5th row


# categorical to numerical
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
colTfrm = ColumnTransformer(transformers = [("encoder", OneHotEncoder(), [3])], remainder="passthrough" )
X_encoded = np.array(colTfrm.fit_transform(X))
        #  last column is now replaced with dummy colums (1st 3 colmns)

# Avoiding dummy-var trap: omit one dummy varable
X_go = X_encoded[:, 1:]         # select all columns starting from 2nd column
y_go = np.array(y) #converting Dataframe to Vector/Array


# split dataset to Train and Test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_go, y_go, test_size = 0.2, random_state = 0)


# Fitting multiple linear regression on traing set
from sklearn.linear_model import LinearRegression
regResor = LinearRegression()
regResor.fit(X_train, y_train)

# predict on the test-set X_test
y_pred = regResor.predict(X_test)


#Checking the score
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
print('\n\n------------ Train Score: ', regResor.score(X_train, y_train))
print('\n\n------------ Test Score: ', regResor.score(X_test, y_test))

#================= building the optimal model using Backward elimination ====================

# ---------- Only P-value implement automatic Backward Elimination: No manual iteration is needed ----------------
"""
import statsmodels.api as sm
def backwardElimination(x, sl):
    numVars = len(x[0])
    for i in range(0, numVars):
        regressor_OLS = sm.OLS(endog = y_go, exog=x).fit()
        # picking the max p-value
        maxPval = max(regressor_OLS.pvalues).astype(float)
        if maxPval >= sl:
            for j in range(0, numVars - i):
                # deleting the clumn
                if (regressor_OLS.pvalues[j].astype(float) == maxPval):
                    x = np.delete(x, j, 1) # deletes j-th column. "1" is used for "column". To delet "row" use "0"
    print(regressor_OLS.summary())
    return x


SL = 0.05
X_pre_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1) # interchange the columns
X_opt = X_pre_opt[:, [0, 1, 2, 3, 4, 5]]
X_Modeled = backwardElimination(X_opt, SL)

"""

import statsmodels.api as sm
def backwardElimination(x, SL):
    numVars = len(x[0])
    temp = np.zeros((50, 6)).astype(int)
    for i in range(0, numVars):
        regressor_OLS = sm.OLS(endog = y_go, exog=x).fit()
        maxVar = max(regressor_OLS.pvalues).astype(float)
        adjR_before = regressor_OLS.rsquared_adj.astype(float)
```

```
        if maxVar > SL:
            for j in range(0, numVars - i):
                if (regressor_OLS.pvalues[j].astype(float) == maxVar):
                    temp[:,j] = x[:, j]
                    x = np.delete(x, j, 1)
                    tmp_regressor = sm.OLS(y, x).fit()
                    adjR_after = tmp_regressor.rsquared_adj.astype(float)
                    if (adjR_before >= adjR_after):
                        x_rollback = np.hstack((x, temp[:,[0,j]]))
                        x_rollback = np.delete(x_rollback, j, 1)
                        print (regressor_OLS.summary())
                        return x_rollback
                    else:
                        continue

    regressor_OLS.summary()
    return x

SL = 0.05
X_pre_opt = np.append(arr = np.ones(shape = (50, 1)).astype(int), values =X_go , axis =1) # interchange the columns
X_opt = X_pre_opt[:, [0, 1, 2, 3, 4, 5]]
X_Modeled = backwardElimination(X_opt, SL)

# python prctc_modl_eval_mul_lin_regsn.py
```

Only p-value

```
==============================================================================
Dep. Variable:                      y   R-squared:                       0.947
Model:                            OLS   Adj. R-squared:                  0.945
Method:                 Least Squares   F-statistic:                     849.8
Date:                Thu, 17 Mar 2022   Prob (F-statistic):           3.50e-32
Time:                        13:37:35   Log-Likelihood:                -527.44
No. Observations:                  50   AIC:                             1059.
Df Residuals:                      48   BIC:                             1063.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const       4.903e+04   2537.897     19.320      0.000    4.39e+04    5.41e+04
x1             0.8543      0.029     29.151      0.000       0.795       0.913
==============================================================================
Omnibus:                       13.727   Durbin-Watson:                   1.116
Prob(Omnibus):                  0.001   Jarque-Bera (JB):               18.536
Skew:                          -0.911   Prob(JB):                     9.44e-05
Kurtosis:                       5.361   Cond. No.                     1.65e+05
==============================================================================
```

Only p-value and Adjusted R-squared: Notice Adj. R-squared increased

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.950
Model:                            OLS   Adj. R-squared:                  0.948
Method:                 Least Squares   F-statistic:                     450.8
Date:                Thu, 17 Mar 2022   Prob (F-statistic):           2.16e-31
Time:                        13:39:25   Log-Likelihood:                -525.54
No. Observations:                  50   AIC:                             1057.
Df Residuals:                      47   BIC:                             1063.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const       4.698e+04   2689.933     17.464      0.000    4.16e+04    5.24e+04
x1             0.7966      0.041     19.266      0.000       0.713       0.880
x2             0.0299      0.016      1.927      0.060      -0.001       0.061
==============================================================================
Omnibus:                       14.677   Durbin-Watson:                   1.257
Prob(Omnibus):                  0.001   Jarque-Bera (JB):               21.161
Skew:                          -0.939   Prob(JB):                     2.54e-05
Kurtosis:                       5.575   Cond. No.                     5.32e+05
==============================================================================
```

2.7.5 Interpreting Linear Regression Coefficients

How do we find/explain/interpret the Coefficient of the final model. Consider the previous ***Venture Capitalist*** example,

```
lm(formula = Profit ~ R.D.Spend + Marketing.Spend, data = dataset)

Residuals:
   Min     1Q Median     3Q    Max
-33645  -4632   -414   6484  17097

Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
(Intercept)     4.698e+04  2.690e+03  17.464   <2e-16 ***
R.D.Spend       7.966e-01  4.135e-02  19.266   <2e-16 ***
Marketing.Spend 2.991e-02  1.552e-02   1.927     0.06 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9161 on 47 degrees of freedom
Multiple R-squared:  0.9505,    Adjusted R-squared:  0.9483
F-statistic: 450.8 on 2 and 47 DF,  p-value: < 2.2e-16
```

☝ ***Notice the following:***

```
R.D.Spend       7.966e-01
Marketing.Spend 2.991e-02
```

So these coefficients here these $B_1 = \mathbf{0.7966}$ and $B_2 = \mathbf{0.0299}$. What they're telling us is that,

☝ ***Sign:*** First of all you look at the ***sign***. If the ***sign*** is ***positive*** that means your variable is ***correlated*** with your ***independent*** variable. Meaning that if you ***change*** the value of your ***independent variable*** then the value then you can see that the ***dependent variable*** will be ***changing*** in the ***same direction***.

- ☞ So basically if you'll be increasing spend on ***R&D*** then you're profitable be ***increasing*** If increasing ***spend of marking*** then your profit is also ***increasing***.
- ☞ If the sign is ***negative*** then it's the opposite effect. So basically ***increase*** your ***independent variable*** and you ***depend variable decreases***. i.e If any one of them has ***−ve*** value then increasing the *corresponding feature* will cause decrease the dependent variable (it is the basic math).

☝ ***Always remember the magnitude:*** So you might think that, Magnitude of coefficient for this ***R&D Spend*** is bigger than the ***Market Spend*** coefficient. So definitely ***R&D spend*** has a bigger impact. But that's not the case, the ***unit of measurement*** is important here, to compare something like that we have to measure all the variables in a same ***measurement scale***.

- ☞ So, even though coefficient for this ***R&D Spend*** is bigger but its magnitude of unit can be lower say 1 Cent i.e ($0.01), and the magnitude of unit for ***Market Spend*** say $1 where its coefficient is lower than ***R&D Spend***.
- ☞ ***Use the term "Per Unit Change":*** R.D. Spend could be million dollar (unit) or it could be in Cents i.e. ($0.01). Or some feature variable can have different units of measure : say Km, or Celsius etc. The magnitude could be different for different feature variables (say ***R&D spend*** could be in **Billion Dollars** and ***Marketing-spend*** could be in **Million Dollars**). In all of those case we should say "***Per Unit Change***".

- ☞ So here we use $1 for measurement unit for ***R&D Spend*** and ***Market Spend*** . Then we can say, for every dollar (for every unit) of R&D spend that you increase, according to his model your profit will increase by ***79 cents*** or ***$0.79***. For every unit that you ***decrease*** in your ***R&D spend*** your profit will ***decrease*** by 0.79unit of profit.
  - ➲ Similarly for every dollar (for every unit) of ***Market Spend*** that you increase, according to his model your profit will increase by 2.99 cents or $0.0299.

2.7.6 FAQ

I understand that in order to *evaluate multiple linear models* I can use the ***Adjusted Rsquared*** or the ***Pearson matrix***. But how can I evaluate ***Polynomial Regression*** models or ***Random Forest Regression*** models which are ***not linear***?

- ☞ You can evaluate *polynomial regression* models and *random forest regression* models, by computing the *"Mean of Squared Residuals" (the mean of the squared errors)*. You can compute that easily with a ***sum*** function or using a ***for loop***, by computing the ***sum of the squared differences*** between the predicted outcomes and the real outcomes, over all the observations of the test set.

☞ You couldn't really apply ***Backward Elimination*** to ***Polynomial Regression*** and ***Random Forest Regression*** models because these models don't have ***coefficients*** combined in a ***linear regression equation*** and therefore don't have p-values.

However in Chapter 9 - Dimensionality Reduction, we will see some techniques like Backward Elimination, to reduce the number of features, based on Feature Selection & Feature Extraction.

❒ What are Low/High Bias/Variance in Machine Learning?

- ***Low Bias*** is when your model predictions are very ***close*** to the ***real values***.
- ***High Bias*** is when your model predictions are ***far*** from the ***real values***.
- ***Low Variance:*** when you ***run your model several times***, the ***different predictions*** of your observation points ***won't vary*** much.
- ***High Variance:*** when you ***run your model several times***, the ***different predictions*** of your observation points will ***vary a lot***.

☝ What you want to get when you build a model is: Low Bias and Low Variance.

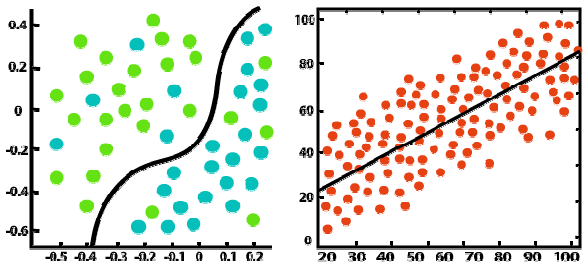
Chapter 3 : part 1

Logistic Regression

3.1.1 REGRESSION vs CLASSIFICATION

Regression and ***Classification*** algorithms are ***Supervised Learning*** algorithms. Both the algorithms are used for prediction in Machine learning and work with the **labeled datasets**. But the difference between both is how they are used for different machine learning problems.

☞ ***The main difference between them is:*** ***Regression algorithms*** are used to ***predict*** the ***continuous values*** such as ***price***, ***salary***, ***age***, etc. and ***Classification algorithms*** are used to ***predict/Classify*** the ***discrete values*** such as ***Male*** or ***Female***, ***True*** or ***False***, Spam or Not Spam, etc.



❒ ***Classification:*** Classification is a process of ***finding*** a ***function*** which helps in ***dividing the dataset*** into ***classes*** based on different parameters. In ***Classification***, a computer program is trained on the training dataset and based on that training, it ***categorizes the data*** into ***different classes***.

☞ The task of the classification algorithm is to ***find the mapping function*** to map the ***input(x)*** to the ***discrete output(y)***.

☯ ***Example:*** The best example to understand the ***Classification*** problem is ***Email Spam Detection***. The model is trained on the basis of millions of emails on different parameters, and whenever it receives a new email, it identifies whether the email is spam or not. If the email is spam, then it is moved to the Spam folder.

☞ ***Types of ML Classification Algorithms:*** Classification Algorithms can be further divided into the following types:

[1] Logistic Regression
[2] K-Nearest Neighbors
[3] Support Vector Machines (SVM)
[4] Kernel SVM
[5] Naïve Bayes
[6] Decision Tree Classification
[7] Random Forest Classification

❒ ***Regression:*** Regression is a process of *finding the* ***correlations*** *between* ***dependent*** *and* ***independent*** *variables*. It helps in ***predicting*** the ***continuous variables*** such as prediction of Market Trends, prediction of House prices, etc.

☞ The task of the Regression algorithm is to find the ***`mapping function`*** to map the ***`input variable(x)`*** to the continuous output ***`variable(y)`***.

☯ ***Example:*** Suppose we want to do weather forecasting, so for this, we will use the ***Regression algorithm***. In weather prediction, the model is trained on the ***past data***, and once the training is completed, it can easily predict the weather for ***future days***.

☞ ***Types of Regression Algorithm:***

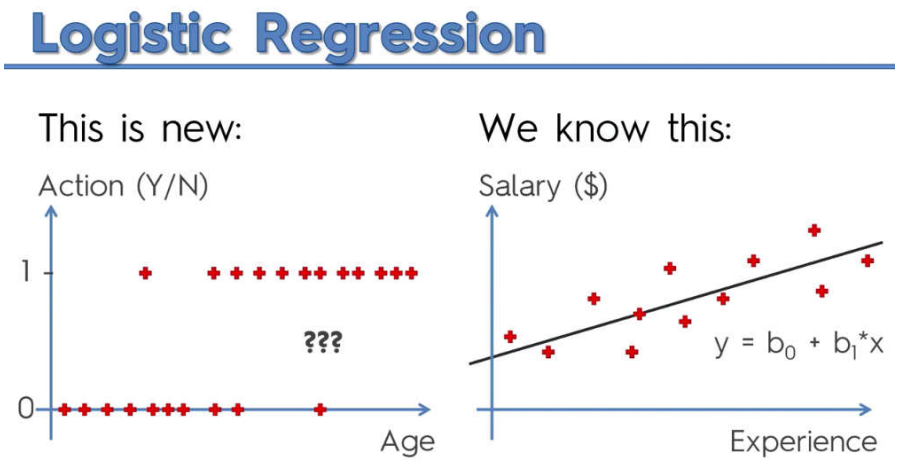
[1] Simple Linear Regression
[2] Multiple Linear Regression
[3] Polynomial Regression
[4] Support Vector Regression
[5] Decision Tree Regression

❒ ***Difference between Regression and Classification:***

Regression Algorithm		Classification Algorithm	
i.	In ***Regression***, the *output variable* must be of *continuous* nature or real value.	ii.	In ***Classification***, the *output variable* must be a *discrete value*.
iii.	The task of the ***regression algorithm*** is to *map* the *input* value (x) with the *continuous output* variable(y).	iv.	The task of the ***classification algorithm*** is to *map* the *input* value(x) with the *discrete output* variable(y).
v.	Regression Algorithms are used with ***continuous data***.	vi.	Classification Algorithms are used with ***discrete data***.
vii.	In ***Regression***, we try to find the ***best fit line***, which can predict the output more accurately.	viii.	In ***Classification***, we try to find the ***decision boundary***, which can divide the dataset into different classes.
ix.	*Regression algorithms* can be used to solve the regression problems such as *Weather Prediction*, *House price* prediction, etc.	x.	*Classification Algorithms* can be used to solve classification problems such as Identification of *spam emails*, *Speech Recognition*, *Identification* of *cancer cells*, etc.
xi.	The regression Algorithm can be further divided into *Linear* and *Non-linear* Regression.	xii.	The Classification algorithms can be divided into *Binary* Classifier and *Multi-class* Classifier.

☝ ***Does regression mean prediction?***
Using *regression* to make *predictions* doesn't necessarily involve *predicting* the *future*. Instead, you *predict the mean* of the *dependent variable* given specific values of the *independent variable(s)*.

3.1.2 Introduction to Classification problem



☞ We already know how to deal with data like in the Right side (with regression). But how do we deal with the data in the Left side?

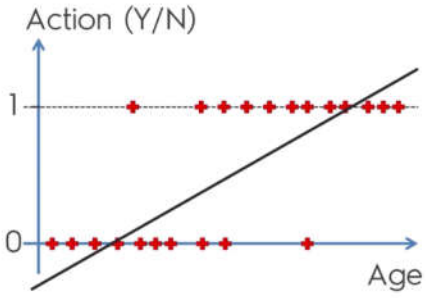
☞ In the Left side data, lets assume the problem as: Action of buying a product, from the left side data we can see there is actually a correlation, which is "older people usually takes action to – Yes and younger people actually doesn't take actions".

☞ We can see that the **observations** on the **bottom** they're a bit more to the **left** and **observations** on the **top** are a bit more to the **right** implying that, ***"probably older people are more likely to take action based on the shopping offer"***.

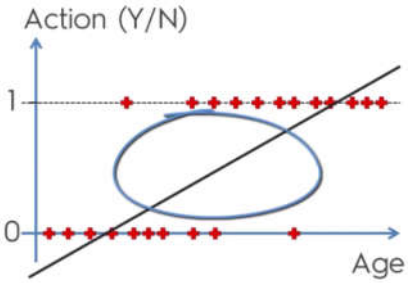
❒ ***Use Linear Regression first:*** We try our existing method in our toolkit which is the *linear regression*, it doesn't look like the best approach and not the best method to solve this problem.

☞ Let's draw a horizontal line $y = 1$. Now, we want to predict for that person of certain age we want to predict whether they will take up the offer or not.

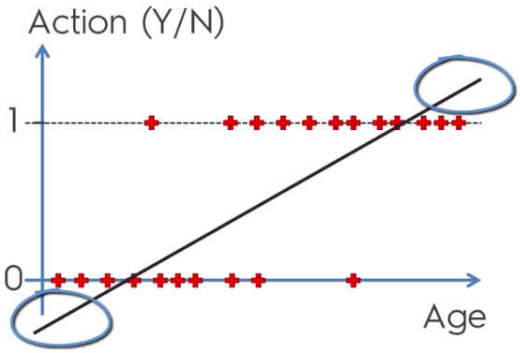
☞ To do that, we will predict the *probability rules* that will state a probability or a likelihood of that person taking up that offer.

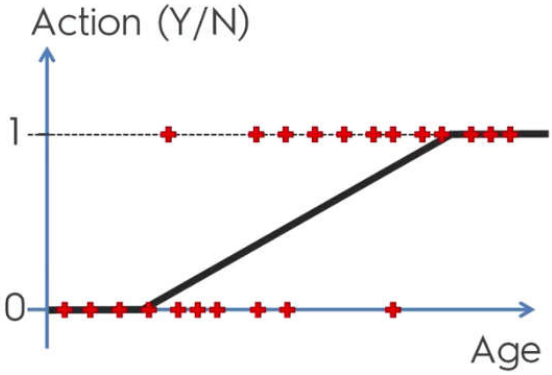


- Now, in our dataset ***taking action*** is indicated by ***0***, ***1*** (the red dots/observations are either 0 or 1). We also know that ***probabilities*** are from ***0*** to ***1***. So basically we could fit in probabilities between 0 and 1.
- Lets consider, the line is crossing $y = 1$ at age 55. And $y = 0$ at age 35.
- Here, the linear regression line, at least the part that's in the middle between 0 and 1. So those people between 35 and 55 there is a probability of them taking up this offer. Their probability is increasing as we move to the right. (more older people the probability is increasing).



- So the **part** of the linear regression in the **middle** kind of predicting the fact but lines at the top at the bottom makes no sense. A probably can never be less than zero. It can never be above 1. (following figure on Left side).
- We could interpret as is that people above 55 age they are very likely take the offer (high probability, say 100%). Anybody below 35 on the other side on the left they're definitely not taking it (probability 0%). Then the Graph becomes as in the right side as below:





In this model, we would still be able to make some sort of predictions and assumptions.

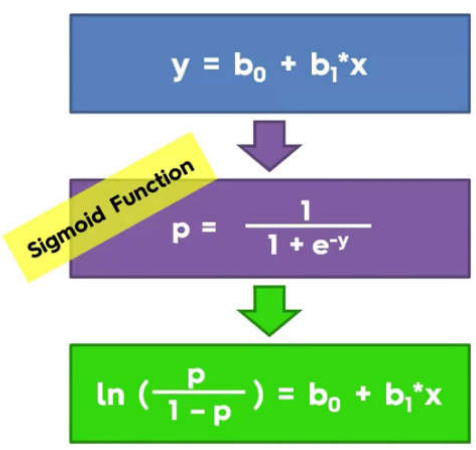
3.1.3 Logistic Regression & Sigmoid Function

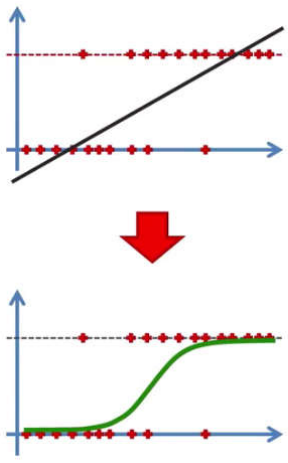
In our first attempt, where we used the Simple Linear Regression, we used the $\boldsymbol{y = b_0 + b_1 x}$ equation.
But in our last attempt we ended up with S-shaped model. For this reason we use the ***Sigmoid –Function***

$$\boldsymbol{p = \frac{1}{1 + e^{-y}}}$$

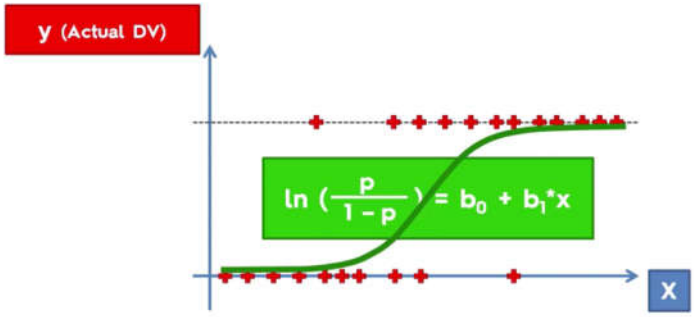
Combining ***Simple Linear equation*** and ***Sigmoid –Function***, replacing y, we get:

$$\ln\left(\frac{p}{1-p}\right) = b_0 + b_1 x$$





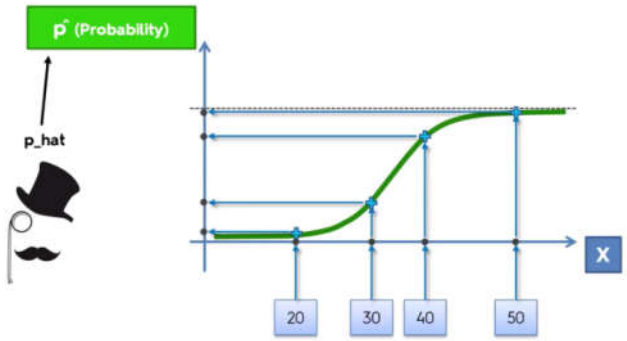
❒ So basically your linear regression will start to look like following. And $ln\left(\frac{p}{1-p}\right) = b_0 + b_1x$ is the formula for logistic regression



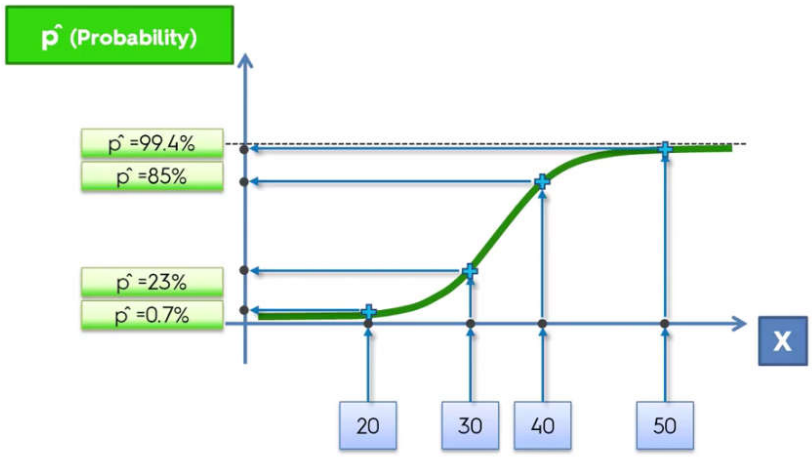
☞ This line for the logistic regression is the same as a slope for a linear regression. So basically this line is the best fitting line that can fit these data which is obtained from that formula. Basically we doing exactly the same thing as a for ***linear regression*** but it just looks ***different.***

❒ ***What can we do if this logistic regression?***

We can use it to ***predict probabilities*** instead of **predicting for sure** that something will or will not happen. Actually we predict probability. The probability here is called $\hat{p}$ (p hat) (anything you see in the ^ in the section means that it's something we're predicting.



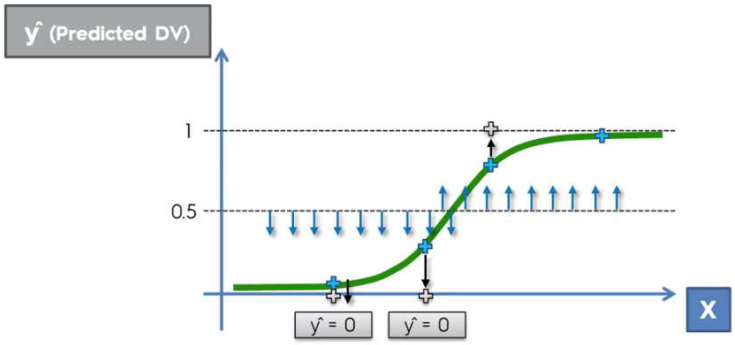
☞ ***How to predict:*** Let's take some random values 20, 30, 40, 50 as ages for the independent variable X. We first project those values to the s-shaped curve and then from the curve we again project to the y-axis (or $\hat{p}$ axis). The projected points on the $\hat{p}$ axis are the corresponding probabilities.



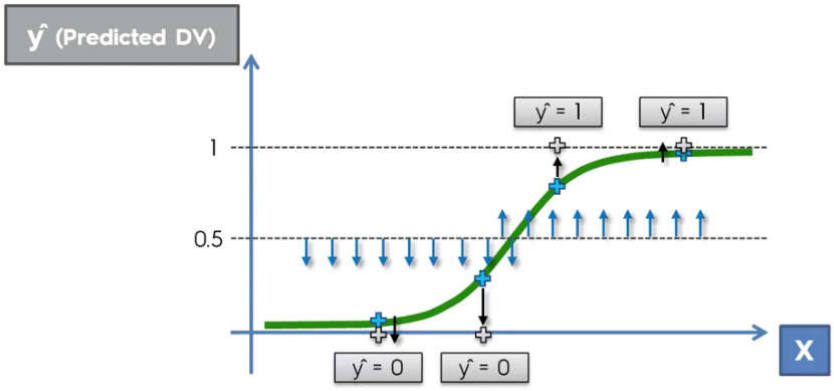
➢ The person who's 20 years old the probability of taking up this offer is very low say 0.7%, 30 years has 23%, 40 years has 85%, and 50 years has 99.4%.

➢ So that's the first thing that you can get out of a logistic regression. We're going to be using it very actively when we're talking about building some ***Demographic Segmentation*** because you use this ***probability as a score***. So it's actually even better than just having a one or a zero.

☞ ***How to Classify:*** Anyway you might want to say: ***Well I don't want probability. I want a prediction as well. I want a prediction for the y value***.



- We can only get a prediction $\hat{p}$. So $\hat{y}$ as it has suggests is a predictive value for the dependent variable. How do you get $\hat{y}$?
- Well the approach is very arbitrary. You have to select a line. In this case we're going to check 50%. You can select it anywhere but ***50%*** is just because it's in the middle and you have symmetry.
- Anything below this line (and down part of the s-curve) will be projected downwards onto the zero line. Similarly for the upper portion, which is projected on the $y = 1$.
- i.e. if your predicted probability of ***taking up this offer*** is less than ***50%***, then you just fall to $y = 0$, i.e. you are not gong to take the offer. So we are deviding up to "yes" or "no" (0 or 1), (instead of continuous [0, 1] interval)



3.1.4 Logistic Regression: Implementation in Python

We will see, how ***logistic regression*** managed just to ***separate*** some ***categories*** and predict a ***binary outcome***. To understand the implementation of ***Logistic Regression*** in ***Python***, we will use the below example:

Example: There is a dataset given which contains the **information** of various **users** obtained from the ***social networking sites***. There is a ***car making company*** that has recently launched a ***new SUV car***. So the company ***wanted to check how many users from the dataset, wants to purchase the car***.

For this problem, we will build a Machine Learning model using the ***Logistic regression algorithm***. The dataset is shown in the beside image. In this problem, we will predict the ***purchased*** variable (*Dependent Variable*) by using ***age*** and ***salary*** (*Independent variables*).

User ID	Gender	Age	EstimatedSalary	Purchased
15624510	Male	19	19000	0
15810944	Male	35	20000	0
15668575	Female	26	43000	0
15603246	Female	27	57000	0
15804002	Male	19	76000	0
15728773	Male	27	58000	0
15598044	Female	27	84000	0
15694829	Female	32	150000	1
15600575	Male	25	33000	0
15727311	Female	35	65000	0
15570769	Female	26	80000	0
15606274	Female	26	52000	0
15746139	Male	20	86000	0
15704987	Male	32	18000	0
15628972	Male	18	82000	0
15697686	Male	29	80000	0
15733883	Male	47	25000	1
15617482	Male	45	26000	1
15704583	Male	46	28000	1
15621083	Female	48	29000	1
15649487	Male	45	22000	1
15736760	Female	47	49000	1

☯ ***Steps in Logistic Regression:*** To implement the Logistic Regression using Python, we will use the same steps as we have done in previous topics of Regression. Below are the steps:

i. ***Data Pre-processing*** step
ii. ***Fitting*** Logistic Regression to the Training set
iii. ***Predicting*** the test result
iv. Test ***accuracy*** of the ***result***(Creation of ***Confusion matrix***)
v. ***Visualizing*** the test set ***result***.

❒ ***Scaling & Splitting:*** Scaling Before Splitting and Scaling after Splitting (Splitted matrices will be different because Standard-Deviation will be different)

Scaling after Splitting

```
# # Data Split : No need for this example
from sklearn.model_selection import train_test_split
# 0.25 test_size means "1/4"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler
# sc_x = StandardScaler()
# X_scaled = sc_x.fit_transform(X)
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)
```

X_test - NumPy object array

	0	1
0	-0.804802	0.504964
1	-0.0125441	-0.567782
2	-0.309641	0.157046
3	-0.804802	0.273019
4	-0.309641	-0.567782
5	-1.1019	-1.43758
6	-0.70577	-1.58254
7	-0.210609	2.15757
8	-1.99319	-0.0459058

Format | Resize | Backgroun

X_train - NumPy object array

	0	1
0	0.581649	-0.886707
1	-0.606738	1.46174
2	-0.0125441	-0.567782
3	-0.606738	1.89663
4	1.37391	-1.40858
5	1.47294	0.997847
6	0.0864882	-0.799728
7	-0.0125441	-0.248858
8	-0.210609	-0.567782

Format | Resize | Background color

Scaling Before Splitting

```
# Feature-Scaling
from sklearn.preprocessing import StandardScaler
#  y need not to be scaled: categorical variable
sc_x = StandardScaler()
X_scaled = sc_x.fit_transform(X)


# Data Split
from sklearn.model_selection import train_test_split
# 0.25 test_size means "1/4"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size= 0.25, random_state = 0)
```

X_test - NumPy object array

	0	1
0	-0.731153	0.506764
1	0.032952	-0.579736
2	-0.253587	0.154386
3	-0.731153	0.271845
4	-0.253587	-0.579736
5	-1.01769	-1.46068
6	-0.63564	-1.60751
7	-0.158074	2.18056
8	-1.87731	-0.0511683

Format | Resize | Backgroun

X_train - NumPy object array

	0	1
0	0.606031	-0.902749
1	-0.540127	1.4758
2	0.032952	-0.579736
3	-0.540127	1.91628
4	1.37014	-1.43132
5	1.46565	1.00597
6	0.128465	-0.814655
7	0.032952	-0.256722
8	-0.158074	-0.579736

Format | Resize | Background color

❒ ***Fit , Predict: & Confusion matrix***

```
# Fit dataset to Logistic regression
from sklearn.linear_model import LogisticRegression # import class
# instead of "regressor" we now use "classifier"
classifer = LogisticRegression(random_state= 0) # create object
classifer.fit(X_train, y_train) # fit the dataset

# Predict
y_pred = classifer.predict(X_test)
```

☞ ***Why Linear?*** It's because the logistic regression is a linear classifier, means that we're in two dimensions and two categories of users are going to be separated by a straight line.

☞ ***Confusion Matrix:*** Evaluating the Model-Performance using Confusion Matrix This confusion matrix is going to contain the correct and incorrect predictions that our model made on the set.

cm - NumPy object array

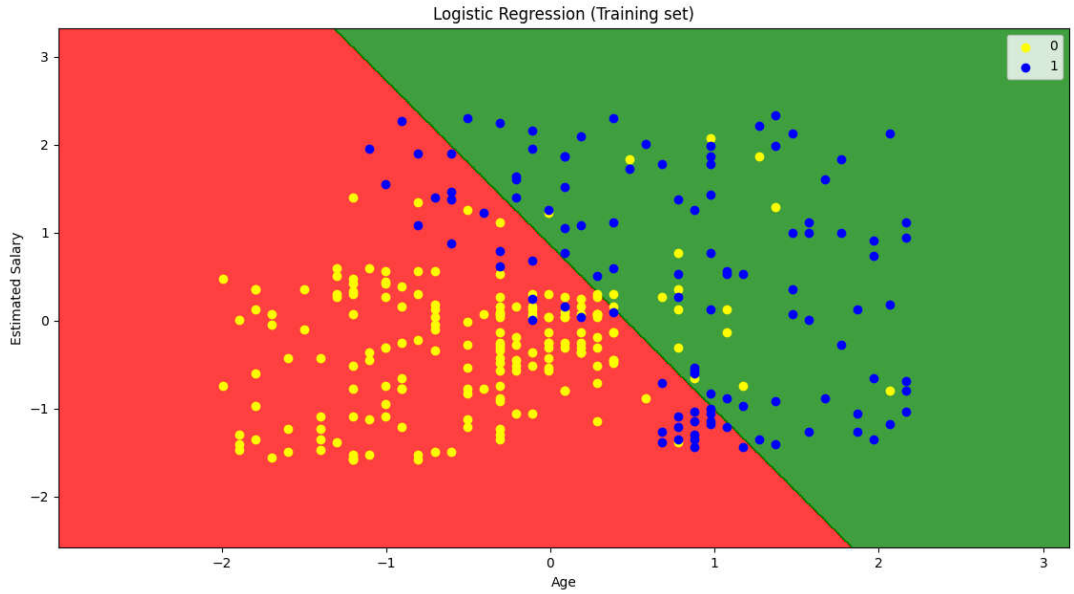
	0	1
0	65	3
1	8	24

☞ ***Test Accuracy of the result:*** We can find the accuracy of the predicted result by **interpreting** the **confusion matrix**. By above output, we can interpret that ***65+24= 89 (Correct Output)*** and ***8+3= 11(Incorrect Output)***.

❒ ***Visualizing the training set result:*** Finally, we will visualize the training set result. To visualize the result, we will use ***ListedColormap*** class of ***matplotlib*** library. Idea is: we ***divide*** the ***region*** into ***pixels*** with ***0.01*** size .

```
# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('yellow', 'blue'))(i), label = j)
pLt.title('Logistic Regression (Training set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()
```

- In the above code, we have imported the ***ListedColormap*** class of ***Matplotlib*** library to create the ***colormap*** for ***visualizing*** the result.
- We have created two new variables ***x_set*** and ***y_set*** to replace ***x_train*** and ***y_train***.
- After that, we have used the ***np.meshgrid*** command to create a ***Rectangular Grid***, which has a range of ***-1(minimum)*** to ***1 (maximum)***. The **pixel points** we have taken are of ***0.01*** **resolution**.
- To create a ***Filled Contour***, we have used ***pLT.contourf*** command, it will create ***regions*** of ***provided colors*** (yellow and blue). In this function, we have passed the ***classifer.predict*** to show the **predicted data points** predicted by the **classifier**.
- ***Output:*** By executing the above code, we will get the below output:



- ***The graph can be explained in the below points:***
  - In the above graph, we can see that there are some **Blue points** within the ***green region*** and **Yellow points** within the ***red region***.
  - All these data points are the **observation points** from the **training set**, which shows the result for ***purchased variables***.
  - This graph is made by using two independent variables i.e., ***Age on the x-axis*** and ***Estimated salary on the y-axis***.
  - The ***Yellow point observations*** are for which purchased (dependent variable) is probably ***0***, i.e., users who ***did not purchase*** the SUV car.
  - The ***Blue point observations*** are for which ***purchased*** (dependent variable) is ***probably 1*** means user who ***purchased*** the SUV car.
  - We can also estimate from the graph that the users who are ***younger*** with ***low salary***, did not purchase the car, whereas ***older users*** with high estimated ***salary*** purchased the ***car***.

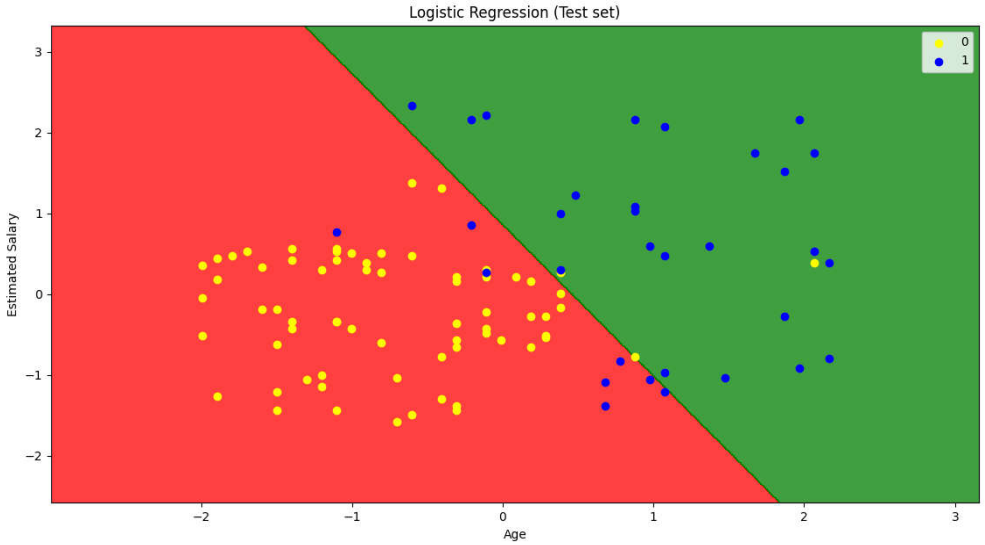
- But there are some ***Yellow points*** in the ***green region*** (Not Buying the car) and some ***Blue points*** in the ***red region***(buying the car). So we can say that younger users with a high estimated salary purchased the car, whereas an older user with a low estimated salary did not purchase the car.

❒ ***The goal of the classifier:*** We have successfully visualized the training set result for the logistic regression, and our goal for this classification is to divide the users who ***purchased the SUV*** car and who ***did not purchase the car*** *(as prediction regions).* So from the output graph, we can clearly see the two regions (Red and Green) with the observation points. The *Red region is for those users who didn't buy the car*, and *Green Region is for those users who purchased the car*. ***Classify right user into right category (categorize the data)***.

❒ ***Linear Classifier:*** As we can see from the graph, the classifier is a ***Straight line*** or *linear* in nature as we have used the ***Linear model*** for ***Logistic Regression***. In further topics, we will learn for ***non-linear Classifiers***. The line is called *"prediction boundary"*.

❒ ***Visualizing the test set result:*** Our model is well trained using the training dataset. Now, we will visualize the result for new observations (***Test set***). The code for the test set will remain same as above except that here we will use ***x_test*** and ***y_test*** instead of ***x_train*** and ***y_train***. Below is the code for it:

```
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('yellow', 'blue'))(i), label = j)
pLt.title('Logistic Regression (Test set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()
```



Full Code (Practiced)

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
```

```python
# Data Split
from sklearn.model_selection import train_test_split
# 0.25 test_size means "1/4"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler
#  y need not to be scaled: categorical variable
# sc_x = StandardScaler()
# X_scaled = sc_x.fit_transform(X)
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)



"""
# Feature-Scaling
from sklearn.preprocessing import StandardScaler
#  y need not to be scaled: categorical variable
sc_x = StandardScaler()
X_scaled = sc_x.fit_transform(X)


# Data Split
from sklearn.model_selection import train_test_split
# 0.25 test_size means "1/4"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size= 0.25, random_state = 0)

"""



# Fit dataset to Logistic regression
from sklearn.linear_model import LogisticRegression # import class
# instead of "regressor" we now use "classifier"
classifer = LogisticRegression(random_state= 0) # create object
classifer.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = classifer.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
# Class in capital letters, functions are small letters
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value



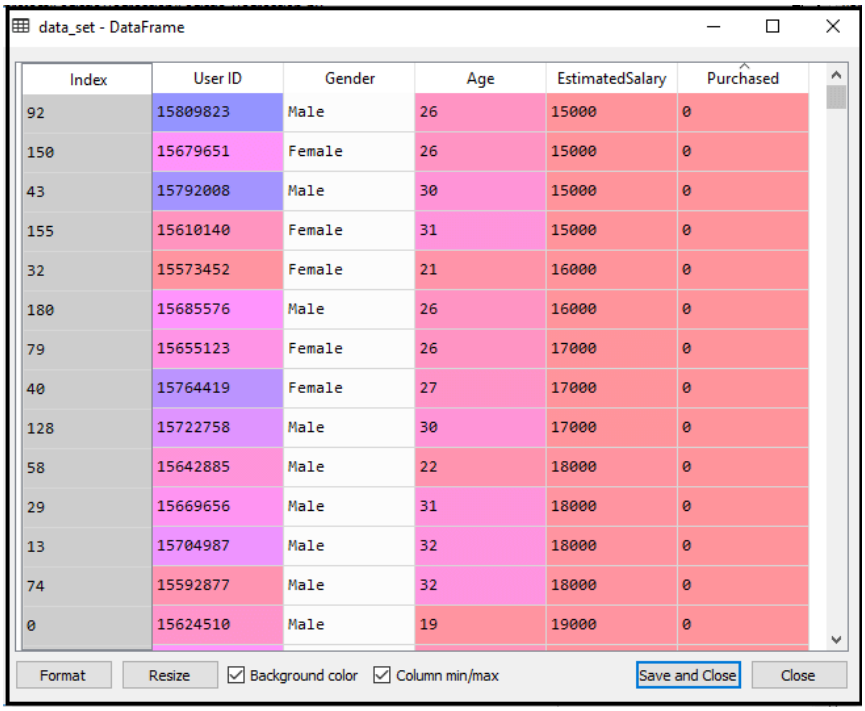
# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('yellow', 'blue'))(i), label = j)
pLt.title('Logistic Regression (Training set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
```

```
pLt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('yellow', 'blue'))(i), label = j)
pLt.title('Logistic Regression (Test set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# python prctc_lgstc.py
```

❒ We will get the dataset as the output. Consider the given image:



Index	User ID	Gender	Age	EstimatedSalary	Purchased
92	15809823	Male	26	15000	0
150	15679651	Female	26	15000	0
43	15792008	Male	30	15000	0
155	15610140	Female	31	15000	0
32	15573452	Female	21	16000	0
180	15685576	Male	26	16000	0
79	15655123	Female	26	17000	0
40	15764419	Female	27	17000	0
128	15722758	Male	30	17000	0
58	15642885	Male	22	18000	0
29	15669656	Male	31	18000	0
13	15704987	Male	32	18000	0
74	15592877	Male	32	18000	0
0	15624510	Male	19	19000	0

New TEMPLATE for Classification Problem

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values


        # Feature-Scaling after Data Split

# Data Split
from sklearn.model_selection import train_test_split
# 0.25 test_size means "1/4"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler
#  y need not to be scaled: categorical variable
# sc_x = StandardScaler()
# X_scaled = sc_x.fit_transform(X)
st_x= StandardScaler()
```

```
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)


        # Feature-Scaling before Data Split
"""
# Feature-Scaling
from sklearn.preprocessing import StandardScaler
#  y need not to be scaled: categorical variable
sc_x = StandardScaler()
X_scaled = sc_x.fit_transform(X)


# Data Split
from sklearn.model_selection import train_test_split
# 0.25 test_size means "1/4"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size= 0.25, random_state = 0)

"""



# Fit train set to Model classifier
from sklearn._ import
clsFier =
clsFier.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = clsFier.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
# Class in capital letters, functions are small letters
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value



# Visualizing the trainig set resultl
# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('yellow', 'blue'))(i), label = j)
pLt.title('Logistic Regression (Training set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

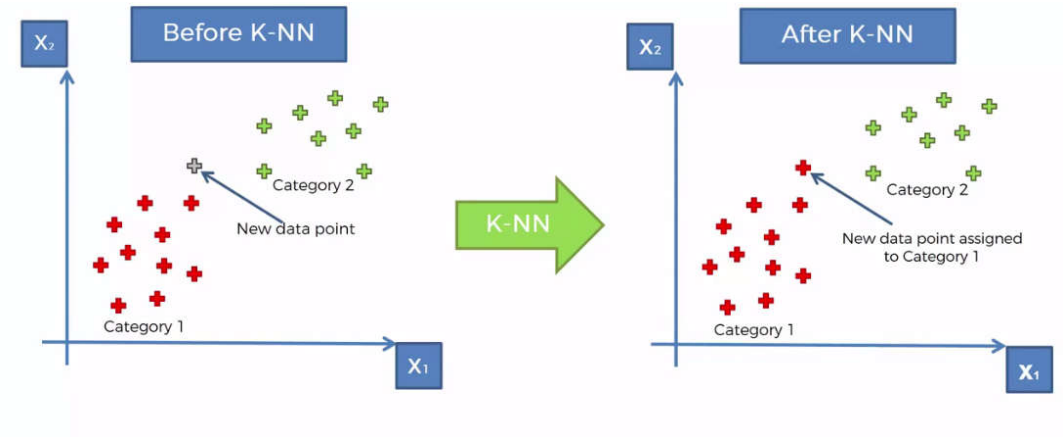
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('yellow', 'blue'))(i), label = j)
pLt.title('Logistic Regression (Test set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()
```

Chapter 3 : part 2

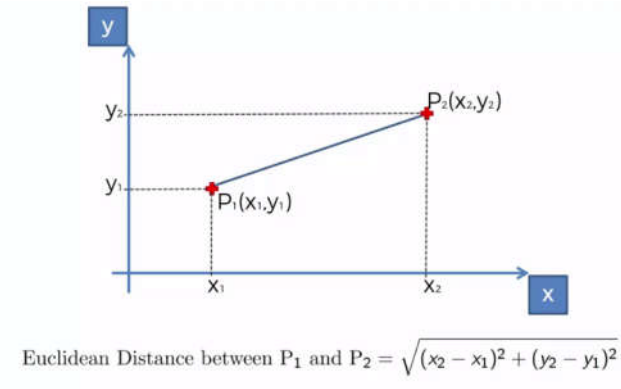
Logistic Regression

3.2.1 Logistic Regression

- Let's imagine that we have ***two categories*** *already present* in our data set so we've identified two categories: Green and Red.
  - For simplicity's let's consider two variables or two columns x_1 and x_2 in our data-set so all of this grouping is happening based on these two columns.
  - Now let's say we ***add a new data point*** into our data set. The question is should it fall into the ***Red category*** or should fall to the ***Green category***.

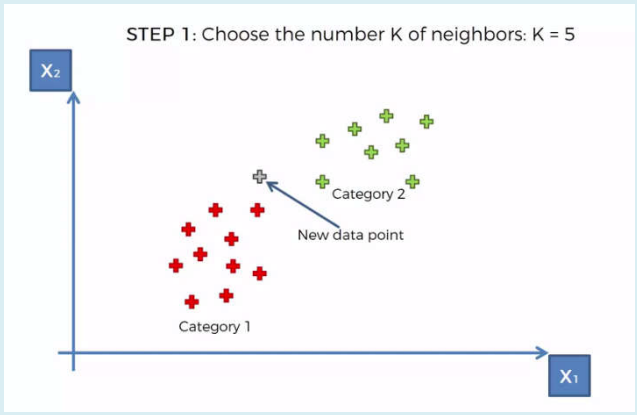


- ***K-Nearest Neighborhoods:*** The steps are given below:
  i. ***STEP 1:*** **Choose** the **number** ***K*** of neighbors (select the ***no. of points*** which will be the Neighborhood).
  ii. ***STEP 2:*** Calculate the ***Euclidean distance*** of ***K number of neighbors***
  iii. ***STEP 3:*** Take the ***K nearest neighbors*** of the ***new data point***, according to the ***Euclidean distance*** (we can also use other kind of distances)
  iv. ***STEP 4:*** Among these ***K neighbors***, ***count the Number*** of data points in **each category** (you might even have more than two categories in your data set).
  v. ***STEP 5:*** ***Assign*** the ***new data point*** to the category where you counted the ***most neighbors***
  vi. ***Final:*** Your Model is Ready

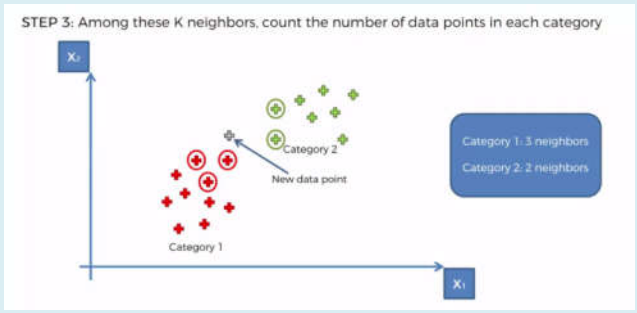


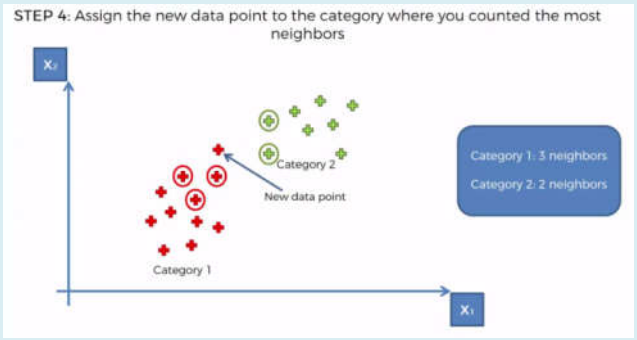
- ***Euclidean distance :*** The formula is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, whwere the points (x_1, y_1) are (x_2, y_2)

- ***How we classify:*** Let's no. of points in our nbd is **K=5**.
  - Now take the K nearest neighbors of the new data point according to their Euclidean distance.
  - Once you've taken the nearest neighbors among these K neighbors you need to count the number of data points in each category.
  - So how many data points fell into each category (you might even have more than two categories in your data set).
  - Just calculate how many fall into each category and then you need to assign the new data point to the category where you counted the most neighbors.









❒ ***How to select the value of K in the K-NN Algorithm?:*** Below are some points to remember while selecting the value of K in the K-NN algorithm:

- ☞ There is ***no particular way*** to determine the best value for "K", so we need to ***try some values*** to find the best out of them. The most ***preferred*** value for ***K*** is ***5***.
- ☞ A very low value for ***K*** such as ***K=1*** or ***K=2***, can be ***noisy*** and lead to the effects of ***outliers*** in the model.
- ☞ Large values for K are good, but it may find some difficulties.

❒ ***Advantages of KNN Algorithm:***

- ➢ It is simple to implement.
- ➢ It is robust to the noisy training data
- ➢ It can be more effective if the training data is large.

❒ ***Disadvantages of KNN Algorithm:***

- ➢ Always needs to determine the value of K which may be complex some time.
- ➢ The computation cost is high because of calculating the distance between the data points for all the training samples.

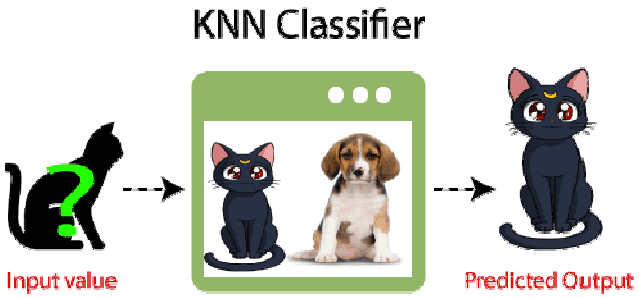
NOTES:

- ☝ K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on ***Supervised Learning*** technique.
- ☝ K-NN algorithm assumes the ***similarity*** between the ***new case/data*** and available cases and put the new case into the ***category*** that is ***most similar*** to the ***available categories***.
  - ☞ K-NN algorithm stores all the available data and classifies a new data point based on the ***similarity***. This means when ***new data*** appears then it can be easily ***classified*** into a ***well suite category*** by using K- NN algorithm.
- ☝ K-NN algorithm can be used for ***Regression*** as well as for ***Classification*** but mostly it is used for the ***Classification*** problems.
- ☝ ***Non-Parametric Algorithm:*** K-NN is a ***non-parametric algorithm***, which means it ***does not make any assumption on underlying data***.

☝ ***Lazy learner algorithm:*** It is also called a ***lazy learner algorithm*** because it *does not learn from the training set immediately instead it stores the dataset and at the time of classification*, it performs an action on the dataset.

☝ KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.

☯ ***Example:*** Suppose, we have an image of a ***creature*** that looks similar to ***cat*** and ***dog***, but we want to know either it is a ***cat*** or ***dog***. So for this ***identification***, we can use the ***KNN algorithm***, as it works on a similarity measure. Our ***KNN model will find the similar features*** of the ***new data set*** to the ***cats*** and ***dogs*** images and based on the ***most similar features*** it will put it in either cat or dog category.



3.2.2 Python implementation of the KNN algorithm

To do the ***Python implementation*** of the ***K-NN*** algorithm, we will use the ***same problem*** and ***dataset*** which we have used in ***Logistic Regression***. But here we will ***improve*** the ***performance*** of the model. Below is the problem description:

❒ ***Problem for K-NN Algorithm:*** There is a Car manufacturer company that has manufactured a new SUV car. The company wants to give the ads to the users who are interested in buying that SUV. So for this problem, we have a dataset that contains multiple user's information through the social network. The dataset contains lots of information but the Estimated ***Salary*** and ***Age*** we will consider for the ***independent*** variable and the ***Purchased*** variable is for the ***dependent*** variable. Below is the dataset:

User ID	Gender	Age	EstimatedSalary	Purchased
15624510	Male	19	19000	0
15810944	Male	35	20000	0
15668575	Female	26	43000	0
15603246	Female	27	57000	0
15804002	Male	19	76000	0
15728773	Male	27	58000	0
15598044	Female	27	84000	0
15694829	Female	32	150000	1
15600575	Male	25	33000	0
15727311	Female	35	65000	0
15570769	Female	26	80000	0
15606274	Female	26	52000	0
15746139	Male	20	86000	0
15704987	Male	32	18000	0
15628972	Male	18	82000	0
15697686	Male	29	80000	0
15733883	Male	47	25000	1
15617482	Male	45	26000	1
15704583	Male	46	28000	1
15621083	Female	48	29000	1
15649487	Male	45	22000	1
15736760	Female	47	49000	1

☯ ***Steps to implement the K-NN algorithm:***

i. Data Pre-processing step
ii. Fitting the K-NN algorithm to the Training set
iii. Predicting the test result
iv. Test accuracy of the result(Creation of Confusion matrix)
v. Visualizing the test set result.

❒ ***Data Pre-Processing Step:*** The ***Data Pre-processing*** step will remain exactly the ***same*** as ***Logistic Regression***.

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# data Extract
dtaSet = pd.read_csv("Social_Network_Ads.csv")
X = dtaSet.iloc[:, [2, 3]].values
y = dtaSet.iloc[:, 4].values
```

```
from sklearn.model_selection import train_test_split
# 0.25 test_size means "1/4"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)
```



❒ ***Fitting K-NN classifier to the Training data:*** Now we will fit the K-NN classifier to the training data. To do this we will import the ***KNeighborsClassifier*** class of ***Sklearn Neighbors*** library. After importing the class, we will create the ***Classifier*** object of the class. The Parameter of this class will be

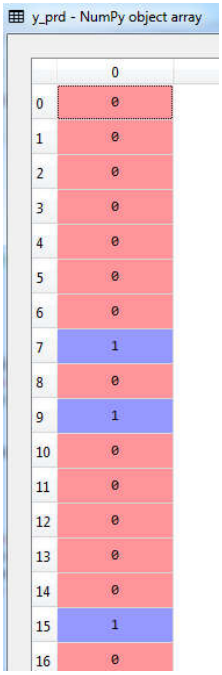
- `n_neighbors:` To define the required neighbors of the algorithm. Usually, it takes 5.
- `metric='minkowski':` This is the default parameter and it decides the distance between the points.
- `p=2:` It is equivalent to the standard Euclidean metric.

And then we will fit the classifier to the training data. Below is the code for it:

```
# Fitting K-NN classifier to the training set
from sklearn.neighbors import KNeighborsClassifier
clsFier = KNeighborsClassifier(p = 2, metric= "minkowski", n_neighbors=5)
clsFier.fit(X_train, y_train) # fit the dataset
```

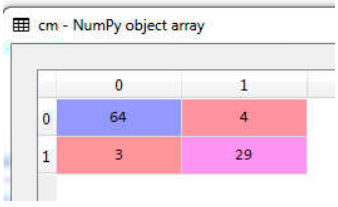
❒ ***Predicting the Test Result:*** To predict the test set result, we will create a y_pred vector as we did in Logistic Regression. Below is the code for it:

```
# Predicting the test set result
y_prd = clsFier.predict(X_test)
```



☞ ***Creating the Confusion Matrix:*** In below code, we have imported the confusion_matrix function and called it using the variable cm.

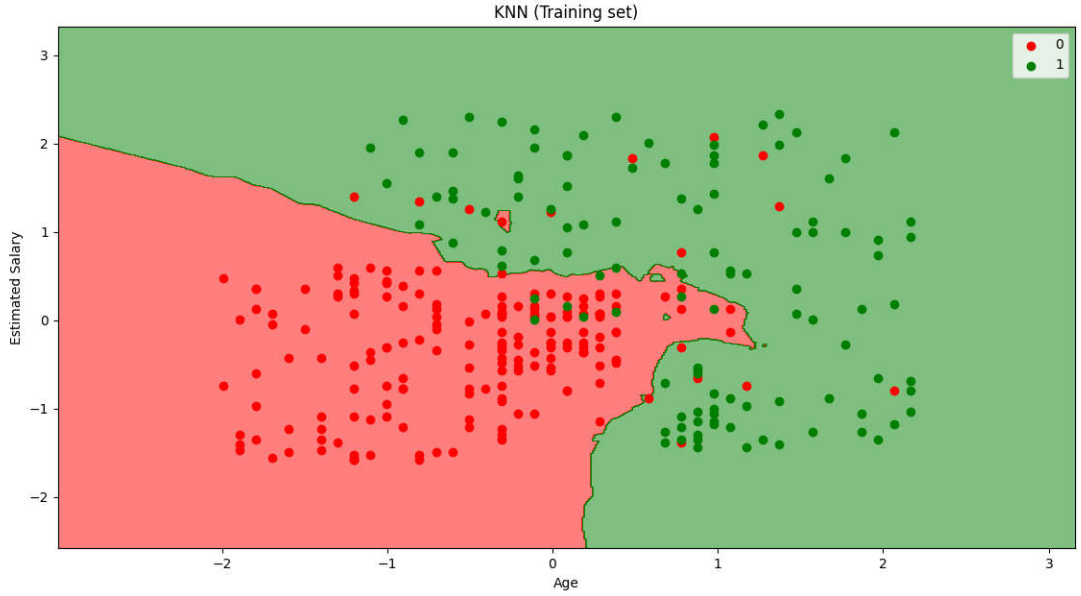
```
# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value
```



We can see there are ***64+29= 93 correct predictions*** and ***3+4= 7 incorrect predictions***, whereas, in ***Logistic Regression***, there were ***11 incorrect predictions***. So we can say that the performance of the model is improved by using the K-NN algorithm.

❒ ***Visualizing the Training set result:*** Now, we will visualize the training set result for K-NN model. The code will remain same as we did in Logistic Regression, except the name of the graph. Below is the code for it:

```
# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Logistic Regression (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```



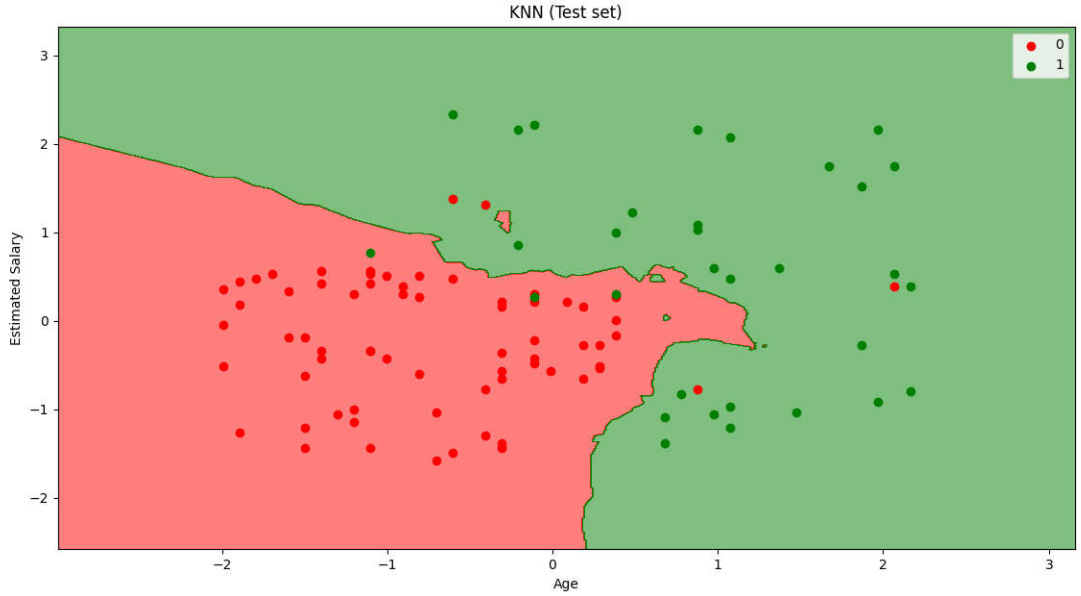
➢ The output graph is different from the graph which we have occurred in ***Logistic Regression***. It can be understood in the below points:

- ❖ As we can see the graph is showing the ***red*** point and ***green*** points. The green points are for ***Purchased(1)*** and Red Points for ***not Purchased(0)*** variable.
- ❖ The graph is showing an *irregular boundary* instead of showing any ***straight line*** or any curve because it is a K-NN algorithm, i.e., finding the nearest neighbor.
- ❖ The graph has classified users in the ***correct categories*** as most of the users who ***didn't buy*** the SUV are in the ***red*** region and users who ***bought*** the SUV are in the ***green*** region.
- ❖ The graph is showing good result but still, there are ***some green points*** in the ***red region*** and ***red points*** in the ***green region***. But this is no big issue as by doing this model is prevented from ***overfitting*** issues.
- ❖ Hence our model is well trained.

❒ ***Visualizing the Test set result:*** After the ***training*** of the model, we will now test the result by putting a new dataset, i.e., ***Test dataset***. Code remains the same except some minor changes: such as ***x_train*** and ***y_train*** will be replaced by ***x_test*** and ***y_test***.

```
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
```

```
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()
```



- ☞ The above graph is showing the output for the test data set. As we can see in the graph, the predicted output is well good as most of the red points are in the red region and most of the green points are in the green region.
- ☞ However, there are few green points in the red region and a few red points in the green region. So these are the incorrect observations that we have observed in the confusion matrix(7 Incorrect output).

☝ ***Nlinear – Non_Linear Classifier:*** If the *"Prediction Boundary"* is linear then the classification model is called linear model.

Practiced version

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# data Extract
dtaSet = pd.read_csv("Social_Network_Ads.csv")
X = dtaSet.iloc[:, [2, 3]].values
y = dtaSet.iloc[:, 4].values


# Data Split
from sklearn.model_selection import train_test_split
# 0.25 test_size means "1/4"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)

# Fitting K-NN classifier to the training set
from sklearn.neighbors import KNeighborsClassifier
clsFier = KNeighborsClassifier(p = 2, metric= "minkowski", n_neighbors=5)
clsFier.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = clsFier.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value
```

```python
# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('KNN (Training set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('KNN (Test set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# python prctc_knn.py
```

Chapter 3 : part 3

Support Vector Machine

3.3.1 Decision Boundary

While ***training*** a ***classifier*** on a ***dataset***, using a specific ***classification algorithm***, it is required to define a ***Set Of Hyper-Planes***, called ***Decision Boundary***, that **separates the data points** into **specific classes**, where the algorithm switches from one class to another. On one side a decision boundary, a data-points is more likely to be called as ***class A*** — on the other side of the boundary, it's more likely to be called as ***class B***.

☯ For example in our ***Logistic regression example***: we had, $p = \frac{1}{1+e^{-y}}$ and $y = b_0 + b_1 x$ implies,

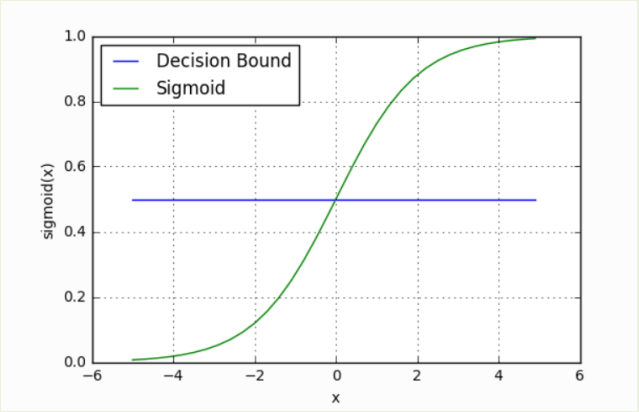
$$p = \frac{1}{1 + e^{-(b_0 + b_1 x)}}$$

☞ So, our current prediction function returns a ***Probability Score*** between ***0*** and ***1***. In order to map this to a ***Discrete Class (A/B)***, we select a ***Threshold Value*** or ***Tipping Point*** above which we will classify values into ***Class A*** and below which we classify values into ***Class B***. This ***Threshold Value*** is frequently denoted by p

$p \geq 0.5$: class=A

$p \leq 0.5$: class=B

➢ If our threshold was ***.5*** and our prediction function returned ***.7***, we would classify this **observation belongs** to ***Class A***. If our prediction was ***.2*** we would classify the **observation belongs** to ***Class B***.



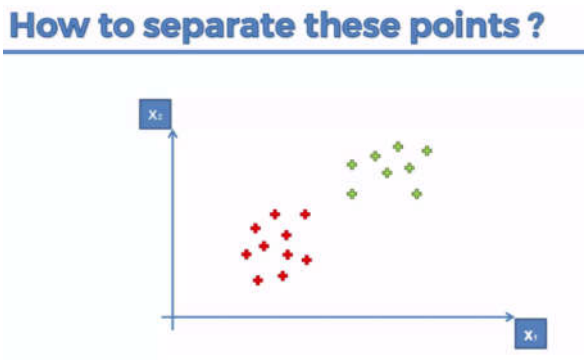
In order to map predicted values to probabilities, we use the ***Sigmoid function***.

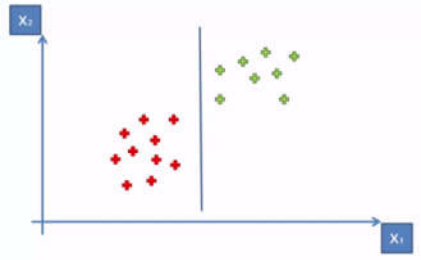
☞ ***Decision Boundary:*** Hence, line with ***0.5*** i.e. the horizontal line $y = 0.5$ is called the ***Decision Boundary***.

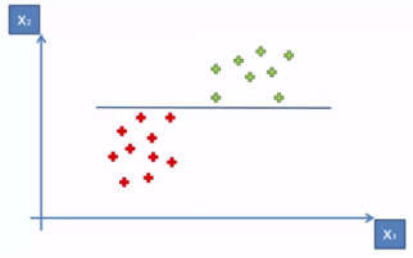
3.3.2 Support Vector Machine

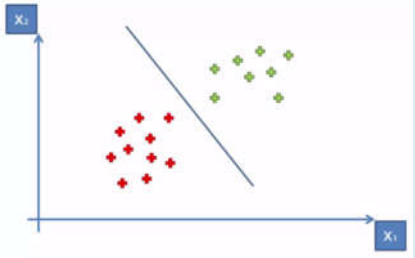
For simplicity's sake consider two columns features x_1 and x_2. And observations are already classified.

❒ The problem is how we divide them? There are some following approaches that we can draw a line (Decision Boundary) that's going to separate them: we can use ***vertical*** or ***horizontal*** line or ***diagonal lines*** as follows



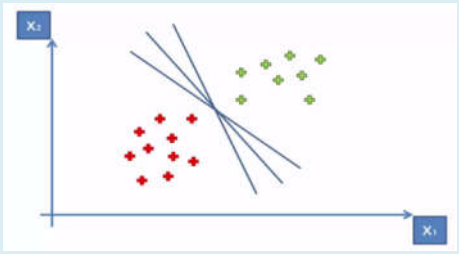






☞ So actually there is so many choice for ***Decision Boundary***. But, ***Decision Boundary*** plays *Main role in classification*, because that's a ***separation***. When we start *adding new points*, to ***classify*** those ***new points*** we need to use the ***Decision Boundary***. So our goal is to find ***"Best Decision Boundary "***.

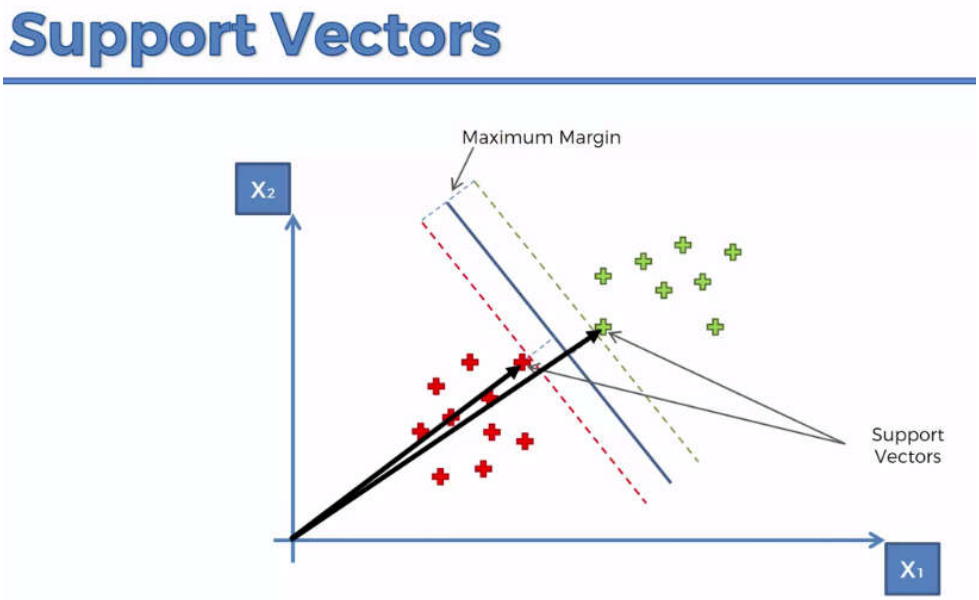
☞ So let's find out how the SVM actually searches for this line (Decision Boundary).



Actually there are so many options for a diagonal line:

❒ ***Decision Boundary using SVM:*** Actually this **Decision Boundary** line is searched through the **maximum margin**. Basically it's *the line that separates these two class's of points* and at the same time it has the ***maximum margin.***

☞ Maximum margins: Which means the distance from the drawn line is equidistant from the points in both sides. So the sum of these two ***distances (total margin) has to be maximized*** in order for this line to be the result of the SVM (***"Best Decision Boundary "***).



❒ ***Support vectors:*** The closest points on which the maximum margin depends (supported) are called the support vectors, because they *literally* **supports** the ***decision boundary***. In above figure, the ***two points*** touching the ***maximum margin***, are actually called the ***support vectors***.

☞ These two points are supporting this whole algorithm. So even if you ***get rid of all the rest of the points*** the algorithm will be exactly the same.

☞ So these other points they don't contribute to the result of the algorithm only these two points are contributing and therefore they called the supporting vectors (you can call them supporting points but in reality they are vectors in multi-dimensions).

☞ Hence these two specific vectors are the ones kind of supporting this decision boundary that's why this whole algorithm is called the support vector machines.

❒ ***Maximum Margin Hyper-planes:*** We've got the line in the middle which is called the ***`maximum margin hyperplane`*** or the ***`maximum margin classifier`***. So in a two dimensional space this classifier is just the ***`line`***, but actually in a multidimensional space it's a ***`hyperplane`***.

☞ You can draw a different hyperplanes, but above is always be less because this is the one with the maximum margin.

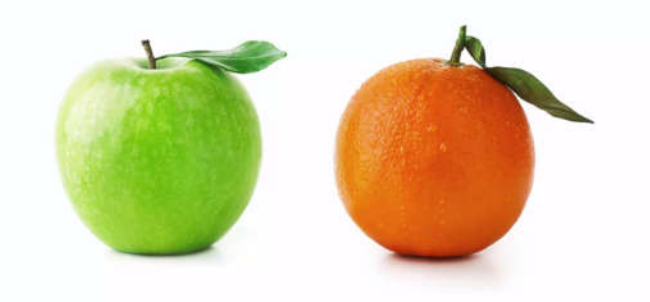
❒ ***Positive Hyperplane And Negative Hyperplane:*** In the above figure, notice the ***green*** and the ***red dotted*** lines. The ***green*** one is called the ***Positive Hyperplane*** and the ***red*** one called the ***Negative Hyperplane***.

☞ Anything to the right of the positive is classified as the green category (or the positive category) anything to the left to classify as a (negative category or) the red category in our case.

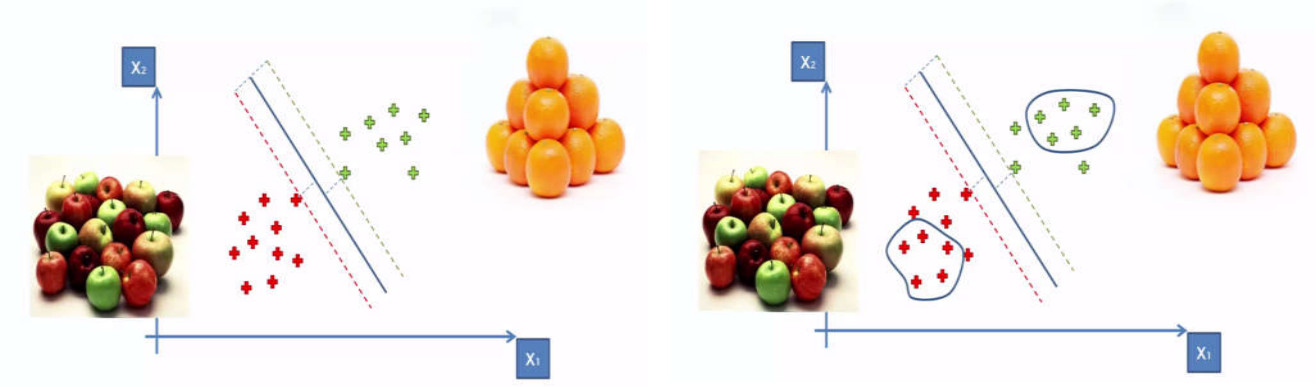
So that's how the supervision machine algorithm works of course there's some complicated mathematics behind it.

3.3.3 What's so special about SVM

Classify a fruit into either an apple an orange: So imagine you're trying to teach a machine how to distinguish between apples and oranges. Now in our case here you can see let's say on the right we have oranges on the left we have apples.

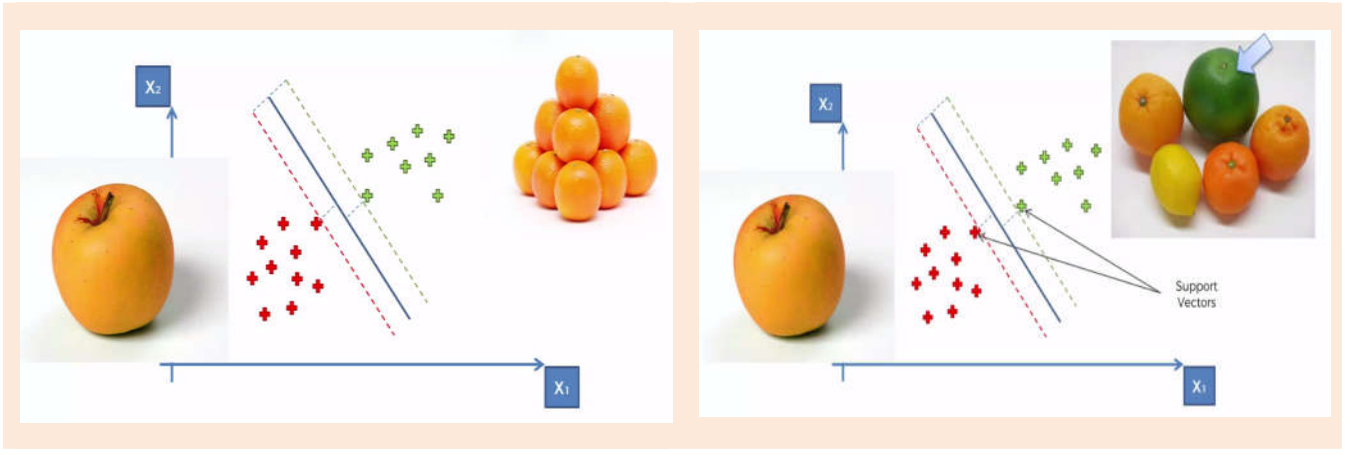


Apple-ly-apple & Orangey—orange



❒ Most of ML algorithms would do is they would look at the ***Most Apple/ Most Orange*** to match the given "***fruit***" (i.e. is the given fruit is most ***"Apple-ly Apple: Apple that is more like Apple"*** or ***"Most Orenge-y Orange: Orange that is more like Orange "***. Maximize the fruit character).

☝ But ***SVM does the reverse***; it looks at the ***marginal values*** (i.e. ***Orenge-y Apple:*** "***Apple*** that is more ***like Orange***" or ***Apple-ly Orange:*** "***Orange*** that is more ***like Apple***"). Those are the ***support vectors***. You can see that they're actually very close to the boundary so their ***"Orange/apple fruit characteristics are very close to each other"***.



☞ Therefore the support vector machine in that sense you can think of it is like a more extreme type of algorithm a very rebellious type of algorithm a very risky type of algorithm because it looks at a very extreme case which is very close to the boundary and it uses that to construct its analysis.

And that in itself makes the ***Support Vector Machine (SVM) Algorithms*** very special very different to most of the other ***Machine Learning Algorithms***.

3.3.4 Python implementation of the SVM algorithm

Practiced Version

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values


        # Feature-Scaling after Data Split

# Data Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler

st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)


# Fit train set to Model classifier
from sklearn.svm import SVC
clsFier = SVC(kernel="linear", random_state=0)
clsFier.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = clsFier.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value


# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('SVM (Training set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('SVM (Test set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# python prctc_SVM.py
```

❒ ***Data Preprocessing*** is Similar to previous KNN Example.

❒ ***Classifier:*** We use the ***`kernel = "Linear"`***. It is the part of Kernel-SVM, default is "***`rbf`***" (***radial basis function kernel***, or ***RBF kernel***, is a popular ***kernel function*** used in various ***kernelized learning algorithms***) whish is mostly "Gaussian". " ***`kernel = "Linear"`***" means we do-not apply any Kernel-trick.

```
# Fit train set to Model classifier
from sklearn.svm import SVC
clsFier = SVC(kernel="linear", random_state=0)
clsFier.fit(X_train, y_train) # fit the dataset
```

❒ ***Prediction and confusion matrix:*** From following figures, we can see that the result is almost same as the "Previous Logistic Regression". Hence model performance is not so much improved. We use kernel-SVM in next section then the performance will increase.

Predicted y-values

y_test - NumPy object arra

	0
30	0
31	1
32	1
33	0
34	0
35	0
36	0
37	0
38	0

Format Resize

y_prd - NumPy object array

	0
30	0
31	0
32	1
33	0
34	0
35	0
36	0
37	0
38	0

Format Resize

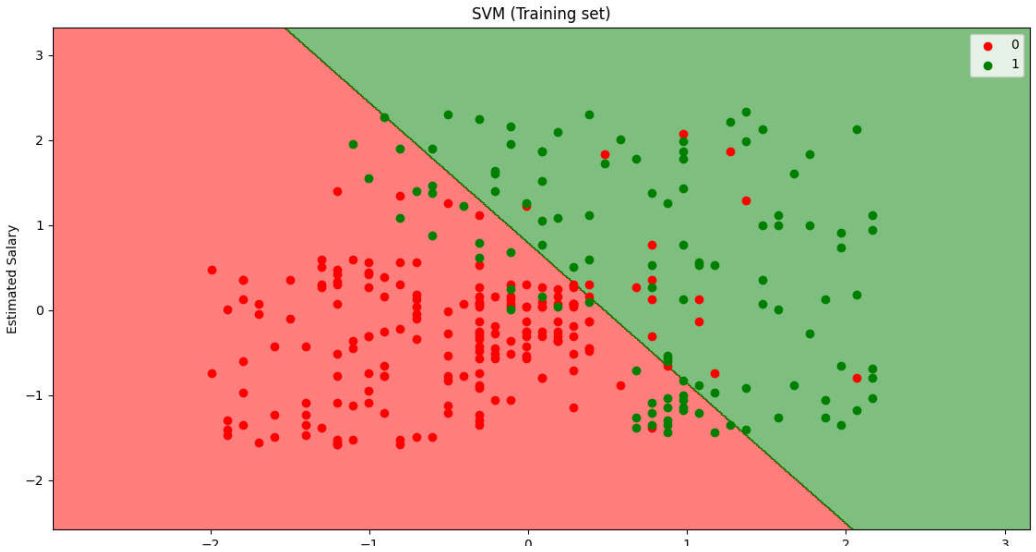
Confusion matrix

cm - NumPy object array

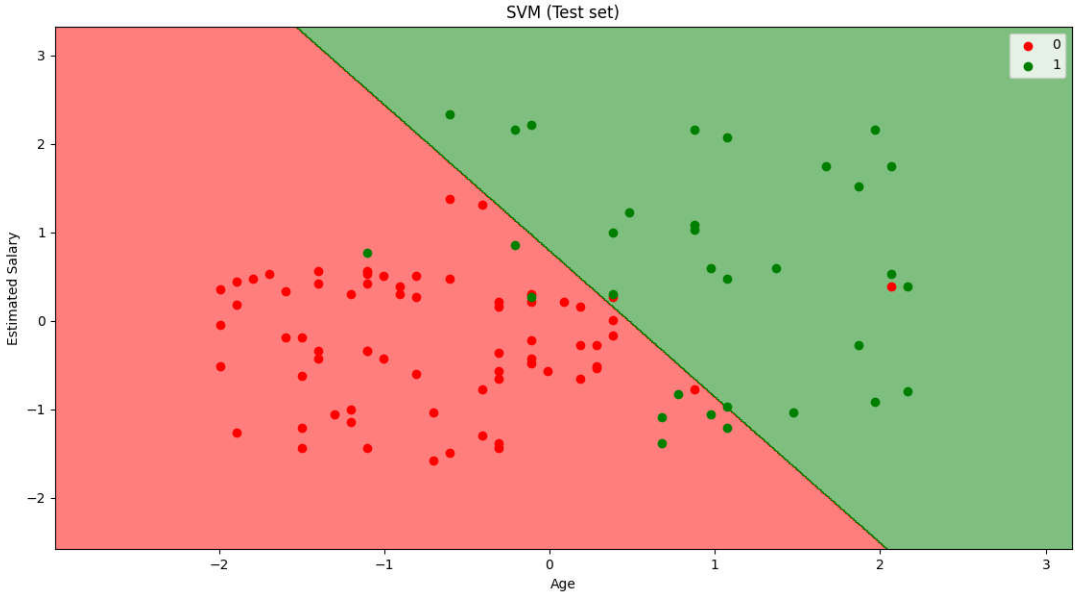
	0	1
0	66	2
1	8	24

❒ ***Visualizing the data:*** Same as KNN-algorithm section

Training Set



Test Set



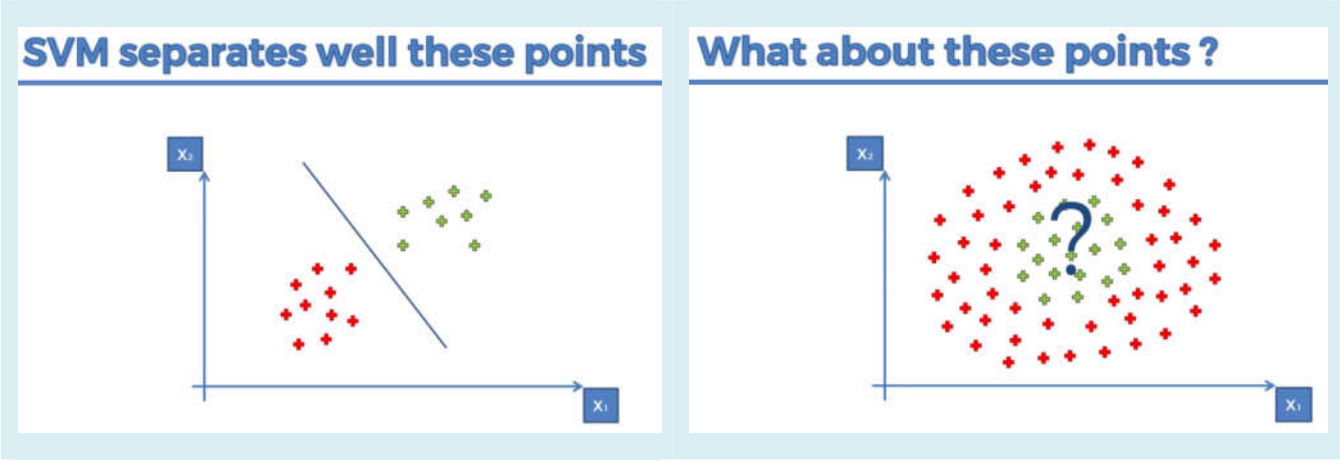
Chapter 3 : part 4

Kernel - SVM

Kernel - Support Vector Machine

3.4.1 Kernel - SVM

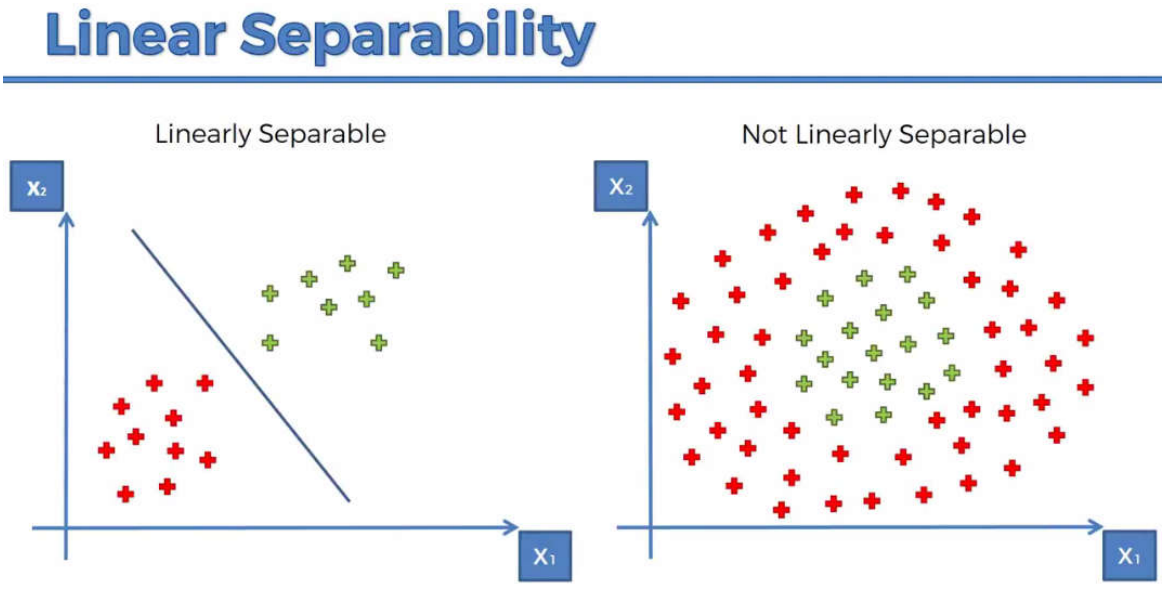
Recall in the ***SVM*** situation we had a set of observations which belong to ***different classes*** and the ***algorithm*** will find the ***decision boundary*** between them so that any ***future observations*** could be ***identified*** which ***Class*** they fall into.



☺ In first case we can see that there a decision boundary and the support vector machine algorithm tells us exactly how to find that boundary.

☹ ***Problem is the data is not always linearly separable:*** In the second case, we cannot separate the points in the same way that the ***SVM algorithm*** told us to. This happens because the data points are not ***LINEARLY SEPARABLE***.

❒ ***Data Is Not Linearly Separable:*** Well this happens because in this case the ***data is not linearly separable***. So here we've got the two examples side by side on the left the linearity separable data and on the right the nonlinear separable data.



☞ ***SVM*** algorithm helps us find that **Decision Boundary** or correctly place that **Decision Boundary**. But it does have an assumption. The ***assumption*** is that the ***data must be separable***.

➢ But in that ***Non Linear Separable Case*** we can't even draw one single ***Decision Boundary*** or ***Linear Decision Boundary*** so therefore the ***Support Vector Machine*** algorithm ***just won't work by definition***.

3.4.2 Tricks to deal with Non Linear Separable Data

[1] ***Mapping to a higher dimension:*** First of all we're going to explore a method called " Mapping to a higher dimension ". In this case we take our dataset and add an extra dimension into our space that we're dealing with and make our data a linearly separable with "some mapping".

[2] ***Kernel trick:*** The kernel trick allows us to make our data separable without having to deal with multiple or higher dimensions.

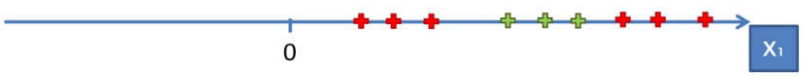
Lastly, we will talk about the different types of kernels that exist.

3.4.3 Mapping to a higher dimension

Use a Higher-Dimensional Space to make a separable data-set.

i. We can take our ***nonlinearly separable data-set*** map it to a higher dimension and get a ***linearly separable data-set***.
ii. Invoke the ***SVM algorithm*** build a ***decision boundary*** for a dataset and
iii. Then ***project*** all of ***that back*** into our ***original dimensions***.

❒ ***1D to 2D:*** First off we're going to look at a simplified example; we're going to look at a one dimensional dataset, so that we can kind of understand how it would work in multiple dimensions.



☞ Here we've got the X_1 dimension we've got some nine data points.

☞ Also, we can see our data is nonlinearity separable. In a single dimensional space a ***linear separator*** would ***not be a line*** or would be a ***dot***. [In a two dimensional space, a linear separator is a line; in a three dimensional space as a hyper-plane; but in a one dimensional space it's a single dot].

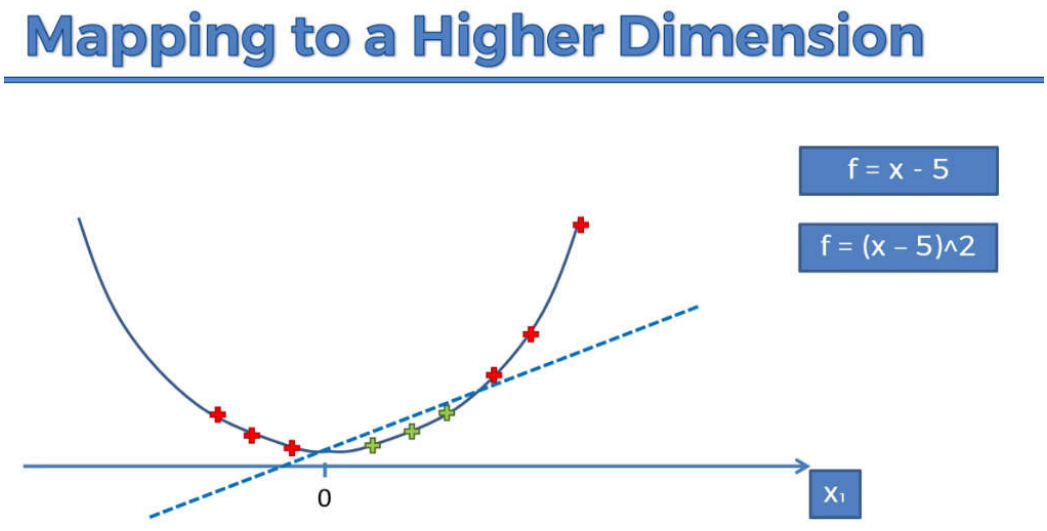
❒ We are going to create this mapping function on the fly. So let's say that the first green dot is after the point 5. So our first step to build the mapping function it can be any mapping functions that you can build. Lets use the following function:

$$f = x - 5$$
$$g = (x - 5)^2$$

☞ We use 2nd-degree function (because it is Convex-curve) and we shift it to point $x = 5$.

☞ If you take $\boldsymbol{x - 5}$ you would get left-side red dots will go into negative. Right-side ones will stay positive. The next step would be to square all to get the curve.

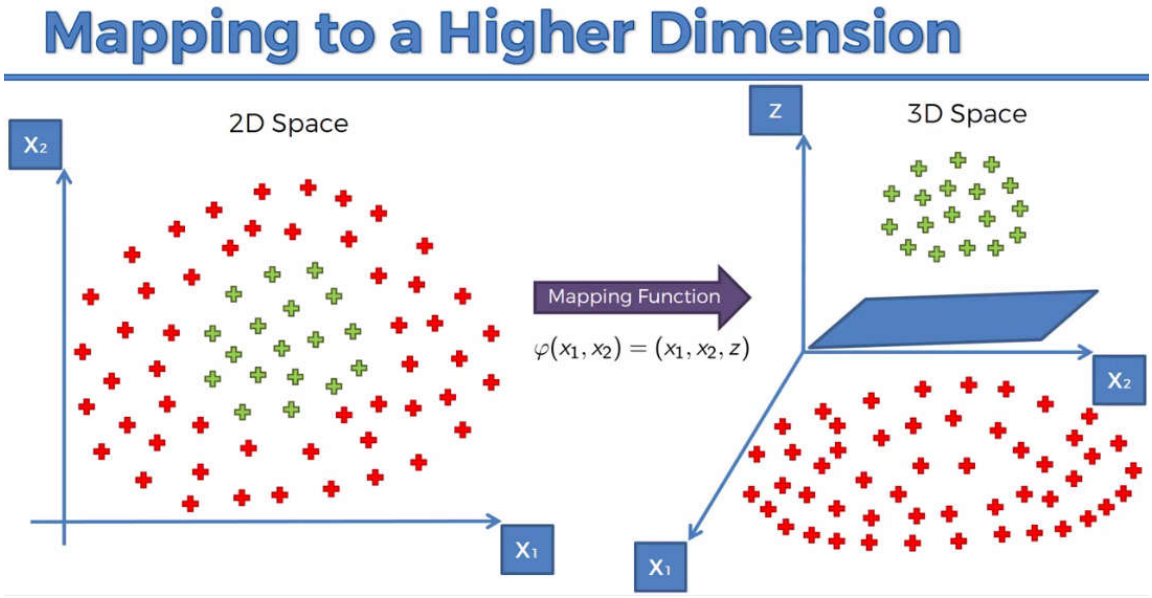
☞ After getting the curve we will project all the observation points on the curve. Then the points can be linearly seperable.



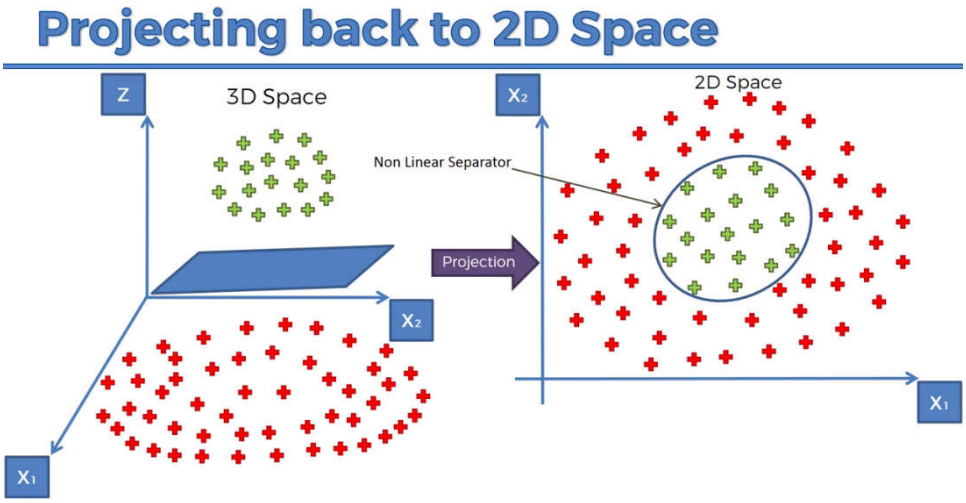
☝ We could use a 3rd degree equation or trigonometric function. This depends on how we want o separate the observation points.

❒ The same thing applies to two dimensional space, moving into three dimensional space. You'd map it into three dimensional space and then somehow it would become a linearly separable dataset.

☞ In this space the linear separator is no longer a line, it's a ***hyperplane***. So this ***hyperplane*** separates the two parts of our dataset in the way we want.

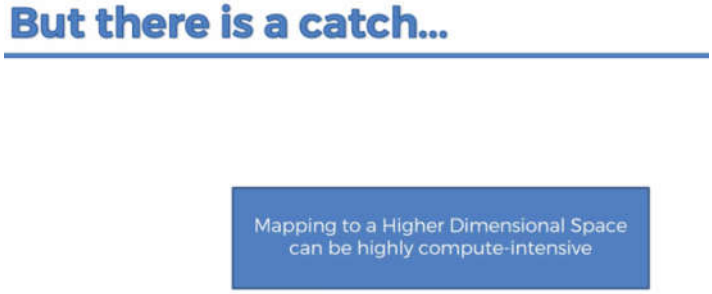


☞ Into this ***higher dimension***, we then apply the ***SVM*** algorithm to get the ***separator-hyperplane***. Once, we've got this result then we just projected back into our ***2D space*** and we've got this circle (which is a non-linear separator in 2D).



📣 With this algorithm, the problem is that mapping to a ***higher dimensional space*** can be ***highly compute intensive*** so it might require a ***lot of computation*** a lot of processing power.

The larger your dataset the more of a problem this can cause and therefore this approach isn't the best because you can imagine like you have a dataset and then mapping it to a higher dimension performing all the calculations there and then coming back to your lower dimension for a computer that can cause a lot of delays it can cause a lot of like processing backlog and issue



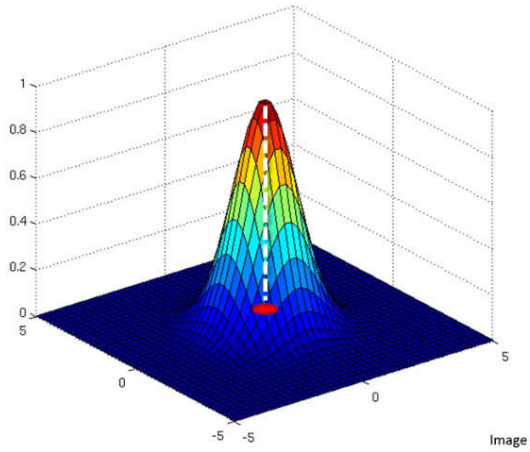
3.4.4 The Kernel Trick

Here we got the ***Gaussian/ Radial Basis Function*** (***Gaussian RBF kernel***) kernel

The Gaussian RBF Kernel

$$K(\vec{x}, \vec{l}^i) = e^{-\frac{\|\vec{x}-\vec{l}^i\|^2}{2\sigma^2}}$$

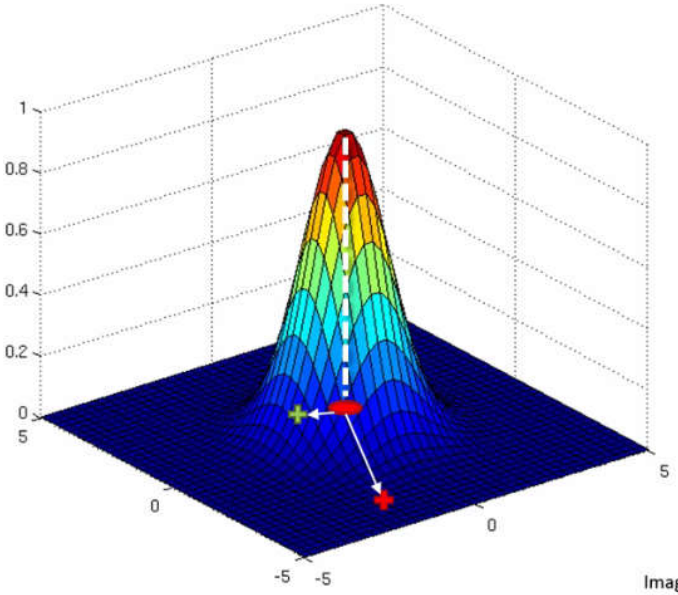
- So $\boldsymbol{K}$ stands for kernel and it's the function applied to two vectors $\vec{x}$, $\vec{l}^i$.
- There is a some sort of point in our dataset and called Landmark.
- $\vec{l}^i$ means there could be several landmark.
- The double vertical lines mean the distance between $\vec{x}$ and the landmark $\vec{l}^i$.
- Sigma is act like radius (variance).



$$K(\vec{x}, \vec{l}^i) = e^{-\frac{\|\vec{x}-\vec{l}^i\|^2}{2\sigma^2}}$$

Image source: http://www.cs.toronto.edu/~duvenaud/cookbook/index.html

☞ In the above figure, at the ***tip of this observation*** is right in the middle, when we project it to ***xy plane***, we get the position of the landmark. It is the point from which we're measuring the distance.



$$K(\vec{x}, \vec{l}^i) = e^{-\frac{\|\vec{x}-\vec{l}^i\|^2}{2\sigma^2}}$$

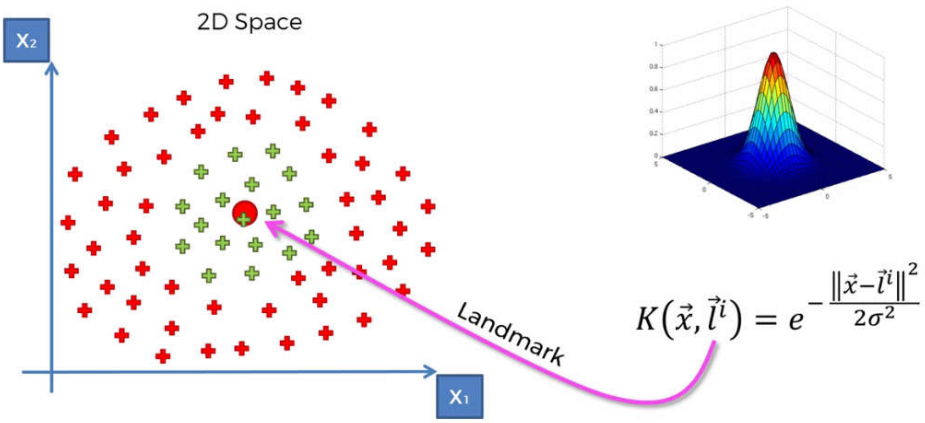
Image source: http://www.cs.toronto.edu/~duvenaud/cookbook/ind

☞ Let there are two points somewhere on our plane. Green one is near the landmark, red one is far from the landmark.

- ***Red*** one quite far away from the landmark. So ***distance*** is ***large***, hence $\frac{\|\vec{x}-\vec{l}^i\|^2}{2\sigma^2}$ is also large. Which makes $K(\vec{x}, \vec{l}^i) \to 0$.
- For the points which are closer to landmark (the green point), distance $\|\vec{x} - \vec{l}^i\|$ is small, hence $\frac{\|\vec{x}-\vec{l}^i\|^2}{2\sigma^2}$ is also small. Which makes $K(\vec{x}, \vec{l}^i) \to 1$. That's the whole methodology on how the machine learning algorithm to find an optimal placement for these landmarks.

☞ ***What's in 2D happening?*** We're going to use this ***kernel function*** to separate our data set to build that decision boundary. So let's have a look there is our two dimensional space.

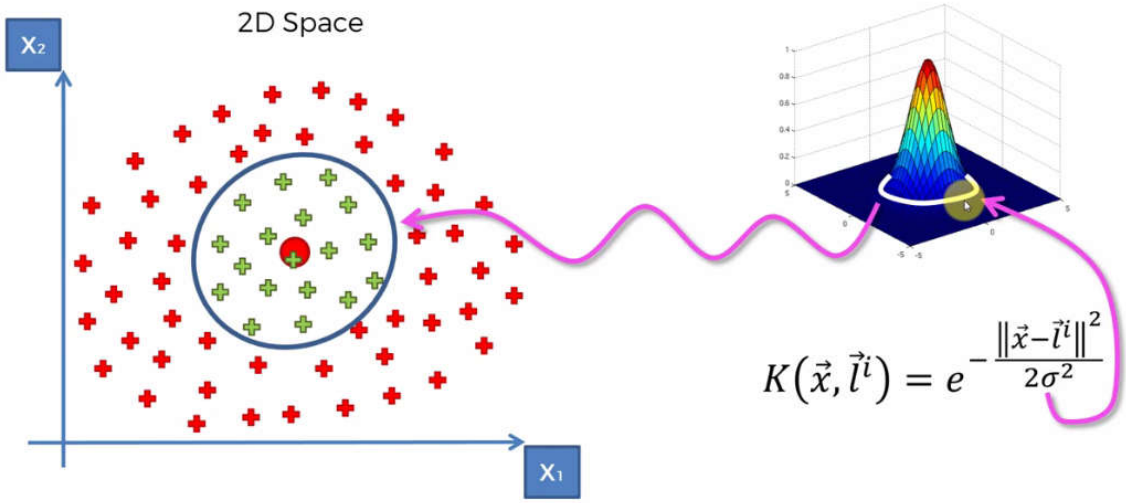
➢ Let's go back to our $x_1 x_2$ plane. Now we're going to take the ***Landmark*** and put it somewhere in our ***Dataset***.

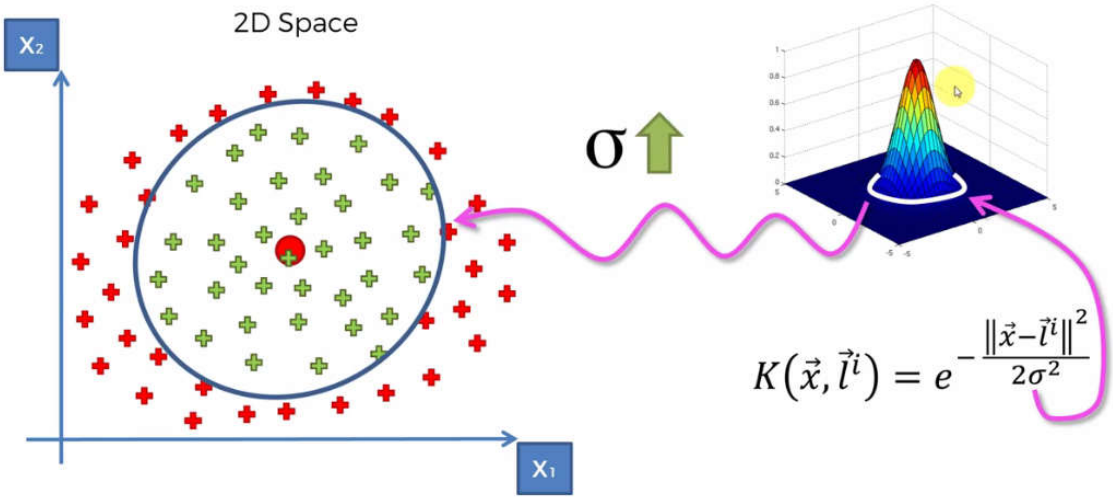


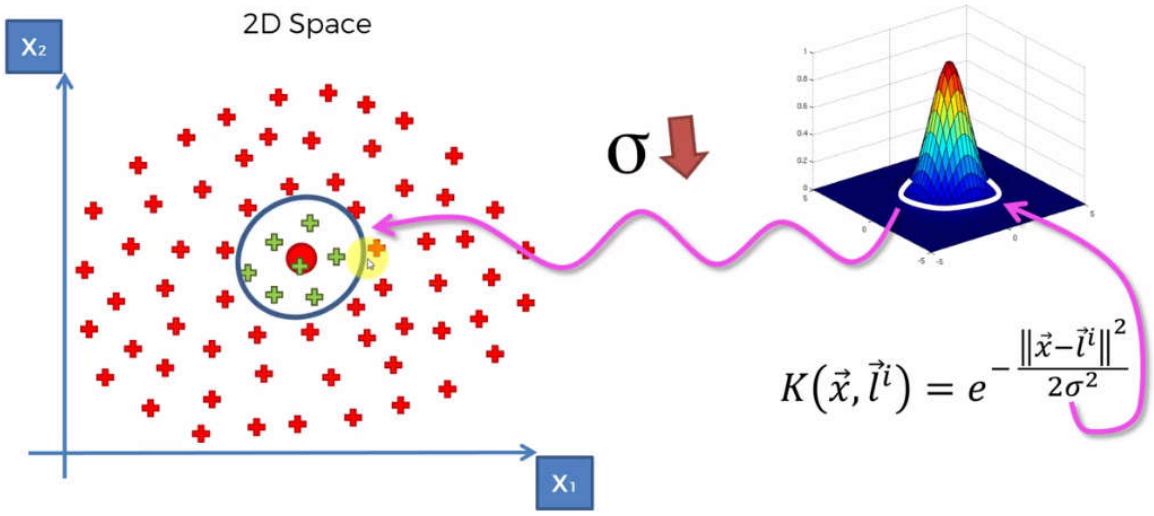
❒ After landmark is placed, the algorithm is set to find the optimal circumference/non-linear boundary using σ. ***When σ increases the boundary spreads, when σ decrease boundary shrinks.*** Kernel function is actually projected here onto our dataset.

☞ All of the points that are within that circumference and have them like assign them a value of above zero $(\mathbf{0}, \mathbf{1}]$ interval. Anything outside the circumference basically all these red points they'll get a value of $\mathbf{0}$. In 3D space, points that are within that circumference are lie on the landmark, but the points outside the circumference are very close to zero lies in the blue-colored plane.

☞ Hence, based on this function we can separate the two classes the green from the red just if we pick the right Sigma $\boldsymbol{\sigma}$.





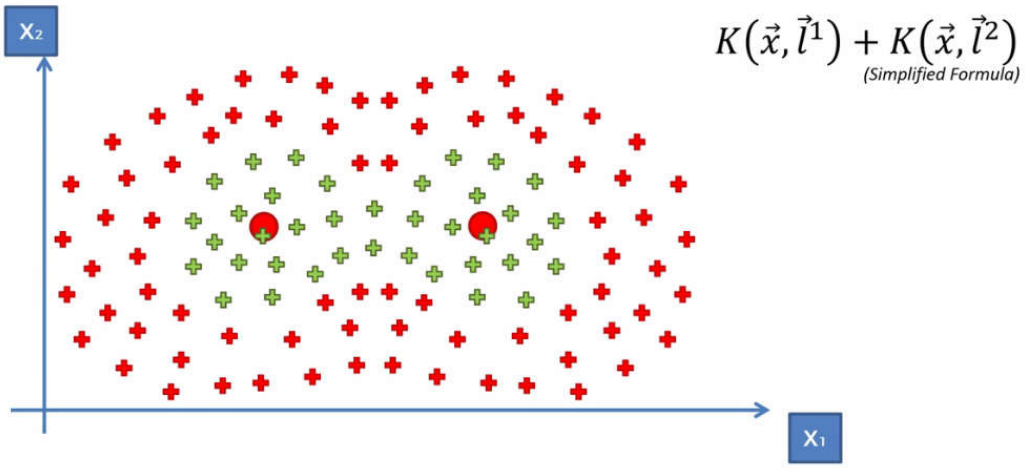


❒ So basically by ***finding the right Sigma σ*** you can set up the ***correct kernel function*** to assign ***0*** values to all of the points that you ***don't want*** in your classification and values ***above zero*** to the point that ***you do want*** in your classification. And that will allow you to ***separate*** the ***two classes***. Allow you to classify each from one. That in essence is a kernel trick

- ☞ Hence, we have created a ***decision boundary*** actually **going into a higher dimensional space without having to project** us back to 2D space.
- ☞ Here ***higher dimensional space is used*** but we're still ***doing the computations in the low dimensional space***.
- ☞ Yes we have this visual representation that involves a higher dimensional space but at the same time if you look at the part we were just calculating this formula in 2D.

3.4.5 The Kernel Trick with Multiple landmark

Say all of a sudden you can't adjust your decision boundary and it's non-linear and moreover you find yourself being able to solve much harder much more complex problems like this for example.

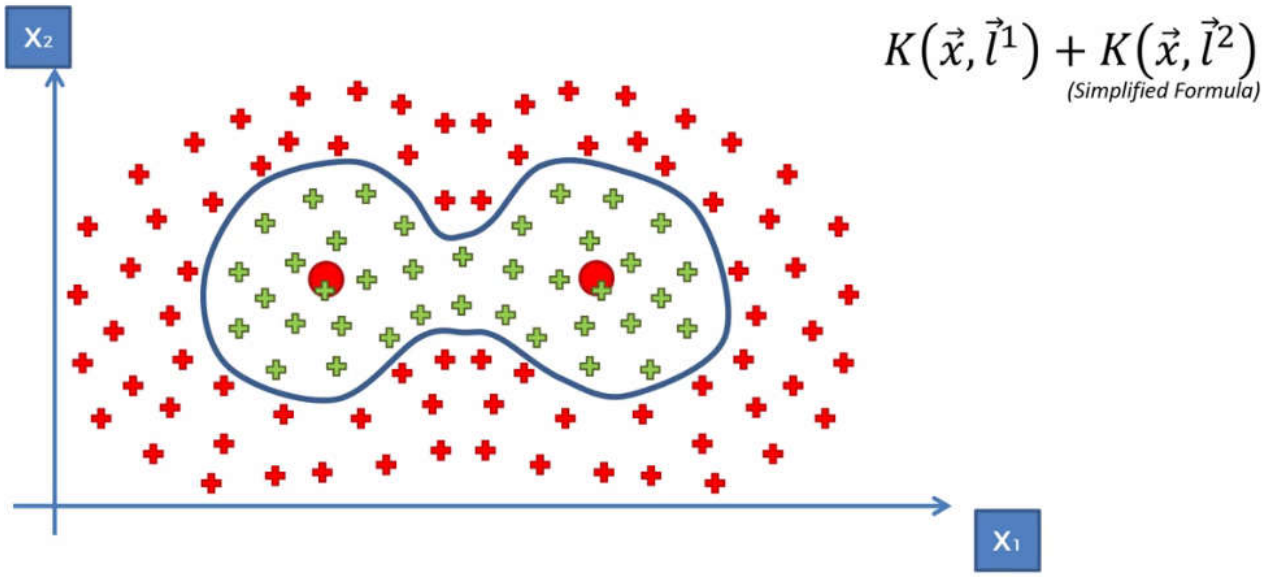


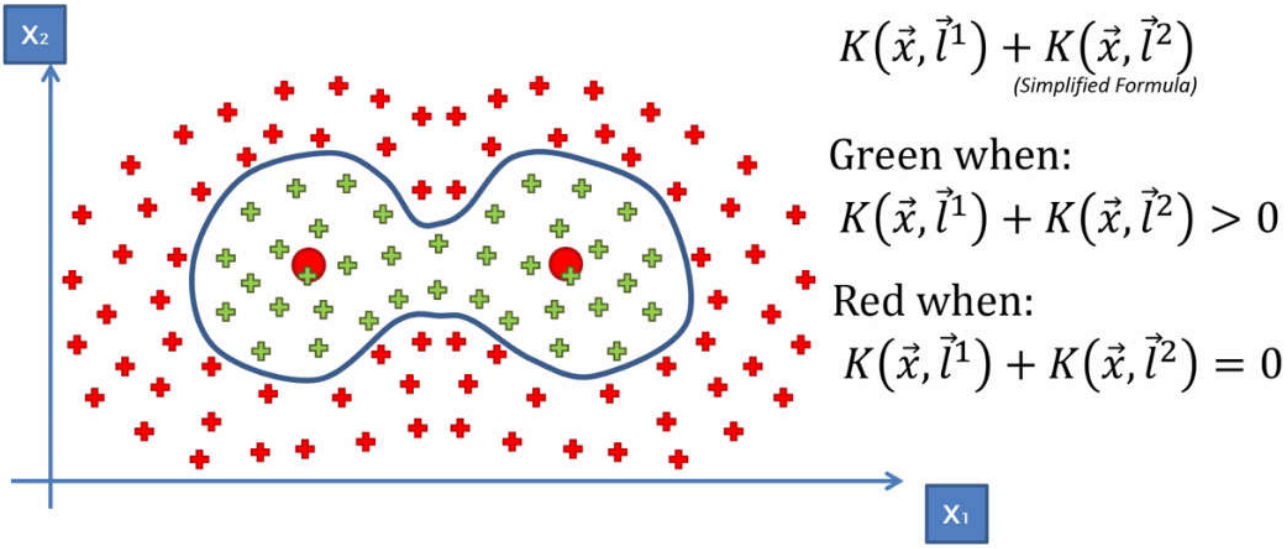
❒ So here is a very simplified formula:

$$K(\vec{x}, \vec{l}^1) + K(\vec{x}, \vec{l}^2)$$

- ☞ If you take ***two kernel function*** and you just ***add*** them up (in reality you need some coefficients). The points ***far*** from ***one landmark*** but ***near*** to ***another landmark*** then it has non-zero value. The points which are far from ***both landmark*** will get a 0 value. That's it.

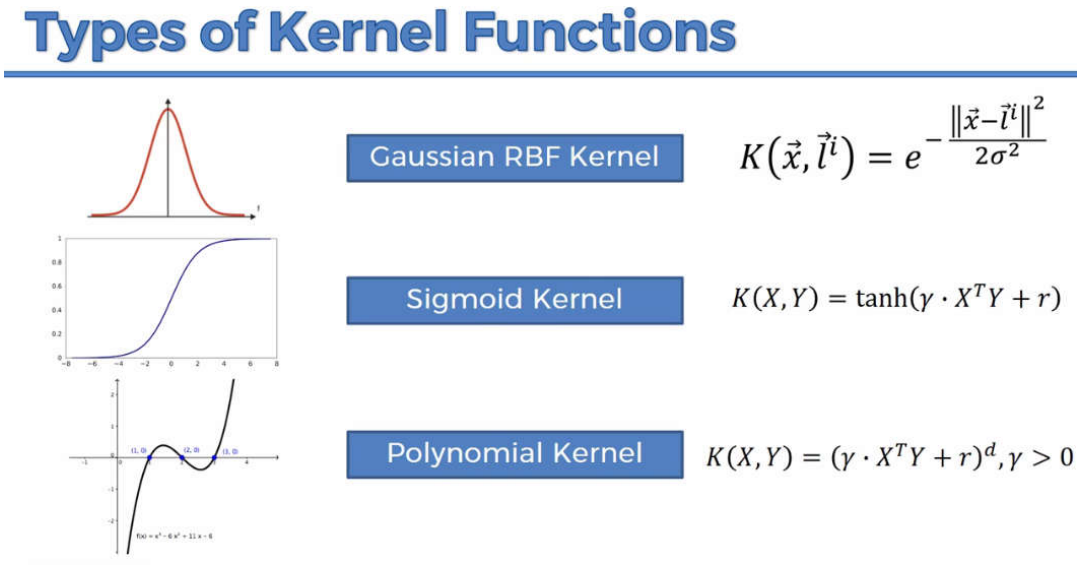
- ☞ And the formula here would be the *point is assigned to the* ***green class***, when this ***equation is greater than zero*** and the *point is assigned to* ***read class*** when this ***equation is equal to zero***.

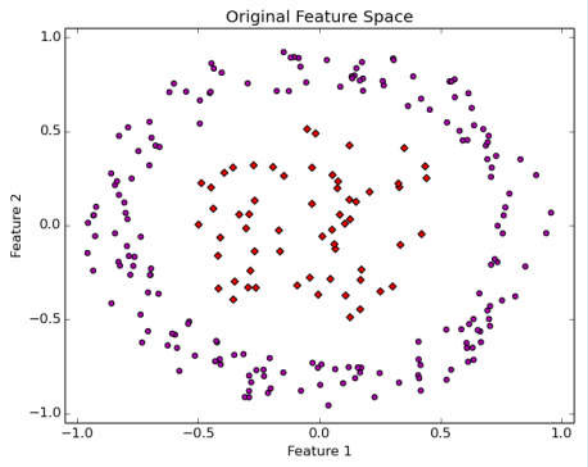


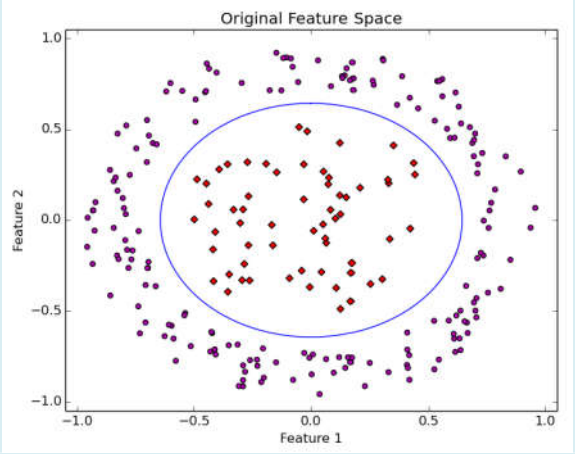


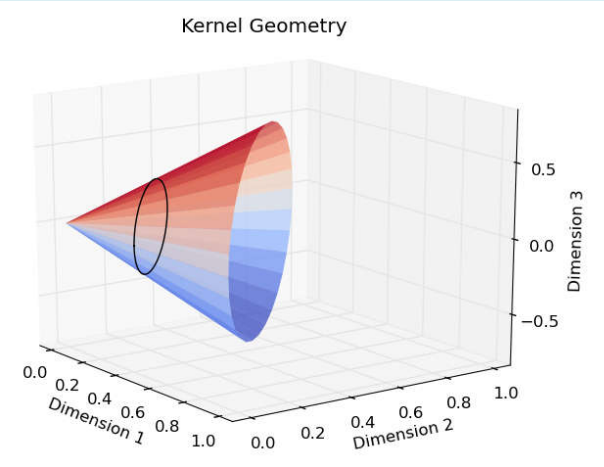
3.4.6 Different types of Kernel Functions

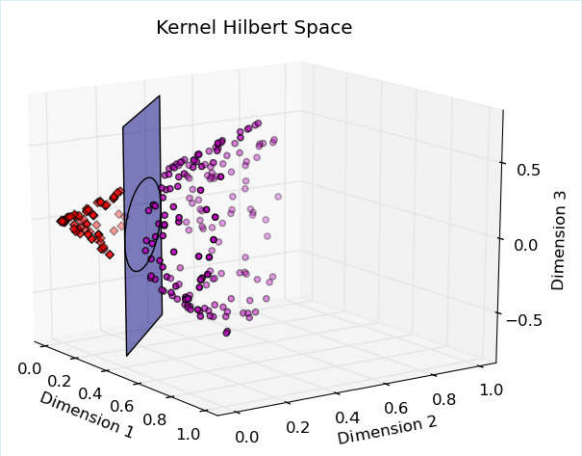
you need to know about the kernel ***SVM*** is that the ***radial basis function*** which also called the ***Gaussian function*** is *not the only Kernel function* that is used in this method. So let's have a look at a couple.



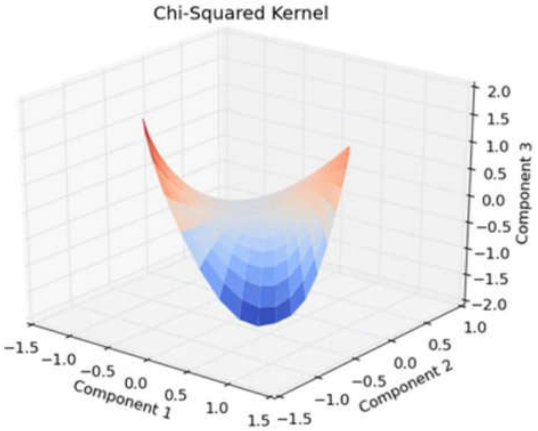


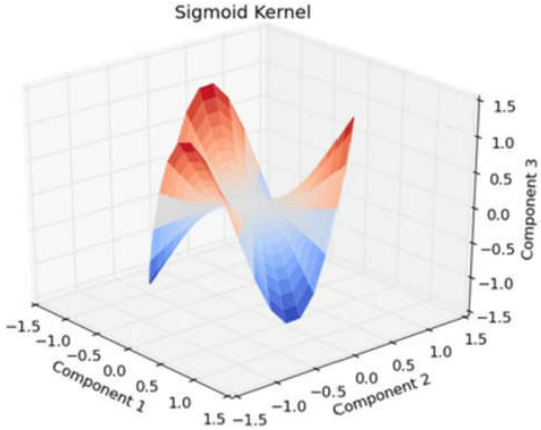


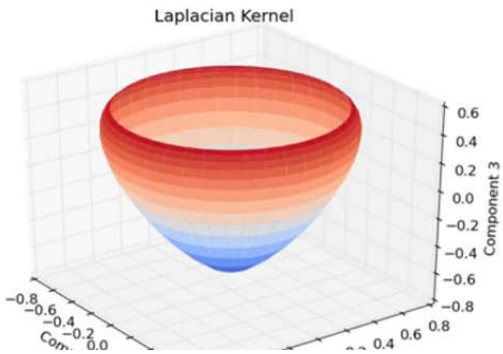


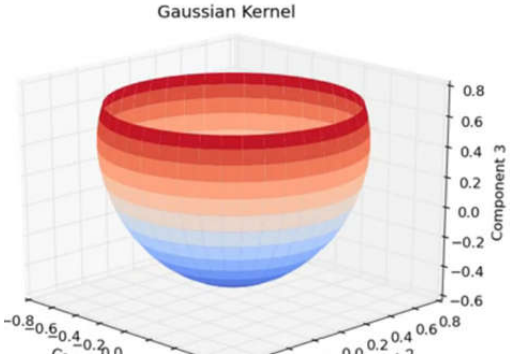


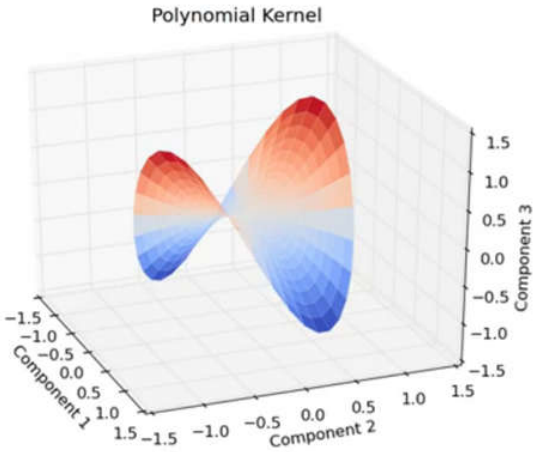
Some more kernel fuunctions

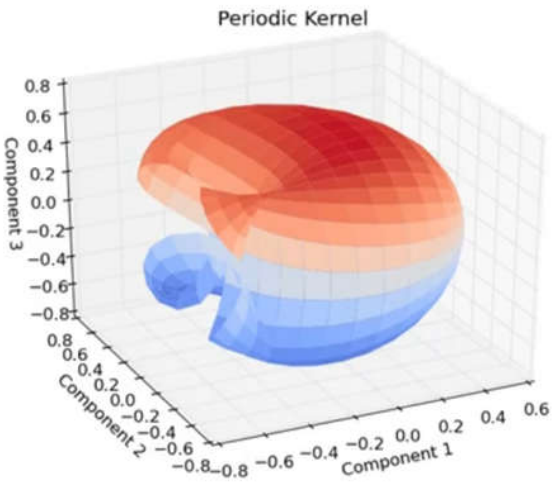












3.4.7 Python implementation of the kernel- SVM

All code remain same as previous SVM-Algorithm. We just need to change the "Kernel": `kernel="rbf"`

```
# Kernel="rbf" instead of kernel="Linear"
clsFier = SVC(kernel="rbf", random_state=0)
```

Practiced version

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values


        # Feature-Scaling after Data Split

# Data Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler

st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)


# Fit train set to Kernel-SVM classifier
from sklearn.svm import SVC
# Since data-points are non-seperable linearly, use "rbf" : Gaussian kernel, gives better result.
# Kernel="rbf" instead of kernel="linear"
clsFier = SVC(kernel="rbf", random_state=0)
clsFier.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = clsFier.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value


# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
```

```
            alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Kernel-SVM (Training set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

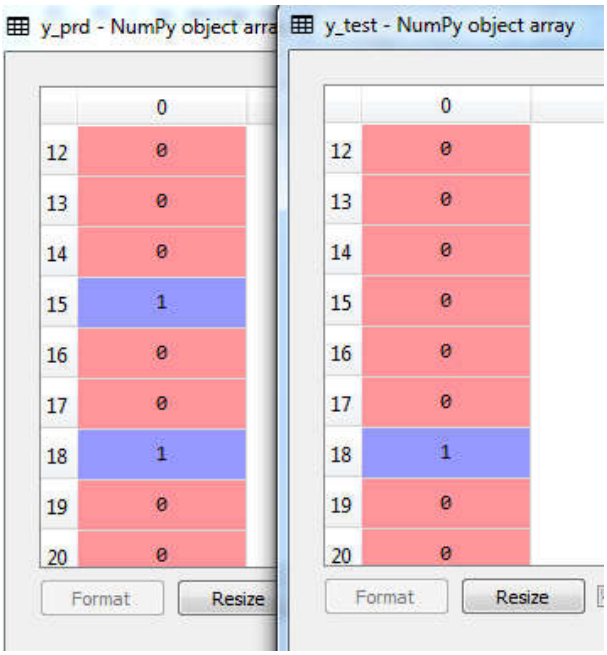
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Kernel-SVM (Test set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# python prctc__kernel_SVM.py
```

☞ ***Confusion matrix:***

	0	1
0	64	4
1	3	29

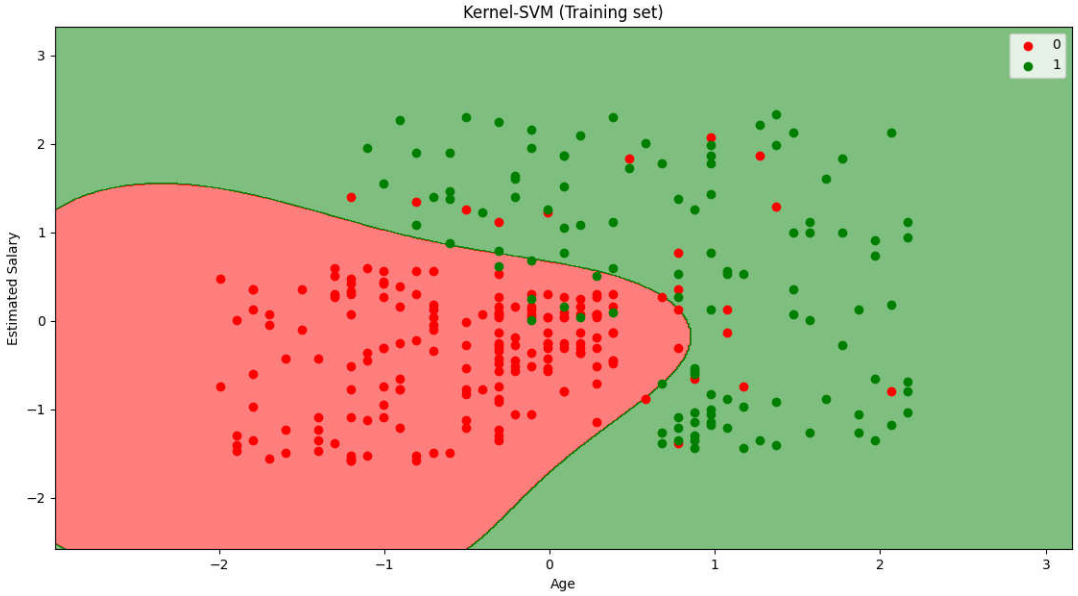
☞ ***Prediction:***



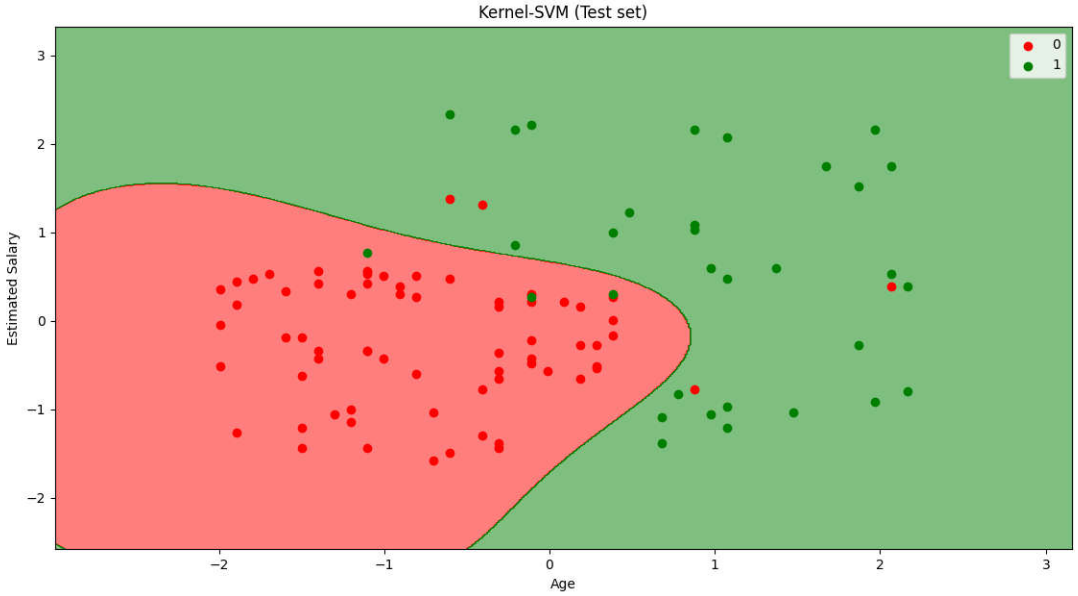
	0
12	0
13	0
14	0
15	1
16	0
17	0
18	1
19	0
20	0

	0
12	0
13	0
14	0
15	0
16	0
17	0
18	1
19	0
20	0

Train data Visualization



Test data Visualization



Chapter 3 : part 5

Naive Bayes

Bayes Theorem, Naive Bayes Algorithm

3.5.1 Bayes Theorem

❒ ***Bayes' Theorem:*** Bayes' theorem is also known as Bayes' Rule or Bayes' law, which is used to determine the ***probability of a hypothesis*** with ***prior knowledge***. It depends on the ***conditional probability***. The formula for Bayes' theorem is given as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

☝ Probability that ***A*** occurs given ***B*** has occurred ***P(A|B)***. The Bayes Theorem is a method of finding what the ***probability is of something occurring*** (A) given that something ***else has just occurred*** (B).

- ***P(A|B) is Posterior probability:*** Probability of hypothesis **A** on the observed event **B**. How often **A** happens given that **B** happens
- ***P(B|A) is Likelihood probability:*** Probability of the evidence given that the probability of a hypothesis is true. How often ***B*** happens given that ***A*** happens
- ***P(A) is Prior Probability:*** Probability of hypothesis before observing the evidence. ***A*** is on its own.
- ***P(B) is Marginal Probability:*** Probability of Evidence. ***B*** is on its own.

- where ***A*** and ***B*** are events and $P(B) \neq 0$. ***A*** and ***B*** must be ***different events***.
- $P(A \mid B)$ ***is a conditional probability:*** The probability of ***event A*** occurring given that ***B*** is ***true***. It is also called the ***posterior probability*** of ***A*** given ***B***.
- $P(B \mid A)$ is also a conditional probability: the probability of event ***B*** occurring given that ***A*** is true. It can also be interpreted as the likelihood of ***A*** given a fixed ***B*** because $P(B \mid A) = L(A \mid B)$.
- $P(A)$ and $P(B)$ are the probabilities of observing ***A*** and ***B*** respectively without any given conditions; they are known as the marginal probability or prior probability.

Example 1: Let us say ***P(Fire)*** means how often there is fire, and P(Smoke) means how often we see smoke, then:

P(Fire|Smoke) means how often there is ***fire*** when we can see ***smoke***
P(Smoke|Fire) means how often we can see ***smoke*** when there is ***fire***

So the formula kind of tells us ***"forwards"*** **P(Fire|Smoke)** when we know ***"backwards"*** **P(Smoke|Fire)**

- dangerous fires are rare (1%), ***P(Fire)***
- but smoke is fairly common (10%) due to barbecues, ***P(Smoke)***
- and 90% of dangerous fires make smoke, ***P(Smoke|Fire)***

We can then discover the probability of dangerous Fire when there is Smoke:

$$P(Fire|Smoke) = \frac{P(Fire)P(Smoke|Fire)}{P(Smoke)}$$
$$= \frac{1\% \; x \; 90\%}{10\%}$$
$$= 9\%$$

So it is still worth checking out any smoke to be sure.

Wrench making machine example: Two machines produce wrenches. We know that:

- ***Machine 1:*** Produces 30 Wrenches Per Hour
- ***Machine 2:*** Produces 20 Wrenches Per Hour
- ***1%*** of all wrenches are defective, where ***50%*** of the wrenches that are defective are from ***machine 1*** and ***50%*** from ***machine 2***.

Question: What is the ***probability*** that a ***part*** produced by ***machine 2*** is ***defective***?

- Total wrench per hour is 30+20 = 50
- **P(M1)** = probability of being from ***machine 1*** M_1 = 30/50 = 0.6
- **P(M2)** = probability of being from ***machine 2*** M_2 = 20/50 = 0.4
- **P(D)** = probability of being defective = 0.01
- **P(M1 | D)** = Probability of being from ***machine 1*** M_1 from **defected wrenches** = 0.5, (probability of being M_1 from a pile of only defective wrenches).
- **P(M2 | D)** = Probability of being from ***machine 2*** M_2 from **defected wrenches** = 0.5, (probability of being M_2 from a pile of only defective wrenches).
- **P(D | M2)** = ***probability*** that a ***Wrench*** produced by ***machine 2*** is ***defective***??? i.e. given that the wrench is from M_2, then we need to find the probability of being it defective. (from pile of wrenches only come from machine 2)

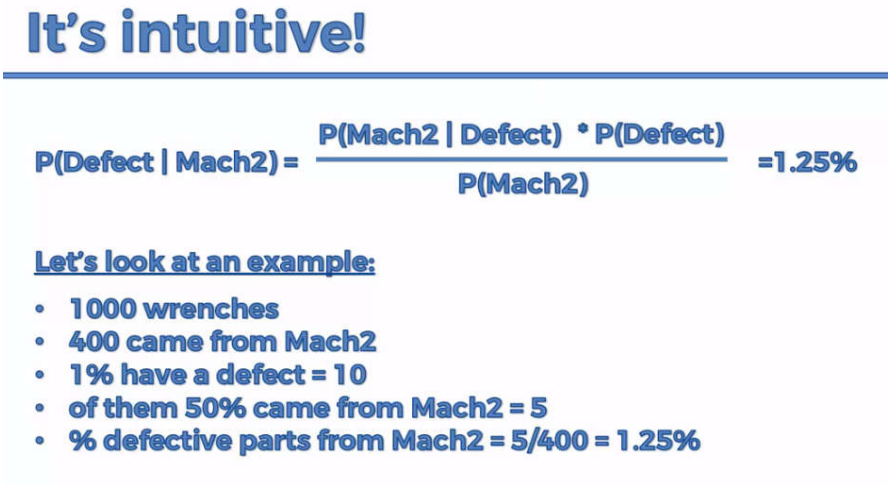
From ***Bayes Theorem***,

$$P(D|M_2) = \frac{P(M_2|D)P(D)}{P(M_2)}$$

$$= \frac{Probability\ of\ being\ from\ machine\ 2\ \ M_2 from\ defected\ wrenches \times \text{probability of being defective}}{probability\ of\ being\ from\ machine\ 2\ M_2}$$

$$= \frac{0.5 \times 0.01}{0.4} = 0.0125 = \frac{1}{80}$$

Its intuitive:



$$\textbf{probability that a Wrench produced by machine 2 is defective} = \frac{\textbf{Number of defective wrenches comes from machine 2}}{\textbf{Number of wrenches comes from machine 2}}$$

$\boldsymbol{P(M_2|D)P(D)}$= Number of defective wrenches comes from machine 2

$\boldsymbol{P(M_2)}$= Number of wrenches comes from machine 2

| : means given in mathematical terms.

Similarly, ***probability*** that a ***Wrench*** produced by ***machine 1*** is ***defective*** $= P(D|M_1) = \frac{P(M_1|D)P(D)}{P(M_1)}$

$$= \frac{0.5 \times 0.01}{0.6} = \frac{1}{120}$$

3.5.2 Naïve Bayes Classifier Algorithm

❒ ***Naïve Bayes*** algorithm is a ***supervised*** learning algorithm, which is based on ***Bayes theorem*** and used for solving ***classification*** problems (used to classify data with previous known classes).

- ☞ It is ***mainly used*** in ***text classification*** that includes a ***high-dimensional training dataset***.
- ☞ ***Naïve Bayes Classifier*** is one of the simple and most effective ***Classification algorithms*** which helps in building the ***fast machine learning models*** that can make ***quick predictions***.
- ☞ It is a ***probabilistic classifier***, which means it ***predicts*** on the ***basis of the probability*** of an ***object***.
- ☞ Some popular examples of ***Naïve Bayes*** Algorithm are ***spam*** filtration, ***Sentimental*** analysis, and classifying ***articles***.

❒ ***Why is it called Naïve Bayes?***

The ***Naïve Bayes algorithm*** is comprised of two words ***Naïve*** and ***Bayes***, Which can be described as:

- ☞ ***Naïve:*** It is called ***Naïve*** because it assumes that the ***occurrence*** of a ***certain feature*** is ***independent of the occurrence of other features***. Such as if the ***fruit*** is identified on the bases of ***color***, ***shape***, and ***taste***, then ***red***, ***spherical***, and ***sweet*** fruit is recognized as an ***Apple***.
- ☞ ***Bayes:*** It is called ***Bayes*** because it depends on the principle of ***Bayes' Theorem***.

❒ ***Advantages of Naïve Bayes Classifier:***

- ➲ ***Naïve Bayes*** is one of the fast and easy ML algorithms to predict a class of datasets.
- ➲ It can be used for ***Binary*** as well as ***Multi-class*** Classifications.
- ➲ It performs well in ***Multi-class*** predictions as compared to the other ***Algorithms***.
- ➲ It is the most popular choice for text classification problems.

❒ ***Disadvantages of Naïve Bayes Classifier:***

- ➲ ***Naive Bayes*** assumes that all ***features*** are ***independent*** or ***unrelated***, so it *cannot learn the relationship* between ***features***.

❒ Applications of Naïve Bayes Classifier:

- ➲ It is used for *Credit Scoring*.
- ➲ It is used in *medical data classification*.
- ➲ It can be used in *real-time predictions* because *Naïve Bayes Classifier* is an *eager learner*.
- ➲ It is used in *Text classification* such as *Spam filtering* and *Sentiment* analysis.

❒ ***Types of Naïve Bayes Model:*** There are three types of ***Naive Bayes*** Model, which are given below:

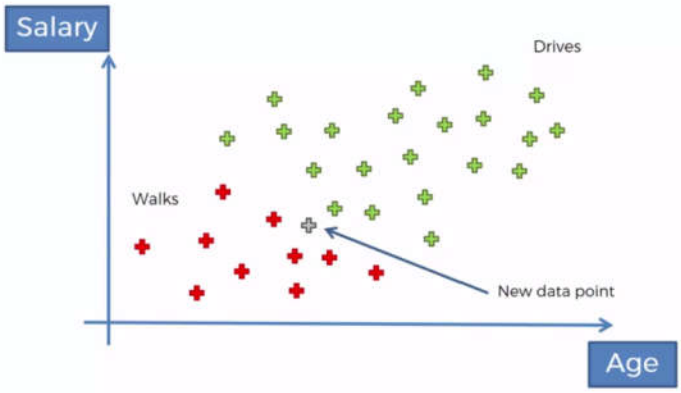
- ▶ ***Gaussian:*** The **Gaussian** model assumes that features follow a **normal distribution**. This means if *predictors take continuous values* instead of *discrete*, then the model assumes that these values are *sampled* from the *Gaussian distribution*.
- ▶ ***Multinomial:*** The Multinomial Naïve Bayes classifier is used when the *data* is *multinomial distributed*. It is primarily used for *document classification* problems, it means a particular *document* belongs to which *category* such as ***Sports***, ***Politics***, *education*, etc. The classifier uses the *frequency of words* for the predictors.
- ▶ ***Bernoulli:*** The *Bernoulli* classifier works similar to the *Multinomial classifier*, but the predictor variables are the *independent Booleans variables*. Such as if a particular word is present or not in a document. This model is also famous for *document classification* tasks.

How it Works:

i. Apply Bayes Theorem in this example we'll use it to find the **probability that a person walks based on his features** (the specific ***age*** and ***salary*** of that data point). ***Calculate*** the ***probability*** of each of the components of ***Bayes Theorem***.

ii. Apply Theorem again, in this case it would be to find the **probability that the new dataset Drives based on its features** (age and salary).

iii. Compare the two and assign a class to the dataset.



☯ Here we've got a data set. It has two features x1 (salary) and x2 (age). And there are ***two categories***: ***Category 1*** which is *RED (Walks to work)* ***Category 2*** which is *GREEN (Drives to work)*.

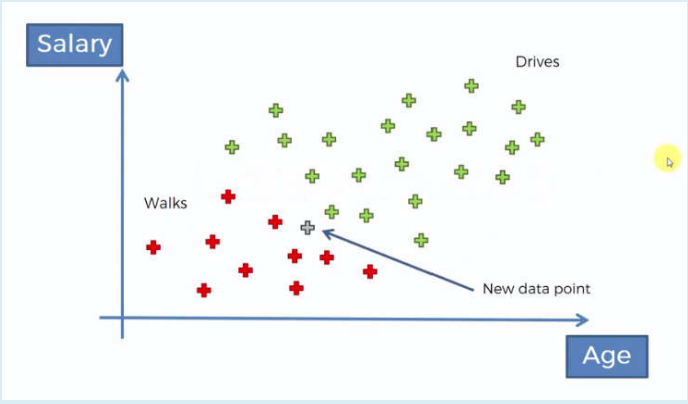
☯ The ***ML problem*** is to ***classify a new data-point***. i.e. what happens if we add a ***new observation*** a ***new data point*** into the set.

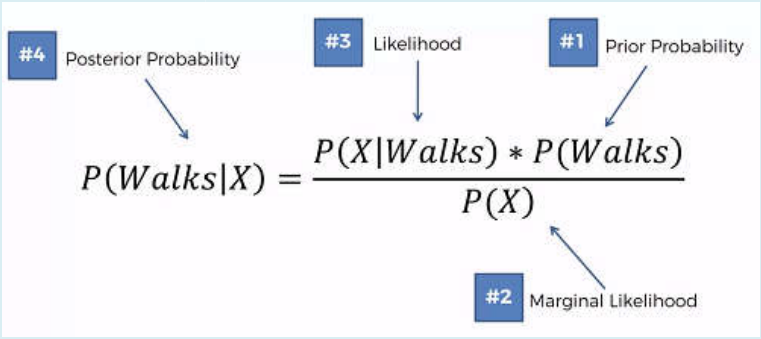
- ☞ So, this is a supervised machine learning algorithm because we're classifying something based on previously known Classes.
- ☞ Here ***Naïve Bayes Algorithm*** is going to help us solve this challenge.

🛠 We're going to take the ***Bayes Theorem*** and we can ***apply it twice***.

[1] ***Step 1:*** First, we're going to apply it to find out what is the probability that a *person(new data-point)* ***walks*** *given that his* ***features*** *X* (i.e. age & salary as features).

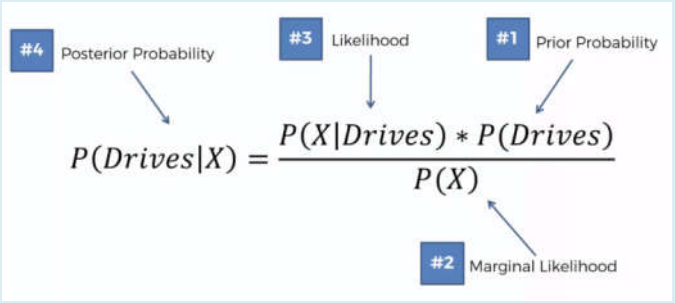
$$P(Walks \mid X) = \frac{P(X \mid Walks)P(Walks)}{P(X)}$$





[2] ***Step 2:*** We're going to calculate the probability that somebody ***drives*** given those same ***features X*** that we see in our ***new data-point***. [Probability that a *person(new data-point)* ***drives*** *given that his* ***features*** *X* (i.e. age & salary as features)]

$$P(Drives \mid X) = \frac{P(X \mid Drives)P(Drives)}{P(X)}$$



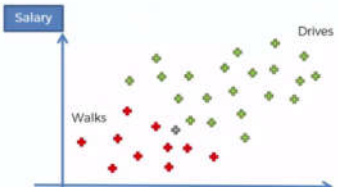
[3] ***Step 3:*** And finally we're going to compare the **probability that somebody WALKS given features X** versus the **probability that somebody DRIVES given features X.** Then from that comparison we'll decide which Class to put that new data point in.

$$P(Walks|X)\ v.s.\ P(Drives|X)$$

☞ You can see that the ***Naïve Bayes classifier*** is a **probabilistic type of classifier** because we're first calculating the probabilities and then based on probabilities we're assigning it to a Class.

❒ ***Step 1:*** $P(Walks \mid X) = \frac{P(X \mid Walks)P(Walks)}{P(X)}$ ***We divide each calculation into 4 steps:***

A. ***Calculate The Prior Probability:*** We're going to calculate the probability that somebody walks. i.e. we're going to add a new observation to our data-set but we don't know their age and salary (without knowing the features).



#1. P(Walks)

$$P(Walks) = \frac{Number\ of\ Walkers}{Total\ Observations}$$

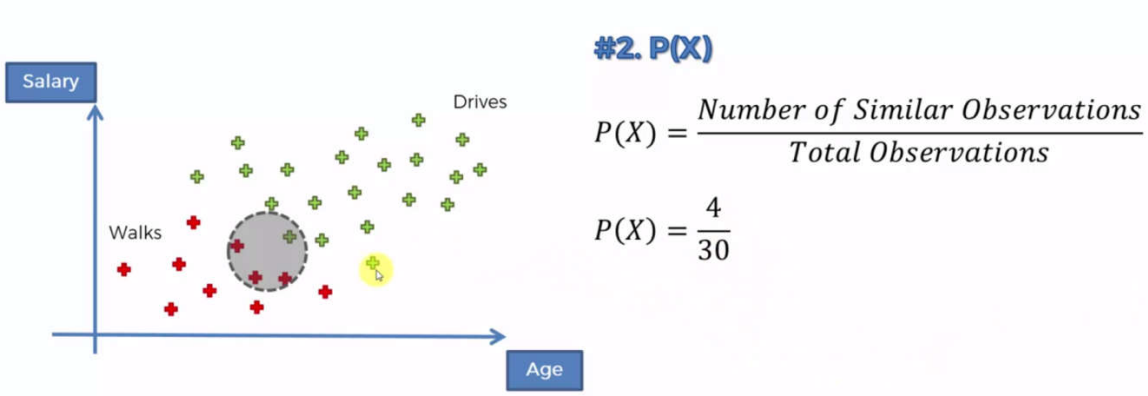
$$P(Walks) = \frac{10}{30}$$

☞ What is the probability that this person that we're

☞ We calculate the ***number*** of ***read observations*** (which is 10) number of people that actually walk and we divide it by the ***number of total observations*** (which is 30).

☞ Hence ***Prior Probability*** is: $P(Walks) = \frac{number\ of\ people\ that\ actually\ walk}{number\ of\ total\ observations} = \frac{10}{30}$

B. ***Calculate the Marginal Likelihood:*** Select a *radius around the data point*. $\boldsymbol{P(X)}$ is the ***probability*** of any ***given point*** to **fall within** that **selected radius**.



➲ We're going to select a radius and we're going to draw a circle around our observation. Now this radius you need to select on your own and you need to decide for you.

➲ We're going to remove our new data-point DOT for now just so that it's not confusing us.

➲ And then we're going to look at all the points that are ***inside*** this ***circle*** and we're going to consider them to be ***similar in terms of features*** to the new data-point that we had (age 25 and salary of $30000 per year).

➲ Let's say anybody between the ***age*** of ***20 to 30*** and in the ***salaries*** of ***$25000 to $35000*** are the ***features*** for the ***circled area***. Anybody that falls in that circle is going to be deemed similar to the new data point that we're adding to our data set.

☝ ***Marginal Likelihood:*** The probability of X, $P(X)$ is the ***probability of a new point that we add to our data set being similar in features to the point that we actually are adding to it***. So basically it's a ***probability*** of that ***new point*** that we're adding (or like any random point) that ***fall into this circle***.

$P(X)$ it tells us what is the **likelihood(possibility)** of any **new random variable** that we add to this data set ***Falling inside the CIRCLE***. And $P(X)$ is calculated as:

$$P(X) = \frac{Number\ of\ Similar\ Observations}{Total\ Observations}$$

➲ We have ***3 red*** dots ***1 green*** dot in the circle. i.e. $P(X) = \frac{4}{30}$

C. ***Likelihood:*** Same radius. $\boldsymbol{P(X \mid Walks)}$: **Probability of the data point would be in this circle *given*** that that **data-point Walks**.

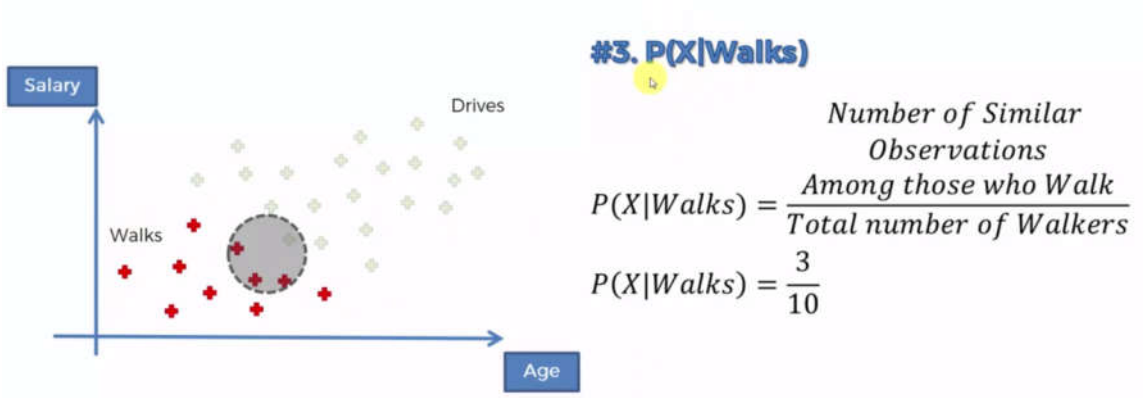
➲ What is the **likelihood** that **somebody who walks** exhibits features **X**?

➲ We're going to draw the ***same circle*** again and once again anything that ***falls inside*** the ***circle*** is deemed to be similar to the point that we're adding.

➲ So, what is the ***likelihood*** that a ***randomly selected data point*** will be fall in this circle given that a person ***Walks***.

☞ Other way to think about this is we're only working with people who walk to work. So we're only working with the red dots which represent people who walk to work. So let's forget about the green dots.

☞ So the question becomes: ***"Given that we're only working with the red dots, then what is the likelihood that a randomly selected data-point from the red dots is (exhibits features similar to the point that we are adding to our Dataset) falling into this circle"***.



D. ***Calculate Posterior Probability:***

$$\text{Posterior Probability, } P(Walks \mid X) = \frac{P(Walks) \times P(X \mid Walks)}{P(X)} = \frac{Prior\ Probability \times\ Likelihood}{Marginal\ Likelihood}$$

$$= \frac{\frac{10}{30} \times \frac{3}{10}}{\frac{4}{30}} = \frac{3}{4}$$

As summery:

[1] ***Prior Probability:*** $P(Walks)$

[2] ***Calculate the Marginal Likelihood:*** Select a radius around the data point. P(X) is the probabilty of any given point to fall within that selected radius.

[3] ***Likelihood:*** Same radius. $\boldsymbol{P(X \mid Walks)}$: Probability of the data point would be in this circle given that that datapoint walks.

[4] ***Calculate Posterior Probability:***

$$\text{Posterior Probability, } P(Walks \mid X) = \frac{P(Walks) \times P(X \mid Walks)}{P(X)} = \frac{Prior\ Probability \times\ Likelihood}{Marginal\ Likelihood}$$

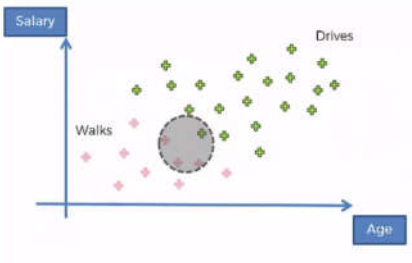
$$= \frac{\frac{10}{30} \times \frac{3}{10}}{\frac{4}{30}} = \frac{3}{4}$$

❒ ***Step 2:*** $P(Drives \mid X) = \frac{P(X \mid Drives)P(Drives)}{P(X)}$

We consider the new data-point and same circle. Then:

A. ***Prior Probability:*** $P(Drives)$

$$P(Drives) = \frac{no.of\ Drivers}{Total\ Observations} = \frac{20}{30}$$



B. ***Calculate the Marginal Likelihood:*** $P(X)$

$$P(X) = \frac{Number\ of\ Similar\ Observations}{Total\ Observations} = \frac{4}{30}$$

C. ***Likelihood:*** $P(X \mid Drives)$: Considering only the ***green points***

$$P(X \mid Walks) = \frac{Number\ of\ similar\ observations\ among\ those\ who\ Drives}{Total\ number\ of\ Drivers} = \frac{1}{20}$$

D. ***Calculate Posterior Probability:***

$$\text{Posterior Probability, } P(Drives \mid X) = \frac{P(Drives) \times P(X \mid Drives)}{P(X)} = \frac{Prior\ Probability \times\ Likelihood}{Marginal\ Likelihood}$$

$$\frac{20}{\quad} \times \frac{1}{\quad}$$

❒ ***Step 3:*** We're going to compare the **probability that somebody WALKS given features X** versus the **probability that somebody DRIVES given features X.**

$P(Walks \mid X) = \frac{P(X \mid Walks)P(Walks)}{P(X)}$	$P(Drives \mid X) = \frac{P(X \mid Drives)P(Drives)}{P(X)}$
#4 Posterior Probability, #3 Likelihood, #1 Prior Probability: $P(Walks\|X) = \frac{\frac{3}{10} * \frac{10}{30}}{\frac{4}{30}} = 0.75$ (#2 Marginal Likelihood)	#4 Posterior Probability, #3 Likelihood, #1 Prior Probability: $P(Drives\|X) = \frac{\frac{1}{20} * \frac{20}{30}}{\frac{4}{30}} = 0.25$ (#2 Marginal Likelihood)
$P(Walks\|X)\ v.s.\ P(Drives\|X)$	
$0.75 > 0.25$	

☞ Since in this case the ***probability*** of the data-point ***walking*** is greater than ***driving*** we can say that the data point is assigned to walking.

📣 Therefore it is more likely that that person with given features ***X*** is going to be a person who ***walks to work*** than the person who drives to work.

That is how the ***Naïve Bayes Algorithm*** in ML works.

☝ ***Why Naïve?:*** Because we assume that the features (age and Salary are independent).

☝ ***We can actually drop P(X).*** But it is good to follow full calculation. Some times the shortcut is used. It is ok when we are comparing but when we calculate the Posterior Probability values, we need to follow the full calculations.

$$\frac{P(X|Walks) * P(Walks)}{\cancel{P(X)}} \quad v.s. \quad \frac{P(X|Drives) * P(Drives)}{\cancel{P(X)}}$$

☝ ***More than 2 classes:*** We need to compare all possible pair of cases.

3.5.3 Python Implementation of the Naïve Bayes algorithm:

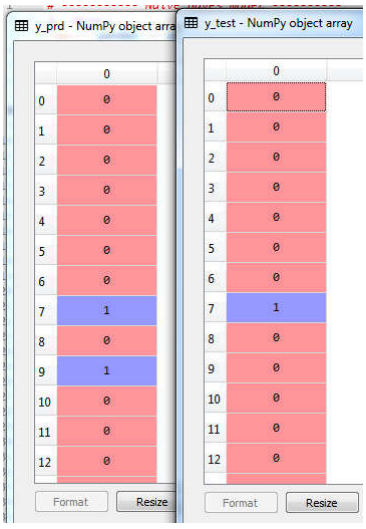
❒ The template is same as before, We only change the algorithm name, and Implement the classifier:

☞ We don't need any parameters for our classifier.

```
from sklearn.naive_bayes import GaussianNB
clsFier = GaussianNB()
clsFier.fit(X_train, y_train) # fit the dataset
```

☞ ***Confusion matrix and prediction:***

	0	1
0	65	3
1	7	25



Practiced version

```
# ---------- Naive Bayes model ----------
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values


        # Feature-Scaling after Data Split

# Data Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler

st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)


# Fit train set to Naïve Bayes classifier: No parmeter is needed
from sklearn.naive_bayes import GaussianNB
clsFier = GaussianNB()
clsFier.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = clsFier.predict(X_test)

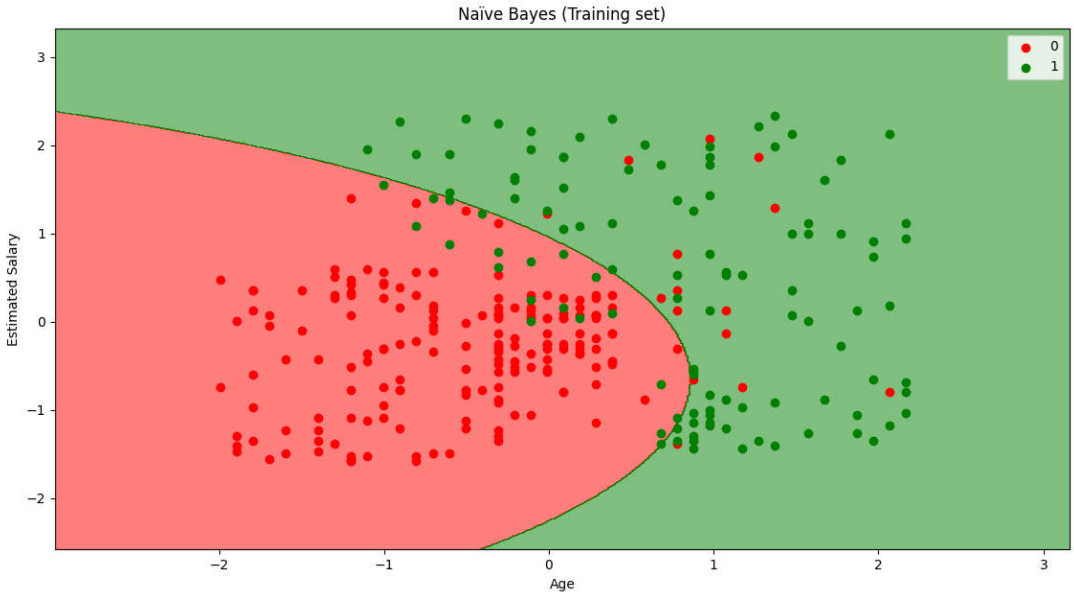
# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value


# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Naïve Bayes (Training set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

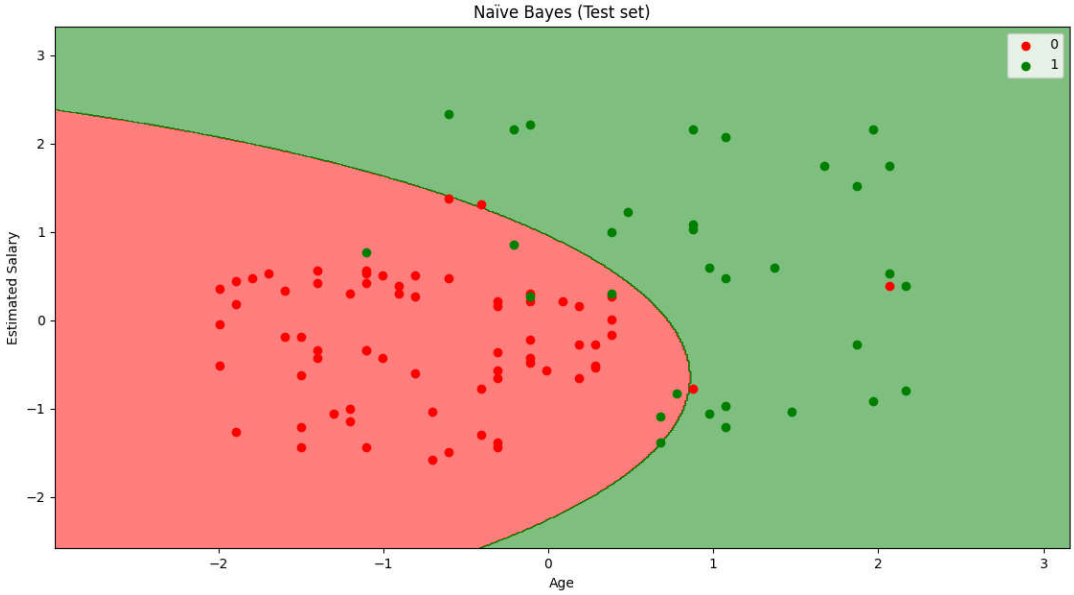
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Naïve Bayes (Test set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# python prctc_niv_bys.py
```

Training Data-set plot



Test Data-set plot



Decision Tree Classification

3.6.1 Decision Tree Regression Algorithm

The goal of the ***Decision Tree Regression Algorithm*** is to ***split data*** into ***similar groups***. The **algorithm uses** something called ***Information Entropy***, which is a mathematical process (that is super complicated).

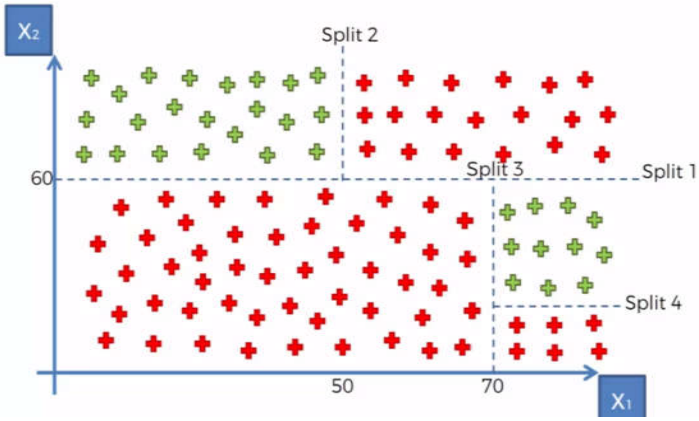
- The point of the algorithm is to ***split the data in such a way that information is added*** and stops splitting the data when it's unable to add any more information to the dataset.

❒ In the ***Regression Chapter***, we saw the '**Decision Tree Regression**'. Regression Trees ***predict data*** of ***real numbers***.

- Classification trees helps us to classify our data, among the categorical variables such as male or female apple or orange or different types of colors and variables of that sort.
- Whereas Regression trees are designed to help you predict outcomes which can be real numbers so for instance the salary of a person or the temperature that's going to be outside.

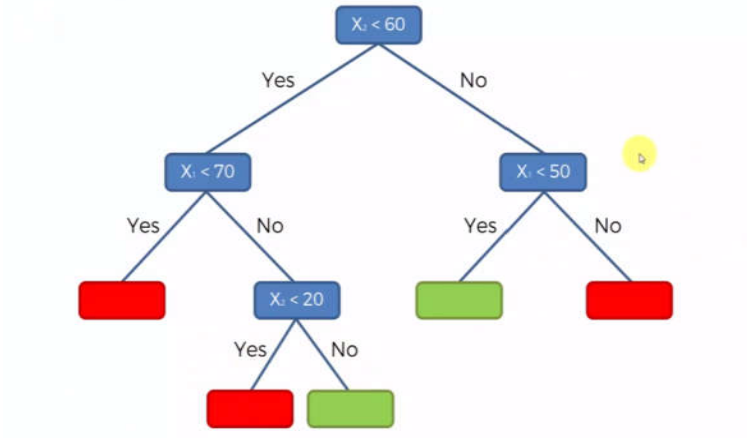
❒ In ***Classification trees***, we ***classify the data*** and ***predict categories*** and. Here in this part we will be looking at but the approach is almost identical to that of the Decision Tree Regression.

- Here we see a graph with ***splits across different categories*** using the ***Decision Tree Algorithm***.
- The decision tree makes splits in a way that would ***maximize the number of points in each class*** and tries to **minimize entropy**. So in plain English, ***it groups as many of the green and red dots together***.



❒ The decision tree would like this:

- This is the method that will be used to add a new data point and classify it to the graph.
- Additionally, we ***don't always*** have to go all the way to the ***very end of the tree***.
- Most of the time the tree will be very long and so at a ***reasonable point*** you ***could stop*** and have it ***run a probability*** of where to ***classify the point***.



- For example, let's say we stop going through the graph at $X_1 < 70$. If the statement is True then we would have, in this case a ***100% chance*** of it being ***red***. Otherwise, ***instead going further down the list***, we could say that since there are more ***Green Dots*** than ***Red*** Ones, I could make a prediction and say that the point will be green. This is extremely useful, as it is a lot ***less computationally*** expensive.

❒ ***How it works:*** Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. The algorithm is just split the data so that point of a class will be maximum.

- ☞ The algorithms going to find optimal splits that are going to maximize the number of different points in each one of leaves (leaves are splitted data pockets, final leaves are actually called a terminal leaves).
- ☞ For example our data-set is splitted as:

$$X_2 < 60;\ x_1 < 70; X_1 < 50; X_2 < 20$$

- Old Method
- Reborn with upgrades
- Random Forest
- Gradient Boosting
- etc.

☝ For more than two feature variables, the **Tree just grow bigger**.

- ▶ It is a ***tree-structured classifier***, where ***internal nodes*** represent the ***features*** of a dataset.
- ▶ ***Branches*** represent the ***decision*** rules and
- ▶ Each ***leaf node*** represents the ***outcome***.

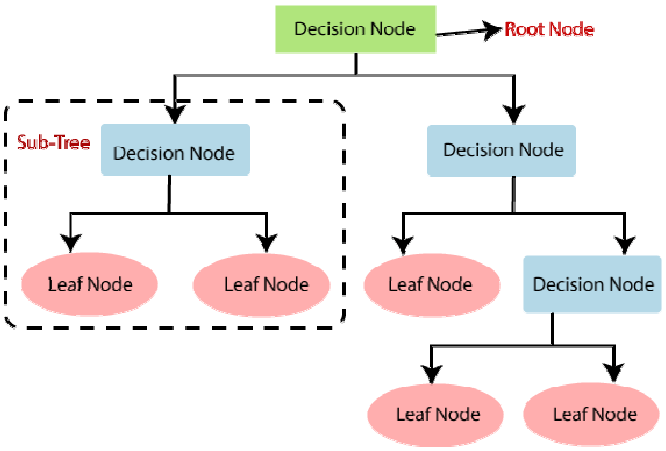
❒ ***Decision Node and Leaf Node:*** In a ***Decision tree***, there are two nodes, which are the ***Decision Node*** and ***Leaf Node***.

- ☞ ***Decision nodes*** are used to make any ***decision*** and have ***multiple branches***, whereas
- ☞ ***Leaf nodes*** are the ***output*** of those ***decisions*** and ***do not contain any further branches***. The decisions or the test are performed on the basis of features of the given dataset.

- ▶ It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.
- ▶ It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.

❒ In order to build a tree, we use the ***CART algorithm***, which stands for ***Classification and Regression Tree algorithm***.

- ☞ A decision tree simply asks a ***question***, and based on the ***answer (Yes/No)***, it further ***split the tree*** into ***subtrees***. Below diagram explains the general structure of a decision tree:



NOTE: A decision tree can contain categorical data (YES/NO) as well as numeric data.

3.6.2 Implementation of Decision Tree in Python

```
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion = 'entropy', random_sate = 0)
classifier.fit(X_train, y_train)
```

❒ ***No need scaled feature:*** Decision Tree is not based on Euclidean distance, so we need not really apply feature scaling here. Feature scaling is important when your algorithm is based on Euclidean distance.

- ☞ However in we are building some trees, so we want to plot those trees in real values rather than scaled values.

☝ If you want to plot some decision trees like the real tree itself then you can remove this ***feature scaling part*** here to have interpretation of your valuables.

❒ ***Parameters:***

criterion : {"**gini**", "**entropy**"}, default="gini"

☞ The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

☞ Most useful classifier and most common decision tree is based on ***"Entropy"***, eg: ***"Maximum Entropy"*** for NLP.

☞ So that the *final nodes* of your tree to be as much *homogeneous* as *possible*. I.e. after each split the *more homogeneous* is a *group of users*, the more the *entropy* is *reduced* from the *parent node* to the *group child node*.

☞ So after the *split* if the *entropy* in the *resulting child node* is *zero* then that means that this *child node* is a *fully homogeneous group of users* (only users of the same class) and the information gain mentioned here is basically the difference between the entropies before and after displays.

Practiced version

```
# ---------- Naive Bayes model ----------
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values


        # Feature-Scaling after Data Split

# Data Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler

st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)


# Fit train set to Naïve Bayes classifier: No parmeter is needed
from sklearn.tree import DecisionTreeClassifier
clsFier = DecisionTreeClassifier(criterion="entropy", random_state=0)
clsFier.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = clsFier.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value

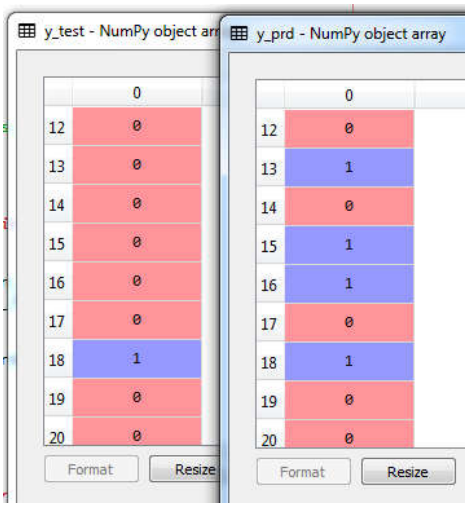

# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Decision Tree (Training set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
```

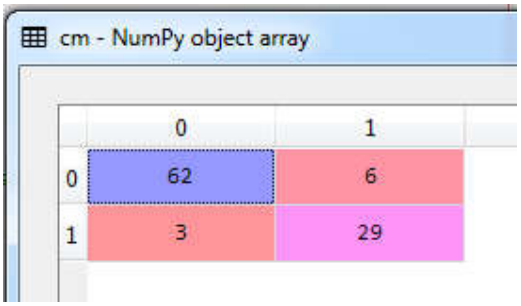
```
pLt.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Decision Tree (Test set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# python prctc_DTC.py
```

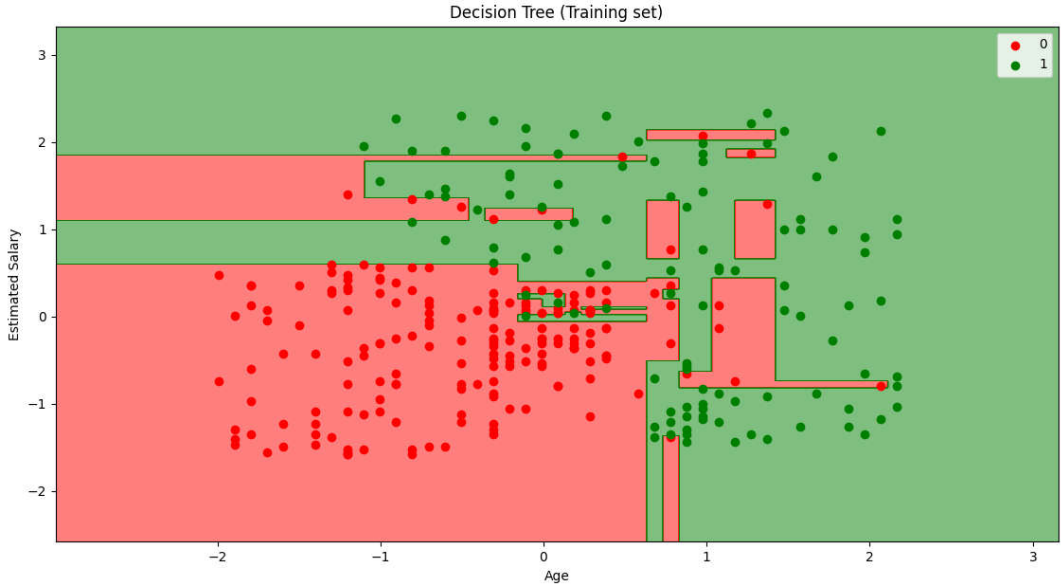
Test Data vs Predicted data and CONFUSION MATRIX



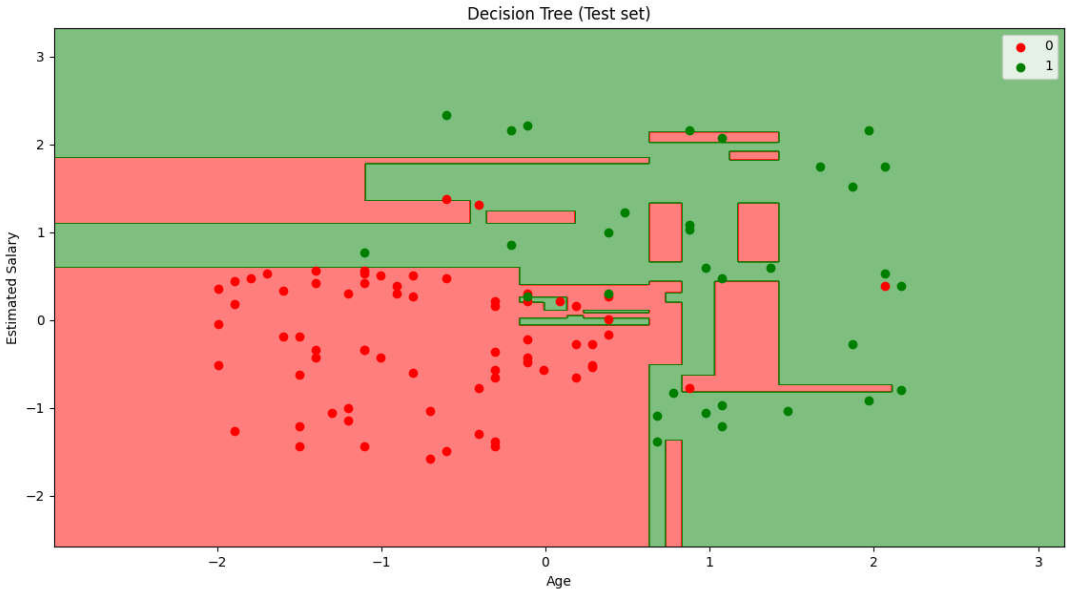


	0	1
0	62	6
1	3	29

Training set Plot



Test set Plot



- From above figures, we see that, ***Prediction boundary*** is composed on Horizontal and vertical lines. This model actually considered to classify all red and green point at 100%.

- ***This model is Kind of Over-fitted:*** This kind of like over-fitting, because you see it's trying to catch every single user into the right category.

Random Forest Classification

3.7.1 Random Forest Classification

Random Forest is a version of ensemble learning (such as gradient boosting). Ensemble learning is when you take multiple algorithms or the same algorithm multiple times and you put them together to make something much more powerful than the original.

❒ ***How it actually works:***

[1] ***STEP 1:*** Pick at ***random K data points*** from the Training set.

[2] ***STEP 2:*** Build the ***Decision Tree*** associated to these ***K data points***.

[3] ***STEP 3:*** Choose the number ***Ntree*** of trees you want to build and **repeat STEPS 1 & 2**

[4] ***STEP 4:*** For a ***new data point***, make ***each*** one of your ***Ntree*** trees ***predict*** the value of ***Y*** to for the ***data point*** in question, and assign the new data point the ***average*** across all of the ***predicted Y values***.

One of decision trees practical example is Microsoft's Xbox's Body-motion Sensor.



Real-Time Human Pose Recognition in Parts from Single Depth Images

Jamie Shotton Andrew Fitzgibbon Mat Cook Toby Sharp Mark Finocchio
Richard Moore Alex Kipman Andrew Blake
Microsoft Research Cambridge & Xbox Incubation

Abstract

We propose a new method to quickly and accurately predict 3D positions of body joints from a single depth image, using no temporal information. We take an object recognition approach, designing an intermediate body parts representation that maps the difficult pose estimation problem into a simpler per-pixel classification problem. Our large and highly varied training dataset allows the classifier to estimate body parts invariant to pose, body shape, clothing, etc. Finally we generate confidence-scored 3D proposals of several body joints by reprojecting the classification result and finding local modes.

The system runs at 200 frames per second on consumer hardware. Our evaluation shows high accuracy on both

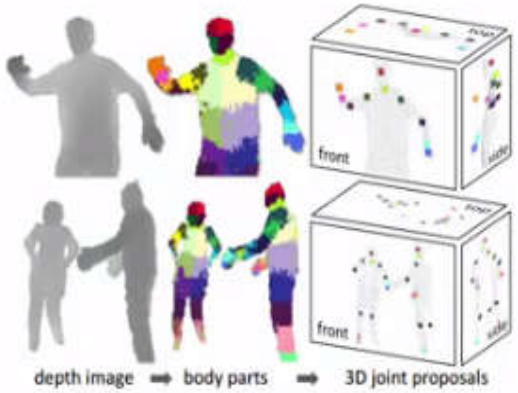
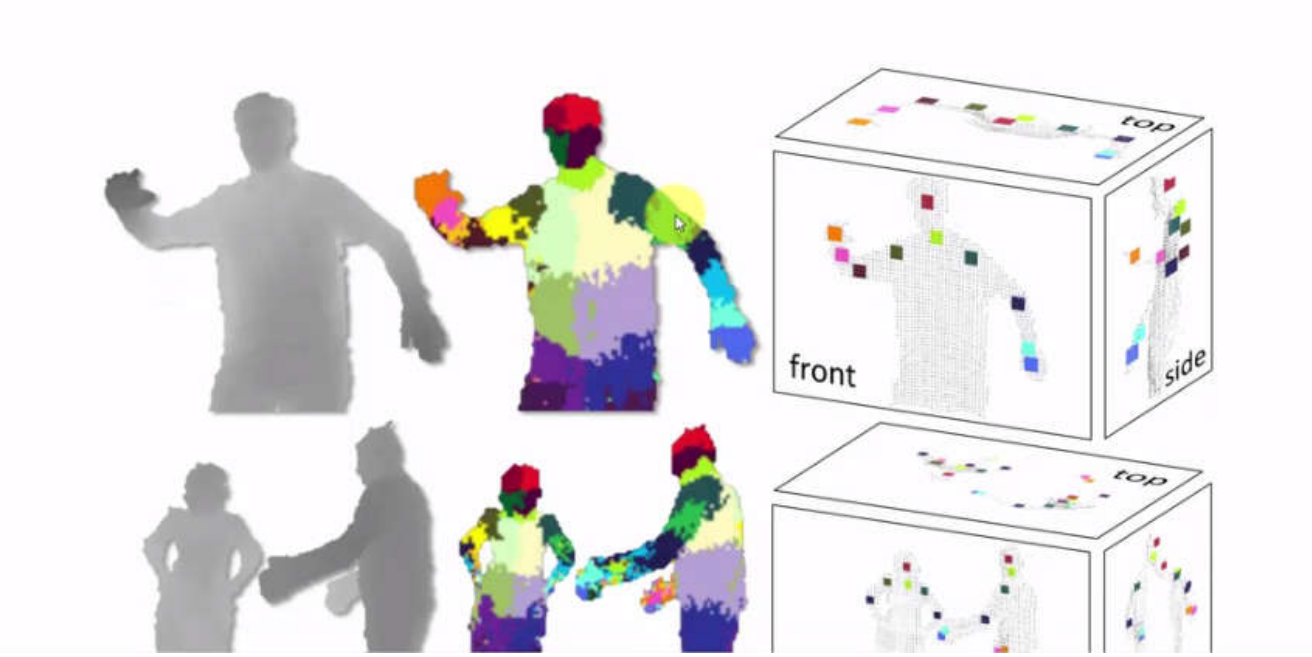


Figure 1. **Overview.** From an single input depth image, a per-pixel



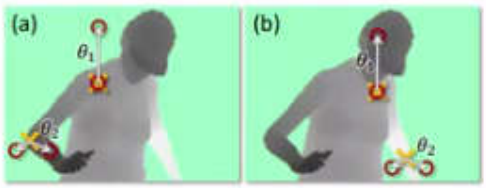


Figure 3. **Depth image features.** The yellow crosses indicates the pixel x being classified. The red circles indicate the offset pixels as defined in Eq. 1. In (a), the two example features give a large depth difference response. In (b), the same two features at new image locations give a much smaller response.

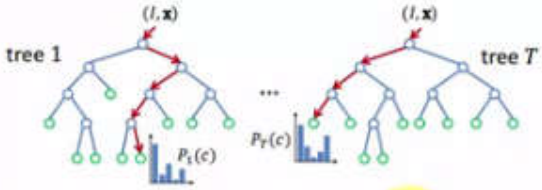


Figure 4. **Randomized Decision Forests.** A forest is an ensemble of trees. Each tree consists of split nodes (blue) and leaf nodes (green). The red arrows indicate the different paths that might be taken by different trees for a particular input.

3.3. Randomized decision forests

3.7. 2 Python Implementation of 3.7.1 Random Forest Classification

```
# Fit train set to Random forest classifier: No parmeter is needed
from sklearn.ensemble import RandomForestClassifier
clsFier = RandomForestClassifier(n_estimators= 10, criterion="entropy", random_state=0)
clsFier.fit(X_train, y_train) # fit the dataset
```

❒ ***Parameters:***

- ***n_estimators= 10***, Number of trees to be used. Try different number of trees to ***detect overfitting***.
- Similar to Decision tree we use ***criterion = "entropy"***

❒ ***Confusion matrix:***

- Some Test values and Predicted values are given to the right:
- Below is the confusion matrix (with 10 trees/estimators) :

	0	1
0	63	5
1	4	28



Practiced Version

```
# ---------- Naive Bayes model ----------
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values


        # Feature-Scaling after Data Split

# Data Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler

st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)


# Fit train set to Random forest classifier: No parmeter is needed
from sklearn.ensemble import RandomForestClassifier
clsFier = RandomForestClassifier(n_estimators= 10, criterion="entropy", random_state=0)
clsFier.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = clsFier.predict(X_test)

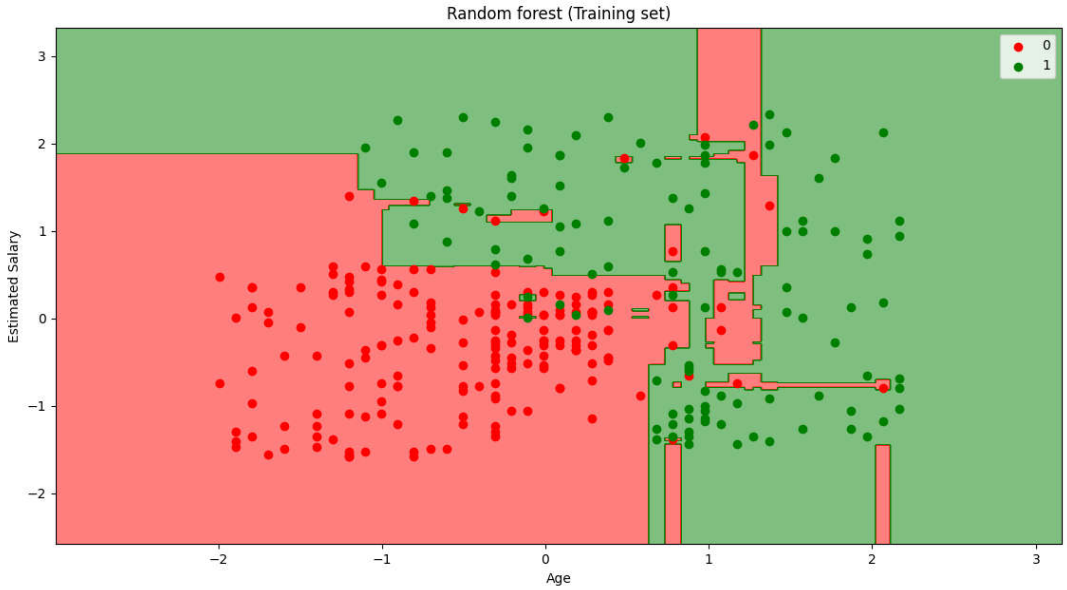
# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value


# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Random forest (Training set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

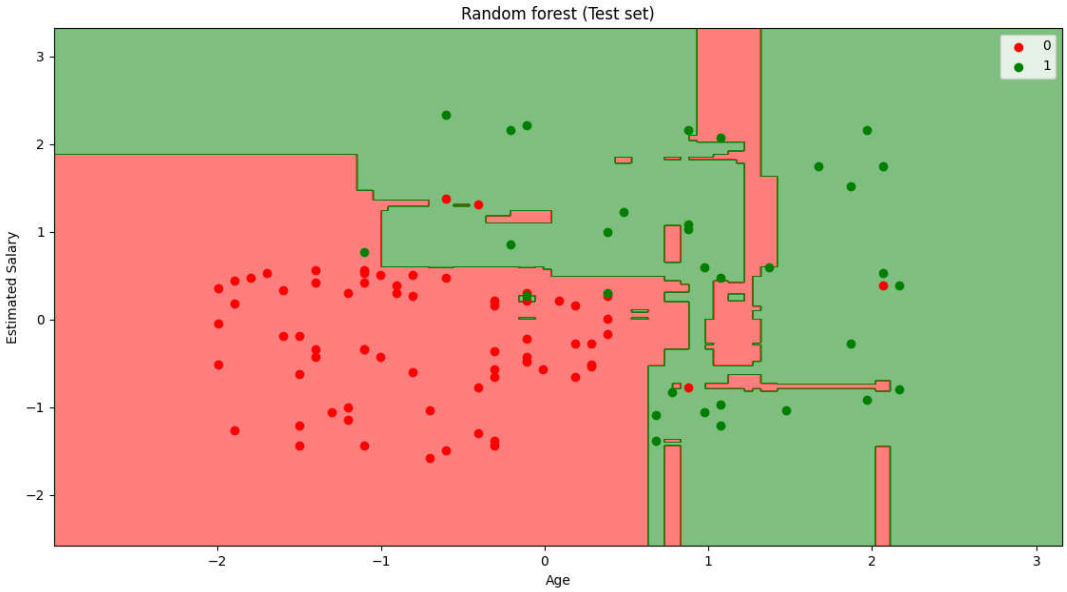
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.5, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Random forest (Test set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# python prctc_RFC.py
```

Train set Plot



Test set Plot



📣 Some ***overfitting*** is appearing in the train set. Notice the ***small red region*** in the ***Green area***.

📣 So as a conclusion we had the best classifier as ***Kernel-SVM*** classifier and ***Naïve Bayes*** Classifier. For our problem we just gonna use ***Kernel-SVM*** classifier.

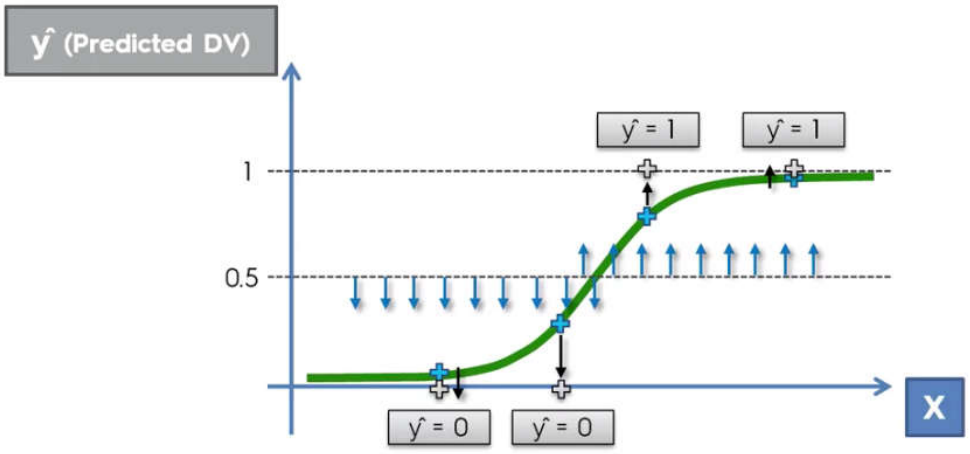
📣 We can start thinking how to choose the best classifier for our problem. And it's always related to this battle between ***accuracy*** (getting the maximum number of correct predictions) and preventing ***overfitting***.

Chapter 3 : part 8

Evaluating Classification Models Performance

False Positives, False Negatives, Confusion Matrix, Accuracy Paradox, CAP Curve, CAP Curve Analysis

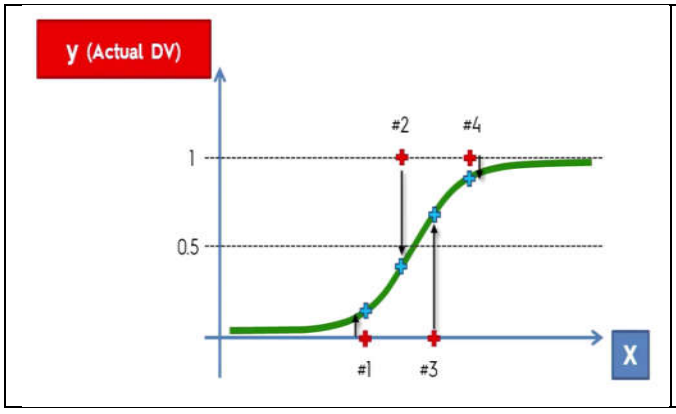
3.8.1 False Positives - False Negatives

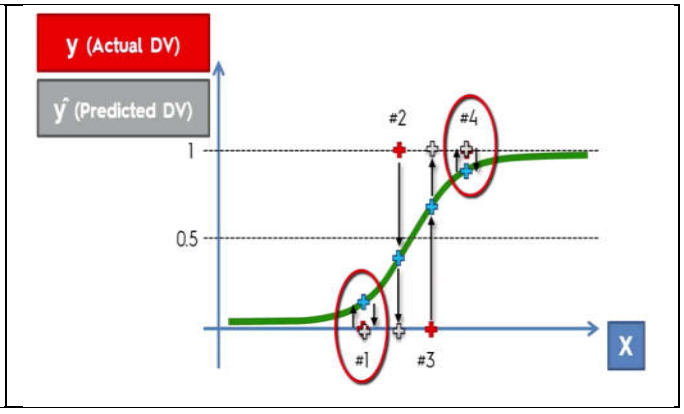


❒ In our ***Logistic Regression Function***, in terms of the **predicted** value for the **dependent variable** (***y***), we agreed that anything ***below*** the ***50%*** line will be ***projected downwards*** onto the ***0*** horizontal line (***y=0***) and anything ***above*** the ***50%*** line will be ***projected upwards*** onto the ***100*** percent horizontal line (***y=1***) and that allowed us to turn probabilities into actual predictions. Either ***Yes*** or ***No***.

❒ Now, let's just pick out ***four values*** that we really know they ***exist*** in our ***data set***. And we use them to create this ***Logistic Regression***. And let's do the same thing with them.

- ☞ Here the actual values of these four observation is Red colored, we then project those in our Logistic curve.
- ☞ Now according to ***50% horizontal line***, ***#2*** and ***#3*** observations falls in the wrong "Horizontal line". So these two are the error of the model (therefore the logistic regression made two mistakes). The 1st observation ***#1*** and 4th observation ***#4*** are correct predications.



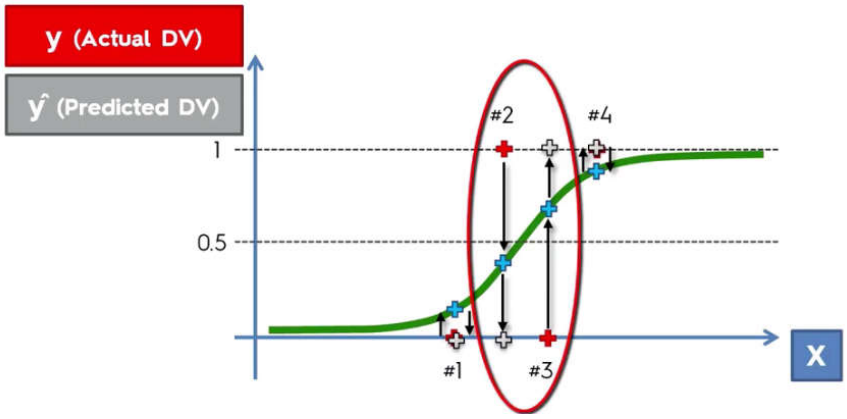


❒ ***False Positive (Type I error):*** The first kind of error is the **mistaken rejection** of a **null hypothesis** (mistaken rejection of an actually true null hypothesis) as the result of a test procedure. This kind of error is called a ***type I error (false positive)*** and is sometimes called an error of the first kind.

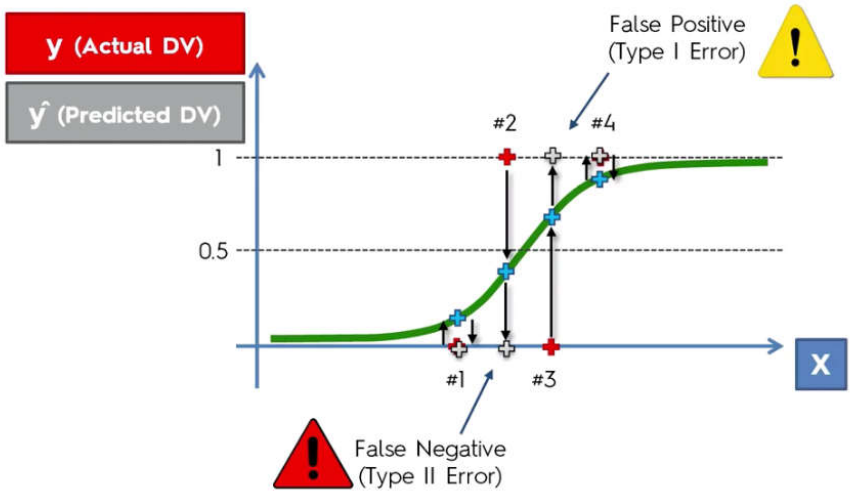
- ☞ It means that we said we predicted a positive outcome but it was false
- ☞ In terms of the courtroom example, a type I error corresponds to *convicting* an *innocent defendant*.
- ☞ The ***#3 point*** is predicted as "it will happen", but in reality this "actually not happened". This is kind of "Warning". **Predicted** as **Positive** but in **reality** they are **Negative**, hence ***False-Positive***.

❒ ***False Negative (Type II error):*** The second kind of error is the **mistaken acceptance** of the **null hypothesis** (mistaken acceptance of an actually false null hypothesis) as the result of a test procedure. This sort of error is called a ***type II error (false negative)*** and is also referred to as an error of the second kind.

- ☞ We predicted that there won't be an effect but the effect actually did occur
- ☞ In terms of the Courtroom example, a type II error corresponds to ***acquitting*** a ***criminal***.
- ☞ The ***#2 point*** is predicted as "it won't happen", but in reality this "actually happened". This is kind of "fatal Error". **Predicted** as **Negative** but in **reality** they are **Positive**, hence ***False-Negative***.



☝ ***False negative is a bit worse:*** Some people think, type 1 as less dangerous than type 2. ***False negative*** is a bit worse in my view, because once you say something's ***not going to happen*** but it ***actually does happen*** then you ***can't even be Prepared*** for it.



3.8.2 Confusion Matrix

❒ ***CONFUSION MATRIX:*** The *confusion matrix* is a matrix used to determine the *performance* of the *classification models* for a given set of `test-data`. It can *only be determined* if ***the true values for test data are known***.

☝ Since it shows the ***errors*** in the ***model performance*** in the form of a ***matrix***, hence also known as an ***ERROR MATRIX***. Some features of Confusion matrix are given below:

- ☞ For the ***2 prediction classes*** of classifiers, the matrix is of ***2*2 table***, for ***3 classes***, it is ***3*3 table***, and so on.
- ☞ The matrix is divided into 2D, that are ***predicted values*** and ***actual values*** along with the total number of predictions.
- ☞ ***Predicted values*** are those values, which are ***predicted by the model***, and ***actual value***s are the ***true values*** for the given observations. It looks like the below table:

n = total predictions	Actual: No	Actual: Yes
Predicted: No	True Negative	False Positive
Predicted: Yes	False Negative	True Positive

☞ The above table has the following cases:

- ***True Negative:*** Model has given ***prediction No***, and the real or actual value was also ***No***.
- ***True Positive:*** The model has ***predicted yes***, and the actual value was also ***true***.
- ***False Negative:*** The model has ***predicted no***, but the actual value was ***Yes***, it is also called as ***Type-II error***.
- ***False Positive:*** The model has ***predicted Yes***, but the actual value was ***No***. It is also called a ***Type-I error***.

❒ ***Need for Confusion Matrix in Machine learning:***

- ✓ It evaluates the **performance** of the **classification models**, when they make ***predictions*** on ***test data***, and tells how good our classification model is.
- ✓ It not only tells the ***error*** made by the ***classifiers*** but also the ***type of errors*** such as it is either ***type-I*** or ***type-II*** error.
- ✓ With the help of the ***confusion matrix***, we can calculate the ***different parameters*** for the model, such as ***accuracy***, ***precision***, etc.

Consider the following example: Suppose we are trying to create a model that ***can predict*** the result for the ***disease*** that is either a ***person has*** that disease or ***not***. So, the confusion matrix for this is given as:

n = 100	Actual: No	Actual: Yes	
Predicted: No	TN: 65	FP: 3	68
Predicted: Yes	FN: 8	TP: 24	32
	73	27	

True Negative: 65
True Positive: 24
False Negative: 8
False Positive: 3

☞ The table is given for the two-class classifier, which has two predictions "***Yes***" and "***NO***." Here, ***Yes*** defines that patient has the ***disease***, and ***No*** defines that patient does ***not has that disease***.

☞ The classifier has made a total of ***100*** predictions. Out of ***100*** predictions, ***89 are true*** predictions, and ***11 are incorrect*** predictions.

☞ The ***model*** has given prediction "***yes***" for ***32*** times, and "***No***" for ***68*** times. Whereas the ***actual*** "***Yes***" was ***27***, and actual "***No***" was ***73*** times.

❒ ***Classification Accuracy/ Accuracy Rate:*** It is one of the important parameters to determine the ***accuracy of the classification*** problems. It defines ***how often the model predicts the correct output***. It can be calculated as the ***ratio*** of the number of ***correct predictions*** made by the ***classifier*** to all number of ***Total predictions***.

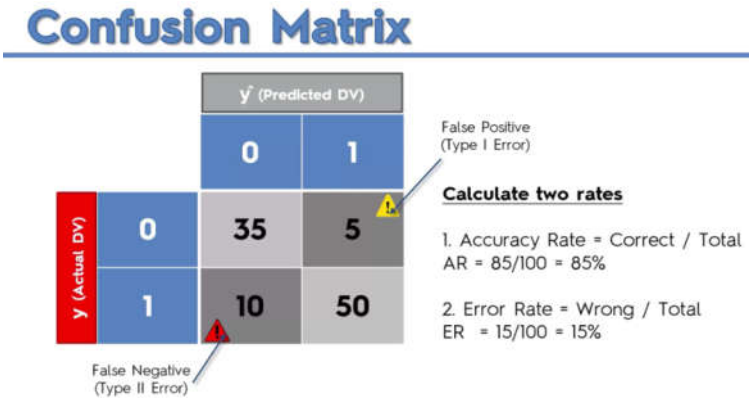
$$A_r = \frac{Total\ correct\ predictions}{Total\ Observations} = \frac{True_{Negative} + True_{Positive}}{True_{Negative} + True_{Positive} + False_{Negative} + False_{Positive}}$$

☞ In our example $A_r = \frac{65+24}{100} = \frac{89}{100} = 89\%$

❒ ***Misclassification rate/ Error Rate:*** It is also termed as ***Error rate***, and it defines ***how often the model gives the wrong predictions***. The value of ***Error Rate*** can be calculated as the number of ***incorrect predictions*** to ***Total predictions***.

$$E_r = \frac{Total\ incirrect\ predictions}{Total\ Observations} = \frac{False_{Negative} + False_{Positive}}{True_{Negative} + True_{Positive} + False_{Negative} + False_{Positive}}$$

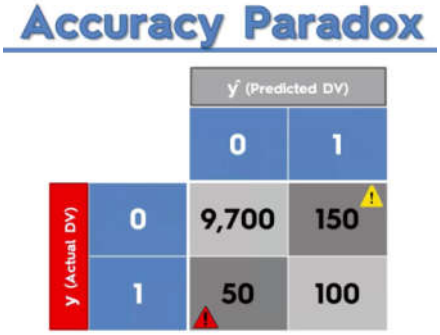
☞ In our example $E_r = \frac{8+3}{100} = \frac{11}{100} = 11\%$



3.8.3 Accuracy Paradox

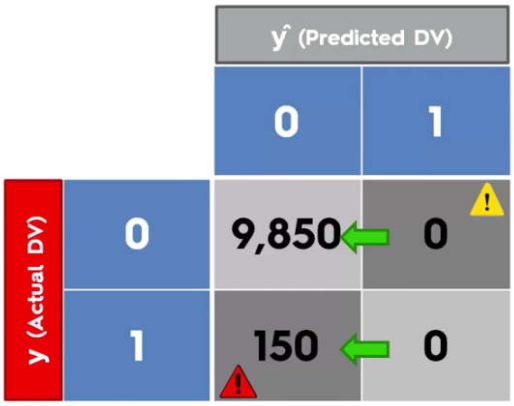
❒ Here a confusion matrix of 10000 records in it. This model has made 150 TYPE-I errors and 50 TYPE-II errors. Now let's calculate the accuracy rate:

$$A_r = \frac{9700 + 100}{10000} = 98\%$$



❒ ***Now we abandon the model:*** We're going to tell the model to stop making predictions, which is going to abandon them all completely. And we're going to say that from now on our prediction is always ***0*** (i.e predict ***0*** only, not ***1***). We're always going to predict that the event is not going to occur.

☞ So basically what will happen to the confusion matrix is these records will move from the right column to the left column and our new confusion matrix will look like this:



Scenario 1:

Accuracy Rate = Correct / Total
AR = 9,800/10,000 = 98%

Scenario 2:

Accuracy Rate = Correct / Total
AR = 9,850/10,000 = 98.5%

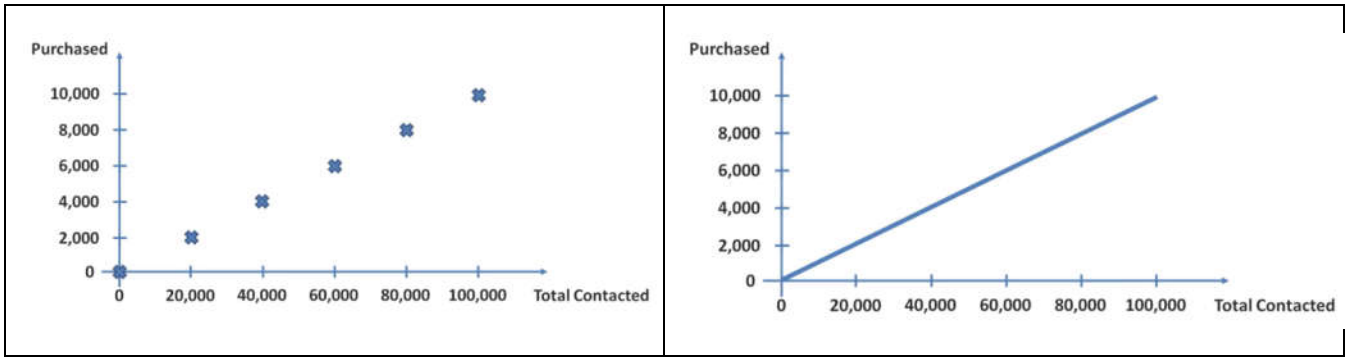
☞ The ***accuracy rises*** when we abandon our model. Now it is 98.5%. And as you can see what we did is we just completely stopped using a model but the accuracy rate went up.

☞ Now you're not applying any kind of logic into your decision making process. And your accuracy rate is going up so it's misleading you into a wrong conclusion that ***You Should Stop Using Models*** and this effect is called the ***Accuracy Paradox***.

3.8.4 CAP Curve Analysis

Lets say a store sells clothes and your store has a total of 100000 customers (horizontal axis). Whenever you send an offer like an e-mail to all 100000 customers approximately 10% of them respond and purchase the product. So I'm going to place 10000 (10% percent of the total) on the vertical axis.

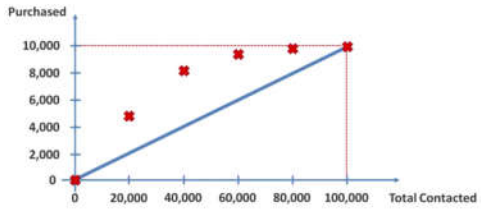
❒ We now draw an ***average line*** (for random values) for our *customer response rate*.

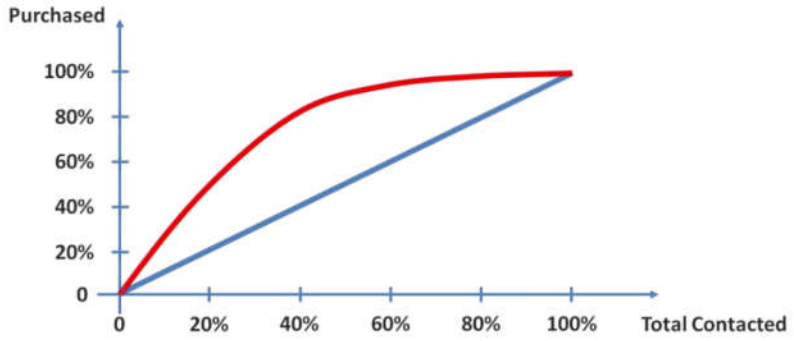


☞ So above represents the line for **average response rate**. Now the question is can we somehow ***improve*** this ***experience***, can we get **more customers** to respond to offers when we send out our **letters**. Can we somehow ***target our customers more appropriately*** so to get a ***better response rate*** instead of sending out these offers ***randomly***.

❒ Well to start off with let's build a model just like we did in the previous section. It will actually predict whether or not they will purchase the product. Because purchasing product is also a binary variable (yes or no). So we can build a logistic model.

- We can take a group of customers who purchased from our store, before we send out the offer. For example: A ***male*** or ***female***. Which ***country*** were they. What ***age*** they were. Were they browsing on ***mobile*** or ***computer***.
- So that we can take them into account and put them into a logistic regression and get a model. Which will help us to predict the likelihood of certain types of customers purchasing from the store (based on their characteristics).
- And once we've built this model how about we apply it to select customers we will send the offer.

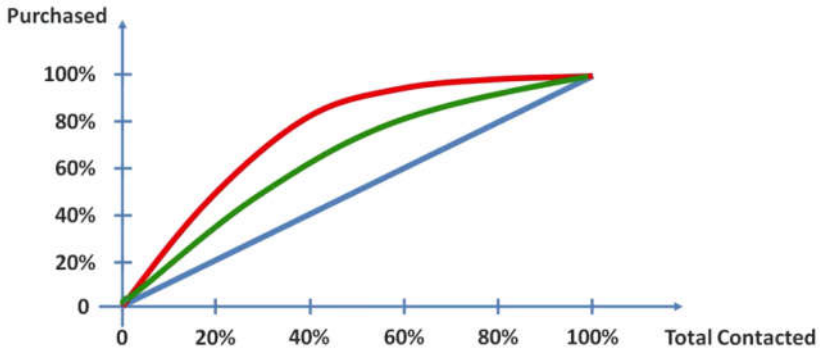




- Lets say Red curve represents the model. This Red line here is called the cumulative accuracy profile of your model. The better your model the ***larger*** will be the ***area*** between ***Red*** and ***Blue*** line.
- If your model is worse, this red line will be closer to the blue line (random line). And this is how the ***CAP-curve*** is normally represented.

❒ Let consider another model represented by ***Green*** curve. Now let's say we had *less access to independent variables* or we didn't see that *there's a multicollinearity* effect in a model or something else that went wrong. And that making this model worse.

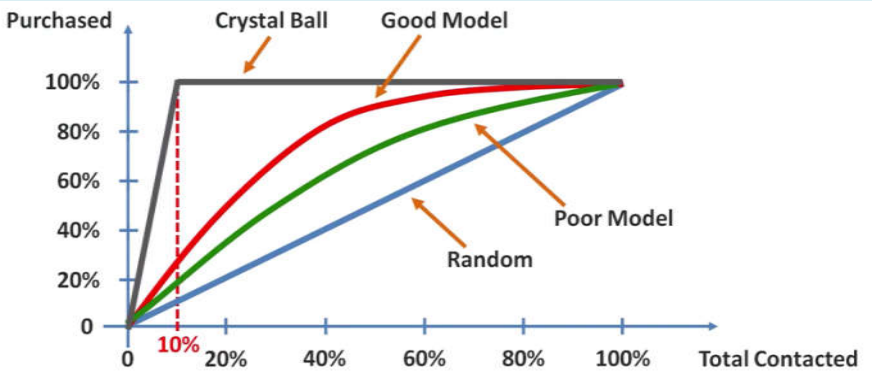
- By plotting the ***CAP CURVES*** (Green and Red) you'll be able to compare models to each other and understand how much ***gain*** this is also sometimes called the ***Gain Chart***.
  - How much gain you get in each of these models compared to the random scenario or how much again you get additional gain you get from switching from one model to the next or from the ***Green*** one to the ***Red*** one for instance. This is how you're improving your hit ratio and therefore you're improving your return on investment.



  - Hence, the read model is better. The green line is a poor model (it's always is better than random but it's still not as good as the red one).

❒ ***Ideal Line:*** And there's one more line, this line is the ***ideal line***. Something like, you had a crystal ball if you could predict exactly who is going to purchase and contact those people. This is what it would look like (the ***Gray*** line).

- We know that only ***10%*** of our customers ever purchase. Notice that the ***Red-Dotted vertical line*** indicates exactly ***10%***.



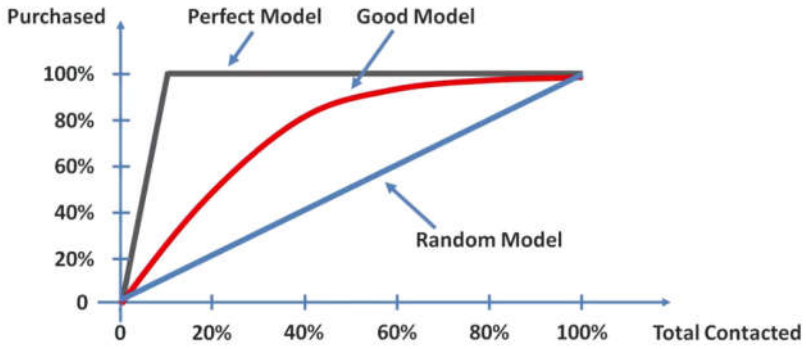
❐ ***CAP curve and ROC curve are not the same:*** Note that ***CAP*** means *Cumulative Accuracy Profile*. There is a ***ROC***-curve which is *Receiver Operating Characteristic*.

CAP = Cumulative Accuracy Profile

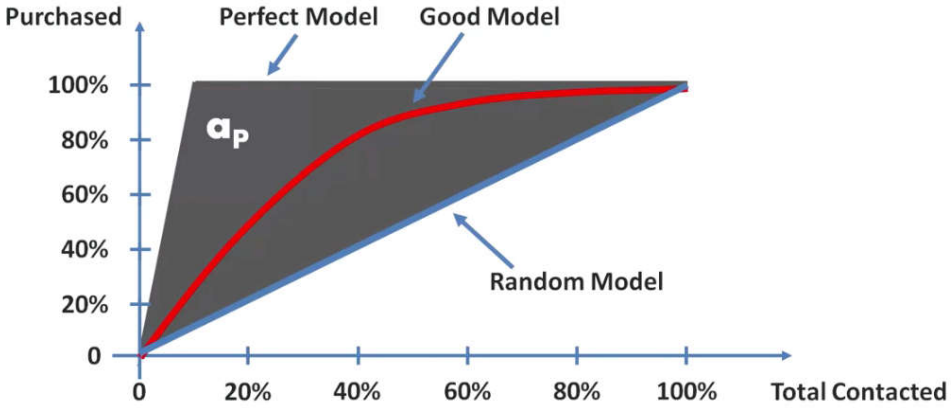
≠

ROC = Receiver Operating Characteristic

🛠 ***How to Analyze:*** So now we have, three lines. One the ***CAP***-Curve ***Red*** line), the ***Blue*** line is the **random line,** the ***Gray*** line which is the *perfect model*.

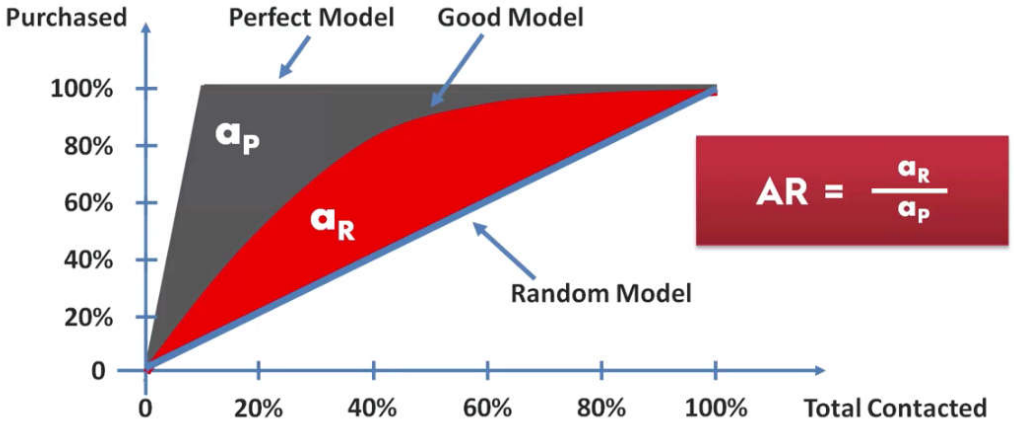


☞ The closer your ***Red*** line to the ***Gray*** line the ***better*** you model, the closer to the ***Blue*** line the ***worse***. So how can we quantify this effect? Well there is a standard approach to **Calculate The Accuracy Ratio** and to calculate accuracy ratios: you take the ***area under the perfect model*** or the ***perfect line*** which is color in ***Gray*** here and it's called a_P. Notice the following figure



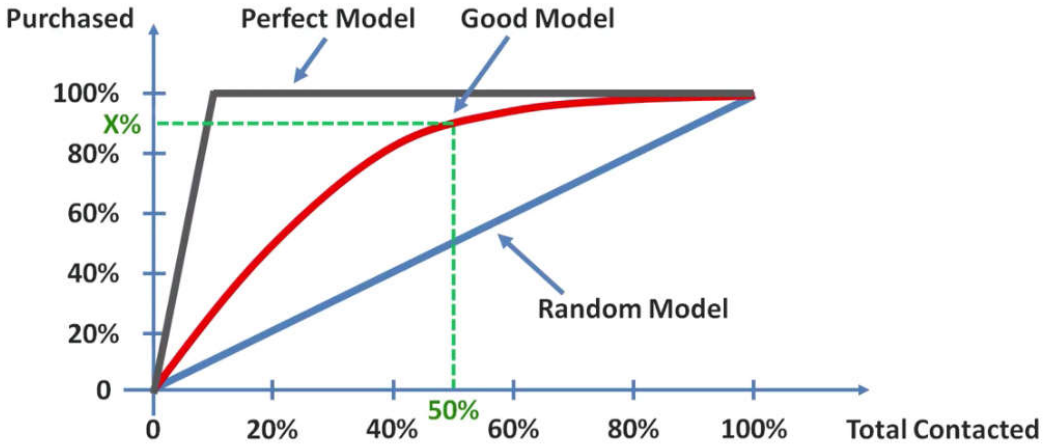
☞ And then you need to take that **area under the Red line** which is **colored** in **Red** here which is a_R and then you need to divide one by the other. This ratio is between 0 and 1.

$$AR = \frac{a_R}{a_p}$$



☞ The closer this ratio is to 1 the better, the further it is away from 1 and close to 0 the worse your model is.

❒ ***Other way:*** Now let's get rid of areas and instead of looking at the area what you can do is look at the 50% line on the **horizontal** axis (X-axis) and look where it **crosses your model-curve** and then project that point on the vertical axis (Y-axis).



☞ And there are some rules for the X% on the Y-axis projected value:

X < 60%: Rubbish model

60% < X < 70%: Poor model

70% < X < 80%: Good model

80% < X < 90%: Very Good model

90% < X < 100%: That's odd !! This model is so good !! Need to check that there is ***OVERFITTING***.

☝ ***OVERFITTING is bad:*** So Remember the ***OVERFITTING***. It could Ruin your model.

Chapter 4 : Part 1

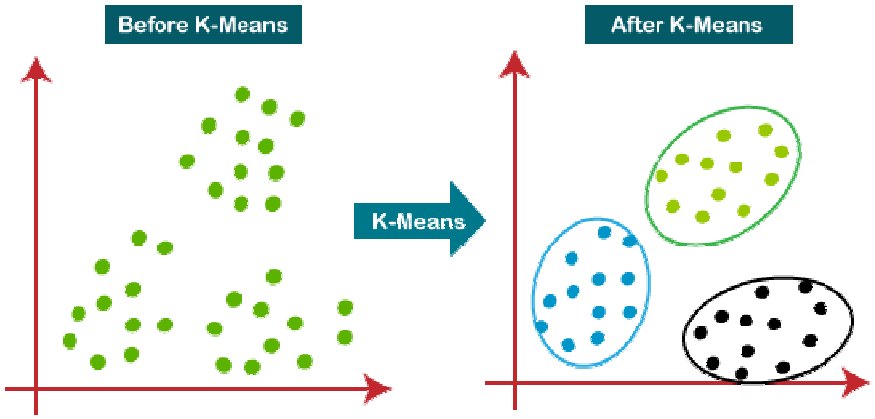
K-Means Clustering

4.1.1 K-Means Clustering

K-Means Clustering is an algorithm that allows you to ***Cluster*** your data. ***K-Means Clustering*** is an ***unsupervised learning*** algorithm that is used to ***solve*** the ***clustering problems*** in *machine learning* or *data science*.

- ***K-Means Clustering*** is an ***Unsupervised Learning*** algorithm, which ***groups*** the ***unlabeled dataset*** into ***different clusters***. Here ***K*** defines the ***number*** of ***pre-defined clusters*** that need to be created in the process, as if ***K=2***, there will be ***`two clusters`***, and for ***K=3***, there will be ***`three clusters`***, and so on.
  - It is an iterative algorithm that ***divides*** the ***unlabeled dataset*** into ***k different clusters*** in such a way that each dataset belongs only one group that has **similar properties**.
  - It allows us to ***cluster*** the ***data*** into ***different groups*** and a convenient way to discover the categories of groups in the unlabeled dataset on its own ***without*** the need for ***any training***.
  - It is a ***`centroid-based algorithm`***, where each ***cluster*** is ***associated*** with a ***centroid***. The main aim of this algorithm is to ***minimize*** the ***sum*** of ***distances*** between the ***data-point*** and their ***corresponding clusters (centroid)***.

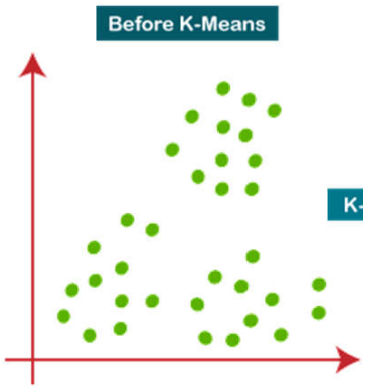
- The algorithm takes the ***unlabeled*** dataset as ***input***, divides the dataset into ***k-number of clusters***, and repeats the process until it does not find the ***best clusters***. The value of ***k*** should be ***predetermined*** in this algorithm.
  - The k-means clustering algorithm mainly performs following tasks:
    - Determines the ***`best value`*** for ***K*** ***`center points`*** or ***`centroids`*** by an iterative process.
    - Assigns each ***data-point*** to its closest ***k-center***. Those ***data points*** which are near to the particular ***k-center***, create a ***cluster***.
    - Hence each cluster has data-points with some commonalities, and it is away from other clusters.
  - The below diagram explains the working of the K-means Clustering Algorithm:



- It is a very convenient tool for discovering ***categories*** or ***groups*** in your data-set.

- Let's imagine that we've got **two variables** in our data set and we decided to plot those two variables on ***X*** and ***Y*** axis. The question is:
  - How our observations are configured according to these two variables?
  - Can we identify certain groups among all variables?

- What the ***K-Means*** does for you is: it takes out the complexity from this decision making process and allows you to very easily identify those clusters (actually called **clusters of data points**) in your data-set.



❒ ***Steps of K-means Algorithm:*** The working of the K-Means algorithm is explained in the below steps:

[1]. ***Step-1 (Choose the number K of clusters):*** Select the number K to decide the number of clusters. Let's imagine that we've agreed on a *number* of *clusters* for a certain challenge, say ***3*** or ***2*** or ***5 clusters***. Once you've done that then you proceed to ***step 2***.

- ☞ We'll talk more about how to select the ***Optimal Number Of Clusters***.

[2]. ***Step-2:*** Select **random K points** or ***CENTROIDS***. (It can be other from the input dataset). The random k points will be the ***centroid*** of your ***clusters*** and ***not necessarily*** these ***points have to be from your dataset***.

- ☞ As you saw we had a **Scatterplot,** we could select any points in that **Scatterplot**. The points *don't have to be part of the observations*, they can be any random ***x*** and ***y*** values on your ***Scatterplot***. (As long as you just selects a ***certain number of centroids*** that are going to equate to the number of clusters that you have decided upon).

[3]. ***Step-3:*** Assign each data point to their **Closest Centroid**, which will form the ***predefined K clusters***.

- ☞ So you're ***starting clusters*** and then there's going to be an ***Iterative process*** to ***Refine*** those ***clusters***.
- ☞ Basically so you just **check** for **every point** in a **data set**, which of them is the closest.
  - ➢ Closest is a kind of vague term here, because it depends on what kind of distance you're measuring (is it Euclidean distances? Or some other sort of distance?).
  - ➢ For the purposes of simplicity, we're going to talk about Euclidean distances (that's basically geometrical distances).

[4]. ***Step-4:*** Calculate the ***Variance*** and *place* a *new centroid* of each *cluster*.

[5]. ***Step-5***: Repeat the ***3rd steps***, which means *reassign* each *datapoint* to the ***New Closest Centroid*** of each ***CLUSTER***.

[6]. ***Step-6:*** If any ***reassignment*** occurs, then go to ***step-4*** else go to ***FINISH***.

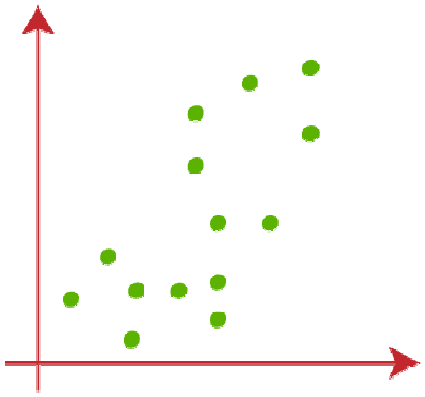
[7]. ***Step-7:*** The model is ready.

4.1.2 K-Means Clustering Example

Now we do these steps manually to Understand the whole process:

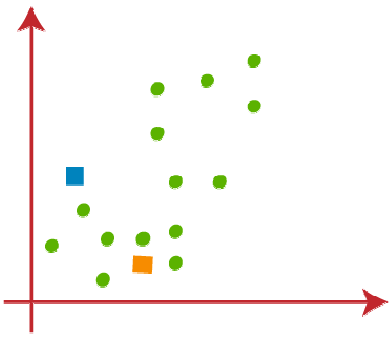
❒ This is our *scatterplot*. We can't just visually *identify* the final *clusters*, although it is *two-dimension* but it's pretty tough.

- ☞ Now imagine how complex a situation would be if we had **three** or **more variables** !! We wouldn't even be able to plot a **five dimensional scatterplot** like that.
- ☝ So that's where it came in is clustering comes into play and that's where this algorithm will help us simplify the process.
- ☯ In this case we're actually going to manually perform the same k-means clustering algorithm (later we will do these in python/R).

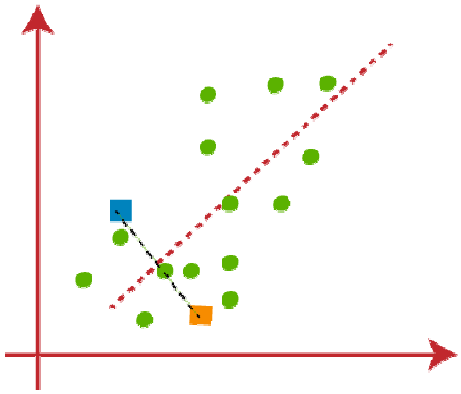


[1] Let's take number ***k*** of clusters, i.e., ***K=2***, to identify the dataset and to put them into different clusters. It means here we will try to group these datasets into ***two*** different ***clusters***.

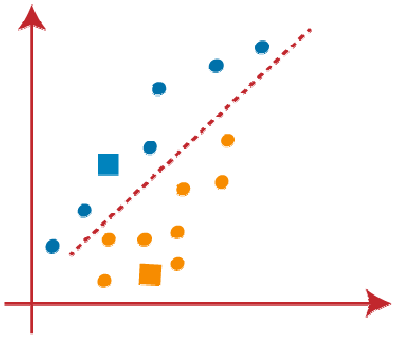
- ☞ We need to choose some ***random k points*** or ***centroid*** to form the cluster.
- ☞ These points can be either the ***points*** from the ***dataset*** or ***any*** other ***point***. So, here we are selecting the two points as k points (in the right-side figure), which are not the part of our dataset. Consider the right-side image:



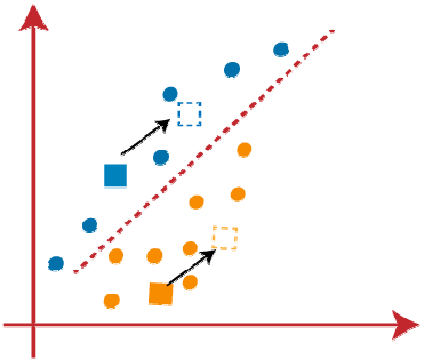
[2] Now we will ***assign*** each ***data point*** of the ***scatter plot*** to its closest ***K-point*** or ***centroid***. We will compute it by applying some ***mathematics*** that we have studied to calculate the distance between two points. So, we will ***draw*** a ***median*** between both the



☞ From the above image, it is clear that points ***left*** side of the ***line*** is near to the ***K1*** or ***blue centroid***, and points to the ***right*** of the line are close to the ***yellow centroid***. Let's color them as ***blue*** and ***yellow*** for clear visualization.

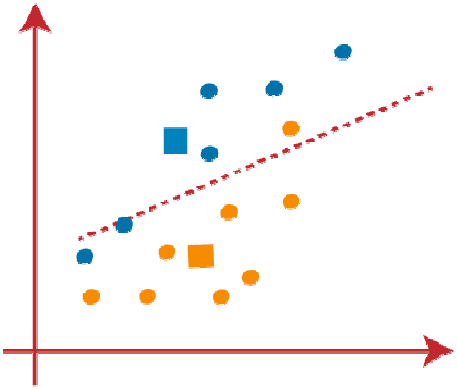


[3] As we need to **find** the **closest cluster**, so we will repeat the process by choosing a **new centroid**. To choose the **new centroids**, we will compute the ***center of gravity*** of these ***centroids***, and will find *new centroids* as below:

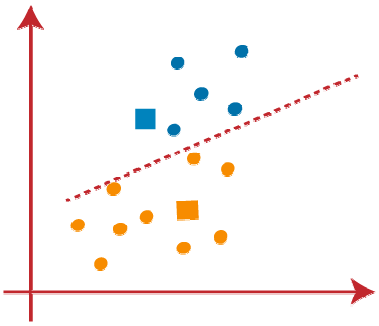


[4] Next, we will ***reassign*** each ***datapoint*** to the ***new centroid***. For this, we will ***repeat*** the ***same process*** of finding a ***median line***. The median will be like right - side image:

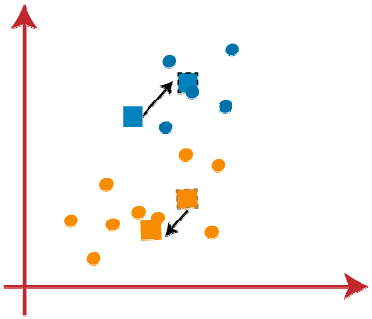
☞ From the beside image, we can see, one yellow point is on the left side of the line, and two blue points are right to the line. So, these three points will be assigned to new centroids.



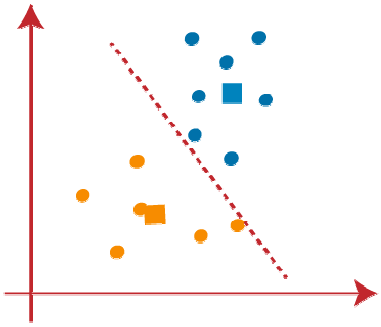
[5] As reassignment has taken place, so we will again go to the ***step-4***, which is finding ***new centroids*** or ***K-points***.



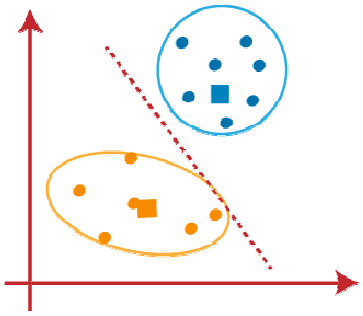
[6] We will repeat the process by finding the ***center*** of ***gravity*** of ***centroids***, so the ***new centroids*** will be as shown in the right-side image:



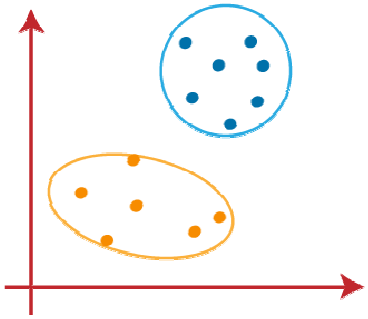
[7] As we got the ***new centroids*** so again will ***draw*** the ***median line*** and ***reassign*** the ***data points***. So, the image will be:



[8] We can see in the above image; there are ***no dissimilar*** data points on ***either side*** of the ***line***, which means our ***model is formed***. Consider the below image:

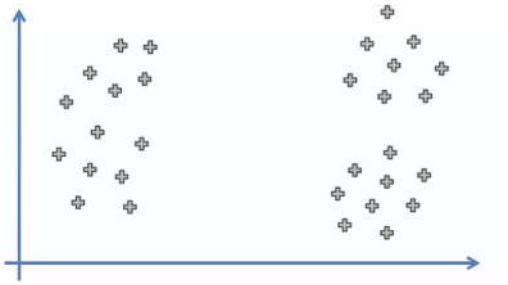


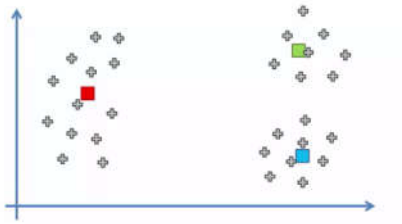
[9] As our model is ready, so we can now ***remove*** the ***assumed centroids***, and the two final clusters will be as shown in the below image:

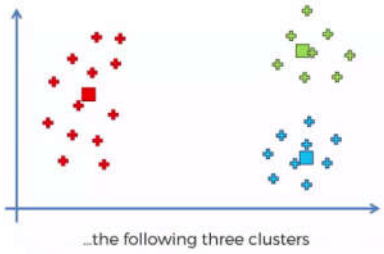


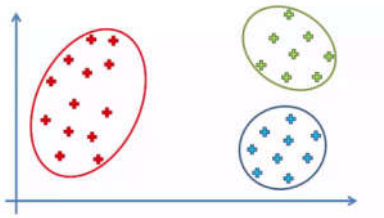
4.1.3 K-Means Random Initialization Trap

Consider the following ***scatterplot***. We have two variables represented by the ***x*** and ***y coordinates***. Let's say we're going to choose three clusters. It does look like you can pretty easily spot them here.



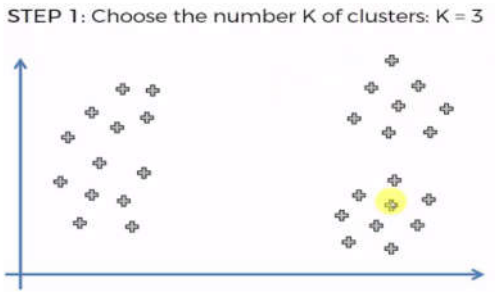


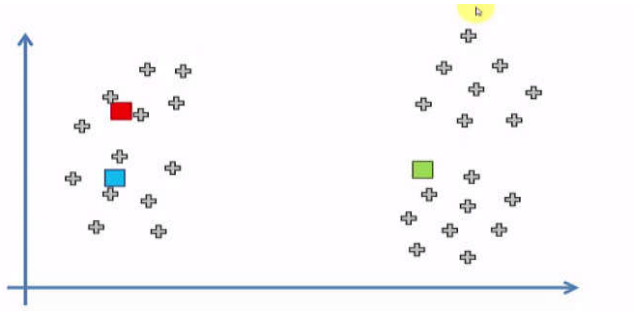




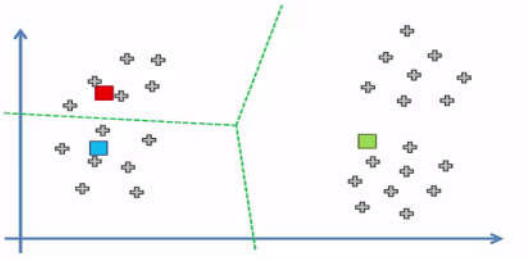
☞ So this is the end result if we choose the ***correct random initialization*** at ***correct location***.

❒ However, if we select a ***centroid*** in ***different locations*** we will end up with different result. By following the steps of the algorithm we will end up as follows:

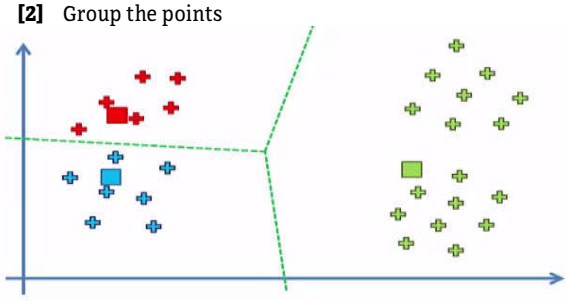




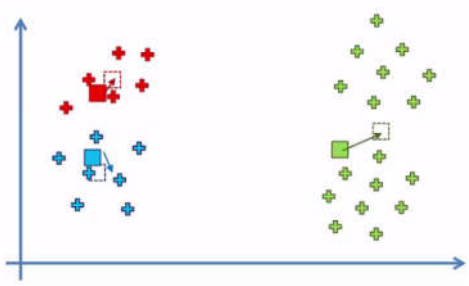
[1] Draw the median lines



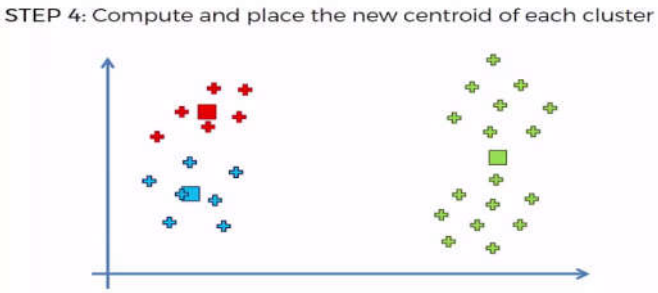
[2] Group the points



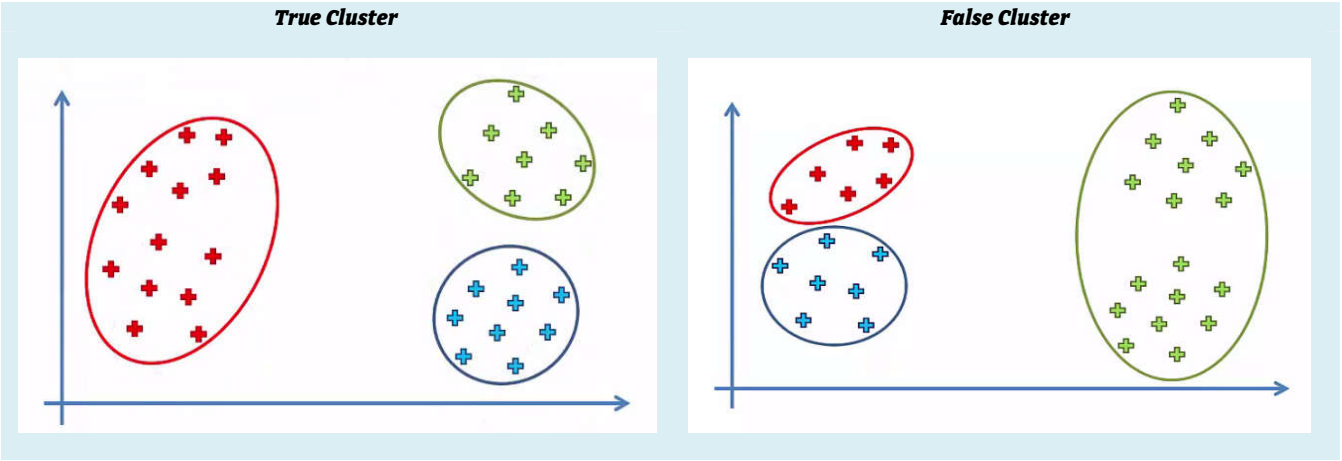
[3] Find new centroids



[4] Then assign the centroids to the new positions:



☞ ***Wrong End result:*** Now if we draw the median lines, we can see there are ***no dissimilar*** data points. So this is the our end point. And it is the final cluster result.



❒ So you can see that the three clusters are different and therefore, the selection of the Centroid is at the very start of the algorithm can potentially dictate the outcome of the algorithm. And that's not a good thing because the centroid are selected at random.

☞ ***K-means++:*** There is a ***modification*** to the ***K-means algorithm*** that allows you to correctly select the Centroid called the ***K-means++*** algorithm.

☝ ***Python Scikit-learn library can handle this automatically:*** From above we can see that it is obviously a trap. However we not need to worry about this because, python implementation of this k-means algorithm can automatically handle this kind of "trap" situation. The good news is that ***K-means++*** algorithm happens in background in either in R or Python or whatever tool you're using. You don't need to actually implement it.

4.1.4 Elbow-Method: Choosing K (Right Number Of Clusters)

The performance of the ***K-means clustering algorithm*** depends upon highly efficient clusters that it forms. But ***choosing the optimal number of clusters*** is a big task. There are some ***different ways*** to find the ***optimal number of clusters***, but here we are discussing the most appropriate method to find the number of clusters or value of K. The method is given below:

❒ ***Elbow Method:*** The Elbow method is one of the most popular ways to find the **optimal number** of **clusters**. This method uses the concept of ***WCSS*** value. ***WCSS*** stands for ***Within Cluster Sum of Squares***, which defines the ***total variations*** within a **cluster**. The formula to calculate the value of ***WCSS (for 3 clusters)*** is given below:

$$WCSS = \sum_{P_i \, in \atop Cluster\ 1} distance(P_i, C_1)^2 + \sum_{P_i \, in \atop Cluster\ 2} distance(P_i, C_2)^2 + m \sum_{P_i \, in \atop Cluster\ 3} distance(P_i, C_3)^2$$

Where P_i are the data-points in corresponding **Clusters** and C_1, C_2, C_3 are the centroids.

☞ In the above formula of WCSS, Following is the ***sum of the square of the distances*** between each ***data point*** and its ***centroid*** within a ***Cluster1*** and the same for the other two terms.

$$\sum_{P_i \, in \atop Cluster\ 1} distance(P_i, C_1)^2$$

☝ To measure the ***distance*** between data ***points*** and ***centroid***, we can use any method such as ***Euclidean distance*** or ***Manhattan distance***.

❒ To find the optimal value of clusters, the elbow method follows the below steps:

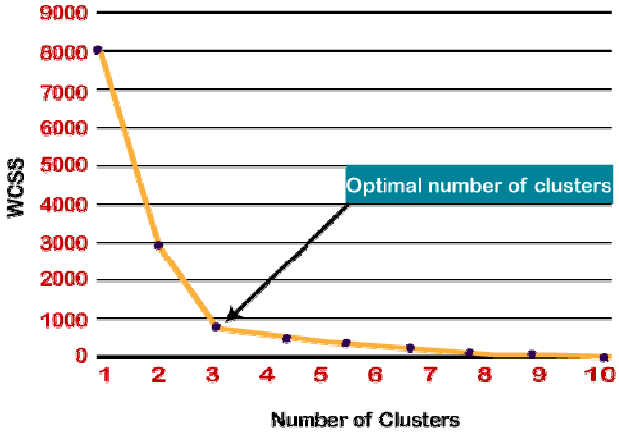
[1]. It executes the ***K-means*** clustering on a given dataset for different ***K values*** (ranges from 1-10).

[2]. For ***each*** value of ***K***, calculates the ***WCSS*** value.

[3]. Plots a ***curve*** between calculated ***WCSS*** values and the ***number*** of clusters ***K***.

[4]. The ***sharp point*** of ***bend*** or a ***point*** of the ***plot*** looks like an ***arm***, then that point is considered as the ***best*** value of ***K***.

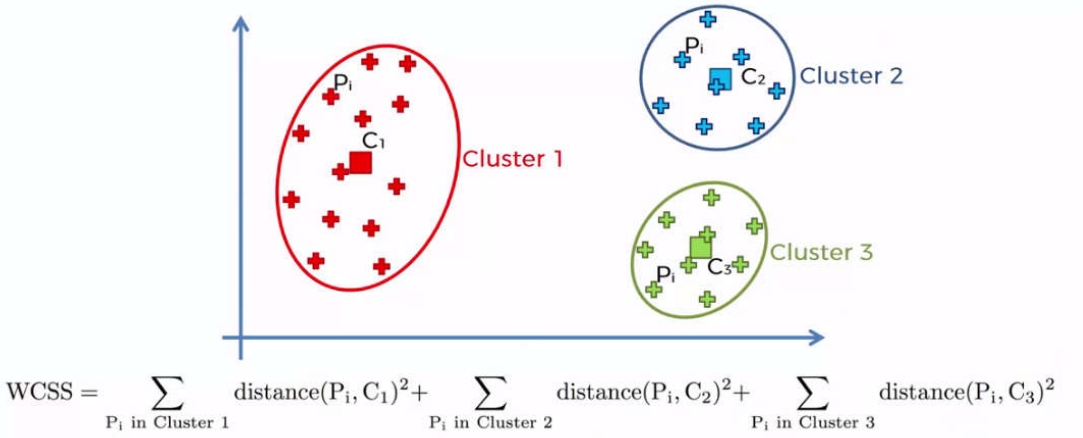
- Since the graph shows the sharp bend, which looks like an elbow, hence it is known as the elbow method. The graph for the elbow method looks like the right side image:



- ***Maximum number of Clusters:*** We can choose the number of clusters equal to the given data points. If we choose the number of clusters equal to the data points, then the value of WCSS becomes zero, and that will be the endpoint of the plot.

  ☞ Actually we can set the K for which WCSS decrease rapidly and after that certain number the WCSS won't change rapidly.

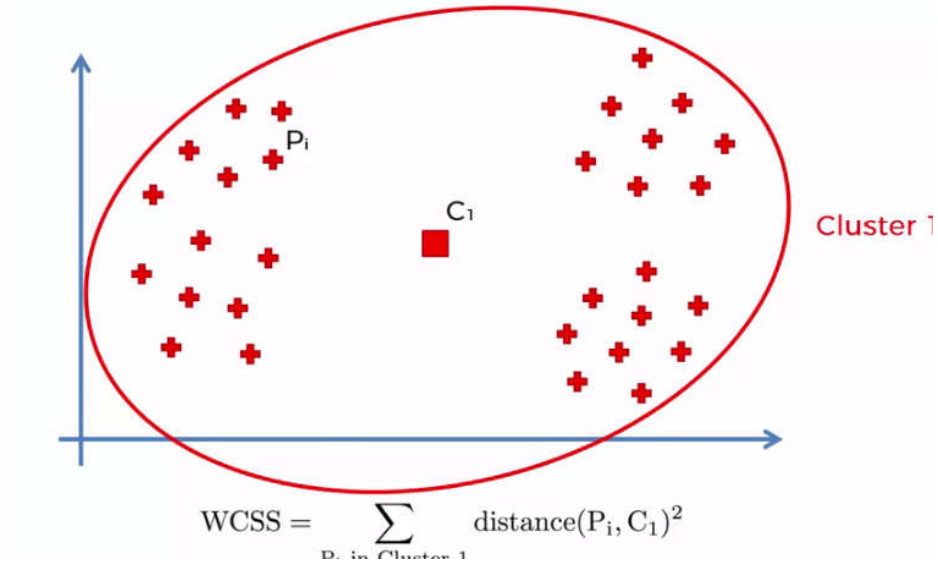
- ***WCSS value is a metric:*** We need a certain metric so that we can understand or evaluate how a certain number of clusters performs compared to a different number of clusters (and preferably that metric should be quantifiable).



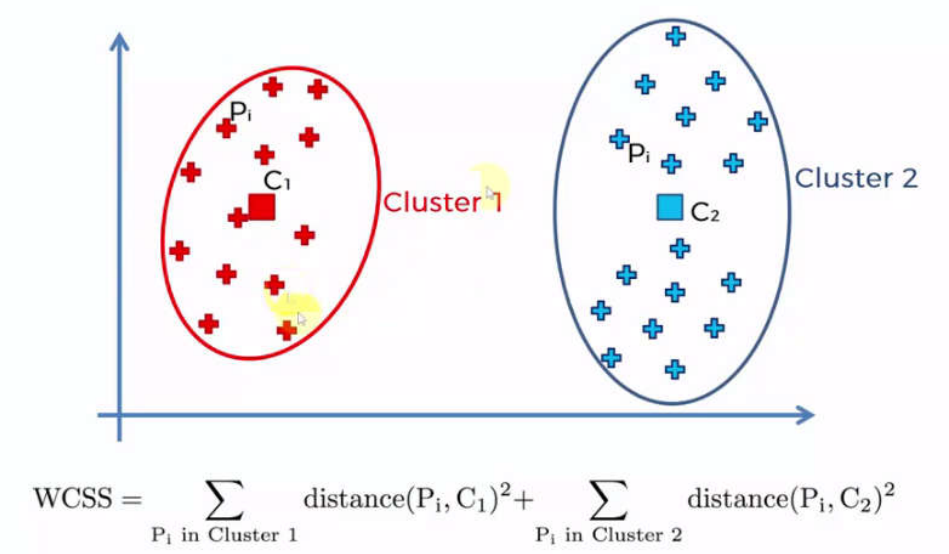
  ☞ Actually it's a quite a good metric in terms of understanding or comparing the goodness of fit between two different K-means clusterings.

- Let's see how that metric ***WCSS*** is going to change as we ***increase*** the ***number*** of ***clusters***.

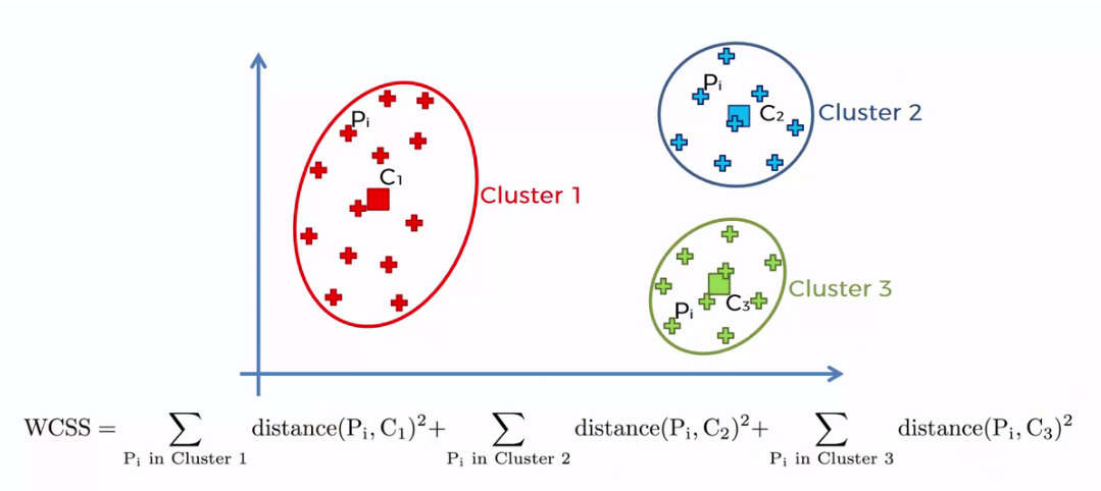
  ☞ If we use one centroid in the middle, the distance from each data point is big and the squared distance will be very large.



☞ If we have 2 clusters then the squared distance reduces.



☞ For 3 clusters we get more reduced ***WCSS***. And for 4, 5, or more ***centroid*** the distance doesn't decrease ***rapidly*** (but it decreases). The WCSS tends to ***0*** as ***K*** reaches to ***number of data points***.



☝ So the ***Elbow-method*** is just an approach that can help you to decide the number of k (number of clusters). But at the end of day it is your decision.

4.1.5 Python Implementation of K-means Clustering

Problem Description: We have a dataset of ***Mall_Customers***, which is the data of customers who visit the mall and spend there.

In the given dataset, we have ***Customer_Id***, ***Gender***, ***Age***, ***Annual Income ($)***, and ***Spending Score*** (which is the calculated value of how much a customer has spent in the mall, the more the value, the more he has spent). From this dataset, we need to calculate some patterns, as it is an unsupervised method, so we don't know what to calculate exactly.

```
# WCSS : Within-Cluster Sum of Square

# K-Means Clustering

# Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# importing data
dataSet = pd.read_csv("Mall_Customers.csv")
X = dataSet.iloc[:, [3, 4]].values
```

```
# There is no y for clustering


# Finding optimal noumber of clusters using elbow method : WCSS
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    # setting paramter for cluster generator
    cluster_generator_elbow = KMeans(n_clusters = i, init="k-means++", random_state=0, max_iter=300, n_init=10)
    cluster_generator_elbow.fit(X) # fit the independent data
    wcss.append(cluster_generator_elbow.inertia_) # capturing wccs data for each i
    # inertia_ :  Sum of squared distances of samples to their closest cluster center. Is actually "wcss"

# visualizing the elbow digram ith clusters vs wcss
# range(1, 11), wcss: both are lists
plt.plot(range(1, 11), wcss)
plt.title("The elbow Method")
plt.xlabel("Number of Clusters")
plt.ylabel("WCSS")
plt.show()

# creating cluster with optimal "n_clusters". From elbow-plot we figured out that 5 is the optimal number of clusters
k_mean_cluster_genrt = KMeans(n_clusters = 5, init="k-means++", random_state=0, max_iter=300, n_init=10)
y_k_mean_cluster = k_mean_cluster_genrt.fit_predict(X)

# plotting the cluster
plt.scatter(X[y_k_mean_cluster == 0, 0], X[y_k_mean_cluster == 0, 1], s = 100, c = "red", label="cluster 1")
plt.scatter(X[y_k_mean_cluster == 1, 0], X[y_k_mean_cluster == 1, 1], s = 100, c = "blue", label="cluster 2")
plt.scatter(X[y_k_mean_cluster == 2, 0], X[y_k_mean_cluster == 2, 1], s = 100, c = "green", label="cluster 3")
plt.scatter(X[y_k_mean_cluster == 3, 0], X[y_k_mean_cluster == 3, 1], s = 100, c = "cyan", label="cluster 4")
plt.scatter(X[y_k_mean_cluster == 4, 0], X[y_k_mean_cluster == 4, 1], s = 100, c = "pink", label="cluster 5")

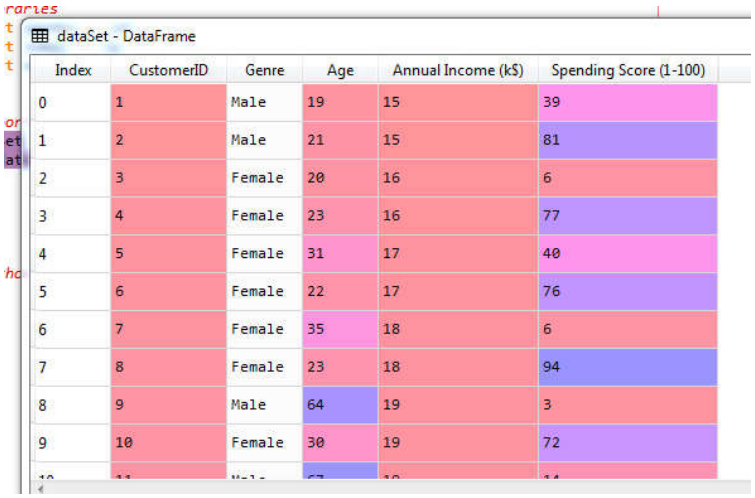
# centroids
plt.scatter(k_mean_cluster_genrt.cluster_centers_[:, 0], k_mean_cluster_genrt.cluster_centers_[:, 1], s=300, c="black"
, label = "Centroids")
plt.title("Clusters Of Clients")
plt.xlabel("Annual income ($)")
plt.ylabel("Spending score (1-100)")
plt.legend()
plt.show()


# python prctc_k_mns.py
```

❒ ***Data preprocessing:*** There is ***no y dependent variable*** in clustering. Here we don't need any dependent variable for data pre-processing step as it is a ***clustering problem***, and we have ***no idea about what to determine***. So we will just add a line of code for the matrix of features.

```
x = dataset.iloc[:, [3, 4]].values
```

☞ As we can see, we are extracting only ***4th and 5th feature-column:*** *Annual Income* and *Spending Score*. It is because we need a ***2D plot to visualize the model***, and some features are not required, such as customer_id, Gender. The dataset is



dataSet - DataFrame

Index	CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40
5	6	Female	22	17	76
6	7	Female	35	18	6
7	8	Female	23	18	94
8	9	Male	64	19	3
9	10	Female	30	19	72

☞ ***Selecting Library and Class:***

```
from sklearn.cluster import KMeans
```

❒ ***The Elbow technique:*** Inside a `for` loop we create the object for ***1*** to ***10 number of clusters*** and we calculate ***WCSS*** for each of them. After creating the ***list*** of ***WCSS*** values we plot the ***Elbow diagram***. We then select the ***optimal number of clusters***.

```
# Finding optimal noumber of clusters using elbow method : WCSS
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    # setting paramter for cluster generator
    cluster_generator_elbow = KMeans(n_clusters = i, init="k-means++", random_state=0, max_iter=300, n_init=10)
    cluster_generator_elbow.fit(X) # fit the independent data
    wcss.append(cluster_generator_elbow.inertia_) # capturing wccs data for each i
    # inertia_ :  Sum of squared distances of samples to their closest cluster center. Is actually "wcss"

# visualizing the elbow digram ith clusters vs wcss
# range(1, 11), wcss: both are lists
plt.plot(range(1, 11), wcss)
plt.title("The elbow Method")
plt.xlabel("Number of Clusters")
plt.ylabel("WCSS")
plt.show()
```

☞ ***We create the object using:***

```
KMeans(n_clusters = i, init="k-means++", random_state=0, max_iter=300, n_init=10)
```

➢ ***n_clusters*** : The number of clusters to form as well as the number of centroids to generate. For Elbow method it is a variable. For final cluster we will use the fixed optimal number of cluster.

➢ ***init = 'k-means++'*** is a Method for initialization. '***k-means++***' : selects initial cluster centers for ***k-mean clustering*** in a smart way to ***speed*** up ***convergence***.

➢ ***n_init*** : Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of ***inertia***.

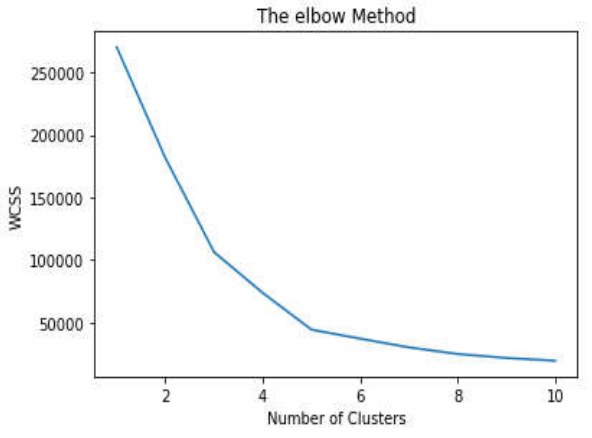
➢ ***max_iter***: Maximum number of iterations of the k-means algorithm for a single run.

☞ After fitting the data we extract the WCSS value using:

`cluster_generator_elbow.inertia_`

➢ **`inertia_`** : Sum of squared distances of samples to their closest cluster center. Is actually "wcss"

☞ ***Optimal Number Of Cluster:*** From the Elbow diagram we notice that, if the number of cluster is more than 5 then WCSS doesn't change rapidly. So 5 is the optimal number of cluster.



☞ From the ***above plot***, we can see the ***elbow point*** is at ***5***. So the ***number of clusters*** here will be ***5***.

wcss_list - List (10 elements)

Index	Type	Size	Value
0	float64	1	269981.28
1	float64	1	181363.59595959596
2	float64	1	106348.37306211118
3	float64	1	73679.78903948834
4	float64	1	44448.45544793371
5	float64	1	37233.81451071001
6	float64	1	30259.65720728547
7	float64	1	25011.83934915659
8	float64	1	21850.165282585633
9	float64	1	19672.07284901432

Save and Close | Close

❒ ***Creating cluster with optimal "n_clusters":*** Finally we analyze the data for 5 clusters. And we use ***fit_predict()*** to find the **y-values**.

☞ As we have got the number of clusters, so we can now ***train the model*** on the dataset.

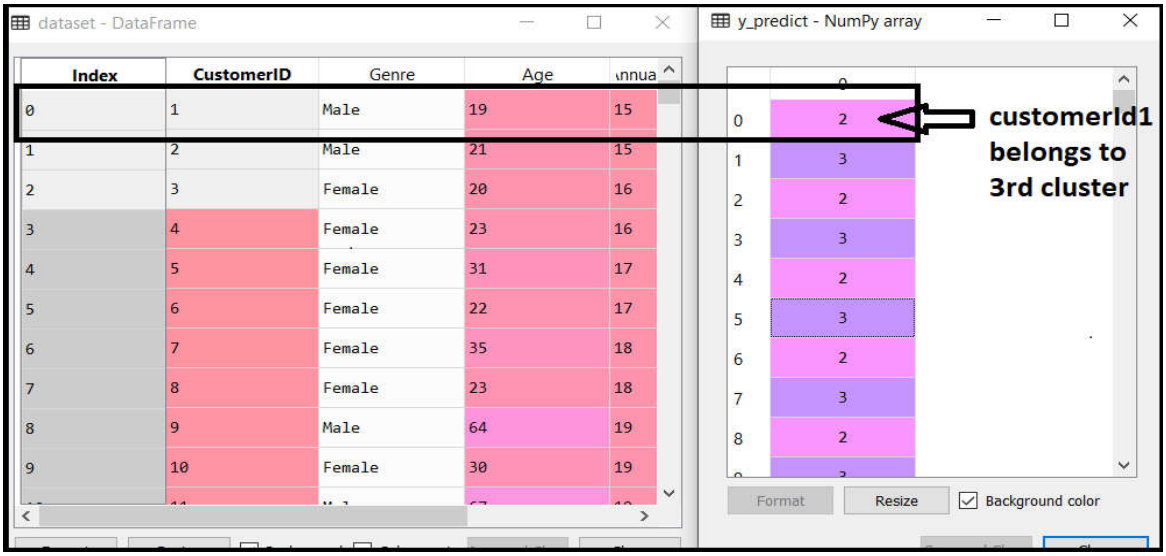
☞ To train the model, we will use the same two lines of code as we have used in the above section, but here instead of using ***i***, we will use ***5***, as we know there are ***5 clusters*** that need to be formed.

```
# creating cluster with optimal "n_clusters". From elbow-plot we got 5 is the optimal number of clusters
k_mean_cluster_genrt = KMeans(n_clusters = 5, init="k-means++", random_state=0, max_iter=300, n_init=10)
y_k_mean_cluster = k_mean_cluster_genrt.fit_predict(X)
```

➢ The first line is the same as above for creating the object of ***KMeans class***.

➢ In the second line of code, we have created the dependent variable ***y_k_mean_cluster*** (we can call it ***y_predict*** also) to train the model.

➢ By executing the above lines of code, we will get the ***y_k_mean_cluster*** variable. We can check it under the variable explorer option in the Spyder IDE. We can now compare the values of ***y_k_mean_cluster*** with our original dataset. Consider the below image:

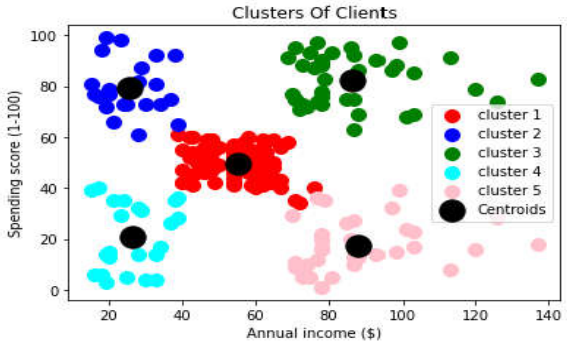


➢ From the above image, we can now relate that the ***CustomerID 1*** belongs to a ***cluster 3***(as index starts from ***0***, hence ***2*** will be considered as ***3***), and ***2*** belongs to ***cluster 4***, and so on

❒ ***Visualizing the Clusters:*** In following lines of code, we have written code for each clusters, ranging from 1 to 5. The first coordinate of the ***plt.scatter***, i.e., ***X[y_k_mean_cluster == 0, 0]*** containing the ***x*** value for the showing the matrix of features values, and the ***y_k_mean_cluster*** is ranging from ***0*** to ***1***.

```
# plotting the cluster
plt.scatter(X[y_k_mean_cluster == 0, 0], X[y_k_mean_cluster == 0, 1], s = 100, c = "red", label="cluster 1")
plt.scatter(X[y_k_mean_cluster == 1, 0], X[y_k_mean_cluster == 1, 1], s = 100, c = "blue", label="cluster 2")
plt.scatter(X[y_k_mean_cluster == 2, 0], X[y_k_mean_cluster == 2, 1], s = 100, c = "green", label="cluster 3")
plt.scatter(X[y_k_mean_cluster == 3, 0], X[y_k_mean_cluster == 3, 1], s = 100, c = "cyan", label="cluster 4")
plt.scatter(X[y_k_mean_cluster == 4, 0], X[y_k_mean_cluster == 4, 1], s = 100, c = "pink", label="cluster 5")

# centroids
plt.scatter(k_mean_cluster_genrt.cluster_centers_[:, 0], k_mean_cluster_genrt.cluster_centers_[:, 1], s=300, c="black"
, label = "Centroids")
plt.title("Clusters Of Clients")
plt.xlabel("Annual income ($)")
plt.ylabel("Spending score (1-100)")
plt.legend()
plt.show()
```



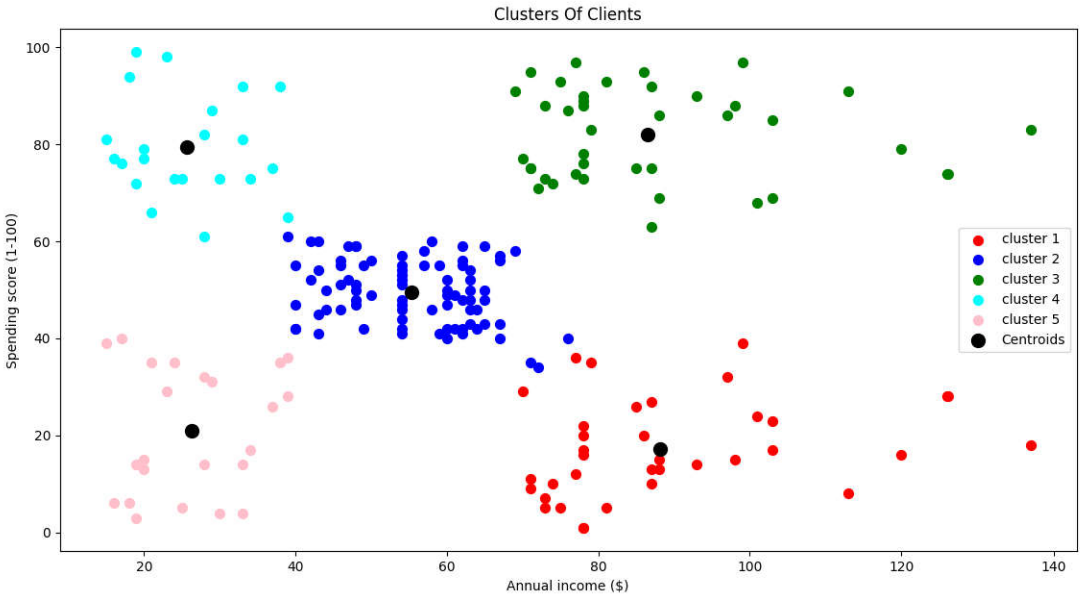
☝ ***X[y_k_mean_cluster == 0, 0]***

☞ ***y_k_mean_cluster == 0*** looks for the matching elements with 1st cluster in the feature matrix ***X***. And the second ***0*** means the 1st column of X, i.e ***Annual income***.

☝ Similarly ***X[y_k_mean_cluster == 0, 1]***

☞ Retrieves all matching data points with 1st cluster and

❒ ***Categorize the customers:*** The output image is clearly showing the ***five different clusters*** with different colors. The clusters are formed between two parameters of the dataset; ***Annual income*** of customer and ***Spending***. We can change the colors and labels as per the requirement or choice. We can also observe some points from the above patterns, which are given below:



[1]. ***Cluster1*** shows the customer has a high income but low spending, so we can categorize them as ***Careful***

[2]. ***Cluster2*** shows the customers with average salary and average spending so we can categorize these customers as ***Standard***

[3]. ***Cluster3*** shows the customers with high income and high spending so they can be categorized as ***Target***, and these customers can be the most profitable customers for the mall owner.

[4]. ***Cluster4*** shows the customers with low income with very high spending so they can be categorized as ***Careless***.

[5]. ***Cluster5*** shows the low income and also low spending so they can be categorized as ***Sensible***.

❒ ***Multi-Dimensional Clustering:*** For 3 feature variable we still can visualize the cluster in 3-D graphics, but more tan 3D we can't plot the clusters.

☞ However later we will learn a technique that allows us to reduce the dimensions of our data. So that you can plot the clusters.

☝ If you are doing clustering in more than two dimensions then don't execute the last code lines in this section to visualize the clusters. Because it's only for two dimensional clustering.

Hierarchical Clustering

4.2.1 Hierarchical Clustering

Hierarchical clustering is another **unsupervised** machine learning algorithm, which is used to ***group*** the ***unlabeled datasets*** into a ***cluster*** and also known as ***hierarchical cluster analysis*** or ***HCA***.

- ***Dendrogram:*** In this algorithm, we develop the hierarchy of clusters in the form of a tree, and this tree-shaped structure is known as the dendrogram.

- Sometimes the results of ***K-Means Clustering*** and ***Hierarchical Clustering*** may look **similar**, but they both differ depending on how they work.
- There is no **requirement** to **predetermine** the ***number of clusters*** as we did in the ***K-Means algorithm***.

- ***Agglomerative and Divisive clustering:*** The hierarchical clustering technique has two approaches:
  - ***Agglomerative***: Agglomerative is a ***bottom-up approach***, in which the algorithm starts with ***taking all data points as single clusters*** and ***merging*** them ***until one cluster is left***.
  - ***Divisive:*** Divisive algorithm is the ***reverse*** of the ***agglomerative algorithm*** as it is a ***top-down approach***.

- ***Why hierarchical clustering?:*** As we already have other clustering algorithms such as ***K-Means Clustering***, then why we need hierarchical clustering?
  - As we have seen in the ***K-means clustering*** that we need a ***predetermined number*** of ***clusters***, and
  - It always tries to ***create*** the ***clusters*** of the ***same size***.

  - To solve these two challenges, we can opt for the ***hierarchical clustering algorithm*** because, in this algorithm, we ***don't need*** to have ***knowledge*** about the ***predefined number of clusters***.

4.2.2 Agglomerative Hierarchical clustering

The ***Agglomerative Hierarchical Clustering*** algorithm is a popular example of ***HCA***. To group the datasets into clusters, it follows the ***Bottom-Up*** approach. It means, this algorithm considers ***each dat-point as a single cluster*** at the **beginning**, and then start ***COMBINING*** the ***closest pair*** of ***clusters*** together. It ***does*** this ***until*** all the clusters are ***merged into a single cluster*** that contains ***all*** the ***datasets***.

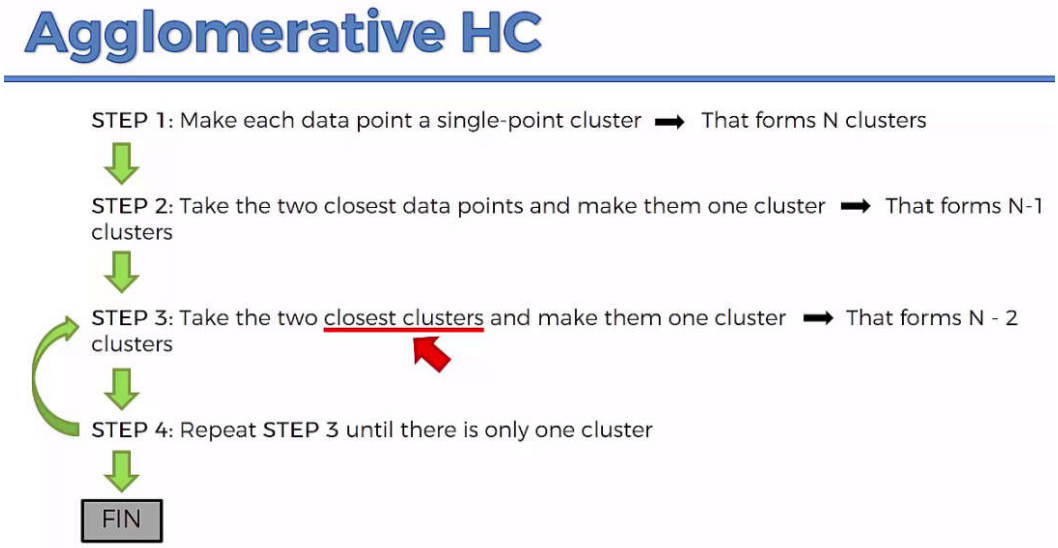
- This ***Hierarchy Of Clusters*** is represented in the form of the ***DENDROGRAM***.

Clustering Pros & Cons

Clustering

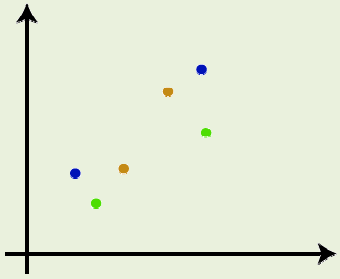
Clustering Model	Pros	Cons
K-Means	Simple to understand, easily adaptable, works well on small or large datasets, fast, efficient and performant	Need to choose the number of clusters
Hierarchical Clustering	The optimal number of clusters can be obtained by the model itself, practical visualisation with the dendrogram	Not appropriate for large datasets

❒ ***How Agglomerative HCA Work:*** The working of the ***AHC*** algorithm can be explained using the below steps:

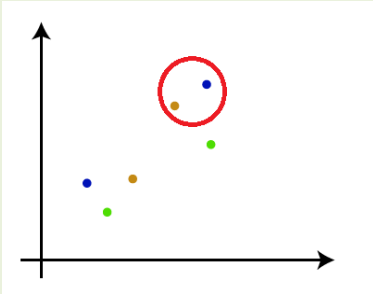


❒ The way the hierarchical clustering algorithm works is that it ***maintains a memory*** of how we went through following process and that ***memory*** is ***stored*** in a ***DENDROGRAM***.

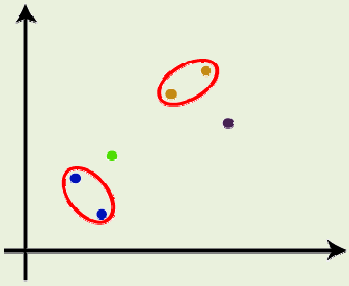
- ***Step-1:*** Create ***each data point*** as a ***single cluster***. Let's say there are ***N data points***, so the number of ***clusters*** will also be ***N***.



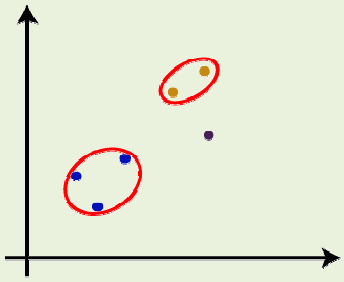
- ***Step-2:*** Take ***two*** closest ***data-points*** or ***clusters*** and ***merge them*** to form one cluster. So, there will now be N-1 clusters.

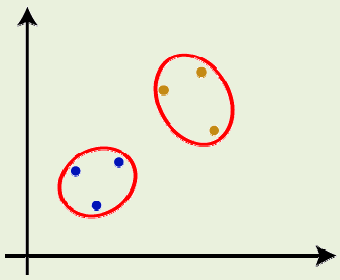


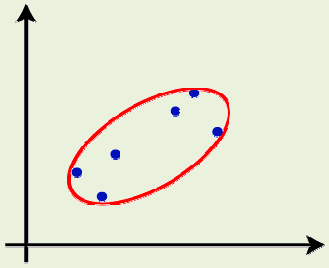
- ***Step-3:*** Again, take the ***two closest clusters*** and ***merge*** them together to form one cluster. There will be ***N-2*** clusters.



- ***Step-4:*** Repeat **Step 3** until only ***one cluster*** left. So, we will get the following clusters. Consider the below images:



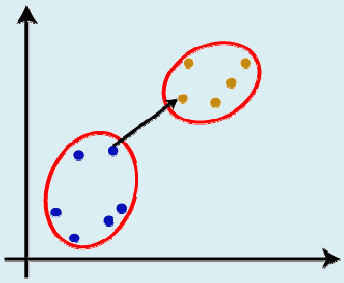




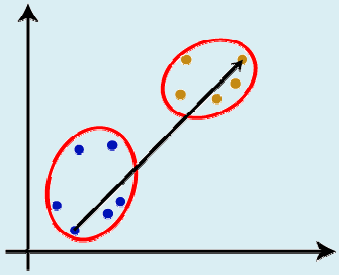
4.2.3 Distance between two clusters

❒ ***Linkage Methods:*** The ***closest distance*** between the two clusters is ***crucial*** for the ***hierarchical clustering***. There are **various ways** to calculate the ***distance*** between ***two clusters***, and these ways decide the rule for clustering. These measures are called ***Linkage methods***. Some of the popular linkage methods are given below:

[1] ***Single Linkage:*** It is the *Shortest Distance* between the *closest points* of the *clusters*. Consider the beside image:

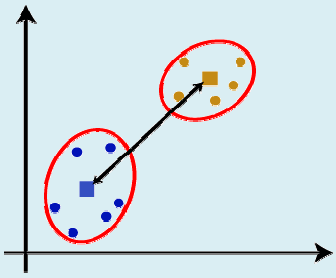


[2] ***Complete Linkage:*** It is the *farthest distance* between the *two points* of two different *clusters*. It is one of the popular linkage methods as it forms ***tighter clusters*** than single-linkage.



[3] ***Average Linkage:*** It is the linkage method in which the *distance between each pair* of ***data-points*** is added up and then divided by the ***total number*** of ***data-points*** to calculate the average distance between two clusters. It is also one of the most **popular linkage methods**.

[4] ***Centroid Linkage:*** It is the linkage method in which the ***distance between*** the ***centroid*** of the ***clusters*** is calculated. Consider the beside image:

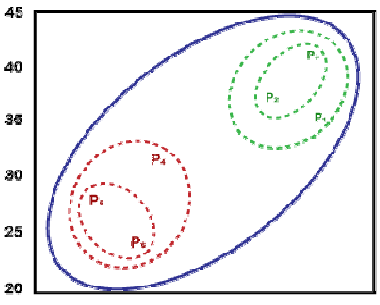


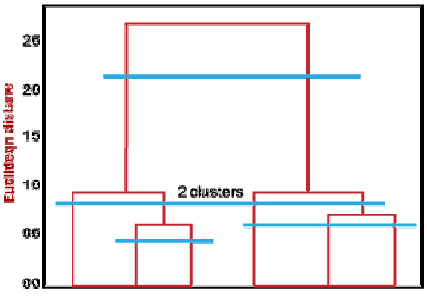
4.2.4 Dendrogram in Hierarchical clustering

The ***Dendrogram*** is a ***Tree-Like Structure*** that is mainly used to *store each step as a memory* that the ***HC algorithm performs***. In the ***dendrogram plot***, the

- ***Y-axis*** shows the *Euclidean distances* between the *data points*, and
- The ***X-axis*** shows *all the data points* of the given dataset.

The working of the dendrogram can be explained using the below diagram:





➢ In the above diagram, the ***left part*** is showing how ***clusters*** are created in ***agglomerative clustering***, and the ***right part*** is showing the ***Corresponding Dendrogram***.

i. Firstly, the datapoints ***P2*** and ***P3*** combine together and form a ***cluster***, correspondingly a ***dendrogram*** is ***created***, which connects ***P2*** and ***P3*** with a ***rectangular shape***. The ***height*** is decided according to the ***Euclidean distance*** between the ***data points***.

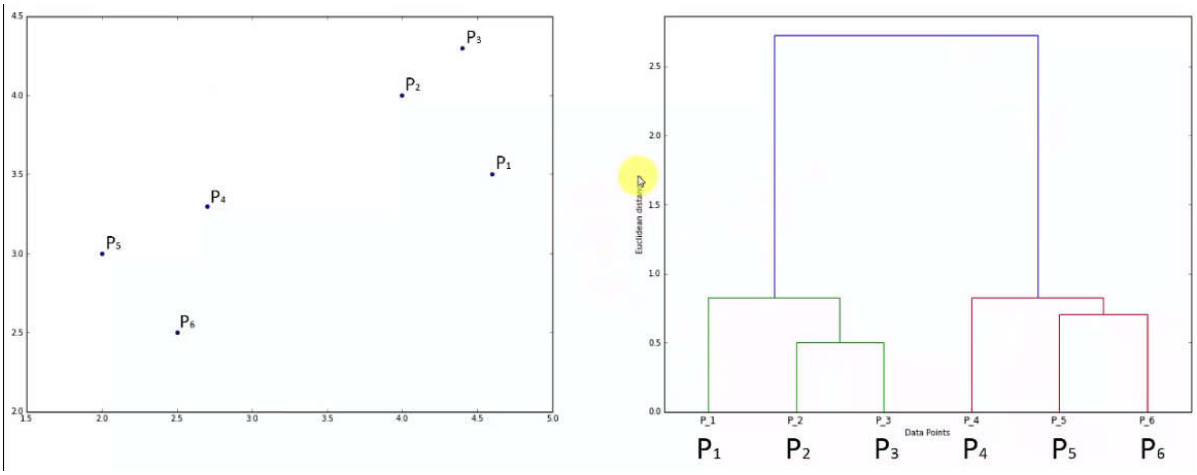
ii. In the next step, ***P5*** and ***P6*** form a ***cluster***, and the corresponding ***dendrogram*** is created. It is **higher** than of **previous**, as the **Euclidean distance** between ***P5*** and ***P6*** is a ***little bit greater*** than the ***P2*** and ***P3***.

iii. Again, **two** new **dendrograms** are created that ***combine P1***, ***P2***, and ***P3*** in ***one dendrogram***, and ***P4***, ***P5***, and ***P6***, in ***another dendrogram***.

iv. At last, the ***final dendrogram*** is created that combines ***all the data points together***.

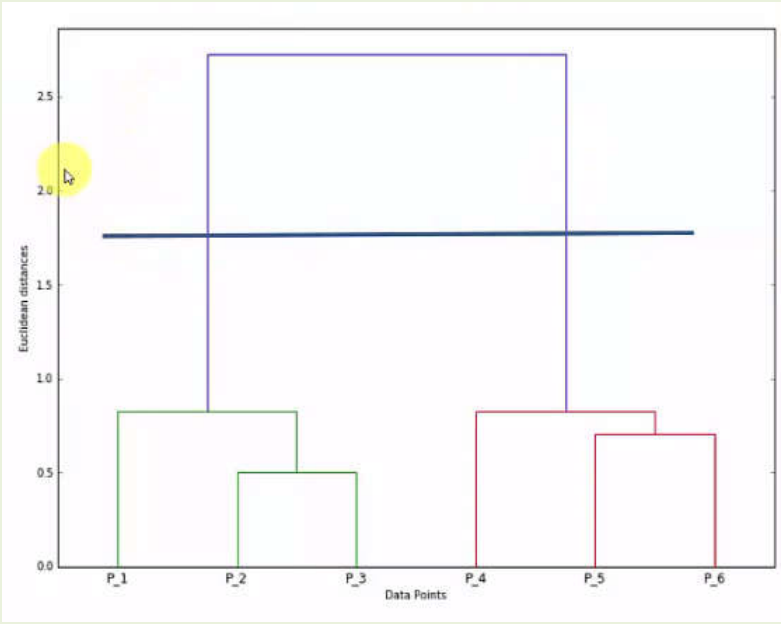
☝ We can ***cut*** the ***dendrogram*** tree structure at **any level** as per our **requirement**.

❒ ***Dendrogram*** contains the ***memory*** of the ***hierarchical clustering algorithm***. You can understand in which order these classes were formed. Just by looking at the ***Dendrogram***. Here I've got an example, this is the actual example generated by computer using HCA.

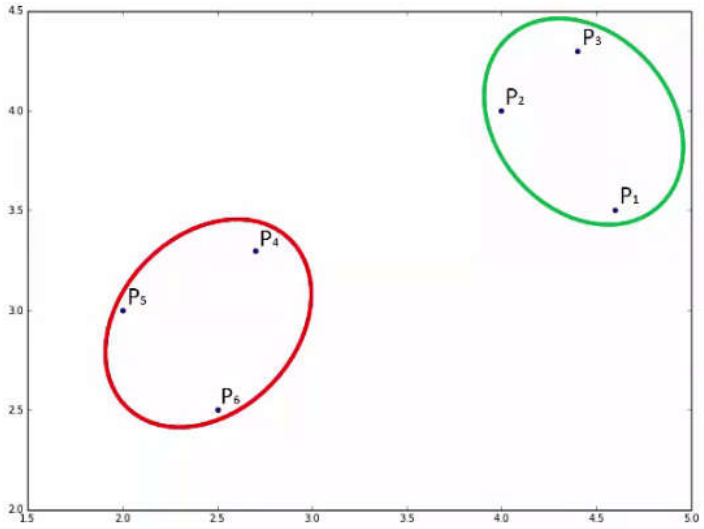


❒ ***How do we get to that right number of clusters:*** Now we have to look at the **horizontal levels** and set **thresholds** so we can set **heights thresholds** or **distance** actually ***distance thresholds*** are also called ***dissimilarity thresholds*** because this vertical axis measures the Euclidean distance between points which also represents the ***dissimilarity*** between ***them*** or ***points*** or ***clusters***.

☞ So what we can do is ***set a threshold*** for all ***dissimilarity*** and we can say that we don't want dissimilarity to be greater than a certain level.



☞ We were setting the dissimilarity threshold by saying that anything if we come across clusters that are above this ***certain threshold*** ***(i.e. above certain Euclidian distance)*** so we ***don't want within a cluster*** to **have dissimilarity above** this **threshold**. Then we end up with two clusters. The clysters are shown in the following figure:

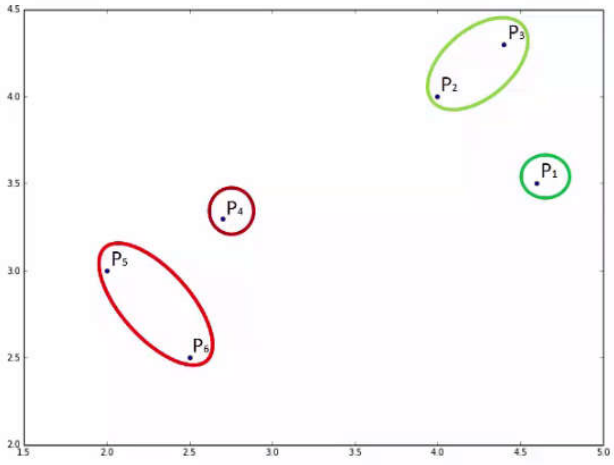


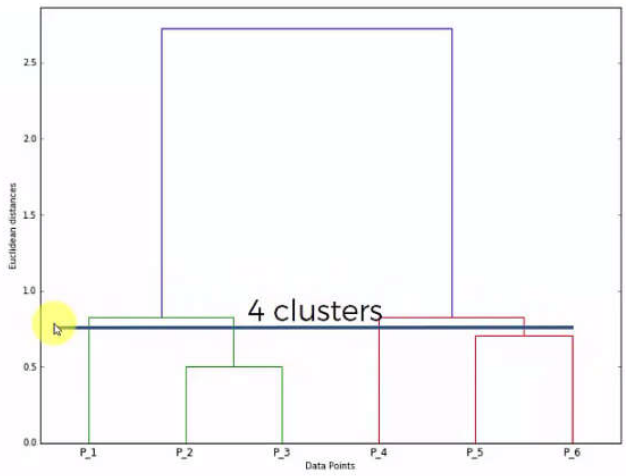
- What this is doing is it is ***not allowing*** any ***`clusters`*** that would have ***`dissimilarity`*** of ***`greater`*** than the ***threshold value*** *(1.7 in the figure)* within them.

❒ ***Calculating clusters numbers:*** You can quickly tell **how many clusters** you will have ***at a certain threshold*** by just looking at ***how many vertical lines*** this ***horizontal threshold*** actually crosses.

- So in above ***Dendrogram figure***, the ***horizontal threshold*** line *crosses two vertical line*, hence we get ***two clusters***.

- In following case we have 4-clusters:





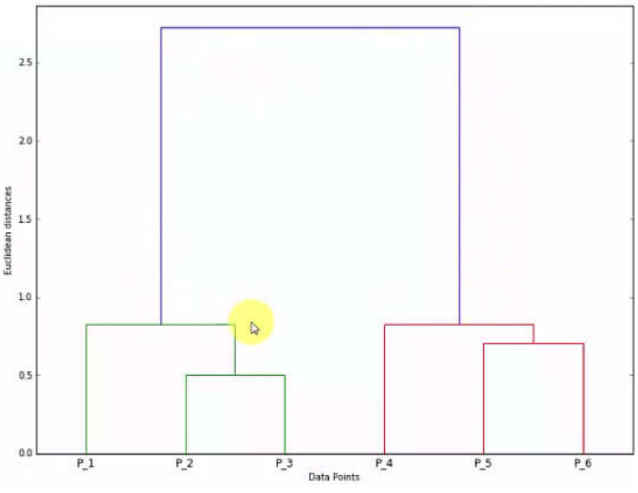
❒ ***Finding Optimal number of Clusters:*** Large vertical line means "Long distance/gap/void between the clusters". So we need to find a ***"threshold" that crosses maximum number "large vertical lines"***.

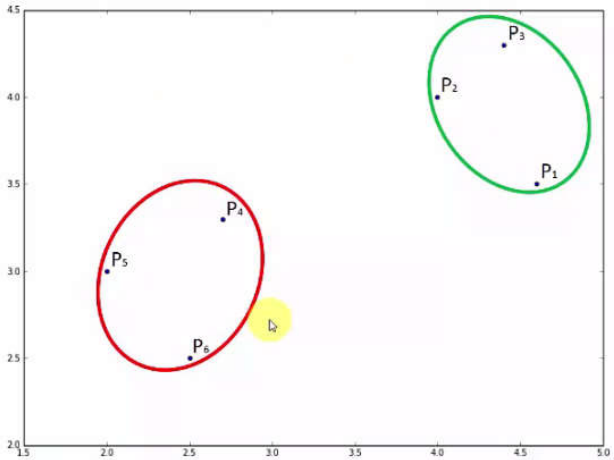
- We cannot consider the vertical lines that crosses "hypothetical extended horizontal line that forms dendrogram".

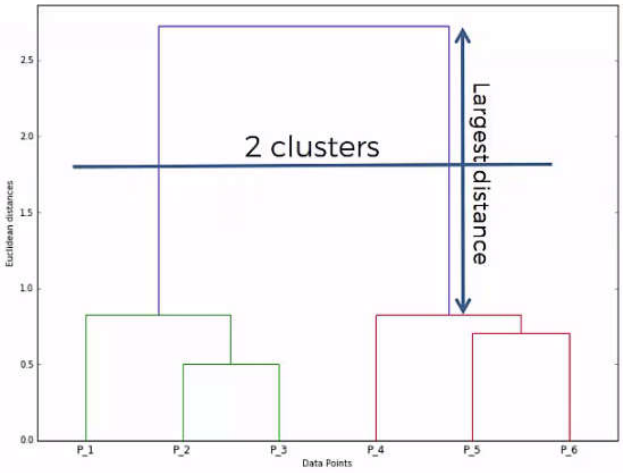
- One of the standard approaches is just to ***look for the highest vertical distance*** that you can find in dendrogram. Any line that ***will not cross any horizontal*** lines.

- For example: line at ***P2*** and ***P3*** can be considered but ***P1*** **cannot** be **considered** because it crosses ***hypothetical horizontal line (extended)*** that joins ***P2*** and ***P3***. Because it does not represents the distance between ***{P2, P3} `cluster`***, it actually distance between the single ***`points` P2***, ***P3***; and the actual distance between ***{P2, P3}*** and ***P1*** is the **short vertical line** (top on P2, P3).

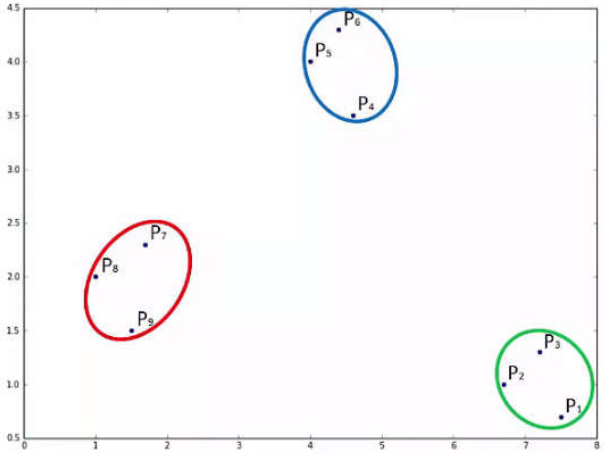
- So we have to find the ***largest vertical line that does not cross any of the existed horizontal lines that forms the dendrogram***.

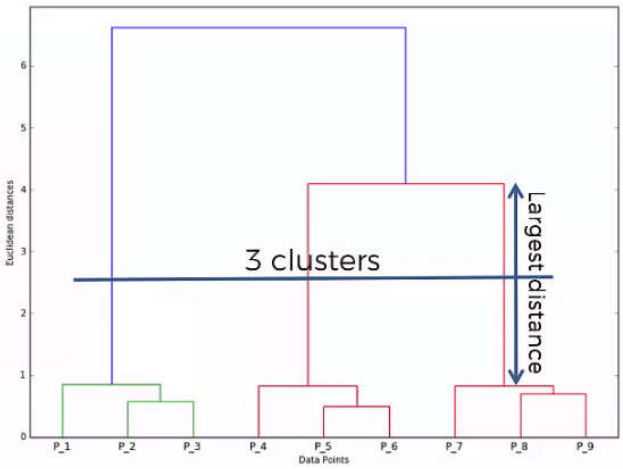






❒ ***One More Example:***





4.2.5 Python Implementation of Agglomerative Hierarchical Clustering

We consider the same problem. The mall customer data.

Problem Description: We have a dataset of ***Mall_Customers***, which is the data of customers who visit the mall and spend there.

- In the given dataset, we have ***Customer_Id***, ***Gender***, ***Age***, ***Annual Income ($)***, and ***Spending Score*** (which is the calculated value of how much a customer has spent in the mall, the more the value, the more he has spent). From this dataset, we need to calculate some patterns, as it is an unsupervised method, so we don't know what to calculate exactly.

```
# Hierachical Clustering

# libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# importing data
dataSet = pd.read_csv("Mall_Customers.csv")
X = dataSet.iloc[:, [3, 4]].values
# There is no y for clustering


# Dendrogram :: Finding optimal noumber of clusters using "Dendrogram"
import scipy.cluster.hierarchy as schr
dendrgm = schr.dendrogram(schr.linkage(X, method="ward"))
plt.title("Dendrogram")
plt.xlabel("Number of Clusters")
plt.ylabel("Euclidean distance")
plt.show()
```

```
# creating cluster with optimal "n_clusters". From "DENDROGRAM" we figured out that 5 is the optimal number of cluster
s
from sklearn.cluster import AgglomerativeClustering
hrcl_cluster_genrt = AgglomerativeClustering(n_clusters = 5, affinity="euclidean", linkage='ward')
y_hrcl = hrcl_cluster_genrt.fit_predict(X)

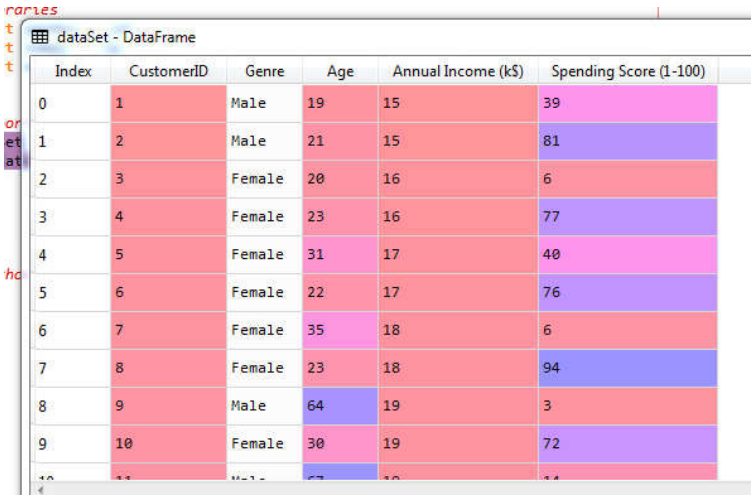
# plotting the cluster
plt.scatter(X[y_hrcl == 0, 0], X[y_hrcl == 0, 1], s = 100, c = "red", label="cluster 1")
plt.scatter(X[y_hrcl == 1, 0], X[y_hrcl == 1, 1], s = 100, c = "blue", label="cluster 2")
plt.scatter(X[y_hrcl == 2, 0], X[y_hrcl == 2, 1], s = 100, c = "green", label="cluster 3")
plt.scatter(X[y_hrcl == 3, 0], X[y_hrcl == 3, 1], s = 100, c = "cyan", label="cluster 4")
plt.scatter(X[y_hrcl == 4, 0], X[y_hrcl == 4, 1], s = 100, c = "pink", label="cluster 5")

# No centroids in Hierarchical clustering
# plt.scatter(hrcl_cluster_genrt.cluster_centers_[:, 0], hrcl_cluster_genrt.cluster_centers_[:, 1], s=300, c="black",
label = "Centroids")
plt.title("Clusters Of Clients")
plt.xlabel("Annual income ($)")
plt.ylabel("Spending score (1-100)")
plt.legend()
plt.show()


# python prctc_hrcl_cltr.py
```

❒ ***Data preprocessing:*** There is ***no y dependent variable*** in clustering.

☞ We are extracting only ***4th and 5th feature-column:*** *Annual Income* and *Spending Score*.



dataSet - DataFrame

Index	CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40
5	6	Female	22	17	76
6	7	Female	35	18	6
7	8	Female	23	18	94
8	9	Male	64	19	3
9	10	Female	30	19	72

☞ ***Selecting Library and Class:***

```
from sklearn.cluster import KMeans
```

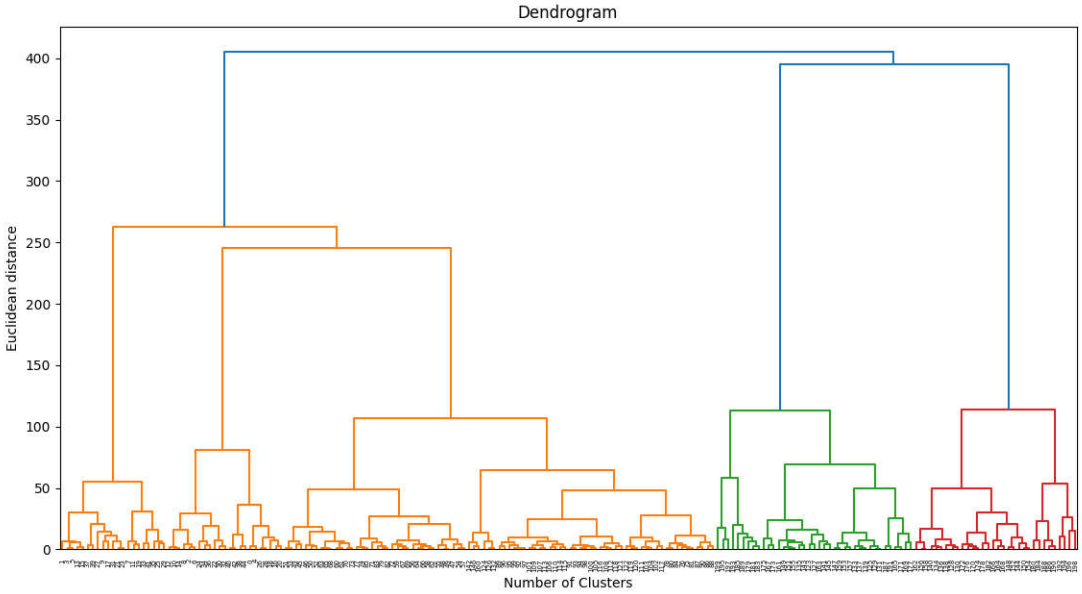
❒ ***The Dendrogram:*** Now we will find the ***optimal number*** of clusters using the ***Dendrogram*** for our model. For this, we are going to use ***scipy*** library as it provides a function that will ***directly return*** the ***dendrogram*** for our code. Consider the below lines of code:

```
# Dendrogram :: Finding optimal noumber of clusters using "Dendrogram"
import scipy.cluster.hierarchy as schr
dendrgm = schr.dendrogram(schr.linkage(X, method="ward"))
plt.title("Dendrogram")
plt.xlabel("Number of Clusters")
plt.ylabel("Euclidean distance")
plt.show()
```

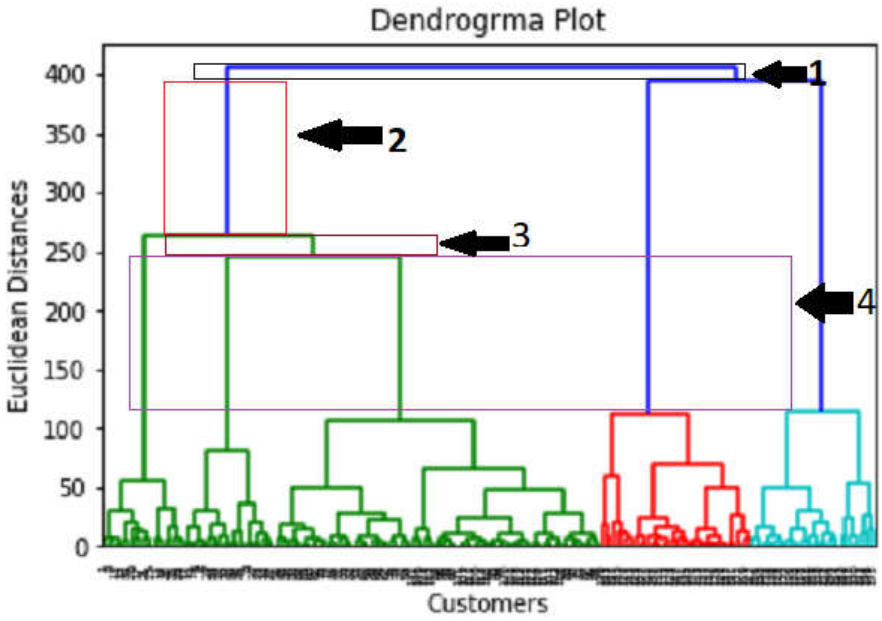
☞ In the above lines of code, we have imported the ***hierarchy module*** of ***scipy*** library. This module provides us a method ***denrogram()***, which takes the ***linkage()*** as a parameter. The ***linkage*** function is used to ***define*** the ***distance between*** two ***clusters***, so here we have passed the ***X*** (matrix of features), and method "***ward***," the popular method of linkage in hierarchical clustering.

☞ The remaining lines of code are to describe the labels for the dendrogram plot.

☞ ***Output:*** By executing the above lines of code, we will get the below output:



❒ ***Optimal Number Of Clusters:*** Using this *Dendrogram*, we will now determine the *optimal number* of *clusters* for our model. For this, we will find the ***maximum vertical distance*** that ***does not cut any horizontal bar***. Consider the below diagram:



☞ In the above diagram, we have shown the ***vertical distances*** that are ***not cutting*** their ***horizontal bars***. As we can visualize, the ***4th distance*** is looking the ***maximum***, so according to this, the number of clusters will be ***5***(the vertical lines in this range). We can also take the ***2nd*** number as it approximately equals the ***4th*** distance, but we will consider the ***5 clusters*** because the same we calculated in the ***K-means*** algorithm.

☞ In the ***Dendrogram*** the largest distance is considered the ***Rightmost Blue Vertical Line***, but it cut by two horizontal lines approximately at ***270*** and ***247***. Hence we take the ***range*** from ***248*** to ***120***, because in this range *no horizontal lines appear*. However we can take the ***upper portion*** of this ***vertical-line***, but as mention above we need to find the number near to ***k-means cluster***.

☞ So, the ***optimal number*** of ***clusters*** will be ***5***, and we will ***train*** the ***model*** in the ***next*** step, using the same as we did in k-means clusters.

❒ ***Training the hierarchical clustering model:*** As we know the required ***optimal number*** of clusters, we can now train our model.

```
from sklearn.cluster import AgglomerativeClustering
hrcl_cluster_genrt = AgglomerativeClustering(n_clusters = 5, affinity="euclidean", linkage='ward')
y_hrcl = hrcl_cluster_genrt.fit_predict(X)
```

☞ In the above code, we have imported the ***AgglomerativeClustering*** class of cluster module of ***scikit*** learn library.

☞ Then we have created the object of this class named as ***hrcl_cluster_genrt***. The ***AgglomerativeClustering*** class takes the following parameters:

- ***n_clusters=5:*** It defines the ***number*** of ***clusters***, and we have taken here ***5*** because it is the ***optimal number*** of clusters.
- ***affinity='euclidean':*** It is a ***metric*** used to ***compute*** the ***linkage***.
- ***linkage='ward':*** It defines the ***linkage criteria***, here we have used the "***ward***" ***linkage***. This method is the ***popular linkage method*** that we have already used for creating the ***Dendrogram***. It reduces the variance in each cluster.

☞ In the last line, we have created the ***dependent*** variable ***y_hrcl*** to *fit or train* the model notice we used ***fit_predict()*** as we did for ***k-means clustering***. It does train not only the model but also returns the clusters to which each data point belongs.

❒ After executing the above lines of code, if we go through the ***variable explorer*** option in our Sypder IDE, we can check the y_pred (in our case ***y_hrcl***) variable. We can compare the ***original dataset*** with the **y_pred** variable. Consider the below image:



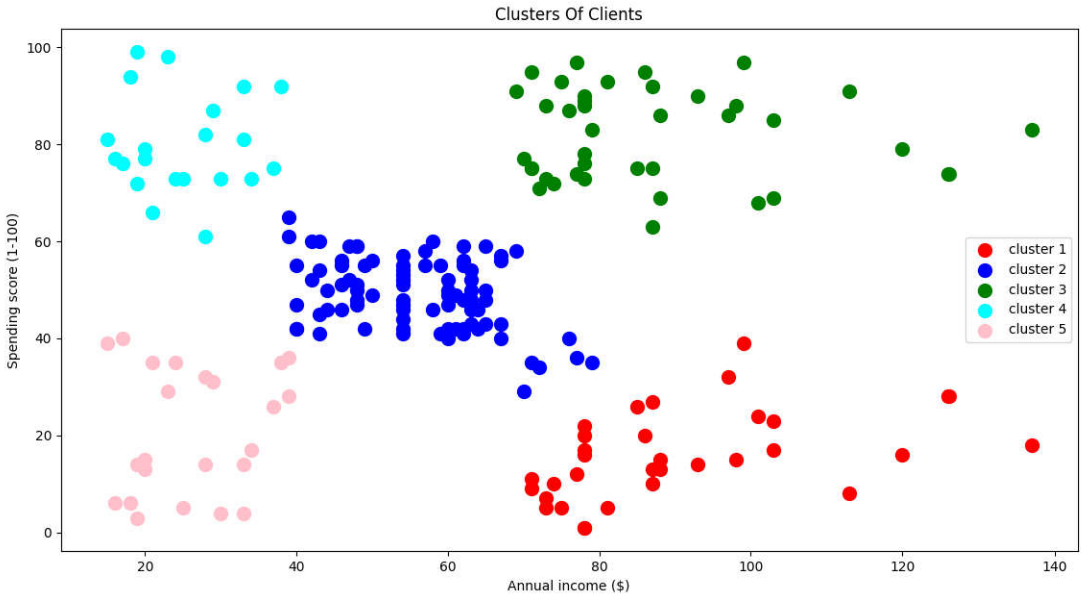
☞ As we can see in the above image, the ***y_pred*** shows the ***clusters value***, which means the customer id ***1*** belongs to the ***5th cluster*** (as indexing starts from ***0***, so ***4*** means ***5th cluster***), the ***customer id 2*** belongs to ***4th cluster***, and so on.

❒ ***Visualizing the Clusters:*** Here we will use the same lines of code as we did in ***k-means clustering***, except one change. Here we will ***not plot*** the ***centroid*** that we did in ***k-means***, because here we have used ***dendrogram*** to determine the ***optimal number*** of clusters. The code is given below:

☞ Notice that there is no ***centroid*** here.

```
# plotting the cluster
plt.scatter(X[y_hrcl == 0, 0], X[y_hrcl == 0, 1], s = 100, c = "red", label="cluster 1")
plt.scatter(X[y_hrcl == 1, 0], X[y_hrcl == 1, 1], s = 100, c = "blue", label="cluster 2")
plt.scatter(X[y_hrcl == 2, 0], X[y_hrcl == 2, 1], s = 100, c = "green", label="cluster 3")
plt.scatter(X[y_hrcl == 3, 0], X[y_hrcl == 3, 1], s = 100, c = "cyan", label="cluster 4")
```

```
# No centroids in Hierarchical clustering
# plt.scatter(hrcl_cluster_genrt.cluster_centers_[:, 0], hrcl_cluster_genrt.cluster_centers_[:, 1]
, s=300, c="black", label = "Centroids")
plt.title("Clusters Of Clients")
plt.xlabel("Annual income ($)")
plt.ylabel("Spending score (1-100)")
plt.legend()
plt.show()
```



❒ ***Categorize the customers:*** The output image is clearly showing the ***five different clusters*** with different colors. The clusters are formed between two parameters of the dataset; ***Annual income*** of customer and ***Spending***. We can change the colors and labels as per the requirement or choice. We can also observe some points from the above patterns, which are given below:

[1]. ***Cluster1*** shows the customer has a high income but low spending, so we can categorize them as ***Careful***

[2]. ***Cluster2*** shows the customers with average salary and average spending so we can categorize these customers as ***Standard***

[3]. ***Cluster3*** shows the customers with high income and high spending so they can be categorized as ***Target***, and these customers can be the most profitable customers for the mall owner.

[4]. ***Cluster4*** shows the customers with low income with very high spending so they can be categorized as ***Careless***.

[5]. ***Cluster5*** shows the low income and also low spending so they can be categorized as ***Sensible***.

❒ ***Multi-Dimensional Clustering:*** For 3 feature variable we still can visualize the cluster in 3-D graphics, but more tan 3D we can't plot the clusters.

☞ However later we will learn a technique that allows us to reduce the dimensions of our data. So that you can plot the clusters.

☝ If you are doing clustering in more than two dimensions then don't execute the last code lines in this section to visualize the clusters. Because it's only for two dimensional clustering.

Chapter 5 : Part 1

Association Rule Learning
Apriori

5.1.1 Association Rule Learning

Association rule learning is a type of ***Unsupervised Learning Technique*** that checks for the ***DEPENDENCY*** of *one data-item* on *another data-item* and ***maps*** accordingly so that it can be more ***profitable***.

☝ It tries to find some ***interesting relations*** or associations among the variables of dataset. It is based on different rules to discover the *interesting relations* between variables in the database.

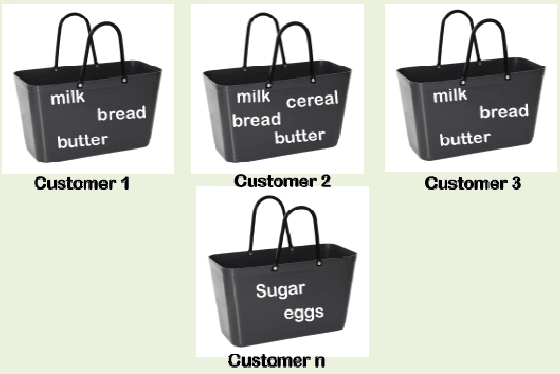
❒ It is employed in-

[1] ***Market Basket analysis:*** For example, if a customer buys *bread*, he most likely can also buy *butter*, *eggs*, or *milk*, so these products are stored *within a shelf* or *mostly nearby*.

[2] ***Web usage mining***

[3] ***Recommendation system***

[4] ***Continuous production***, etc.



❒ Here market basket analysis is a technique used by the various big retailer to discover the *associations between items*. We can understand it by taking an example of a supermarket, as in a supermarket, *all products* that are *purchased together* are *put together*.

❒ Association rule learning can be divided into three types of algorithms:

I. APRIORI
II. ECLAT
iii. F-P GROWTH Algorithm

Diapers and Beer Example: Very often during certain times of the day, when people shop in the afternoon between 6 and 9 p.m.





☞ People who buy *diapers* also buy ***beer.*** Which makes no sense at first but a plausible explanation might be: when the husband gets home and husband and the wife are taking care of their baby.

➢ They sometimes find that they run out of diapers and most of the time, the husband has to go pick up the diapers. While he's picking up the ***Diapers*** because it's really after hours after work, he also picks up some ***Beer***.

☞ And based on that you can decide how to arrange products in your store.

➢ So some stores might decide to put these two items (Beer & Diapers) closer to ***entice people*** to *buy a beer when they're*

- But actually a lot of stores do the opposite. For example, you probably noticed this from your convenience store that they try to separate **bread** and **milk** as ***far as possible***.
  - Because that way they really know that ***these two products are bought together***. And so you actually have to ***walk through the whole store*** to pick up.
  - So you've picked up your bread and then to get to the milk you have to get all the way through the whole store to the completely opposite corner of the store.
  - When you're ***walking through*** the store you ***see more other products*** and you're more likely to pick up an ***additional item*** that you ***weren't actually planning on buying*** when you got to the store in the first place.

- So there's a lot of interesting marketing tactics that are used based on this data.

5.1.2 How does Association Rule Learning work?

Association rule learning works on the concept of ***If*** and ***Else*** Statement, such as **`if`** A **`then`** B.



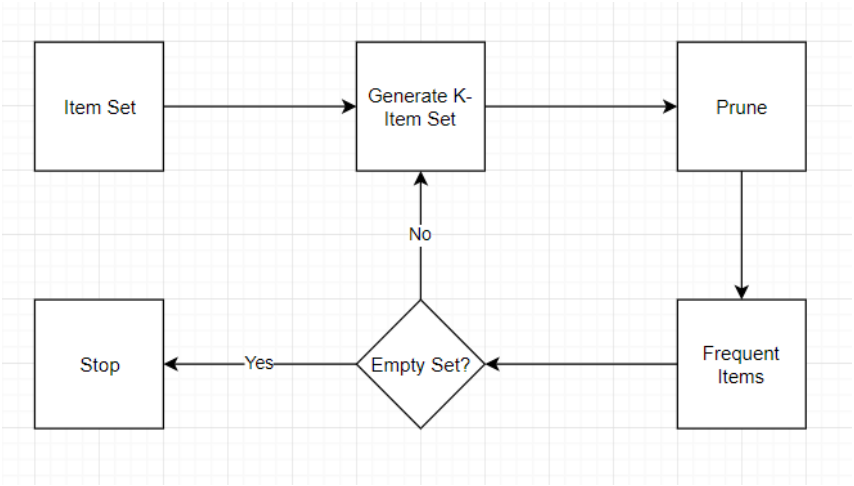
- Here the *`If element`* is called ***Antecedent***, and *`then statement`* is called as ***Consequent***. These types of **relationships** where we can find out some ***association or relation*** between ***two items*** is known as ***Single Cardinality***.
- It is all about ***creating rules***, and if the ***number of items*** *increases*, then ***cardinality*** also *increases* accordingly.

5.1.3 Apriori Algorithm

The ***Apriori*** algorithm uses ***frequent itemsets*** to generate ***association rules***, and it is designed to work on the databases that contain **transactions**. With the help of these **association rule**, it determines how **strongly** or how **weakly** ***two objects are connected***. This algorithm uses a ***breadth-first search*** and ***Hash Tree*** to calculate the **itemset associations** efficiently. It is the *`iterative`* process for finding the ***frequent itemsets*** from the ***large dataset***.

- This algorithm was given by the ***R. Agrawal*** and ***Srikant*** in the year 1994. It is mainly used for ***market basket analysis*** and helps to ***find those products that can be bought together***. It can also be used in the ***healthcare*** field to find ***drug reactions*** for patients.

- The control flow diagram for the Apriori algorithm



- **Apriori** uses a **"bottom up"** approach, where frequent subsets are extended one item at a time (a step known as **candidate generation**), and groups of candidates are tested against the data.
- The algorithm terminates when no further successful extensions are found.
- Apriori uses breadth-first search and a Hash tree structure to count candidate item sets efficiently. It generates candidate item sets of length from item sets of length . Then it prunes the candidates which have an infrequent sub pattern.
- According to the downward closure lemma, the candidate set contains all frequent -length item sets. After that, it scans the

❒ Apriori is all about:

- People who bought something also bought something else or
- Watched something also watched something else
- Did something also did something else

☞ This whole ***association rule learning*** part is all about analyzing when things come in ***pairs*** or in ***triplicates*** or in certain ***sequence*** but they are combined together for some reason looking for those ***rules*** (***reasons***, finding specific ***pattern***) and those ways that this happens.

Consider the following example:

User ID	Movies liked
46578	Movie1, Movie2, Movie3, Movie4
98989	Movie1, Movie2
71527	Movie1, Movie2, Movie4
78981	Movie1, Movie2
89192	Movie2, Movie4
61557	Movie1, Movie3

Potential Rules:

Movie1 ⇨ Movie2

Movie2 ⇨ Movie4

Movie1 ⇨ Movie3

☞ From above we can easily tell that there are some potential rules. For example everybody who watches Movie1 also like Movie2. People who like Movie2 also like Movie4. And people who like Movie1 are also quite likely to be like Movie3.

But from those rules some rules are "**strong**" and some rules are "**weak**". We want to find the ***very strong ones*** in order to build our ***business decisions*** or other decisions on those rules.

We don't want to go to the people and asking their opinions instead we extract those information/rules from our data. So if we have a large ***sample size*** say, ***50000*** or ***500000*** people then by we're analyzing that data we can come up with some quite solid rules.

5.1.4 How Does Apriori algorithm works

The apriori algorithm has three parts to it: the ***support***, the ***confidence*** and the ***lift***.

❒ ***Calculation metrices:*** So, to measure the associations between ***thousands*** of ***data-items***, there are ***several metrics***. These metrics are:

i. Support ii. Confidence iii. Lift

[1] ***Support:*** We're going to start up with the support and you will see that it's very similar to the way we talked about the ***Naïve Bayes classifiers***.

☞ ***Support*** is the ***frequency*** of ***A*** or ***how frequently an item appears in the dataset***. It is defined as the fraction of the ***transaction T*** that contains the itemset ***X***. If there are ***X datasets***, then for ***transactions T***, it can be written as:

$$support(X) = \frac{Freq(X)}{T}$$

Movie Recommendation: $\text{support}(M) = \frac{\#\text{ user watchlists containing } M}{\#\text{ user watchlists}}$

Market Basket Optimisation: $\text{support}(I) = \frac{\#\text{ transactions containing } I}{\#\text{ transactions}}$

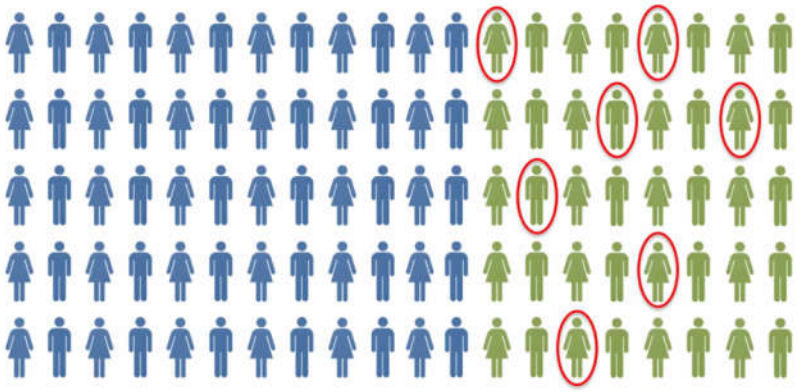
☯ ***For example:*** Say from ***100 people***, ***10 people*** watched ***Ex-Machina***, then $support = \frac{10}{100} = 10\%$,

[2] ***Confidence:*** Confidence indicates **how often the *rule* has been found** to be ***true***. Or, how often the ***items X*** and ***Y*** occur ***together*** in the **dataset** when the ***occurrence*** of ***X*** is ***already given***. It is the ratio of the ***transaction*** that contains ***X*** and ***Y*** to the number of ***records*** that contain ***X***.

$$Confidence(X \rightarrow Y) = \frac{Freq(X,Y)}{Freq(X)}$$

☯ ***For example:*** Now consider we are testing a rule, ***"People watched Intersteller are also watched Ex-Machina "***. So in the following diagram say ***green people watched Intersteller*** (the number is ***40 among 100***), and ***7*** of them watched ***Ex-Machina***, then the confidence is:

$$Confidence = \frac{7}{40} = 17.5\%$$



Movie Recommendation: $\text{confidence}(M_1 \rightarrow M_2) = \frac{\#\text{ user watchlists containing } M_1 \text{ and } M_2}{\#\text{ user watchlists containing } M_1}$

Market Basket Optimisation: $\text{confidence}(I_1 \rightarrow I_2) = \frac{\#\text{ transactions containing } I_1 \text{ and } I_2}{\#\text{ transactions containing } I_1}$

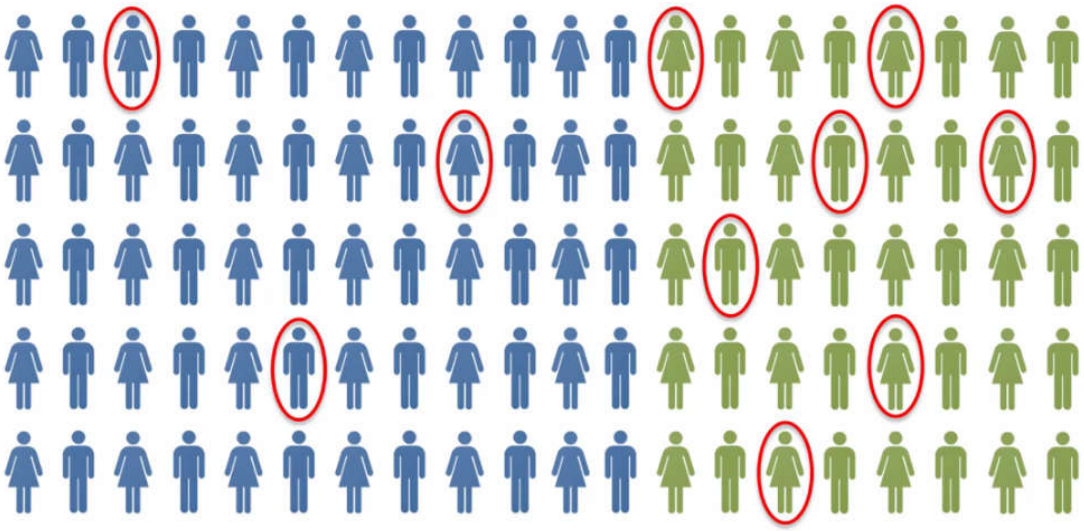
[3] ***Lift:*** Lift is very similar to the ***Naïve Bayes classifiers***. Lift is basically is the ***ratio*** of ***Confidence*** and ***Support***. It is the ***strength of any rule***, which can be defined as below formula:

$$Lift = \frac{Support(X,Y)}{Support(X) \times Support(Y)}$$

Movie Recommendation: $\text{lift}(M_1 \rightarrow M_2) = \frac{\text{confidence}(M_1 \rightarrow M_2)}{\text{support}(M_2)}$

Market Basket Optimisation: $\text{lift}(I_1 \rightarrow I_2) = \frac{\text{confidence}(I_1 \rightarrow I_2)}{\text{support}(I_2)}$

☯ ***Example:*** So in below ***Green*** people watched ***Intersteller*** and ***Red-circle*** represents the people watched ***Ex-Machina***.



☯ Out of this population we know that 10% actually likes ***Ex-Machina.*** So if we take another random population and then what is the ***likelihood that if we recommend to a random person*** in that brand ***new population*** will recommend the ***Ex-Machina*** movie, what is the likelihood that they will like it.

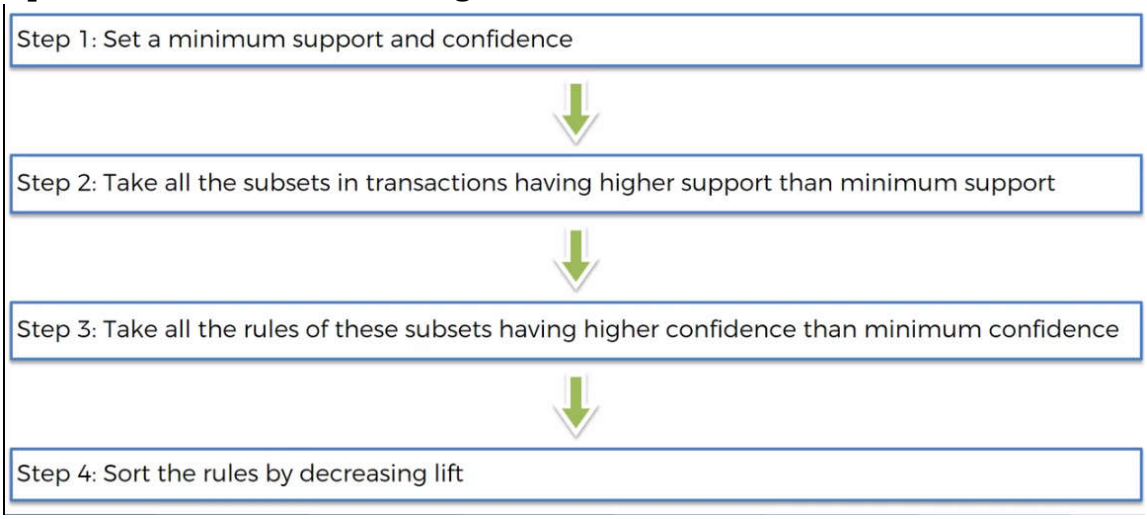
☯ Well the likelihood is 10%. But now the question is: ***Can we prove that result by using some prior knowledge***. That's why the algorithm is called ***Apriori***.

☯ In that new population let's only recommend ***Ex-Machina*** to people who have already seen ***Interstellar***. In that case the likelihood as we've calculated out of the ***Green*** people ***17.5%*** actually liked ***Ex-Machina***. So the ***Lift*** is the improvement in your prediction.

$$Lift = \frac{17.5\ \%}{10\ \%} = 1.75$$

☯ So, out of your new population, if you first ask the question "***Have you seen and liked interstellar?***". If they say "***yes***" and then you recommend "***Ex-Machine***" the likelihood of a successful recommendation is ***17.5%***. So the lift is by definition is ***1.75***.

5.1.5 Steps in Association Rule Learning



❒ ***Apriori*** is actually ***quite a slow algorithm*** because it just goes through all of these different combinations of rules. For example it will calculate rules for ***Movie1***, ***Movie2***, ***Movie3***, ***Movie4***, ... so on for pair (***Movie1*** and ***Movie2***) or triplet (***Movie1*** and ***Movie2 and Movie3***) so on.

❒ So there is so many different rules and we pick only strong rules. The rule with ***highest lift*** is the ***strongest*** rule.

5.1.6 Types of Association Rule Learning

[1] ***Apriori Algorithm:*** This algorithm uses frequent datasets to ***generate*** **association rules**. It is designed to work on the ***databases*** that ***contain transactions***. This algorithm uses a ***Breadth-First Search*** and ***Hash Tree*** to calculate the ***itemset*** efficiently.

- ☞ It is mainly used for **market basket analysis** and helps to understand the products that can be bought together. It can also be used in the healthcare field to find drug reactions for patients.

[2] ***Eclat Algorithm:*** *Eclat algorithm* stands for *Equivalence Class Transformation*. This algorithm uses a ***depth-first search*** technique to find ***frequent itemsets*** in a ***transaction*** database. It performs ***faster execution*** than ***Apriori*** Algorithm.

[3] ***F-P Growth Algorithm:*** The ***F-P growth*** algorithm stands for ***Frequent Pattern***, and it is the ***improved version*** of the ***Apriori*** Algorithm. It represents the ***database*** in the form of a ***tree structure*** that is known as a ***frequent pattern*** or ***tree***. The purpose of this ***frequent tree*** is to ***extract*** the most ***frequent patterns***.

❒ ***Applications of Association Rule Learning:*** It has various applications in **machine learning** and **data mining**. Below are some popular applications of **association rule learning**:

- ☞ ***Market Basket Analysis:*** It is one of the popular examples and applications of association rule mining. This technique is commonly used by big ***retailers*** to determine the ***association between items***.
- ☞ ***Medical Diagnosis:*** With the help of association rules, patients can be cured easily, as it helps in ***identifying the probability of illness*** for a particular disease.
- ☞ ***Protein Sequence:*** The association rules help in determining the **synthesis** of **artificial Proteins**.
- ☞ It is also used for the ***Catalog Design*** and ***Loss-leader Analysis*** and many more other applications.

❒ ***What is Frequent Itemset:*** Frequent itemsets are those items whose ***Support*** is greater than the ***Threshold Value*** or user-specified ***Minimum Support***. It means if ***A*** & ***B*** are the frequent ***itemsets together***, then ***individually A*** and ***B*** should also be the ***frequent itemset***.

- ☞ Suppose there are the two transactions: $A = \{1, 2, 3, 4, 5\}$, and $B = \{2, 3, 7\}$, in these two transactions, ***2*** and ***3*** are the ***frequent itemsets***.

Advantages of Apriori Algorithm

- This is easy to understand algorithm
- The ***join*** and ***prune*** steps of the algorithm can be easily implemented **on large datasets**.

Disadvantages of Apriori Algorithm

- The ***Apriori*** algorithm works ***Slow*** compared to other algorithms.
- The overall ***performance*** can be ***reduced*** as it scans the database for ***multiple times***.
- The ***time complexity*** and ***space complexity*** of the ***Apriori*** algorithm is ***O(2D),*** which is ***very high***. Here ***D*** represents the ***horizontal width*** present in the database.

5.1.7 Python Implementation of Apriori Algorithm

We have a problem of a retailer, who wants to find the ***association between*** his shop's ***product***, so that he can provide an offer of ***"Buy this and Get that"*** to his customers.

The retailer has a dataset information that contains a list of transactions made by his customer. In the dataset, each ***row shows the products purchased by customers*** or ***transactions*** made by the customer.

We are going to make this machine learning model to create some added value in some specific business. This business problem is going to be about ***optimizing*** the ***sales*** in a ***grocery store***. Using Association rule learning we want to know exactly where to place the products in the store. ***For example:*** If someone buys some ***Cereals*** the same person is very likely to buy some ***Milk*** as well.

❒ we're making this *Apriori model* for a store in the **south of FRANCE**. And so we want to find out the ***association rules*** *of the* ***different products*** of this Store to see how the Manager of this store can *optimize the placement* of its different products to *optimize the sales*.

- Imagine this store is located in one of the most popular places in the south of France. This place is a very Convivial place, a very friendly place where people love to hang out relax talk to each other.

📣 So these people come very often to the store to meet their friends. The manager of the store noticed and calculated that on ***average*** each ***customer*** goes and buys something to the store ***once a week***.

☯ So this dataset contains the 7500 transactions of all the different customers that bought a basket of products in a whole week.

☯ Indeed the manager took it as the basis of its analysis because since each customer is going an average once a week to the store then the transaction ***registered*** over a ***week*** is quite ***representative*** of what ***customers want to buy***.

📣 So based on all these ***7500 transactions*** our ***Apriori ML model*** our model is going to learn the different associations it can make to actually understand the rules. Such as: if customers buy certain product then they're likely to buy other set of products.

❒ ***Data Pre-processing:*** First, we will perform the importing of the libraries. Before importing the libraries, we will install the ***apyori*** package to use further. ***Apyori*** is a simple implementation of ***Apriori algorithm*** with Python 2.7 and 3.3 - 3.5, provided as ***APIs*** and as ***commandline*** interfaces. (We have to use the external .py file ***"apyori.py"*** after installing it using ***pip***).

pip install apyori

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

❒ ***Importing the dataset:*** Now, we will import the dataset for our apriori model. To import the dataset, there will be some changes here.

☞ All the rows of the dataset are showing different transactions made by the customers. The first row is the transaction done by the first customer, which means ***there is no particular name for each column*** and have their own individual value or product details(See the dataset given below after the code).

☞ So, we need to ***mention in our code*** that there is ***no header specified***. The code is given below:

Index	shrimp	almonds	avocado	vegetables mix	green grapes	whole weat
0	burgers	meatballs	eggs	nan	nan	nan

☞ What we can see here is: it contains some different names of different products. And this line is supposed to be the line containing the titles of the columns (but this is not the case here).

➢ But these names here are not the ***actual names*** of the ***columns*** because these names are simply the ***names of the products*** in the ***first transaction registered*** by the ***Manager*** of this store.

➢ And ***Pandas*** in ***Python*** thought that this ***first line*** of the data set contains the ***titles*** of the ***column***.

➢ So we need to specify to Python that there is no titles in the data set. So the code become as follows:

```
dataset = pd.read_csv("Market_Basket_Optimisation.csv", header = None)
```

dataset - DataFrame

Index	0	1	2	
0	shrimp	almonds	avocado	vegeta
1	burgers	meatballs	eggs	nan
2	chutney	nan	nan	nan

☝ ***No data-split:*** Because the Apriori model is a special type of machine learning model so we won't need to do any splitting of the data-set into a ***training*** set and a ***test*** set.

❒ ***Convert Data-set to "List of Lists":*** So, when we use ***apriori*** implementation of the ***apyori*** we need to import the data set in a specific way. And it expecting the data as "***List of Lists***".

☞ But our data-set is not in that form, so we need to convert our data set into "***List of Lists***" format. i.e. a ***list*** of customer's bought ***item-set***.

☞ It's going to require ***TWO FOR loops*** because we're going to **loop over** all the **transactions** in the data-set. That's the first loop. Which has ***7500*** rows.

☞ And the second loop will be about to ***loop over all the products*** in each of the ***transaction***. Which has ***20*** columns.

```
# importing data
dataset = pd.read_csv("Market_Basket_Optimisation.csv", header = None)

# creating "List of Lists": List of product-set
# there are 7500 rows of transections and 20 columns of product
transacTions = []
for i in range(0, 7501):
    # notice :: "List comprehension is used". Items are converted to strings
    # transacTions.append([str(dataset.values[i, j]) for j in range(0, 20)])
    transacTions.append([str(dataset.values[i, j]) for j in range(0, 20) if (str(dataset.values[i, j]) != "nan")])
```

☞ **dataset.values[i, j]:** We cannot take the values of the data set as ***dataset[i, j]*** we need to add remember ***.values***.

☞ Apriori is also expecting the items as ***strings*** so we need to convert the data in to string when we creating the lists ***str(dataset.values[i, j]***. To create list we used **[]** braces.

dataset - DataFrame

Index	0	1	2	3
0	shrimp	almonds	avocado	vegetables mix
1	burgers	meatballs	eggs	nan
2	chutney	nan	nan	nan
3	turkey	avocado	nan	nan
4	mineral water	milk	energy bar	whole wheat rice
5	low fat yogurt	nan	nan	nan
6	whole wheat pasta	french fries	nan	nan
7	soup	light cream	shallot	nan
8	frozen vegetables	spaghetti	green tea	nan
9	french fries	nan	nan	nan

Format | Resize | Background color | Column min/max | Save and Close | Close

transacTions - List (7501 elements)

Index	Type	Size	Value
0	list	20	['shrimp', 'almonds', 'avocado', 'vegetables mix', 'green grapes', 'wh ...
1	list	3	['burgers', 'meatballs', 'eggs']
2	list	1	['chutney']
3	list	2	['turkey', 'avocado']
4	list	5	['mineral water', 'milk', 'energy bar', 'whole wheat rice', 'green tea ...
5	list	1	['low fat yogurt']
6	list	2	['whole wheat pasta', 'french fries']
7	list	3	['soup', 'light cream', 'shallot']
8	list	3	['frozen vegetables', 'spaghetti', 'green tea']
9	list	1	['french fries']

☞ So as you can see this list contains ***7501 lists*** and ***each list*** corresponds to one ***transaction***.

❒ ***Training the model:*** We import ***apriori*** class from ***apyori*** package/api. We create our object and name it ***rUles*** , because this ***class*** take ***list of lists*** as input and output the ***rules***. This rules needs to converted into list so we used `list(rUles).`

☞ ***apriori*** is python 2.7 module so this is not working on python3. The updated module ***apyori.py*** needs to put/copy in the working directory

☞ ***Parameters:*** These arguments will actually depend on your business problem, depend on the ***number of observations*** you have in your ***data-set***. Of course your ***minimum support, confidence*** and ***lift*** is not going to be the same whether you have **1000 transactions** or **100000 transactions**.

```
# Train the apriori model
from apyori import apriori
rUles= apriori(transactions= transacTions, min_support=0.003, min_confidence = 0.2, min_lift=3, mi
n_length=2, max_length=2)
results= list(rUles)
for asocitn_item in results:
    print(f"{asocitn_item}\n")
```

In the above code, the first line is to import the apriori function. In the second line, the apriori function returns the output as the rules. It takes the following parameters:

- ***transactions:*** A list of transactions. We pass our ***list of lists*** data-set ***transacTions***.
- ***min_support=*** To set the minimum support float value. Here we have used 0.003 that is calculated by taking 3 transactions per customer each week to the total number of transactions.
  - ✓ We need to look at the products that are purchased rather frequently like at least three or four times a day.
  - ✓ It depends on your business goal. Now assume, we find some strong rules about items that are bought at least three or four times a day then by associating them and placing them together customers will be more likely to put them in their basket and therefore more of these products will be purchased and therefore the sales will increase.
  - ✓ Now, to set the ***minimum support*** we are going to consider the products that are purchased three or four times a day and then we will look at the rules. And of course if we're ***not convinced/satisfied*** by the ***rules*** we will change this value of the ***minimum support*** later.
  - ✓ Now if a product is bought 3 times per day then in 1 week it is $3 \times 7 = 21$. And we have total 7500 transactions. Then

$$min\_support = \frac{3 \times 7}{7500} = 0.0028 \approx 0.003$$

    So all the products of our rules will have a higher support than this support here.

- ***min_confidence:*** To set the minimum confidence value. Here we have taken 0.2. It can be changed as per the business problem.
  - ✓ confidence of 0.8 means that the rules has to be correct in 80 percent of the time.
  - ✓ But if you get some rules containing some products for example ***Mineral-water*** and ***Eggs***. In hot day people bought them frequently. But these two products has no connection between them. So they end up in the same basket. Right reason: not because they associate well together but because they're purchased all the time.
  - ✓ There is no logical association between these two products and that's why it's not very relevant. And unfortunately that's what we'll get if we ***set*** the ***confidence too high***.
  - ✓ Hence we set the confidence 20% i.e. `min_confidence = 0.2.`
- ***min_lift=*** To set the minimum lift value.
  - ✓ We also try ***different values*** of the ***minimum lift***.
  - ✓ So for now we try to get some rules that have lift above 3. Well these are actually some good rules because you know the left is a great insight of the relevance and the strength of rule. We're hoping to find some rules having lift to four, five or even six.

- ✓ And remember those parameters/arguments depends on your ***business problem***, on the number of ***observations*** in your data set. So you might ***spend*** a ***little time*** on this ***choice***.

- ***min_length=*** It takes the minimum number of products for the association. ***Minimum number of products*** we want to have in our rules. We want minimum 2 products that a customer would buy.
- ***max_length =*** It takes the maximum number of products/items for the association. i.e. maximum items in our rules. We can left it empty. Here we set it 2 because we want ***pair-of-items*** in our rules.

❒ ***Visualizing the result:*** Now we will visualize the output for our apriori model. Here we will follow some more steps, which are given below:

☞ Now we first train our model for ***min_length=2, and max_length=2.*** Then we get 8 rules.

```
# visualizing the rules
for item in results:
    pair = item[0]
    items = [x for x in pair]
    print("Rule: " + items[0] + " -> " + items[1])

    print("Support: " + str(item[1]))
    print("Confidence: " + str(item[2][0][2]))
    print("Lift: " + str(item[2][0][3]))
    print("====================================")
```

The rules

```
RelationRecord(items=frozenset({'light cream', 'chicken'}), support=0.004532728969470737,
ordered_statistics=[OrderedStatistic(items_base=frozenset({'light cream'}), items_add=frozenset({'chicken'}),
confidence=0.29059829059829057, lift=4.84395061728395)])

RelationRecord(items=frozenset({'mushroom cream sauce', 'escalope'}), support=0.005732568990801226,
ordered_statistics=[OrderedStatistic(items_base=frozenset({'mushroom cream sauce'}), items_add=frozenset({'escalope'}),
confidence=0.3006993006993007, lift=3.790832696715049)])

RelationRecord(items=frozenset({'pasta', 'escalope'}), support=0.005865884548726837,
ordered_statistics=[OrderedStatistic(items_base=frozenset({'pasta'}), items_add=frozenset({'escalope'}), confidence=0.3728813559322034,
lift=4.700811850163794)])

RelationRecord(items=frozenset({'honey', 'fromage blanc'}), support=0.003332888948140248,
ordered_statistics=[OrderedStatistic(items_base=frozenset({'fromage blanc'}), items_add=frozenset({'honey'}),
confidence=0.2450980392156863, lift=5.164270764485569)])

RelationRecord(items=frozenset({'ground beef', 'herb & pepper'}), support=0.015997866951073192,
ordered_statistics=[OrderedStatistic(items_base=frozenset({'herb & pepper'}), items_add=frozenset({'ground beef'}),
confidence=0.3234501347708895, lift=3.2919938411349285)])

RelationRecord(items=frozenset({'ground beef', 'tomato sauce'}), support=0.005332622317024397,
ordered_statistics=[OrderedStatistic(items_base=frozenset({'tomato sauce'}), items_add=frozenset({'ground beef'}),
confidence=0.3773584905660377, lift=3.840659481324083)])

RelationRecord(items=frozenset({'olive oil', 'light cream'}), support=0.003199573390214638,
ordered_statistics=[OrderedStatistic(items_base=frozenset({'light cream'}), items_add=frozenset({'olive oil'}),
confidence=0.20512820512820515, lift=3.1147098515519573)])

RelationRecord(items=frozenset({'whole wheat pasta', 'olive oil'}), support=0.007998933475536596,
ordered_statistics=[OrderedStatistic(items_base=frozenset({'whole wheat pasta'}), items_add=frozenset({'olive oil'}),
confidence=0.2714932126696833, lift=4.122410097642296)])

RelationRecord(items=frozenset({'shrimp', 'pasta'}), support=0.005065991201173177,
ordered_statistics=[OrderedStatistic(items_base=frozenset({'pasta'}), items_add=frozenset({'shrimp'}), confidence=0.3220338983050847,
lift=4.506672147735896)])
```

☞ ***Notice the above structure:***

```
RelationRecord( items=frozenset({'light cream', 'chicken'}),
                support=0.004532728969470737,
                ordered_statistics= [ OrderedStatistic(items_base=frozenset({'light cream'}),
                                      items_add=frozenset({'chicken'}),
                                      confidence=0.29059829059829057,
                                      lift=4.84395061728395)
                                      ]
                )
```

- It is a ***tuple*** form, contains 1st item as pair.
- 2nd element as float
- 3rd element as list

☞ Hence we can access the required items and format them as we want. The code given above.

☞ Finally we didn't set any ***max_length***.

```
rUles= apriori(transactions = transacTions, min_support=0.003, min_lift = 3, min_confidence
                                        =0.2, min_length = 2)
```

❒ ***The final code are given below:***

Practiced version

```
# -------- Association rule :: Apriori --------------

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# importing data
dataset = pd.read_csv("Market_Basket_Optimisation.csv", header = None)

# creating "List of Lists": List of product-set
# there are 7500 rows of transections and 20 columns of product
transacTions = []
for i in range(0, 7501):
    # notice :: "List comprehension is used". Items are converted to strings
    # transacTions.append([str(dataset.values[i, j]) for j in range(0, 20)])
    transacTions.append([str(dataset.values[i, j]) for j in range(0, 20) if (str(dataset.values[i, j]) !=
 "nan")])


# Train the apriori model
from apyori import apriori
# rUles= apriori(transactions= transacTions, min_support=0.003, min_confidence = 0.2, min_lift=3, min_len
gth=2, max_length=2)
rUles= apriori(transactions = transacTions, min_support=0.003, min_lift = 3, min_confidence=0.2, min_leng
th = 2)
results= list(rUles)
for asocitn_item in results:
    print(f"{asocitn_item}\n")

# visualizing the rules
for item in results:
    pair = item[0]
    items = [x for x in pair]
    rul = "Rule: "
    for itm in items:
        rul += (itm + " -> " )

    print(rul)
    print(f"Rule: {items}")
    print("Support: " + str(item[1]))
    print("Confidence: " + str(item[2][0][2]))
    print("Lift: " + str(item[2][0][3]))
    print("====================================")



# python prctc_apriori.py
```

❒ ***As a result:*** Following are first few rules that have ***lift*** more than ***4***. And we got final 77 total rules.

```
Rule: light cream -> chicken ->
Rule: ['light cream', 'chicken']
Support: 0.0045327289694707 37
Confidence: 0.29059829059829057
Lift: 4.84395061728395
====================================
Rule: escalope -> mushroom cream sauce ->
Rule: ['escalope', 'mushroom cream sauce']
Support: 0.005732568990801226
Confidence: 0.3006993006993007
Lift: 3.790832696715049
====================================
Rule: escalope -> pasta ->
Rule: ['escalope', 'pasta']
Support: 0.005865884548726837
Confidence: 0.3728813559322034
Lift: 4.700811850163794
====================================
Rule: honey -> fromage blanc ->
Rule: ['honey', 'fromage blanc']
Support: 0.003332888948140248
Confidence: 0.2450980392156863
Lift: 5.164270764485569
====================================
Rule: ground beef -> herb & pepper ->
Rule: ['ground beef', 'herb & pepper']
Support: 0.015997866951073192
Confidence: 0.3234501347708895
Lift: 3.2919938411349285
====================================
Rule: ground beef -> tomato sauce ->
Rule: ['ground beef', 'tomato sauce']
Support: 0.005332622317024397
Confidence: 0.3773584905660377
Lift: 3.840659481324083
====================================
Rule: light cream -> olive oil ->
Rule: ['light cream', 'olive oil']
Support: 0.003199573390214638
Confidence: 0.20512820512820515
Lift: 3.1147098515519573
====================================
Rule: olive oil -> whole wheat pasta ->
Rule: ['olive oil', 'whole wheat pasta']
Support: 0.007998933475536596
Confidence: 0.2714932126696833
Lift: 4.122410097642296
====================================
Rule: shrimp -> pasta ->
Rule: ['shrimp', 'pasta']
Support: 0.005065991201173177
Confidence: 0.3220338983050847
Lift: 4.506672147735896
====================================
. . . . .
 . . . . .
```

☞ ***Analyzing the rules:*** From the above output, we can analyze each rule. The first rules, which is ***Light cream → chicken***, states that the ***light cream*** and ***chicken*** are bought frequently by most of the customers. The ***support*** for this rule is ***0.0045***, and the ***confidence*** is ***29%***. Hence, if a customer buys ***light cream***, it is ***29% chances*** that he also buys ***chicken***, and it is ***.0045*** times appeared in the ***transactions***. We can check all these things in other rules also.

☝ ***Conclusion:*** Usually we try **several values** of the **parameters** here which will give us some rules and then ***experiment these rules*** in ***real life*** and then according to the results we can update the parameters and get more appropriate parameters.

☝ Of course ***data-scientists*** can combined these rules with other ***recommendation system techniques*** like ***Collaborative Filtering*** with you know the ***user profiles*** that can add some ***additional relevant info*** and also other more advanced techniques like the ***Neighborhood Model*** or ***Latent Factor Models***.

Well they combine a lot of models to increase the sales and the revenue.

Chapter 5 : Part 2

Association Rule Learning
Eclat

Data Types, Numbers, Operators, Type Conversion, f-strings

5.2.1 Eclat algorithm

Eclat stands for ***Equivalence Class Clustering*** and ***Bottom-Up Lattice Traversal*** and it is an algorithm for ***association rule mining*** (which also regroups frequent itemset mining).

- ***Eclat:*** In this model, only ***Support value*** is used, which shows ***how frequent a set of itmes occur***. Therefore, Eclat is a ***simplified version of Apriori model***.

- ***Association rule mining*** and ***frequent itemset mining*** are easiest to understand in their applications for ***basket analysis***: the goal here is to understand which ***products*** are often ***bought together by shoppers***.

- These association rules can then be used for example
  - for ***recommender engines*** (in case of ***online shopping***) or
  - for ***store improvement*** for ***offline shopping***.

- ***ECLAT for association rule:*** The ECLAT algorithm is ***not the first algorithm for association rule mining***. The ***foundational algorithm*** in the domain is the ***Apriori algorithm***. Since the **Apriori algorithm** is the **first algorithm** that was proposed in the domain, it *has been improved upon in terms of computational efficiency* (i.e. they made *faster alternatives*).

- ***ECLAT vs FP Growth vs Apriori:*** There are two faster alternatives to the **Apriori algorithm** that are state-of-the-art:
  - **[1]** one of them is ***FP Growth*** and
  - **[2]** the other one is ***ECLAT***.
  - Between ***FP Growth*** and ***ECLAT*** there is **no obvious winner** in terms of **execution times**: it will *depend on different data and different settings* in the algorithm.

- It also talks about the ***"people who bought also bought …"***. So it's kind of like a ***recommender system*** and similar to what we had in the ***Apriori algorithm***.

User ID	Movies liked
46578	Movie1, Movie2, Movie3, Movie4
98989	Movie1, Movie2
71527	Movie1, Movie2, Movie4
78981	Movie1, Movie2
89192	Movie2, Movie4
61557	Movie1, Movie3

Potential Rules:
- Movie1 ⇨ Movie2
- Movie2 ⇨ Movie4
- Movie1 ⇨ Movie3

- ***Eclat Algorithm:*** `Eclat algorithm` stands for ***Equivalence Class Transformation***. This algorithm uses a ***depth-first search*** technique to find ***frequent itemsets*** in a ***transaction*** database. It performs ***faster execution*** than ***Apriori*** Algorithm.

- In ***Apriori*** we worked with "***rules***" and these ***rules*** will have different strengths, and based on the ***Lift*** we could judge the strength of a ***rule***.
  - But in ***Eclat*** we are ***not*** actually going to be ***talking*** about ***rules***. Here we are going to be talking about sets.
  - In here we aren't judging the rules with their strength. We're not selecting rules and we're just saying ***what could potentially be*** and then the **Eclat model** is responsible for actually going ***through all of these combinations*** and telling us ***what we should focus on***.

☞ In ***Eclat*** model we only have ***support***: *In following* ***M*** *and* ***I*** *stands for* ***Set of Movies*** *or* ***Set of Items***. *That is there will be more than one item in the set.*

Movie Recommendation: $$\text{support}(M) = \frac{\#\text{ user watchlists containing } M}{\#\text{ user watchlists}}$$

Market Basket Optimisation: $$\text{support}(I) = \frac{\#\text{ transactions containing } I}{\#\text{ transactions}}$$

☞ So, how often does this happen - ***Some people are watching a certain combinations of movies***. So in ***Eclat model***, we don't have the ***`confidence`*** and the ***`lift factors`*** we're only looking at ***`support`*** and ***how frequently does this set of items occur***.

Example: We check that how many people watched ***`Ex-Machina`*** and ***`Interstellar`*** both in given data. And then when we get the new data we find the people who watched either ***`Ex-Machina`*** or ***`Interstellar`*** and we recommend them the other movie (I.e watched one movie and recommend other related movie).

❒ ***Steps:***

Eclat - Algorithm

Step 1: Set a minimum support

↓

Step 2: Take all the subsets in transactions having higher support than minimum support

↓

Step 3: Sort these subsets by decreasing support

☞ It's much faster and the steps involved are set a minimum support. So you want to ***set*** up a ***support level*** then you take all the subsets in transactions ***having higher support*** and then you set the ***subset*** in ***decreasing support***.

☞ And basically at the top you will have the most the strongest combinations of items which you should look at (e.g. end up with ***top 10*** or ***top five***).

Chapter 6 : Part 1

Reinforcement Learning

UCB: Upper Confidence Bound

6.1.1 Reinforcement Learning

Reinforcement Learning(RL) is a type of machine learning technique that enables an ***agent*** to *learn* in an *interactive environment* by ***trial and error*** using feedback from its *own actions* and *experiences*.

❒ Though both ***supervised*** and ***reinforcement learning*** use ***mapping*** between ***input*** and ***output***, unlike **supervised learning** where the **feedback** ***provided to the agent*** is ***correct set of actions*** for ***performing a task***, reinforcement learning uses ***REWARDS*** and ***PUNISHMENTS*** as signals for ***positive*** and ***negative*** behavior.

☞ As compared to ***unsupervised learning***, ***reinforcement learning*** is different in terms of goals. While the goal in *unsupervised learning* is to ***find similarities*** and ***differences*** between ***data-points***, in the case of *reinforcement learning* the goal is to *find* a *suitable action model* that would *maximize the total* ***cumulative reward*** of the agent.

❒ ***Reinforcement learning*** is an area of Machine Learning. It is about taking suitable action to ***maximize reward in a particular situation***. It is employed by various software and machines to find the best possible behavior or path it should take in a specific situation.

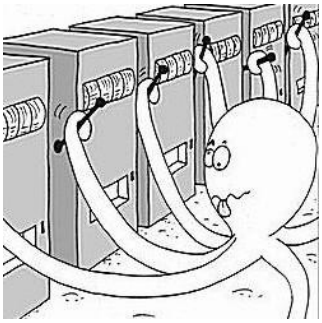
☞ *Reinforcement learning* differs from ***supervised learning*** in a way that in *supervised learning* the ***training data*** has the ***answer key with it*** so the model is ***trained with*** the ***correct answer*** itself whereas in ***reinforcement learning***, there is ***no answer*** but the reinforcement agent decides what to do to perform the given task. In the ***absence*** of a ***training dataset***, it is bound to ***learn from its experience***.

6.1.2 the Multi-Armed Bandit problem

❒ ***Multi-armed bandit:*** In **probability theory** and **machine learning**, the ***Multi-Armed Bandit Problem*** (sometimes called the ***K or N-armed*** *bandit problem*) is a problem in which

☞ A ***fixed limited set of resources*** must be ***allocated between competing (alternative) choices*** in a way that ***maximizes their expected gain***, when

☞ Each ***choice's properties*** are only ***partially known*** at the ***time*** of ***allocation***, and may become better understood ***as time passes*** or by ***allocating resources to the choice***.



☯ ***Classic Reinforcement Learning Problem:*** This is a classic reinforcement learning problem that exemplifies the exploration–exploitation tradeoff dilemma. The name comes from imagining

➲ A gambler at a ***row*** of ***slot machines*** (Those *slot machines* are sometimes known as ***"one-armed bandits"***), who has to decide which machines to play,

➲ how many ***times*** to play ***each machine*** and

➲ in ***which order*** to play them, and

➲ whether to ***continue*** with the ***current*** machine or ***try*** a ***different*** machine.

The multi-armed bandit problem also falls into the broad category of stochastic scheduling.

❒ ***What is a multi armed bandit:*** First thing that comes to mind is like a robber going into a bank and so on. But actually a ***bandit*** or more specifically *one armed bandit* is a ***slot machine*** (gambling).



☞ ***Why is it called the one arm bandit:*** Notice the *handle* on the right, you have to *pull* that lever to *initiate* the game (today most of those slot machines are electronic and you have to push the button).

⇨ These machines and you, there are ***50-50 chance to win/lose***. But in some casino this **machines** are ***designed*** with a ***bug***, so that the user ***lose*** their ***money*** very fast (i.e. more than 50% chance to lose). And they became the quickest way to lose your money in a casino. Hence the name ***bandit*** because it was ***basically robbing you***.



☞ Since those slot machines has one arm/handle they are called ***One Armed Bandit***. For more than one slot machine, i.e. if you play with a series of those slot machines they become ***`Multi-Armed Bandit`***.

☞ The ***`Multi-Armed Bandit problem`*** is kind of the challenge that a person is faced when he comes up to a whole set of these machines. This is the classic example. But there are more problems similar to this which are also called ***`Multi-Armed Bandit problem`***. Consider the ***Baby Robot*** problem given below.

☯ ***Another Example:*** *Baby Robot* is lost in the *mall*. Using *Reinforcement Learning* we want to help him find his *way back* to his *mum*. However, before he can even begin looking for her, he needs to ***recharge***, from a ***set of power sockets*** that each give a ***slightly different amount*** of ***charge***.

➢ Using the ***Strategies*** from the ***Multi-Armed Bandit Problem*** we need to find the ***`best socket`***, in the ***`shortest amount of time`***, to allow Baby Robot to get charged up and on his way.

➢ Baby Robot has entered a ***`charging room`*** containing ***`5 different power sockets`***. Each of these sockets returns a ***`slightly different amount`*** of ***`charge`***. We want to get ***`Baby Robot`*** charged up in the ***`minimum amount`*** of ***`time`***, so we need to ***`locate the best socket`*** and then use it until charging is complete.



☞ This is identical to the ***`Multi-Armed Bandit problem`*** except that, instead of looking for a ***`slot machine`*** that gives the ***`best payout`***, we're looking for a ***`power socket`*** that gives the ***`most charge`***.

❒ ***The Multi-Armed Bandit problem:*** Assume that you've got five of these machines. Your want to play them to *maximize* your *return* from the *number of games* that you can *actually play*. You decided how many times you're going to play, 100 times or 1000 times and you want to maximize return.

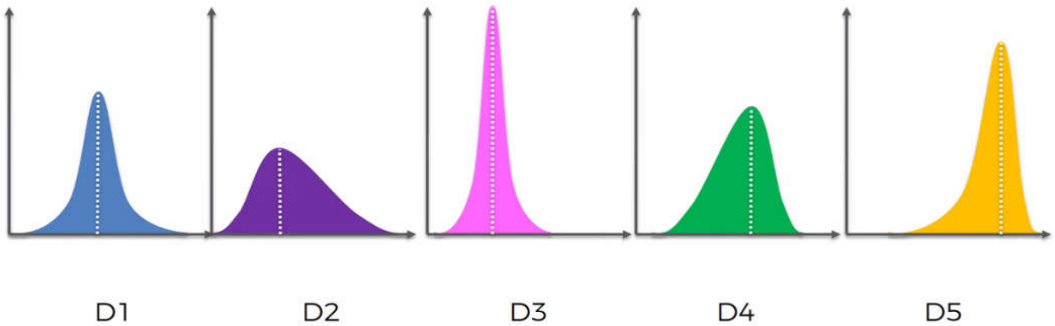
☯ How do you figure out ***`which ones`*** of them to play in order to ***`maximize`*** your ***`returns`***.

☯ ***The assumption here is:*** Each one of these machines has a distribution behind it (So there's a distribution of numbers for outcomes. You pull the trigger and it just picks out randomly out of its distributed numbers).

[1] Each ***machine*** has ***different distribution***. Sometimes it can be *similar* in *some* of the *machines* but by default they are *different*.

[2] You don't know these distribution.

[3] Your goal is to ***figure out*** which of these distributions is the ***best one for you***. With *minimum try* you have to figure out the *best distribution*.



☯ ***The best Distribution:*** The orange one is the best machine because:

- ➲ It's the most ***left skewed*** because the ***tails on the left***.
- ➲ So it's got the most favorable outcomes.
- ➲ The highest ***mean***, ***median*** and ***mode***.

If you knew these distributions, you can just go to the fifth machine and get the maximum outcome. But you don't know that in advance.

☯ And your goal is to ***figure out the best distribution for you***, with ***minimum trial***. So there are these two factors that are in play:

[1] ***Exploration*** : You don't know which one of these machine is the best, you're going to figure it out.

[2] ***Exploitation*** : But at the same time you are already spending your money doing each machine **Trial**.

i.e. the longer you take to figure out ***"the best distribution"*** there's a ***tradeoff***. The longer you take to figure it out the more money you'll probably spend on the wrong ones. Therefore you have to figure out very quickly.

☝ We call this ***regret*** and ***regret*** is **mathematically defined**.

☝ ***Paper:*** And if you can read more about this in this paper: ***Using Confidence Bounds for Exploitation-Exploration Trade-offs***, *Peter Auer; Institute for Theoretical Computer Science, Graz University of Technology*.

☯ ***REGRET:*** Basically regret is a suffering when you're using an **non optimal method**.

- ➲ So whenever you are using the ***non-optimal machine*** you have a ***regret***. Which can be quantified as: the difference between the ***best outcome*** and the ***non-best outcome***. So the longer you explore ***other non-optimal machines*** the ***higher REGRET***.
- ➲ But at the same time if you don't explore for long enough. Then a ***sub-optimal machine*** might appear ***as*** an ***optimal machine*** for you. In this case you **didn't get** the **best Machine** (real **optimal machine**).
- ➲ For instance: if we don't spend enough time exploring we might think that the green-colored distribution is the best machine because it's got quite a good (but not as good as orange one).

So our goal is ***find*** the ***best distribution*** but at the same time we have to ***minimize the exploration***.

🖒 ***Real-world application:*** One of the real-world application is **"to find out the best advertising idea for a certain community"**. Consider the following image as the ideas for "Coca-Cola" company:



☞ So there is a distribution behind it but that distribution will only become known after thousands and thousands of people look at these ads and click or not click on these ads.

☞ ***AB test:*** One way to approach a problem is just run an **A/B test**. So take your five or 50 or 500 ads and run a huge A/B test (with multiple A/B test) and wait until you have a large enough sample and then conclude which is the best with certain confidence.

- ⇨ But the problem of that is that you would ***spend*** a ***lot of time and money*** doing that. Because an A/B test is **pure exploration**. You're ***not exploiting the best*** option.

☯ So the challenge is to ***find out which is the Best One*** but do it ***while*** you're ***Exploring***.

- ⇨ So you have to do the **A/B test** and then use them to find out the best one in the ***quickest way possible*** and ***start exploiting*** it.

So that's the challenge here and that's what we're going to be solving. And that's the modern application of the ***Multi Armed Bandit Problem***.

☝ ***A/B testing:*** **A/B testing**, also known as **split testing**, refers to a ***randomized experimentation process*** wherein two or more versions of a ***variable*** (web page, page element, etc.) are shown to different segments of ***website visitors*** at the same time to determine which version leaves the ***maximum impact*** and ***drive business metrics***.

More on Bandit

[1] ***Know the The Bandit Framework:*** A description of the code and test framework.

[2] ***Bandit Algorithms:***

i. The Greedy Algorithm
ii. The Optimistic-Greedy Algorithm
iii. The Epsilon-Greedy Algorithm (s-Greedy)
iv. Regret

6.1.3 How Reinforcement learn is used to train a Robot Dog

We're going to be looking at different ways that we can solve the multi armed bandit problem and comparing the results. But remember that the ***Multi-Armed Bandit problem*** is not the only problem that can be solved with ***Reinforcement Learning***. For instance **Reinforcement Learning** is used to train **Robot Dogs** to **Walk**.

Assume that you have a ***programmable robot dog*** in which you can ***implement an algorithm*** inside the robot dog which will tell it how to walk.

☞ You can use two type of algorithms:

[1] You can actually give the sequence of actions such as to accomplish a task (walking): Move ***front right foot*** and then move ***left back foot*** and then ***Front Left Foot***, ***Right Back Foot*** and so on.

[2] Or you can implement ***Reinforcement Learning*** algorithm which will train the dog to walk in a very interesting way.

- ➲ We give the dog a **set** of **instructions** (actions dog can take), for instance: Instructions for **leg movements**.
- ➲ Then we ***set a goal:*** goal is to make a ***step forward***.
- ➲ And we give it ***rewards/Punishments*** on its action: Every time dog make a ***step forward*** it is given a ***reward*** every time, if it ***fall over*** it will given a ***punishment***.

⇨ **Reward** is basically a ***1***, **punishment** is ***0***. You just give it a ***1*** in algorithm and a **punishment** is ***0***. So basically you will try all these random sets of actions and see what they lead to every time it takes a step forward.

⇨ And every time the dog takes a step forward it knows it's got a reward and it's good for it. It remembers that those were good actions and will try to repeat them more and more and it can learn to walk.

⇨ So you ***don't have to program an actual walking algorithm*** into Dog will figure out the steps it needs to take on its own.

Now this topic is more of on the side of ***Artificial Intelligence*** rather than just ***Machine Learning***. Now we focus on our ***Multi-Armed Bandit problem***.

6.1.4 Upper Confidence Bound Algorithm in Reinforcement Learning

In Reinforcement learning, the ***agent*** or ***decision-maker*** generates its ***training data*** by ***interacting*** with the ***world***. The agent must learn the consequences of its actions through ***trial*** and ***error***, rather than being ***explicitly*** told the ***correct*** action. Consider the problem:

❒ ***Multi arm bandit Problem:***

[1] We have ***d*** arms. For example, arms are ***Ads*** : $\{ad_1, ad_2, \dots, ad_d\}$ that we display to users each time they connect to a **web page**.

[2] Each time a user connects to this web page, that makes a ***round***.

[3] At each ***round n***, we choose ***one ad*** to display to the user.

[4] At each round n, ad_i gives reward $r_i(n)$ defined as:

$$r_i(n) \in \{0,1\} : r_i(n) = \begin{cases} 1 & if \text{ the user clicked on the ad} \\ 0 & if \text{ the user didn't click} \end{cases}$$

[5] Our goal is to maximize the total reward we get over many rounds.

❒ ***UPPER CONFIDENCE BOUND ALGORITHM:***

[1] ***Step 1:*** At each round ***n***, we consider two numbers for each ad ***i***, ad_i

- $N_i(n)$ = The number of the times the ad_i was ***selected*** up to round n.
- $R_i(n)$ = The number of ***rewards*** of the ad_i up to round n.

[2] ***Step 2:*** From these two numbers we compute:

- The ***average reward*** of ad_i up to round n

$$\bar{r}_i(n) = \frac{R_i(n)}{N_i(n)}$$

- The confidence interval at round n is:

$$[\bar{r}_i(n) - \Delta_i(n) \quad , \quad \bar{r}_i(n) + \Delta_i(n)]$$

Where:

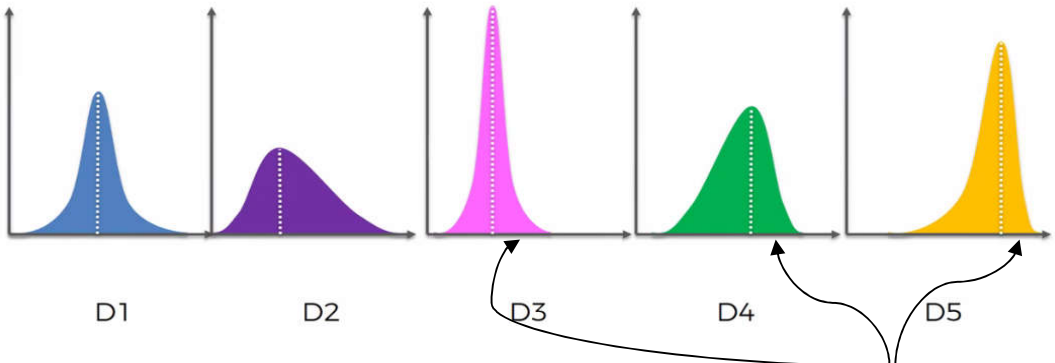
$$\Delta_i(n) = \sqrt{\frac{3\log(n)}{2\,N_i(n)}}$$

[3] ***Step 3:*** We select the ad_i that has the maximum ***UCB***, $\bar{r}_i(n) + \Delta_i(n)$.

❒ ***Walk through the example:*** Following are our ***slot machines*** or ***one arm bandits*** and each one of them has a ***distribution*** behind it.

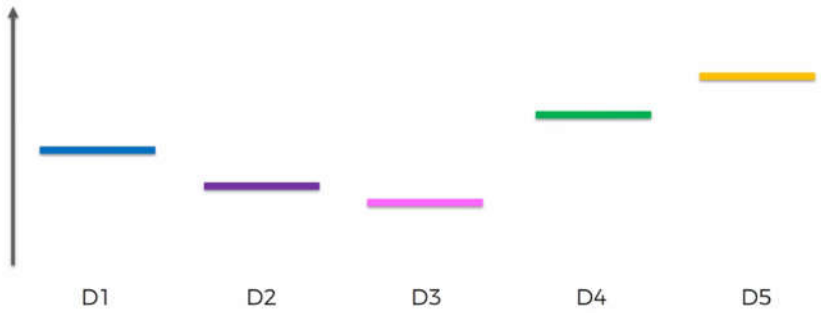


☞ We want to find the ***best one***. Looking at them we ***can't tell*** which one it is. But let's say we do know. Let's say we know the end result. Just for argument's sake the distributions look like. Obviously, the ***Orange*** one is the best distribution. But we don't know that. And we want to find that out in the process of playing these machines.



⇨ We're going to take the ***Actual Expected Return*** from the distributions (the ***right side portions*** from the ***mode***), and we're going to put them onto a ***vertical axis***.

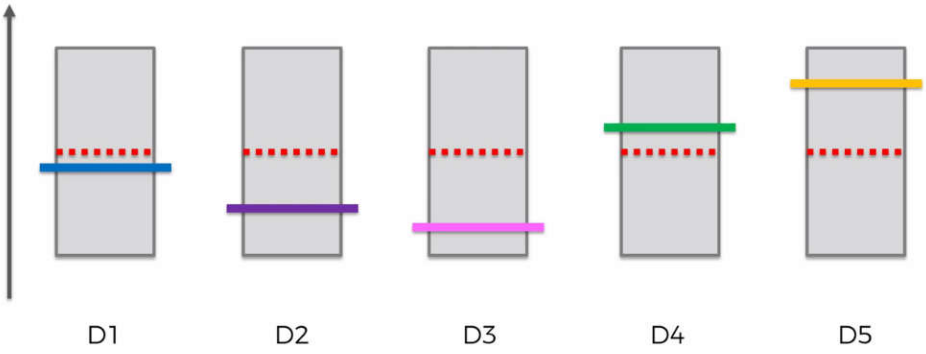
☞ So those are the ***expected values*** or ***returns*** for each of those ***distribution*** for each ***machine*** that's why are on **y** axis. But



⇨ Well, we assumes some ***starting*** point for ***every distribution***. Let's just assume that all returns the same, i.e. at same level because we we can't discriminate against these machines at the very beginning. They all look the same.



⇨ The algorithm (formulas that are behind algorithm) create a ***`confidence band`***. And it is ***designed in such a way*** that we have a very ***high level of certainty*** that ***`confidence band`*** will include the ***actual return*** or the ***actual expected return***.

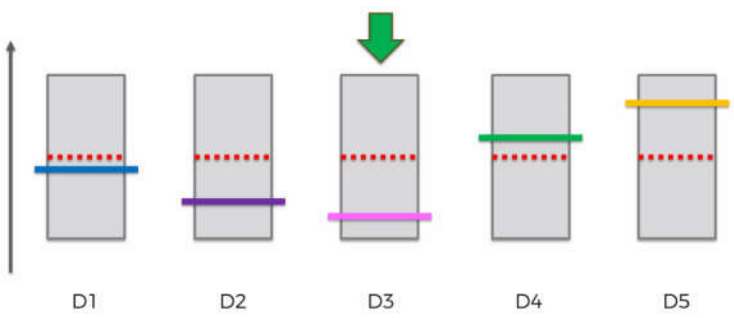


⇨ So ***`basically`*** the ***`first couple`*** of ***`rounds`*** are going to be ***`trial runs`***. We're going to ***intentionally*** just ***try out the machines*** at least ***one time each*** in order for us to be able to place this ***actual return value*** in a ***confidence band***.

⇨ At first ***confidence band*** is going to be very large so that the ***actual return*** or the ***actual expected return*** falls inside this confidence level, with a very high degree of certainty.

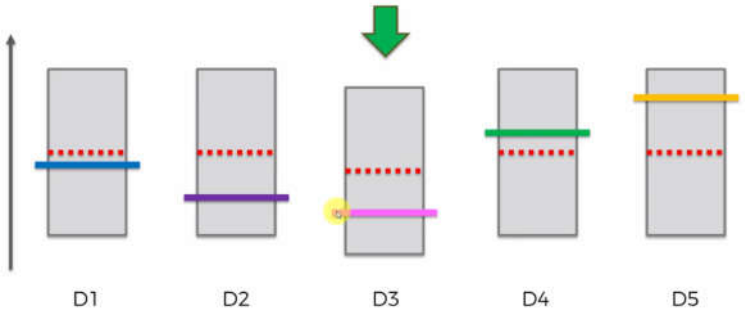
⇨ This ***confidence band*** is built around this ***red-dotted empirical value*** which are, at very start all the same.

☞ ***How does this algorithm work?:*** Out of all of them. We pick the machine with the ***`highest confidence`***. But right now it can be ***any*** of these ***machines***. They all have the ***same confidence***.

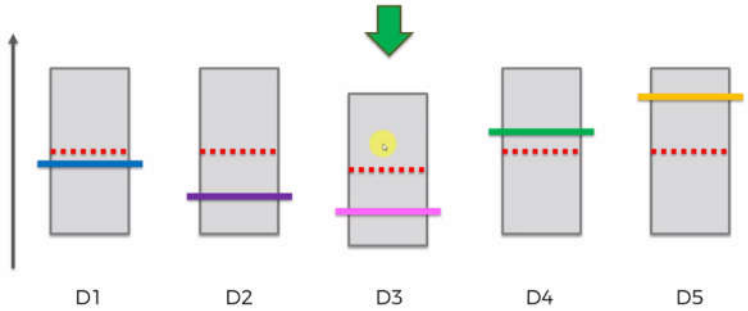


⇨ So we're going to pick any one of them. Next we actually pull the lever of that selected machine. (Or, in case of ***Ads picking problem:*** we **display that Ad** and next we want to see ***did the person click on it*** or did the person ***not click on it***.)

⇨ Lets assume, in this case the person loss hit money to the machine (or user didn't click on that Ad). So this ***red-dotted value goes down*** because it is like the ***observed average*** the for a *large numbers of observation* this ***red-dotted value*** is always going to ***converge to*** the ***expected return*** or ***expected average*** or ***expected value*** for this ***distribution***.



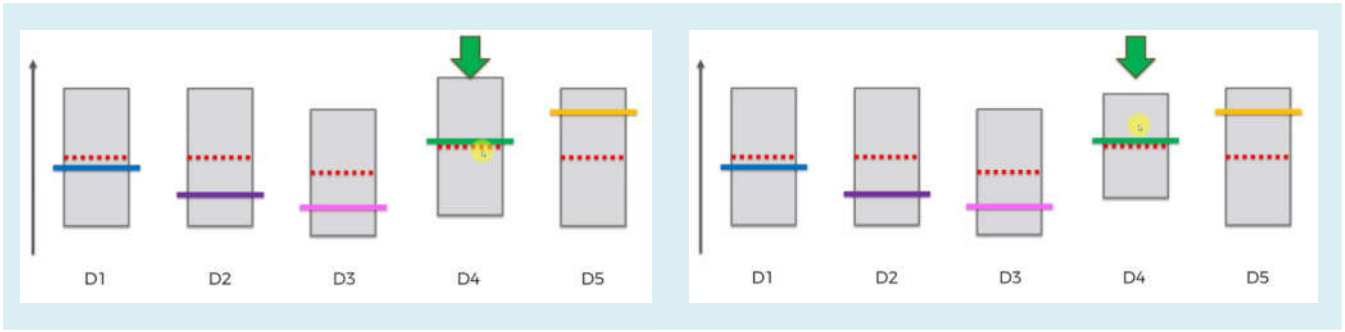
Red-dotted value goes down



Confidence bond Shrinks

☞ And now because we have an ***extra observation***, the ***second thing*** happens is the ***confidence*** bond ***Shrinks***. That ***confidence interval*** become ***smaller*** because we have an ***additional observation*** we are more ***confident*** in our ***predictions***.

☞ So the next step is now **we find the next one** with the highest ***confidence bond***. Obviously it's one of these remaining machines, and we are picking a random one. Say we choose the Green one.



⇨ Say we got positive result i.e. user wins with this machine (or, visitor clicked in our Ad), in this case the ***red-dotted value goes up***. And the confidence band shrinks.

- ❖ Because we got an additional observation in our sample. ***Confidence bands*** only purpose is to include the **actual expected value** wherever it is. When the sample grows, we become more confident about the **expected value**, hence this **confidence band** shrinks.

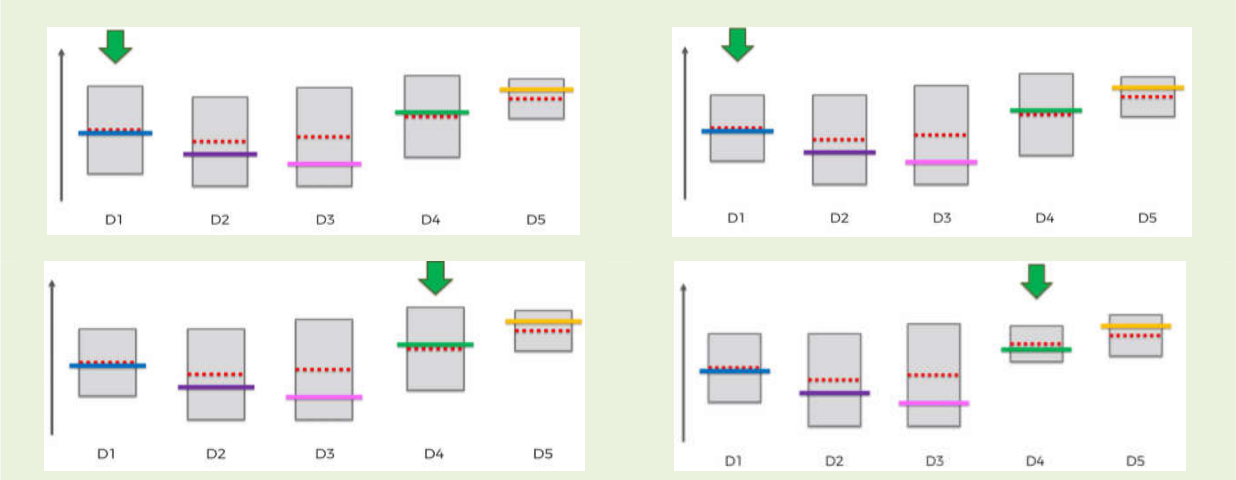
⇨ Now in this iteration we can ***stop looking*** for the ***best distribution*** and we can conclude that this ***current machine*** (or current Ad) is with the ***best distribution*** (which is not). In this case algorithm fails.

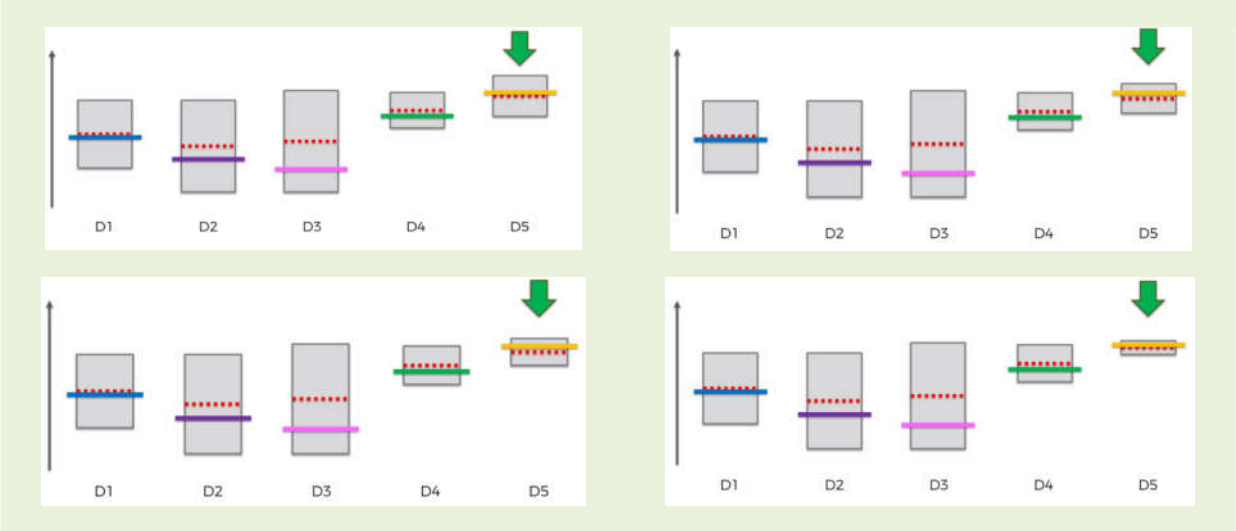
☞ Similarly we pick a **machine/Ad randomly** from the **remaining ones**. We do trial and we got positive/negative result the **red-dotted line** goes **up/down** according to the results and ***Confidence band Shrinks***.



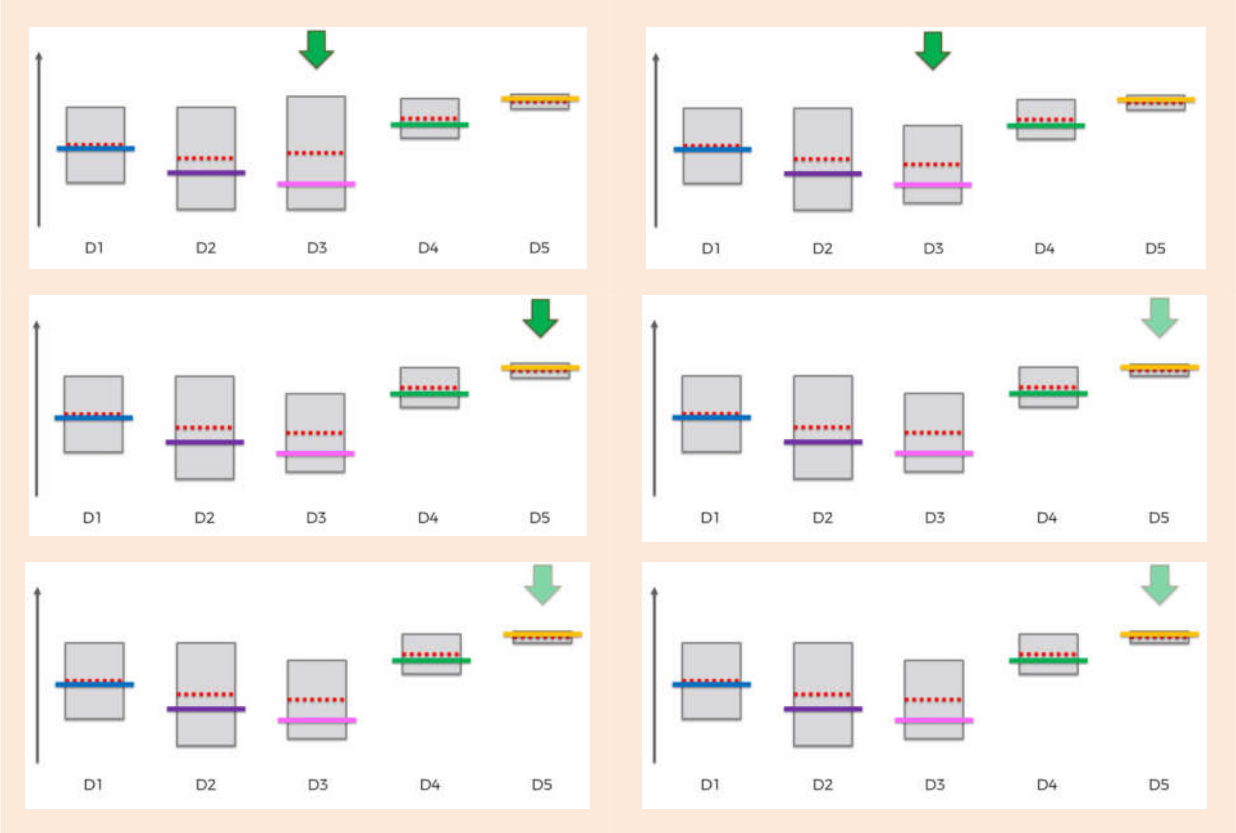
☞ For this example (we knew this is the best distribution) we exploit the Orange one, and its ***Confidence band*** gets so small. The main purpose is "if we found that the red-dotted line goes up for twice or more" the distribution is selected more. But also we pick some random machines.

☞ So even though we exploited the best option but for any option if it keeps going up, it's keeps being good. And at the same time we're building up the sample size hence ***decreasing*** the ***conference band***. But the point is: ***we select the distribution more with the positive outcome***.





☞ And some times algorithm looks for other distributions also. But more trials go the algorithm choose the best distribution (converging) more often (because we found out there is a good possibility to be the best one).

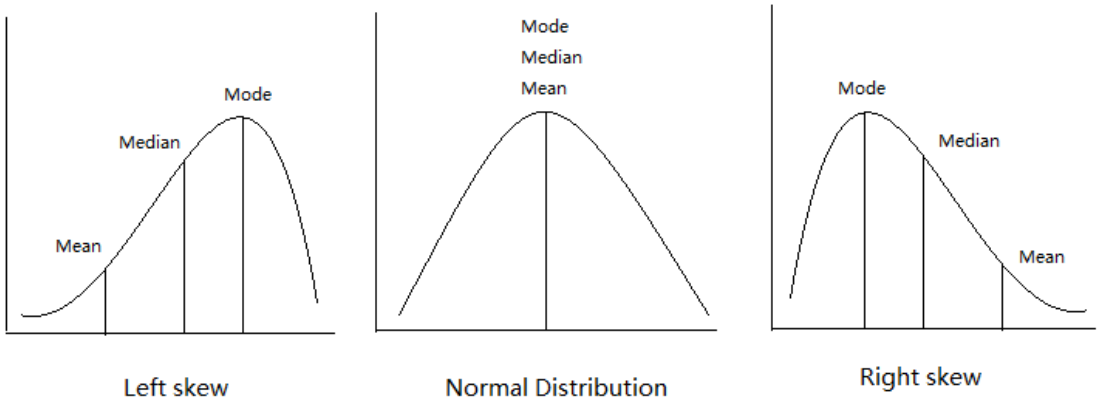


🛠 That's how the ***algorithm converges*** to the ***best distribution*** using ***minimal trial***: It's ***Focus More And More*** on the ***Distributions*** that are giving more ***Positive Results*** and *ignores the unlucky distribution*, as sample is growing. To avoid trap random selection is used.

☝ So that is in essence the whole concept behind this UPPER CONFIDENCE BOUND ALGORITHM and that's how it solves the multi arm bandit problem.

☝ It's a very interesting solution & much more sophisticated and better than *just selecting randomly* or running an ***A/B test and then selecting the option***. A very powerful algorithm.

❒ ***Mean, Median, and Mode:*** Every numerical data set has an ***average value*** that represents the ***weight*** of its ***array*** value. There are many different types of average! 3 of the most popular average values: ***Mean***, ***Median*** and ***Mode***. ***Mean***, ***Median*** and ***Mode*** are average values or ***central tendency*** of a ***numerical data set***.



[1] ***Mean:*** Mean can be calculated by adding all data points and dividing by the number of data points.

$$\mu = \frac{\sum_{i=0}^{n} x_i}{n}$$

[2] ***Median:*** Median is the ***middle value*** of a ***sorted data set***; found by **ordering all data points** and **picking** out the **one in the middle** (or if there are two middle numbers, taking the mean of those two numbers).

[3] ***Mode:*** Mode is the **most frequent number** — that is, the number that occurs the **highest number of times**.

6.1.5 Implementation of UCB in Python

Reinforcement learning is taking us closer to the field of ***Artificial Intelligence*** because *robots* and *artificial intelligence* that comes with it are *partly* built with *reinforcement learning*.

❒ ***Problem description:*** Our goal is ***CTR optimization***. **CTR** means *Click Through Rates*. We are going to try to optimize the *click through rates* of different *users* on an ***Ad*** that we put on a *social network*.

- For some **Car company** marketing campaigns on the ***social network*** about their new SUV Car model. They want to put ***Ads*** on the ***social network***.
- Now the Department of Marketing of that Car company prepared some different *versions* of the *same Ad*. They're actually not very sure which ad to put on the social network. They want to put the ad that will get the maximum clicks. So that most users buy the SUV.
- Now our goal is to find out the best version of the ***SUV car AD***. We consider this as the ***Bandit Problem***.

We want to find the ads that will get the most clicks.

6.1.6 No Data-set for Reinforcement Learning

Here dataset is not important. In reality there is no dataset. In this example dataset is used to ***simulate*** the **"CTR process"**.

❒ ***Real life situation:*** So in real life, we are going to start experimenting by placing those ***Ads*** on a ***social network*** (the different versions of the ad).

- According to the results we observe we will change our strategy to place these Ads on the social network. That is our observation will determine **"which particular version of Ad will be used next"**.

So there is a key difference between what we're about to do now and what we've done in previous chapters (i.e. Regression, Classification, Association-Rule) because ***earlier*** we had a data-set with some data containing ***independent variables*** and one ***dependent variable*** which are used for clustering/classification/regression.

- So things are different now. We start with no data (the given data-set is actually for simulation of CTR).
- The ***data-set*** we are using here is ***just*** for ***simulation*** and we're going to pretend we're in real life, we select an Ad and then we ***generate Click/No-click event*** using the given Data-set. We're going to pretend that we don't have any data yet.

Description of the Problem: We have ***ten versions*** of the ***same Ads***. 10 versions of these Ads trying to sell this cheap luxury SUV. And each time a user of the social network will ***log into*** his ***account*** we will place ***one version*** of these 10 Ads.

- Then we will ***observe its response*** and give Rewards:
  a) If the user clicks on the ***Ad***. We get a ***Reward*** equal to ***1***.
  b) And if the user doesn't click on the ad we get a ***Reward*** equals to ***0***.
  c) We are going to do this for ***ten thousand users*** on the social network. Then we observe if the user clicks: ***yes/no*** on the ***Ad***.
  d) The user click will be simulated using the given dataset.

- ***What's in the Data-set:*** This data-set actually mimic the real-time situation, which we are choose an Ad and this dataset generates User-Click (returns 1) or No-Click (returns 0).

dataset - DataFrame

Index	Ad 1	Ad 2	Ad 3	Ad 4	Ad 5	Ad 6	Ad 7	Ad 8	Ad 9	Ad 10
0	1	0	0	0	1	0	0	0	1	0
1	0	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0
5	1	1	0	0	0	0	0	0	0	0

- This is just some simulation of what is going to happen when we show the ads to the users: It's telling is for each round an user connecting to his account on which versions of the user is going to click on (or not).

- For example consider the five first users. Let's take first user indexed by 0.
  - At the first round, according to this simulation, this first user of the social network is going to click on the Ad if we show him the **1st**, **version** or the **5th version** or the **9th version**. And he/she not going to click the Ad if we choose ***2, 3, 4*** or ***6, 7, 8*** and ***10th version*** of the ***Ad***.
  - At the 2nd round, for the 2nd user, he/she not going to click the Ad except ***9th Version*** of the Ad.

Reinforcement learning in Action: The key thing to understand about ***reinforcement learning*** is that this *strategy will depend at* ***each round*** *on the* ***previous results*** *we* ***observed*** *at the* ***previous rounds***.

- So for example when we are at ***round 10***, the algorithm will look at the different results observed during the **first ten rounds** and **according** to **these results** ***UCB algorithm*** will decide which ***version*** of the ***Ad*** it will show to the user. That's why reinforcement learning is also called ***Online Learning*** or ***Interactive Learning***.

So how can we choose the different versions of the ads? There's going to be a ***specific strategy*** to do this. We compare two version of Algorithm. One is ***no-UCB*** i.e. we select Ads ***randomly***. The other version is: we use ***UCB*** to choose Ads ***according to observations***.

a) At our ***first version*** no Algorithm or no Strategy is used, we run ***10,000 trial***(round) one by one and we are ***not going to use UCB*** to choose the ***version*** of ***Ad*** instead we pick the ***Versions randomly*** for each round.
   - Each time a user connects to its account we're displaying one version of these ten at totally random. And we count the rewards.

b) In our ***second version*** we also run the ***10,000 trials*** and in this case we will use UCB to pick Ad-versions this algorithm will decide from here which version of the ad to show to the user.
   - And depending on the reward (***0*** or ***1***) of the current observation, the UCB-Algorithm will decide which Ad to show to the user at the next round.
   - The goal of the algorithm is to ***maximize*** the ***total reward***.

6.1.7 Random Selection Algorithm

The ***Random Selection Algorithm***, just consists of selecting one ***random version*** of the ***Ad*** each time a user connects on his/her social network accounts.

- The most important variable is the " ***total_reward***" the sum of the different rewards up to the 10000th user.
- If we run this following code, the total reward ***"total_reward"*** will be around 1200. That is around 1200 user clicks if we choose the Ads randomly.

```
# Random Selection

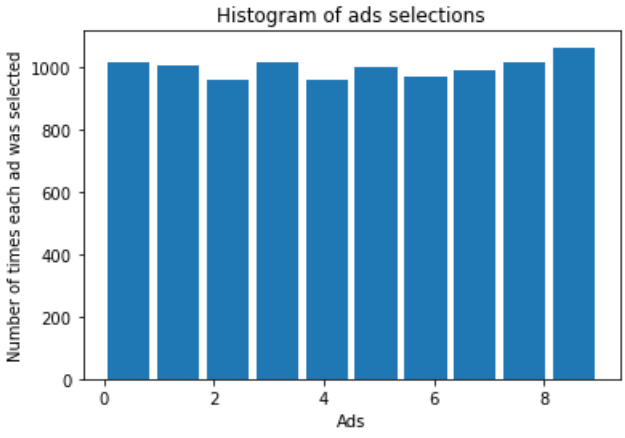
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random

# Importing the dataset
dataset = pd.read_csv('Ads_CTR_Optimisation.csv')

# Implementing Random Selection
import random
N = 10000
d = 10
ads_selected = []
total_reward = 0
for n in range(0, N):
    ad = random.randrange(d)
    ads_selected.append(ad)
    # in this case "reward" indicates the total clicks if we select the ads "randomly".
    # in UCB we do not select ads randomly.
    reward = dataset.values[n, ad]
    total_reward = total_reward + reward

# Visualising the results
plt.hist(ads_selected, rwidth=0.85)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()

"""
The parameter rwidth specifies the width of your bar relative to the width of your bin. For
 example, if your bin width is say 1 and rwidth=0.5, the bar width will be 0.5. On both sid
e of the bar you will have a space of 0.25.
"""
```



Note: ***Bar width of histogram:*** The parameter ***rwidth*** specifies the ***width*** of your ***bar*** relative to the ***width*** of your ***bin***. For example, if your ***bin*** width is say ***1*** and ***rwidth=0.5***, the ***bar*** width will be ***0.5***. On both side of the ***bar*** you will have a space of ***0.25***.

- In the for loop, we are selecting **Ad** randomly between **1** to **10**, using ***ad = random.randrange(d)*** where ***d = 10***, now we store this random number in the array ***ads_selected = []*** using ***ads_selected.append(ad)***, that is at the end the ***length*** of this array will be ***10,000***.

ads_selected	list	10000	[0, 4, 0, 4, 4, 6, 4, 4, 9, 9, ...]

☞ And this array ***ads_selected*** will be used to create ***"histogram"***. This histogram counts the randomly selected and displays through the histogram.

total_reward	int64	1	1236

☞ ***reward = dataset.values[n, ad]***

- Retrieves the reward ***0*** or ***1*** from the given dataset (simulation is happening in this line), at the ***n*** round/trial. Then "***total_reward = total_reward + reward***" counts the reward.
- For example, if the **1st version** of the ***Ad*** is selected for the first round, user will click it (according to simulation from dataset). Now in 2nd round/trial the 5th **version** of the ***Ad*** is selected then user will not click it.

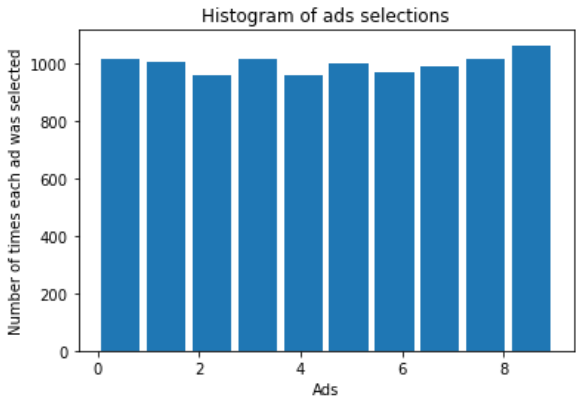
☝ NOTE: Notice the ***index*** of the ***add***, and ***random.randrange(d)*** generates integers **0, 1, 2, … , 9**.

☞ In the histogram we see that all the ***bars*** have ***nearly same height***, because ***Ads*** are ***selected randomly***.

☞ So let's keep this in our mind, that ***total_reward*** is always near ***1200*** (in case of random selection) because then we'll compare it to the total reward that we get from ***Upper Confidence Bound*** and then the ***Thompson Sampling*** algorithm.

❒ ***Visualizing the result:*** In this part of reinforcement learning the visualization of the results will consist of visualizing the histogram where we see the **different selections** of the **different versions** of the ***Ad***.

☞ Since our algorithm ***randomly selected*** the ***different versions*** of the ***Ads*** at each round. Hence we get a ***nearly uniform distribution*** of the different versions of the ***Ads***, where we selected ***All 10 Versions*** more or less the same number of times.



6.1.8 Coding the UCB algorithm in python from scratch

Currently we have no package on UCB. So we will code the following steps in Python:

❒ ***UPPER CONFIDENCE BOUND ALGORITHM:***

[1] ***Step 1:*** At each round ***n***, we consider two numbers for each ad ***i***, ad_i

- $N_i(n)$ = The number of the times the ad_i was ***selected*** up to round n.
- $R_i(n)$ = The number of ***rewards*** of the ad_i up to round n.

[2] ***Step 2:*** From these two numbers we compute:

- The ***average reward*** of ad_i up to round n

$$\bar{r}_i(n) = \frac{R_i(n)}{N_i(n)}$$

- The confidence interval at round n is:

$$[\bar{r}_i(n) - \Delta_i(n) \quad , \quad \bar{r}_i(n) + \Delta_i(n)]$$

Where:

$$\Delta_i(n) = \sqrt{\frac{3\log(n)}{2\,N_i(n)}}$$

[3] ***Step 3:*** We select the ad_i that has the maximum ***UCB***, $\bar{r}_i(n) + \Delta_i(n)$.

❒ ***Step 1:***
Lets set,

numbers_of_selections = The number of the times the ***ad_***i was selected up to ***round n***.
numbers_of_rewards = The number of rewards of the ***ad_i*** up to ***round n***.

☞ We consider it as a ***d size vecto***r. Here ***d*** indicates the ***number of bandits*** (in this case number of the ***Ads***, i.e. ***d = 10***). Hence we first create `numbers_of_selections` as a **`list of 0's`** of size ***d*** with all elements equal to **0**:

```
numbers_of_selections = [0]*d
```

☞ Similarly we create: `numbers_of_rewards = [0]*d`

☞ So that we can ***access*** and ***update*** the *"Number of Selection"* and *"Number of Rewards"* of the ***d-th Ad,*** by accessing the corresponding elements of these two vectors/lists

```
d = 10
numbers_of_selections = [0]*d
numbers_of_rewards = [0]*d
```

NOTE:

☝ ***Creating an Empty List of Length 10:***

```
l = [None] * 10
l = [None, None, None, None, None, None, None, None, None, None]
```

☞ Assigning a value to an existing element of the above list:

```
l[1] = 5
l = [None, 5, None, None, None, None, None, None, None, None]
```

Similarly we've created **`10 size`** list of **0**'s as:

```
name_list[0]*10
```

❒ ***Step 2:*** Notice we have two ***indices***, **n-th-round** and **i-th Ad**. ***n*** increments the ***10,000 trials*** and ***i*** increments ***10 Ads***. Hence we need two ***for*** loops.

☞ We do the following things:

i. The average reward of ad_i up to round n
ii. $\Delta_i(n)$ and
iii. Upper bound of the confidence interval i.e. Upper-Confidence-Bound for each Ad at round n:

```
for n in range (0, N):
    for i in range (0, d):
        avg_reward = numbers_of_rewards[i]/numbers_of_selections[i]
        # log(n+1) because of index
        delta = math.sqrt((3*math.log(n+1))/(2*numbers_of_selections[i]))
        upper_conf_bound =  avg_reward + delta
```

```
# UCB Implemnetation
import math
# here d is No. of Ads or Number of Bandits
d = 10
N = 10000   # Total number of trials

numbers_of_selections = [0]*d
numbers_of_rewards = [0]*d

for n in range (0, N):
    for i in range (0, d):
        avg_reward = numbers_of_rewards[i]/numbers_of_selections[i]
        # log(n+1) because of index
        delta = math.sqrt((3*math.log(n+1))/(2*numbers_of_selections[i]))
        upper_conf_bound =  avg_reward + delta
```

❒ ***Step 3:*** We select the i-th Ad that has the maximum ***UCB***, ***upper_conf_bound***

☞ For this we have to create a ***vector/list*** which contains selected ***Ads*** for all ***10,000 trials***. So first we'll create an empty list:

ads_selected = []

ads_selected will be a vector of 10000 elements and each of these elements will be the ***Ad*** that was ***selected*** at ***each round***. We are going to append this vector by the ***UCB selected Ads***.

☞ Now the question is how are we going to append the different versions of the Ad in this ***ads_selected*** vector?

⇨ So we created ***max_upper_bound*** and we initialize it for each ***trial/round*** (i.e. it is inside 1st ***for*** loop).

⇨ But also we need to ***keep track*** of the ***index*** of the Ad that has the ***max_upper_bound***. So that we can identify an Ad is selected. For this reason we used: ***ad_max_ucb = i***.

```
for n in range (0, N):
    ad_max_ucb = 0
    max_upper_bound = 0
    for i in range (0, d):
        avg_reward = numbers_of_rewards[i]/numbers_of_selections[i]
        # log(n+1) because of index
        delta = math.sqrt((3*math.log(n+1))/(2*numbers_of_selections[i]))
        upper_conf_bound =  avg_reward + delta
        if upper_conf_bound > max_upper_bound:
            max_upper_bound = upper_conf_bound
            ad_max_ucb = i
```

☞ ***10 Ads at the beginning (d Bandits at the beginning):*** At the beginning you know during the 10 first rounds (generally first ***d*** rounds) we **don't have much information** of the ***Ads***. We don't have much information about their ***reward***.

⇨ Basically for first 10 Ads we select them serially (not randomly), i.e. at ***round 1 Ad1*** is selected, ***round 2 Ad2*** and so on up to 10th round, afte 10th round/trial we are going to use UCB.

⇨ To implement this, we use an ***if-else*** condition inside the **second for-loop (i)**. And under ***else*** condition we put a very large ***upper_conf_bound***.

```
# UCB Implemnetation
import math
# here d is No. of Ads or Number of Bandits
d = 10
N = 10000   # Total number of trials

numbers_of_selections = [0]*d
numbers_of_rewards = [0]*d
slected_ads = []

for n in range (0, N):
    ad_max_ucb = 0
    max_upper_bound = 0
    for i in range (0, d):
        if(numbers_of_selections[i] > 0):
            avg_reward = numbers_of_rewards[i]/numbers_of_selections[i]
            # log(n+1) because of index
            delta = math.sqrt((3*math.log(n+1))/(2*numbers_of_selections[i]))
            upper_conf_bound =  avg_reward + delta
        else:
            upper_conf_bound = 1e400 # i.e. 10^400

        if upper_conf_bound > max_upper_bound:
            max_upper_bound = upper_conf_bound
            ad_max_ucb = i
```

⇨ ***Lets walk through the iterations:***

🕸 At first ***for*** loop ***n=0***, ***ad_max_ucb*** and ***max_upper_bound*** both set to ***0***.

✓ Inner ***for loop*** begins with ***Ad_0***, for ***Ad_0,***

`numbers_of_selections[0] > 0`

is ***false*** so ***upper_conf_bound*** is set to ***1e400***.

In following ***if*** condition, ***upper_conf_bound > max_upper_bound*** become ***true***. So ***max_upper_bound*** (which was **0**) is set to ***1e400***. And ***Ad_0*** is selected by this statement: ***ad_max_ucb = 0***.

- ✓ Then in 2nd iteration in inner ***for*** loop (***i = 1***).

  `numbers_of_selections[1] > 0` is ***false*** so ***upper_conf_bound*** is set to ***1e400*** again.

  But in following ***if*** condition, ***upper_conf_bound > max_upper_bound*** become ***False (1e400>1e400)***. Hence ***Ad_1*** is ***not going to selected*** by this statement: ***ad_max_ucb = i***.

- ✓ And this is happening for the remaining iterations in inner ***for*** loop (***i = 2, 3, . . . , 9***), ***Ad_0*** stays as the **selected Ad**. The inner ***for*** loop (***i***) ends for the first round ***n=0***.

Now at the first ***for*** loop in 2nd iteration ***n=1***, ***ad_max_ucb*** and ***max_upper_bound*** both set to ***0*** again.

- ✓ Inner ***for loop*** begins with ***Ad_0***, for ***Ad_0,***

  `numbers_of_selections[0] > 0`

  is ***True*** so ***upper_conf_bound*** is set by ***avg_reward + delta*** which is **smaller < *1e400***.

  - ➢ In following ***if*** condition, ***upper_conf_bound > max_upper_bound*** become ***true***, because ***max_upper_bound*** both set to ***0*** again.
  - ➢ So ***max_upper_bound*** (which was **0**) is set to ***avg_reward + delta*** which is **smaller < *1e400***.
  - ➢ And ***Ad_0*** is ***selected (temporarily)*** by this statement: ***ad_max_ucb = 0***.

- ✓ Then in 2nd iteration in inner ***for*** loop (***i = 1***). For ***Ad_1***

  `numbers_of_selections[1] > 0` is ***false*** so ***upper_conf_bound*** is set to ***1e400*** again.
  (Which was set to ***avg_reward + delta*** < ***1e400*** in previous iteration).

  - ➢ Hence in following ***if*** condition, ***upper_conf_bound > max_upper_bound*** become ***True***

    (1e400> avg_reward + delta).
  - ➢ So ***max_upper_bound*** (which was ***avg_reward + delta***) is set to ***1e400*** again.
  - ➢ Hence ***Ad_1*** is ***going to selected*** by this statement: ***ad_max_ucb = 1***.

- ✓ Then in 3rd iteration in inner ***for*** loop (***i = 2***). For ***Ad_2***

  `numbers_of_selections[2] > 0` is ***false*** so ***upper_conf_bound*** is set to ***1e400*** again.

  But in following ***if*** condition, ***upper_conf_bound > max_upper_bound*** become ***False (1e400>1e400)***. Hence, 3rd Ad, ***Ad_2*** is ***not going to selected*** by this statement: ***ad_max_ucb = i***.

- ✓ And this is happening for the remaining iterations in inner ***for*** loop (***i = 3, 4, . . . , 9***), So the 2nd Ad, ***Ad_1*** stays as the **selected Ad** after the inner ***for*** loop (***i***) ends for the second round ***n=1.***

This is how ***all 10 (d) Ads*** gets ***selected*** for the first ***10 rounds/trials*** for ***n= 0, 1, 2, . . . , 9***.

And from ***n = 11*** to ***9,999*** the following statement never going to executed and ***numbers_of_selections[i] > 0*** remains ***true*** for all following iterations (i.e. **Ads will be selected by the maximum UCB**):

```
else:
        upper_conf_bound = 1e400 # i.e. 10^400
```

- ✓ And UCB is always calculated using: ***upper_conf_bound = avg_reward + delta***

- ✓ And the Ads are selected by the following:

```
if upper_conf_bound > max_upper_bound:
    max_upper_bound = upper_conf_bound
    ad_max_ucb = i
```

☞ Storing Selected Ads to ***slected_ads***, tracking ***numbers_of_selections***, ***numbers_of_rewards*** and calculating the ***total_reward***.

```python
# Reinforcement lrarning : ----------  UCB. ---------  Select an add by Click on Ad

import pandas as pd
import numba as np
import matplotlib.pyplot as plt


# importing dataset
dataSet = pd.read_csv("Ads_CTR_Optimisation.csv")


# UCB Implemnetation.
    # 3 parameters are important:
        # no. of bandit d,
        # No. of trials N,
        # initial max_upper_bound (1e400)

import math
# here d is No. of Ads or Number of Bandits
d = 10
N = 10000   # Total number of trials

numbers_of_selections = [0]*d
numbers_of_rewards = [0]*d
slected_ads = []
total_reward = 0


for n in range (0, N):
    ad_max_ucb = 0
    max_upper_bound = 0

    for i in range (0, d):
        if(numbers_of_selections[i] > 0):
            avg_reward = numbers_of_rewards[i]/numbers_of_selections[i]
            # log(n+1) because of index
            delta = math.sqrt((3*math.log(n+1))/(2*numbers_of_selections[i]))
            upper_conf_bound =  avg_reward + delta
        else:
            upper_conf_bound = 1e400 # i.e. 10^400

        if upper_conf_bound > max_upper_bound:
            max_upper_bound = upper_conf_bound
            ad_max_ucb = i

    # storing selected Ad
    slected_ads.append(ad_max_ucb)

    # updating "numbers_of_selections" and "numbers_of_rewards" of selected Ad "ad_max_ucb"
    numbers_of_selections[ad_max_ucb] += 1
    reward = dataSet.values[n, ad_max_ucb] # generating reward-simulation from given Dataset
    numbers_of_rewards[ad_max_ucb] += reward

    total_reward += reward


# python prctc_UCB.py
```

Moment of Truth:

☞ After running the above code we get **total reward** ***2178***, which is more than **1200** (in **random Ad selection**). So **UCB** improved the Ad selection.

☞ From the following Data we see that the ***5th Ad*** (i.e. index 4), is the ***best*** version of the ***Ad***.

☞ The Ad selection process of UCB is visible from the 31st round/iteration. And for most iterations Ad of index 4 (i.e. 5th Ad) is being selected. At the down 5th Ad selected mostly.

☞ And we can only observe ***4***, but be careful this is the ***index*** this the actually the ***5th Ad***.

numbers_of_rewards	***numbers_of_selections***	***UCB selected Ads***

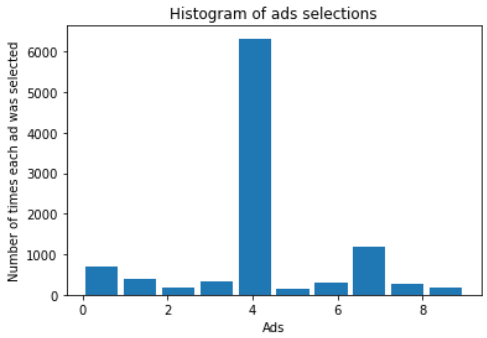
Index	Type	Size	Value
0	int64	1	120
1	int64	1	47
2	int64	1	7
3	int64	1	38
4	int64	1	1675
5	int64	1	1
6	int64	1	27
7	int64	1	236
8	int64	1	20
9	int64	1	7

Index	Type	Size	
0	int	1	705
1	int	1	387
2	int	1	186
3	int	1	345
4	int	1	6323
5	int	1	150
6	int	1	292
7	int	1	1170
8	int	1	256
9	int	1	186

24	int	1	3
25	int	1	4
26	int	1	5
27	int	1	6
28	int	1	7
29	int	1	8
30	int	1	8
31	int	1	9
32	int	1	0
33	int	1	8
34	int	1	1
35	int	1	2
36	int	1	3

❒ ***Visualizing the result:*** As we did before for **"Random Ad selection"**. We use the list ***slected_ads*** to draw the histogram.

```
#visualizing the result
plt.hist(slected_ads, rwidth=0.85)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()
```



Shows 5th Ad with maximum exploitation

Practiced version

```
# Reinforcement lrarning : ----------  UCB. ---------  Select an add by Click on Ad

import pandas as pd
import numba as np
import matplotlib.pyplot as plt


# importing dataset
dataSet = pd.read_csv("Ads_CTR_Optimisation.csv")


# UCB Implemnetation.
    # 3 parameters are important:
        # no. of bandit d,
        # No. of trials N,
        # initial max_upper_bound (1e400)

import math
# here d is No. of Ads or Number of Bandits
d = 10
N = 10000   # Total number of trials

numbers_of_selections = [0]*d
numbers_of_rewards = [0]*d
slected_ads = []
total_reward = 0


for n in range (0, N):
    ad_max_ucb = 0
    max_upper_bound = 0

    for i in range (0, d):
```

```
        if(numbers_of_selections[i] > 0):
            avg_reward = numbers_of_rewards[i]/numbers_of_selections[i]
            # log(n+1) because of index
            delta = math.sqrt((3*math.log(n+1))/(2*numbers_of_selections[i]))
            upper_conf_bound =  avg_reward + delta
        else:
            upper_conf_bound = 1e400 # i.e. 10^400

        if upper_conf_bound > max_upper_bound:
            max_upper_bound = upper_conf_bound
            ad_max_ucb = i

    # storing selected Ad
    slected_ads.append(ad_max_ucb)

    # updating "numbers_of_selections" and "numbers_of_rewards" of selected Ad "ad_max_ucb"
    numbers_of_selections[ad_max_ucb] += 1
    reward = dataSet.values[n, ad_max_ucb] # generating reward-simulation from given Dataset
    numbers_of_rewards[ad_max_ucb] += reward

    total_reward += reward


#visualizing the result
plt.hist(slected_ads, rwidth=0.85)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()


# python prctc_UCB.py
```

Instructor version

```
# Upper Confidence Bound (UCB)

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Ads_CTR_Optimisation.csv')

# Implementing UCB
import math
N = 10000
d = 10
ads_selected = []
numbers_of_selections = [0] * d
sums_of_rewards = [0] * d
total_reward = 0
for n in range(0, N):
    ad = 0
    max_upper_bound = 0
    for i in range(0, d):
        if (numbers_of_selections[i] > 0):
            average_reward = sums_of_rewards[i] / numbers_of_selections[i]
            delta_i = math.sqrt(3/2 * math.log(n + 1) / numbers_of_selections[i])
            upper_bound = average_reward + delta_i
        else:
            upper_bound = 1e400
        if upper_bound > max_upper_bound:
            max_upper_bound = upper_bound
            ad = i
    ads_selected.append(ad)
    numbers_of_selections[ad] = numbers_of_selections[ad] + 1
    reward = dataset.values[n, ad]
    sums_of_rewards[ad] = sums_of_rewards[ad] + reward
    total_reward = total_reward + reward

# Visualising the results
plt.hist(ads_selected)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()
```

Chapter 6 : Part 2

Reinforcement Learning
Thompson Sampling

6.2.1 Thompson Sampling

We're going to be using this algorithm to solve the ***Multi Armed Bandit Problem***.

❒ ***Multi Armed Bandit Problem:*** We have several slot machines each one of them has a distribution behind it. We don't know what these distributions are and we need to start playing these machines and at the same time figure out which one has the best distribution.



☞ So we have to find that ***ideal balance*** or ***tradeoff*** between ***exploration*** and ***exploitation***.

Multi arm bandit Problem Assumptions

[1] We have ***d*** arms. For example, arms are ***Ads*** : $\{ad_1, ad_2, \dots, ad_d\}$ that we display to users each time they connect to a **web page**.

[2] Each time a user connects to this web page, that makes a ***round***.

[3] At each ***round n***, we choose ***one ad*** to display to the user.

[4] At each round n, ad_i gives reward $r_i(n)$ defined as:

$$r_i(n) \in \{0,1\} : r_i(n) = \begin{cases} 1 & if \text{ the user clicked on the ad} \\ 0 & if \text{ the user didn't click} \end{cases}$$

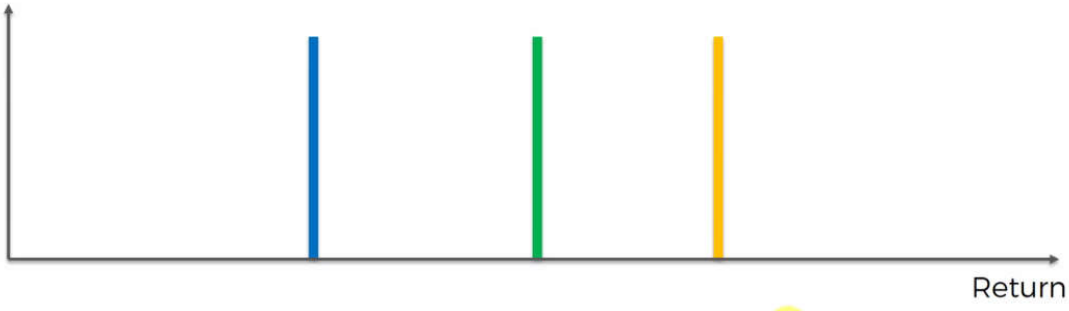
[5] Our goal is to maximize the total reward we get over many rounds.

Bayesian Inference

- Ad i gets rewards $\mathbf{y}$ from Bernoulli distribution $p(\mathbf{y}|\theta_i) \sim \mathcal{B}(\theta_i)$.
- θ_i is unknown but we set its uncertainty by assuming it has a uniform distribution $p(\theta_i) \sim \mathcal{U}([0,1])$, which is the prior distribution.
- Bayes Rule: we approach θ_i by the posterior distribution

$$\underbrace{p(\theta_i|\mathbf{y})}_{\text{posterior distribution}} = \frac{p(\mathbf{y}|\theta_i)p(\theta_i)}{\int p(\mathbf{y}|\theta_i)p(\theta_i)d\theta_i} \propto \underbrace{p(\mathbf{y}|\theta_i)}_{\text{likelihood function}} \times \underbrace{p(\theta_i)}_{\text{prior distribution}}$$

- We get $p(\theta_i|\mathbf{y}) \sim \beta(\text{number of successes} + 1, \text{number of failures} + 1)$
- At each round n we take a random draw $\theta_i(n)$ from this posterior distribution $p(\theta_i|\mathbf{y})$, for each ad i.
- At each round n we select the ad i that has the highest $\theta_i(n)$.



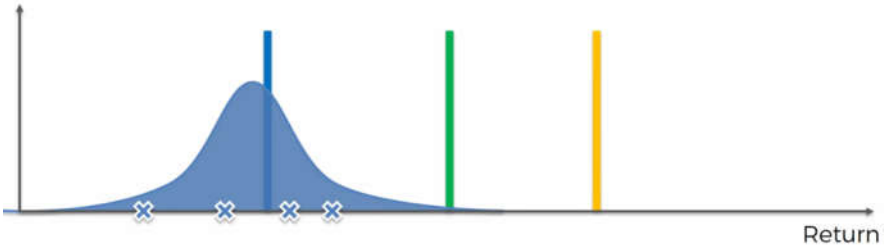
☞ So here we've got a scale. For simplicity we consider ***3 bandits***. ***Vertical*** line represents the ***expected value*** and ***horizontal*** line represents ***return***.

☞ Those **colored-vertical lines** are **central distribution** i.e. the real expected values of the machine. The algorithm of course doesn't know about those.

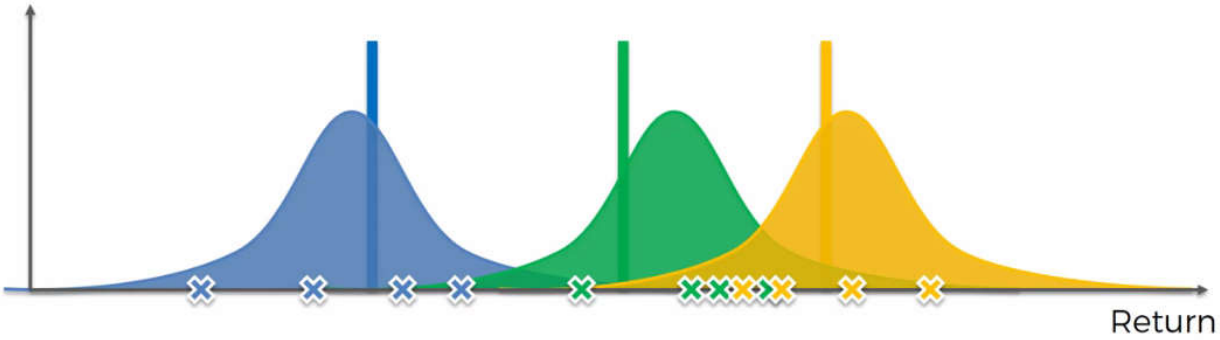
❒ At first all the machines are considered as similar. You have to have at least ***one*** or even a ***couple*** of ***trial*** rounds just to get ***some data*** to ***analyze***.

☞ Say we run some trail in the blue machine, then the algorithm tries to make a distribution





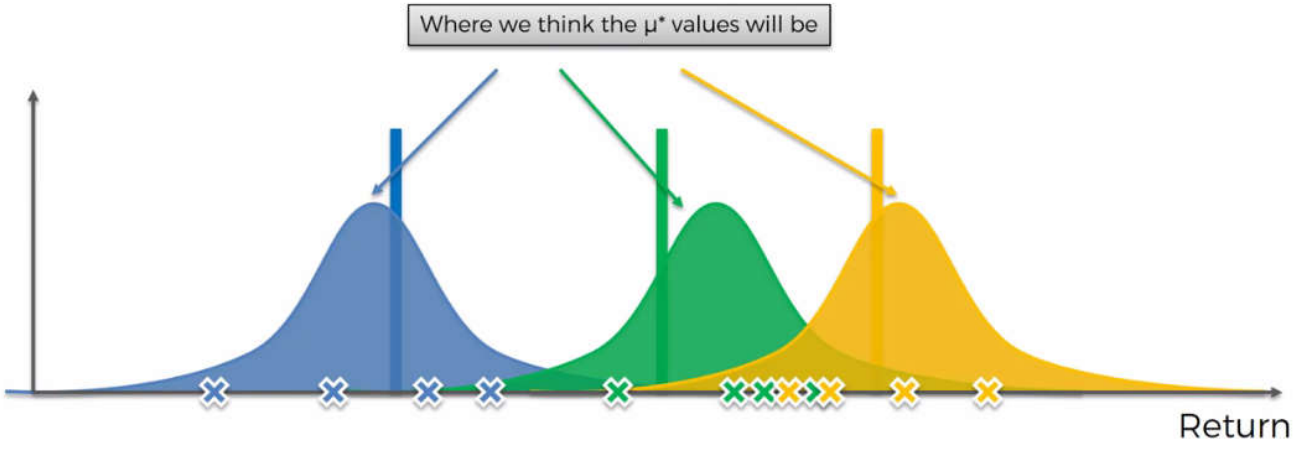
☞ So we do the same things for other two, we pull the machines arm several times and draw a distribution for each machine.



☝ The actual meaning of this these distributions is not what you might think at first, these distributions are ***not trying to guess*** the ***distributions behind*** the ***machines***.

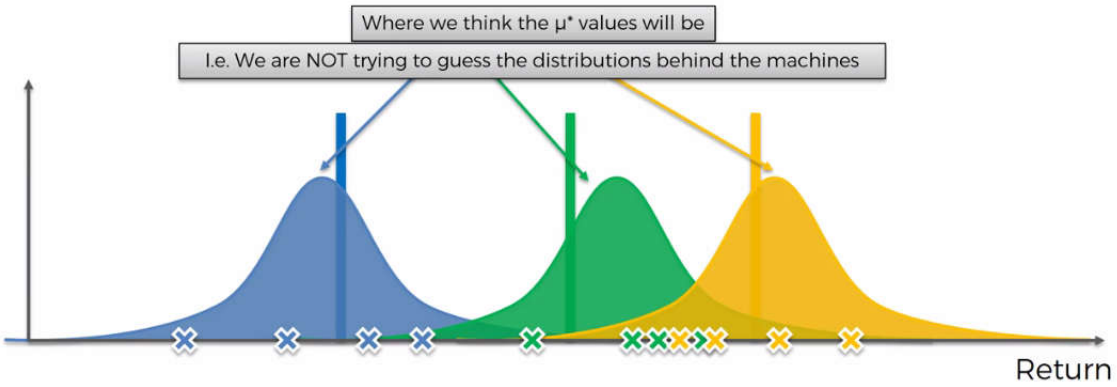
The first thing that might come to mind is that (for instance): the *blue distribution* is *attempting* to *guess* the *actual distribution* behind the *blue machine* right. Or, same for the green or Orange !!!

☞ ***But in reality they are not doing that:*** we are actually ***constructing distributions*** of where we think the actual ***expected value might lie***.



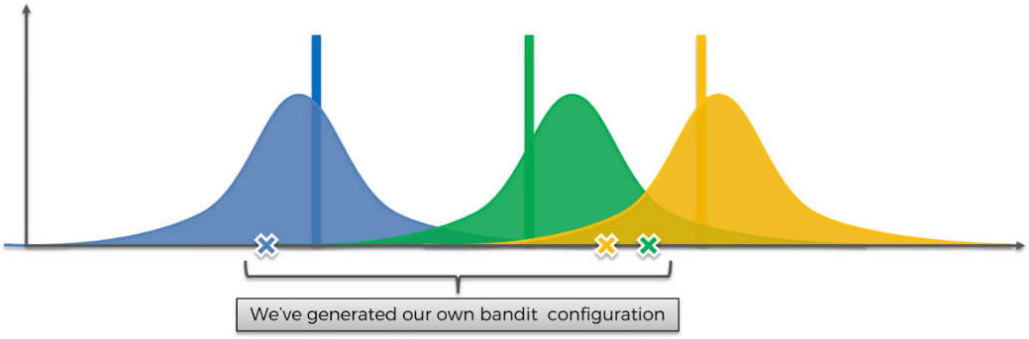
☝ We're creating an ***auxiliary mechanism*** for us to ***solve*** the ***problem***. So we're not we're **not trying** to **recreate** these **machines** we're recreating the **possible way** these **machines could** have been **created.**

☞ For example consider the ***four Blue-dots***, they are the **possible position** for the actual **Blue-colored-vertical line** or **central distribution**. And based on those ***four*** values we've constructed this ***blue-distribution*** which will showing us ***possible positions*** for that value μ^*. It shows the ***high likelihood*** or ***low likelihood*** for the ***position*** of the actual ***Blue-Vertical*** line. And same goes for the ***Green*** and ***Orange distribution***.

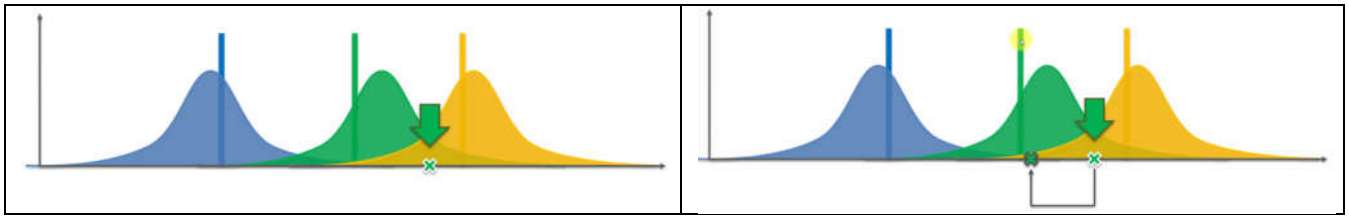


☝ Hence the ***Thompson Sampling*** is Kind of a ***Probabilistic algorithm***. Where Upper-bound is a ***Deterministic Algorithm***.

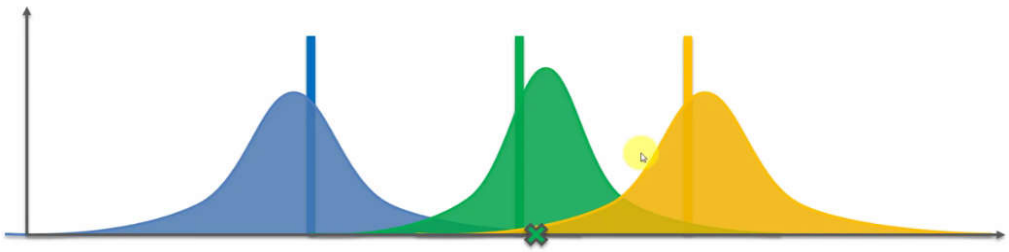
❒ So in simple word, we've created our own ***Bandit-Configuration***, where we know the all distribution. It is some kind of imaginary distribution.



☞ Now from above three points we choose the ***green one*** (because it lies in the ***right side*** of the ***imaginary central tendency***, hence it **may** give **positive result**). According to our own ***Bandit-Configuration***, we generate a value and we ***compare*** that *value* with the *value generated in the real world*. Then our sample grows and we make correction to our own version of distribution.



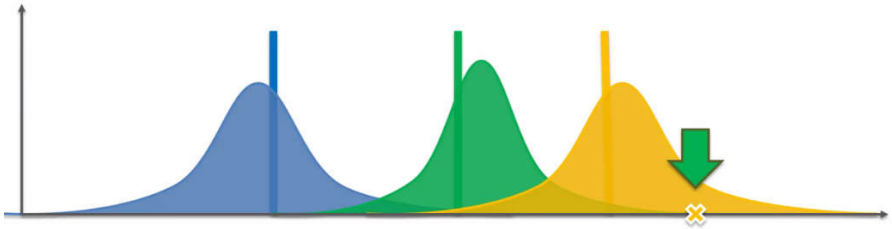
☞ So our green version ***shifted*** a little bit and got ***narrow*** (because sample is increased) and ***increased*** little higher. These happens because in our real world value lies in the ***left side*** of the ***imaginary central tendency,*** i.e. we get the ***wrong expected value*** in our ***imaginary distribution*** (however we were lucky to got positive value in real world). Hence we shifted the ***imaginary distribution*** to the ***left***. Since the ***sample grows*** bigger the distribution ***narrowed*** and grows ***higher***.



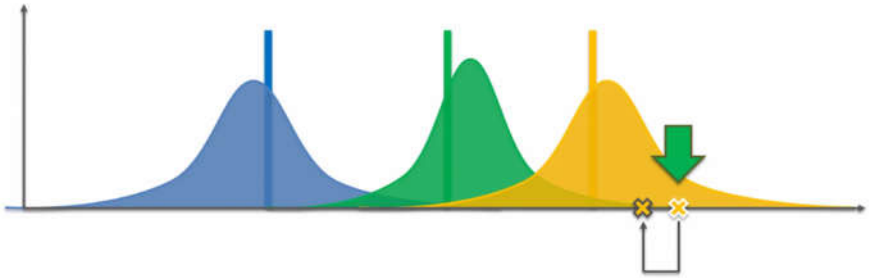
☞ That's the ***point*** that ***every time we add New INFORMATION*** our distribution becomes more and more ***REFINED***.

❒ ***New Round:*** Similarly we pick three values (randomly ?) for three ***Blue***, ***Green*** and ***Orange*** as the ***expected Return***, for our new **imaginary Version of Bandit Problem**. Out of these three we pick the best bandit. Which is in this new round, in our imaginary is ***Bandit-Configuration***, the ***Orange*** one:





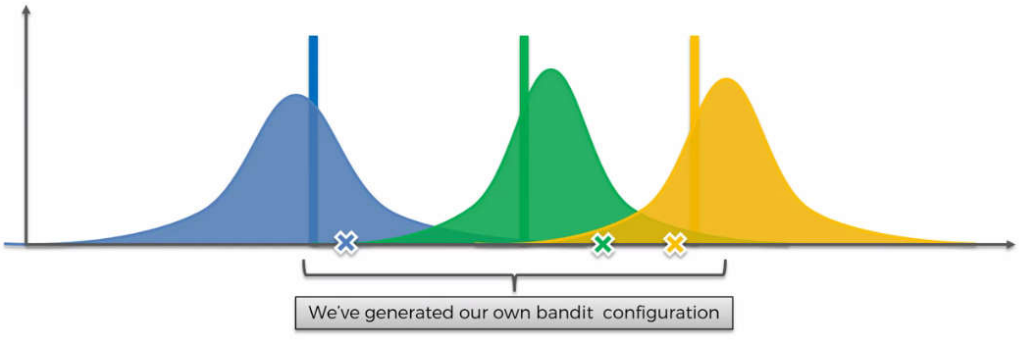
☞ Now we ***generate*** the ***real value*** by pulling the ***hand*** of the ***Orange machine***, we get the following result:

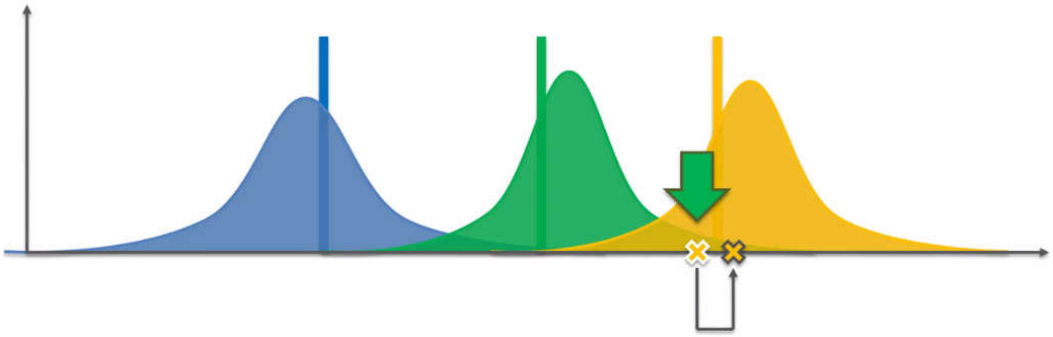


☞ Then we refine our Orange distribution.

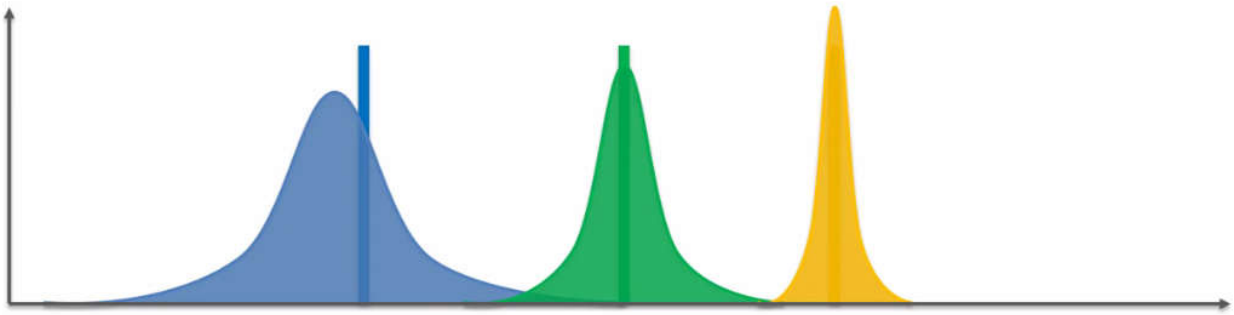


❒ ***Another Round:*** Now for another round we get the following:





❒ After certain amount of Round we end up like follows. Where we used ***Orange machine, more and more*** because of the ***correct prediction***. That's how this ***Thompson sampling*** algorithm ***converges*** to the correct ***best distribution***. For this reason the ***Green*** and ***Blue*** distribution ***doesn't get refined*** as ***Orange*** one.



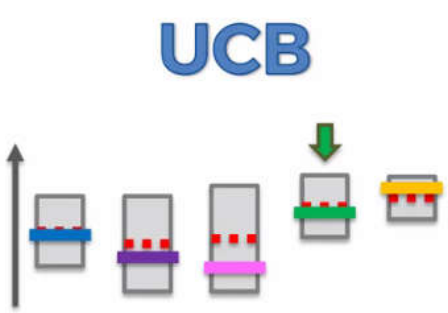
☞ Which is totally fine because our point is to ***Get*** to the ***Best Machine*** to find it and ***Exploit*** it ***As Much As We Can***.

☝ Every time we're generating these values, and they are kind of creating this ***hypothetical set up of the bandits*** and then we're ***solving that*** and then we're applying the ***results*** to the ***real world***.

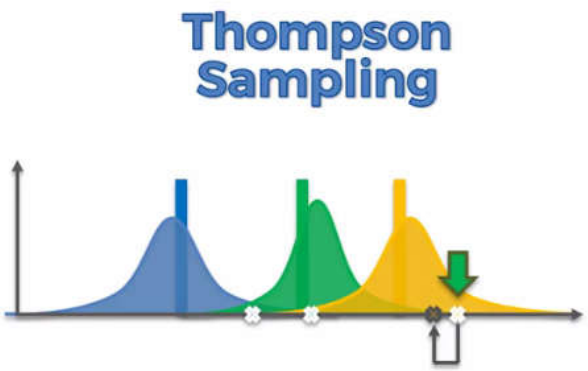
➲ We're adjusting our **perception** of **reality** based on the **new information** that **generates** and then we're repeating the whole process.

6.2.2 UCB vs Thompson Sampling

We're going to compare the two because they do solve the same problem ***"The Multi Armed Bandit"*** and let's have a look at some of the ***pros*** and ***cons*** of each of the ***algorithms***.



- Deterministic
- Requires update at every round



- Probabilistic
- Can accommodate delayed feedback
- Better empirical evidence

i. ***UCB*** is a **deterministic** algorithm, there's lots of different modifications to these algorithm, all belongs to a family of UCB. They are all deterministic and basically what that means is that it is very straightforward.

- ☞ So once you have *certain round* you just look at the *upper confidence* and you going to Pick highest.
- ☞ You pull the lever then you do get a *random value from* the *machine*. But that's on the *side* of the *machine*, when we get the value it is very ***determined***. There's no ***randomness*** in the ***algorithm itself***.

ii. On the other hand the ***THOMPSON SAMPLING ALGORITHM*** is a **probabilistic** algorithm because in that there are distributions which represent our perception of the world and where we think the actual expected returns of each of those machines might lie.

- ➲ And therefore every time we are *implementing* or *iterating* in the *Thompson Sampling Algorithm* we actually *generate random* values from those *distributions*.
- ➲ So it's always going to be *different* because you're always *sampling* from your *distributions* which *characterize* your *perception* of the *world* and that is a whole different type of algorithm. It's a probabilistic algorithm.

iii. UCB requires an update at every round (for each round). So once you've pulled the lever and you get a *value back* from that *machine* that value you have to *incorporate* it right away in order to *proceed to the next* round. You cannot proceed to the next round until you have incorporated that value.

- ☞ Until you have made an adjustment to the algorithm based on that value because if you *don't make the adjustment* then nothing changes and you're going to be *stuck*.

iv. Whereas in the Thompson Sampling Algorithm can *accommodate delayed feedback* and it's very important. This basically means that if you pull the lever and you only keep record of the results (say, pulling the lever 500 rounds), and input those data to the Thompson sampling, Thompson Sampling Algorithm will still work.

- ☞ Why will it work because if you now run the algorithm without even updating your perception of the world you're still going to get a *new set of hypothetical bandits*. Because you are generating them in a probabilistic manner.
- ☞ And this is very important to understand because this gives the ***Thompson sampling*** that ***advantage*** that you ***don't have to update the algorithm with the result every time***.

- ☝ In terms of just terms of ***Bandits*** it doesn't really matter that much because if you're playing in the casino and out of sight if some hypothetical person is playing in a casino and they're pulling these lever's they get to the results right away. So they could update Algorithm.
- ☝ But in terms of Web sites and ads that is a big deal. Just not even just ***displaying ads*** on a Web site or you could use this for like ***A/B testing*** the different layouts of your Web site right. You could you could use a Thompson sampling algorithm to have that ***balance*** between ***exploitation*** and ***exploration***
- ☝ This sampling algorithm allows you to do is to update your dataset or your information algorithm in a batch manner.

v. ***Thompson sampling algorithm*** is actually it has ***better empirical evidence*** than they used to be.

Hence we can conclude that ***Thompson sampling algorithm*** would be better choice.

Thompson Sampling Algorithm

Step 1. At each round n, we consider two numbers for each ad i:

- $N_i^1(n)$ - the number of times the ad i got reward 1 up to round n,
- $N_i^0(n)$ - the number of times the ad i got reward 0 up to round n.

Step 2. For each ad i, we take a random draw from the distribution below:

$$\theta_i(n) = \beta(N_i^1(n) + 1, N_i^0(n) + 1)$$

Step 3. We select the ad that has the highest $\theta_i(n)$.

6.2.3 Implementation of Thompson sampling algorithm in Python

We introduced a multi armed benefit program for an Ad click through rates (CTR) optimization problem.

❒ We observed the ***Total Reward*** for Random Selection algorithm (1200 on average, every ad was selected more or less the same number of times) and UCB algorithm. Clearly UCB did the great job (with reward 2170 and we figured out 5th Ad version was the best). Now this **Thompson sampling algorithm** is even better than ***UCB***. Because it will figure out which version was the best more quickly (so the reward will be high).

❒ ***So we will observe:***

[1] How better **Thompson sampling** is w.r.t ***UCB*** according to **total reward**

[2] Which Ad is selected for Maximum Exploitation

❒ We will ***change*** the ***previous code*** for ***UCB*** rather than writing it from ***scratch***.

a) ***Step 1 :*** First we create the two variables:

- $N_i^1(n)$ = No. of times ***Ad_i*** get reward **1** up to round n i.e. No. of ***REWARDS***
- $N_i^0(n)$ = No. of times ***Ad_i*** get reward **0** up to round n i.e. No. of ***PUNISHMENTS***

```
"""
numbers_of_selections = [0]*d
numbers_of_rewards = [0]*d

changed to

        # no. of reward of Ad i up-to trial n i.e. reward = 1,
        # No. of punishment of Ad i up-to trial n i.e. reward = 0,
"""
reward_count_of_ads = [0]*d
punish_count_of_ads = [0]*d
```

⇨ As before we created ***two vector/list of 10 elements*** (initialized to **0**). These 2 vectors will keep track to the ***Rewards*** and ***Punishments*** for each ***10 Ads*** up to trial/round ***n***.

b) ***Step 2:*** For each Ad_i we take a random draw from following distribution, which is the ***beta distribution***.

$$\theta_i(n) = \beta\left(N_i^1(n) + 1, \quad N_i^0(n) + 1\right)$$

☞ We have two important assumptions here which are related to Bayesian inference.

Bayesian Inference

- Ad i gets rewards $\mathbf{y}$ from Bernoulli distribution $p(\mathbf{y}|\theta_i) \sim \mathcal{B}(\theta_i)$.
- θ_i is unknown but we set its uncertainty by assuming it has a uniform distribution $p(\theta_i) \sim \mathcal{U}([0,1])$, which is the prior distribution.
- Bayes Rule: we approach θ_i by the posterior distribution

$$\underbrace{p(\theta_i|\mathbf{y})}_{\text{posterior distribution}} = \frac{p(\mathbf{y}|\theta_i)p(\theta_i)}{\int p(\mathbf{y}|\theta_i)p(\theta_i)d\theta_i} \propto \underbrace{p(\mathbf{y}|\theta_i)}_{\text{likelihood function}} \times \underbrace{p(\theta_i)}_{\text{prior distribution}}$$

- We get $p(\theta_i|\mathbf{y}) \sim \beta(\text{number of successes} + 1, \text{number of failures} + 1)$
- At each round n we take a random draw $\theta_i(n)$ from this posterior distribution $p(\theta_i|\mathbf{y})$, for each ad i.
- At each round n we select the ad i that has the highest $\theta_i(n)$.

[1] So the first assumption is this first line here we suppose that each ***Ad_i*** gets ***y*** rewards from a ***Bernouli distribution*** of parameter $\boldsymbol{\theta_i}$. Where $\boldsymbol{\theta_i}$ is the ***probability of success***.

❖ And you can picture this ***probability of success*** by showing the Ad to a **huge amount of users** like **1000000** users and θ_i could be interpreted as the **number of times the outcomes** were ***1*** (i.e. the number of successes) divided by the **total number of times** we selected the Ad that is 1 million.

$$\theta_i = \frac{number\ of\ times\ the\ outcomes\ were\ 1}{total\ number\ of\ times\ we\ selected\ the\ Ad}$$

So basically $\boldsymbol{\theta_i}$ is the **probability of success** that is **the probability of getting *Reward 1*** when we select the Ad.

[2] ***Second Assumption:*** [Recall ***Bayes's Theorem*** in 3.5.1 of Naive Bayes] The second assumption are less stronger than the first one. We assume that θ_i has a ***uniform distribution*** which has the **prior distribution** $p(\theta_i)$.

[3] Then we use the ***Bayes Rule*** to get to **posterior distribution** which is $p(\theta_i|y)$. Where y given the rewards that we got up to the round **n**.

☞ So by using ***Bayes rule*** that's how we get this β***-distribution*** in the ***step 2***.

$$p(\theta_i|y) = \beta(number\ of\ success + 1, \quad number\ of\ failure + 1)$$

☞ So at each ***n round*** we take a ***random draw*** from this ***β-distribution***. Since these ***random draws*** represent the ***probability of success*** θ_i, we get this strategy: the ***maximum*** of these ***random draws*** is ***approximating*** the ***highest probability of success***. i.e. At each round n, we select the Ad_i that has the height ***probability of success*** θ_i: $\theta_i(n)$.

☝ That's the whole idea behind Thompson Sampling: We are trying to

- Estimate these parameters $\theta_i, \theta_1, \theta_2, \ldots, \theta_{10}$, the **probabilities of success**, of each of these ***10 Ads*** then
- By taking these ***random draws*** and taking the ***highest of them*** we're estimating the ***highest probability of success*** and this ***highest probability of success*** corresponds to ***one Specific Ad at each round***.
- So for small amount of round, we might be wrong, but when we take these ***random draws*** over ***thousands of rounds*** we obtain over all the θ_i that corresponds to the ***Ad*** that has the ***Highest Probability Of Success*** (highest probability of getting reward = 1).

c) ***Step 3:*** What we just did. Taking these maximum of these random draw that is the maximum of these estimations of the probability of getting reward equals 1.

☞ Now we'll implement the strategy composed of ***step 2*** and ***step 3*** in python. Actually it is easier than UCB, we just need to use one method "***random.betavariate()***" .

⇨ This ***random.betavariate()*** will calculate the ***β- distribution*** for each i-th ***Ad*** (i.e each of 10 Ad). Then we select the ***Ad_i*** which has maximum value of the calculated ***β-distribution***.

```
for n in range (0, N):
    slct_ad = 0
    max_random_beta = 0

    for i in range (0, d):
        random_beta = random.betavariate(reward_count_of_ads[i]+1, punish_count_of_ads[i]+1)

        if  random_beta > max_random_beta:
            max_random_beta = random_beta
            slct_ad = i
```

⇨ Next we simulate reward/punish situation and then update Reward and Punish for Each Ad. Also we append the selected Ad to out 10000 long ***slected_ads*** list.

```
for n in range (0, N):
    slct_ad = 0
    max_random_beta = 0

    for i in range (0, d):
        random_beta = random.betavariate(reward_count_of_ads[i]+1, punish_count_of_ads[i]+1)

        if  random_beta > max_random_beta:
            max_random_beta = random_beta
            slct_ad = i

    # storing selected Ad
    slected_ads.append(slct_ad)

    # updating "reward_count_of_ads" and "punish_count_of_ads" of selected Ad "slct_ad"
    reward = dataSet.values[n, slct_ad] # generating reward-simulation from given Dataset
    if reward == 1:
        reward_count_of_ads[slct_ad] += 1
    else:
        punish_count_of_ads[slct_ad] += 1

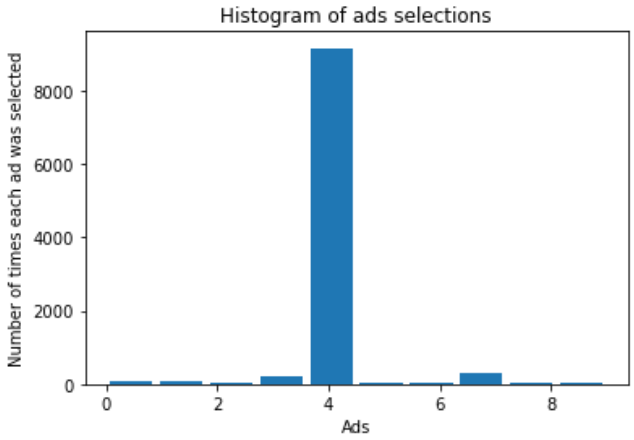
    total_reward += reward
```

☞ ***Now we visualize our result:***

```
#visualizing the result
plt.hist(slected_ads, rwidth=0.85)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()
```

☞ ***Moment of truth:*** We conclude that the ***Thompson Sampling Algorithm*** is better than ***UCB Algorithm***. It finds the same ***Ad*** (5th) from ***UCB*** but more quickly. Also it doubles the ***Total reward*** from the ***Random Selection***.

reward_count_of_ads	list	10	[9, 8, 0, 36, 2481, 0, 3, 62, 2, 2]
slct_ad	int	1	4
slected_ads	list	10000	[5, 3, 4, 6, 3, 1, 1, 0, 0, 7, ...]
total_reward	int64	1	2603



Practiced Version

```
# Reinforcement lrarning : ----------  Thompson Sampling. ---------
#  Ad Click through rate optimization

import pandas as pd
import numba as np
import matplotlib.pyplot as plt


# importing dataset
dataSet = pd.read_csv("Ads_CTR_Optimisation.csv")


# Thompson Sampling Implemnetation.
"""    # 2 parameters are important:
        # no. of bandit d,
        # No. of trials N,
        """

import random
# here d is No. of Ads or Number of Bandits
d = 10
N = 10000   # Total number of trials

"""
numbers_of_selections = [0]*d
numbers_of_rewards = [0]*d

changed to

        # no. of reward of Ad i at trial n i.e. reward = 1,
        # No. of punishment of Ad i at trial n i.e. reward = 0,
"""
reward_count_of_ads = [0]*d
punish_count_of_ads = [0]*d
```

```
slected_ads = []
total_reward = 0


for n in range (0, N):
    slct_ad = 0
    max_random_beta = 0

    for i in range (0, d):
        random_beta = random.betavariate(reward_count_of_ads[i]+1, punish_count_of_ads[i]+1)

        if  random_beta > max_random_beta:
            max_random_beta = random_beta
            slct_ad = i

    # storing selected Ad
    slected_ads.append(slct_ad)

    # updating "reward_count_of_ads" and "punish_count_of_ads" of selected Ad "slct_ad"
    reward = dataSet.values[n, slct_ad] # generating reward-simulation from given Dataset
    if reward == 1:
        reward_count_of_ads[slct_ad] += 1
    else:
        punish_count_of_ads[slct_ad] += 1

    total_reward += reward


#visualizing the result
plt.hist(slected_ads, rwidth=0.85)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()


# python prctc_tmpsn_sml.py
```

Instructor Version

```
# Thompson Sampling

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Ads_CTR_Optimisation.csv')

# Implementing Thompson Sampling
import random
N = 10000
d = 10
ads_selected = []
numbers_of_rewards_1 = [0] * d
numbers_of_rewards_0 = [0] * d
total_reward = 0
for n in range(0, N):
    ad = 0
    max_random = 0
    for i in range(0, d):
        random_beta = random.betavariate(numbers_of_rewards_1[i] + 1, numbers_of_rewards_0[i] + 1)
        if random_beta > max_random:
            max_random = random_beta
            ad = i
    ads_selected.append(ad)
    reward = dataset.values[n, ad]
    if reward == 1:
        numbers_of_rewards_1[ad] = numbers_of_rewards_1[ad] + 1
    else:
        numbers_of_rewards_0[ad] = numbers_of_rewards_0[ad] + 1
    total_reward = total_reward + reward

# Visualising the results - Histogram
plt.hist(ads_selected)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()
```

Chapter 7

Natural Language Processing

Here we will learn ***how*** to ***clean text*** to prepare them for ***machine learning models***.
How to create a ***Bag Of Words Model*** and apply ML models onto this ***Bag Of Words*** model.

7.1 NLP: Natural Language Processing

❒ ***NLP:*** ***Natural Language Processing*** or ***NLP*** is an area of **Computer Science** and **Artificial Intelligence** concerned with ***interactions*** between ***computers*** and ***human*** through ***natural languages***.

- ☞ NLP is used to apply ***Machine Learning*** models to ***text*** and ***language***.
- ☞ Teach machines to understand what is said in ***spoken*** and ***written*** word is the focus of ***Natural Language Processing***. Whenever you dictate something into your **iPhone** / **Android** device that then **converted** to text, that's an **NLP algorithm** in **action**.
- ☞ You can also think of Apple's ***Siri*** or Amazon's ***Alexa***. These are NLP related algorithms, which is processing your language and converting them into data. Making possible to run the algorithm for the desired purposes.

❒ ***NLP history:*** The history of ***Natural-Language Processing*** generally started in the 1950s, although work can be found from earlier periods. In 1950, ***Alan Turing*** published an article titled ***"Computing Machinery and Intelligence"*** which proposed what is now called the ***Turing Test*** as a criterion of intelligence.

- ☞ Up to the 1980s, most ***`natural-language processing systems`*** were based on ***complex*** sets of ***hand-written rules***. Starting in the late 1980s, however, there was a *revolution in natural-language processing* with the introduction of **Machine Learning Algorithms** for **Language Processing**.

- 📣 But even more recent the ***research*** has ***focused*** on ***unsupervised*** and ***semi supervised*** learning algorithm (especially as of lately a huge push into ***Deep Learning*** techniques for NLP). *Research in this area is* ***`highly active`*** *and it's an exciting time for* ***NLP*** *related* ***`algorithms`***.

7.2 Uses of NLP

☝ ***Sentiment analysis:*** Identifying the mood or subjective opinions within large amounts of text, including average sentiment and opinion mining. We are going to be building a similar model in this section with ***restaurant reviews*** being ***positive*** or ***negative***.

☝ ***Use it to predict the genre of the book:*** You can also use it to predict the genre of a book.

☝ ***Question Answering:*** use NLP for question and trade with Chatbot.

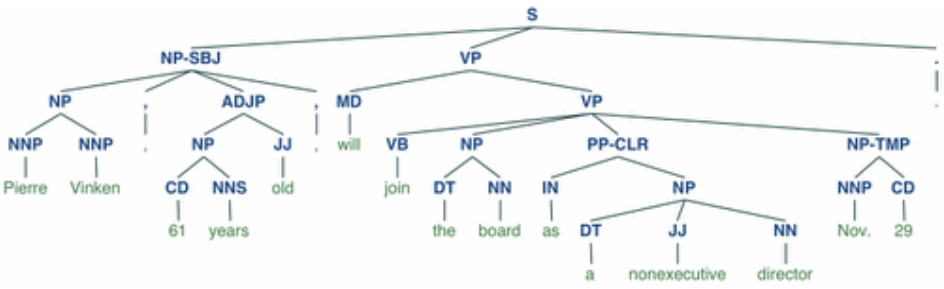
☝ ***Translator:*** Use NLP to build a machine translator or a speech recognition system

☝ ***Document Summarization***

7.3 NLP Libraries

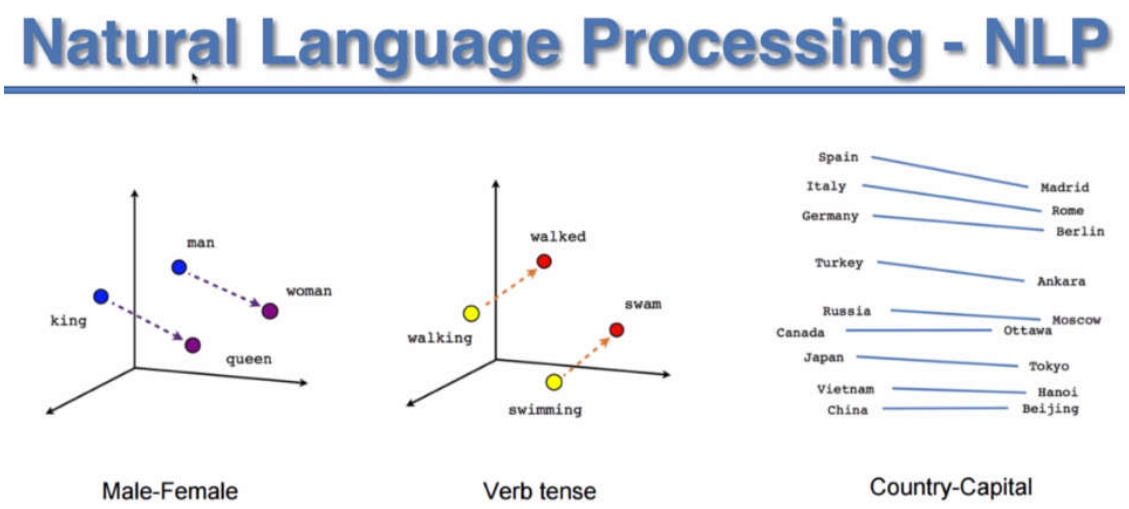
We also have a range of NLP related libraries that include the following examples:

[1] Natural Language Toolkit - NLTK
[2] SpaCy
[3] Stanford NLP
[4] OpenNLP



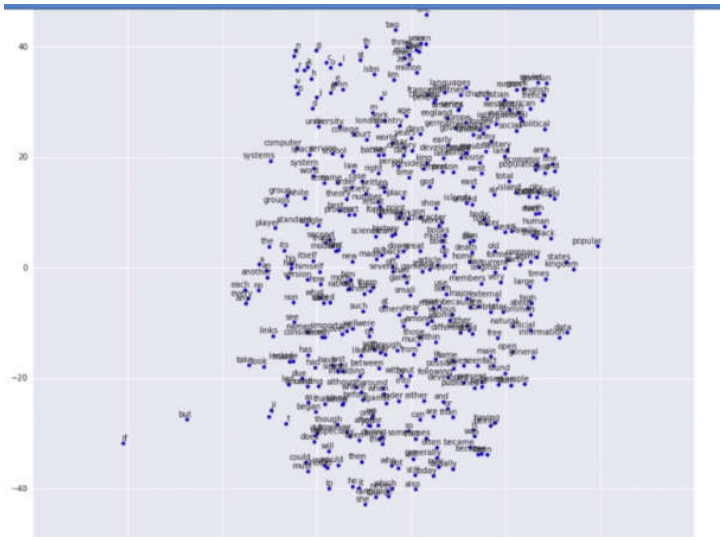
❒ ***Example of NLTK:*** We need to take a look at what NLTK can do by breaking down a sentence. In this above example we can see the ***POS*** or ***part of speech tree***. It's breaking down the sentence. You can see for example we have a ***determiner*** or ***DT***. you have the ***cardinal number CD*** and an example of an ***adjective*** for the ***JJ*** tag.

☞ You can run operations such as here to visualize ***semantic information*** about ***words*** and their relationships to one another.



https://www.tensorflow.org/tutorials/word2vec

☞ You can also visualize learned embeddings or have words that are similar **cluster nearby each other**.



❒ ***NLP - Bag of Words:*** For the purpose of the practical work in this section you'll be working with the **bag of words**.

📣 ***Bag of Words:*** A model used to ***preprocess the texts*** to classify before fitting the ***classification algorithms*** on the observations containing the text.

📣 It is Very popular ***NLP model*** - It is a model used to ***preprocess*** the ***texts*** to ***classify*** before fitting the ***classification algorithms*** on the observations containing the texts. It involves two things:

1. A ***vocabulary*** of ***known words***.
2. A ***measure*** of the ***presence*** of ***known words***.

The things will learn in this section are give below:

1. ***Clean texts*** to prepare them for the ***Machine Learning models***,
2. Create a ***Bag*** of ***Words model***,
3. Apply ***Machine Learning*** models onto this ***Bag*** of ***Worlds*** model.

7.5 CSV files vs TSV files

❒ ***CSV:*** A **comma-separated values (CSV)** file is a ***delimited text file*** that uses a comma to separate values. Each line of the file is a ***data record***. Each ***record*** consists of one or more ***fields***, separated by ***commas***. The use of the comma as a field separator is the source of the name for this file format. A CSV file typically stores ***tabular data*** (numbers and text) in ***plain text***, in which case each line will have the same number of fields.

☞ The CSV file format is not fully standardized. Separating fields with commas is the foundation, but commas in the ***data*** or ***embedded line breaks*** have to be handled specially. Some implementations disallow such content while others ***surround the field with quotation marks***, which yet again creates the need for escaping if quotation marks are present in the data.

☞ The term "***CSV***" also denotes several closely-related ***delimiter-separated formats*** that use other field ***delimiters*** such as ***semicolons***. These include ***tab-separated values*** and ***space-separated values***. A delimiter guaranteed not to be part of the data greatly simplifies ***parsing***.

❒ ***TSV:*** A ***tab-separated values (TSV)*** file is a simple text format for storing data in a ***tabular structure***, e.g., a ***database table*** or ***spreadsheet*** data, and a way of ***exchanging information*** between ***databases***. Each record in the table is one line of the text file. Each field value of a record is separated from the next by a tab character. The TSV format is thus a variation of the comma-separated values format.

☞ TSV is a simple file format that is widely supported, so it is often used in data exchange to move tabular data between different computer programs that support the format. For example, a TSV file might be used to transfer information from a database program to a spreadsheet.

\n for **newline**,

\t for **tab**,

\r for **carriage return**,

**** for **backslash**

❒ ***Delimiter:*** A ***delimiter*** is a sequence of ***one or more characters*** for specifying the ***boundary between*** separate, independent regions in ***plain text***, mathematical ***expressions*** or other ***data streams***. An example of a delimiter is the ***comma*** character, which acts as a field delimiter in a sequence of ***comma-separated values***. Another example of a delimiter is the ***time gap*** used to separate ***letters*** and ***words*** in the ***transmission*** of ***Morse code***.

Example: The most common CSV escape format uses **quotes** to **delimit fields** containing ***delimiters***. Quotes must also be escaped, this is done by using a pair of quotes to represent a single quote. Consider the data in this table:

Field-1	Field-2	Field-3
abc	hello, world!	def
ghi	Say "hello, world!"	jkl

☯ In ***Field-2***, the first value contains a ***comma***, the second value contains ***both quotes and a comma***. Here is the CSV representation, using escapes to represent commas and quotes in the data.

```
Field-1,Field-2,Field-3
abc,"hello, world!",def
ghi,"Say ""hello, world!""",jkl
```

☯ In the above example, only fields with delimiters are **quoted**. It is also common to quote all fields whether or not they contain delimiters. The following CSV file is equivalent:

```
"Field-1","Field-2","Field-3"
"abc","hello, world!","def"
"ghi","Say ""hello, world!""","jkl"
```

☯ Here's the same data in ***TSV***. It is ***much simpler*** as ***no escapes*** are ***involved***.

```
Field-1      Field-2      Field-3
abc    hello, world! def
ghi    Say "hello, world!" jkl
```

❒ ***TSV or CSV in NLP:*** Now the question is do we need a data-set where the columns are separated by a ***Comma*** or by a ***Tab***. We used csv for previous ML models but in NLP we are going to use TSV. The reason is:

➲ Because it *doesn't create any problem* with ***commas***, ***quotes*** and hence we don't need any ***escapes***.

```
Review     Liked
Wow... Loved this place.   1
Crust is not good.    0
Not tasty and the texture was just nasty.  0
```

☯ ***For example:*** Our dataset is in tsv format like above. ***0*** or ***1*** means ***negative*** or ***positive***. Reviews like following in CSV can cause problem:

```
Wow, this place is amazing!!!, 1
```

➲ It is one sentence, but due to "**,**" spate ***Wow*** and ***this place is amazing !!!*** it makes them to be in ***different fields***. Then ***1*** goes to the ***next field***. Hence algorithm doesn't work.

➲ Also we can use double-quote, but Reviews can also contain Double-quotes susc as:

```
"Good lord !!"  , The place is Horrible., 1
```

☯ It's way better to take ***tabs*** here because you know when people ***write reviews*** they ***do not put tabs in the review***. That would be very rare.

➲ Because by pressing the ***Tab button*** when you're writing your review you would ***not be able to continue to write*** it. So we will ***never find a tab in the review*** and that's why we will *never have this problem of getting these anomalies due to* ***duplicate delimiters*** in one specific review.

☝ To avoid those problems we use ***tsv*** files. It makes easier to import dataset. So in NLP it is recommend to prepare your **text datasets** using tab-separator (TSV).

7.4 NLP in python

Natural Language Processing (NLP) is a branch of machine learning where we do some ***predictive analysis*** on ***text*** mostly. NLP is about ***analyzing text***. These texts can be ***books reviews***, some ***HTML*** web pages that you extract from ***web-scrapping***, all sorts of Texts.

📣 ***Problem Description:*** Here we will analyze some ***written reviews of restaurants***. So we will make some ***ML models*** that will ***predict*** if the review is ***positive*** or ***negative*** (according to 1 or 0 in our tsv file).

- The algorithm we will making this part is a general model, it will be very well applicable to other kinds of text such as:
- To predict the ***Genre*** of a book whether it is ***thriller***, ***comedy*** or ***romance***.
- Use it on ***HTML*** web pages to do whatever kind of analysis you want to do on those pages.
- You can also play it on ***newspapers*** you know to predict in which ***category*** an ***article*** belongs to.

we need to add some parameters because of course we are importing a ***tsv*** file whereas this ***read_csv*** function by ***Pandas*** is expecting a ***csv***.

```
dataset = pd.read_csv("Restaurant_Reviews.tsv", delimiter = "\t", quoting = 3)
```

delimiter: This parameter ***delimiter = "\t"*** specify that it's ***not*** the ***comma ","*** but the **tab "\t"** `delimiter` .

Pandas will know that our two columns are separated by a tab.
quoting: But we will not stop here because when looking at the data sets we saw that we might have some other problems related to the ***double quotes "***. That's why we used ***quoting = 3***, to make sure that we won't have any problem with these double quotes.

We need to make sure that we have our 1000 reviews.
Also make sure that we don't have any ***review*** in the ***like column*** or a ***1 or 0*** in the ***review column***.

```python
# NLP: Natural Language Precessing

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# -------------- Data preprocessing -------------
# importing dataset
        # since we are using "tsv" ins tead of "csv" we need to specify some parameters.
        # Because "Pandas" expecting some "csv" files
dataset = pd.read_csv("Restaurant_Reviews.tsv", delimiter = "\t", quoting = 3)
```

dataset - DataFrame

Index	Review	Liked
0	Wow... Loved this place.	1
1	Crust is not good.	0
2	Not tasty and the texture was just nasty.	0
3	Stopped by during the late May bank holiday off Rick Steve recommendation and loved it.	1
4	The selection on the menu was great and so were the prices.	1
5	Now I am getting angry and I want my damn pho.	0
6	Honeslty it didn't taste THAT fresh.)	0
7	The potatoes were like rubber and you could tell they had been made up ahead of time being kept under a warmer.	0
8	The fries were great too.	1

7.4.1 Clean the Data

Next step is to clean the different reviews. This is a ***compulsory*** step in ***natural language processing (NLP)*** which consists of ***cleaning the text*** to make it **ready** for our **future machine learning algorithms**.

❒ We have to do this because in the end we will create a ***bag of words model*** or **bag of words representation** and this will consist of getting ***only the relevant words*** and the different reviews here.

☞ That means that we'll **get rid of** all the ***unnecessary words***. EG: The, on, am etc. and these are not relevant words because these are not the words that will help the machine learning algorithm to predict if the review is positive or negative.

☞ ***Punctuation:*** We will also get rid of the ***punctuation*** like ***"..."***.

☞ ***numbers :*** We will also get rid of ***numbers*** unless numbers can have a significant impact.

☞ ***Stemming:*** And we will also do what's called ***stemming*** which consist of taking the ***roots*** of some ***different versions*** of a ***same word***.

- For example we have this "**Loved**" word here **past tense** of the verb **love** and so ***apply stemming on this word*** will consist of only taking ***love*** here because whether we have ***loved*** or ***love*** that gives the ***same hints*** on whether the review is ***positive*** or ***negative***.
- We need to apply ***stemming*** in order **not to have too many words** in the end. And to ***regroup*** the ***same versions*** of a same word like ***love*** and ***love*** or even ***loving*** into a same word ***love***.

☞ ***Capitals:*** we will only have the reviews in lower text.

☞ ***Tokenization:*** Once all the reviews are ***cleaned***, we'll proceed to the last step of creating our **Bag Of Words Model** which is the ***tokenization process***.

- It splits all the different reviews into different words (only the relevant words because we did text pre-processing).
- And then we'll take ***all*** the ***words*** of the ***different reviews*** and we will ***create one column for each word*** (we'll have a lot of columns).
- And then for each **review**, each **column** will contain the **number of times** the associated **word appears** in the ***review***. So we'll have a ***lot of 0's*** because for each review a ***lot of words don't appear in the review*** and we'll have a ***couple of 1's*** for the words that appear once/twice in the review.
- We will get here ***sparse matrix*** because we'll get a lot of zeros in the sparse matrix because for all the reviews ***most*** of the ***words*** will ***not*** be in the ***reviews*** so we will have to do something about it. That will be the ***last step*** of the creation of our ***bag of words***.

❒ ***Cleaning the text:*** At the beginning, we apply the ***different steps*** of cleaning process on the ***first review***, and then we use ***for –loop*** for the ***rest*** of the ***reviews***. So let's do this let's clean this first review and then we'll clean the rest.

☞ ***Regular Expression:*** Python has a built-in package called ***re***, which can be used to work with ***Regular Expressions***. A ***RegEx***, or ***Regular Expression***, is a sequence of characters that forms a ***search pattern***. ***RegEx*** can be used to check if a string contains the ***specified search pattern***. We are going to use it for cleaning text.

```
# Cleaning the text
import re
print(f"{dataset['Review'][0]}")
rev_W = re.sub("[^a-zA-Z]", " ", dataset["Review"][0])
rev_W = rev_W.lower()        #converting to Lower case
```

i. In ***re.sub()***, the 1st parameter means ***remove all characters excluding a-z*** and ***A-Z***. " ***^*** " means ***Excluding***.
ii. The 2nd parameter " " means we want to use a space as a separator.
iii. `dataset["Review"][0]` is the sentence where re.sub() will be applied, it is the first review of our Data-set.

rev_W	str	1	Wow Loved this place

iv. Then we convert the all words into lower case.

rev_W	str	1	wow loved this place

❒ ***Removing non-Significant words:*** That is we are going to remove all the words like: ***the***, ***that***, ***and***, ***in***, all the **articles** all the **propositions**. For example in ***"Wow loved this place"***: ***Loved*** is the **important** word here, this indicates **positive**. But the word "***this***" is not important because it may occur in negative review such as: "***this place sucks!!***" .

☞ Our goal is to avoid too much ***sparsity*** in the ***sparse matrix***.

☞ To do this we will need a new library which is the very famous ***NLTK*** library

➢ ***module nltk:*** The ***Natural Language Toolkit (NLTK)*** is an ***open source Python library*** for Natural Language Processing. A free online book is available. (If you use the library for academic research, please cite the book.)

➢ Steven Bird, Ewan Klein, and Edward Loper (2009). Natural Language Processing with Python. O'Reilly Media Inc. http://nltk.org/book

```
import nltk
nltk.download("stopwords")
```

➢ Now we'll make a for-loop for each review ***" rev_W "*** our first review: "Wow loved this place", to go through all the different words of this ***rev_W***, then **for each** of the **different words** we see if the word is ***present*** in this **stop words list** and if that's the case we ***remove*** the ***word*** from the ***review***.

```
# Cleaning the text
import re
import nltk
# nltk.download("stopwords")
from nltk.corpus import stopwords
print(f"{dataset['Review'][0]}")
rev_W = re.sub("[^a-zA-Z]", " ", dataset["Review"][0])
rev_W = rev_W.lower()    # converting to Lower case
rev_W = rev_W.split()    # converting "string" to "list"
# reView = [wrd for wrd in rev_W if wrd not in stopwords.words("english")]
reView = [wrd for wrd in rev_W if wrd not in set(stopwords.words("english"))]

# import nltk
# from nltk.corpus import stopwords
# print(stopwords.words('english'))
# print(set(stopwords.words('english')))
```

➢ ***nltk.download("stopwords")*** not needed if the package already up-to-date.

➢ We use *list-comprehension*, and ***set*** is used to make a faster algorithm. ***set*** works faster than ***list***:

```
reView = [wrd for wrd in rev_W if wrd not in set(stopwords.words("english"))]
```

In case you have some ***bigger text (eg: Articles or Book review)*** than this ***simple review*** here (because right now you know our review is composed of only 4 words) it is highly recommended to add this set() function here, makes your algorithm work much faster.

☝ ***Corpus:*** This word means a collection of written texts, especially the entire works of a particular author or a body of writing on a particular subject

☝ ***Stopwords:*** A stop word is a commonly used word (such as “the”, “a”, “an”, “in”) that a search engine has been programmed to ignore, both when indexing entries for searching and when retrieving them as the result of a search query.

```
import nltk
from nltk.corpus import stopwords
print(stopwords.words('english'))
```

❒ ***Stemming:*** For example: keeping only the root of the word i.e. ***love*** because whether we have ***love***, ***loved*** or ***loving*** or ***will love*** in a review, gives the same hints whether the review is ***positive*** or ***negative*** so we don't need this word conjugated in the ***past*** or in the ***future*** or in the ***present***.

☞ Because if we keep all the different versions of the same word, then in the end in our ***sparse matrix*** we'll have ***tons of words*** and therefore ***huge sparsity*** and therefore ***our algorithm will have trouble to run*** because we will simply have ***too many columns*** because in the end you know in the ***sparse matrix*** each ***word*** will have its ***own column***. So that's why we are doing some simplification here.

☞ We import the ***PorterStemmer***, then create an object of it. And we apply it to our words in to our ***list-comprehension.***

➢ Notice we changed ***wrd*** into ***prt_stmr.stem(wrd)*** in our list comprehension.

```
from nltk.stem.porter import PorterStemmer
```

and

```
prt_stmr = PorterStemmer()
reView = [prt_stmr.stem(wrd) for wrd in rev_W if wrd not in set(stopwords.words("english"))]
```

☞ And then we join the resulting ***list*** of words into a ***string***

```
reVw = " ".join(reView)
```

Before				***After***			
rev_W	str	1	Wow Loved this place	reVw	str	1	wow love place

```
# Cleaning the text
import re
import nltk
# nltk.download("stopwords")
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
print(f"{dataset['Review'][0]}")
rev_W = re.sub("[^a-zA-Z]", " ", dataset["Review"][0])
rev_W = rev_W.lower()    # converting to Lower case
rev_W = rev_W.split()    # converting "string" to "list"
# reView = [wrd for wrd in rev_W if wrd not in stopwords.words("english")]
# Removing stopwords. And stemming : Finding the root of different versions of a same word
prt_stmr = PorterStemmer()
reView = [prt_stmr.stem(wrd) for wrd in rev_W if wrd not in set(stopwords.words("english"))]
reVw = " ".join(reView)
```

❒ ***Loop through al 1000 reviews:*** We'll create a new list. And then inside our for-loop we are going to append it through the iterations.



dataset - DataFrame

Index	Review	Liked
0	Wow... Loved this place.	1
1	Crust is not good.	0
2	Not tasty and the texture was just nasty.	0
3	Stopped by during the late May bank holiday off Rick Steve recommendation and loved it.	1
4	The selection on the menu was great and so were the prices.	1
5	Now I am getting angry and I want my damn pho.	0
6	Honeslty it didn't taste THAT fresh.)	0
7	The potatoes were like rubber and you could…	0
8	The fries were great too.	1
9	A great touch.	1

coRpus_reViw - List (1000 elements)

Index	Type	Size	Value
0	str	1	wow love place
1	str	1	crust good
2	str	1	tasti textur nasti
3	str	1	stop late may bank holiday rick steve recommend love
4	str	1	select menu great price
5	str	1	get angri want damn pho
6	str	1	honeslti tast fresh
7	str	1	potato like rubber could tell made ahead time kept warmer
8	str	1	fri great
9	str	1	great touch

```python
# Cleaning the text
import re
import nltk
# nltk.download("stopwords")
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
# print(f"{dataset['Review'][0]}")
coRpus_reViw = []

for i in range(0, 1000):
    rev_W = re.sub("[^a-zA-Z]", " ", dataset["Review"][i])
    rev_W = rev_W.lower()   # converting to Lower case
    rev_W = rev_W.split()   # converting "string" to "List"
    # reView = [wrd for wrd in rev_W if wrd not in stopwords.words("english")]
    # Removing stopwords. And stemming : Finding the root of different versions of a same word
    prt_stmr = PorterStemmer()
    reView = [prt_stmr.stem(wrd) for wrd in rev_W if wrd not in set(stopwords.words("english"))]
    reVw = " ".join(reView)
    coRpus_reViw.append(reVw)
```

☝ All ***stopwords*** are ***removed*** and all words are ***stemmed***.

☝ We will simplify those reviews even more. Because we will ***actually keep*** only the ***words*** that appear a ***minimum number*** of ***times***. We'll remove some ***irrelevant words*** by ***filtering*** all the ***words*** that ***appear rarely*** and we'll do that while creating the ***Bag of words*** model.

7.4.2 Clean the Data

What is the ***bag of words model*** and why we need to create it? By understanding why we need to create the ***bag of word model*** you will understand even more why we ***had to clean*** the ***text/the 1000 reviews***.

☞ Basically the ***first big step*** of natural language processing (NLP) not only ***Cleaning The Texts*** but also creating a ***Corpus***.

❒ Now from this corpus we will create our bag of words model.

☞ The bag words model basically really simple. We're going to take all the different words of the 1000 reviews here but ***without*** taking ***twice*** or ***three*** times (***duplicates*** or ***triplicates***) i.e. the unique words of these 1000 reviews. Then we create a ***table*** where ***one column*** for each ***word***. So there will be lots of columns.

☞ Then we will put all these columns in a table where the rows are nothing else than the 1000 reviews.

❒ So basically what we'll get is a ***table*** containing ***1000 rows*** where the ***rows*** correspond to the ***reviews*** and a lot of columns where the ***columns*** correspond to each of the ***different words*** in the ***corpus*** of the ***reviews***.

☞ Each cell of this table will correspond to ***one specific review and one specific word*** of this corpus. And in this ***cell*** we're going to have a ***number*** which represents the ***number of times*** the ***word*** corresponding to the column ***appears*** in the ***review***.

☞ So if a word appear in a review, we place a 1, and for all rest of the words we place 0's in the rows. Hence in each row(or review) of the table we **mostly** have ***0***s and **few** ***1***'s.

❒ ***Sparse Matrix:*** And this table is actually a matrix, and it is called ***sparse matrix***. In numerical analysis and scientific computing, a ***sparse matrix*** or ***sparse array*** is a matrix in which most of the elements are ***zero***.

☞ And the fact that we have a lot of ***0***s is called ***sparsity*** and that's a very common notion in ***machine learning***. We work a lot with ***sparse matrix*** and we're trying to ***reduce sparsity*** as much as possible when we work with ML models.

☞ ***Creating this Sparse Matrix*** is actually the bag of words model itself. The bag of words model is basically to ***simplify*** all the ***reviews***, ***clean*** all the ***reviews*** to ***simplify the words*** and try to ***minimize the number of words***.

☞ ***Tokenization:*** And it's also about **creating** the **sparse matrix** through the process of ***tokenization***. Tokenization is the process of taking all the different words of the review and creating one column for each of these words (which we just discussed).

❒ ***Our goal:*** In the end our goal is to predict if a **review** is **positive** or **negative**. Our ML model needs to be trained on all these reviews, and find the correlations between the ***hints*** (the words) that tell if the review is ***positive*** or ***negative*** and it's ***true*** result whether it is ***positive*** or ***negative***.

☞ Now remember we did this kind of stuff in classification models. So, here we also ***classify*** a review is ***positive*** or ***negative*** according to the given data (to train the model).

☞ Now ***reducing the words*** and ***sparsity*** actually ***reducing*** the ***"independent variable"*** of our classification problem.

☞ The ***dependent variable*** is a **categorical variable** a binary outcome: ***1*** is **a review positive** or ***0*** if **the reviews negative**. So we are doing nothing else than classification.

☞ So, as soon as we managed to create this bag of words, then we simply need to copy or classification teplates and create our ML model.

☞ At the end we will have our ***matrix of features*** or ***matrix of independent variables*** which will be the **different word appearing** in all the **reviews** here that will be the columns of our matrix and we'll have our ***dependent variable vector*** which will be the ***result*** whether the review is ***positive*** **(1)** or ***negative*** **(0)**.

☝ So that's why we need to create this bag of words model and now we understand very well why we had to clean all the text all the reviews. Because since we created one column for each word that is one independent variable for each word.

▶ Hence we ***clean*** the ***reviews*** and ***simplify*** them as much as ***possible*** to ***reduce*** the total number of ***words*** in the ***corpus*** and therefore the total number of ***independent variables***.

❒ ***Creating the bag of words:*** we will create this **Bag Of Word Model** through the process of **tokenization.** To create our *Bag Of Word* we need to use **CountVectorizer** of **sklearn.feature_extraction.text**. It will simply create the ***sparse matrix***. Since it's a ***class*** we will create an ***object*** of this class

```
from sklearn.feature_extraction.text import CountVectorizer
cntVctzr = CountVectorizer()
```

➲ Now we apply ***fit_transform()*** to our ***coRpus_reViw.*** And also we convert the result to an array (matrix). We call it ***X*** since it will be our ***feature matrix***.

```
X = cntVctzr.fit_transform(coRpus_reViw).toarray()
```

☞ At this stage we don't need any parameters for ***CountVectorizer()*** but we should have a look at all the parameters because you'll see that some of them are very useful:

- ***stop_words:*** It is what we did before in the ***cleaning*** part. But we can remove those ***stopwords*** by using this parameter directly.
- ***lowercase:*** Again it is what we did already, but can directly apply here.
- ***token_pattern:*** basically it is same as we did in ***Regular Expressions***. i.e. removing all characters other than ***a-z*** and ***A-Z***.

Basically what I'm showing you here is that what we did before in the text-cleaning manually, we can do directly in this ***CountVectorizer()*** by playing with ***different parameters***.

☝ But using these parameters are not the best way to do it. There are two reasons:

☯ The ***first reason*** is that we get to see how to ***clean*** the ***text*** step by step for ***learning purposes***.

☯ ***Second reason*** is by cleaning the reviews ***manually*** that gives you ***more options***. Sometimes you will need to do further cleaning to clean whatever text you're working with. For example:

➲ If you are doing ***NLP*** for ***web scrapping***, in that case the text are going to be some ***HTML pages*** and in those ***HTML pages*** you'll get some ***HTML code***. So you would need to add ***another option*** to clean these ***HTML texts***. Basically doing this ***manually*** gives us more control and ***more options***.

❒ ***Improving the Bag of Word:*** In our sparse matrix we can see, it has ***1000 lines*** and ***1565 words***. But we can ***reduce*** the ***sparsity*** by using ***another parameter*** called max_features. It will keep the most frequent words in your reviews so you know that will remove the ***non-relevant words*** that appear only ***once*** or ***twice*** of use. Lets set this ***max_features= 1500***

☞ Now our sparse matrix of features contains ***1500 columns*** instead of ***1565***. And not only that reduces the sparsity but also that gives us **most relevant words** to train our algorithm which therefore has more chance to ***make better correlations*** between the ***presence of the words*** in the reviews and the ***outcome of the reviews*** whether they are ***positive*** or ***negative***.

☞ Now about sparsity, we can also REDUCE ***sparsity*** using ***dimensionality reduction*** techniques which we will see later in this course in Chapter 9: Dimensionality Reduction.

☞ Then we create the dependent variable (positive or negative) from our given dataset.

```
# creating the bag of words model
from sklearn.feature_extraction.text import CountVectorizer
# cntVctzr = CountVectorizer()
cntVctzr = CountVectorizer(max_features= 1500)
X = cntVctzr.fit_transform(coRpus_reViw).toarray()
y = dataset.iloc[:, 1].values
```

❒ ***Building the model:*** We now build the model using **any classification algorithm**.

☞ Now we have ***X*** and ***y***, exactly what we had in ***Chapter 3: Classification*** that is we have a ***matrix*** **of independent variable X** here that contains 1500 independent variables (which are words actually). Each line here corresponds to one specific review and ***1***: word is **in the review**, ***0***: word is **not in the review**.

☞ So basically that gives us a classification model, we will train a ML model that will try to understand the ***correlations*** between the ***presence of the words*** in the reviews and the outcome (is ***0*** if it's a ***negative*** review and ***1*** if it's a ***positive*** review).

☞ And we will use ***Dimensionality Reduction*** to ***reduce sparsity*** of this ***Sparse Matrix***. But we will do this in Chapter 9.

❒ ***Choosing the Classification Models:*** We just need to do some copy-paste of our **Machine Learning Classification Models** that we built in ***chapter 3***. In ***chapter 3***, we have

i. Logistic Regression,
ii. KNN
iii. SVM,
iv. Kernel SVM,
v. Naïve Bayes,
vi. Decision tree classification and
vii. Random forest classification.

So which one would be the best for natural language processing? Well we have two options her:

[1] Since we already have our ***matrix of features*** and our ***dependent variable vector*** and since ***everything*** is ***well-prepared*** for our ***ML classification models***. So we can apply all of them and compare the ***Confusion Matrix***. By looking at the ***accuracy*** the number of ***false positive*** and ***false negatives*** and look at all their ***performance criteria*** to decide what would be the best model.

[2] Secondly, from experience, most Data-Scientists commonly use ***Naïve Bayes***, ***Decision tree*** classification and ***Random forest*** classification for ***Natural Language Processing***.

☞ For this section we choose the ***Naïve Bayes***.

```
# NLP: Natural Language Precessing

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# -------------- Data preprocessing --------------
# importing dataset
        # since we are using "tsv" ins tead of "csv" we need to specify some parameters.
        # Because "Pandas" expecting some "csv" files
dataset = pd.read_csv("Restaurant_Reviews.tsv", delimiter = "\t", quoting = 3)


# Cleaning the text
import re
import nltk
# nltk.download("stopwords")
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
# print(f"{dataset['Review'][0]}")
coRpus_reViw = []

for i in range(0, 1000):
    rev_W = re.sub("[^a-zA-Z]", " ", dataset["Review"][i])
    rev_W = rev_W.lower()   # converting to Lower case
    rev_W = rev_W.split()   # converting "string" to "list"
    # reView = [wrd for wrd in rev_W if wrd not in stopwords.words("english")]
    # Removing stopwords. And stemming : Finding the root of different versions of a same word
    prt_stmr = PorterStemmer()
    reView = [prt_stmr.stem(wrd) for wrd in rev_W if wrd not in set(stopwords.words("english"))]
    reVw = " ".join(reView)
    coRpus_reViw.append(reVw)
```

```python
# creating the bag of words model
from sklearn.feature_extraction.text import CountVectorizer
# cntVctzr = CountVectorizer()
cntVctzr = CountVectorizer(max_features= 1500)
X = cntVctzr.fit_transform(coRpus_reViw).toarray()
y = dataset.iloc[:, 1].values


# using ---------------- Naïve Bayes --------------
# Data Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.20, random_state = 0)

# # Feature-Scaling
# from sklearn.preprocessing import StandardScaler

# st_x= StandardScaler()
# X_train= st_x.fit_transform(X_train)
# X_test= st_x.transform(X_test)


# Fit train set to Naïve Bayes classifier: No parmeter is needed
from sklearn.naive_bayes import GaussianNB
clsFier = GaussianNB()
clsFier.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = clsFier.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value

accuracy = (cm[0][0] + cm[1][1])/X_test.shape[0]

# How can I find the length of a row (or column) of this matrix? Equivalently, how can I know the number
of rows or columns?

# shape is a property of both numpy ndarray's and matrices.
    # A.shape
# will return a tuple (m, n), where m is the number of rows, and n is the number of columns.



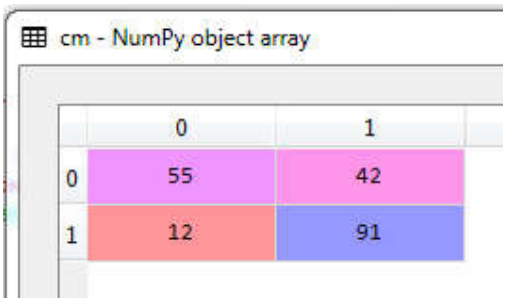
# import nltk
# from nltk.corpus import stopwords
# print(stopwords.words('english'))
# print(set(stopwords.words('english')))

# python prctc_nlp.py
```

- We take the ***test size*** 20%.
- We also ***don't need*** any ***feature scaling*** because we are only dealing with ***0***s and ***1***'s.

- From the confusion matrix we that out of 200 test points, ***146***(true negative 55 + true positive 91) are ***correct*** predictions and ***54*** (false positive 42 + false negative 12) are ***incorrect*** predictions and the accuracy is 73%.
  - From ***97*** **negative** review we predicted ***55*** **correct**
  - From ***103*** **positive** review we predicted ***91*** **correct**

```
posve = [k for k in y_test if k==1]
total_posve = len(posve)
```



Chapter 8 : Part 1

Deep Learning

ANN: Artificial Neural Network

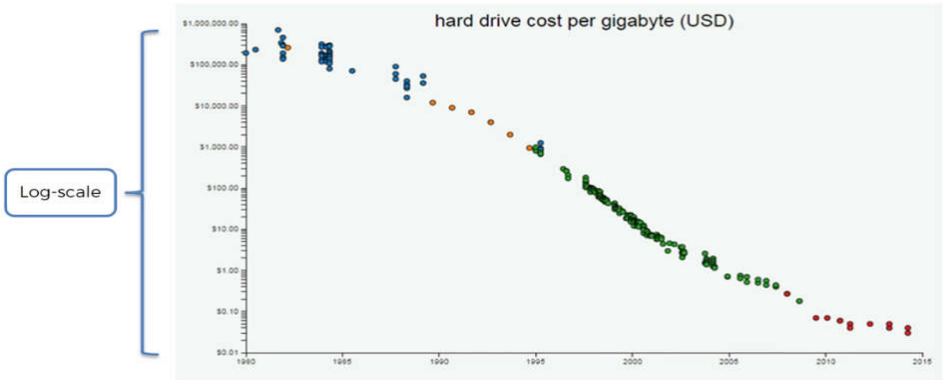
Introduction to Neural Network

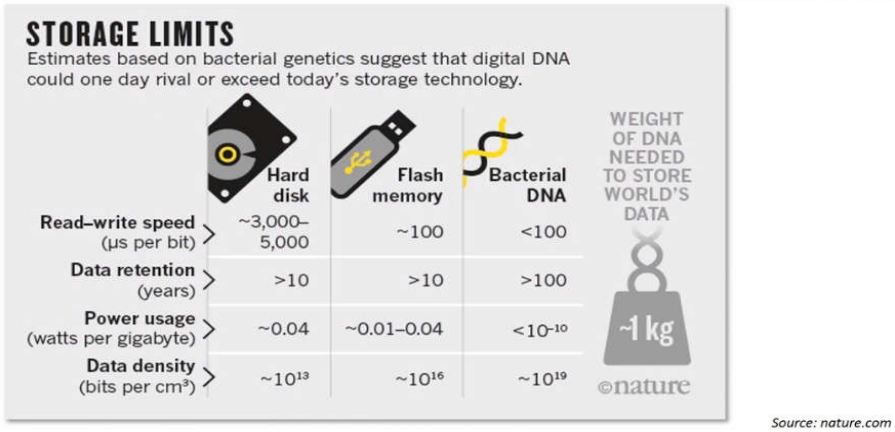
8.1.1 History of Deep Learning

Deep learning was invented in the 60s and 70s, in the 80s so people are talking about them a lot. But that trend became slow and died during the following decades. The reason was, the idea was not so clear and technology wasn't ready.

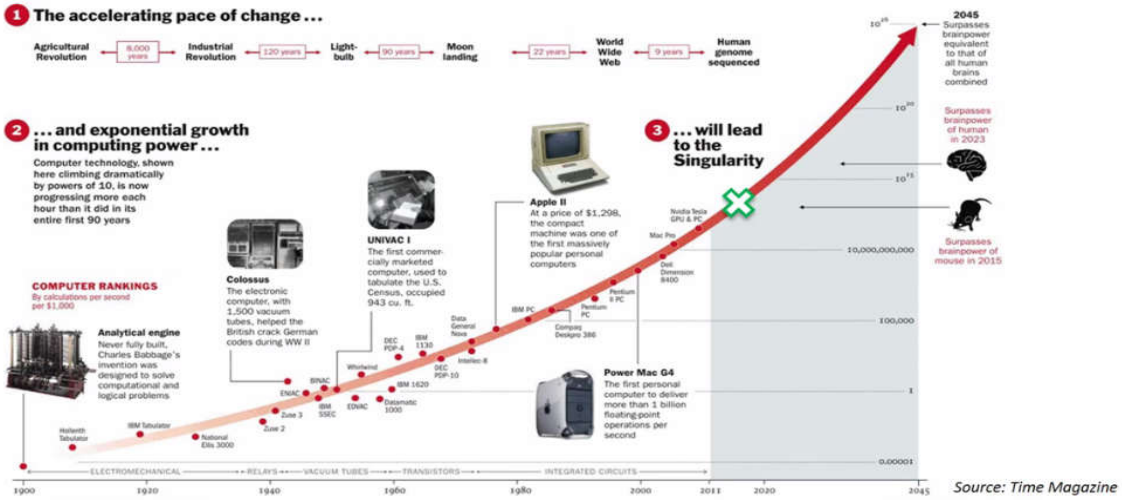
From 2000 to 2020 the ***storage capacity*** become ***large enough*** and the ***processing power*** of the computers ***increased***. That's why this Deep Learning concept became so popular in the recent years.

And in deep learning to work properly, you need two things: a ***lot of data*** and ***processing power*** (strong computers to process that data).



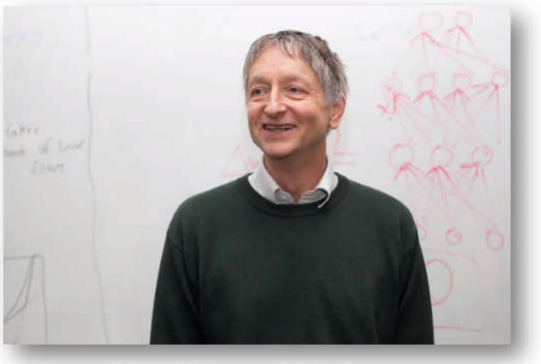


DNA storage Comparison



Processing capacity is increasing

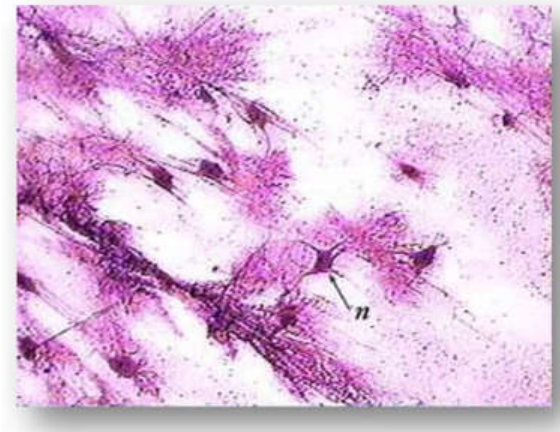
❒ ***What is deep learning:*** This gentleman over here is ***Jeffrey Hinton***, known as the **godfather of Deep Learning**. And he did research on deep learning in the 80s and he's done lots and lots of work lots of research papers he's published in deep learning. Right now he works at **Google**.

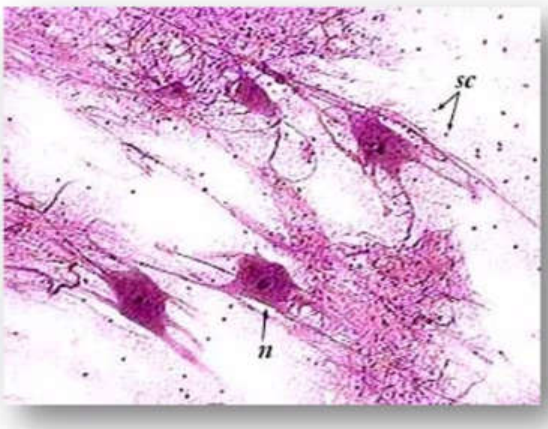


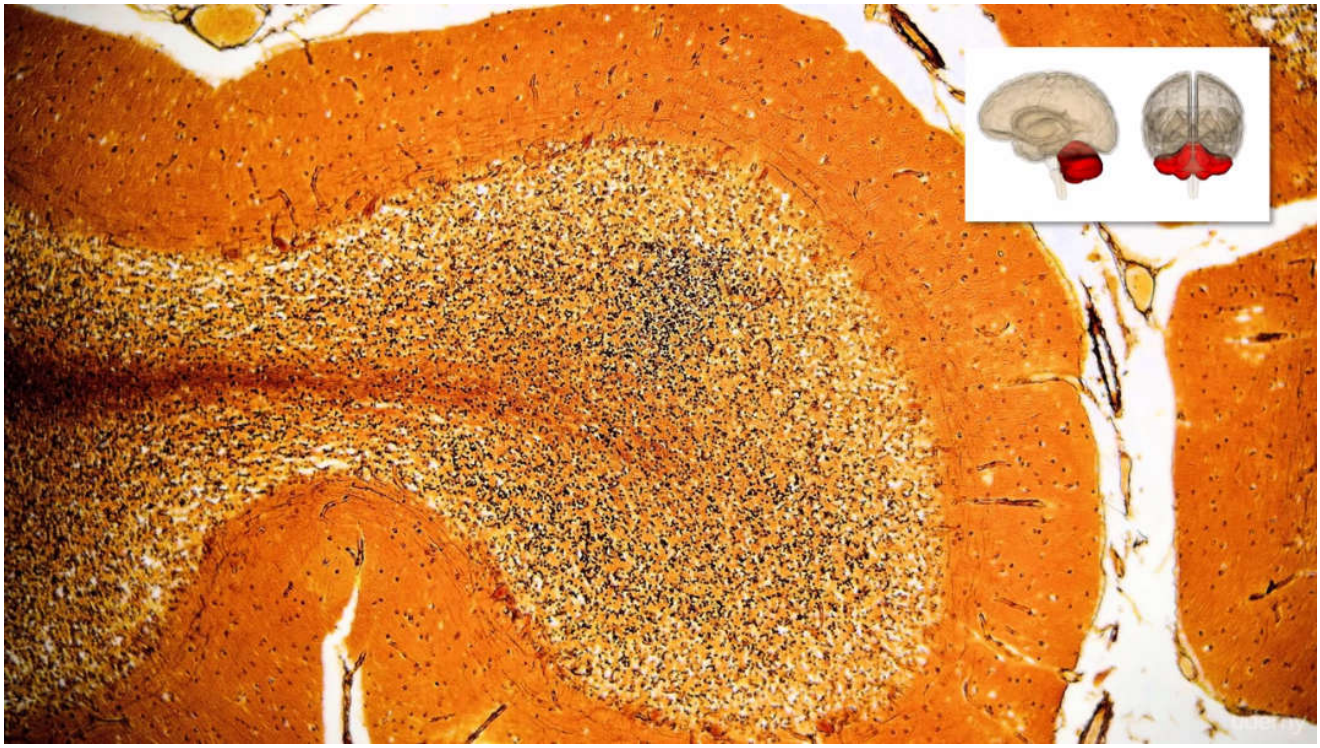
Geoffrey Hinton

❒ ***The idea behind deep learning is:*** to look at the human brain. Here we've got some neurons, they have a body, branches, tails and a nucleus in the middle and that's that's basically what a neuron looks like in the human brain.

- ☞ There's approximately ***100 billion neurons*** all together so these are individual neurons these are actually motor neurons because they're bigger they're easier to see but nevertheless there's a **hundred billion** neurons in the human brain.
- ☞ And each of neurons connected to as many as about a ***thousand*** of its ***neighbors***.



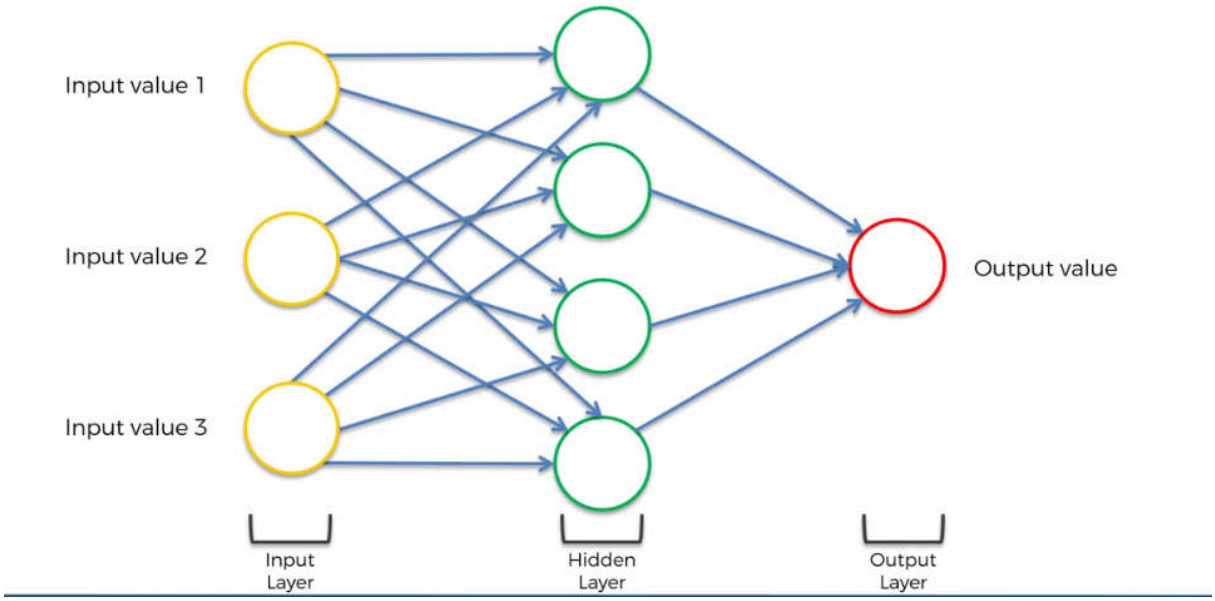




Actual section of ***Cerebellum***

Above image shows how many ***neurons*** there are, like billions and billions and billions of neurons all connecting each other forming a ***network***. And that's what we're going to be trying to ***recreate*** in our computer.

❒ ***How do we recreate this in a computer:*** We'll create an artificial structure called an ***artificial neural net*** where we have ***nodes*** or ***neurons***. We're going to have some



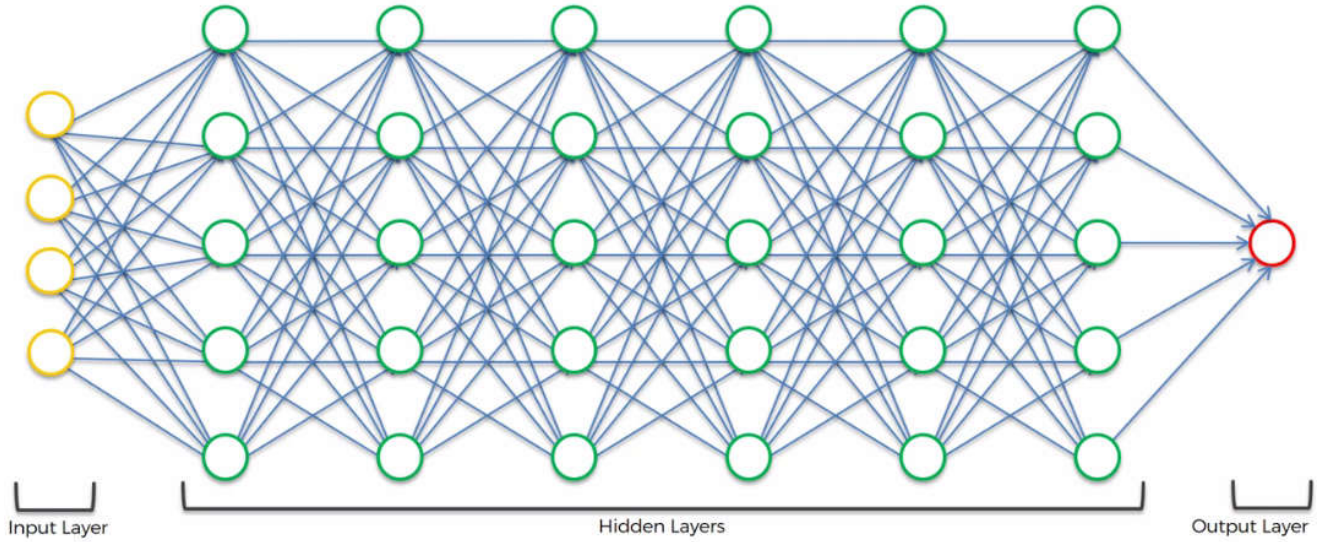
Shallow-Learning

☞ ***Input layer:*** Neurons for input values: These are values that you know about a certain situation. For instance: you're ***modeling something***, you want to ***predict something*** you always could have some ***input*** something to ***start***., then that's called the ***input layer***.

☞ ***Output layer:*** Then you have the output. That's of value that you ***want to predict***. EG: For a transaction in a bank, is this a fraud-transaction it's a real-transaction and so on. So that's going to be ***output layer***.

☞ ***Hidden layer(s):*** Between ***Input layer*** and ***Output layer*** we're going to have a ***hidden layer***. The **input layers** neurons connected to a **hidden layer** neurons that neurons are connect to **output**.

❒ ***Shallow-Learning and Deep-Learning:*** For a few Hidden Layers it is called Shallow-Learning. When the Hidden Layer increases then it is called Deep Learning.



Deep Learning

☞ And that's how the ***input values*** are ***processed*** through all these ***hidden layers*** just like in the human brain. Then we have an output. So that's what Deep-learning is all about on a very abstract level.

What we will learn in this section

i. ***The Neuron:*** There'll be a little bit of **Neuroscience** and we'll find out a bit about **how** the **human brain works** and why we are **trying to replicate** that. And we'll also see what the *main building block* of a ***Neural Network*** of the ***Neuron*** looks like.

ii. ***The Activation Function:*** We'll talk about the *activation function* and we'll look at a couple of *examples* of *activation functions* that you could use in your *neural networks* and we'll find out which ones of them is the most *commonly* used in *neural networks* and in which *layers* you'd rather use.

iii. ***How do Neural Networks work? (example):*** We're not going jump into the *learning part* directly, instead we're actually going to go into the *working of the Neural Networks first* because that way by *seeing a Neural Network in action* that will allow us to understand what we're ***aiming*** towards what ***our goal*** is.
   - So here we'll look at an example of a neural network: we're going to look at a very simplified hypothetical example: to *predict housing prices*.

iv. ***How do Neural Networks learn?:*** We will move on to understanding **how Neural Networks learn**.

v. ***Gradient Descent:*** This is also part of neural networks learning and we'll find out how great the advantage of **Gradient Descent** are.

vi. ***Stochastic Gradient Descent:*** It's a it's a continuation of the **Gradient Descent** tutorial but it's an even better and even stronger method and we'll find out exactly how it works.

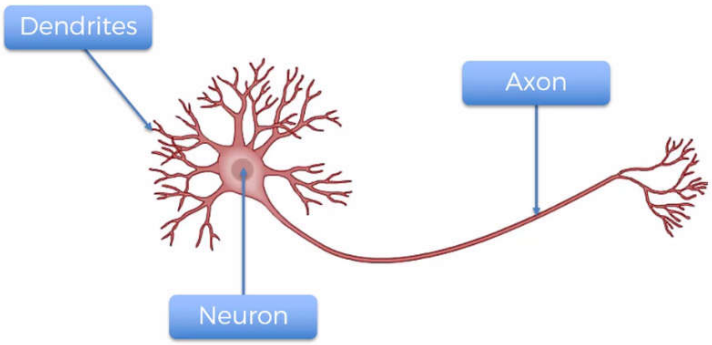
vii. ***Backpropagation:*** And finally we'll wrap things up by mentioning the important things about back propagation and summarizing everything in a step by step set of instructions for running your artificial neural networks.

8.1.2 The Neuron

Neuron is the basic building block of **Artificial Neural Networks**. Now before recreate it, lets talk about Biological Neuron.

❒ Neurons by themselves are pretty much useless. It's like an **ant**. For example an ant can't do anything, but **five ants** together can pick something up. And if you have a **million ants** they can build a whole **colony** they can build an **anthill**. They can act like an **Organism**.

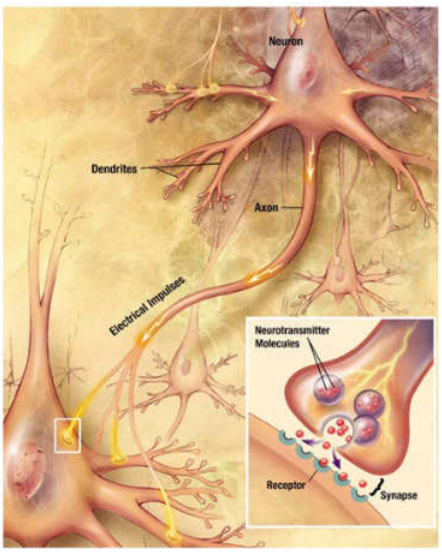
☞ Same thing with the neurons. By itself it's not that strong but when you have lots of neurons together they work together to do magic.



❒ ***How do they work together:*** That's what the ***Dendrites*** and ***Axons***.

☞ ***Dendrites*** are kind of like the ***receivers*** of the signal for the neuron and

☞ ***Axon*** is the ***transmitter*** of the signal for the neuron. And here's an image of how it all works conceptually.



☞ We can see ***Dendrites*** are connected to ***Axons*** of other neurons that are like even further away above it (***axon*** doesn't actually touch the ***dendrite***, it has been proven that there is no physical connection there).

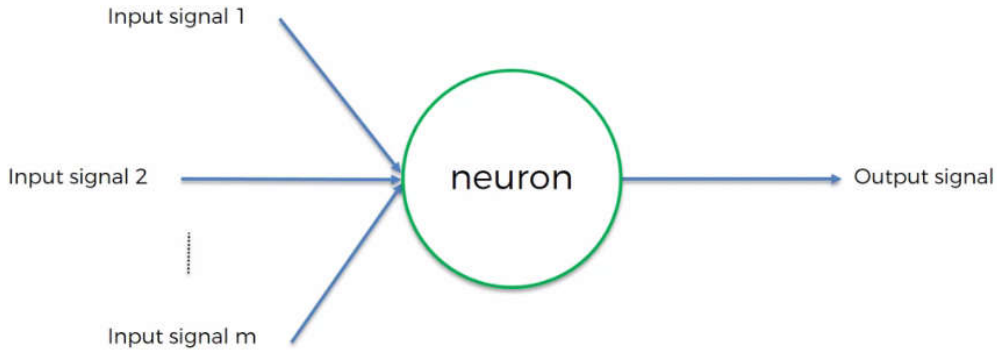
☞ And then the ***signal*** travels ***down*** its ***axon*** and connects or passes on to the ***dendrites*** of the next neuron and that's how they're connected.

☞ Connection between the neurons are called the ***Synapse (neuronal junction)*** you can see over there in that little image that's figure bracket is a ***Synapse***.

❒ And we will use those terminology in our **Artificial Neural Networks (ANN)**. So instead of calling our *artificial neurons linking lines* ***axons*** or ***dendrites*** (whose connection it is, coming or going signals) we're going to call them just ***Synapses***.

☞ Because the important thing is ***where the signal is passed*** *doesn't matter who that element belongs to*. They're just a representation of the **signal pass**. So basically that's how a neuron works.

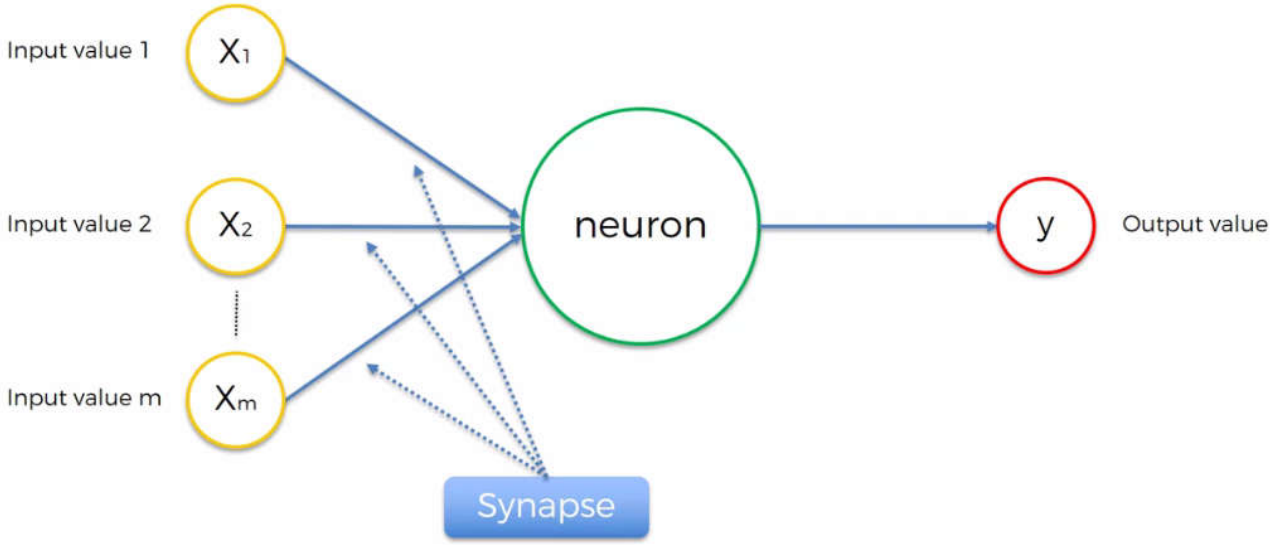
❒ ***How are we going to represent neuron in a machine:*** So here's our ***neuron*** also sometimes called the ***node***. The ***neuron*** gets some ***input signals*** and it has an ***output signal*** (Dendrites and Axons). We are call them ***Synapses***.



☞ These ***input signals***, we're going to present them of other ***neurons*** as well. So in this specific case you can see that this neuron is a ***Green neuron*** and is getting signals from ***Yellow neurons***.

❒ ***Color indication:*** In this section we're stick to a certain **color coding regime** where ***yellow*** means an ***input layer*** (all of the neurons that are on the **outer layer** on the first **front** of where are the signals **coming in**).

🜲 And by ***signal***, we mean just basically ***input values***. Like in a simple linear regression we have input values and then we have a predicted value. Same thing is in here.



☞ So we have ***input values*** as the ***Yellow*** ones and then on the right ***output-value*** will be ***Red***. And ***hidden layers*** will be ***Green***.

☞ In terms of the ***input layer*** the way to think about it as of the ***human brain*** the ***input layer*** is your ***senses*** i.e.whatever you can see, hear, feel, touch or smell.

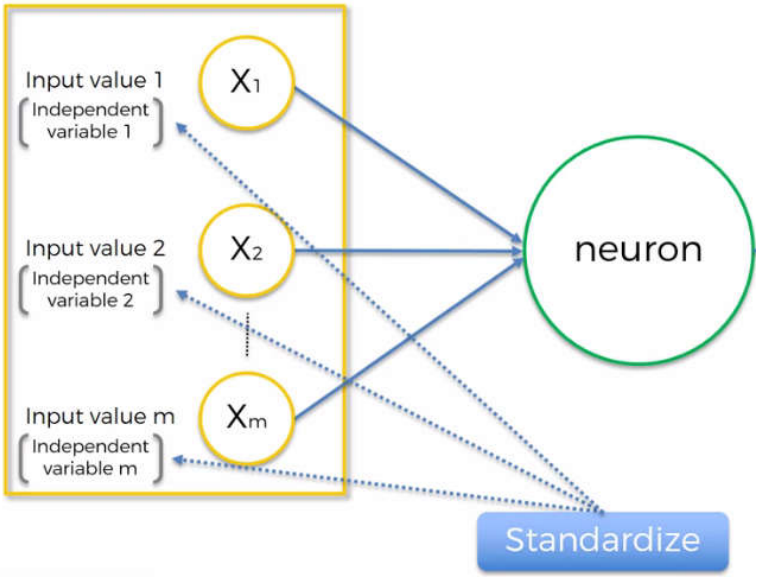
- More simply your brain is sitting in your head (dark box) it cannot feel anything and inputs are coming as electric signals from the senses.
- So for ***Humans*** it is our ***Five-senses*** as Input-layer and for our ANN it is just ***input-data***.

🜲 One other thing, here in this specific example we're looking at a ***neuron*** which is getting its signals from the ***input layer (yellow) neurons***. Sometimes you'll have ***neurons*** which get their ***signal*** from other ***hidden layer neurons*** (i.e. from other ***green neurons***) and the concept is going to be exactly the ***same***. For simplicity we're portraying his example.

8.1.3 How Neuron works in ANN : Terminologies

❒ ***Input-layers:*** In the input layer, ***inputs*** are in fact ***independent variables***. These ***independent variables*** are all for ***one single observation***. So think of it as just ***one row*** in your database (think your observation points as (a, b, c, d, . . .) a multi-dimensional vectors).

☞ Eg: independent variables maybe age, deposits in the bank, drive/walk, salary etc. But that's all descriptors of one specific person (***one single observation***.) that can be either in your training model or prediction.



❒ ***Standardize/Normalize the variables:*** You need to ***standardize*** these variables so you have a ***mean of zero*** and a ***variance one***.

☞ Or you can also sometimes ***normalize*** them instead of ***standardize*** them. Meaning that instead of making a ***mean of zero*** and a ***variance one,*** you just ***subtract*** the ***minimum value*** from ***each element*** of a ***column*** and then ***divide*** by the **(maximum – minimum)** by the range of your values and therefore you get values ***between 0 and 1***.

☞ And it depends on this scenario you might want to do ***standardize*** or ***normalize***.

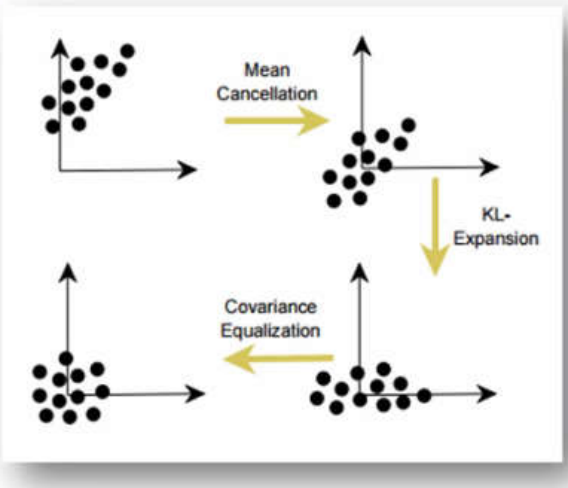
Basically you want all of these variables to be quite similar because these values are going to go into a neural network and is going to be easier for the neural network to process them if they're all about the same.

If you want to read more about standardization/normalization and other things read following paper: ***Yann LeCun*** is a leading ***Deep Learning*** expert works at ***Meta (facebook)*** as Chief Scientist. And is close friend to ***Geoffrey Hinton***. In this paper you'll learn more about ***centralization*** and ***normalization***.

Additional Reading:

Efficient BackProp

By Yann LeCun et al. (1998)



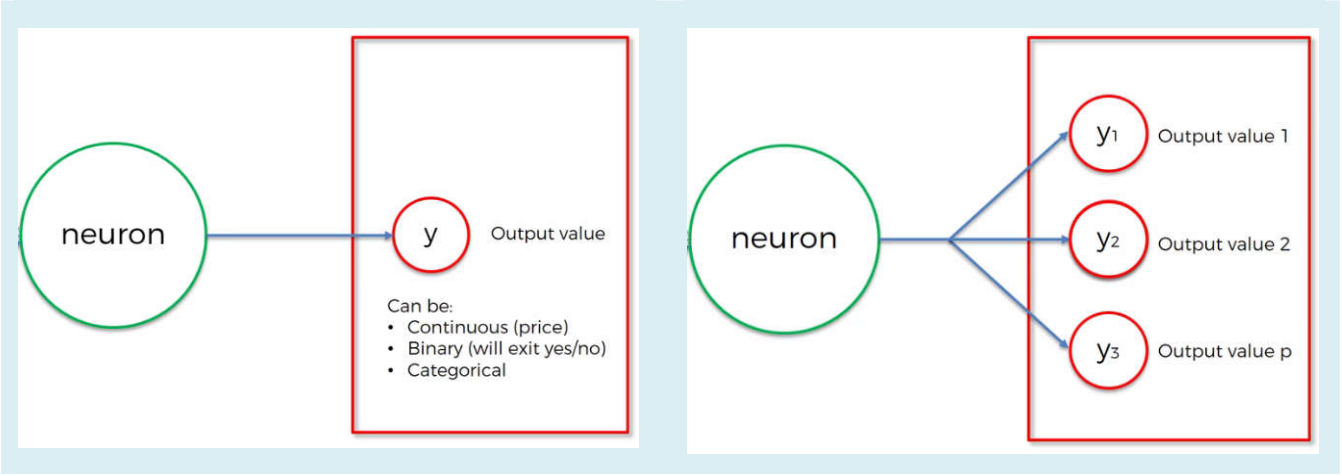
Link:

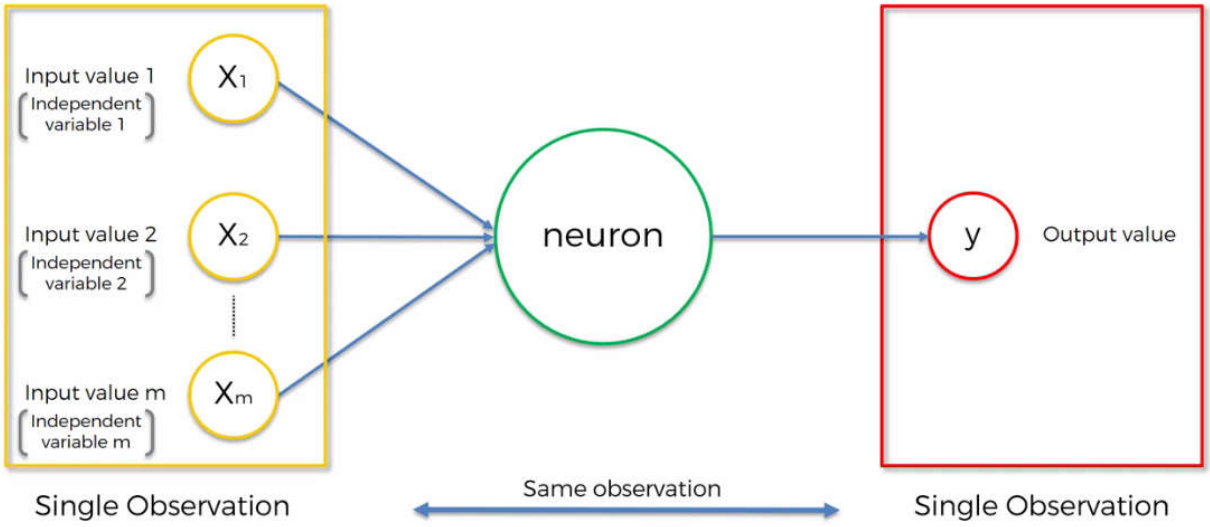
http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

❒ ***Output values:*** These values can be different types. Output value can be:

i. ***Continuous***. For instance price it can be
ii. ***Binary*** (yes/no). For instance a person will exit or will stay in a shop.
iii. ***Categorical*** verbal.

In the case of categorical variable, your **output** value won't be just ***one***, it'll be ***several output values*** because these will be a ***dummy variables*** which will be representing your ***categories***.





❒ ***All those things are happening to singe observation:*** Now consider simple case of one output value. Whatever ***inputs*** you putting in, that's for ***one row*** and then the ***output*** you get that is for that ***same exact row***. Or if you're **training** your **neural network** then you're putting the inputs in for that one row you're getting the output in for that one specific row.

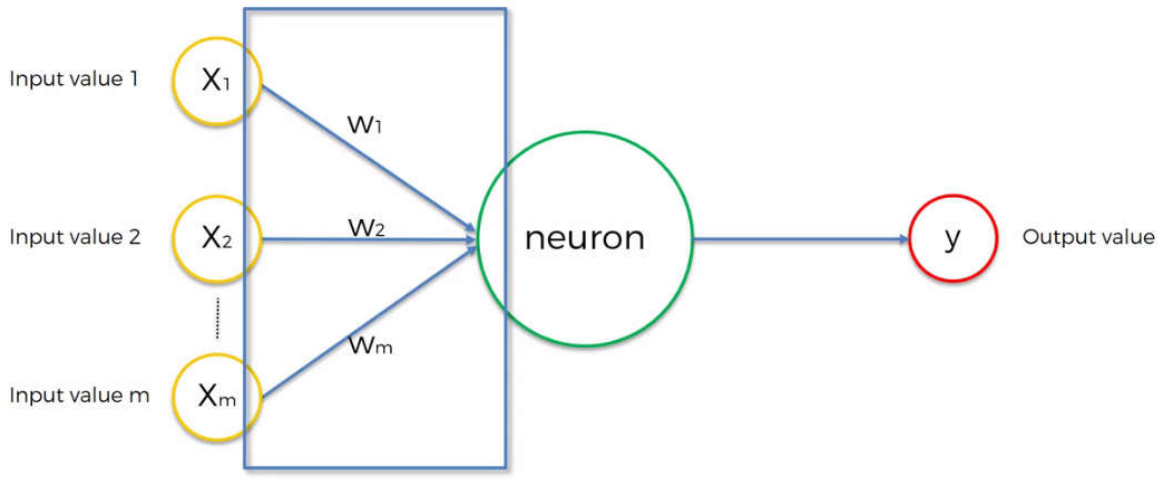
☞ For simplicity, you can think it like a ***simple (multivariate) regression***. Same thing here it's nothing too complex. We're just putting in values we are getting output. But just remember that ***every time*** it's ***one row*** you're dealing with. *Don't get confused and start thinking that* ***these are different rows that you're putting into your artificial neural network***.

❒ ***Synapses:*** The Synapses, they all actually get assigned with weights.
In short weights are **crucial** to **ANN functioning**. Because **weights** are how **neural networks** ***learn***. By adjusting the ***weights*** the ***neural network decides*** in every ***single case*** what ***signal*** is ***important*** and what ***signal*** is ***not important*** to certain ***neuron***.

☞ What ***signal*** gets passed along or not. Or what strength to what extent signals get passed along. So weights are crucial. They are the things that **get adjusted through** the **process of learning**.

☞ When you're ***training*** an ***ANN*** you're basically ***adjusting*** all of the ***weights*** in all of the ***Synapses*** across this whole **neural network**.

☞ And that's where ***Gradient Descent*** and ***Back Propagation*** come into play.



Synapses assigned with weights

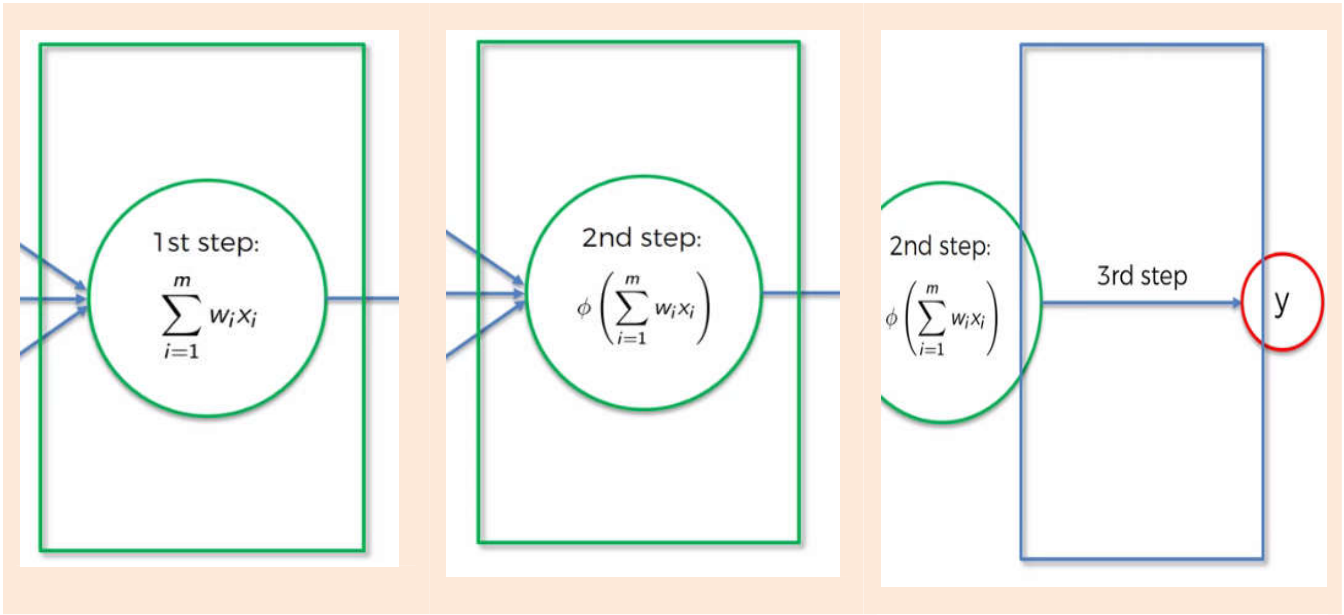
❒ ***Neurons:*** Signals go into the neuron and a few things happen.

A. First thing and the ***first step*** is corresponding weights of variables will be added together i.e. the ***weighted sum*** of all of the ***input values***.

B. Secondly an ***activation function*** is applied to this ***weighted sum***.

- Basically ***activation function*** is a function that is ***assigned*** to a ***neuron*** or *to this* **whole layer** *and it is applied to this* **weighted sum**.
- The ***`activation function`*** decides if it ***needs to pass on a signal*** to the **next neuron (or layer)**. i.e. depending on the ***`activation function`*** the **Neuron** will **either pass on a signal** it or it **won't pass the signal** on.

C. And that's exactly what happened here in ***step three***. The ***`neuron`*** passes on that ***`signal`*** to the ***`next neuron`*** down the line.



And that's how you got input values, weights. And what happens in neuron where you've got weighted sum and activation function applied on weighted sum that is passed on line and that is just ***repeated throughout*** the ***whole neural network on and on and on*** and depending on how ***big***, how many ***neurons*** you have how many ***Synapses*** you have your ***neural network***.

NOTES:

Normalization is a *data preparation technique* that is frequently used in machine learning. The process of ***transforming the columns*** in a dataset to the ***same scale*** is referred to as ***normalization***. Every dataset ***does not need*** to be ***normalized*** for machine learning. It is only required when the ranges of characteristics are different.

[1] ***Min-Max Scaling (Normalization):*** Subtract the ***minimum*** value from each column's ***highest value*** and ***divide*** by the ***range***. Each new column has a ***minimum*** value of ***0*** and a ***maximum*** value of ***1***.

[2] ***Standardization Scaling:*** The term "***standardization***" refers to the process of ***centering*** a variable at ***zero*** and ***standardizing*** the ***variance*** at ***one***. Subtracting the mean of each observation and then dividing by the standard deviation is the procedure:

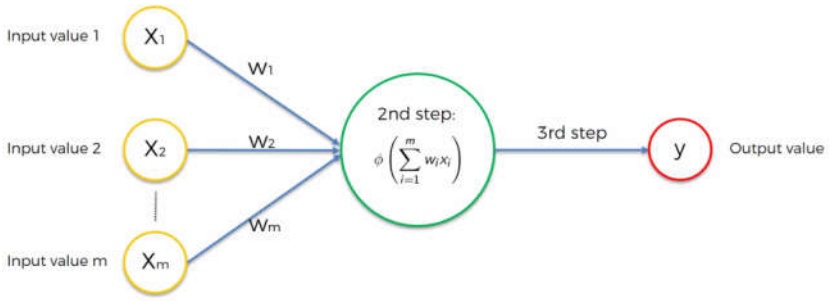
☞ ***Normalization and standardization:*** Normalization and standardization are not the same things. ***Standardization***, interestingly, refers to ***setting the mean to zero*** and the ***standard deviation to one***. ***Normalization*** in machine learning is the process of ***translating*** data into the ***range [0, 1]*** (or any other range) or simply transforming data onto the unit sphere.

EPOCH: An epoch is a term used in machine learning and indicates the ***number of passes*** of the entire ***training dataset*** the machine learning algorithm has completed. Datasets are usually grouped into ***batches*** (especially when the amount of data is very large). Some people use the ***term iteration*** loosely and refer to putting ***one batch*** through the model as ***an iteration***.

☞ If the ***batch size*** is the whole training dataset then the number of epochs is the number of iterations. For practical reasons, this is usually not the case. Many models are created with ***more*** than one ***epoch***. The general relation where ***dataset size*** is ***d***, number of ***epochs*** is ***e***, number of ***iterations*** is ***i***, and ***batch size*** is ***b*** would be ***d*e = i*b***.

8.1.4 The ACTIVATION FUNCTION

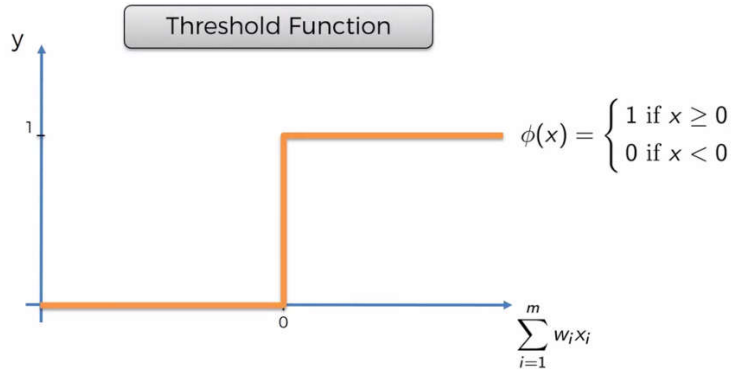
We talked about the ***structure*** of one ***neuron***, it has some **inputs values** coming in. **Synapses** got some **weights** then it ***adds*** up the ***weights*** and then apply the ***activation function***. In step 3, it passes on the signal to the next ***neuron*** (the decision of pass/no-pass comes from the ***activation function***.).



❒ ***Types of activation function:*** We're going to look at **four** different **types** of **activation functions**. Of course there are more different types of activation function but these are the popular ones.

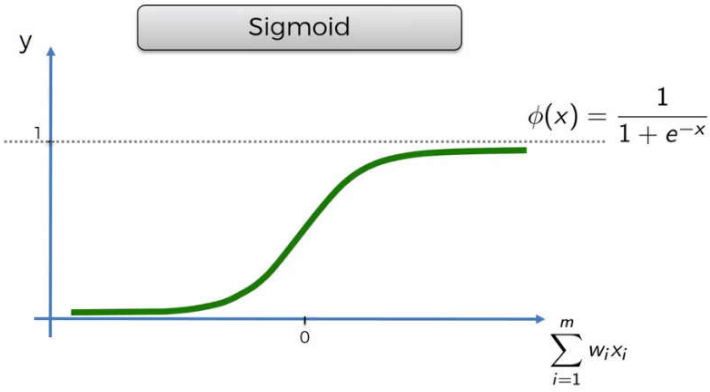
[1] ***Threshold function:*** On the ***X axis*** you have the ***weighted sum*** of ***inputs***. On the y axis, you have 0 to 1 scale. It's basically kind of, ***yes-no*** type of function.

- If the value is ***less than zero*** then the threshold function passes on ***0***.
- If the value is ***more than zero or equal to zero*** then threshold function passes on a ***1***.



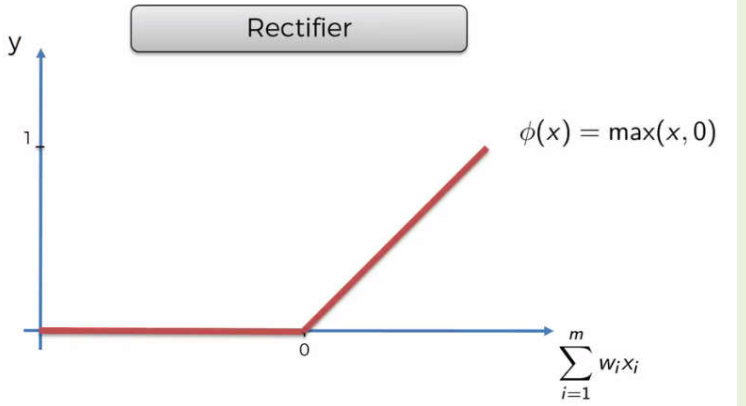
[2] ***Sigmoid Function:*** It's a function which is used in the *logistic regression*.

- The ***benefit*** of this function is that it is ***smooth*** (unlike the ***threshold function***). It's just nice and smooth **gradual progression**. So anything below ***0*** it **doesn't suddenly** goes to ***0***, actually it tends to ***0*** **gradually**. And it approximates towards ***1*** for ***+ve*** values.
- ***Sigmoid function*** is very ***useful*** in the ***final layer*** (the ***output layer***) of ANN. Especially when you're trying to ***predict probabilities***.

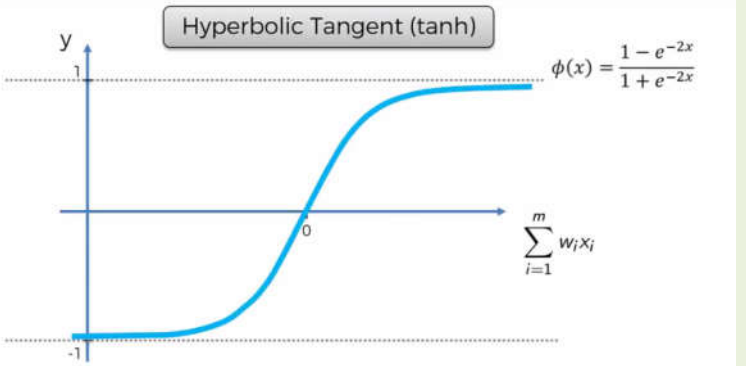


[3] ***Rectifier Function:*** **Rectifier Function** is one of the most used functions in **artificial neural networks - ANN**. even though it has a kink is one of the

- It goes all the way to zero (for ***–ve*** values) and then from there it's ***gradually progresses*** as the ***input value increases*** (for +ve values). We use this function in this section.



[4] ***Hyperbolic Tangent Function:**** It's very similar to the ***sigmoid function*** but here the ***hyperbolic tangent*** function goes ***below 0***. So the values go from ***0*** to ***1*** or approximately to ***1*** and go from ***0*** to (–ve) ***–1*** on the other side.



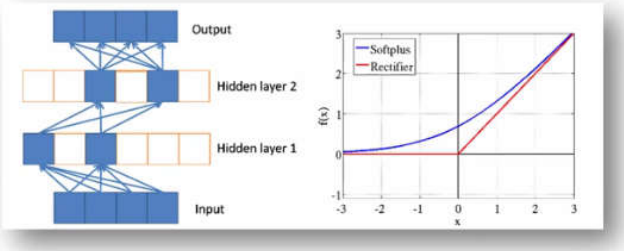
❒ ***Read more in the following paper:*** There you will find out exactly why the ***rectifier function*** is such a ***valuable function***, why it's so ***popularly*** used.

☞ For now we ***don't really need to know*** all of those things. We're just going to start ***applying them*** which you start ***using*** them ***more and more*** . And so when you feel ***comfortable*** with the ***practical side*** of things then you can go and refer to this paper and then you will be able to ***soak in that knowledge much quicker*** and it will make ***much more sense***.

Additional Reading:

Deep sparse rectifier neural networks

By Xavier Glorot et al. (2011)



Link:

http://jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf

❒ ***How to apply different activation function:*** Which type of **activation function** is used in which **layer**.

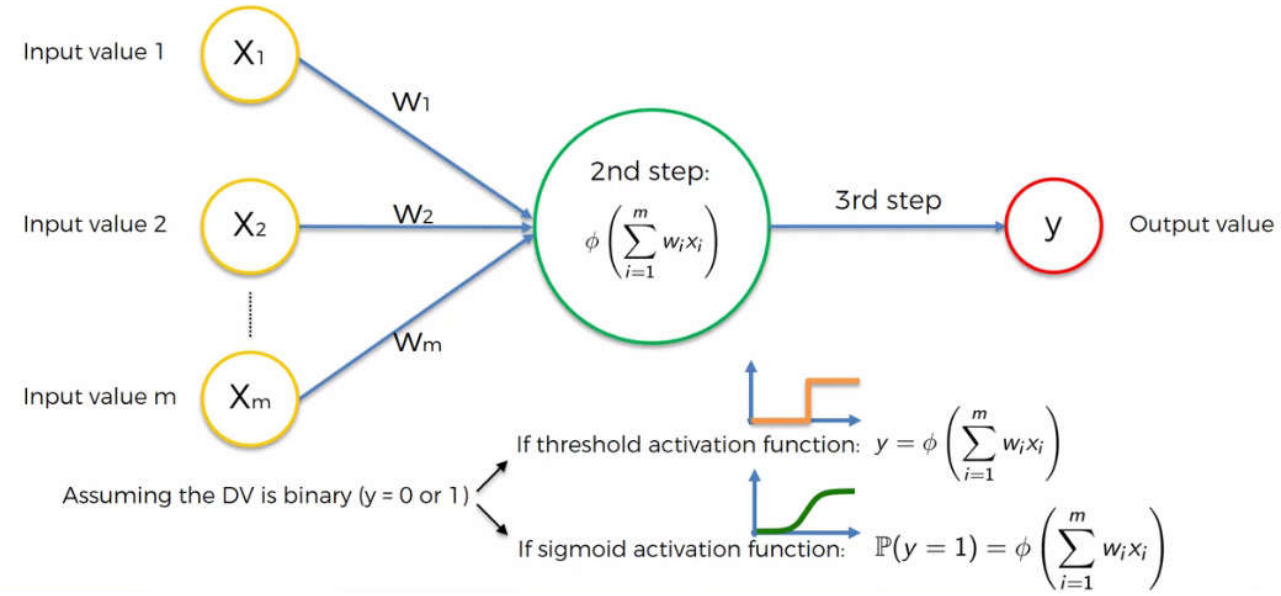
Example 5.1.1: We've got an example here of a ***neural network*** of just ***one neuron*** and an output layer. The question is ***assuming*** that your ***Dependent Variable (Output)*** is ***binary*** (either ***0*** or ***1***) which ***activation function*** would you use.

☯ ***There are two options:*** And those are just two examples. If you have a ***binary output (dependent) variable***.

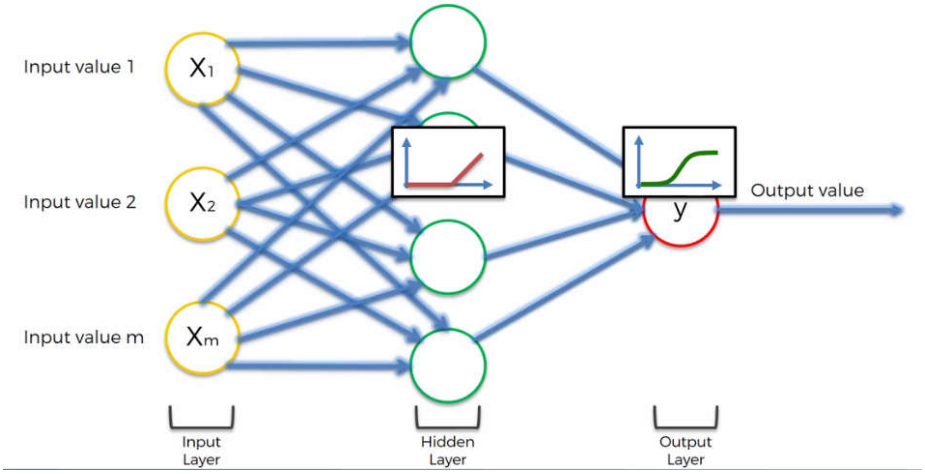
i. ***Threshold Activation Function:*** Because we know that it's between ***0*** and ***1***. It fits perfectly to this requirement and therefore you could you say ***Y*** equals the ***threshold function*** of your ***weighted-sum*** and that's it.

ii. ***Sigmoid Activation Function:*** It is actually between 0 and 1 just what we need. But at the same time you could use it as is the ***probability*** of ***Y*** being ***yes*** or ***no***.

- So we want ***Y*** to be ***0*** or ***1*** but instead we'll say that the **Sigmoid Activation Function** tells us of the probability of ***Y*** being ***equal*** to ***1***.
- That's very similar to the ***logistic regression approach***.



Example 5.1.2: Now let's have a look at another practical application. What about if we had in your all natural like follows?



- In the ***first layer*** we have some ***inputs***. They are sent to our *first hidden layer* and then *an activation function* is applied. And usually we will apply a ***Rectifier Activation Function*** here.

- And then from there the ***signals*** would be **passed** on to the **output layer** where the ***Sigmoid Activation Function*** would be applied and that would be our ***`final output`*** (predict a probability).

And this ***combination of two- activation function*** is going to be quite common where in the **hidden layers** we apply the ***`rectifier function`*** and then for **output layer** we apply the ***`sigmoid function`***.

8.2.1 How an NN works

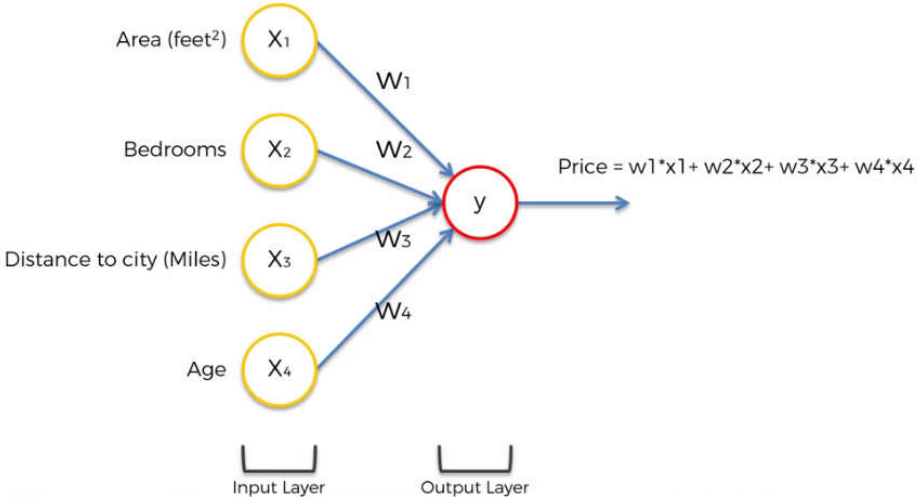
In an NN where ***multiple neuron*** present in a ***hidden layer***, different neuron takes different decisions. Each neuron picks ***different input*** variables according to the ***conditions*** given to the ***neurons***.

Walk through an Example: We're going to be looking at a **property evaluation problem**. We're going to look at a ***neural network*** that takes in some ***parameters*** of our property and ***evaluates*** it.

- We are **not going to train the network** (a very important part in neural networks is training them up). We're going to pretend is ***already trained up*** and that will allow us to focus on the ***application side***.
- Let's say we have some input parameters: ***area*** in square feet, ***number of bedrooms***, ***distance to the city*** in Miles, ***age of the property***. All of those four are going to comprise our ***input layer***.
- There could be ***more parameters***, now for ***simplicity sake*** we're going to look at just this ***four*** for now.

Most of the ***ML algorithms*** (regression/classification) that exist can be ***represented*** in this form and this is basically a ***diagrammatic representation*** of how you deal with.

- This shows us how powerful NNs are. Even ***without the hidden layers*** we are ready. We have a representation that works for most other M L algorithms.



The basic form: It's very basic form of a ***neural network***. It only has an ***input layer*** and an ***output layer*** and no ***hidden layers***. Our ***output layer*** is the ***price*** that we're ***predicting***.

- In this form these inputs variables would be ***weighted up*** by the ***Synapses***, and then the output would be calculated. For instance the ***price*** could be calculated as the ***weighted sum*** of all of the inputs.
- Here we could use any of the ***activation functions***. ***sigmoid*** or ***threshold***.

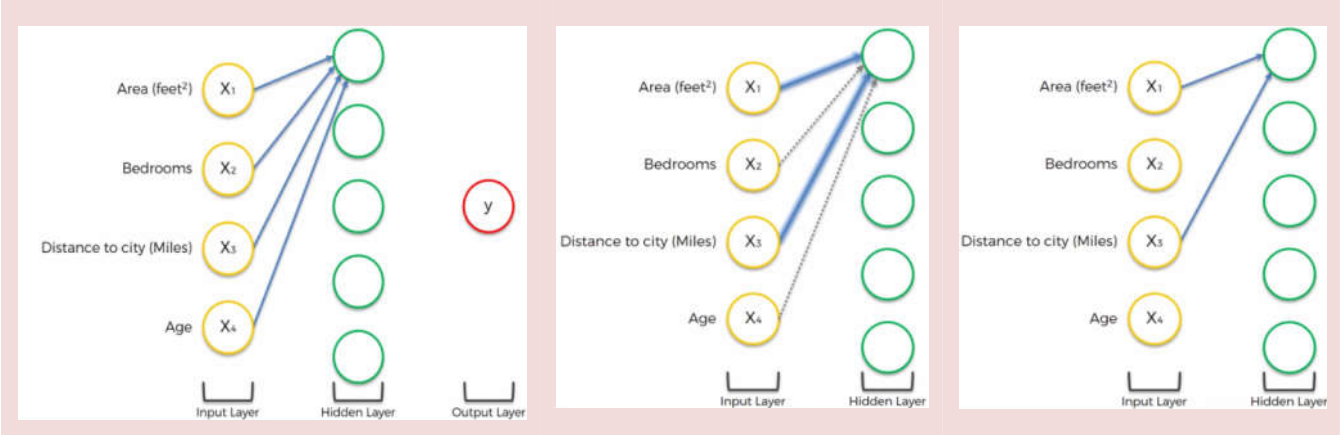
The Hidden layers- The advantage of NNs: In neural networks we have **hidden layers**, which is an advantage that gives us lots of **flexibility** and **power** that increase the accuracy.

- Now we're going to understand how that hidden layer gives us that extra power. We're going to walk through this example. Since we assume that this **NN has already trained up**, then we're just going to ***walk step by step through*** how the ***neural network*** will deal with the ***input variables*** and calculate in the ***hidden layers*** and then calculate the ***output-layer***.
- We've got ***all four variables*** on the left and we're going to start with the ***top Neuron*** on the ***hidden layer***.



❒ ***Not all variables are important for some neuron:*** Now we previously saw that ***all*** of the ***neurons*** from the ***input layer*** they have ***synapses*** connecting each one of them to the ***top neuron*** in the ***hidden layer***.

☛ But those ***synapses*** have ***WEIGHTS***. Now some **weights** will have a **non-zero value** and some **weights** will have **zero value**, because ***not all inputs*** will be ***valid*** or all inputs ***won't be important*** for ***every single neuron***. Some inputs **will not be important** and **neglected** by **some neurons**.



☛ Here we can see two variables x_1 and x_3 the ***area*** and the ***distance to the city*** are important *for that first neuron* whereas ***bedrooms*** and ***age*** are not.

☛ ***We can explain as:*** The further away you get from the **city** the **cheaper** real estate becomes, hence the **space in square feet** of properties becomes ***larger***. So for the **same price** you can get a **larger property** the further away you go from the city.

☛ ***And probably what this neuron is doing:*** it is looking for ***area*** variable which are **not so far from the city** but have a ***large area***. So for their **distance from the city** they have an **unfair area**.

☯ So that **neuron** might be picking out those **specific properties** and it will **activate** the **activation function** only when the ***certain criteria is met***. It performs on calculations inside itself and it combines those two variables ***area*** and the ***distance to the city*** and that contributes to the **price** in **output**.

⇨ And therefore this **first neuron** doesn't really care about the variables ***bedrooms*** and ***age*** because it's focused on that specific thing.

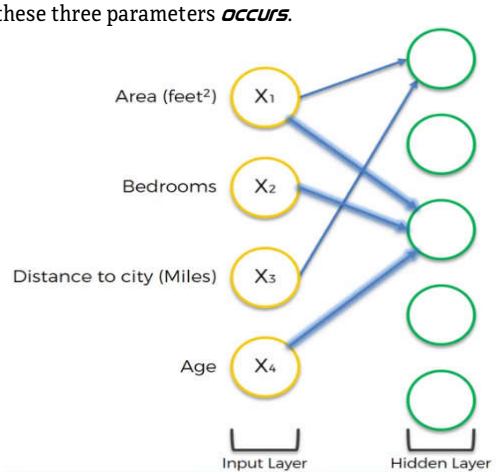
⇨ Now let's not even draw these lines for the synapses that are neglected.

☝ That's where the ***Power*** of the ***Neural Network comes from*** because you have ***so many of these Neurons*** each focusing on specific ***criteria.***

☯ Let's take one in the middle neuron. Here we've got **three parameters** feeding ***Area***, ***Bedrooms*** and ***Age***. So what exactly that neuron is doing? Why this neuron through all of the thousands of examples of properties has found out that the ***Area***, ***Bedrooms*** and ***Age,*** combination of those parameters is important?

⇨ ***The reason could be:*** In the area/city data this model is trained, there are some people looking for ***Larger properties*** with ***lots of bedrooms*** and the ***age of the property is low*** (i.e. new property). Thos people could be *New couples with new jobs and better income* or could be *larger families with old parents and grandchild's*. The common thing about those people that, they don't care about the **distance from the city**.

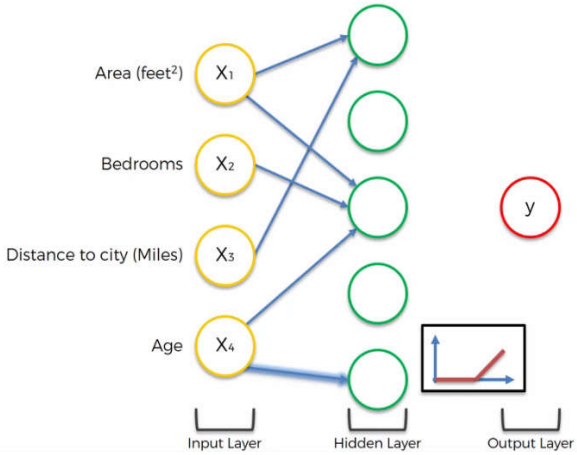
⇨ Hence this specific neuron is looking for these three properties (variables), as soon as that **criteria** is **met** the **neuron fires up** and the ***combination*** of these three parameters ***occurs***.



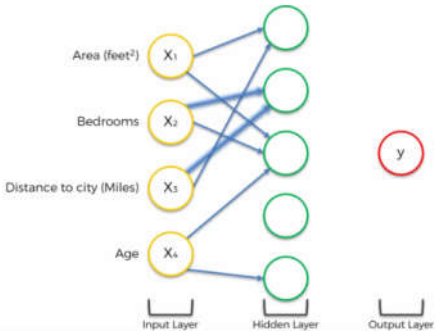
So this is the ***Power of the Neural Network***, it ***combines these parameters into a brand new parameter*** that helps with the ***evaluation*** of the ***property***.

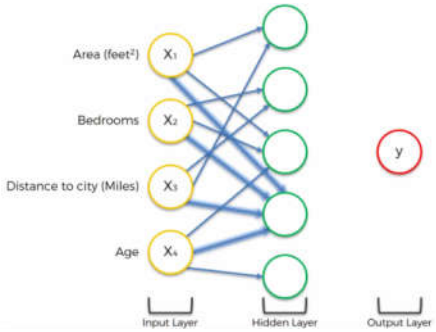
Let's look at another neuron, at the very bottom one, for instance this neuron could be picked up just one property: Age. The criteria behind can be: some properties are more valuable when it is too old. For example: a 100 year old property can be a historic place and some **Elite/Rich Family** want to buy it for "show off their friends". Hence this neuron only aims to the ***Age variable***(property).

⇨ This can be perfect to apply ***Rectifier Activation function***, because after certain ***age limit***, the ***value*** of the property ***gets higher***.



And also a neuron can consider only no. of bedrooms and distance. Another neuron can consider all of the variables. And so on. The pint is there can be so many options for the neurons. That's the power of the NNs.

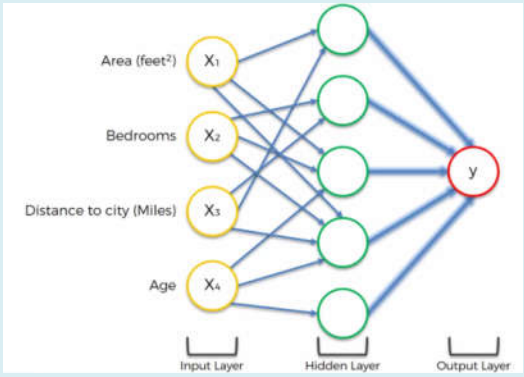




So you can see that these ***neurons*** and this whole ***hidden layer situation*** allows you to ***increase*** the ***flexibility*** of your ***neural network*** and allows you to look for very specific things and then in combination they predict the price.

That's the power of NNs. Like an ***ant***, by itself cannot build anything. But when you have 100000 ants they can build an ***Anthill*** together. And that's the situation here.

Each one of these ***neurons*** by itself ***cannot predict*** the ***price***. But together they have ***super powers*** and they ***predict the price*** and they can do quite an ***accurate*** job if trained ***properly,*** set up ***properly***.



8.2.2 How an NN Learns

There are two fundamentally different approaches to getting a program to do what you want it to do.

a) ***Hard coded source-code:*** where you actually tell the program's ***specific rules*** and what ***outcomes*** you want. Guide the program throughout the whole way and define all the ***possible options*** that the program has to deal with.

b) ***Neural Networks:*** On the other hand you have neural networks where you **automate** the program to be able to understand what it needs to do **on its own**. In this **NN** you provided **inputs**, tell it what you want as **outputs** and then you let it **figure** everything **out** on its **own**.

Our goal is to create this network ***which learns on its own***. We going to avoid trying to put in the rules.





For example: Distinguish between a ***Dog*** and ***Cat***.

- ***Option 1:*** You would use hard-coded program using different characters: like the ***cat's ears***, look out for ***type of nose***, look out for ***type of shape*** or ***colors*** etc.
- ***Option 2:*** On the other hand for a **neural network** you just code the **neural networks architecture** and then you ***point*** the ***neural network*** at a ***folder*** with ***images*** of all these ***cats*** and ***dogs***, which are already categorized. From those images of cats and dogs ***NN*** going to ***learn by itself*** what a cat or dog looks like.
  - Once ***NN is trained up*** then you give it a ***new image*** of a ***cat*** or ***dog*** it will be able to ***understand*** what it was.

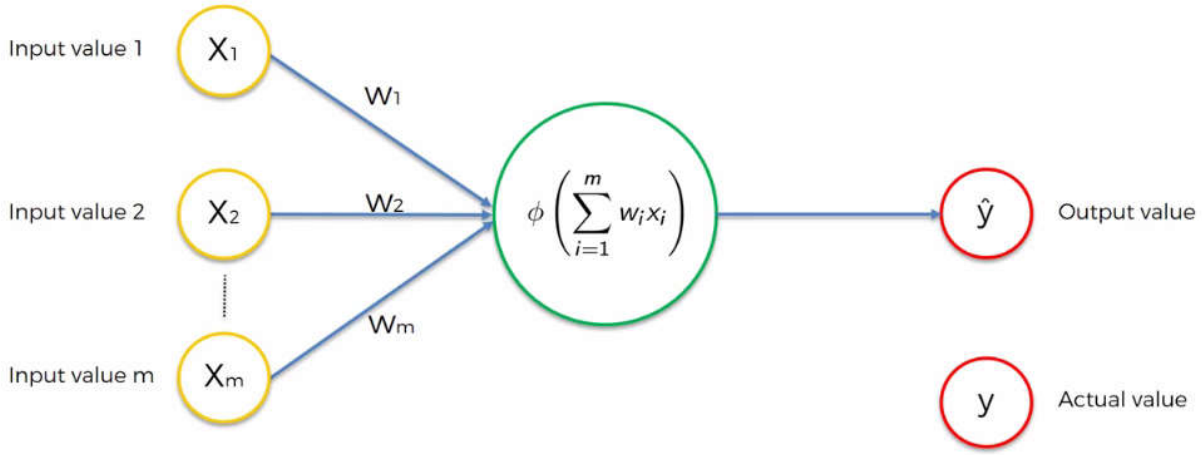


8.2.3 PERCEPTRON

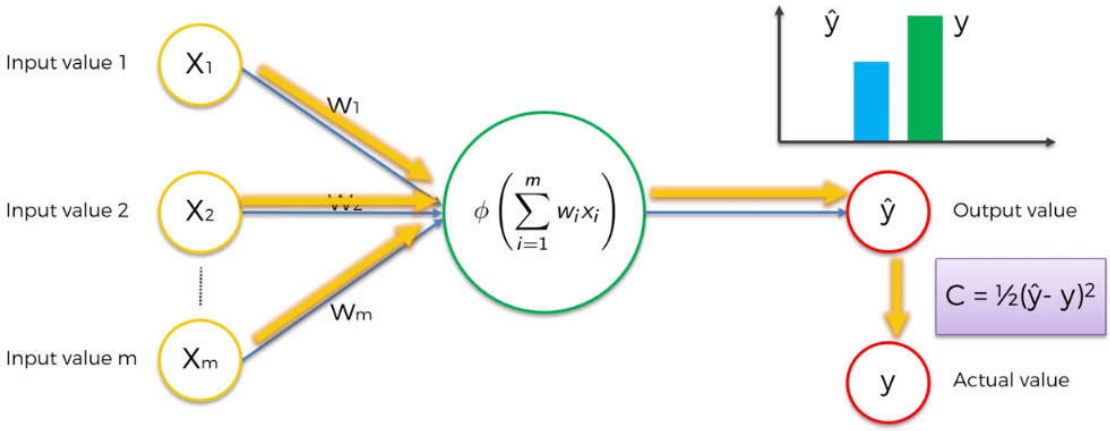
Here we have a very ***basic neural network*** with a ***one layer***. It is called a ***perception***.

y stands for the actual value
$\hat{y}$ output value, it is predicted by the neural network.

And the perception that was first invented in 1957 by ***Frank Rosenblatt*** and his whole idea was to create something that can actually ***learn*** and ***adjust itself***.



- ***Let's see how our perception learns:*** Say we have some ***input values*** that have been supplied to the ***PERCEPTION*** (basic NN).
  - Then the ***activation*** function is ***applied*** and we get an ***output.***
  - Now we're going to plot the output value $\hat{y}$ and actual value y on a chart.



  - To make our NN to be able to learn we need to ***compare*** the ***output value*** to the ***actual value***. So we use a function called the ***cost function*** and is calculated as:

$$C = \frac{1}{2}(\hat{y} - y)^2$$

Now there are many ways of calculating ***cost function***. There are **many different cost functions** that you can use. We used the simplest form here.

➲ Basically the ***cost function*** is telling us about the ***error*** that we have in our ***prediction***. And our goal is to ***minimize*** the ***cost function*** because the lower the ***cost function*** the closer ***output value*** $\hat{y}$ and ***actual value*** y.

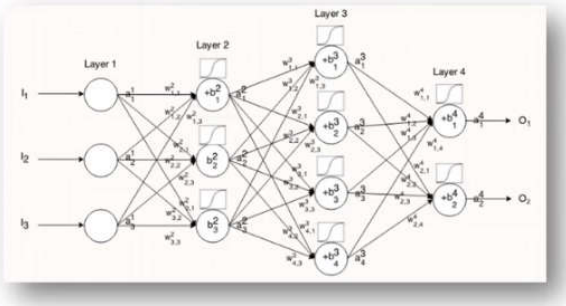
Additional Reading:

A list of cost functions used in neural networks, alongside applications

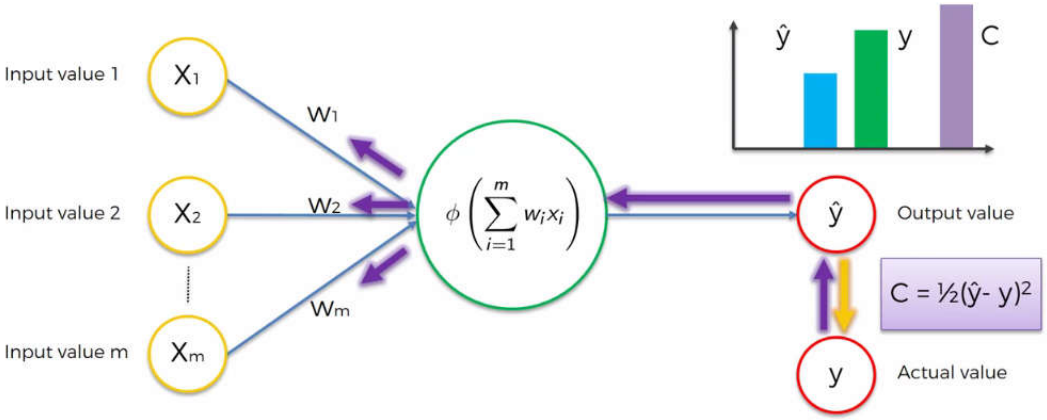
CrossValidated (2015)

Link:

http://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications

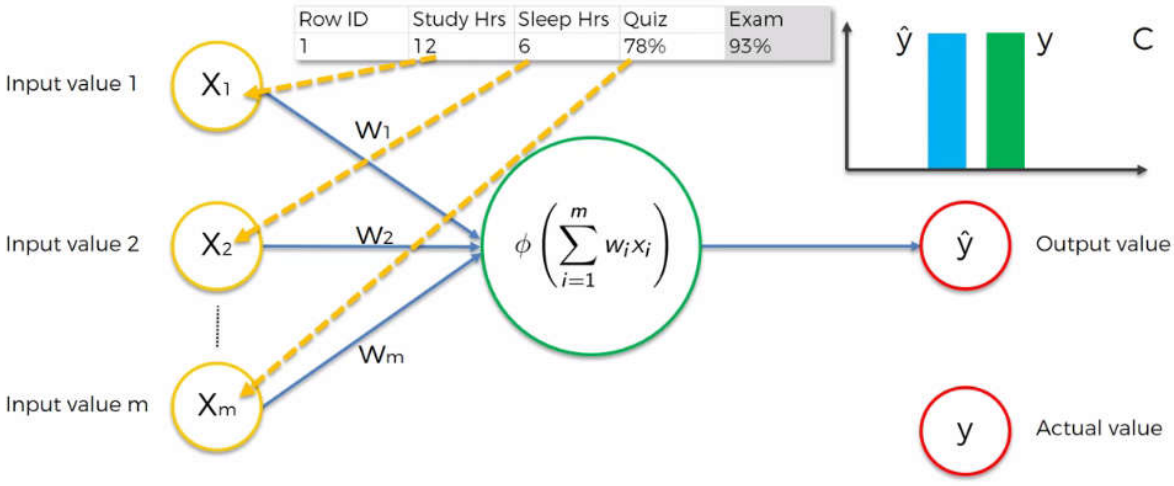


➲ Next step once we've compared and calculated the cost function, now we're going to ***feed this information back*** into the ***Neural Network***. The ***information*** going ***back*** into the ***NN*** and the ***weights*** get ***updated***.

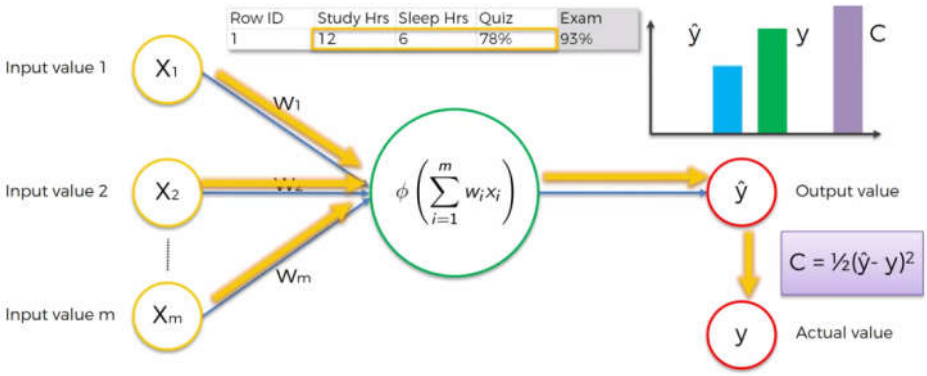


8.2.4 How an NN learns: Single Row of a Dataset

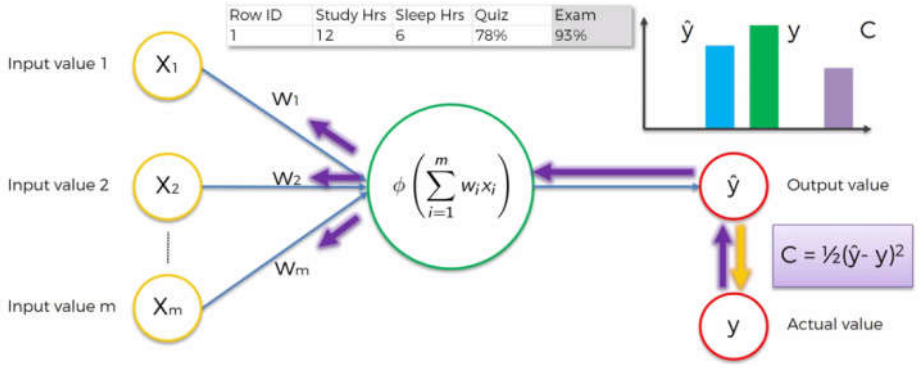
Example: Let's consider following screenshot of the data. From a dataset, of a row where we have: these independent variables- how long you **study**, how long you **sleep** and your **quiz score**. We're ***predicting*** the ***result*** you're going to get on an exam.



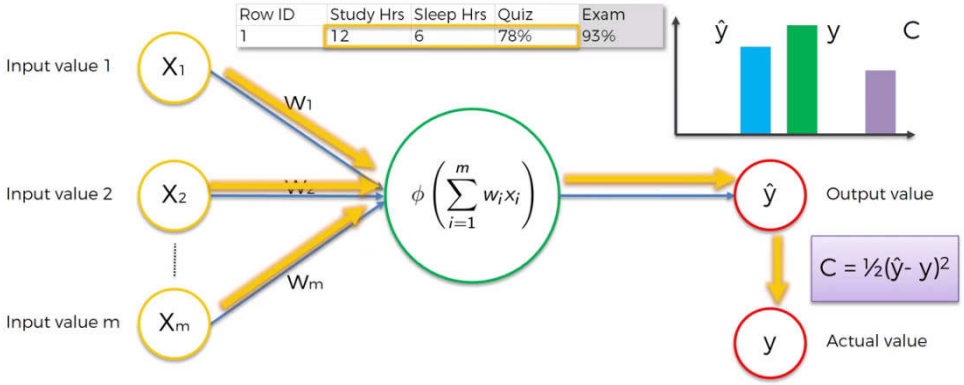
☯ So for first iterartion, we input these input values (Study hr, Sleep Hr, Quiz) in to NN, then we get $\hat{y}$. Comparing to actual value y, we get the cost value C.



☯ Once we've compared and calculated the ***cost function***, we're going to ***feed this information back*** into the ***NN*** and the ***weights*** get ***updated***.

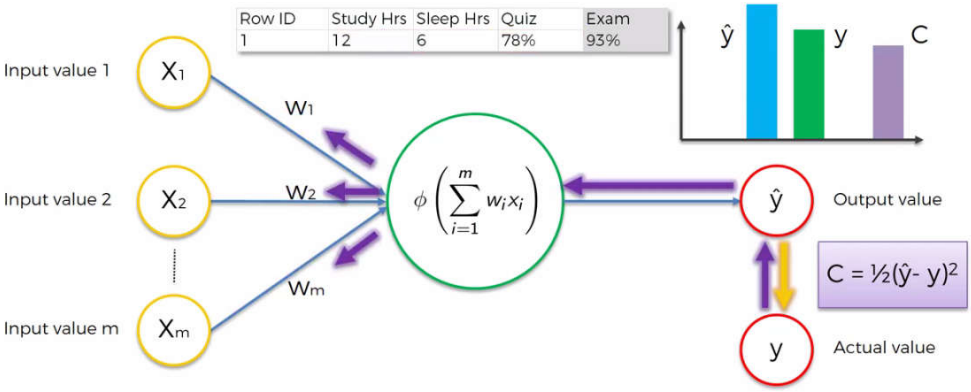


☯ So we feed these three values into the NN again for the second time (now the ***weights*** are ***updated***) then we're going to be comparing the result $\hat{y}$ to y.



Iteration 2

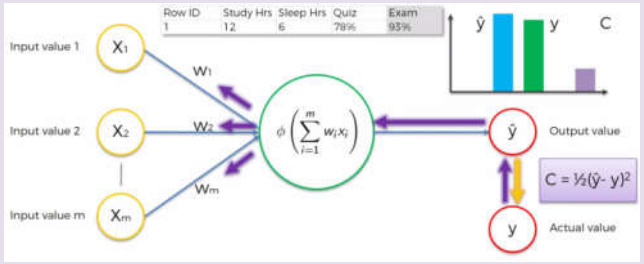
☯ ***Cost function*** is calculated again and ***weights*** are ***adjusted*** again.

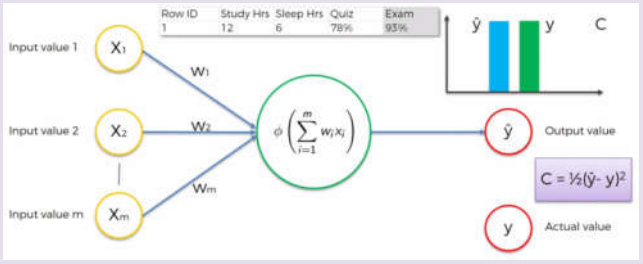


☝ We continue this iteration until cost-function, $C = \frac{1}{2}(\hat{y} - y)^2$ is minimum. That is $\hat{y}$ and y gets equal. Usually you won't get cost function equal to zero.

➢ Every time $\hat{y}$ is changing because we've tweaked the weights. Hence ***cost function*** changing also. We get information, then feedback this information to the NN so that the ***weights*** get ***adjusted again***.

☝ Here every time we feed in exactly that ***same row*** because just in this case we're ***dealing with that one row*** into our ***neural network***.





8.2.5 How an NN learns: Multiple Rows of a Dataset

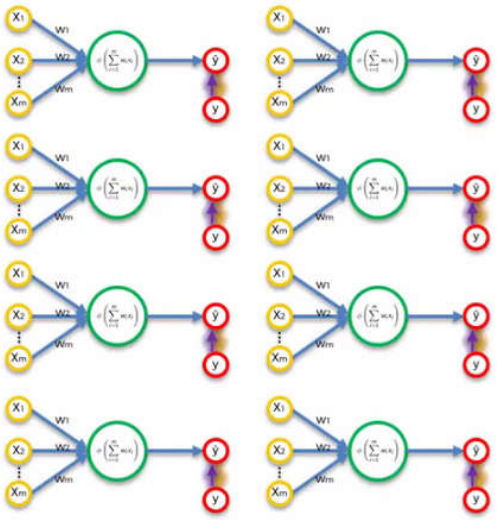
❒ Up until now we've been dealing with just that **one row**. So here's the full data set. We have eight rows of how long you **study**, how long you **sleep** and your **quiz score**.

Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

❒ And as you can see here on the left we've got eight of these PERCEPTRON. They are all the same PERCEPTRON so this is also important.

➢ I just duplicated eight times for the learning purpose. It is the same NN, we're going to be feeding these data into.

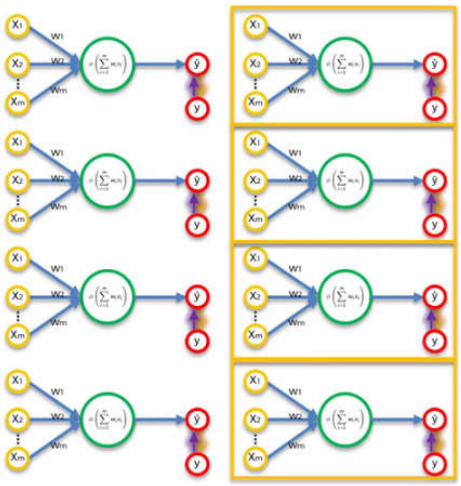
☝ ***Epoch:*** One epoch is when we go through a whole dataset and we train our neural network on all of these rows.



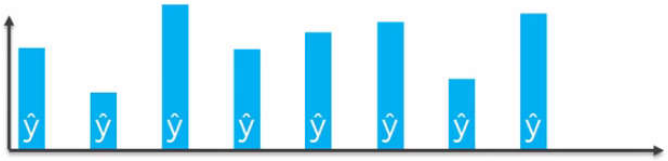
Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%



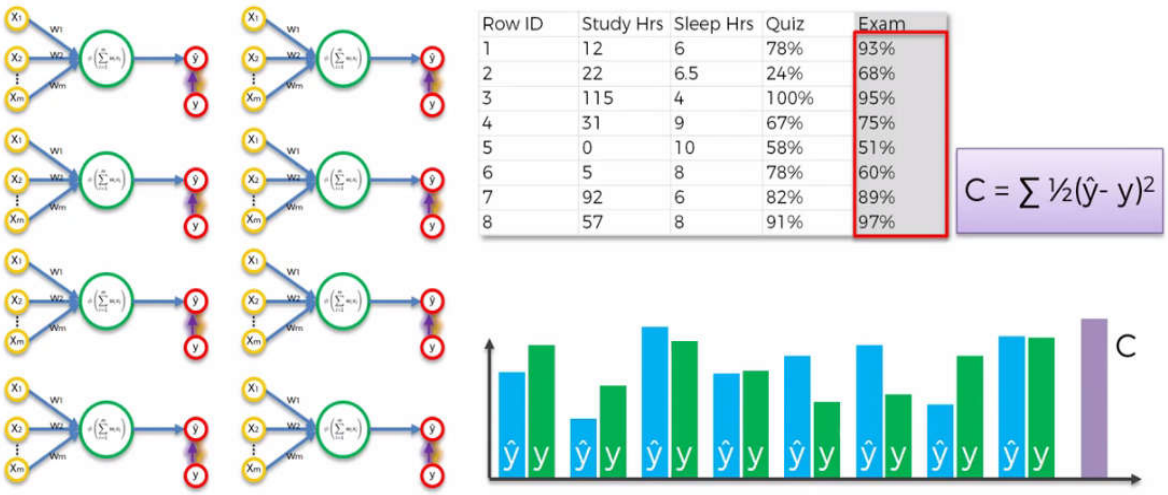
☯ For first iteration we input the rows one by one and get output $\hat{y}$ for each row. In following graph we calculated outputs for all 8 rows.



Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%



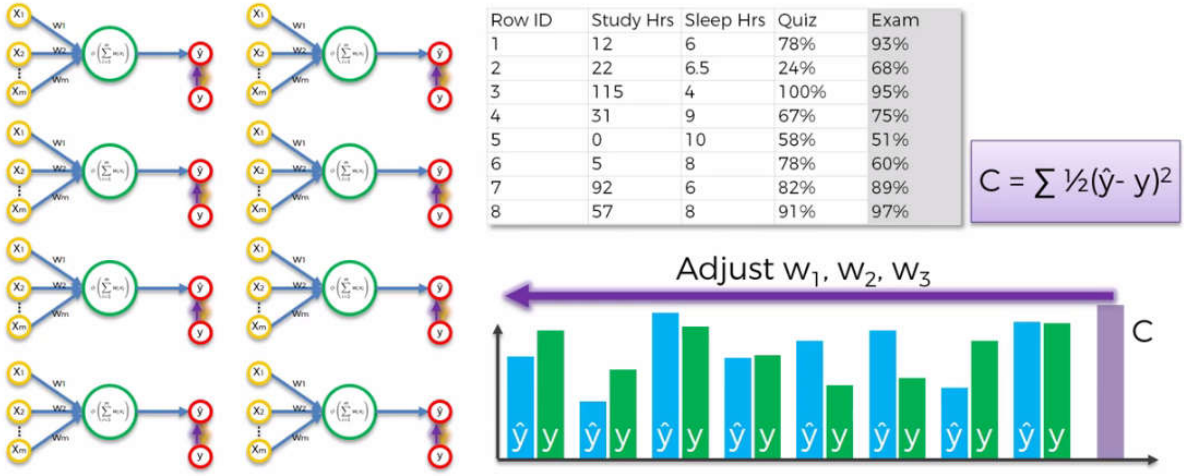
☯ Then we compare the ***outputs*** to the ***actual*** values. For **every single row** we have an **actual value** and corresponding **output value**. And now based on all of these **differences** between $\hat{y}$ and y we can calculate the ***cost function*** which is the **sum of all** of those **squared differences** between $\hat{y}$ and y.



☯ ***Total Cost Function:***

$$\text{Cost function, } C = \sum_{i=1}^{8} \frac{1}{2}(\hat{y}_i - y_i)^2$$

☯ After we have the ***full cost function*** we go back and we ***update*** the ***weights***. So we're going to ***update*** the weights in that ***one neural network*** (there are not eight of NNs there's just one NN) so basically the ***weights*** are going to be ***the same for all of the rows***. All the rows share the same weights.



[So it's ***not the case*** that ***each row has its own weight***.] Now that was just ***one iteration*** (we used all 8 rows, hence ***1 epoch*** is complete).

➢ Next we're going to run this whole ***process again***. We're going to ***feed every single row*** into the ***neural network*** find out our ***cost function*** and adjust the weight of the Synapses again.

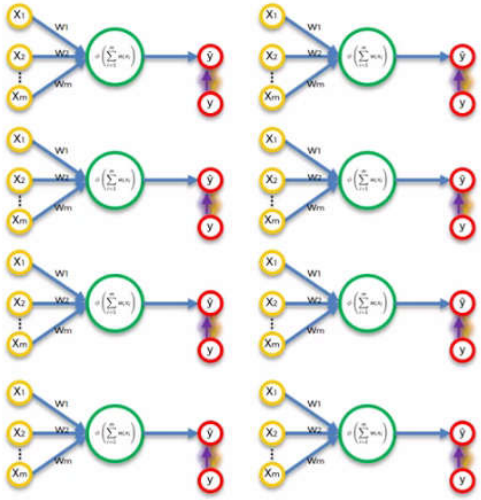
➢ We iterate until the ***cost function is minimum (or stable)***. What we did for one row. But now we're going to be doing it for 8 rows.

☝ We have here ***8 rows*** but it could be ***800 rows*** or ***eight thousand*** rows however many rows we have in our ***data-set***. You do this process and then you calculate the ***cost function***.

☞ And the goal here is to ***minimize*** the ***cost function***

☝ And **as soon as** you found a ***minimum cost function***, that is your ***Final Neural Network*** that means your ***Weights*** have been ***Adjusted*** and you have found the **optimal weights** for this dataset.

☞ You are done ***training*** your ***dataset*** and you're ready to the ***testing phase*** or ***application phase***. And this whole process is called ***Back Propagation***.



Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

$C = \sum \frac{1}{2}(\hat{y} - y)^2$



8.2.6 Additional Reading : List Of Cost Functions

❒ So some additional reading that you might want to do for the ***cost function*** and good article is located on ***"cross-validated" website***. It's called ***"A List Of Cost Functions Used In Neural Networks Alongside Applications"***. You can just Google for that exact search term go to the website and read the article.

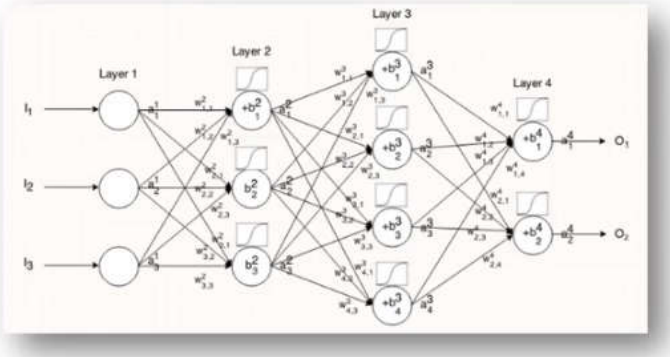
☞ It's actually got some good examples and application or use cases for different cost functions so if you're interested to learn more about cost functions Check out this article.

Additional Reading:

A list of cost functions used in neural networks, alongside applications

CrossValidated (2015)

Link:



http://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications

Chapter 8 : Part 3

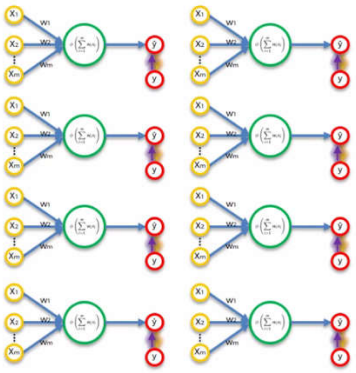
Deep Learning

ANN: Gradient Decent

Gradient Decent, Stochastic Gradient Decent and Backpropagation details

8.3.1 Why Gradient Decent?

In this section we're talking about ***gradient descent***. We saw in previous section, that an ***NN*** learns using ***back-propagation***. That is, when the **error/difference** or the **sum of squared differences** between y and $\hat{y}$ is ***back propagated*** through the ***neural network*** and the ***weights*** are ***adjusted*** accordingly. Now we're going to learn exactly **how** these **weights** are **adjusted**.



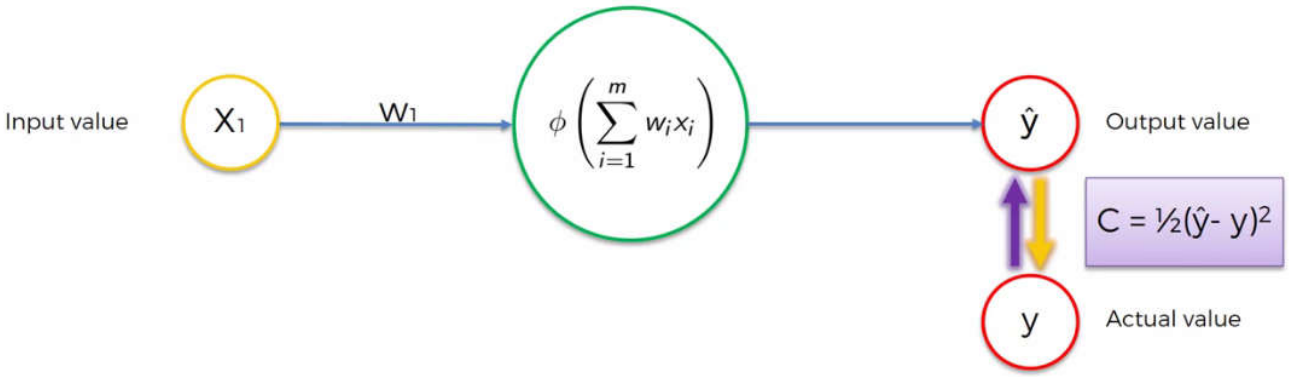
Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

$C = \sum \frac{1}{2}(\hat{y} - y)^2$



❒ ***Gradient Descent*** is kind of **similar** to some **Numerical techniques** like ***Bisection*** or ***Regula falsi*** methods.

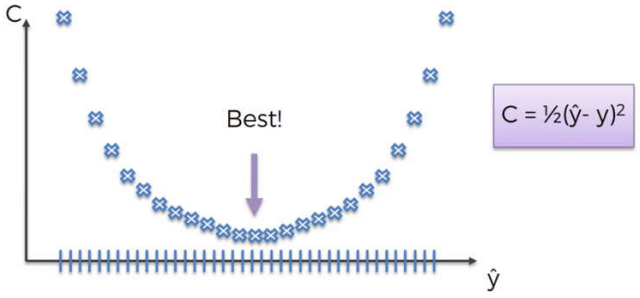
❒ Now consider the very *simple version of a neural work* or a **PERCEPTRON**, a **single layer feed-forward NN**. We can see here is the whole process in action: we've got some input value, we've got a weight then a activation function is applied. Then we got **predicted** value $\hat{y}$ and we compare it to the **actual** value y, then we calculate the **cost function**.



☞ Now the question is: ***How can we minimize the cost function?***

☞ Well one approach to do it is a **Brute Force Approach** where we just take all **lots of different *possible weights*** and look at them and see which one looks best.

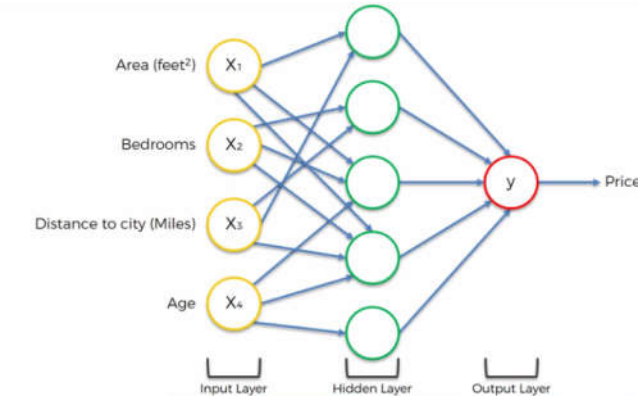
- In this approach we would try out weights , for example: 1000 weights. Then we get something like this:
- For the cost function and this is a chart. On the Y axis we have cost function and on X-axis we have output value $\hat{y}$. You'd find the ***best value*** at the **bottom** of that **curve**.



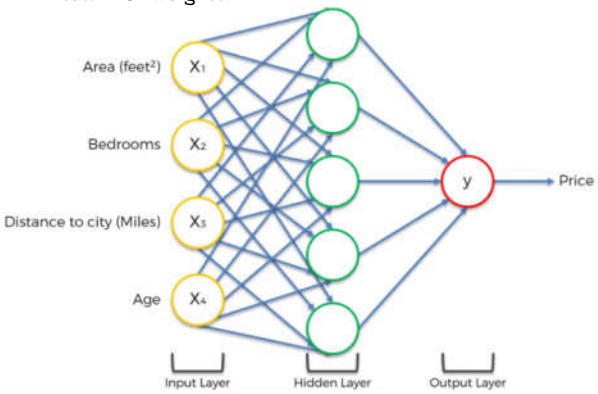
❒ ***Why this brute force method is not efficient?:*** Well if you have just **one weight** to **optimize** this might work but as you increase the *number* of ***weights*** (i.e. *increase* the *number* of ***Synapses*** in your network) you have to face the ***Curse of Dimensionality***.

❒ ***Curse of Dimensionality:*** Recall the example, when we were talking about **how neural networks actually work** where we were building a NN for a **Property Evaluation** (Recall Section 8.2.1).

☞ So this is what it looked like when it was ***Trained Up Already***. Here we know which one what are the best weights.



☞ But ***Before Training*** we didn't know about the optimal weights. The actual NN before training looks like this. Where we have all these different possible Synapses and we still have to train up the weights. Here we have a total of 25 weights: ***input*** ($4 \times 5 = 20$) and ***output*** (5) so total ***25*** weights.



☞ And let's see how we could possibly brute force 25 ways. Now it is a very simple neural network we have right now. We have just one Hidden Layer.

☞ But to do this we have to try 10^{75} possible combination. Which is impossible even we use a super computer (using 93 peta FLOPS. 1 FLOP = 1 floating point operation per second. Normal computer can do several giga FLOPs).

$1{,}000 \times 1{,}000 \times \ldots \times 1{,}000 = 1{,}000^{25} = 10^{75}$ combinations

Sunway TaihuLight: World's fastest Super Computer

93 PFLOPS

93×10^{15}

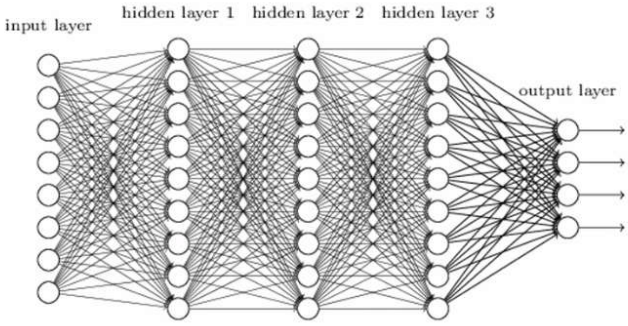
$10^{75} / (93 \times 10^{15})$

$= 1.08 \times 10^{58}$ seconds

$= 3.42 \times 10^{50}$ years ✖



☝ Now for an NN like following it is just impossible to try out the Bruit-Force technique.



NOTES:

☝ ***Brute-Force Search:*** In computer science, ***Brute-Force*** search or ***Exhaustive*** search, also known as ***Generate And Test***, is a very general problem-solving technique and *algorithmic paradigm* that consists of *systematically enumerating* all *possible candidates* for the *solution* and checking whether *each candidate satisfies* the problem's statement.

☝ ***Brute Force*** Algorithms are exactly what they sound like – ***straightforward*** methods of ***solving*** a ***problem*** that rely on ***sheer computing power*** and trying ***every possibility*** rather than ***advanced techniques*** to improve ***efficiency***.

☞ For example, imagine you have a **small padlock** with 4 digits, each from **0-9**. You **forgot** your **combination**, but you don't want to buy another padlock. Since you can't remember any of the digits, you have to use a **brute force method** to open the lock.

☞ So you set all the numbers back to 0 and try them one by one: 0001, 0002, 0003, and so on until it opens. In the **worst case scenario**, it would take 10^4, or 10,000 tries to find your combination.

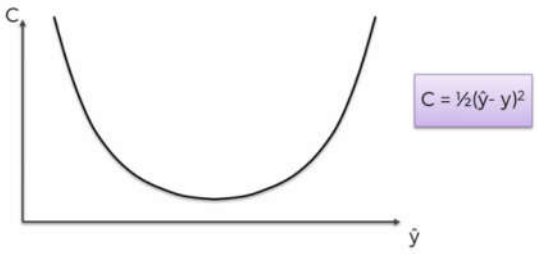
☞ A classic example in computer science is the **Traveling Salesman Problem** (TSP). Suppose a salesman needs to visit ***10 cities*** across the country. How does one determine the order in which those cities should be visited such that the total ***distance traveled*** is ***minimized***?

☞ The brute force solution is ***simply to calculate*** the ***total distance*** for ***every possible route*** and then select the ***shortest one***. This is not particularly ***efficient*** because it is possible to eliminate many possible routes through ***clever algorithms***.

Gradient Decent: Similar to Numerical methods: Bisection/Regula-falsi

8.3.2 Gradient Decent

It is called gradient descent because we calculate the Gradient every-time and check if it is descending.



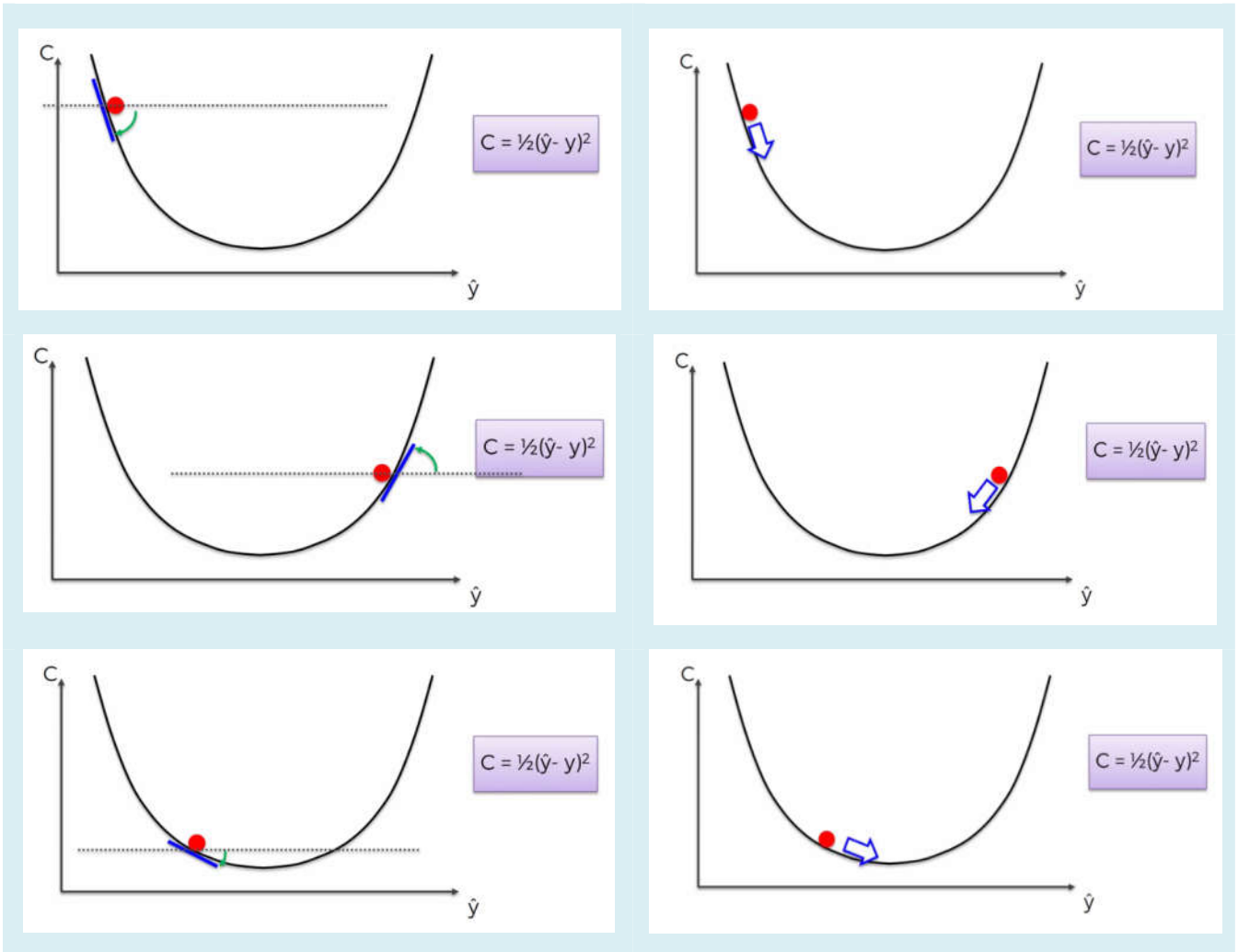
❒ So here is our **cost function**, and now we going see **how** we can find our ***best value*** faster.

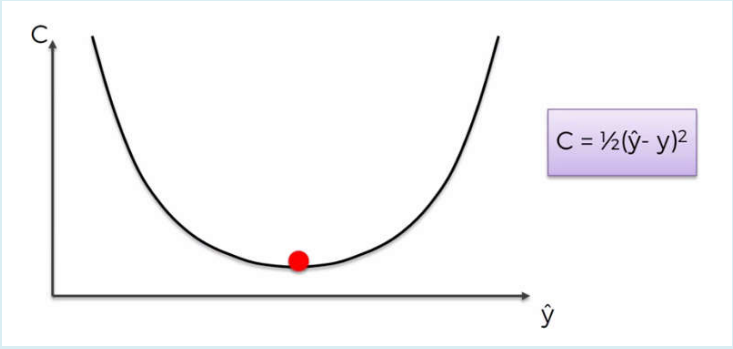
☞ Lets say we **start** somewhere at point in the **top left**, we're going to look at the **angle** of the **tangent** drawn at that **point** of our **cost function** then we calculate the gradient (you have to differentiate).

☞ You just need to ***differentiate*** to find out what the ***slope*** is in that ***specific point*** and find out if the ***slope*** is ***positive*** or ***negative***.

➢ If the ***slope*** is ***negative*** (like in this case), means that you're ***going downhill*** (so to the ***right*** is ***downhill*** to the ***left*** is ***uphill***), so you need to ***go right*** i.e. you need to ***go downhill*** (find the best/optimal point. Something like ***"finding global minima"***).

☞ Lets say we go to ***right***, and again we calculate the ***gradient*** and find the ***slope*** is ***positive***. That means we are ***too right*** we need to go ***left***.

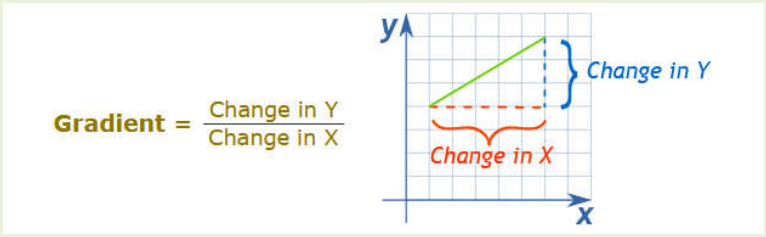




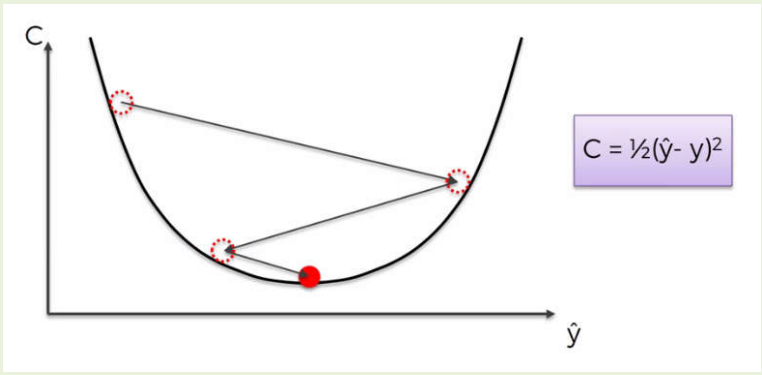
☝ ***Gradient:*** The **Gradient** (also called **Slope**) of a **straight line** shows how **steep** a straight line is.

☞ To calculate the Gradient, divide the ***change in height*** by the ***change in horizontal distance.***

$$Gradient = \frac{change\ in\ height}{change\ in\ horizontal\ distance}$$



☝ To remember it as a fun way to think it as a ball rolling. But in reality it's going to be like a step by step approach is going to be a **zigzag** type of **method**.

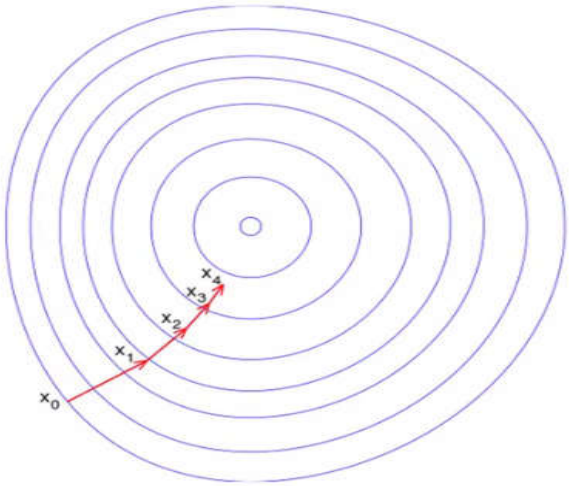


☞ There's also lots of other elements to it. For instance: Why does it go down why does it ***not go way over*** the line or it could have gone ***upwards*** instead of ***downwards***. So there are ***parameters*** that you can tweak.

☞ Here, we are getting to the bottom to understanding which way we need to go. Instead of ***Brute-force*** through ***billions and quadrillions*** of ***combinations***. We can simply look at which way is it ***sloping***. ***right/left***. Then we try to get to the ***bottom***. (like you're standing on a hill. Which way does it feel that it's going downwards).

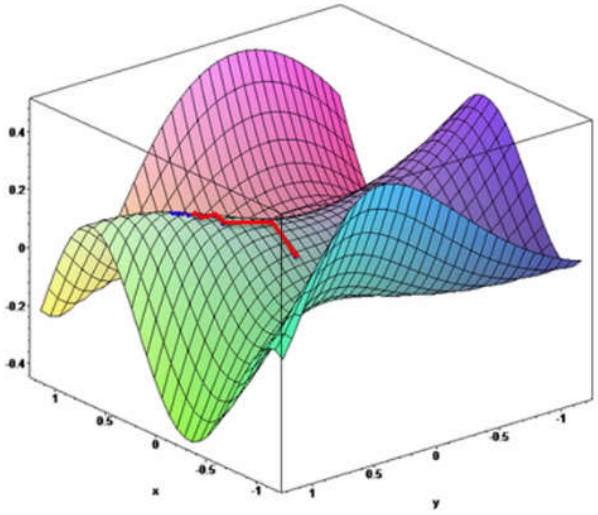
8.3.3 Gradient Descent in 2D

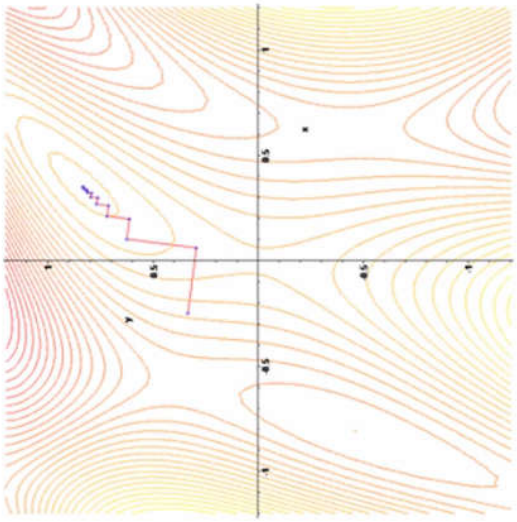
Here's an example of **gradient descent** applied in a **two dimensional** space. You can see it's getting **closer** to the **minimum** and hence it is called ***gradient descent*** because you're ***descending*** into the ***minimum*** of the ***cost function***.



8.3.4 Gradient Descent in 3D

Here is a gradient descent applied in 3D. And if you projected onto **two dimensions** you can see ***zigzagging*** its way into the ***minimum***.

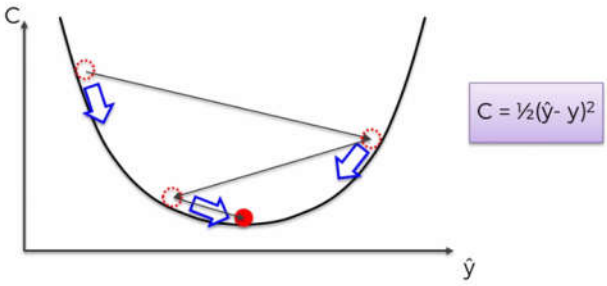




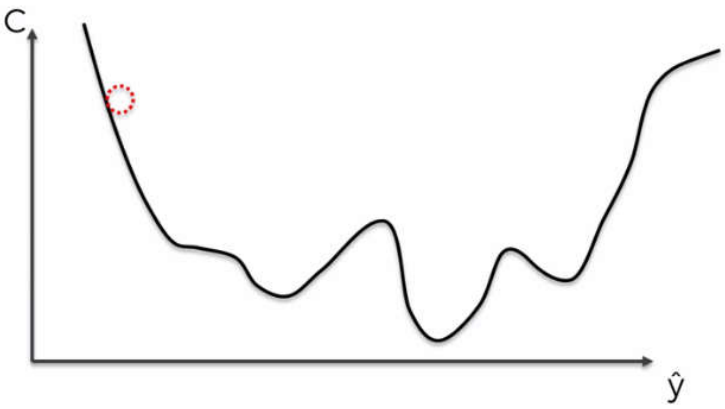
8.3.5 Stochastic Gradient Decent

Above we see that gradient descent is a very efficient method to solve our ***optimization problem*** where we're trying to ***minimize*** the ***cost function***.

It basically takes us from 10^{57} years to solving a problem within ***minutes*** or ***hours*** or within a ***day*** or so. And it really ***helps speed things up*** because we can see which way is ***downhill*** and we can just go in that ***direction*** and get to the ***minimum*** faster.



❒ ***Non- CONVEX cost-function:*** But the problem is, this method ***requires*** for the ***cost function*** to be ***convex***. Which has ***only one*** **global minima**. What if our **cost-function** is ***not CONVEX***. What if it looks something like this:

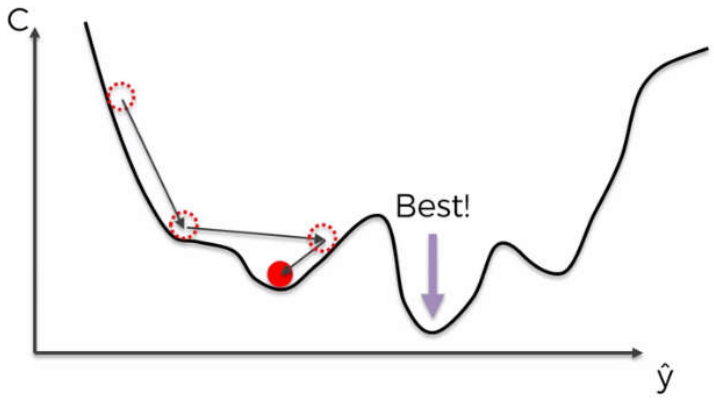


☞ ***This could happen:***

a) If we choose a **cost function** which is **not the square difference** between $\hat{y}$ and y or

b) If we do choose the cost function as **square difference** between $\hat{y}$ and y but in a ***multi dimensional space*** it can actually turn into something that is ***not convex***.

☞ ***The local-minima trap:*** In these cases if we apply our normal **gradient decent method** we will end up something like following. Which is not the **global minima,** we'll end up with the **local-minima**. But the ***best value*** is the ***Global-minima***.



We could find a **local minimum** of the cost function rather than the **global** one.

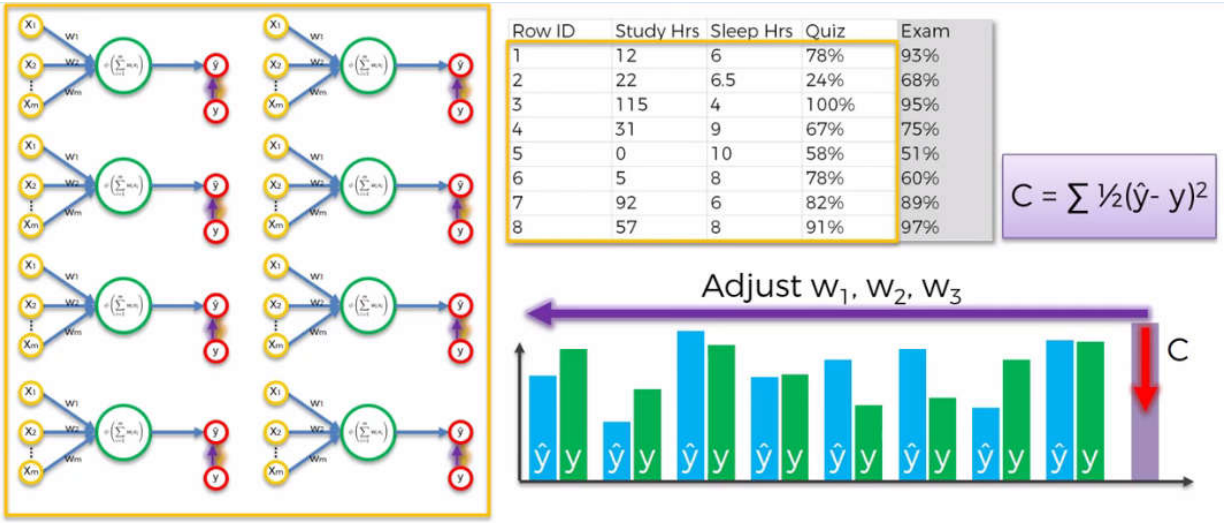
➲ *Global-minima* is the ***best*** one and we found the wrong one (the *Local-minima*) and therefore we **don't have** the **correct *weight***. As a result we **failed** to find an **optimized neural network**.

❒ ***So how to avoid this trap?***

The solution is ***Stochastic Gradient descent***. *Stochastic gradient descent* doesn't require for the **Cost-function** to be **convex**.

☞ So let's have a look at the **differences** between the normal ***Gradient Descent*** and the *stochastic gradient descent*.

☞ ***Batch Selection in normal Gradient descent:*** In normal ***Gradient Descent*** we take **all of our rows** we ***`plug them`*** into our ***`neural network`*** all at once.



Batch Selection in normal Gradient descent

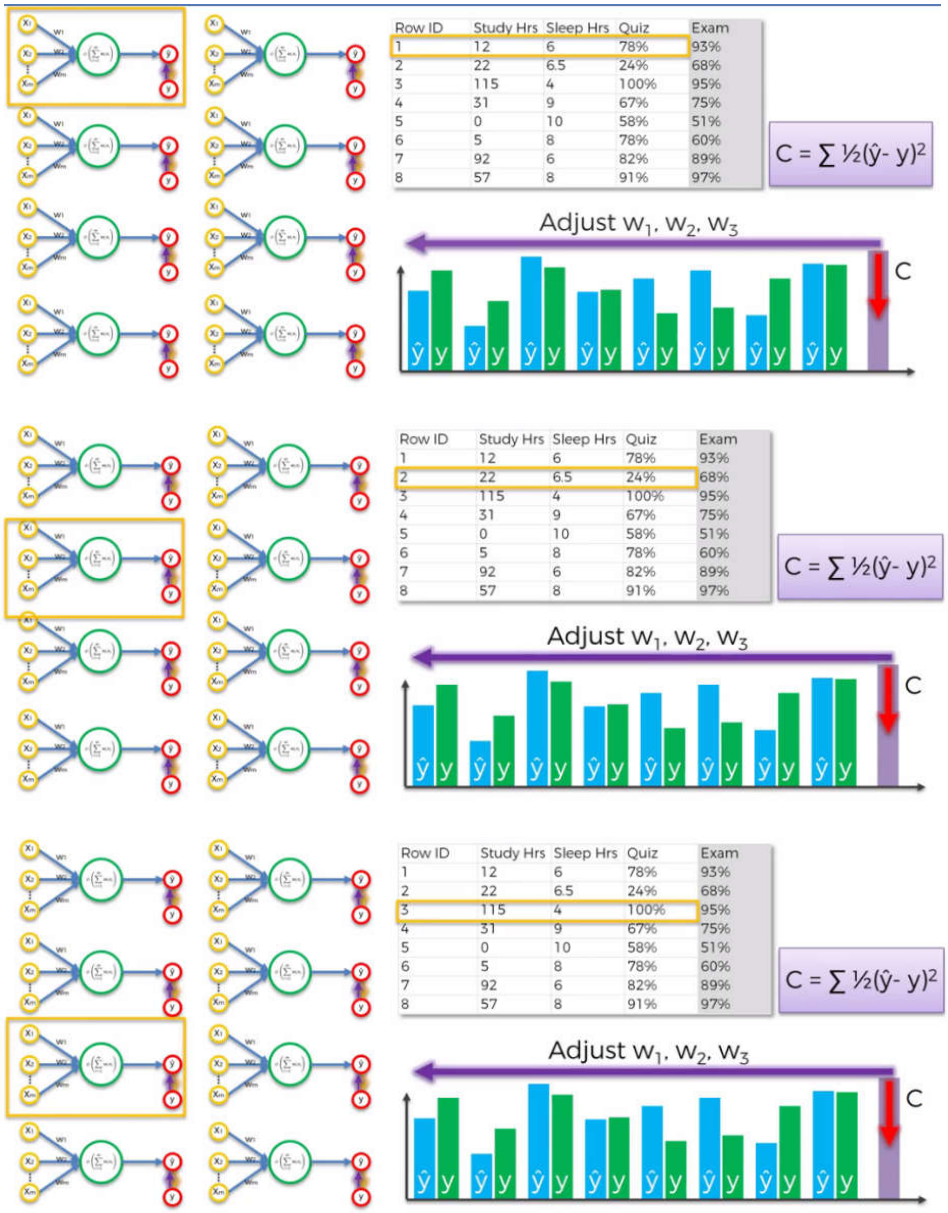
➲ Here we've got the *neural network* (copied over several times) the ***rows*** are being plugged into that *same neural* network every time.

➲ Once we plug them in, we've *calculated* our *cost function* and then we ***adjust*** the ***weights***. This is called the ***`gradient descent method`*** or in the proper term ***`"batch gradient descent method"`***.

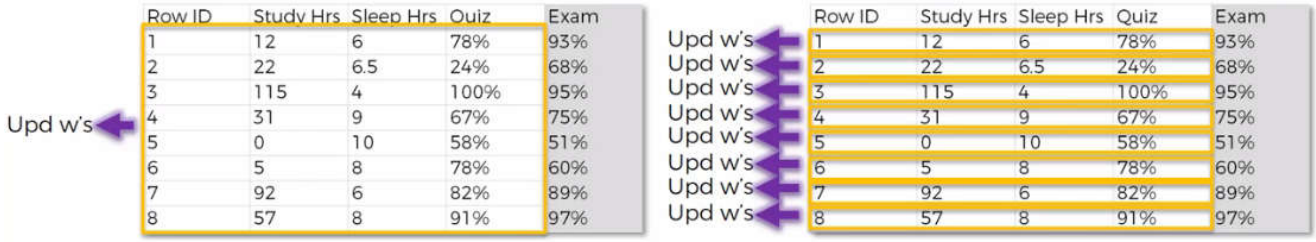
❒ ***Stochastic Gradient Descent:*** The stochastic gradient descent method is a bit different. Here we take the ***rows*** **one by one**. We take that row, we run our NN and then we ***adjust*** the ***weights***.

☞ Then we move onto the **second row** we run our NN again, then ***adjusted*** the ***weights*** again. Then we move to 3rd row. And so on.

☞ So basically we're *adjusting* the *weights* after *every single row* rather than doing everything at once.



☞ And now we're going to just compare the two ***side by side***. For visually remember them.



Batch Gradient Descent

Stochastic Gradient Descent

☞ In the **Batch Gradient Descent**, you are **adjusting** the **weights** *after you've run all of the rows* in your **NN** and then **adjust** the **weights** and that's the **first iteration**.

➲ For the ***2nd iteration*** we repeat whole thing ***adjusting*** the ***weights*** and then we do everything again for ***3rd iteration***.

☞ The ***Stochastic Gradient Descent*** method helps you to avoid the problem with ***local extrema*** or ***local minima*** . It helps to find ***global minima***.

➲ And the reason is that ***SGD*** or ***STOCHASTIC GRADIENT DESCENT*** method has much higher fluctuations, because it is doing ***one iteration*** on ***one row at a time*** and therefore the **fluctuations are much higher** and it is ***much more likely to find*** the ***global minimum*** rather than just the ***local minimum***.

☞ ***SGD is more faster than BGD:*** **Stochastic Gradient Descent** has more **advantages** over the **Batch Gradient descent** method. At the first impression you might have think that **Batch Gradient Descent** is more faster because it's doing **all row once** at a time. But in reality **Stochastic Gradient Descent** is faster because **it doesn't have to *load up all the data*** into ***memory*** and *run and wait* until all of those rules are on altogether.

➲ The main advantage is that ***BGD*** is a **Deterministic algorithm** but ***SGD*** being a **Stochastic algorithm** (meaning it's random).

➲ With BGD as long as you have the ***same*** starting ***weights*** for your ***NN***. Every time you run, you will get the **same iterations** to update the **weights**.

➲ But for ***SGD*** you won't get that because it is a ***stochastic method*** you're picking your ***rows*** possibly at ***random*** and you are updating your NN in a ***stochastic manner***. Therefore in ***SGD*** even if you have the **same weights** at the **start** you're going to have a **different iterations** to get there.

8.3.6 Mini Batch Gradient Descent

There's a method in-between the ***BGD*** and ***SGD***, it is called the ***Mini Batch Gradient Descent*** (mini BGD) method. Where you are running ***smaller batches of rows*** rather than running ***whole batch*** once at a time.

❒ From your ***whole dataset***, from ***all the rows***, you set that ***batch size*** (number of rows). You ***divide*** all the rows into ***several groups*** and you run your ***NN*** through each group and update the ***weight***.

And that's called the ***Mini Batch Gradient Descent*** method if you'd like to learn more about gradient

❒ ***Additional Reading:*** There's a great article which you can have a look at. It's called ***A Neural Network In 13 Lines Of Python Part 2 Gradient Descent*** by ***Andrew Trask*** and the links below it's an good 2015 article very well-written, very simple terms. You will got some very cool **tips-tricks** and **hacks**.

☺ It discussed on how to apply ***Gradient***,
☺ know **advantages** and **disadvantages** and
☺ how to do things in certain situations so

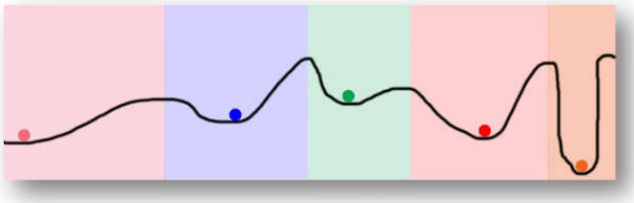
Additional Reading:

A Neural Network in 13 lines of Python (Part 2 - Gradient Descent)

Andrew Trask (2015)

Link:

https://iamtrask.github.io/2015/07/27/python-network-part2/



❒ ***Book:*** There is another article, it's a bit more heavier to read. For ***those*** of you who are ***into mathematics*** who want to get to the ***bottom of the mathematics***. What is Gradient descent is specifically. ***What are the formulas*** that are ***driving Gradient*** and ***how is it calculate*** and so on. Check out the ***Article*** or actually the ***Book***.

☞ It's a free online book called ***Neural Networks And Deep Learning*** by ***Michael Nielsen*** 2015 book.

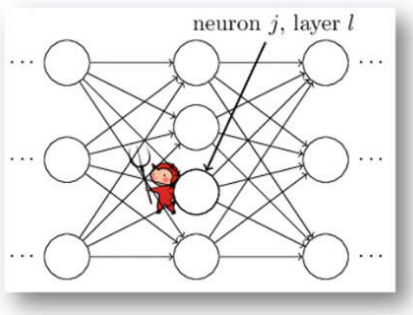
Additional Reading:

Neural Networks and Deep Learning

Michael Nielsen (2015)

Link:

http://neuralnetworksanddeeplearning.com/chap2.html

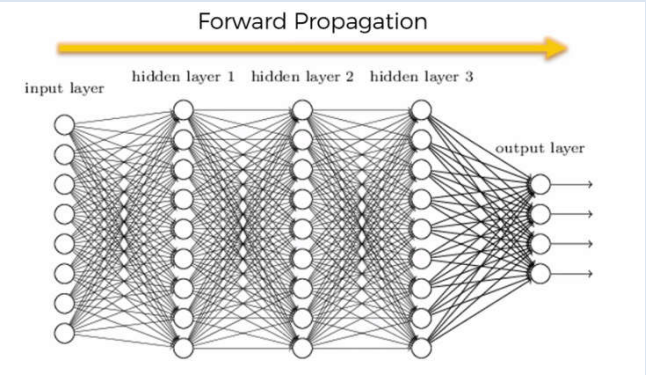


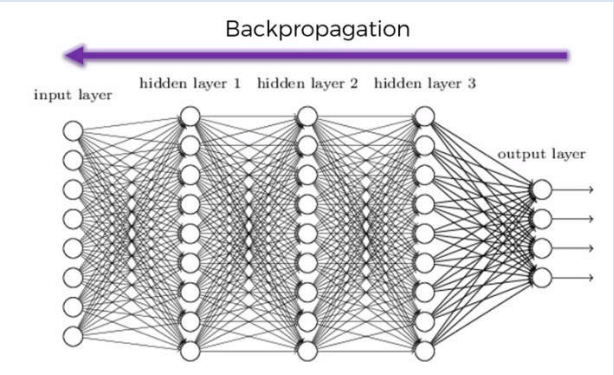
8.3.7 Back-propagation

❒ ***Forward propagation:*** There's a process called Forward propagation where information is entered into the input layer and then it's *propagated forward* to get our *output* value $\hat{y}$.

☞ Then we compare those to the *actual* value y in our ***Training set***.

☞ And then we *calculate* the *errors* then the *errors* are ***Back Propagated*** through the *network* in the *opposite direction* and that ***allows*** us to ***train the network*** by ***adjusting*** the ***weights***.





❒ ***Back propagation adjusts all the Weights simultaneously:*** **Back propagation** is an advanced algorithm driven by very **interesting** and **sophisticated mathematics** which allows us to *adjust* the *weights*. All the **weights** at the **same time** are **adjusted simultaneously**.

☞ The huge advantage of **Back propagation** and it a key thing to remember is that during the process of back propagation, you are able to adjust all the *weights* at the *same time*, because that's the way the algorithm is structured. This is the key fundamental underlying principle of back propagation.

☞ So you can track which part of the *error* each of your weights in the *neural network* is responsible for.

❒ ***Adjust Each Of The Weights Independently/Individually:*** If we were doing this *manually* or if we're coming up a *very different type of algorithm* than even if we calculated the ***error*** and then we were trying to understand **what *effect* each of the *weights* has on the *error*** then we'd have *adjust* each of the *weights independently/individually*.

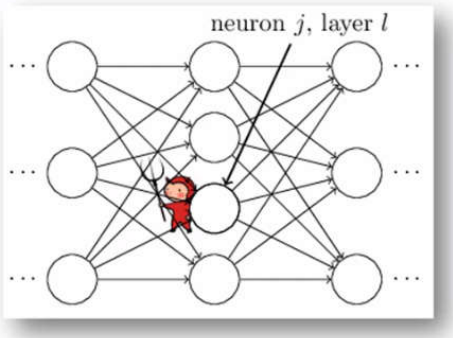
☞ And if you'd like to learn more about that and how exactly the mathematics works in the background then a good article which we've already mentioned is the ***Neural Networks And Deep Learning*** is actually a book by ***`Michael Nielsen`***. You'll find the mathematics written out and it will help you understand how exactly this is possible.

Additional Reading:

Neural Networks and Deep Learning

Michael Nielsen (2015)

Link:



http://neuralnetworksanddeeplearning.com/chap2.html

8.3.8 Steps on Training Of A Neural Network

Now we're going to just wrap everything up with a ***Step By Step Walkthrough*** of what happens in the ***Training Of A Neural Network***.

➲ ***STEP 1:*** We **randomly initialize** the ***`weights`*** to ***`small numbers close`*** to ***`0`*** but **not zero**. i.e. They are initialized with random values near zero. And from there through the process of **Back Propagation** these *weights* are *adjusted* until the *error/ cost function* is *minimized*.

- ***STEP 2:*** Input the *first observation* of your *dataset* in the *input layer*, ***each feature*** in ***one input node***. That is the first row into input layer and ***each feature*** in ***one input node.***

- ***STEP 3:*** ***Forward-Propagation:*** from *left* to *right*, the neurons are activated in a way that the *impact of each neuron's activation* is *limited* by the ***weights*** *(the* ***weights*** *basically* ***determine*** *how important each* ***neurons activation*** *is).* ***Propagate*** the ***activations*** until getting the ***predicted result*** $\hat{y}$.

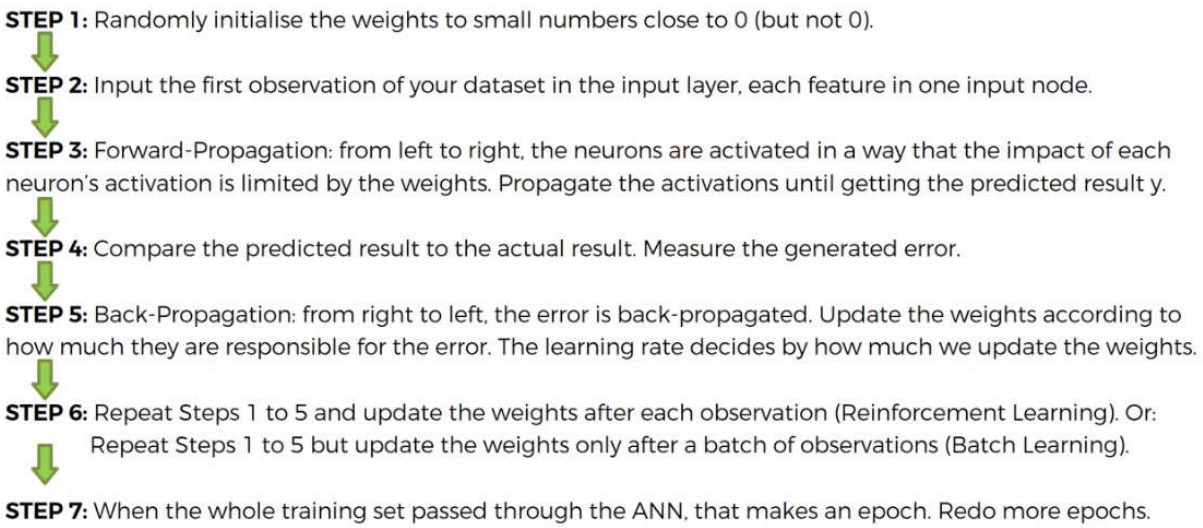
- ***STEP 4:*** Compare the *predicted* result $\hat{y}$ to the *actual* result y. Measure the ***generated ERROR***.

- ***STEP 5:*** ***Back-Propagation:*** from *right* to *left*, the ***Error*** is ***Back-Propagated***. *Update* the ***weights*** according to *how much they are* ***responsible*** for the *error*. The ***learning rate*** decides by ***how much*** we **update** the **weights**. The *learning rate*, as a ***parameter*** you can control it in your neural network.

- ***STEP 6:***
  - ***SGD:*** Repeat ***Steps 1 to 5*** and *update* the ***weights*** after each *observation* (this is called *reinforcement learning* and in our case that was *stochastic gradient descent*).
  - ***BGD/Mini-BGD:*** Repeat ***Steps 1 to 5*** but update the ***weights*** only after a *batch of observations* (***Batch Learning***).

- ***STEP 7:*** When the ***whole training set*** passed through the ***ANN***. that makes an ***Epoch***. Redo **More Epochs**. Just keep doing that to allow your ***NN*** to `train` better and better and `constantly adjust itself` until you ***minimize*** the ***cost function***.

Those are the steps you need to take to ***build*** your ***artificial neural networks (ANN)*** and ***train*** it.



Chapter 8 : Part 4

Deep Learning

ANN: Artificial Neural Network

`Python Implementation`

📣 Here in the first branch of deepening (ANN), we're going to study this classification problem using ANN.

📣 In the next chapter we will study ***CNN (convolutional neural networks)*** another branch of deep learning which are which work very well for ***computer vision tasks*** unlike ***artificial neural networks (ANN)***.

8.4.1 Business Problem Description

Deep learning is one of the ***most fascinating*** and also the ***most powerful*** branch of ***Machine Learning***. We will build a very powerful ***ML models*** based on **Deep Neural Networks**.

- ☞ Today deep learning is used for many of demanding and highly ***compute intensive*** tasks for example ***computer vision*** (like recognizing faces and pictures or videos) & ***medicine*** (recognizing tumors in some brain images).
- ☞ In fact deep learning can be used for making ***predictions (regression)*** or ***classifications*** for **Business Problems**.
- ☞ And it is also used for ***recommended systems*** with the use of ***deep Boltzmann machines***.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProduct	HasCrCard	IsActiveMemb	EstimatedSalar	Exited
2	1	15634602	Hargrave	619	France	Female	42	2	0	1	1	1	101348.88	1
3	2	15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1	112542.58	0
4	3	15619304	Onio	502	France	Female	42	8	159660.8	3	1	0	113931.57	1
5	4	15701354	Boni	699	France	Female	39	1	0	2	0	0	93826.63	0
6	5	15737888	Mitchell	850	Spain	Female	43	2	125510.82	1	1	1	79084.1	0
7	6	15574012	Chu	645	Spain	Male	44	8	113755.78	2	1	0	149756.71	1
8	7	15592531	Bartlett	822	France	Male	50	7	0	2	1	1	10062.8	0
9	8	15656148	Obinna	376	Germany	Female	29	4	115046.74	4	1	0	119346.88	1
10	9	15792365	He	501	France	Male	44	4	142051.07	2	0	1	74940.5	0
11	10	15592389	H?	684	France	Male	27	2	134603.88	1	1	1	71725.73	0
12	11	15767821	Bearce	528	France	Male	31	6	102016.72	2	0	0	80181.12	0
13	12	15737173	Andrews	497	Spain	Male	24	3	0	2	1	0	76390.01	0
14	13	15632264	Kay	476	France	Female	34	10	0	2	1	0	26260.98	0
15	14	15691483	Chin	549	France	Female	25	5	0	2	0	0	190857.79	0
16	15	15600882	Scott	635	Spain	Female	35	7	0	2	1	1	65951.65	0

❒ ***Data-set:*** We have a data-set, containing columns: ***RowNumber***, ***CustomerId***, ***Surname***, ***Geography***(country), and some other info.. Those data are the customers of a Bank. The dependent variable is, "***Exited***", it indicates if a customer ***leaves*** a bank or not. The data-set contains data of 10000 customers.

- ☞ The bank has been seeing unusually ***high churn rates***. **Churn** is when people **leave** the **company**. They want to understand what the problem is. We are here to look into this data-set for them and give them some insights.
- ☞ This fictional bank operates in Europe in three countries ***France***, ***Spain*** and ***Germany***. They have lots and lots of customers so they took this sample of 10000 customers and they measured six months ago everything they knew about them,
  - ▶ Their ***CustomerId***, their ***Surname***, ***CreditScore***, ***Geography***, ***Gender*** , ***Age*** their ***Tenure*** (how long they've been with the bank),
  - ▶ The ***Balance*** of the customers at that point in time, the ***NumberOfProducts*** they had at that point in time (i.e. savings account, credit card or loan), ***HasCrCard*** (did the customer have a credit card or not),
  - ▶ ***IsActiveMember*** (active member or not active member can be measured differently by different organizations. It could be whether or not the customer ***logged*** into their ***online banking*** in the past month whether they did a ***transaction*** in the past two months or some other measurement like that.),
  - ▶ ***EstimatedSalary*** (the bank doesn't know the salary of the customers but based on the other things they know they could estimate a salary for that customer).
- ☞ So six months ago they measured all of these things and they waited six months and observed which customer is living the bank and which stayed.
  - ▶ In the data-set, the column ***Exited*** tells you whether or not the person left the bank within those six months. Here ***1 means*** that the person is ***left*** the bank and ***0 means*** the person ***stayed*** in the bank.

☝ ***Churn rate:*** Churn rate, in its broadest sense, is a measure of the number of individuals or items moving out of a collective group over a specific period.

❒ ***Goal:*** Our goal is to create a ***Geodemographic segmentation model*** to tell the bank which of their customers are at ***highest risk of leaving***. We are going to solve this business problem using ***Artificial Neural Networks***.

❒ ***Other applications:*** For a lot of customer centric organization this is going to be valuable, it doesn't have to be for churn rates.

- ⇨ ***Geodemographics segmentation models*** can be applied to millions of scenarios, for instance even in a bank the same scenario could work should the person ***get a loan or not***, should the person be approved for ***credit-card or not***.
- ⇨ A person is reliable or not: You'd have a binary outcome. So based on prior experience you would know whether or not ***a person is reliable*** and you'd build a model and say which people are more likely to be reliable and which people are more likely to default.
- ⇨ And that could govern the Bank's decision on whether **"to give"** or **"not to give"** loans.
- ⇨ You could apply a demographic segmentation and it doesn't even have to be demographic!! For example, when you have a binary outcome and you have lots of independent variables you can apply this Technique.

8.4.2 Data Preprocessing

We already know that the problem that we're about to deal with is a classification problem.
We have these ***independent variables***: ***CustomerId***, ***Surname***, ***CreditScore***, ***Geography***, ***Gender*** , ***Age Tenure***. And based on these ***independent variables*** we are going to ***predict*** which customers are ***leaving the bank***.

❒ ***Installing Libraries:*** We need to install 3 libraries: ***Theano***, ***TensorFlow*** and ***Keras***.

- ☞ ***Theano:*** Theano is a Python library that allows you to define, optimize, and efficiently evaluate mathematical expressions involving multi-dimensional arrays. Primarily developed by the ***Montreal Institute for Learning Algorithms (MILA) at the Université de Montréal.***.
  - ⇨ What is also great about this library is that it can run on CPU and also on GPU.
- ☞ ***TensorFlow:*** TensorFlow is another open source Numerical-computation library that runs very fast inn you CPU or GPU. This library was originally developed by the Google brain team at Google and it's now under the Apache 2.0 license.
- ☞ These two libraries ***Theano*** and ***TensorFlow*** are used mostly for research and development purposes. You have to use these two libraries to build a ***Deep Neural Network*** from ***scratch***. That is with many lines of code. If you want to do some ***Research*** in order to ***Improve*** the ***Deep Neural Networks*** invent and **Develop** a new kind of **Neural Network** or any other kind of **Deep Learning Models**.
  - ⇨ ***Keras:*** Now we are not going to directly use ***Theano*** and ***TensorFlow***, we're going to use another library called ***Keras*** that's in some way ***wraps*** the ***two libraries*** (Theano and TensorFlow).
  - ⇨ ***Keras*** helps to build ***Deep Neural Networks*** in a very few lines of code. We will use ***Keras*** to build ***Deep Learning Models*** very efficiently as we used ***scikit-learn*** to build ***ML models***.

❒ ***TensorFlow download & setup:*** Tensorflow runs at 64bit OS. Just need python 3.7 or python 3.8 version and then just use this command: "***pip install tensorflow***". Use command prompt to install following:

☞ Anaconda Environment for TF
☞ Create conda environment for TF. Choose the Python version. Execute on cmd.
☞ After that install TF using pip.
☞ Choose and install for GPU or CPU.
☞ Finish the installation using command
 	
Following might be useful
 	
`pip install scikit-learn`
`pip install numpy`
`pip install matplotlib`
`pip install statsmodels`
	Conda Environment: You might need to re-install ***spyder*** in the created ***environment***. Also install ***scikit-learn***, ***matplotlib*** etc.
	
	`conda create --name tensorflow python=3.8`
	`conda activate tensorflow`
	
	Above will install ***tensorflow***
	
	conda install jupyter
	`conda install scipy`
	`## pip install tensorflow-gpu`
	`pip install tensorflow`
	
	`conda update -n base -c defaults conda`
	Install Keras and Theano:
	`pip install keras`
	`pip install Theano`

☝ ***Another method:*** Without creating virtual environment in Anaconda/Conda:
1. Open anaconda Powershell prompt
2. Run following codes to install TensorFlow in base(root):
`conda activate base`
`pip install --user tensorflow`
	--user makes pip install packages in your home directory instead, which doesn't require any special privileges.
	
	☠ To **remove conda virtual environment**, use:
	conda env remove --name env_name

❒ ***Data Preprocessing:*** We're going to build this model in two parts:

[1]. Data processing

[2]. Creating the ANN model.

☞ Since our business problem is ***classification problem***. The *independent variables* are some **information** about customers in a bank. We are trying to predict a ***binary outcome*** for the *dependent variable* which is:

1: if the customer ***leaves*** the bank

0: if the customer ***stays*** in the bank.

☞ So we will use our ***classification template*** (or previously done classification model). Also we **don't need** the ***visualization part***, because current problem has 10 *independent variables*.

☞ We will use the ***confusion matrix*** to evaluate the performance of our ***ANN model***.

❒ ***Choosing the independent variables:*** We don't need the first 3 – variables, ***RowNumber***, ***CustomerId***, ***Surname***. Because these variables has no impact on the "Exited" (dependent variable). We take the rest of the variables excluding the last "Exited". "Exited" will be our dependent variable.

```
X = dataSet.iloc[:, 3:13].values
```

☞ In `[:, 3:13]`, `3:13` means "all columns from index 3, excluding 13 indexed column". If the index of last column is unknown, then we can use "`3:-1`" which means "all columns from index 3, excluding the last column"

❒ ***How to find out the most impact independent variables:*** So with **our own logic** we are **able** to say which **independent variables** might have impact on the **dependent variable**. But we don't know which ***independent variable*** has the ***most impact*** on the ***dependent variable***. And that's what our ANN will spot and find the correlations by giving the **bigger weights** in the *neural network* to those **independent variables** that have the most impact.

❒ ***Encode Categorical Data:*** `[1, 2]` means that we want to convert 2nd and 3rd columns. We need to encode before splitting the data.

```
ct = ColumnTransformer(transformers = [("encoding", OneHotEncoder(), [1, 2])], remainder = 'passthrough')
```

☞ However we are not using the above code to encode columns.

⇨ We will use ***Label-encoder*** for "**Gender**", variable because it has only **2-categories**.

⇨ We will use ***One-hot-encoder*** for "**Geography**" (country variable), because it has **more than two** categories. So it will create 3 dummy variables.

☞ ***Encode "Gender" column:*** "**.values**" converts data-set into **array**. If we use `X = dataSet.iloc[:, 3:13]` instead of `X = dataSet.iloc[:, 3:13].values`
Then X won't be an Array it will be a Dataset. And we cannot use `X[:, 2]` for encoding. To encode a column in a Dataset, we need to use "key" of the column. We have to use ***X["Gender"]*** instead of ***X[:, 2]***.

X["Gender"] = label_encode.fit_transform(X["Gender"]) *# in case of data-set instead of Array*

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

#Encode "Gender" using LabelEncoder."Gender" is in "3rd-column", hence X[:, 2]
label_encode = LabelEncoder()
# Following is applicable to numpy array, if we used "X = dataSet.iloc[:, 3:13]"
# X = np.array(X) # it is needed if X is not an Arry. i.e. ".values" not applied
X[:, 2] = label_encode.fit_transform(X[:, 2])
print(X)

# For a data-set we can still encode it using Columns "key"
# X["Gender"] = label_encode.fit_transform(X["Gender"])
```

☞ ***Encode "Geography" column:*** Using ***OneHotEncoder*** will create 3-dummy variables.

⇨ ***Avoiding dummy variable trap:*** To avoid the dummy variable trap, we exclude our 1st column (index 0). Using

X = X[:, 1:]

```
ct = ColumnTransformer(transformers = [("encoding", OneHotEncoder(), [1])], remainder = 'passthrough')
X = ct.fit_transform(X)
X = np.array(X) # convert this output to NumPy array
X = X[:, 1:] # Excluding 0-index column to avoid dummy-variable trap
```

❒ ***Data-Spit:*** We use ***test_size = 0.2*** because of 10000 observations, so 8000 for training and 2000 for testing.

```
# Data Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2, random_state = 0)
```

❒ ***Feature-Scaling:*** In ***ANN*** and in general ***deep learning*** we ***need feature scaling***. It is thoroughly ***compulsory*** and that is because there is going to be a ***lot of computation*** and it is all ***parallel computations***. So we need to ***apply feature-scaling to ease of these calculations***.

☞ And besides we don't want to have one **independent** variable **dominating** another one.

```
from sklearn.preprocessing import StandardScaler
#  y dependent variable, need not to be scaled: categorical variable, 0 and 1
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)
```

Data-Preprocessing All at once

```
# Artificial Neural Network
# Install : Tensorflow, Keras and Theano libraries.

# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Churn_Modelling.csv")
# X = dataSet.iloc[:, 3:-1].values # this can be used too
X = dataSet.iloc[:, 3:13].values # all columns from index 3, excluding 13 indexed column
y = dataSet.iloc[:, 13].values # the last column


# ------------- Encode Categorical Data  -----------
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

#Encode "Gender" using LabelEncoder."Gender" is in "3rd-column", hence X[:, 2]
label_encode = LabelEncoder()
# Following is applicable to numpy array, if we used "X = dataSet.iloc[:, 3:13]"
# X = np.array(X) # it is needed if X is not an Arry. i.e. ".values" not applied
X[:, 2] = label_encode.fit_transform(X[:, 2])
print(X)

# For a data-set we can still encode it using Columns "key"
# X["Gender"] = label_encode.fit_transform(X["Gender"])


#Encode 'Gegraohy' using OneHotEncoder. "Gegraohy" is in "2nd-column", hence [1]
ct = ColumnTransformer(transformers = [("encoding", OneHotEncoder(), [1])], remainder = 'passthrough')
# remainder = 'passthrough' for remaining columns to be unchanged
X = ct.fit_transform(X)
X = np.array(X) # convert this output to NumPy array
print(X)
X = X[:, 1:] # Excluding 0-index column to avoid dummy-variable trap



# ------------------ Data Split -----------------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2, random_state = 0)

        # Feature-Scaling after Data Split
```

```
# ------------------ Feature-Scaling ---------------------
from sklearn.preprocessing import StandardScaler
#  y dependent variable, need not to be scaled: categorical variable, 0 and 1
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)
```

dataSet - DataFrame

Index	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	15634602	Hargrave	619	France	Female	42	2	0	1	1	1	101349	1
1	2	15647311	Hill	608	Spain	Female	41	1	83807.9	1	0	1	112543	0
2	3	15619304	Onio	502	France	Female	42	8	159661	3	1	0	113932	1
3	4	15701354	Boni	699	France	Female	39	1	0	2	0	0	93826.6	0
4	5	15737888	Mitchell	850	Spain	Female	43	2	125511	1	1	1	79084.1	0
5	6	15574012	Chu	645	Spain	Male	44	8	113756	2	1	0	149757	1

Dataset

Feature matrix | ***Output-vector***

X - NumPy object array (read only)

	0	1	2	3	4	5	6	7	8	9	10
0	0.0	0.0	619	0	42	2	0.0	1	1	1	101348.88
1	0.0	1.0	608	0	41	1	83807.86	1	0	1	112542.58
2	0.0	0.0	502	0	42	8	159660.8	3	1	0	113931.57
3	0.0	0.0	699	0	39	1	0.0	2	0	0	93826.63
4	0.0	1.0	850	0	43	2	125510.82	1	1	1	79084.1
5	0.0	1.0	645	1	44	8	113755.78	2	1	0	149756.71

y - NumPy object array

	0
0	1
1	0
2	1
3	0
4	0
5	1
6	0
7	1

8.4.3 Creating the ANN model

Now everything is ready and we can eventually get into make the artificial neural network.

❒ ***Import Keras and packages:*** First step is to ***import*** the ***Keras*** libraries and the ***required packages (***some modules of the Keres library***)*** to build the Neural Network.

```
# importing "keras" libraries and packages
# from tensorflow import keras
import keras # using TensorFlow backend
from keras.models import Sequential
from keras.layers import Dense
```

☞ Actually ***keras*** is using ***TensorFlow*** as **back-end**. That is the ***keras*** library will build the *Deep Neural Network* based on ***TensorFlow***. You can also use ***Theano*** as **backend**. But TF will be fast.

☞ Also we need to import two modules here.

⇨ The ***Sequential*** module that is required to ***initialize*** our *neural network* and

⇨ the ***Dense*** module is required ***to build the layers*** of our ANN.

❒ ***Initializing the ANN:*** Initializing the ANN means ***"defining it as a sequence of layers"***. There are actually ***two ways*** of **initializing** a *Deep Learning Model*. It's either by defining the ***sequence of layers*** or defining a ***graph***.

☞ Since we'll make **ANN** with **successive layers** (as you saw in pervious sections of this chapter), we'll initialize our ***Deep Learning Model*** by defining it as a **Sequence Of Layers**.

☞ We just need to create an Object of the ***Sequential*** class. This ***object*** that we're going to create is *nothing else than the* ***model itself***. i.e. the **Neural Network** that will have a ***role of a Classifier*** (this NN-model is going to be a classifier) here because our problem is a ***Classification Problem*** where we have to ***predict a class***.

```
ann_classifier = Sequential()
```

So this classifier object is nothing else than the future ANN that we're going to build.

- ⇨ We *don't need to use any* ***arguments*** because we will **define the *layers* step by step** afterwards.
- ⇨ We will start with the ***`input layer`*** and the **first *`Hidden-layer`*** and then we'll add some more ***`hidden-layers`*** then finally we'll add the ***`output layer`***.

So that's how we initialize our *Artificial Neural Network Classifier*.

❒ ***INPUT LAYER*** and first ***HIDDEN LAYER***: We are going to add the first layer of our ANN-model which is the **input-layer** and the first **hidden-layer**. Recall the ANN has 7-steps to follow:

SGD

STEP 1: Randomly initialise the weights to small numbers close to 0 (but not 0).

STEP 2: Input the first observation of your dataset in the input layer, each feature in one imput node.

STEP 3: Forward-Propagation: from left to right, the neurons are activated in a way that the impact of each neuron's activation is limited by the weights. Propagate the activations until getting the predicted result y.

STEP 4: Compare the predicted result to the actual result. Measure the generated error.

STEP 5: Back-Propagation: from right to left, the error is back-propagated. Update the weights according to how much they are responsible for the error. The learning rate decides by how much we update the weights.

STEP 6: Repeat Steps 1 to 5 and update the weights after each observation (Reinforcement Learning). Or: Repeat Steps 1 to 5 but update the weights only after a batch of observations (Batch Learning).

STEP 7: When the whole training set passed through the ANN, that makes an epoch. Redo more epochs.

[1]. Step-1 ***"randomly initialize the weights of each of the nodes to small numbers close to 0" `Dense`*** module is going to take care of this first step.

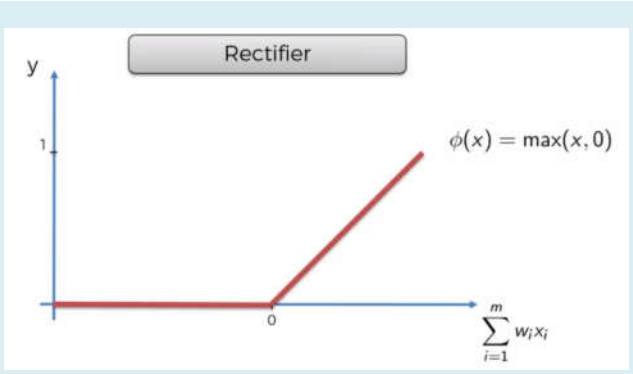
[2]. Step-2: Our ***first observation*** goes into the ***NN*** and ***each feature*** as an ***input node***. We already know the **number of nodes** of the **input layer** which is the **number of independent variables** we have in our ***Feature-Matrix***.

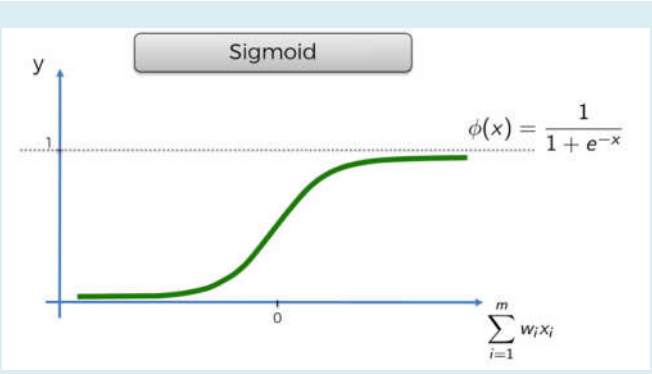
☞ After data preprocessing, we had ***11*** **independent variables**. Hence in our **input-layer** will have ***11*** **input nodes**.

[3]. Step-3 is forward propagation. So from ***left to right*** the neurons are ***activated*** by the ***activation function*** in such a way that the "***higher*** the ***value*** of the ***activation function*** is for the ***neuron*** the ***more impact*** this ***neuron*** is going to have" in the network.

☞ ***Choosing an activation function:*** We'll choose the ***Rectifier-Activation-Function*** for the ***Hidden-Layers*** and the ***Sigmoid-Activation-Function*** for the ***Output-Layer***.

☞ The best one based on experiments research is the **rectifier-activation-function** for **Hidden-layers**





☞ We also use **sigmoid-rectifier-function** for **Output Layer**. Since using the ***SIGMOID FUNCTION for the Output Layer*** will allow us to get the probabilities of the different.

i.e. for each ***observations*** of the ***test set*** we'll get the ***probability*** that the *customer leaves the bank* and the probability that the *customer stays in the bank*.

☞ Since we are trying to build a ***SEGMENTATION MODEL*** and by getting the probability we will be able to see which *customers* have the ***highest probabilities*** to *leave the bank*. So we'll be able to make a ***ranking of the customers*** by their probability to leave the bank.

And then you can ***Segment*** your ***Customers*** according to their ***probability to leave the bank***. So that you can decide what to do in terms of *business constraints* and *business goals*

[4]. Step-4 the algorithm ***compare*** the **predicted-result** and **actual-result** and generates ***Error***.

[5]. Step-5 the Error will be ***back-propagated*** and algorithm ***updates*** the ***weights*** of Synapses. ***Weights*** are ***updated*** according to how much they are responsible for this ***generated error***.

☞ There are several ways of updating these weights. It is defined by the ***learning rate parameter*** which decides by *how much the weights are updated*.

[6]. In step-6 and step-7 above steps are repeated and minimize the cost-function.

☯ ***Adding input and first-hidden layer:*** We take the object ***ann_classifier*** and use the ***add()*** method. ***add()*** method is used to ***add the different layers*** in our NN.

📌 ***Parameters of add()***: There is only one arguments and this argument is the ***layer*** that we want to add to our CNN. We are going use ***Dense*** function to define this ***layer*** argument.

👽 ***Parameters of Dense():*** We're going to add two layers the *input-layer* and the *first hidden-layer*. There are a ***lot of arguments*** for ***Dense*** function.

➢ These arguments are going to be all the parameters, such as: how the ***weights are updated***, the type of ***activation*** function, number of ***nodes for layers***, number of ***input nodes*** etc. Those things happens in this ***Dense()*** function.

➢ **output_dim:** That is simply the number of nodes you want to add in this hidden layer.

✓ Here ***add()*** function doesn't know it is adding *input-layer* & the *first hidden-layer*. It just adding a ***hidden layer*** so for these ***hidden layers*** we have to specify the ***number of inputs*** in the ***previous layers*** (which is in the input-layer at very first).

✓ ***Choosing the number of nodes:*** It is the ***Art***. There is no rule of thumb on what would be the optimal number of nodes in these hidden layers. However, we can give some rules like for example :

- If your data is linearly separable, you don't even need a hidden layer and in fact you don't even need a neural network.
- Choose the **number of nodes** in the **hidden-layer** as the ***average*** of the number of nodes in the ***input layer*** and the number of nodes in the ***output layer***. It is not a rule but as a tip, if you ***don't want to be an Artist***. It is ***not based on theory*** but rather based on ***experiments***.

$$no.\,of\ nodes\ = \frac{input\ layer\ nodes\ +\ output\ layer\ nodes}{2}$$

- ***PARAMETER TUNING:*** If you want to be an artist (i.e. pro), then you have to experimenting with a technique called **PARAMETER TUNING**. ***Parameter Tuning*** is about using some techniques like ***K-fold cross-validation*** (we will study it In "Model selection and Ensemble model" later).
  - ***K-fold cross-validation*** technique consists of creating a separate set in your data set besides the ***training-set*** and the ***test-set*** that is called a ***cross-validation-set***.
  - Basically in this **cross-validation-set**, you ***experiment different parameters*** of your *model*. Such as: ***number of hidden layers*** and the ***number of nodes*** in the ***hidden layers***. And then you test the ***performance*** of your ***different models*** with the ***different parameters***.

✓ We won't do this PARAMETER TUNING here, we will study ***K-fold cross-validation*** later. which will help us choose the ***optimal parameters*** of our model. But for now we're going to take the average of the *number of* ***nodes*** *in the* ***input layer*** and a *number of* ***nodes*** *in the* ***output layer***.

- In our case, number of nodes in the input layer is ***11***, because the *number of* ***nodes*** *in the* ***input layer*** is the *number of* ***independent variables***.
- And the *number of* ***nodes*** *in the* ***output layer*** is ***1*** because we have a ***binary outcome one*** or ***zero***. So the ***average*** is ***6***, i.e. ***six nodes*** in the *hidden layer*.

```
output_dim = 6
```

NOTE: In the new documentation of ***Dense*** function ***output_dim*** is replaced by ***units***, and the ***input_dim*** is replaced by ***input_shape***. However in the ***input_shape*** argument you have to specify a ***tuple***.

➢ **init:** It corresponds to the step-1 of SGD algorithm: "randomly initialized the weights as small numbers close to zero". We can randomly initialize them with a uniform-function.

✓ For example: "***glorot_uniform***" to initialize the weights. Or,

✓ More simple "***uniform***" it's a simple-uniform-function to initialize the weights according to a ***uniform distribution,*** it will also make sure that the weights are small numbers close to zero.

```
init = "uniform"
```

NOTE: In the new documentation of ***Dense*** function ***init*** is replaced by ***kernel_initializer***.

➢ **activation:** Here we specify the **type of activation-funstion**. We use the ***Rectifier-Activation-Function*** for the ***Hidden-Layers***. The parameter is corresponds to Rectifier-Activation-Function the is "***relu***".

```
activation = "relu"
```

➢ **input_dim:** It is a compulsory agument. It specify the *number of nodes in the* ***input-layer***. (i.e. *number of* ***independent variables***.). Without ***input_dim*** our ANN is simply initialized, *we haven't created any layer yet* and it doesn't know which nodes this hidden layer is expecting as inputs. Since we have 11 ***independent variables***.

```
input_dim = 11
```

Since this first layer will already be created we ***won't need to specify*** this ***input_dim*** for the ***next hidden layers*** because the next hidden layers will know what to expect.

```
# ann_classifier.add(Dense(output_dim = 6, init = "uniform", activation = "relu", input_dim = 11))
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim
= 11))
```

☯ ***Adding second hidden-layer:*** The next step is to add a second hidden layer. To be honest, it is not necessarily useful for our data set but we're going to add it anyway for two reasons.

- ❖ First of all because the deep learning is defined as an artificial neural network with many hidden layers.
- ❖ And the second reason is simply that you need to add more hidden layers in your neural networks.
- ❖ We will use the same method ***add()*** and ***Dense()***, without ***input_dim*** (because it is specified in previous layer). So we use following code:

```
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))
```

- ❖ The number of nodes will be ***(11+1)/2 = 6*** i.e. `(input layer nodes+ output layer nodes)/2`.
- ❖ Also we ***initialize*** the ***weights*** using *uniform-initializer* to randomly initialize the ***weights*** to given *small numbers* close to ***zero***.
- ❖ Since we are creating second hidden-layer, we use ***Rectifier-Activation-Function*** i.e. "***relu***" again (we'll use *sigmoid function* for the *output layer*).

☯ ***Output-layer:*** Since we already have two hidden layers in ANN (one with input-layer & another is 2nd hidden layer). Now we add the output-layer. The code is similar to ***2nd hidden layer*** but with ***sigmoid activation function***. From the *logistic regression* of chapter 3, we know that, the *sigmoid function* is the heart of this *probabilistic approach*.

- ❖ Since we are making a ***Geo-demographic segmentation model*** we want to have *probabilities for the outcome* because we want to have the **probability** of each **customer leaves** the **Bank**.

```
ann_classifier.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))
```

- ❖ We need to change the output parameter "***units***" because in our *output-layer* we want *only one node* because our dependent variable is a ***categorical*** variable with a ***binary*** outcome ***0 (stay)*** and ***1 (out)***.
- ❖ We will keep the ***uniform-initializer***.

☝ If you are dealing with a *dependent variable* that has *more than two categories,* for example: **three categories** then you will need to ***change two things*** here.

☞ First is the output parameter "units" that will be set as the number of classes(because it will be based on the *one-VSL method* while the dependent variable is *one-hot-encoded*.). So it will be "***units = 3***".

☞ Second thing that you need to *change* is the *activation function* that in this situation would be ***Softmax-activation-function***, it is actually the *Sigmoid function* but applied to a *dependent variable* that has more than *two categories*.

☞ The ***Softmax*** function, also known as ***Softargmax*** or ***Normalized Exponential Function***, is a **generalization** of the ***logistic function*** to ***multiple dimensions***. It is used in *multinomial logistic regression* and is often used as the last *activation function* of a *neural network* to *normalize the output* of a network to a **probability distribution** over ***predicted output classes***, based on ***Luce's choice axiom***.

8.4.4 Compile and Train the ANN model

❒ ***Compile the ANN model:*** We are done adding the layers of our artificial neural network. Now what we're going to do is to ***compile*** the ANN, applying ***SGD*** on the whole ANN.

```
ann_classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

We apply ***compile()*** method on our ***ann_classifier*** object. This ***compile()*** method contains several parameters.

[1]. ***optimizer:*** "***optimizer***" is simply the ***Algorithm*** you want to use to find the ***OPTIMAL set of WEIGHTS*** in the neural networks. This algorithm is the SGD algorithm.

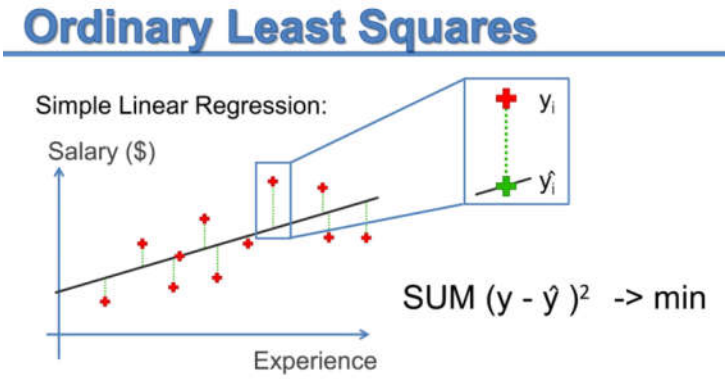
☞ There are several types of SGD algorithms. A very efficient one is called "***adam***" we're going to use it here.

```
optimizer = "adam"
```

[2]. ***loss:*** Loss-function. And this corresponds to the ***Loss function*** within the ***SGD algorithm*** (i.e. within the "***adam***" algorithm).

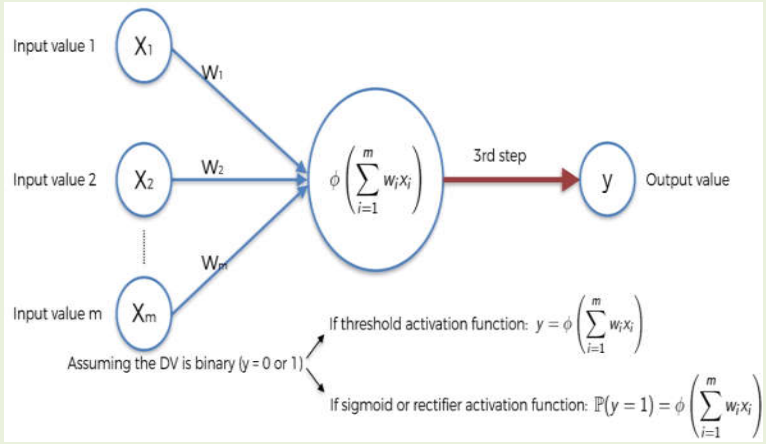
☞ ***Loss function:*** If you go deeper into the ***mathematical details*** of ***SGD***, you will see that it is based on a ***Loss-function***, that you need to ***optimize to find the optimal weights***.

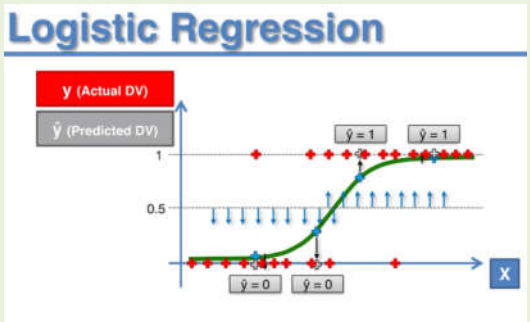
☞ For examples, we saw the **loss-function** when we studied ***Linear Regression***. The ***Loss-Function*** was the **sum of the squared errors** (sum of the square differences between the real value and the predicted value). It is used to optimize the parameters of the regression model.



☞ Now the idea is exactly same in the case of ***SGD*** here. We have some **parameters** which are the ***weights*** in the ***neural network*** and so we need to specify a ***Loss-function*** that will ***optimize*** through ***SGD*** to find ***optimal weights***.

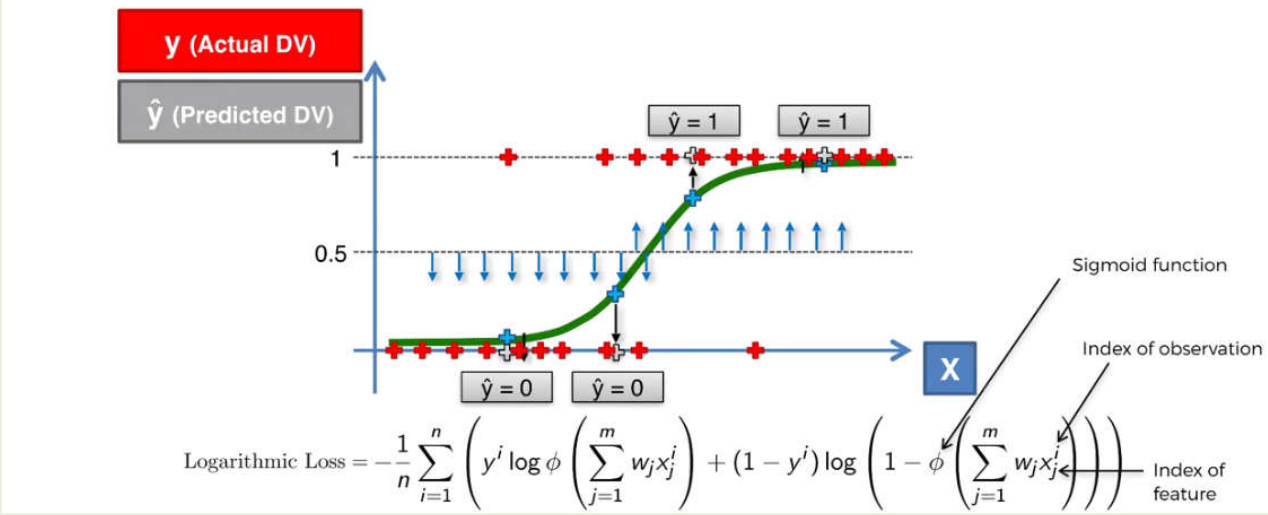
☝ ***LOGARITHMIC-LOSS: Loss-function for the neural networks:*** This ***Loss-function*** is going to be kind of the same as for ***Logistic Regression*** because when you take a simple neural network (a PERCEPTION) and if you use a ***Sigmoid Activation Function*** for this ***PERCEPTION*** then you obtain a ***Logistic Regression Model***





✓ If we go for the mathematical details of ***SGD*** for ***Logistic Regression***, you will find out that the ***Loss-Function*** is ***not the*** **sum of the squared errors** like for ***Linear Regression***. It's going to be a ***Logarithmic-Function*** that is called the ***Logarithmic-Loss***.

Logarithmic Loss



✓ Since the ***activation function*** for ***output layer*** is nothing else than the ***sigmoid function***, the ***Loss-function*** that we're going to use compile our ANN and on which ***SGD-adam algorithm*** is based on: is the ***Logarithmic-Loss***.

☞ If your dependent variable has a ***binary outcome*** (like here) then ***Logarithmic-Loss*** is called ***'binary_crossentropy'***,

☞ If your dependent variable has more than two outcomes, like **three categories** then this ***logarithmic function*** is called ***'categorical_crossentropy'***.

```
loss = "binary_crossentropy"
```

[3]. ***metrics:*** It is just a ***criterion*** that you choose to ***evaluate your model***. Typically we use the ***'accuracy'*** criterion. Basically what happens is that, when the weights are updated after **each observation** or after **each batch of observations**, the ***algorithm*** uses this ***'accuracy'*** criterion to *improve* the *models performance*.

☞ When we fit the ANN into our training-set, the ***accuracy*** is going to **increase little by little** until **reaching a top accuracy** and that will happen because we choose here the ***"accuracy" metric***.

```
metrics = ['accuracy']
```

☞ Since this ***metrics*** argument is expecting a *list of metrics*. But here we only use **one metric** which is the ***accuracy metric*** we need to add this *accuracy metric* as a ***list***. This list will only contain *one element* which is the *accuracy metric*.

❒ ***Train the ANN model:*** We are going to use the ***fit()*** method into the ***training-set***.

☞ We will apply ***fit()*** to our object ***ann_classifier***.

☞ We pass the parameters ***X_train*** and ***y_train***.

☞ We will add two **additional arguments** for the ***"batch_size"*** (separate dataset into several batches) and ***"number of epoch"***. This is where you're ***Deep Learning Artist Soul*** comes into play, because there is no rule of thumb. We need to experiment to find some *optimal choice* for this *batch size* here and this *number of epochs*.

1. ***batch_size:*** We can choose to *update the weights* either after *each observation passing* through the ***ANN*** or after a *batch of observations*.
   - ***batch_size*** is the number of observations after which you want to ***update*** the ***weights***. This could be any number depends on how many parts you want *to divide* the *train-data-set*.
2. ***epochs:*** An epoch is basically a round when the whole training set passed through the ANN. And in reality *training ANN* consists of *applying all steps of the ANN* over *many epochs*.

☞ Right now we're not going to experiment. We will go with some fixed choice of ***batch_size*** and number of ***epochs***:

```
batch_size = 10,  epochs = 100
```

So that, we can see the *algorithm in action* and so that we can see the *accuracy improving over the rounds/epochs.*

❒ We can execute the model now. And eventually our model will be ready. And if we check the result we will converge to the accuracy of 86%.

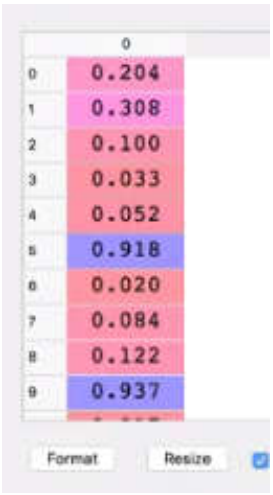
```
Epoch 97/100
800/800 [==============================] - 1s 904us/step - loss: 0.3412 - accuracy: 0.8596
Epoch 98/100
800/800 [==============================] - 1s 791us/step - loss: 0.3409 - accuracy: 0.8589
Epoch 99/100
800/800 [==============================] - 1s 845us/step - loss: 0.3405 - accuracy: 0.8601
Epoch 100/100
800/800 [==============================] - 1s 834us/step - loss: 0.3406 - accuracy: 0.8614

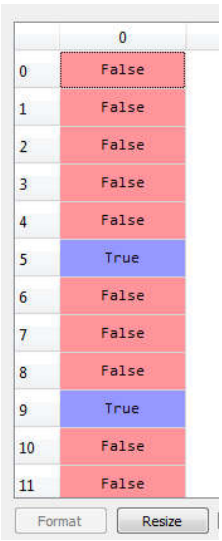
D:\1_Development_2.0\ML_phase_3_ML_Intro\ml_p3_ch8_dp_1_ANN>
```

8.4.5 Prediction and Evaluation of trained ANN model

We just trained ANN on the ***train-set*** and now time to make the ***predictions*** and the ***test-set***. We can execute following line as we did our classification models.

```
y_prd = ann_classifier.predict(X_test)
```





❒ After execution we get the ***predicted probabilities***. These are the **probabilities of *leaving the bank*** of the 2000 customers of the **test-set**.

☞ Now we can obtain the ***accuracy*** of the model using ***Confusion-matrix***. We got accuracy **86%** on **training-set**, now we're going to use test-set.

☞ Then if we get a good accuracy on the test-set , then the Bank is use this model on **all the customers of the bank** by ranking the *probabilities* from the ***highest to the lowest*** of the customers **most likely** to **leave** the **Bank**.

- ▶ So then for example the bank to have a look at the **10 percent highest probabilities** of their customers ***to leave the bank*** and so make it a ***SEGMENT*** and then **analyzed in more depth** using ***Data Mining Techniques*** to understand why the ***customers*** of this ***segment*** are the *most likely to leave the bank*.
- ▶ Then the Bank itself can take some measures to prevent these customers from leaving.

☞ However, ***predict()*** method returns the ***probabilities*** in range ***[0, 1]***. But in ***confusion matrix*** we just need **0** or **1** i.e. "**false**" or "**true**". So we need to **convert** this **predicted probabilities** into "**false**" or "**true**".

- ▶ We have to set a ***threshold value*** to decide when the predicted result is ***1(true)*** and when the predicted result is ***0 (false)***. And the ***threshold value*** we set is 0.5 or 50%.
- ▶ The code is simple, we just need to apply a ***condition*** over the ***y_pred*** **vector**.

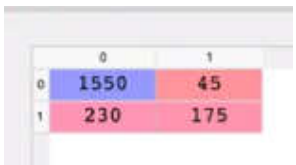
```
y_pred = (y_pred > 0.5)
```

- ▶ ***y_pred > 0.5*** is used because ***1*** means a customer will ***leave*** the ***bank***. So all values of ***y_pred*** greater than ***0.5*** will become ***1***.
- ▶ Now we can *proceed* to the *confusion-matrix*.

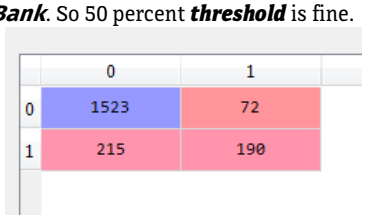
☝ In ***medicine*** we can take a **higher *threshold*** if what we have to **predict** is **sensitive information** like for example if we have to ***predict*** if a ***tumor*** is ***malignant***.

☞ But here we're just predicting if a customer is ***leaving*** or ***staying*** in the ***Bank***. So 50 percent ***threshold*** is fine.

Instructor-version:



Practiced-version:



☞ So out of ***2000*** new observations we get ***1550 + 175 correct*** **predictions** and ***230 + 45 incorrect*** **predictions**.

$$Accuracy = \frac{correct\ predictions}{totalpredictions} = \frac{1550 + 175}{2000} = 0.8625 = 86.25\%$$

☞ On new observations i.e **observations** on which we **didn't train ANN** we got an accuracy of 86%. (We get this prediction without being an Pro ! We didn't do any parameter tuning. So maybe we can still get an even better accuracy.)

Practiced-Version

```
# Artificial Neural Network
# Install : Tensorflow, Keras and Theano libraries.

# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np



# ------------------------------- Part 1 : Data Preprocessing ---------------------------------------
# Data Extract
dataSet = pd.read_csv("Churn_Modelling.csv")
# X = dataSet.iloc[:, 3:-1].values # this can be used too
X = dataSet.iloc[:, 3:13].values # all columns from index 3, excluding 13 indexed column
y = dataSet.iloc[:, 13].values # the last column


# ------------- Encode Categorical Data  -----------
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

#Encode "Gender" using LabelEncoder."Gender" is in "3rd-column", hence X[:, 2]
label_encode = LabelEncoder()
# Following is applicable to numpy array, if we used "X = dataSet.iloc[:, 3:13]"
# X = np.array(X) # it is needed if X is not an Arry. i.e. ".values" not applied
X[:, 2] = label_encode.fit_transform(X[:, 2])
print(X)

# For a data-set we can still encode it using Columns "key"
# X["Gender"] = label_encode.fit_transform(X["Gender"])


#Encode 'Gegraohy' using OneHotEncoder. "Gegraohy" is in "2nd-column", hence [1]
ct = ColumnTransformer(transformers = [("encoding", OneHotEncoder(), [1])], remainder = 'passthrough')
# remainder = 'passthrough' for remaining columns to be unchanged
X = ct.fit_transform(X)
X = np.array(X) # convert this output to NumPy array
print(X)
X = X[:, 1:] # Excluding 0-index column to avoid dummy-variable trap



# ------------------ Data Split -----------------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2, random_state = 0)

        # Feature-Scaling after Data Split



# ------------------ Feature-Scaling ----------------------
from sklearn.preprocessing import StandardScaler
#  y dependent variable, need not to be scaled: categorical variable, 0 and 1
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)



# ------------------------------- Part 2 : Creating ANN model ---------------------------------------

    # 1. importing "keras" libraries and packages
```

```
# from tensorflow import keras
import keras # using TensorFlow backend
from keras.models import Sequential
from keras.layers import Dense

    # 2. initialize the ANN
ann_classifier = Sequential()

    # 3. Add the "input-layer" and  "first Hidden-layer"
# ann_classifier.add(Dense(output_dim = 6, init = "uniform", activation = "relu", input_dim = 11))
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11))

    # 4. Add the "second Hidden-layer"
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))

    # 5. Add the "output-layer"
ann_classifier.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))

    # 6. Compile the ANN
ann_classifier.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"])

    # 7. Train the model: fit the ANN to Training-set (batch_size and epoch)
ann_classifier.fit(X_train, y_train, batch_size= 10, epochs= 100)




# --------------------- Part 3 : Predictions and Evaluating the model -------------------------------

# Predict
y_prd = ann_classifier.predict(X_test)

# coverting probabilities into "true/false" form. because 1 for leaving the Bank
y_prd = (y_prd > 0.5)

# Making the confusion matrix use the function "confusion_matrix"
# Class in capital letters, functions are small letters
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value

accURacy = (cm[0][0] + cm[1][1])/X_test.shape[0]
print(f"Accuracy = {accURacy}%")


# python prctc_ANN.py
```

Instructor-Version (updated?)

Now in ***latest*** version of ***tensorflow*** the ***Dummy-variable trap*** can be fixed automatically. So in following version there are all 3-dummy variables.

```
# Artificial Neural Network

# Importing the libraries
import numpy as np
import pandas as pd
import tensorflow as tf
tf.__version__

# Part 1 - Data Preprocessing

# Importing the dataset
dataset = pd.read_csv('Churn_Modelling.csv')
X = dataset.iloc[:, 3:-1].values
y = dataset.iloc[:, -1].values
print(X)
print(y)

# Encoding categorical data
# Label Encoding the "Gender" column
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
```

```
X[:, 2] = le.fit_transform(X[:, 2])
print(X)

# One Hot Encoding the "Geography" column
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1])], remainder='passthrough')
X = np.array(ct.fit_transform(X))
print(X)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)
print(X)

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

# Part 2 - Building the ANN

# Initializing the ANN
ann = tf.keras.models.Sequential()

# Adding the input layer and the first hidden layer
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))

# Adding the second hidden layer
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))

# Adding the output layer
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))

# Part 3 - Training the ANN

# Compiling the ANN
ann.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])

# Training the ANN on the Training set
ann.fit(X_train, y_train, batch_size = 32, epochs = 100)

# Part 4 - Making the predictions and evaluating the model

# Predicting the Test set results
y_pred = ann.predict(X_test)
y_pred = (y_pred > 0.5)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

NOTE

- ***Deep learning on GPU:*** GPU is a processor for graphic purposes. In terms of **power** and in terms of **computations efficiency** well the **GPU** is much **more powerful** because it has many more cores and it's able to run a lot **more floating points calculations** per second than the **CPU**.
  - GPU is much more specialized for highly **compute intensive task** and **parallel computations** exactly as it is the **case** for **neural networks**.
  - When we are for ***propagating*** the ***activations*** of the different ***neurons*** in the **NN** that exactly involves **parallel computations** and same when the **error** is **back propagated** and the NN. So lots of parallel computing, hence GPU is the better option.

- ***Tensorflow warnings:*** The ***W*** in the beginning stands for ***warnings***, ***errors*** have an ***E*** (or ***F*** for ***fatal errors***)
- In conda environment you need to install all the packages, Tensorflow, Theano, Keras, pandas, Scikit-learn, matplotlib etc.

- The updated instructor version uses the ***TF.keras*** and we used only ***keras***.

New documentation:

Add the first ANN layers (Input and Hidden Layers)

```
classifier.add(Dense(units=6, activation='relu', kernel_initializer='uniform', input_dim = 11))
```

Adding the second hidden layer

```
classifier.add(Dense(units = 6, kernel_initializer = 'uniform', activation = 'relu'))
```

To get ***TF 1.x*** like behaviour in ***TF 2.0*** one can run

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

but then one cannot benefit of many improvements made in TF 2.0. For more details please refer to the migration guide https://www.tensorflow.org/guide/migrate

Test tensorflow for first time:

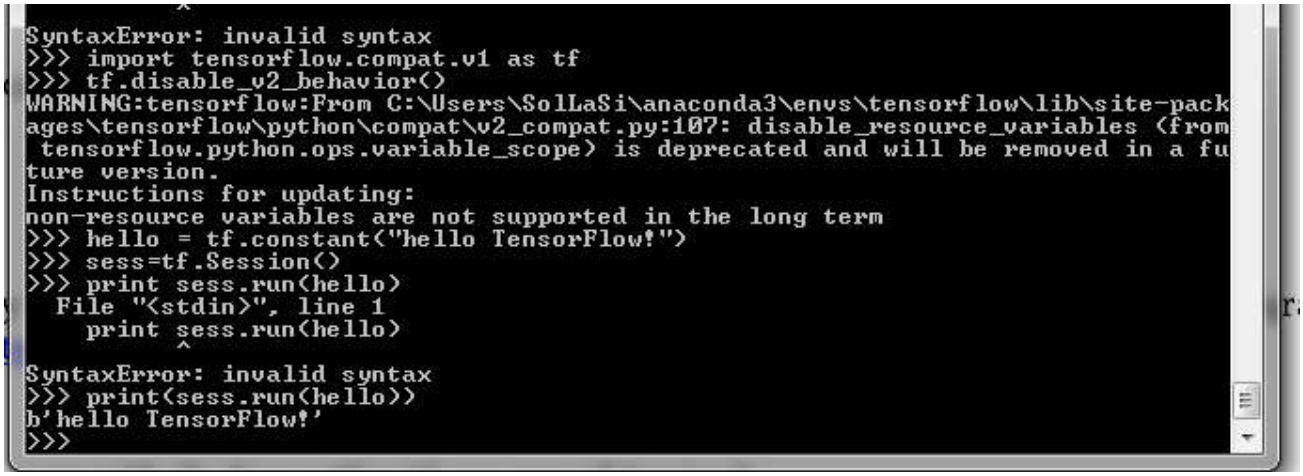
```
>>> import tensorflow as tf >>> hello = tf.constant("hello TensorFlow!") >>> sess=tf.Session()
```

To verify your installation just type:

```
>>> print(sess.run(hello))
```

If the installation is okay, you'll see the following output:

```
Hello TensorFlow!
```



Deep Learning

ANN: Predict new Data-point

📣 Here we already trained our ANN, using the given data (six-month observation of customers of a Bank).

📣 Now a new customer's data is arrived to us. We have to predict if the customer will ***leave*** or ***stay*** in the bank by using our ANN-model that we just built. The new data is given below:

☯ Use our ***ANN model*** to ***predict*** if the customer with the ***following informations*** will ***leave*** the bank:

Geography:	France
Credit Score:	600
Gender:	Male
Age:	40 years old
Tenure:	3 years
Balance:	$60000
Number of Products:	2
Does this customer have a ***credit card*** ?	Yes
Is this customer an ***Active Member***:	Yes
Estimated Salary:	$50000

So should we say goodbye to that customer ?

📣 ***What we need to do***:

[1] ***Arrange*** the new data in correct order (same order of our data-set),

[2] Find the ***right dummy variable*** for categorical variables (Geography, Gender etc),

[3] Transform the data into a ***NumPy array***,

[4] ***Scale*** the data-point,

[5] Make the ***prediction*** by converting the ***probability***.

❐ ***Arrange the new data in correct order:*** Let's compare our ***original data-set*** to ***Encoded-data-set*** feature-matrix ***X***.

☞ i.e. Feature-matrix ***X*** after ***encoding-categorical data*** and before **train-test split** and **feature-scaling**, so that we can compare the variables to ***original data-set***.

Original-Data

dataSet - DataFrame

Index	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	15634602	Hargrave	619	France	Female	42	2	0	1	1	1	101349	1
1	2	15647311	Hill	608	Spain	Female	41	1	83807.9	1	0	1	112543	0
2	3	15619304	Onio	502	France	Female	42	8	159661	3	1	0	113932	1
3	4	15701354	Boni	699	France	Female	39	1	0	2	0	0	93826.6	0
4	5	15737888	Mitchell	850	Spain	Female	43	2	125511	1	1	1	79084.1	0
5	6	15574012	Chu	645	Spain	Male	44	8	113756	2	1	0	149757	1
6	7	15592531	Bartlett	822	France	Male	50	7	0	2	1	1	10062.8	0
7	8	15656148	Obinna	376	Germany	Female	29	4	115047	4	1	0	119347	1
8	9	15792365	He	501	France	Male	44	4	142051	2	0	1	74940.5	0

Encoded Feature-matrix ***X***

X - NumPy object array (read only)

	0	1	2	3	4	5	6	7	8	9	10
0	0.0	0.0	619	0	42	2	0.0	1	1	1	101348.88
1	0.0	1.0	608	0	41	1	83807.86	1	0	1	112542.58
2	0.0	0.0	502	0	42	8	159660.8	3	1	0	113931.57
3	0.0	0.0	699	0	39	1	0.0	2	0	0	93826.63
4	0.0	1.0	850	0	43	2	125510.82	1	1	1	79084.1
5	0.0	1.0	645	1	44	8	113755.78	2	1	0	149756.71
6	0.0	0.0	822	1	50	7	0.0	2	1	1	10062.8
7	1.0	0.0	376	0	29	4	115046.74	4	1	0	119346.88
8	0.0	0.0	501	1	44	4	142051.07	2	0	1	74940.5

```
new_dt_pt = np.array([[0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0]])
```

☞ From, ***row*** no. ***0***, ***1*** and ***7*** we notice ***France*** = `(0.0, 0.0)`, ***Spain*** = `(0.0, 1.0)`, ***Germany*** = `(1.0, 0.0)` represented using the ***dummy variables***.

➲ Since Dummy variables of ***Geography*** appears ***first*** in our ***feature matrix***, we need not to ***rearrange*** the ***columns*** for our new data-point **`new_dt_pt`**.

➲ Also we not need to re-arrange other columns, they are in right order.

☞ From, ***row*** no. ***0*** to ***4*** and ***5***,***6*** we notice ***Male*** = ***1*** and ***female*** = ***0***

☞ **Credit card** : ***yes*** = ***1***, ***no*** =***0***

☞ **Active-member** : ***yes*** = ***1***, ***no*** =***0***

☞ Hence we represent our new data-point **`new_dt_pt`** as:

```
0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0
```

☝ Now if we use it as a ***list***, it ***won't be a row***, it will be a **`vector/column`**:

```
[0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0]
```

➲ To make it as ***row*** of the ***feature matrix*** we define it as a ***list-of-list*** i.e. **`[[]],`** as 1x11 matrix

```
[[0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0]]
```

☝ ***NumPy array:*** We also need to **convert it to *NumPy array***, we use ***`np.array`*** (here numpy imported as **`np`**):

```
np.array([[0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0]])
```

We put it to the variable called **`new_dt_pt`**.

☞ ***Scale:*** Then we ***scale*** this new data-point **`new_dt_pt`**, using our *Standard Scaler* **`st_x`**.

☞ ***Predict:*** Finally ***predict*** the new data-point **`new_dt_pt`**, using the classifier `ann_classifier` and ***transform*** the returned ***probability*** into **`True/False`** using ***threshold*** value ***0.5***.

```
new_dt_pt = np.array([[0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0]])
new_dt_pt= st_x.transform(new_dt_pt) # scaling
predict_data_pt = (ann_classifier.predict(new_dt_pt) > 0.5)
```

All code for new data-point prediction

```
# first create a 2D "NumPy array" in our X_train's format.
    # it will be smilar to a single row of our X_train
    # 2 "[" used to define a single row of a 2-D array
new_dt_pt = np.array([[0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0]])
new_dt_pt= st_x.transform(new_dt_pt) # scaling
predict_data_pt = (ann_classifier.predict(new_dt_pt) > 0.5) # Predict the data-point
```

☝ ***Result:*** The prediction is "**`False`**". That is the customer ***not going to leave the Bank***. Since `leave = 1, stay = 0` in dependent variable, "**`Exited`**". Here ***`True/False`*** is represented by ***1*** or ***0***.

new_dt_pt	Array of float64	(1, 11)	[[-0.5698444 -0.57369368 … -0. ...
predict_data_pt	Array of bool	(1, 1)	[[False]]

Hence we ***don't say Goodbye*** to that customer.

Chapter 9 : Part 1

Deep Learning

CNN: Convolutional Neural Network

Introduction to CNN

9.1.0 Overview of what we will learn

What we will learn in this section:

- What are Convolutional Neural Networks?
- Step 1 - Convolution Operation
- Step 1(b) - ReLU Layer
- Step 2 - Pooling
- Step 3 - Flattening
- Step 4 - Full Connection
- Summary

- EXTRA: Softmax & Cross-Entropy

[1]. ***What Convolutional-Neural-Networks (CNN) actually are:*** We'll have a look at a few examples. We'll compare the human brain to ***Artificial Neural Networks*** in terms of ***Image Recognition***.

[2]. ***Step 1 –Convolution Operations:*** This is a part of the steps to build a CNN. We'll learn about feature detectors, filters, feature maps, and the different parameters- what they mean and have a look at some visual examples.

[3]. ***Step 1 (b) – ReLU Layer:*** It is the Rectified Linear Unit (ReLU) and we'll talk about why ***linearity is not good*** and how we want ***more nonlinearity*** in our ***network*** for ***`image recognition`***.

[4]. ***Step 2 – Pooling:*** We'll understand how **pooling** works. We'll talk specifically about **Max-pooling** and also **mean-pooling** or **sum-pooling** and other approaches that you can take to the process of **pooling**. We'll see some example.

[5]. ***Step 3 – flattening:*** It's going to be a quick tutorial on how to proceed from your ***`Pooled-layers`*** to ***`Flatten-layer`***.

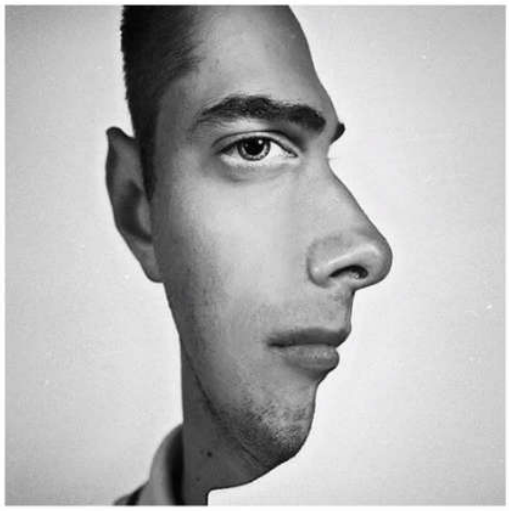
[6]. ***Step 4 - Full Connection:*** In this section we put everything together and everything into perspective. Here we will understand how everything works. How those final neurons understand how to classify Image.

[7]. ***Summary:*** Summarize everything we've talked about.

[8]. ***Softmax and Cross-Entropy:*** Not compulsory but these are terms that you will come across when dealing with CNN.

9.1.1 Convolutional-Neural-Networks (CNN)

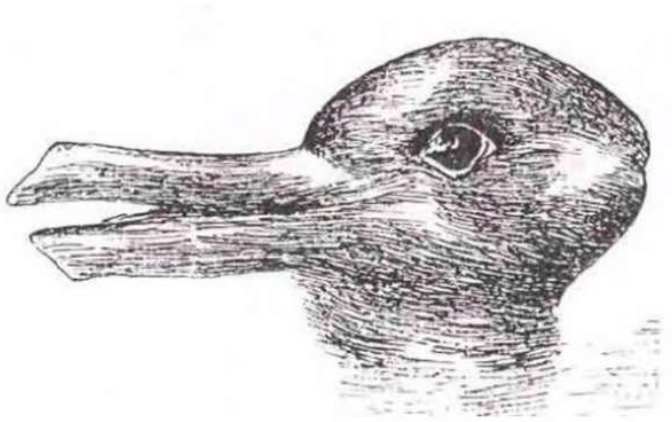
Look at this image. Do you see a person looking at you or do you see a person looking to the right. Here your brain is struggling to adjust, if you look to the right side of the image you'll see a person looking to the right. If you look at the left side of the image you'll see a person looking at you.

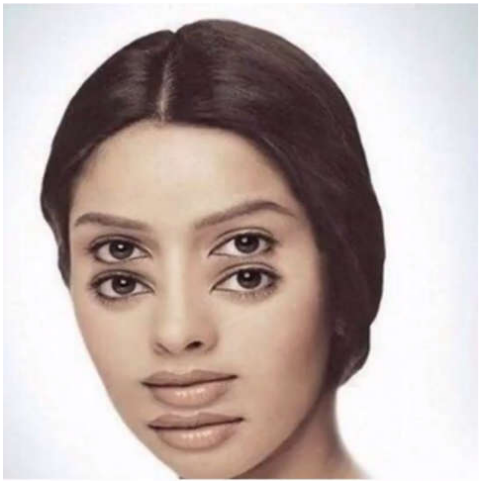


❒ This proves that when we see things is actually its ***features***. Depending on the features our brain process.

- ☞ So when you look on the ***right side*** of the ***image*** you see ***certain features*** of a person ***looking to right*** because they're closer to your center of focus and therefore your brain ***classifies*** as a ***person looking to the right***.
- ☞ When you look to the ***left side of the image*** you see more ***features*** of a ***person looking at you*** and therefore your brain classifies it as such.







- Most of this kind of illusion image we can see two in one and ***depending*** on which ***features*** our ***brain picks up*** it will ***switch between*** classifying each image as one or the other. The oldest one of these illusions recorded in the printed work is ***"duck or the rabbit"***. Here is also ***"Young girl or Old lady"***.

  - For this kind of image your brain is trying to understand what is it. what it is like it's trying to. This is a classic example of when there are certain features where your brain cannot decide.

- All these examples illustrate to us ***how the brain processes certain features*** on an image or on whatever you see in real life and it classifies that as.

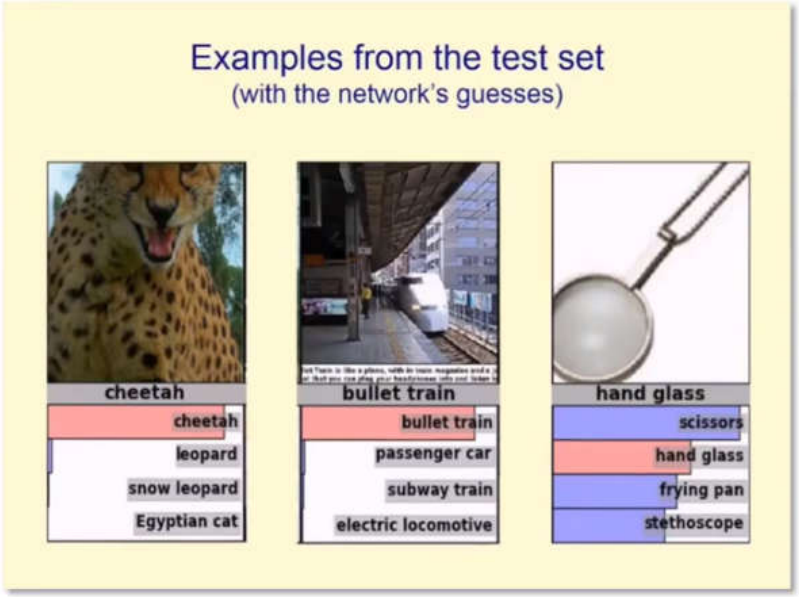
  - You probably have been in **situations** when you ***look over your shoulder quickly*** and you see something it's like a ***Ball*** but it turns out to be a ***Cat*** because you don't have enough time to **process** those **features** or you don't have enough **features** to **classify**.

  - The CNN works in very similar way. And computers can interpret them as we do.

- ***Here's an experiment*** done on computers on CNN: here you see three images and we're going to go through them with left to right and see how you would classify then we see how computer classify.

  - So on the left you probably say ***Cheetah*** and computer said so (we're ***going to learn how to read these images*** because if you going to go deep into CNN we're going to start learning more and more and see a ***lot of these kind of images***.)

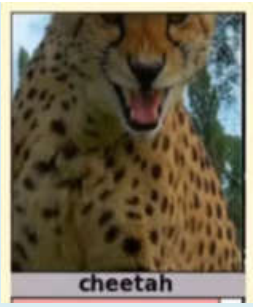
  - In 2nd image the ***neural network*** was able to ***distinguish*** between ***bullet-train/ passenger-car/subway-train/electric-locomotive***.



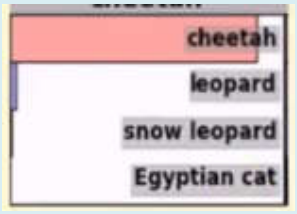
- Actually there could be ***many more options*** and the ***NN*** learn to ***distinguish*** from those ***categories*** at the ***same time***.
- For the ***third image***, there are couple of options and it's not very clear what is it could be a ***frying-pan/magnifying-glass/pair of scissors***
  - You can see that the ***Probabilities*** are not as clear here so the ***neural network*** was a bit ***confused***.
  - Basically here you can see that ***scissors*** was its ***first guess*** but the correct option was number two and that's why it's ***highlighted*** in ***red***.

Later in this chapter we will learn, what these ***VOTES mean*** and how they are ***derived***.

How we read these image: So that's the actual correct label of the image "cheetah" in the ash color. That's the label of the images without any processing.

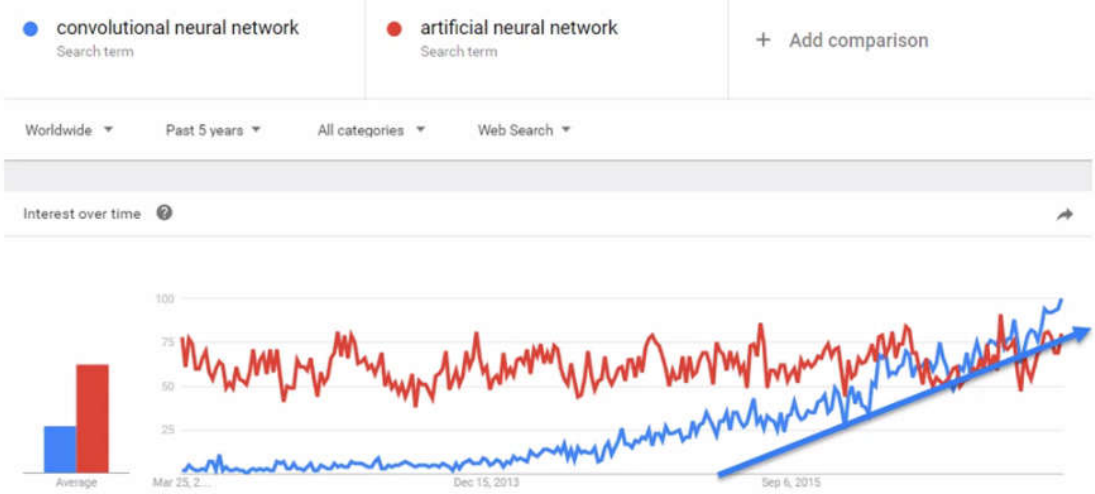


- And the computer vision (prediction) are here: the guesses the top four or five. They're given the probabilities so the computer said or the CNN said.
- It said with a high probability "it's a cheetah" about like 95% or 99%.





CNN gained so much ***popularity*** over ***ANN***. Because it is a very important field: that that is where all like ***self-driving cars***, ***recognize people*** on the ***road***, how to recognize ***stop signs***, and things like that are build. How Facebook able to tag people in images also based on CNN.





Yann Lecun

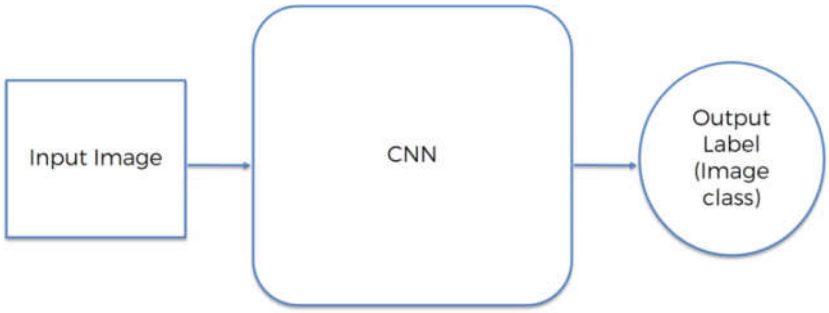
Google Facebook



☞ If ***Jeffrey Hinton*** is the godfather of ***ANN*** and deep learning. Then ***Yann Lecun*** is the grandfather of ***CNN***. ***Lecun*** is a student of ***Jeffrey Hinton's***.

9.1.2 How CNN works

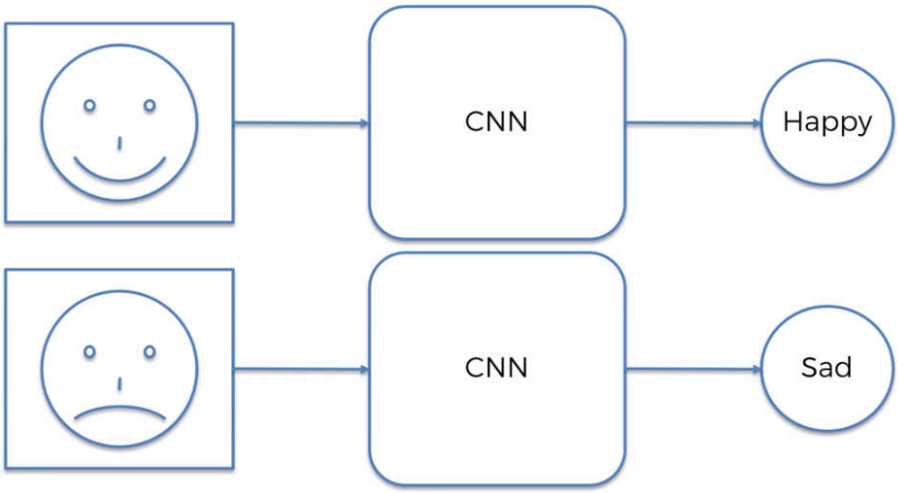
You have an ***input image*** it goes through the ***CNN*** and you then get a ***labeled-image*** as an ***output***. So it classifies that image. As something like has a ***Cheetah*** or a ***Bullet Train*** or something else.



☞ A CNN can **trained** upon certain **classified images** and it can **classify similar images** from a given ***test-image***. Let's say a ***CNN*** has been trained up to recognize ***facial expressions***. You can give it a face of a ***smiling person*** and also a face of ***sad person*** and you train CNN from bunch of ***pre-classified images***. Then after training the ***CNN*** can detect the ***Happy/Sad*** person from a ***test image***.

☞ CNN gives you the ***probability***, for example ***85% chance of happy-Person*** or ***95% Sad***.

☞ And CNN can get confused sometime (as we get for some image/illusions).



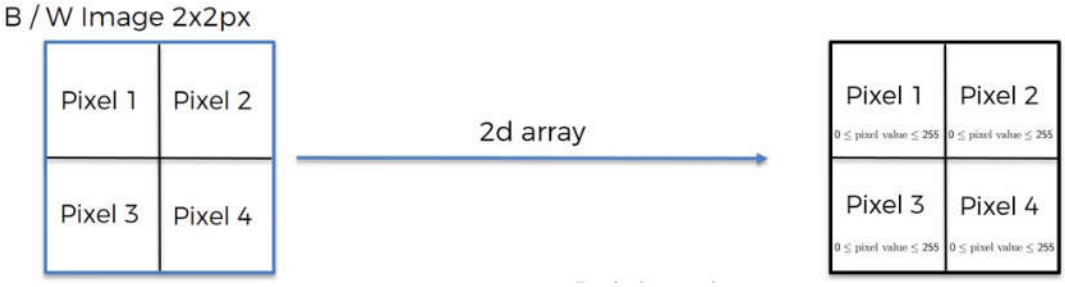
❒ ***How does CNN able to recognize these features:*** Let's say you have two images one is ***black and white image*** of ***2x2 pixels*** and Other one is a ***colored image*** of ***2x2 pixels***.

☞ ***Black & White - image:*** NN takes the black and white image is a two dimensional array with **each of those pixels** having a value between ***0*** and ***255*** (that's 8 bits of information $2^8 = 256$.).

➲ The values from ***0*** to ***255*** and that's intensity of the color, ***0 = black pixel*** and ***255 = white pixel*** and between them you have the grayscale range of possible options for this pixel.

➲ And based on that information computers are able to work with the image as: Any image is actually has a digital representation/digital-form. And those are just basically ***1*** and ***0*** that form a number ***0*** to ***255*** for every ***single pixel***.

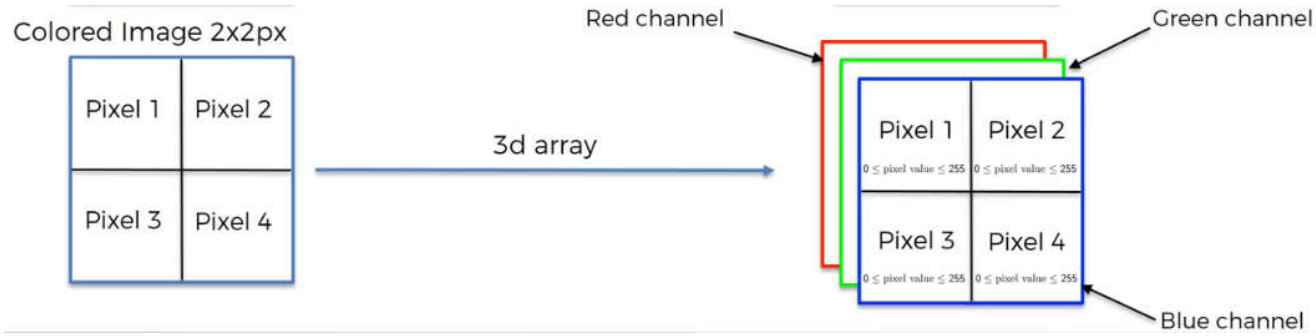
➲ It ***doesn't*** actually work with ***colors*** or anything, it works with the ***1***s and ***0***s.



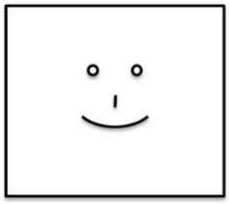
☞ ***Colored image:*** And in a color image it's actually a 3-dimensional array. You've got in ***RGB***, ***Blue-layer***, ***Green-layer*** and the ***Red-layer***. And ***each one*** of those ***colors*** has its ***own intensity***.

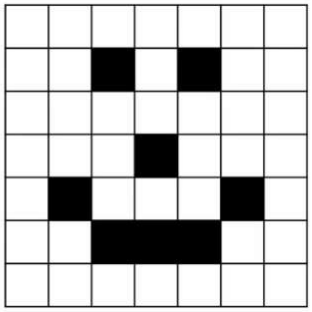
➲ So basically a ***pixel*** has ***3 values*** assigned to it. Each one of them is between ***0*** and ***255***. Computers are going to work by combining those three values.

➲ Those are the ***red channel***, the ***green channel***, the ***blue channel***.

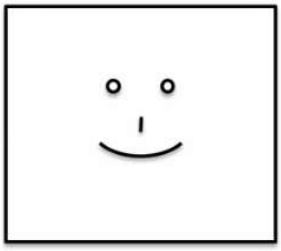


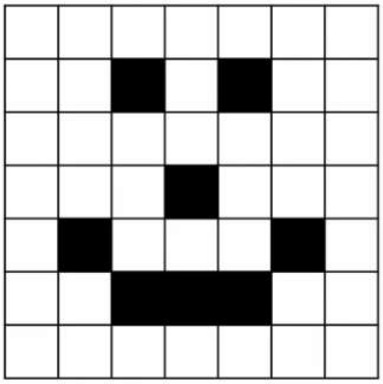
Example: Let's have an example of a smiling face. If we simplify things, instead of having values from ***0*** to ***255*** if we use only ***1***'s and ***0***s, ***0 = white*** and ***1 = black***, then we can see that that image can be represented as below.





In this chapter we will more discuss about images like this, which is very simple having only ***1***'s and ***0***s, but at the same time all those concepts can applied to the ***0*** to ***255*** range of values.





0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

❒ ***Steps to process images in CNN:*** And the steps are we're going to process these images are:

STEP 1: Convolution

⇩

STEP 2: Max Pooling

⇩

STEP 3: Flattening

⇩

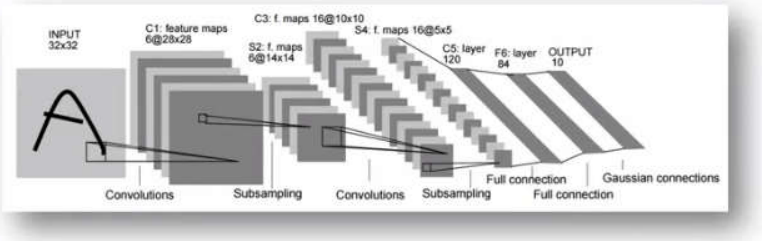
STEP 4: Full Connection

❒ ***Paper (Additional reading):*** This is the ***Yann LeCun's*** original paper that gave rise to ***CNN***. It's called **"Gradient Based Learning Applied To Document Recognition".** If you want to go back to the very beginnings of how it all happened, where it all came from this is the paper to look into.

Additional Reading:

Gradient-Based Learning Applied to Document Recognition

By Yann LeCun et al. (1998)



Link:

http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

Chapter 9 : Part 2

Deep Learning

CNN: Convolution Operations & ReLU layer

9.2.1 Convolution

❒ ***Convolution:*** Following is the ***convolution function***. A ***convolution*** is basically a ***combined integration*** of the two ***functions*** and it shows you ***how one Function modifies the other*** or (modifies the shape of the other). If you've done any ***Signal Processing*** or ***Electrical Engineering*** then you are already familiar with it.

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$

❒ ***Additional Reading:*** If you want to know the mathematics of ***convolution function*** and CNN read the article below. is called *"Introduction to convolutional neural networks"* by ***Jianxin Wu*** who is a professor at ***Nanjing University*** in China. It is oriented specifically at people who are ***beginners*** to know ***CNN*** so the mathematics there should be accessible.

Additional Reading:

Introduction to Convolutional Neural Networks

By Jianxin Wu (2017)

$$\begin{aligned}\frac{\partial z}{\partial(\mathrm{vec}(\boldsymbol{y})^T)}(F^T \otimes I) &= \left((F \otimes I)\frac{\partial z}{\partial\, \mathrm{vec}(\boldsymbol{y})}\right)^T \\ &= \left((F \otimes I)\,\mathrm{vec}\left(\frac{\partial z}{\partial Y}\right)\right)^T \\ &= \mathrm{vec}\left(I\frac{\partial z}{\partial Y}F^T\right)^T \\ &= \mathrm{vec}\left(\frac{\partial z}{\partial Y}F^T\right)^T,\end{aligned}$$

Link:

http://cs.nju.edu.cn/wujx/paper/CNN.pdf

https://cs.nju.edu.cn/wujx/index.htm

❒ ***Convolution in intuitive terms:*** Here we got an input image, just ***ones*** and ***zeros*** to simplify things. You can see the ***smiley face*** there.

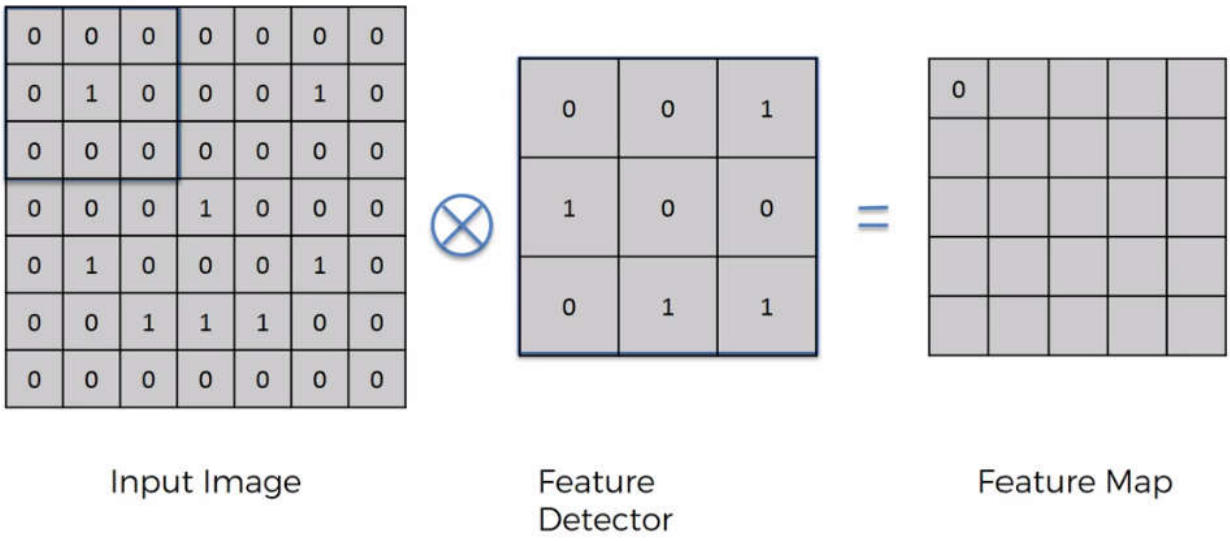
☞ ***Feature Detector:*** We've also got a feature detector. This feature detectors a 3×3 Matrix. However it ***could be any size***. Also the ***feature detectors*** called ***Kernel*** or you might hear it being called ***Filter***. We're going to be using either ***filter*** or a ***feature detector*** interchangeably. And a ***Convolution operation*** is signified by ⊗ ***"x in a circle"***.

0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input Image

0	0	1
1	0	0
0	1	1

Feature Detector



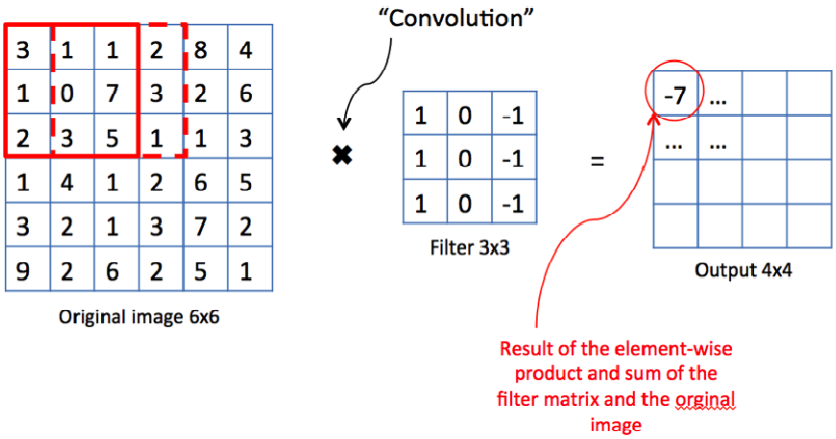
❒ ***How Feature Detector works:*** Here we are going to see what is actually happening in the background rather than the mathematics.

☞ You take the ***feature detector*** or ***filter*** and you put it on your image (like you see on the left). For instance in above case the top **left corner-nine pixels** and you basically ***multiply each value*** by ***respective value***. It's ***not*** the ***matrix multiplication***, its actually ***"element-wise multiplication and sum of the values"***. So

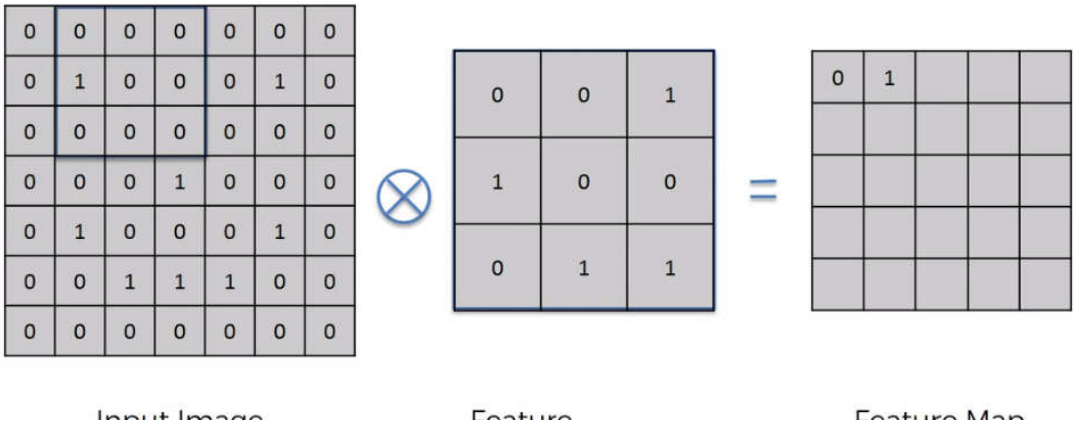
$$a_{11}b_{11} + a_{12}b_{12} + a_{13}b_{13} + \cdots$$

☞ Actually we are looking for how many matches are found in ***"Input image"*** w.r.t. ***"Feature Detector"***. So in this first case nothing matches up (it's always either 0×0 or by 0×1) so the result is ***0***.

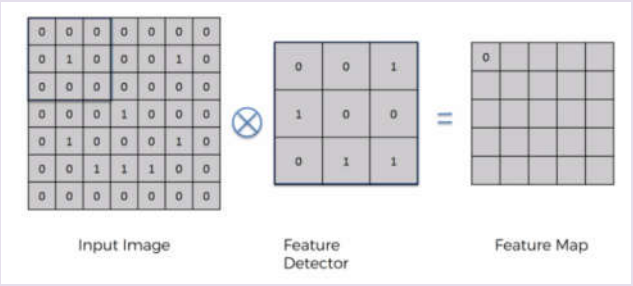
☞ Feature detector can also contain negative values:

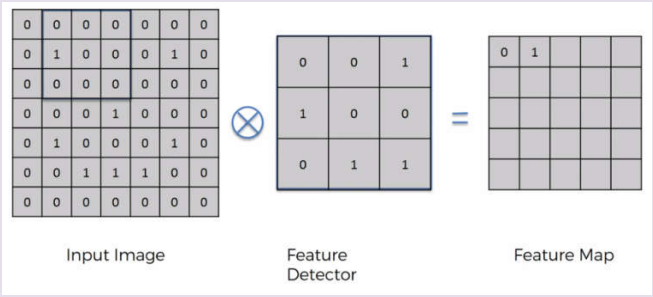


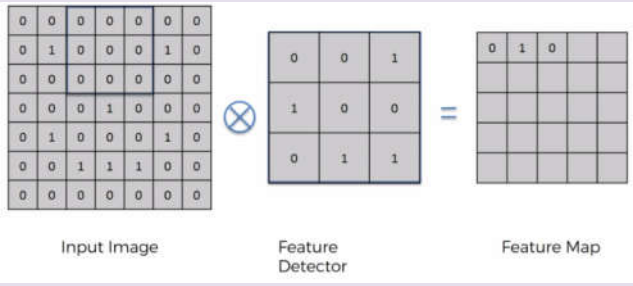
☞ If we shift 1px to right we can see that one of the 1 is matched up And therefore we've got a 1 here.

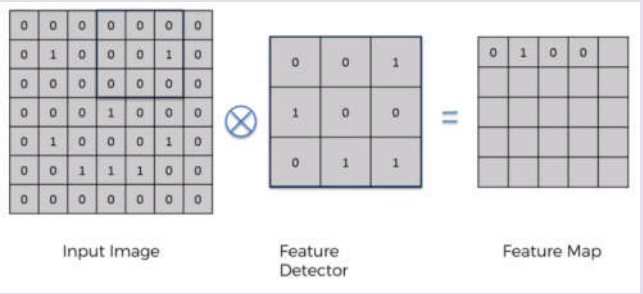


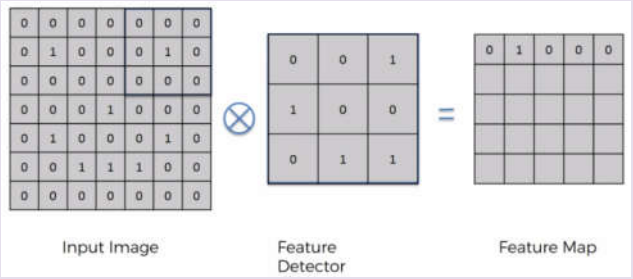
Hence we get the followings

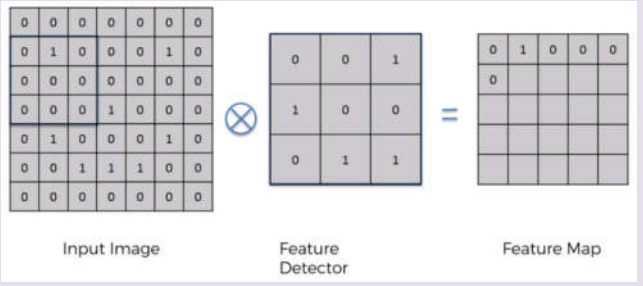


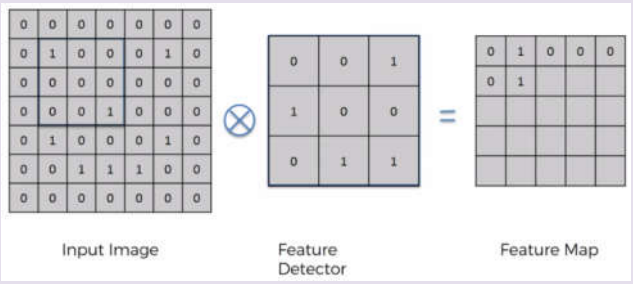


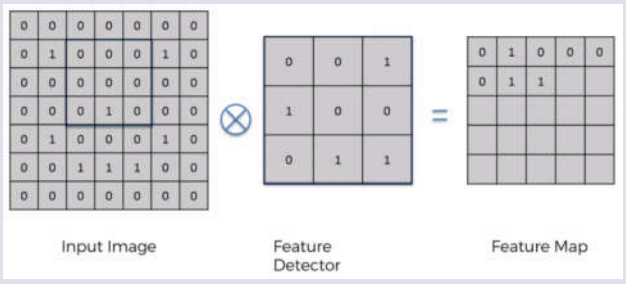


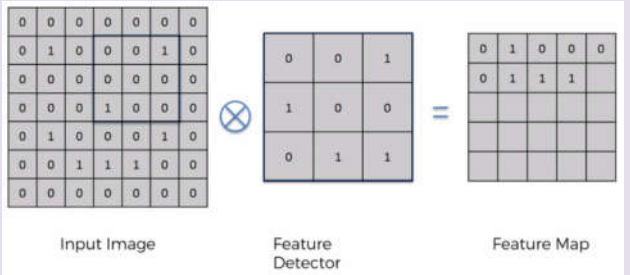


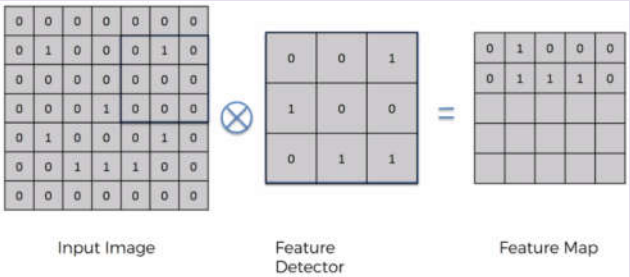


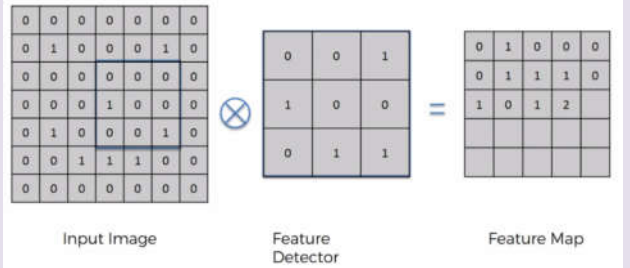


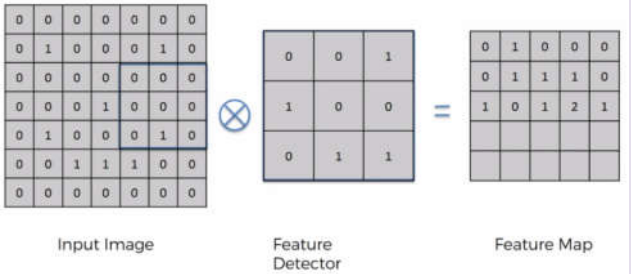


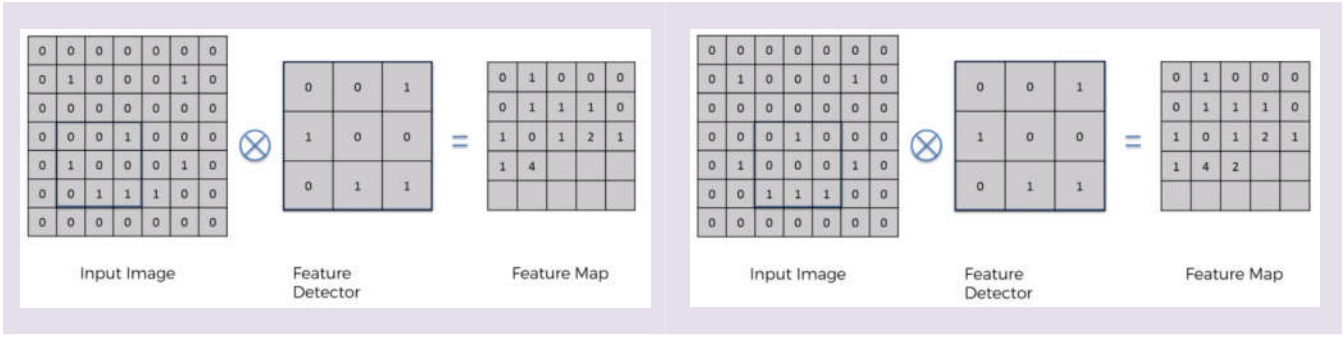




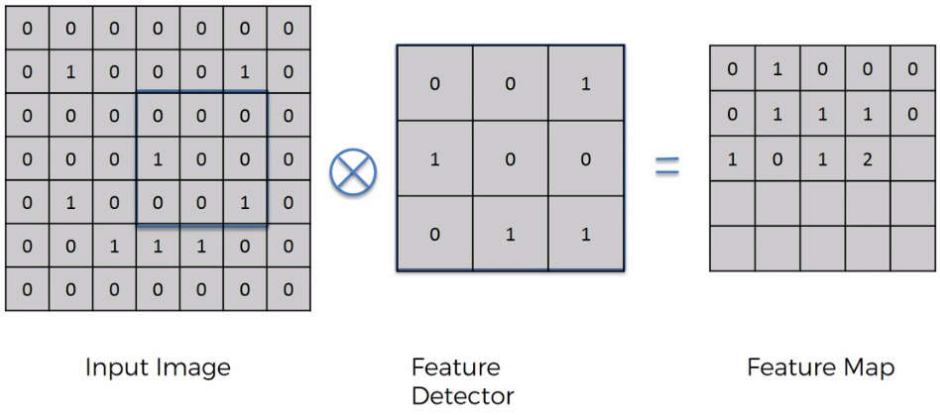




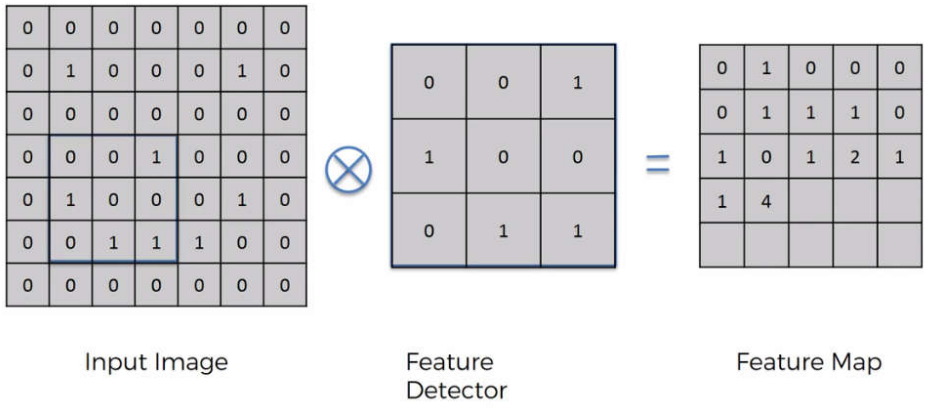




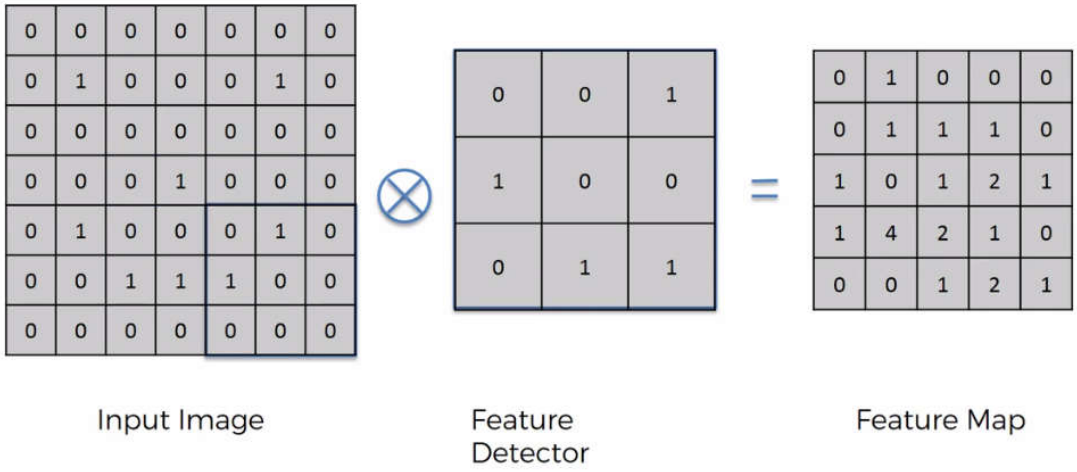
☞ Notice that, following has matching ***two 1's*** hence ***sum*** is 2.



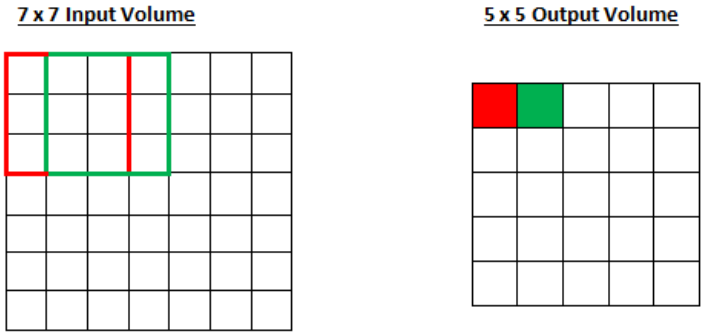
☞ Also notice that we got ***All Matching*** four 1's. Hence the sum is 4.



☞ Finally we got the following ***feature map***.

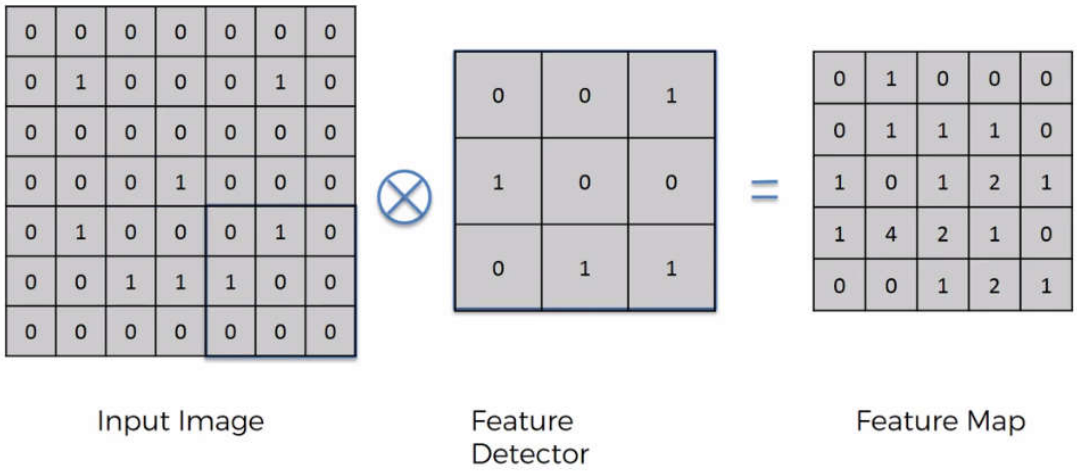


❒ ***Stride:*** ***Steps*** at which we're moving this whole filter is called the ***stride***. So here we have a ***stride*** of ***one pixel***. Stride is a component of CNN, or neural networks tuned for ***the Compression of Images*** and ***Video*** data. ***Stride*** is a ***parameter*** of the ***neural network's filter*** that modifies ***the amount of movement over the image*** or ***video***. For example, if a neural network's stride is set to 1, the filter will move one pixel, or unit, at a time. The size of the filter affects the encoded output volume, so stride is often set to a whole integer, rather than a fraction or decimal.



☞ Imagine a convolutional neural network is taking an image and analyzing the content. If the filter size is 3x3 pixels, the contained nine pixels will be converted down to 1 pixel in the output layer. Naturally, as the stride, or movement, is increased, the resulting output will be smaller.

❒ ***Feature map:*** The image on the right is called a ***feature map*** also called ***convolved feature*** can also be called the ***activation map***. So when you apply ***convolution operator*** to something, it ***doesn't*** become ***convoluted*** it becomes ***convolved***.



☞ Here we've actually reduced the size of the image. If you have a **stride of one** you can see the image reduced a bit, but if you have **stride of two** the image is going to reduce more, so the ***feature-map*** is going to be even ***smaller***.

☞ The purpose of this whole convolution step is to make the image smaller, so that, it'll be easier & faster to process it. Therefore **feature detectors** will ***reduce*** the ***size*** of the image and therefore **stride of two** is actually beneficial.

❒ ***Information loss:*** Some information we are losing because we have less values in resulting matrix. But at the same time the purpose of the ***feature detector*** is to **detect certain features** at certain parts of the image that are integral.

☞ For instance if you think about it this way: the ***feature detector*** has a certain ***pattern*** on it, the highest number in your feature map is when that ***pattern matches*** up. As in our case it is **4 the highest value**, because ***feature-detector*** **fully matched**.

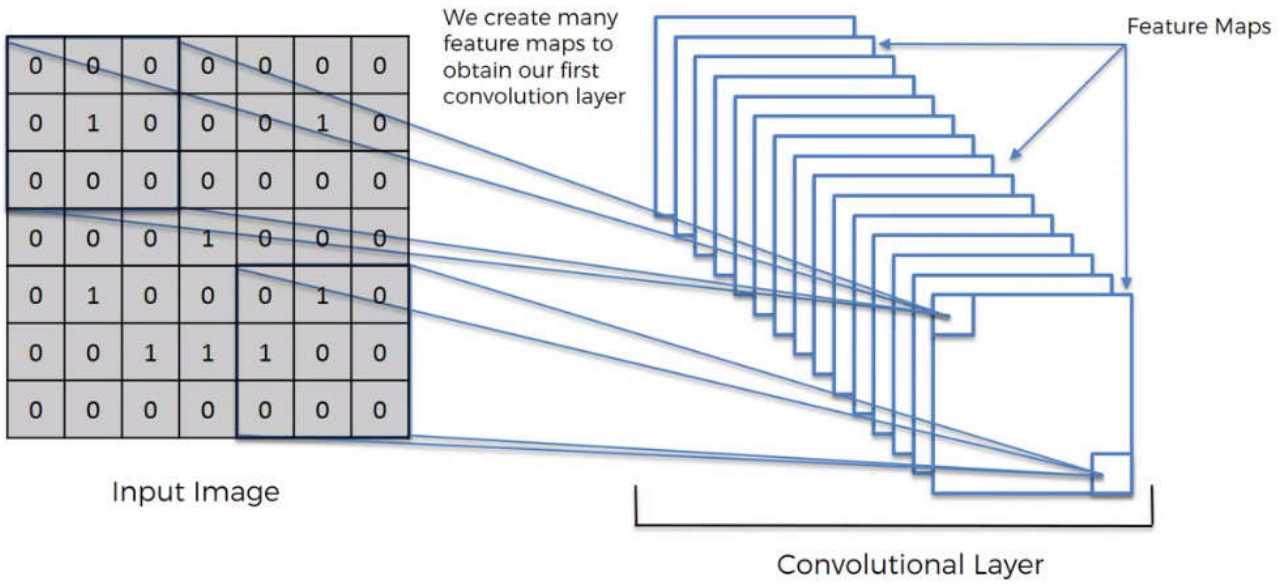
☞ It helps us to ***focus*** on the ***features***, as for detect a "***Cheetah***" we look at its "***eye mark***". We don't consider all of the pixels.

❒ ***Multiple FEATURE MAPS with different FILTERS/FEATURE-DETECTOR:*** For our input image, and we create multiple feature maps because we use different filters/feature-detector. That's another way that we preserve lots of the information. (Kind of extracting features).

☞ Here we don't just have one feature map. Basically we want to **detect *features***. Each ***layer*** has its ***own feature map*** for certain ***features***.

☞ When we look for ***certain features,*** the ***network decides*** through its ***training*** which ***features*** are ***important*** for certain ***types/categories*** and it looks for them using different ***filters/feature-detectors***. It'll apply these ***filters/feature-detectors*** so to get certain ***feature map***.

☞ That's why the term ***Feature Detector*** is better than the term ***Filters***.



Lets see some example of filters for an image app. We are using ***Gimp.org***. Here we have a picture of the ***Taj Mahal*** and you can choose ***which filter*** you want to apply. So if you download this program and you upload a photo into it and then you can actually start a ***Convolution-matrix*** and ***apply filters***. Following are some commonly used filters.

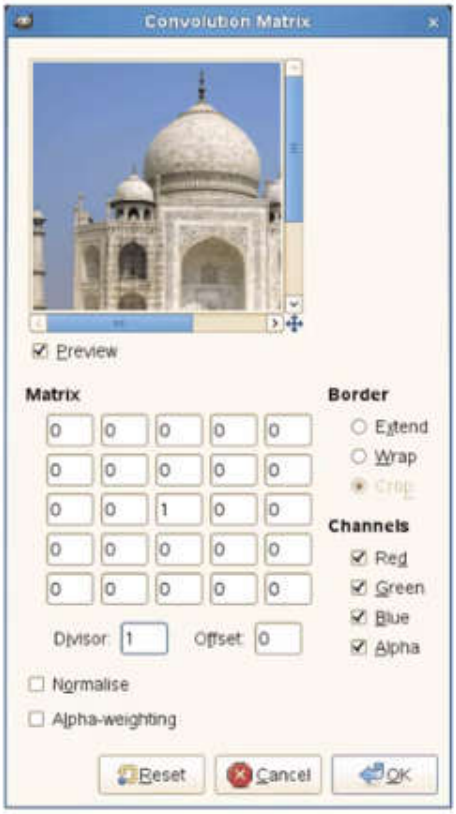
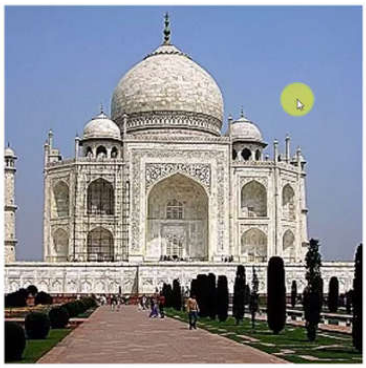


Image Source: docs.gimp.org/en/plug-in-convmatrix.html

☯ ***Sharpen:*** If we apply this filter, we can see that it sharpens the image. 5 in the middle as main pixel and four -1's. Kind of reduces the pixels around.

Sharpen:

0	0	0	0	0
0	0	-1	0	0
0	-1	5	-1	0
0	0	-1	0	0
0	0	0	0	0



☯ ***Blur:*** All six 1's. Basically takes equal significant to all of the pixels and therefore it combines them together and you get a blur.

Blur:

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0



☯ ***Edge Enhance:*** So here you can see that's -1 and 1 and then you get all 0's (i.e. remove the pixels around the main one). It gives you an edge.

Edge Enhance:

	0	0	0	
	-1	1	0	
	0	0	0	



☯ ***Edge Detect:*** So here we are detecting edge. You reduce the middle one as -4. And use 1's around it.

Edge Detect:

	0	1	0	
	1	-4	1	
	0	1	0	



☯ ***Emboss:*** So the key point here is that this matrix is asymmetrical and you can see the image becomes asymmetrical.

Emboss:

	-2	-1	0	
	-1	1	1	
	0	1	2	



☝ So we can see these are great examples of the same image but we're getting different ***feature maps*** using different ***filters/feature-detectors***.

✋ *But these terms:* Emboss/Sharpen are not applicable to us (we are not working with Gimp). We are not going to use those terms in CNN. However, ***`Edge-Detect`*** is quiet important for CNN but not the others.

✋ CNN will decide for itself what's important what's not and it probably won't be even recognizable to the human eye. You won't be able to understand **what those *features* mean**.

☞ That's the beauty of CNN. They can process so many different things and understand without even having that explanation *why they will understand which features are important to them* whether we have a **term/name for those features** or not.



$*$

1	0	-1
2	0	-2
1	0	-1



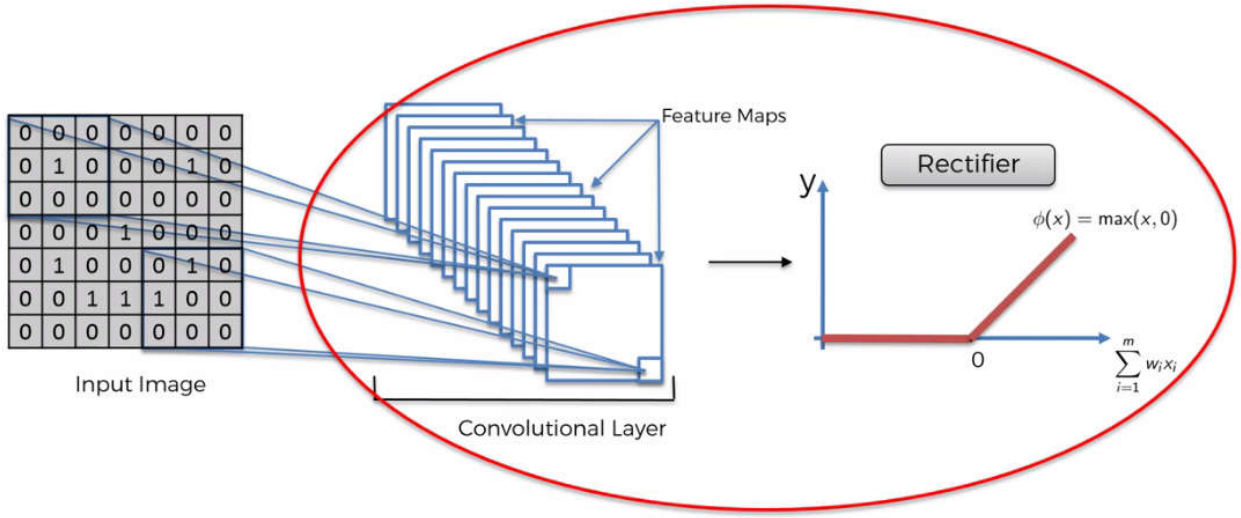
Here's a image of Geoffrey Hinton passed through one of these filters.

☝ ***The key-point is:*** the primary purpose of ***Convolution*** is to find ***features*** in your **image** using the ***feature detector*** put them into a ***feature map*** by preserving the spatial relationships between pixels (if they are completely jumbled up then we've lost the pattern).

☞ It's also important to understand that most of the time the ***features*** a ***neural network*** will **detect/use** to **recognize** certain **images** and **classes**, it will mean nothing to humans.

9.2.2 Rectified Linear Unit (ReLU)

ReLU (Rectified Linear Unit) is an ***additional step*** on top of our ***Convolution*** step. Here we have our ***input image*** and we have all ***Convolution Layer*** then on top of that we're going to apply ***rectifier function***. The reason why we're applying the rectifier we want to ***Increase Non-Linearity*** in our image or in our CNN. And rectifier acts as that filter or that function which breaks up that linearity.



❒ ***Why increase Non-linearity:*** We want to increase nonlinearity in our network is because ***images*** themselves ***are*** highly ***non-linear*** especially if you're ***recognizing*** different ***objects next to each other***.

☞ Images are going to have lots of ***nonlinear elements*** and the ***transition*** between ***pixels-adjacent pixels*** is often would be ***nonlinear***, because of its borders, different colors, different elements in your images.

☞ When we're applying a ***mathematical operation*** such as ***convolution***, and running this ***feature detection*** to create our ***feature maps*** there is a possibility that we might create something ***linear*** and therefore we need to ***break up the linearity***.

Example: Here is a image an original image. Now when we apply a ***feature detector*** to this image we get following:	
☞ You can see that ***black*** is ***negative*** and ***white*** is ***positive***. When you apply a feature detector to a real-life image, (not just 1 or 0 but lots of different values, including negative values) we get this kind of result.	

☝ Now a ***Rectified Linear Unit (ReLU)*** function removes all the black (anything **below zero** it turns into **zero**).



☞ Here it's pretty hard to see what exactly is the benefit in terms of breaking up linearity. This is a very mathematical concept and would have to go into a lot of math to really explain what is going on. Now we try to figure out without math.

❒ ***What is LINEARITY in here:*** For instance consider the shadows of the white building. In ***black-negative*** and ***white-positive*** version notice the light. You see that it's ***White*** (the reflection of the light) and then it's a ***Gray*** and then it gets ***Darker*** and then it gets ***Darker*** again. It looks like when you go from ***White*** to ***Gray*** the next step would be ***Black***. So it's a ***Linear Progression*** from ***Bright*** to ***Dark*** and therefore this is kind of ***`linear`*** situation.

☞ When you take out the ***`black`*** you ***`break up`*** the ***`linearity`***. There **won't be** any **gradual progression** like ***White-Gray-Black***.



Real image



Linear: Bk = -ve, W=+ve



Non-Linear: only +ve

☞ Like having an ***abrupt change***, which helps to introduce ***non-linearity*** into your ***image***.

☝ ***Additional reading:*** But if you'd like to learn more there's a good paper. This one is by ***C.C. Jay Kuo*** from the ***University of California*** and it's called ***"Understanding Convolutional neural networks with a mathematical model"***.

☞ ***There he answers this question:*** Why a ***nonlinear activation function*** is essential at the ***filter output*** of all ***intermediate layers***. It explains it in a bit more detail both in terms of intuition and mostly in terms of ***mathematics***.

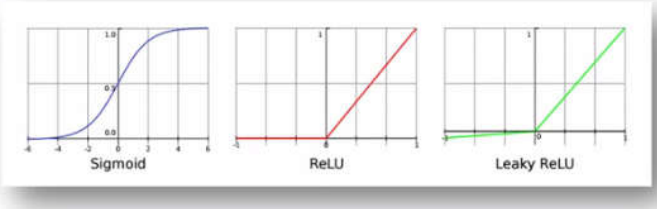
Additional Reading:

Understanding Convolutional Neural Networks with A Mathematical Model

By C.-C. Jay Kuo (2016)

Link:

https://arxiv.org/pdf/1609.04112.pdf



📣 And if you really want to dig in and explore more. Then there's another paper that you might be interested in. It's called ***"Delving Deep Into Rectifiers: Surpassing Human Level Performance on ImageNet classification"*** by ***Kaiming He*** from ***Microsoft Research***.

☞ They proposed a different type of ***ReLU function*** which you see here on the right. Which is parametric ReLU function. And they argue that it delivers better results ***Without Sacrificing PERFORMANCE***.

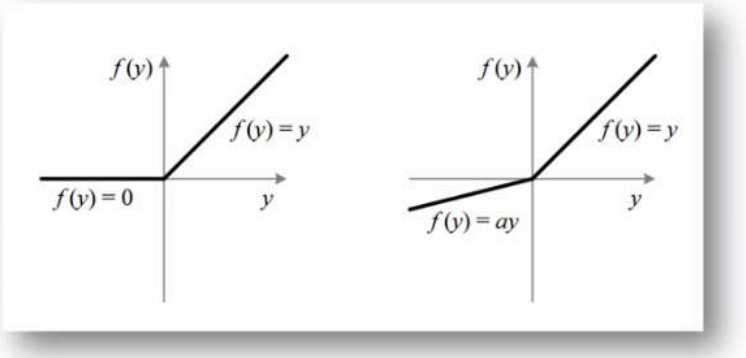
Additional Reading:

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

By Kaiming He et al. (2015)

Link:

https://arxiv.org/pdf/1502.01852.pdf



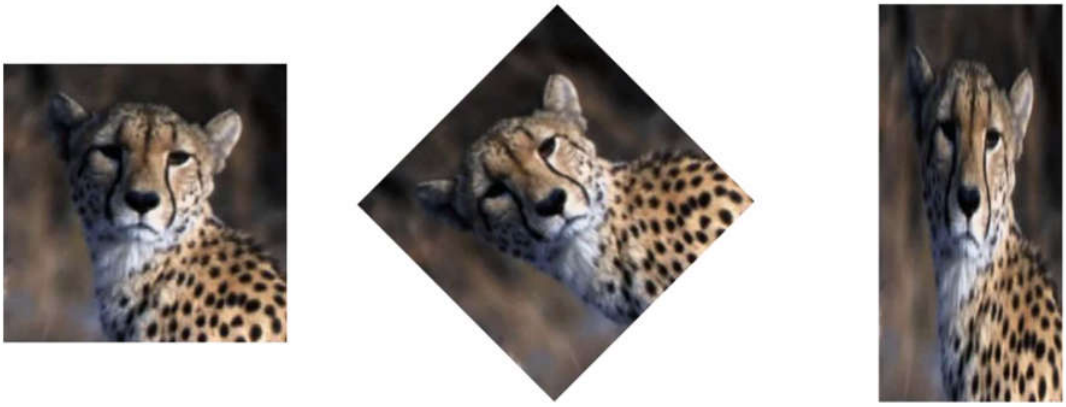
Chapter 9 : Part 3

Deep Learning

CNN: Pooling, flattening & Full Connection

9.3.1 Pooling

Here we'll talk about ***Max pooling***. And what is ***pooling*** and why do we need it. Well to answer that question let's have a look at these following.



❒ We've got a ***cheetah***. Same ***cheetah*** but in 2nd image its ***rotated*** and in 3rd image its a bit ***squashed***. We want the ***neural network*** to be able to recognize the cheetah in every single one of these images.

❒ However, this is just one cheetah. What happens if we have lots of different Images of different Cheetahs?

☞ We want the ***neural network*** to recognize all of these Cheetahs. So how can it do that if they're all looking at ***different directions***, they're in ***different parts*** of the image, they're ***faces are positioned differently*** and many other differences like ***light*** and ***texture*** etc.

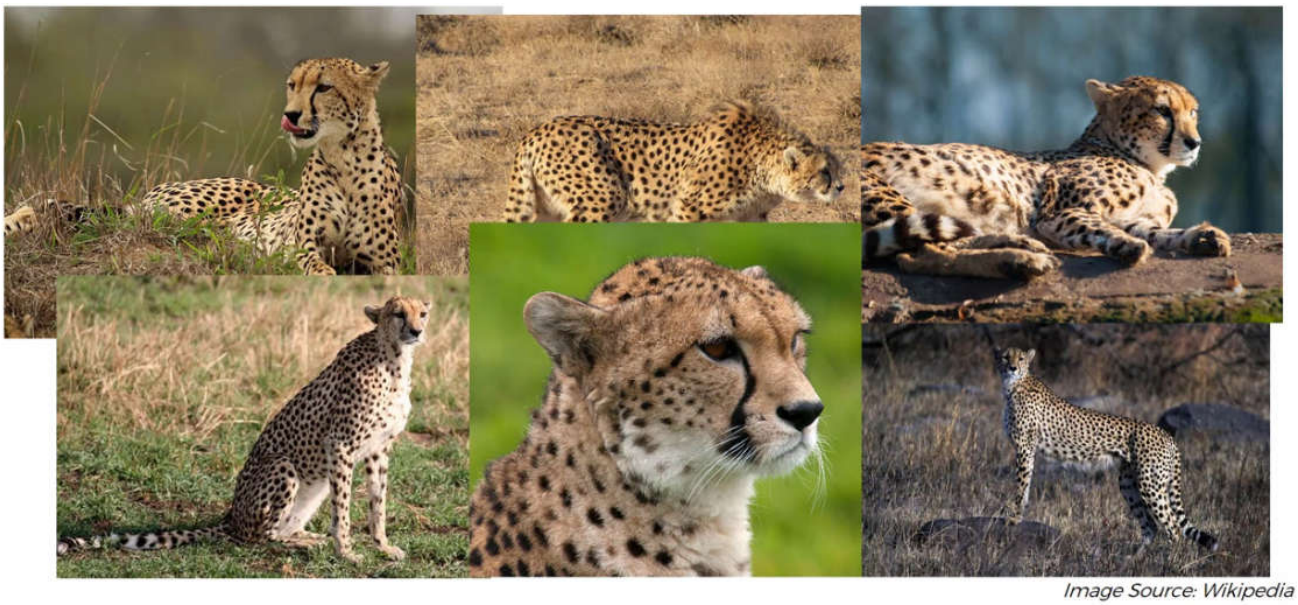


Image Source: Wikipedia

☞ There's lots of little differences and so if the neural network looks for a certain feature for instance: A distinctive feature of the Cheetah is the ***tear-mark*** that are on its ***face*** going from the ***eyes***.

☞ But if NN is looking for that feature which it **learned** from **certain cheetahs** in an ***exact location*** or an ***exact shape*** it will never find these for other Cheetahs.

❒ So we have to make sure that our NN has a property called ***"spatial invariance"*** meaning that it doesn't care where the features are. i.e. the ***NN*** doesn't have to ***care*** if the features are a ***bit tilted***, bit ***different in texture***, if the features are a ***bit closer*** or a ***bit further*** apart relative to each other.

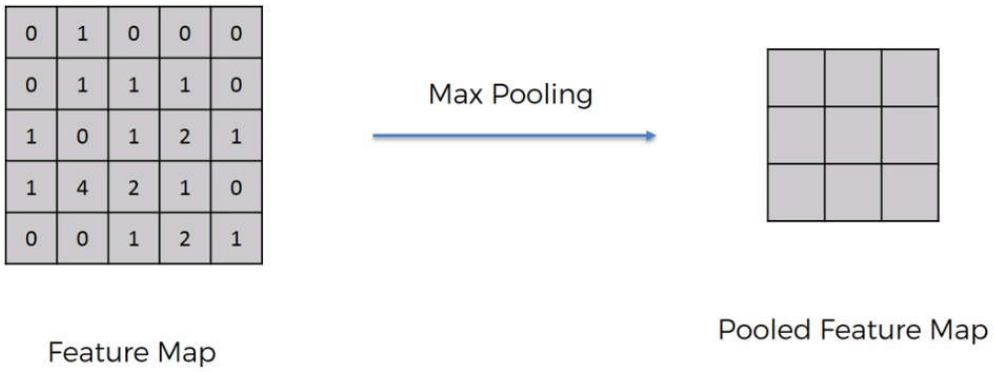
☞ So if the feature itself is a bit ***distorted*** our ***neural network*** has to have some ***level of flexibility*** to be able to ***still find*** that ***feature***. And that is what ***POOLING*** is all about.

9.3.2 Max Pooling

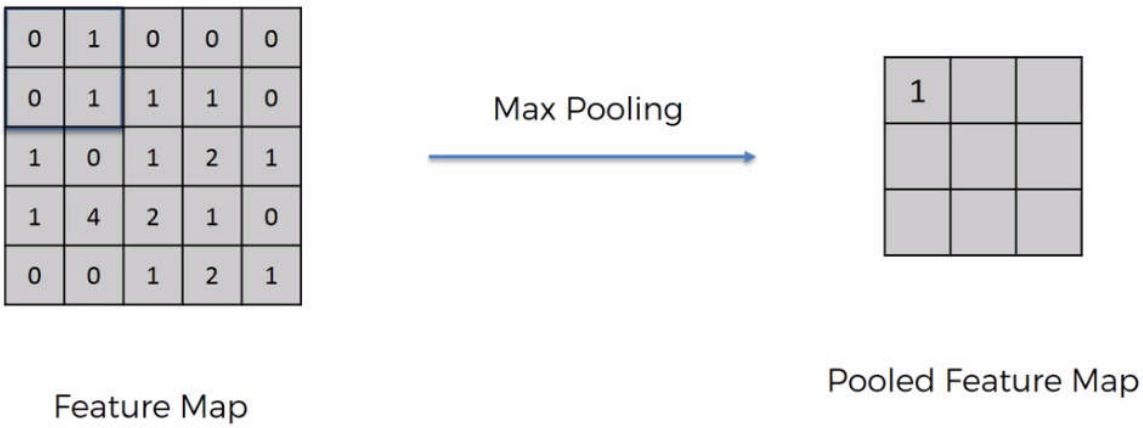
Consider following as our ***feature map*** (already done our convolution). Now we're going to apply ***pooling*** (also called ***Down-Sampling***). There are different types of ***pooling***

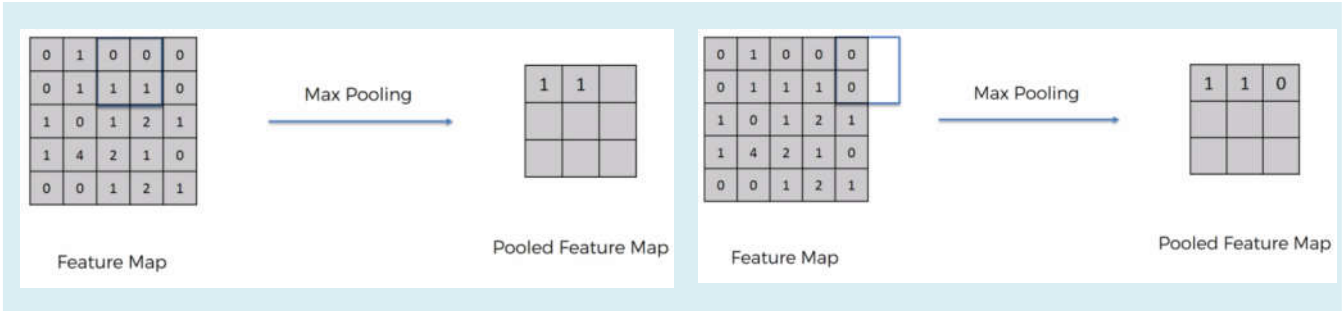
- ***Max pooling:*** The maximum pixel value of the batch is selected.
- ***Min pooling:*** The minimum pixel value of the batch is selected.
- ***Average pooling:*** The average value of all the pixels in the batch is selected.
- ***Sum Pooling:*** This is a variation of the Max pooling. Here, instead of average or max value, the sum of all the pixels in the chosen region is calculated.

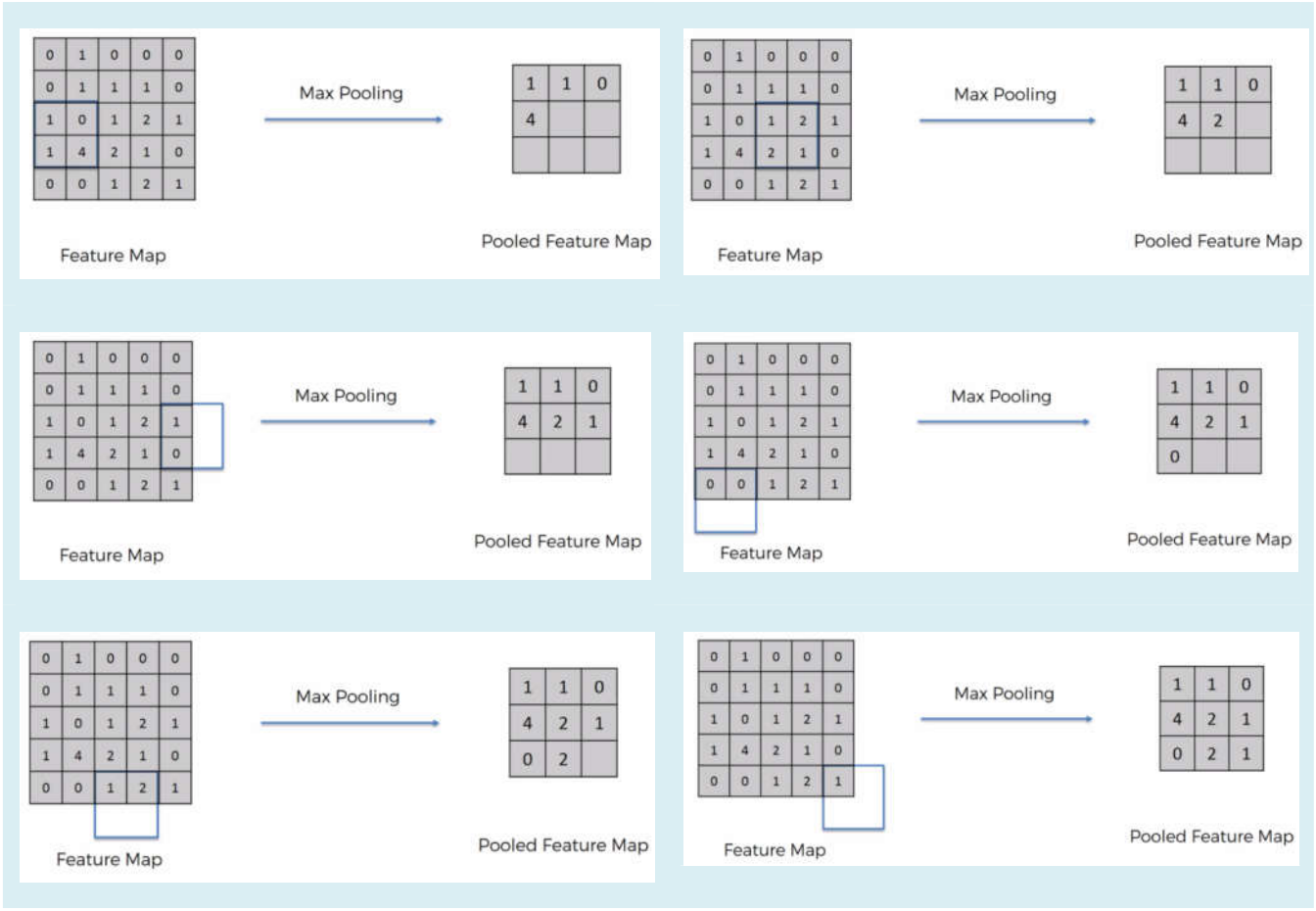
Here we'll apply ***Max-Pooling***.



❒ ***Max- pooling:*** We take a ***box of*** 2×2 ***pixels***. And select ***Stride size*** of ***2*** (if you want overlapping box then select ***stride size*** of ***1***). However, you can choose ***any size of box*** and place it in the ***top left hand corner*** and you ***find*** the ***maximum value*** in that ***box*** and then you record only that max value.







☞ Here we're selecting a ***stride of two*** and this size is ***commonly used***. After repeating the process we record that maxim values shown in above pictures.

☞ ***Here following things happened:***

i. ***Preserving the features:*** First of all we still were able to preserve the features, the **maximum numbers** they represents the features. These ***large numbers*** in your ***feature-map*** represent where you actually found the ***closest similarity*** to a ***feature***.

ii. We're introducing spatial invariants: And because we are taking the ***maximum*** of the ***pixels*** we are gaining ***"spatial invariance"***. For example: ***Cheetah's Tear-mark*** in ***different version*** of the image (***rotated*** and ***squashed***) or ***other position*** can ***end-up*** with same ***Pooled-Feature-Map***.

iii. ***Reducing the size of the data:*** We're reducing the ***size by 75%*** which is really going to help us in terms of ***processing***. By **Pooling** these **features** we are ***getting rid of 75% of the information*** that is not the feature (not related to the feature) by picking ***1px*** out of ***4px*** using ***box of*** 2×2 ***pixels***.

iv. ***Preventing Overfitting:*** Another benefit of pooling is we are ***reducing*** the ***number*** of ***parameters***. We're reducing 75% of parameters that are going to go into our final Layers of the neural network and therefore we're ***preventing Overfitting*** (i.e. model doesn't get dependent on train images).

It is a very important benefit of pooling that we're ***removing information*** because that way our model won't be able to ***Overfit onto*** that ***information***. Because it's important to see ***exactly*** the ***features*** rather than all other ***noise*** that is coming into our eyes. Same thing goes for neural networks by ***eliminating*** the ***unnecessary information*** we're helping with preventing of ***overfitting***.

That's the point of ***pooling*** that we're still being able to **preserve the features** account for their possible ***spatial*** or ***textural*** or other kind of ***distortions***.

❒ ***Why Max pooling (Additional Reading):*** There's lots of different types of pooling and why stride of two and 2×2 ***pixels*** box size. On that note I'd like to introduce you to this lovely research paper called ***"Evaluation Of Pooling Operations In Convolutional Architectures For Object Recognition"*** by ***Dominic Scherer*** from ***University of Bonn***.

☞ They talk about a concept called ***Subsampling*** which is basically ***average pooling***.

☞ In average pooling, instead of taking the ***Max value*** we take the ***average***.

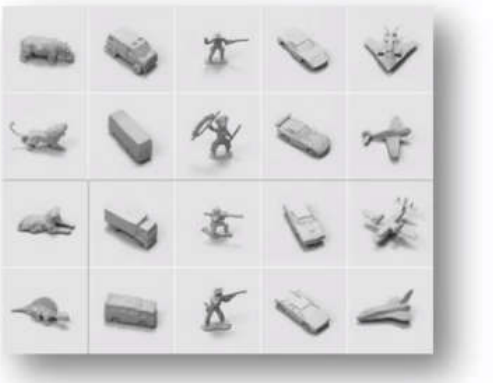
Additional Reading:

Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition

By Dominik Scherer et al. (2010)

Link:

http://ais.uni-bonn.de/papers/icann2010_maxpool.pdf



Let's recap what we have done so far.

☝ On our input image, we applied the ***convolution operation*** and we got the ***convolution-layer*** (collection of ***Feature-Maps***).

☝ And then to each of those **`feature-maps`** we've applied the "**`Max-Poolling`**" and we get a **`Pooling-Layer`**.

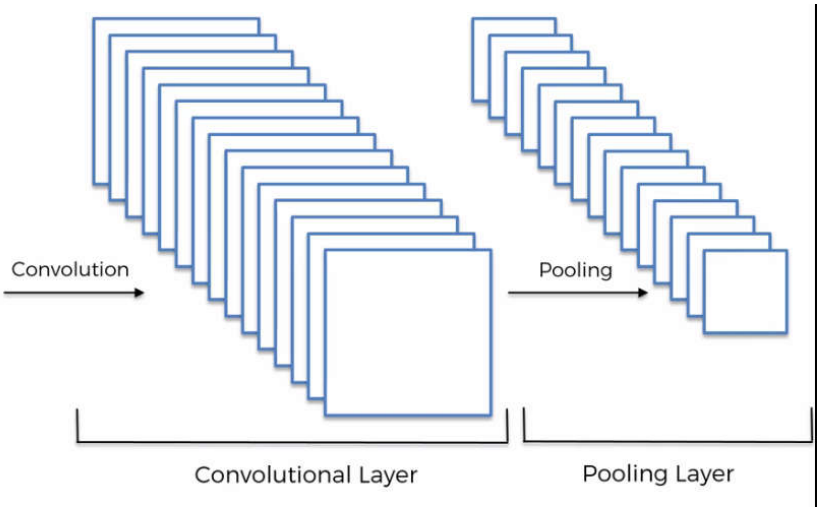
🖐 So we've done these two steps:

[1]. Convolution and

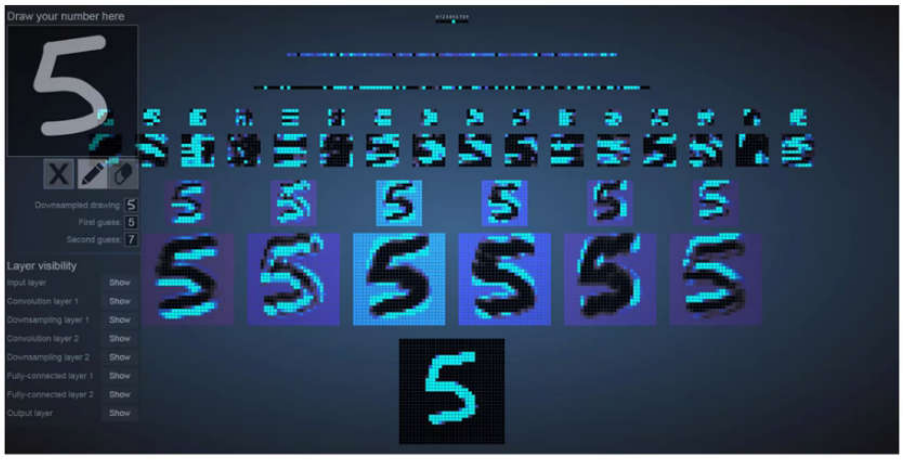
[2]. Pooling

0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

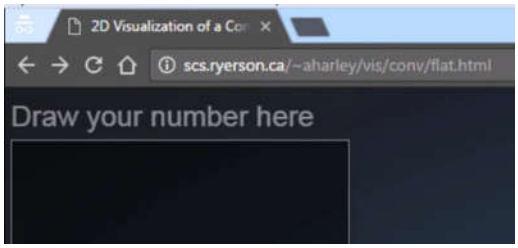
Input Image



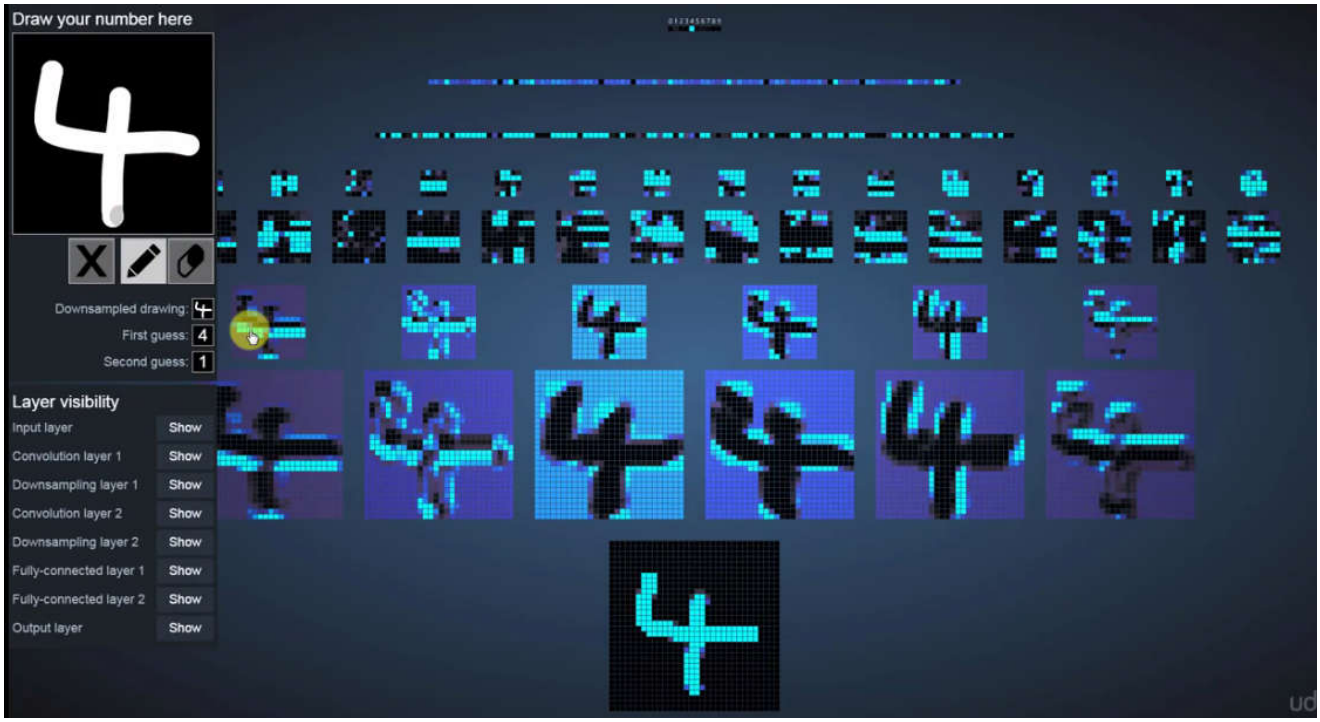
❒ ***Example:*** There is a fun example of CNN. Follow the link (may be deleted now).



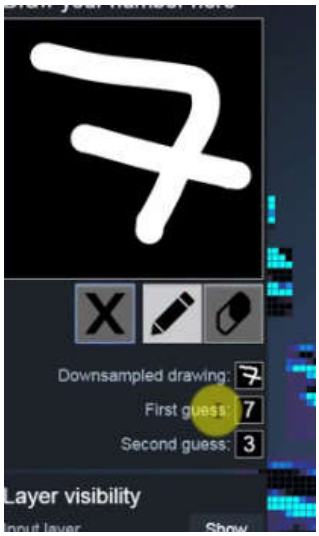
❒ And basically this is exactly what we're doing but we can visualize it here (kind of). So here we draw a number, say we draw number 4, and in this tool will put the number ***four*** at the bottom (that's what we drawn).

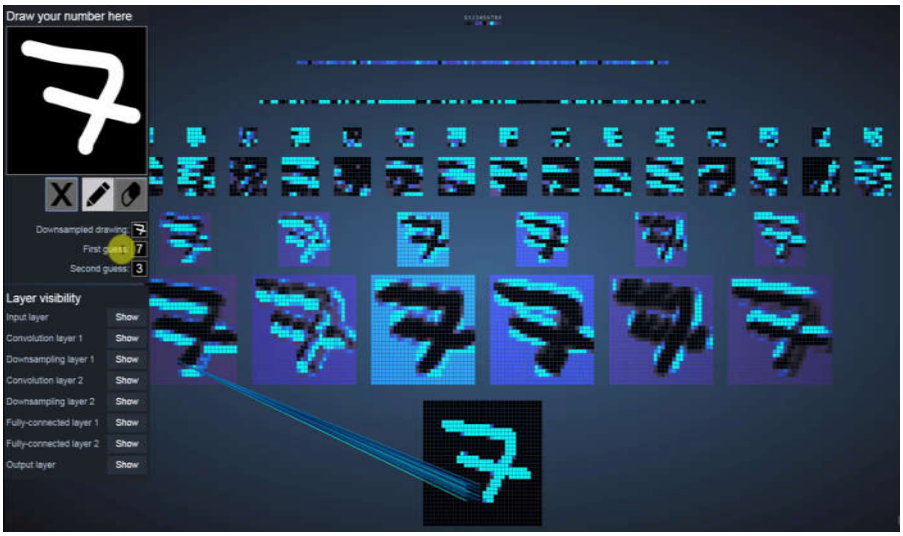


❒ After that, the ***1st*** row is the ***convolution step***, the ***2nd*** row is the ***Polling step*** (also called ***Down-Sampling***). When it's applied ***pooling*** it's ***reducing the size*** and you can see here that ***pooled image*** has ***same features*** that having just ***less information***.

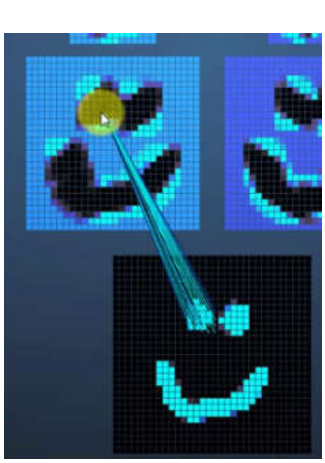


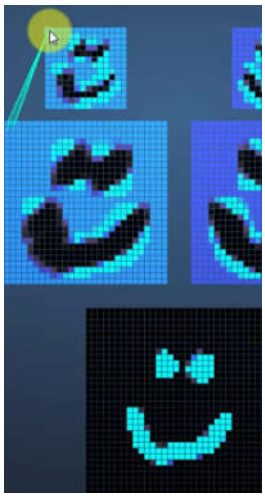
❒ It also gives the 2nd guess.

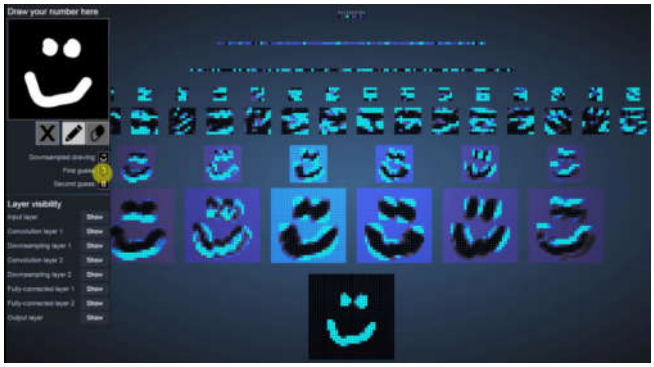




☞ We can also see the pixels of the image how it is reduced. And also we can see the ***Stride size*** (pointy thing). The 3rd row is also ***convolution layer*** of the previous ***Pooling layer***. 4th row is another ***Pooling***. Last two rows are ***flattened-layer***.





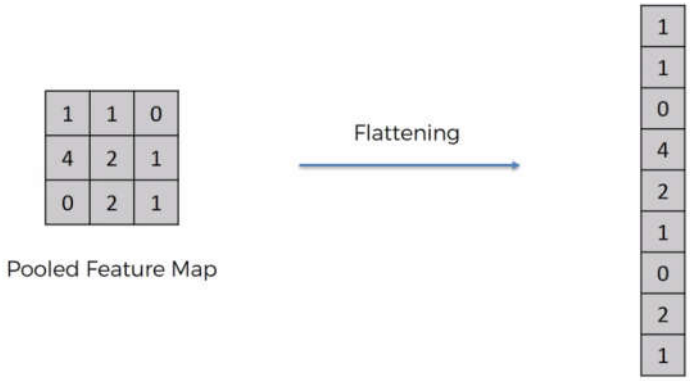


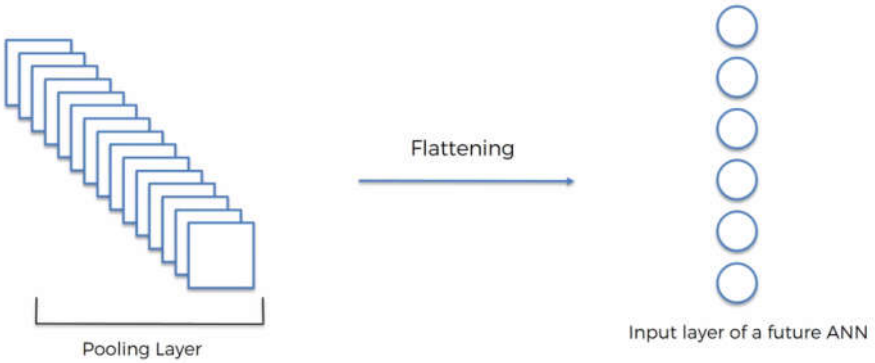
9.3.3 Flattening

After we get the ***Pooled Feature Map*** (in ***Pooled layer***) we're going to ***flatten*** it into a ***column***. Basically just take the values of "***row by row***" and put them into one long column. i.e. we are putting ***2-D matrix*** into ***1-D vector***.

❒ The reason is we want to input this ***vector*** into an ANN for further processing. This ***vector*** will be ***Input Layer*** of the ***ANN***.

☞ When flattened the pooling layers (with multiple Pooled Feature Map) into ***one long column sequentially*** then we end up like following:

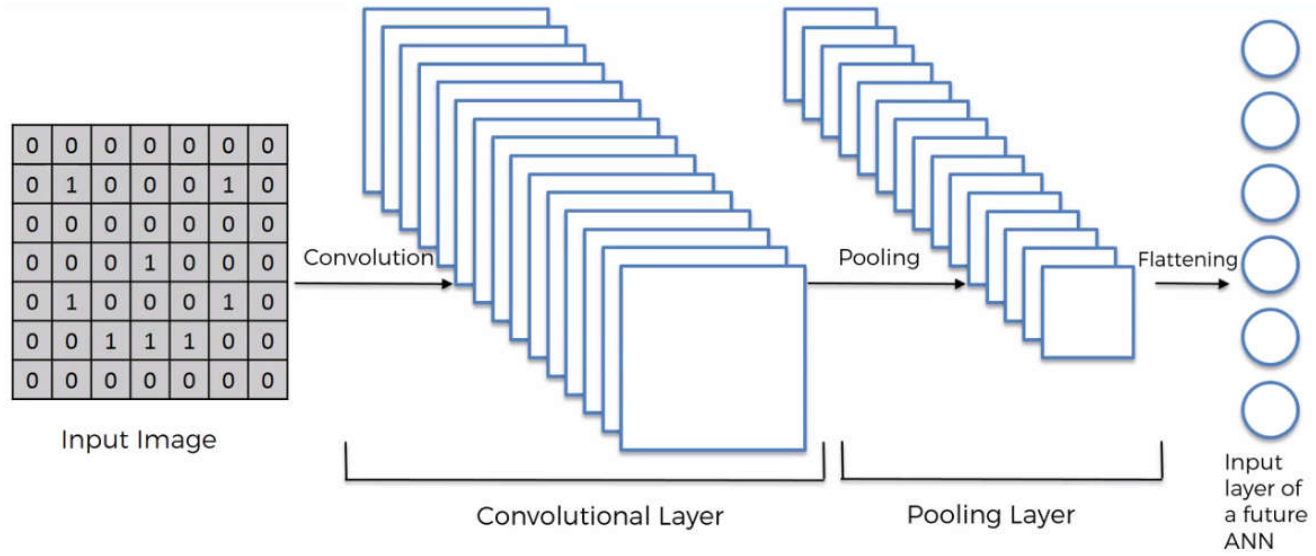




☞ So we put them one after the other and we get **one huge vector of inputs** for an ***Artificial Neural Network***.

<u>So at this point we applied following steps:</u>

[1]. We've got an input image.
[2]. We apply a ***Convolutional-Layer*** and ***ReLU*** function
[3]. Then we apply pooling
[4]. After that we flatten everything into a long vector which will be our ***input layer*** for an ***ANN***.



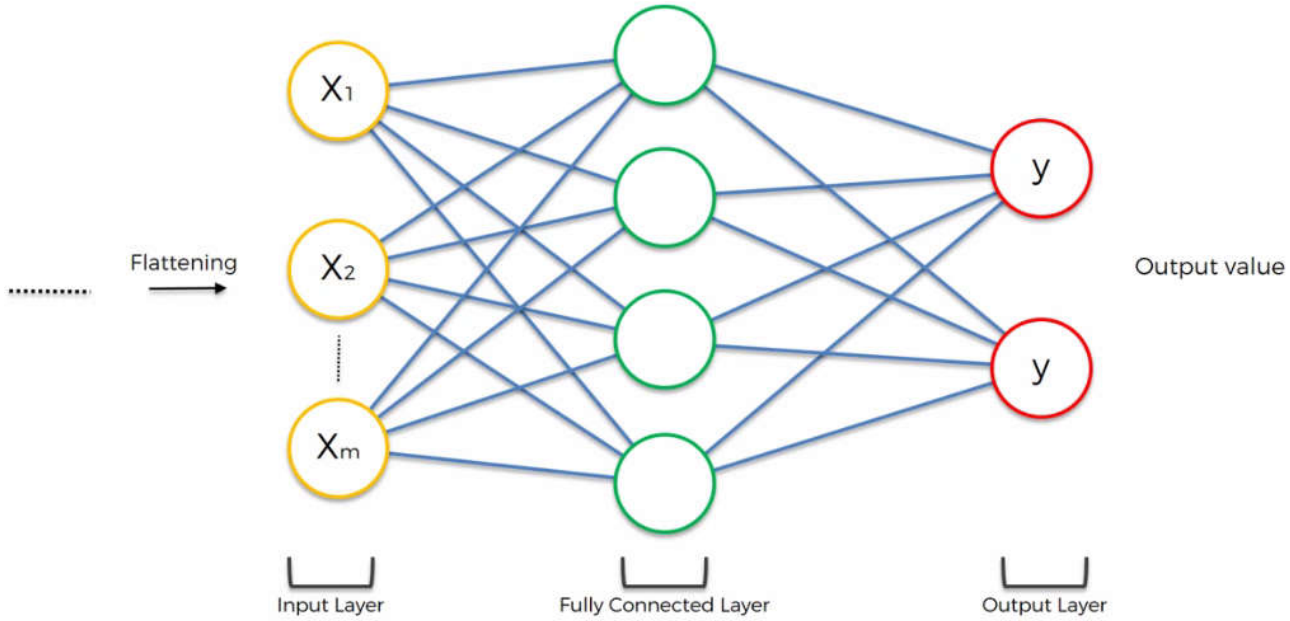
9.3.4 Full Connection

In this step we're adding a whole ***Artificial Neural Network (ANN)*** to our ***Convolutional Neural Network (CNN)***.

❒ So here we've got the ***input-layer***, ***fully-connected-layer***, and ***output-layer***. In ***ANN*** we used to call them as "***Hidden-Layers***" but in ***CNN*** we call them "***fully-connected-layer***"

☞ Because, indeed they are ***hidden layers*** but at the same time they're a more ***specific type of hidden layers*** that are ***fully connected***.

☞ In ***ANN***, ***hidden letters don't need to*** be ***fully connected***. But in ***CNN*** those layers are ***fully connected***, hence called ***fully-connected-layer***.



☞ Above is a very simplified example of ANN. The main purpose of this ANN is to ***combine*** our ***features*** into ***more attributes*** that predict the Classes even better. Here the column or ***vector*** of ***flattened-layer*** is passed into the ***Input Layer***.

☞ In our vector of ***vector*** of ***flattened-layer*** we have some ***features*** encoded in the ***values*** in that ***vector***. Those ***features*** can ***already*** predict what class we're looking at, for instance, whether it's a ***Dog*** or a ***Cat***.

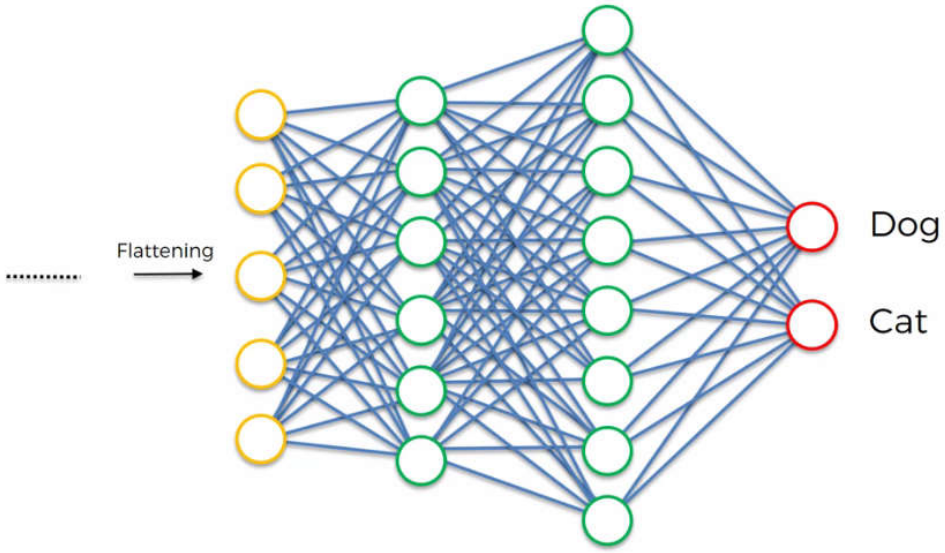
☞ But at the same time the ANN dealing with ***features*** and coming up with new ***attributes*** and combining ***attributes*** together to give even better prediction.

So we pass on those values (***Flattened-Layer Vector***) into an ***ANN*** and let it even further ***optimize*** everything that we're doing.

9.3.5 How Fully-Connected-Layers Work

Let's look at a more realistic example. Here we've got an ANN where we have

i. **five *attributes*** on the ***inputs*** that
ii. we have in the first ***hidden-layer [fully-connected-layer]*** we have ***six neurons***,
iii. in the ***second fully connected layer*** we have ***eight neurons*** and
iv. then we have ***two outputs***, one for ***Dog*** and one for ***Cat***.



❒ ***Why do we have two outputs here:*** Before in ANN we used to have one output.

☞ Note that, ***one output*** is for kind of when you're ***predicting*** a ***numerical value*** and running a ***regression*** type of ***problem***.

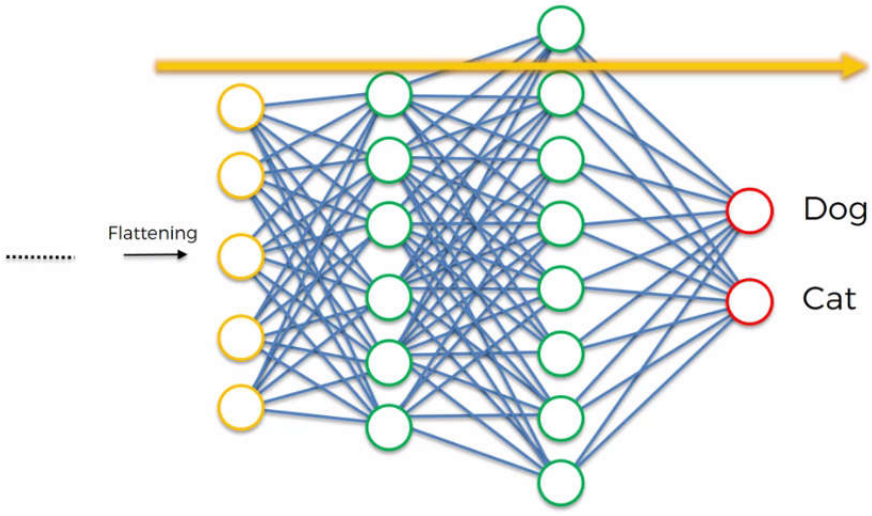
☝ Here we are doing **classification**. In our case we have two classes : ***Cats*** and ***Dogs***. We can do our job by using one output (binary output: ***1*** is a ***dog*** and ***0*** is a ***cat*** and that would have worked totally fine).

☞ But when you're doing classification, where you have ***more than two categories*** for instance ***dogs***, ***cats*** and ***birds*** then you have to have a ***neuron*** for ***each category*** and that's why we're going to ***practice*** with ***two categories***.

❒ So we've already done the *convolution*, *pooling* and *flattening* and now the information is gonna go through the ANN.

a) At the first step the *convolution*, *pooling* and *flattening* happens and the information going through the ANN and a ***prediction*** is made.

b) And for instance let's say, it predicted 80% that given image is a ***Dog's*** image. But it turns out that its actually a ***Cat***.

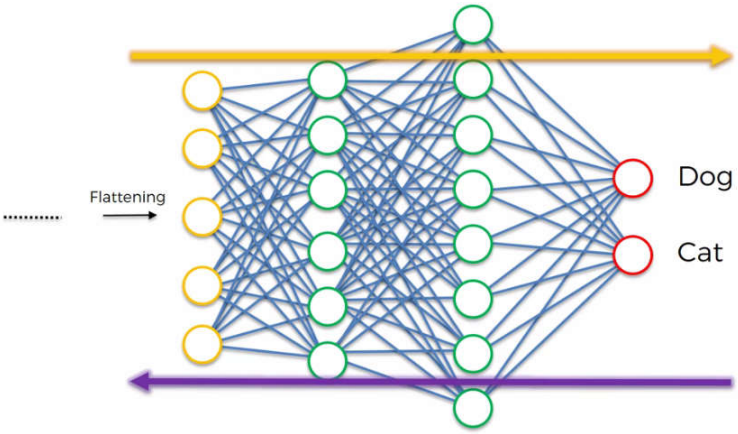


c) Then an error is calculated. "*Cost function*" in ANN, where we used to calculate "*mean square error*". In CNN it's called a "***Loss Function***" and we use a "***Cross Entropy Function***" for that (we will talk about *Cross Entropy Function* later).

➢ This ***Loss Function*** tells us how well our network is performing and we're trying to optimize or minimize that ***Loss Function*** to *optimize* our *Network*.

d) After the error is calculated, it's ***Back Propagated*** through the network just like ANN and couple of things are adjusted:

1. The ***weights*** in the ANN Synapses (blue connected lines) and
2. The ***feature detectors (filters)*** are also adjusted (those little 3×3 matrices that we had in convolution step). It updates the **features** if looking for the **wrong features**. So that next time the NN gets improved. Done with GD (or SGD) and Back-propagation.



e) After Adjusting the ***weights*** and ***feature detectors*** the ***forward-propagation*** happens again. Then ***Error calculation*** and after that again ***Back-propagation***.

☞ This ***processes*** are ***going*** on until our ***CNN is optimized*** and the network gets **trained** on the **data**.

✋ ***Important:*** Note that that the ***data goes through*** the whole ***Network*** from the ***very start*** (i.e ***convolution***, ***pooling*** and ***flattening*** again) to the very end.

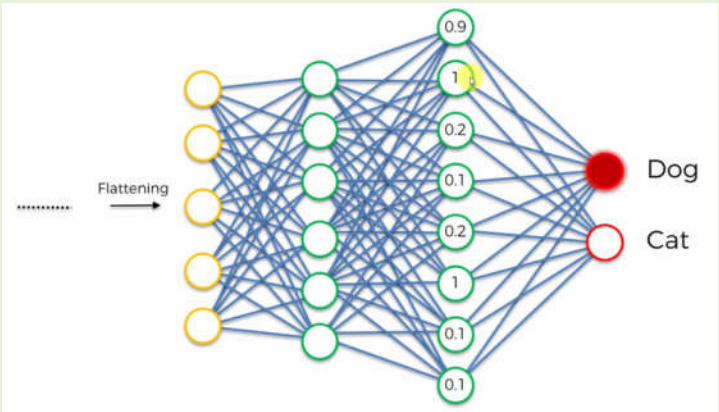
☝ So same thing as ***ANN*** but a bit longer because of that the first three steps ***convolution***, ***pooling*** and ***flattening***.

❒ ***How do these two classes/output neurons work:*** Let's see how does this **image classification** works.

☯ Let's start with the ***top output-neuron*** (indicates ***Dog***).

➲ First of all let's ***assume*** (hypothetically) the **weights** of the **Synapses** that are connected to **Dog**. These numbers can be absolutely anything we are just doing this for learning purpose. Sot that, we can say which of the previous-layer neurons are actually important for the ***Dog***.

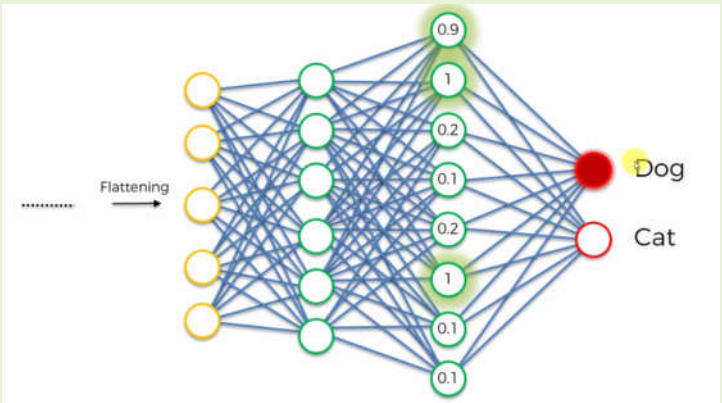
So let's say we've got **above numbers** in our ***previous*** (final) ***fully connected layer***. Those can be any numbers but just for argument's sake we're assuming these numbers between 0 and 1 i.e. in [0, 1] interval.

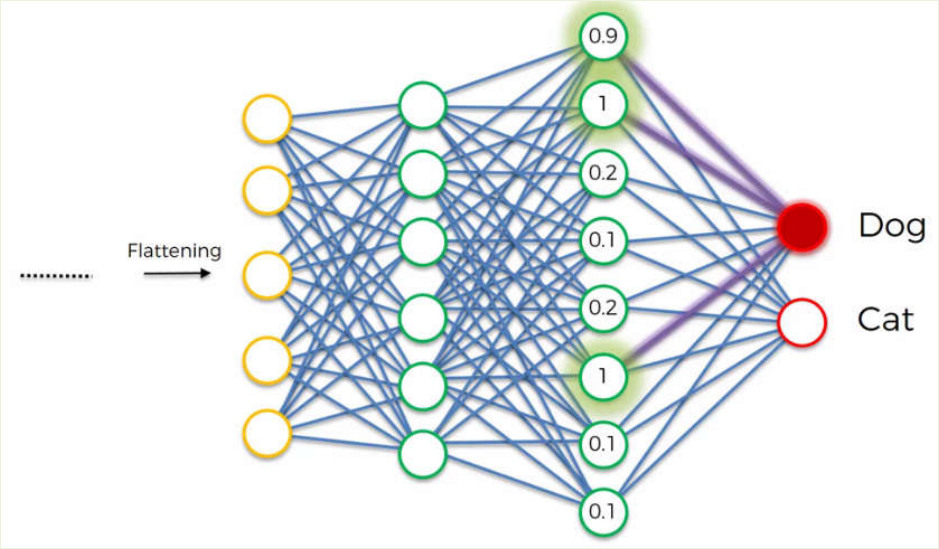


➲ These ***neurons*** of **final- fully connected layer** are actually finding the ***features***. And ***0*** means that that neuron **didn't find** a feature is looking for. so because at the end of the day these neurons like anything else on this from on this left side

➲ Even though These ***neurons are*** already processed, but still they're detecting a **certain feature** or a **combination of features** on the image.

For example following glowing neurons are for features of dogs ***nose***, ***floppy ears*** and ***eyebrows***. So these neurons do indeed fire up when the feature belongs to a dog.





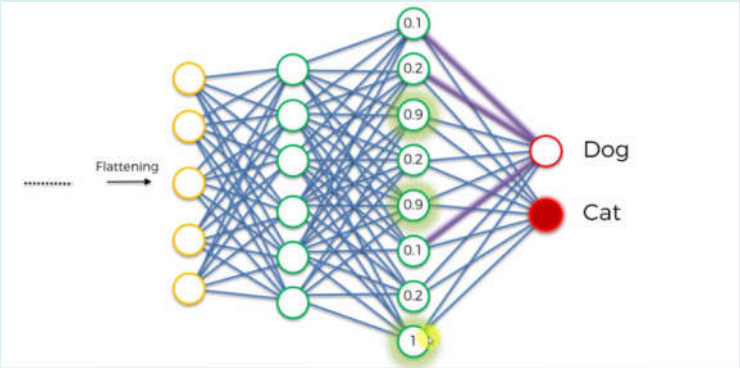
- On the other hand the ***Cat neuron*** will know that it's not a ***Cat***. Even though it got also ***floppy ears*** (***one neuron*** with ***1***) and ***Cat neuron*** will ignore this neuron.
- Through lots and lots of iterations if this happens often. The ***Cat neuron*** will ignore this neuron more (Synapse with ***Cat-Neuron*** will be weak. i.e. weight ***gets*** smaller). And ***Dog-neuron*** will pick up those ***3 neurons***. It will start attributing *higher importance/weight* to those *3 neuron*.
- Hence we're going to say that this ***Dog neuron*** learned that these three neurons ***(eyebrow neuron*** and ***big nose neuron*** and ***floppy ear neuron) are*** contribute to the ***classification*** of ***Dog*** and ***Cat*** through this iterative process with many ***samples*** & ***epochs***.

Important: One thing to note that the ***signals*** from those ***3-neurons*** are going actually to both ***Cat-Neuron*** and ***Dog-neuron*** simultaneously.

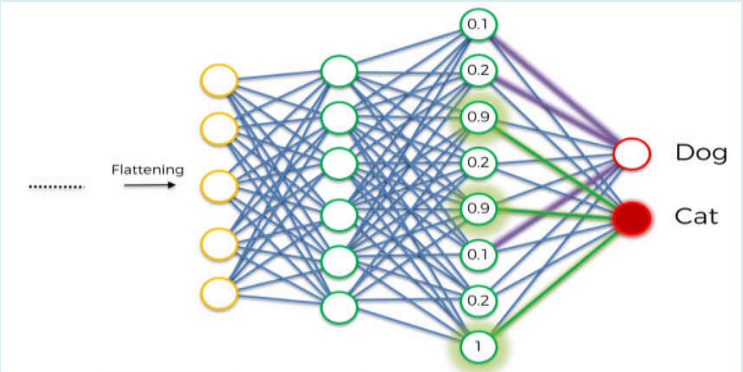
Remember, these ***ears*** and ***nose*** and ***eyebrows*** those are very approximate (because of going through ***Convolution***, ***Pooling***, ***Flatenning***) or ***unrecognizable*** what they're looking for but at the same time it is ***something*** in the ***features*** of ***dogs***.

Now let's move on to ***Cat-neuron***. But those 3-Synapses (how we've sorted out the dog using these weights) of ***eyebrow neuron*** and ***big nose neuron*** and ***floppy ear neuron*** are remembered.

- Now how is the ***Cat-neuron*** works? Well whenever it is actually a cat, there will be ***different values/numbers*** for the ***last fully-connected-layer-neurons*** for different neurons. Here let's consider again ***3 different neurons***.



- Notice these three neurons ***0.9***, ***0.9*** and ***1***. They're ***interacting*** to both the ***Dog*** and the ***Cat*** (remember that the ***Signals*** from those 3-neurons are going actually to both ***Cat-Neuron*** and ***Dog-Neuron*** simultaneously).



Important: This is again important to remember that those **output signal** goes both ways to ***Cat-Neuron*** and ***Dog-Neuron***.

Remember: It's up to the ***Cat-Neuron*** and ***Dog-Neuron*** to decide whether to take into account that signal and learn from it or not (using Synapse weights).

And ***Cat-Neuron*** and ***Dog-Neuron*** analyze the given photo/image of Cat/Dog.

- So basically the ***Dog-Neuron*** will recognize Cats ***whiskers*** and Cats ***pointy triangle ears*** and Cats ***vertical eye pupil***. It will recognize because every time these neurons fired up the prediction is ***not Dog***.
- On the other hand the ***Cat-Neuron*** recognize those three neurons (***whiskers*** , ***pointy triangle ears*** , ***vertical eye pupil***) because most of the time these three lights up it matches ***Cat-Neuron's*** expectation for Cat.
- ***Cat-Neuron*** is going to listen to those three neurons more and more. So basically it's listening to these three and it's ignoring the other five.

And that is how these final neurons (***Cat-Neuron*** and ***Dog-Neuron***) learn: *"which neurons in the **Final Fully Connected Layer** to listen"*.

- That's how the features are propagated through the network and conveyed to the **output**.

In reality these ***features*** don't have that ***much meaning*** to them like ***floppy ears*** or ***whiskers*** but they do have some ***distinctive feature*** of that specific ***class***.

That's how the network is trained because during the ***back propagation***, we're not just adjust the ***weights*** but also the ***feature-detectors***.

If a **feature** is **useless to the output**, it's going to be **disregarded** and ***replaced*** with ***feature*** which is ***useful*** because this happens through thousands and thousands of iterations.

So at the end in this ***Final Fully Connected Layer*** of neurons have lots of ***features*** or ***combinations of features*** from the image that are indeed ***representative*** or ***descriptive*** of ***dogs*** and ***cats***.

How prediction/recognition occurs : Once your network is trained up, let's say we pass on an ***image of a Dog***. The values are propagated through a network we get certain values.

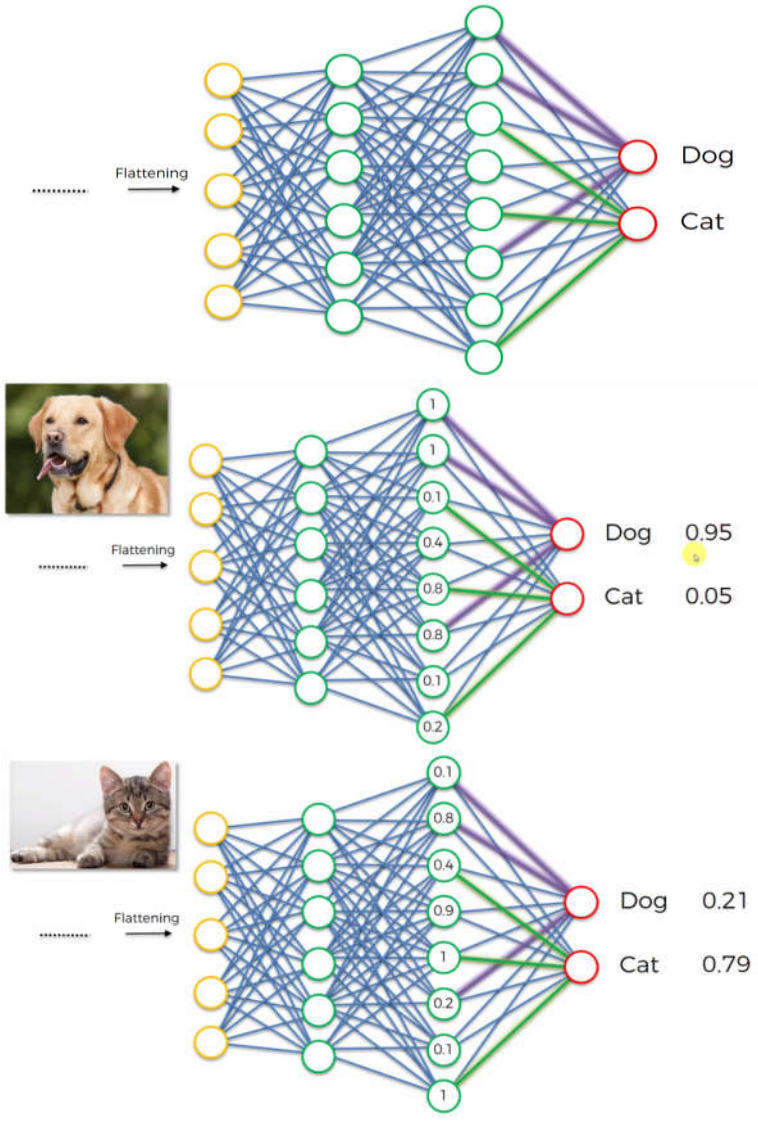
- ***Dog*** and the ***Cat Neurons*** don't know that it's a **dog** or a **cat**. But they have **learned to listen** to what is being shown here. ***Dog-Neuron*** listens to these three neurons (**Purple Synapse**) a ***Cat-Neuron*** listens to other three neuron (**Green Synapse**) for features.
- ***Purple weights*** are how the ***Dog neuron*** views their votes. How much importance is it assigns to these neurons and those votes. And ***Green weights*** are how much importance the ***Cat's neuron***.

A. So when the Dog neuron ***founds*** its ***corresponding feature neurons*** pretty ***high*** then it give a ***high probability*** that *"given image is probably a dog"*.

B. And Cat neuron ***founds*** its ***corresponding feature neurons*** very ***low***. Then it give a ***low probability*** that *"given image is a Cat"*.

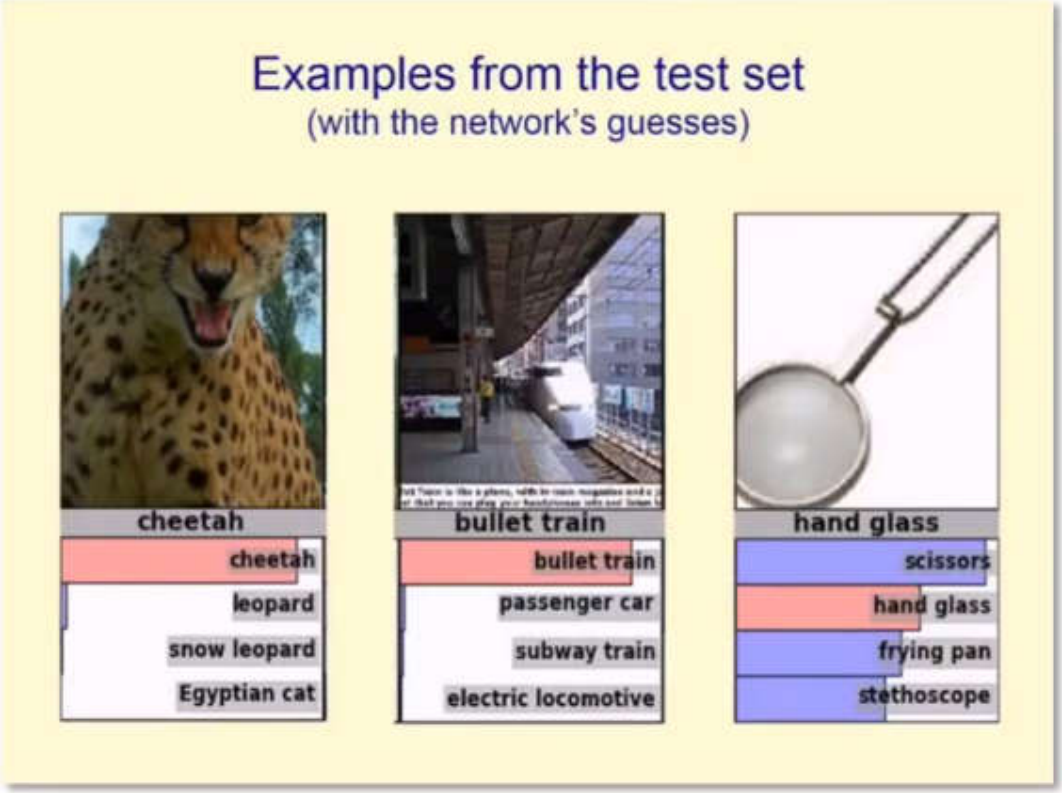
And that's how we you get our prediction. So the answer is Dog. NN is voted ***high*** for a ***Dog***. ***Voting*** is a term that is used for ***neurons*** in the ***Final Fully Connected Layer***. The corresponding ***weights*** are the ***importance*** of their ***vote***.

C. Same thing happens when you pass an ***Image Of A Cat***.



So these neurons vote the ***Dog*** and the ***Cat*** based on their ***learned weights*** for ***Final Fully Connected Layer***, then they make their ***predictions***.

And that's how you get images like this where you have a ***Cheetah*** (voted high). And we can see the ***voting is different*** for ***different images***, because corresponding CNN recognize differently.



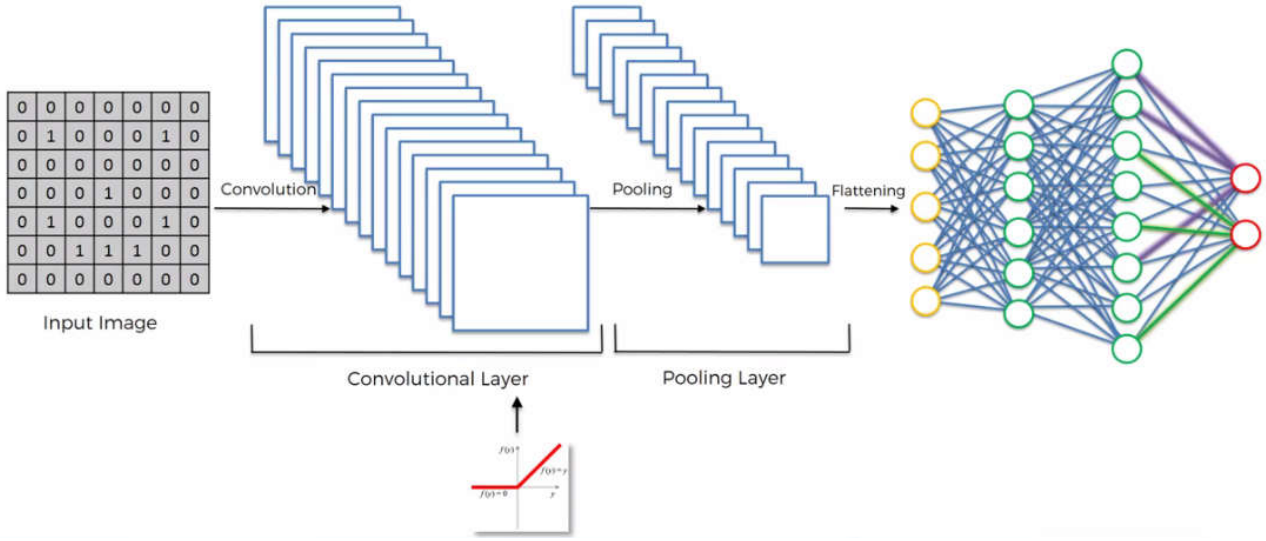
So that's how the ***Full Connection*** works.

Chapter 9 : Part 4

Deep Learning

CNN: Softmax and Cross-Entropy & Summery

9.4.1 Summery



❒ Let's summarize what we've talked about.

- ***Step-1:*** We started with an *input image*. On the *input image* we applied multiple different ***Feature-Detectors /Filters*** to create these ***feature maps***. And this is our ***Convolutional Layer***.
  - Then on top of that ***Convolutional Layer*** we applied the ReLU to *remove linearity* or *increase non-linearity* in our images.

- ***Step-2:*** Then we applied a ***Pooling Layer*** to our ***Convolutional Layer***. From every single ***Feature-Map*** we created a ***Pooled-Feature-Map***.
  - The main purpose of the ***Pooling Layer*** is to make sure that we have ***Spatial Invariants*** in our images.
  - So basically if an ***object*** in an image *tilts* or *twists* or is a *bit different* to the **ideal scenario** then we can **still pick up** that objects ***feature***
  - Also ***Pooling*** significantly *reduces the* size of our images.
  - Moreover ***pooling*** helps with ***avoiding*** any kind of ***overfitting*** of our data or overall by getting rid of a lot of that data.
  - But at the same time ***Pooling*** preserves the ***main features*** that we're after just because ***Max pooling*** is used.

- ***Step-3:*** Then we ***flattened*** all of the ***pooled images*** into one ***long vector*** or ***column*** of all of these values and we put that into an ANN.

- ***Step-4:*** We then introduce a *Fully Connected Artificial Neural Network* where all of these features are processed through a NN.
  - Then we have ***Final Fully Connected Layer*** which performs the ***voting*** towards the ***classes*** that we're after and then all of this is *trained* through a *forward propagation* and *back propagation* process. With lots of ***iterations*** and ***Epochs*** and in the end we have a very well defined ***neural network***.
  - Another important thing is not only the ***weights*** are trained in ANN part but also the ***feature detectors/filters*** are trained.
    - Both are adjusted in that same ***Gradient Decent*** process and that allows us to come up with the ***best feature maps***.

In the end we get a ***fully trained CNN*** which can recognize images and classify them. That's how **Convolutional Neural Networks** work.

9.4.2 Additional Reading on CNN

If you'd like to do some additional reading then there's a great blog by ***Adit Deshpande*** from 2016. So the blog is called "**The Nine deep learning papers you need to know about (understanding CNN's part 3)**".

❒ This blog actually gives you a short overview of ***9 different CNN's*** that have been created by people like you and others which you can then go ahead and study further.

☞ Just keep this blog in mind are these nine papers maybe after the ***practical tutorials*** or maybe after you do some ***additional training*** in the space of ***deep learning*** slowly you can then ***reference*** these works.

☞ You will get a lot of value by looking through other people's NNs and how they structured. Their CNN will help you understand ***what are the best practices*** and why people did certain things in a certain way and that will ***help you with your architecture of NN*** because ***ANN*** and ***CNN*** are not an exception.

☞ They are like an ***architecture challenge***, you have to come up with an ***idea*** and then ***structure*** and then ***adjust*** it and ***tweak*** it to get the ***best possible design*** and the ***best possible*** and ***optimal performance***.

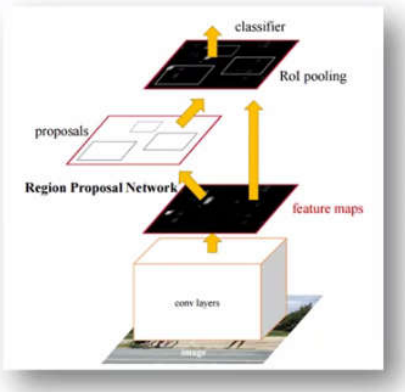
Additional Reading:

The 9 Deep Learning Papers You Need To Know About (Understanding CNNs Part 3)

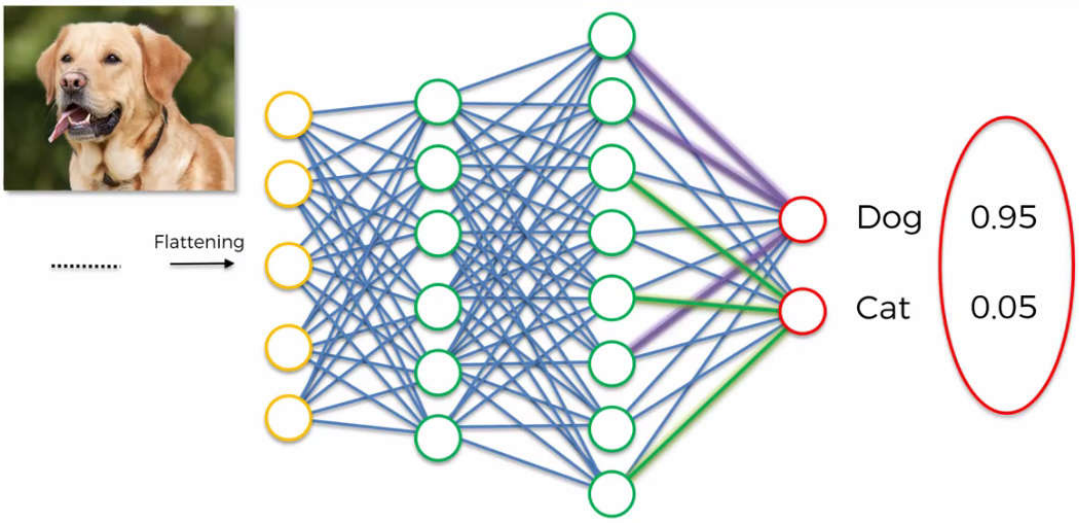
Adit Deshpande (2016)

Link:

https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html



9.4.3 SoftMax function



❒ Here is the CNN that we built to ***Recognize Cat/Dog*** in a given image. Now notice the output ***probabilities 0.05 for Cat*** and ***0.95 for Dog***.

👽 ***Now the question is:*** How the ***sum*** of these two values is ***1***. (in case of more than two class **Dog**, **Cat** & **Bird** this Sum of **output probabilities** is always 1).

☝ Because as far as we know from ANN there is nothing to say that these two ***output-neurons*** are connected between each other.

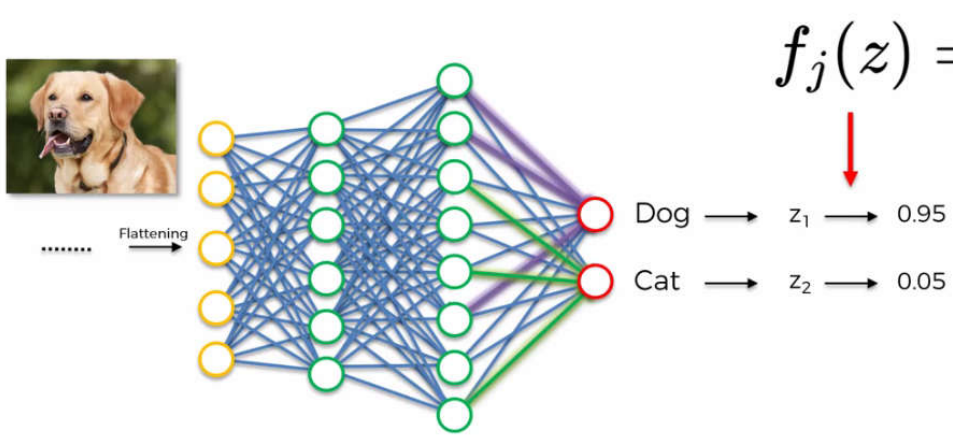
So how would they know about the value that other one is holding? And how would they know to add their probability-values up to 1.

☝ ***The answer is :*** They wouldn't in the classic version of our ANN.

❒ ***SoftMax:*** The only way that they would know about those values if we use a *special function* called the ***SoftMax function***.

☞ So normally the **Dog** and the **Cat Neurons** would have any kind of real values that their sum don't have to be ***1***. Say these values are z_1 and z_2. And $z_1 + z_2 \neq 1$.

☞ Then we would apply the ***SoftMax*** function upon z_1 and z_2 so that the sum of the output becomes 1.



☞ Here the ***SoftMax function*** or the *Normalized Exponential Function* is a ***Generalization*** of the ***Logistic Function*** that ***Squashes*** a **k-dimensional vector of arbitrary real values** to a *k-dimensional vector of real values in the range of [0, 1]* that add up to 1. i.e. for example ***(23, 45, 79, 45)*** becomes something like ***(0.1, 0.25, 0.4, 0.25)***, it ***normalize*** the ***vector*** in a way so that the sum of the ***elements*** of the vector becomes ***1***.

☞ The *elements* of the *vector* in our case the *values* returned by *Cat-Neuron* and *Dog-neuron*. ***SoftMax function*** brings these values to be in range ***[0, 1]*** and make sure that they add up to ***1***.

❒ ***Why SoftMax function is used in CNN:*** It makes sense to introduce the ***SoftMax function*** into CNN because it is strange that a probability of being a dog is 95% and also probability of being a cat is 65%.

Therefore it's much better when you use ***SoftMax function*** to CNN

9.4.4 Cross-Entropy function

The ***SoftMax function*** is mostly used with the ***Cross Entropy function***. The ***Cross Entropy function*** looks like following:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

We're actually going to be using a different representation of the ***Cross Entropy function***. that looks like follows:

$$H(p,q) = -\sum_x p(x)\,\log q(x)$$

The results are basically the same. This is just easier to calculate.

❒ ***Cross Entropy function:*** In ANN we had a function called the *Mean Squared Error (MSE) Function* which we used as the ***Cost Function*** and our goal was to ***minimize*** the **MSE** in order to **optimize** our **network performance**.

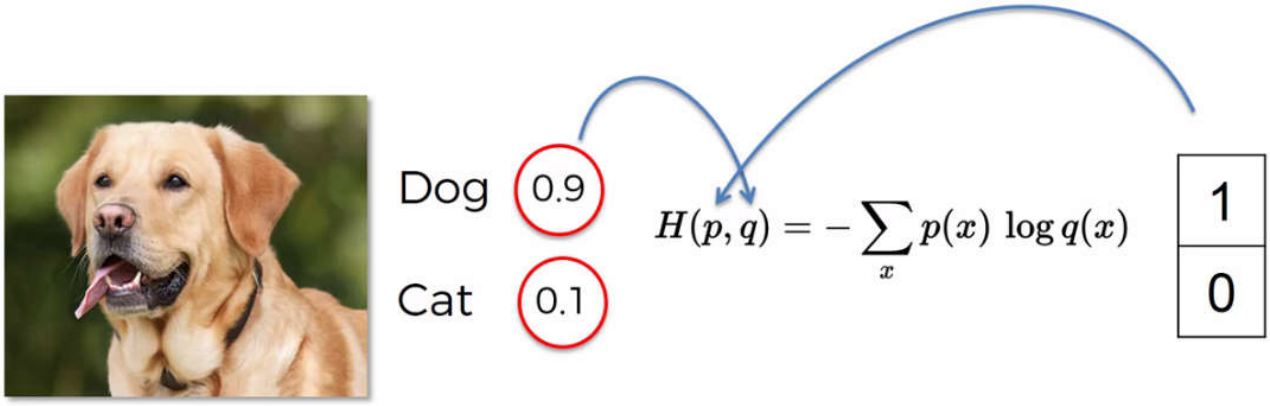
☞ Now in CNN we can still use ***MSE*** but after applying the ***Soft-Max function*** we better use *Cross Entropy function* because here "*log*" is applied and we can work with very *small numerical values*.

☝ Note that, in ***CNN*** we call "***Loss function***" instead of "***Cost function***". "***Loss function***" are "***Cost function***" not same but they are very similar. Here we want to ***minimize Loss function*** in order to ***maximize*** the ***performance*** of our ***network***.

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

$$H(p,q) = -\sum_x p(x)\,\log q(x)$$

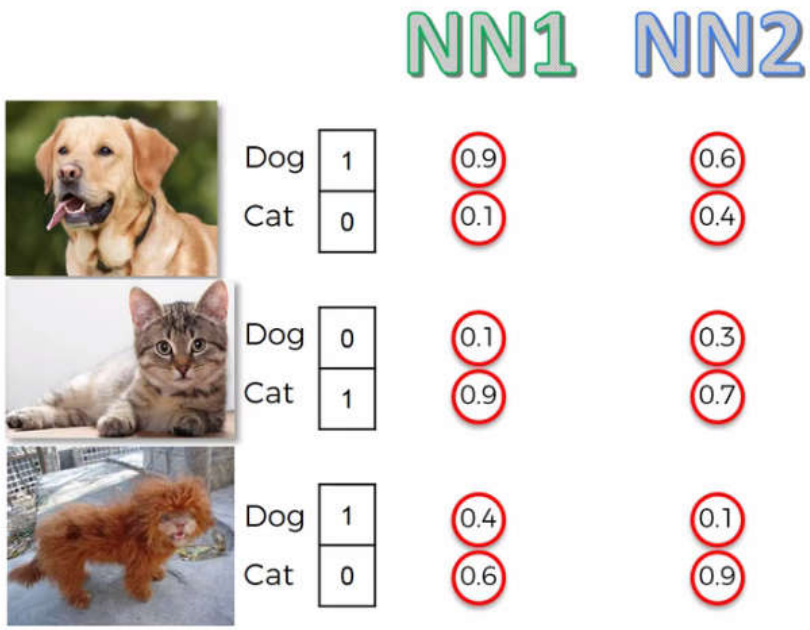
Example: Let's see an example on how ***Cross Entropy function*** can be ***applied***: Here ***p*** is the value of the range i.e. ***1*** or ***0***. And we put the ***output-probability*** into ***q***. The figure shown above.



9.4.5 Cross Entropy function calculates NNs performance more accurately

Now we consider ***two neural networks*** and then we pass ***three images*** of a ***Dog,*** a ***Cat*** and another animal (which is actually another dog if you look closely, but difficult to recognize). Those NNs doesn't know abut those animals.

- In following figure the boxed ***0*** and ***1*** represents the ***actual values*** of being ***Cat/Dog***. So we want to see the performances of these following two NNs.
- Below the values inside the ***"Red-Circle"*** are the ***predicted values*** from these two NNs.



- So the ***key*** here is that even though both NNs got it **wrong** in the **last case**, but ***NN1*** shows ***better performance*** than ***NN2***. Because in the last case NN1 gave a dog 40% vote where NN2 gave 10% vote.

❒ ***Different functions to measure performance of NNs:*** Now we're going to look at the different functions that they can measure performance of NN1 and NN2

- ☑ In following tables: "***Dog^***" and "***Cat^***" columns represents the ***predicted*** values. And "***Dog***" and "***Cat***" columns represents the ***actual*** values.
- ☑ Notice that even though ***NN2*** was ***correct*** for ***first 2 trials*** but its ***performance*** were ***poor***.
  - ***NN2*** votes ***60%*** for Dog where NN1 votes ***90%*** in ***first trial***.
  - Also ***NN2*** votes ***70%*** for Cat where NN1 votes ***90%*** in ***second trial***.
  - In the ***last trial*** both ***NN1***, ***NN2*** are incorrect, but NN1's performance was better in that trial. In this case ***NN1*** gave a dog ***40%*** vote where ***NN2*** gave ***10%*** vote.

☞ And so now let's see what kind errors we can calculate to ***estimate*** the ***performance*** and ***monitor*** the ***performance of our NNs***.

NN1

Row	Dog^	Cat^	Dog	Cat
#1	0.9	0.1	1	0
#2	0.1	0.9	0	1
#3	0.4	0.6	1	0

NN2

Row	Dog^	Cat^	Dog	Cat
#1	0.6	0.4	1	0
#2	0.3	0.7	0	1
#3	0.1	0.9	1	0

Classification Error

1/3 = 0.33 | 1/3 = 0.33

Mean Squared Error

0.25 | 0.71

Cross-Entropy

0.38 | 1.06

🛠 ***Classification error:*** It is basically just asking it "Did you get it right or not". So for both NNs we got ***1 incorrect*** out of ***3 trials*** (animals). So for both NNs we got ***1/3 = 0.33***. So in this case we cannot find the difference between the performance of the NNs (butt we know NN1 performed better in the last case).

❍ So in case of ***Classification error***, both NNs perform at the same level (but we know that's not true.)

☝ That's why a **classification error** is ***not a good measure*** especially for the purposes of ***Back Propagation***.

🛠 ***Mean Square Error (MSE):*** Basically take the ***sum of squared errors*** and then just take the ***average*** across your ***observations***.

❍ Here ***NN1*** gets **25% error rate** and ***NN2*** gets **71% error rate** (because even though NN2 was correct for first 2 trials but its performance were poor, also for 3rd trial NN2's performed poorly).

☝ So we can see ***Mean Square Error*** is more accurate than ***Classification error***. ***MSE*** telling us that NN1 has a much lower error rate than NN2.

🛠 ***Cross Entropy:*** Cross Entropy gives ***error rate 0.38*** for ***NN1*** and ***1.06*** for ***NN2***. We can see that the results are a bit different.

Why would you use Cross Entropy over MSE

There's several advantages of Cross Entropy over MSE. For instance:

✋ If your ***output value*** is ***too small*** at the very ***start*** of your ***back propagation***. Then at the ***very START*** the ***Gradient*** in your ***Gradient-Decent*** will be very small and it ***won't be enough*** for our NN to ***start adjusting*** the ***weights*** and ***propagating*** in the ***right direction***.

☞ But in case of ***Cross Entropy***, there is ***logarithm*** in it. It actually helps the NNs to work with a ***very small error*** to adjust ***Weights*** and ***propagate*** in the ***right direction***.

i.e. for a little error decrease like ***0.0000001***, NN will ***adjust weights*** and Propagate. So ***NNs*** will detect ***very tiny improvement*** and works on it. This is possible because there is "***Logarithm***" in the ***Cross-Entropy***.

✋ So ***CROSS ENTROPY*** will help your ***neural network*** get to the ***optimal state*** more accurately than ***MSE***. In CNN it is important because we applying ***SoftMax-function***, which normalizes results in between ***[0, 1]***, and hence we have to deal with ***very small numbers***. Thus ***Cross-Entropy*** is a better option.

☞ Actually ***Cross-Entropy*** will improve your network significantly so that that jump from 1000000 to 1000 in MSE these jump will be very low. In that case MSE won't guide your gradient boosting process or your back propagation in the right direction.

☞ Even if MSE guide the NN into right direction, but it'll be like a very slow guidance it won't have enough power. But if you do ***Cross-Entropy*** , it will understand that even though these are ***very small adjustments*** and a ***tiny changes in absolute terms***, but in ***relative*** terms it's a ***huge improvement***. Because we have used SoftMax before.

NOTE:

📢 ***Important:*** It is important to note that, ***CROSS ENTROPY*** is only the ***preferred*** method for ***classification***. But it we deal with ***Regression*** which we had in ANN then ***MSE*** is a ***Better option***.

➲ ***`Cross entropy`*** is better for ***`classification`*** and ***MSE*** is better for ***`regression`***.

9.4.6 Additional Reading

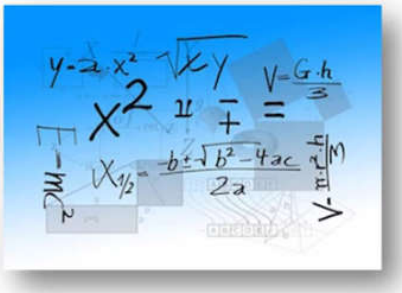
❒ To know more about usage of ***`Cross entropy`*** over ***MSE***, watch Geoffrey Hinton's ***"The SoftMax output function"*** video. He explains it very well there.

❒ If you'd like a light introduction into ***Cross Entropy***, then a good article to check out is called ***"A Friendly Introduction To Cross Entropy Loss"*** by ***Rob DiPietro***.

Additional Reading:

A Friendly Introduction to Cross-Entropy Loss

By Rob DiPietro (2016)



Link:

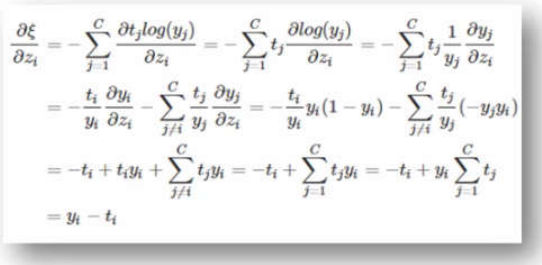
https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/

❒ If you'd like to dig into the mathematics behind ***Cross Entropy*** & ***SoftMax*** then check out an article by or a blog called: "***How To Implement A Neural Network Intermezzo 2***" (intermezzo means an intermediate thing).

Additional Reading:

How to implement a neural network Intermezzo 2

By Peter Roelants (2016)



$$\frac{\partial \xi}{\partial z_i} = -\sum_{j=1}^{C} \frac{\partial t_j log(y_j)}{\partial z_i} = -\sum_{j=1}^{C} t_j \frac{\partial log(y_j)}{\partial z_i} = -\sum_{j=1}^{C} t_j \frac{1}{y_j} \frac{\partial y_j}{\partial z_i}$$
$$= -\frac{t_i}{y_i} \frac{\partial y_i}{\partial z_i} - \sum_{j \neq i}^{C} \frac{t_j}{y_j} \frac{\partial y_j}{\partial z_i} = -\frac{t_i}{y_i} y_i (1 - y_i) - \sum_{j \neq i}^{C} \frac{t_j}{y_j} (-y_j y_i)$$
$$= -t_i + t_i y_i + \sum_{j \neq i}^{C} t_j y_i = -t_i + \sum_{j=1}^{C} t_j y_i = -t_i + y_i \sum_{j=1}^{C} t_j$$
$$= y_i - t_i$$

Link:

http://peterroelants.github.io/posts/neural_network_implementation_intermezzo02/

CNN: Image Recognition Project

9.5.1 Problem Description

A ***CNN*** is just an ***ANN*** on which you use ***convolution trick*** to add some ***Convolutional layers***. We use this ***Convolutional layers*** to preserve the ***special structure*** in ***images*** So that we can ***classify*** some ***images***.

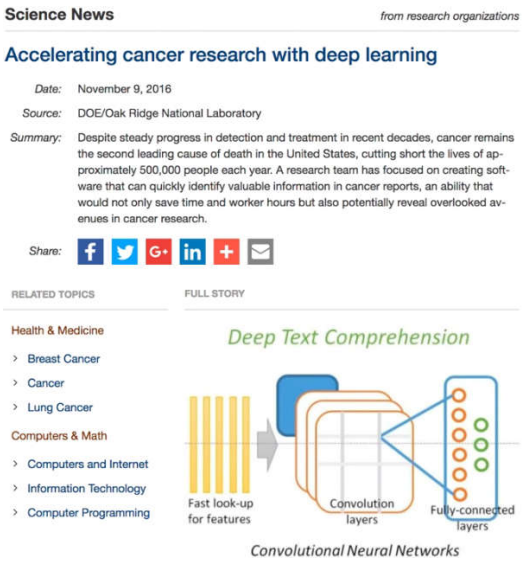
- CNN are great deep learning models for computer vision, to classify some images/photographs, or even some videos.

Problem Description: In this section, we are not going to solve any **business problem** as we used to do in the **previous sections**. Here we are simply going to solve an ***image classification problem***.

- We will have some images of ***cats*** and ***dogs*** and we will train a ***CNN*** to predict if the ***image*** is a photo of a ***dog*** or of a ***cat***.
- Where we'll have a ***folder*** full of ***images***, these ***images will be some*** images of ***cats*** and ***dogs***.

Classify any image: Once we build our ***CNN model*** you will simply need to ***change*** the images of ***cats*** and ***dogs*** in the folder and replace them by the ***images*** you want to work with.

- For example, you will be able to ***replace*** these ***cats*** and ***dogs*** images by some ***medical images*** such as: ***brain image*** contains a ***tumor*** or not. If you know the answers of ***enough observations*** (like ***thousands*** of ***observations***), then you will be able to **train** a **CNN** to predict if some new ***brain image*** contains a ***tumor*** or not.
- CNN can use to accelerate cancer research as we can see in this article right here.



9.5.2 Work environment for CNN

Work **environment** for **CNN** will be different than other projects that are done in previous chapters. Because remember, the **data-set** we used to work with were **tables**, containing ***some independent*** variables and ***one dependent*** variable.

- Now we have some images, so we need to do some image pre-processing to be able to input these images in our CNN.
- We have a folder named "***dataset***" which contains ***10,000 images*** of ***cats*** and ***dogs***. These images are ***pre-categorized*** and named as "***cat.0001.jpeg***" or "***dog.0067.jpeg***" in *jpeg-image-format*.

This ***"dataset"*** folder must be a ***sub-folder*** of our ***working directory*** where we have ***CNN.py*** file.

9.5.3 Data preprocessing : folder structure

Previously we had ***.csv*** file but here the independent variables are now the ***pixels*** distributed in ***3D arrays***, and therefore we ***cannot*** add ***explicitly*** the ***dependent variable*** in our ***dataset*** because it wouldn't make much sense to add this dependent variable column along the ***3D*** arrays ***representing*** the images.

Remember, when we ***train*** a ***ML*** model we always need the ***dependent variable*** to have the ***real results*** that are required to ***understand*** the ***correlations*** between the ***information***.

- But here, ***since*** we cannot add this ***dependent variable column*** in the same table, how can we extract the info of this dependent variable?

❒ ***We have several solutions:***

[1]. ***Categorize*** each ***image*** by giving ***category*** & ***Number*** to its ***image file*** name. Eg: "***cat.0001.jpeg***" or "***dog.0067.jpeg***". Then split all files into train-test.

⇨ A classic solution is to only have a dataset containing our images, separated in two different folders, ***training_set*** and ***test_set***.

⇨ Name each of these images by the category, for example, as "***cat.0001.jpeg***" or "***dog.0067.jpeg***" a number to differentiate all the images. In each folder the ***training set*** and the ***test set***, we would get, for example, ***5,000*** images of ***cats*** and ***5,000*** images of ***dogs***.

⇨ Then we can ***write*** some kind of ***code*** to ***extract*** the ***label*** name ***Cat*** or ***Dog*** from the ***name*** of the ***image file*** to specify to the ***algorithm*** whether this ***image*** belongs to the ***class-Cat*** or belongs to the ***class-Dog***.

⇨ And in some way, we get the our ***dependent*** variable ***vector***, by ***filling*** this ***dependent variable vector*** with the ***label names*** (cat/dog) that we managed to ***extract*** from the ***image file names*** of all our images.

[2]. ***Categorize*** each ***image*** by creating ***different folder***, eg: "**`cats`**", "**`dogs`**" for each train & test folder. We're gonna use that in **`Keras`**.

⇨ Other solution, comes with ***Keras***, it contains some ***tricks*** and ***tools*** to ***import*** some ***images*** in a very efficient way. And that's the solution we'll use.

⇨ ***Folder Structure for Keras:*** To import the images with ***Keras***, we only need a ***special folder structure*** for our dataset.

☑ To split training set and a test set, we create **two sub-folders** inside ***"dataset"*** folder named "***test_set***" and "***training_set***". Inside each of those 2 folders we create two more folders named "***cats***" and "***dogs***".

☑ Each of "***cats***" and "***dogs***" inside "***training_set***" have **4000 cats-images** and **4000 dogs-images** respectively.

☑ Each of "***cats***" and "***dogs***" inside "***test_set***" have **1000 cats-images** and **1000 dogs-images** respectively.

☑ Then for total ***10000*** images we've divided the data into ***0.8*** for ***train-data*** and ***0.2*** for ***test-data***.

⇨ The first pillar of the structure is to separate your images into two separate folders, we already said that, a training set folder and a test set folder.

✓ But that's ***not*** the ***main point*** !!! remember we want to have a simple way to **differentiate** the **class labels** (i.e. ***cats*** and ***dogs***).

✓ To differentiate the ***cat images*** and the ***dog images***, the simple trick is to make **two** different **folders** named "***cats***" and "***dogs***" one folder for the cats and one folder for the dogs. We have to make "***cats***" and "***dogs***" folders inside of each "***test_set***" and "***training_set***" folders.

✓ But remember it is not essential to name each image as: "***cat.0001.jpeg***" or "***dog.0067.jpeg***", it could be "***0001.jpeg***" or "***0067.jpeg***" because those images are now in separate folders. Images are now categorized by the folder now. (But the files that we got from ***Kaggle*** are already named as "***cat.0001.jpeg***" or "***dog.0067.jpeg***").

✓ And that's how **Keras** will understand how to **differentiate** the **labels** of your **dependent variable**.

☝ Those ***images*** can be ***any kind*** of ***images*** you have on your ***computer***, can take some pictures of your ***friends*** and replace these ***dog's*** pictures (☺ because dogs are also our best friend ☺) by the pictures of your ***friends*** and then you'll be able to ***train*** an algorithm that will ***predict***, which ***friend*** of yours is in the ***pictures***. So that can be pretty fun to do, but ***remember***, you need a ***lot of images***.

❒ ***Where to find this dataset:*** This dataset is a very well-known dataset in ***computer vision***, it can be used as a ***performance benchmark***, to test your ***Deep Learning models*** on this to simply see if you get some ***good accuracy*** and so it's a very useful dataset.

☞ Our dataset here is actually a subset of the whole dataset, that you can find on ***Kaggle***. Because the original whole dataset contains ***25,000*** images, but here is just a subset we have ***10000*** image.

❒ ***Size Of Our Dataset:*** The size of our dataset is same as the dataset of the business problem we had in the ANN. The train-test split will be similar.

☞ We have ***10,000*** images in total in the dataset, ***8,000*** images in the ***training set*** and ***2,000*** images in the ***test set***. So that's an 80% 20% split, then in the training set we have ***4,000*** images of ***dogs*** and ***4,000*** images of ***cats***. And in the test set we have ***1,000*** images of ***dogs*** and ***1,000*** images of ***cats***.

❒ ***No encoding needed:*** We don't need to encode any ***categorical data*** because, of course, our ***independent variables*** are in someway the ***pixels*** in the three ***R-G-B channels***. So, there is no categorical data here and therefore we don't need to do any ***encoding***.

❒ ***splitting the datasets:*** We did it already splitted into two folders "***test_set***" and "***training_set***".

❒ ***Feature scaling:*** Of course we *need feature scaling*. Feature scaling is *100% compulsory* in *deep learning* and especially for *computer vision*.

☞ Previously *feature scaling* section was associated to *data pre-processing* part. But here we are not using any previous pre-processing techniques, so we will take care of *feature scaling later*, just before we fit our *CNN* to our images.

✋ So some part of *data pre-processing* was done *manually*. We do *feature scaling* and image *augmentation*, so that our deep learning *models* can run the most *efficiently* as possible.

✋ Hence the first part of our *CNN* model *won't be* our usual *data pre-processing*, we built the *CNN first*.

9.5.4 Import packages for CNN

The first step is to import all the *Keras* packages. Following are the only packages we'll need to make our *convolutional neural network*.

```
# =========== ====  Convolutional Neural Network : CNN  ==== ============

# ----------- Install following  packages -------------
    # Install Theano
    # Install Tensorflow
    # Install Tensorflow
    # Install Keras



# ---------- Part 1 - Building the CNN ----------------
# Importing the Keras libraries and packages

from keras.models import Sequential     # to initialize as sequence-of-layers
from keras.layers import Convolution2D  # Convolution step for images
from keras.layers import MaxPooling2D   # not "MaxPool2D". Pooling step for images
from keras.layers import Flatten        # Flattening step
from keras.layers import Dense          # ads fully-connected-layers to classic ANN
```

i. The first package is *Sequential*, and we already in *ANN*. we'll use it to *initialize* our *neural network* because, remember, there are *two ways* of initializing the neural network, either as a **sequence of layers** or as a **graph**. And since a *CNN* is still a *sequence* of *layers*, well, we use the *Sequential* package to **initialize** our **neural network**.

ii. Second package, *Convolution2D*, is the package is used for the *first step* of making the *CNN*, the *Convolution Step*, in which we add the *convolutional* layers.

iii. Since we're working on *images* and since images are in *two dimensions* (unlike, for example, *videos* that are in *three dimensions* with the *time*) hence, *Convolution2D* package to deal with *images*.

iv. *MaxPooling2D*, used to proceed to step two, the *Pooling Step*, that will add our *pooling layers*.

v. Next package, *Flatten* used for 3rd step, *flattening*, in which we *convert* all the pooled *feature maps* (that we created through *Convolution* and *MaxPooling*) into a *large feature vector* that is then becoming the *input* of our *fully connected layers*.

vi. The *Dense* package is used it in the *ANN* section. This is the package we use to *add* the *fully connected layers* (of CNN) in a classic *ANN*.

Basically each of above packages corresponds to one step of the construction of the CNN.

9.5.5 Initialize our CNN

To initialize our CNN, we provoke the *Sequential* package. It's exactly the same as initializing a *classic ANN*. We are going to create an **object** of the *Sequential* class, and we're gonna call this object *cnn_classifier*.

☞ This *classifier* will classify some images, to tell if each image is a picture of a *dog* or a *cat*. So we're doing nothing *else* than *classification*.

```
# initializing the CNN
cnn_classifier = Sequential()
```

9.5.6 Adding Layers: step 1 : Convolution - layer

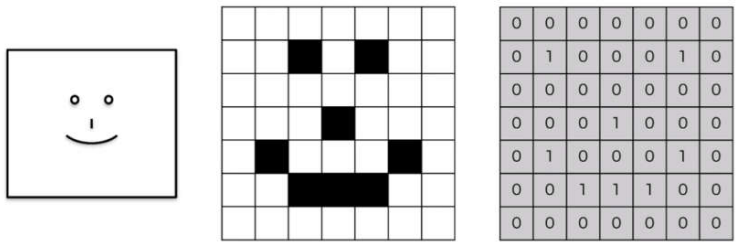
Here we add different ***layers***. The ***first layer*** that we're going to add is the ***convolutional*** layer.

A quick reminder of the building process: the ***CNN building process*** takes ***four steps***.

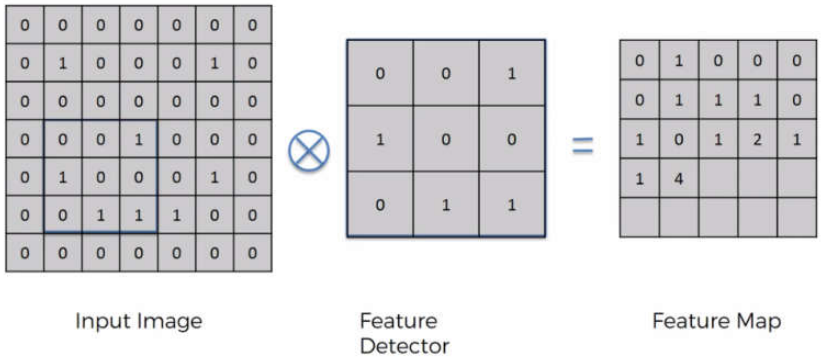
i. Step one, ***Convolution***
ii. step two, ***Max Pooling***,
iii. step three, ***Flattening*** and
iv. step four, ***Full Connection***.

Here we will complete the first step, ***Convolution***.





To simplify, let's take a simple black and white image with 0 and 1 pixels



❒ ***Convolution Step:*** Here we have an input image of a ***cat*** or a ***dog***, We convert into a ***table*** of ***pixel values***. ***Convolution step*** consists of applying ***several feature detectors*** on this ***input image***.

```
# step 1 : Convolution - layer
cnn_classifier.add(Convolution2D(32, 3, 3, input_shape = (64, 64, 3), activation= "relu"))
```

For example: In our intuition section we studied a ***smile face*** image (above),

☯ Here the ***feature detector*** is the **feature detector of the left side of a smiling mouth** when we ***slide it*** all over the ***input image*** and when the ***feature detector*** ***passes over*** the part of the ***face*** that contains this ***left side*** of the ***mouth*** that is ***smiling***, we get a ***high number*** in this table (notice the 4 on ***feature map***).

☞ So for each **feature detector** that we apply on the input image we get a **feature map**.

☞ The ***feature map*** contains some ***numbers*** and the ***highest numbers*** of the ***feature map*** is where the *feature detector* could ***detect*** a ***specific feature*** in the *input image*.

❒ ***Number of feature detectors:*** We will choose the ***number*** of ***feature detectors***, therefore the ***number*** of ***feature maps*** in this step.

⇨ We get as many ***feature maps*** as ***feature detectors*** we use to detect some ***specific features*** in the ***input image***. And those ***feature maps*** will form our ***Convolution-layer***.

❒ We are going to apply an ***add()*** method on our ***cnn_classifier*** object (we used it in ***ANN*** to create a ***classic layer*** composed of ***several nodes***, but here we used it to create a ***Convolution-layer***).

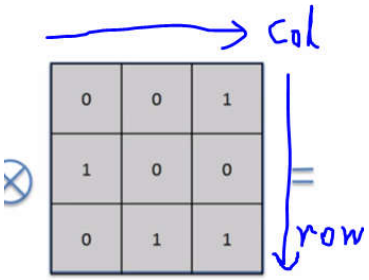
❒ ***Parameters of our Convolution-layer:*** Remember when we added the classic layer in the ***ANN***, we used the ***dense()*** function, which is used to add a ***fully connected layer (hidden-layer)*** in the ANN and therefore this is not the ***function*** that we're gonna use here (we'll use it later).

☞ The function that we're gonna use is ***Convolution2D***,

❒ Parameters for ***Convolution2D()***:

⇨ ***nb_filter***: the number of ***filters***. It specifies the number of ***feature-detectors/filters*** that we're going to apply on our input image to get this ***same number*** of ***feature maps***.

- So the ***number*** of ***filters*** that we choose here is the ***feature maps*** that we want to create as well because there will be ***one feature map*** created for ***each filter*** used.



⇨ ***row and column size of filter/feature detector:*** number of feature detectors is not the only thing that we need to choose here, we also need to specify row and column size of filter/feature detector/convolution kernel (convolution kernel is just another name for feature detector or filter).

```
cnn_classifier.add(Convolution2D(32, 3, 3,
```

Here we set ***no. of filters*** = **32**, ***row_size*** = **3**, ***column_size*** = **3.**

Why we choose number of filters as 32? Because,

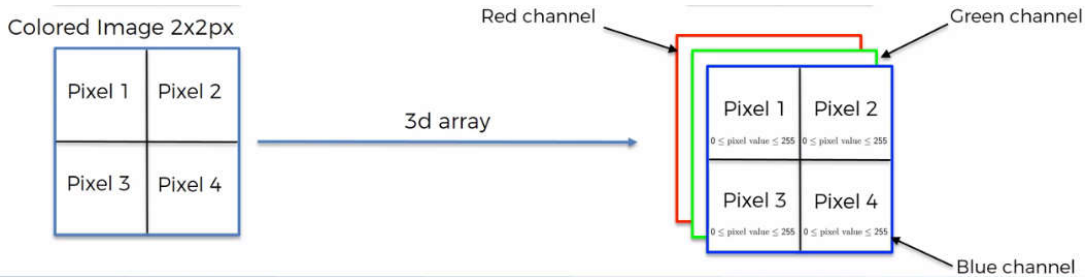
- ***Common Practice is 32:*** Most of the CNN architectures, the common practice is to start with ***32***. Therefore our ***convolutional layer*** will be composed of ***32 feature maps***.
  - If we start with ***32 feature detectors*** in the ***first convolutional*** layer and then we can add other ***convolutional layers*** with more feature detectors like ***64*** and then ***128*** and then ***256***.
- ***We have no GPU:*** The second reason we are working on a CPU.

⇨ ***border_mode*** (optional): Just to specify how the feature detectors will ***handle*** the ***borders*** of the ***input image***. But most of the time we choose keyword- "**same**", as the default value.

- We don't need to input it because default is automatically applied.

⇨ ***input_shape:*** Very important argument. ***input_shape*** is the ***shape*** of your ***input image*** on which you are ***going to apply*** your ***feature detectors*** through the **convolution operation**.

- Since all of our images don't have the same size, same format and therefore we need to force them in some way having the same format.
- Therefore, specifying ***input_shape*** will ***convert*** all of our images into one same ***single format*** and therefore ***one fixed size*** of the image.
- Remember, we will do this **conversion** during the ***Image Pre-Processing*** **part**, right ***after*** we ***build*** our ***CNN*** and just ***before*** we ***fit*** our ***CNN*** to our ***images***.



- We know that:
  - If the image is a ***colored*** image then input images are converted into ***3D arrays***,
  - If the image is a ***black-and-white*** image then input images are converted into ***2D arrays***.

- Since we are working with ***colored images***, our images will be ***converted*** into ***3D arrays*** during the ***image pre-processing*** part.
  - This ***3D array*** is composed of ***three channels***, each channel corresponding to one color, ***Red***, ***Green*** or ***Blue (R-G-B),*** and each channel corresponds to **single** ***2D array*** that contains the ***pixels*** of our ***images***.

    For example, use `input_shape=(128, 128, 3)` for 128x128 RGB pictures in RGB channel. You can use `None` when a dimension has variable size.

    ```
    cnn_classifier.add(Convolution2D(32, 3, 3, input_shape = (64, 64, 3),
    ```

  - ***3*** is the ***number of channels*** (since we are dealing with the **colored image**), it will only be **1** if we're dealing with a **black and white image**.
  - ***64*** and ***64*** are the ***dimensions*** of the ***2D array*** in ***each*** of the ***channel***.
  - All of that means that we are expecting ***colored*** images of 64×64 pixels.

✋ ***Why small size:*** Here we are using a smaller format because we're using a **CPU**. This will be way enough to get some good accuracy results if you're working on a **GPU**, and we don't wanna wait **too many hours** to execute the code.

☝ You can choose a larger format like *128* by *128* or even *256* by *256*, but either you need to use a *GPU* or run your code before sleeping for *8 hours*.

✋ ***Why 3 channels (colored):*** We are keeping *three channels* of color information because *cats* and *dogs* don't have the *same colors* and therefore *differentiating* them with the *colors* can be helpful to classify them.

✋ Notice the ***order*** of the ***input_shape*** parameters here.

☝ ***Theano back-end:*** *Number of channels* first and then the *Dimensions of the 2D arrays*. This order is used in **Theano back-end** (old version ??).

```
input_shape=(3, 128, 128)
```

☝ ***TensorFlow back-end:*** The *Dimensions Of The 2D Arrays* first and then *Number of channels*. This order is used in **TensorFlow back-end**.

```
input_shape=(64, 64, 3)
```

⇨ ***activation:*** We also specify **rectifier activation function** type in ***Convolution2D***,

➢ We already used it in *Hidden-layers (fully connected layers)* in the ***ANN*** in the previous chapter. In ANN we used **activation= "relu"** to *activate* the *neurons* in the NN.

➢ But in ***CNN*** we used **activation= "relu"** to make sure we get *non-linearity* in all of our *feature maps*.

➢ It will make sure that we don't have any **negative pixel values** in our **feature maps**. Because depending on the **parameters** that we use for our **convolution operation**, we can get some **negative pixels** in the **feature map**.

➢ We need to **remove** these **negative pixels** in order to have **non-linearity** in our **CNN**.

➢ Because *classifying* some images is a *nonlinear problem* so we need to have *non-linearity* in our *model*.

```
cnn_classifier.add(Convolution2D(32, 3, 3, input_shape = (64, 64, 3), activation= "relu"))
```

So that's our **convolution-layer**:

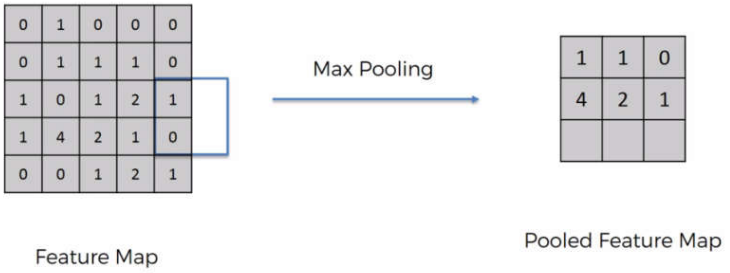
```
# step 1 : Convolution - layer
cnn_classifier.add(Convolution2D(32, 3, 3, input_shape = (64, 64, 3), activation= "relu"))
```

9.5.7 Adding Layers: Step 2 : Pooling - layer

We are ready to move on to 2nd Step, **Pooling step**. This **pooling step** is very easy, it just consists of *reducing* the *size* of your *feature maps*.

❒ In Previous convolution step we've used a 3×3 *px* size *table/filter* for *convolution operation*. We've also use ***Stride-size*** of *1* (the filter moves *1px* at a time, with **2px overlapping**).

❒ Now in this pooling step we're gonna use a 2×2 *px* size *table* for *Max-pooling operation* (choosing *maximum value* of the *four cells* inside this table/square). Now we're gonna use ***Stride-size*** of *2* (the table moves *2px* at a time, with **no overlapping**).

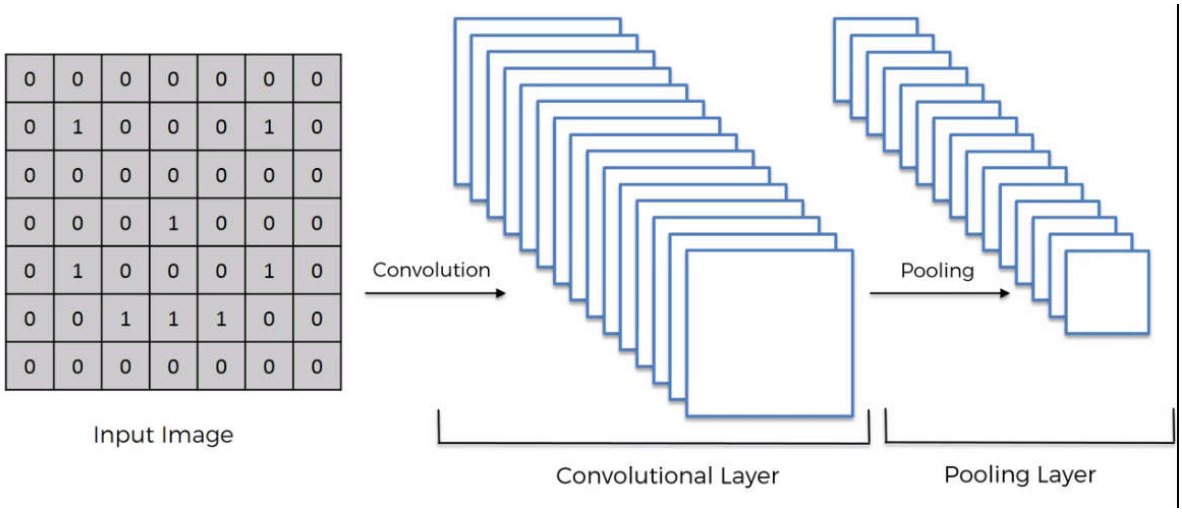


☞ Since *each time* we take the *max* of a *2-by-2 table*, at the *end*, we get a *new feature map [pooled-feature-map]*, with a *reduced size*. And more precisely, the *size* of the *pooled feature map* will be the *half* of the *original feature-map's size*.

☞ And then we obtain our next layer ***composed*** of all these ***pooled-feature-maps*** and that is called the ***Pooling Layer***.

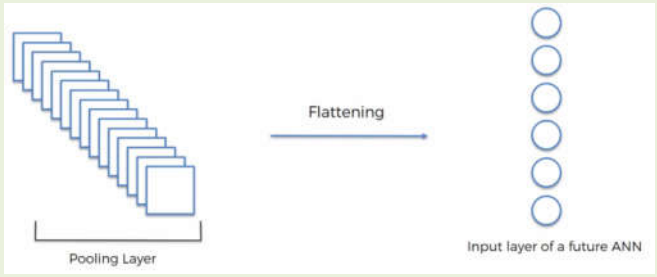
☞ This will reduce the complexity and the ***time execution*** but without the ***losing*** too much ***information*** because we are using ***max-pooling***.

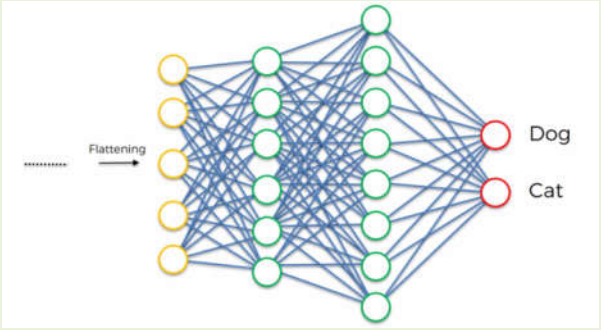
☞ Using max-pooling we are ***keeping track*** of the ***most-important features*** of the image, that contained the ***high numbers*** corresponding to where the ***feature detectors***, detected some ***specific features*** in the input image.



✋ Remember, we apply this ***max pooling*** to reduce the ***number of nodes*** for the ***Flattening*** step, it is some kind of ***reduction of independent variables***.

- ☝ Here we are actually ***reducing the size of*** flattened one-dimensional vector (which will be huge) for the Full Connection step.
- ☝ If we don't reduce the **size** of these **feature maps**, we'll get to **too large *flattened* one-dimensional *vector*** and then we'll get ***too many nodes*** in the ***fully connected layers*** in the ***NN*** part and therefore our model will be highly ***compute-intensive***.
- ☝ So we don't lose the ***`spatial structure`*** information, hence we don't lose the performance of the model.
- ☝ But at the same time, we managed to reduce the ***time complexity*** and we make it ***less compute-intensive***.





❐ ***Creating Pooling-layer:*** We'll apply same ***`add()`*** method to our classifier ***`cnn_classifier`*** to add a pooling-layer. And inside ***`add()`*** we use ***`MaxPooling2D`***:

☞ Parameters of ***`MaxPooling2D`***.

➢ **`pool_size`:** We are gonna use ***`pool_size = (2, 2)`***. Most of the time we take a ***`2-by-2`*** **`pool_size`** when we apply **max pooling** on our **feature maps**. Because we don't wanna lose the information. We're still being precise on where we have the **high numbers (max-pooling)** in the **feature maps**.

☞ Basically adding following line will ***reduce*** the ***size*** of your ***feature maps***, and ***pooled-feature-map's size*** will be half of ***original-feature-map's*** size.

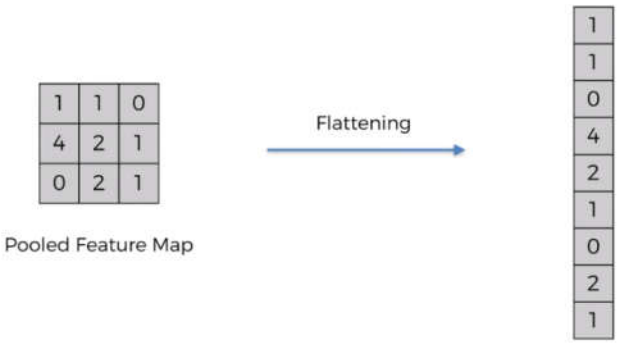
```
# step 2 : Pooling  - layer
cnn_classifier.add(MaxPooling2D(pool_size = (2, 2)))
```

So we just reduced the ***complexity*** of our model without reducing its performance.

9.5.8 Adding Layers: Step 3 : Flattening for ANN-part

Flattening step consists of taking all our **Pooled Feature Maps** and put them into **one single vector**.

- This is going to be a ***huge vector*** of course because even if we ***reduced*** the size of the ***Feature Maps*** in ***pooled-feature-maps***, we still have ***many Pooled Feature Maps***.
- This ***single vector*** is going to be the ***Input Layer*** of a future ***ANN***, this ***ANN*** is similar to the previous chapter's ***ANN*** (which had ***hidden layers***), but this ***ANN*** has ***Fully Connected Layers*** (hidden-layers with full connection with all nodes).



Now we can ask ourselves two very important questions:

- ***Why don't we lose the*** `spatial structure` ***by*** `Flattening` ***all these*** `Feature Maps` ***into one same,*** `single vector`***?***
  - By creating ***convolution-layer*** of ***feature-maps***, we extracted the ***spatial structure information***, by getting some ***high numbers*** in ***each Feature Map***, using specific ***Feature-Detector/filter*** that we applied on the Input Image.
  - These ***high numbers*** in each ***Feature Map*** represent the ***spatial structure*** of our images, because these ***high numbers*** are associated to a ***specific feature*** in the input image.
  - When we apply the ***Max Pooling*** Step, we ***keep*** these ***high numbers*** because we take the ***Max-values***.
  - The ***Flattening Step*** just consists of putting all the ***numbers*** in the cells of each ***pooled-Feature-Map*** of the ***pooling-layer*** into one, same, single ***vector***. Hence those ***high numbers*** that are associated to a ***specific feature*** in the ***input image*** (represents the ***spatial*** structure) are ***still*** in the ***flattened-1D-vector***.

  Hence all the spatial structure information preserved in this one, huge, single flattened-1D-vector.

- ***Why didn't we*** `directly` ***take all the*** `pixels` ***of the*** `Input Image` ***and*** `Flatten` ***them into this one, same,*** `single vector`***, without applying the previous steps:*** `Convolution` ***and*** `Max-Pooling`***?***
  - If we directly ***Flatten*** all the ***Input Image pixels*** into this huge, single, ***one-dimensional vector***, then each node of this huge vector will represent ***one independent pixel*** of the Image.
    - Then we only get information of the ***pixel itself*** not the other ***pixels*** that are ***around*** it.
    - We **don't get information** of how this **pixel** is **spatially connected** to the **other pixels around** it. We don't get any information of the ***spatial structure*** around this pixel.
  - If we apply the ***Convolution*** and the ***Max Pooling*** step to create all the ***pooled-Feature-Maps***, and ***Flatten*** all these Maps into this huge, single, one-dimensional vector:
    - Then since each ***Feature Map*** corresponds to one ***specific feature*** of the image, then ***each node*** of this ***huge flattened-1D-vector*** will represent the ***information*** of a ***specific feature***, a ***specific detail*** of the ***Input Image***, (for example, the upper left border of a dog nose).
    - Because this ***high number*** doesn't represent a ***unique pixel*** by ***itself*** but the study of ***specific feature***, that the **Feature Detector** extracted from the **Input Image**, through the **Convolution Operation**.

  And therefore eventually, we keep the spatial structure information of the Input Image.

- ***Flattening the Pooling-layer:*** As usual, we're gonna take our classifier ***`cnn_classifier`***, apply same ***`add()`*** method to our classifier and inside ***`add()`*** we use ***`Flatten()`*** to flatten the pooling-layer.

```
# step 3 : Flattening
cnn_classifier.add(Flatten())
```

  - There is no need to specify any parameters, because ***Keras*** will understand that,
    - Since we're taking a ***`cnn_classifier`*** object, that it needs to Flatten the previous layer that we obtain in the Max Pooling Step, after the Convolution Step.

Now this huge, single vector is created and basically it contains all the information of the spatial structure of our images. Then the 4th step is to create a ***classic ANN***, that will classify the images using this ***huge, single vector***, as the ***Input Vector***.

9.5.9 Adding Layers: Step 4 : ANN – Full Connection

The ***full connection*** step, which basically consists of making a classic ***ANN*** composed of some ***fully-connected layers*** (instead of hidden layer).

❒ We managed to ***convert*** our ***input image*** into this ***one-dimensional vector*** that contains some information of the ***spatial structure*** or of some ***pixel patterns*** in the image.

☞ We use this **input vector** as the **input layer** of a **classic ANN**. A classic ANN, can be a great classifier for ***nonlinear problem*** like ***image classification***.

☞ Since we already have our ***input layer (flattened-vector)***, now we create a ***hidden layer*** i.e a ***fully-connected layer***.

❒ ***Fully-connected-layer:*** This is similar to previous chapter, but output-layer will have two nodes.

☞ We use ***`Dense()`*** method inside ***`cnn_classifier.add()`*** to add the fully-connected layer (hidden layer).

➢ Parameters:

i. ***`units:`*** is the number of nodes in the **hidden layer**. ***How many nodes do we need to input here?***

- Because, there was ***no rule of thumb*** to choose a ***number of nodes*** in the ***hidden layer***.
- We saw that a common practice is to choose a number of hidden nodes ***between*** the ***number of input nodes*** and the ***number of output nodes***.
- But here we have too many input nodes, because, we built ***32*** **pooled-feature-maps** and each contains many ***cells***, those are contained in ***flattened-one-single-vector*** which is the ***input layer*** of our ***fully-connected layer***. Thus we end up with a lot of input nodes.
- So we are not gonna count all of them right now, just remember that we ***shouldn't take*** a ***too-small number***.
- We are gonna ***choose*** here ***128***.
- Remember this choice of numbers ***results*** from ***experimentation***. **128** is not so small and not too big to make it highly ***compute-intensive***.
- By experimenting on this outputting parameter, we realize that a number around 100 is a good choice. We could have picked 100, but it is a common practice to pick a power of 2 (i.e $2^7 = 128$).

ii. ***`activation`***: The activation function, will be "relu" for the fully connected layers ***`activation= "relu"`***.

```
cnn_classifier.add(Dense(units= 128, activation= "relu")) # fully connected layers
```

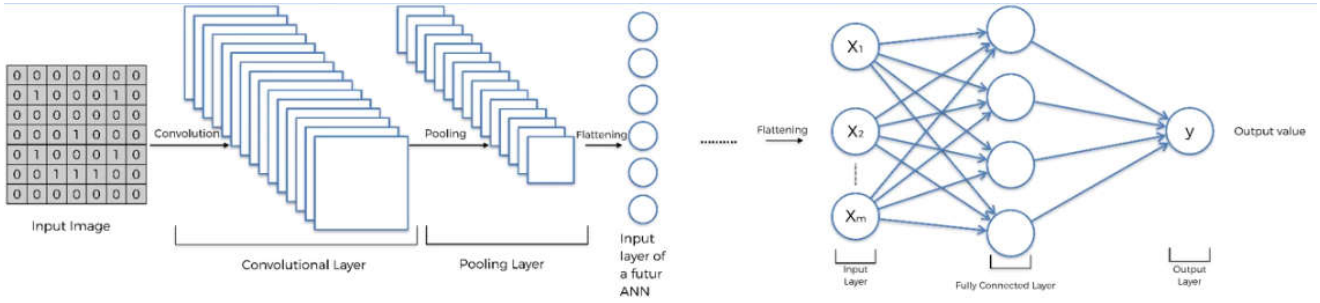
❒ ***Output-layer:*** The last layer that we need to add is the output layer. The value of ***`output_dim = 1`*** because out output will be cat/dog. We are just expecting one node that is going to be the predicted probability of one class, the dog or the cat.

☞ The activation function is the ***SIGMOID activation function***. Because we have a binary outcome, cats or dog.

☞ If we had an outcome with ***more*** than ***two categories***, we would need to use the ***SOFTMAX activation*** function.

```
cnn_classifier.add(Dense(units= 1, activation= "sigmoid")) # output layer
```

❒ This ***full connection step*** only consisted of adding the ***fully-connected layer***, that is the ***hidden layer***, and then the ***output layer*** to get the ***final predictions***.




```
# step 4 : ANN - full connection
cnn_classifier.add(Dense(units= 128, activation= "relu")) # fully connected layers
cnn_classifier.add(Dense(units= 1, activation= "sigmoid")) # output layer
```

9.5.10 Compilation: compile - CNN model

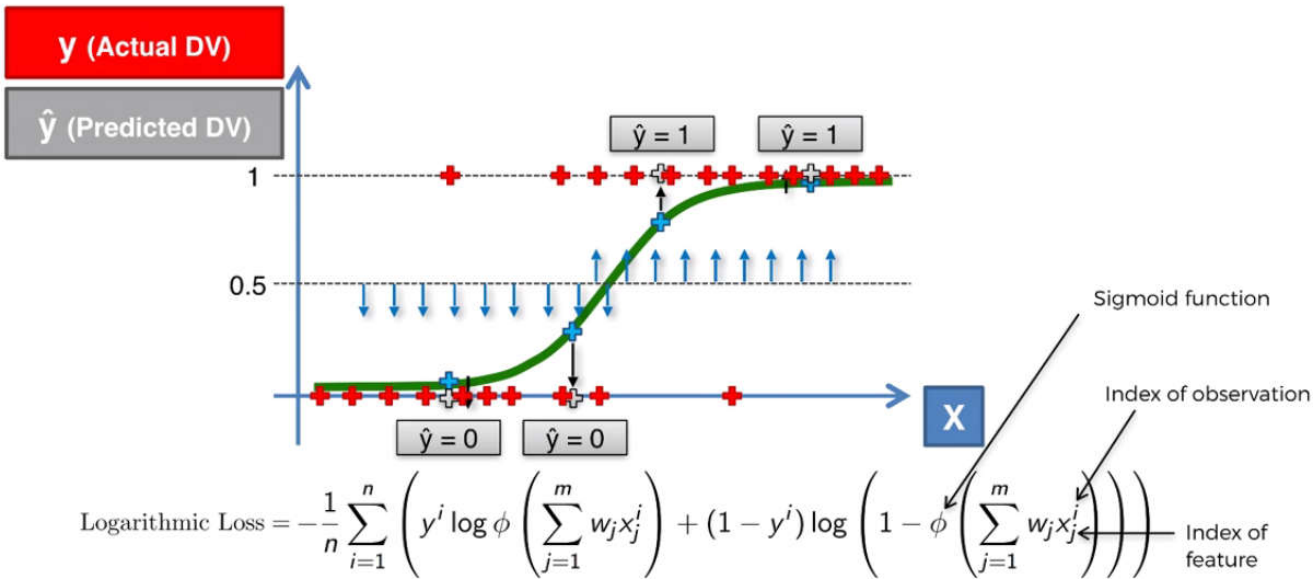
We just need to compile the whole NN by choosing a ***SGD algorithm***, a ***loss function*** and, eventually, a ***performance metric***. It is same as the ***classic-ANN*** in the ***previous chapter***.

```
# compile the NN
cnn_classifier.compile(optimizer= "adam", loss="binary_crossentropy", metrics = ["accuracy"])
```

❒ We apply ***compile()*** over our classifier ***cnn_classifier***.

☞ ***Parameters:***

- optimizer: " ***adam*** " specifies the ***SGD algorithm***. It is the Adam algorithm.
- loss: We choose the ***binary_crossentropy*** for two reasons:
  - ➲ First of all, because this function corresponds to the ***logarithmic loss***, that is the loss function that we use in general for ***classification*** problems using a ***classification model*** like ***logistic regression***.
  - ➲ The second reason is that we have a binary outcome, cat or dog. and therefore, we need to choose the **binary-cross-entropy** ***loss function***.
  - ➲ If we had more than two outcomes, like cats, dogs, and birds, well, we would need to choose **categorical-cross-entropy** as ***loss function***.
- metrics: To choose the ***performance metric***. The most common **performance metric** is the ***accuracy*** metric.



9.5.11 Image Preprocessing

We just completed ***part one: Building the CNN***. We designed the architecture of our CNN.

❒ Now we're beginning ***part two: Image Preprocessing***, where we will *fit our CNN to our images*.

☞ We will actually do it in **one step** because we're gonna use a **shortcut**, the ***Keras Documentation***.

❒ We will use ***Keras Documentation*** for a process called ***image augmentation***, that basically consists of ***preprocessing*** your ***images*** to prevent ***overfitting***.

☞ If we don't do this ***Image Augmentation***, we might get is a *great accuracy* result on the *training set*, but a much *lower accuracy* on the *test set*.

❒ ***Image Augmentation Process:*** We know that one of the situations that lead to ***overfitting*** is when we have *few data* to *train* our *model*. In that situation, our model finds some *correlations* in the *few observations* of the *training set*, but *fails* to generalize this *correlations* on some *new observations*.

☞ And when it comes to **images**, we actually need a **lot of images** to find and **generalize** some **correlations**, because in ***Computer Vision***, our **model** ***doesn't need*** to find some *correlations* between *independent and dependent variables*. It needs to find some *patterns* in the *pixels*, and to do this it requires a *lot* of *images*.

☞ Right now, we are working with 10,000 images, 8,000 images on the training set, and that is actually not much to get some **great performance** on results. We either need some ***more images***, or we can use ***data augmentation*** trick.

🛠 **Image augmentation** will create many **batches** of our **images**, and in each **batch** it will apply some ***random transformations*** on a ***random selection*** of our ***images***, like ***rotating*** them, ***flipping*** them, ***shifting*** them, or even ***shearing*** them, and eventually we'll get many more ***diverse images*** inside these ***batches***, and therefore a lot ***more material*** to ***train***.

☝ That's why it is called ***image augmentation***. That's because the ***amount*** of our training images is ***augmented***. Besides, because of the ***random transformations***, our model will ***never*** find the ***same picture*** across the ***batches***.

☝ In summary, ***image augmentation*** is a technique that allows us to ***enrich*** our ***data set***, ***without adding*** more ***images*** and therefore that allows us to ***get good performance*** results with ***little or not overfitting***, even with a ***small amount*** of ***images***.

❒ We're gonna use this **TensorFlow-Keras Documentation** shortcut. In your browser, you can type ***"keras image preprocessing tensorflow"***. Search the link for ***Documentation***.

☞ We will go to the link "imagedatagenerator": However the code might change in latest version. Now we use following link:

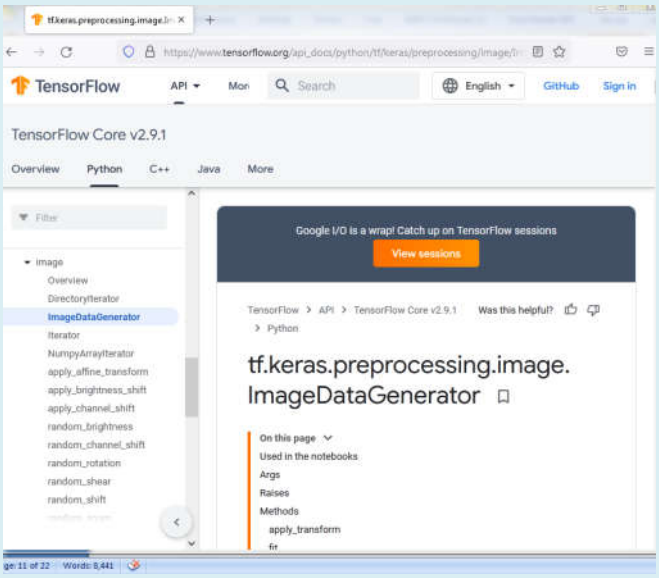
https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator

The ***Keras Library Documentation*** is also available currently on:

https://faroit.com/keras-docs/1.2.0/preprocessing/image/

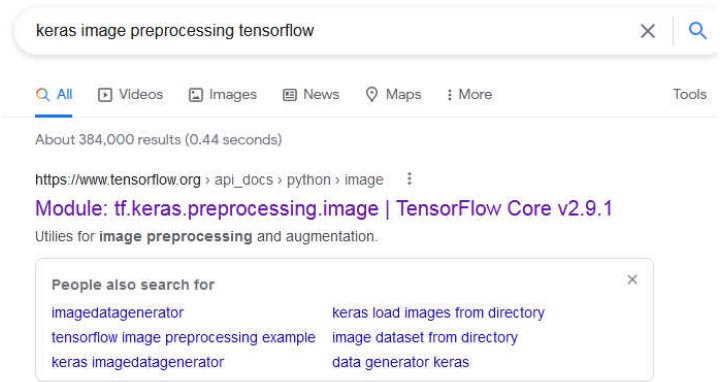
☞ Open `tf.keras`. You get a lot of ***informations*** about ***Keras*** and ready-to-use codes that you can take for your deep learning project.

```
tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    zca_epsilon=1e-06,
    rotation_range=0,
    width_shift_range=0.0,
    height_shift_range=0.0,
    brightness_range=None,
    shear_range=0.0,
    zoom_range=0.0,
    channel_shift_range=0.0,
    fill_mode='nearest',
    cval=0.0,
    horizontal_flip=False,
    vertical_flip=False,
    rescale=None,
    preprocessing_function=None,
    data_format=None,
    validation_split=0.0,
    interpolation_order=1,
    dtype=None
)
```



☞ We're gonna look for Preprocessing.

☞ Note that, deep learning can also be applied to text in a very powerful way. There we use " text preprocessing ".



❒ ***ImageDataGenerator:*** That's the first function that we're gonna use to ***generate*** this ***image augmentation***.

☞ From following page we'll take ready-to-use code and that corresponds very well to how we structured our data set,

https://faroit.com/keras-docs/1.2.0/preprocessing/image/

❒ There are ***two ways*** to ***preprocess*** our ***images*** by applying ***image augmentation*** on them:

☞ It's either by using this code that is based on the flow method:
Example of using `.flow(X, y):` Following doesn't match to our ***folder-structure***.

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)

# compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied)
datagen.fit(X_train)

# fits the model on batches with real-time data augmentation:
model.fit_generator(datagen.flow(X_train, Y_train, batch_size=32),
                    samples_per_epoch=len(X_train), nb_epoch=nb_epoch)

# here's a more "manual" example
for e in range(nb_epoch):
    print 'Epoch', e
    batches = 0
    for X_batch, Y_batch in datagen.flow(X_train, Y_train, batch_size=32):
        loss = model.train(X_batch, Y_batch)
        batches += 1
        if batches >= len(X_train) / 32:
            # we need to break the loop by hand because
            # the generator loops indefinitely
            break
```

☞ Or this code that is based on the `flow_from_directory` method. Example of using `.flow_from_directory(directory):`

➢ Following matches to our ***folder-structure***. We'll use this code section because we structured our folder in this specific way so that our image classes of cat or dogs can be well identified in the separate folders.

```
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        'data/train',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

model.fit_generator(
        train_generator,
        samples_per_epoch=2000,
        nb_epoch=50,
        validation_data=validation_generator,
        nb_val_samples=800)
```

➢ Since our dataset is on our **working directory**, we need to change few things on ***flow_from_directory()*** function.

- Basically above ***copied code-segment*** has everything that we need to preprocess-augment our images, and even fitting our CNN that we just built on our images.
- That's the end of the code, because this ***fit_generator()*** method will not only fit our ***CNN*** to the ***training_set***, but at same time it will also ***test*** its ***performance*** on some ***new observations*** of our **test set**, (i.e the images of our ***test_set*** folder).

❒ First we need to import ***ImageDataGenerator*** fronn ***keras***.

☞ We'll create ***two data-generator/augmentation objects*** for train and test data named ***train_datagen*** and ***test_datagen***.

☞ For those 2 objects we specify the ***transform parameters***. In this ***image augmentation*** part, we apply several transformations like :

i. **rescale:** rescaling factor. Defaults to None. If **None** or **0**, ***no rescaling*** is applied, otherwise we multiply the data by the value provided (before applying any other transformation).

⇨ ***rescale*** is always compulsory and it corresponds to the feature scanning part of the data preprocessing phase that we know.

⇨ ***Why we have to rescale by 1./255:*** ***rescale*** is a value by which we will ***multiply the data*** before any ***other processing***. Our original images consist in ***RGB coefficients*** in the ***0-255***, but such values would be **too high** for our **models** to process (given a typical learning rate), so we target **values between *0*** and ***1*** instead by scaling with a **1./255** factor.

ii. **shear_range:** (Float). Shear Intensity (Shear angle in counter-clockwise direction as radians).

⇨ **sharing** is a ***geometrical transformation*** that is also called ***transvection***. Here the pixels are moved to a fixed direction over a proportional distance from a line that is parallel to the direction they're moving to. So basically that is just a ***geometrical transformation*** for ***augmenting*** our images.

iii. **zoom_range:** (Float) Range for random zoom. If a float, `[lower, upper] = [1-zoom_range, 1+zoom_range].` This is some sort of ***random zoom*** that we apply on our images.

iv. **horizontal_flip:** (Boolean) Randomly **flip** inputs-images **horizontally**.

However we can also have ***vertical_flip***, but that is not used here.

We can have fun and apply all the image transformations that there are in this Keras Documentation, but for now we will just use what we have in this code segment. That will be way enough and you'll see that we get good results.

☞ Rename ***train_generator*** and ***validation_generator*** to ***training_set*** and ***test_set*** respectively. These two are the instances of objects ***train_datagen*** and ***test_datagen*** and we apply ***flow_from_directory*** method to import the train and test images to apply augmentation and then CNN on them. We need to ***specify*** following ***parameters*** for ***flow_from_directory***

i. **directory:** path to the ***target directory***. It should contain one ***subdirectory per class***. Any PNG, JPG or BNP images inside each of the subdirectories directory tree will be included in the generator.

⇨ We have to replace "**directory**" with corresponding **file path**. Set the image folder paths **'dataset/training_set'** and **'dataset/test_set'** for ***training_set*** and ***test_set*** respectively. (notice back slash **/** used instead of ****)

⇨ We ***don't*** have to ***specify*** the ***whole path*** that leads to this ***dataset***, because this dataset is already in the ***working directory*** folder.

ii. **target_size:** ***Tuple*** of integers, default: ***(256, 256)***. The ***dimensions*** to which all images found will be ***resized***. In our case since we are working with ***CPU***, we set it to ***(64, 64)*** it should be same as we set ***input_shape*** (dimension of expected resized images) in ***Convolution2D***.

iii. **batch_size:** Size of the **batches of data** (default: 32). ***batch_size*** is not related to **no. of filters** used in ***Convolution2D***.

⇨ It specifies the ***number*** of ***random samples*** of our images that will go through the ***CNN***, after which the ***weight*** will be updated.

iv. **class_mode:** one of "**categorical**", "**binary**", "**sparse**" or **None**. Default: "**categorical**". Here we use ***"binary"*** because we are working with ***binary category*** cat/dog.

⇨ That's the parameter ***indicating*** if your ***class***, your ***dependent variable***, is ***binary*** or has ***more than two categories***, and therefore since we have **two classes** here, ***cats*** and ***dogs***, well the **class_mode = "binary"**.

- ***class_mode*** Determines the ***type*** of ***label arrays*** that are returned:
  - "***categorical***" will be ***2D one-hot encoded labels***,
  - "***binary***" will be ***1D binary labels***,
  - "***sparse***" will be ***1D integer labels***.
  - If ***None***, ***no labels*** are returned.

- These two sections actually create the **training-set** and the **test-set**. Basically in this section we will create this ***training_set*** composed of all these ***augmented images*** extracted from our ***ImageDataGenerator***.
- Also ***test_set*** will create our test set from the images of the ***test_set folder*** that are extracted from our *ImageDataGenerator*.
- ***test_set*** will be used to ***evaluate*** the ***model performance*** in ***fit_generator()*** part of the code.

- Finally last code section, the ***model.fit_generator***, where we ***fit*** our ***CNN*** to the ***training set***, while also ***testing*** its ***performance*** on the ***test set***. Replace "**model**" with "**`cnn_classifier`**".
  - ***fit_generator()*** at the end of the code section, will fit our ***CNN*** model on the ***training_set***, as well as ***testing*** its *performance* on the ***test_set***.
  - We are using this **`fit_generator()`** method to fit our ***CNN*** to our ***training_set*** and test its **performance** on the ***test_set*** at the same time, and this **`fit_generator()`** method is applied onto our ***CNN model***. We named our ***model*** as **`cnn_classifier.`**
    i. The first argument is our ***training_set***.
    ii. ***samples_per_epoch:*** Is the **number of images** we have in our **training set**. Remember **all** the **observations** of the **training set** pass through the **CNN** during **each epoch**, and since we have **8,000** images in our **training set**, we set ***samples_per_epoch = 8000***.
    iii. ***nb_epoch:*** That's the number of epochs we wanna choose to train our CNN. And here, **25** might be a good choice. So that we don't have to **wait** for **too long** to get our **results**.
    iv. ***validation_data:*** It corresponds to the ***test_set***, on which we want to evaluate the performance of our CNN, so we set ***validation_data = test_set.***
    v. ***nb_val_samples:*** It corresponds to the number of images in our test_set, and that is 2,000.

```
# ---------- Part 2 - Image Preprocessing & fit CNN to our images ----------------

from keras.preprocessing.image import ImageDataGenerator

# creating two data-generator/augmentation obgects for train and test data
        # here we specify the transform parameters

train_datagen = ImageDataGenerator(
                                   rescale=1./255,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True )

test_datagen = ImageDataGenerator(rescale=1./255)


# applying augmentation on training data : training folder path needed
                                      # target_size = input_shape (dimension of expected resized images)
                                      # batch_size is not related to no. of filters
training_set = train_datagen.flow_from_directory('dataset/training_set',
                                                 target_size=(64, 64),
                                                 batch_size=32,
                                                 class_mode='binary')


# applying augmentation on test data : test folder path needed
test_set = test_datagen.flow_from_directory('dataset/test_set',
                                            target_size=(64, 64),
                                            batch_size=32,
                                            class_mode='binary')
```

```
cnn_classifier.fit_generator(training_set,
 # due to incompatibility between Tensorflow and Keras version, "samples_per_epoch", "nb_epoch",
"nb_val_samples" may not work
 # Then use "steps_per_epoch", "epochs",  "validation_steps" instead
                             # samples_per_epoch = 8000,
                             # nb_epoch=25,
                             # nb_val_samples = 2000,
                             steps_per_epoch = 250,
                             epochs = 25,
                             validation_data=test_set,
                             validation_steps = 62)
```

☝ NOTE: Due to ***incompatibility*** between ***Tensorflow*** and ***Keras*** version, "**samples_per_epoch**", "**nb_epoch**", "**nb_val_samples**" may not work.

➲ Then use "**steps_per_epoch**", "**epochs**", "**validation_steps**" instead.

➲ Remember in ***image-augmentation*** we used ***32*** as ***batch_size*** in ***training_set***. Then we need to set **steps_per_epoch= 250**. Because we are dividing our dataset into several batches. And **32*250 = 8000**. Similarly **validation_steps= 62** because ***test_data*** size is ***2000*** we have 200 test image and we also used we used ***32*** as ***batch_size*** in ***test_set*** (**32*62 = 1984**).

☠ ***Errors:*** Make sure that your dataset or generator can generate at least "**steps_per_epoch * epochs**" batches
Epoch 1/25
25/32 [======================>.....] - ETA: 1s - loss: 0.6942 - accuracy: 0.5238WARNING:tensorflow:Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least "**steps_per_epoch * epochs**" batches (in this case, 800 batches). You may need to use the ***repeat()*** function when building your dataset.

✋ To avoid this, instead of manually setting "**steps_per_epoch**", "**epochs**", "**validation_steps**": we use following code:

```
cnn_classifier.fit_generator(training_set,
                             steps_per_epoch=math.floor((training_set.samples)/(training_set.batch_size)),
                             epochs=25,
                             validation_data=test_set,
                             validation_steps=math.floor((test_set.samples)/(test_set.batch_size)),
                                   )
```

❒ After the training is over. We obtained an ***accuracy*** of ***84%*** for the ***training set***, and ***75%*** for the ***test set***. Well, not too bad, but not too good either.

☞ The ***difference*** between the ***accuracy*** of the ***training set*** and the ***test set*** indicating whether there's ***overfitting*** or not.

☞ So ***75%*** accuracy on the ***test set*** is not bad. That means that we get ***three correct predictions*** out of ***four***, so that's actually not too bad.

☞ When we get quite a large difference between the ***accuracy on the training set*** and the ***accuracy on test set***. It's indicating that there is important ***overfitting***.

☞ Notice how ***test_set*** and ***training_set*** classifies the images:

```
In [12]: runfile('D:/1_Development_2.0/ML_phase_3_ML_Intro/
ml_p3_ch9_dp_2_CNN/prctc_cnn.py', wdir='D:/1_Development_2.0/
ML_phase_3_ML_Intro/ml_p3_ch9_dp_2_CNN')
Found 8000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
D:\1_Development_2.0\ML_phase_3_ML_Intro\ml_p3_ch9_dp_2_CNN
\prctc_cnn.py:72: UserWarning: `Model.fit_generator` is deprecated
and will be removed in a future version. Please use `Model.fit`,
which supports generators.
  cnn_classifier.fit_generator(training_set,
Epoch 1/25
  20/8000 [..............................] - ETA: 36:18 - loss:
0.7452 - accuracy: 0.4516  Traceback (most recent call last):
```

❒ Now we'll improve our model ***by*** adding an ***extra layer***, to make this ***accuracy*** on ***test-set*** will reach an accuracy ***over 80%*** and decrease this difference between the ***training_set accuracy*** and the ***test_set accuracy***.

9.5.12 Increasing Accuracy

To ***improve*** our model's accuracy we need make a ***deeper NN*** that is a ***deeper CNN***.

❒ We have two option:

[1]. First option is to add another ***convolutional layer*** and ***pooling layer***.

[2]. Second option is to ***add*** another ***fully connected layer*** (hidden-layer).

☞ The best solution is actually to add a ***convolutional layer***. And its very easy. We have to add just 2-lines of code, after our ***first pooling layer***. But you can always improve your model by considering the two options that is adding a ***convolutional layer*** as well as a ***fully connected layer***.

☞ However, we can to reach our goal of getting a **test set accuracy** of more than 80% (also ***reducing*** the ***over-fitting***) by only adding a **second convolutional layer**.

☞ We apply the ***2nd convolution operation*** to this ***first pooling layer***.

➢ So we build this ***2nd convolution*** and ***2nd pooling layers*** right after the first two layers, i.e. after ***1st convolution*** and ***1st pooling layers*** .

➢ And before the **flattening step**.

```
# step 1 : Convolution - layer
cnn_classifier.add(Convolution2D(32, 3, 3, input_shape = (64, 64, 3), activation= "relu"))

# step 2 : Pooling  - layer
cnn_classifier.add(MaxPooling2D(pool_size = (2, 2)))

# improving step : Adding 2nd-Convolution and  2nd-Pooling layers
cnn_classifier.add(Convolution2D(32, 3, 3, activation= "relu"))
cnn_classifier.add(MaxPooling2D(pool_size = (2, 2)))
```

✋ Notice, we don't need ***input_shape*** parameter, because there is no new ***input-images*** for this ***layer***, we are just taking the ***pooled-feature-maps*** coming from the previous step the ***first-pooling-layer***.

☝ So we're going to apply the ***convolution*** trick and the ***max pooling*** trick, ***not on the images*** but ***on the pooled feature maps***.

☞ Hence we don't need ***input_shape***. Because ***input_shape*** corresponds to our images dimensions that our CNN should expect.

❒ Therefore when you're adding an ***additional convolutional layer***, you just need

- a ***number of features detectors***,
- the ***dimensions*** of these ***feature detectors***
- and an ***activation function***.

☞ And then you apply ***max pooling*** with only ***pool_size*** parameter.

☺ If you want to have fun adding **new additional convolutional layers**, then you can ***increase*** the number of ***feature detectors*** and ***double*** it ***each time***.

☺ So for example, you can add a ***third convolutional layer*** with ***64 feature detectors***. That's a common practice and that leads to great results.

❒ After ***25 epoch***, are about to get an accuracy of ***85%*** for the ***training set*** and ***82%*** for the ***test set***.

☞ That's great!! We're not only reached our goal to obtain the test set accuracy over 80% and also we ***reduced*** the ***difference*** between the ***training set accuracy*** and the ***test set accuracy***. Because now indeed we get a ***difference*** of ***3%*** as opposed to this ***10%*** difference that we got in the ***previous result***. (***84%*** on ***training set*** and ***75%*** on ***test set***)

✋ ***Getting even-more accuracy:*** Of course, adding more **convolutional layers** will help get an even **better accuracy**. But if you increase more, ***increase*** the ***input_shape*** of image.

☝ ***Higher image-size:*** Using higher ***input_shape*** & ***target_size*** gives a better accuracy for your images of the ***train set*** and the ***test set*** so that you get **more information** of your **pixel patterns**.

☝ Because if you increase the size of your images, all your **images** will be **resized**, you will get ***a lot more pixels*** in the ***rows*** and a ***lot more pixels*** in the ***columns*** in your ***input images***, therefore you will have more information on the pixels.

☝ ***GPU and more time:*** To do this, we recommend using a ***GPU*** or trying this ***before*** getting to ***sleep*** and you might even be able to get an ***accuracy*** **over 90%.**

Training-set accuracy & test-set accuracy:

```
Epoch 1/100
250/250 [==============================] - 86s 340ms/step - loss: 0.6881 - accuracy: 0.5411 -
val_loss: 0.6721 - val_accuracy: 0.5796
```

➢ ***accuracy: 0.5411*** is the accuracy on ***training_set***.

➢ ***val_accuracy: 0.5796*** is the accuracy on ***test_set***

Practiced version

```
# ========== ====  Convolutional Neural Network : CNN  ==== ============

# ---------- Install following  packages -------------
    # Install Theano
    # Install Tensorflow
    # Install Tensorflow
    # Install Keras



# ---------- Part 1 - Building the CNN ----------------
# Importing the Keras libraries and packages

from keras.models import Sequential     # to initialize as sequence-of-layers
from keras.layers import Convolution2D  # Convolution step for images
from keras.layers import MaxPooling2D   # not "MaxPool2D". Pooling step for images
from keras.layers import Flatten        # Flatenning step
from keras.layers import Dense          # ads fully-connected-layers to classic ANN

# initializing the CNN
cnn_classifier = Sequential()

# step 1 : Convolution - layer
cnn_classifier.add(Convolution2D(32, 3, 3, input_shape = (64, 64, 3), activation= "relu"))

# step 2 : Pooling  - layer
cnn_classifier.add(MaxPooling2D(pool_size = (2, 2)))

# improving step : Adding 2nd-Convolution and  2nd-Pooling layers
cnn_classifier.add(Convolution2D(32, 3, 3, activation= "relu"))
cnn_classifier.add(MaxPooling2D(pool_size = (2, 2)))


# step 3 : Flattening
cnn_classifier.add(Flatten())

# step 4 : ANN - full connection
cnn_classifier.add(Dense(units= 128, activation= "relu")) # fully connected layers
cnn_classifier.add(Dense(units= 1, activation= "sigmoid")) # output layer

# compile the NN
cnn_classifier.compile(optimizer= "adam", loss="binary_crossentropy", metrics = ["accuracy"])



# ---------- Part 2 - Image Preprocessing & fit CNN to our images ----------------

from keras.preprocessing.image import ImageDataGenerator
import math

# creating two data-generator/augmentation obgects for train and test data
        # here we specify the transform parameters

train_datagen = ImageDataGenerator(
                                   rescale=1./255,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True )

test_datagen = ImageDataGenerator(rescale=1./255)


# applying augmentation on training data : training folder path needed
                                       # target_size = input_shape (dimension of expected resized
images)
                                       # batch_size is not related to no. of filters
training_set = train_datagen.flow_from_directory('dataset/training_set',
                                                 target_size=(64, 64),
                                                 batch_size=32,
                                                 class_mode='binary')
```

```
# applying augmentation on test data : test folder path needed
test_set = test_datagen.flow_from_directory('dataset/test_set',
                                            target_size=(64, 64),
                                            batch_size=32,
                                            class_mode='binary')


"""
cnn_classifier.fit_generator(training_set,
 # due to incompatibility between Tensorflow and Keras version, "samples_per_epoch", "nb_epoch", "nb_val_samples" may not work
 # Then use "steps_per_epoch", "epochs",  "validation_steps" instead
                         # samples_per_epoch = 8000,
                         # nb_epoch=25,
                         # nb_val_samples = 2000,
                         steps_per_epoch = 250,  # training_set_size/batch_size
                         epochs = 25,
                         validation_data=test_set,
                         validation_steps = 62   # test_set_size/batch_size
                         )

"""


# In this case no need to explicitly specify training_set's or test_set's sample-size: i.e 8000, 2000 or 800, 200
cnn_classifier.fit_generator(
                            training_set,
                            steps_per_epoch=math.floor((training_set.samples)/(training_set.batch_size)),
                            epochs=25,
                            validation_data=test_set,
                            validation_steps=math.floor((test_set.samples)/(test_set.batch_size))
                            )


# history = model.fit_generator(train_gen,
#                                   steps_per_epoch=(train_gen.samples/batch_size),  # len(train_gen)
#                                   epochs=100,
#                                   validation_data=validation_gen,
#                                   validation_steps=(validation_gen.samples/batch_size),
#                                   callbacks=[checkpointer],
#                                   workers=4
#                                   )


# python prctc_cnn.py
```

ImageDataGenerator

```
keras.preprocessing.image.ImageDataGenerator(featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    rotation_range=0.,
    width_shift_range=0.,
    height_shift_range=0.,
    shear_range=0.,
    zoom_range=0.,
    channel_shift_range=0.,
    fill_mode='nearest',
    cval=0.,
    horizontal_flip=False,
    vertical_flip=False,
    rescale=None,
    dim_ordering=K.image_dim_ordering())
```

Generate batches of tensor image data with real-time data augmentation. The data will be looped over (in batches) indefinitely.

- **Arguments**:
    - **featurewise_center**: Boolean. Set input mean to 0 over the dataset, feature-wise.
    - **samplewise_center**: Boolean. Set each sample mean to 0.

- **featurewise_std_normalization**: Boolean. Divide inputs by std of the dataset, feature-wise.
- **samplewise_std_normalization**: Boolean. Divide each input by its std.
- **zca_whitening**: Boolean. Apply ZCA whitening.
- **rotation_range**: Int. Degree range for random rotations.
- **width_shift_range**: Float (fraction of total width). Range for random horizontal shifts.
- **height_shift_range**: Float (fraction of total height). Range for random vertical shifts.
- **shear_range**: Float. Shear Intensity (Shear angle in counter-clockwise direction as radians)
- **zoom_range**: Float or [lower, upper]. Range for random zoom. If a float, `[lower, upper] = [1-zoom_range, 1+zoom_range]`.
- **channel_shift_range**: Float. Range for random channel shifts.
- **fill_mode**: One of {"constant", "nearest", "reflect" or "wrap"}. Points outside the boundaries of the input are filled according to the given mode.
- **cval**: Float or Int. Value used for points outside the boundaries when `fill_mode = "constant"`.
- **horizontal_flip**: Boolean. Randomly flip inputs horizontally.
- **vertical_flip**: Boolean. Randomly flip inputs vertically.
- **rescale**: rescaling factor. Defaults to None. If None or 0, no rescaling is applied, otherwise we multiply the data by the value provided (before applying any other transformation).
- **dim_ordering**: One of {"th", "tf"}. "tf" mode means that the images should have shape `(samples, height, width, channels)`, "th" mode means that the images should have shape `(samples, channels, height, width)`. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "tf".

- **Methods**:
  - **fit(X)**: Compute the internal data stats related to the data-dependent transformations, based on an array of sample data. Only required if featurewise_center or featurewise_std_normalization or zca_whitening.
    - **Arguments**:
      - **X**: sample data. Should have rank 4. In case of grayscale data, the channels axis should have value 1, and in case of RGB data, it should have value 3.
      - **augment**: Boolean (default: False). Whether to fit on randomly augmented samples.
      - **rounds**: int (default: 1). If augment, how many augmentation passes over the data to use.
      - **seed**: int (default: None). Random seed.
  - **flow(X, y)**: Takes numpy data & label arrays, and generates batches of augmented/normalized data. Yields batches indefinitely, in an infinite loop.
    - **Arguments**:
      - **X**: data. Should have rank 4. In case of grayscale data, the channels axis should have value 1, and in case of RGB data, it should have value 3.
      - **y**: labels.
      - **batch_size**: int (default: 32).
      - **shuffle**: boolean (defaut: True).
      - **seed**: int (default: None).
      - **save_to_dir**: None or str (default: None). This allows you to optimally specify a directory to which to save the augmented pictures being generated (useful for visualizing what you are doing).
      - **save_prefix**: str (default: `''`). Prefix to use for filenames of saved pictures (only relevant if `save_to_dir` is set).
      - **save_format**: one of "png", "jpeg" (only relevant if `save_to_dir` is set). Default: "jpeg".
    - **yields**: Tuples of `(x, y)` where `x` is a numpy array of image data and `y` is a numpy array of corresponding labels. The generator loops indefinitely.
  - **flow_from_directory(directory)**: Takes the path to a directory, and generates batches of augmented/normalized data. Yields batches indefinitely, in an infinite loop.
    - **Arguments**:
      - **directory**: path to the target directory. It should contain one subdirectory per class. Any PNG, JPG or BNP images inside each of the subdirectories directory tree will be included in the generator. See this script for more details.
      - **target_size**: tuple of integers, default: `(256, 256)`. The dimensions to which all images found will be resized.
      - **color_mode**: one of "grayscale", "rbg". Default: "rgb". Whether the images will be converted to have 1 or 3 color channels.
      - **classes**: optional list of class subdirectories (e.g. `['dogs', 'cats']`). Default: None. If not provided, the list of classes will be automatically inferred (and the order of the classes, which will map to the label indices, will be alphanumeric).
      - **class_mode**: one of "categorical", "binary", "sparse" or None. Default: "categorical". Determines the type of label arrays that are returned: "categorical" will be 2D one-hot encoded labels, "binary" will be 1D binary labels, "sparse" will be 1D integer labels. If None, no labels are returned (the generator will only yield batches of image data, which is useful to use `model.predict_generator()`, `model.evaluate_generator()`, etc.).
      - **batch_size**: size of the batches of data (default: 32).
      - **shuffle**: whether to shuffle the data (default: True)
      - **seed**: optional random seed for shuffling and transformations.
      - **save_to_dir**: None or str (default: None). This allows you to optimally specify a directory to which to save the augmented pictures being generated (useful for visualizing what you are doing).
      - **save_prefix**: str. Prefix to use for filenames of saved pictures (only relevant if `save_to_dir` is set).
      - **save_format**: one of "png", "jpeg" (only relevant if `save_to_dir` is set). Default: "jpeg".

- **follow_links**: whether to follow symlinks inside class subdirectories (default: False).

- **Examples**:

Example of using **`.flow(X, y)`**:

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)

# compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied)
datagen.fit(X_train)

# fits the model on batches with real-time data augmentation:
model.fit_generator(datagen.flow(X_train, Y_train, batch_size=32),
                    samples_per_epoch=len(X_train), nb_epoch=nb_epoch)

# here's a more "manual" example
for e in range(nb_epoch):
    print 'Epoch', e
    batches = 0
    for X_batch, Y_batch in datagen.flow(X_train, Y_train, batch_size=32):
        loss = model.train(X_batch, Y_batch)
        batches += 1
        if batches >= len(X_train) / 32:
            # we need to break the loop by hand because
            # the generator loops indefinitely
            break
```

Example of using **`.flow_from_directory(directory)`**:

```
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        'data/train',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

model.fit_generator(
        train_generator,
        samples_per_epoch=2000,
        nb_epoch=50,
        validation_data=validation_generator,
        nb_val_samples=800)
```

Example of transforming images and masks together.

```
# we create two instances with the same arguments
```

```
data_gen_args = dict(featurewise_center=True,
                     featurewise_std_normalization=True,
                     rotation_range=90.,
                     width_shift_range=0.1,
                     height_shift_range=0.1,
                     zoom_range=0.2)
image_datagen = ImageDataGenerator(**data_gen_args)
mask_datagen = ImageDataGenerator(**data_gen_args)

# Provide the same seed and keyword arguments to the fit and flow methods
seed = 1
image_datagen.fit(images, augment=True, seed=seed)
mask_datagen.fit(masks, augment=True, seed=seed)

image_generator = image_datagen.flow_from_directory(
    'data/images',
    class_mode=None,
    seed=seed)

mask_generator = mask_datagen.flow_from_directory(
    'data/masks',
    class_mode=None,
    seed=seed)

# combine generators into one which yields image and masks
train_generator = zip(image_generator, mask_generator)

model.fit_generator(
    train_generator,
    samples_per_epoch=2000,
    nb_epoch=50)
```

Deep Learning

CNN: Predict new single Data-point

9.6.1 Predicting Cat or Dog

Before proceed, do some online research on what tools to use to make single predictions with CNN. The deep learning scientist spends a lot of his time doing research on how to implement models, or even sometimes on how to use them.

❒ Add a new sub-folder in the ***dataset*** folder called ***single_prediction***. This new folder contains two images, a cat and a dog.

☞ We need ***NumPy***. We will actually use a function by ***NumPy*** to ***pre-process*** the image that we are going to ***load***, so that it can be accepted by the ***predict()*** method.

```
# Part 3 - Making new predictions
import numpy as np
from tensorflow.keras.preprocessing import image
# from keras.preprocessing import image # for old versions


# we can also use 'utils ' module
# from keras import utils
# test_image = utils.load_img('dataset/single_prediction/cat_or_dog_2.jpg',target_size=(64, 64))
# test_image = utils.img_to_array(test_image)

test_image = image.load_img('dataset/single_prediction/cat_or_dog_1.jpg', target_size=(64, 64))
# test_image = image.load_img('dataset/single_prediction/cat_or_dog_1.jpg',target_size=(64, 64))
test_image = image.img_to_array(test_image)

test_image= np.expand_dims(test_image, axis = 0)
result = cnn_classifier.predict(test_image)
idx = training_set.class_indices
if result[0][0]== 1:
    prediction = 'dog'
else:
    prediction = 'cat'
```

❒ ***image*** module: Is the image module from Keras. (***from*** `keras.preprocessing` ***import image*** *# used in old version*).

```
from tensorflow.keras.preprocessing import image
```

☞ We can also use:

```
from keras import utils
```

It can do the same job.

❒ **`load_img()`**: To load our image on which we wanna make our prediction, we use **`load_img()`**. Here we specify the destination folder and the size for the image. Since we used $\mathbf{64 \times 64}$ image size to train our model, we have to set the same size for the new image.

```
test_image = image.load_img('dataset/single_prediction/cat_or_dog_1.jpg',target_size=(64, 64))
```

or

```
test_image = utils.load_img('dataset/single_prediction/cat_or_dog_2.jpg',target_size=(64, 64))
```

☞ Do not forget to ***specify*** the ***extension***, We have to include it, which is ***JPG***.

❒ ***Conversion to array:*** To convert our image into a 3D array we use

```
test_image = image.img_to_array(test_image)
```

or

```
test_image = utils.img_to_array(test_image)
```

☞ Remember, the input shape in the input layer of our CNN has three dimensions, each of 64×64 layer, because it's a colored image **`img_to_array()`** will allow to create this 3D array that will have the same format as the input shape in the input layer of our CNN.

❒ Using ***NumPy*** to add extra ***dimension***: We have to add a new dimension to our test image using ***`expand_dims()`*** function. Using 3-dimension, will get an error saying that we will need four dimensions instead of three dimensions.

```
test_image= np.expand_dims(test_image, axis = 0)
```

☞ What this ***dimension*** corresponds to?

➢ It corresponds to the ***batch*** because, in general, the functions of neural networks, like the ***predict function***, cannot accept a ***single input*** by itself, like the image we have here.

➢ It only accepts ***`inputs`*** in a ***batch***. Even if the ***`batch`*** contains ***`one input`***. The ***`input`*** must be in the ***`batch`***, and this ***`new dimension`*** that we are creating right now corresponds to the ***`batch`***. Whether there is ***one input*** or ***several inputs***.

➢ So here, we will have ***one batch*** of ***one input***, but then in general, we can have ***several batches*** of ***several inputs***, and we can apply the ***predict*** method on that.

☞ ***`axis`***: ***`axis`*** is to specify the position of the index of the dimension that we are adding.

➢ We need to add this ***dimension*** in the ***first position***, therefore, we will specify ***`axis = 0`***. because ***`axis = 0`*** means that that ***index*** of this new dimension we are adding is gonna have the ***first index***, that is ***index zero***.

❒ ***Next we predict the image:***

```
result = cnn_classifier.predict(test_image)
```

☞ We put the result of that single prediction in a new variable ***`result`***, The new single prediction, will be ***`1`*** or ***`0`*** (binary-classification).



☞ ***Next we check the indices of our training set:*** Does ***`1`*** correspond to cat or to dog? To check that we need to use following code:

```
idx = training_set.class_indices
```

```
In [10]: training_set.class_indices
Out[10]: {'cats': 0, 'dogs': 1}
```

➢ We can clearly see that cats correspond to ***`0`*** and dogs correspond to ***`1`***. Perfect, so that means that the prediction by our CNN model for this first image, ***`cat_or_dog_1.jpg`***, is correct because this image contains a dog.

☞ If you want to make it even more simple you can add following codes. Remember you need to first check out the ***`class_indices`*** indices, then you can use the following conditions:

```
idx = training_set.class_indices
if result[0][0]== 1:
    prediction = 'dog'
else:
    prediction = 'cat'
```

➢ Notice, this ***`result`*** is an array of two dimensions. **`result[0][0]`**, Used to get the **first row** and the **first column**.

☯ When we look at the test-set validation accuracy ***`val_accuracy`***., well, remember we obtained between ***`81`*** and ***`83%`***, so you know our model had ***`82%`*** chance to make ***`correct predictions`***, and that's how we got these good results.

All code at once (practiced)

```
# ========== ====  Convolutional Neural Network : CNN  ==== ============

# ---------- Install following  packages -------------
    # Install Theano
    # Install Tensorflow
    # Install Tensorflow
    # Install Keras


# ---------- Part 1 - Building the CNN ----------------
# Importing the Keras libraries and packages

from keras.models import Sequential     # to initialize as sequence-of-layers
from keras.layers import Convolution2D  # Convolution step for images
from keras.layers import MaxPooling2D   # not "MaxPool2D". Pooling step for images
from keras.layers import Flatten        # Flatenning step
from keras.layers import Dense          # ads fully-connected-layers to classic ANN

# initializing the CNN
cnn_classifier = Sequential()

# step 1 : Convolution - layer
cnn_classifier.add(Convolution2D(32, 3, 3, input_shape = (64, 64, 3), activation= "relu"))

# step 2 : Pooling  - layer
cnn_classifier.add(MaxPooling2D(pool_size = (2, 2)))

# improving step : Adding 2nd-Convolution and  2nd-Pooling layers
cnn_classifier.add(Convolution2D(32, 3, 3, activation= "relu"))
cnn_classifier.add(MaxPooling2D(pool_size = (2, 2)))

# step 3 : Flattening
cnn_classifier.add(Flatten())

# step 4 : ANN - full connection
cnn_classifier.add(Dense(units= 128, activation= "relu")) # fully connected layers
cnn_classifier.add(Dense(units= 1, activation= "sigmoid")) # output layer

# compile the NN
cnn_classifier.compile(optimizer= "adam", loss="binary_crossentropy", metrics = ["accuracy"])


# ---------- Part 2 - Image Preprocessing & fit CNN to our images ----------------

from keras.preprocessing.image import ImageDataGenerator
import math

# creating two data-generator/augmentation obgects for train and test data
        # here we specify the transform parameters

train_datagen = ImageDataGenerator(
                                   rescale=1./255,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True )

test_datagen = ImageDataGenerator(rescale=1./255)

# applying augmentation on training data : training folder path needed
                                       # target_size = input_shape (dimension of expected resized images)
                                       # batch_size is not related to no. of filters
training_set = train_datagen.flow_from_directory('dataset/training_set',
                                                 target_size=(64, 64),
                                                 batch_size=32,
                                                 class_mode='binary')

# applying augmentation on test data : test folder path needed
test_set = test_datagen.flow_from_directory('dataset/test_set',
                                            target_size=(64, 64),
                                            batch_size=32,
                                            class_mode='binary')

"""
cnn_classifier.fit_generator(training_set,
 # due to incompatibility between Tensorflow and Keras version, "samples_per_epoch", "nb_epoch", "nb_val_samples" may
not work
 # Then use "steps_per_epoch", "epochs",  "validation_steps" instead
                           # samples_per_epoch = 8000,
                           # nb_epoch=25,
                           # nb_val_samples = 2000,
                           steps_per_epoch = 250,  # training_set_size/batch_size
                           epochs = 25,
```

```
                         validation_data=test_set,
                         validation_steps = 62   # test_set_size/batch_size
                         )

"""


# In this case no need to explicitly specify training_set's or test_set's sample-size: i.e 8000, 2000 or 800, 200
cnn_classifier.fit_generator(
                         training_set,
                         steps_per_epoch=math.floor((training_set.samples)/(training_set.batch_size)),
                         epochs=25,
                         validation_data=test_set,
                         validation_steps=math.floor((test_set.samples)/(test_set.batch_size))
                         )

# history = model.fit_generator(train_gen,
#                                 steps_per_epoch=(train_gen.samples/batch_size),  # len(train_gen)
#                                 epochs=100,
#                                 validation_data=validation_gen,
#                                 validation_steps=(validation_gen.samples/batch_size),
#                                 callbacks=[checkpointer],
#                                 workers=4
#                                 )




# Part 3 - Making new predictions
import numpy as np
from tensorflow.keras.preprocessing import image
# from keras.preprocessing import image # for old versions


# we can also use 'utils ' module
# from keras import utils
# test_image = utils.load_img('dataset/single_prediction/cat_or_dog_2.jpg',target_size=(64, 64))
# test_image = utils.img_to_array(test_image)

test_image = image.load_img('dataset/single_prediction/cat_or_dog_2.jpg',target_size=(64, 64))
# test_image = image.load_img('dataset/single_prediction/cat_or_dog_1.jpg',target_size=(64, 64))
test_image = image.img_to_array(test_image)

test_image= np.expand_dims(test_image, axis = 0)
result = cnn_classifier.predict(test_image)

idx = training_set.class_indices
if result[0][0]== 1:
    prediction = 'dog'
else:
    prediction = 'cat'



# python prctc_cnn.py
```

☯ We can also use '**utils**' module

```
from keras import utils
test_image = utils.load_img('dataset/single_prediction/cat_or_dog_2.jpg',target_size=(64, 64))
test_image = utils.img_to_array(test_image)
```



cat_or_dog_1.jpg



cat_or_dog_2.jpg

Name	Type	Size	Value
idx	dict	2	{'cats':0, 'dogs':1}
prediction	str	1	cat
result	Array of float32	(1, 1)	[[0.41072974]]
test_datagen	preprocessing.im…	1	ImageDataGenerator object of keras.preprocessing.image module
test_image	Array of float32	(1, 64, 64, 3)	[[[[239. 239. 239.] [239. 239. 239.]
test_set	preprocessing.im…	1	DirectoryIterator object of keras.preprocessing.image module
train_datagen	preprocessing.im…	1	ImageDataGenerator object of keras.preprocessing.image module

Chapter 10

Dimensionality Reduction

Principal Component Analysis (PCA),
Linear Discriminant Analysis (LDA),
Kernel PCA

10.1 Dimensionality Reduction

❒ ***Dimensionality:*** The number of *input features*, *variables*, or *columns* present in a given *dataset* is known as *dimensionality*, and the process to reduce these features is called *dimensionality reduction*.

☞ Huge *number* of input *features* in various cases makes the predictive modeling task more *complicated*. It is very *difficult* to *visualize* or make predictions for the *training dataset* with a *high number* of *features*.

❒ ***Dimensionality Reduction:*** It is a way of *converting* the *higher dimensions* dataset into *lesser dimensions* dataset ensuring that it provides *similar information*. These techniques are widely used in machine learning for *obtaining* a better *fit predictive model* while solving the *classification* and *regression* problems.

☞ It is commonly used in the fields that deal with ***high-dimensional data,*** such as *speech recognition*, *signal processing*, *bioinformatics*, etc. It can also be used for *data visualization*, *noise reduction*, *cluster analysis*, etc.

❒ ***Curse of Dimensionality:*** If the dimensionality of the input dataset increases, any ***ML algorithm and model*** becomes more complex. As the number of *features increases*, the *number* of *samples* also gets *increased proportionally*, and the *chance of* ***OVERFITTING*** also *increases*.

❒ ***Benefits of Dimensionality Reduction:*** Some benefits of applying dimensionality reduction techniques are:

i. By reducing the *dimensions* of the *features*, the *space* required to *store the dataset* also gets *reduced*.
ii. *Less Computation* training time is required for reduced dimensions of features.
iii. Reduced dimensions of features of the dataset *help in visualizing* the data quickly.
iv. It *removes* the *redundant features* (if present) by taking care of ***multicollinearity***.

❒ ***Disadvantages of dimensionality Reduction:*** Some disadvantages of applying the dimensionality reduction are:

i. Some *data* may be *lost* due to dimensionality reduction.
ii. In the *PCA* dimensionality reduction technique, sometimes the ***principal components*** *required* to consider are *unknown*.

10.2 Approaches of Dimension Reduction

There are two ways to apply the dimension reduction technique, which are: ***Feature Selection*** and ***Feature Extraction***.

❒ ***Feature Selection:*** *Feature selection* is the process of selecting the *subset* of the *relevant features* and leaving out the *irrelevant features* present in a dataset to build a *model* of *high accuracy*. In other words, it is a way of ***selecting the Optimal Features*** from the input dataset.

☞ Three methods are used for the feature selection:

[1]. ***Filters Methods:*** In this method, the dataset is filtered, and a subset that contains only the relevant features is taken. Some common techniques of filters method are:

a) **Correlation**
b) **Chi-Square** Test
c) **ANOVA**
d) **Information Gain,** etc.

[2]. ***Wrappers Methods:*** In this method, some features are fed to the ML model, and evaluate the performance. The performance decides whether to add those features or remove to increase the accuracy of the model. ***Wrappers Methods*** are more accurate than the *Filtering Method* but *complex* to work. Some common techniques of wrapper methods are:

a) **Forward** Selection
b) **Backward** Selection
c) **Bi-directional** Elimination

[3]. ***Embedded Methods:*** Embedded methods check the different training iterations of the machine learning model and evaluate the importance of each feature. Some common techniques of Embedded methods are:

a) **LASSO**
b) **Elastic Net**
c) **Ridge Regression,** etc.

❒ ***Feature Extraction:*** Feature extraction is the process of *transforming* the *space* containing *many dimensions* into *space* with *fewer dimensions*. This approach is useful when we want to *keep* the *whole information* but use *fewer resources* while ***processing*** the ***information***. Some common feature extraction techniques are:

[1]. *Principal Component Analysis* (PCA)
[2]. *Linear Discriminant Analysis* (LDA)
[3]. Kernel ***PCA***
[4]. Quadratic Discriminant Analysis

The first three techniques we are gonna discuss in this chapter.

10.3 Principal Component Analysis (PCA)

PCA is considered to be one of the *most* used *unsupervised algorithms* and can be seen as the most *popular dimensionality reduction* ***algorithm***.

❒ ***PCA*** is a *statistical process* that *converts* the observations of *correlated features* into a set of *linearly uncorrelated features* with the help of *orthogonal transformation*. These new *transformed features* are called the **`Principal Components`**.

☞ It is a technique to draw strong patterns from the given dataset by reducing the variances. ***PCA*** is used for operations such as:

- *Visualization*
- *Feature extraction*
- *Noise filtering*

It can also be seen in algorithms used for

- *Stock market predictions* and
- *Gene* data *analysis*.

❒ ***Goal Of PCA:*** The goal of PCA is to *identify* and *detect* the *correlation* between *variables*.

- Identify *patterns* in *data*
- *detect* the *correlation* between *variables*

☞ If there is a **strong correlation** found then you could ***Reduce*** the ***Dimensionality***.

☞ Find the *directions* of *maximum variance* in *high dimensional data* and then you *project* it into a *smaller dimensional subspace* while retaining most of the information

☞ The goal of ***PCA*** to *reduce* the *dimensions* of a ***d-dimensional dataset*** by projecting onto a ***k-dimensional*** subspace where k is less than d ($k < d$)and for a

❒ ***Steps for the PCA:*** That the main functions of the PCA algorithm are followed by the following steps:

[1]. ***Standardize*** the data.
[2]. Obtain the ***Eigenvectors*** and ***Eigenvalues*** from the *covariance matrix* or *correlation matrix*, or perform **Singular Vector Decomposition**.
[3]. ***Sort eigenvalues*** in *descending* order and choose the ***k-eigenvectors*** that correspond to the ***k largest eigenvalues*** where ***k*** is the number of ***dimensions*** of the ***new*** feature ***subspace*** ($K < d$).
[4]. Construct the ***projection matrix W*** from the selected ***K*** **eigenvectors**.
[5]. Transform the ***original*** **dataset** ***X*** via ***W*** to obtain a ***K*****-dimensional** feature **subspace** ***Y***

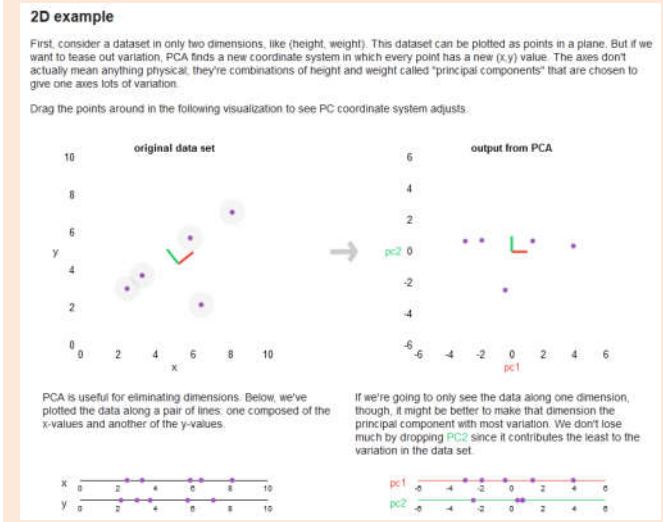
https://plot.ly/ipython-notebooks/principal-component-analysis/

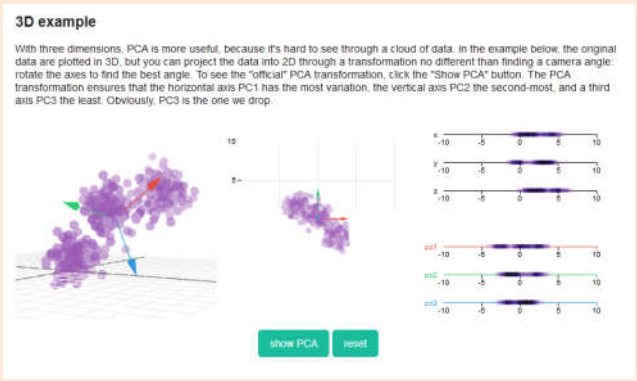
❒ ***Visualization of PCA:*** The visualization of PCA will really helpful if we visit the following link. It's going to take us to this page where we can actually view it in 2D and 3D examples.

https://setosa.io/ev/principal-component-analysis/

☞ With ***PCA*** in a ***2D*** you can start to see the *relationship* in how *PCA is playing* out among the *variables* in the *data*.
- You can also *drag* the *data point around* to see the ***PCA*** coordinates *adjust* within the *system*.

☞ The ***3D*** example is also very *helpful*. You can actually see the *relationship* the *data* within this model and comparing it to the ***2D*** within the *higher dimensional space*. Obviously it can be a much *easier visualization*.
- In *3D plot*, we can actually *move* the *model*.





NOTES:

- PCA is not like ***linear regression*** although it may look like it because ***rather than attempting to predict*** the values, PCA is attempting to ***learn about the relationship*** between **x** and **y** values quantified by finding a ***list*** of ***principal axes***.
  - ***learn about the relationship*** between **x** and **y** values
  - Find a ***list*** of ***principal axes***
- To understand PCA, the best ways is to look at the visualizations (the link is given above).
- ***PCA does have a weakness:*** It is highly ***affected*** by ***outliers*** in the ***data*** but PCA is considered to be one of the most used and it's extremely popular.

10.4 PCA in Python: part 1 – Problem description

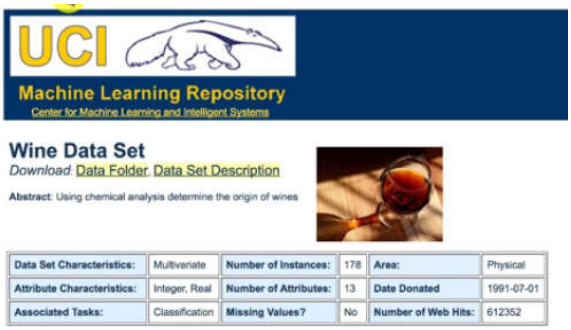
- In ***dimensionality reduction*** there are **two** techniques ***feature selection*** and ***feature extraction***.
  - We did feature selection in ***Chapter 2: Machine Learning***, when we implemented the ***backward elimination*** model to select the ***most relevant features*** of our ***feature- matrix***.

Feature Selection	Feature Extraction
Backward Elimination	PCA
Forward Selection	LDA
Bidirectional Elimination	Kernel PCA
Score Comparison	

- Now we are starting ***feature extraction*** technique of ***dimensionality reduction***, and ***PCA: principal component analysis*** is one of feature extraction techniques.
  - From the ***m*** **independent variables** of your dataset, ***PCA*** extracts $p \leq m$ **new independent variables** that explain the ***most*** the ***variance*** of the ***dataset***, regardless of the ***dependent variable***.
    - i.e. PCA will extract a smaller number of your independent variables that are going to be new independent variables (like new dimensions) and these new independent variables explain the most the variance of your data set.
- Notice that, we are not considering the ***dependent variable (DV)*** in the ***PCA*** model, for this reason ***PCA*** considered as an ***unsupervised model***.
- Recall, in ***Chapter 2*** and ***Chapter 3*** we worked with ***one*** or ***two independent variables***, because we needed a ***graphic visualization*** of our ***results***.
- ***PCA will help us to visualize the results:*** Using ***PCA dimensionality reduction technique***, we'll reduce the dimension of our dataset by taking relevant independent variables that will explain the most the variance of our dataset.
  - Since we can ***reduce*** this number of ***independent variables***, we can end up with ***two*** or ***three independent variables*** and therefore ***visualize*** the ***results***.

Problem description: We'll apply **PCA** to our **Logistic-Regression** model that we built in **Chapter 3: Classification**. So we are applying ***PCA*** to a ***Classification problem***.

- Following is a very ***famous dataset***, well-known in the ***machine learning literature*** and that you can find on the ***UCI Machine Learning Repository***.



dataSet - DataFrame

Index	Alcohol	Malic_Acid	Ash	Ash_Alcanity	Magnesium	Total_Phenols	Flavanoids	Nonflavanoid_Phenols	Proanthocyanins	Color_Intensity	Hue	OD280	Proline	Customer_Segment
0	14.23	1.71	2.43	15.6	127	2.8	3.06	0.28	2.29	5.64	1.04	3.92	1065	1
1	13.2	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.4	1050	1
2	13.16	2.36	2.67	18.6	101	2.8	3.24	0.3	2.81	5.68	1.03	3.17	1185	1
3	14.37	1.95	2.5	16.8	113	3.85	3.49	0.24	2.18	7.8	0.86	3.45	1480	1
4	13.24	2.59	2.87	21	118	2.8	2.69	0.39	1.82	4.32	1.04	2.93	735	1
5	14.2	1.76	2.45	15.2	112	3.27	3.39	0.34	1.97	6.75	1.05	2.85	1450	1
6	14.39	1.87	2.45	14.6	96	2.5	2.52	0.3	1.98	5.25	1.02	3.58	1290	1
7	14.06	2.15	2.61	17.6	121	2.6	2.51	0.31	1.25	5.05	1.06	3.58	1295	1
8	14.83	1.64	2.17	14	97	2.8	2.98	0.29	1.98	5.2	1.08	2.85	1045	1
9	13.86	1.35	2.27	16	98	2.98	3.15	0.22	1.85	7.22	1.01	3.55	1045	1
10	14.1	2.16	2.3	18	105	2.95	3.32	0.22	2.38	5.75	1.25	3.17	1510	1
11	14.12	1.48	2.32	16.8	95	2.2	2.43	0.26	1.57	5	1.17	2.82	1280	1
12	13.75	1.73	2.41	16	89	2.6	2.76	0.29	1.81	5.6	1.15	2.9	1320	1

- ***Independent Variables:*** The independent variables are: ***Alcohol***, ***Malic_Acid***, ***Ash***, ***Ash_Alcanity***, ***Magnesium***, ***Total_Phenols***, ***Flavanoids***, ***Nonflavanoid_Phenols***, ***Proanthocyanins***, ***Colorjntensity***, ***Hue***, ***OD280*** and ***Proline***

- ***Dependent Variables:*** Last variable " ***Customer_Segment*** " is the ***dependent variable***. In the original data set this dependent variable is not called " ***Customer_Segment*** " this is actually "***origin_of_the_wine***".

Imagine that, a business owner gathered all the information of these ***independent variables*** here that are ***chemical's informations*** of several ***wines***.

- Also this business owner applied some ***clustering technique*** to find some ***segments of customers*** that like a ***specific wine*** depending on the ***information of the wine***.
- This business owner identified ***three segments*** of customers. Numbered: as 1, 2, 3 in ***Customer_Segment*** column.
- So basically this business owner found ***three types*** of wines ***each type*** of one corresponding to ***one segment of customers*** and therefore ***three segments of customers***.

- ***Goal of our model:*** This business owner can take all these information of the wines (all independent variables) and the information about the customer segments (***Customer_Segment*** as one dependent variable) and make a ***classification*** model like ***logistic regression***.
  - Then for each ***new wine*** the model can predict to ***which customer segment*** it should recommend this ***new wine***.
  - So our logistic regression model is going to return the customer segment that each ***new wine*** should be recommended to.

- ***Role of PCA in our model:*** But to have a clear ***visual*** look at the ***prediction regions*** and the ***prediction boundary*** of the classification model we use PCA as dimensionality reduction technique.
  - We'll ***reduce*** the ***dimensions*** i.e. we gonna find the most ***important two independent-variables*** that ***explain*** the ***most*** the ***variance*** in data.
  - Then we use those ***two independent-variables*** to ***visualize*** the ***prediction regions*** and the ***prediction boundary***.

- ***Principal Components:*** These ***extracted features*** *(most important two independent-variables)* by ***PCA*** are called the ***principal components***.

10.5 PCA in Python: part 2 – Data pre-processing

We'll ***copy*** all code from ***`logistic.py`*** source file (classification-template for logistic regression) and ***paste*** it into a new ***`.py`*** file called ***`logistic_rgsn_pca.py`***.

❒ By applying **PCA** inside this *logistic regression model* we end up with *two independent variables* that explain the *most variance* in the data, therefore we will be able to ***visualize*** the ***results***.

❒ Now will change a few things and then we will implement PCA.

☞ ***Change dataset:*** We first change our *.csv* dataset. Now we use a *multidimensional* (more than 3 independent variables) *dataset*. We'll use the wine.csv file as our dataset.

```
dataSet = pd.read_csv("Wine.csv")
```

☞ ***Fixing the Index:*** We **change** the **indexes** in the ***`feature-matrix`*** and also fix the index of the ***dependent*** variable.

```
# Data Extract
dataSet = pd.read_csv("Wine.csv")
X = dataSet.iloc[:, :13].values
y = dataSet.iloc[:, 13].values
```

X - NumPy object array

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	14.23	1.71	2.43	15.6	127	2.8	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	13.2	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.4	1050
2	13.16	2.36	2.67	18.6	101	2.8	3.24	0.3	2.81	5.68	1.03	3.17	1185
3	14.37	1.95	2.5	16.8	113	3.85	3.49	0.24	2.18	7.8	0.86	3.45	1480
4	13.24	2.59	2.87	21	118	2.8	2.69	0.39	1.82	4.32	1.04	2.93	735
5	14.2	1.76	2.45	15.2	112	3.27	3.39	0.34	1.97	6.75	1.05	2.85	1450
6	14.39	1.87	2.45	14.6	96	2.5	2.52	0.3	1.98	5.25	1.02	3.58	1290
7	14.06	2.15	2.61	17.6	121	2.6	2.51	0.31	1.25	5.05	1.06	3.58	1295
8	14.83	1.64	2.17	14	97	2.8	2.98	0.29	1.98	5.2	1.08	2.85	1045

y - NumPy object array

	0
0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1

☞ ***Split the data:*** We set ***`20%`*** of observations for the ***`test_set`*** and ***`80%`*** for ***`training_set`***.

```
# Data Split
from sklearn.model_selection import train_test_split
# 0.20 test_size means "1/5"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.20, random_state = 0)
```

☞ ***Feature-scaling:*** **Features scaling** must be applied when we apply *dimensionality reduction techniques* like ***PCA*** or ***LDA***.

```
# Feature-Scaling
from sklearn.preprocessing import StandardScaler
#  y need not to be scaled.
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)
```

☞ ***Fit dataset to Logistic regression:*** It will be used after applying ***PCA***.

✋ ***When we apply PCA to a model:*** Remember, we apply any ***Dimensional Reduction Technique*** (like PCA, LDA, Kernel-PCA), right *after* the **Data Processing** phase (right after the feature is getting scaled) and just ***before fitting*** the ***logistic regression*** model or any other ***Classification Model***.

10.6 PCA in Python: part 3 – applying PCA

In this part we comment-out the following sections: **fitting logistic model**, **prediction**, **confusion matrix**, **visualization-part**. Because the independent variable are not fixed yet. Here's what we gonna do next:

i. ***Import*** PCA

ii. ***First*** time applying PCA for all ***13 independent variables*** we'll ***examine*** all 13 variables and their ***impact*** (i.e. variance) on the data.

- Find the ***most two important independent*** variables, i.e ***Principle Component - PC***.

iii. Then we'll apply ***PCA*** second time for ***2 independent variables***. (after finding the no. of ***PC***s we just fix the number in ***PCA*** object).

```
# Library
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Wine.csv")
X = dataSet.iloc[:, :13].values
y = dataSet.iloc[:, 13].values


# Data Split
from sklearn.model_selection import train_test_split
# 0.20 test_size means "1/5"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.20, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler
#  y need not to be scaled.
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)

# Applying PCA



# # Fit dataset to Logistic regression
# from sklearn.linear_model import LogisticRegression # import class
# # instead of "regressor" we now use "classifier"
# classifer = LogisticRegression(random_state= 0) # create object
# classifer.fit(X_train, y_train) # fit the dataset

# # Predict
# y_prd = classifer.predict(X_test)

# # Making the confusion matrix use the function "confusion_matrix"
# # Class in capital letters, functions are small letters
# from sklearn.metrics import confusion_matrix
# cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# # parameters of cm: y_true: Real values, y_pred: Predicted value




# # Visualising the Training set results
# from matplotlib.colors import ListedColormap
# X_set, y_set = X_train, y_train
# X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
#                      np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
# plt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
#              alpha = 0.30, cmap = ListedColormap(('red', 'green', 'orange')))
# plt.xlim(X1.min(), X1.max())
# plt.ylim(X2.min(), X2.max())
# for i, j in enumerate(np.unique(y_set)):
#     plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
#                 c = ListedColormap(('red', 'green', 'orange'))(i), label = j)
# plt.title('Logistic Regression (Training set)')
# plt.xlabel('PC_1')
# plt.ylabel('PC_2')
# plt.legend()
# plt.show()

# # Visualising the Test set results
# from matplotlib.colors import ListedColormap
```

```
# X_set, y_set = X_test, y_test
# X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
#                      np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
# plt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
#              alpha = 0.30, cmap = ListedColormap(('red', 'green', 'orange')))
# plt.xlim(X1.min(), X1.max())
# plt.ylim(X2.min(), X2.max())
# for i, j in enumerate(np.unique(y_set)):
#     plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
#                 c = ListedColormap(('red', 'green', 'orange'))(i), label = j)
# plt.title('Logistic Regression (Test set)')
# plt.xlabel('PC_1')
# plt.ylabel('PC_2')
# plt.legend()
# plt.show()

# # python prctc_logistic_rgsn_pca.py
```

❒ ***Import PCA:*** First we import the right package and more precisely the right class to use ***PCA***. Then we create an ***object*** of this ***class*** and we'll apply the ***fit_transform*** and ***transform*** methods respectively on the ***training set*** and the ***test-set***.

```
from sklearn.decomposition import PCA
dcmPose_pca = PCA(n_components= None)

X_train = dcmPose_pca.fit_transform(X_train)
X_test = dcmPose_pca.transform(X_test)
```

☞ By applying the ***fit_transform*** method to this ***dcmPose_pca*** object can see how the ***training set*** is ***structured*** and therefore how ***it*** can ***extract*** some ***new independent variables*** that explain the ***most variance***. Once the object is fitted to the **training-set**, then use the ***transform*** method to transform the **test-set** that is ***X_test***.

That will ***fit*** our object the ***training set*** and ***transform*** it at the same time i.e. ***extracting*** all the ***PCs***.

☞ ***n_components:*** int, float or 'mle', default=None

➲ Number of components to keep. if ***n_components*** is ***not set*** all components are ***kept***. It is the number of ***principal components***. Basically this is the ***number*** of ***extracted features*** you want to get to explain the most the variance. You choose the number depending on how much variance you would like to be explained.

➲ For now we set : **n_components= None**, because we know we want to get ***two*** **principal components** to visualize the result, but we don't know how much variance these two components explain.

➲ We need to make sure that the two first principal components (PCs) that explain the most variants don't explain the low variance.

➲ We used ***None*** because we'll create a vector called ***explnd_vrince*** by using a ***PCA*** attribute ***explained_variance_ratio_*** to see the ***cumulative explained variance*** of different ***principal components***.

❒ ***explained_variance vector:*** This ***explnd_vrince*** vector going to contain the ***percentage*** of ***variance explained*** by ***each*** of the ***principal components*** that we ***extracted*** here.

```
explnd_vrince = dcmPose_pca.explained_variance_ratio_
```

☞ ***explained_variance_ratio_*** returns the list of all the principal components and we will get the percentage of variance explained by each of them.

```
from sklearn.decomposition import PCA
dcmPose_pca = PCA(n_components= None)
X_train = dcmPose_pca.fit_transform(X_train)
X_test = dcmPose_pca.transform(X_test)
explnd_vrince = dcmPose_pca.explained_variance_ratio_
```

❒ Now we can have a look at this returned explained variance, ***explnd_vrince*** of all the principal components.

☞ In the ***explained variance vector***, since we originally had 13 independent variables, it extracted 13 principal components.

☞ But these are not the ***original independent variables*** that we had in our ***data-set***. These are the new ***extracted independent variables*** that explained the ***most*** the ***variance***.

➲ You can see they are ***ranked***, from the ***first*** (**0th**) ***PC*** that explains the ***most*** the ***variance*** down to the 1***2th*** and last PC that explains the ***least*** the ***variance***.

➲ First PC that will explain **37%** of the variance. And 2nd PC will explain **19%** of the variance. So if we take ***first two PCs*** that will explain $37 + 19 = 56\%$ of the ***variance***.

➲ For 3D visualization, if we take the top three PCs that will explain $37 + 19 + 11 = 67\%$

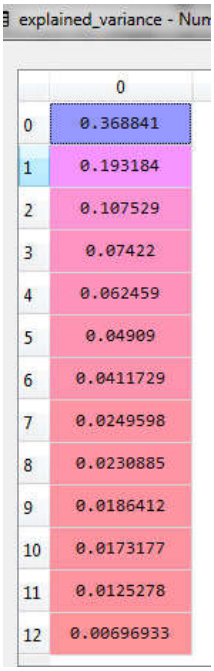
☞ We chose the first two PCs because we want to get **2D visualization** of the result

➲ Explaining 56% of the variance is pretty good to make a classification out of it.

☞ Now we edit above PCA code-segment and replace `n_components= None` with `n_components= 2`.

```
from sklearn.decomposition import PCA
# dcmPose_pca = PCA(n_components= None)
dcmPose_pca = PCA(n_components= 2)
X_train = dcmPose_pca.fit_transform(X_train)
X_test = dcmPose_pca.transform(X_test)
explnd_vrince = dcmPose_pca.explained_variance_ratio_
```

☞ Then the PCA object ***dcmPose_pca*** will return 2-most important independent variables (represents 2 PCs). And ***X_train*** and ***X_test*** will be transformed into ***2D-feature-matrices*** from ***13D-feature-matrices***.



☞ These first ***two PCs*** are going to be the two new ***independent variables*** of our dataset (originally we had 13 independent variables).

☞ When we have a look at ***X_train*** and ***X_test*** right now well it contains only two independent variables that are of course the top ***two principal components*** that explain the most ***variance***.



Since these two retuned ***independent variables*** are already scaled, we're ready to ***fit*** the ***logistic regression model*** and ***visualize*** its results in ***2D***.

10.7 PCA in Python: part 4 – Logistic model & Visualize the result

We are ready to ***fit*** a ***logistic regression model*** to classify ***new wines*** and tell in which ***segment of customers*** they ***belong to***. At the same time we'll ***predict*** the ***tested results*** to ***evaluate*** the model ***performance*** using the ***confusion matrix***.

❒ So basically all things are pre-coded, now we just execute the following codes:

```
# Fit dataset to Logistic regression
from sklearn.linear_model import LogisticRegression # import class
# instead of "regressor" we now use "classifier"
classifer = LogisticRegression(random_state= 0) # create object
classifer.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = classifer.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
# Class in capital letters, functions are small letters
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value
```

❒ ***Confusion matrix:***

	0	1	2
0	14	0	0
1	1	15	0
2	0	0	6

☞ Here we see the accuracy of our ***classification*** model with the ***two extracted principal components***.

☞ Since these ***PCs*** were chosen to explain the ***most variance*** that they ***explained*** around ***60 percent*** of the ***variance***. ***Therefore*** we should get ***good accuracy*** because our two ***PCs*** are actually the ***directions*** of ***maximum variance*** in our dataset.

☞ It's a ***confusion matrix*** with ***three classes***, so it won't be a **confusing matrix** of **2x2** but since we now have ***three classes*** this will be a **confusion matrix** of **3x3**.

☞ You can see we get excellent results because in the diagonal we get all the correct prediction.

☞ ***14 correct predictions*** of the class zero for the ***customer*** of ***segment number one***.

- ***15 correct predictions*** of the **customer segment number two**.
- ***6 correct predictions*** of the **customer segment number three**.

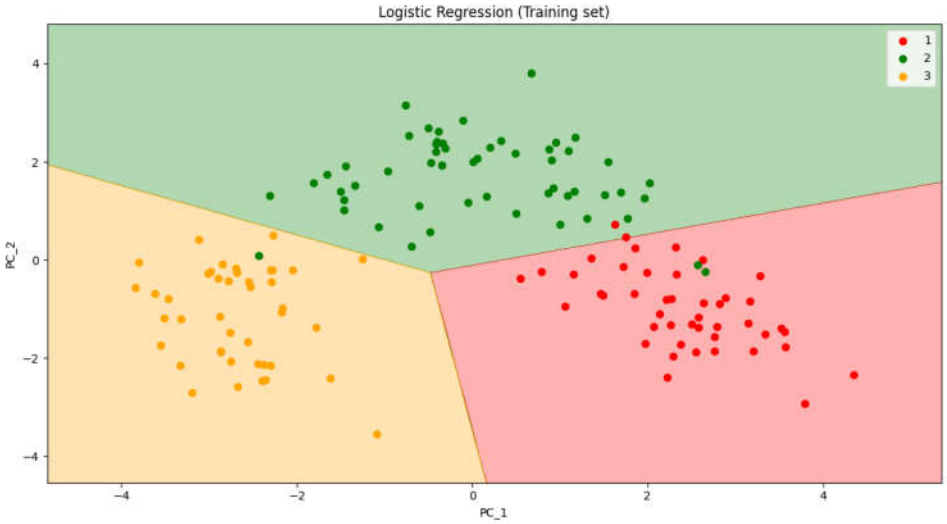
☝ So using PCA we got a very good prediction. ***35 correct*** prediction out of ***36 total*** observations. i.e. $35/37 = 97.2\%$ accuracy.

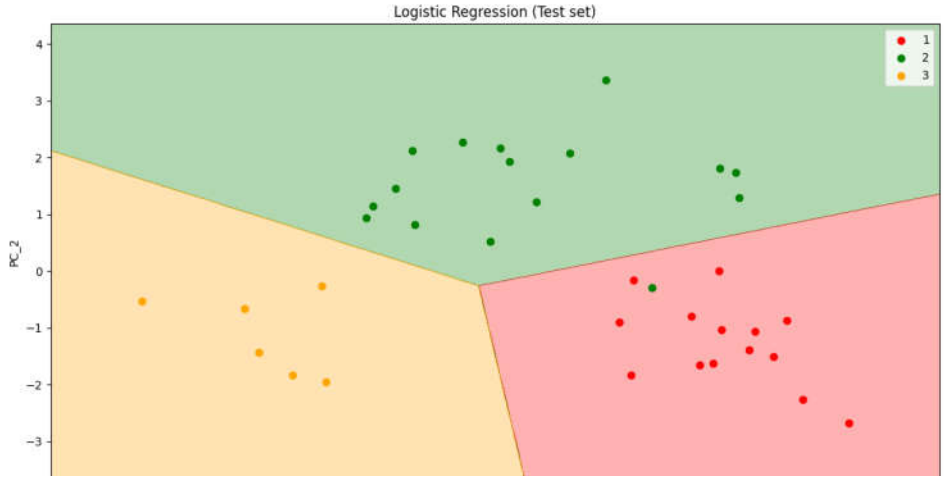
❒ ***Visualization:*** We edit the label of the variables.

```
plt.xlabel('PC_1')
plt.ylabel('PC_2')
```

☞ And we use 3-colors to visualize our three customer_segments.

```
ListedColormap(('red', 'green', 'orange'))
```





All code at once (practiced version)

```
# --------- Dim Reduction (Feature extraction): Princple Component Analysis (PCA) ---------
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Wine.csv")
X = dataSet.iloc[:, :13].values
y = dataSet.iloc[:, 13].values


# Data Split
from sklearn.model_selection import train_test_split
# 0.20 test_size means "1/5"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.20, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler
#  y need not to be scaled.
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)

# Applying PCA
from sklearn.decomposition import PCA
# dcmPose_pca = PCA(n_components= None)
dcmPose_pca = PCA(n_components= 2)
X_train = dcmPose_pca.fit_transform(X_train)
X_test = dcmPose_pca.transform(X_test)
explained_variance = dcmPose_pca.explained_variance_ratio_

# Fit dataset to Logistic regression
from sklearn.linear_model import LogisticRegression # import class
# instead of "regressor" we now use "classifier"
classifer = LogisticRegression(random_state= 0) # create object
classifer.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = classifer.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
# Class in capital letters, functions are small letters
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value


# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.30, cmap = ListedColormap(('red', 'green', 'orange')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'orange'))(i), label = j)
pLt.title('Logistic Regression (Training set)')
pLt.xlabel('PC_1')
pLt.ylabel('PC_2')
pLt.legend()
pLt.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
```

```
pLt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.30, cmap = ListedColormap(('red', 'green', 'orange')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'orange'))(i), label = j)
pLt.title('Logistic Regression (Test set)')
pLt.xlabel('PC_1')
pLt.ylabel('PC_2')
pLt.legend()
pLt.show()


# python prctc_logistic_rgsn_pca.py
```

All code at once (instructor version)

```
# PCA

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Wine.csv')
X = dataset.iloc[:, 0:13].values
y = dataset.iloc[:, 13].values

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Applying PCA
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
explained_variance = pca.explained_variance_ratio_

# Fitting Logistic Regression to the Training set
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green', 'blue')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'blue'))(i), label = j)
```

```
plt.title('Logistic Regression (Training set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green', 'blue')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'blue'))(i), label = j)
plt.title('Logistic Regression (Test set)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.show()
```

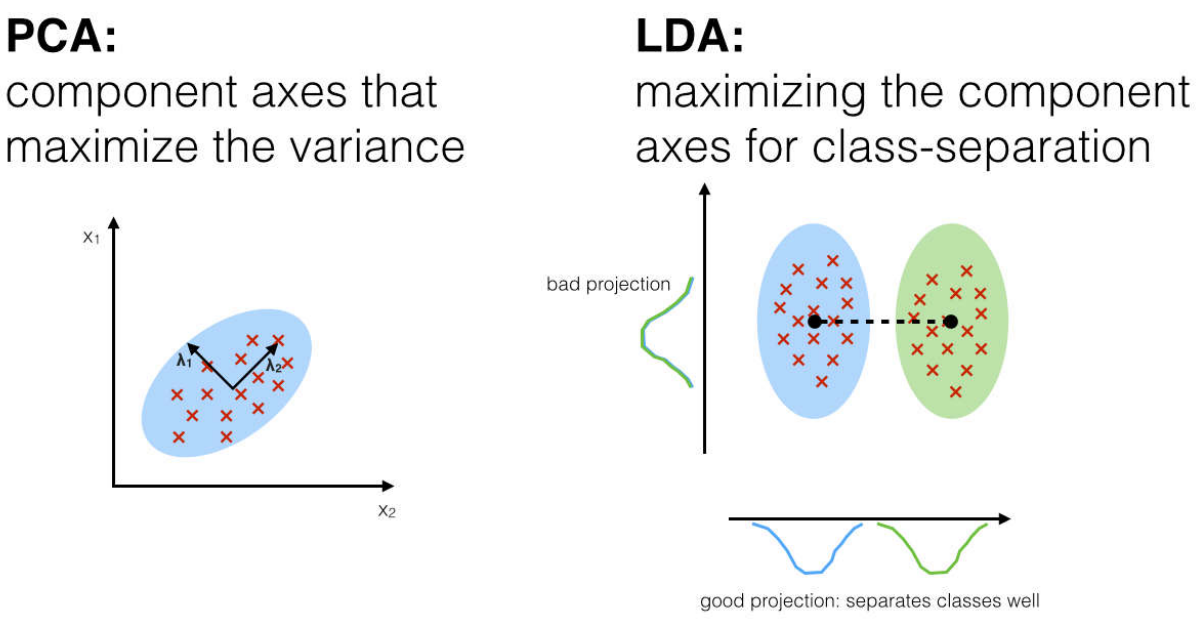
10.8 Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) bit similar to Principle Component Analysis (PCA).

- ☞ LDA is commonly used as a dimensionality reduction technique.
- ☞ It's used in the ***pre-processing*** step for ***Pattern Classification*** and ***Machine Learning Algorithms***.
- ☞ Its goal is to project a data set onto a lower dimensional space.

❒ ***Comparison between PCA & LDA***

- ☞ LDA differs because in addition to finding the ***component axises*** with ***LDA*** we are interested in the **axes** that **maximize** the **separation** between **multiple classes**.
  - ➲ In PCA we are just finding the principal components (the axes) within the data.
- ☞ The ***goal*** of ***LDA*** is to project a ***feature space*** (a dataset ***n-dimensional*** samples) onto a small subspace ***subspace k***(where $k \leq n - 1$) while maintaining the class-discriminatory information.
- ☞ Both ***PCA*** and ***LDA*** are **linear transformation techniques** used for ***dimensional reduction***.
- ☞ ***PCA*** is described as ***unsupervised*** but ***LDA*** is ***supervised*** because of the relation to the ***dependent variable***.



☞ We can see the main differences between ***PCA*** and ***LDA*** from above visualization. In ***LDA*** we're looking for the ***class separation*** within the data. Key points are:

- ***LDA*** is that ***class separation*** technique
- ***LDA*** is a ***supervised learning*** technique

1. Compute the d-dimensional mean vectors for the different classes from the dataset.
2. Compute the scatter matrices (in-between-class and within-class scatter matrix).
3. Compute the eigenvectors ($\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_d$) and corresponding eigenvalues ($\boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2, \ldots, \boldsymbol{\lambda}_d$) for the scatter matrices.
4. Sort the eigenvectors by decreasing eigenvalues and choose k eigenvectors with the largest eigenvalues to form a $d \times k$ dimensional matrix $\boldsymbol{W}$ (where every column represents an eigenvector).
5. Use this $d \times k$ eigenvector matrix to transform the samples onto the new subspace. This can be summarized by the matrix multiplication: $\boldsymbol{Y} = \boldsymbol{X} \times \boldsymbol{W}$ (where $\boldsymbol{X}$ is a $n \times d$-dimensional matrix representing the n samples, and $\boldsymbol{y}$ are the transformed $n \times k$-dimensional samples in the new subspace).

https://sebastianraschka.com/Articles/2014_python_lda.html

Summarizing the LDA approach in 5 steps. Listed below are the 5 general steps for performing a linear discriminant analysis (similar to PCA);

[1]. Compute the **d-dimensional mean vectors** for the different classes from the dataset.

[2]. Compute the **scatter matrices** (in-between-class and within-class scatter matrix).

[3]. Compute the **eigenvectors** ($e_1, e_2, \ldots, e_d$) and corresponding eigenvalues ($\lambda_1, \lambda_2, \ldots, \lambda_d$) for the **scatter matrices**.

[4]. ***Sort*** the ***eigenvectors*** by ***decreasing eigenvalues*** and choose k ***eigenvectors*** with the ***largest eigenvalues*** to form a $\boldsymbol{d} \times \boldsymbol{k}$ ***dimensional matrix*** $\boldsymbol{W}$ (where every ***column*** represents an ***eigenvector***).

[5]. Use this $\boldsymbol{d} \times \boldsymbol{k}$ ***eigenvector matrix*** to transform the ***samples*** onto the ***new subspace***. This can be ***summarized*** by the matrix ***multiplication***: $\boldsymbol{Y} = \boldsymbol{X} \times \boldsymbol{W}$ (where $\boldsymbol{X}$ is a $\boldsymbol{n} \times \boldsymbol{d}$-**dimensional matrix** representing the $\boldsymbol{n}$ samples, and $\boldsymbol{y}$ are the transformed $\boldsymbol{n} \times \boldsymbol{k}$-**dimensional samples** in the **new subspace**).

❒ ***Additional reading:*** **Linear Discriminant Analysis – Bit by Bit** by ***Sebastian Raschka*** (Aug 3, 2014).

[RSS]

Linear Discriminant Analysis

– Bit by Bit

Aug 3, 2014
by Sebastian Raschka

https://sebastianraschka.com/Articles/2014_python_lda.html

PCA vs LDA :

❒ PCA has no concern with the class labels. PCA summarizes the feature set without relying on the output.

☞ PCA tries to find the directions of the maximum variance in the dataset. In a large feature set, there are many features that are merely duplicate of the other features or have a high correlation with the other features. Such features are basically redundant and can be ignored.

☞ The role of PCA is to find such highly correlated or duplicate features and to come up with a new feature set where there is minimum correlation between the features or in other words feature set with maximum variance between the features. Since the variance between the features doesn't depend upon the output, therefore PCA doesn't take the output labels into account.

❒ Unlike PCA, LDA tries to **reduce dimensions** of the **feature** set while **retaining** the **information** that **discriminates output classes**. LDA tries to find a **decision boundary around** each **cluster** of a **class**. It then **projects** the **data points** to **new dimensions** in a way that the **clusters** are as **separate from each other** as possible and the individual elements within a cluster are as close to the centroid of the cluster as possible.

☞ The new dimensions are ranked on the basis of their ability to maximize the distance between the clusters and minimize the distance between the data points within a cluster and their centroids. These new dimensions form the linear discriminants of the feature set.

10.9 LDA in Python: Logistic model & Visualize the result

LDA is another technique for feature extraction. We consider the same **`Wine.csv`** dataset from **PCA**. We are going to ***extract*** some ***new independent variables*** that will ***reduce*** the ***dimensionality*** of our dataset as we did in **PCA**.

PCA feature extraction technique reduced the dimensionality of our problem by **extracting** the **variables** that **explain** the **most variants**.

LDA is quite different. Here, we are extracting some ***new independent variables*** (from the n ***independent variables*** of your dataset ***LDA*** extracts $p \leq n$ *new independent variables*) that will ***separate*** the ***most the classes*** of the ***dependent variable***.

The fact that the DV (dependent variable) is considered makes LDA a supervised model.

LDA is a ***Supervised Feature-Extraction*** model: Since ***LDA*** **considers** the **classes** of the **dependent variable**. LDA is going to extract the ***`independent variables`*** that ***`separate`*** the ***`most`*** the ***`classes`*** and the classes are information related to the dependent variable. i.e. ***LDA*** works with ***dependent variable*** to proceed ***feature extraction***.

That makes ***LDA*** *supervised dimensionality reduction model* (on the other hand, PCA was unsupervised because we didn't consider the dependent variable).

Problem Description: We are going to work with the same business problem *"the wine chemical component* ***`Customer_Segment`*** *problem"* and we're going to see if ***LDA*** beat the ***`accuracy`*** obtained with ***PCA*** that is ***97%***, i.e we will see if we get a ***perfect accuracy*** of ***100%*** (no incorrect predictions).

We will take the PCA code that we just wrote in the previous section and we will have very few things to change.

Data preprocessing: This is going to be the **same** as in **PCA** because we just need a training set and a test set on which we apply features screening.

- We can already execute that without changing anything.
- We'll have our **original data set** with all the chemical information about the different Wines and the dependent variable that contains the different segments of customers that already clustered.
- We have our **training set** that is **scaled** but so far contains the **13 independent variables**.

We're going to get a **new selection** of **extracted independent** variables that separate the most the different classes of the dependent variable that we cannot see here.

Importing LDA: Your classification model doesn't have to be a ***logistic regression,*** it can be ***SVM*** or a ***decision tree*** classification.

- But you need to **apply *LDA*** just **before *fitting*** your **classification model** to the *training set*.
- We import ***`LinearDiscriminantAnalysis`*** from ***`sklearn.discriminant_analysis`***. Then we create an object called ***`dcmPose_lda`***.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
dcmPose_lda = LinearDiscriminantAnalysis(n_components= 2)
```

- ***Linear Discriminants:*** The ***extracted features*** in **PCA** we called them **Principal Components**. In LDA ***extracted features*** are called **Linear Discriminants**.
- **`n_components:`** It specifies the number of components i.e the number of **Linear Discriminants**.
  - We don't need to build a vector of ***`explained_variance`*** or any other kinds of class ***`observability vector`***. We'll directly take **`n_components = 2`**.
- There is no need to ***`explained_variance`***, because we're not looking for the **independent variables** that explain the most variance. We're now looking for the independent variables that ***separate*** the ***most*** the ***classes*** of the ***dependent variable*** (already done three classes customer segment).
  - Here our goal is to **get** some **independent variables** that allows us to **visualize** the ***training-set results*** and the *test-set results*. So we already know we're looking for **2 Linear Discriminants**. For this reason we directly choose **`n_components = 2`**.

Building the LDA model: Also since ***LDA*** is a *supervised* technique, we need to use *independent (feature matrix)* and *dependent-vector* both with our ***`fit_transform()`*** method.

- Notice for ***training*** set we used ***`fit_transform(X_train, y_train)`***, but for ***test*** set we used only the *independent (feature matrix)* ***`transform(X_test)`***. Because we'll predict the dependent variable by fitting the test data.

```
X_train = dcmPose_lda.fit_transform(X_train, y_train)
X_test = dcmPose_lda.transform(X_test)
```

- The `fit_transform` method fit the object in the training set (both `X_train, y_train` are used) and transform it and at the same time it extracts **2 Linear Discriminants** so that ***X_train*** becomes a matrix of two new features (instead of 13-features).
- Then we use the `transform` method on the ***test set***, so that ***X_test*** becomes a matrix of features containing the 2 same **Linear Discriminants**.
- We don't need to include ***y_test*** in `transform`, because ***y_train*** in `fit_transform` is just used to **build/fit** the **LDA object** to the **training set** to build the LDA-model.

```
# Applying LDA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
dcmPose_lda = LinearDiscriminantAnalysis(n_components= 2)
# notice in LDA "fit_transform" takes both independent & dependent: X_train, y_train
X_train = dcmPose_lda.fit_transform(X_train, y_train)
X_test = dcmPose_lda.transform(X_test)
# explained_variance = dcmPose_lda.explained_variance_ratio_
```

NOTE that **LDA** is a **supervised model** it needs **dependent variables** data, hence both ***X_train***, ***y_train*** are used, because we are looking to separate the most the different classes of the dependent variable.

- That's the key difference between the **unsupervised** model **PCA** and **supervised** model **LDA**.

To build/fit the logistic model we don't need to change anything just set the plot-labels:

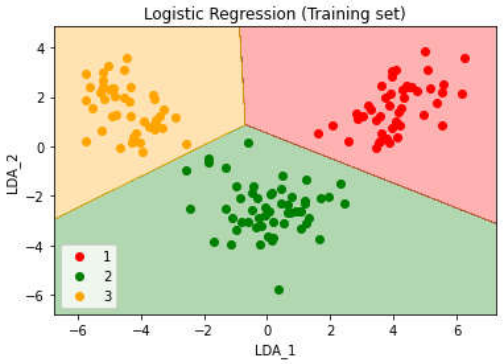
```
plt.xlabel('LDA_1')
plt.ylabel('LDA_2')
```

Confusion matrix: Since LDA is looking to separate the most the classes, we expect that the classes will be perfectly well separated and therefore if that's the case we should get an accuracy of 100% that depend on how the dataset is structured.

	0	1	2
0	14	0	0
1	0	16	0
2	0	0	6

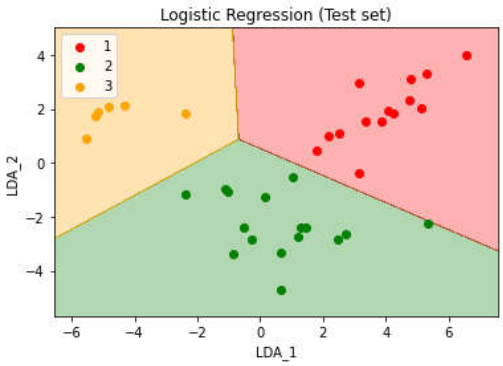
- **No incorrect predictions!!** Our test-set contains 36 observations and here we have all 36 correct predictions. We get an accuracy of 100%.
- That was ***not totally unexpected*** because this perfect 100% accuracy results from the perfect ***Seperability*** of our ***classes*** and ***LDA*** extracted the ***independent*** variables that separate the most ***3 classes*** in ***customer_segments***.

Visualization: We'll get perfectly well separated prediction regions, as we can see the prediction boundary is slightly different from PCA.



- We can clearly see that each **straightline** composing this **prediction boundary** is separating.
- If we take the closest points to prediction boundaries we can see that the points are approximately equally distant to this line. That's **class seperability**.

Outliers points: If we look at the prediction boundary between the ***Green*** region and the ***Red*** region corresponding to respectively ***customers*** number ***1*** and ***2***.



- We can see that this **class seperability** is **less obvious** when we look at these closest **green** points closest **red** points.
- That's due to the fact that these points were considered as outliers by the LDA model, and it considers **other points** between these two region to make boundary **equally distant**.

That's how we visualize the result and ***reduce dimensionality*** using ***PCA*** & ***LDA***. But these are for **linear-models**. Next we see a new technique ***kernel-PCA*** for *non-linear models*.

10.10 Kernel-PCA in Python: Part 1: Linear model & non-linear-dataset

PCA and ***LDA*** feature **extraction techniques** work on **linear problems** (i.e. when the data is **linearly separable**).

❒ Here we'll consider a non-linear problems where the data is **not linearly separable**. Hence we'll apply a new feature extraction technique. This technique is called ***kernel-PCA***.

❒ ***kernel-PCA:*** ***kernel-PCA*** is a ***kernelized*** version of ***PCA*** where we ***map*** the ***data*** to a **higher dimension** using the ***kernel trick***. From there we **extract** some new **principal components** (PCs) and we're going to see how it manages to deal with **non-linear problems**.

Problem description: We're not going to work on the same problem as we did in PCA i.e. we're not using ***Wine.csv***. We're using the data-set of the Chapter 3: Classification, that we are used to model logistic-regression.

This dataset ***Social_Network_Ads.csv*** contains the **information** of various **users** obtained from the ***social networking sites***. There is a ***car making company*** that has recently launched a ***new SUV car***. So the company ***wanted to check how many users from the dataset, wants to purchase the car***.

For this problem, we will build a Machine Learning model using the ***Logistic regression algorithm***. The dataset is shown in the beside image. In this problem, we will predict the ***purchased*** variable (*Dependent Variable*) by using ***age*** and ***salary*** (*Independent variables*).

User ID	Gender	Age	EstimatedSalary	Purchased
15624510	Male	19	19000	0
15810944	Male	35	20000	0
15668575	Female	26	43000	0
15603246	Female	27	57000	0
15804002	Male	19	76000	0
15728773	Male	27	58000	0
15598044	Female	27	84000	0
15694829	Female	32	150000	1
15600575	Male	25	33000	0
15727311	Female	35	65000	0
15570769	Female	26	80000	0
15606274	Female	26	52000	0
15746139	Male	20	86000	0
15704987	Male	32	18000	0
15628972	Male	18	82000	0
15697686	Male	29	80000	0
15733883	Male	47	25000	1
15617482	Male	45	26000	1
15704583	Male	46	28000	1
15621083	Female	48	29000	1
15649487	Male	45	22000	1
15736760	Female	47	49000	1

We'll use ***Social_Network_Ads.csv*** because it is **easy to *visualize* the result** *(this dataset already has 2 main independent variable, so it is easy to visualize but at the same time it its non-linear)*, what happens if we use kernel-PCA to solve a **non-linear-problem** using **linear model** like ***Logistic-Regression***.

- ***Kernel-PCA*** manages to ***extract*** some new ***independent variables*** (PCs) when the problem is ***non-linear***. i.e. when the data is not linearly separable, because ***Social_Network_Ads.csv*** contains non-liner-data.
- Remember it was clearly a **nonlinear problem** because nonlinear classifiers (KNN, kernel-SVM & Naïve-Bayes) showed much **better performance** than *logistic-regression*.

❒ Before applying ***kernel-PCA*** we like to visualize again why this linear model is not appropriate for this data set. So we first use ***Logistic-Regression*** without using ***kernel-PCA*** to compare the result.

Logistic-regression only

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values


# Data Split
from sklearn.model_selection import train_test_split
# 0.25 test_size means "1/4"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler
#  y need not to be scaled: categorical variable
# sc_x = StandardScaler()
# X_scaled = sc_x.fit_transform(X)
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)
```

```
# Fit dataset to Logistic regression
from sklearn.linear_model import LogisticRegression # import class
# instead of "regressor" we now use "classifier"
classifer = LogisticRegression(random_state= 0) # create object
classifer.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = classifer.predict(X_test)

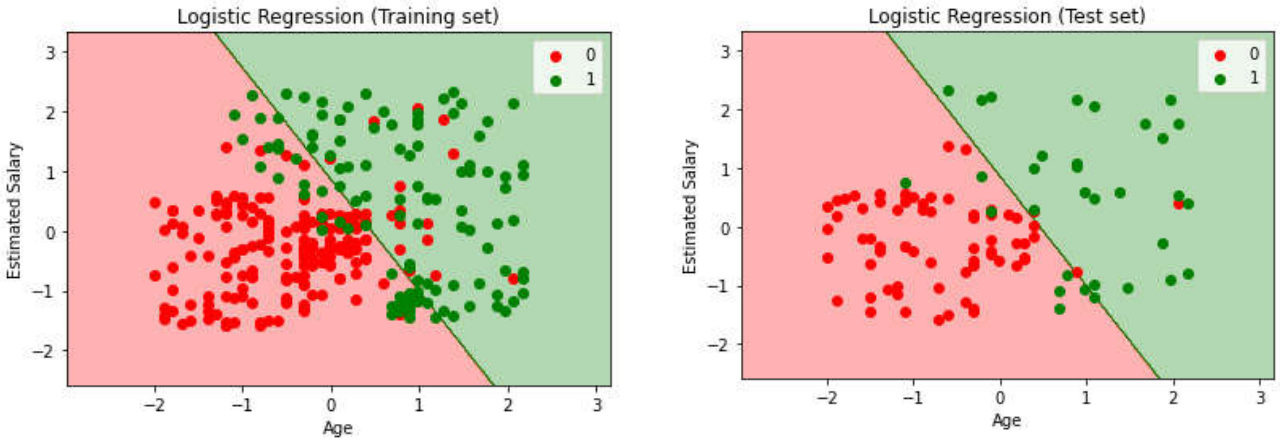
# Making the confusion matrix use the function "confusion_matrix"
# Class in capital letters, functions are small letters
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value



# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.30, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Logistic Regression (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.30, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Logistic Regression (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

☝ Following are the visual result for training-set and test-set. **`Without`** using **`kernel-PCA`**.



☝ ***Logistic regression*** model is a ***linear classifier*** therefore it will not be ***appropriate*** for our ***current problem*** because our data is not ***linearly separable***.

☞ Customers in the social network are represented by their **age** and their **estimated salary**. And our predictions are represented by these Red and Green regions.

☞ **Red region** predicts that the customer will not **click** on the **Ad**. **Green region** predicts that the customers will **click** on the **ad** and buy the **SUV**. The **straight line** is the ***prediction boundary*** generated by the logistic regression model.

The problem was: since the **logistic regression** model is a **linear classifier** then it has to be a **straight line**, it cannot generate **curve**. So it cannot separate the **Green-points** and **Red-points** ***properly*** (notice the figure).

☞ It can't make some kind of ***curve*** to catch these ***green users*** that should be in the ***green region***. Right now they're in the ***red region*** as well as some ***Green-points*** are in the ***red region***. This clearly represents the fact that our data is not linearly separable. Because those users are not in the right region.

The solution is to use a non-linear classifier, i.e. ***KNN***, ***kernel-SVM*** or ***Naïve-Bayes*** or ***Random-forest***. But we are not gonna do those ,instead, we'll use **Kernel-PCA** to keep this straight line as a as the **prediction boundary** of the **Logistic regression linear classifier**.

10.11 Kernel-PCA in Python: Part 2: Kernel-PCA with Logistic Regression

Since we're going to apply ***kernel-PCA***, this will apply ***kernel trick*** to map the data into a ***higher dimension*** and then apply ***PCA*** to extract ***new components*** that will be new dimensions that explain the most variants.

☞ It'll manage to get some ***new dimensions*** in which the ***data*** will be ***linearly separable*** even by a ***linear classifier*** like ***logistic regression***.

❒ Now we're going to apply ***kernel PCA*** inside of this **Logistic regression linear classifier** to see how ***kernel-PCA*** will **save** the **situation**. We'll observe how the ***kernel PCA*** managed to extract new PCs from this nonlinearly separable data.

☯ We need to apply **kernel PCA** right ***after*** the **data preprocessing phase** and just **before fitting** our **classifier** (building classification model) like ***logistic regression*** to our ***training-set***.

☯ We'll use ***KernelPCA*** class from ***sklearn.decomposition***.

☯ Then we create an object of this ***KernelPCA*** class naming ***dcomposer_kr_PCA***.

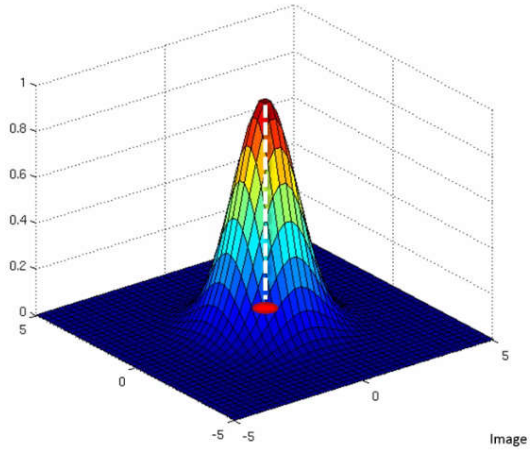
☯ ***Parameters:*** We use ***n_components= 2***, because we only have 2-independent variables.

⇨ And we need to choose the kernel because Kernel-PCA will use a kernel trick to our dada-set. We set **`kernel="rbf"`**.

⇨ Here we have ***kernel*** parameter which is exactly similar to ***kernel-SVM***, we have the same options. Here we choose ***'rbf'*** that represents ***Gaussian RBF kernel***.

```
from sklearn.decomposition import KernelPCA
dcomposer_kr_PCA = KernelPCA(n_components= 2, kernel="rbf")
```

Gaussian RBF kernel



$$K(\vec{x}, \vec{l}^i) = e^{-\frac{\|\vec{x}-\vec{l}^i\|^2}{2\sigma^2}}$$

Image source: http://www.cs.toronto.edu/~duvenaud/cookbook/index.html

☝ When we apply ***kernel PCA*** to our **data set**, our data set will be ***mapped*** to a ***higher dimension*** using the ***kernel trick*** that already create some new dimensions a new feature space where data will be literally separable. (You can revisit Kernel-SVM for more).

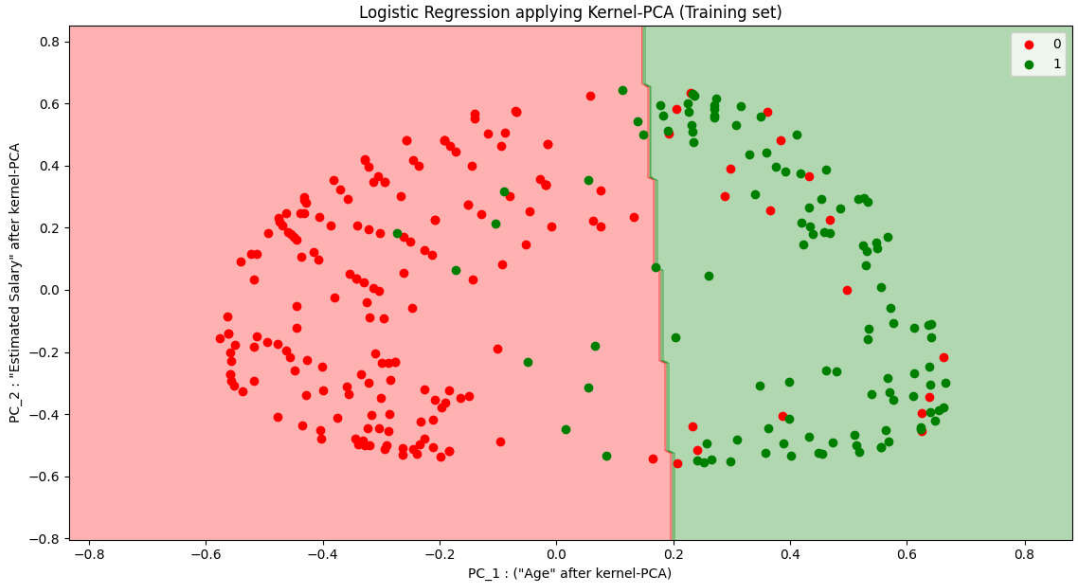
➲ And then since we are in this **new feature space** where the **data** is **linearly separable** Well ***PCA*** will be **applied** to **reduce** the **dimensionality** by **extracting** the **new principal components**.

```
# applying Kernel-PCA : Unsupervised
from sklearn.decomposition import KernelPCA
dcomposer_kr_PCA = KernelPCA(n_components= 2, kernel="rbf")
X_train = dcomposer_kr_PCA.fit_transform(X_train)
X_test = dcomposer_kr_PCA.transform(X_test)
```

❒ ***Now we execute all the steps:*** Data ***processing*** phase, applying ***kernel PCA*** to our dataset, ***fitting*** the logistic regression model to the train-set, creating the ***test results***, making the ***confusion matrix*** and the ***visualization*** of the train-set & test-set results.

✋ Remember we're ***not expecting*** a ***nonlinear*** classifier with a ***curved*** prediction ***boundary***. We're still expecting a ***straight line*** but you will see that this time the ***straight*** line is perfectly going to ***separate*** our ***dataset***, the two classes in our data set thanks to these new extracted features (new ***PCs***).

Training-set result



Test-set result



❒ For us this is actually kind of new. These are the results of **Kernel-PCA** combined to a **logistic regression** model that we apply on a ***non-linear separable*** dataset.

❒ All the different elements of this plot represented the same thing

- The ***red points*** are the customers that in reality ***didn't click*** on the ***ad*** by the SUV
- The ***green points*** are the customers that in reality ***clicked*** on the ***ad*** to buy the SUV

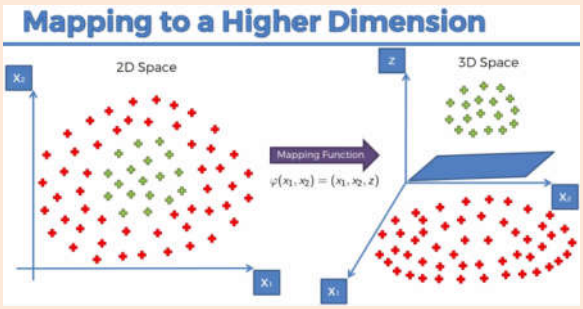
☯ The most important thing is that our two classes the **red class** and the **green class** are now much better **separated** by the **straight line (**prediction boundary**)**.

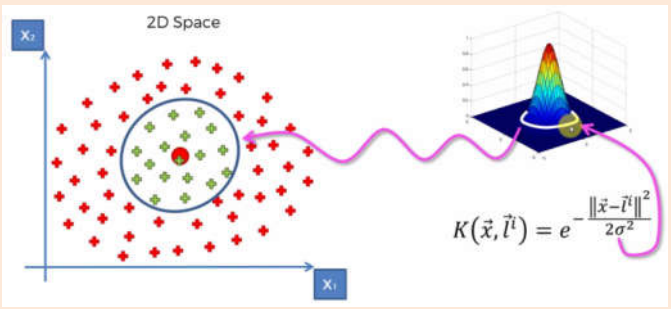
⇨ Still some ***green points*** in the ***red region*** and some ***red points*** in the ***green region*** but now we are in a ***new feature space*** where the ***observation points*** of the two different classes are now much ***better separated***.

🖐 And this **new feature space** that we're in right now is formed by these **principal components (PCs)** that were extracted through **Kernel-PCA**.

☝ So these new-independent variables here are ***not*** the ***age*** and ***not*** the ***estimated salary***. Those are now a PC_1 and PC_2 as principal components. And these are the **dimensions** of this **new feature space** where our **data** is now well **generally separable** by this **straight line** *(prediction boundary of the logistic regression classifier)*.

👽 ***What happened behind the scenes:*** Our original ***feature space*** was mapped to a ***higher dimension*** using the ***kernel trick*** to avoid to highly ***compute intensive*** computation.



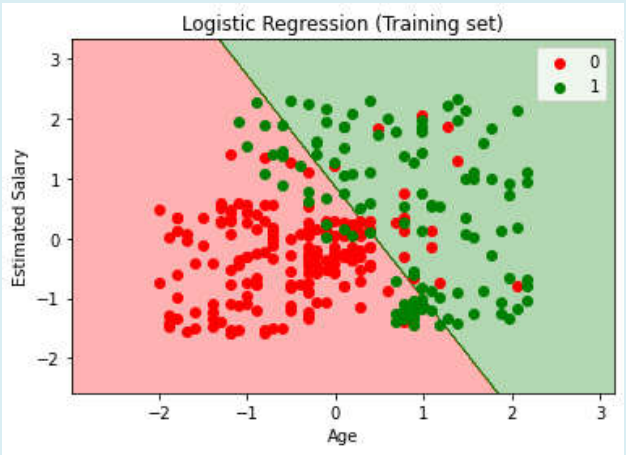


☞ And since we are in higher dimension, going to ***higher dimension***, creates some ***new dimensions*** and mostly that ***created a new feature space*** where our data was ***linearly separable***.

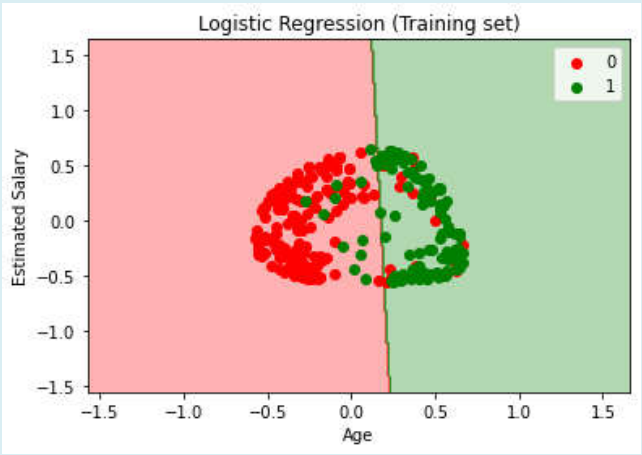
☞ And since we are in ***higher dimension***, more ***dimensions*** than the ***original*** number of ***dimensions***. So we still needed to apply the ***PCA dimensionality reduction technique*** to end up with a **lower dimensions**. So then ***PCA*** was ***applied*** to this ***new feature space*** where the data was ***linearly separable*** and it extracted new independent variables i.e. ***principal components*** (PCs).

☞ And eventually we obtain this ***new Feature Space*** formed by these ***two*** new extracted ***PCs***. Where our data is ***linearly separable*** and much better separated by a ***linear classifier***.

Before Kernel-PCA



After Kernel-PCA



All code at once (practiced version)

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values


# Data Split
from sklearn.model_selection import train_test_split
# 0.25 test_size means "1/4"th of the total observation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)


# applying Kernel-PCA : Unsupervised
from sklearn.decomposition import KernelPCA
dcomposer_kr_PCA = KernelPCA(n_components= 2, kernel="rbf")
X_train = dcomposer_kr_PCA.fit_transform(X_train)
X_test = dcomposer_kr_PCA.transform(X_test)


# Fit dataset to Logistic regression
from sklearn.linear_model import LogisticRegression # import class
# instead of "regressor" we now use "classifier"
classifer = LogisticRegression(random_state= 0) # create object
classifer.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = classifer.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
# Class in capital letters, functions are small letters
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value



# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.30, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Logistic Regression applying Kernel-PCA (Training set)')
pLt.xlabel('PC_1 : ("Age" after kernel-PCA)')
pLt.ylabel('PC_2 : "Estimated Salary" after kernel-PCA')
pLt.legend()
pLt.show()

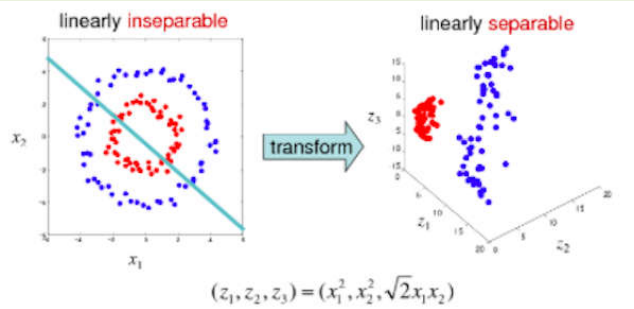
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.30, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
```

```
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Logistic Regression applying Kernel-PCA (Test set)')
plt.xlabel('PC_1 : ("Age" after kernel-PCA)')
plt.ylabel('PC_2 : "Estimated Salary" after kernel-PCA')
plt.legend()
plt.show()

# python prctc_lgstc_krnl_PCA.py
```

- Thus when we dealing with some data, which is ***completely linearly inseparable***, like **red points** surrounded by a circle of **blue points** for the original dataset. Then by applying ***Kernel-PCA*** we can make them completely ***linearly separable***.

- ***We can think Kernel-PCA as:*** We are taking our **non-linear-separable data-set** into **higher dimension** using ***kernel-trick*** and finding a plane/space where we get the **`projection`** of our **`data-points`** that they are **`linearly separable`**.



https://towardsdatascience.com/dimension-reduction-techniques-with-python-f36ca7009e5c

Chapter 11 : Part 1

Model Selection

K-fold Cross Validation, Grid Search

11.1.1 Evaluating & Improving Model Performance

❒ ***Model parameters and Hyper parameters:*** In a ***ML model*** we have two types of parameters. ***Model parameters*** and ***Hyper parameters***.

☞ ***Model parameters:*** These are the parameters that the *model learns* and found *optimal values* by running the *model*.

☞ ***Hyper parameters:*** This type of *parameters* are the *parameters* that we *choose ourselves*. For example the *kernel parameter* in the *kernel-SVM* or the ***penalty parameter*** or even some ***regularisation parameter***.

So there is still room to improve the model because we can still choose some optimal values for these parameters.

❒ In this chapter we will do two things:

[1]. ***Evaluating our model performance:*** To evaluate our models in a efficient way, we use ***K-fold-Cross-validation*** instead of just ***train-test-split***. Then to tune- Hyper parameters we'll use ***Grid Search***.

[2]. ***Improving our model performance:*** We'll use the most powerful algorithm in ML***, XGBoost***.

➲ Improving the model performance can be done with a technique called *model selection*. That consists of ***choosing*** the ***best parameters*** of your ***ML model***.

❒ ***Grid Search:*** Since the ***model parameter*** is **learned** by the **model** then we need to **figure** out a **way** to choose the **optimal values** for these ***Hyper parameters***. One of the powerful techniques to tune- Hyper parameters is ***Grid Search***.

❒ ***K-fold-Cross-validation:*** We need to optimize a way to evaluate our models before we start grid search. Previously we just *split* our *dataset* between the *training set* and a *test set*.

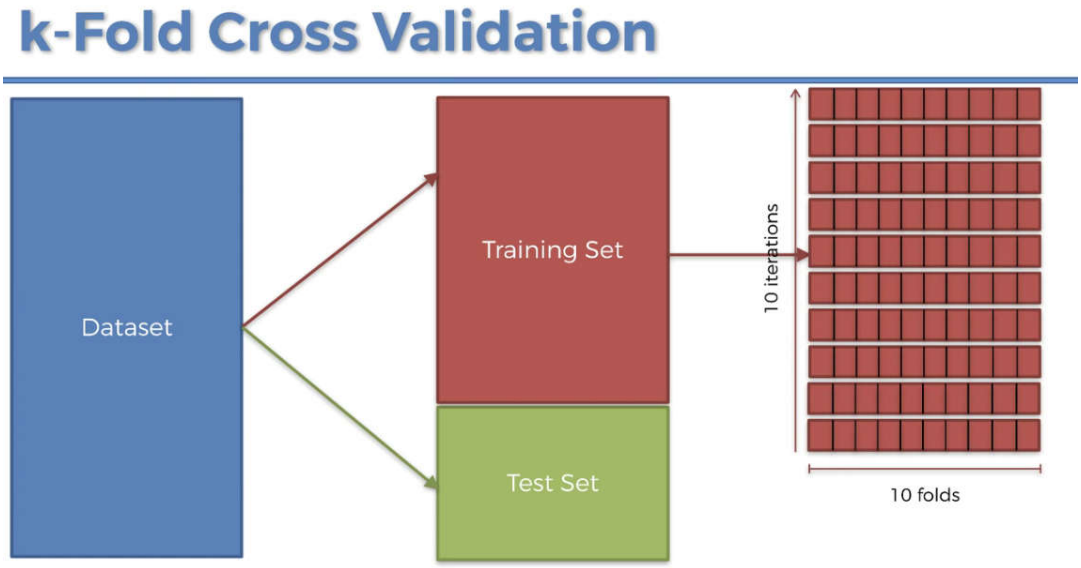
☞ We trained our model on the *training set* and we *tested* its *performance* on *the test*. That's a correct way of ***evaluating*** the ***model performance***.

➲ But that's not the *best* one because we actually have a *variance problem*.

➲ The ***variance problem*** can be explained by the fact that when we get the *accuracy* on the *test set*. And if we **run** the **model again** and **test** again it's **performance** on **another *test set***, we can get a very different accuracy.

➲ Hence judging our *model performance* only on one *test set* is actually not a *good idea*. That's not the most relevant way to evaluate the model performance.

☞ ***K-fold-Cross-validation:*** K-fold-Cross-validation is a technique that resolve the *variance problem* by *splitting* the *training set* into ***K-fold***.

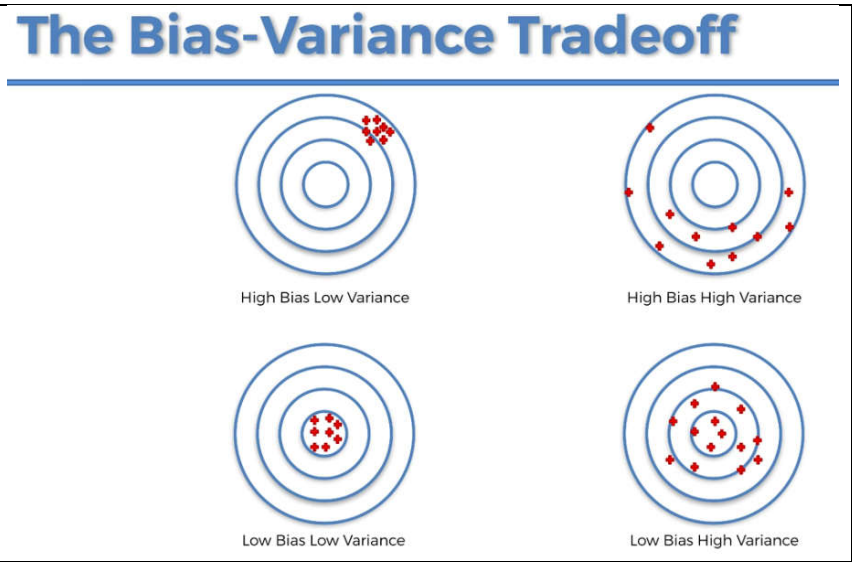


☝ ***No. of Folds:*** Most of the time ***K = 10***, and we **train** our model a **nine fold** and we **test** it on the **last** remaining **fold**.

☝ ***No. of Iterations:*** Since we have ***10 folds***, we can make ***10*** different ***combinations*** of ***9+1 fold*** to train the model. Thus the model can be train in 10 iterations for each combination of 10-fold.

☞ Once we ***train*** the model and ***test*** them all on ***ten combinations*** (10 iterations) of ***training*** and ***test*** sets. We'll get ***10 different accuracies***.

➲ We take an ***average*** of ***10*** different **accuracies** to model-performance-evaluations and also compute the ***standard deviation*** to have a look at the ***variance***. These will give us a better idea of our model's performance.

✋ Eventually our analysis will be much more relevant and we'll know in which of these four categories will be: i. ***Good accuracy*** (low bias) and a ***small variance*** - lower left ii. ***Large accuracy*** and a ***high variance*** - lower right iii. ***Small accuracy*** (high bias) & ***low variance*** - upper left iv. ***Low accuracy*** and a ***high variance*** – upper right.	

❒ ***K-fold cross-validation*** approach ***divides*** the ***input dataset*** into ***K groups*** of ***samples*** of equal sizes. These ***samples*** are called ***folds***. For each learning set, the ***prediction*** function uses ***k-1 folds***, and the rest of the ***folds*** are used for the ***test set***. This approach is a very popular CV approach because it is easy to understand, and the output is less biased than other methods.

➲ The ***steps*** for ***k-fold cross-validation*** are:

[1]. Split the input dataset into K groups

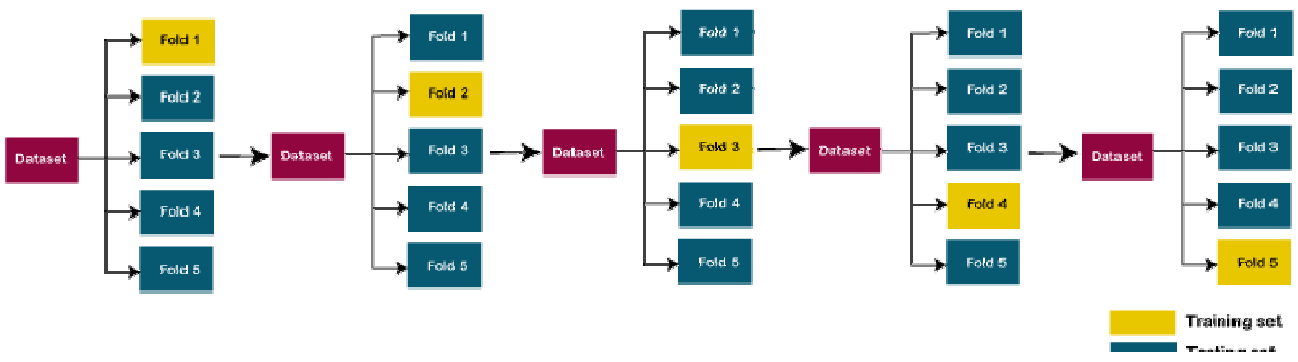
[2]. For each group:

- ✓ Take **one group** as the **reserve** or **test data set**.
- ✓ Use **remaining groups** as the **training dataset**
- ✓ ***Fit*** the model on the ***training set*** and ***evaluate*** the ***performance*** of the model using the ***test set***.

Let's take an example of ***5-fold cross-validation***. So, the dataset is ***grouped*** into ***5 folds***.

☯ On 1st iteration, the ***first fold*** is reserved for ***test*** the model, and rest are used to ***train*** the model.

☯ On 2nd iteration, the ***second fold*** is used to ***test*** the model, and rest are used to ***train*** the model. This process will continue until each fold is not used for the test fold. Consider the below diagram:



11.1.2 K-fold cross-validation : Evaluate model performance

Let's start with this **K-fold cross-validation**, our first technique of model selection. We are going to **use** one of the **model** that we've **built** previously and apply **K-folds cross-validation** on it.

❒ We're gonna use the ***kernel-SVM*** model from ***Chapter -3 : Classification***. Where we used the **Social Network Ads** data and ***kernel-SVM*** used to predict if the customers are going to click on the ads on the social network to buy the SUV (yes or no).

☞ We'll use the dataset ***Social_Network_Ads.csv*** contains the **information** of various **users** obtained from the ***social networking sites***. There is a ***car making company*** that has recently launched a ***new SUV car***. So the company ***wanted to check how many users from the dataset, wants to purchase the car***.

❒ Since the model is ***already built*** and we already have ***everything***. We just ***copy*** the whole ***code*** and implement **K-fold cross-validation** in right place (on a new section of code).

❒ ***Where to apply K-folds cross-validation code section:*** Since that consists of evaluating the model performance, the most relevant location to put it is ***right after*** we ***build*** our ***kernel-SVM model*** that is right we built the model.

☞ ***After y_pred and confusion-matrix?:*** Since getting the predictions of the test results and evaluating the confusion-matrix is actually a good way of evaluating the model. But not the best way, we apply **K-fold cross-validation** after ***y_pred*** and ***confusion-matrix***.

☯ We import the ***cross_val_score*** class from ***sklearn.model_selection*** (the same module already used for ***train_test_split***).

```
from sklearn.model_selection import cross_val_score
```

☯ Now we apply *k-fold-cross-validation* on our ***training set***: we create an object of ***cross_val_score*** called ***accuRacies*** (this is actually a vector).

➲ ***accuRacies*** will return ***10 accuracy's*** (a *vector of accuracies* of the model) for each one of the **10 combinations** that will be created through ***10-fold-cross-validation***, (K =10 here).

```
accuRacies = cross_val_score(estimator=clsFier, X = X_train, y = y_train, cv= 10)
```

➲ ***Parameters:***

i. ***estimator :*** estimator is the object that implementing 'fit' on the data. This is our SVM-classifier object, so we set **`estimator = clsFier`**,

ii. ***X:*** is the data to fit. It is actually our feature-matrix of the training sets ***X_train***. So we set **`X = X_train`**,

iii. ***y :*** is the target variable, (i.e the ***dependent variable vector***) to try to predict in the case of supervised learning. So we set **`y = y_train`**,

iv. ***cv :*** Determines the cross-validation splitting strategy. It is the number of folds of *k-fold-cross-validation*. Here we wanna apply ***10 folds*** so we set **`cv = 10`**.

- ❖ The most common choice for this CV number is actually 10. Most of the time you'll use ***10-fold-cross-validation*** where you'll get ***10*** **accuracy's** and ***10*** **accuracy's** is actually enough to get a ***relevant idea*** of the *model performance*.

v. **`n_jobs`** (optional): Number of jobs to run in parallel. Training the **estimator** (classifier/regressor) and **computing** the **score** are **parallelized** over the **cross-validation (CV) splits**.

- ❖ **None** means **1** unless in a joblib.parallel_backend context.
- ❖ **-1** means using **all processors**. It means that you will use all the CPU on your machine and therefore your can run *k-fold-cross-validation* even faster in case you are working on a ***very large dataset***.

❒ ***Calculating Mean of accuracies and slandered deviation:*** We use `mean()` to return mean, and `std()` to return standard deviation.

```
mean_accu = accuRacies.mean()
std_accu = accuRacies.std()
```

```
# Applying K-folds cross-validation
from sklearn.model_selection import cross_val_score
accuRacies = cross_val_score(estimator=clsFier, X = X_train, y = y_train, cv= 10)
mean_accu = accuRacies.mean()
std_accu = accuRacies.std()
```

❒ After executing following code:

```
# --------------- Model-Selection and Boosting ------------------
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values



# Data Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler

st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)

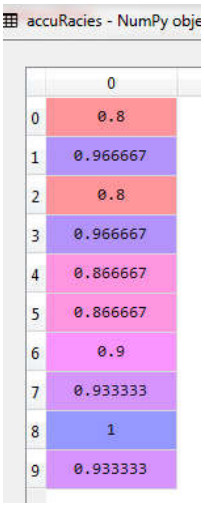

# Fit train set to Kernel-SVM classifier
from sklearn.svm import SVC
# since data-points are non-seperable linearly, use "rbf" : Gaussian kernel, gives better result.
# kernel="rbf" instead of kernel="linear"
clsFier = SVC(kernel="rbf", random_state=0)
clsFier.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = clsFier.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value

# --------  Applying K-folds cross-validation  -------------
from sklearn.model_selection import cross_val_score
accuRacies = cross_val_score(estimator=clsFier, X = X_train, y = y_train, cv= 10)
mean_accu = accuRacies.mean()
std_accu = accuRacies.std()
```

➲ ***Accuracy vector:*** Here is our accuracy's vector. The first accuracy is 80% but then the second accuracy is 96% and then 80% followed by other accuracies.



- ✓ It clearly shows that, the performance of your model changes when test-set varies. So evaluating the medel performance on one test set is not very relevant.
- ✓ With 10-fold-cross-validation we are testing it on 10 test sets.
- ✓ ***Mean:*** Now we're going to take the ***mean*** of all these ***10 accuracies*** and that **better idea** of the **average model-performance** of our **model**.

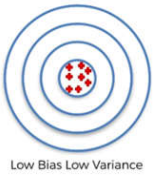
```
mean_accu = accuRacies.mean()
```

The mean of these 10 accuracy's here is actually 90.33%. So in conclusion this 90% accuracy is the ***relevant evaluation*** of our ***model performance***.

- ✓ ***Standard Deviation:*** If we want to put the analysis further, we can compute also the **standard deviation** of this accuracy's vector that will tell us if there is a high variance or low variance.

```
std_accu = accuRacies.std()
```

- ❖ We get a 6.5% standard deviation. That means the average of the differences between the different accuracies and 90.3% is 6.5%.
- ❖ That's actually not too high variance. It means that when we evaluate our model performance, most of the time will be around 84% and 96% so eventually that means that we are in this ***low bias*** and ***low variance*** category.


Low Bias Low Variance

All code for k-fold-cross-validation at once (practiced version)

```
# --------------- Model-Selection and Boosting ------------------
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values


# Data Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler

st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)


# Fit train set to Kernel-SVM classifier
from sklearn.svm import SVC
# since data-points are non-seperable linearly, use "rbf" : Gaussian kernel, gives better result.
# kernel="rbf" instead of kernel="linear"
clsFier = SVC(kernel="rbf", random_state=0)
clsFier.fit(X_train, y_train) # fit the dataset

# Predict
y_prd = clsFier.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value


# Applying K-folds cross-validation
from sklearn.model_selection import cross_val_score
accuRacies = cross_val_score(estimator=clsFier, X = X_train, y = y_train, cv= 10)
mean_accu = accuRacies.mean()
std_accu = accuRacies.std()

# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.3, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Kernel-SVM (Training set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()

# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
pLt.contourf(X1, X2, clsFier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.3, cmap = ListedColormap(('red', 'green')))
pLt.xlim(X1.min(), X1.max())
pLt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    pLt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
pLt.title('Kernel-SVM (Test set)')
pLt.xlabel('Age')
pLt.ylabel('Estimated Salary')
pLt.legend()
pLt.show()
```

11.1.3 Grid-Search : Improve models performance

In the previous section we used ***K-fold-cross-validation*** to ***evaluate*** our model ***performance***. In this section we're going to learn ***Grid-Search*** which is used to ***improve*** models ***performance***.

❒ We'll ***improve*** our model by finding the ***optimal*** values of the ***hyper parameters***, the parameters that we choose.

☞ This ***Grid Search*** technique will help us to choose **appropriate hyper parameters** for our model and ***tune*** them to the ***optimal value***.

*How do I know **which model** to choose for my **machine learning problem**? I have a machine learning problem that comes with a specific dataset, how do I know which model to choose to solve my business problem? Which model would be the best one?*

✓ ***Step 1:*** Figure out, if your problem is a ***regression*** problem or a ***classification*** problem or a ***clustering*** problem.

☑ You just need to look at your dependent variable.

❍ If you don't have a dependent variable then it's a **clustering** *problem*.

❍ And if you have a ***dependent variable*** you see if it's a ***continuous outcome*** or a ***categorical outcome***.

❍ If it's a ***continuous outcome*** then your problem is a **Regression** problem.

❍ If it's a ***categorical outcome*** then your problem is a **Classification** problem.

✓ ***Step 2:*** find out your problem is ***linear*** problem or ***nonlinear*** problem.

☑ When you have a ***large data set*** you ***cannot figure out*** the ***linearity*** of your dataset ***easily***. For large data-set you have to try both situation, to choose a ***linear*** model like ***SVM*** or a ***nonlinear*** model like ***kernel-SVM*** (if you're doing classification).

☑ In this kind of situation where we have ***large-dataset*** we can get help from ***Grid search***. ***Grid search*** will ***tell*** us if we should rather choose a ***linear model*** *like* ***SVM*** or a ***non-linear model*** *like* ***kernel-SVM***.

Problem description: We're going to work on the same problem as in the previous section (k-fold-cross-validation). ***Classify*** the ***users*** in the ***social network*** and predict if they're going to click ***yes*** or ***no*** on the ad to ***buy*** the ***SUV***.

☯ So we use the same data-set ***Social_Network_Ads.csv***.

☯ ***Grid search*** will tell us if we should rather choose an ***SVM*** model or a ***kernel-SVM*** model.

☯ Since the ***kernel SVM*** model has ***many parameters***, like ***penalty***, ***gamma*** parameter and ***grid search*** will tell us exactly which ***values*** we should choose for these ***hyper-parameters***.

❒ ***Fitting grid search:*** Basically ***grid search*** can be ***applied*** after ***fitting*** your ***model*** to the ***training set*** because one of the ***paramter*** of ***grid search*** will be the ***classifier***.

☞ Since we first used ***K-fold-cross-validation*** to ***evaluate*** the ***model performance*** and now we are working on ***improving*** the ***model performance***, we put the **grid search** section right **after** ***K-fold-cross-validation*** section.

☞ We import ***GridSearchCV*** package from ***sklearn.model_selection*** because grid search is a *model selection technique*.

```
sklearn.model_selection import GridSearchCV
```

☞ ***Specifying the different parameters:*** To use **Grid-search** for **Hyper-parameter tuning**, we need to ***input*** those ***Hyper-parameters*** as a ***list*** of ***dictionary***. So first we create this dictionary of parameters. We name this dictionary of parameters as ***params_dic***.

⇨ When we built our model we only used 2 parameters: `clsFier = SVC(kernel="rbf", random_state=0)`

⇨ But there are other parameters for ***SVC()*** that we didn't specify, and we used their default values. For example: penalty parameter ***C***, ***degree*** (in case of polynomial), ***gamma*** etc.

⇨ Say we want to tune C, kernel & gamma, 3 hyper-parameters. Then we create **dictionaries *"key-value"*** pair, **parameter-name** as ***key*** and ***list of values*** that we want to set for tuning as ***value*** of the dictionary. Those values will be tested by the grid search model. And among these values the grid search model will find the best one.

```
# --------   Grid-Search to find the "Best model" and 'Best hyper-parameters'-------------
from sklearn.model_selection import GridSearchCV
params_dic = [
            {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
            {'C': [1, 10, 100, 1000], 'kernel': ['rbf'], 'gamma' : [0.7, 0.5, 0.1, 0.01 ,0.001, 0.0001]},
            ]
```

❍ This ***penalty*** parameter is for ***regularization*** to prevent ***overfitting***, the default value is 1. We tune this for our model to prevent overfitting.

❍ The more you increase this ***penalty*** parameter ***C*** the ***more*** it will prevent ***overfitting***. But be careful you should ***not increase*** it ***too much*** because otherwise you will get a new problem which will be ***Underfitting***. For example 10000 would be too much for penalization.

- ***kernel*** parameter also included it in the dictionary as well, to find out the best model, a *linear* model or a *non-linear* (Eg: **rbf**)model. If you are dealing with large data-set then it will be very useful for you to decide if you should take a linear or non-linear model.
- We're not going to include ***degree*** in the dictionary because we won't test if we should take polynomial Kernel. We're just examine between *linear* model or a *non-linear* model.
- ***gamma*** is a parameter for the nonlinear kernels like **'rbf'**, **'poly'** and **'sigmoid'**. Since we are using **'rbf'**, We'll try to optimize this ***gamma*** parameter and find the ***best value***.
  - The default value is **auto** and if gamma is **auto** than **1/n** features will be used instead. So we set the values in the ***range [0, 1]***.
  - Here in this problem, we have ***2 features*** so we set **(1/2)=0.5**, also **0.7** and other values if **0.5** is not the optimal.
  - If you have a ***lot more features*** for example **100** ***features***, or **1000** ***features*** you can even try a ***smaller number*** for the **gamma** parameter like **0.001**.

⇨ ***The first option is:*** **Grid Search** will investigate is a linear model that is a classic SVM with a linear kernel. And it will try to find the optimal value for the penalty parameter C.

⇨ ***The second option is:*** **Grid Search** will investigate the nonlinear model 'rbf' kernel-SVM, it will tune gamma, and also the penalty parameter C.

Eventually what we'll get is a ***single combination*** of all this ***different parameters*** and options. And ***grid-search*** will find that ***combination*** for us.

☞ ***Applying Grid-search:*** In this step we are going to implement ***grid search***. We do that right after the *list* of *parameters-dictionary* that we just created.

⇨ We'll create an *object* of ***GridSearchCV*** class and we are going to fit this *object* on our *training* set ***X_train***.

```
griD_Srch = GridSearchCV(estimator = clsFier,
                         param_grid= params_dic,
                         scoring= 'accuracy',
                         cv = 10,
                         n_jobs= -1     )
griD_Srch = griD_Srch.fit(X_train, y_train)
best_accu = griD_Srch.best_score_
best_parameters = griD_Srch.best_params_
```

- ***Parameters of*** GridSearchCV***:***

  i. ***estimator :*** This is our SVM-classifier object, so we set **estimator = clsFier**,

  ii. **param_grid:** is the *list* of *parameters-dictionary* that we want to tune using ***Grid-Search***.

  param_grid= params_dic,

  iii. **scoring:** Is the ***scoring matrix*** that is used to decide the best ***parameters***. We set **scoring= 'accuracy'**,
  - ***Grid search*** is going to ***select*** the best parameters based on one **performance matrix**.
  - It can be the ***accuracy*** matrix or it can be the ***precision*** matrix, it can be the ***recall***. So, it can be different performance metrics.
  - We are going to take the most common one like we did for deep learning the accuracy matrix. So , we set **scoring= 'accuracy'**.

  iv. ***cv :*** ***grid search*** is going to evaluate the ***performance*** of each of the model with their own set of parameters, using ***K-fold-cross-validation***. So we set ***10 folds***, ***cv = 10***.
  - The most common choice for this CV number is actually 10. Most of the time you'll use ***10-fold-cross-validation*** where you'll get ***10*** **accuracy's** and ***10*** **accuracy's** is actually enough to get a ***relevant idea*** of the ***model performance***.

  These above 4-parameters will do the job for grid-search. We can also use following optional parameter.

  v. **n_jobs** (optional): If you are working on a very large dataset you can set, ***n_jobs= -1.*** It should get all the power available from your machine.
  - **-1** means using **all processors**. It means that you will use all the CPU on your machine and therefore your can run ***k-fold-cross-validation*** even faster in case you are working on a ***very large dataset***. (otherwise use **None**)

- We then use ***fit()*** to apply this **Grid-search** to our train-set ***X_train*** and ***y_train*** (in case of unsupervised model we don't need ***y_train***).

```
griD_Srch = griD_Srch.fit(X_train, y_train)
```

- To look at the results, we interested in best accuracy and best parameters. We'll ***usebest_score_*** and ***best_params_*** attributes
  - ***Best Accuracy:*** ***best_accu = griD_Srch.best_score_*** returns the best accuracy that **Grid-search** found for the given hyper-parameters.
  - ***Best Parameters:*** ***best_parameters = griD_Srch.best_params_*** returns the best dictionary of parameters that **Grid-search** found for the given hyper-parameters.

best_accu	float64	1	0.9099999999999999
best_parameters	dict	3	{'C':1, 'gamma':1, 'kernel':'rbf'}

Above result shows us, we should use a non-linear model ***kernel SVM*** using "**rbf**", with C=1 and gamma =1.

```
# --------   Grid-Search to find the "Best model" and 'Best hyper-parameters'-------------
from sklearn.model_selection import GridSearchCV
params_dic = [
            {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
            {'C': [1, 10, 100, 1000], 'kernel': ['rbf'], 'gamma' : [1, 0.9, 0.8, 0.7, 0.5, 0.1, 0.01]},
            ]

griD_Srch = GridSearchCV(estimator = clsFier,
                         param_grid= params_dic,
                         scoring= 'accuracy',
                         cv = 10,
                         n_jobs= -1     )
griD_Srch = griD_Srch.fit(X_train, y_train)
best_accu = griD_Srch.best_score_
best_parameters = griD_Srch.best_params_
```

All code at once (with k-fold-cross-validation and Grid-search)

```
# --------------- Model-Selection and Boosting ------------------
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np

# Data Extract
dataSet = pd.read_csv("Social_Network_Ads.csv")
X = dataSet.iloc[:, [2,3]].values
y = dataSet.iloc[:, 4].values


        # Feature-Scaling after Data Split

# Data Split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.25, random_state = 0)

# Feature-Scaling
from sklearn.preprocessing import StandardScaler

st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)


# Fit train set to Kernel-SVM classifier
from sklearn.svm import SVC
# since data-points are non-seperable linearly, use "rbf" : Gaussian kernel, gives better result.
# kernel="rbf" instead of kernel="linear"
clsFier = SVC(kernel="rbf", random_state=0)
clsFier.fit(X_train, y_train) # fit the dataset
```

```
# Predict
y_prd = clsFier.predict(X_test)

# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value


# --------   K-folds cross-validation  -------------
from sklearn.model_selection import cross_val_score
accuRacies = cross_val_score(estimator=clsFier, X = X_train, y = y_train, cv= 10)
mean_accu = accuRacies.mean()
std_accu = accuRacies.std()



# --------   Grid-Search to find the "Best model" and 'Best hyper-parameters'-------------
from sklearn.model_selection import GridSearchCV
params_dic = [
            {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
            {'C': [1, 10, 100, 1000], 'kernel': ['rbf'], 'gamma' : [1, 0.9, 0.8, 0.7, 0.5, 0.1, 0.01]},
            ]

griD_Srch = GridSearchCV(estimator = clsFier,
                         param_grid= params_dic,
                         scoring= 'accuracy',
                         cv = 10,
                         n_jobs= -1     )
griD_Srch = griD_Srch.fit(X_train, y_train)
best_accu = griD_Srch.best_score_
best_parameters = griD_Srch.best_params_
```

11.2.1 XGBoost: Installation & Preparing the environment

Congratulations for reaching this last section of the Machine-Learning !!! We hope that you feel expert in machine learning and you're very confident in this world and that you are working on some fascinating machine learning projects.

❒ ***XGBoost*** is one of the most ***popular algorithm*** in machine learning that is quite recently popular but still a ***very powerful*** model especially if you work on ***large data-sets***.

☞ It will offer you very ***high performance*** while being ***fast*** to ***execute***.

✋ It is important to remind that ***XGBoost*** is the most powerful implementation of ***Gradient Boosting*** in terms of *model performance* and *execution speed*.

✋ ***XGBoost (eXtreme Gradient Boosting)*** is an **open-source software library** which provides a *regularizing Gradient Boosting* framework for ***C++***, ***Java***, ***Python***, ***R***, ***Julia***, ***Perl***, and ***Scala***.

☝ This is only going to be an introduction so we will make a simple implementation of ***XGBoost***. But this simple implementation will definitely give you some excellent performance.

❒ ***Install XGBoost and integrate it in spider:*** Visit the Official website: https://xgboost.ai/

☞ ***XGBoost*** is an optimized distributed *gradient boosting library* designed to be highly efficient, ***flexible*** and ***portable***. It implements *machine learning* algorithms under the *Gradient Boosting framework*. ***XGBoost*** provides a *parallel tree boosting* (also known as ***GBDT***, ***GBM***) that solve many *data science problems* in a ***fast*** and ***accurate*** way. To install latest version and installation guide visit following:

https://xgboost.readthedocs.io/en/stable/

https://xgboost.readthedocs.io/en/stable/install.html



❒ ***Install in windows:***

```
pip install --user xgboost
```

```
C:\Users\SolLaSi>pip install --user xgboost
Collecting xgboost
  Downloading xgboost-1.6.1-py3-none-win_amd64.whl (125.4 MB)
     |████████████████████████████████| 125.4 MB 56 kB/s
Requirement already satisfied: numpy in c:\users\sollasi\anaconda3\lib\site-packages (from xgboost) (1.22.3)
Requirement already satisfied: scipy in c:\users\sollasi\anaconda3\lib\site-packages (from xgboost) (1.5.2)
Installing collected packages: xgboost
Successfully installed xgboost-1.6.1

C:\Users\SolLaSi>
```

❒ ***Conda:*** You may use the Conda packaging manager to install XGBoost:

```
conda install -c conda-forge py-xgboost
```

☞ Conda should be able to detect the existence of a GPU on your machine and install the correct variant of XGBoost. If you run into issues, try indicating the variant explicitly:

```
# CPU only
conda install -c conda-forge py-xgboost-cpu

# Use NVIDIA GPU
conda install -c conda-forge py-xgboost-gpu
```

11.2.2 XGBoost usin Python: Problem description & Data-preprocessing

❒ ***Problem description:*** Remember this is the ***Churn Modeling problem*** where we need to predict the customers of the bank that will leave the bank. We solved it using ***ANN*** in ***Chapter 8: Deep Learning – ANN***. **1** for ***leave*** and **0** for ***stay***.

☞ Where we classify the customers in two classes: those who will ***leave the bank*** and those who will ***not leave the bank***.

☞ ***Accuracy using ANN:*** Remember for this problem we obtained an ***accuracy*** of ***86%*** but that took quite a while because we trained an ***ANN*** with many ***epochs***.

☯ In this section we're going to apply ***XGBoost*** on this ***Churn Modeling Problem*** and you're going to see that we will get the ***same accuracy***. We'll probably get ***86% accuracy*** but this will ***execute in no time*** like ***instantly*** compared to ANN.

[We can not get a higher accuracy anyway because the accuracy is limited by the problem itself in the sense that there isn't a 100% correlation between the informations of the customers and their decision to leave the bank (yes or no). So 86% is very close to the best accuracy we can obtain for this specific business problem.]

✋ Also notice that, this dataset only contains ***13 features*** so it's not a ***large dataset***.

✋ It is important to note that even if this was a very large data-set, ***XGBoost*** would be one of the ***best model*** in terms of ***performance***. That is to get a ***good accuracy*** and ***execution speed***.

☝ So if you are working with a ***large data-set*** I strongly encourage you to test ***XGBoost***.

❒ ***Data preprocessing:*** We'll take the pre processing phase from our ANN file.

☞ ***Feature scaling:*** Feature scaling is compulsory for Deep-Learning so we used it in ANN.

➢ But in ***XGBoost*** feature-scaling is ***not necessary***. Since ***XGBoost*** is a ***Gradient Boosting*** model with decision trees hence, ***features scaling is totally unnecessary***.

➢ That's one of the very good thing about ***XGBoost*** beside its high performance & high execution speed, you can ***keep*** the ***interpretation*** of your ***problem/data-set*** and of the ***results*** you'll get after building the model.

✋ That's why ***XGBoost*** is so popular because it has the three qualities:

☝ High **performance**

☝ Fast **execution speed**.

☝ You can **keep** all the **interpretation** of your problem and your model (**no feature-scaling**).

Data pre-processing for XGBoost

```
# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np



# --------------------------- Part 1 : Data Preprocessing --------------------------------
# Data Extract
dataSet = pd.read_csv("Churn_Modelling.csv")
# X = dataSet.iloc[:, 3:-1].values # this can be used too
```

```
X = dataSet.iloc[:, 3:13].values # all columns from index 3, excluding 13 indexed column
y = dataSet.iloc[:, 13].values # the last column


# ------------- Encode Categorical Data  -----------
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

#Encode "Gender" using LabelEncoder."Gender" is in "3rd-column", hence X[:, 2]
label_encode = LabelEncoder()
# Following is applicable to numpy array, if we used "X = dataSet.iloc[:, 3:13]"
# X = np.array(X) # it is needed if X is not an Arry. i.e. ".values" not applied
X[:, 2] = label_encode.fit_transform(X[:, 2])
print(X)

# For a data-set we can still encode it using Columns "key"
# X["Gender"] = label_encode.fit_transform(X["Gender"])


#Encode 'Gegraohy' using OneHotEncoder. "Gegraohy" is in "2nd-column", hence [1]
ct = ColumnTransformer(transformers = [("encoding", OneHotEncoder(), [1])], remainder = 'passthrough')
# remainder = 'passthrough' for remaining columns to be unchanged
X = ct.fit_transform(X)
X = np.array(X) # convert this output to NumPy array
print(X)
X = X[:, 1:] # Excluding 0-index column to avoid dummy-variable trap



# ------------------ Data Split ----------------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2, random_state = 0)
```

11.2.3 XGBoost usin Python: Apply the model

Here we are going to implement ***XGBoost*** by importing the ***classifier*** and creating an ***object*** of that classifier.

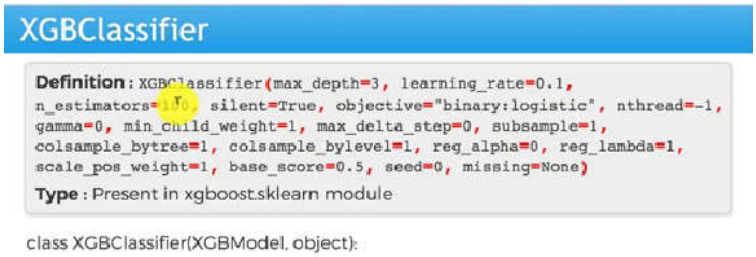
- Then we will ***fit*** this ***object*** to the ***training set*** and
- Then we will ***evaluate*** model ***performance*** making ***confusion matrix***.
- And we also apply ***K-fold-cross-validation*** for evaluating the ***model performance***

❒ ***Importing the Class:*** We import **XGBClassifier** from the module **xgboost** (we just installed it !!!).

```
from xgboost import XGBClassifier
```

- ***Parameters:*** When we're creating the object from **XGBClassifier**, we noticed that there is ***very little information*** about this class when we hit (ctrl+I or cmd+I), very ***little documentation***. The only thing we get is the ***list*** of the ***parameters*** we can ***input***.

- ***Some parameters are:*** ***learning_rate*** (as we had in deep learning), ***n_estimators*** *(the number estimators, because* ***xgboost*** *is actually a* ***Gradient Boosting Algorithm*** *with* ***Trees****. So the number of estimators is actually the number of trees)* and then we have some other parameters but we're not going to play with those parameters now.



- However, this is only an introduction to this very powerful algorithm but of course you can find a lot of information on the ***internet***.

- Now we're not going to input any parameters we're going to be satisfied with the default values here in this **XGBClassifier** class.

- So we are going to take **max_depth**= 3, **n_estimators** =100 (one hundred trees) and the rest is fine because actually we have a binary outcome.

- Also you can do a little practice and try to do some **Grid Search Parameter Tuning** for example to find the optimal parameters for the learning_rates or the n_ estimators. It's exactly the same **technique** as how we did when we **implemented *Grid Search*** in the ***Previous Section***.
  - ❖ We have another ***gamma*** parameter here that you can try to ***tune*** with ***grid search***.

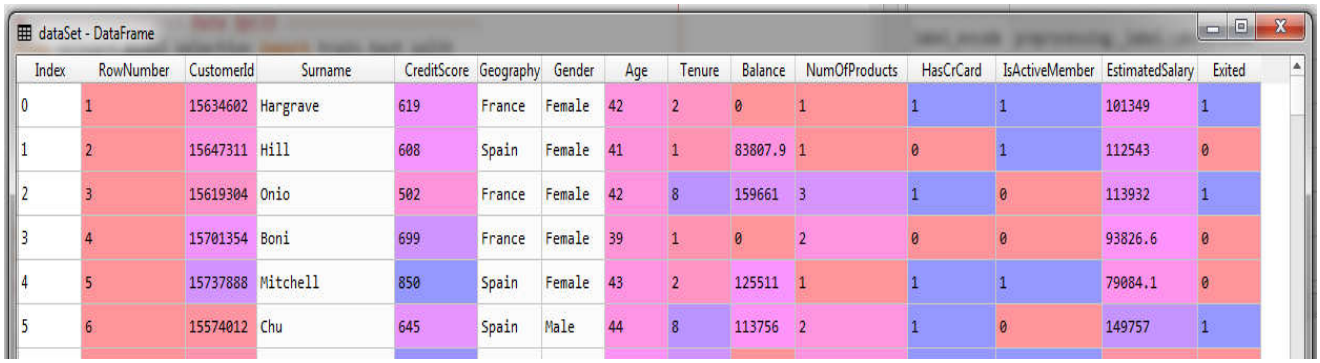
❒ ***Fitting this classified object to the training set:*** We just take our classifier object and fit the training-set. Exactly the same as when we implemented the ***other classification models***.

```
claSifire_XGB.fit(X_train, y_train)
```

```
# -------------- Fitting XGBoost in the training set -----------
from xgboost import XGBClassifier
claSifire_XGB = XGBClassifier()
claSifire_XGB.fit(X_train, y_train)
```

❒ ***Prediction, confusion matrix, k-fold-cross-validation:*** All are copied from previous projects. Just change the classifier name.

- We make some ***predictions*** on the test-set
- We also calculate the ***confusion-matrix***.
- Lastly we apply ***K-fold-cross-validation*** to get relevant performance metrics to assess the performance of our ***XGBoost*** model.



dataSet - DataFrame

Index	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	15634602	Hargrave	619	France	Female	42	2	0	1	1	1	101349	1
1	2	15647311	Hill	608	Spain	Female	41	1	83807.9	1	0	1	112543	0
2	3	15619304	Onio	502	France	Female	42	8	159661	3	1	0	113932	1
3	4	15701354	Boni	699	France	Female	39	1	0	2	0	0	93826.6	0
4	5	15737888	Mitchell	850	Spain	Female	43	2	125511	1	1	1	79084.1	0
5	6	15574012	Chu	645	Spain	Male	44	8	113756	2	1	0	149757	1

☯ The dataset contains some information of customers in a bank.

➲ The independent variables that we selected are all independent variables from ***credit score*** to ***estimated salary*** and the dependent variable is ***exited*** variable ***1*** for ***leave*** and ***0*** for ***stay***. These are the data of the previous six months that the Bank recorded.

➲ We were training this ***XGBoost*** model in the data set so that it understands the correlations between these information like the *credit score, the geography, the gender, age, the estimated salary* and the ***decision*** of the customer to ***leave*** the bank and therefore that's a classification problem.

➲ ***XGBoost*** will classify the customers between two classes the ones that leave and the ones that stay and then our goal is to make some ***predictions*** for the ***future customers*** and predict if they're going to ***leave*** the bank ***yes*** or ***no***.

```
# -------------------- Part 3 : Predictions and Evaluating the model ----------------------

# Predict
y_prd = claSifire_XGB.predict(X_test)


# Making the confusion matrix use the function "confusion_matrix"
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
print(f"\nConfusion Matrix :\n {cm}")
# parameters of cm: y_true: Real values, y_pred: Predicted value

accURacy_by_Confusion_matrix = (cm[0][0] + cm[1][1])/X_test.shape[0]
print(f"\nAccuracy = {accURacy_by_Confusion_matrix}%")


# --------   K-folds cross-validation  -------------
from sklearn.model_selection import cross_val_score
accuRacies = cross_val_score(estimator=claSifire_XGB, X = X_train, y = y_train, cv= 10)
mean_accu = accuRacies.mean()
std_accu = accuRacies.std()

print("\nAccuracy using k-fold-cross-validation: {:.2f} %".format(mean_accu*100))
print("\nStandard Deviation from k-fold-cross-validation: {:.2f} %".format(std_accu*100))
```

❒ ***Result:*** We have a lot of correct predictions we have 1491 correct predictions of customers that don't leave the bank and 218 of customers that leave the bank.

```
Confusion Matrix :
 [[1491  104]
 [ 187  218]]

Accuracy = 0.8545%

Accuracy using k-fold-cross-validation: 85.29 %

Standard Deviation from k-fold-cross-validation: 1.16 %
```

- ☞ And then we have ***104 + 187*** incorrect predictions. The accuracy is ***(1709/2000)=0.855*** i.e. ***85.5%*** accuracy.
- ☞ We know that's not the most relevant accuracy. The relevant accuracy we get ***85.29%*** from K-fold-cross-validation.
- ☞ We also get ***1.16 % deviation*** i.e. our models accuracy range is ***84.13%*** to ***86.45%***, $(85.29\% \pm 1.16\%)$
- ☞ Besides it's very difficult to improve the accuracy because as we said before it is limited by the problem itself.

All code at once (practiced version)

```
# ----------------- XGBoost instead of ANN -----------------


# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np



# -------------------------- Part 1 : Data Preprocessing ----------------------------
# Data Extract
dataSet = pd.read_csv("Churn_Modelling.csv")
# X = dataSet.iloc[:, 3:-1].values # this can be used too
X = dataSet.iloc[:, 3:13].values # all columns from index 3, excluding 13 indexed column
y = dataSet.iloc[:, 13].values # the last column


# ------------- Encode Categorical Data  -----------
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

#Encode "Gender" using LabelEncoder."Gender" is in "3rd-column", hence X[:, 2]
label_encode = LabelEncoder()
# Following is applicable to numpy array, if we used "X = dataSet.iloc[:, 3:13]"
# X = np.array(X) # it is needed if X is not an Arry. i.e. ".values" not applied
X[:, 2] = label_encode.fit_transform(X[:, 2])
print(X)

# For a data-set we can still encode it using Columns "key"
# X["Gender"] = label_encode.fit_transform(X["Gender"])


#Encode 'Gegraohy' using OneHotEncoder. "Gegraohy" is in "2nd-column", hence [1]
ct = ColumnTransformer(transformers = [("encoding", OneHotEncoder(), [1])], remainder = 'passthrough')
# remainder = 'passthrough' for remaining columns to be unchanged
X = ct.fit_transform(X)
X = np.array(X) # convert this output to NumPy array
print(X)
X = X[:, 1:] # Excluding 0-index column to avoid dummy-variable trap



# ------------------ Data Split -----------------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2, random_state = 0)
```

```python
# --------------------------- Part 2 :Fitting XGBoost in the training set---------------------------
from xgboost import XGBClassifier
claSifire_XGB = XGBClassifier()
claSifire_XGB.fit(X_train, y_train)


# --------------------------- Part 3 : Predictions and Evaluating the model ---------------------------

# ----- Predict -----
y_prd = claSifire_XGB.predict(X_test)


# -------- confusion matrix ------
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
print(f"\nConfusion Matrix :\n {cm}")
# parameters of cm: y_true: Real values, y_pred: Predicted value

accURacy_by_Confusion_matrix = (cm[0][0] + cm[1][1])/X_test.shape[0]
print(f"\nAccuracy = {accURacy_by_Confusion_matrix}%")


# --------   K-folds cross-validation  -------------
from sklearn.model_selection import cross_val_score
accuRacies = cross_val_score(estimator=claSifire_XGB, X = X_train, y = y_train, cv= 10)
mean_accu = accuRacies.mean()
std_accu = accuRacies.std()

print("\nAccuracy using k-fold-cross-validation: {:.2f} %".format(mean_accu*100))
print("\nStandard Deviation from k-fold-cross-validation: {:.2f} %".format(std_accu*100))

# python prctc_XGBoost.py
```

Chapter 11 : Part 3

Deep Learning

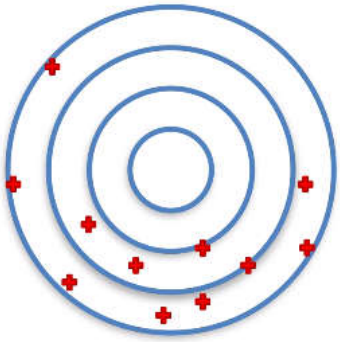
ANN: Evaluate performance using K-fold-CV

&

Hyper parameter tuning

K-fold-CV or K-fold-Cross-Validation, keras-wrapper, Dropout Regularization

11.3.1 K-fold-CV in ANN



- ***K-fold-CV in ANN:*** In the previous section we discussed ***K-fold-CV*** and applied to ***kernel-SVM*** & ***XGBoost***. In this section we'll apply ***K-fold-CV*** to ***ANN***, it is our first Deep-learning Algorithm.

- In ***Chapter 8: ANN***, we trained our ***artificial neural network*** twice and we noticed that the ***accuracies*** are different. So we want to apply ***K-fold-CV*** to evaluate Model performance accurately.

- ***Bias-Variance Tradeoff:*** We're trying to ***train*** a ***model*** that will not only be ***accurate***, but also that should not have ***too much variance*** of ***accuracy***, when we ***train*** it ***several times***.
  - Previously we trained our ANN twice and we obtained two **different accuracies**, ***85%*** and then ***84%***. Which one of these two accuracies we should take to ***evaluate*** our ***models performance***?
  - We did ***split*** our data set between a ***training-set*** and a ***test-set***. It results the ***Variance Problem***. When we get the ***accuracy*** on the ***test set***, if we ***run*** the model ***again*** and ***test again*** it's performance on ***another test set***, well, we can get a very ***different accuracy***.
  - So, judging our model performance only on one accuracy on one test set, is not the most relevant way, to evaluate the model performance.
  - That's why we use where train-set (or whole dataset) divided into equal k-groups and hence k-combination of (k-1)-train-group and 1-test-group, hence k-iterations. Finally we get a vector of k-accuracies for whicg we calculate the mean-accuracy and standared-deviation. It gives us the better evaluation of the model performance.

- Most of the time, K= 10, we train our model on 9-folds and we test it on the last remaining fold. With 10-folds there are 10 different combinations of 9-folds-train & 1-fold-test.
  - That means we can ***train*** and ***test*** the ***model*** on ten ***combinations*** of training and test sets. Hence ***10*** different ***accuracies***.
  - That will give us a much better idea of the model performance, by taking average of the different accuracies of the ten evaluations and also compute the standard deviation.
  - We then categorize the model-performance into 4-categiries:
    i. good accuracy - small variance;
    ii. large accuracy - high variance;
    iii. small accuracy - low variance;
    iv. low accuracy - high variance

11.3.2 Implement K-fold CV in ANN

We implement K-fold CV after the data pre-processing phase.

- ***Keras wrapper:*** We implemented our ***ANN*** model with ***`Keras-TensorFlow`***. But the ***`K-fold-CV`*** belongs to ***`Scikit-learn`***.
  - ***`Keras wrapper`*** combines ***`Keras`*** and ***`Scikit-learn`*** together. ***`Keras wrapper`*** module belongs to ***`Keras`***. It will wrap ***`K-fold cross validation`*** by ***`Scikit-learn`***, into the ***`Keras model`***.
  - We will be able to include K-fold cross validation in our Keras classifier.

```
from keras.wrappers.scikit_learn import KerasClassifier      # to combine Keras & Scikit-learn
from sklearn.model_selection import cross_val_score
```

➢ We use ***KerasClassifier*** to prepare to ***fit*** an ANN for **each iteration** of ***K-fold-CV***. That ANN is invoked using ***cross_val_score***. That's how both ***Keras*** & ***Scikit-Learn*** are ***combined*** together.

❒ ***build_ANN_clsfire():*** Since the ANN-model is not an *one-line code* like previous ***Regression/Classification models***, we need a function to define our ANN-model. That's why we need to create ***build_ANN_clsfire()***, it just *define* our ***ANN-model*** and returns the ***ANN-classifier***.

☞ Remember we don't need to ***fit*** the model *inside* this defined *function*.

☞ Also notice we ***Compile*** the ANN-classifier inside this ***build_ANN_clsfire()*** function, i.e. a *compiled ANN-classifier* is ***returned*** from this function.

☞ This function is simply a function that *returns* the *classifier* that we made here with all this *architecture for our ANN-model* (the initial-layer, different layers and ***Compile***). Basically this function just builds the architecture of our ANN.

```
def build_ANN_clsfire():
        # initialize the ANN
        ANN_clsfire = Sequential()

        # "input-layer" &  "first Hidden-layer"
        ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11))

        #  "second Hidden-layer"
        ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))

        # "output-layer"
        ANN_clsfire.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))

        # Compile the ANN
        ANN_clsfire.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"])

        return ANN_clsfire
```

➢ It is exactly as we first built our ANN. Basically we just copied all the code from that section including compiling (part 2) except fitting part. Later we'll fit each ANN-model (returned by this function) using ***cross_val_score***.

➢ We need to return the ANN-classifier before this function get out from the scope. Hence the return statement.

☞ We're also gonna define this function because the ***KerasClassifier()*** expects for one of its arguments a function, **build_fn =** `build_ANN_clsfire`. Following is called after the defined function:

```
ann_clsfier_for_eval = KerasClassifier(build_fn= build_ANN_clsfire, batch_size= 10, epochs= 100)
```

❒ ***KerasClassifier():*** The ***KerasClassifier*** is the wrapper of *K-fold cross validation*.

☞ The variable ***ann_clsfier_for_eval*** stores the classifier that is created by ***KerasClassifier*** (by invoking ***build_ANN_clsfire***, with specified ***batch_size*** & ***epochs***).

☞ ***KerasClassifier*** actually prepare each ANN for ***sklearn*** class ***cross_val_score***, which expecting a ***sklearn*** based classifier:

```
ann_clsfier_for_eval = KerasClassifier(build_fn= build_ANN_clsfire, batch_size= 10, epochs= 100)
```

☞ Here ***ann_clsfier_for_eval*** is the ***global classifier*** variable (is the object of ***KerasClassifier*** class), because the classifier `ANN_clsfire` inside `build_ANN_clsfire()` is a ***local variable*** because it is a variable inside the function.

☞ Classifier ***ann_clsfier_for_eval*** will be built through *k-fold cross validation* on *10* different *training folds* and by each time measuring the model performance on *one test fold*.

☞ ***Parameters:***

➢ **build_fn:** It takes the defined function ***build_ANN_clsfire*** that builds the architecture of our ANN. So, we set **build_fn=** `build_ANN_clsfire`

➢ ***batch_size:*** Since we're not using ***fit()***, we need to specify the ***batch-size*** here. In the `cross_val_score` we just use ***X_train***, ***y_train***. We set `batch_size= 10`

➢ ***epoch:*** Same goes for epoch we also need to specify it here. We set `epochs= 100`

❒ ***cross_val_score():*** As we did previously, we create a variable for accuracy-vector named **acuRacies** and we set our ***global classifier*** variable ***ann_clsfier_for_eval*** as the ***estimator***.

```
acuRacies = cross_val_score(estimator= ann_clsfier_for_eval, X = X_train, y = y_train, cv = 10, n_jobs = None)
```

☞ All other things are same as we did before.

☞ ***n_jobs*** *(-1 is important in DL):* Note that here ***n_jobs = -1*** is ***crucial*** for **ANN** and all **Deep-Learning techniques**. Because all ***Deep-Learning techniques*** are ***parallel computation process*** that makes them ***slow***. Moreover we're applying **K-fold-CV**, that makes the model **K-time slower**, because the model literally runs **K-times** and execution time is ***K-time longer***.

➢ Hence we need to use ***CPU's full capacity*** for faster computing. For this reason we have to use ***n_jobs = -1*** for all *Deep-learning techniques*. (here we use `n_jobs = None` in case you need to do other works in your PC) .

```
# ------------------ Part 4 : Evaluating, Improving  & Tuning the ANN --------------------------------

# --------  Evaluating ANN : K-fold-CV -----------
from keras.wrappers.scikit_learn import KerasClassifier     # to combine Keras & Scikit-Learn
from sklearn.model_selection import cross_val_score

# ----------- make a function that creates ANN ------------
from keras.models import Sequential
from keras.layers import Dense

# structure of our ANN model
def build_ANN_clsfire():
    ANN_clsfire = Sequential() # initialize the ANN
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11)) # "input-Layer" &  "first Hidden-Layer"
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu")) #  "second Hidden-Layer"
    ANN_clsfire.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid")) # "output-Layer"
    ANN_clsfire.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"]) # Compile the ANN
    return ANN_clsfire

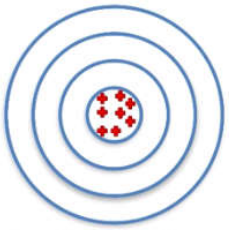
ann_clsfier_for_eval = KerasClassifier(build_fn= build_ANN_clsfire, batch_size= 10, epochs= 100) # creates/compile an ANN classifier
acuRacies = cross_val_score(estimator= ann_clsfier_for_eval, X = X_train, y = y_train, cv = 10, n_jobs = None) # compile ANN 10-times with 10-fold

mean_accu = acuRacies.mean()
st_dvi_accu = acuRacies.std()
```

❒ To calculate mean and standard deviation we use following:

```
mean_accu = acuRacies.mean()
st_dvi_accu = acuRacies.std()
```

☞ The mean accuracy is 83.3% and standard-deviation is 1%. So we are in following category:



Low Bias Low Variance

acuRacies - NumPy object array

	0
0	0.8325
1	0.83875
2	0.83375
3	0.8275
4	0.85
5	0.83375
6	0.8325
7	0.83375
8	0.81
9	0.84625

mean_accu	float64	1	0.8338750004768372
st_dvi_accu	float64	1	0.010253814766620795

All code at once (practiced)

```
# Artificial Neural Network
# Install : Tensorflow, Keras and Theano libraries.

# Library
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

```
# ------------------------- Part 1 : Data Preprocessing --------------------------------
# Data Extract
dataSet = pd.read_csv("Churn_Modelling.csv")
# X = dataSet.iloc[:, 3:-1].values # this can be used too
X = dataSet.iloc[:, 3:13].values # all columns from index 3, excluding 13 indexed column
y = dataSet.iloc[:, 13].values # the last column


# ------------- Encode Categorical Data  -----------
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

#Encode "Gender" using LabelEncoder."Gender" is in "3rd-column", hence X[:, 2]
label_encode = LabelEncoder()
# Following is applicable to numpy array, if we used "X = dataSet.iloc[:, 3:13]"
# X = np.array(X) # it is needed if X is not an Arry. i.e. ".values" not applied
X[:, 2] = label_encode.fit_transform(X[:, 2])
print(X)

# For a data-set we can still encode it using Columns "key"
# X["Gender"] = label_encode.fit_transform(X["Gender"])



#Encode 'Gegraohy' using OneHotEncoder. "Gegraohy" is in "2nd-column", hence [1]
ct = ColumnTransformer(transformers = [("encoding", OneHotEncoder(), [1])], remainder = 'passthrough')
# remainder = 'passthrough' for remaining columns to be unchanged
X = ct.fit_transform(X)
X = np.array(X) # convert this output to NumPy array
print(X)
X = X[:, 1:] # Excluding 0-index column to avoid dummy-variable trap



# ------------------ Data Split --------------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2, random_state = 0)

        # Feature-Scaling after Data Split



# --------------- Feature-Scaling ------------------
from sklearn.preprocessing import StandardScaler
#  y dependent variable, need not to be scaled: categorical variable, 0 and 1
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)




# ------------------------ Part 2 : Creating ANN model ----------------------------

    # 1. importing "keras" libraries and packages
# from tensorflow import keras
import keras # using TensorFlow backend
from keras.models import Sequential
from keras.layers import Dense

    # 2. initialize the ANN
ann_classifier = Sequential()

    # 3. Add the "input-layer" and  "first Hidden-Layer"
# ann_classifier.add(Dense(output_dim = 6, init = "uniform", activation = "relu", input_dim = 11))
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11))

    # 4. Add the "second Hidden-Layer"
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))

    # 5. Add the "output-layer"
ann_classifier.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))

    # 6. Compile the ANN
ann_classifier.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"])

    # 7. Train the model: fit the ANN to Training-set (batch_size and epoch)
ann_classifier.fit(X_train, y_train, batch_size= 10, epochs= 100)
```

```python
# ------------------ Part 3 : Predictions and Evaluating the model ---------------------------------

# Predict
y_prd = ann_classifier.predict(X_test)

# coverting probabilities into "true/false" form. because 1 for leaving the Bank
y_prd = (y_prd > 0.5)

# Making the confusion matrix use the function "confusion_matrix"
# Class in capital letters, functions are small letters
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value

accURacy = (cm[0][0] + cm[1][1])/X_test.shape[0]
print(f"Accuracy = {accURacy}%")



# ------- prediction of new data-point using the built model ----------------

"""
Use our ANN model to predict if the customer with the following informations will leave the bank:

    Geography:      France
    Credit Score:   600
    Gender:         Male
    Age:            40 years old
    Tenure:         3 years
    Balance:        $60000
    Number of Products:     2
    Does this customer have a credit card ?     Yes
    Is this customer an Active Member:  Yes
    Estimated Salary:   $50000

So should we say goodbye to that customer ?
"""

# first create a 2D "NumPy array" in our X_train's format.
    # it will be smilar to a single row of our X_train
    # 2 "[" used to define a single row of a 2-D array

# new_dt_pt = np.array([[0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0]])
# new_dt_pt= st_x.transform(new_dt_pt) # scaling
# predict_data_pt = (ann_classifier.predict(new_dt_pt) > 0.5) # Predict the data-point




# ------------------ Part 4 : Evaluating, Improving  & Tuning the ANN ---------------------------------

# --------  Evaluating ANN : K-fold-CV -----------
from keras.wrappers.scikit_learn import KerasClassifier     # to combine Keras & Scikit-Learn
from sklearn.model_selection import cross_val_score

# ----------- make a function that creates ANN ------------
from keras.models import Sequential
from keras.layers import Dense

# structure of our ANN model
def build_ANN_clsfire():
    ANN_clsfire = Sequential() # initialize the ANN
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11)) # "input-Layer" &  "first Hidden-Layer"
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu")) #  "second Hidden-Layer"
    ANN_clsfire.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid")) # "output-Layer"
    ANN_clsfire.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"]) # Compile the ANN
    return ANN_clsfire

ann_clsfier_for_eval = KerasClassifier(build_fn= build_ANN_clsfire, batch_size= 10, epochs= 100) # creates/compile an ANN classifier
acuRacies = cross_val_score(estimator= ann_clsfier_for_eval, X = X_train, y = y_train, cv = 10, n_jobs = None) # compile ANN 10-times with 10-fold

mean_accu = acuRacies.mean()
st_dvi_accu = acuRacies.std()

# -------- Improving ANN : Dropout-Regularization ------------



# --------  Tuning the ANN : Grid-Search ------------



# python prctc_ANN_eval_KFldCV.py
```

11.3.3 Dropout-Regularization: Improving the ANN

From above we see that our models accuracy is ***83%***, using ***K-fold-CV***, here we can improve that by modifying our ***ANN-layer*** using ***Dropout-regularization***.

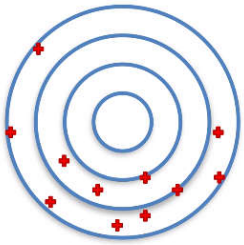
❒ In deep learning ***Dropout Regularization*** is a very important ***technique*** that ***prevents Overfitting***. (***Overfitting:*** Model was trained too much on the training-set, that it becomes much less performance on the test-set.)

❒ In that case we have a large difference of ***accuracies*** between ***training set*** and the ***test set***.

☞ Generally, when ***overfitting*** happens, you have a much ***higher accuracy*** on the ***training set*** than the ***test set***.

☞ And another way to detect overfitting is when you observe a ***high variance*** when applying ***k-fold cross-validation***

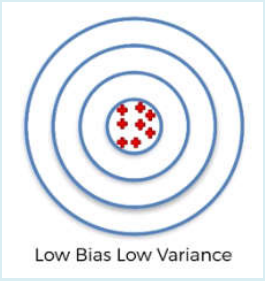
➢ In this case, in your ***vector*** of ***accuracies*** of k-fold-CV, you'll get some ***high accuracies*** and some ***low accuracies*** and therefore you have ***high variance*** and that's how you detect ***overfitting*** in your model.



High Bias High Variance

✋ Since we definitely ***didn't get overfitting*** in our ***ANN-model***, because we get **"Low-Bias & Low-variance accuracy"**, we actually don't need to use ***Dropout*** (however we're doing it here for learning purpose). The accuracies were more or less around 83%.

mean_accu	float64	1	0.8338750004768372
st_dvi_accu	float64	1	0.010253814766620795



Low Bias Low Variance

👽 ***Where do we need to apply dropout to our ANN:***

☞ ***Dropout works as:*** At each iteration of the training, some ***neurons*** of your ***ANN*** are ***randomly disabled*** to prevent them from being ***too dependent*** on ***each other*** when they learn the correlations.

☞ Therefore, by **over-writing** these **neurons**, the ***ANN*** learns ***several independent correlations*** in the data, because each time there is not the **same configuration** of the neurons.

☞ We get these ***independent correlations*** of the data, because now the ***neurons*** work more ***independently***, that prevents the ***neurons*** from learning ***too much*** and therefore that ***prevents overfitting***.

So we need to apply ***Dropout*** to our Hidden-layers, after defining a layer. For example: following Dropout applied to first-hidden layer.

```
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11))
ann_classifier.add(Dropout(p = 0.1))
```

🛠 ***Implementing Dropout-Regularization:***

☞ First we need to import a new class besides the ***Sequential*** class and the ***Dense*** class to modify our ANN This class is the ***Dropout*** class, from ***keras.layers***.

```
import keras # using TensorFlow backend
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
```

☞ Where exactly in the ***neural network*** are we going to apply dropout?

➢ Since ***Dropout*** is applied to the ***neurons*** so that some of them ***randomly*** become ***disabled*** at each ***iteration***. So basically we need to ***apply Dropout*** to the ***layers***. It can be to ***one layer***, or it can be to ***several layers***.

➲ Here we're going to apply to ***several layers*** (for demonstration purpose). We'll apply it to the ***first hidden layer*** and to the ***second hidden layer***.

✋ One advice is that when you have ***overfitting***, you should apply **Dropout** to ***all the layers***, because that will give you ***more chance*** to ***reduce overfitting***.

🛠 ***Applying dropout:*** As usual we use ***add()*** to apply ***Dropout*** in a layer. We do it after defining each layer.

```
    # 3. Add the "input-layer" and  "first Hidden-layer"
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11))
ann_classifier.add(Dropout(p = 0.1))

    # 4. Add the "second Hidden-layer"
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))
ann_classifier.add(Dropout(p = 0.1))
```

☞ **`p = 0.1: p`** is *floating-point-number* in range ***(0, 1)***. It's the *fraction* of the *input* you want to ***drop***. Basically that's the ***fraction*** of the ***neurons*** you want to ***drop***, or *disable* at each *iteration*. In newer version ***p*** is replaced with ***`rate`***.

☯ ***For example:*** Suppose we have ***10 neurons***, if we choose ***P*** equals ***0.1***, i.e. ***10%***, that means that at each iteration, ***one*** *neuron* will be ***disabled***. If ***p*** equals ***0.2***, ***two*** *neurons* will be ***disabled***.

☯ ***Which value of P(`rate`) should we input?*** The key "***p***" is replaced with "***`rate`***."

➲ Our advice is that when you have *overfitting*, you start trying with ***p*** or ***`rate`*** equals ***0.1***, ***10%***, and then if it *doesn't solve* the problem, if you still have *overfitting*, then you try a *higher* value of ***`rate`***. And you *increment* it for example by ***0.1***.

➲ So you first try with ***0.1*** and then if you still have *overfitting*, you try with ***0.2***. If you still have *overfitting* you try with ***0.3***. And until you manage to *reduce overfitting*.

➲ ***Too higher value*** of ***`rate`*** (***p***) is ***bad***, when you disable *most* of the *neurons* of a *layer*, you'll get is *not overfitting* but *underfitting*. For example if *p=0.99* nothing will be *learnt*, *most neuron* will be *turned off*.

➲ So in general ***don't try*** to go ***over 0.5*** because then you'll get ***`too close`*** to ***`underfitting`***.

➲ And so what we recommend is to try with ***0.1***, then try ***0.2***, ***0.3***, ***0.4*** and that should do it.

☞ After applying ***Dropout*** to *first-hidden layer*, we just need to ***copy*** the ***line-of-code*** and apply it to the *second-hidden-layer*.

All ANN-structure at once with Dropout applied

```
# ------------------------ Part 2 : Creating ANN model ----------------------------

    # 1. importing "keras" libraries and packages
import keras # using TensorFlow backend
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout

    # 2. initialize the ANN
ann_classifier = Sequential()

    # 3. Add the "input-layer" and  "first Hidden-layer"
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11))
ann_classifier.add(Dropout(rate = 0.1))     # used 'rate' instead of 'p'

    # 4. Add the "second Hidden-layer"
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))
ann_classifier.add(Dropout(rate = 0.1))     # used 'rate' instead of 'p'

    # 5. Add the "output-layer"
ann_classifier.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))

    # 6. Compile the ANN
ann_classifier.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"])

    # 7. Train the model: fit the ANN to Training-set (batch_size and epoch)
ann_classifier.fit(X_train, y_train, batch_size= 10, epochs= 100)
```

11.3.4 ANN Hyper-Parameter tuning

❒ ***Model-parameter:*** Are the parameters that the model learn by itself.

❒ ***Hyper parameters:*** Are the parameters that we fix to build a model. Those are the number of ***epoch***, the ***batch size***, the ***optimizer***, or the ***number of neurons*** in the **layers**.

🛠 ***Parameter-Tuning & Grid-Search:*** It's a real deal about ***improving*** our model's ***performance*** because we're going to tune our model and we're gonna find the best ***hyper parameters***, like the best of number of ***epoch***, the best ***batch size***, the best ***optimizer***, so that we get the best ANN that will allow us to maximize our accuracy.

☞ So *Parameter tuning* is all about *finding* the *best values* of these *hyper parameters* and we are gonna do this using ***Grid Search*** technique.

☞ ***Grid Search*** will test several *combinations* of these *values* and will eventually return the *best selection* that leads to the *best accuracy* with ***K-fold-CV***.

❒ ***Implement parameter tuning in ANN:*** It's actually quite the same as implementing ***K-fold-CV*** because we will use the ***KerasClassifier*** class to wrap our ANN in a classifier, so that it can be used for ***scikit_learn***.

☞ Because we'll use another class ***GridSearchCV*** from the same ***sklearn.model_selection*** (used for k-fold-CV, *cross_val_score* **function**).

➢ For old version, try to import it from ***scikitlearn.grid_search***

```
from sklearn.model_selection import GridSearchCV
```

☞ Basically we will **create** an **object** from ***GridSearchCV*** we name it ***gridsearch_ann*** that will apply parameter tuning on our ***KerasClassifier*** that is our *traditional NN*.

➢ So we can copy all our code from ***K-fold-CV section except*** the accuracy-vector ***acuRacies***, and edit it for Grid-Search. (changed lines are highlighted).

```
from keras.wrappers.scikit_learn import KerasClassifier     # to combine Keras & Scikit-Learn
from sklearn.model_selection import GridSearchCV

# ----------- make a function that creates ANN ------------
from keras.models import Sequential
from keras.layers import Dense

# structure of our ANN model
def build_ANN_clsfire():
    ANN_clsfire = Sequential() # initialize the ANN
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11))
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))
    ANN_clsfire.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))
    ANN_clsfire.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"])
    return ANN_clsfire

ann_clsfier_for_eval = KerasClassifier(build_fn= build_ANN_clsfire) # creates/compile an ANN classifier
```

➢ But in our ***KerasClassifier*** object we will ***not input*** the ***batch_size*** and ***epoch*** arguments because that's the arguments we're gonna ***tune*** in ***Grid-Search***.

☞ ***Creating parameter dictionary:*** To use **Grid-search** for **Hyper-parameter tuning**, we need to ***input*** those ***Hyper-parameters*** as a ***list*** of ***dictionary***. So first we create this dictionary of parameters. We name this dictionary of parameters as ***params_dic***.

➢ It is the same procedure described in the **section *11.1.3 Grid-Search*** earlier in this chapter.

➢ In this dictionary we'll define different *combinations* of *hyper-parameters* and the *Grid-Search* uses the *k-fold-CV* with those different combinations and at the end it will return the *best accuracy* with the *best selection* of these *parameters*.

❒ ***Selecting parameters:***

i. ***batch size:*** we can try several batch sizes because one of them will lead us to a better accuracy. We're going to input different values of the ***batch_size*** in the dictionary.
ii. ***epoch:*** We can also tune the number of epoch, there is an optimal number of epoch.
iii. ***Optimizer:*** We can also tune some hyper parameters in our ANN architecture, like the optimizer.

```
params_dic = [
    {
        'batch_size': [25, 32],
        'epochs' : [2, 3],
        'opTmzr' : ["rmsprop"]
    },
    {
        'batch_size': [24],
        'epochs' : [5],
        'opTmzr': ["adam", "rmsprop"]
    }
]
```

Tuning optimizer: We want to tune the optimizer. How do we input some different values, considering that we ***already*** have a ***values*** of the optimizer in ***`compile()`***?

You know we didn't have values for the number of ***epoch*** and the ***batch size***, so we can try several of them here, but here we already have a value for the ***`optimizer`***. So how can we test some other ones?

We have to use a ***parameter*** in the ***`build_ANN_clsfire`***, function, and this new argument is of course going to be, an argument that will give us choice for the optimizer.

```
def build_ANN_clsfire(opTmzr):
```

We replace the Adam optimizer **`'adam'`** by this optimizer argument **`opTmzr`** that now plays the role of a variable.

```
ANN_clsfire.compile(optimizer = opTmzr, loss = "binary_crossentropy", metrics= ["accuracy"])
```

Note that: the parameter **`opTmzr`** in the defined function "**`def`** build_ANN_clsfire(**`opTmzr`**):" must be the same as the key in the parameters-dictionary:

```
{
    'batch_size': [25, 32],
    'epochs' : [2, 3],
    'opTmzr' : ["rmsprop"]
},
```

i.e. ***`parameter`*** in the **function** & ***`key`*** in the **dictionary** must be the same.

And so now, we have the ***right*** to ***input different values*** that we want to test for our optimizer this key **`opTmzr`**, will be ***associated*** to ***key*** of the ***dictionary*** and so when we give ***different*** values for this ***key***, well the ***`different values`*** are going to be tested in this optimizer in ***`compile()`***.

So that's the trick, and therefore if you want to tune a hyper parameters that are in this architecture here, you have to create a new parameter in the ***`buid_function`***,

```
build_ANN_clsfire(param1, param2, . .)
```

```
def build_ANN_clsfire(opTmzr):
    ANN_clsfire = Sequential() # initialize the ANN
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11))
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))
    ANN_clsfire.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))
    ANN_clsfire.compile(optimizer = opTmzr, loss = "binary_crossentropy", metrics= ["accuracy"])
    return ANN_clsfire

ann_clsfier_for_tune = KerasClassifier(build_fn= build_ANN_clsfire) # creates/compile an ANN classifier
params_dic = [
    {
        'batch_size': [25, 32],
        'epochs' : [2, 3],
        'opTmzr' : ["rmsprop"]
    },
    {
        'batch_size': [24],
        'epochs' : [5],
        'opTmzr': ["adam", "rmsprop"]
    }
]
```

➲ For optimizers, we're going to try "**adam**", and '**rmsprop**'.

- Sometimes ***rmsprop*** optimizer is better one for ***deep learning models***. It is another excellent optimizer based on ***stochastic gradient descent***.
- ***Keras*** documentation recommend to use this ***rmsprop*** for ***RNN***, this is generally a ***better choice*** and indeed this is the ***optimizer*** that we're going to use for ***RNN***.
- But lets still try it for our ***ANN***, maybe this will lead us to better results.

❒ ***Implement Grid Search:*** To implement grid search, we use the same code that we did earlier in this chapter.

```
grid_sch = GridSearchCV(
    estimator= ann_clsfier_for_tune,
    param_grid= params_dic,
    scoring= 'accuracy',
    cv = 5
)
```

- **estimator**: is the classifier that we created for tuning, ***ann_clsfier_for_tune***
- **param_grid**: to use different parameter combination we use the list of parameter-dictionaries ***params_dic***
- **scoring**: is the scoring matrix , we use ***'accuracy'*** as scoring matrix.
- **cv** = is the no. of ***fold*** for ***k-fold-CV***. We use ***5 folds*** here. When we tune our model with ***gridsearch***, ***k-fold cross validation*** is going to be used to ***evaluate*** the ***accuracy***.

❒ ***To fit grid search object:*** We need to fit grid search to the training set, it is same as before.

```
grid_sch = grid_sch.fit(X_train, y_train)
```

❒ To view the best parameter-combination use the following lines of code:

```
best_parametrs = grid_sch.best_params_
best_accuracy = grid_sch.best_score_
print(f"\n\tBest parameters = {best_parametrs} \n\tBest Accuracy = {best_accuracy}")
```

Hyper-Parameter Tuning Part

```
# --------  Tuning the ANN : Grid-Search ------------
from keras.wrappers.scikit_learn import KerasClassifier     # to combine Keras & Scikit-Learn
from sklearn.model_selection import GridSearchCV

# ----------- make a function that creates ANN ------------
from keras.models import Sequential
from keras.layers import Dense

# structure of our ANN model
def build_ANN_clsfire(opTmzr):
    ANN_clsfire = Sequential() # initialize the ANN
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim =
11))
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))
    ANN_clsfire.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))
    ANN_clsfire.compile(optimizer = opTmzr, loss = "binary_crossentropy", metrics= ["accuracy"])
    return ANN_clsfire

ann_clsfier_for_tune = KerasClassifier(build_fn= build_ANN_clsfire) # creates/compile an ANN classifier
params_dic = [
    {
        'batch_size': [25, 32],
        'epochs' : [2, 3],
        'opTmzr' : ["rmsprop"]
    },
    {
```

```
        'batch_size': [24],
        'epochs' : [5],
        'opTmzr': ["adam", "rmsprop"]
    }
]

grid_sch = GridSearchCV(
    estimator= ann_clsfier_for_tune,
    param_grid= params_dic,
    scoring= 'accuracy',
    cv = 5
)

grid_sch = grid_sch.fit(X_train, y_train)
best_parametrs = grid_sch.best_params_
best_accuracy = grid_sch.best_score_
print(f"\n\tBest parameters = {best_parametrs} \n\tBest Accuracy = {best_accuracy}")

# python prctc_ANN_imprv_DrpRg_grd_sch.py
```

❒ ***Execute-by-part (do not execute all-code):*** Since we're not gonna execute previous ANN-part-2 with Dropout-regularization and Evaluations-part (k-fold-CV):

[1]. First we need to import the data set and run the data reprocessing phase,

[2]. Finally ***execute*** the ***last section***, ***parameter tuning***, we have the whole tuning ready.

All code at once (practiced, Used small values in parameter tuning)

```
# Artificial Neural Network
# Install : Tensorflow, Keras and Theano libraries.

# Library
import pandas as pd
import matplotlib.pyplot as pLt
import numpy as np




# -------------------------- Part 1 : Data Preprocessing --------------------------------
# Data Extract
dataSet = pd.read_csv("Churn_Modelling.csv")
# X = dataSet.iloc[:, 3:-1].values # this can be used too
X = dataSet.iloc[:, 3:13].values # all columns from index 3, excluding 13 indexed column
y = dataSet.iloc[:, 13].values # the last column



# ------------- Encode Categorical Data  -----------
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

#Encode "Gender" using LabelEncoder."Gender" is in "3rd-column", hence X[:, 2]
label_encode = LabelEncoder()
# Following is applicable to numpy array, if we used "X = dataSet.iloc[:, 3:13]"
# X = np.array(X) # it is needed if X is not an Arry. i.e. ".values" not applied
X[:, 2] = label_encode.fit_transform(X[:, 2])
print(X)

# For a data-set we can still encode it using Columns "key"
# X["Gender"] = label_encode.fit_transform(X["Gender"])



#Encode 'Gegraohy' using OneHotEncoder. "Gegraohy" is in "2nd-column", hence [1]
ct = ColumnTransformer(transformers = [("encoding", OneHotEncoder(), [1])], remainder = 'passthrough')
# remainder = 'passthrough' for remaining columns to be unchanged
X = ct.fit_transform(X)
X = np.array(X) # convert this output to NumPy array
print(X)
X = X[:, 1:] # Excluding 0-index column to avoid dummy-variable trap



# ------------------ Data Split --------------------
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2, random_state = 0)
```

```
        # Feature-Scaling after Data Split



# ---------------- Feature-Scaling ------------------
from sklearn.preprocessing import StandardScaler
#  y dependent variable, need not to be scaled: categorical variable, 0 and 1
st_x= StandardScaler()
X_train= st_x.fit_transform(X_train)
X_test= st_x.transform(X_test)




# -------------------- Part 2 : Creating ANN model with  "Dropout-Regularization" ---------------------

    # 1. importing "keras" libraries and packages
# from tensorflow import keras
import keras # using TensorFlow backend
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout


    # 2. initialize the ANN
ann_classifier = Sequential()

    # 3. Add the "input-Layer" and  "first Hidden-Layer" and applying "Dropout-Regularization"
# ann_classifier.add(Dense(output_dim = 6, init = "uniform", activation = "relu", input_dim = 11))
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11))
# ann_classifier.add(Dropout(p = 0.1))
ann_classifier.add(Dropout(rate = 0.1)) # used 'rate' instead of 'p'

    # 4. Add the "second Hidden-Layer" and applying "Dropout-Regularization"
ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))
# ann_classifier.add(Dropout(p = 0.1))
ann_classifier.add(Dropout(rate = 0.1)) # used 'rate' instead of 'p'

    # 5. Add the "output-Layer"
ann_classifier.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))

    # 6. Compile the ANN
ann_classifier.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"])

    # 7. Train the model: fit the ANN to Training-set (batch_size and epoch)
ann_classifier.fit(X_train, y_train, batch_size= 10, epochs= 100)




# ------------------ Part 3 : Predictions and Evaluating the model ----------------------------------

# Predict
y_prd = ann_classifier.predict(X_test)

# coverting probabilities into "true/false" form. because 1 for leaving the Bank
y_prd = (y_prd > 0.5)

# Making the confusion matrix use the function "confusion_matrix"
# Class in capital letters, functions are small letters
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_true= y_test, y_pred= y_prd)
# parameters of cm: y_true: Real values, y_pred: Predicted value

accURacy = (cm[0][0] + cm[1][1])/X_test.shape[0]
print(f"Accuracy = {accURacy}%")



# ------- prediction of new data-point using the built model ----------------

"""
Use our ANN model to predict if the customer with the following informations will leave the bank:

    Geography:      France
    Credit Score:   600
    Gender:         Male
    Age:            40 years old
    Tenure:         3 years
    Balance:        $60000
    Number of Products:     2
    Does this customer have a credit card ?     Yes
    Is this customer an Active Member:  Yes
    Estimated Salary:   $50000

So should we say goodbye to that customer ?
"""
```

```python
# first create a 2D "NumPy array" in our X_train's format.
    # it will be smilar to a single row of our X_train
    # 2 "[" used to define a single row of a 2-D array

new_dt_pt = np.array([[0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0]])
new_dt_pt= st_x.transform(new_dt_pt) # scaling
predict_data_pt = (ann_classifier.predict(new_dt_pt) > 0.5) # Predict the data-point




# ------------------ Part 4 : Evaluating, Improving  & Tuning the ANN ---------------------------------

# --------  Evaluating ANN : K-fold-CV -----------
from keras.wrappers.scikit_learn import KerasClassifier     # to combine Keras & Scikit-Learn
from sklearn.model_selection import cross_val_score

# ----------- make a function that creates ANN ------------
from keras.models import Sequential
from keras.layers import Dense

# structure of our ANN model
def build_ANN_clsfire():
    ANN_clsfire = Sequential() # initialize the ANN
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11)) # "input-layer" &  "first Hidden-layer"
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu")) #  "second Hidden-layer"
    ANN_clsfire.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid")) # "output-layer"
    ANN_clsfire.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"]) # Compile the ANN
    return ANN_clsfire

ann_clsfier_for_eval = KerasClassifier(build_fn= build_ANN_clsfire, batch_size= 10, epochs= 100) # creates/compile an ANN classifier
acuRacies = cross_val_score(estimator= ann_clsfier_for_eval, X = X_train, y = y_train, cv = 10, n_jobs = None) # compile ANN 10-times with 10-fold

mean_accu = acuRacies.mean()
st_dvi_accu = acuRacies.std()




# -------- Improving ANN : Dropout-Regularization ------------
# Done in "part -2 Creating ANN model"




# --------  Tuning the ANN : Grid-Search ------------
from keras.wrappers.scikit_learn import KerasClassifier     # to combine Keras & Scikit-Learn
from sklearn.model_selection import GridSearchCV

# ----------- make a function that creates ANN ------------
from keras.models import Sequential
from keras.layers import Dense

# structure of our ANN model
def build_ANN_clsfire(opTmzr):
    ANN_clsfire = Sequential() # initialize the ANN
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu", input_dim = 11))
    ANN_clsfire.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))
    ANN_clsfire.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))
    ANN_clsfire.compile(optimizer = opTmzr, loss = "binary_crossentropy", metrics= ["accuracy"])
    return ANN_clsfire

ann_clsfier_for_tune = KerasClassifier(build_fn= build_ANN_clsfire) # creates/compile an ANN classifier
params_dic = [
    {
        'batch_size': [25, 32],
        'epochs' : [2, 3],
        'opTmzr' : ["rmsprop"]
    },
    {
        'batch_size': [24],
        'epochs' : [5],
        'opTmzr': ["adam", "rmsprop"]
    }
]

grid_sch = GridSearchCV(
    estimator= ann_clsfier_for_tune,
    param_grid= params_dic,
    scoring= 'accuracy',
    cv = 5
)

grid_sch = grid_sch.fit(X_train, y_train)
best_parametrs = grid_sch.best_params_
best_accuracy = grid_sch.best_score_
print(f"\n\tBest parameters = {best_parametrs} \n\tBest Accuracy = {best_accuracy}")
```

❒ ***To get even more accuracies you will have several options:***

[1]. Change the architecture of your neural network

[2]. Do some more parameter tuning.

Result: best parameters are ***bach size*** of ***25*** a number of ***epoch*** of ***500*** and an ***rmsprop*** optimizer and so it's with these parameters that we manage to get an 85% accuracy.

NOTE: You result may be different due to choice of different parameters & parameters value.

I'll give you some hints and a solution to achieve more accuracy

So much repetition, so we have to manage the code. Our code is so wet.

Chapter 12 : Part 1

Deep Learning

RNN: Recurrent Neural Network

`Introduction`

12.1.1 What we will learn in this Chapter

RNN is one of the most advanced algorithms that exists in the world of ***Supervised Deep Learning***. We will cover following topics in this chapter.

[1]. ***The idea behind Recurrent Neural Networks (RNN):*** We'll see how they compare to the ***human brain***, we'll understand what makes them ***unique*** and ***special*** as compared to regular ***ANN***.

[2]. ***The Vanishing Gradient Problem:*** It has been a major **road block** in the **development** and **utilization** of ***RNN***, something that prevented ***RNNs*** from being what they are now.

[3]. ***Long Short-Term Memory (LSTM):*** One of the most popular solutions to the ***Vanishing Gradient Problem*** is the ***Long Short-Term Memory*** or ***LSTM*** neural networks.

- Here we'll talk about ***LSTM architecture***.
- We will find out exactly LSTMs work and what that complex structure is inside them by break it down into simple terms.

[4]. ***LSTM Variations:*** We'll see some other options of ***LSTMs*** exist out there, what other architectures you might come across in your work.

[5]. ***Step by Step Example:*** We'll look at some great examples posted by one of the researchers.

- We'll understand even better on an ***intuitive level*** how ***LSTMs*** actually work, how they think.
- Here we'll be like ***neuroscientists*** trying to understand what's going on in the ***brain*** of an ***LSTM***.

12.1.2 Deep Learning Methods and Human Brain

❒ Here we have break down some ***`Deep Learning Algorithms`*** into ***`supervised`*** and ***`unsupervised`*** categories. So ***ANN***, ***CNN*** and ***RNN*** are supervised algorithms. And **SOM**, **Deep Boltzmann Machines** and **AutoEncoders** are unsupervised algorithms.

Supervised	Artificial Neural Networks	Used for Regression & Classification
	Convolutional Neural Networks	Used for Computer Vision
	Recurrent Neural Networks	Used for Time Series Analysis
Unsupervised	Self-Organizing Maps	Used for Feature Detection
	Deep Boltzmann Machines	Used for Recommendation Systems
	AutoEncoders	Used for Recommendation Systems

❒ We can compare the whole concept behind deep learning with the human brain. We can get kind of similar functions as the human brain has.

☞ It would be pretty cool if we could somehow link the different ***branches*** of ***deep learning*** that we've discussed, or the algorithms that we discussed.

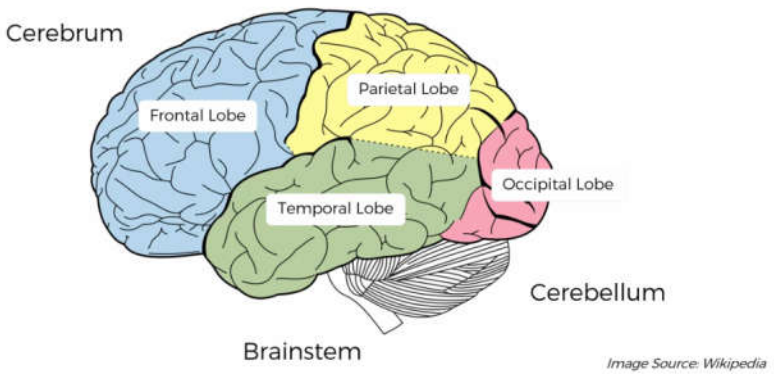


Image Source: Wikipedia

☞ Here we've got the brain, it's got main three parts.

i. ***Cerebrum:*** Which is all of this colored part. Frontal lobe, Temporal lobe, Occipital lobe, Parietal lobe.

ii. ***Cerebellum:*** Which is underneath Cerebrum there and that's the little brain.

- ➢ We discussed cerebellum in ANNs chapter (that was a big orange picture). But it doesn't represent ANN.

iii. ***Brainstem:*** Which connects the brain to the organs such as: arms, legs and so on.

☞ Now, in the ***CEREBRUM*** has four lobes:

[1]. Frontal lobe,

[2]. Temporal lobe,

[3]. Occipital lobe,

[4]. Parietal lobe

☞ ***Weights are Long Term Memory of a neural network - TEMPORAL lobe:*** ***ANN*** is the main part of ***deep learning***. In ANN we had, *Input/Output* layers and *Hidden layers*. Those layers are created by nodes called *neurons*. Inside each *neuron* we had "*activation-function*" also we had "*weights*" for *Synapses*. We had *forward/back propagation*, ***cost*** function.

- ➢ The fact that ANNs can learn through prior experience, or through prior impulse, and through prior observations.

- But the main thing about ANNs, are the ***Weights***. And philosophically those ***weights*** represent ***long term memory***.
- So once you've run your **ANN** and you've **trained** it, you can **switch it off**. But it **remembers** the ***weights***. So whatever input you put into it, it will process it the same way as it would ***yesterday***, as it will ***tomorrow***, as it will the day after.
- So, the ***weights*** are ***long term memory*** of a ***neural network***.

☑ That's why the ***ANN*** similar to ***temporal lobe***. Philosophically, ***ANNs*** are a start to deep learning and they represent ***long term memory***. So, we've to put them in the ***temporal lobe*** because the ***Temporal Lobe*** is responsible for long term memory.

☞ ***CNN represents OCCIPITAL lobe:*** Then, ***CNN*** represents ***vision***, ***recognition*** of images and objects and so on, so that's the ***OCCIPITAL lobe***.

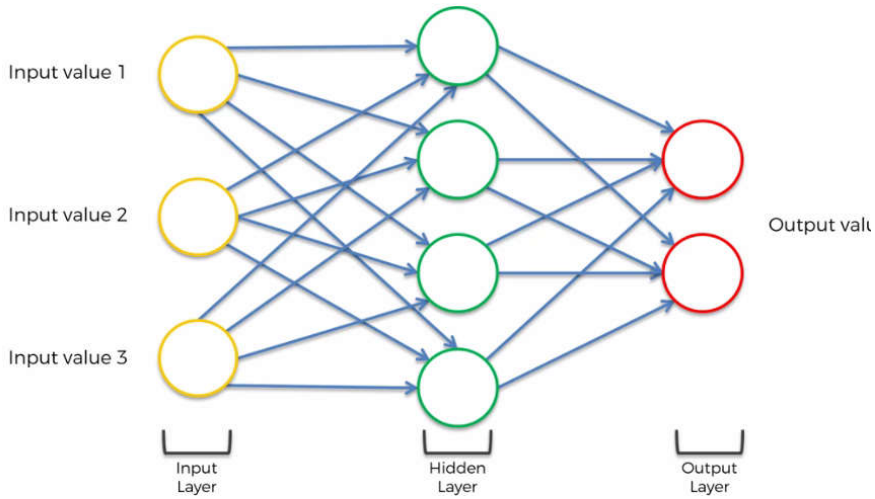
☞ ***RNN represents Frontal lobe:*** ***RNNs*** are much more like ***short term memory***. They can remember things that just happened in the previous couple of observations and apply that knowledge going forward.

- So, ***RNNs*** similar to the ***frontal lobe***. That's where we have a lot of the short term memory.
  (The frontal lobe also is responsible for personality, behavior, motor cortex, working memory, and lots of other things.)

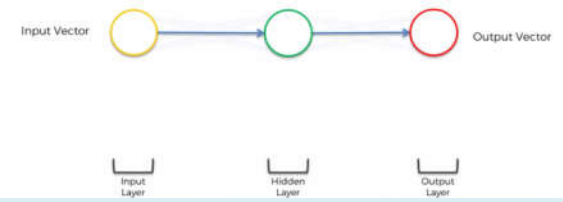
12.1.3 How RNN works

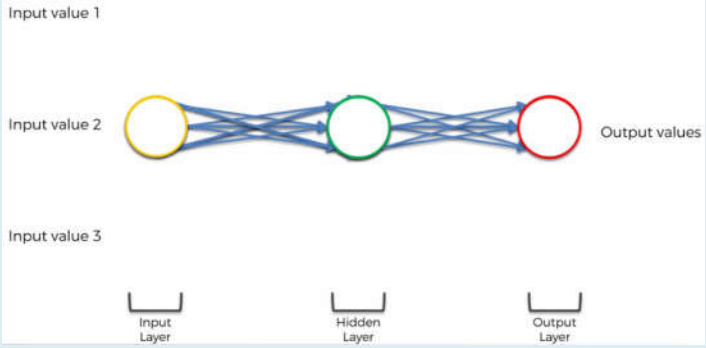
Here, we've got a simple ANN. Three inputs, two outputs and one hidden layer.

❒ We turn this into an RNN by squashing it. But remember that those neurons, the whole network, is still there. Nothing changed, we just squashed it for our purposes.



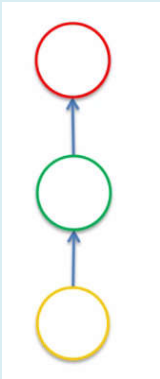
☞ Then to simplify things we're just going to change these ***multipliers/synapses*** into ***two***, then we're gonna ***twist*** thing whole thing, to make it ***vertical*** because that's the ***standard representation***.

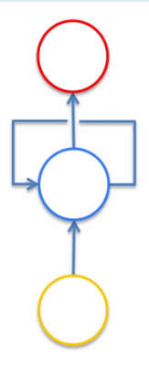




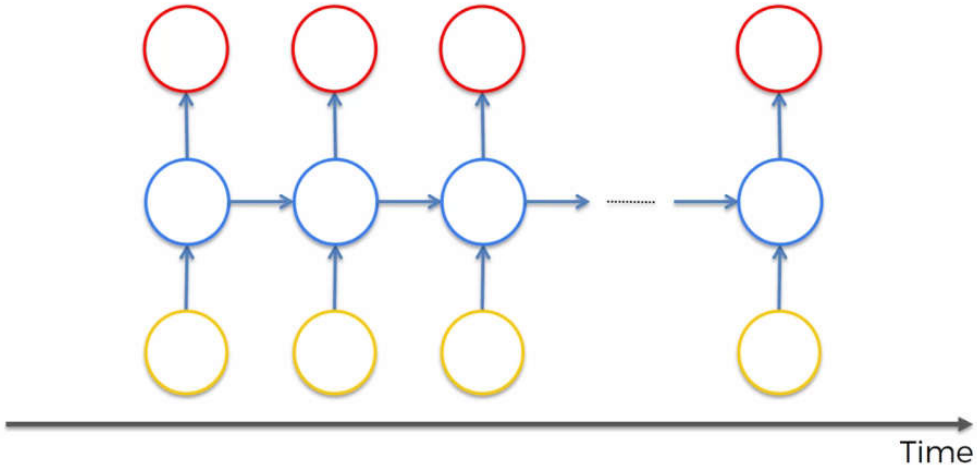
☞ Then in terms of ***neural metrics*** we're gonna ***color*** them, instead of ***green*** we're gonna color the ***hidden layers*** in ***blue***.

☞ And we're gonna add a ***loop***, represents the ***temporal loop***. Means that this hidden layer not only gives an output but also feeds back into itself.





☞ If we ***unwind***, or ***unroll***, this temporal loop then it represents the following ***ANNs***



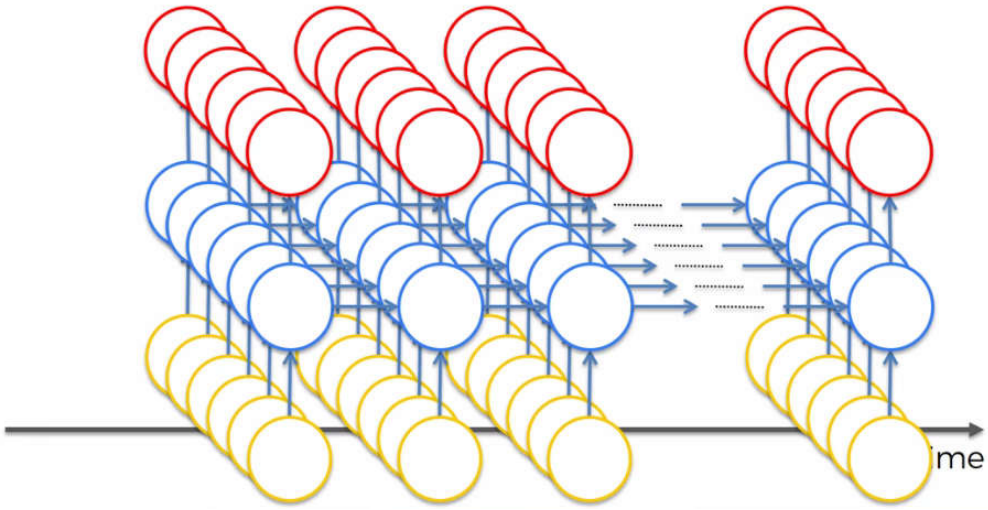
☝ ***But keep in mind:*** we're looking in a new dimension, that the layers are actually still there (all ***circles*** now represents ***layers***).

☞ We just remember that each one of these circles is not one neuron, it's a whole layer of neurons.

❒ ***Short Term Memory:*** In above figure we can see that, we've ***inputs*** coming into the ***neurons***, then we got ***outputs***, but also the neurons are ***connecting*** to ***themselves*** through ***time***.

☞ So, that's the whole ***concept*** of ***short term memory***, that they ***remember*** what was in that ***neuron*** just ***previously*** and before that.

☞ It just ***remembers*** what it was ***previously***, and that allows them to ***pass information*** on to ***themselves*** in the ***future*** and analyze things.



☯ ***For example:*** In our case we are talking about RNN in this chapter and to figure it out we learned ANN in previous chapters (i.e. we keep in our memory ANN stuffs like: Activation-function, neuron, weights etc from previous chapter so that we can understand RNN). It enables us to understand RNN in this chapter. So to learn new things in this current chapter we need to remember previous chapters (so some kind of short term memory is happening in our Brain).

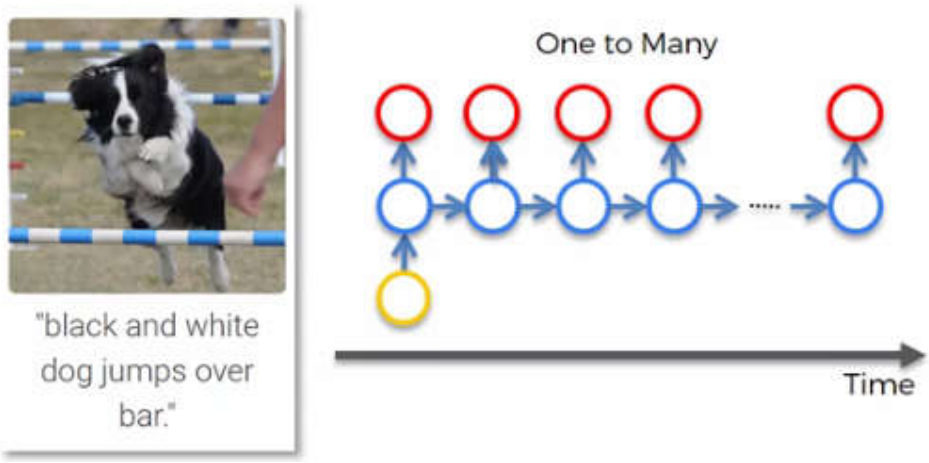
➲ And same thing here, we are mimicking the human brain.

☝ ***Short Term Memory is powerful:*** Long term memory is great, but short term memory is more powerful. And that's where recurrent neural networks come in.

Example: Following are some examples from Karpathy blog, ***karpathy.github.io***,

☯ ***One to many relationships:*** This is when you have ***one input*** and have ***multiple outputs***. An example of this is an image where a ***Computer Describes The Image***. So, you have one ***input***, the ***image***, and that would go through a ***CNN*** and then it would be fed into an ***RNN***, and then the computer would come up with words to describe the image. And this is an actual computer describing the image.

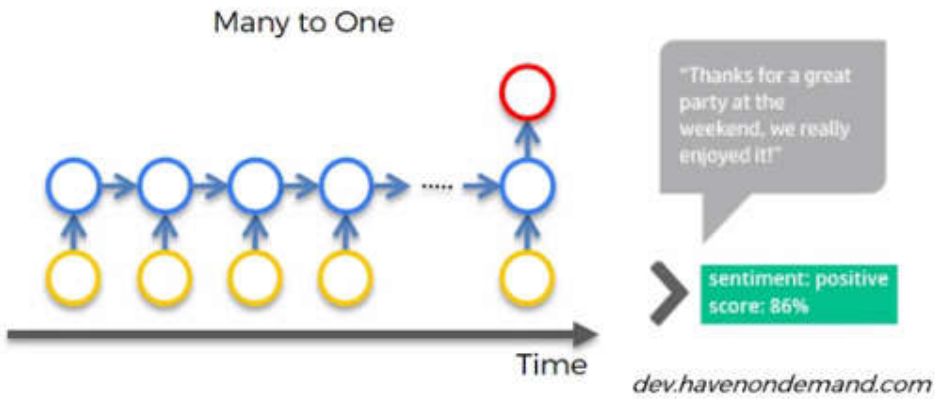
"***Black and white dog jumps over bar***".



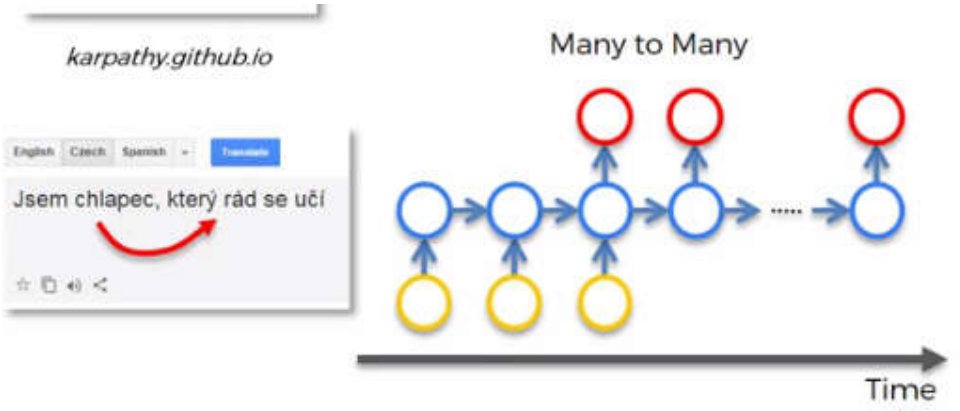
➲ This is a computer that looked at this image and it recognize the "black and white dog" from its training using , the long term memory (ANN ***weights***), and using ***CNN***.

➲ And then the RNN allows it to ***make sense out of the sentence***. So, you can see that the sentence actually flows. There's an ***and***, there's an ***over the bar***, and then there's like a ***verb***, there's a ***noun***, and so on. So, basically the ***RNN*** is what allows it to put a sentence together in this case.

☯ ***Many to one:*** An example would be ***sentiment analysis***. So, when you have a lot of text and from that text you kind of need to find out that: ***is this a Positive comment/ Negative comment***? What's the chance that it's a positive, or how negative is that comment?



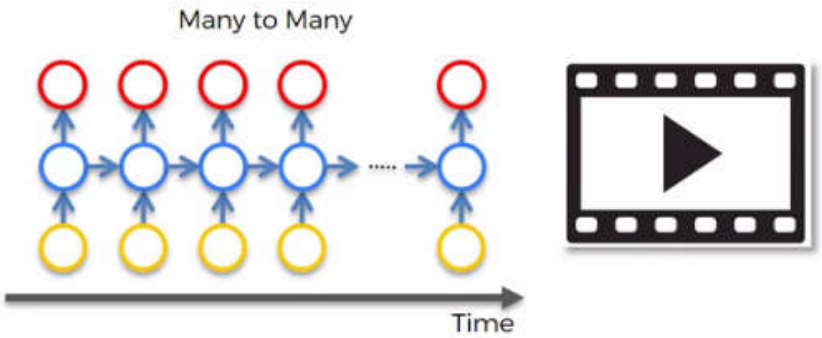
☯ ***Many to many:*** Here, we've got an example of Google Translator. In translation it find out the related word.





➲ So, if I say here from **English** to **Czech**. I say, **"I am a boy who likes to learn"**. In other languages it is important what **gender** your person is, right? So, here **boy** is **male**. It finds the *male-gender related word*. If I change this to ***girl*** in ***English*** nothing changes. But in ***Czech***, the other words ***change***.

➲ Another many to many example, you can use RNNs to subtitle movies.



Reference: karpathy.github.io

❒ ***Additional watching:*** Here is a movie called ***Sunspring*** in 2016 directed by ***Oscar Sharp***. And this movie was entirely written by ***Benjamin*** who is an RNN, an LSTM to be specific.

- ☞ There's actually an interview of ***Benjamin*** and he actually gave himself the ***name*** of ***Benjamin***, that's why they call him ***Benjamin***.
- ☞ What you'll find amazing is that Benjamin is able to ***construct sentences*** which kind of ***make sense*** for the most part, which is good, but what he lacks is kind of the ***bigger picture*** (relation ***between*** the ***subjects*** that makes ***sense***.). He cannot come up with a ***plot*** that ***consistently makes sense***.
- ☞ When you're watching, ***separate*** the ***sentences*** and you'll see that each sentence on its own more or less, ***90%*** of the time, makes sense. But overall, he can't properly ***link sentences together*** (link is lost to different sentence). And that's the next step for RNNs, we have to fix this in future.

Additional Reading:

Sunspring (movie, 2016)

- Directed By Oscar Sharp
- Written by Benjamin



Link:

https://arstechnica.com/the-multiverse/2016/06/an-ai-wrote-this-movie-and-its-strangely-moving/

12.1.4 Vanishing gradient Problem

Vanishing gradient was first discovered by ***Sepp Hochreiter***. And the second person is ***Yoshua Bengio*** (professor at the University of Montreal).

The Vanishing Gradient Problem

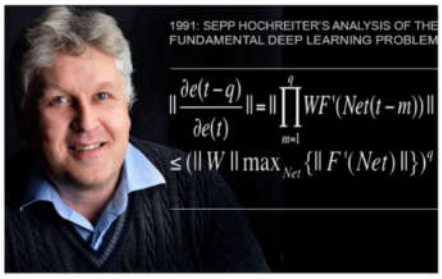
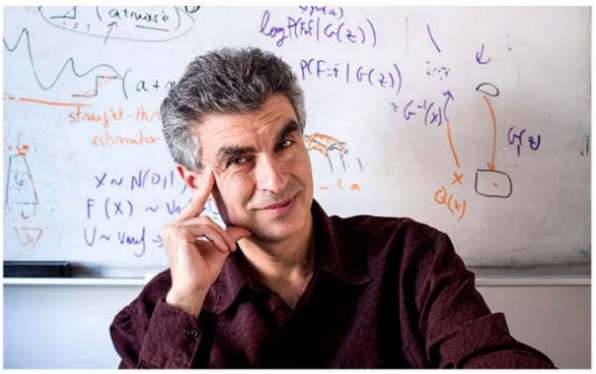
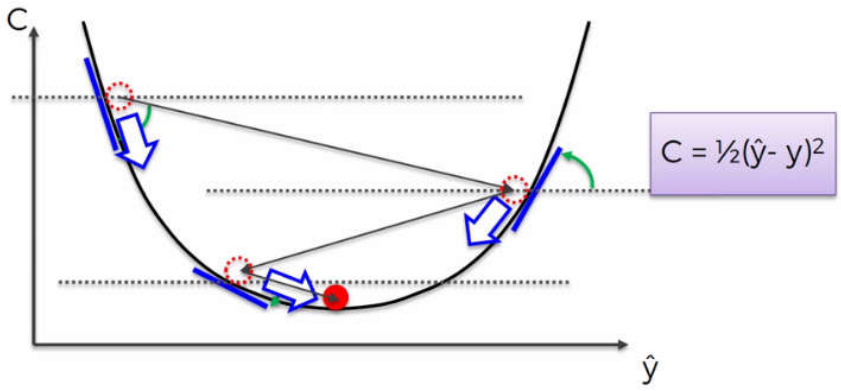


Image Source: people.idsia.ch





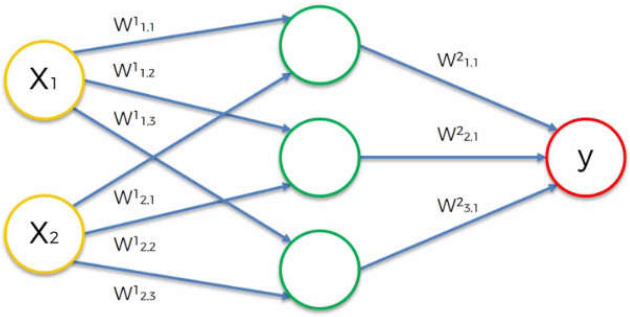
❒ ***Vanishing gradient problem:*** So, as you remember, following is the ***gradient descent algorithm***. We're trying to find the ***global minimum*** of your ***cost function***, and that's gonna be the ***optimal solution***, optimal ***setup*** for your ***neural network***.

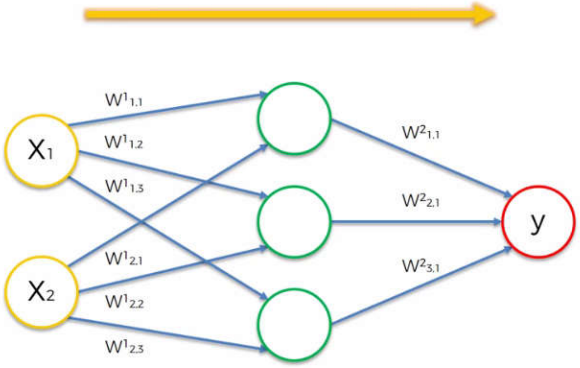


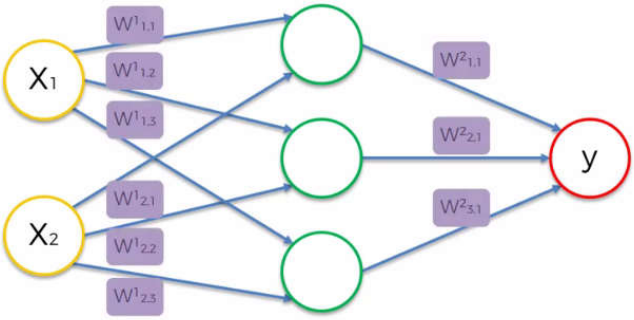
☞ Our information ***travels through*** NN to get output, and then the ***error*** is calculated and is ***propagated back*** through the NN to ***update*** the ***weights***.

☞ It is same as ***ANN*** but here all the ***circles*** represents ***Layers*** (not nodes).

☞ Every single ***node*** here is not just a node, it's a representation of a ***whole layer of nodes***. There's lots more neurons behind the ones that we can actually see because each one represents a layer.

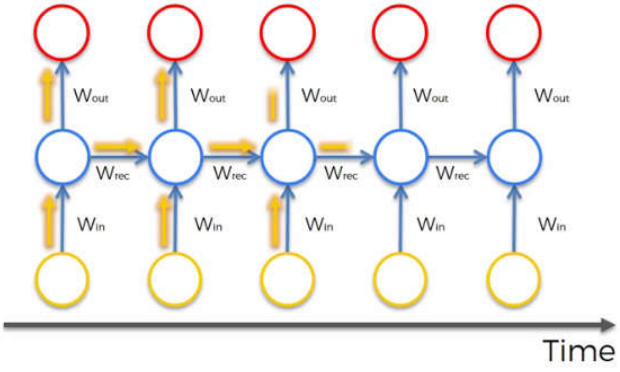




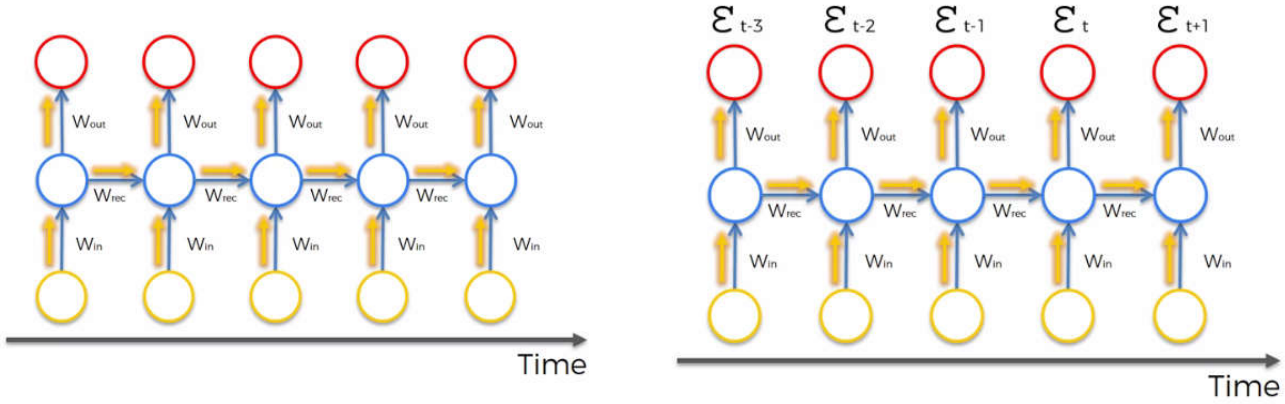


☞ In a RNN is a similar thing, when your information travels through the network it travels like this:

☞ It travels through ***time*** and ***information flows*** from previous ***time-points***, keeps coming/going through the network, and remember that every node here is a ***whole layer of nodes***.

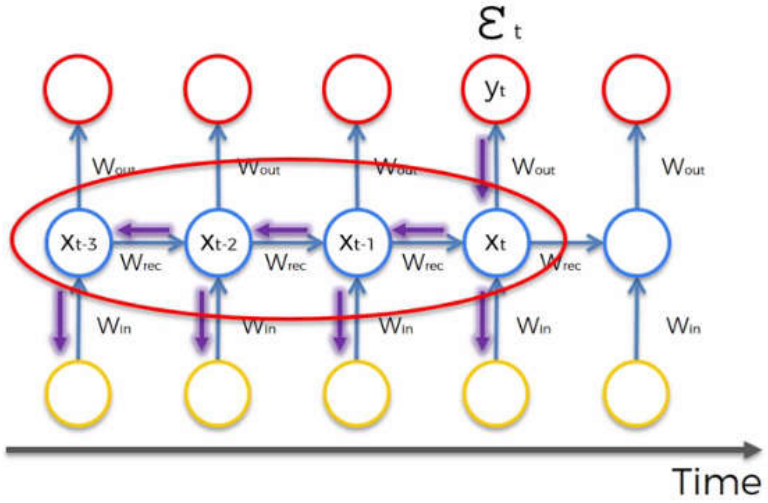


☞ So, at ***each point*** in ***time*** you can calculate your ***`cost function`***, or your ***`error`***. During the training, ***cost function*** compares your ***output*** (the red circle) to your ***desired output***. Then you get these ϵ_t values throughout the ***time series*** (for a single ***red circle***, calculates the ***cost function***).



❒ ***How weights got updated (Gradient decent occurs):*** Now let's focus on one value, lets consider just single ϵ_t. Here we've calculated the ***cost function*** epsilon-t: ϵ_t. Now we want to propagate that ***cost function*** back through the ***network***.

☞ To do this, every ***single neuron*** which ***participated*** in the calculation of the ***output associated*** with this ***cost function*** : ϵ_t, should ***update*** their ***weights***, in order to ***minimize*** that ***error***.

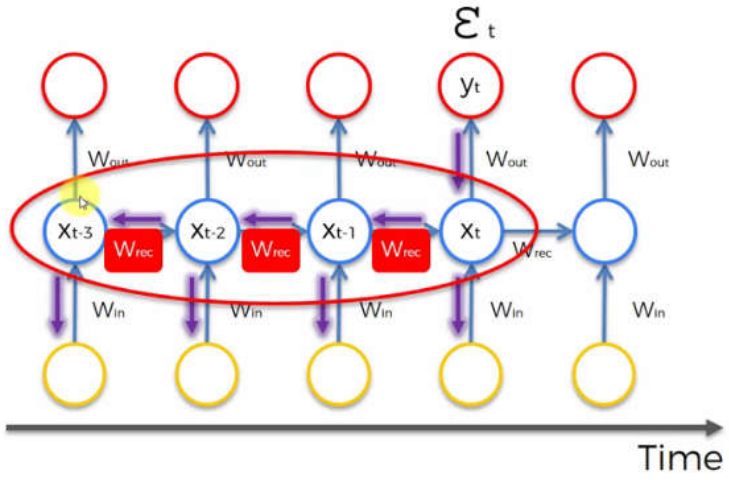


☞ But we have to note that, it's not just the neurons directly below (directly below that red circle) : ϵ_t.

➢ It's all the neurons that contributed (i.e. all previous time-point), all of these ***neurons*** as far back as you go. Depending on how many ***time-steps*** you take.

➢ For example, you might take ***50 time-steps*** before, then you have to update weights for all previous ***50 time-steps***. You have to ***propagate*** all the way ***back through time*** to those ***neurons***.

❒ Following is the math behind RNNs, we've got W_{rec} , and W_{rec} (stands for ***weight recurring***), and that is the ***weight*** that is used to ***connect*** the ***hidden layers*** to ***themselves*** in the ***unrolled*** temporal ***loop***.



$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \le t \le T} \frac{\partial \mathcal{E}_t}{\partial \theta} \quad (3)$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right) \quad (4)$$

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \ge i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \ge i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1})) \quad (5)$$

☞ Here we can see that to propagate from the layer X_{t-3} to the next layer X_{t-2}, we need to apply W_{rec}.

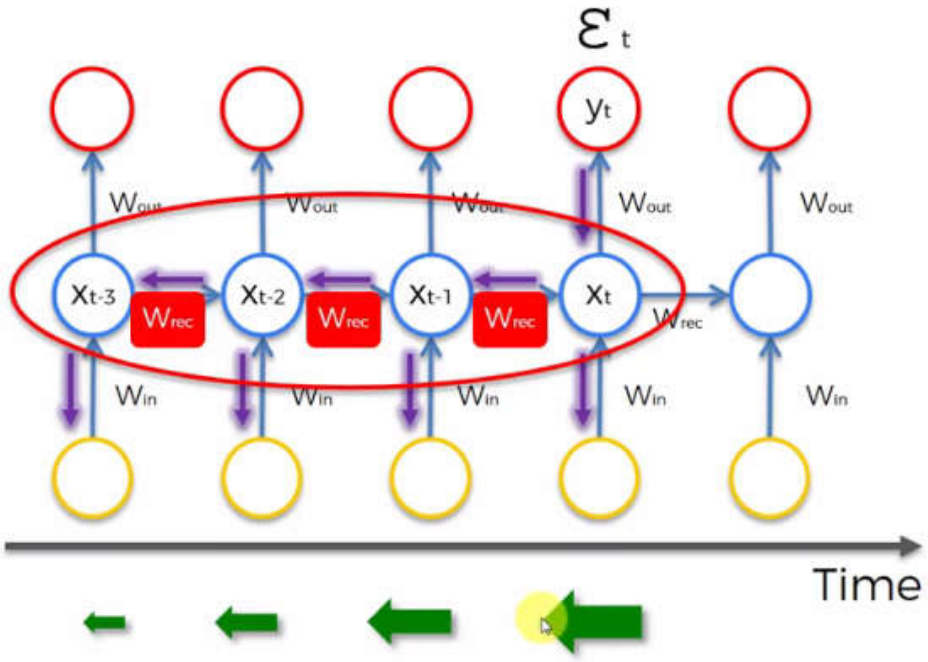
- ➢ In simple word, we are simply multiplying the output by the weight, and then we get to the next layer.
- ➢ Here we're multiplying by the ***same*** exact ***weight*** in ***multiple times***, as many times as we need to go through this temporal loop.

✋ And this is where the problem arises, because when you ***multiply*** by something ***small*** your value decreases very quickly (eg: 0.2*0.2 = 0.04), and from the above formula we can see that.

- ➢ Now remember that at the ***very-start*** of the ***propagation process*** the weights are assigned ***randomly*** to ***NN*** and those ***random-weights*** are close to ***0***. Hence due to ***multiplication*** of such small values, our ***"gradient"*** decreases from ***one layer*** *to* ***previous layer*** during ***Back-Propagation***.

❒ A **vanishing gradient** is **bad** for the **network**. Because when the ***gradient*** as it goes back through the network, it is used to ***update*** the ***weights***, and the ***lower*** the ***gradient*** is the ***harder*** it is for the ***network*** to ***update*** the ***weights***.

➲ The lower the gradient gets, the updating the weights get slower. (the higher the gradient the faster it's going to update the weights).



➲ So for instance say we have 1,000 epochs. Then ***some layers*** and ***part*** of our ***network*** cannot get ***trained*** properly.

➲ Having their ***gradient's*** so much smaller, they're gonna be updated slower. Therefore by the end of the ***1,000 epochs*** you might not have the final results there, and some ***part*** of the ***network*** is ***trained*** and some ***part*** is ***not trained*** (based on their cost function).

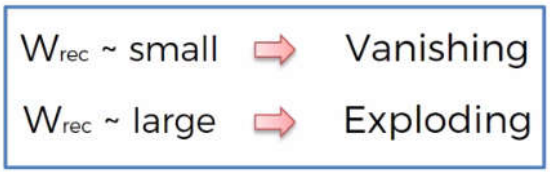
☝ But the problem here is not just that *half of our network* is **not trained properly**, but also that those ***weights***, on those ***untrained layer*** generating ***wrong outputs***. And those ***wrong-outputs*** are being used as ***inputs*** for further ***layers***.

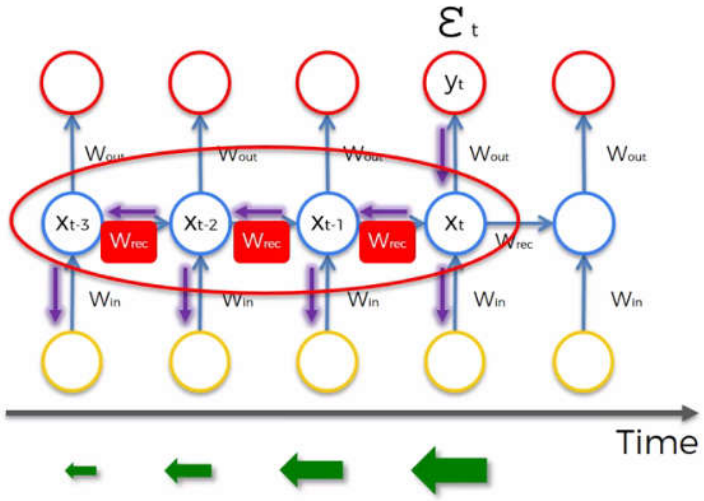
☝ So, the ***training*** here has been ***happening*** all along based on, inputs that are coming from **untrained** ***neurons***, **untrained** ***layers***.

In simple word that's the ***vanishing gradient problem*** for RNN.

12.1.5 Exploding Gradient Problem

Exploding –gradient: If W_{rec} is small, then you have a ***vanishing gradient problem***. If W_{rec} is large you have an ***Exploding Gradient Problem***. Of course there is so much parameters in the formula, activation-functions weights etc but in a nutshell these two thing can happen.





$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \le t \le T} \frac{\partial \mathcal{E}_t}{\partial \theta} \quad (3)$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right) \quad (4)$$

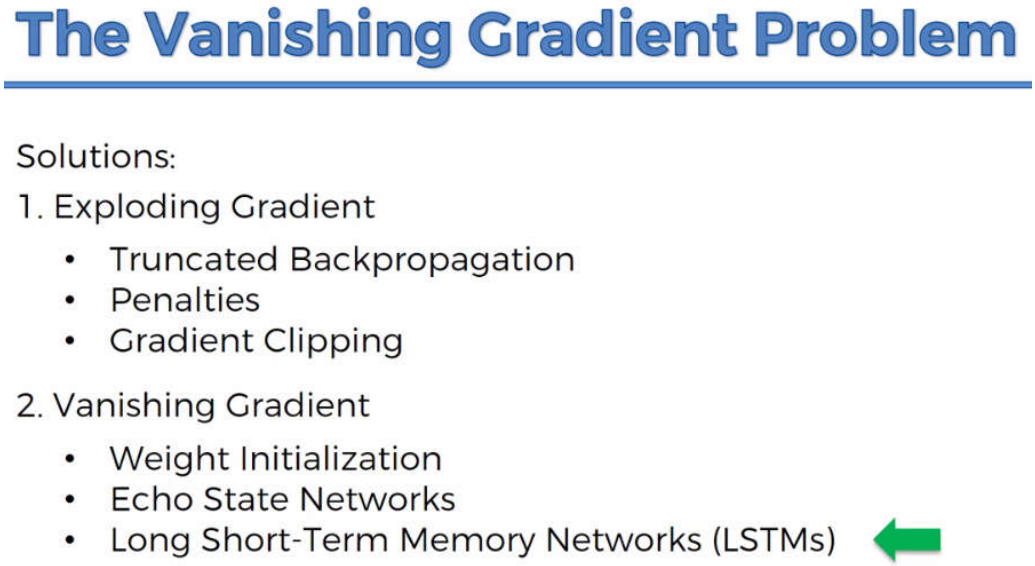
$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \ge i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \ge i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1})) \quad (5)$$

W_{rec} ~ small ⇨ Vanishing

W_{rec} ~ large ⇨ Exploding

Formula Source: Razvan Pascanu et al. (2013)

12.1.6 Solutions for Vanishing/Exploding Gradient Problem



❒ ***For the Exploding Gradient problem***

[1]. ***Truncated Back Propagation:*** For the exploding gradient you can have *Truncated Back Propagation*. So, you stop *back propagating* after a *certain point*, but that's probably not **optimal** because then you're **not** updating **all** the **weights**.

[2]. ***Penalties:*** You can have *Penalties*. The gradient being penalized and being artificially reduced.

[3]. ***Gradient clipping:*** You can have *Gradient Clipping*. So, you could have a *maximum limit* for the *gradient*. *Gradient* never go over this *value*, and if it does, then that value just *stays* at *same* as it *propagates further* down through a network.

❒ ***For the Vanishing Gradient problem***

[1]. ***weight initialization:*** You have *weight initialization*, where you are smart about how you *initialize* your *weights* to *minimize* the *potential* for *vanishing gradient*.

[2]. ***Echo State Networks:*** You can have, other type of network called the *echo state networks*. They are designed to solve the vanishing gradient problem.

[3]. ***Long Short-Term Memory networks, OR THE LSTMS:*** There's also a different type of network called the long short-term memory networks, or the LSTMs (which are extremely popular). We will talk about it in next section.

12.1.7 Additional reading

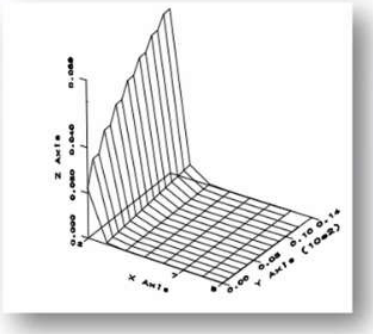
The original works by ***Sepp Hochreiter*** and ***Yoshua Bengio*** are good to read.

❒ So, this is ***Sepp's*** paper in 1991. It's completely in German. If you understand and can read German, then definitely this could be a good read for you.

Additional Reading:

Untersuchungen zu dynamischen neuronalen Netzen

By Sepp (Josef) Hochreiter (1991)



Link:

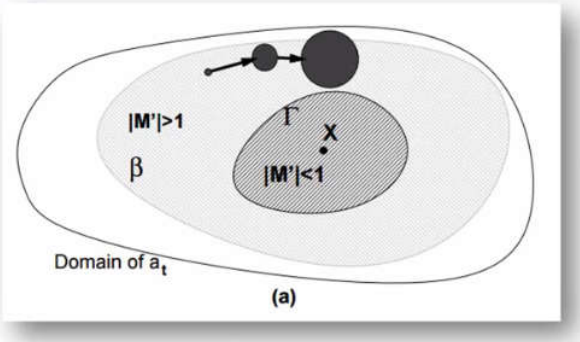
http://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf

❒ Then there's ***Yoshua Bengio's*** paper which is called ***Learning Long Term Dependencies with Gradient Descent is Difficult***, 1994.

Additional Reading:

Learning Long-Term Dependencies with Gradient Descent is Difficult

By Yoshua Bengio et al. (1994)



Link:

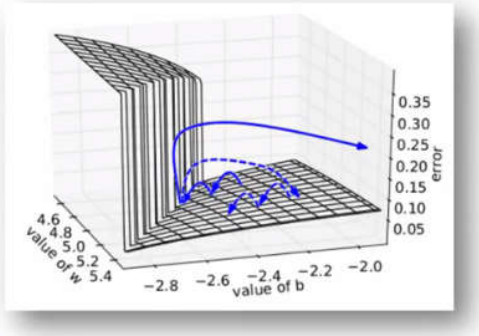
http://www-dsi.ing.unifi.it/~paolo/ps/tnn-94-gradient.pdf

❒ We also recommend looking into this paper called ***On The Difficulty Of Training Recurrent Neural Networks*** by ***Razvan Pascanu***.

Additional Reading:

On the difficulty of training recurrent neural networks

By Razvan Pascanu et al. (2013)



Link:

http://www.jmlr.org/proceedings/papers/v28/pascanu13.pdf

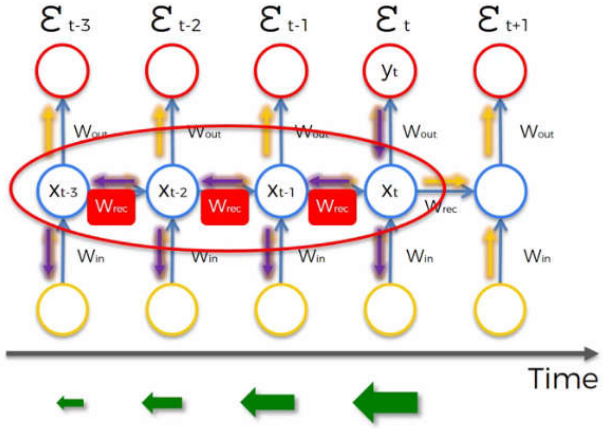
Chapter 12 : Part 2

Deep Learning

RNN: LSTM - Long Short-Term Memory

long short-term memory Introduction

12.2.1 LSTM



$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \le t \le T} \frac{\partial \mathcal{E}_t}{\partial \theta} \quad (3)$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right) \quad (4)$$

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \ge i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \ge i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1})) \quad (5)$$

W_{rec} ~ small ⇨ Vanishing

W_{rec} ~ large ⇨ Exploding

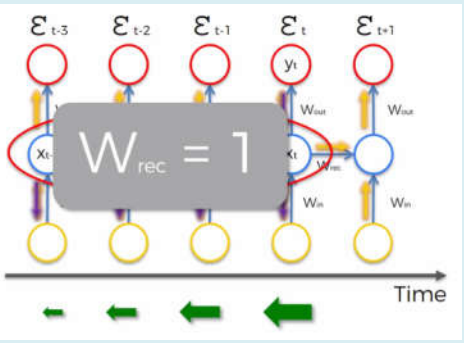
Formula Source: Razvan Pascanu et al. (2013)

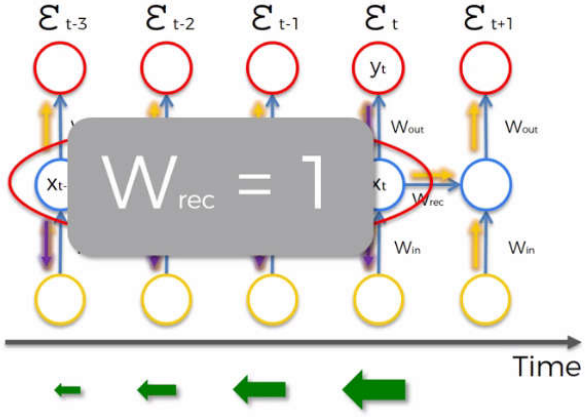
- Here we've got vanishing gradient problem. And as a rule of thumb, we can see here that if $\boldsymbol{W_{rec}}$ is **small**, then we have ***Vanishing Gradient***, if $\boldsymbol{W_{rec}}$ is **large**, then we have ***Exploding Gradient***. To solve this problem there is couple of solutions. But here we only focus on LSTM.

$W_{rec} < 1$ ⇨ Vanishing

$W_{rec} > 1$ ⇨ Exploding

- The first thing that comes to mind is to make $\boldsymbol{W_{rec}}$ equal one, $\boldsymbol{W_{rec}} = 1$, and that's exactly what was done in the ***LSTMs***.





$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \le t \le T} \frac{\partial \mathcal{E}_t}{\partial \theta} \quad (3)$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right) \quad (4)$$

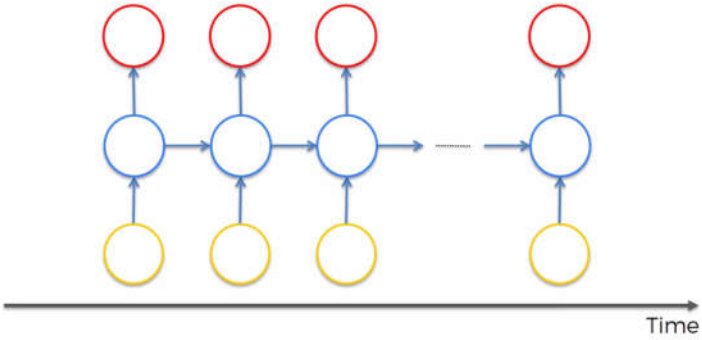
$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \ge i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \ge i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1})) \quad (5)$$

W_{rec} < 1 ⇨ anishing

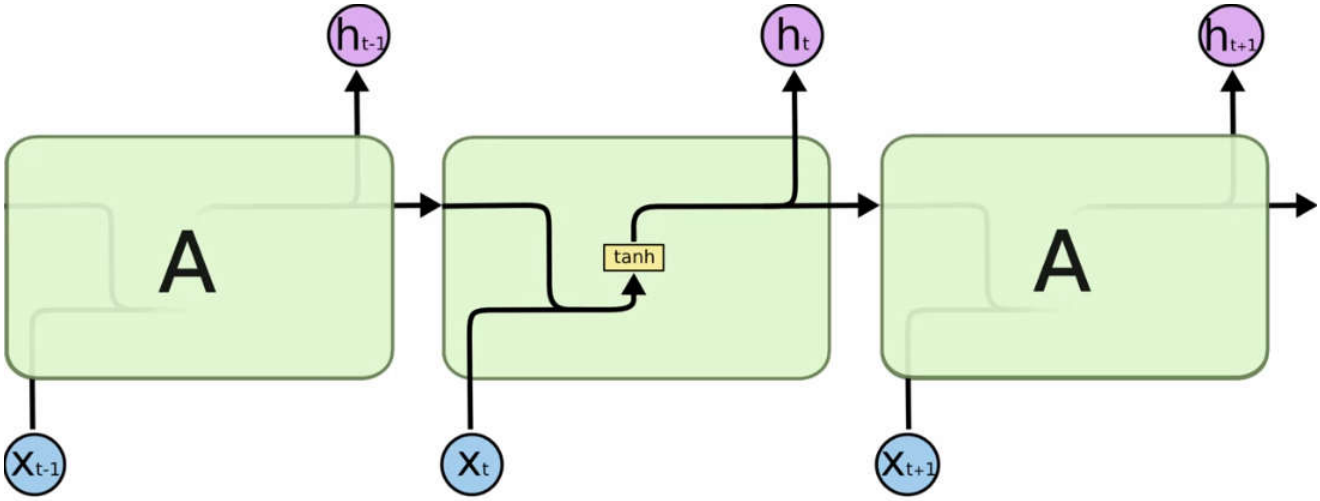
W_{rec} > 1 ⇨ xploding

Formula Source: Razvan Pascanu et al. (2013)

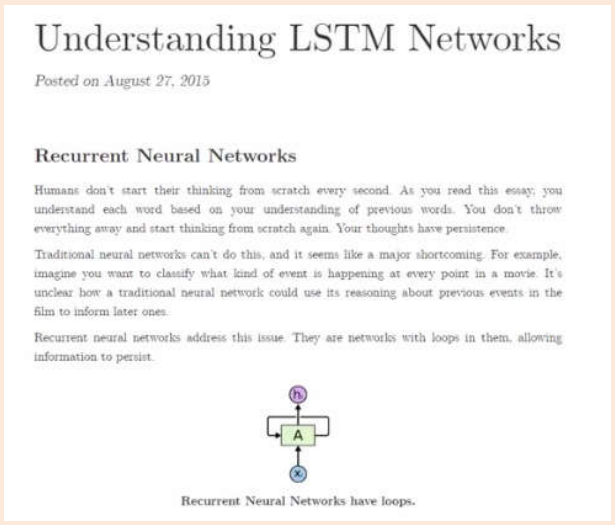
❒ Here we've got a recurrent neural network. With unraveled ***temporal loop***.



❒ This is what it looks like if you dig inside the RNN. The images are taken from Christopher Olah.



❒ Here is his blog Very well-written blog with amazing images.



Understanding LSTM Networks

Posted on August 27, 2015

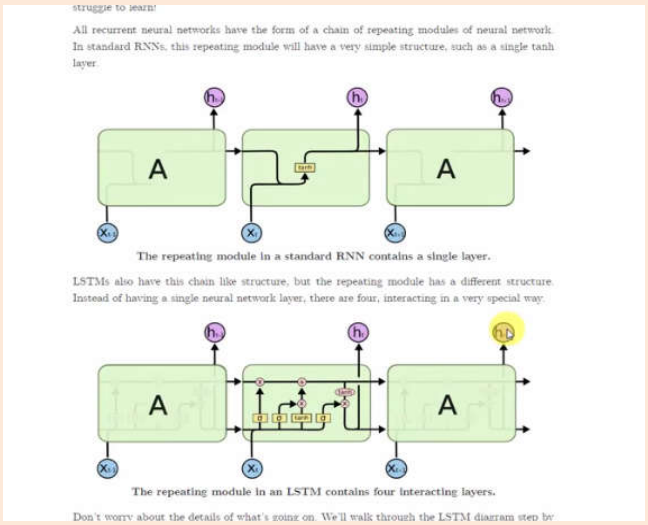
Recurrent Neural Networks

Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.

Recurrent Neural Networks have loops.



struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

The repeating module in a standard RNN contains a single layer.

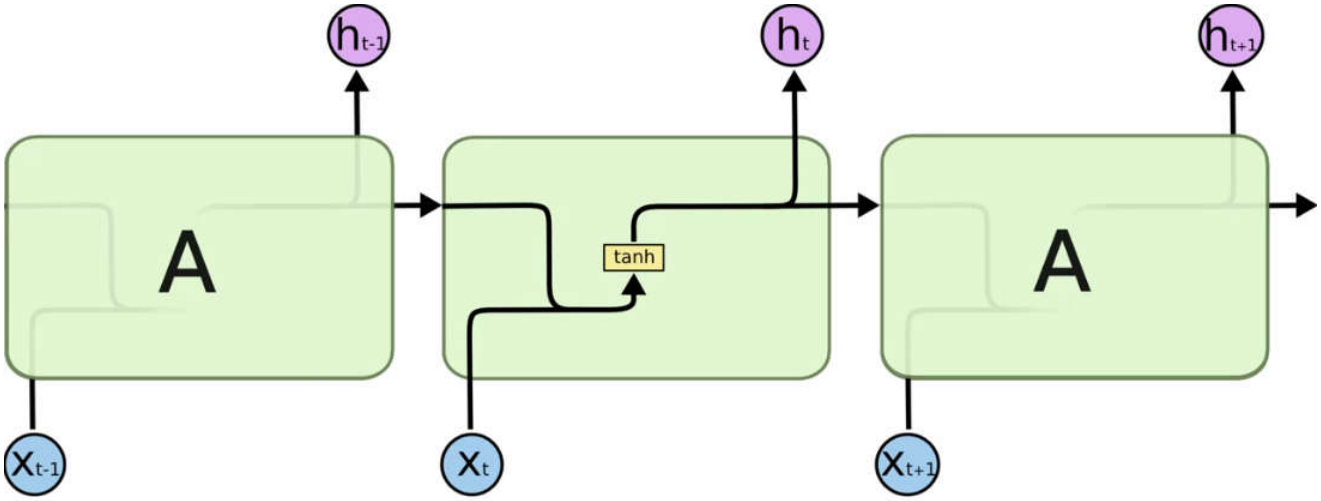
LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

The repeating module in an LSTM contains four interacting layers.

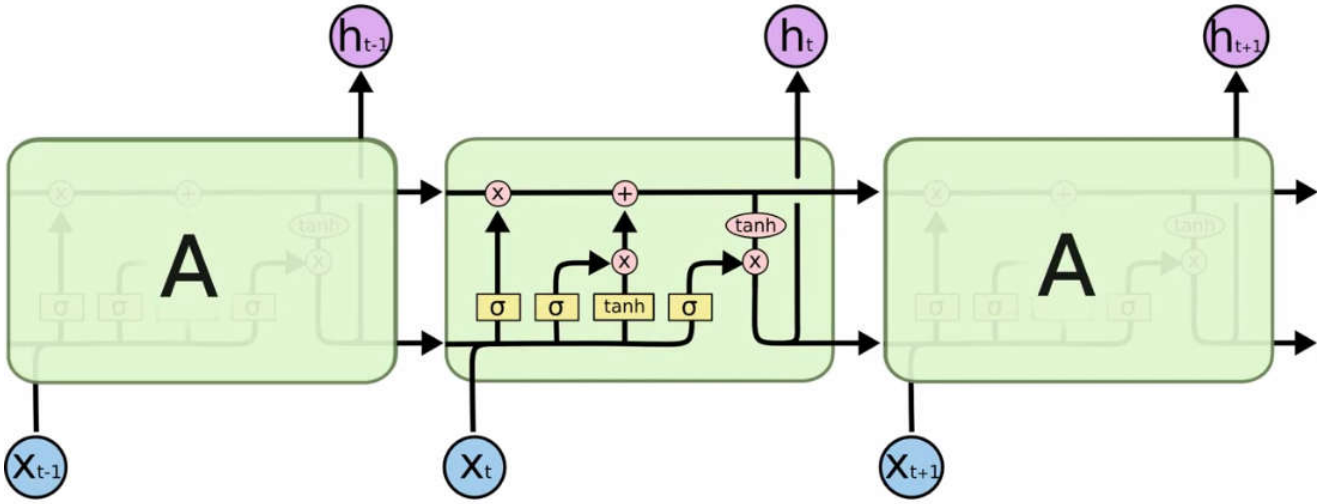
Don't worry about the details of what's going on. We'll walk through the LSTM diagram step by

12.2.2 RNN vs LSTM-RNN

In Following images the first one represents the Simple RNN and 2nd image represents "RNN with LSTM". Here LSTM is a special kind of network applied to RNN.

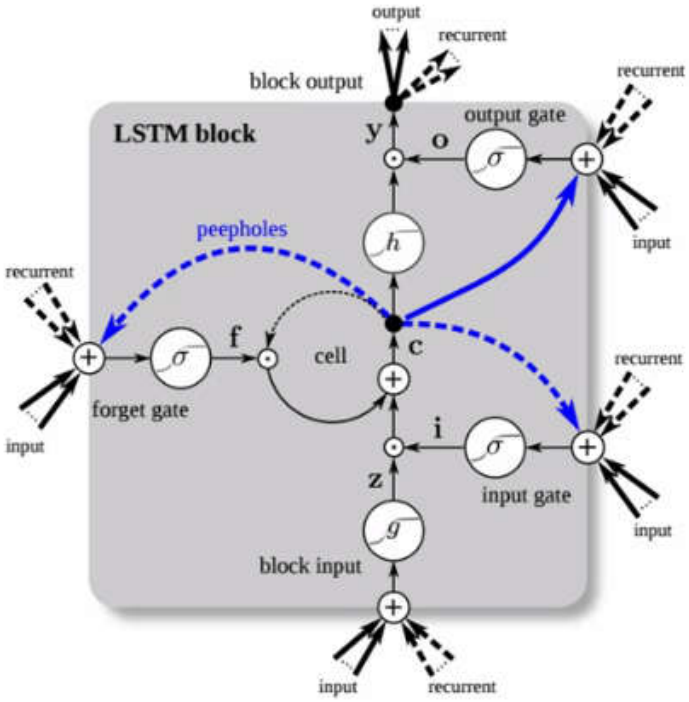


Standard RNN



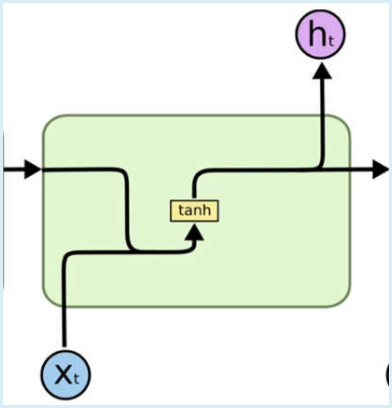
RNN with LSTM applied

❒ Above is a simple representation of LSTM. Following is the detailed representation. Image Source: ***arxiv.org/pdf/1503.04069.pdf***



12.2.3 How LSTM works

❒ So here we've got the ***inside-look*** of our ***simple-Standard-RNN***. And this is where the problem lies.

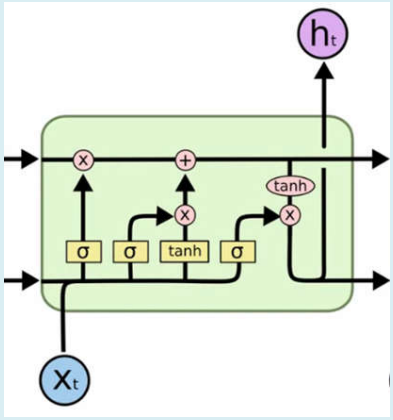


☞ This operation that happens here is actually a ***neural-network-layer-operation***.

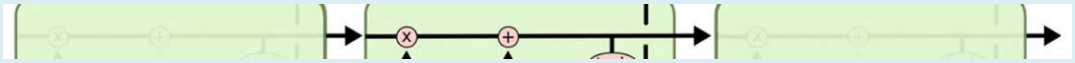
In a simple word: outputs ***coming*** into this ***module*** and this operation's applied and then goes into the ***next module*** operation's applied, and so on.

☞ When we ***back propagate***, it goes through all of those ***operation***, and that's where the ***weights*** are applied (that's where the W_{rec} is sitting). And through this ***back propagation***, the ***gradient vanishes***, which means that the ***weights cannot*** be ***updated*** properly or fast enough to ***train*** the network ***properly***. This is the ***standard RNN***,

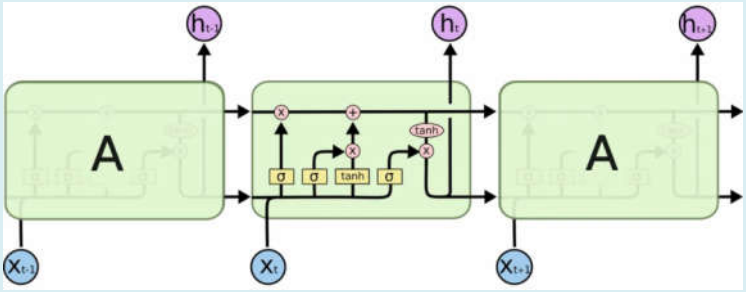
❒ Now Following image is the ***LSTM*** version of ***RNN***.



☞ Notice that the main point here was setting $W_{rec} = 1$. Well that's this line that ***flows straight*** here (the upper line), that ***pipeline*** at the top of the ***LSTM*** version of ***RNN***.



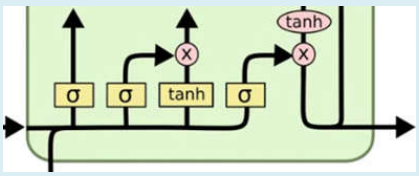
☞ There's not much going on, just two ***pointwise operations*** (removal & addition) and ***no Complex Neural Network Layered Operations*** happening in this line.



☞ Actually ***LSTMs*** have this pipeline as a ***memory cell*** or we can call it ***memory pipeline***. This ***pipeline*** flows through ***time***, sometimes it faces ***pointwise operations*** to remove/add something.

☞ But mostly it ***flows through times*** freely and therefore when you ***back propagate*** through these LSTMs, you don't face ***vanishing gradient*** problem.

❒ However, all complex-operations are brought out to this down part (image on the Right).



12.2.4 LSTM operations

Now let's focus on just one time-step or a single module/block of LSTM. In following representation we have:

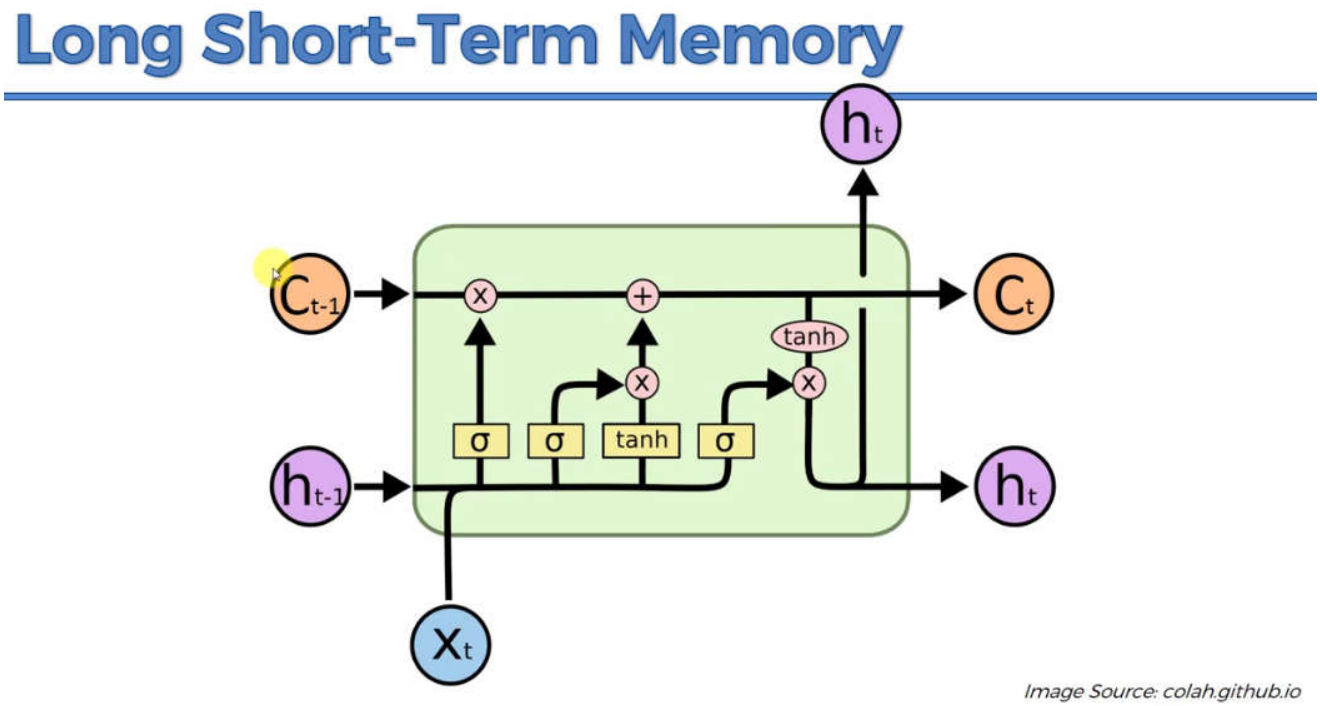


Image Source: colah.github.io

❒ ***The VARIABLES in this Diagram:***

⇨ C_t stands for ***memory*** or ***memory-cell***.

- C_t represents the *memory-cell* for ***current*** LSTM-module/block.
- And C_{t-1} is the *memory-cell* coming from ***previous*** LSTM- block.

⇨ h_t is ***output***. Here we can see there is two h_t and one h_{t-1}

- One h_t (goes up) goes out into the world, and
- another h_t (on the Right) goes to the next LSTM-module/ block.
- h_{t-1} represents the ***output*** from the ***previous*** module.

⇨ X_t represents the input for ***current*** LSTM-block

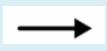
☝ So an LSTM a module takes in ***three inputs***: input data X_t, previous modules output-data h_{t-1}, memory-cell C_{t-1}.

☝ And it produces ***two outputs***: current modules memory-cell C_t and output-data h_t, (one copy of h_t goes to next LSTM-module).

✋ ***All variables are vectors:*** The important thing is that everything here is a vector. I.e. C_t , h_t, and X_t, are all ***vectors***.

☝ Each of them *doesn't represent single value*, they are actually ***vectors*** containing ***multiple values***.

❒ ***The LEGENDS in this Diagram:*** And let's go through the legend.

	***Vector transfers:*** Any *line* here (in this drawn LSTM-module) is a *vector* being *transferred*.
	***Concatenation:*** Anywhere in this drawn LSTM-module, if see that there's two lines ***combining*** into one. Actually they don't become one single line. But these two lines are *running* in *parallel*.  ☞ You're *not actually combining*, ***concatenation*** means that you're combing these two lines *on top of each other*. Basically you have two pipes running in parallel feeding into these neural network layer operations (simultaneously).



Copy: In this drawn LSTM-module, if see that there's one line ***splitting*** into two (or more).

 or  or 

☞ It means that the memories go straight ahead and just copy it when the line splits.



Pointwise operations: We've got a couple of *pointwise operations* here (5 *pointwise operations* are used in this diagram).



Valves: The X's are valves and they all have names. There are three kind of valves.



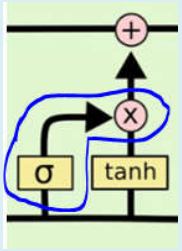
Forget valve, ***F***.
Memory valve, ***V***.
Output valve, ***O***.

Forget valve is basically controlled by the *layer operation* σ. σ represents *sigmoid function* (decide between ***0*** and ***1***). Based on the decision made by the *layer operation* σ foregut valve ***F*** will close/open.

If it's open, *memory* flows through *freely* through *memory-pipeline*.

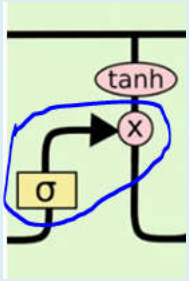
If it's closed then *memory* is *cut off.* Therefore it's not transferred further and then **new memory** just will be added in next $\oplus$ (in the joint) based on the *memory valve* opened/closed.

Memory valve, which is also controlled by *layer operation* σ. σ represents *sigmoid activation function*. The value of $tanh$ *- layer operation*, is added (or somewhat added) or not-added to the *memory-pipeline* based on the decision of the memory-valve ***V***.



Why we're using ***Sigmoid Activation Function*** is because they are from ***0*** to ***1***. ***0*** stands for ***close***, ***1*** stands for ***open***.

Output valve, also controlled by *layer operation* σ and it decides whether the output X_t can go or not.





T-shaped joint: And then we've got *T-shaped joint*, where *connects* the memory coming form *memory-valve* to the *memory-pipeline*. Here additional memories got added if the *memory-valve* is opened.



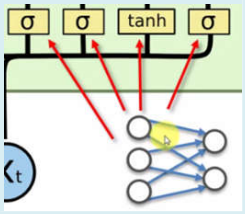


Tangent operation: Tangent operation, works with values between **minus one** and **one**. It's another *pointwise* operation (*layer operation*).



The neural network layer operations: Above we talked about *pointwise operations*. (Pointwise is like ***element by element*** over vector, if you wanna multiply a vector by zero, you multiply every element by zero. Or multiply a vector by certain amount is multiply each element of the vector with that amount).

☞ But the things going on inside those ***NN-layer operations*** are bit more complex than *pointwise operations*.



☞ Here on those ***NN-layer operations*** we've got a ***NN-layer*** coming in and then we've got a ***NN-layer*** coming out.

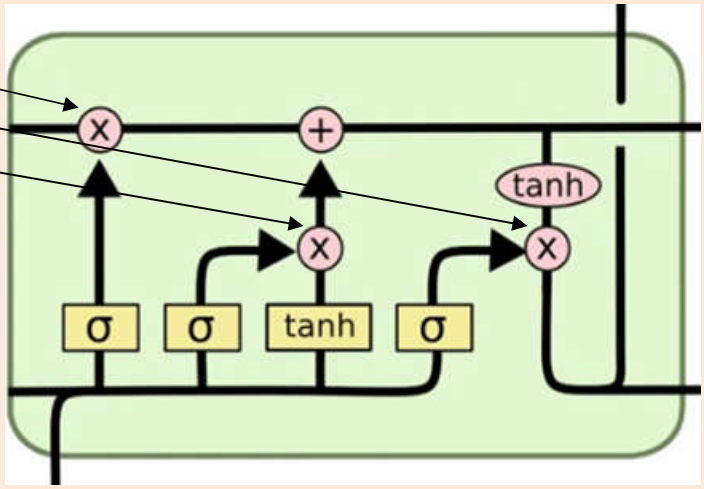
☞ Everything here is a vector. So we've got a ***Layer of these Sigmoids*** (σ or $tanh$) which ***controlling*** the ***valve*** for each one of these ***elements*** in the ***vector*** of ***memory***.

So it's ***not*** just ***one Sigmoid***. Its a whole ***layer of Sigmoids*** coming in and then we've got a layer coming out.

Forget valve, ***F***.

Output valve, ***O***.

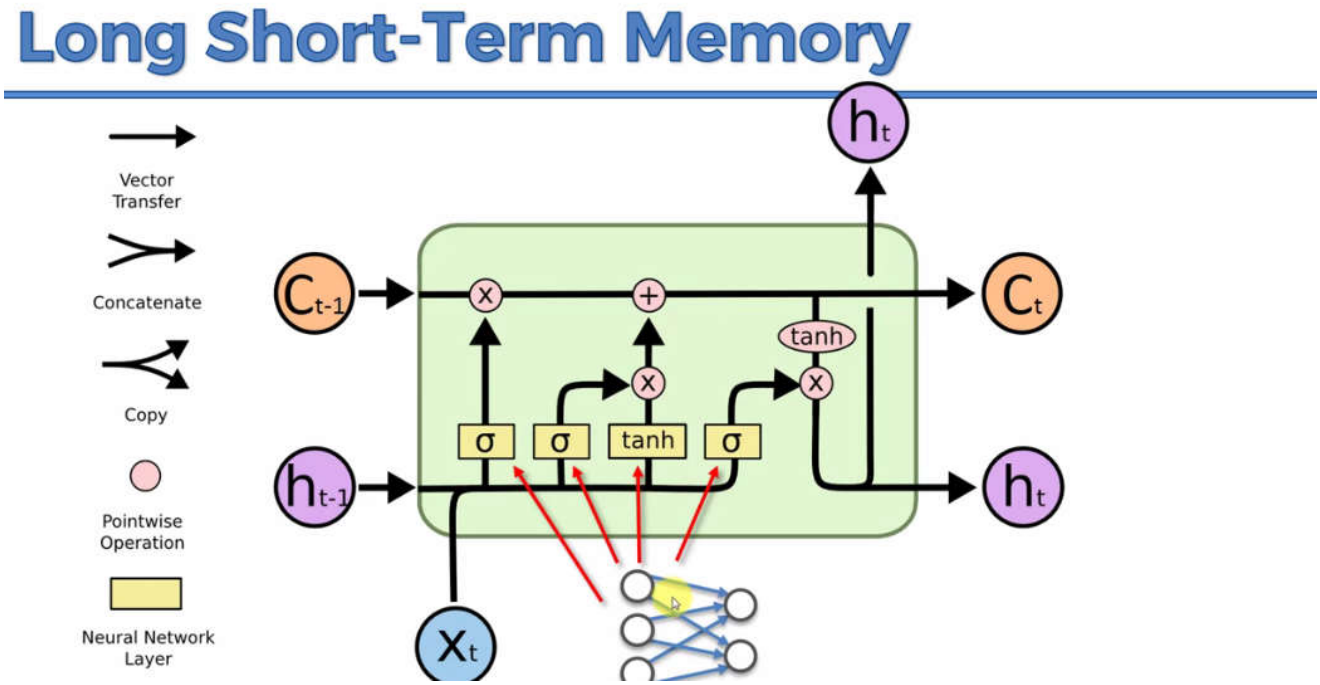
Memory valve, ***V***.



☞ In literature you will see letters ***F***, ***V***, and ***O*** in the actual formulas representing these valves.



☞ But we imagine it like a real valve (in plumbing). Where we've got *water* or basically *something* **flowing through**, and then you can either ***close*** it or you can ***open*** it, or you can ***close it to some extent***.

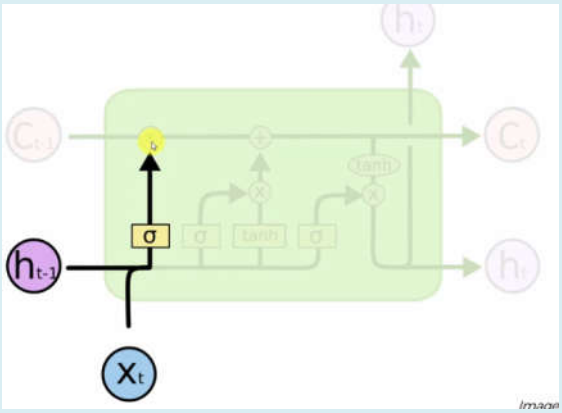


12.2.5 LSTM steps

So we're ready to look into above LSTM diagram in step by step.

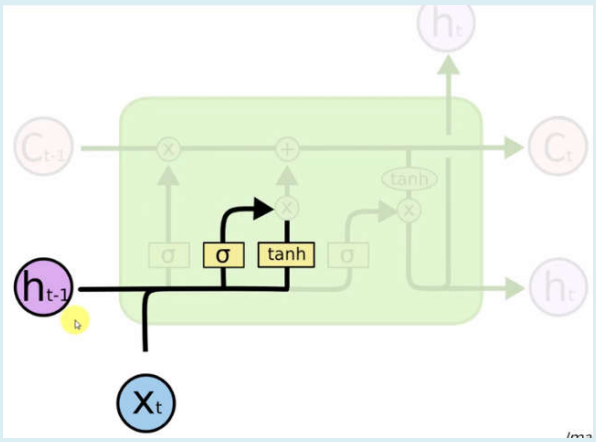
Step 1: Here two values enter to the LSTM-block. One is *input-value* X_t and other is the *output-value* h_{t-1} coming from a *previous LSTM-block*.

They are *combined*, *layer-operation* σ decides whether this value should go ahead or not (i.e. *forget-valve* should be *closed* or *open* or *somewhat closed* or *open*)



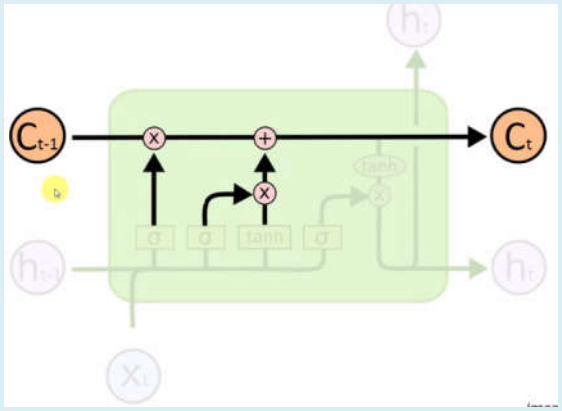
Somewhat means to ***some degreee***, but not to a ***large degree***.

Step 2: In 2nd step we've got X_t and h_{t-1} again they first flow parallel, and combined in σ or $tanh$ operation. Basically X_t and h_{t-1} are *layers of neurons*, σ or $tanh$ decides which values *can* pass and which *cannot* or *somewhat* pass.

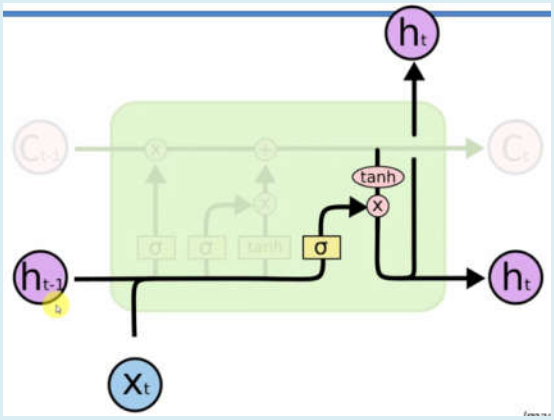


Step 3: Then we've got the ***memory flowing*** through *memory pipeline*. We've got the *forget valve*, *joint-operation* and *memory valve*. , closed.

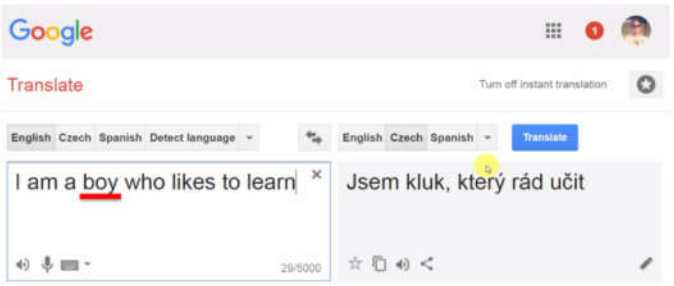
- If *forget valve* is opened ***memory*** can *flow* and we're adding in some memory if *memory-valve* is *open*.
- Or we can let this whole memory flow through, then keep *memory-valve* closed, keep *forget valve* open, the memory won't change.
- Or we can keep *forget valve* closed and keep *memory-valve* open, then we can update the memory completely.

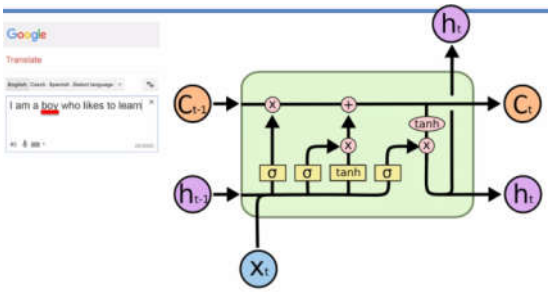


Step 4: Finally we've got these two values X_t and h_{t-1} combined and σ decide what part of the *memory pipeline* is going to become *output* of this *LSTM-module* : is it going to be *full-output* or *partial-output*.



Example: ***Google Translate*** example for ***English*** to ***Czech.***





Now for example, we can consider our ***Google Translate*** example. Here we are translating ***English*** to ***Czech***. Remember in ***Czech*** gender matters. So changing ***Boy*** into ***Girl*** in ***Czech*** we need to change gender related word. Here ***RNN*** works as follows:

- First if it is ***Boy*** then ***Memory*** flows ***freely*** in the ***memory pipeline***.
- If it is ***Girl*** (or female name like "Amanda") ***Sigmoid-layer-function*** can detect that and ***Forget-valve*** is closed. Old memory ***cannot pass*** through the memory ***pipeline***. Then new memory added via ***Memory-valve***.
  - In addition we have ***extracted*** gender and other data (***capitalized***, ***singular/plural***, no. of letters etc.).
- ***Output-valve*** **extract** the information. For example ***Gender*** is extracted and send this output as an information to the ***next LSTM-module***, so that **next** module can act according to this **Gender** (change the gender related word).

12.2.6 Additional Readings

- In terms of additional reading, you could definitely reference the original paper by our two authors who created LSTMs.

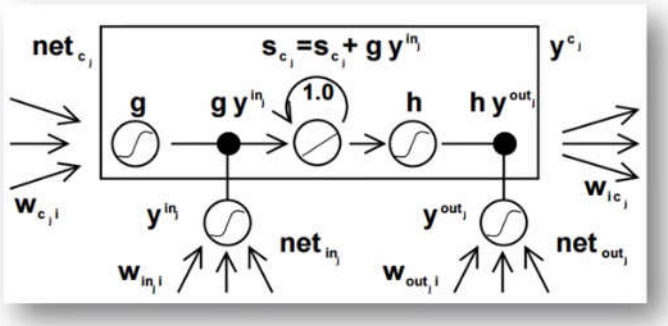
Additional Reading:

Long Short-Term Memory

By Sepp Hochreiter &
Jurgen Schmidhuber (1997)

Link:

http://www.bioinf.jku.at/publications/older/2604.pdf



- If you ***don't wanna get*** that ***deep*** into ***mathematics*** and into the technical stuff there's the great blog by ***Christopher Olah***,

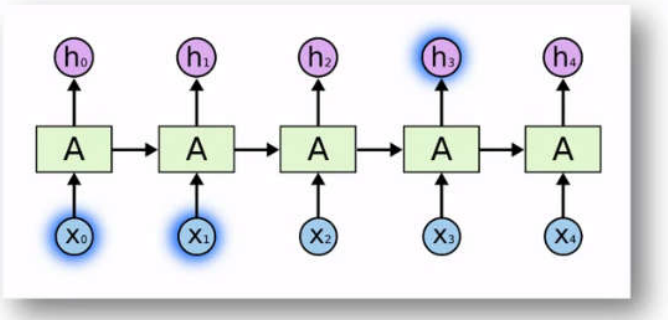
Additional Reading:

Understanding LSTM Networks

By Christopher Olah (2015)

Link:

http://colah.github.io/posts/2015-08-Understanding-LSTMs/



❒ And there's another blog by ***Shi Yan***. ***Understanding LSTM and its diagrams***. Those **Diagrams** are a bit more **in-depth**, so there's a bit less space saving, but **diagrams** might be **easier** to **understand** in some cases. No mathematics whatsoever, just plain intuition. So also highly recommend this blog.

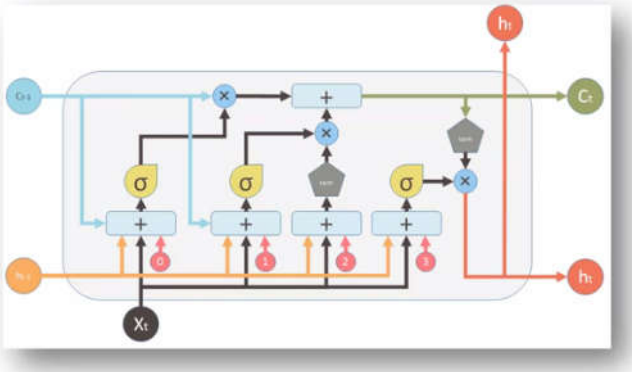
Additional Reading:

Understanding LSTM and its diagrams

By Shi Yan (2016)

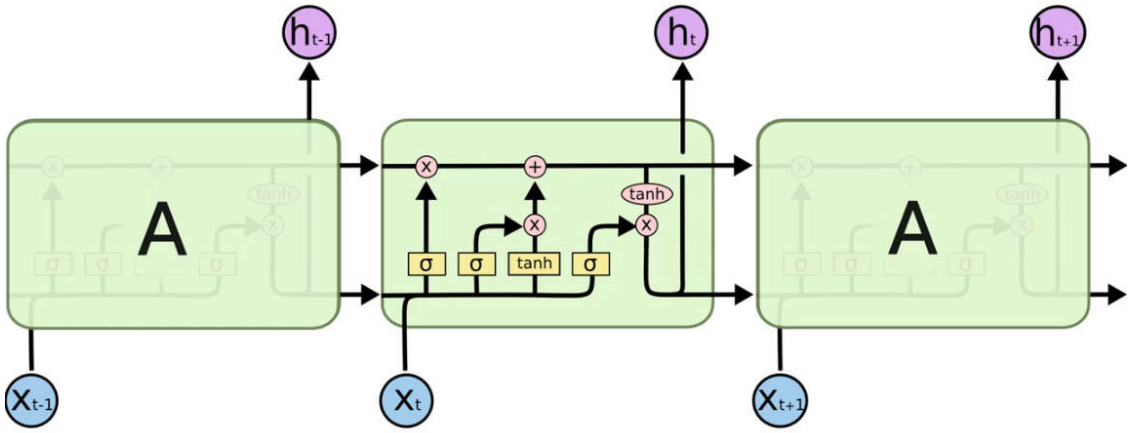
Link:

https://medium.com/@shiyan/understanding-lstm-and-its-diagrams-37e2f46f1714



12.2.7 LSTM in action - Examples

Today we're going to look at some practical applications where LSTM in action. We're gonna look at how LSTMs work inside those applications. Here we've got our LSTM architecture.

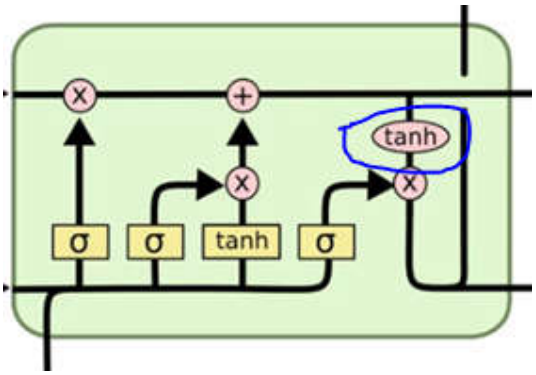


❒ At first we're going to look to the ***poinwise-tangent-function*** and how it ***fires up***.

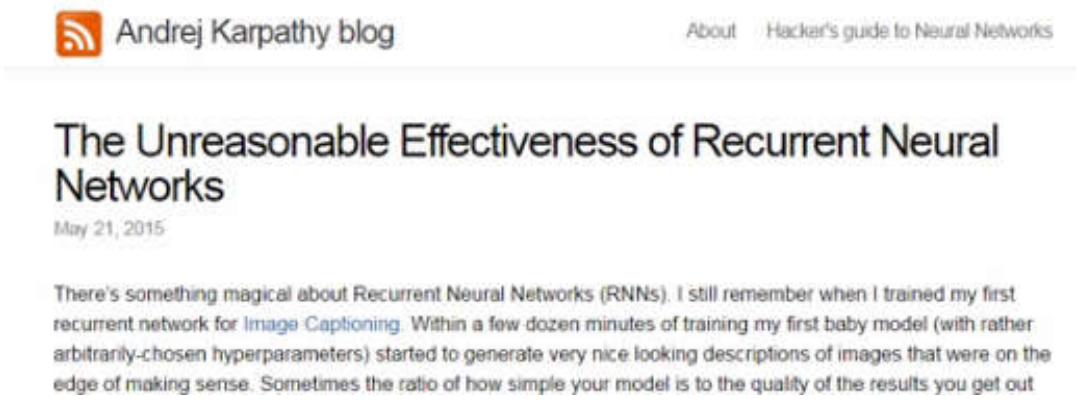
☞ The images that we're going to look are all from ***Andrej Karpathy's*** blog. This blog is called The ***"Unreasonable Effectiveness of Recurrent Neural Networks"***. And the paper that ***Andrej*** published along with that will be linked at end.

☞ The ***poinwise-tangent-function*** that we are looking at works in the ***range [-1, 1]***.

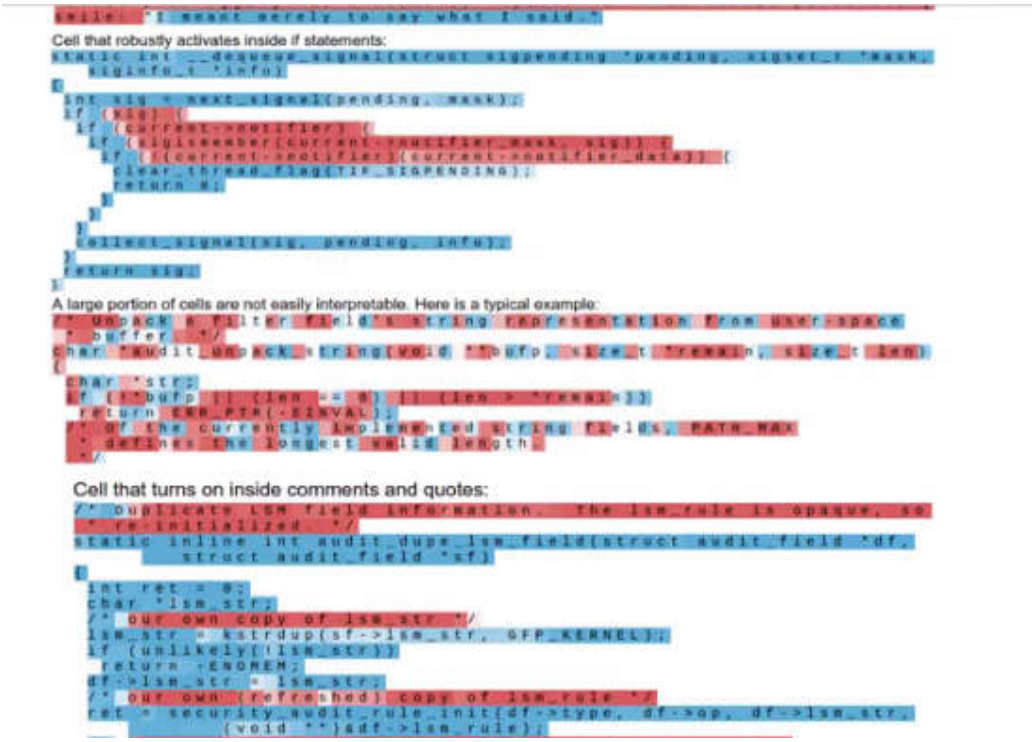
☞ According to the paper, the color for ***-1*** gonna be ***red***, ***+1*** is gonna be ***blue***.



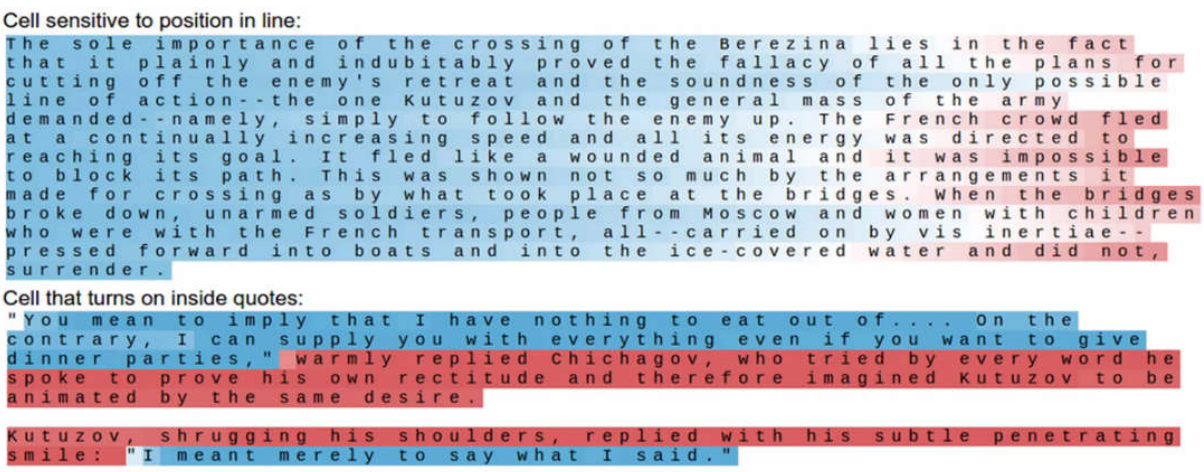
❐ ***Andrej Karpathy's blog:*** Unreasonable Effectiveness of Recurrent Neural Networks



☞ We are going to look following kind of image where the color codes are done by RNN.



☞ ***Example 1:*** ***Example of Trained RNN on text:*** Here's some text, which is given to an RNN, which ***learned*** to ***read text*** and kind of ***create text*** and ***predict*** what ***text*** is coming ***next***. And here, this is a snippet from the ***War and Peace***.



☞ Here two neurons are acting,

i. **Cell sensitive** to **position** of **line**
ii. **Cell that** turns on inside quotes

☯ ***Cell sensitive to position of line (detecting the end of the line):*** We can see that when we get ***towards*** the ***end*** of the ***line*** it's activating. How does it know when it's the end of the line?

- Here in this text we have about 80 symbols per line approximately, so the ***neuron*** is ***counting*** how many ***symbols*** have ***passed*** and it's trying to ***predict*** when the ***new line character*** (is an invisible character. i.e. "\n") is going to appear.

☯ ***Cell that turns on inside quotes:*** Then you've got a cell that turn on inside quotes. It detects the texts that are inside a quotation mark.

- But from the color code (red = turn on), we can see the cell is activating outside the quotes. This cell may acting wrong but one way or another, it's ***activating*** either ***inside*** the ***quotes*** or ***outside*** the ***quotes.*** It is keeping track of what's happening.
- So very similar to what we discusses previously, where we were ***keeping track*** of the ***subject***.
- The subject could be ***gender*** (male or female) or understand things like if it's a ***singular*** or ***plural*** so that that would effect our ***verbs*** in our ***translation***.
- Same thing is happening here, the ***cell*** is detecting ***quotes***, if you're ***inside*** or ***outside quotes*** because that effects the ***rest*** of the ***text***. So if the ***cell*** find a ***quotation mark*** then its expecting other ***quotation mark*** for the ***end*** of the ***quote***.

Example 2: Example of Trained RNN on Programming language Source code. Here we got some ***code*** of the ***Linux OS***.

Cell that robustly activates inside if statements:

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
        siginfo_t *info)
{
  int sig = next_signal(pending, mask);
  if (sig) {
    if (current->notifier) {
      if (sigismember(current->notifier_mask, sig)) {
        if (!(current->notifier)(current->notifier_data)) {
          clear_thread_flag(TIF_SIGPENDING);
          return 0;
        }
      }
    }
    collect_signal(sig, pending, info);
  }
  return sig;
}
```

Cell that is sensitive to the depth of an expression:

```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
  int i;
  if (classes[class]) {
    for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
      if (mask[i] & classes[class][i])
        return 0;
  }
  return 1;
}
```

Image Source: karpathy.github.io

☯ ***Cell activates inside the if statements (detects the if statemnet):***

- We can see that a cell activates inside if statements.
- If we look closely then we can see that the cell is actually active on the ***if*** statements ***conditions***. It detects the ***curly-braces*** "***{}***" and ***normal braces*** "***()***".
- That's how it detects the ***if*** statements ***appearance***.

☯ ***Cell that is sensitive to the depth of an expression:*** In the above image we can also see that the cell is sensitive to how ***deep*** you are inside of a ***nested expression***.

- As we go deeper and the expression gets more and more nested, this cell keeps track of that (notice the ***red-color gets deeper*** in the p of nested if statement).
- So here the ***cell*** is ***using*** it's ***memory*** to keep track of that.
- It's doing that by tracking the **curly-brace** "**{}**" and **indent**.

NOTE:

It's very important to remember that none of this is actually hard coded into the neural network. It's doing those things on its own. All of this is learned by network itself.

- Through **thousands of iterations** and **Epochs**, using many **hidden states** (or the ***actual memory cells***, and it assigns them to keep ***track*** of certain ***things*** based on what it thinks is ***important***).. It assigns ***different hidden states*** of it to keep ***track*** of ***different things/subject***.

So it's really evolving on its own and deciding what's important and what's not,

Example 3: Example of Trained RNN on some text. But it is unrecognizable to human. It's portion of some source-code of a program.

- ***A large portion of cells are not easily interpretable***. Here is a typical example of a cell that *we can't really understand what it's doing*. And according to ***Andrej Karpathy***, about ***95%*** of the ***cells*** are ***like this***.

A large portion of cells are not easily interpretable. Here is a typical example:

```
/* Unpack a filter field's string representation from user-space
 * buffer. */
char *audit_unpack_string(void **bufp, size_t *remain, size_t len)
{
  char *str;
  if (!*bufp || (len == 0) || (len > *remain))
    return ERR_PTR(-EINVAL);
  /* Of the currently implemented string fields, PATH_MAX
   * defines the longest valid length.
   */
```

- The cell is doing something here, but it's just not recognizable to us what is happening there.
- It's much more like the ***example*** of ***CNNs*** of the ***previous chapter***, where after applying the ***filters*** or ***feature-detectors*** if we're looking out for the ***detected-features***, in the ***`Pooled-layer`***, ***`Flattened-Layer`***, ***CNN*** can recognize it but ***human*** cannot.
- Same thing is happening here, by the time the ***processed feature*** get to the ***last layer***, they're completely unrecognizable to the human eye. But they make sense to the machines. Most the time, 95% of the time, you can't really tell what's going on.

But those 5% of the time, those were the ***Example 1*** and ***Example 2*** that we looked at.

Example 4: Now we are going to look another output (We are now analyze tyhe output h_t in our ***LSTM-RNN*** after it's going through the ***pointwise-tangent-operation***, passed through the ***output-valve*** or ***output-gate*** and now were gonna be looking at what's being produced over there in "***output*** h_t ".)

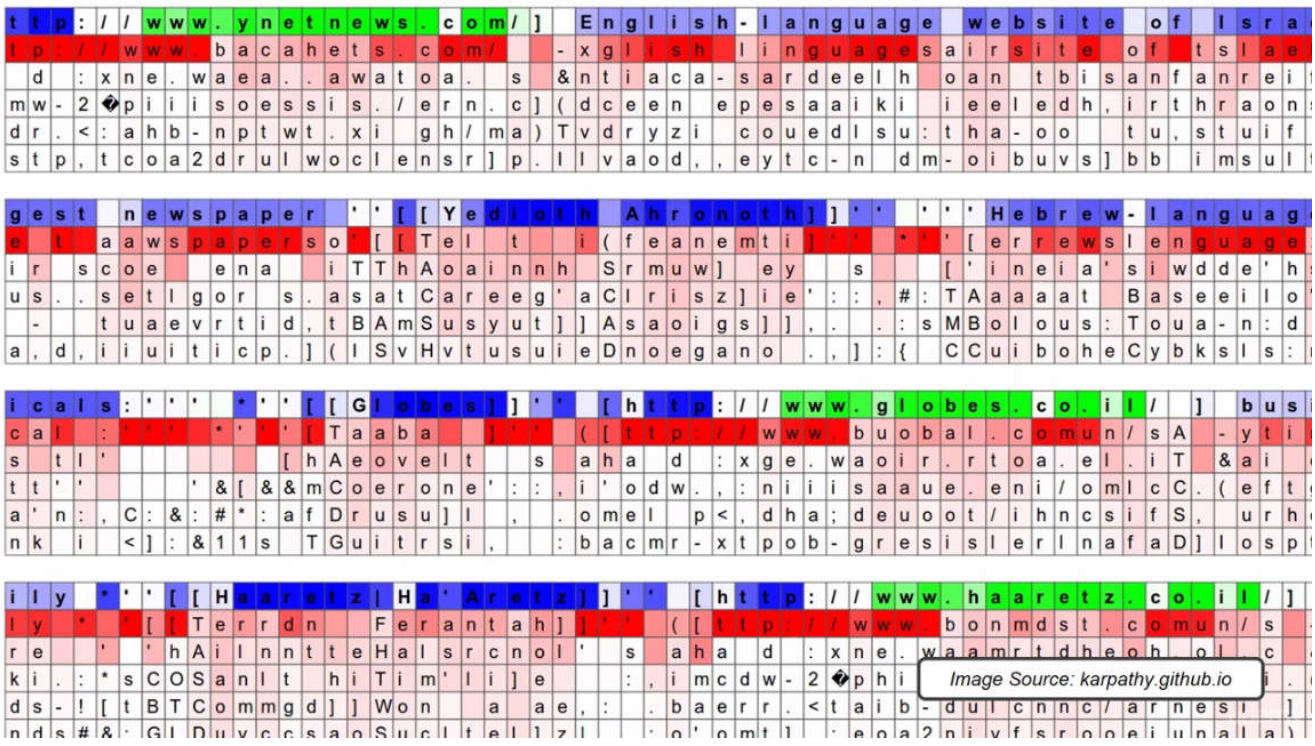


Image Source: karpathy.github.io

☯ We're going to look at the actual output. Above image is another example from ***Andrej Karpathy's*** blog. Here, it's not just showing us if it's active or not but also we can see the guessed letters.

☞ There are ***Six rows***.

- ➢ At the top row, its showing if it's active or not. ***Green*** means active and ***Blue*** means ***not active***.
- ➢ In the other five lines it is saying what's the neural network is predicting, what letter is going to appear in next. Notice the 2nd row letters some letters matched diagonally-left (next in 1st row) to some letters of 1st-row. More ***deep-red*** means more possibility. ***Dark-red*** means a very ***likely prediction***, and ***light red*** means ***unlikely prediction***.
- ➢ We can also see the NN also detecting the ***non-active blue*** text. For example: "**`English-language`**".



☞ If we look closely we can see that this network is looking for ***URLs***. That's why all ***URLs*** are ***green*** here. Following image shows how the NN is guessing the next appearing letters.





- ➢ Some most-probable-predictions (deep-red) also gets wrong. Eg: "**`.com`**" for "**`.co.il`**".
- ➢ Most of the time it can predict "www" successfully.

12.2.8 Additional Readings

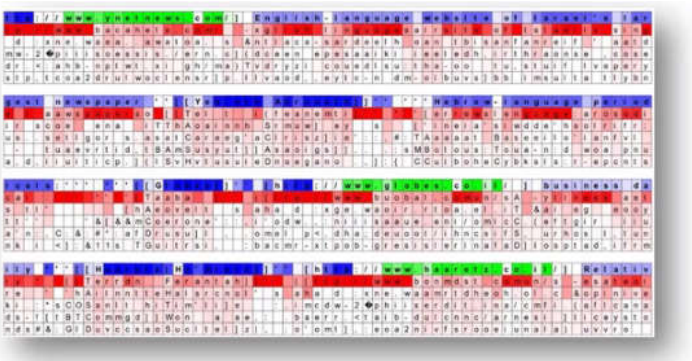
❒ For more, check out his blog, ***karpathy.github.io***. There's a couple more of these examples. And more of the previous examples that we looked at.

☞ Actually it is important to know what's going on inside the neural network. Because RNN, CNN, ANN those are so advanced (and complex) that we need to analyze what's going on inside those architectures (we need to study them and treat them as an alien-being !!!!).

Additional Reading:

The Unreasonable Effectiveness of Recurrent Neural Networks

By Andrej Karpathy (2015)



Link:

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

❒ Also, we've got ***Andrej Karpathy*** and others research paper. Which was published in ***2015***. It's called ***Visualizing and Understanding Recurrent Networks***. There's not too much math.

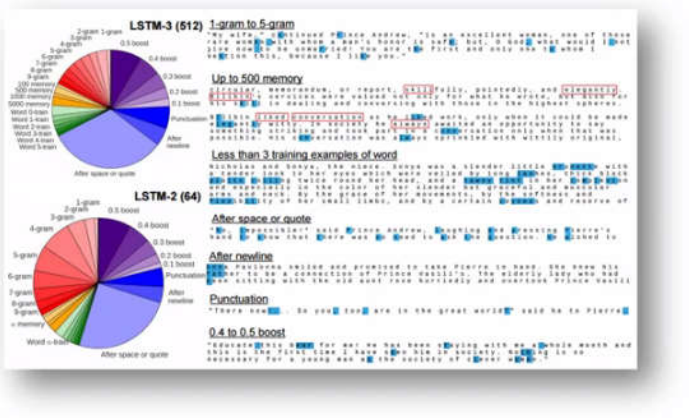
Additional Reading:

Visualizing and Understanding Recurrent Networks

By Andrej Karpathy et al. (2015)

Link:

https://arxiv.org/pdf/1506.02078.pdf



☞ In the paper, they're like **neuroscientists** trying to understand what's going on. So they open up the **brain** of the **neural network** and **monitor** what's happening in one **specific neuron**, or different **neurons**.

☞ As if they're exploring some **alien**, as if they're exploring some kind of **extra-terrestrial being** and how it thinks.

☞ We know that humans created these ***LSTMs*** and ***RNNs***, these are just things that work on our computers. But because they are so ***advanced*** and involve so many ***different elements*** to them, they became so ***complex***, we now need to study them as if they're ***separate beings*** that exists on its own.

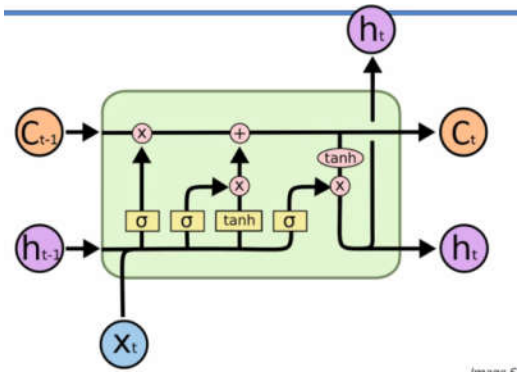
In a few more years or maybe a decade from now, these things are going to be able to think completely on their own.

12.2.9 Different Versions of LSTMs

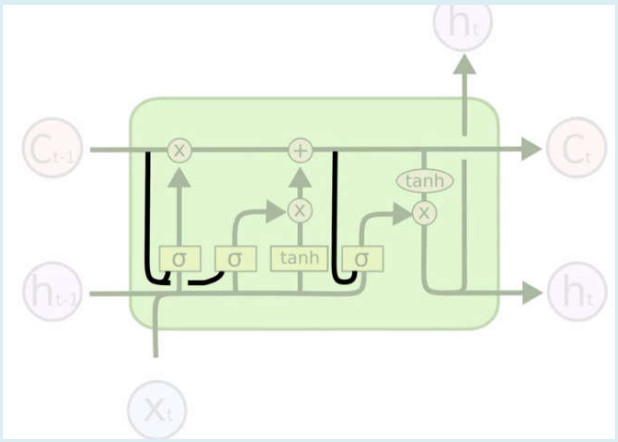
Here we have studied the ***Standard version*** of ***LSTM***. But there could be different version. Here we are going to quickly cover off the variations of long short-term memory architectures.

Standard LSTM.

So here is the **standard LSTM** which we've discussed.



Variations of LSTM



❒ ***Variation no. 1:*** When you add ***Peepholes***.

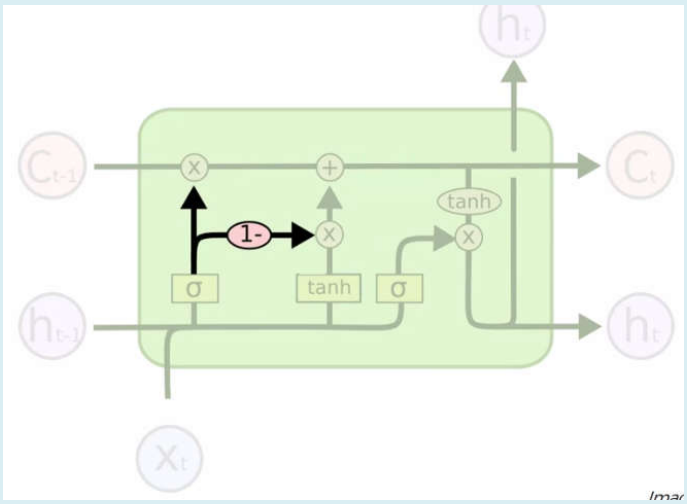
☞ Notice these lines are added, connecting these sigmoid activation functions (NN-layer-operations).

☞ These lines providing the information about the ***current state*** of the ***memory cell (memory-pipeline)*** to the ***NN-layer-operations*** like a ***peephole***.

☞ These allow ***NN-layer-operations*** to decide about the valves with taking into account what is actually sitting there in the memory.

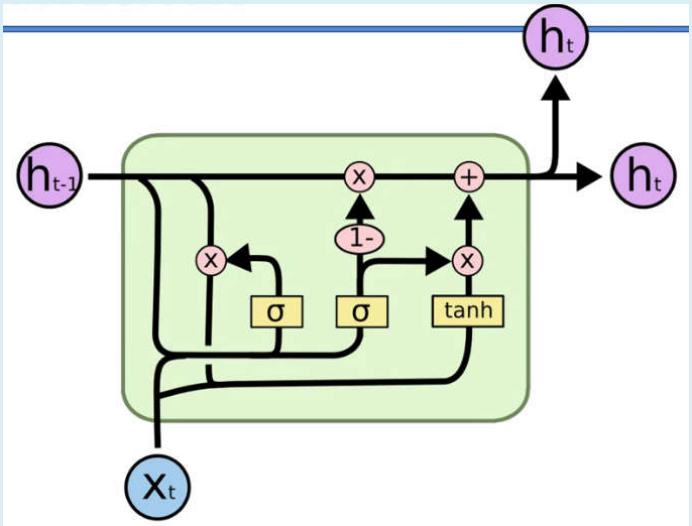
❒ ***Variation no. 2:*** Connected ***Forget-valve*** and ***Memory-valve*** with ***-1 pointwise-operator***.

- ☞ Instead of having a separate decision for the ***memory-valve***, now you have a ***combined decision*** for the ***forget-valve*** and the ***memory-valve***.
- ☞ Whenever you add something into memory, so whenever you close the ***forget-valve*** off whenever this is ***0***, ***memory-valve*** becomes the opposite i.e. it opened. ***-1 pointwise-operator*** turns ***0*** into ***1***. So it makes sense to combine them sometimes.



❒ ***Variation no. 3:*** A very popular modification called ***Gated Recurring Units***, ***GRUs***, for short.

- ☞ It completely get rid of the **C-pipeline** (**cell-pipeline** or **Memory-Pipeline**), and they replace it with the ***H-pipeline*** (h_t the ***Output-pipeline***), which is the ***hidden-pipeline***, which we had before at the ***bottom***.
- ☞ It simplifies things, bit ***less flexible***, but in terms of how many things are being ***controlled*** and ***monitored***. It might look a bit more ***convoluted***, but in reality it is a bit ***simpler***.
- ☞ You only have 3 valves, two are connected as well.
- ☞ The constant behind it is to get rid of the **memory cell** (**memory-pipeline**) and just have this ***one pipeline*** to takes care of everything.



❒ ***Additional reading:*** A good paper to check out is called "**LSTM A Search Space Odyssey**" by ***Klaus Greff*** and others, 2015.

- ☞ There they compared quite a few different LSTMs. You might like this research that they did.

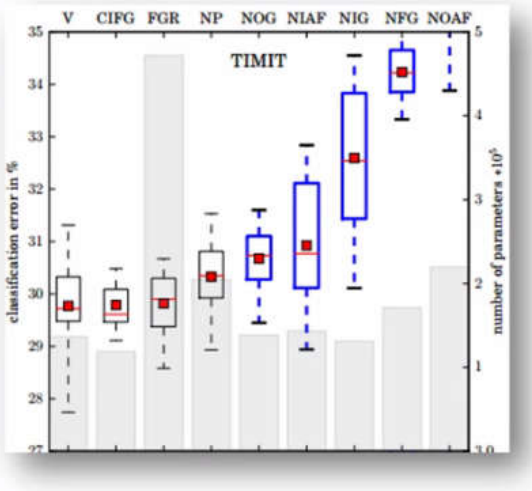
Additional Reading:

LSTM: A Search Space Odyssey

By Klaus Greff et al. (2015)

Link:

https://arxiv.org/pdf/1503.04069.pdf



Deep Learning

RNN: Building a RNN

Python Implementation

12.3.1 Problem Description

Here we predict the ***stock price*** of ***Google***. If you have some notions in ***financial engineering***, you already know that it's ***pretty challenging***, since indeed there is the ***Brownian Motion*** that states that the ***future variations*** of the ***stock price*** are ***independent*** from the ***past***.

- So it's actually ***impossible*** to predict ***exactly*** the ***future stock price*** otherwise we would ***all*** become ***billionaires*** but it's actually possible to ***predict*** some ***trends***.

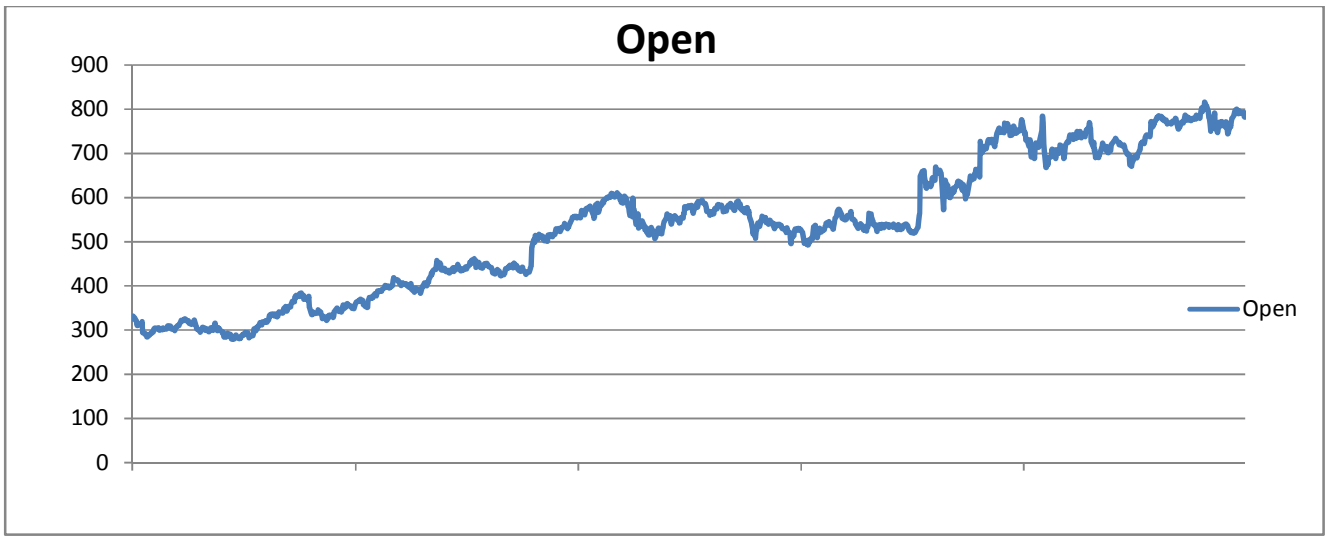
- So we're gonna try to predict the upward and downward trends that exist in the Google stock price.

- ***Our model:*** The model that we will implement will be an LSTM.
  - Our LSTM will try to ***capture*** the ***downward*** and ***upward trend*** of the ***Google stock price***.
  - We're not gonna implement a ***simple LSTM***, it's gonna be super ***robust*** with some ***high-dimensionality***, several ***layers***, it's gonna be a ***stacked LSTM***, then we're gonna add some ***dropout regularization*** to avoid overfilling and we will use the ***most powerful optimizer*** that we have in the ***Keras*** Library.

- ***Our approach:*** We're gonna train our ***LSTM*** model on five years of the ***Google stock price***, from the ***beginning*** of ***2012*** to the ***end*** of ***2016***.
  - Based on this training, and on the correlations identified by the ***LSTM*** of the ***Google stock price***, we will try to predict the **first month**, ***January 2017***.
  - Remember, we're ***not*** going to ***try*** to ***predict*** exactly the ***stock price***, we're gonna try to ***predict*** the ***trend***, the ***upward*** or ***downward trend*** of the Google stock price.

- ***Data-set:*** In our working directory, there are two ***.csv*** files, one for ***train*** and other for ***test***.
  - The train data, ***Google_Stock_Price_Train.csv*** contains the ***Google stock price from beginning 2012 to the end of 2016***, and then the test data ***Google_Stock_Price_Test.csv*** that contains the ***first month of 2017***, that is, the whole financial month of January 2017.
  - In the ***Google_Stock_Price_Train.csv*** and we're gonna try to predict the **Open** i.e. the stock price at the ***beginning*** of the ***financial day***. Let's go to Google Sheet, Excel or any spreadsheet to have closer look at the Google stock price.
  - ***MsExcel:*** Choose columns and **Insert** > **Line** : choose the **style**. So let's have a look at the Google stock price during this period.

	A	B	C	D	E	F
1	Date	Open	High	Low	Close	Volume
2	1/3/2012	325.25	332.83	324.97	663.59	7,380,500
3	1/4/2012	331.27	333.87	329.08	666.45	5,749,400
4	1/5/2012	329.83	330.75	326.89	657.21	6,590,300
5	1/6/2012	328.34	328.77	323.68	648.24	5,405,900
6	1/9/2012	322.04	322.29	309.46	620.76	########
7	1/10/2012	313.7	315.72	307.3	621.43	8,824,000
8	1/11/2012	310.59	313.52	309.4	624.25	4,817,800
9	1/12/2012	314.43	315.26	312.08	627.92	3,764,400
10	1/13/2012	311.96	312.3	309.37	623.28	4,631,800
11	1/17/2012	314.81	314.81	311.67	626.86	3,832,800
12	1/18/2012	312.14	315.82	309.9	631.18	5,544,000
13	1/19/2012	319.3	319.3	314.55	637.82	########
14	1/20/2012	294.16	294.4	289.76	584.39	########



☞ You can see, the ***beginning*** of ***2012*** to the end of 2016 we have some ***upward trends***, globally, with several ***downward trends***. So that's the ***training set***.

☝ Remember that these are financial days, so there's no ***Saturday*** or ***Sunday***.

☞ The LSTM model predict the ***test-data***. It will have no idea of what the ***Google stock price*** will be right after ***2016***.

☞ Once our model is ***trained***, we will predict the ***Google stock price*** for the whole month of ***January***, and we will compare our predictions to the ***actual*** results that are in our ***test-dataset***.

We proceed the following steps:

- ***Data-preprocessing:*** We do it from scratch. Different from ANN/CNN.
- ***Building RNN-Architecture:*** Define the layer, no. of neurons, output-layer.
- ***Prediction:*** We use our RNN to predict the test-dataset. We will make the predictions on January 2017.
- ***Visualization:*** Finally we visualize and compare our Train & Test results.

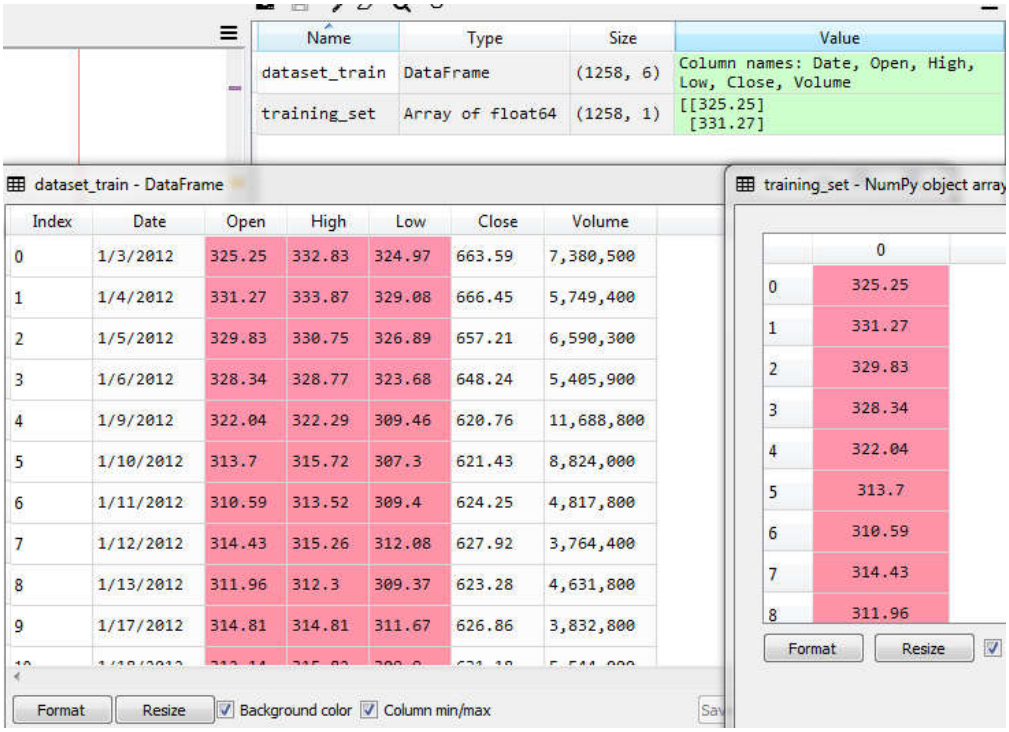
12.3.2 Data-preprocessing

It is different than ANN & CNN. We're gonna implement it from scratch.

❒ ***Libraries:*** Following are the essential libraries we're gonna use to implement the RNN.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

❒ ***Import the training sets:*** Notice, we are importing the ***`training-set`*** and not the whole ***`data-set`***. Because we are gonna train our ***RNN*** on only the ***training sets***. The RNN will have ***no idea*** of what's going on in a ***test set***.



Name	Type	Size	Value
dataset_train	DataFrame	(1258, 6)	Column names: Date, Open, High, Low, Close, Volume
training_set	Array of float64	(1258, 1)	[[325.25] [331.27]

dataset_train - DataFrame

Index	Date	Open	High	Low	Close	Volume
0	1/3/2012	325.25	332.83	324.97	663.59	7,380,500
1	1/4/2012	331.27	333.87	329.08	666.45	5,749,400
2	1/5/2012	329.83	330.75	326.89	657.21	6,590,300
3	1/6/2012	328.34	328.77	323.68	648.24	5,405,900
4	1/9/2012	322.04	322.29	309.46	620.76	11,688,800
5	1/10/2012	313.7	315.72	307.3	621.43	8,824,000
6	1/11/2012	310.59	313.52	309.4	624.25	4,817,800
7	1/12/2012	314.43	315.26	312.08	627.92	3,764,400
8	1/13/2012	311.96	312.3	309.37	623.28	4,631,800
9	1/17/2012	314.81	314.81	311.67	626.86	3,832,800

Format Resize Background color Column min/max

training_set - NumPy object array

	0
0	325.25
1	331.27
2	329.83
3	328.34
4	322.04
5	313.7
6	310.59
7	314.43
8	311.96

Format Resize

☞ It's like the **test-set *doesn't exist*** for the ***RNN*** (not even for validation). We're not importing the training-set right now, we do that after train the ***RNN***. Once the training is done, we will introduce the test-set to the ***RNN***.

```
dataset_train = pd.read_csv('Google_Stock_Price_Train.csv')      # Notice train-set is now our main dataset
training_set = dataset_train.iloc[:, 1:2].values
```

⇨ First line import the data as a DataFrame using Pandas.

⇨ The second line creates the **NumPy** array using the **right columns**. We did two things, selecting the ***`right column`*** and creating a ***NumPy array***.

⇨ It's the real ***training set*** containing the *input* data of the ***NN***.

☝ ***iloc[:, 1:2]*** Start from column *indexed* 1 and end at ***(2-1)=1*** *indexed* column and ***exclude*** column *indexed* ***2***. First '**:**' take all the rows,

☑ ***data.iloc[3]***
Gives the 4th row.

☑ ***data.iloc[1:3]***
will give row indexed ***1***, ***2*** and exclude 3 (i.e. 2nd, 3rd rows excluding 4th row.). ***iloc[k:m]*** starts from ***k*** (including ***k***), ends at ***m-1*** excluding ***m***.

☑ ***data.iloc[i:j, n:m]***
Gives ***rows*** starting from i and ends at ***j-1*** excluding ***j***. And ***columns*** starting from ***n*** (including ***n***), ends at ***m-1*** excluding ***m***.

☝ "**.values**" transform the data to NumPy-array. So we don't have a ***vector***, we have a ***NumPy array*** of one column.

12.3.3 Feature scaling

We know there are ***standardization*** and ***normalization***. What are we going to use this time for ***RNN***? In previous chapter we used ***standardization***.

Standardization	***Normalization***
$x_{stand} = \frac{x - mean(x)}{standard\ deviation\ (x)}$	$x_{norm} = \frac{x - min(x)}{max(x) - min(x)}$

❒ ***Normalization is more relevant for RNN:*** Whenever you build an ***RNN*** and especially if there is a ***sigmoid function*** as the ***activation*** function in the ***output layer*** of you ***RNN***, well I recommend to apply ***Normalization***.

☞ We're going to use the ***MinMaxScaler*** class from *preprocessing* module of *sklearn*.

```
from sklearn.preprocessing import MinMaxScaler
```

☞ Next we create scale object ***sc***, we have to input some arguments, ***feature_range*** equals ***(0, 1)***, that's the **default** feature **range**.

```
sc = MinMaxScaler(feature_range= (0, 1))
```

➢ The new *scaled stock prices* will be between *zero* and *one*.

☞ The last thing we need to do, is of course to apply this ***sc*** object to apply normalization to ***training_set***.

```
train_set_scled = sc.fit_transform(training_set)
```

➢ We create a new variable ***train_set_scaled***, because it is ***recommended*** to ***keep*** your *original datasets*.

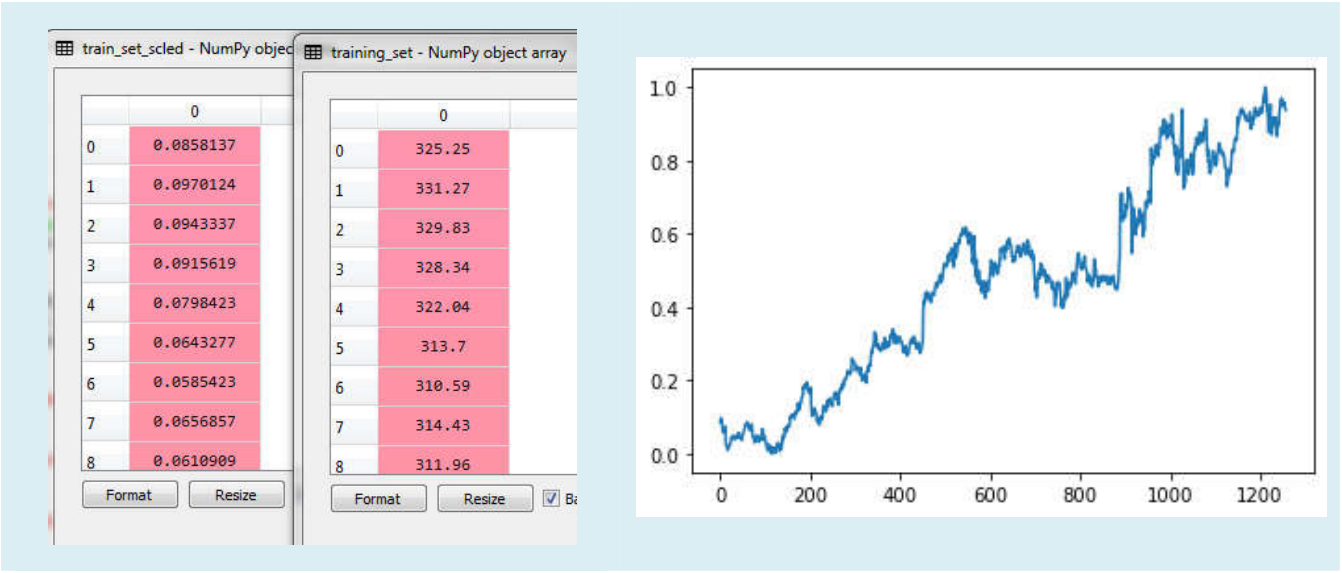
➢ Then we apply ***fit_transform*** as we did in **previous chapters**, which will not only fit your object **sc** to the **training_set**, also *transform* it, that is ***scale*** it.

☝ Here ***fit*** means it just going to get the *min* and the *max* of the *data*.

☝ The ***transform*** method, it's going to compute for each of the stock prices of the training set, the scaled stock prices according to normalization formula.

```
# feature scaling
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range= (0, 1))
train_set_scled = sc.fit_transform(training_set)
```

❒ After *executing* the code, we obtained our training set *scaled/normalized* between ***0*** and ***1***. The **last values** are **close** to ***1*** because remember, the **stock** price was **going up** between **2012** and **2016** i.e. getting closer to max value.



☞ Now we have now the ***right values*** for our future ***RNN*** that we're going to ***build***.

12.3.4 Time steps – Data-structure

In this step, we'll create ***a specific Data Structure***, that's the most important step actually of **data pre-processing** for RNN.

- We're going to create a ***Data Structure*** specifying what the RNN will ***need to remember*** when *predicting* the *next stock price*.
- And this is called the ***number of time steps***. This is important to have the right number of ***time steps*** because a ***wrong*** number of time steps could ***lead*** to ***over fitting*** or nonsense predictions.

❒ ***time-steps:*** We have to create a *special data structure* with, ***60 time-steps*** and ***one output***.

☞ ***60 time-steps*** means that at each time ***T***, the RNN is going to look at the ***60*** **stock prices** before ***time T***, that is ***the stock prices*** between ***60*** days before time ***T*** and time ***T***.

- Based on the ***trends*** of these ***60*** previous ***timesteps***, it will try to predict the ***next output***.
- So ***60 timesteps*** of the ***past information*** from which our ***RNN*** is gonna try to ***learn*** and understand some ***correlations***, or some ***trends***, and based on its ***understanding***, it's going to try to ***predict*** the ***next output***. That is, the stock price at time ***T + 1***.

☞ ***Why 60 time steps:*** It is fixed by some experiments, trying different number of time steps.

- ***One*** time ***step*** first, which is completely stupid, because it led to ***overfitting***. The model won't learn anything.
- Then ***20 timesteps***, which was ***not enough*** to be able to ***capture*** some ***trends***, then 30, 40.
- Eventually the **best number** of **timesteps** we ended up with was ***60***.

☞ ***60 timesteps*** correspond to the ***60*** previous ***financial days***, and since ***20 financial days*** in **one month**, well ***60 timesteps*** correspond to **three months**.

- That means that each day we're gonna look at the ***three previous month*** to try to ***predict*** the stock price the ***next day***.
- So we're gonna have ***60 timesteps*** and ***one output***, which will be the stock price at time ***T + 1***.

❒ **X_train *and* y_train:** These are created **differently** in ***RNN***. In previous chapters we just choose the columns from data-set. And *splitted* those using *train-test-split*.

☞ Here in RNN we have to make these manually. Let ***n*** is our data-size.

i. We first create a list of list, ***2D-list***. Each list is a row of ***60 days data***. This will be ***X_train*** matrix. (i.e. $n \times 60$ *matrix*).

ii. Then we also create another vector a ***1D-list***, starting from ***61st data***, i.e. (i.e. $(n - 60) \times 1$ vector). This will be ***y_train vector***.

iii. Then we ***Convert*** these to ***NumPy*** array to implement in our ***RNN***.

☞ ***X_train*** will be the input of the neural network, and ***y_train***, will contain the output.

- ➢ For each observation, that is for each ***financial day***, ***X_train*** will contain the ***60*** **previews stock prices**, before that financial day.
- ➢ ***y_train*** will contain the stock price the ***next financial day***.

☞ We use a ***for loop*** to append data points to ***X_train*** and ***y_train***.

```
print(f"Size of train set : {train_set_scled.shape}")
print(f"No. of data-points : {train_set_scled.shape[0]}")


# creating a data structure with 60 time-steps and 1 output.
data_size = train_set_scled.shape[0]
X_train = []
y_train = []

for i in range(60, data_size):
    X_train.append(train_set_scled[(i-60):i, 0])     # append a list of 60 data-points in one row and (data_size-60) rows
    y_train.append(train_set_scled[i, 0]) # vector of (data_size-60) rows

# Convert X_train and y_train to NumPy array
X_train, y_train = np.array(X_train), np.array(y_train)
```



	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
0	0.044	0.045	0.046	0.044	0.037	0.045	0.051	0.052	0.056	0.058	0.065	0.069	0.072	0.08	0.078	0.08	0.085
1	0.045	0.046	0.044	0.037	0.045	0.051	0.052	0.056	0.058	0.065	0.069	0.072	0.08	0.078	0.08	0.085	0.086
2	0.046	0.044	0.037	0.045	0.051	0.052	0.056	0.058	0.065	0.069	0.072	0.08	0.078	0.08	0.085	0.086	0.085
3	0.044	0.037	0.045	0.051	0.052	0.056	0.058	0.065	0.069	0.072	0.08	0.078	0.08	0.085	0.086	0.085	0.075
4	0.037	0.045	0.051	0.052	0.056	0.058	0.065	0.069	0.072	0.08	0.078	0.08	0.085	0.086	0.085	0.075	0.079
5	0.045	0.051	0.052	0.056	0.058	0.065	0.069	0.072	0.08	0.078	0.08	0.085	0.086	0.085	0.075	0.079	0.072
6	0.051	0.052	0.056	0.058	0.065	0.069	0.072	0.08	0.078	0.08	0.085	0.086	0.085	0.075	0.079	0.072	0.067
7	0.052	0.056	0.058	0.065	0.069	0.072	0.08	0.078	0.08	0.085	0.086	0.085	0.075	0.079	0.072	0.067	0.063

	0
0	0.086
1	0.085
2	0.075
3	0.079
4	0.072
5	0.067
6	0.063
7	0.068
8	0.068

i. We can see the 60 previous stock prices in ***X_train***, and the next stock price in ***y_train***.

ii. Notice in ***y_train*** 1st entry is 61st data point, then 62nd data-point and so on.

iii. In ***y_train*** each **n-th element** is the **last element** of ***X_train***'s **(n+1)-th** row.

iv. So there is total ***1258-60 = 1198 time-steps***, and as ***time step*** increase as ***1-stride*** the row's ***element's*** move ***1-stride*** to the ***left***.

☞ The first line of observation here corresponds to time $T = 60$. At the 60th financial day of our training dataset.

- ➢ Because since here we're getting the 60 stock prices before $T = 60$, and in 2nd row, the 60 stock prices before $T = 61$, hence we have 59 stock prices in common.
- ➢ Because we have this **sliding window** of size ***60***, sliding with a ***stride of one*** at each observation, from one observation to the next.

☝ That's exactly the idea of the ***RNN***. It is memorizing what's happening in the ***60 previous*** **timesteps** to predict the next value at time ***T+1***.

That's the approach, the ***data structure*** we need to create as the ***input*** of a ***RNN***.

12.3.5 Adding Extra Dimension to our Data Structure

This is the last step of the ***Data Preprocessing***, we will add a ***new dimension*** to this ***structure***, which makes it more ***powerful***.

- You don't have to do it with only one column, ***Open*** stock price, but with several other indicators.
- These ***indicators*** can be some ***other columns*** of our dataset, like for example the ***Close*** column, or the ***Volume***.
- Or even some other ***different stock prices/other-companies*** related to ***Google***. For example: ***Apple*** is related to ***Samsung***.

This step is about ***reshaping*** the ***data*** that is adding some even ***more dimensionality*** to the ***data structure*** that we just made.

- This ***dimension*** is actually the ***unit***, that is, the number of ***predictors*** used to predict Google stock price at **T+1**. In this financial engineering problem, where we try to predict the ***trend*** of the ***Google Stock Price***, these ***predictors*** are ***indicators***.
- Right now we have ***one indicator***, which is the ***Open*** Google Stock Price.
- Now in this ***new dimension*** that we're gonna add to our ***data structure***, we will be able to add some ***more indicators***.
- That could help ***predict*** even better the ***upward*** and ***downward*** trends of the ***Google Stock*** Price (we're not gonna do it in this implementation. We will just use the ***open*** *Google Stock Price*).

Implementing Reshape:

- It's actually really simple, it just takes **one line of code**. We're gonna use the ***reshape()*** function. Anytime you want to add a **dimension** in a ***NumPy array*** you always need to use the ***reshape()*** function.
- We just need to do this for ***X_train***, because ***X_train*** actually contains the inputs of the ***NN***.

Also we need to create this new dimensionality of this new data structure, because this shape is expected by the future RNN that we're gonna build in the next.

- So that's **not only** for use some more **indicators**, but also to be **compatible** with the **input format/shape** of the ***RNN***.

reshape() is taken from ***NumPy*** library. In this ***reshape()*** function, we need to input two arguments.

- The first argument is our current-data-structure that we want to reshape.
- In the second argument we need to specify this new structure. We're gonna include three elements (to add these three dimensions), because right now, our data structure has two dimensions, a NumPy array of 1198 rows and 60 columns.
  - After adding a new dimension it will be a Stack of 1198×60 NumPy array.

```
# Reshaping
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
```

  - `X_train.shape[0]` = no. of ***rows*** of `X_train = 1198`
  - `X_train.shape[1]` = no. of ***columns*** of `X_train = 60`
  - No. of indicators = 1. We set it 1, because we're not gonna use other indicators like "Close" or "Volume", we stick to our only indicator "Open".
  - Our ***2D matrix*** is now the ***first-layer*** of a ***3D tensor***.

- Function ***numpy.reshape*** gives a new shape to an array without changing its data. It is a ***numpy*** package function. First of all, it needs to know **what to reshape**, which is the **first argument** of this function (in your case, you want to reshape ***X_train***).
- Then it needs to know what is the ***shape*** of your ***new matrix***. This argument needs to be a ***tuple***. For 2D reshape you can pass ***(W,H)***, for three dimensional you can pass ***(W,H,D)***, for four dimensional you can pass ***(W,H,D,T)*** and so on.
- However, you can also call ***reshape*** a ***Numpy matrix*** by ***X_train.reshape((W,H,D))***. In this case, since ***reshape*** function is a method of ***X_train*** object, then you do not have to pass it and only pass the new shape.
- It is also worth mentioning that the total number of element in a matrix with the new shape, should match your original matrix. For example, your 2D **X_train** has **X_train.shape[0]** $\times$ **X_train.shape[1]** elements. This value should be equal to $W \times H \times D$.

Tensor is like 3D version of an array. The input should be a 3D array, containing the following three dimensions.

(batch_size, time_steps, input_dim)

- First dimension is ***batch_size***, which will correspond to the total number of observations we have (no. of rows of ***X_train***).
- The second dimension is the ***time_steps***, (no. of columns of ***X_train***).
- The third dimension, ***input_dim*** is the no. of indicators that we're adding (no. of the predictors i.e. ***Open***, so we used ***1***). We are doing it because our RNN takes a 3D tensor, but:
  - This can be some ***new financial indicators***, that could help predict the Google stock price trends.
  - For example, that can be the "***Closed***" Google stock price or even some other stock prices from other companies that are related to Google.
  - Other example can be ***Apple*** and ***Samsung***, you know that in ***an iPhone's*** most of the ***material*** is coming from ***Samsung***. And therefore ***Apple*** is highly ***dependent*** on ***Samsung***, but at the same time, ***Samsung*** is highly dependent on ***Apple***, because simply, ***Apple*** is their ***best customer***. And therefore the stock prices of Apple and Samsung might be highly correlated.
  - Since we have one indicator, So we just inputted ***1***, but don't forget to change it if you have **several indicators**.
- Instead of using numbers we use ***X_train.shape*** so that later we can use any **size** of **data-structure**.

So now we have our ***final structure***, that is expected by the ***neural network***, our future ***RNN***.

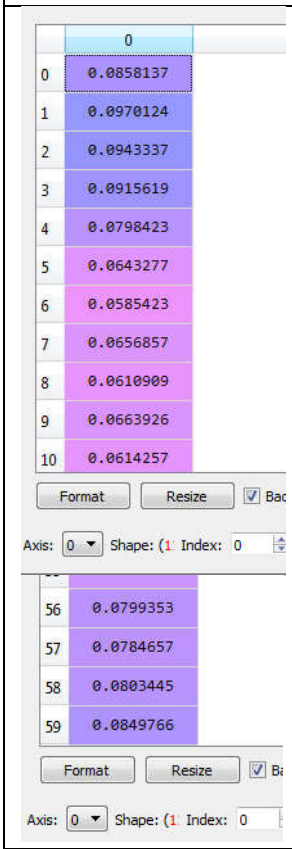
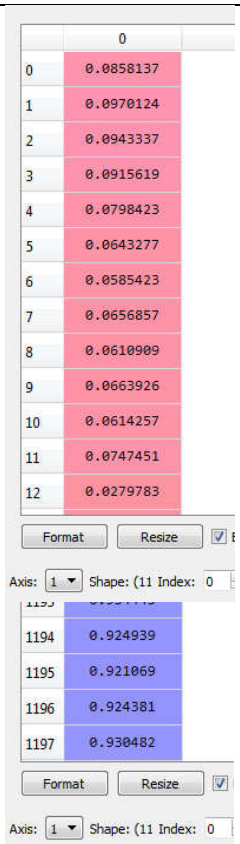
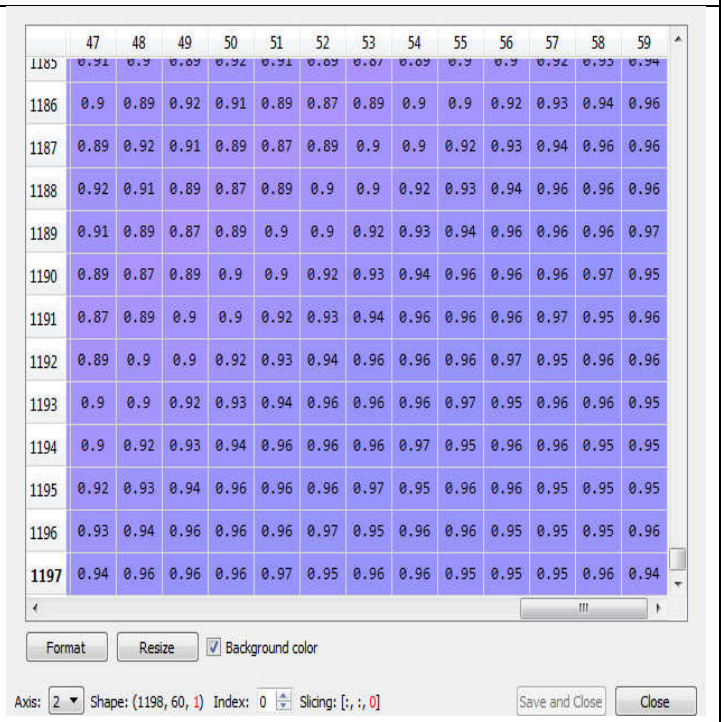
[1]. First dimension corresponding to the number of ***stock prices***.

[2]. The second dimension corresponding to the number of ***time steps***.

[3]. And the third dimension corresponding to the number of ***indicators***.

X_train	Array of float64	(1198, 60, 1)	[[[0.0858] [0.0970]

You can see its changing the axis here.

Dimension 1 (Axis : 1)	***Dimension 2 (Axis : 2)***	***Dimension 3 (Axis : 3)***
		

- At each time ***T***, starting from ***60***. The ***60th*** financial day.

12.3.6 Building the RNN: Structure of our RNN

Now we're going to build the RNN, the whole architecture of our LSTM.

- ☞ It's not gonna be an LSTM, it's gonna be a stacked LSTM with several LSTM layers and we're gonna make it perfect.
- ☞ We're gonna make it robust by adding some ***dropout regularization*** to avoid ***overfitting***.

☝ But keep in mind: ***"All models are wrong, but some are useful."***

❒ ***Libraries and Packages:***

```
# importing keras libraries and packages
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
```

- ☑ **`Sequential`** class will allow us to create a neural network object representing a sequence of layers (other representation is graph).
- ☑ **`Dense`** class to add the output layer.
- ☑ **`LSTM`** class, to add the LSTM layers.
- ☑ **`Dropout`** class to add some dropout regularization.

❒ ***Initialize our RNN:*** We gonna initialize our RNN as a sequence of layers.

```
# Initialize the RNN
regressor_rnn = Sequential()
```

☞ We're gonna use the ***`Sequential`*** class from ***keras*** to introduce our ***regressor*** as a ***sequence of layers***. [Since we're dealing with regression problem, we used `'regressor'`, for classification problem we used `'classifier'`]

➢ This time we're predicting a **continuous** output, the **Google stock price**, And therefore we are doing some **regression**.

➢ Remember, **regression** is about predicting the **continuous value** and **classification** is about predicting a **category/class**.

✋ We also gonna build ***Computational Graph*** in *following chapters*. To build some **computational graphs**, we use **`Pytorch`**, because for this, which is much more powerful tool for dynamic graphs.

❒ ***Add The Different Layers:*** And now we gonna add the **different layers**, to make it a powerful **stacked LSTM**. We add the first ***LSTM layer***, of our ***RNN*** which was introduced as a ***sequence*** of layers and also some ***`Dropout` regularization*** to avoid ***`Overfitting`***.

```
#adding first LSTM layer & some Dropout-Regularization
regressor_rnn.add(LSTM(units= 50, return_sequences= True, input_shape = (X_train.shape[1], 1)))
regressor_rnn.add(Dropout(rate = 0.2))
```

☞ Inside the ***`add()`*** method, we need to *input* the *layer*, the *type of layer* we want to add.

☞ Since we want to add is an ***LSTM layer***, so we use ***LSTM*** class to add first layer (and ***`Dense`*** to add output layer).

☞ ***Parameters:***

➢ **`units:`** It is the number of LSTM cells or units in our LSTM layer. We set a very high number of LSTM cells, or memory units, (for simplicity's let's just call them ***neurons***).

✋ ***Why more dimensions:*** Now *already* our model will have a very ***high dimensionality***, because we are going to ***`stack`*** many ***`layers`***. But we can increase this *dimensionality* even more by including a *large number* of *neurons* in *each* of the ***LSTM layers***.

☝ Since capturing the ***trends*** of a ***stock price*** is pretty ***complex***, we need to have this high dimensionality and therefore we also need to have a large number of neurons in each of the multiple LSTM layers.

☝ And therefore, number of neurons we'll choose for this first LSTM layer is gonna be 50. The next layers, will also have 50 neurons, that will get us a model with high dimensionality and will lead us to better results.

➢ **`return_sequences:`** We set it ***`True`***, because we are building a ***stacked LSTM*** which have ***several LSTM layers***. When you add another LSTM layer one after another, you have to set *`return_sequences`*= **`True`**.

- ❍ Once you are done with your LSTM layers, you are not gonna add another one after that, you will set it *`return_sequences`*= **`False`**, (however we don't have to do it because it's a default value of ***`return_sequences`***).

➢ **`input_shape:`** It is exactly the shape of the input containing ***`X_train`*** that we created in the last step of the data preprocessing part. It's an input shape in 3D, corresponding to the ***`observations`***, the ***`time_steps`***, and the ***`indicators`***.

- But in this third argument of the LSTM class, we won't have to include all the three dimensions, only the ***`time_steps`*** and the ***`indicators`***,
- i.e. from `(X_train.shape[0], X_train.shape[1], 1)` we need to input **`(X_train.shape[1], 1)`**. The first one `X_train.shape[0]`, corresponding to the observations, will be **automatically** taken into account.

☞ ***Adding Dropout-Regularization:*** We recommend to use 20% i.e. 0.2 because it's quite relevant. 20% of the neurons i.e. 10 out of 50 neurons of the LSTM layer, will be ignored during the training, during the forward propagation and back propagation.

```
regressor_rnn.add(Dropout(rate = 0.2))
```

We're gonna add a ***total*** of ***4 LSTM layers***, so that will make a big, stacked LSTM.

12.3.7 Building the RNN: extra LSTM layers

Now we add some **extra LSTM layers**, with ***dropout regularization***.

❒ ***Adding 2nd LSTM layer:*** We only need to do one change, which is the ***`input_shape`***.

```
regressor_rnn.add(LSTM(units= 50, return_sequences= True))
regressor_rnn.add(Dropout(rate = 0.2))
```

☞ We had to specify the ***`input_shape`*** in *first* input *layer* but we don't need it for the ***second layer***, because it's recognized automatically. Thanks to this ***`units`*** argument, which tells exactly that we have ***`50 neurons`*** in the ***`previous layer`***. So no need to specify any ***`input_shape`*** here, when you're adding your ***next LSTM layers***, after the ***first one***.

☞ ***50 neurons*** in *2nd hidden layer* adds some *high dimensionality* to our model, to be able to handle the *complexity* of the *problem*.

☞ *Augmenting* the *dimensionality* of our model, will *augment* at the same time the *complexity*, and therefore will *respond better* to the *complexity* of the *problem* and that will eventually lead us to *better results*.

☞ We're keeping a ***`20% dropout`*** for the regularization, since that's a relevant choice.

❒ ***Adding 3rd LSTM layer:*** We only need to copy the 2nd LSTM layer. All are just same.

```
regressor_rnn.add(LSTM(units= 50, return_sequences= True))
regressor_rnn.add(Dropout(rate = 0.2))
```

❒ ***Adding 4th LSTM layer:*** We only need to copy the ***2nd LSTM layer***. The parameter ***`return_sequences`*** is set to ***`False`*** because it is the *last LSTM layer* and next *layer* is the *output layer*.

```
regressor_rnn.add(LSTM(units= 50, return_sequences= False))
```

☞ Because we're ***not going to return anymore sequences***, and therefore, since the ***default value*** of the ***`return_sequence`***'s parameter is **`False`**, we just removing this parameter.

☞ Be careful, this is ***not*** the ***final layer*** of our RNN, this is the fourth LSTM layer, but after that, comes the output layer, with the output dimension, ***`units`*** will be ***`1`*** of course, because we're predicting just ***`one value`***, the ***`value`*** of the ***`stock`*** price at ***`time T+1`***.

```
regressor_rnn.add(LSTM(units= 50))
regressor_rnn.add(Dropout(rate = 0.2))
```

✋ However we can specify the ***number*** of ***neurons*** in the ***LSTM layers***, and we're keeping ***`50 neurons`*** to have the same goal of having a **high dimensionality**.

☝ Also we're *keeping* the *20% dropout regularization*. We can also change that.

12.3.8 Building the RNN: output layer, compile & fit

❒ ***output-layer:*** For an **output-layer** we're ***not adding*** an ***LSTM*** layer, but a classic ***fully connected layer*** (similar to CNN). The output layer is a fully connected to the previous (4th-last) LSTM layer.

☞ To make a full connection we need to use the ***Dense*** class exactly as we did for ANN and CNN.

```
# the output Layer
regressor_rnn.add(Dense(units = 1))
```

☞ We specified ***units = 1***, because we're predicting a ***real value*** corresponding to the ***stock price***, i.e. the ***output*** has only ***one dimension***. So there will be ***one neuron*** in ***output-layer***. This will ***output*** our stock price, at time ***T +1***.

Now we're done with the architecture of our super robust LSTM RNN.

❒ ***Compiling the RNN:*** Our optimizer will be ***'adam'*** and the loss-function will be the ***mean_squared_error*** because we're doing some regression.

☞ We use the ***compile()*** method, it is another method of the ***Sequential*** class.

```
# ------ Compiling the Model ----------
regressor_rnn.compile(optimizer='adam', loss='mean_squared_error')
```

➢ ***optimizer:*** For RNN ***rmsprop*** is recommended in the ***keras documentation***.

- ***rmsprop*** is some kind of an advanced ***SGD*** optimizer that usually a good choice for ***RNN***.
- However, with our implementation we're not gonna use an ***rmsprop***, here we use the ***adam*** optimizer.
- The ***adam*** optimizer is always a safe choice for ***SGD***. It's always a ***good choice*** because it is very powerful and it always performs some relevant ***updates*** of the ***weights***.

➢ ***loss:*** Since, we're not doing classification anymore. We do not use ***cross-entropy***. So the loss is not gonna be '**binary_crossentropy**' or '**categorical_crossentropy**'.

- Now we're dealing with a ***regression problem*** because we have to predict a ***continuous value*** and the loss for this kind of problem is the ***mean_squared_error***.
- So that the error can be measured by the ***mean*** of the ***squared differences*** between the ***predictions*** and the ***targets***. ***Targets*** means the ***real values***.
- ***Mean squared error***, also sometimes called ***MSE***.

Now our ***regressor*** is compiled with a powerful ***optimizer 'adam'*** and the right ***loss*** function '*mean_squared_error*'.

❒ ***Fit/train this RNN to our training set:*** Notice we did not use ***training_set*** or ***train_set_scled***. Our ***training set*** is composed of ***X_train***, that's the right ***data structure*** that is ***expected*** by the ***neural networks***.

☞ So we need to take ***X_train*** and not ***training_set*** or ***train_set_scled***.

☞ We also need to ***specify*** the ***output*** (i.e ***y_train***), when fitting the ***regressor*** to our ***training sets*** because the ***output*** contains the ***ground truth*** that is the stock price at time ***T +1***.

➢ We're training the ***RNN*** on the ***true stock price*** at time ***T +1***, after the ***60 previous stock prices*** during the ***60 previous financial days***. So that's why we also need to include the ***ground truth*** (dependent - variable) and therefore ***y_train***.

☝ It is the final step of part two, ***building*** the ***RNN***. Here we fit our LSTM-RNN to training set, on ***X_train*** and ***y_train***.

☝ The ***training*** will happen in ***this part*** and in the end we'll have a ***robust RNN***, but mostly, we will have a smart one. Ours ***RNN*** will be ***intelligent***, ***trained***, neural network, to be able to ***predict*** in some way, the ***upward trends***, and ***downward trends*** of the Google stock price.

☞ We need to input four arguments, which are going to be: Feature/independent ***X_train***, output/dependent ***y_train***, no. of ***epochs*** and ***batch_size*** for the sample.

➢ ***X_train:*** The *independent variables* of the *training* set. Which will be the ***input*** of the ***neural network*** and will be ***forward propagated*** to the ***output***, which will be the ***prediction*** and that will compare to the ***ground truth*** that is contained in ***y_train***.

➢ ***y_train:*** Are the real values for predictions, it is the dependent variables.

➢ ***number of epochs:*** That is on how many iterations do you want your RNN to be trained. Or how many times do you want the whole training data to be forward propagated inside the NN and then back propagated to update the weight.

- ***What is the best number:*** You have to test it several times. For example:
  - I tried with first ***25 epochs*** and noticed that there wasn't a ***convergence*** of the ***loss***.

- Then I tried, 50 still not some convergence.
- Finally I tried 100, and there I observed some convergence of the ***loss***.

- So I think 100 is the right number of epochs. For this kind of problem, besides we don't have that much data. We only have the Google stock prize for 5 years.

➢ ***batch_size:*** We set this so that our ***RNN*** trained on ***batches*** of ***observations/stock prices*** going into the NN. So instead of updating the weight for each observation (stock price) during ***forward-propagation*** generating ***error*** and ***back-propagation***, we do that, every 32 stock prices.

If you want you can train this ***RNN*** on even ***more*** than ***5 years***. The data is available online, it's the real ***Google stock price*** that you can take from many financial sources. One of them can be for example, ***Yahoo finance***. So feel free to ***play*** with it and ***experiment*** even more to create some maybe even more robust RNN.

But anyway with ***5 years*** of training data well ***100 epochs*** turned out to be a good choice, with some ***convergence***.

All code at once (only fit)

```
# Recurrent Neural Netwark: RNN

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# ------------------- Data Preprocessing ----------------------
dataset_train = pd.read_csv('Google_Stock_Price_Train.csv')     # Notice train-set is now our main dataset
training_set = dataset_train.iloc[:, 1:2].values

# feature scaling
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range= (0, 1))
train_set_scled = sc.fit_transform(training_set)


print(f"Size of train set : {train_set_scled.shape}")
print(f"No. of data-points : {train_set_scled.shape[0]}")


# creating a data structure with 60 time-steps and 1 output.
data_size = train_set_scled.shape[0]
X_train = []
y_train = []

for i in range(60, data_size):
    X_train.append(train_set_scled[(i-60):i, 0])    # append a list of 60 data-points in list of (data_size-60) rows
    y_train.append(train_set_scled[i, 0]) # vector of (data_size-60) rows

# Convert X_train and y_train to NumPy array
X_train, y_train = np.array(X_train), np.array(y_train)

# Reshaping
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))


# ------------------- Building RNN ----------------------

# importing keras libraries and packages
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout

# Initialize the RNN
regressor_rnn = Sequential()

# adding first LSTM layer & some Dropout-Regularization
regressor_rnn.add(LSTM(units= 50, return_sequences= True, input_shape = (X_train.shape[1], 1)))
regressor_rnn.add(Dropout(rate = 0.2))

# second LSTM layer with Dropout-Regularization
regressor_rnn.add(LSTM(units= 50, return_sequences= True))
regressor_rnn.add(Dropout(rate = 0.2))
```

```
# third LSTM layer with Dropout-Regularization
regressor_rnn.add(LSTM(units= 50, return_sequences= True))
regressor_rnn.add(Dropout(rate = 0.2))

# forth LSTM layer (last LSTM layer) with Dropout-Regularization
regressor_rnn.add(LSTM(units= 50))
regressor_rnn.add(Dropout(rate = 0.2))

# the output Layer
regressor_rnn.add(Dense(units = 1))


# ------ Compiling the Model ----------
regressor_rnn.compile(optimizer='adam', loss='mean_squared_error')


# ------- Fit the model to Training dataset (model learns) ----------
# The training will happen in this part.
regressor_rnn.fit(X_train, y_train, epochs= 10, batch_size=32)
```

```
Epoch 1/10
38/38 [==============================] - 10s 71ms/step - loss: 0.0435
Epoch 2/10
38/38 [==============================] - 3s 78ms/step - loss: 0.0063
Epoch 3/10
38/38 [==============================] - 3s 75ms/step - loss: 0.0055
Epoch 4/10
38/38 [==============================] - 2s 65ms/step - loss: 0.0053
Epoch 5/10
38/38 [==============================] - 2s 62ms/step - loss: 0.0063
Epoch 6/10
38/38 [==============================] - 2s 62ms/step - loss: 0.0053
Epoch 7/10
38/38 [==============================] - 2s 62ms/step - loss: 0.0053
Epoch 8/10
38/38 [==============================] - 2s 62ms/step - loss: 0.0047
Epoch 9/10
38/38 [==============================] - 2s 62ms/step - loss: 0.0039
Epoch 10/10
38/38 [==============================] - 2s 62ms/step - loss: 0.0044
```

Notice the ***convergence*** of ***loss*** values during the ***training***. So we started with a ***loss*** near ***4%, 0.0435***. The final loses in the end in the last 20 epochs, it's staying around ***0.0015*** i.e. ***0.15%***.

Too small loss indicates overfitting: So if you want to try more epochs, I think the loss remain around 0.0015.

- That's because we added a ***Dropout Regularization***. And if you obtain a ***loss*** too small in the end well you might get some ***over fitting***.
- In that case you might get some ***small loss*** on the ***training data*** and really ***large loss*** on the ***test data***. So be careful not to obtain ***overfitting*** and therefore ***not to try to decrease*** the ***loss*** as much as possible.

And that's why here it seems that we get really ***good results***. Next we're going to make the ***Predictions*** and ***Visualizing*** the Results to observe the ***predictions***.

12.3.9 Prediction from the RNN

In this part we'll make the predictions and visualize the results. There is three steps:

i. First, we're gonna get the ***Real Google stock price of 2017-January*** from `Google_Stock_Price_Test.csv`.

ii. In the second step, we're gonna get the ***Predicted Google stock price of 2017-January*** from our trained **model**.

iii. And then in the final step, we will ***visualize*** the results.

Getting the real Google stock price of January 2017: We simply get it from the test dataset that we have, in the CSV file, ***Google_Stock_Price_Test.csv***.

- So first we create a data-frame using ***Pandas***
- Then we will select the right column, the ***Open*** Google stock price.
- And make it a ***NumPy*** **array**.

```
dataset_test = pd.read_csv('Google_Stock_Price_Test.csv')
real_stock_price = dataset_test.iloc[:, 1:2].values
```

❒ ***Data-preprocess for Test-Set before prediction:*** This part is little bit tricky. There are some important tricks that we will make sure to understand and apply, and some mistakes to absolutely avoid.

☝ Here we're gonna use our ***regressor***, and we're gonna ***predict*** the Google ***stock prices*** of ***January, 2017***.

☞ But the thing is, we ***trained*** our ***model*** to be able to ***predict*** the ***stock price*** at time ***T +1***, based on the ***60*** previous ***stock prices***, and, therefore, to ***predict*** each ***stock price*** of ***each*** financial day of ***January, 2017***, we will need the ***60 previous stock prices*** of the 60 previous financial days, before the actual day.

☞ In order to get at ***each*** day of ***January, 2017***, the ***60*** previous ***stock prices*** of the ***60 previous days***, well, ***we*** will ***need both*** the ***training set*** and the ***test set***, because we will have ***some of the 60 days*** that will be ***from*** the ***training set***, because they will be from ***December 2016***, and we will also have some ***stock prices*** of the ***test set***, because some of them will come ***from January 2017***.

⇨ Therefore, we need to do some concatenation of the ***training set*** and the ***test set***,

☝ ***How do we concatenate:*** If you think of making this concatenation by ***concatenating*** the training-set " **training_set** " and the test-set " **real_stock_price** ", it will lead us to a problem and we have to ***re-scale*** the result(we have to apply the ***fit_transform*** method again). We should never do this, we have to keep the actual test values as they are.

➢ The trick is, here we actually concatenate the original DataFrames, ***dataset_train*** and ***dataset_test***.

➢ From this concatenation, we will get the input of each prediction. Then we will ***scale*** it by our ***sc*** object. That's the way we are only ***scaling*** the ***input***, and not changing the ***actual test values***.

➢ We have to scale the input, because our ***RNN*** model was trained on the scaled values of the train set. That's why we use scaled input to get our predictions.

➢ The scaled input should be based on the same scaling i.e. normalization with our ***sc*** object.

☞ Here ***dataset_total*** will contain the whole concatenated dataset. We're gonna use ***concat()*** function from this ***Pandas*** library.

☞ We need to input two arguments:

i. The first one are the ***two DataFrames*** we want to concatenate,

ii. The ***second*** argument is the ***axis*** along which we want to make that concatenation. i.e. do we want to concatenate the ***lines/rows-wise***? Or do we want to concatenate the ***column-wise***?

☞ ***dataset_train['Open']*** contains the ***'Open'*** ***Google stock prices*** from 2012-2016, and and ***dataset_test['Open']***, contains ***'Open'*** ***Google stock prices*** of January, 2017.

```
dataset_total = pd.concat((dataset_train['Open'], dataset_test['Open']), axis = 0)
```

⇨ The trick is to specify a **column** in the **DataFrame** we can use their **name** inside **[]**, as we did ***['Open']***.

⇨ So *(dataset_train['Open'], dataset_test['Open'])* is the first argument of the ***concat*** function.

⇨ Now the ***axis*** we want to ***concatenate*** is the ***lines***, i.e. we concatenate the ***rows***.

➢ Therefore, we need to make a concatenation along the vertical axis. To do this we need to use ***axis = 0***. Because ***vertical axis*** is labeled by ***0***.

➢ For a ***horizontal*** concatenation (columns), use ***axis = 1***,

➢ For ***vertical*** concatenation use ***axis = 0***.

```
# getting the Predicted Google stock price of 2017-January from our trained model
dataset_total = pd.concat((dataset_train['Open'], dataset_test['Open']), axis = 0)
inputs_for_predict = dataset_total[len(dataset_total)-len(dataset_test)-60 : ].values
# values makes it array, otherwise it is just a series
# it is 60 + 20 = 80 stock-prices, combining october, november, December 2016 and january 2017
print(len(inputs_for_predict))
```

⇨ To get the inputs, we need to specify the index. Since we need ***60*** from ***test-set*** and we get ***20*** from ***train-set***, our input size will be ***80***.

```
inputs_for_predict = dataset_total[len(dataset_total)-len(dataset_test)-60 : ].values
```

inputs_for_prd	Array of float64	(80, 1)	[[779. [779.6
inputs_for_predict	Array of float64	(80,)	[779.

- Here `len(dataset_test)+60 = 80`. Thus we starting from the index `len(dataset_total)- 80`, to get last 80 data-points.
- Because to predict each financial day of ***January 2017***, we need to get the 60 previous stock prices from ***October, November, December*** of ***2016*** (the 60 previous financial days).
- That's why we fix the ***lower-bound*** as: `len(dataset_total) - len(dataset_test)-60`. And there is no ***upper-bound*** because we wand all values to the last of the ***January 2017***.
- ***.values*** is used to convert the selected data-points into ***NumPy Arrays***.

⇨ Since we haven't used the ***iloc*** method from **Pandas** to get these inputs, and therefore, it is not still shaped the right way. It is not properly ***shaped*** like a ***NumPy Array***. So we need to use a ***simple reshape*** of the ***inputs_for_predict*** object (not the ***NumPy***-`reshape` to convert 3D tensor) just to format the data:

```
# simple reshape for the input: to make it a vector, similar job as 'iloc'
inputs_for_prd = inputs_for_predict.reshape(-1, 1)
```

☝ Next we use NumPy's ***reshape*** to create ***3D structure*** of our ***inputs*** (observations, time_steps, indicators). We'll just copy what we've done before and make the proper changes.

☞ ***Scaling the prediction-inputs:*** Before we reshape our input to 3D format to put in RNN we must scale those inputs. We do not use ***fit_transform()*** because we did it already with training set ***train_set_scled***. And we just need to use ***transform()*** to bring input ***inputs_for_predict*** same scale of ***train_set_scled***.

```
inputs_for_prd = sc.transform(inputs_for_prd) # scaling, only "transform" , no "fit"
```

➲ Since the RNN was trained on the scaled values. we need to scale the inputs.

➲ So we're ***not gonna use*** the ***fit_transform()*** because our sc object was already fitted to the training set ***train_set_scled***. I'm directly gonna use the ***transform()*** method because the scaling we need to apply to our input must be the ***same scaling*** that was applied to the ***training set***.

➲ Therefore we must ***not fit*** our *scaling object* ***sc*** again. We must directly apply the ***transform*** method to get the ***previous scaling*** on which our ***regressor*** was ***trained***.

The ***Data Pre-Processing*** for ***Prediction*** is given below:

```
# getting the Real Google stock price of 2017-January from Google_Stock_Price_Test.csv
dataset_test = pd.read_csv('Google_Stock_Price_Test.csv')
real_stock_price = dataset_test.iloc[:, 1:2].values

# getting the Predicted Google stock price of 2017-January from our trained model
dataset_total = pd.concat((dataset_train['Open'], dataset_test['Open']), axis = 0)
inputs_for_predict = dataset_total[len(dataset_total)-len(dataset_test)-60 : ].values
# values makes it array, otherwise it is just a series
# it is 60 + 20 = 80 stock-prices, combining october, november, December 2016 and january 2017
print(len(inputs_for_predict))

# simple reshape for the input: to make it a vector, similar job as 'iloc'
inputs_for_prd = inputs_for_predict.reshape(-1, 1)
inputs_for_prd = sc.transform(inputs_for_prd) # scaling, only "transform" , no "fit"
```

❒ ***Data-Preprocess for test-set, creating 3D data-structure:*** Here we prepare the special 3D structure of ***inputs_for_prd*** that expected by the NN for the training, but also for the predictions. It is similar as we did before:

```
# creating the data structure for test-set
test_data_size = inputs_for_prd.shape[0]
X_test = []
# There is no "y_test = []" we'll predict it
```

```
for i in range(60, test_data_size):
    X_test.append(inputs_for_prd[(i-60):i, 0])

# Convert X_test to NumPy array
X_test = np.array(X_test)
```

☞ We don't need "**`y_test = []`**", we'll predict it. Instead of ***`train_set_scled`*** we use our input for test ***`inputs_for_prd`***.

☞ Then we convert it to ***NumPy*** array.

☞ ***Reshaping:*** It is exactly same as we did for ***`X_train`***, but here we do it for ***`X_test`***.

```
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
```

❒ ***Getting the Predicted Google stock price of January 2017:*** The predicted values are returned in Scaled format, so we need to inverse-scale it to get the real values

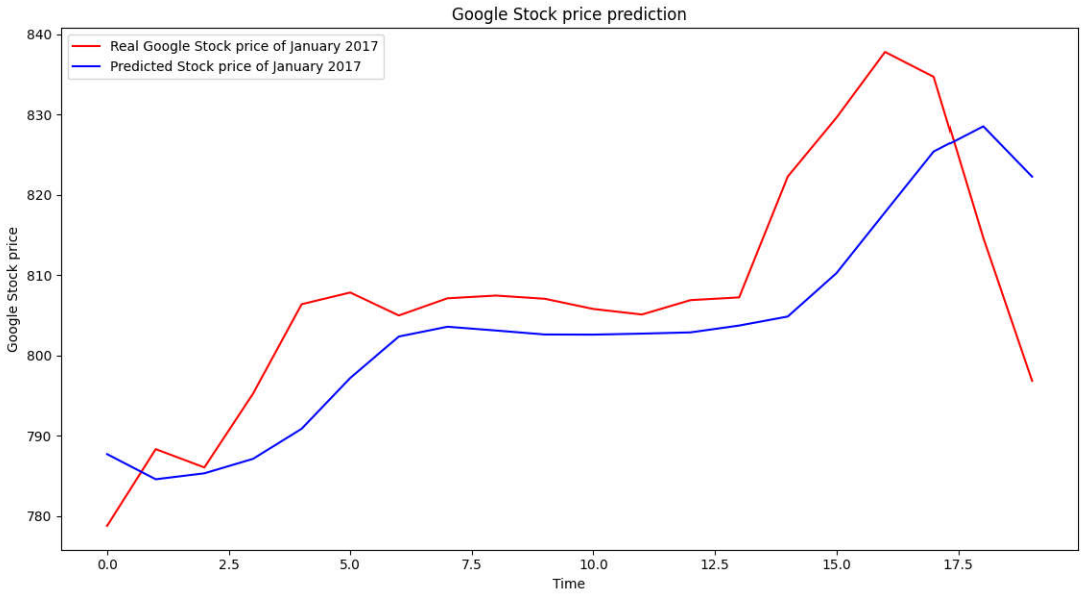
```
# ---- Making Prediction !!! -----
y_pred = regressor_rnn.predict(X_test)
predicted_stock_price = sc.inverse_transform(y_pred)
```

12.3.10 Visualizing the result

We are just going to use the ***`plot`*** function to separately plot the ***real Google stock price*** and then the ***predicted Google stock price***. We will give a ***title*** and different ***colors*** and a ***label*** for the ***legend***, and a title to our chart and some labels for the x- axis and the y- axis.

☝ Keep in mind that we're ***plotting*** the ***first month*** of ***January 2017***, and the ***predictions*** of ***January 2017***.

```
plt.plot(real_stock_price, color = 'red', label = 'Real Google Stock price of January 2017')
plt.plot(predicted_stock_price, color = 'blue', label = 'Predicted Stock price of January 2017')
plt.title('Google Stock price prediction')
plt.xlabel('Time')
plt.ylabel('Google Stock price')
plt.legend()
plt.show()
```



☞ We have the real ***Google stock price*** in ***red*** and our ***predicted Google stock price*** in ***blue***. And we get this comparison of the real and the predicted Google stock prices for the whole month of January 2017.

❒ We see a big **spike**, like a **stock time singularity** at last week of **January**, and our predictions did not follow that, but that is completely normal. (There is another spike around the first week also.)

☞ Our model just lags behind because it cannot ***react*** so ***fast***, these kind of ***nonlinear changes***.

➢ That's totally fine because, indeed, according to the ***Brownian Motion*** Mathematical Concept in ***financial engineering***, the ***future variations*** of the ***stock price*** are ***independent*** from the ***past***.

➢ And therefore, this ***future variation*** that we see here around the ***spike***, well, is a ***variation*** that is indeed totally ***independent*** from the ***previous stock prices***.

☞ The good news is that our ***RNN model*** reacts okay to ***smooth changes***.

➢ For the parts of the predictions ***containing smooth changes***, our model reacts ***pretty well*** and manages to follow the ***upward*** and ***downward trends***.

➢ It manages to follow the ***upward trend***, then the ***stable trend***, and again, the ***upward trend***. Then, there is a ***downward trend*** in the ***last*** financial days of ***January***, and it started to capture it.

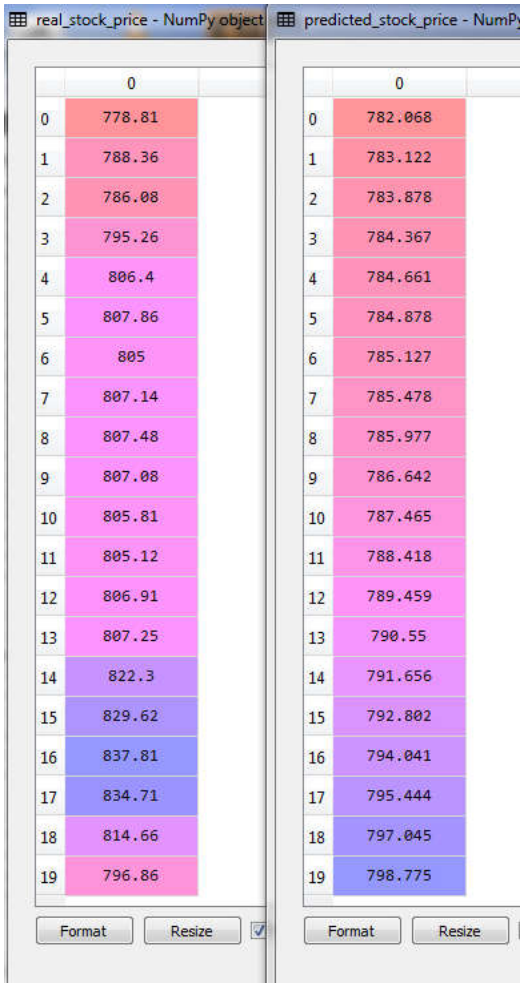
What is Next?

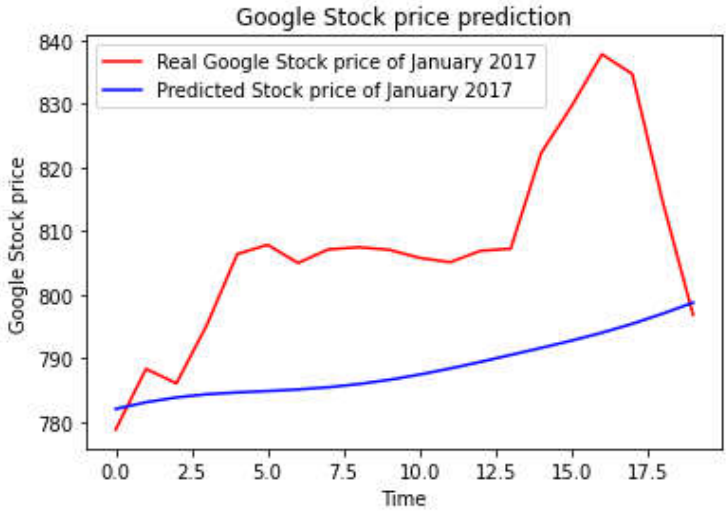
In the next chapters of ***unsupervised deep learning***, we are going to start to play with one of the most powerful tools for deep learning and artificial intelligence, the **`Pytorch`**,

`Pytorch` is much more powerful than ***Keras***, thanks to the ***dynamic graphs***.

With these ***dynamic graphs***, we will be able to build some very ***powerful, unsupervised, deep learning models*** especially the ***Boltzmann machines*** and the ***Auto-encoders*** which will implement, build two different recommended systems. One that will ***predict*** the **`ratings`** given by the users to movies, and one other that will ***predict*** if a user will **`like`** yes or no, a movie.

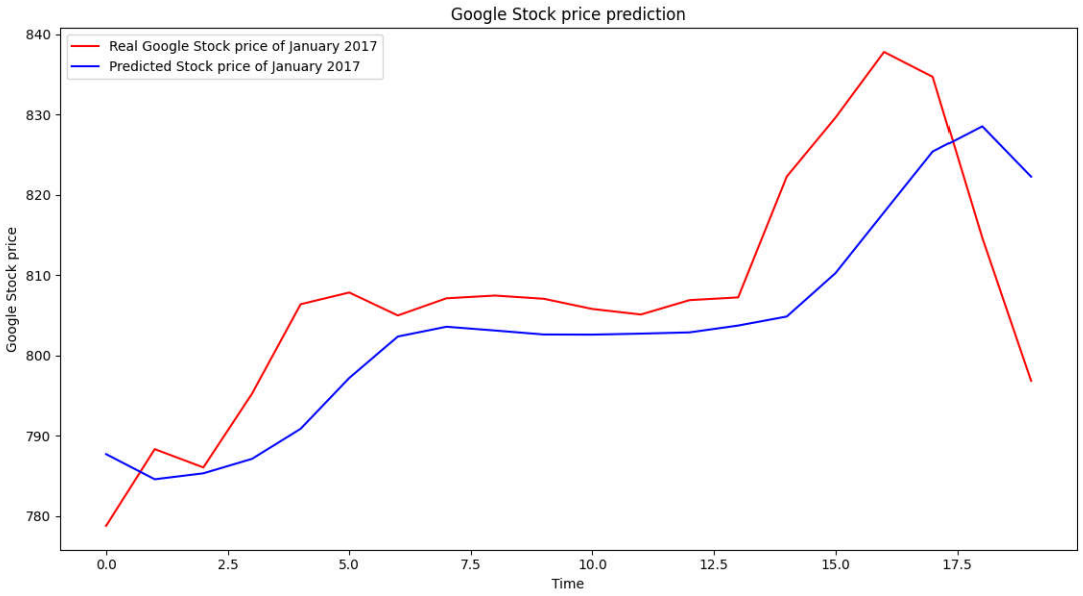
Using 10 epochs only.





Using 100 Epochs

	----- ***Real Stock prices***---------	----- ***Predicted Stock prices***--------
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags. Epoch 1/100 38/38 [==============================] - 7s 60ms/step - loss: 0.0360 Epoch 2/100 38/38 [==============================] - 2s 61ms/step - loss: 0.0062 Epoch 3/100 38/38 [==============================] - 2s 61ms/step - loss: 0.0052 Epoch 4/100 38/38 [==============================] - 2s 60ms/step - loss: 0.0054 .... Epoch 97/100 38/38 [==============================] - 2s 61ms/step - loss: 0.0015 Epoch 98/100 38/38 [==============================] - 2s 61ms/step - loss: 0.0013 Epoch 99/100 38/38 [==============================] - 2s 61ms/step - loss: 0.0012 Epoch 100/100 38/38 [==============================] - 2s 62ms/step - loss: 0.0014 1/1 [==============================] - 1s 1s/step	[[778.81] [788.36] [786.08] [795.26] [806.4] [807.86] [805.] [807.14] [807.48] [807.08] [805.81] [805.12] [806.91] [807.25] [822.3] [829.62] [837.81] [834.71] [814.66] [796.86]] ------ ---------	[[787.7355] [784.6061] [785.3503] [787.1467] [790.8928] [797.22] [802.37866] [803.59607] [803.12555] [802.6338] [802.61597] [802.74005] [802.89484] [803.7443] [804.8699] [810.2623] [817.8498] [825.4131] [828.5456] [822.2784]] ------ ---------



All RNN code at once (practiced version)

```
# Recurrent Neural Network: RNN

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# ------------------- Data Preprocessing ----------------------
dataset_train = pd.read_csv('Google_Stock_Price_Train.csv')      # Notice train-set is now our main dataset
training_set = dataset_train.iloc[:, 1:2].values

# feature scaling
from sklearn.preprocessing import MinMaxScaler
```

```
print(f"Size of train set : {train_set_scled.shape}")
print(f"No. of data-points : {train_set_scled.shape[0]}")


# creating a data structure with 60 time-steps and 1 output.
data_size = train_set_scled.shape[0]
X_train = []
y_train = []

for i in range(60, data_size):
    X_train.append(train_set_scled[(i-60):i, 0])     # append a list of 60 data-points in list of (data_size-60) lists
    y_train.append(train_set_scled[i, 0]) # vector of (data_size-60) rows

# Convert X_train and y_train to NumPy array
X_train, y_train = np.array(X_train), np.array(y_train)

# Reshaping
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))


# ------------------- Building RNN ---------------------

# importing keras libraries and packages
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout

# Initialize the RNN
regressor_rnn = Sequential()

# adding first LSTM layer & some Dropout-Regularization
regressor_rnn.add(LSTM(units= 50, return_sequences= True, input_shape = (X_train.shape[1], 1)))
regressor_rnn.add(Dropout(rate = 0.2))

# second LSTM layer with Dropout-Regularization
regressor_rnn.add(LSTM(units= 50, return_sequences= True))
regressor_rnn.add(Dropout(rate = 0.2))

# third LSTM layer with Dropout-Regularization
regressor_rnn.add(LSTM(units= 50, return_sequences= True))
regressor_rnn.add(Dropout(rate = 0.2))

# forth LSTM layer (last LSTM layer) with Dropout-Regularization
regressor_rnn.add(LSTM(units= 50))
regressor_rnn.add(Dropout(rate = 0.2))

# the output Layer
regressor_rnn.add(Dense(units = 1))


# ------ Compiling the Model ----------
regressor_rnn.compile(optimizer='adam', loss='mean_squared_error')


# ------- Fit the model to Training dataset (model learns) ----------
# The training will happen in this part.
regressor_rnn.fit(X_train, y_train, epochs= 100, batch_size=32)



# ------------------- Making Prediction ---------------------
# getting the Real Google stock price of 2017-January from Google_Stock_Price_Test.csv
dataset_test = pd.read_csv('Google_Stock_Price_Test.csv')
real_stock_price = dataset_test.iloc[:, 1:2].values

# getting the Predicted Google stock price of 2017-January from our trained model
dataset_total = pd.concat((dataset_train['Open'], dataset_test['Open']), axis = 0)
inputs_for_predict = dataset_total[len(dataset_total)-len(dataset_test)-60 : ].values
# values makes it array, otherwise it is just a series
# it is 60 + 20 = 80 stock-prices, combining october, november, December 2016 and january 2017
print(len(inputs_for_predict))

# simple reshape for the input: to make it a vector, similar job as 'iloc'
inputs_for_prd = inputs_for_predict.reshape(-1, 1)
```

```python
inputs_for_prd = sc.transform(inputs_for_prd) # scaling, only "transform" , no "fit"


# creating the data structure for test-set
test_data_size = inputs_for_prd.shape[0]
X_test = []
# There is no "y_test = []" we'll predict it

for i in range(60, test_data_size):
    X_test.append(inputs_for_prd[(i-60):i, 0])

# Convert X_test to NumPy array
X_test = np.array(X_test)

# Reshaping
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))


# ---- Making Prediction !!! -----
y_pred = regressor_rnn.predict(X_test)
predicted_stock_price = sc.inverse_transform(y_pred)


# ------------------- Visualizing the Result ---------------------
print(f"\n------ Real Stock prices--------- \n ")
print(f"{real_stock_price}")
print(f"\n------ --------- \n ")
print(f"\n------ Predicted Stock prices--------- \n ")
print(f"{predicted_stock_price}")
print(f"\n------ --------- \n ")


plt.plot(real_stock_price, color = 'red', label = 'Real Google Stock price of January 2017')
plt.plot(predicted_stock_price, color = 'blue', label = 'Predicted Stock price of January 2017')
plt.title('Google Stock price prediction')
plt.xlabel('Time')
plt.ylabel('Google Stock price')
plt.legend()
plt.show()



# python prctc_rnn.py
```

Deep Learning

RNN: Evaluating, Improving and Tuning the RNN

`Evaluation & Performance`

12.4.1 RMSE for to evaluate the model performance

In the previous sections, the ***RNN*** we built was a ***regressor***. Indeed, we were dealing with ***Regression*** because we were trying to ***predict*** a ***continuous outcome*** (the Google Stock Price). For ***Regression***, the way to ***evaluate*** the ***model performance*** is with a ***metric*** called ***RMSE (Root Mean Squared Error)***. It is calculated as the ***root*** of the ***mean*** of the ***squared differences*** between the ***predictions*** and the ***real values***.

❐ ***RMSE doesn't help here:*** However for our specific Stock Price Prediction problem, evaluating the model with the ***RMSE*** does **not make much sense**, since we are more ***interested*** in the ***directions*** taken by our ***predictions***, rather than the ***closeness*** of their ***values*** to the ***real stock price***. We want to ***check*** if our ***predictions follow*** the ***same directions*** as the ***real stock price*** and *we don't really care whether our predictions are close the real stock price*. The ***predictions*** could indeed be ***close*** but often taking the ***opposite direction*** from the ***real stock price***.

Nevertheless if you are interested in the code that computes the ***RMSE*** for our ***Stock Price Prediction problem***, please find it just below:

```
import math
from sklearn.metrics import mean_squared_error
rmse = math.sqrt(mean_squared_error(real_stock_price, predicted_stock_price))
```

☞ Then consider ***dividing*** this ***RMSE*** by the ***range*** of the ***Google Stock Price*** values of ***January 2017 (that is around 800)*** to get a ***relative error***, as opposed to an ***absolute error***.

☞ It is more relevant since for example if you get an ***RMSE*** of ***50***, then this ***error*** would be ***very big*** if the ***stock price values*** ranged ***around 100***, but it would be ***very small*** if the stock price values ranged around ***10000***.

12.4.2 Different ways to improve RNN model

[1]. ***Getting more Training Data:*** we trained our model on the past ***5 years*** of the ***Google Stock Price*** but it would be even better to train it on the ***past 10 years***.

[2]. ***Increasing the number of Timesteps:*** the model remembered the stock prices from the ***60 previous financial days*** to predict the stock price of the next day. That's because we chose a number of ***60 timesteps (3 months)***. You could try to ***increase*** the number of ***timesteps***, by choosing for example ***120 timesteps (6 months)***.

[3]. ***Adding some other Indicators:*** if you have the ***financial*** instinct that the ***stock price*** of some other ***companies*** might be ***correlated*** to the one of ***Google***, you could add this other stock price as a ***new indicator*** in the ***training data***.

[4]. ***Adding more LSTM layers:*** we built a ***RNN*** with ***four LSTM layers*** but you could try with ***even more***.

[5]. ***Adding more Neurones in the LSTM layers:*** we ***highlighted*** the fact that we needed a ***high number of neurones*** in the ***LSTM layers*** to respond better to the complexity of the problem and we chose to include ***50 neurones*** in each of our ***4 LSTM layers***. You could try an ***architecture*** with even ***more neurones*** in each of the ***4 (or more) LSTM layers***.

12.4.3 Parameter tuning to improve RNN model

You can do some ***Parameter Tuning*** on the ***RNN*** model we ***implemented***. Remember, this time we are dealing with a ***Regression problem*** because we predict a ***continuous outcome*** (the Google Stock Price).

❐ ***Parameter Tuning*** for ***Regression*** is the same as ***Parameter Tuning*** for ***Classification*** which you learned in ***ANN*** and ***Chapter 11: Model Selection***, the only difference is that you have to replace:

```
scoring = 'accuracy'
```

by:

```
scoring = 'neg_mean_squared_error'
```

in the ***GridSearchCV*** class ***parameters***.

Chapter 12 : Part 4

Deep Learning

RNN: GRU

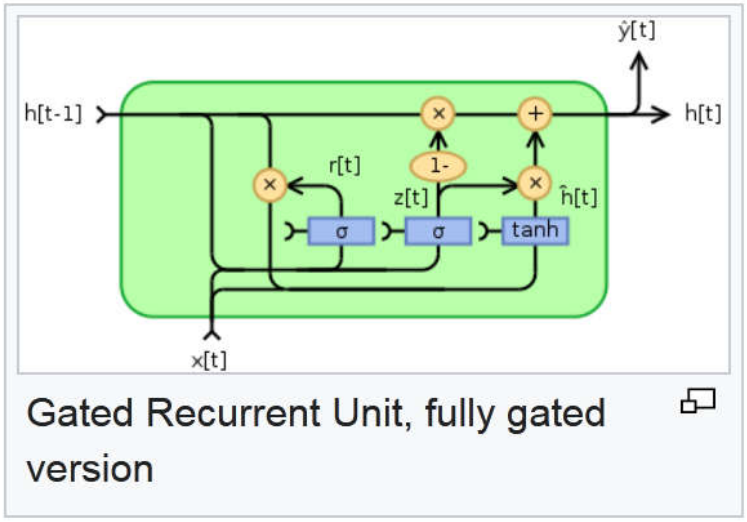
Variation of RNN

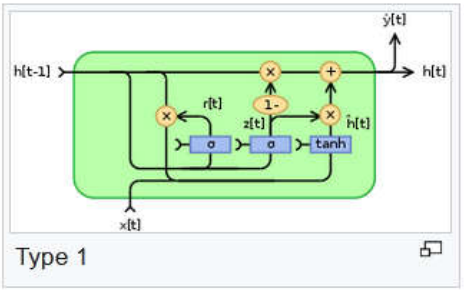
12.4.1 GRU : Gated recurrent units

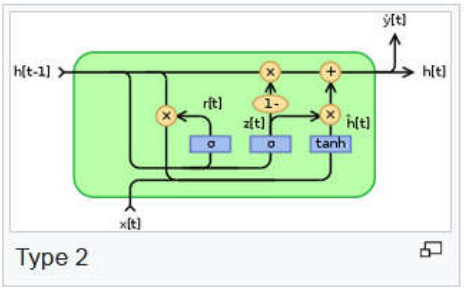
❒ ***Gated recurrent units (GRUs)*** are a ***gating mechanism*** in ***RNN***, introduced in 2014 by Kyunghyun Cho et al. The ***GRU*** is like a ***long short-term memory (LSTM)*** with a ***forget gate***, but has ***fewer parameters*** than ***LSTM***, as it ***lacks*** an ***output gate***.

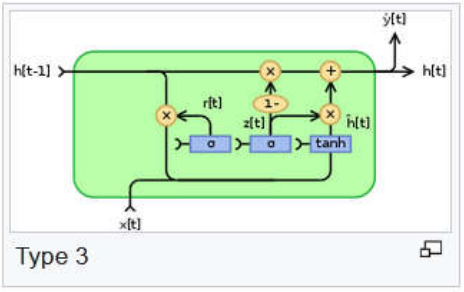
☞ GRU's performance on certain tasks of ***polyphonic music*** modeling, ***speech signal*** modeling and ***natural language processing (NLP)*** was found to be similar to that of ***LSTM***.

☞ ***Better for Smaller dataset:*** GRUs have been shown to exhibit better performance on certain smaller and less frequent datasets.[6][7]







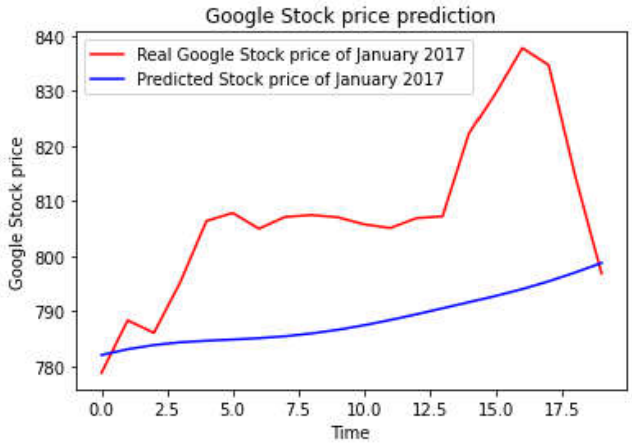
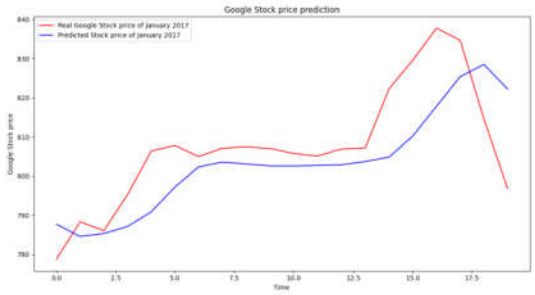
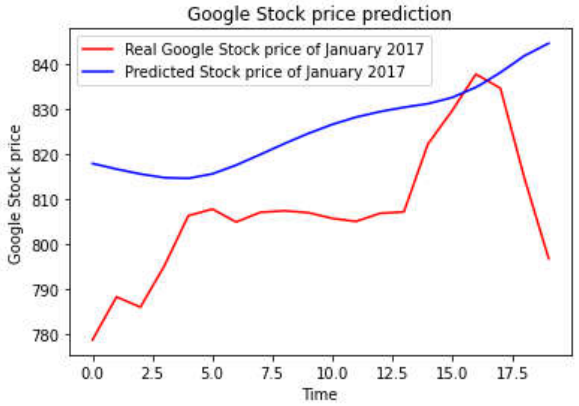
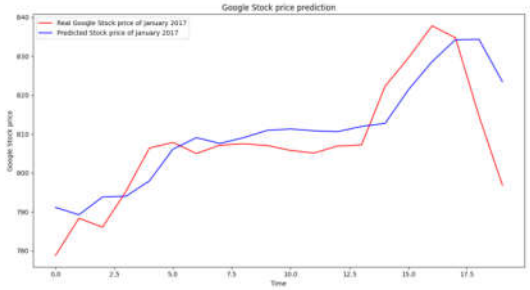
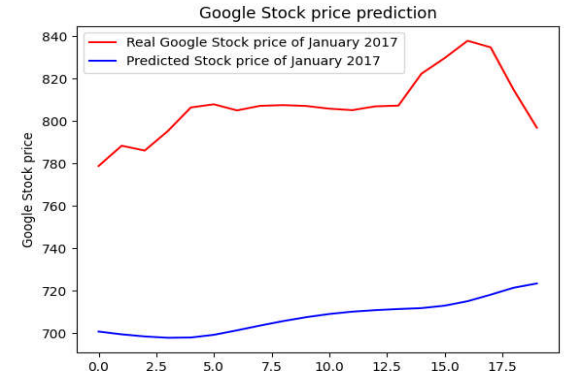
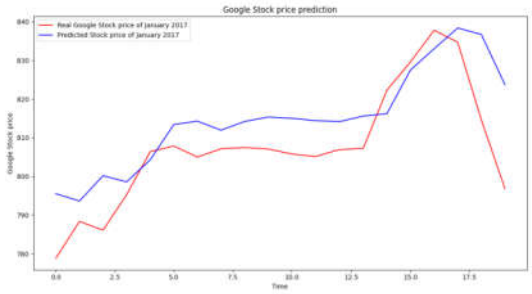


☝ There are several ***variations*** on the ***full gated unit***, with ***gating*** done using the ***previous hidden state*** and the ***bias*** in ***various combinations***, and a simplified form called ***minimal gated unit***.

☞ The code is Exactly same, we just replace LSTM by GRU.

	------ Real Stock prices---------	------ Predicted Stock prices---------
100 epochs GRU, ADAM,		
Epoch 1/100		
38/38 [==============================] - 8s 65ms/step - loss: 0.0390	[[778.81]	[[791.1578]
Epoch 2/100	[788.36]	[789.262]
38/38 [==============================] - 2s 65ms/step - loss: 0.0054	[786.08]	[793.8223]
Epoch 3/100	[795.26]	[794.02014]
38/38 [==============================] - 2s 65ms/step - loss: 0.0053	[806.4]	[797.9758]
Epoch 4/100	[807.86]	[806.1135]
38/38 [==============================] - 2s 65ms/step - loss: 0.0049	[805.]	[809.07086]
---------- ---------- ----------	[807.14]	[807.58527]
Epoch 97/100	[807.48]	[809.06055]
38/38 [==============================] - 3s 66ms/step - loss: 0.0013	[807.08]	[810.9257]
Epoch 98/100	[805.81]	[811.3189]
38/38 [==============================] - 3s 66ms/step - loss: 0.0013	[805.12]	[810.8287]
Epoch 99/100	[806.91]	[810.64606]
38/38 [==============================] - 2s 66ms/step - loss: 0.0015	[807.25]	[811.96246]
Epoch 100/100	[822.3]	[812.74835]
38/38 [==============================] - 2s 65ms/step - loss: 0.0014	[829.62]	[821.4289]
	[837.81]	[828.5962]
	[834.71]	[834.26086]
	[814.66]	[834.3416]
	[796.86]]	[823.4504]]

Epoch 1/100 38/38 [==========] - 8s 64ms/step - loss: 0.0242 Epoch 2/100 38/38 [==========] - 2s 64ms/step - loss: 0.0100 Epoch 3/100 38/38 [==========] - 2s 64ms/step - loss: 0.0081 Epoch 4/100 38/38 [==========] - 2s 64ms/step - loss: 0.0084 ---------- ---------- ---------- Epoch 96/100 38/38 [==========] - 2s 64ms/step - loss: 0.0017 Epoch 97/100 38/38 [==========] - 2s 64ms/step - loss: 0.0017 Epoch 98/100 38/38 [==========] - 2s 65ms/step - loss: 0.0017 Epoch 99/100 38/38 [==========] - 2s 65ms/step - loss: 0.0016 Epoch 100/100 38/38 [==========] - 2s 64ms/step - loss: 0.0018	------ Real Stock prices--------- [[778.81] [788.36] [786.08] [795.26] [806.4] [807.86] [805.] [807.14] [807.48] [807.08] [805.81] [805.12] [806.91] [807.25] [822.3] [829.62] [837.81] [834.71] [814.66] [796.86]]	------ Predicted Stock prices--------- [[795.52484] [793.6181] [800.1723] [798.59094] [804.2728] [813.4076] [814.29333] [811.95935] [814.19464] [815.3559] [815.036] [814.4354] [814.17615] [815.6241] [816.2212] [827.5328] [833.02094] [838.3989] [836.74066] [823.7365]]

10 epochs	*100 epochs*
Adam LSTM	***Adam LSTM***
	
Adam GRU	***Adam GRU***
	
Rmsprop GRU	***Rmsprop GRU***
	

Chapter 13 : Part 1

Deep Learning

SOM: Self-Organizing Maps

`Introduction`

13.1.1 What we will learn in this Chapter

[1]. ***How do Self-Organizing Maps work?:*** First of all we will talk about how ***self-organizing maps (SOMs)*** work. It will help us understand what to expect, what we're aiming for. We'll know the end goal that we're working towards.

[2]. ***K-Means Clustering:*** Then we'll talk about ***K-Means Clustering***, it will be a review for us what we've done in ML in **Chapter 4: Clustering**.

[3]. ***How do Self-Organizing Maps Learn? (Part 1 & Part 2):*** We'll talk about how do ***self-organizing maps*** learn, in these 2-part. We do this in two parts because we'll dive deep into the topic here.

- ❖ We'll walk through them step by step example for better understanding.

[4]. ***Live SOM example:*** Here we'll have a live SOM example, a very simple one. Here you will see how a SOM ***structures itself*** and preserves ***similarities*** & ***correlations*** in your data set and portrays them in a ***lower dimensionality*** representation (***2-D map***).

[5]. ***Reading an Advanced SOM:*** Finally we will talk about ***reading*** an ***advanced SOM***. This shows you how to read those ***SOMs***.

- ❖ We'll have different maps on one screen and by looking at them you can ***read/understand*** them.
- ❖ We'll discuss some examples of ***map implementations*** to guide you in the direction of where you can do ***further study*** in the space of ***SOMs***.

13.1.2 SOM (Self-Organizing Map)

We already talked about ***ANN***, ***CNN*** and ***RNN*** those are collectively called ***supervised*** deep learning.

❒ However, ***SOM*** is a ***unsupervised*** deep learning method. ***Self-organizing maps (SOM)*** were invented in the 1980s by, ***Teuvo Kohone***.

☞ Sometimes ***SOMs*** even called the ***Kohonen maps***.



❒ ***Usage of SOMs:***

☞ SOMs are used for ***reducing dimensionality***.

☞ SOMs can be used in ***astronomy***. Here's a great example from the paper.

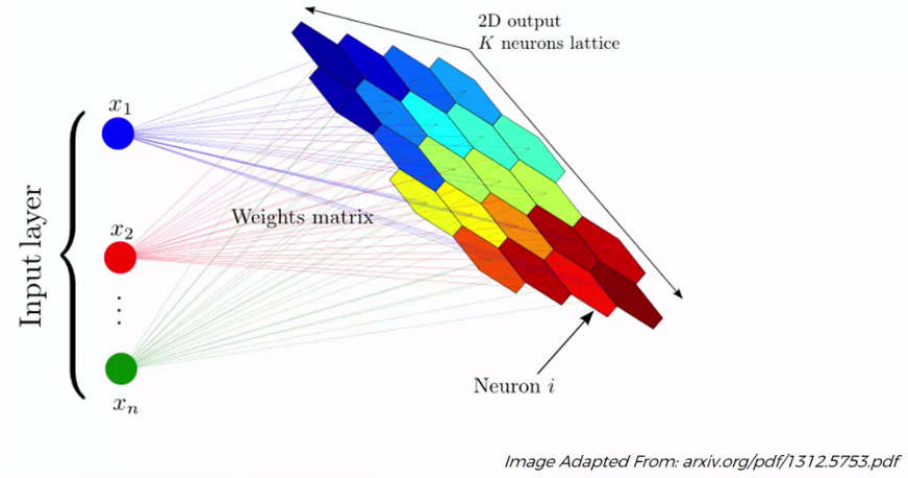


Image Adapted From: arxiv.org/pdf/1312.5753.pdf

Here we got a beautiful visualization of how ***self-organizing maps*** actually work.

❒ ***SOMs*** take a ***multi-dimensional data set*** (with ***lots*** of ***columns*** which are the ***dimensions*** of the data set, and ***lots*** of ***rows***) and they ***reduce*** the ***dimensionality*** of these ***data sets***.

☞ So basically, instead of having ***20***, ***30*** or a ***hundred*** or even more ***columns*** (20, 30 or 100 dimension), you end up with a ***map***.

☞ That's why they called ***Self-Organizing Maps (SOMs)*** because we end up with a ***two-dimensional representation*** of your ***data set***.

☞ The purpose of the SOM is to ***reduce*** the amount of ***columns***. And represent the data into ***2D-map***.

Example: Here is an actual SOM. which was produced, from the ***data*** of the different ***states*** of ***prosperity*** and ***poverty*** in different ***countries***.

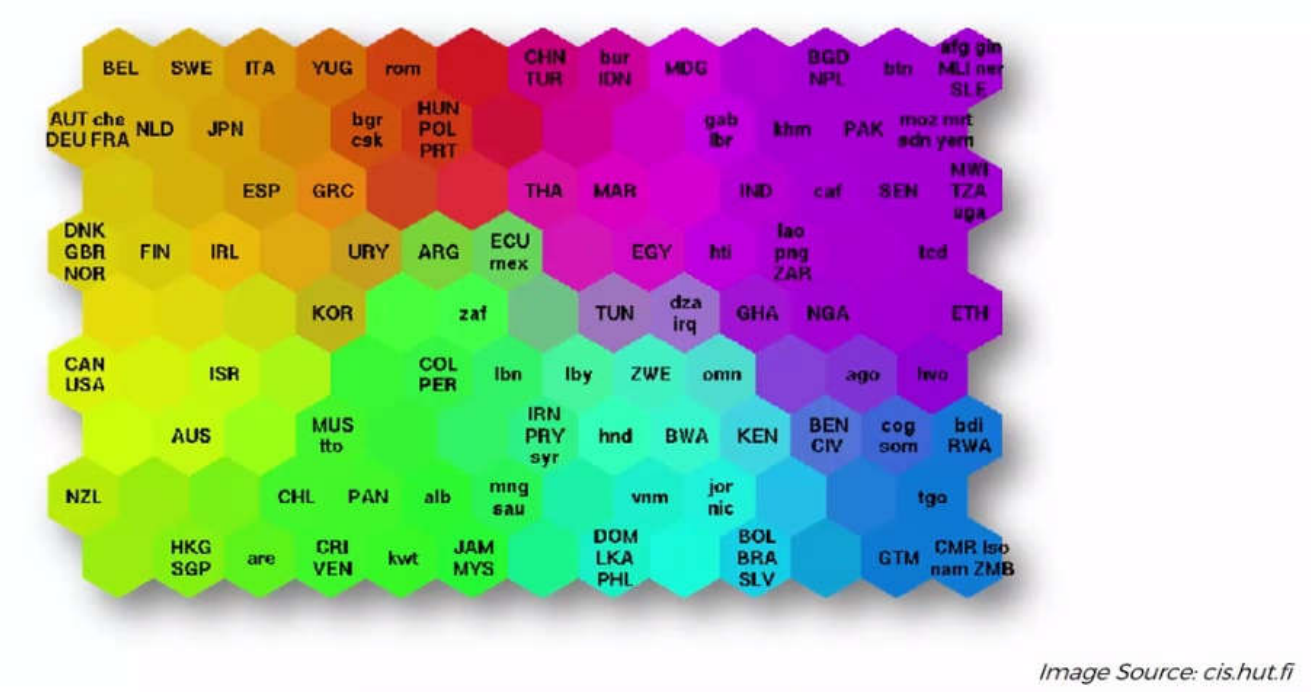


Image Source: cis.hut.fi

☯ Those names actually represents ***countries*** of the world and this ***SOM*** has put them into ***clusters*** based on lots of ***different indicators***.

☯ In this specific example, **39** different **indicators** were used. And **indicators** are **parameters** describing things such as ***quality of life***, ***factors***, the ***state of health*** in a country, ***nutrition***, ***educational services***, and so on.

☯ We can see that in the top left corner, we have ***countries*** with the ***best*** or the ***least alarming*** state of ***poverty***. Those countries are ***Belgium***, ***Sweden***, ***Japan***, ***Spain***.

☯ We also notice that it's slowly going towards the other end of spectrum where you have countries with the most ***alarming*** state of ***poverty***, like ***Ethiopia*** or ***Zimbabwe***.

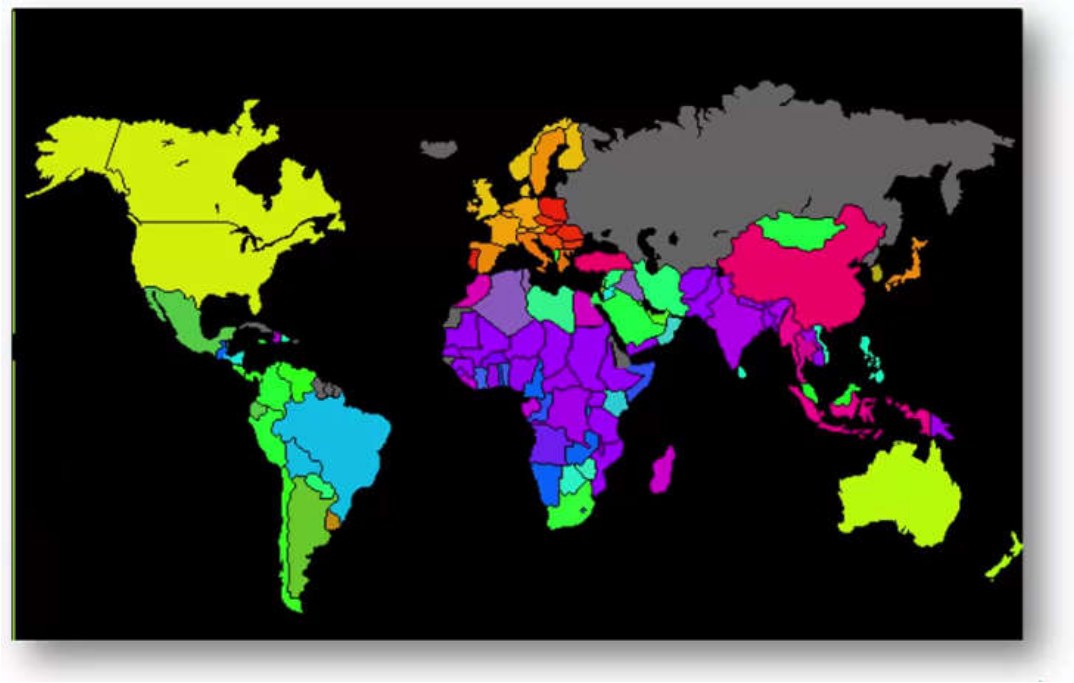
☞ So if we have a huge data set containing 200 plus ***countries*** as ***rows***, and 39 different columns (dimensions. Indicators in SOM). So it's impossible to visualize.

☞ But using a ***SOM***, we can ***reduce*** the ***dimensionality*** and present it as a ***map*** like above.

❒ ***Remember that, SOMs are unsupervised techniques:*** It has ***training data*** but it doesn't have any ***labels*** in the ***training data***. So its learning on its own.

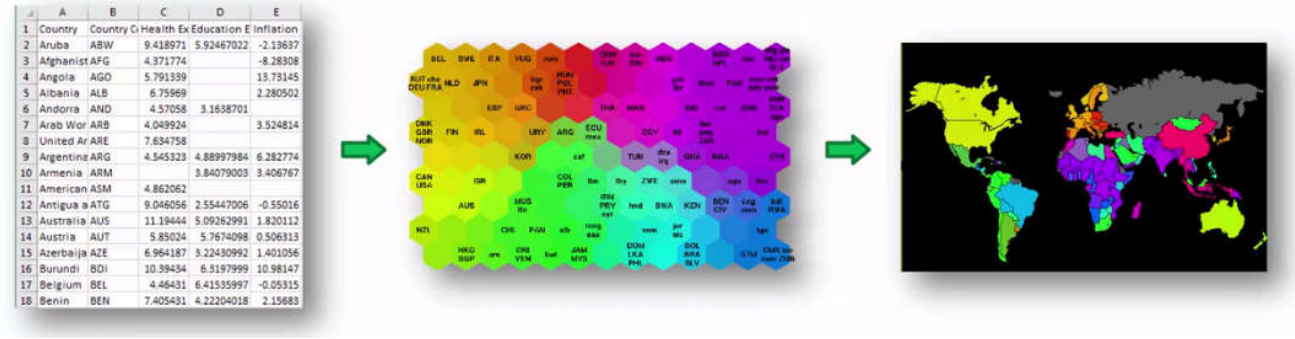
➲ Basically just given data, and then it **learns to *group*** these data (countries). It is much more like ***clustering*** in ***ML***.

❒ ***What else can we do with this map:*** We take the ***color-codes*** for the different ***countries*** and ***color*** them in the ***world map***. In this world map we can determine first world countries and third world countries, and where countries are developed, where countries are still developing.



❒ To summarize, we had the data, which you can get from the ***World Bank*** (you can just download data sets from there),

- ☞ We ***reduce*** the ***columns*** using ***SOM*** and get a ***2D-map*** with different ***countries grouped together*** according their ***color code***. It would help you group your data set. SOM map is still a good representation for the data but we can use the colors in the world map.
- ☞ Next, we can color the countries in world map using SOM map.



	A	B	C	D	E
1	Country	Country Co	Health Ex	Education E	Inflation
2	Aruba	ABW	9.418971	5.92467022	-2.13637
3	Afghanist	AFG	4.371774		-8.28308
4	Angola	AGO	5.791339		13.73145
5	Albania	ALB	6.75969		2.280502
6	Andorra	AND	4.57058	3.1638701	
7	Arab Wor	ARB	4.049924		3.524814
8	United Ar	ARE	7.634758		
9	Argentina	ARG	4.545323	4.88997984	6.282774
10	Armenia	ARM		3.84079003	3.406767
11	American	ASM	4.862062		
12	Antigua a	ATG	9.046056	2.55447006	-0.55016
13	Australia	AUS	11.19444	5.09262991	1.820112
14	Austria	AUT	5.85024	5.7674098	0.506313
15	Azerbaija	AZE	6.964187	3.22430992	1.401056
16	Burundi	BDI	10.39434	6.3197999	10.98147
17	Belgium	BEL	4.46431	6.41535997	-0.05315
18	Benin	BEN	7.405431	4.22204018	2.15683

❒ ***SOM can be applied to visualize different kind of data:*** It groups the data, so that you would understand different similarities based on all of your data. You wouldn't have to go through hundreds and hundreds of columns.

- ☞ Grouping different types of ***equipment*** that you might be considering or you might be ***selling*** through your organization.
- ☞ Grouping different types of ***stock*** and ***inventory***.

You would be able to just look at this map and quickly understand all of the similarities.

❒ ***Additional reading:*** If you'd like to get some additional reading a good paper to check out by Teuvo Kohonen from 1990, it's called ***"The Self-Organizing Map"***.

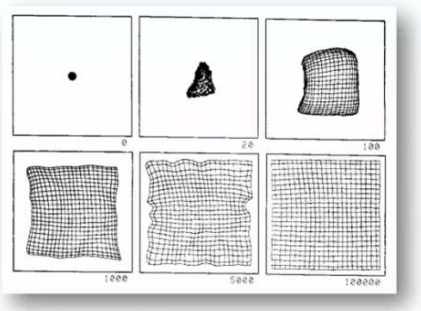
Additional Reading:

The Self-Organizing Map

By Tuevo Kohonen (1990)

Link:

http://sci2s.ugr.es/keel/pdf/algorithm/articulo/1990-Kohonen-PIEEE.pdf



Chapter 13: Part 2

Deep Learning

SOM: Revisit K-means Clustering

`Python Implementation`

13.2.1 Why Revisit K-means Clustering

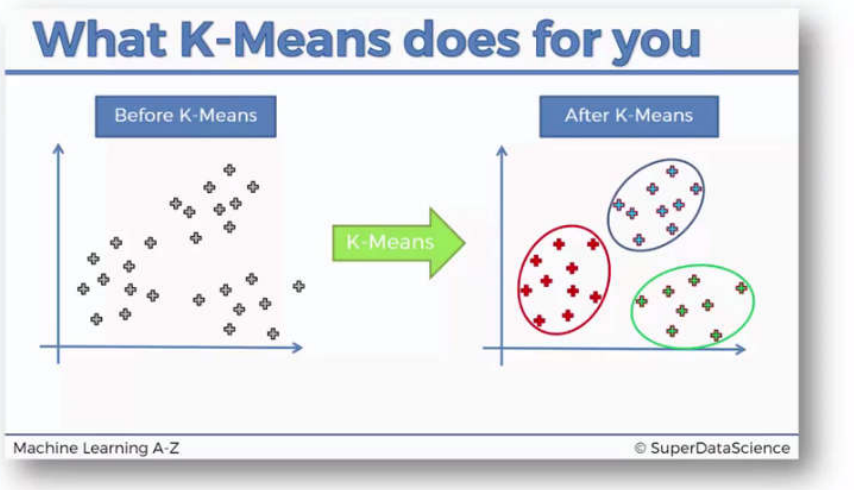
Knowing **K-Means clustering** will be helpful when you want to understand **SOMs**. ***K-Means clustering*** is relevant to ***SOMs***. It's not exactly ***identical***. But it will prepare you for understanding ***SOMs***.

❒ In the next section we will see the process of SOM jumping around, like, ***pushing*** and ***pulling*** is happening as those ***nodes***, or as those ***centroids*** are ***traveling across*** the ***map***.

☞ And how they're being ***pulled*** and ***pushed*** around by the **actual data points**. That kind of similar process that we saw in ***Chapter 4.1: K-means Clustering***.

☞ So revisiting ***Chapter 4.1: K-means Clustering*** will prepare you for the mood of what's going to be happening in SOMs,

❒ Also note that, K-Means clustering is a unsupervised type of algorithm. But it's not a neural network. It's just an unsupervised DL algorithm.



❒ We don't need the python implementation (application). Just revisit the following topics.

4.1.1 K-Means Clustering: What is K-Means Clustering and its Steps.

4.1.2 K-Means Clustering Example

4.1.3 K-Means Random Initialization Trap

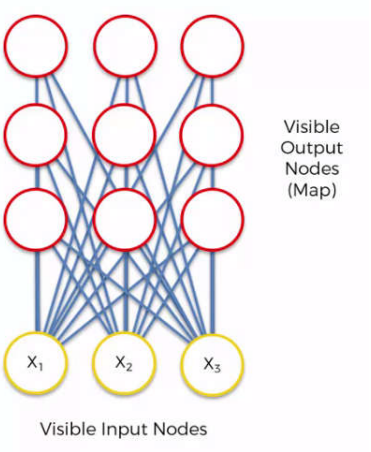
4.1.4 Elbow-Method: Choosing K (Right Number Of Clusters)

Deep Learning

SOM: How it Works

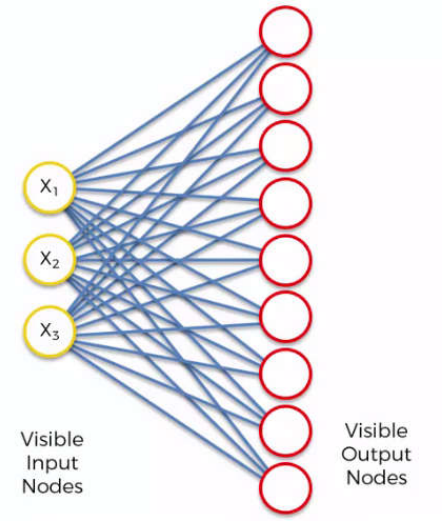
13.3.1 How SOMs work

In this section we want to find out how ***SOMs*** learn.



❒ Here we've got a very simple example of a SOM. We've got **three features (3 columns)** in our **input vector**, and we've got ***nine nodes*** in the output. Here ***each node*** represents a ***data-point*** (a ***single row***).

- ☞ Don't let this representation confuse your understanding of SOM. Here we have ***3 columns of features*** and we might have ***thousands*** of ***rows*** (each row has 3 columns, represents a data-point).
  - ➢ We map these ***rows*** (data-points) in a ***2D map*** and ***Group*** (cluster) them using ***SOM***.
- ☞ It means that our ***input data set*** is actually ***three dimensional*** (we reduce this dimension), whereas our ***output data set*** in a ***SOM*** is always a ***two-dimensional map***, and therefore we are reducing the dimensionality from ***3D*** to ***2D***.

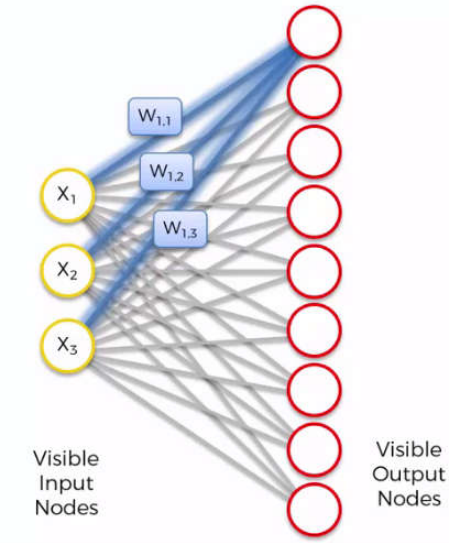


❒ Now we're going to turn this ***SOM*** into an ***Network representation***. Right-hand image shows us what it would look like.

- ☞ It is the ***same network*** (as ***above***), the only difference is how we've ***positioned*** the ***nodes***.
- ☞ We still have the ***same amount*** of ***connections***, ***inputs*** and ***outputs***, it's just the ***visual representation*** has changed so that it would be easier for us to understand what's going on.

❒ Also note that, ***SOMs*** are different than ***NN***.

- ☞ First of all, SOMs are much easier than other NN-supervised-techniques. The whole concept behind them is very simple and straightforward.
- ☞ Secondly it's also important to note that SOMs are different than ANN/CNN/RNN.
  - ➢ The concepts that might have the same names/terms (such as ***weights*** and ***synapses***) have different meanings so don't get confused with ANN/CNN/RNN terms.
  - ➢ Just be careful when we're talking about things like ***weights*** and ***synapses*** and other things, those are different than ***ANN/CNN/RNN***.



❒ Let's consider the ***top node*** of our ***output-nodes***. Notice the ***three synapses*** connecting it to our **three input-nodes**.

- ☞ We saw in ***ANN***, ***weights*** were used to multiply the ***value*** of a node, we ***added them up***, and then we ***applied*** an ***activation function***.
- ☞ ***Weights in SOMs:*** The weights in ***SOMs*** are different and there is ***no activation function***. Here weights are ***characteristics*** of the ***node itself***.
  - ➢ In ***SOM*** the weights are act as ***coordinates***. For example, $W_{1,1}$, $W_{1,2}$ and $W_{1,3}$ as an ***input vector***, in 3D ***input space***.
  - ➢ There are **three *weights*** for those 3 synapses $W_{1,1}$, $W_{1,2}$ and $W_{1,3}$. Now

    First index means that it's the first ***output-node***. ***Second index*** means the ***input nodes***.
  - ➢ Here the first node $(W_{1,1}, W_{1,2}, W_{1,3})$ trying to fit in our ***input space***.

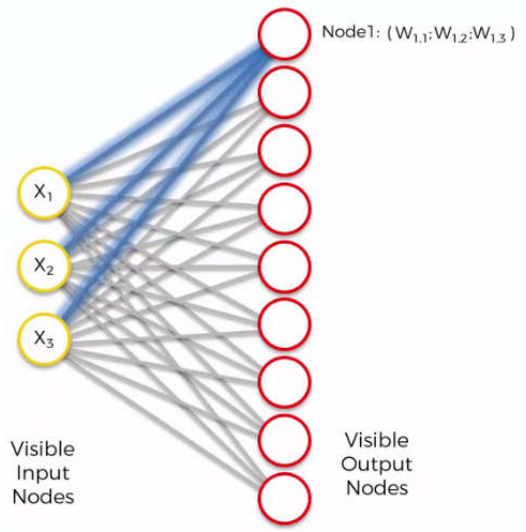
❒ So in ***twenty-dimensional input space***, there are ***20 columns*** in your **inputs**. Then each ***output-node*** would have ***20 weights*** and act as a ***20-dimensional-vector*** in the ***20D-input-space***.

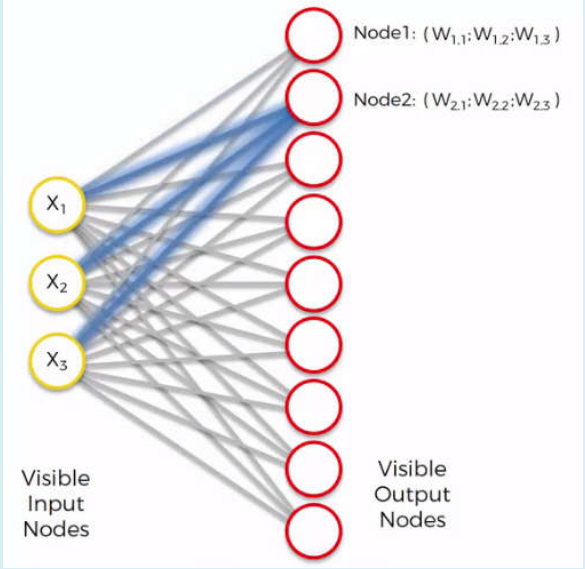
☞ Basically just think of these ***output nodes***, each one of them is a ***imaginary data point*** in our ***input space***.

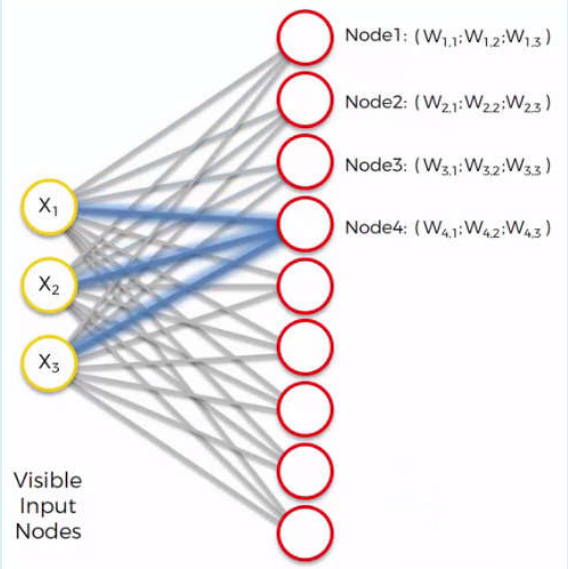
☞ Same thing applied to 2nd , 3rd output-nodes and all other output-nodes.

❒ So, each one of the nodes, in our case **`nine`** (there could be many more), has its ***own weights***. At the start of the algorithm, those weights are randomly selected a small value near ***0***.

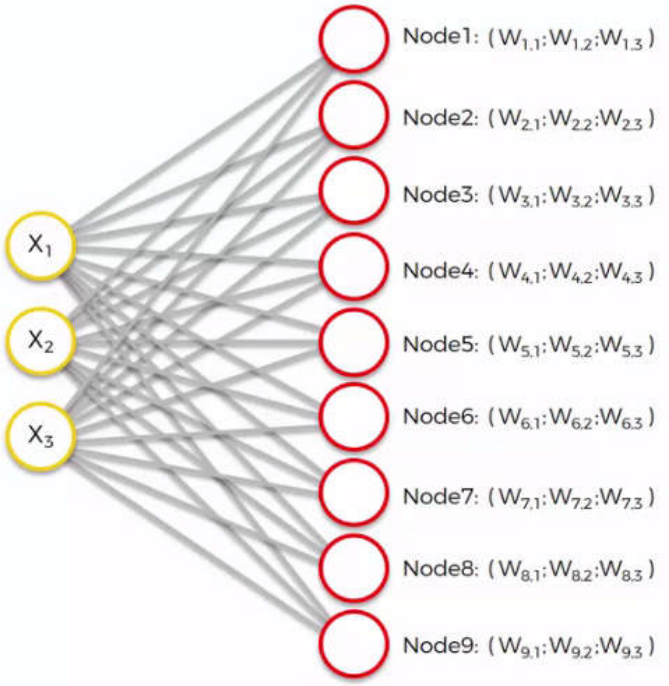
☞ Each one of these nodes has its own imaginary place in the **input space**.







❒ After assigning all weights, finally we get Following.



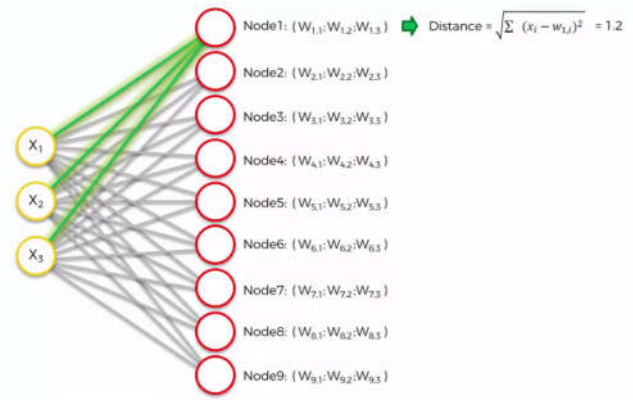
13.3.2 How SOMs organize itself: Distance calculation

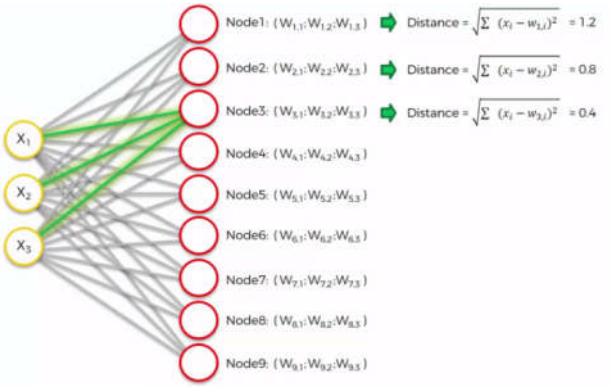
Now we go through each of ***rows*** of our ***data set***, and we're going to find out which of the ***weight assigned*** ***output-nodes*** $(W_{i,1}, W_{i,2}, W_{i,3}), i = 1, 2, 3, \dots, 9$ (from SOM's first iteration) is closest to each of our rows (**inputs**) in our data set.

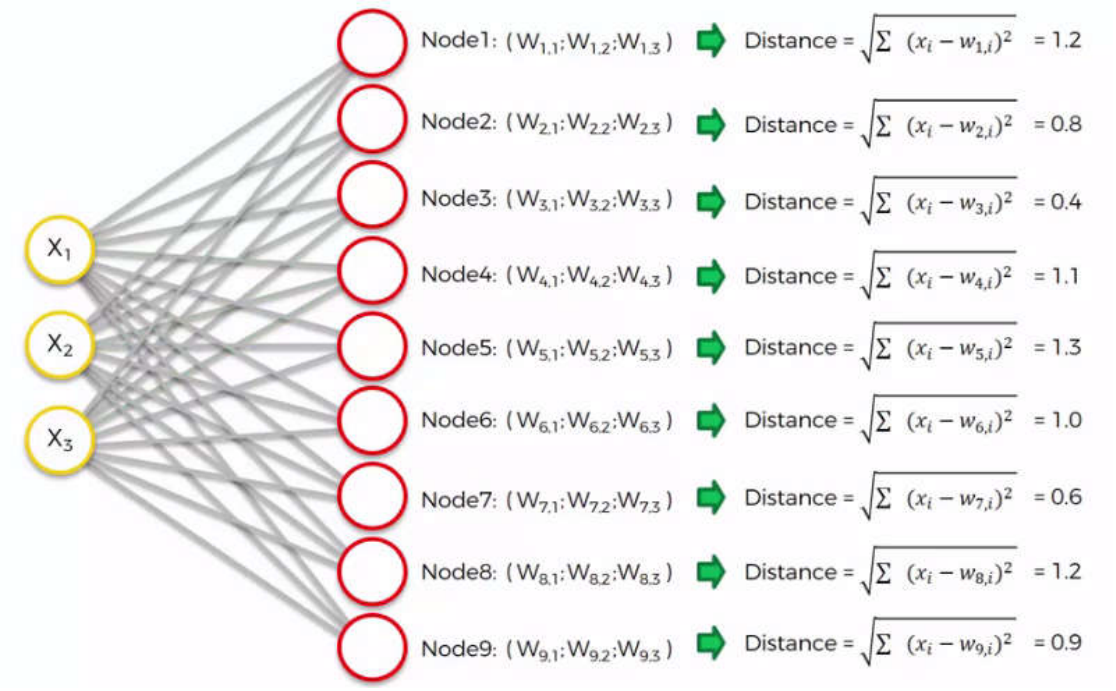
❒ Let's start with row number one:

☞ We first put the ***first row's 3 values*** into our ***3 input nodes***.

☞ After we've inputted ***first row*** from our ***data set*** into our ***input nodes***. We'll go through all 9 output-nodes and calculate the distance from the ***first row*** in ***input nodes***.







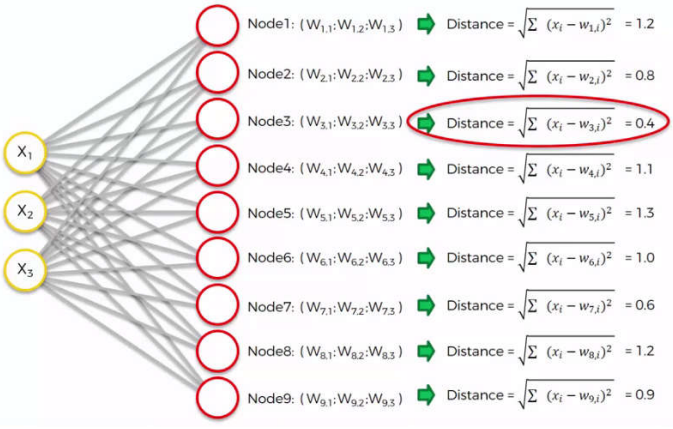
✋ Remember, here (X_1, X_2, X_3) inside ***input-nodes*** is the inputs from ***first-row*** of our ***Data-set***. This (X_1, X_2, X_3) inside ***input-nodes*** does ***not changes*** while we are calculating ***distance*** from SOMs ***output-nodes***. We're not changing the row.

❒ We're going to go through every single one of these nodes, and find out which of these is the ***closest*** in that ***original input space***, which of these nodes is ***closest*** to our ***first-row***.

☞ We calculate the ***distance*** as a ***Euclidean distance***.

✋ ***Feature scaling:*** Also note that we should get a value close to 1. Because our inputs are between 0 and 1, to make this algorithm work properly.

☞ So we need to apply ***normalization*** or ***standardization*** to our data-set. Then we input the ***data*** into the ***SOM***.



❒ After calculating distance for all 9-**output-nodes**, we can see that ***first-row*** or ***first-input*** in our data-set is *three times closer* to ***3rd output-node*** than ***1st output-node***.

☞ ***Best Matching Unit (BMU):*** Now we calculated all of the ***distances*** between ***first-row (first-input)*** and ***9-output-nodes*** then we found that the closest one is ***3rd output-node***.

☞ Here ***3rd output-node*** is the ***BMU***, or the best matching unit.

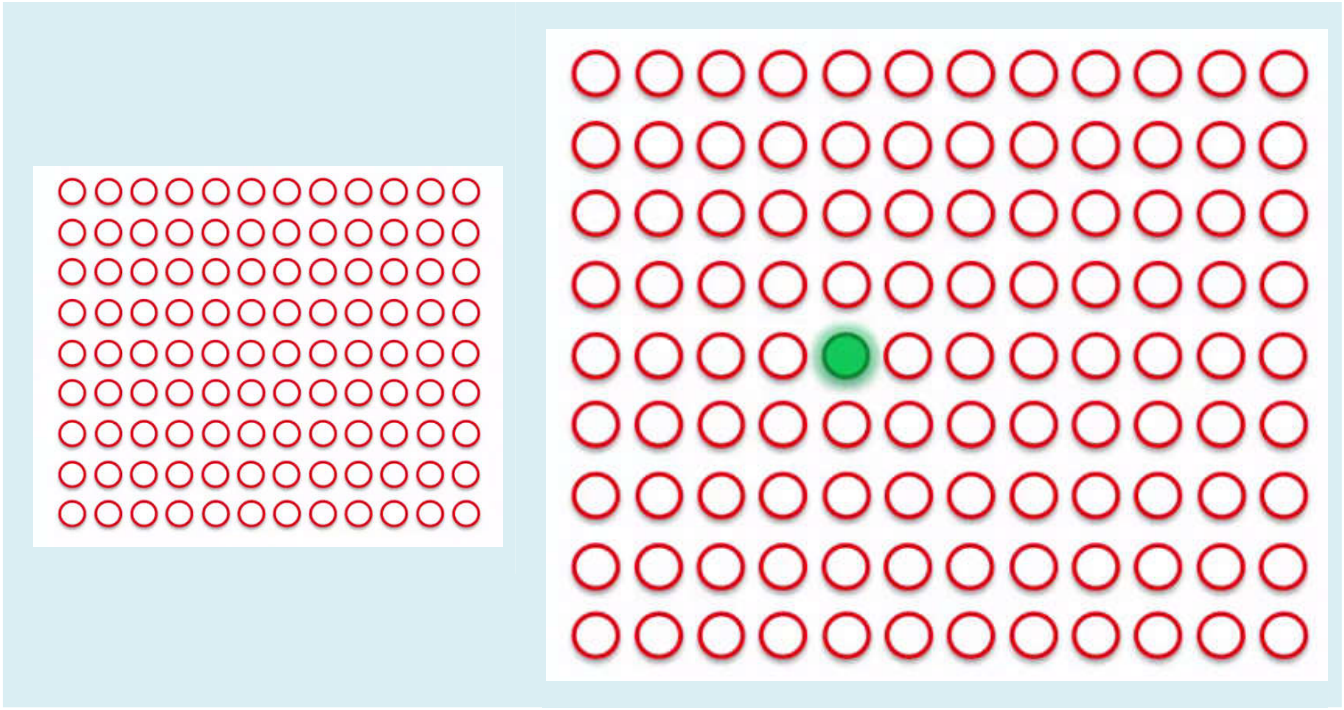
13.3.3 SOM moves toward the Data-points

Let's look at a larger SOM of 9x12 size (i.e. 108 output-nodes).

❒ Let's say in this larger map we found the BMU for first-row (the Green colored point).

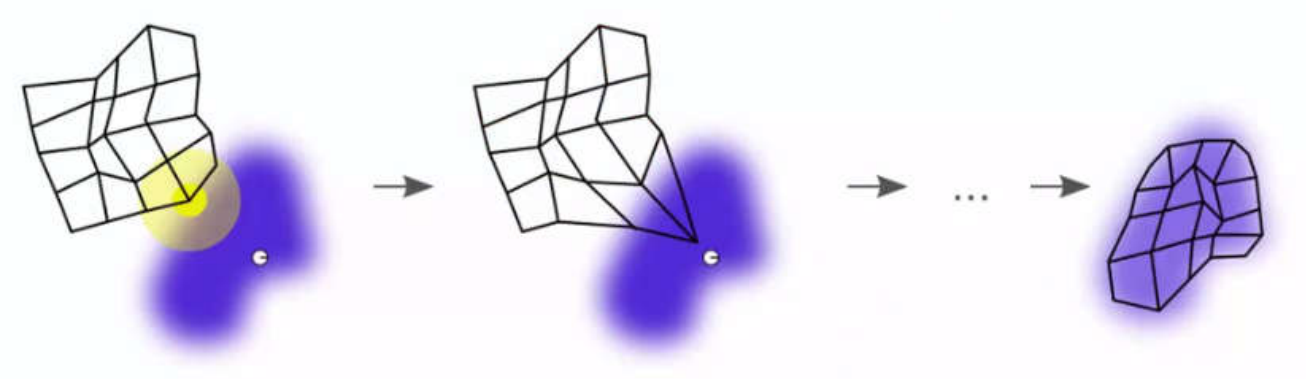
☞ Next **SOM** is actually going to **update** the **weights**, according to **BMU**, so that the ***output-node*** gets even ***closer*** to our ***first-input*** (first row) in our ***data-set***.

☞ The reason we are updating the weights is because we ***don't have control*** of our ***inputs***, the only thing that we can control in that formula are the ***weights***.



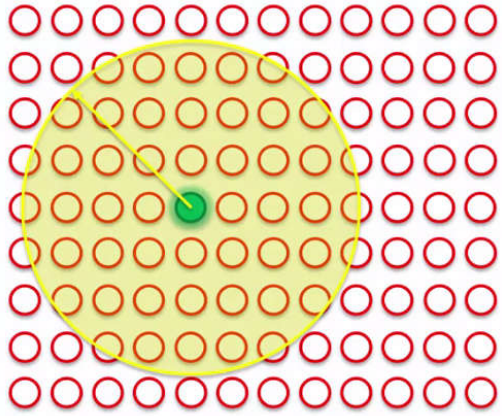
❒ In simple words, the ***SOM*** (white structure/grids) is coming ***closer*** to that ***data point*** (***purple*** colored ***cloud***). In the following image we can see that the ***SOM*** is moving over the ***data-point-cloud*** (***Purple*** colored).

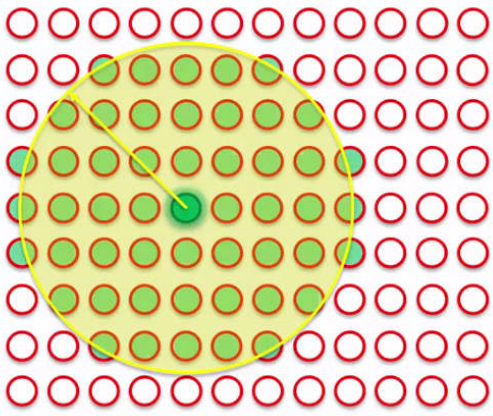
☞ At the very start ***SOM*** is ***initialized*** with some ***weights*** and then it ***updates*** the ***weights*** and move over the ***data-points***.



❒ ***Setting radius (how grouping happens):*** In the next step we set a **radius** around this **BMU**, and every single **point** (**output-node**) of our SOM that falls inside that ***radius*** is going to have its ***weight updated*** to come ***closer*** to that ***row*** that we ***matched up*** with.

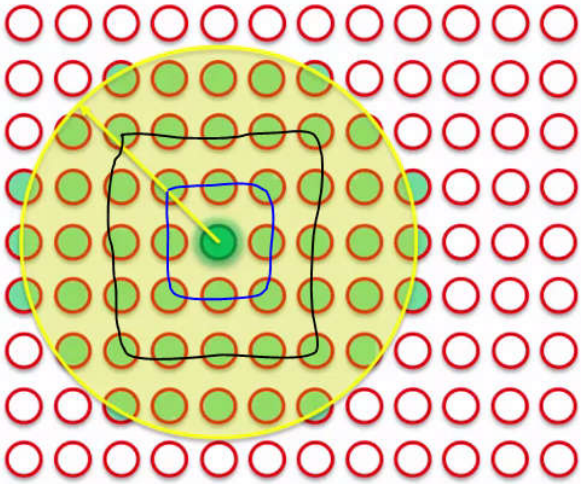
☞ The ***closer*** to the ***BMU***, the ***heavier*** are the ***weights*** being updated. So weights ***near BMU*** are going to be ***updated*** the ***most***, weights ***far*** from ***BMU*** are going to be updated ***less***.





☞ Think it as the **nodes** are **dragging** each other. Whole structure **near** the **BMU** is slowly **pulled towards** the same direction.

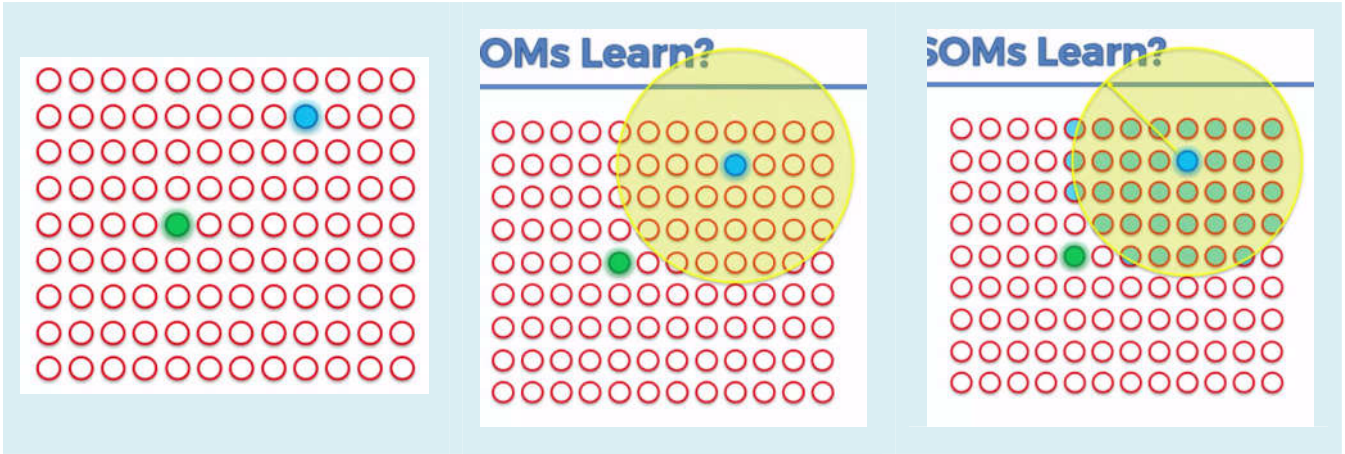
☞ The **closer** nodes are to this **BMU**, the **harder** they will get **pulled** towards that row (input) that the **BMU** matched up with.



So that's how the radius concept works.

❒ ***How nodes fall into different groups:*** Now let's have a look at ***2nd-row*** (**2nd-input**). Let's say ***2nd-row*** has its ***BMU*** at the ***Blue-Colored*** node.

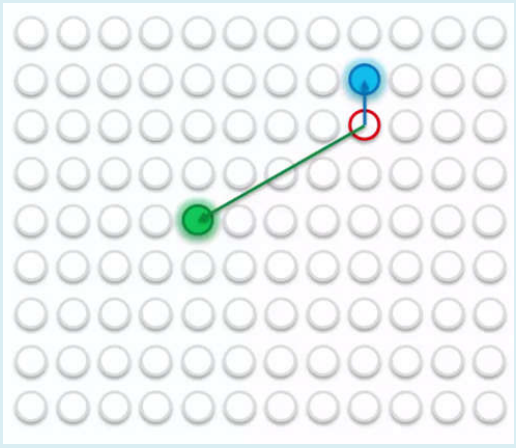
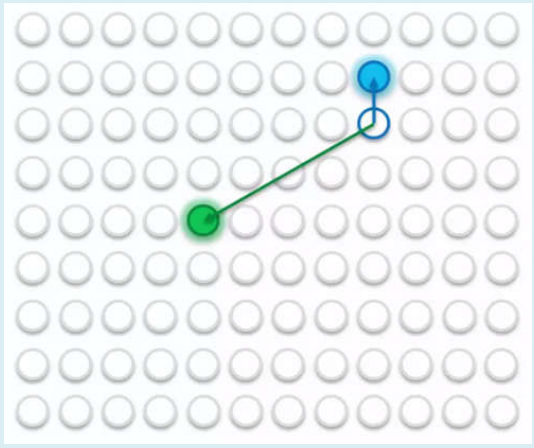
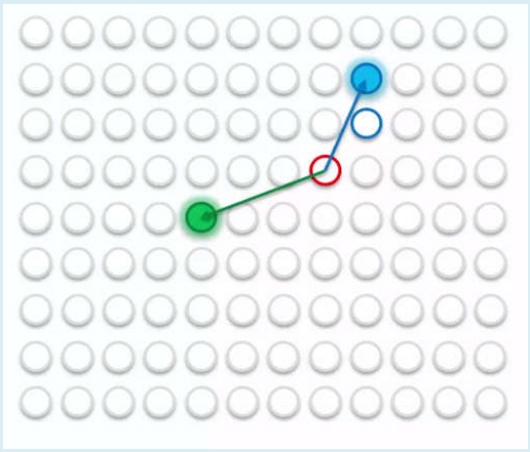
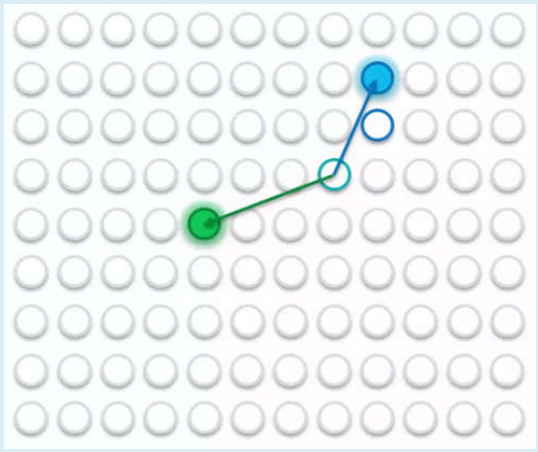
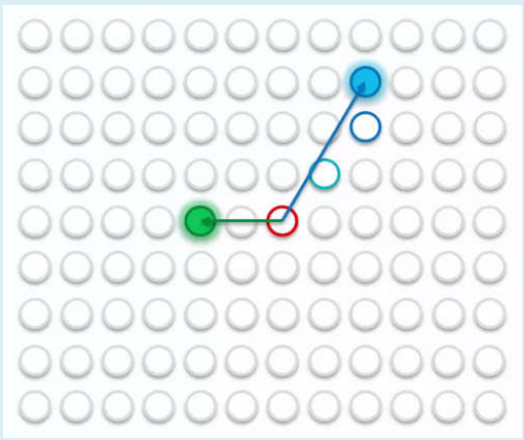
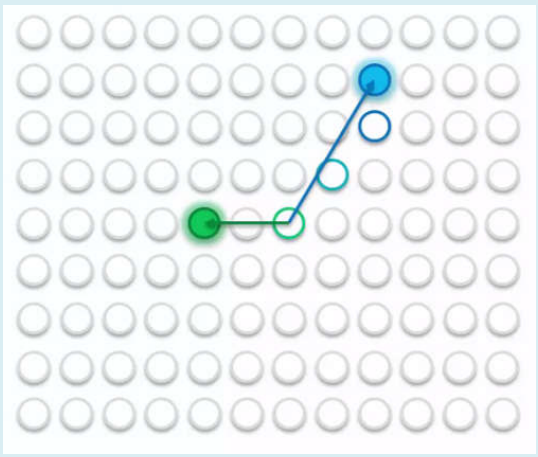
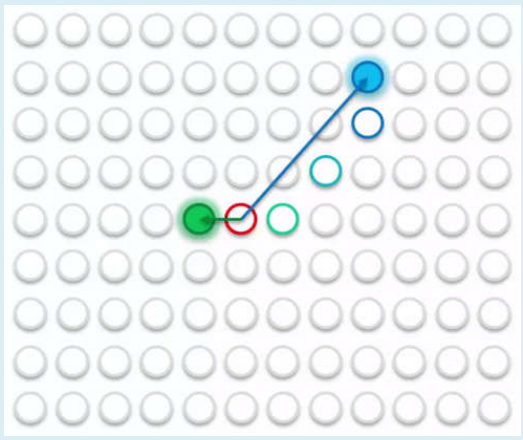
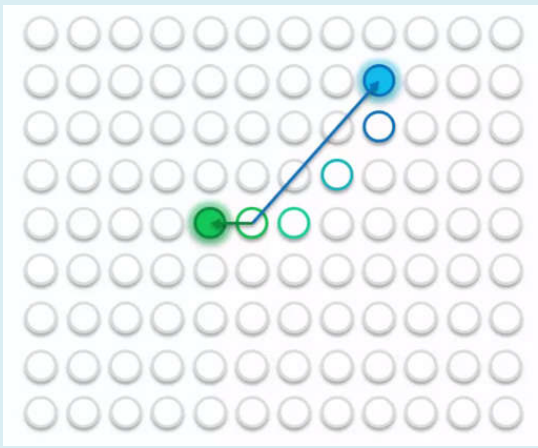
☞ Here again this BMU **drags** all the nodes that are **close** to it that are falls into **BMU's radius**.

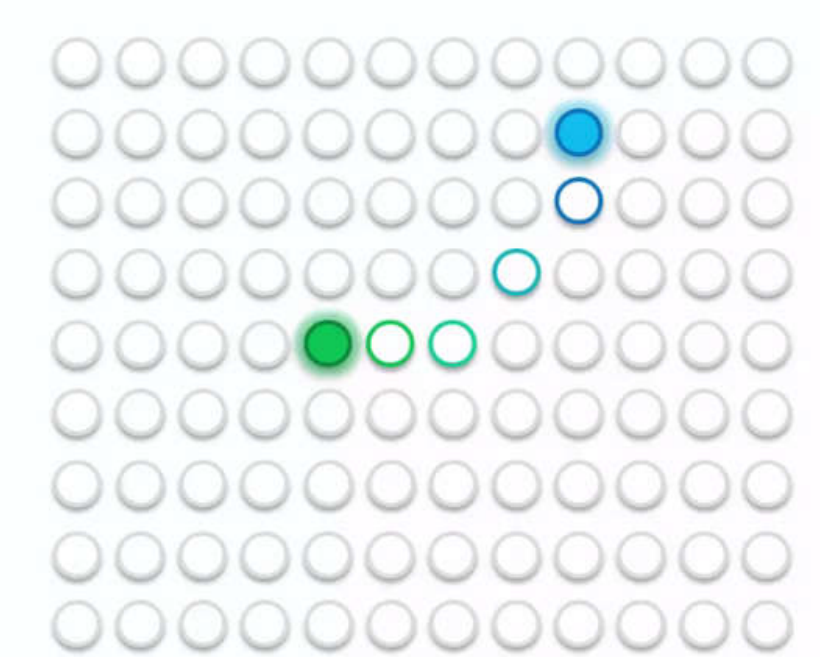


☞ So for above two **BMUs** the nodes closer to **Green BMU** falls in "**Green-group**" and nodes are closer to **Blue BMU** falls in "**Blue-group**". It's pretty simple.

➢ For example, a node far away from the green ***BMU***, is close to the ***blue BMU***, it is pulled much harder with the ***blue BMU***, and therefore it becomes like the blue ***BMU***.

So now we've got some idea about how this SOM works.

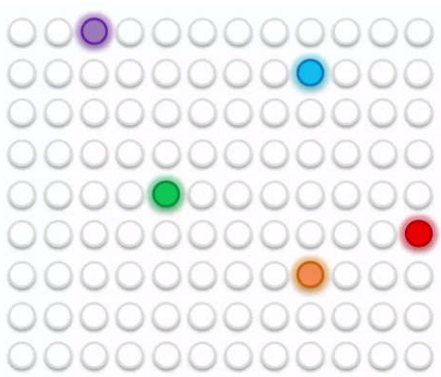


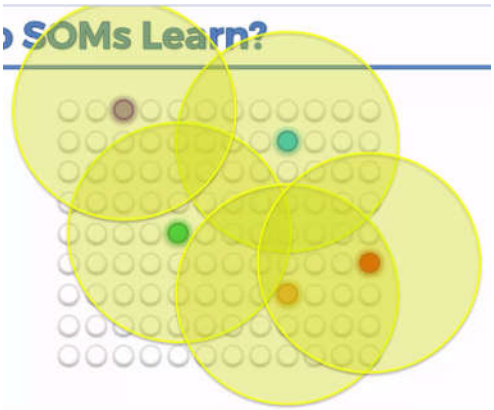
13.3.4 How SOM updates itself

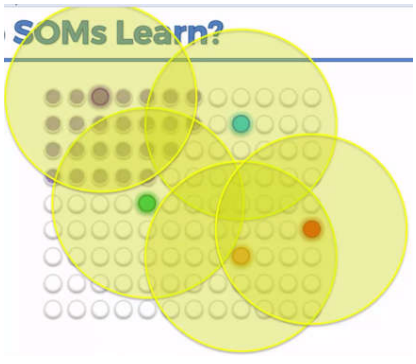
❒ Here we have ***5 BMUs***. Now each these ***BMUs*** are going to be ***updated***.

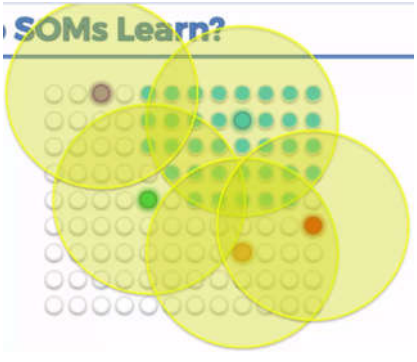
- ☞ Then each one of these ***BMUs*** is going to be ***assigned*** an ***area*** around it, and that area is going to be calculated through a ***radius***.
- ☞ Here we can see that there's some values that don't **fall** under the **radius**, that usually doesn't happen in **SOM** this is just our **visual** example.
- ☞ Normally the **radius** at the **start** is selected as **quite large** so that it **covers** nearly the **whole** *self-organizing map*.

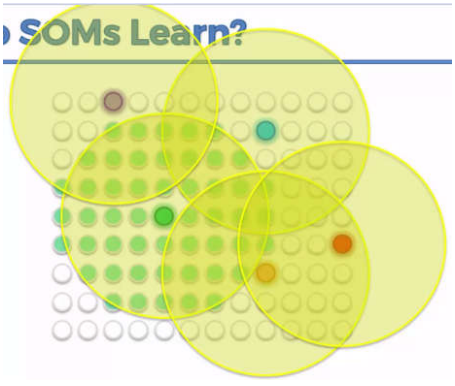
Here all of the nodes that fall into these areas are updated.

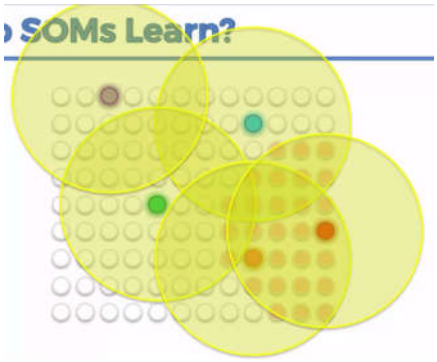


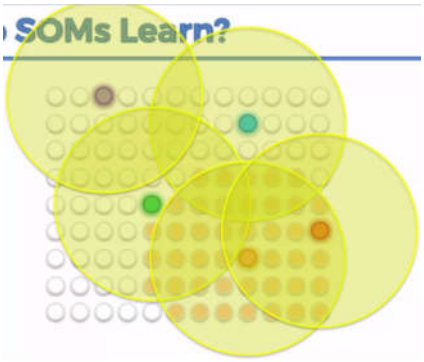


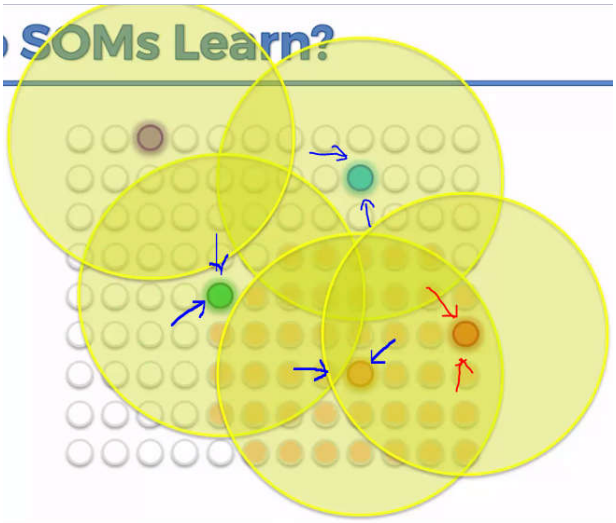






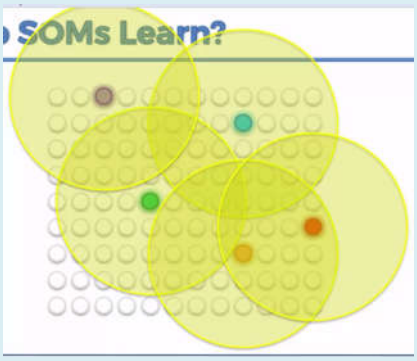


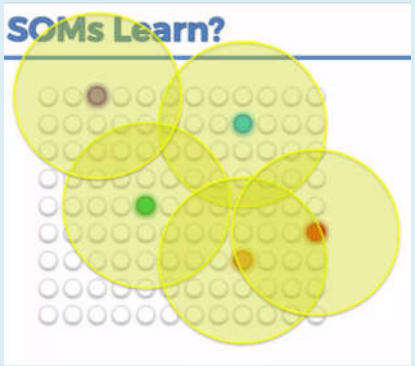


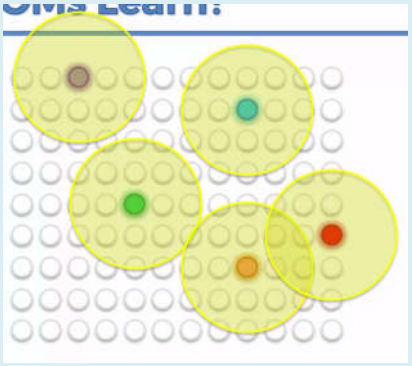


- So ***BMUs*** dragged the ***nearby nodes*** closer and closer and there's a ***competition*** between them each node feels attraction for multiple BMUs. That's normal, that's what happens in the ***SOM***.

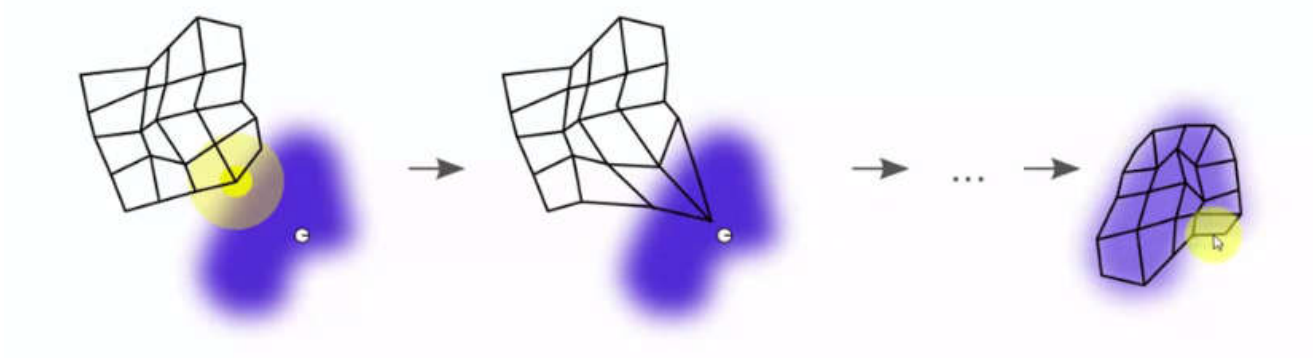
- ***BMUs radiuses shrink:*** One ***epoch*** completes after you go through ***all of your rows*** in your ***dataset*** and all of these ***updates*** happen. In a **new** ***epoch*** when you go through your **rows** again, a unique feature of the ***Kohonen Learning Algorithm*** is applied and makes all the BMUs ***radiuses shrink***.
  - When the ***radiuses*** of all BMUs become a bit ***smaller***, and this time when you're going through your ***dataset***, your ***BMUs*** are pulling ***less nodes*** than previous ***epoch***.
  - And again, the ***radiuses shrink***. Then again ***less nodes*** are ***pulled*** towards the BMUs. The process becomes more and more ***accurate*** as epoch passed.



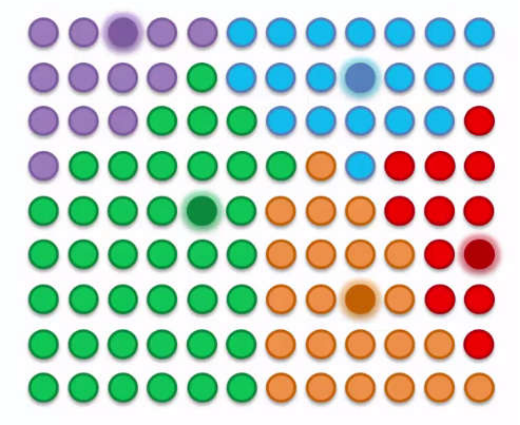


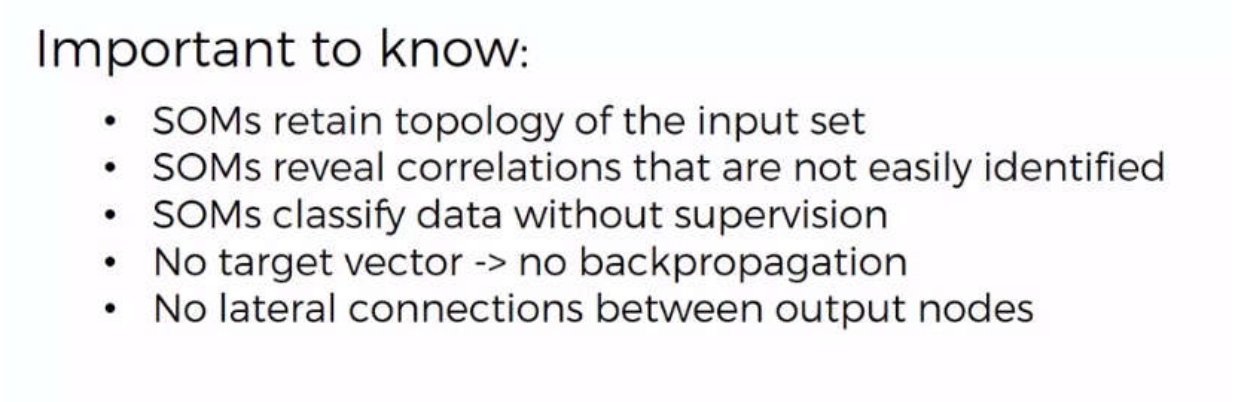


- At the ***very-start*** you were just trying to ***get*** all of your ***SOMs*** very ***close*** to your ***data***. Then as you go through more and more and more ***epochs*** you are adjusting your ***SOM*** in a more ***precise*** manner. As a result ***SOM*** slowly ***goes over*** your ***data*** and becomes kind of ***musk*** for your data, as we can see in this example down here.



❒ Then through lots and lots of ***epochs***, our SOM might look something like this:





NOTE:
A couple of things that are important to know:

[1]. ***SOMs retain topology of your input set:*** The map is slowly become a musk of your data. Your data might have some topology, some interrelations in your data. SOM retain those.

[2]. ***SOMs actually reveal correlations between data that are not easily identified:*** Say you **hundreds** of **columns** in your dataset it can be very challenging to find kind of correlations or similarities that might be present in your dataset.

☞ A SOM can neatly analyze all that for you, and then put all of that for you into a map.

[3]. ***SOMs classify data without supervision:*** SOM is unsupervised Deep learning technique. SOMs don't need any labeled data.

☞ It doesn't need any supervision, the SOM will, extract features on its own, show us features, dependencies, correlations, and similarities. They can be used in *scenarios* where you don't actually know *what* you're *looking for* but you want to find any kind of *correlations* in your data.

[4]. ***SOMs don't require a target vector:*** There is no backpropagation in the training of a SOM (unlike ANN). SOM doesn't have a target vector. There is no error to backpropagate.

☞ In *ANN* the *data* would go through the *NN*, we'd get a result we'd *compare* it to the *target vector*, we'd find the *error* and then we'd *backpropagate* that *error* through the *NN* to update the *weights*.

[5]. ***There is no lateral connections between output-nodes:*** We didn't actually need any connection between the nodes.

☞ The only thing that happens between the nodes is: when you **pull** on one **node**, the other **close** ones get **pulled**.

☞ When we say that ***there's no lateral connections between the output-nodes*** means that there's ***no*** actual ***NN type*** of ***connections*** there's ***no activation functions*** between them and so on.

☞ Sometimes you'll see images where the *output-nodes* are *connected*. There's a *grid* behind the *nodes*. They're just showing that these are ***output-nodes*** on a ***SOM***. But there is *no* actual *formulas* or *equations* going on between those *output-nodes*.

❒ ***Additional reading:*** If you'd like to study with some soft introduction into mathematics behind self-organizing maps, about how the radius changes and how the weights are updated based on how close on the proximity to your best-matching unit. Read the blog "*Kohonen's Self Organizing Feature Maps*" by *Mat Buckland*.

☞ You also get some introduction to programming SOMs.

☞ Several useful blogs are available at *ai-junkie.com*.

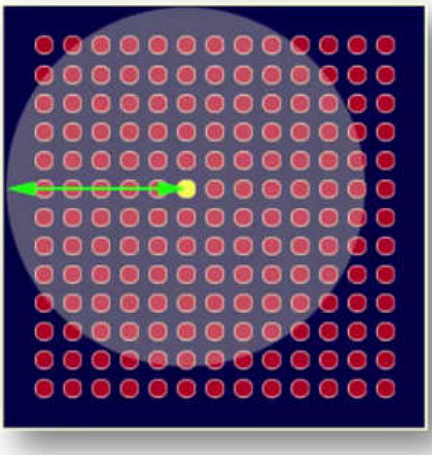
Additional Reading:

Kohonen's Self Organizing Feature Maps

By Mat Buckland (2004?)

Link:

http://www.ai-junkie.com/ann/som/som1.html



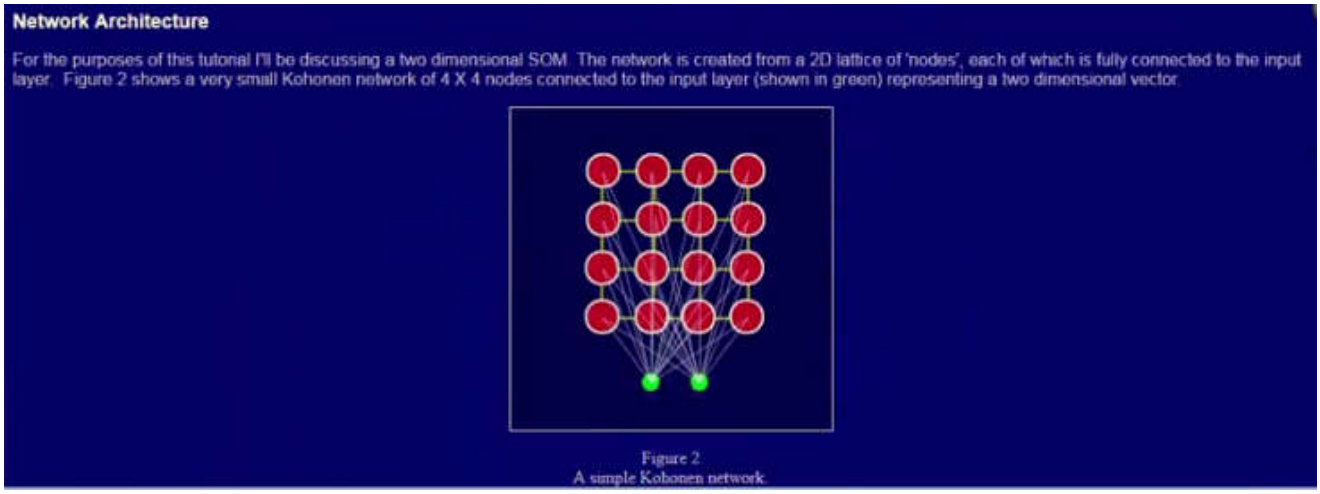
13.3.5 Example: Live example of a SOM

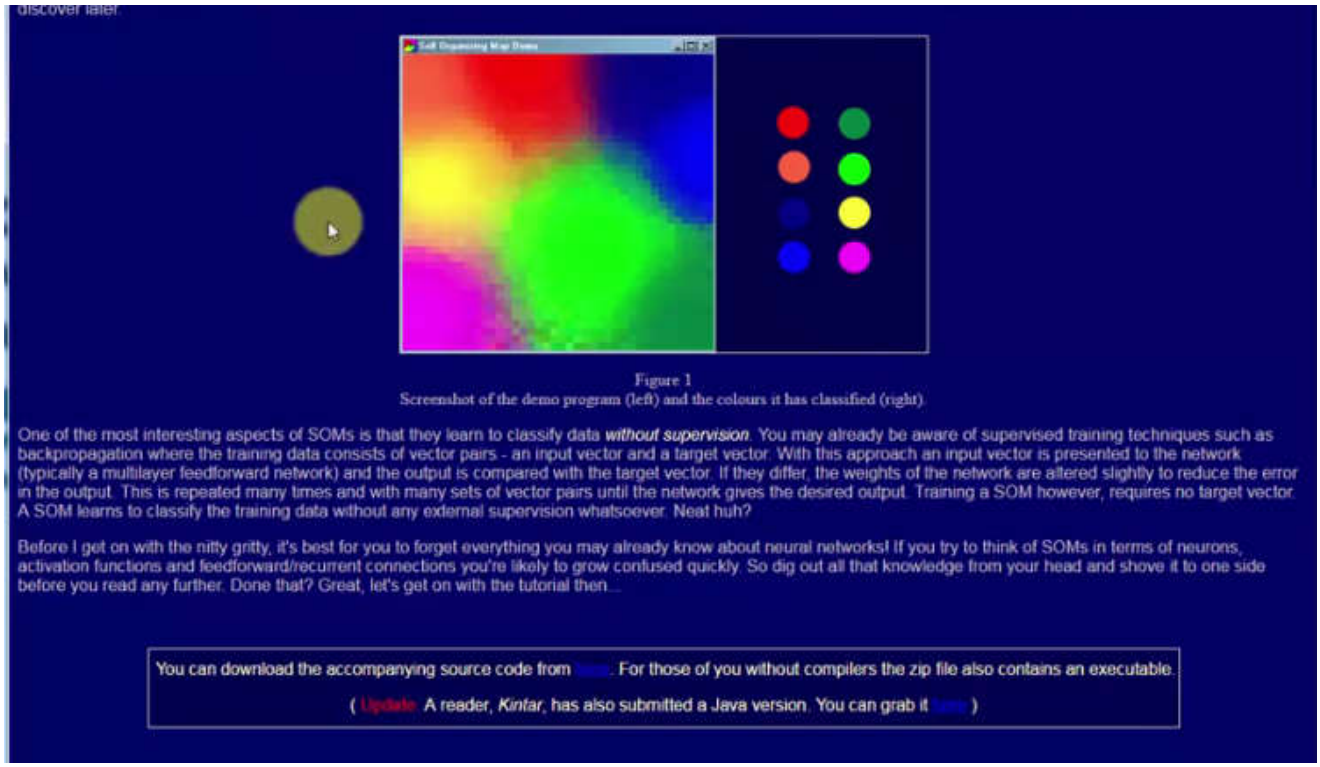
Now we are going to see a live example of a SOM. Here we use ***.exe*** (executable) file provided by ***AI junkie***. *ai-junkie.com* is the blog that (we mentioned above section).

❒ In this blog you can get a brief introduction into ***SOMs*** and the ***mathematics*** behind them, and as well get some ***programming*** examples.

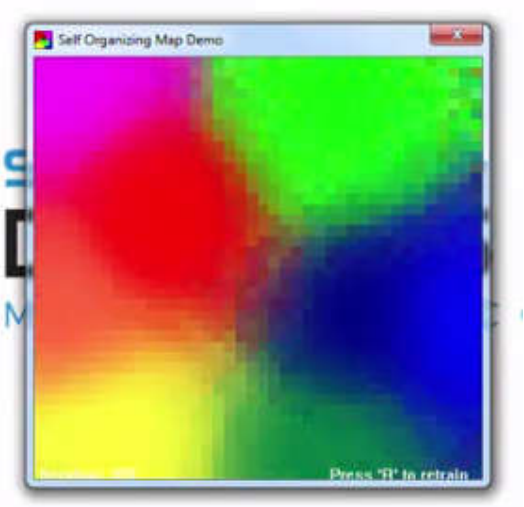
☞ You will be able to download source codes for the example we're going to be looking at.





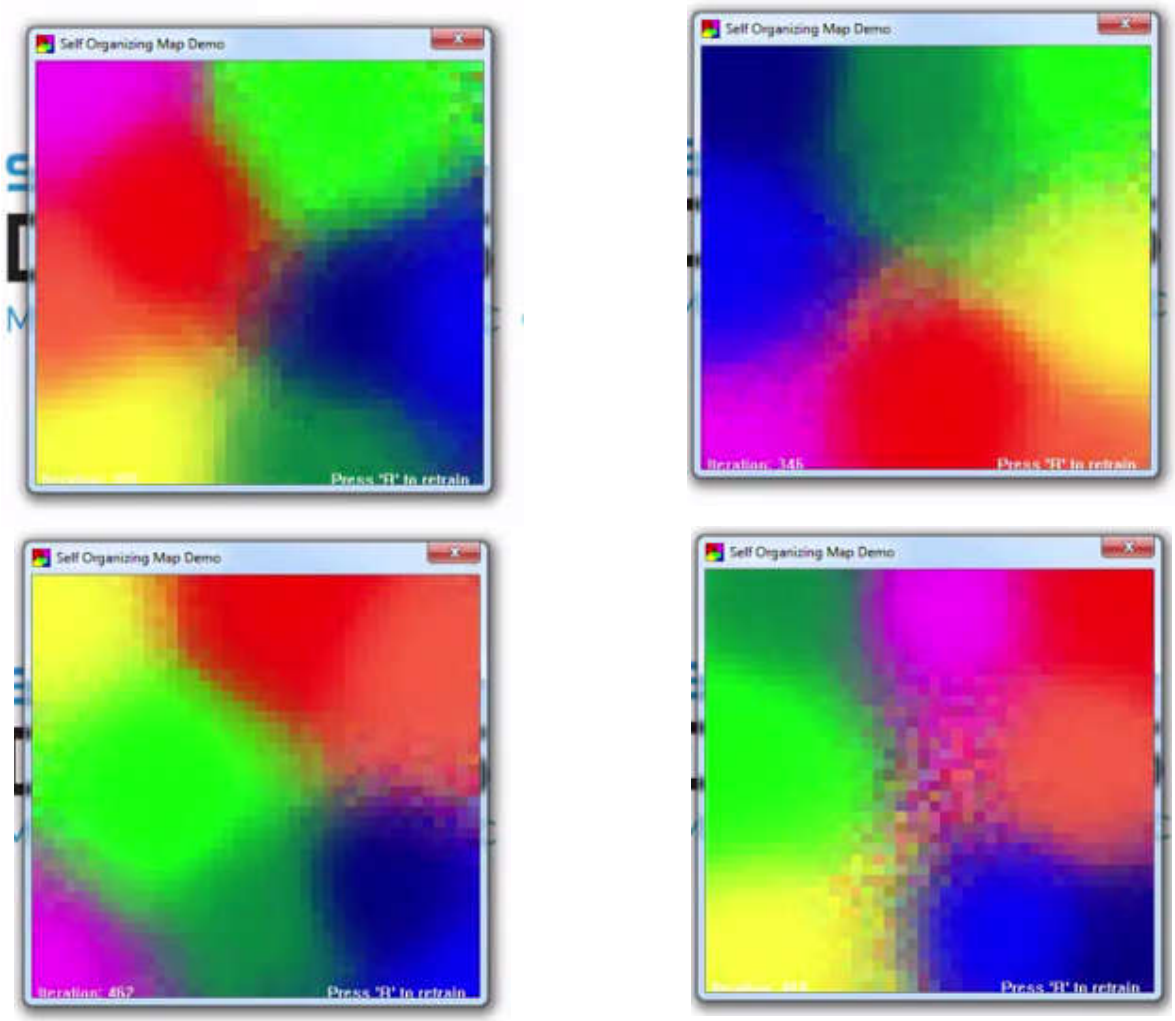


❒ We're going to be creating a self-organizing map like this by running the provided .exe file.



- ☞ Once we run it, this app will create a SOM, which has, as inputs, these eight colors: **red**, **orange**, **dark blue**, **light blue**, **dark green**, **light green**, **yellow** and **magenta**.
- ☞ So, each of the ***rows*** in the data set, is ***R-G-B*** code for each of these ***colors***. And those codes are actually ***normalized*** on scale on ***range*** **`[0, 1]`**.
- ☞ The point is we have got **eight rows** on our dataset (**8-colors**) and each one of them has **three columns** (**`R, G, B`**) and we are going to put them into a SOM.

Once you run it, you will see that there is the number of iterations and then you can press ***R*** to ***retrain*** this ***SOM***.

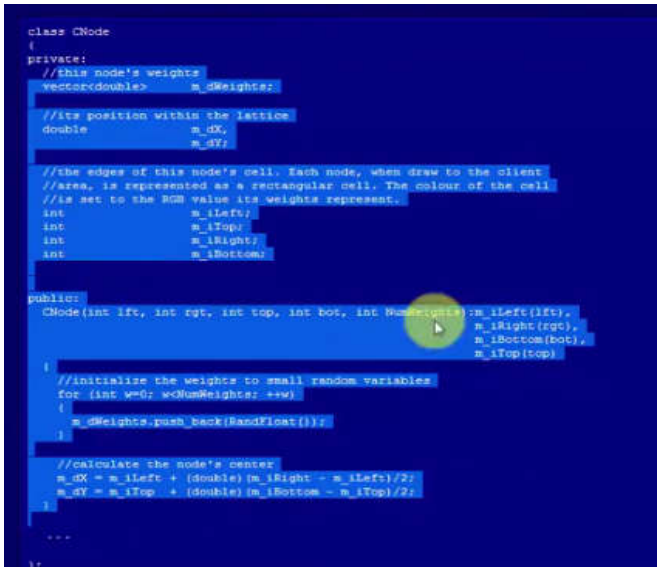


- You will notice that it ***organizes*** itself every time ***differently*** that's because your ***weights*** are ***initialized*** at ***random***.

- ***But the main point is:*** it's putting a ***`dark blue`*** next to the ***`light blue;`*** and the ***`light green`*** next to the ***`dark green;`***, and most of the time ***magenta red***, ***orange*** together. That's what we mean that ***SOM*** does ***preserve topology*** or it does ***find*** and ***preserve similarities*** in your data set. So here SOM groups **similar colors** together.

- Once again, check out the website, here at ***AI junkie***, and you will be able, get the codes here, you can actually get it in Java as well. And you can get some lovely examples of how all this works.

  ☞ You'll see that self-organizing maps are not that hard mathematics and to code.



```
class CNode
{
private:
  //this node's weights
  vector<double>      m_dWeights;

  //its position within the lattice
  double              m_dX,
                      m_dY;

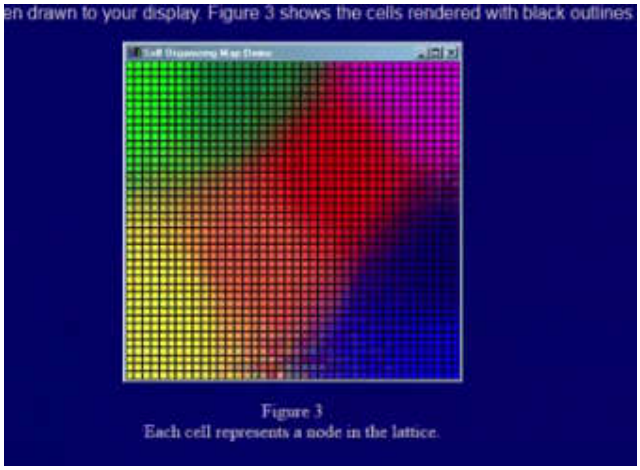
  //the edges of this node's cell. Each node, when draw to the client
  //area, is represented as a rectangular cell. The colour of the cell
  //is set to the RGB value its weights represent.
  int                 m_iLeft;
  int                 m_iTop;
  int                 m_iRight;
  int                 m_iBottom;

public:
  CNode(int lft, int rgt, int top, int bot, int NumWeights):m_iLeft(lft),
                                                            m_iRight(rgt),
                                                            m_iBottom(bot),
                                                            m_iTop(top)
  {
    //initialize the weights to small random variables
    for (int w=0; w<NumWeights; ++w)
    {
      m_dWeights.push_back(RandFloat());
    }

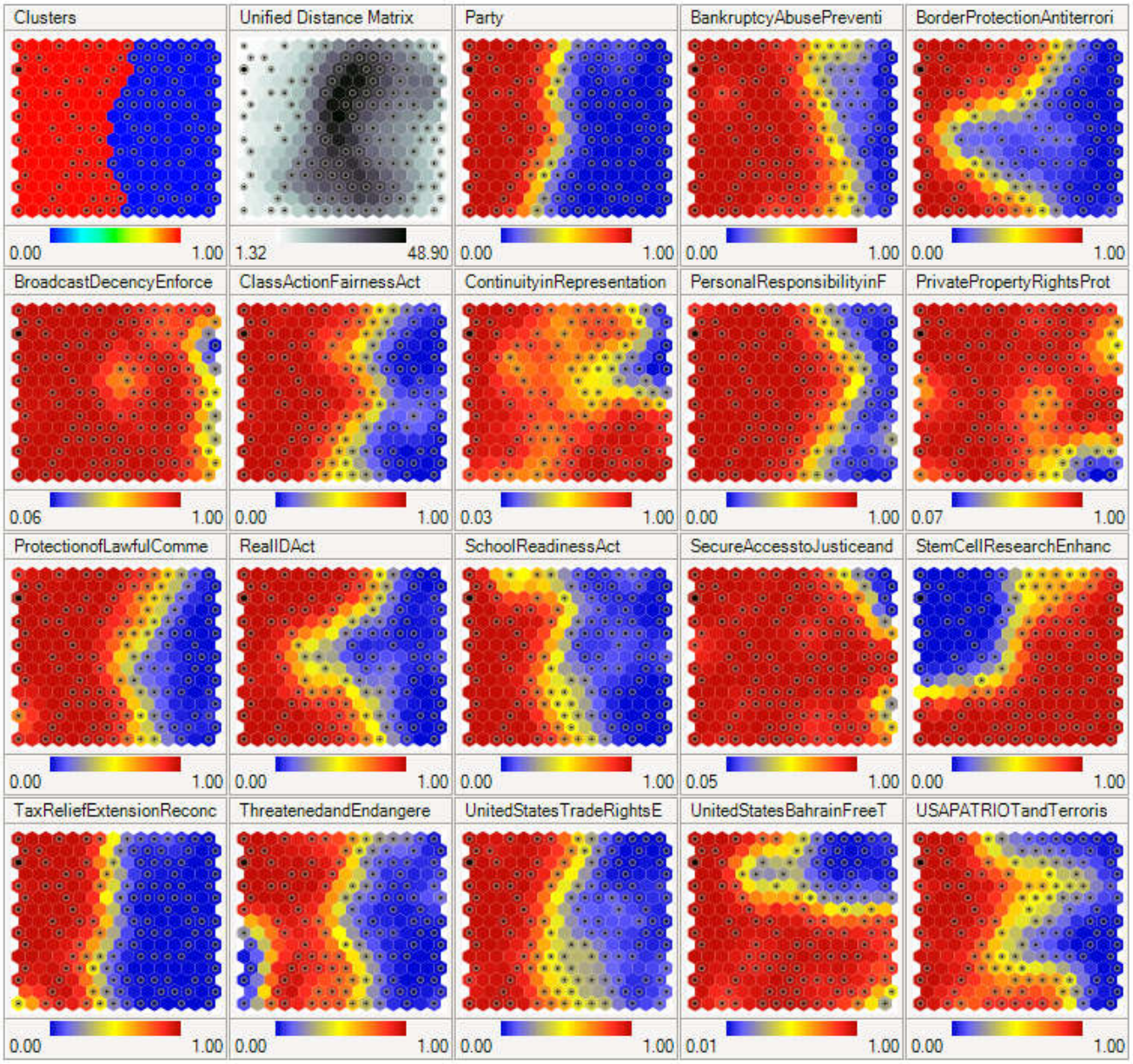
    //calculate the node's center
    m_dX = m_iLeft + (double)(m_iRight - m_iLeft)/2;
    m_dY = m_iTop  + (double)(m_iBottom - m_iTop)/2;
  }

  ...

};
```



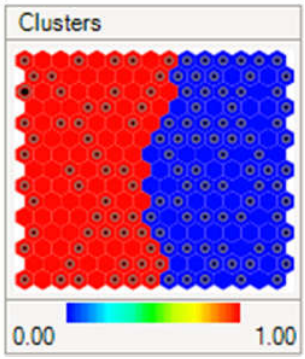
13.3.6 Example: Reading a SOM



❒ Here we're going to have a look at how to read advanced SOM. You might come across a ***SOM*** that looks like ***above***. There's actually a ***lots*** of ***representations*** inside this ***SOM***. This is an example from ***Wikipedia***, and is a ***SOM*** of ***voting*** results/ ***patterns*** in the ***US Congress***.

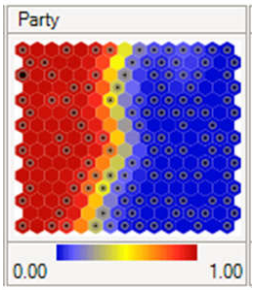
- ☞ The ***input data*** for this map was like a ***data sets*** of ***Members*** of ***Congress*** in the ***US Congress*** like over ***500*** or ***535 Members***. Where each Member say about a certain question that they were voting on. Did they say yes/ no/absent from voting.
- ☞ And based on that information, the ***SOM*** group them which ***Members*** of ***Congress*** are ***close***, are ***similar*** to each other, which are ***dissimilar***, and place them onto a ***map***.

❒ ***CLUSTERS:*** So here we've got a map of the ***Members*** of Congress, where ***SOM*** splits the data set into ***two classes***.



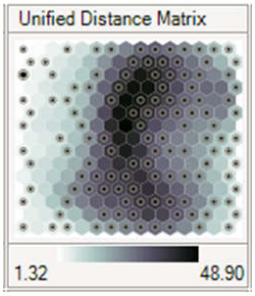
- ☞ This is the overall cluster, it is bit different. This is how the **SOM**, **splits** the Members of **Parliament**, just based on all of the things that they **voted** on all of the different questions (it is ***not solely based*** on ***Republican -Democratic***).
- ☞ Here ***red*** **doesn't** mean ***Republican*** or ***blue*** **doesn't** mean ***Democratic***, just red and blue other two colors that are used to identify the two clusters.
- ☞ This cluster is based on overall results on the voting about: 17 topics –***Bankruptcy Abuse Prevention, Border Protection Anti-terrorist, Broadcast Decency Enforce, Class Action Fairness Act, Continuity in Representation, Personal Responsibility inF, Private Property Rights Prot*** etc (these are the actual topics that being voted).

❒ ***PARTY:*** The actual ***split*** between the ***parties*** is modeled over here. So when you ask the SOM which Member belongs to which party, the answer is like this, so ***red*** here is ***Republican*** and ***blue*** is ***Democratic*** Party.



☞ This is the clusters of members that belonging to either Democratic Party or the Republican Party.

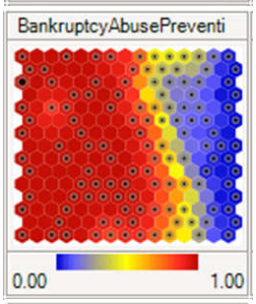
❒ ***UNIFIED DISTANCE MATRIX:*** In the second representation, we see the ***unified distance matrix***, is also called the ***U matrix*** and it shows is the **distance between points/nodes** on the ***SOM***.



☞ At the ***middle*** it's ***darker***, means that these points are ***further apart*** from each other, ***lighter***-points are ***close*** together.

☞ It makes sense over here, because the ***darker points*** are exactly on that ***ridge*** (border between two clusters) and there's a lot of ***dissimilarity***.

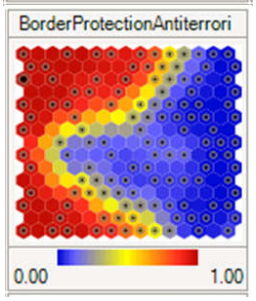
❒ ***Bankruptcy abuse prevention:*** in this case Republicans voted yes and maybe that is a corporate bankruptcy, rather than an individual purpose bankruptcy.



☞ Here is an invisible line (in some SOM represented by a black line/border) is the border between ***Republican –Democratic***. It can show us: that most of the Republicans according to SOM voted yes, some of the Democrats voted yes and other Democratic voters voted no.

☞ red = yes, and blue = no.

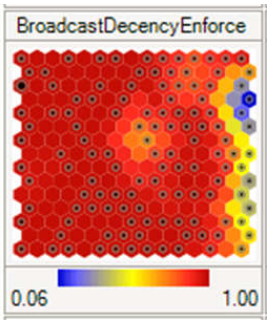
❒ ***Border Protection, anti terrorism:*** Here you can see a very interesting split (keep in mind that there is **invisible border** from the main cluster: "**Clusters**", the first map.).



☞ We can see that **some Democrats** voted **no** over here, but ***most Democrats*** voted ***yes***, and then ***some*** even ***Republicans*** voted ***yes***.

☞ red = yes, and blue = no.

❒ ***Broadcast Decency Enforce:*** Looks like there's a lot of yes-vote on this question. All of the **Republicans** definitely voted **yes**. **Most** of the **Democrats** voted **yes**. Some people voted no.



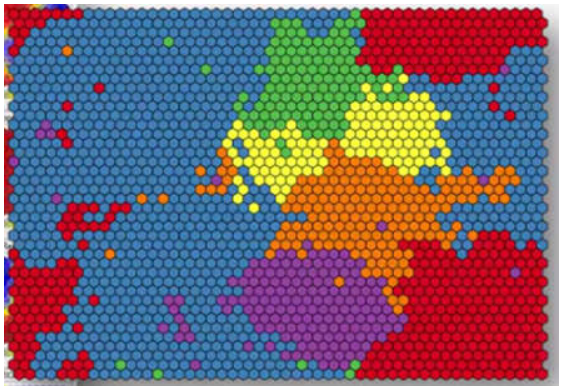
☞ You can see a bit of yellow. The reason for this most likely, is that here we've got 15×15 nodes. So that means we have about 225 nodes in our SOM, whereas in the US Congress, you have over 500 Members. So meaning that the way this map has been overlaid over our data, is not a one to one relationship. There are several Members of Congress represented within every node.

☞ This yellow basically means that maybe ***most*** are ***yes-voting***, but there was like one or two people who voted ***no***. And same thing over other yellow areas, means there's kind of ***inconsistency*** there.

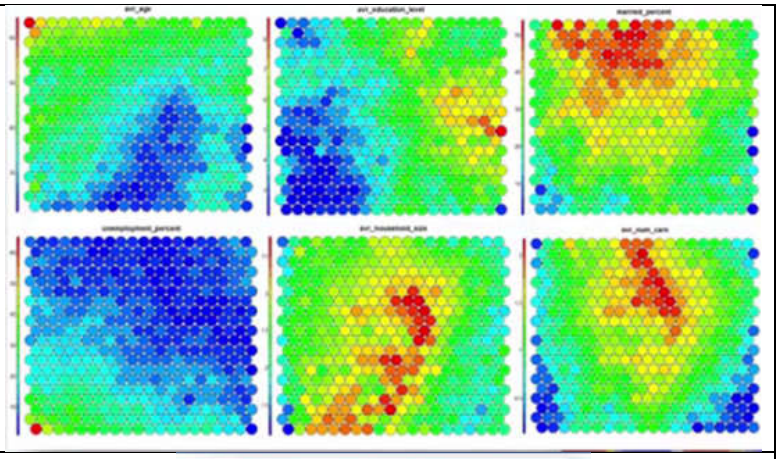
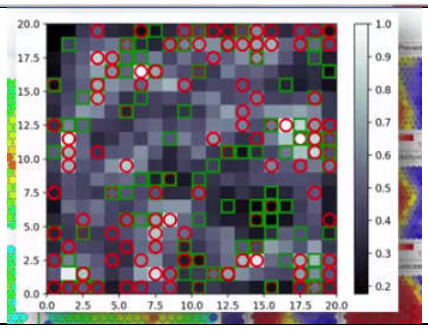
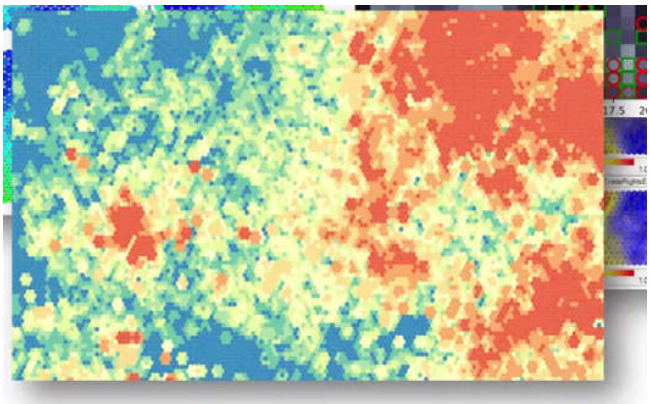
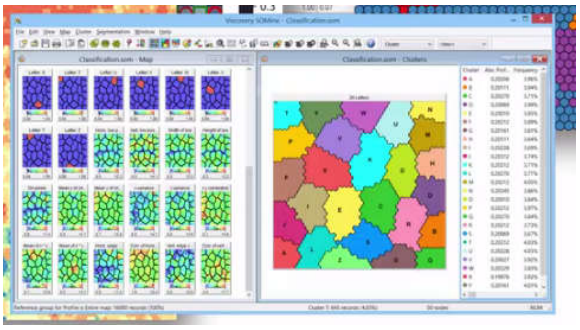
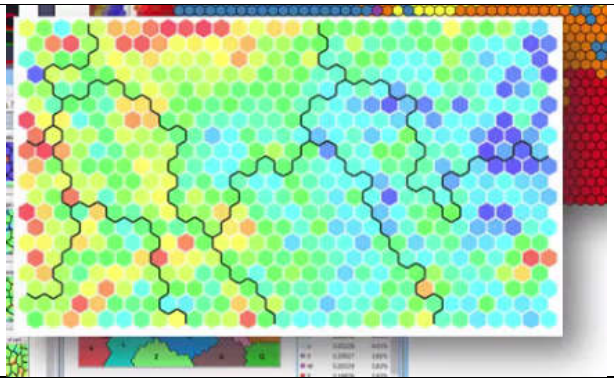
☞ red = yes, and blue = no.

And that's how we read an Advanced SOM.

❒ Because of the simplicity of SOMs, you will find that there are lots and lots of different versions and variations of implementations of SOMs.



☞ Here's another example from another website, ***boyeltab.org***

☞ So this one is from *R-bloggers*, it shows you can create a self organizing map in ***R***.	
☞ This is a self organizing map which we'll use in *this Chapter*.	
☞ This SOM from *StackOverflow*.	
☞ This one from *viscovery*. ☞ You can see these are the clusters that have been identified and then you have separate maps for each one of your features. Like the voting-SOM from the Wikipedia.	
☞ Another from *visualcinnamon.com*. ☞ Here you can actually see those **lines** for **highlights** and **identify** the **clusters**. And then as you go through those separate ***features*** of your ***data set***, you still keep these ***clusters*** in mind.	

❒ ***Additional readings:*** We reference this one below because it's actually quite a cool representation. This one is coded in ***D3.js***, so it's a ***JavaScript*** library.

☞ This is by ***Nadieh Bremer*** from ***Netherlands Amsterdam*** and she was kind enough to actually explain how she created this visualization and all the hexagons and you've got the codes here.

☞ https://www.visualcinnamon.com/2013/07/self-organizing-maps-creating-hexagonal/

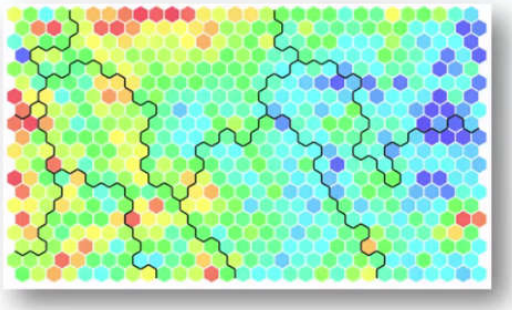
Additional Reading:

SOM - Creating hexagonal heatmaps with D3.js

By Nadieh Bremer (2003)

Link:

https://www.visualcinnamon.com/2013/07/self-organizing-maps-creating-hexagonal.html



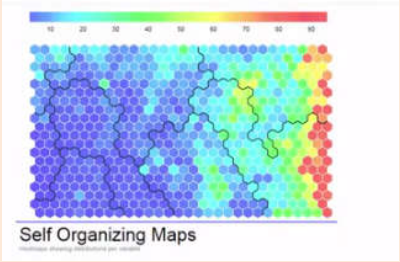
https://www.visualcinnamon.com/2013/07/self-organizing-maps-creating-hexagonal/

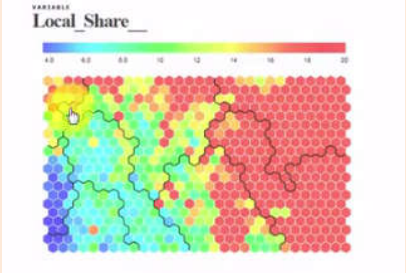
WORK ART SHOP ABOUT CONTACT BLOG

Posted on July 22, 2013

Creating hexagonal heatmaps with d3.js

In my previous post I spoke a bit about a program I wrote in R that helps me perform **self organizing map** (SOM) analyses and create heatmaps. From the cleaned data file all the way to the visualization and analysis of the heatmaps.







D3.js

D3.js is a JavaScript library for producing dynamic, interactive data visualizations in web browsers. It makes use of Scalable Vector Graphics, HTML5, and Cascading Style Sheets standards. It is the successor to the earlier Protovis framework.
Wikipedia

Chapter 13: Part 4

Deep Learning

SOM: Building a SOM

`Python Implementation`

Yo!!! Welcome to **Unsupervised Deep Learning**.
It's been a looooong journey. Let's dive into the topics.....

13.4.1 Objectives

- How to ***build*** a ***SOM***
- How to return the specific features (like ***frauds***) ***detected*** by the ***SOM***
- How to make a ***Hybrid Deep Learning Model***



In this section we will implement our very first ***unsupervised deep learning model***, which is the ***self organizing map*** or ***SOM***.

13.4.2 Problem Description

Here we try to solve a fraud detection problem.

❒ Let's assume a ***data-set*** is given to us that contains ***information*** of ***customers*** of a ***Bank*** *applying* for an ***advanced credit card***. Basically, these ***informations*** are the ***data*** that customers had to ***provide*** when filling the ***application form***. And our mission, is to ***detect*** potential ***Fraud*** within these ***applications***.

☞ At the ***end*** of the ***mission***, we have to give the ***explicit list***, of the ***customers*** who potentially ***cheated*** (list of ***Potential Fraudulent Customers***).



It's not a Supervised model: We'll ***not make*** a *supervised deep learning model* with a ***dependent variable*** that has binary values: ***{yes, no}*** and try to predict if each customer potentially cheated, yes or no.

☞ Our models will be an ***Unsupervised Deep Learning Model***, which means that we will ***identify*** some ***patterns*** in a ***High Dimensional data sets*** full of ***nonlinear relationships***. And one of these ***patterns*** will be the ***potential fraud***. That is the customers who potentially cheated.

13.4.3 Data Set Description

We first Import the essential libraries and then our data-set.

```
# importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# importing the Data-set
dataset = pd.read_csv("Credit_Card_Applications.csv")
```

dataset - DataFrame

Index	CustomerID	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	Class
0	15776156	1	22.08	11.46	2	4	4	1.585	0	0	0	1	2	100	1213	0
1	15739548	0	22.67	7	2	8	4	0.165	0	0	0	0	2	160	1	0
2	15662854	0	29.58	1.75	1	4	4	1.25	0	0	0	1	2	280	1	0
3	15687688	0	21.67	11.5	1	5	3	0	1	1	11	1	2	0	1	1
4	15715750	1	20.17	8.17	2	6	4	1.96	1	1	14	0	2	60	159	1
5	15571121	0	15.83	0.585	2	8	8	1.5	1	1	2	0	2	100	1	1

❒ ***Data-set description:*** There are ***690 customers*** in the given dataset. It contains the ***applications*** for the ***advanced credit card***.

☞ This data set is taken from the ***UCI Machine Learning Repository***, it is called the ***Statlog (Australian Credit Approval Data Set)***. Link is in the picture.



Statlog (Australian Credit Approval) Data Set

Download: Data Folder, Data Set Description

Abstract: This file concerns credit card applications. This database exists elsewhere in the repository (Credit Screening Database) in a slightly different form

Data Set Characteristics:	Multivariate	**Number of Instances:**	690	**Area:**	Financial
Attribute Characteristics:	Categorical, Integer, Real	**Number of Attributes:**	14	**Date Donated**	N/A
Associated Tasks:	Classification	**Missing Values?**	Yes	**Number of Web Hits:**	87829

☞ Link: https://archive.ics.uci.edu/ml/datasets/statlog+(australian+credit+approval)

❒ ***Data Set Information:*** This file concerns credit card applications. All ***attribute names*** and ***values*** have been changed to ***meaningless symbols*** to ***protect confidentiality*** of the data.

☞ This dataset is interesting because there is a good ***mix*** of ***attributes*** -- ***continuous***, *nominal with small numbers of values*, and *nominal with larger numbers of values*. There are also a few ***missing values***.

☝ That makes this problem even more *complex*, and *difficult* to solve for *human*. So we clearly need a *deep learning model* to find the ***cheaters***.

❒ ***Attribute Information:*** There are ***6 numerical*** and ***8 categorical*** attributes. The ***labels*** have been ***changed*** for the ***convenience*** of the ***statistical algorithms***. For example, **`attribute 4`** originally had *3 labels* **`p,g,gg`** and these have been changed to ***`labels 1,2,3`***.

```
A1: 0,1 CATEGORICAL (formerly: a,b)
A2: continuous.
A3: continuous.
A4: 1,2,3 CATEGORICAL (formerly: p,g,gg)
A5: 1, 2,3,4,5, 6,7,8,9,10,11,12,13,14 CATEGORICAL (formerly: ff,d,i,k,j,aa,m,c,w, e, q, r,cc, x)
A6: 1, 2,3, 4,5,6,7,8,9 CATEGORICAL (formerly: ff,dd,j,bb,v,n,o,h,z)
A7: continuous.
A8: 1, 0 CATEGORICAL (formerly: t, f)
A9: 1, 0 CATEGORICAL (formerly: t, f)
A10: continuous.
A11: 1, 0 CATEGORICAL (formerly t, f)
A12: 1, 2, 3 CATEGORICAL (formerly: s, g, p)
A13: continuous.
A14: continuous.
A15: 1,2 class attribute (formerly: +,-)
```

☞ Basically, the ***independent variables*** are some ***categorical*** and ***continuous*** independent variables. And inside all these variables are ***hidden some frauds*** that we have to detect.

❒ ***Role of SOM:*** The first thing important to understand here is that the ***columns*** are the ***attributes***, those are the ***informations*** of the ***customers***. There are ***16 columns***, we consider first ***15*** as ***features***. And the ***lines/rows*** are the ***customers***.

☞ The unsupervised deep learning model is going to identify some patterns among the customers. It's going to do some kind of ***customer segmentation*** to identify ***segments*** of customers and one of the segments will contain the ***Fraud customers***.

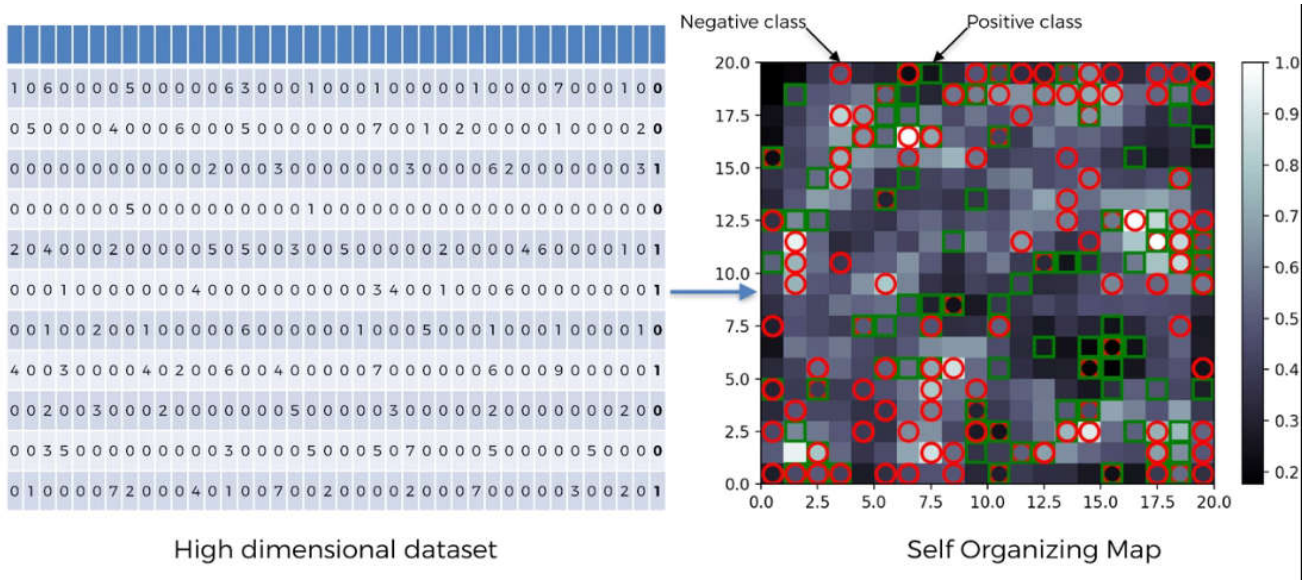
☞ All those segments are going to be on the self organizing map (SOM). It will actually be very explicit, it corresponds to one specific range of values in the SOM.

☞ All these customers/rows are the inputs of our neural network. These input points are going to be mapped to a new output space.

- Between the ***input space*** and the ***output space***, we have ***NN*** composed of ***neurons***, each ***neuron*** being initialized as a ***vector of weights***, this weight vector has ***same size*** as the ***vector*** of ***inputs*** (i.e a vector of 15 elements).
- For each ***observation point*** the ***output*** will be the ***neuron*** that is the ***closest*** to the ***point***. Basically, in the network, we pick the ***neuron*** that is the ***closest*** to the ***customer***, this neuron is called the ***winning node***.
- For each customer, the ***winning node*** is the most similar neuron to the customer. We use a ***neighborhood function*** like the ***Gaussian Neighbourhood function***, to ***update*** the ***weight*** of the ***neighbors*** of the ***winning node*** to ***move*** them ***closer*** to the ***point***.

- We do this for all the customers in the ***input space***. It is a repeating process. And each time we'll repeat it, the ***output space decreases*** and ***loses dimensions*** (shift its shape).
- It ***reduces*** its ***dimension*** little by little.
- And then it reaches a ***point*** where the ***neighborhood stops decreasing***, where the ***output space*** stops decreasing and become ***steady***.

And that's the moment where we obtained our SOM in 2D map, with all the ***winning nodes*** that were eventually ***identified***.



- ***How we detect the frauds:*** The ***frauds*** are ***outliers***, because the fraud basically is defined by something that is ***far from*** the general ***rules***. The ***rules*** that must be ***respected*** when applying to the ***credit card***.
  - The ***frauds*** are actually the ***outlying neurons*** in this **2D-SOM**, because they are far from the ***majority*** of neurons that ***follow*** the ***rules***.

- How can we detect the outlier neurons in the SOM?
  - For this, we need the ***MID***, the ***Mean Interneuron Distance***.
  - That means that in our self organizing map for each neuron, we're going to compute the ***mean*** of the ***Euclidean distance*** between this neuron and the neurons in its neighborhood.
  - We have to define a ***neighborhood*** for each neuron ***manually***. And we compute the ***mean*** of the ***Euclidean distance*** between this ***neuron*** that we ***picked*** and all the ***neurons*** in the ***neighborhood*** that we defined.
  - And by doing that we can ***detect outliers***, because ***outliers*** will be ***far from*** all the neurons in its ***neighborhood***. That's how we detect fraud from SOM.
  - Also we'll use an ***inverse mapping function*** to ***identify*** which customers in the input space is an ***outlier*** that are associated to a winning node, that.

13.4.4 Data preprocessing

We split the data sets into two subsets,

[1]. The sets that contain all the variables from customer ID to attribute number 14 (i.e. first 15 columns).

[2]. The class that is the variable that ***tells*** if the application of the customer was ***approved*** or ***rejected*** yes or no.

- So ***0*** is ***no***, the *application* was *not approved*. ***1*** means ***yes***, the *application* was *approved*.

❒ We create two sets of variables, so that we can clearly ***distinguish*** the ***customers*** whose applications are ***rejected*** and the customers who got ***approval***. It will be useful, for example, if we want to ***detect*** the ***fraudulent customers*** who got their applications ***approved***.

```
X = dataset.iloc[:, :-1].values
```

☞ ***-1*** is used, because we want ***all*** the ***columns*** except the ***last one***.

☞ To take the ***last column*** we used ***-1*** again.

```
y = dataset.iloc[:, -1].values
```

❒ ***X*** contains all the ***variables*** except the ***last one***, and ***y*** contains the ***last variable*** that tells if ***yes*** or ***no***, the application was *approved* or *rejected*.

✋ Note that we ***splitted*** our data-sets into ***X*** and ***y*** but it is ***not for the Supervised Learning***. We're not trying to make a model that will predict ***0*** or ***1*** in the end (that will make it supervised model).

☝ We're just doing this to ***make*** the ***distinction*** in the end between the customers who were ***approved*** and the customers who were ***rejected***.

☝ You will see that when we ***train*** our ***SOM*** we will only use ***X*** because we are doing some ***unsupervised deep learning***, that means that ***no dependent variable is considered***.

X - NumPy object array

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1.57762e+07	1	22.08	11.46	2	4	4	1.585	0	0	0	1	2	100	1213
1	1.57395e+07	0	22.67	7	2	8	4	0.165	0	0	0	0	2	160	1
2	1.56629e+07	0	29.58	1.75	1	4	4	1.25	0	0	0	1	2	280	1
3	1.56877e+07	0	21.67	11.5	1	5	3	0	1	1	11	1	2	0	1
4	1.57158e+07	1	20.17	8.17	2	6	4	1.96	1	1	14	0	2	60	159
5	1.55711e+07	0	15.83	0.585	2	8	8	1.5	1	1	2	0	2	100	1
6	1.57265e+07	1	17.42	6.5	2	3	4	0.125	0	0	0	0	2	60	101
7	1.56604e+07	0	58.67	4.46	2	11	8	3.04	1	1	6	0	2	43	561
8	1.56639e+07	1	27.83	1	1	2	8	3	0	0	0	0	2	176	538

Format Resize Background color

y - NumPy object array

	0
0	0
1	0
2	0
3	1
4	1
5	1
6	0
7	1
8	0

Format Resize

Feature matrix X and y before scaling.

❒ ***Feature scaling:*** Most of the time feature scaling is ***compulsory*** for ***deep learning***. Because there will be ***high computations*** to make, since we are dealing a ***high dimensional data set***, with lots of ***nonlinear relationships***.

☞ We're going to use ***normalization*** (instead of standardization). For ***RNN*** we also used ***normalization***. We'll get all our features between ***0*** and ***1***.

```
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range= (0, 1))
X = sc.fit_transform(X)
```

☞ ***fit*** means ***sc*** gets all the informations of ***X*** like the ***minimum***, the ***maximum***, well all the ***informations*** that it needs to apply ***normalization*** to x.

☞ After that we ***transform X***, that is to apply ***normalization*** to ***X*** and therefore we used ***fit_transform()*** on ***X***.

☞ Finally the ***fit_transform()*** returns, the normalized version of ***X***.

X - NumPy object array

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0.843	1	0.125	0.409	0.5	0.231	0.375	0.0556	0	0	0	1	0.5	0.05	0.0121
1	0.696	0	0.134	0.25	0.5	0.538	0.375	0.00579	0	0	0	0	0.5	0.08	0
2	0.389	0	0.238	0.0625	0	0.231	0.375	0.0439	0	0	0	1	0.5	0.14	0
3	0.488	0	0.119	0.411	0	0.308	0.25	0	1	1	0.164	1	0.5	0	0
4	0.601	1	0.0965	0.292	0.5	0.385	0.375	0.0688	1	1	0.209	0	0.5	0.03	0.00158
5	0.0217	0	0.0313	0.0209	0.5	0.538	0.875	0.0526	1	1	0.0299	0	0.5	0.05	0
6	0.644	1	0.0552	0.232	0.5	0.154	0.375	0.00439	0	0	0	0	0.5	0.03	0.001
7	0.379	0	0.675	0.159	0.5	0.769	0.875	0.107	1	1	0.0896	0	0.5	0.0215	0.0056
8	0.393	1	0.212	0.0357	0	0.0769	0.875	0.105	0	0	0	0	0.5	0.088	0.00537

Feature matrix X after scaling. (y is not scaled)

```
# importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# -------------- Data Preprocessing -------------------
# importing the Data-set
dataset = pd.read_csv("Credit_Card_Applications.csv")
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

#feature scaling
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range= (0, 1))
X = sc.fit_transform(X)
```

13.4.5 Training the SOM

We have two options.

[1]. The first option is to ***implement*** the ***SOM*** from ***scratch***

[2]. The second option is to use a ***code*** or a ***class*** made by ***another developer***,

❐ ***MiniSom 1.0: SOM*** doesn't have an implementation in ***Scikit-learn***. So we'll need to take it from another developer. Fortunately there is one excellent ***implementation*** of ***SOM***, called ***MiniSom 1.0***.

☞ ***MiniSom*** is a ***minimalistic*** and ***Numpy*** based implementation of the ***Self Organizing Maps (SOM)***. Developed by " ***Giuseppe Vettigli*** ".

☞ Remember, you will not always find libraries for a Deep-Learning model. Sometimes you will need to implement your own models from scratch.

☞ We downloaded the ***minisom.py*** (you can download from ***github***) in our working directory.

```
from minisom import MiniSom
```

➲ We import this ***MiniSom*** class from the ***minisom.py*** "**som**" is the object of this class. It is going to be trained only on ***X*** and ***not*** **y** because we're doing some unsupervised learning.

➲ We are trying to identify some ***patterns*** inside the ***independent variables*** that are contained in ***X***.

```
MiniSom(x, y, input_len, sigma=1, learning_rate=0.5, decay_function=None, random_seed=None)
```

✓ ***x=10, y=10*** defines the ***size*** of the ***Map***. Since out dataset is not too big we used $10 \times 10 = 100$ ***grid size***.

✓ ***input_len=15*** specifies the ***number of features*** in feature –matrix ***X***. We have 15 features in X. (However ***customer_id*** has no significance on the patterns but we're gonna keep it to identify the potential cheaters)

✓ ***sigma*** is the radius of the different neighborhoods in the grid. So we will keep its default value 1.0.

✓ ***learning_rate*** specifies how much the weights are updated during each iteration.

Higher learning rate, the convergence will be fast.

Lower learning rate, makes SOM build slowly. We're gonna keep it to default value 0.5

✓ ***decay_function*** can be used to improve the convergence. But here we're going to leave it to none and not use a decay.

✓ We don't need a ***random_seed***. That will be fine as well.

```
Initializes a Self Organizing Maps. x,y - dimensions of the SOM input_len - number of the elements of the vectors in input sigma - spread of the neighborhood function (Gaussian), needs to be adequate to the dimensions of the map. (at the iteration t we have sigma(t) = sigma / (1 + t/T) where T is #num_iteration/2) learning_rate - initial learning rate (at the iteration t we have learning_rate(t) = learning_rate / (1 + t/T) where T is #num_iteration/2) decay_function, function that reduces learning_rate and sigma at each iteration
        default function: lambda x,current_iteration,max_iter: x/(1+current_iteration/max_iter)
random_seed, random seed to use.
```

❒ ***Initialize the weight:*** Before train this ***som*** object on **X**, we have to ***randomly initialize*** the values of the ***weight vectors*** to small numbers ***close to zero***, but not zero.

☞ We do this by using a Class called ***random_weights_init*** from ***minisom.py***.

```
som.random_weights_init(X)
```

❒ ***Train the SOM:*** To train the self-organizing map on ***X***. We use the method called ***train_random***. We need to specify the ***no. of iterations*** and the ***dataset***.

```
som.train_random(data=X, num_iteration=100)
```

```
# Training the SOM
from minisom import MiniSom
som = MiniSom(x=10, y=10, input_len=15, sigma=1.0, learning_rate=0.5)
som.random_weights_init(X)
som.train_random(data=X, num_iteration=100)
```

Now basically our ***SOM*** is ***trained*** on our matrix of features ***X*** and the ***patterns*** are already ***identified***.
So its time to ***visualize*** the ***results***, to identify the outline neurons inside the map.

13.4.6 Visualize the SOM

```
# Visualize the result
from pylab import bone, pcolor, colorbar, plot, show
bone()
pcolor(som.distance_map().T)
colorbar()

markers = ['o', 's']
colors = ['r', 'g']
for i, x in enumerate(X):
    w= som.winner(x)  # notice lowercase 'x' used
    plot(
        w[0]+0.5,
        w[1]+0.5,
        markers[y[i]],
        markeredgecolor = colors[y[i]],
        markerfacecolor = 'None',
        markersize = 10,
        markeredgewidth = 2)
show()
```

❒ Here we are about to see a **2D grid**, that will contain all the ***final winning nodes***. For each of these winning nodes we will get the ***MID***; the ***Mean Inter-neuron Distance***.

☞ ***MID*** of a specific ***winning node*** is the ***mean*** of the ***distances*** of all the ***neurons around*** the winning node ***inside*** a ***neighborhood***. Neighborhood is defined by setting ***sigma=1.0*** is the radius of this neighborhood.

☞ ***Higher*** is the ***MID***, the more the ***winning node*** will be ***far away*** from it's ***neighbors***.

☞ Therefore the ***higher*** is the ***MID***, the ***more*** the ***winning node*** is an ***outlier***.

☝ Since the ***majority*** of the ***winning nodes*** respects the ***general rules*** (of the bank). We will detect the ***outliers/frauds*** that are ***far from*** this ***majority***, therefore far from the ***general rules***.

☝ Since for **each neuron** we will get the **MID**, so we will simply need to take the ***winning nodes*** that have the ***highest MID***.

👽 To visualize the result we only use ***colors***. The winning nodes will be ***colored*** by different ***colors*** in such a way that the larger is the ***MID***, the closer to ***white*** the color will be.

❒ We will not use ***matplotlib***, we're not plotting a ***classic graph***, like a ***histogram*** or a ***curve***. We're building a ***SOM***, and therefore we're gonna make it from ***scratch***.

☞ From ***pylab*** we import ***bone***, ***pcolor***, ***colorbar***, ***plot***, ***show*** functions.

```
from pylab import bone, pcolor, colorbar, plot, show
```

❒ ***Creating the map:***

```
bone()
pcolor(som.distance_map().T)
colorbar()
```

☞ ***bone*** function initialize the figure, the window that will contain the map.

☞ On the map we are going to add the ***information*** of the ***MID*** for all the ***winning nodes*** that the ***SOM identified***.

➢ We're *not going to add the figures* of all these ***MID***, instead we will use ***different colors*** corresponding to the ***different range values*** of the ***Mean Interneuron Distances***.

➢ We are gonna put the different winning nodes on the map, using `pcolor()`. We're going to add all the values of the MID for all the winning nodes of our SOM.

➲ To get these MID, we use `distance_map()` Method. It will return all the MID in one matrix.

➲ To get things in the right order for the ***pcolor()*** function, we just need to **take** the ***transpose*** of this matrix. That's why we used '**.T**'. T= transpose.

```
pcolor(som.distance_map().T)
```

☞ Actually if we select these two lines

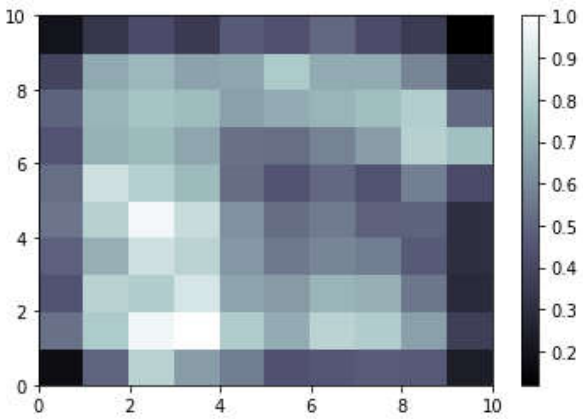
```
bone()
pcolor(som.distance_map().T)
```

and execute, well we get the SOM. With all the different colors corresponding to the MID.

❒ ***Describing the SOM using legends, colorbar:***

☞ To add one more information. We would like to see if the ***white color*** corresponds to a ***high MID*** or a ***low MID***. We use `colorbar()`

```
bone()
pcolor(som.distance_map().T)
colorbar()
```



☞ we can clearly see that the ***highest MIDs***, the highest Mean Interneuron Distances - MID, correspond to the ***white color***. And on the other hand, the ***smallest MID*** correspond to the ***dark colors***.

☞ So the ***white neurons*** are the ***frauds***, they are far from the ***general rules***. White winning nodes here have large MIDs and therefore they're outliers and accordingly potential frauds.

☞ Notice all these ***majority*** of points here with ***dark colors*** are ***close to each other*** because their ***MID*** is ***pretty low***. That means that all other ***winning nodes*** in the ***close-neighborhood*** of one ***central winning*** that creates ***clusters*** of ***winning nodes*** all close to each other.

❒ ***Adding Markers:*** We could stop here and get to the next step, to get the explicit list of the customers, by using the inverse mapping of those winning nodes.

☞ But we can do better by adding some markers. That marks the customers who got ***approval*** and the customers who got ***rejected***. Because the customers ***who cheated*** and ***got approval*** are more *relevant targets* to *fraud detection*.

☞ We're going to create two markers, some ***red circles*** and some ***green squares***.

➲ The ***red circles*** are going to correspond to the customers who ***didn't get approval***.

➲ And the ***green squares*** will correspond to the customers who ***got approval***.

```
markers = ['o', 's']
colors = ['r', 'g']
for i, x in enumerate(X):
    w= som.winner(x)  # notice lowercase 'x' used
    plot(
        w[0]+0.5,
        w[1]+0.5,
        markers[y[i]],
        markeredgecolor = colors[y[i]],
        markerfacecolor = 'None',
        markersize = 10,
        markeredgewidth = 2)
show()
```

```
'o' = circle,
's' = square,
'r' = red,
'g' = green.
```

☞ We're going to ***loop*** over all the ***customers*** and ***for each*** customer we're going to get the ***winning node*** and dependent on whether the ***customer*** got ***approval*** or ***not***.

- We actually need two looping variables that are going to be i and x (not capital X).
  - ***i*** is just going be the different values of all the indexes of our customer database, that is ***i = 0, 1, 2, 3, . . . , 689***.
  - ***x*** is going to be ***different vectors*** of ***customers***. So ***x*** will ***start*** by being equal to the ***vector*** that corresponds to the ***first customer***, then at the ***next iteration x*** will be equal to the ***second vector***, that corresponds to the ***second customer*** and down to the last customer.

☞ `enumerate()` return an enumerate object. The enumerate object yields ***pairs*** containing a ***count*** *(from start, which defaults to zero)* and a ***value yielded*** by the ***iterable argument***.

- ***enumerate*** is useful for ***obtaining*** an ***indexed list***: That's why we used it here so that we can get *(0, customer_1), (0, customer_2), (0, customer_3), . . . etc.*

```
(0, seq[0]), (1, seq[1]), (2, seq[2]), ...
```

- Inside the loop, `w= som.winner(x)` will get the ***winning node*** for the ***customer x*** (since each customer has a winning node).
- ***w[0]*** and ***w[1]*** are the ***coordinates*** for the ***winning-node***, and it is a ***Square*** of unit ***1***. So we add ***0.5*** to ***w[0]*** and ***w[1]***, so that our ***marker*** is in the ***center*** of the ***Winning-node-square***.

```
w[0]+0.5, w[1]+0.5,
```

☞ ***Using the dependent variable:*** Remember we didn't use ***dependent variable*** to train our ***model***, now we use it to ***mark*** our ***result***.

- Using ***y[i]*** in the ***loop*** generates either ***0*** or ***1*** according to ***reject/approve***. We use is to access our ***marker*** and ***color***.
- Notice both **circle** "**o**" and **color** "**r**" indexed ***0*** in ***markers*** and ***colors***, so that we can use them using same index. And that's enable us to use ***y[i]*** as index value.

```
markers[y[i]],
markeredgecolor = colors[y[i]],
```

- To specify ***fill-color***, *(no fill-color because some maker will overlay)* ***width*** of the marker, and ***size*** we use following:

```
markerfacecolor = 'None',
markersize = 10,
markeredgewidth = 2
```

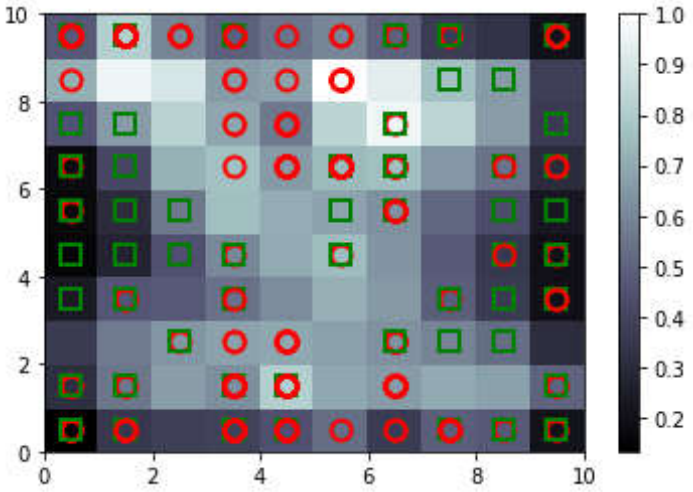
- Finally we use ***show()*** to display the plot, after the loop.

❒ Now not only we have the Mean Interneuron Distances but also we get the information of whether the customers got approval or didn't get approval for each of the winning nodes.

☞ We see some of the ***customers*** got ***rejected*** even though they are ***not fraud***. We also notice that some ***customer*** that ate ***potentially fraud*** got ***approved***.

☞ Now we have to do is ***catch*** these ***potential cheaters*** in the winning nodes who got ***approval***, because it's much more relevant to the bank to catch the ***cheaters*** who ***got away*** with ***this***.

We're going to use this map to ***catch*** these ***potential cheaters*** and to do this we're going to add just three lines of code.



13.4.7 Detecting the fraud

We don't have an inverse mapping function to directly get the list of customers from the coordinates of the winning nodes.

☞ However, there is another solution. It's to use a ***dictionary*** that we can obtain by using a ***method*** available in ***minison.py*** and that will contain all the ***different mappings*** from the ***winning nodes*** to the customers.

❒ First we get all these ***mappings*** (and ***their coordinates***) and then we'll use the ***coordinates*** of our ***outliers winning nodes*** that we identified (white ones), and that will give us the ***list of customers***.

❒ We have to concatenate the ***outliers winning nodes***. Since we actually identified two outlying winning nodes, we used the concatenate function to concatenate the two lists of customers so that we can have a whole list of the potential cheaters.

☞ ***win_map(X)*** will return the dictionary of all the mappings from the ***winning nodes*** to the ***customers***. ***mappins*** is the returned dictionary.

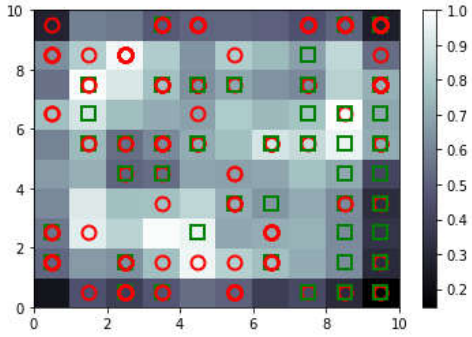
➢ We can see in variable explorer, the leftmost column of ***mappins*** consists the coordinates of the winning nodes in SOM.

➢ Each ***winning-node*** is a ***list of customers***, the sizes are shown in the 3rd column.

mappins - Dictionary (63 elements)

Key	Type	Size	Value
(0, 1)	list	39	[Numpy array, Numpy array, Numpy array, Numpy array, Numpy array, Nump ...
(0, 2)	list	8	[Numpy array, Numpy array, Numpy array, Numpy array, Numpy array, Nump ...
(0, 6)	list	3	[Numpy array, Numpy array, Numpy array]
(0, 8)	list	11	[Numpy array, Numpy array, Numpy array, Numpy array, Numpy array, Nump ...
(0, 9)	list	1	[Numpy array]
(1, 0)	list	1	[Numpy array]
(1, 2)	list	1	[Numpy array]
(1, 5)	list	7	[Numpy array, Numpy array, Numpy array, Numpy array, Numpy array, Nump ...
(1, 6)	list	6	[Numpy array, Numpy array, Numpy array, Numpy array, Numpy array, Nump ...

☞ Now from the ***SOM*** we need to identify the ***coordinates*** of the ***outlier nodes***.

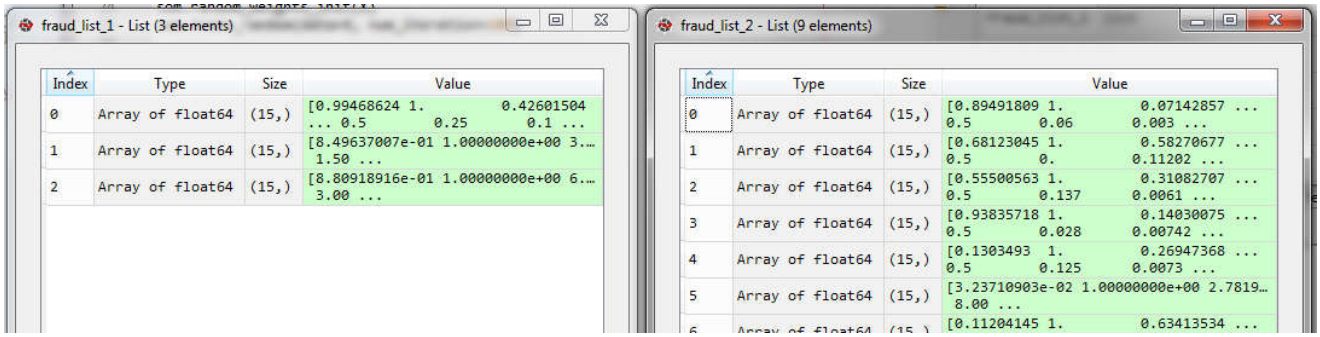


☞ From the map (from the right side) we see that, ***(8,6)*** and ***(1, 7)***, are the ***outlier-nodes*** from which some ***fraud-customers*** got approved. So we now concatenate them.

➢ We use `concatenate()`, it is a method from Numpy (np).

➢ However, for a ***single node***, we can ***get*** the ***list*** as follows:

```
mappins = som.win_map(X) # not 'x' its our dataset "X", capital X
fraud_list_1 = mappins[(8, 6)]
fraud_list_2 = mappins[(1, 7)]
```



NOTICE that above are ***2 lists***, to rescale/inverse-scale we need to convert them into ***NumPy array***.

➢ Now if we concatenate above two lists we can do as follows:

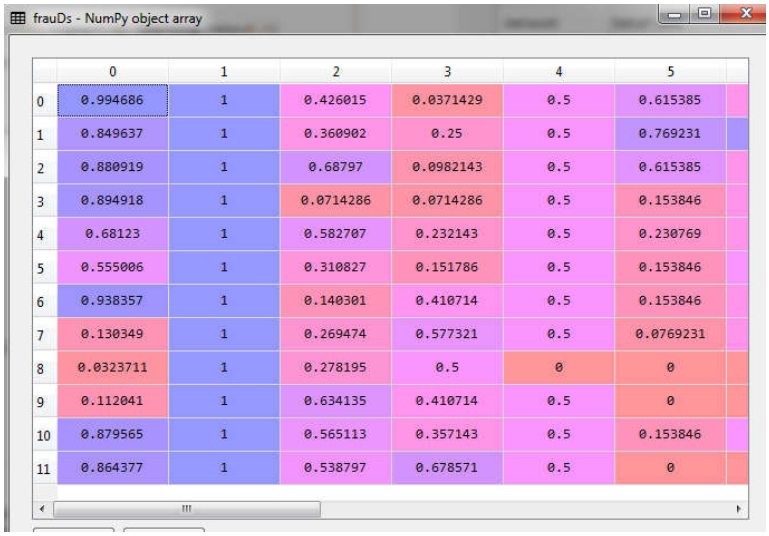
```
frauDs = np.concatenate((mappins[(8, 6)], mappins[(1, 7)]), axis=0)
```

axis=0 means, we concatenate the data ***vertically*** (i.e adding ***more rows***).

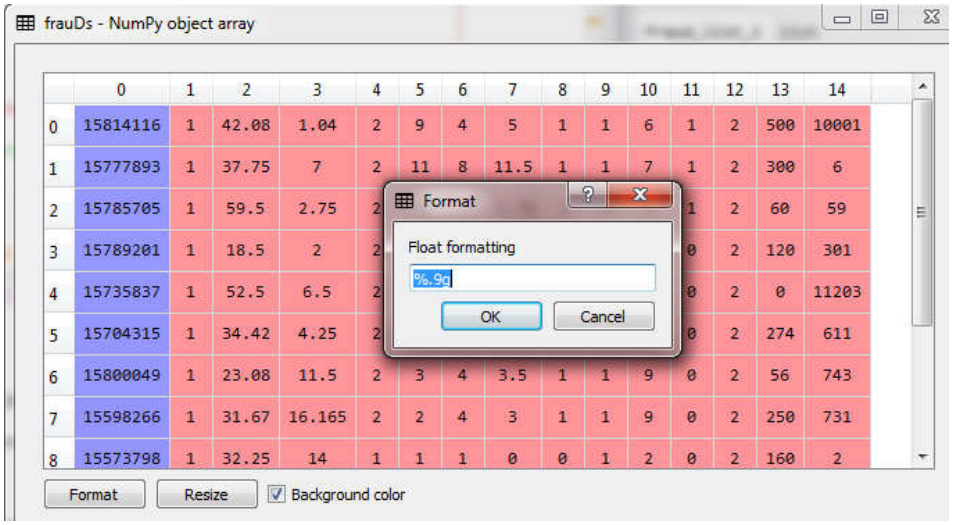
➢ It returns an array of floats of 3 + 9 = 12 total fraud-customers. Also notice using ***NumPy*** returns an ***array*** not ***list***, so it will be easy to ***inverse scale*** the ***resulted array*** to get the real values.

➢ ***Inverse scaling:*** Since out data is scaled, we need to undo it. That is done by following inverse scaling:

```
frauDs = sc.inverse_transform(frauDs)
```




```
# finding the Fruds
mappins = som.win_map(X) # not 'x' its our dataset "X", capital X
fraud_list_1 = mappins[(8, 6)]
fraud_list_2 = mappins[(1, 7)]
frauDs = np.concatenate((mappins[(8, 6)], mappins[(1, 7)]), axis=0)
frauDs = sc.inverse_transform(frauDs)
```



We ***reformat***, float-formatting to view the ***Customers-Id***.

We did our job, we gave the list of ***potential cheaters*** to the ***bank***, so now the bank side's got the ***ball***. Their ***analyst*** will ***investigate*** this list of potential cheaters and take in priority the ones that ***got approved*** to ***revise*** the application, and then by investigating ***deeper***, they will find out if the customer ***really cheated*** somehow.

All code at once (practiced)

```
# Self Organizing Maps (SOM)

# importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# -------------- Data Preprocessing ------------------
# importing the Data-set
dataset = pd.read_csv("Credit_Card_Applications.csv")
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

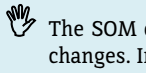
#feature scaling
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range= (0, 1))
X = sc.fit_transform(X)


# Training the SOM
from minisom import MiniSom
som = MiniSom(x=10, y=10, input_len=15, sigma=1.0, learning_rate=0.5)
som.random_weights_init(X)
som.train_random(data=X, num_iteration=100)


# Visualize the result
from pylab import bone, pcolor, colorbar, plot, show
bone()
pcolor(som.distance_map().T)
colorbar()

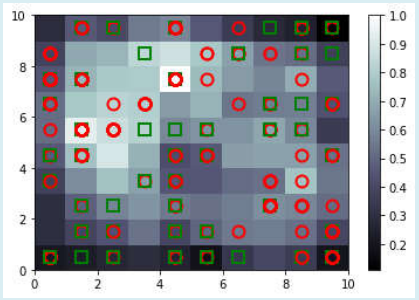
markers = ['o', 's']
colors = ['r', 'g']
for i, x in enumerate(X):
    w= som.winner(x)
    plot(
        w[0]+0.5,
        w[1]+0.5,
        markers[y[i]],
        markeredgecolor = colors[y[i]],
        markerfacecolor = 'None',
        markersize = 10,
        markeredgewidth = 2)
show()

# finding the Fruds
mappins = som.win_map(X) # not 'x' its our dataset "X", capital X
fraud_list_1 = mappins[(8, 6)]
fraud_list_2 = mappins[(1, 7)]
frauDs = np.concatenate((mappins[(8, 6)], mappins[(1, 7)]), axis=0)
frauDs = sc.inverse_transform(frauDs)
```



The SOM changes every time, when we build it, so the position of outlier nodes also changes. In this case

```
frauDs = np.concatenate((mappins[(1, 5)], mappins[(4, 7)]), axis=0)
```



Chapter 13: Part 5

Deep Learning

SOM: Hybrid Model

Python Implementation

13.5.1 Problem description

Here we'll make a ***Hybrid Deep Learning Model***. We'll combine two deep learning models **ANN** and **SOM** (**supervised** and **unsupervised**).

- Our ***data-set*** will be, the *credit card applications* dataset to identify the ***frauds***.
- The idea is to make an even more ***advanced deep learning model*** where we can ***predict*** the ***probability*** that each customer ***cheated***.

❒ ***It takes two parts:***

- 🛠 In the first part, we'll make the unsupervised deep learning branch of our hybrid deep learning model using SOM.
- 🛠 And in the second part, we'll make the supervised deep learning branch using ANN.
- ☝ And, in the end, we'll get this ***hybrid deep learning model*** composed of both ***unsupervised*** and ***supervised*** deep learning.

❒ We will use the ***self-organizing map*** exactly as we did ***previously***, to identify the ***fraud***. So, there will be nothing new about that.

- ☞ But then, the ***challenge*** is to use the ***results*** of this ***self-organizing*** map to ***ANN*** model. ***ANN*** will take, as ***input***, the ***results*** given by your ***SOM***.
- ☞ The challenge is to ***combine two models***. In the end, what you must obtain is a ***ranking*** of the ***predicted probabilities*** that ***each*** customer ***cheated***.

NOTE that, you will get very ***small probabilities***, that's normal. It's because there are few ***frauds*** identified by the ***SOM***, but that ***doesn't matter***. What's important is that you get this ***ranking*** of ***probabilities***.

13.5.2 SOM part

- ☞ We run the SOM code to get the map. Execute all the code from beginning to ***show()*** method.
- ☞ After obtaining the map, we choose a ***threshold*** for ***outlier neuron*** and we find and select those neurons.
- ☞ We then ***concatenate*** those data.

✋ NOTE that we execute the following code after selecting the outlying neurons. ***(8, 6)***, ***(1, 7)*** are just for example, yours could be different.

```
# finding the Fruds
mappins = som.win_map(X) # not 'x' its our dataset "X", capital X
fraud_list_1 = mappins[(8, 6)]
fraud_list_2 = mappins[(1, 7)]
frauDs = np.concatenate((mappins[(8, 6)], mappins[(1, 7)]), axis=0)
frauDs = sc.inverse_transform(frauDs)
```

13.5.3 ANN part & Prediction

Creating the Feature matrix: We'll take all the columns except first column, i.e. ***customer_id***.

- ☝ NOTE: the last column ***approved/rejected*** is not a dependent variable here (it is done by the ***Bank*** they don't know who is fraud). We create the ***dependent*** variable from ***SOM output***.

```
# creating the matrix of features
cuStomers = dataset.iloc[:, 1:].values
```

❒ ***Creating the dependent variable for ANN:*** This dependent variable is about fraud, ***0 = no fraud*** and ***1 = fraud***.

- ☞ Since from ***SOM*** we extracted some ***frauds***, we can use them as our ***dependent variable***.

☞ We first create a vector of **`690 0`**'s i.e. at first we assume that there is no ***cheaters/fraud***. Then we ***generate*** 1 for the Id's that ***falls*** into our ***fraud list*** from ***SOM***.

☞ In a loop we check each customer that are inside this list of fraud.

➢ ***`dataset.iloc[i, 0]`*** means ***i***th row and 1st column (i.e. ***`customer_id`***), we don't need ***`.values`***, it used for ***NumPy array***.

```
# creating the dependent variable
is_fraud = np.zeros(len(dataset)) # creating 690 size vector of zeros
for i in range(len(dataset)):
    if dataset.iloc[i, 0] in frauDs:
        is_fraud[i]=1
```

❒ ***Creating ANN model:***

☞ ***Feature scaling:*** We replace ***`X_train`*** by ***`cuStomers`*** and remove X_test.

```
# ------------------ Feature-Scaling ----------------------
from sklearn.preprocessing import StandardScaler
#  y dependent variable, need not to be scaled: categorical variable, 0 and 1
st_x= StandardScaler()
cuStomers= st_x.fit_transform(cuStomers)    # replace "X_train" by 'cuStomers'
```

➢ We remove ***2nd hidden-layer*** (we keep our model simple for learning purpose, however in general this model could be far more complex.)

➢ We set ***`units = 2`*** instead of ***6*** (no. of neurons).

➢ Since we have ***15*** columns, so our ***`input_dim = 15`***.

☞ Also in ***`ann_classifier.fit()`*** we use as feature-matrix and as output/dependent variable/data. We also change the ***`batch_size`*** and no. of ***epochs***

```
ann_classifier.fit(cuStomers, is_fraud, batch_size= 1, epochs= 2)
```

NOTE, do not train your ***Deep Learning models*** in ***too many epochs*** if you have ***few observations*** and ***few features***.

```
# ----------------------------------  Creating ANN model ------------------------------------------

    # 1. importing "keras" libraries and packages
# from tensorflow import keras
import keras # using TensorFlow backend
from keras.models import Sequential
from keras.layers import Dense

    # 2. initialize the ANN
ann_classifier = Sequential()

    # 3. Add the "input-layer" and  "first Hidden-layer"
# ann_classifier.add(Dense(output_dim = 6, init = "uniform", activation = "relu", input_dim = 11))
ann_classifier.add(Dense(units = 2, kernel_initializer = "uniform", activation = "relu", input_dim = 15))

    # 4. Add the "second Hidden-layer"
# ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))

    # 5. Add the "output-layer"
ann_classifier.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))

    # 6. Compile the ANN
ann_classifier.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"])

    # 7. Train the model: fit the ANN to Training-set (batch_size and epoch)
ann_classifier.fit(cuStomers, is_fraud, batch_size= 1, epochs= 2)



# ---------------------------------- Part 3 : Predictions  ------------------------------------------

# Predicting the probabilities of frauds
y_prd = ann_classifier.predict(cuStomers)
```

13.5.4 Ranking the customers

❒ We will sort these probabilities. But before that, we add ***customer_id*** to ***y_pred*** because it will be easy to identify the fraud.

☞ It will be ***2D array*** of 1st column hold the ***ids*** and ***2nd column*** is probabilities of being a fraud.

🛠 ***Concatenate the columns:***



☞ Notice ***y_pred*** is a NumPy array of float, of size ***(690, 1)***. So to concatenate the ***customer_id***, it need to be a ***NumPy array*** of float.

☞ ***dataset.iloc[:, 0:1].values***, selects the 1st column i.e. ***customer_id*** from the dataset. ***0:1*** and ***.values*** is used to convert it to ***NumPy array***.

☞ Also we changed the ***axis***. Since we now concatenate side-by-side, we set ***axis=1.***

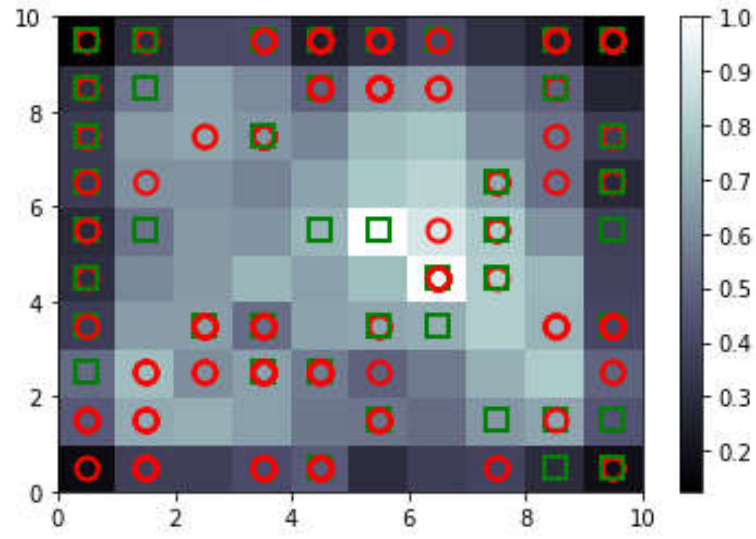
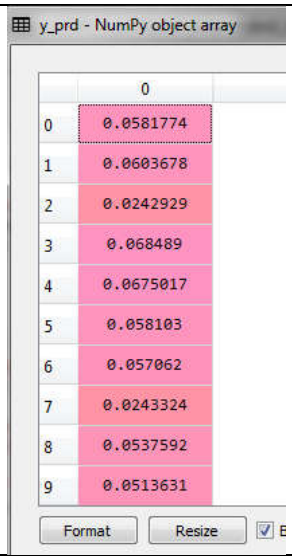
```
# concatenate columns to add customer_id
y_prd = np.concatenate((dataset.iloc[:, 0:1].values, y_prd), axis=1)
```

❒ Now ***sorting*** is ***bit-Tricky***, because ***NumPy*** will ***sort*** both columns. We don't want that, we just want to ***sort*** 2nd column of ***y_pred***, i.e. the probabilities only.

☞ In ***Excel*** it is ***easy***, the ids always ***stay paired*** if we ***sort*** the ***probability***. In NumPy it is different.

```
# sorting trick
y_prd = y_prd[y_prd[:, 1].argsort()] # y_prd[:, 1] selects the 2nd column of y_pred
```

☞ ***y_prd[:, 1]*** selects the 2nd column of ***y_pred***.

SOM	***Prediction probabilities***
	


```
# ---------------------------- Part 3 : Predictions  ----------------------------------

# Predicting the probabilities of frauds
y_prd = ann_classifier.predict(cuStomers)

# concatenate columns to add customer_id
y_prd = np.concatenate((dataset.iloc[:, 0:1].values, y_prd), axis=1)

# sorting trick
y_prd = y_prd[y_prd[:, 1].argsort()] # y_prd[:, 1] selects the 2nd column of y_pred
```

All code at once (practiced version)

```
# --------- mega case study : Make a Hybrid Deep learning model ----------

# importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# -------------- Data Preprocessing -------------------
# importing the Data-set
dataset = pd.read_csv("Credit_Card_Applications.csv")
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

#feature scaling
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range= (0, 1))
X = sc.fit_transform(X)



# ------ Part 1 : Identify the frauds using Self Organizing Maps (SOM) ---------
# Training the SOM
from minisom import MiniSom
som = MiniSom(x=10, y=10, input_len=15, sigma=1.0, learning_rate=0.5)
som.random_weights_init(X)
som.train_random(data=X, num_iteration=100)


# Visualize the result
from pylab import bone, pcolor, colorbar, plot, show
bone()
pcolor(som.distance_map().T)
colorbar()

marKers = ['o', 's']
coLors = ['r', 'g']
for i, x in enumerate(X):
    w= som.winner(x)
    plot(
        w[0]+0.5,
        w[1]+0.5,
        marKers[y[i]],
        markeredgecolor = coLors[y[i]],
        markerfacecolor = 'None',
        markersize = 10,
        markeredgewidth = 2)
show()

# finding the Fruds
mappins = som.win_map(X) # not 'x' its our dataset "X", capital X
fraud_list_1 = mappins[(5, 5)]
fraud_list_2 = mappins[(6, 4)]
frauDs = np.concatenate((mappins[(5, 5)], mappins[(6, 4)]), axis=0)
frauDs = sc.inverse_transform(frauDs)

# saving the list to a csv
# save numpy array as csv file
"""
from numpy import asarray
from numpy import savetxt
# define data
data = frauDs
# save to csv file
savetxt('frauds.csv', data, delimiter=',')
"""


# part 2: Going from Unsupervised to Supervised Deep Learning

# creating the matrix of features
```

```python
cuStomers = dataset.iloc[:, 1:].values


# creating the dependent variable
is_fraud = np.zeros(len(dataset)) # creating 690 size vector of zeros
for i in range(len(dataset)):
    if dataset.iloc[i, 0] in frauDs:
        is_fraud[i]=1


# creating ANN model
# ------------------ Feature-Scaling ---------------------
from sklearn.preprocessing import StandardScaler
#  y dependent variable, need not to be scaled: categorical variable, 0 and 1
st_x= StandardScaler()
cuStomers= st_x.fit_transform(cuStomers)    # replace "X_train" by 'cuStomers'




# ---------------------------------  Creating ANN model ------------------------------------------

    # 1. importing "keras" libraries and packages
# from tensorflow import keras
import keras # using TensorFlow backend
from keras.models import Sequential
from keras.layers import Dense

    # 2. initialize the ANN
ann_classifier = Sequential()

    # 3. Add the "input-layer" and  "first Hidden-layer"
# ann_classifier.add(Dense(output_dim = 6, init = "uniform", activation = "relu", input_dim = 11))
ann_classifier.add(Dense(units = 2, kernel_initializer = "uniform", activation = "relu", input_dim = 15))

    # 4. Add the "second Hidden-layer"
# ann_classifier.add(Dense(units = 6, kernel_initializer = "uniform", activation = "relu"))

    # 5. Add the "output-layer"
ann_classifier.add(Dense(units = 1, kernel_initializer = "uniform", activation = "sigmoid"))

    # 6. Compile the ANN
ann_classifier.compile(optimizer = "adam", loss = "binary_crossentropy", metrics= ["accuracy"])

    # 7. Train the model: fit the ANN to Training-set (batch_size and epoch)
ann_classifier.fit(cuStomers, is_fraud, batch_size= 1, epochs= 2)




# ---------------------------------- Part 3 : Predictions  ------------------------------------------

# Predicting the probabilities of frauds
y_prd = ann_classifier.predict(cuStomers)

# concatenate columns to add customer_id
y_prd = np.concatenate((dataset.iloc[:, 0:1].values, y_prd), axis=1)

# sorting trick
y_prd = y_prd[y_prd[:, 1].argsort()] # y_prd[:, 1] selects the 2nd column of y_pred

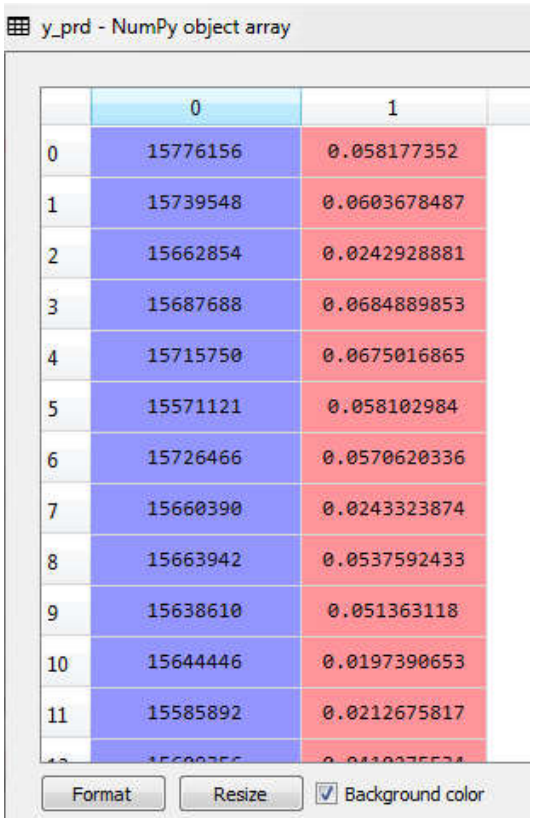
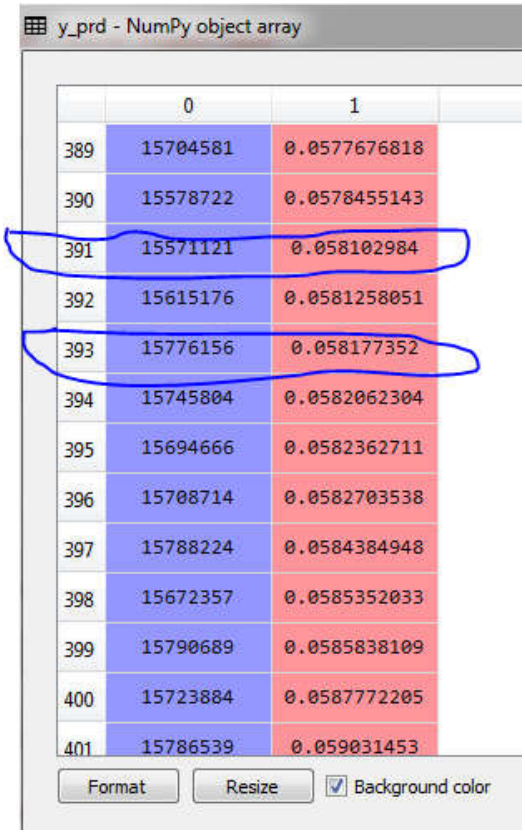

# python prctc_mega_Hybrid_SOM_ANN.py
```

```
2022-07-17 14:38:08.982247: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]
Ignore above cudart dlerror if you do not have a GPU set up on your machine.
Epoch 1/2
690/690 [==============================] - 2s 987us/step - loss: 0.4558 - accuracy:
0.9623
Epoch 2/2
690/690 [==============================] - 1s 859us/step - loss: 0.1636 - accuracy:
0.9623
Out[13]: <keras.callbacks.History at 0x1cf65cd0>

In [14]: y_prd = ann_classifier.predict(cuStomers)
22/22 [==============================] - 0s 1ms/step

In [15]: y_prd = np.concatenate((dataset.iloc[:, 0:1].values, y_prd), axis=1)

In [16]:
    ...: y_prd = y_prd[y_prd[:, 1].argsort()] # y_prd[:, 1] selects the 2nd column
of y_pred
```

Un-sorted Customer_id & Probability	***Sorted Customer_id & Probability***
	

y_prd - NumPy object array

	0	1
0	15776156	0.058177352
1	15739548	0.0603678487
2	15662854	0.0242928881
3	15687688	0.0684889853
4	15715750	0.0675016865
5	15571121	0.058102984
6	15726466	0.0570620336
7	15660390	0.0243323874
8	15663942	0.0537592433
9	15638610	0.051363118
10	15644446	0.0197390653
11	15585892	0.0212675817

Format Resize Background color

y_prd - NumPy object array

	0	1
389	15704581	0.0577676818
390	15578722	0.0578455143
391	15571121	0.058102984
392	15615176	0.0581258051
393	15776156	0.058177352
394	15745804	0.0582062304
395	15694666	0.0582362711
396	15708714	0.0582703538
397	15788224	0.0584384948
398	15672357	0.0585352033
399	15790689	0.0585838109
400	15723884	0.0587772205
401	15786539	0.059031453

Format Resize Background color

Conclusion: The purpose here to know how to ***combine*** two ***models*** to create a ***Hybrid model***.

Chapter 14 : Part 1

Deep Learning

BM: Boltzmann Machines

`Introduction`

14.1.1 What we will learn in this Chapter

[1]. ***The Boltzmann Machine:*** We'll talk about the **Boltzmann machine**, and how it's ***structured***, how it ***works***, its ***architecture***. Here we're talking about the actual Boltzmann machine, the big one, not just the ***RBM (Restricted Boltzmann machine)***.

[2]. ***Energy-Based Models (EBM):*** We're going to be talking about **Energy-Based Models (EBM)**, that will help you understand the inspiration for ***Boltzmann machines***, where it comes from, We're going to understand what goes into their ***architecture***, what goes into their background that allows them to come up with the ***results*** that they are able to come up with.

- ✓ And how we actually control them in terms of weights and what that means in terms of energy.

[3]. ***Restricted Boltzmann Machine (RBM):*** Here we're going to be talking about the ***Restricted Boltzmann machine***, or the ***RBM***. This architecture was proposed to solve the problem of ***computational*** issues with the ***Boltzmann machine***.

- ✓ We'll also ***walkthrough*** an ***example*** to know how an RBM ***trains*** and how an RBM is applied on a ***practical*** example.
- ✓ It will help us on implementation the BM in Python.

[4]. ***Contrastive Divergence (CD):*** Here we'll talk about ***Contrastive Divergence***, the algorithm that allows us to find the ***weights*** for research in ***Boltzmann machines***.

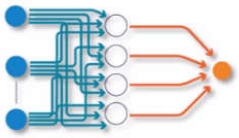
[5]. ***Deep Belief Networks (DBN):*** Here we'll talk about ***Deep Belief Networks*** or ***DBNs*** in very short.

[6]. ***Deep Boltzmann Machines (DBM):*** We'll discuss the ***DBMs***, ***Deep Boltzmann Machines*** very shortly.

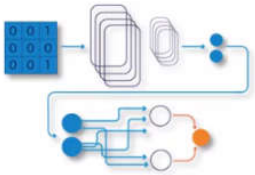
14.1.2 The Boltzmann Machine (BM)

Boltzmann Machine (BM) is very ***advanced*** topic. We've discussed ***ANN***, ***CNN***, ***RNN*** for ***supervised*** deep learning, also we've discussed ***SOM*** which is an ***unsupervised*** technique. Now ***BM*** is also ***unsupervised*** technique. So let's have a look at all of these in a diagrammatical representation.

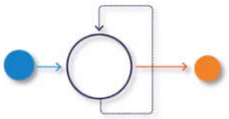
[1]. So here we've got an ***artificial neural network (ANN)***. With the ***input*** layer, and ***multiple hidden*** layers and an ***output*** layer.



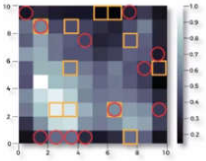
[2]. And then we've got a ***convolutional neural network (CNN)*** with the ***convolution*** layer, the ***pooling*** layer, the ***flattening*** layer and then an ***ANN*** sitting on the end.



[3]. Then we've got the ***recurrent neural network (RNN)*** where the ***hidden*** layer feeds back into itself, and therefore facilitates analysis of ***temporal data***.

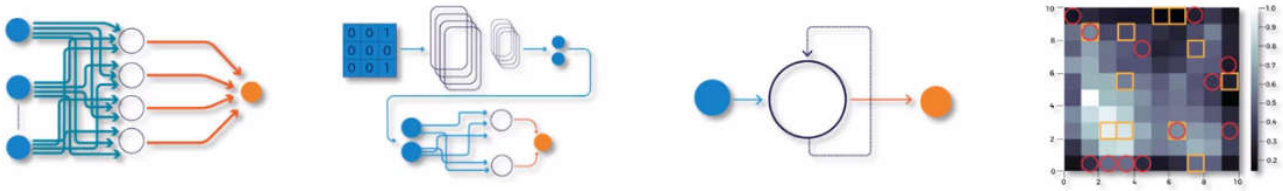


[4]. Here we've got the ***self organizing map (SOM)***, which helps you ***reduce*** your ***dimensionality*** and represents something, it represents your data in a more understandable way.



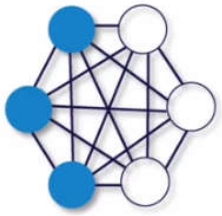
❒ ***Directed Models:*** Now, even though the SOM is a unsupervised type of deep learning model, but ANN, CNN, RNN & SOM all these four are actually ***directed models***.

☞ There is a direction in which the model operates. For example in SOM, we've got the input-data and then the information goes through the nodes creates an output map.



ANN, CNN, RNN & SOM all these four are Directed Models

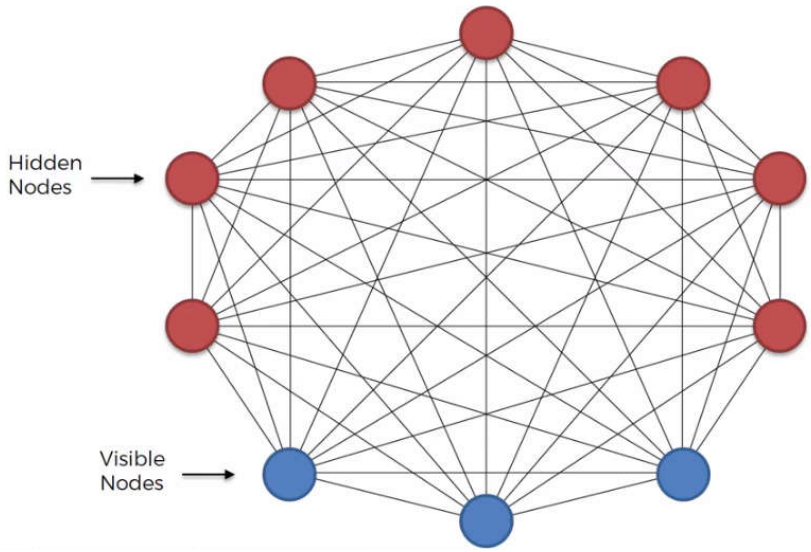
❒ ***Boltzmann machines (Undirected Models):*** Boltzmann machines are *Unsupervised Deep Learning Model* and it is an ***Undirected Model***. All kind of *Boltzmann machines* are *undirected models*.



☞ In the diagram, notice that there are actually no arrows in the connections between nodes, all these connections go **both ways**.

☞ Let's have a look a bit closer at a Boltzmann machine and understand what's going on. Following is a ***Boltzmann machine (BM)***, we've got hidden nodes in red and we've got visible nodes here in blue. (Actually there is no difference between ***hidden nodes*** and ***visible nodes.*** In ***BM*** all nodes connected to each other. For learning purpose we used two colors.).

☞ Here, we've got ***three visible nodes*** (input nodes) at the bottom, and we've got seven hidden nodes.



❒ ***Three things to notice:***

[1]. This model *doesn't* have an *output layer*. There's an *input layer*, there's a *hidden layer*, but *no output layer*.

[2]. All nodes connected to each other. *Everything is connected* (hyper connected) to everything, there's **no specific layering** per say.

[3]. There is *no direction* in these *connections* (Undirected). All connections in ***BM*** are ***bi-directional***. The connection happens both ways.

❒ ***The Visible Nodes (input nodes) are connected each other:*** Notice that the visible nodes are all connected between each other,

What's the point of connecting visible nodes? Once you input data, it's fixed, right?

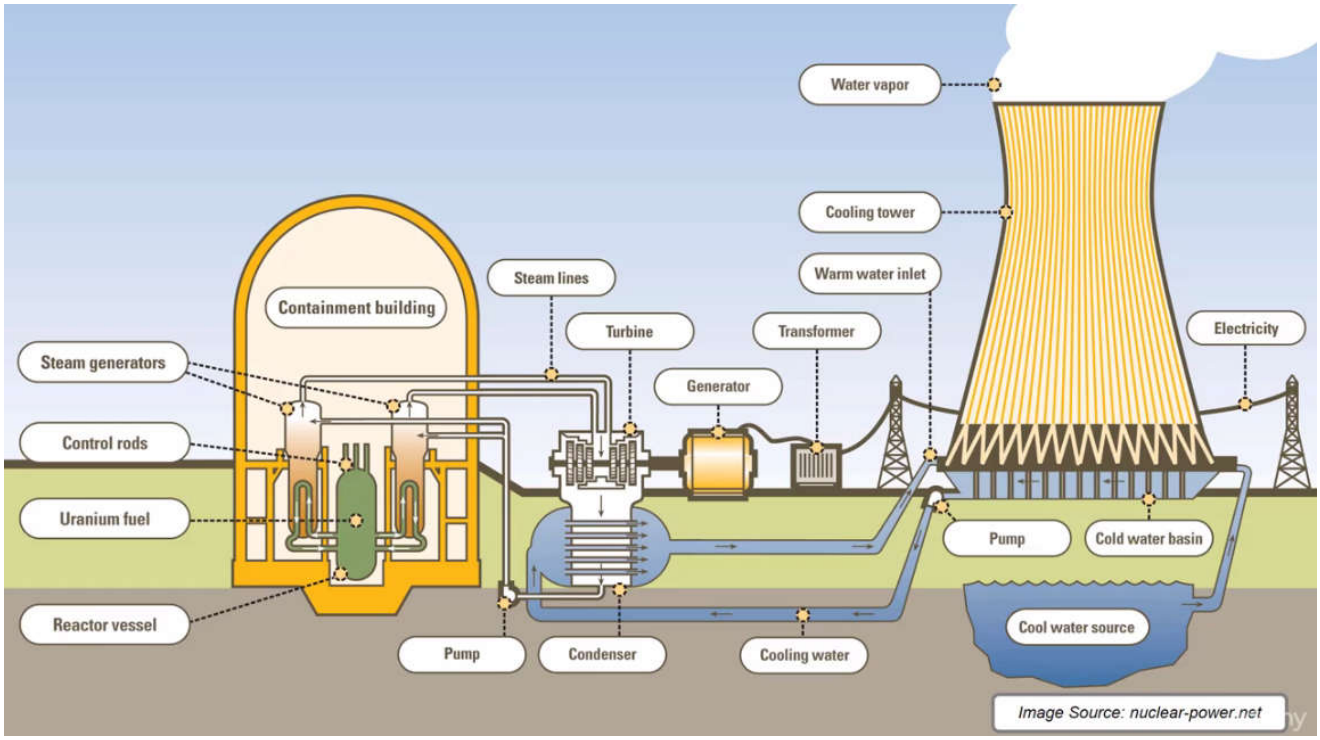
☞ You have a *row* of *data* and you just input it to the *model* and there's no point in these *connections* between these *visible nodes*. You're not going to be *adjusting* the *weights* or *training*, because that data is *fixed*.

And that's the Twist!! ***Boltzmann machines*** are **fundamentally different** to all other algorithms.

☝ ***Boltzmann machines*** don't just **expect input** data they also ***generate data***. They ***generate information*** in all of available nodes, regardless of whether there's and **input node** or a **hidden node**. For a ***Boltzmann machine*** **all of these nodes** are the ***same***.

☝ For a ***BM*** this whole thing is a *system*, it's generating *states of this system*.

❒ ***Analogy of nuclear power plant:*** The best way to think about BM is through an example of a ***nuclear power plant*** that once given by ***Geoffrey Hinton***.



☞ Let's say that there are certain things that we can measure about the nuclear power plant, for instance, ***temperature*** inside the ***containment unit***, how quickly ***pump turbine*** is ***spinning***, the ***pressure*** inside the ***turbine pump***, how much ***electricity*** it is ***outputting*** etc.

☝ But at the same time there could be a ***lot*** of ***odd things*** that ***we're not measuring***. For example, the ***speed*** of the ***wind***, the ***moisture*** of the ***soil*** (where pump operates), ***thickness*** of the ***cooling tower*** wall at a height 20 meters. So there can be lots of ***different parameters*** of the nuclear power plant that we're ***not measuring***,

☝ ***All*** these ***parameters*** (those we are measuring and we aren't measuring) all together, ***form one system*** and they all ***work together***, and that is what the ***Boltzmann machine*** represents.

☝ The BM is a ***representation*** of a ***certain system*** (in our case, a nuclear power plant). The ***visible nodes*** are just merely things that we can and do ***measure***, and the ***hidden nodes*** are things that we ***can't*** or ***don't measure***.

☞ The ***Boltzmann machine (BM)*** is capable of ***generating values*** instead of just waiting for us to ***input values***.

☞ It just generates ***different states*** of our system. For example: in case of ***nuclear power plant*** BM looks at a state with a certain ***temperature***, certain ***speed*** of the ***wind***, and the ***moisture*** or the ***pressure*** in the ***pump***.

That is how the ***Boltzmann Machine*** works, it's not a ***Deterministic Deep Learning*** model, it's a ***Stochastic Deep Learning*** model or a ***Generative Deep Learning*** model.

14.1.3 How BM works

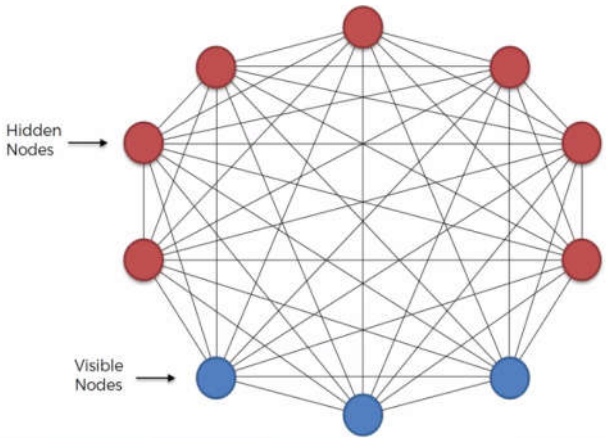
In a simple word, ***BM*** describes ***different possible states*** of our system. It can ***generate*** possible scenarios from the given data.

- For example: If we give our ***nuclear power plants normal state data*** (how it operates in normal condition) to a **BM**, then the **BM** recognize the **normal state** for the **power plant**. And also BM can generate ***abnormal state***, such as ***"Core-meltdown"*** or ***"Nuclear Explosion"*** and shows us the corresponding parameters for the scenario.
  - This could be very helpful to avoid such ***fatal failure*** and ***accidents***.

❒ We feed in ***BM*** our ***training data*** (thousands of rows that we have) as the ***inputs*** to help it ***adjust*** the ***weights*** of this system accordingly, so that it actually ***resembles*** our ***system***.

☞ ***For example:*** ***BM*** doesn't ***resemble*** any ***nuclear power plant*** in the ***real-world***, it actually learns from what we feed it in. It learns how, what are the ***possible connections*** between all of these ***parameters***.

❒ ***BM*** becomes a ***system***, a machine that represents our ***system*** (our nuclear power plant).



☞ Once ***BM*** done the ***learning***, once all of these ***weights*** are ***adjusted***, BM ***understands*** how all ***parameters interact*** with ***each other*** and what kind of constraints should exist between them in order for this system to be the system that we're modeling.

☞ Once that's all done we can use the BM to ***monitor*** our ***nuclear power plant***.

☞ Because we've modeled it using ***good behavior***, that hasn't led to any ***meltdown*** or any ***explosions***. BM knows what is ***normal*** for the ***nuclear power plant***.

☞ Then the BM will help us ***understand*** what is ***abnormal behavior***.

☞ In **real-life** we cannot really **model** a **nuclear meltdown** through **supervised learning**, we don't really have that ***luxury*** of having all of this training data with ***actual nuclear meltdowns***.

- You have to ***model*** it in an ***unsupervised manor***, and that's exactly what a ***Boltzmann machine*** does.

☞ ***Learning*** through good ***examples***, it understands how the system works in it's ***normal state***, and through that it helps us model what the system would look like in an ***abnormal state*** and model those ***scenarios*** and recognize those ***scenarios***, and therefore ***monitor the nuclear power plant***.

That now explains the ***design*** of these ***Boltzmann machines***:

⇨ Why we don't have an ***output layer*** because we're not ***outputting*** any value. We are creating a ***model*** that ***describes*** our ***system***.

⇨ Why all of these ***nodes*** are ***interconnected***. Because all of the ***parameters*** are ***interconnected*** with each other. For example: the ***speed of turbine*** and the ***moisture in the air*** on that day are connected.

⇨ The ***directionality***. The ***visible nodes*** and ***hidden nodes*** all of the nodes ***are equal*** in a ***BM***. All of these ***parameters*** are ***interconnected*** and therefore it had to be ***undirected***. All the **connections** are going **both ways**.

14.1.4 Energy based Models (EBM)

Energy-Based Models helps us to understand the stochastic processes that describe systems in BM.

❒ ***Boltzmann distribution:*** Notice the following equation. This is the ***Boltzmann distribution***. That's why the method is called Boltzmann machines (BM). This ***equation*** comes from ***physics***, and it is used in the ***sampling distribution*** for the ***Boltzmann machine***.

$$p_i = \frac{e^{-\varepsilon_i/kT}}{\sum_{j=1}^{M} e^{-\varepsilon_j/kT}}$$

☞ Using this ***Boltzmann distribution*** the ***Boltzmann machine*** constantly creates ***different states*** of our ***system***.

☞ From the equation we see that ***Boltzmann distribution*** talks about ***probability***. It describes the probability p_i of a certain state of a system. Assuming there is total M states for the system.

i = is a state. Represents different states.

ε_i = energy of the state i.

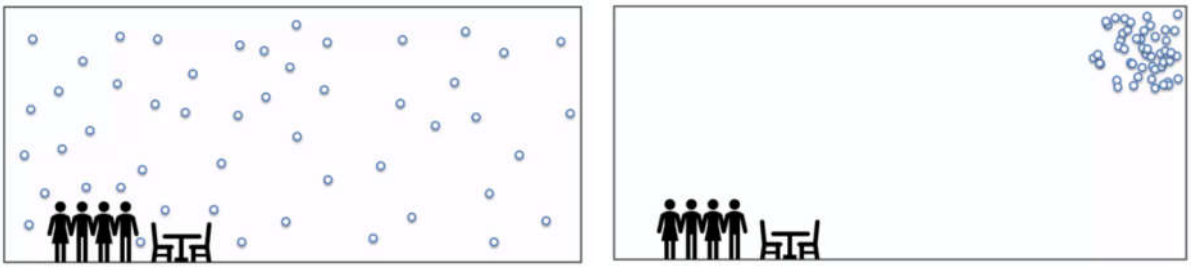
p_i = probability of your system being in state i.

K = is constant and

T = is the temperature of your system.

☞ The important thing is $e^{-\varepsilon_i/KT}$ means the ***higher*** the ***energy*** of a certain state at ***fixed temperature***, the ***lower*** the ***probability***. So, ***probability*** is **inversely** related with **energy**.

☞ In a ***thermodynamical system***, we're looking at the system at a ***given temperature***, for instance a ***gas***. The equation says that, the ***probability*** of your system being in a ***certain state*** is ***inversely related*** to the ***energy*** of your system in that state.



❒ To explain ***Boltzmann distribution*** consider we've got a room, and we have a ***gas***, for instance ***air***.

- Now the question is how the gas is ***distributed*** across the whole room?
- Why is the gas, not all in one corner?

📣 The answer is: They could be anywhere. **Statistically**, they could end up in **one corner**, in any room, at any given point in time. This is one of the possible states of this system.

☝ But the Boltzmann distribution is saying that the ***probability*** of that state **"all gas in one corner"** occurring is ***very low***.

⇨ Because the ***energy*** in that state (all gas in one corner), would be ***very high***.

⇨ Since the ***molecules*** are very ***close*** to each other, making a lot of ***chaos*** and they would be moving very ***quickly***. Given the ***constant temperature*** of the room.

⇨ What we normally observe is gas (air molecules are equally spaced apart) are distributed all over the room. Because it is the **lowest energy state** for that ***system***.

☝ It applies to everything. For instance, if you ***dropped*** some ***ink*** into water, it will start ***spreading evenly*** in all directions. It won't form a ***star***, because that's ***not*** the ***lowest energy state***.

⇨ On the other hand, if you drop, if you put a drop of oil into water, then it won't start spreading, because that is the lowest energy state for that specific system.

❒ ***Boltzmann machines*** are constructed with a ***similar concept***. The ***energy*** is defined in BMs through the ***weights*** of ***synapses***.

☞ Once the system is ***trained up***, once the ***weights*** are ***set***, the ***system***, based on those ***weights***, will always try to find the ***lowest energy state*** for ***itself***.

☞ It has lots of different options, but the weights will dictate what is the lowest energy state for the system, and it will constantly try to get to the lowest energy state possible.

☝ That's why they're called ***Energy-Based Models***.

14.1.5 Energy function for a restricted Boltzmann machine (RBM)

Here's an example of an energy function for a ***`Restricted Boltzmann Machine (RBM)`***. A bit ***different*** to the case of the ***Boltzmann distribution energy***, because it's ***not a gas***. You can see that it is defined through the weight.

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j v_i w_{i,j} h_j$$

a_i, b_j are ***biases*** in the system, just ***constants***.
v_i, h_j are, the ***visible nodes*** and ***hidden nodes*** respectively.
$w_{i,j}$ are the weights between the visible and the hidden node.

☞ And then the ***probability*** of being in a ***certain state*** is given by, through the ***energy***, just like in the ***Boltzmann distribution***. It is following equation:

$$P(v,h) = \frac{1}{Z} e^{-E(v,h)}$$

- ⇨ Where ***Z***, is the ***sum*** of all of the ***values***, for all of the ***possible states***, just like in the ***Boltzmann distribution***.
- ⇨ The ***probability*** of being in a ***certain state*** is ***inversely*** related with the ***energy*** of that ***state***.

☞ And the ***system*** going to play by those ***rules***, it's going to find the ***lowest energy state***, just because of the way we ***set*** it ***up***.

The functioning of a BM is very different than what we had in just NNs.

❒ ***Additional reading:*** A great paper actually, by Yann LeCun, from 2006. It's called ***"A Tutorial on Energy-Based Learning"***. So if you really want to dig into **Energy-Based Learning** and ***Energy-Based Models*** then this is probably the best place to start.

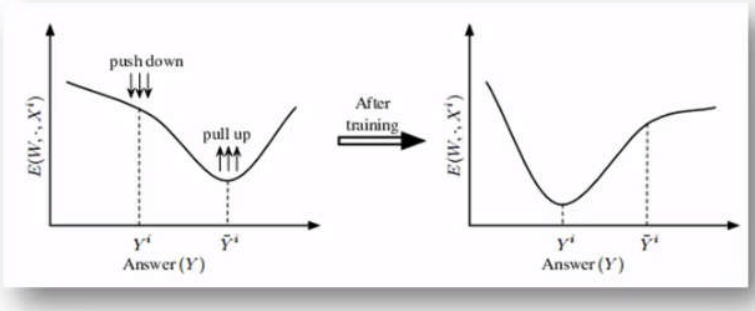
Additional Reading:

A Tutorial on Energy-Based Learning

By Yann LeCun et al. (2006)

Link:

http://yann.lecun.com/exdb/publis/pdf/lecun-06.pdf



❒ ***Watch:*** If you don't have time to read an article. Watch "***MR. Nobody***" by ***Jaco Van Dormael***. A film starring, by **Jared Leto**.

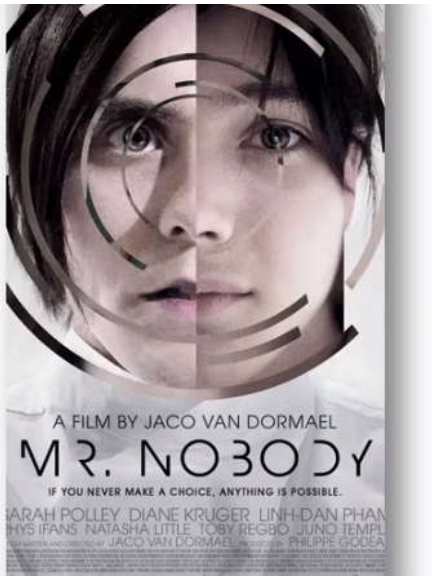
Additional Reading:

Mr. Nobody (film)

By Jaco Van Dormael (2009)

Link:

http://www.imdb.com/title/tt0485947/



- Why does ***time*** travel ***forwards*** and not ***backwards***?
- Why if you mix ***two liquids*** together, you can't then ***un-mix them***?
- Why ***smoke*** goes ***out*** of a ***cigarette*** not ***back into*** a ***cigarette***?
- Why you can ***remember*** the ***past*** and you ***can't remember*** the ***future***?
- These kind of very philosophical questions, we're getting very close to that when we're talking about ***Energy-Based Models*** and things like ***Entropy***.

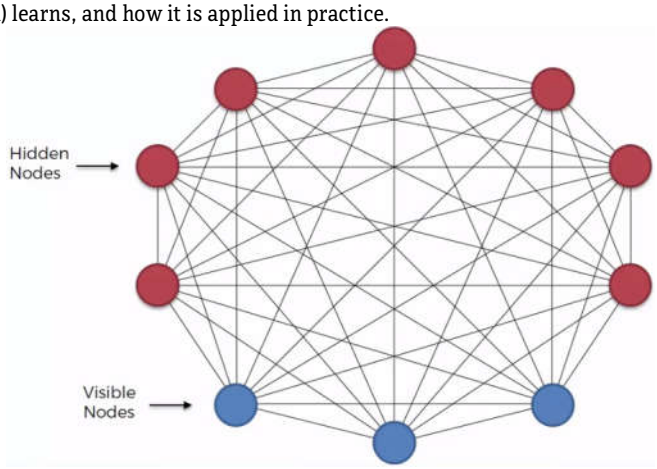
Deep Learning

BM: Restricted Boltzmann Machines (RBM)

`Introduction`

14.2.1 Restricted Boltzmann Machines (RBM)

And we're going to see how Restricted Boltzmann Machines (RBM) learns, and how it is applied in practice.



❒ ***BM:*** Here we've got the ***standard Boltzmann machine***, we've got all of these **intra connections**. Every ***single node*** connects to ***each other***.

☞ In ***theory*** this is a great model and you can ***solve*** lots of different ***problems***. But in ***practice*** it's very ***hard to implement***. We simply cannot compute a ***full Boltzmann machine***, the reason is as you ***increase number of nodes***, the number of ***connections*** between them ***grows exponentially***.

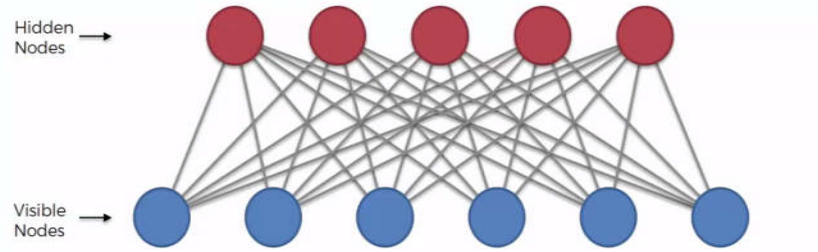
☞

❒ ***RBM:*** Therefore, a different type of architecture was proposed which is called the ***Restricted Boltzmann Machine (Rbm)***.

☞ Here in **RBM**, we've got exactly the same concept with the **simple restriction** that

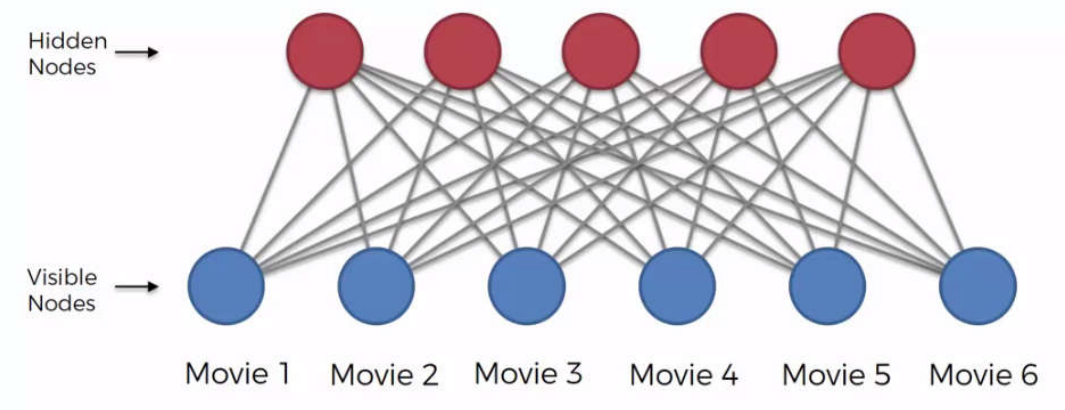
- ***Hidden nodes*** *cannot connect* to each other and
- ***Visible nodes*** *cannot connect* to each other.

☞ Everything else is the same to BM. We've also got undirected connections.



14.2.2 How a RBM trained

Now we're going to talk about how a ***RBM works***, how it's ***trained*** and then how it's ***applied*** in practice. For example, here we use ***RBM*** for **movie recommendation system**.

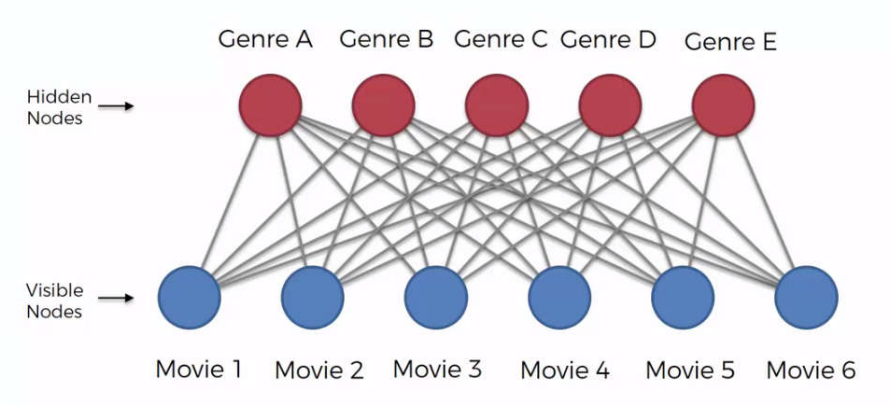


☞ Let's say our ***RBM - recommender system*** is going to be working on ***six movies***.

❒ As you remember, a ***Boltzmann machine*** is a ***generative type*** of ***model***, it always constantly ***generates states***. By training the ***BM*** through feeding it ***training data*** and through a process called ***contrastive divergence*** (it will be discussed next), the **Boltzmann machine** become a **representation** of **our specific system** *(rather being a recommender system for any kind of possible impossible movies)*.

☞ We make it the ***recommender system*** that is ***associated*** with our ***specific set of movies*** that we are ***feeding*** into this system and with our ***specific training data***.

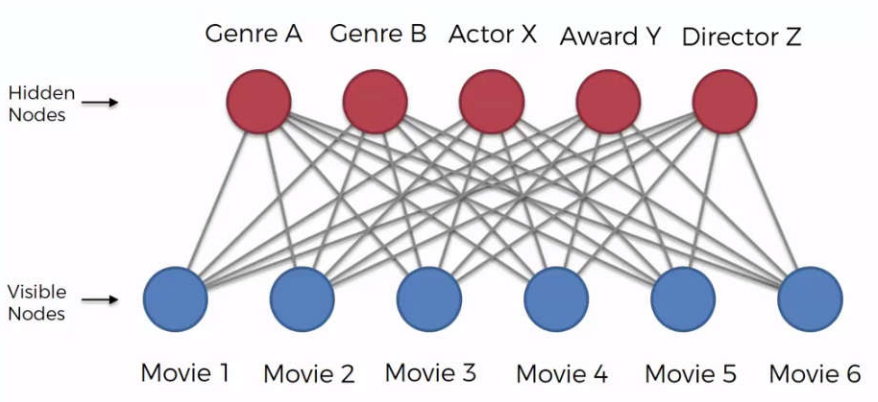
☞ Through that process, ***RBM*** is going to ***learn*** how to ***allocate*** its ***hidden nodes*** to certain ***features***.



☞ This process is very similar that we discussed in the CNN (for processing images). For example, through the training process, the RBM might identify some genres A, B, C, D and E.

⇨ But the important thing to understand here is that ***RBM*** **doesn't know** about those **genres**. It's just identifying ***certain features***. Actually it ***doesn't have to be genres***, for example, it could identify that ***genre A*** and ***B*** are important for the ***recommender system*** but there can be other important features such as an actor, an award or a director.

⇨ Hence those genres could be ***Genre of certain class*** A or B (eg: Action or Drama) or ***Genre of Actors*** (Tim Robbins or Morgan Freeman), ***Genre of Awards*** (Oscars or Golden globe), ***Genre of Directors*** (Tarantino or Cameron).



❒ ***How RBM identify an important feature:*** During the **training process**, we're feeding in lots and lots of **rows** to the **RBM** and for example, these rows could look something like ***Table in Right-hand***, where we've got ***movies*** as ***columns*** and then the ***users*** as ***rows***.

☞ We've got the ratings ***`1 = user liked it`***, and ***`0 = user didn't like it`***.

☞ The **empty cells** means that person **hasn't watched** that movie.

☝ Through this process as we're feeding this data to this RBM, it's able to understand better our system.

	Movie 1	Movie 2	Movie 3	Movie 4	Movie 5	Movie 6
User 1	1	0		1	1	1
User 2	0	1	0	0	1	0
User 3		1	1	0	0	
User 4	1	0	1	1	0	1
User 5	0		1	1		1
User 6	0	0	0	0	1	
User 7	1	0	1	1	0	1
User 8	0	1	1		0	1
User 9		0	1	1	1	1
User 10			0	0		0
User 11	0	1	1	1	0	1

☝ And ***adjust itself*** to be a better ***representation*** of our ***system***, and *understand* and *reflect* all of the ***intra connectivity*** that might be ***present*** in the ***data***.

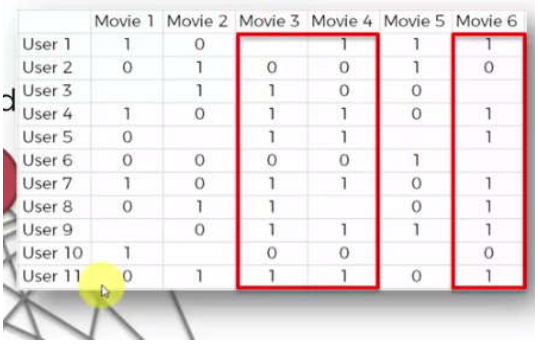
✋ Because ultimately, people have ***biases***, ***preferences***, ***tastes*** and that is what is ***reflected*** in the ***data***.

☝ If somebody liked ***Movie-2*** and ***Movie-3*** and didn't like ***Movie-1*** just means that that's what's their preferences.

☝ Somebody else might have liked ***Movie-1*** and might have not liked ***Movie-2*** but liked ***Movie-3***.

☝ So basically the data is talking about the **preferences** of people, their **tastes** and how they're **biased** towards **different movies** and that's what the **RBM** is trying to **find out**.

☞ ***RBM*** would ***identify*** those in the ***training*** and it would assign a ***node*** to look out for ***certain feature*** (even ***without knowing*** what that ***feature*** is, since all the input are **1**'s and **0**'s).



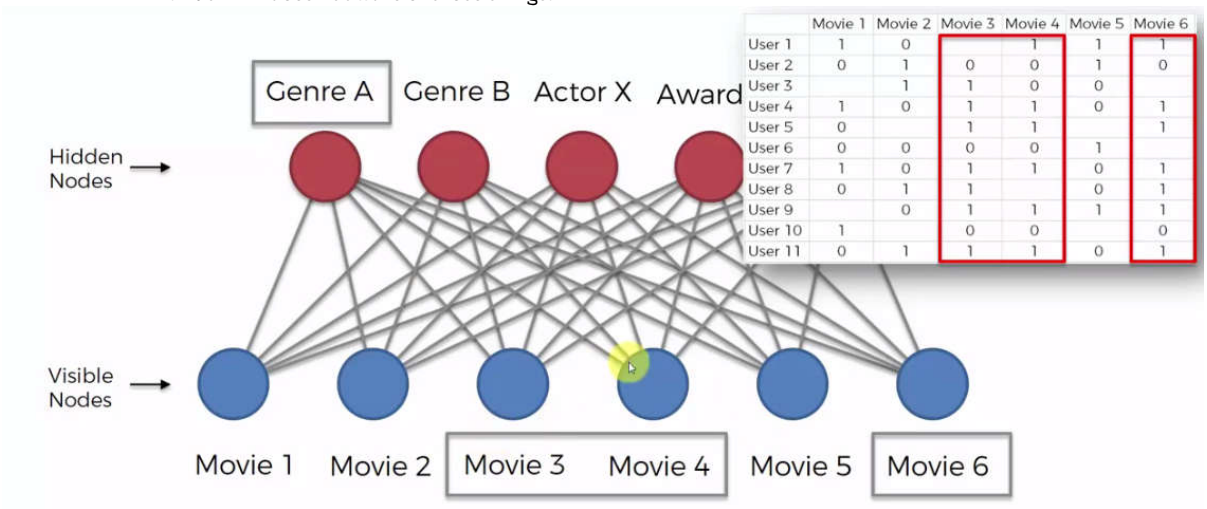
	Movie 1	Movie 2	Movie 3	Movie 4	Movie 5	Movie 6
User 1	1	0		1	1	1
User 2	0	1	0	0	1	0
User 3		1	1	0	0	
User 4	1	0	1	1	0	1
User 5	0		1	1		1
User 6	0	0	0	0	1	
User 7	1	0	1	1	0	1
User 8	0	1	1		0	1
User 9		0	1	1	1	1
User 10	1		0	0		0
User 11	0	1	1	1	0	1

⇨ It's ***not getting*** the ***genre*** of the movies or list of ***actors*** or list of ***awards***, it's only getting just these **1**'s and **0**'s.

⇨ From that kind of data it can ***establish*** that there probably is ***some feature*** that these ***movies*** have in ***common*** that is making people ***like*** them.

⇨ So people who ***like*** these ***movies***, actually they ***like that feature*** and therefore any ***other movie*** with that ***feature***, is highly likely to be ***enjoyed*** by those specific ***people***.

In our understanding, as humans that ***feature*** might be **Genre**, specific **Actors** or **Award** winning or some specific **Directors**. But RBM doesn't aware of those things.



	Movie 1	Movie 2	Movie 3	Movie 4	Movie 5	Movie 6
User 1	1	0		1	1	1
User 2	0	1	0	0	1	0
User 3		1	1	0	0	
User 4	1	0	1	1	0	1
User 5	0		1	1		1
User 6	0	0	0	0	1	
User 7	1	0	1	1	0	1
User 8	0	1	1		0	1
User 9		0	1	1	1	1
User 10	1		0	0		0
User 11	0	1	1	1	0	1

📣 In short ward RBM takes those ***input***, and through the ***training process*** it understands what ***features*** among these ***movies*** and it's assigning its ***hidden nodes*** or the ***weights*** are being ***assigned*** in such a way that the ***hidden nodes*** are becoming ***reflective*** of those specific ***features***. So that's how the training of the RBM happens.

14.2.3 Trained RBM in action

Consider we've ***trained*** up our ***RBM***, it is able to pick out these ***certain features*** and based on data of ***thousands*** of users and their ***ratings***.

❒ Now we're going to look at specific features. Let's say as ***features***, we are considering ***Drama*** or ***Action***, ***Leonardo DiCaprio*** as the actor, ***Oscar*** as an award (whether or not the movie has won an ***Oscar*** for the ***Best Picture***), and ***Quentin Tarantino*** as director.

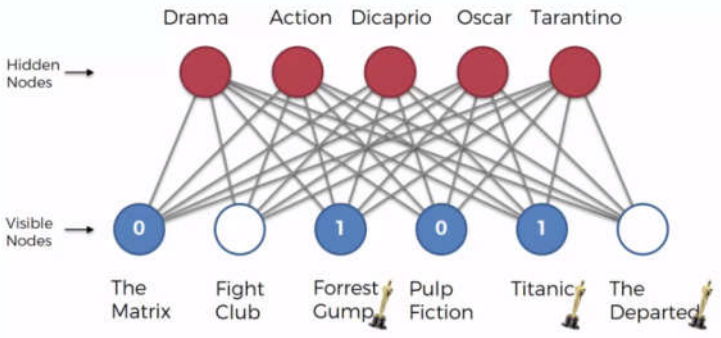
❒ These ***named-features*** are just for our ***learning purpose***. In reality, the ***RBM*** has no idea about those ***names***, ***Genres***, ***Actors*** or ***Directors***. It's just ***picking*** out a ***feature***.

Let's look at a couple of movies. We're going to input a new row into this **RBM-recommender system** and we're going to see how it ***predict*** whether a ***person*** will ***like*** certain movies or ***not***

☯ We've got movies ***The Matrix***, the ***Fight Club***, ***Forrest Gump***, ***Pulp Fiction***, ***Titanic*** and ***The Departed***.

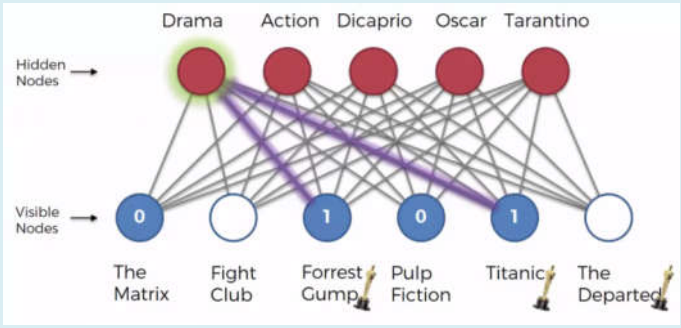
☯ Now the person that we're trying to make a recommendation for gives following rating :

- ***The Matrix:*** Seen & Didn't liked it (**0**),
- ***Fight Club:*** didn't Seen the Fight Club.
- ***Forrest Gump:*** Seen & liked it (**1**)
- ***Pulp Fiction:*** Seen & Didn't liked it (**0**)
- ***Titanic:*** Seen & liked it (**0**)
- ***The Departed:*** they haven't seen that movie

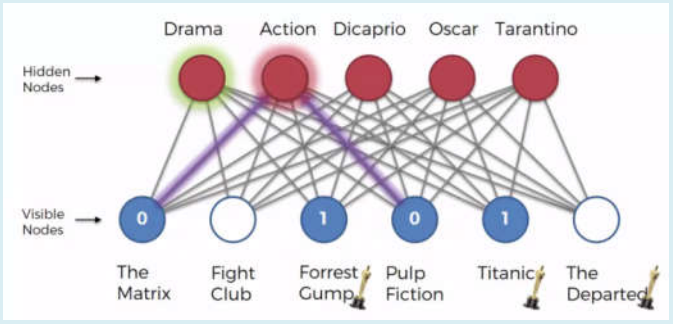


- Now since the user haven't seen ***Fight Club*** and ***The Departed*** our ***RBM*** will ***predict*** that the user will ***like*** those movies or ***not***, so that we can ***recommend*** movies to that ***user***.
- We're gonna go through this ***step by step*** to see how ***RBM*** takes those decisions. We're going to assess which of these nodes (features) are going to activate for this specific user.
  - ⇨ As in the ***CNN analogy***, there, we would feed in a ***picture*** into our ***CNN*** and it would, ***certain features*** would ***highlight***. Certain features would ***light*** up if they're ***present*** in that ***picture***.
  - ⇨ Same thing here we're ***feeding*** in a ***row*** into our ***RBM*** and certain ***features*** are going to ***light up*** if they are ***present*** in this user's ***tastes*** and ***preferences*** and ***likes*** and ***biases***.

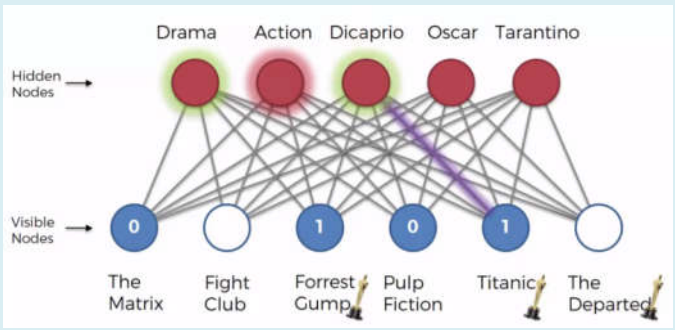
- ***Drama:*** ***Forrest Gump***, ***Titanic*** and ***The Departed*** are ***Drama***.
  - ⇨ (We don't have rating-data for The Departed, RBM can only learn from other two.)
  - ⇨ Since this person ***liked*** **Forest Gump** and **Titanic** and based on that this node is gonna **light up Green**.
  - ⇨ Symbolically **green** means node is **activated** and that means this person ***likes Drama movies***.



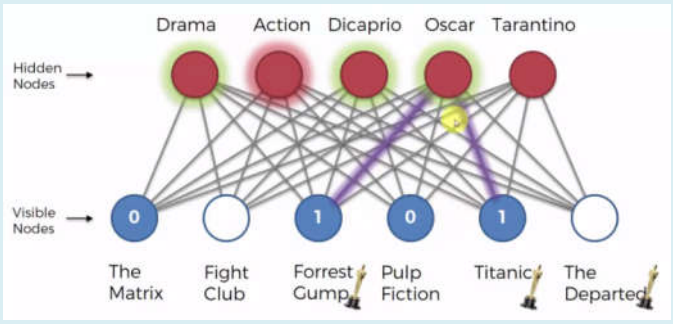
- ***Action:*** The Action movies we have here are ***The Matrix***, ***Fight Club*** and ***Pulp Fiction*** and ***Departed*** (it is also Drama).
  - ⇨ We have four Action movies but out of them we only have ***rating-data*** for ***The Matrix*** and ***Pulp Fiction*** and both of these, this person ***didn't like***. So it's gonna ***light*** up in ***Red***.



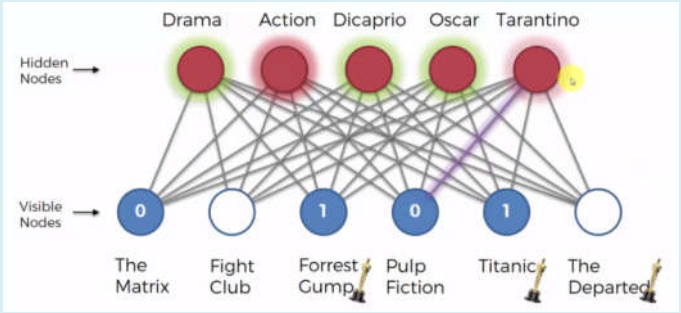
- ***Dicaprio:*** ***Leonardo DiCaprio*** is present in ***Titanic*** and ***The Departed***.
  - ⇨ We only have ***rating-data*** for ***Titanic*** and user **liked** it. So the ***DiCaprio node*** is going to **light** up **green**.



- ***Oscar:*** Here we've got three Oscar movies. We only have rating-data for ***Forrest Gump*** and ***Titanic***. The person **liked** both. The **Oscar-node** is gonna just **light** up **green**.



- ***Tarantino:*** The only **Tarantino** movie we have here is **Pulp Fiction**. The person ***did not like*** it. Therefore this Tarantino-node is gonna ***light*** up ***Red***.



☝ Now that's the ***first pass (forward-pass)***. Everything from our *visible nodes* goes into our *hidden nodes* and now we know which ones of our *hidden nodes* are *activated*.

✋ ***Backward Pass:*** When the ***backward pass*** happens, the ***Boltzmann machine*** try to *reconstruct* our *input*. It happens during training as well. So during ***training*** the ***test*** is also happening.

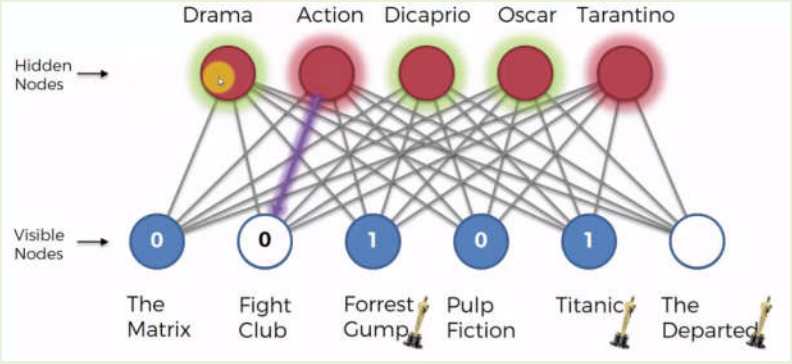
☝ *BM* first accept values into the *hidden nodes* and then it tries to *reconstruct* your *inputs* based on those *hidden nodes*. If during *training* the *reconstruction* is *incorrect*, then everything is ***adjusted*** (*weights* are *adjusted*) and then ***reconstruction*** happens ***again***.

☯ Since we working with **trained *RBM***, we're actually *inputting* a certain ***row*** and we want to get our ***predictions***. So basically, there is not gonna be any *adjusting* of *weights*. We're just going to see how the ***BM*** basically *reconstructs* these *rows*.

- ⇨ We're not going to care about the *movies* that we *already* have *ratings*, that's the training part of BM.
- ⇨ Here we're only going to care about the movies that **don't have *ratings***, we're gonna use ***RBM*** to ***reconstructs*** the **ratings** as **predictions**.

☯ ***Fight Club:*** Fight Club is going to look at all of the *hidden-nodes* and based on training it's going to find out which *nodes* actually connect to *Fight Club*.

- ⇨ It's **not a Drama** movie.
- ⇨ It's an **Action** movie.
- ⇨ It *doesn't have DiCaprio* in it.
- ⇨ This movie *hasn't win* an *Oscar*.
- ⇨ *Tarantino* is *not* the *director* of this movie.



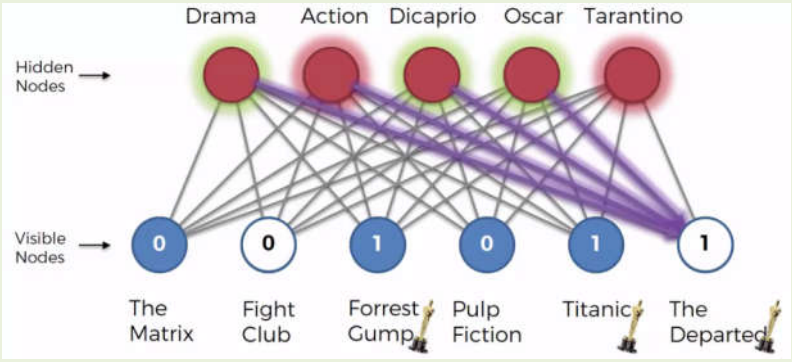
➲ Hence from all 5 hidden-nodes it only connects to **Action (node)**. But this node is lit Red (not active).

➲ *RBM* recognize these *associated connections*, based on the *weights* that it had determined *during training*.

➲ Based on **Action's (node)** connection, we know this one ***lit*** up in ***Red*** and therefore *Fight Club* is going to be a movie that this *person* is *not going* to *like*. The predicted rating will be **0**.

☯ ***The Departed:***

- ⇨ It's **a Drama** movie. Connected to this node (active).
- ⇨ It's an **Action** movie. Connected to this node (not active).
- ⇨ It *does have DiCaprio* in it. Connected to this node (active).
- ⇨ This movie *has win* an *Oscar*. Connected to this node (active).
- ⇨ *Tarantino* is *not* the *director* of this movie. It's ***not connected*** to this node. The *weight* here is *low* or very *insignificant*.



➲ Among *5-hidden-nodes* 4 are connected and 3 of them are active, hence *The Departed* is going to be a movie that this *person* is *going* to *like*. The predicted rating will be **1**.

So there we go, that's how the ***RBM*** works. ***RBM*** is detecting some kind of **sequence** from the given data and filling the gaps (predicting unknown) according to presented data.

Chapter 14 : Part 3

Deep Learning

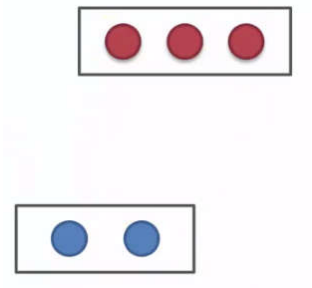
BM: Contrastive Divergence & DBN, DBM

Contrastive Divergence & Advanced topics

14.3.1 Contrastive Divergence: Diagrammatically

In this part we'll discuss about ***Contrastive Divergence***. This is the algorithm that actually allows RBM to learn.

❒ Here we've got a diagrammatic representation of our **RBM**. We've got **2 input Nodes** (***Blue***), and we've got **3 hidden nodes** in ***Red***. Here we focus on a specific part of the learning process. In previous section of this chapter

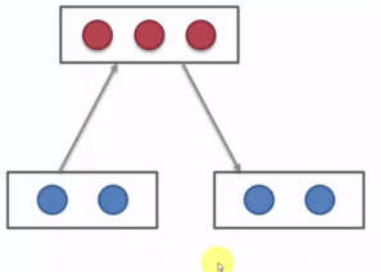


❒ We discussed exactly how we feed in different ***values*** in ***RBM***, and how it looks at them and looks for ***features*** and then ***assign*** certain ***nodes*** to those ***features***.

- ☞ But we didn't discuss how **RBM adjust** those **weights** to connect the nodes.
- ☞ Previously in the other NN, we had the ***gradient descent*** process, which allowed us to ***back propagate*** the ***error*** through the ***NN***, and ***adjust*** the ***weights*** to ***minimize*** that ***error***.
  - ⇨ Previously those ***NNs*** are ***directed*** network. But in ***RMB***, we don't have an ***undirected*** network.
  - ⇨ This is where the Contrastive Divergence comes in. It is used to adjust weights in ***undirected*** network.

✋ We're going to look at it in two ways.

- ➢ ***Diagrammatically*** like this and
- ➢ Later through an ***Energy graph***.

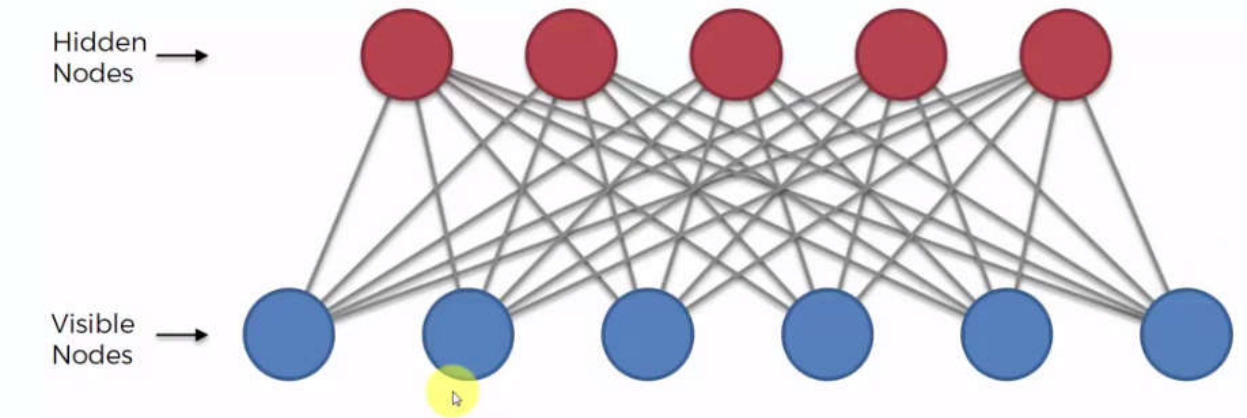


☞ Here we've got 2 input nodes. At the very start the RBM calculates the ***hidden nodes*** using some ***randomly assigned weights***. Then those **hidden nodes** are going to use the **exact same weights** to ***reconstruct*** the **input nodes**.

☞ It important to understand that the ***reconstructed inputs*** are not going to equal to the ***original inputs***, even though the ***weights*** are the ***same***.

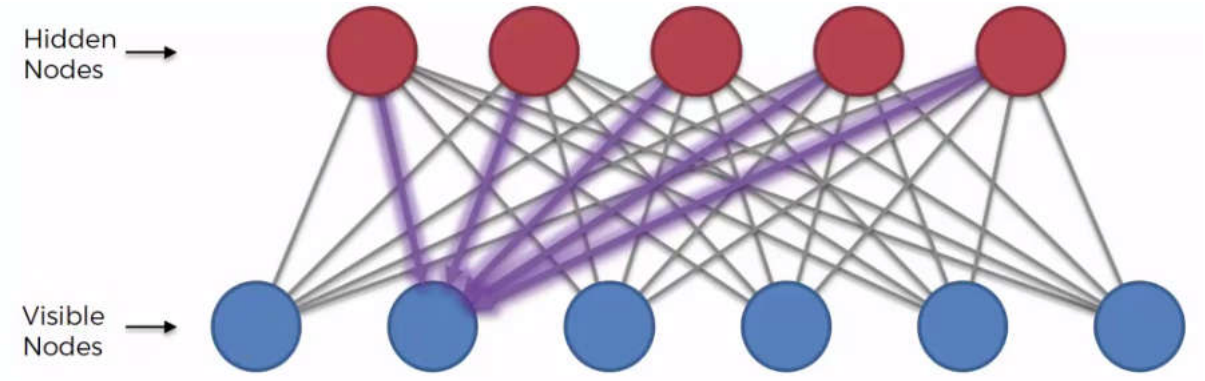
✋ Once we've ***reconstructed the visible-nodes***, they're not **identical** to the **original visible nodes**. Even though we're using the same ***weights***.

☝ The reason for that is because these nodes are not ***initially interconnected***. There's no specific connection, and there's no formula or equation that's connecting them at **very beginning**.



❒ Lets say **second input-node** (visible-node) get ***reconstructed***. It gets reconstructed using ***all*** the ***hidden nodes*** (all five hidden nodes). But those hidden nodes first created from our **original input-nodes** (all six ***input-nodes/visible-node*** contributed to create each ***hidden node***).

☞ At the very-beginning in the RBM, these **initial input-values** will ***initiate*** some ***values*** in your ***hidden nodes***. Then once we run it **backwards**, these **hidden nodes** will **reconstruct**, all of **input-nodes** including this ***second*** **input-node**.

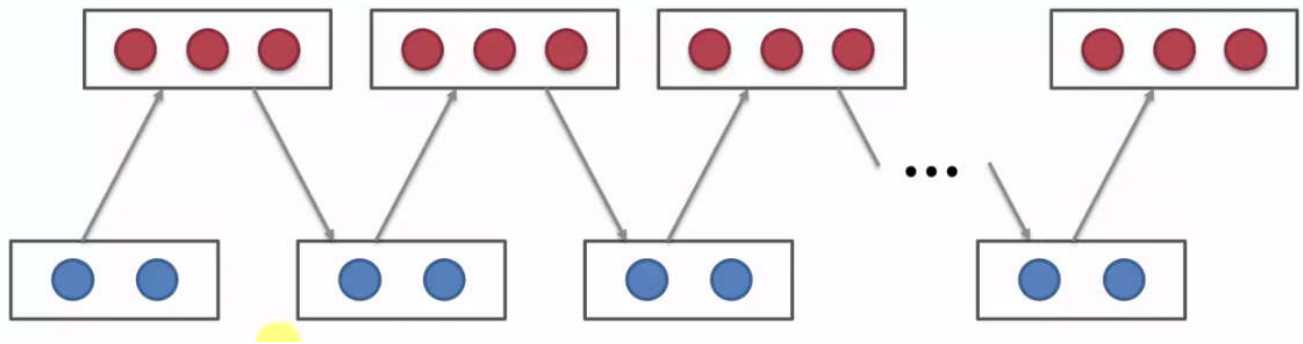


- If ***only one input-node*** is used to create ***all hidden-node*** then the ***reconstructed input-node*** would be same to **input-node**.

- Since all ***initial-input-nods*** contributed to create ***each*** of ***hidden nodes***, and all **hidden nodes** are trying to **reconstruct** each of **input-nodes**. That's why ***reconstructed input-nodes*** won't be same as ***initial-input-nods*** *(because indirectly all initial-input-nods are using to reconstruct an input-node).*

That's very important to understand, that's why this whole Contrastive Divergence process exists.

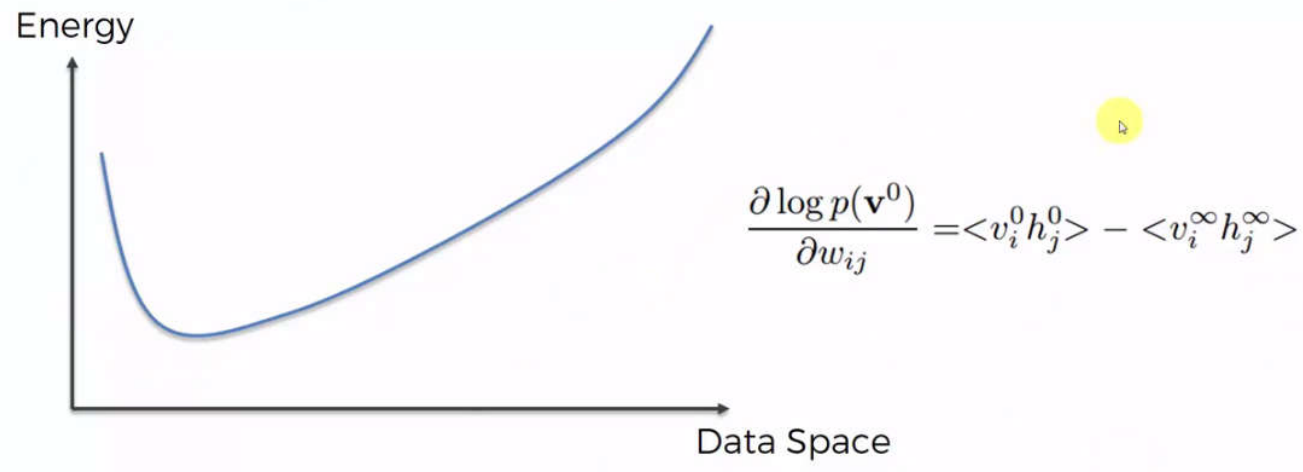
- The RBM repeats this process: feed ***reconstructed node values*** of our inputs into the ***RBM*** and we're going to get some ***values*** for ***hidden nodes***. Then based on these ***hidden values***, we're going to ***reconstruct*** the ***inputs*** again, and again. This whole process is called ***GIBBS SAMPLING***.
- At the ***end***, we're going to get some ***reconstructed input*** values such that when we feed them into the ***RBM***, and then we try to ***reconstruct*** them again, we will get those ***same values***. That is the ***reconstructed input*** values ***converges*** to some values.



- In this **final scenario**, our **network** is modeling **exactly** our **inputs**. So basically, we can **input** in and we will always get the same **output**. This process has finally **converged** and our network is finally **trained**.

14.3.2 Contrastive Divergence: Energy graph

In terms of the curve, Contrastive Divergence looks like follows. We've got two parts here, we'll start with the formula.

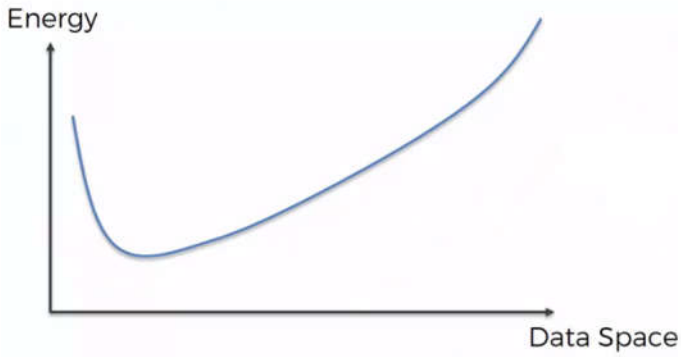


❒ ***Gradient Formula:*** This is a gradient formula, can see the ***gradient*** on the ***left*** here. We've got the ***gradient*** of the ***log probability*** of a ***certain state*** of our ***system***, based on the ***weights*** in the ***system***.

$$\frac{\partial \log p(\mathbf{v}^0)}{\partial w_{ij}} = <v_i^0 h_j^0> - <v_i^\infty h_j^\infty>$$

❒ ***Weights are Constant:*** Remember, through this whole process, the **weights** are **constant**. We're not changing the ***weights***, we're just using ***random weights***.

☞ When we talked about energy based systems we said in ***RBM*** **energy** is defined through the ***weights***. Following is the curve for the weights, "**Energy**" is actually weights here.



☞ Since we are using **random weights**, ***reconstructed input-values*** won't be same to our ***real-input-values*** even though the system is converged. We need to adjust the weights in such a way that the system reach to the low energy state.

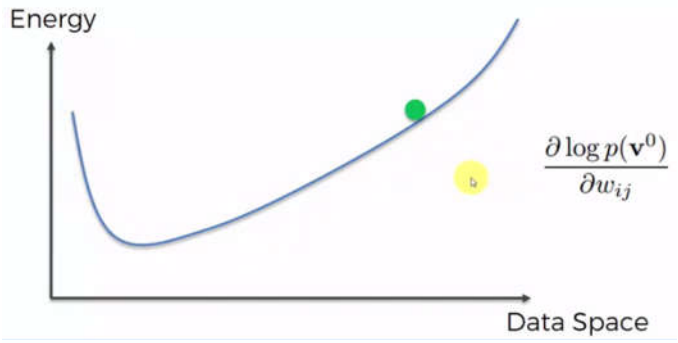
☞ It's telling us is how the ***weights*** affect the ***log probability***.

☞ $< v_i^0 h_j^0 >$ is the ***initial state*** of the system, v_i^0 is ***visible*** *vector*, h_j^0 is ***hidden*** *vector*. $< v_i^\infty h_j^\infty >$ represents ***next states***.

❒ The ***weights*** are ***fixed*** and we're going to define this ***energy curve*** is based on the ***weight***.

☞ ***Weights*** dictate the ***shape*** of this ***energy curve***,

☞ For our ***first pass*** through the ***RBM*** we place our ***initial inputs*** (the ***green point***, since the ***weights*** are initialized ***randomly*** it could be anywhere).

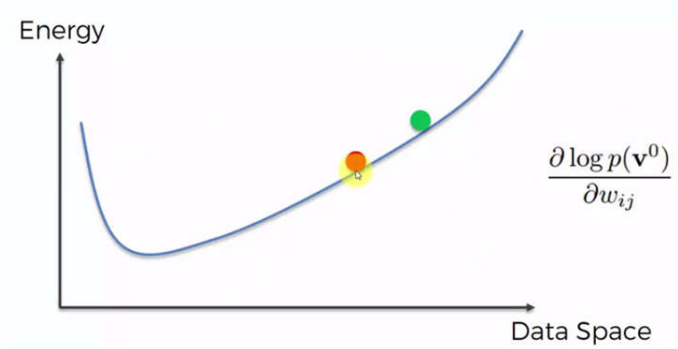


☞ After the ***second pass***, we end up at the ***red point***. Since a system which is governed by its energy will always try to ***end*** up in the ***lowest energy state*** possible.

☞ So as you can see, this ***ball/point*** is rolling towards the ***bottom*** (minima).

☞ That's exactly what's happening through that ***Contrastive Divergence process***, where ***reconstructed input*** values ***converges*** to some values.

⇨ We're going ***closer*** and ***closer*** to our ***lowest energy state***.



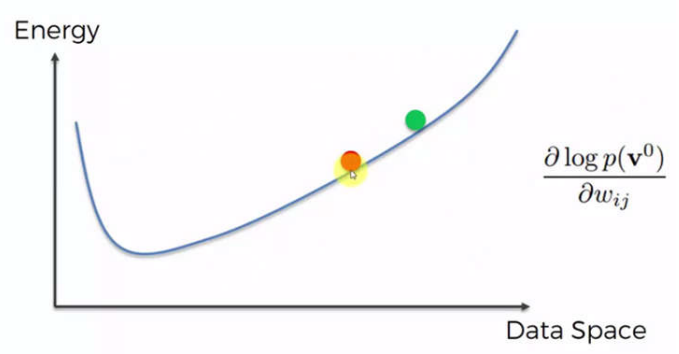
✋ But remember, the ***weights*** are ***not changed***, but we get certain ***reconstructed input*** values and certain **hidden values**, that bring the system to the **minimal energy state** at the end of this ***Contrastive Divergence process***.

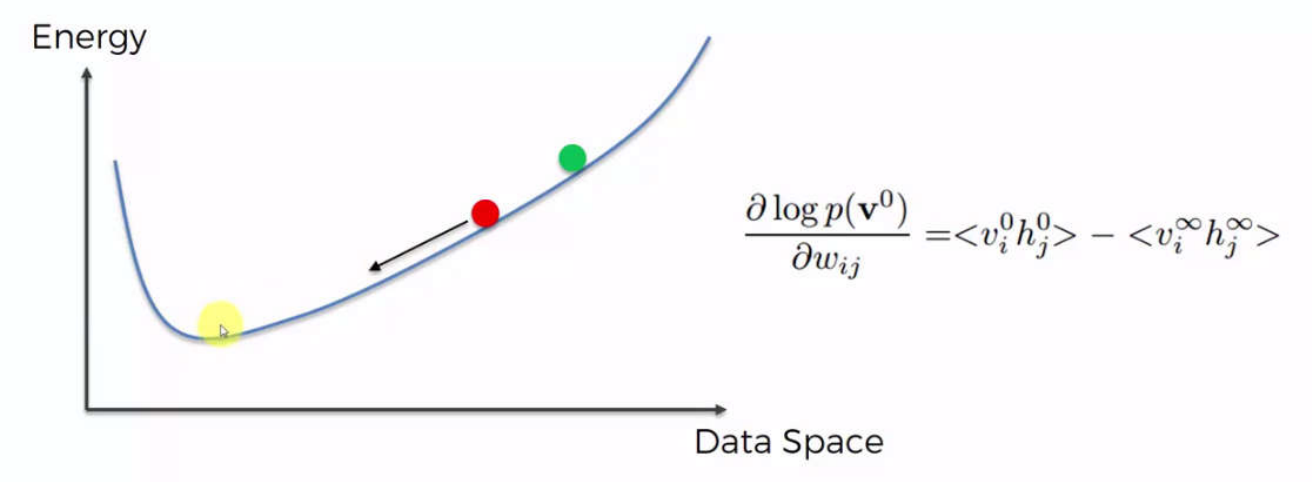
☝ And what this formula is telling us is, once you have that ***lowest energy state*** if you subtract $< v_i^\infty h_j^\infty >$ value from $< v_i^0 h_j^0 >$ value, it will tell you how ***adjusting*** your ***weights*** will affect the ***log probability*** of the system being in this ***lowest energy state***.

☝ So basically, this formula is a recipe for ***adjusting*** your ***curve*** or for modifying your ***energy curve***, so that you can make sure that ***initial state*** (where we have ***initial-input-values***) is inside an ***lowest energy state***, so that you can get an desired effect.
[From this formula we adjust the weights in such a way that the ***reconstructed input*** values are same as **initial-input-values**.]

❒ Right now (before adjusting weights) it's like getting towards a certain **minimal energy state** but the ***inputs*** **are completely different** to our ***real-inputs***, we want to change that.

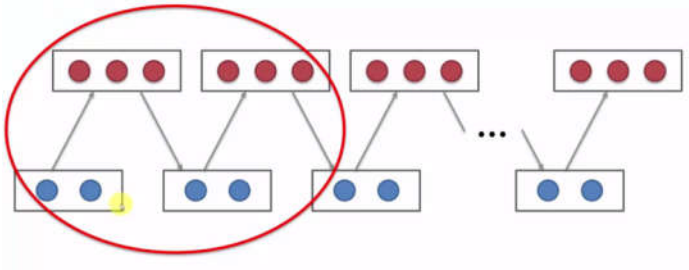
☞ We want to use this **gradient formula** to ***adjust*** our ***curve***, so this, the ***energy minimum*** is actually ***next*** to our ***inputs*** rather than some ***random reconstructed inputs*** which are defined by the ***randomly initialized weights***.





❒ ***Hinton's- shortcut:*** In 1998, **Jeffrey Hinton** discovered a shortcut that: *"Even if you take just the **first two passes**, you **don't wait** until it **convergences** to the **end**. This is **sufficient** to understand how to **adjust** your **curve** as far as is the **initial stage**."*

☞ **CD-1** means **Contrastive Divergence 1 pass**. You might hear that term CD-1, CD-3, CD-5, CD-9, So if you do a CD-1, Contrastive Divergence one pass, its enough for you to know which way the ball/point is rolling.

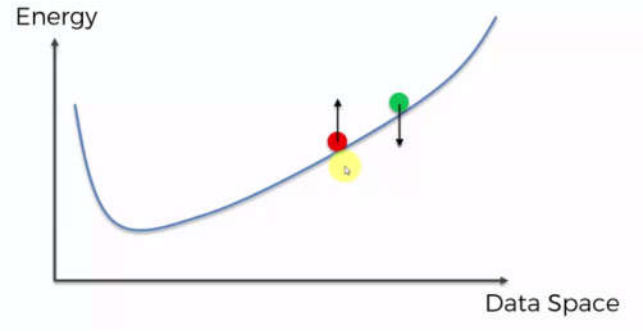


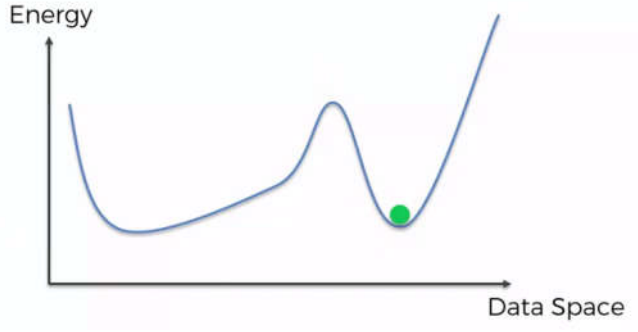
☞ It is kind of similar to what we had in ***gradient descent's*** downhill/uphill. In ***gradient decent*** we just had a curve and we were like ***finding*** the ***minimum***.

☝ But here we have ***control*** over the ***curve***, now want ***to adjust/change*** this ***curve*** so that the ***minima*** will be at the ***point*** where ***we want***. We are ***adjusting*** the ***weights*** because it's an ***energy based process***.

🛠 We're adjusting the weights so that minimum is actually going to be at our ***initial state*** rather than some ***random state***.

☝ Since in our case the ***balls*** are going ***downhill***, we ***pull*** this curve ***down*** where the ***green-ball*** is, make it ***minimum-energy-state*** and we want to ***push*** it up over the point where the ***red ball*** is.





☞ So you can see ***green-ball (initial state)*** is already inside the **minimum**, and we don't even have to go through *the long process of sampling* to get to that ***recipe*** of how to ***adjust*** the ***curve***, but you can just ***adjust*** the ***weights*** using ***Hinton's- shortcut***.

❒ ***So in a summary:***

- ⇨ We have an ***energy curve*** and the ***shape*** of this ***energy curve***, is governed by the ***weights*** in the system,
- ⇨ We also know the ***RBM*** will always get to the ***minimal energy state***. Minimal energy state found through the ***Gibbs sampling process***.
- ☝ ***Gibbs sampling process:*** *where* ***RBM*** *feeds* ***input****, set the* ***hidden values*** *and then it* ***reconstruct*** *the* ***inputs*** *then it* ***feeds*** *again from* ***reconstructed inputs*** *and the process goes on until* ***reconstructed inputs*** *converges to* ***some values*** (at **lowest energy state**).
- ⇨ We then ***redesign*** the ***system*** by ***redesigning*** the ***energy curve***. We ***adjust*** the ***weights*** in such a way that when we ***input*** our values, our ***training*** values, the system is ***already*** going to be in the ***lowest energy possible***.
- ⇨ And for that we use this ***Gradient Formula***.

$$\frac{\partial \log p(\mathbf{v}^0)}{\partial w_{ij}} = <v_i^0 h_j^0> - <v_i^\infty h_j^\infty>$$

i. We start with some randomly initialize weights,

ii. We input a value like one of our ***rows*** into the ***RBM***,

iii. We go through this process of Gibbs sampling, we calculate $< v_i^0 h_j^0 > - < v_i^\infty h_j^\infty >$, we find out how to ***adjust*** our ***curve***.

iv. Plus on top of all of that, there's ***Hinton's- shortcut***. We don't actually have to go through to the very end $< v_i^\infty h_j^\infty >$ of the sampling process,

- ⇨ We can just do ***two passes***, we go ***first pass***, ***second pass***, we do a **CD-1, Contrastive Divergence one**, and that will tell us how to ***adjust*** the ***curve***.

v. We're trying to ***adjust*** the ***energy curve*** by **modifying** the **weights**, in order to create a system which in the best way possible resembles our ***input values***, our ***training values*** and we do that, using above ***gradient formula***.

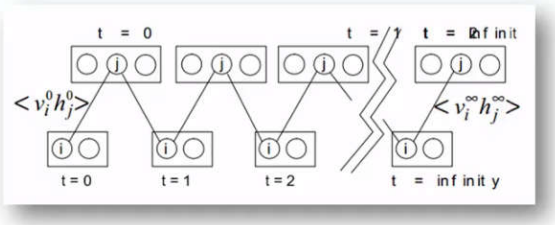
14.3.3 Additional Reading

❒ If you'd like to get into more depth on this topic, on ***Contrastive Divergence***. The first paper is by ***Jeffrey Hinton*** and others 2006, it's called a ***Fast Learning Algorithm For Deep Belief Nets***. And you can see exactly the diagram here which we discussed.

Additional Reading:

A fast learning algorithm for deep belief nets

By Geoffrey Hinton et al. (2006)



Link:

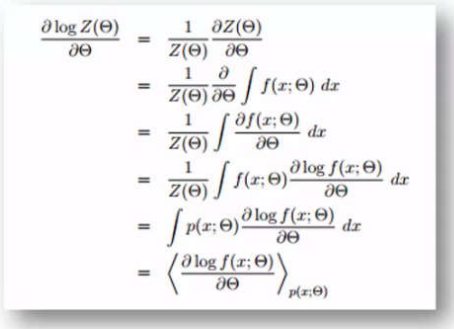
https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf

❒ Another paper if you'd like to get a bit more mathematical on the Contrastive Divergence and really understand the math behind it. And what's exactly going on with the gradients and so on, a good paper to look at is called ***Notes on Contrastive Divergence***, it's not actually a paper it's just some **notes** is a **three pager** by ***Oliver Woodford***.

Additional Reading:

Notes on Contrastive Divergence

By Oliver Woodford (2012?)

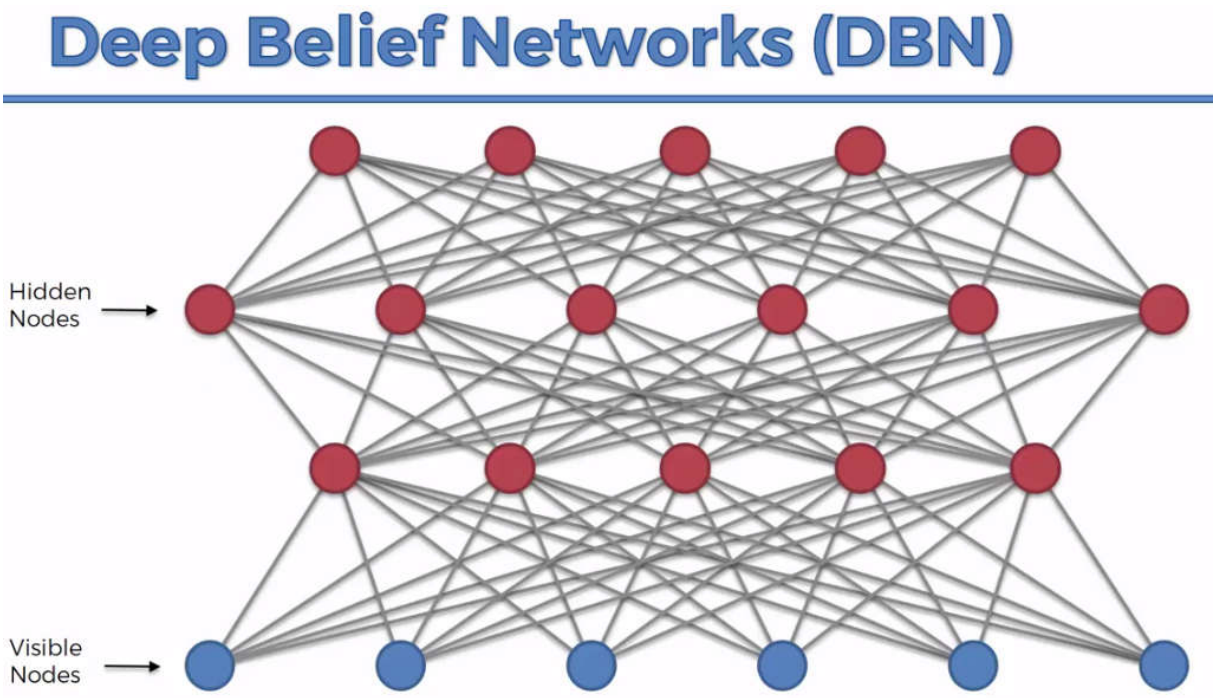


Link:

http://www.robots.ox.ac.uk/~ojw/files/NotesOnCD.pdf

14.3.4 Deep Belief Network (DBN)

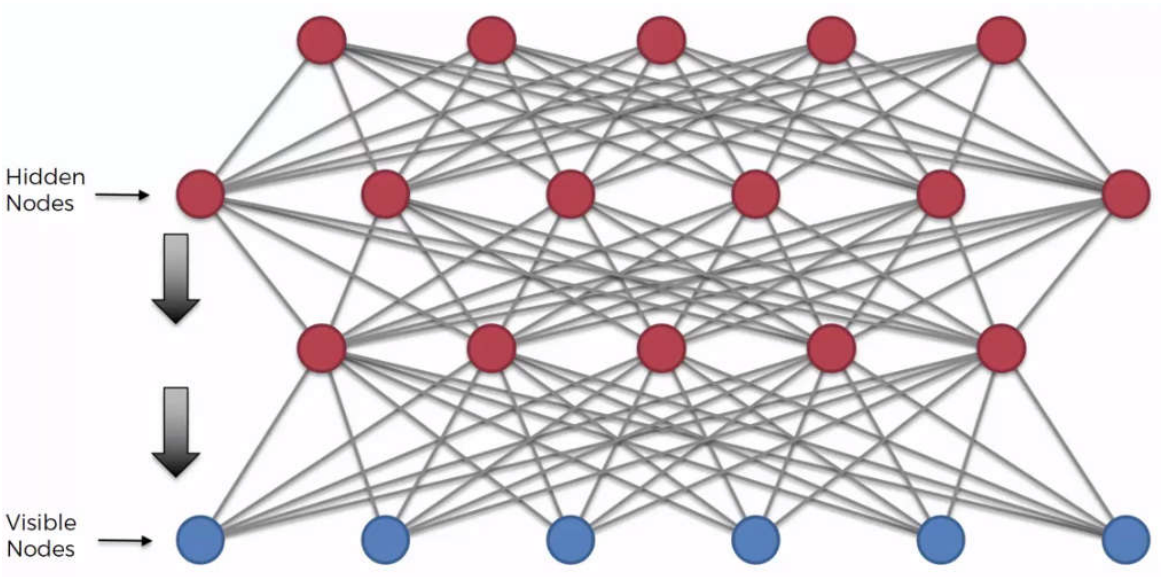
❒ ***DBN:*** A Deep Belief Network (DBN) is a stack (on top of each other) of several RBMs. Where the ***1st RBM's*** **hidden nodes** are the **input nodes** of ***2nd RBM*** and ***2nd RBM's*** **hidden nodes** are the **input nodes** of ***3rd RBM*** and so on.



❒ ***Directed & Undirected connections in DBN:*** In a Deep Belief Network, you make sure that your directionality is in place for all of the layers except for the top two.

☞ Basically you make sure that these layers ***one***, ***two*** and ***three***, and there are ***directed connections*** between them, and they are ***directed downwards***. Whereas there is no direction in the top two layers *(undirected connections)*.

☝ It's quite hard to explain what's going on here because this is a very ***complex***, a very ***advanced*** type of ***network***. When ***Hinton*** and his ***team*** find that's ***DBN idea*** when the whole ***interest*** in ***Deep Learning*** got revived in ***2000's***.



❒ In terms of training, there's two types of algorithms in DBN

i. ***Greedy layer-wise training algorithm:*** First you train the RBMs, with the undirected connections. You train them layer by layer as RBMs, 1st layer, 2nd layer and then 3rd layer (down to top). The directionality is set it up after you've trained up the weights.

ii. ***Wake-sleep algorithm:*** It is basically you train all the way up, then you train all the way down your layers.

❒ ***In simple word DBN is:*** You stack up RBMs, you train them up, then you, once you've got the weights, you make sure that these connections only work downwards (except top two layers).

☞ If you ever do need to use it in practice, if you have to design your own ***DBN***, then you have to do some ***research*** and read some ***papers*** and ***understand*** the ***design*** that goes onto it.

14.3.5 Additional Readings

❒ The ***Greedy layer-wise training of Deep Belief Networks*** is an article by ***Yoshua Bengio*** and others.

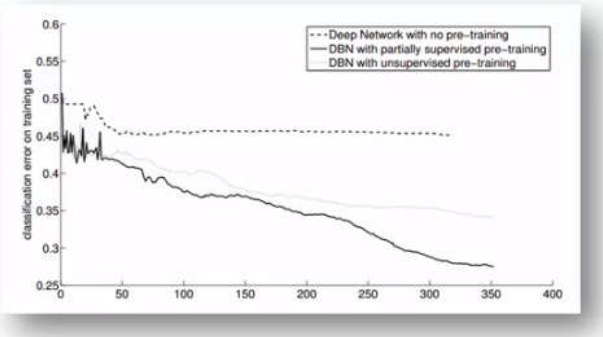
Additional Reading:

Greedy Layer-Wise Training of Deep Networks

By Yoshua Bengio et al. (2006)

Link:

http://www.iro.umontreal.ca/~lisa/pointeurs/BengioNips2006All.pdf



❒ But the article that we mentioned in the previous section (14.3.3) by ***Jeffrey Hinton*** and others 2006, it's called a ***Fast Learning Algorithm For Deep Belief Nets***. Where we were talking about how the ***training*** of an ***RBM*** happens. Well that article actually also has, that's where the ***Greedy layer-wise*** is described before it was used in ***Bengio's*** article.

☞ So, make sure to check out that article first, and then move onto the above paper by Bengio. There they explore the whole concept of ***Greedy layer-wise training*** further.

☞ If you do want to get into DBNs more then those two articles, those two papers, will help you get, actually do exactly that. You will also learn about Greedy layer-wise training and how it's happening.

❒ Another one is the ***wake-sleep algorithm*** by Hinton. This one is about the ***wake-sleep algorithm***.

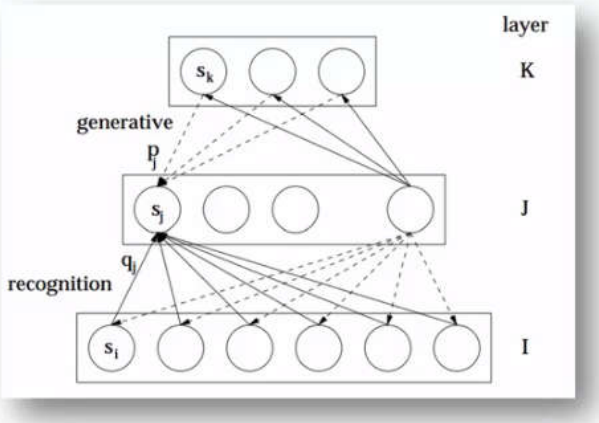
Additional Reading:

The wake-sleep algorithm for unsupervised neural networks

By Geoffrey Hinton et al. (1995)

Link:

http://www.gatsby.ucl.ac.uk/~dayan/papers/hdfn95.pdf

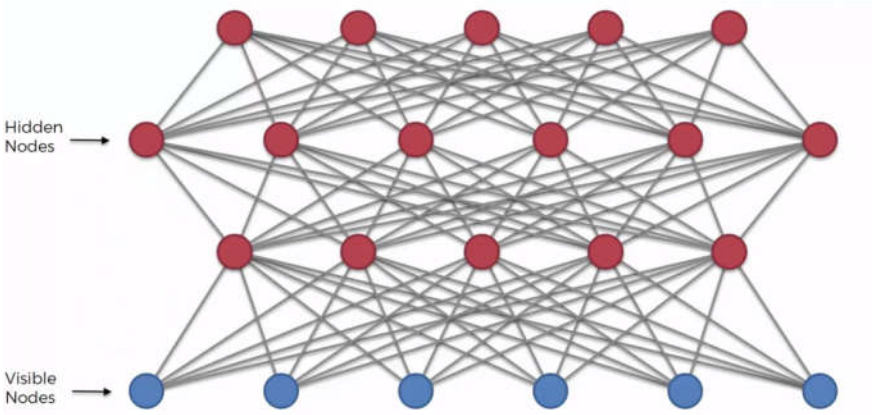


☞ If you feel that you need to get into DBNs, and it's something that you need for your work or something you need for a project or you want to explore further, these are the 3 papers that are a good start to get you into the world of Deep Belief Networks.

14.3.6 Deep Boltzmann Machines (DBM)

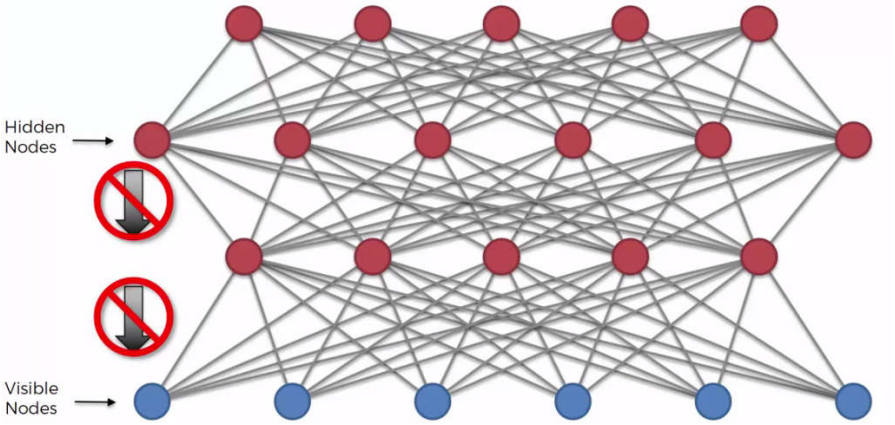
DBN <> DBM

❒ ***Deep Boltzmann Machines (DBM)*** is another topic, just like the ***DBNs***. ***Deep Belief Networks (DBN)*** are not the same as ***Deep Boltzmann Machines (DBM)***. Deep Boltzmann Machines actually can extract features that are more sophisticated, more complex, and therefore they could potentially be used for more complex tasks.



❒ ***The main difference is, in DBM is totally undirected network:*** Previously in **DBN** we had **stacked RBMs**, after the training has happened, you make sure that ***all of your layers***, ***except*** for the ***top two***, are ***directed layers*** (connections between them are directed downwards).

☞ In ***Deep Boltzmann Machine (DBM)***, you don't have that, ***no directed layers***. But in ***DBM*** we also have ***stacked RBMs***.



❒ ***Additional readings:*** There's a great paper, which we're going to direct you towards in terms of Deep Boltzmann Machines. It's called ***Deep Boltzmann Machines*** by ***Rusian Salakhutdinov***. Hinton is also a co-author here.

☞ ***Ruslan*** actually has ***quite few papers*** on ***Boltzmann Machines*** and ***RBMs*** and stuff like that.

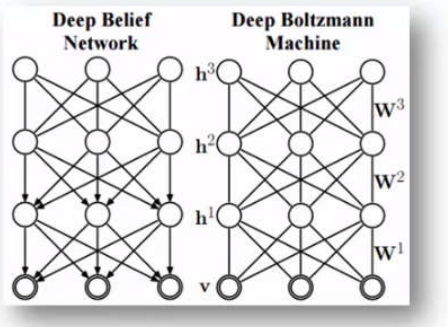
Additional Reading:

Deep Boltzmann Machines

By Ruslan Salakhutdinov et al. (2009?)

Link:

http://www.utstat.toronto.edu/~rsalakhu/papers/dbm.pdf



Chapter 14: Part 4

Deep Learning

BM project - part 1: Building RBM class

`Data preprocessing & Building RBM class`

14.4.0.1 Objectives

Boltzmann Machines can be seen from ***two*** different ***points of view***.

[1]. An ***Energy-Based Model***

[2]. A ***Probabilistic Graphical Model***

☞ In the ***Intuition Lectures*** we focused on the ***Energy-Based Model*** point of view, and then for the ***Practical Lectures*** we will focus more on the ***Probabilistic Graphical Model*** point of view.

❒ In these last two parts (***Boltzmann Machines*** and ***AutoEncoders***) of this book, we will create two types of Recommender Systems:

[1]. One that predicts **binary ratings** "***Like***" or "***Not Like***". We will build it in this section with a ***Boltzmann Machine***.

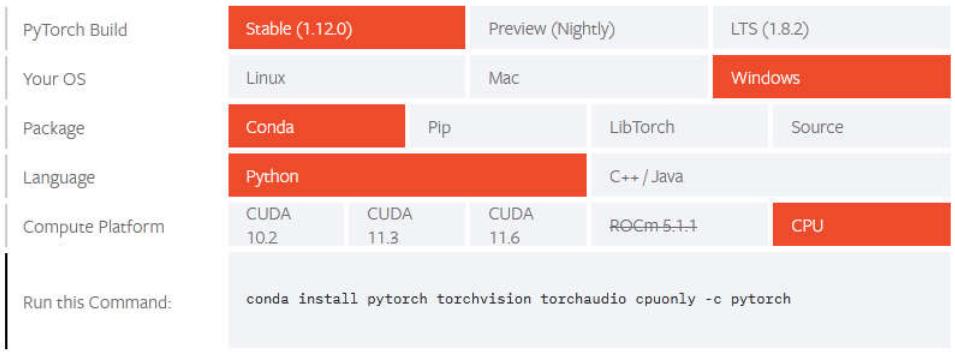
[2]. Another one that **predicts ratings** from ***1*** to ***5***. We will build it in next chapter with an ***AutoEncoder***.

☞ We will implement these two Deep Learning models with ***PyTorch***, a highly advanced Deep Learning platform more powerful than ***Keras***.

☞ Every single line of code will be explained in details but I would recommend to have a first look at the ***PyTorch documentation*** to start getting familiar with ***PyTorch***.

14.4.0.2 Installing Pytorch

Using Anaconda (conda)




```
conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

Using pip:




```
pip3 install torch torchvision torchaudio
```

☝ However, following also install ***PyTorch*** with ***anaconda***. If you do it, no need to run ***conda installer***. If you want to install in ***Local Python directory***, rename the folder "**anaconda3**" in: C:\Users\user_name

☞ Rename "**anaconda3**" as "**anaconda3n**" or "**anaconda3bak**" or whatever you want. It is temporary, we'll undo it after Pytorch installed in Local Python.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\SolLaSi>pip3 install torch torchvision torchaudio
Collecting torch
  Downloading torch-1.12.0-cp38-cp38-win_amd64.whl (161.9 MB)
     |████████████████████████████████| 161.9 MB 913 bytes/s
Collecting torchvision
  Downloading torchvision-0.13.0-cp38-cp38-win_amd64.whl (1.1 MB)
     |████████████████████████████████| 1.1 MB 731 kB/s
Collecting torchaudio
  Downloading torchaudio-0.12.0-cp38-cp38-win_amd64.whl (969 kB)
     |████████████████████████████████| 969 kB 369 kB/s
Requirement already satisfied: typing-extensions in c:\users\sollasi\anaconda3\lib\site-packages (from torch)
(3.7.4.3)
Requirement already satisfied: requests in c:\users\sollasi\anaconda3\lib\site-packages (from torchvision) (2.24.0)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\sollasi\anaconda3\lib\site-packages (from
torchvision)
(8.0.1)
Requirement already satisfied: numpy in c:\users\sollasi\anaconda3\lib\site-packages (from torchvision) (1.22.3)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\sollasi\anaconda3\lib\site-packages (from requests-
>torchv
ision) (2020.6.20)
Requirement already satisfied: idna<3,>=2.5 in c:\users\sollasi\anaconda3\lib\site-packages (from requests-
>torchvision)
 (2.10)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in c:\users\sollasi\anaconda3\lib\site-packages
(
from requests->torchvision) (1.25.11)
Requirement already satisfied: chardet<4,>=3.0.2 in c:\users\sollasi\anaconda3\lib\site-packages (from requests-
>torchvi
sion) (3.0.4)
Installing collected packages: torch, torchvision, torchaudio
Successfully installed torch-1.12.0 torchaudio-0.12.0 torchvision-0.13.0

C:\Users\SolLaSi>
```

PyTorch Documentation:
https://pytorch.org/docs/master/

☝ ***Another method (Anaconda/Conda):*** Without creating virtual environment in Anaconda/Conda: No need to use ***pip3 installer***.

1. Open anaconda Powershell prompt, run as adminstrator
2. Run following codes to install TensorFlow in base(root):

```
conda activate base
conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

```
(base) PS C:\Windows\system32> conda activate base
(base) PS C:\Windows\system32> conda install pytorch torchvision torchaudio cpuonly -c pytorch
Collecting package metadata (current_repodata.json): done
Solving environment: |
The environment is inconsistent, please check the package plan carefully
The following packages are causing the inconsistency:

  - defaults/win-64::anaconda==2020.11=py38_0
  - defaults/win-64::astropy==4.0.2=py38he774522_0
  - defaults/win-64::bkcharts==0.2=py38_0
  - defaults/win-64::bokeh==2.2.3=py38_0
  - defaults/win-64::bottleneck==1.3.2=py38h2a96729_1
  - defaults/noarch::dask==2.30.0=py_0
  - defaults/win-64::h5py==2.10.0=py38h5e291fa_0
  - defaults/noarch::imageio==2.9.0=py_0
  - defaults/win-64::matplotlib==3.3.2=0
  - defaults/win-64::matplotlib-base==3.3.2=py38hba9282a_0
  - defaults/win-64::mkl_fft==1.2.0=py38h45dec08_0
  - defaults/win-64::mkl_random==1.1.1=py38h47e9c7a_0
  - defaults/win-64::numba==0.51.2=py38hf9181ef_1
  - defaults/win-64::numexpr==2.7.1=py38h25d0782_0
  - defaults/win-64::numpy==1.19.2=py38hadc3359_0
  - defaults/win-64::pandas==1.1.3=py38ha925a31_0
  - defaults/win-64::patsy==0.5.1=py38_0
  - defaults/win-64::pytables==3.6.1=py38ha5be198_0
  - defaults/win-64::pywavelets==1.1.1=py38he774522_2
```

```
  - defaults/win-64::scikit-image==0.17.2=py38h1e1f486_0
  - defaults/win-64::scikit-learn==0.23.2=py38h47e9c7a_0
  - defaults/win-64::scipy==1.5.2=py38h14eb087_0
  - defaults/noarch::seaborn==0.11.0=py_0
  - defaults/win-64::statsmodels==0.12.0=py38he774522_0
  - defaults/win-64::tifffile==2020.10.1=py38h8c2d366_2
failed with initial frozen solve. Retrying with flexible solve.
Solving environment: failed with repodata from current_repodata.json, will retry with next repodata
source.
Collecting package metadata (repodata.json): done
Solving environment: -
The environment is inconsistent, please check the package plan carefully
The following packages are causing the inconsistency:

  - defaults/win-64::anaconda==2020.11=py38_0
  - defaults/win-64::astropy==4.0.2=py38he774522_0
  - defaults/win-64::bkcharts==0.2=py38_0
  - defaults/win-64::bokeh==2.2.3=py38_0
  - defaults/win-64::bottleneck==1.3.2=py38h2a96729_1
  - defaults/noarch::dask==2.30.0=py_0
  - defaults/win-64::h5py==2.10.0=py38h5e291fa_0
  - defaults/noarch::imageio==2.9.0=py_0
  - defaults/win-64::matplotlib==3.3.2=0
  - defaults/win-64::matplotlib-base==3.3.2=py38hba9282a_0
  - defaults/win-64::mkl_fft==1.2.0=py38h45dec08_0
  - defaults/win-64::mkl_random==1.1.1=py38h47e9c7a_0
  - defaults/win-64::numba==0.51.2=py38hf9181ef_1
  - defaults/win-64::numexpr==2.7.1=py38h25d0782_0
  - defaults/win-64::numpy==1.19.2=py38hadc3359_0
  - defaults/win-64::pandas==1.1.3=py38ha925a31_0
  - defaults/win-64::patsy==0.5.1=py38_0
  - defaults/win-64::pytables==3.6.1=py38ha5be198_0
  - defaults/win-64::pywavelets==1.1.1=py38he774522_2
  - defaults/win-64::scikit-image==0.17.2=py38h1e1f486_0
  - defaults/win-64::scikit-learn==0.23.2=py38h47e9c7a_0
  - defaults/win-64::scipy==1.5.2=py38h14eb087_0
  - defaults/noarch::seaborn==0.11.0=py_0
  - defaults/win-64::statsmodels==0.12.0=py38he774522_0
  - defaults/win-64::tifffile==2020.10.1=py38h8c2d366_2
done

## Package Plan ##

  environment location: C:\Users\SolLaSi\anaconda3

  added / updated specs:
    - cpuonly
    - pytorch
    - torchaudio
    - torchvision


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    _anaconda_depends-2020.07  |           py38_0           6 KB
    anaconda-custom            |           py38_1          36 KB
    certifi-2022.6.15          |   py38haa95532_0         153 KB
    conda-4.12.0               |   py38haa95532_0        14.5 MB
    cpuonly-2.0                |                0           2 KB  pytorch
    gmpy2-2.1.2                |   py38h7f96b67_0         160 KB
    libllvm9-9.0.1             |       h21ff451_0          61 KB
    libuv-1.40.0               |       he774522_0         255 KB
    mpc-1.1.0                  |       h7edee0f_1         260 KB
    mpfr-4.0.2                 |       h62dcd97_1         1.5 MB
    mpir-3.0.0                 |       hec2e145_1         1.3 MB
    openssl-1.1.1q             |       h2bbff1b_0         4.8 MB
    pytorch-1.12.0             |      py3.8_cpu_0       133.7 MB  pytorch
    pytorch-mutex-1.0          |             cpu           3 KB  pytorch
    snappy-1.1.9               |       h6c2663c_0         2.2 MB
    tbb-2021.5.0               |       h59b6b97_0         149 KB
    torchaudio-0.12.0          |         py38_cpu         3.5 MB  pytorch
    torchvision-0.13.0         |         py38_cpu         6.2 MB  pytorch
    ------------------------------------------------------------
                                           Total:       168.7 MB

The following NEW packages will be INSTALLED:

  _anaconda_depends  pkgs/main/win-64::_anaconda_depends-2020.07-py38_0
  cpuonly            pytorch/noarch::cpuonly-2.0-0
  gmpy2              pkgs/main/win-64::gmpy2-2.1.2-py38h7f96b67_0
  libllvm9           pkgs/main/win-64::libllvm9-9.0.1-h21ff451_0
  libuv              pkgs/main/win-64::libuv-1.40.0-he774522_0
  mpc                pkgs/main/win-64::mpc-1.1.0-h7edee0f_1
```

```
  mpfr              pkgs/main/win-64::mpfr-4.0.2-h62dcd97_1
  mpir              pkgs/main/win-64::mpir-3.0.0-hec2e145_1
  numpy-base        pkgs/main/win-64::numpy-base-1.19.2-py38ha3acd2a_0
  pytorch           pytorch/win-64::pytorch-1.12.0-py3.8_cpu_0
  pytorch-mutex     pytorch/noarch::pytorch-mutex-1.0-cpu
  snappy            pkgs/main/win-64::snappy-1.1.9-h6c2663c_0
  tbb               pkgs/main/win-64::tbb-2021.5.0-h59b6b97_0
  torchaudio        pytorch/win-64::torchaudio-0.12.0-py38_cpu
  torchvision       pytorch/win-64::torchvision-0.13.0-py38_cpu

The following packages will be UPDATED:

  ca-certificates                            2020.10.14-0 --> 2022.4.26-haa95532_0
  certifi           pkgs/main/noarch::certifi-2020.6.20-p~ --> pkgs/main/win-64::certifi-2022.6.15-
py38haa95532_0
  conda                                4.9.2-py38haa95532_0 --> 4.12.0-py38haa95532_0
  openssl                                 1.1.1h-he774522_0 --> 1.1.1q-h2bbff1b_0

The following packages will be DOWNGRADED:

  anaconda                                    2020.11-py38_0 --> custom-py38_1


Proceed ([y]/n)? y


Downloading and Extracting Packages
openssl-1.1.1q       | 4.8 MB    | ##################################### | 100%
pytorch-mutex-1.0    | 3 KB      | ##################################### | 100%
cpuonly-2.0          | 2 KB      | ##################################### | 100%
libuv-1.40.0         | 255 KB    | ##################################### | 100%
libllvm9-9.0.1       | 61 KB     | ##################################### | 100%
snappy-1.1.9         | 2.2 MB    | ##################################### | 100%
mpc-1.1.0            | 260 KB    | ##################################### | 100%
torchaudio-0.12.0    | 3.5 MB    | ##################################### | 100%
_anaconda_depends-20 | 6 KB      | ##################################### | 100%
mpir-3.0.0           | 1.3 MB    | ##################################### | 100%
pytorch-1.12.0       | 133.7 MB  | ##################################### | 100%
torchvision-0.13.0   | 6.2 MB    | ##################################### | 100%
mpfr-4.0.2           | 1.5 MB    | ##################################### | 100%
gmpy2-2.1.2          | 160 KB    | ##################################### | 100%
anaconda-custom      | 36 KB     | ##################################### | 100%
tbb-2021.5.0         | 149 KB    | ##################################### | 100%
certifi-2022.6.15    | 153 KB    | ##################################### | 100%
conda-4.12.0         | 14.5 MB   | ##################################### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
(base) PS C:\Windows\system32>
```

❒ ***Verification:*** To ensure that PyTorch was installed correctly, we can verify the installation by running sample PyTorch code. Here we will construct a randomly initialized tensor.

From the command line, type:

```
python
```

then enter the following code:

```
import torch
x = torch.rand(5, 3)
print(x)
```

The output should be something similar to:

```
tensor([[0.3380, 0.3845, 0.3217],
        [0.8337, 0.9050, 0.2650],
        [0.2979, 0.7141, 0.9069],
        [0.1449, 0.1132, 0.1375],
        [0.4675, 0.3947, 0.1426]])
```

Additionally, to check if your GPU driver and CUDA is enabled and accessible by PyTorch, run the following commands to return whether or not the CUDA driver is enabled:

```
import torch
torch.cuda.is_available()
```

☞ ***Use Conda powershell prompt:***

```
(base) PS C:\Windows\system32> python
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32

Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> x = torch.rand(5, 3)
>>> print(x)
tensor([[0.0298, 0.0806, 0.8720],
        [0.0490, 0.3365, 0.4080],
        [0.7628, 0.1640, 0.0817],
        [0.0627, 0.7596, 0.3849],
        [0.1151, 0.4519, 0.9773]])
>>>
```

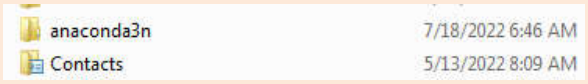
☟ ***No Anaconda (Local Python):*** Intalling on `C:\Users\SolLaSi\AppData\Local\Programs\Python\Python38`. Alongside ***anaconda***.

☝ ***Without anaconda*** or *No Anaconda environment*. (if any "***anaconda***" version installed previously)

NOTE: If ***Anaconda3*** or any version installed in your pc, goto: ***C:\Users\user_name*** in my case ***C:\Users\SolLaSi***

☞ Find the folder named " ***anaconda3***" rename it, so that ***pip*** *doesn't* ***locate*** *it* and by default it uses ***local python directory***.

☞ Rename "***anaconda3***" as "***anaconda3n***" or "***anaconda3bak***" or whatever you want. It is temporary, we'll undo it after ***Pytorch*** installed in ***Local Python***.




```
pip install torch
pip install torchvision
pip install torchaudio
```

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\SolLaSi>pip install torch
Collecting torch
  Using cached torch-1.12.0-cp38-cp38-win_amd64.whl (161.9 MB)
Requirement already satisfied: typing-extensions in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-pac
kages (from torch) (4.2.0)
Installing collected packages: torch
Successfully installed torch-1.12.0
WARNING: There was an error checking the latest version of pip.

C:\Users\SolLaSi>pip install torchvision
Collecting torchvision
  Using cached torchvision-0.13.0-cp38-cp38-win_amd64.whl (1.1 MB)
Requirement already satisfied: torch==1.12.0 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-package
s (from torchvision) (1.12.0)
Requirement already satisfied: requests in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-packages (fr
om torchvision) (2.27.1)
Requirement already satisfied: numpy in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-packages (from
torchvision) (1.22.3)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site
-packages (from torchvision) (9.1.0)
Requirement already satisfied: typing-extensions in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-pac
kages (from torchvision) (4.2.0)
Requirement already satisfied: idna<4,>=2.5 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-packages
 (from requests->torchvision) (3.3)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-pa
ckages (from requests->torchvision) (2021.10.8)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site
-packages (from requests->torchvision) (1.26.9)
Requirement already satisfied: charset-normalizer~=2.0.0 in c:\users\sollasi\appdata\local\programs\python\python38\lib\
site-packages (from requests->torchvision) (2.0.12)
Installing collected packages: torchvision
Successfully installed torchvision-0.13.0
WARNING: There was an error checking the latest version of pip.

C:\Users\SolLaSi>pip install torchaudio
Collecting torchaudio
  Using cached torchaudio-0.12.0-cp38-cp38-win_amd64.whl (969 kB)
Requirement already satisfied: torch==1.12.0 in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-package
s (from torchaudio) (1.12.0)
Requirement already satisfied: typing-extensions in c:\users\sollasi\appdata\local\programs\python\python38\lib\site-pac
kages (from torch==1.12.0->torchaudio) (4.2.0)
Installing collected packages: torchaudio
```

```
Successfully installed torchaudio-0.12.0
WARNING: There was an error checking the latest version of pip.

C:\Users\SolLaSi>
```

Verification:

```
C:\Users\SolLaSi>python
Python 3.8.10 (tags/v3.8.10:3d8993a, May  3 2021, 11:48:03) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> x = torch.rand(5, 3)
>>> print(x)
tensor([[0.0342, 0.2619, 0.0458],
        [0.4530, 0.1518, 0.9431],
        [0.3442, 0.6217, 0.1544],
        [0.8963, 0.9547, 0.4306],
        [0.7427, 0.4610, 0.1228]])
>>>
```

☞ Now rename "***anaconda3n***" to "***anaconda3***" or again.

14.4.1 Problem description

📣 We are gonna make a ***recommender system*** that will ***predict*** if a user is going to like a movie: ***yes*** or ***no***. This one predicts a binary outcome, ***1*** or ***0***, that is ***yes*** or ***no***.

📣 We'll build another ***recommender system*** that is going to ***predict*** the ***rating*** of a movie by a user using ***Auto Encoders*** in next chapter (predicts a rating from ***1*** to ***5***).

☝ And this way you have the ***two recommender systems*** that are mostly used in the ***industry***.

❒ So we're gonna make the **recommender system** that **predicts** a ***binary outcome***: ***yes*** or ***no*** with our **Restricted Boltzmann machines *(RBM)***.

❒ ***Dataset:*** For both these recommended systems (with ***RBM*** and ***AE***) we're gonna start with the same data set. This is the most real world data set that you can find online. It is called ***MovieLens***.

☞ We downloaded the dataset for ***100k*** and ***1m***. We extracted the files in following folders: "***movie_lens_100k***" and "***movie_lens_1m***".

☞ There are different size dataset in MovieLens. In our Project we use 100k and 1million ratings dataset.

The dataset source is MovieLens: https://grouplens.org/datasets/movielens/

MovieLens 25 million: https://files.grouplens.org/datasets/movielens/ml-25m.zip
MovieLens 10 million: https://files.grouplens.org/datasets/movielens/ml-10m.zip
MovieLens 1 million: https://files.grouplens.org/datasets/movielens/ml-1m.zip
MovieLens 100k: https://files.grouplens.org/datasets/movielens/ml-100k.zip

14.4.2 Overview of the project

For two different recommender systems with ***RBM*** and ***AE*** (next chapter), we have a common data preprocessing phase.

[1]. We're going to *import* the data set,

[2]. prepare the ***training set*** and the ***test set***.

[3]. We're going to get the ***number of users*** and the ***number of movies***.

[4]. And then we're going to ***convert*** our ***data*** into an ***array*** where we have our ***users*** in *lines*, and ***movies*** in *columns*.

[5]. And finally, in the **last step** of this data preprocessing phase, we will convert the data into ***torch.Tensor***.

☞ After these ***data preprocessing*** steps we will start the steps that are specific to ***Boltzmann machines***. We're going to start dealing with ***binary ratings***, then create an *architecture of a neural network*, that will be a ***Restricted Boltzmann Machine (RBM)*** and note that, it is a ***Probabilistic Graphical Model***.

❒ We have our two data sets, the ***1 million ratings data set***, and the ***100K ratings data set***.

☞ Remember, we're going to train our RBM on this one, the one 100k ratings, but of course you can practice on this data set as well, if you want to evaluate more the performance.

❒ ***AItRBM:*** We have this PDF file named ***AItRBM-proof.pdf***, also available online, that contains all the ***theory behind the neural network*** that we are about to make. So I strongly encourage to have a look at this ***PDF file***, if you can read it, that's really, ***really good***. It contains all the ***theory***, ***explains*** on the ***intuitive level***, but also it goes into the ***math*** pretty much in ***detailed***.

☞ Here all you need to know about ***graphical models***, because a **Restricted Boltzmann Machine (RBM)** is a ***Probabilistic Graphical Model***.

☞ Here you have the theory of ***graphical models***, with ***Undirected Graphs*** and ***Markov Random Fields***.

☞ You have a section on ***Unsupervised Learning***.

☞ You have the mathematical details on the ***computations of the likelihood***.

☞ The ***KL-divergence***, and some optimization theory, which is of course very useful for ***Boltzmann machines***.

☞ And then you have some theory about ***MCMC*** techniques (***Markov chain Monte Carlo techniques***), very important, because we're going to use ***Gibbs sampling*** to **estimate** the **gradient** of the **likelihood**, and ***Gibbs*** sampling is based on ***MCMC***.

- ➢ You have the definition of a ***Markov chain***, if you want to go deeper in mathematics,
- ➢ And then you have of course ***Gibbs sampling*** that's very important, at the heart of ***Boltzmann machines***,
- ➢ We will remind the intuition behind ***Gibbs sampling***, when we ***implement*** our ***Boltzmann machines***.

with the edge between units V_j and H_i and b_j and c_i are real valued bias terms associated with the jth visible and the ith hidden variable, respectively.

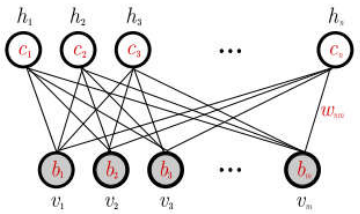


Fig. 1. The undirected graph of an RBM with n hidden and m visible variables

☞ Finally, you have, of course, our ***Restricted Boltzmann Machines*** (RBM).

- ➢ There is the formula for the energy that we're going to try to ***minimize***, because we're trying to ***minimize*** the ***free energy***, and that is by ***maximizing*** the ***log likelihood***.
- ➢ Here you have the ***architecture*** of the ***Restricted Boltzmann Machines***, with the ***visible nodes***, that's our ***input***, which are going to be the ***ratings*** of the ***movies*** by the ***users***, and
- ➢ The users are going to be the ***different observations*** going into the ***network***, one by one.
- ➢ You have the ***hidden nodes***, and so all this makes the architecture of the ***RBM***, and that's exactly what we're going to make, in ***Python***, by building a ***class***. And mostly this class will contain the ***Contrastive Divergence technique***, that will be used to ***maximize*** the ***likelihood***.

❒ In this class, that we're going to call ***RBM***, we will implement the ***Contrastive Divergence*** technique. We will implement the following algorithm, ***k-step contrastive divergence*** this is this algorithm, the ***heart*** of our ***RBM***.

Algorithm 1. k-step contrastive divergence

Input: RBM $(V_1, \ldots, V_m, H_1, \ldots, H_n)$, training batch S
Output: gradient approximation Δw_{ij}, Δb_j and Δc_i for $i = 1, \ldots, n$, $j = 1, \ldots, m$

```
init Δw_ij = Δb_j = Δc_i = 0 for i = 1,...,n, j = 1,...,m
forall the v ∈ S do
    v^(0) ← v
    for t = 0,...,k − 1 do
        for i = 1,...,n do sample h_i^(t) ~ p(h_i | v^(t))
        for j = 1,...,m do sample v_j^(t+1) ~ p(v_j | h^(t))
    for i = 1,...,n, j = 1,...,m do
        Δw_ij ← Δw_ij + p(H_i = 1 | v^(0)) · v_j^(0) − p(H_i = 1 | v^(k)) · v_j^(k)
        Δb_j ← Δb_j + v_j^(0) − v_j^(k)
        Δc_i ← Δc_i + p(H_i = 1 | v^(0)) − p(H_i = 1 | v^(k))
```

14.4.3 Data Preprocessing 1 : Train & Test set

Now we're going to import the libraries that will be using to implement our RBM,

```
# Importing the libraries
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable
```

i. ***NumPy***, because we will be working with ***NumPy*** arrays.
ii. ***Pandas*** to import the dataset and create the ***training set*** and the ***test set***,

☞ And then we have all the Torch libraries,

iii. So, for example this ***nn*** is the module of ***Torch*** to implement ***Neural Networks***,
iv. ***parallel*** for the ***Parallel Computations***,
v. ***optim*** for the ***Optimizer***,
vi. ***Variable*** is for ***Stochastic Gradient Descent***.

❒ ***Importing Movie dataset:*** Now we're going to import the dataset. This will be different, because we'll import the dataset part-by-part, because the dataset is not that simple and we'll need to use some of the arguments:

☞ The first dataset we're going to import is all our ***movies***. We have to import ***movies.dat***.

```
# importing the dataset
movies = pd.read_csv("./movie_lens_1m/movies.dat", sep= "::", header=None, engine="python", encoding="latin-1")
```

➢ The path that contains the dataset, **"./movie_lens_1m/movies.dat"**

➢ The separator the ***default separator*** is a ***comma***, that works for ***csv*** file, where the features are ***separated*** by ***commas***.

➲ But here that's not the case because some of the ***titles*** of the movies contain ***commas inside the title***, hence we cannot use the ***comma*** as a ***separator***. Therefore the separator is a ***double colon***, like this **'::'**. So **sep= "::"**

➲ If you open the ***movies.dat*** file, you will see that the movies are ***separated*** by their ***ratings*** and their other ***features***, by a ***double colon*** like this.

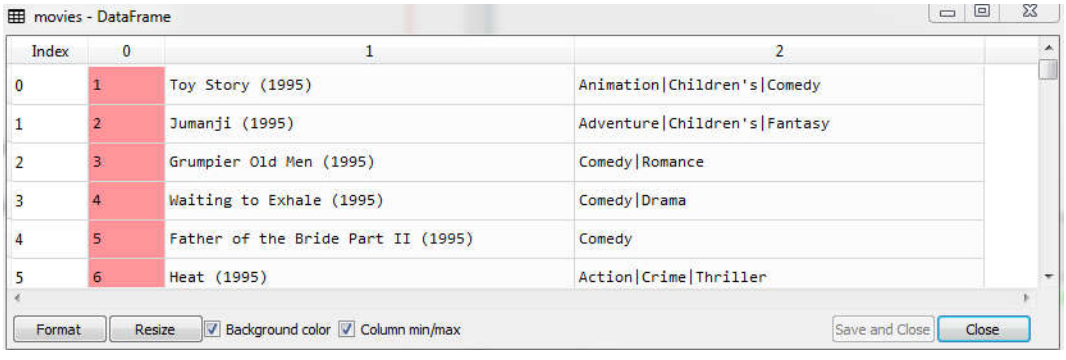
➢ The third parameter is the header. Because actually the file movies.dat, doesn't contain ***headers***, that is, ***names of columns***. And therefore we need to specify this because, the **default value** of **header** is not ***None*** (no column names), and therefore we need to specify that there is no column names, and to do this we put **header=None**.

➢ Then the next parameter is going to be ***engine***, and this is to make sure that the dataset gets imported correctly. And we will use the Python engine, ***engine = "python"***.

➢ Last argument, is the encoding. And we need to input a different encoding than usual because some of the movie titles contain special characters that cannot be treated properly with the classic encoding, ***UTF-8***. So, we're just adding this encoding argument because of some of the special characters in the movie titles. ***encoding = "latin-1"***.

movies = pd.read_csv("./movie_lens_1m/movies.dat", sep= "::", header=None, engine="python", encoding="latin-1")

☝ NOTICE we didn't use ***header="None"***, because ***"None"*** is string, and ***None*** is a **data-type**.



☝ In this ***Movieland*** database, we have thousands of movies, and for each of these movies you have this ***first column*** which is the ***movie ID***, we will use the ***movie ID*** to make a ***recommender system***, we will ***not*** be using the ***titles***. It will be much ***more simple*** with the ***movies' IDs***.

☝ Actually we will not be using this dataset to make the ***training set*** or the ***test set***. It is just to show you what's going on with all the movies.

❒ ***Users data-set:*** Same As above we just ***copy-paste*** and give ***new variable***. All augments are same, we just need to change the file path.

☞ These are all the information is about the different users.

- ***First column*** is the ***user ID***,
- the ***second column*** is the ***gender***,
- the ***third column*** is the ***age***,
- the ***fourth column*** is some codes that correspond to the ***user's job***, and
- the ***last column*** is the ***zip code***.

useRs - DataFrame

Index	0	1	2	3	4
0	1	F	1	10	48067
1	2	M	56	16	70072
2	3	M	25	15	55117
3	4	M	45	7	02460
4	5	M	25	20	55455

❒ ***User ratings dataset:*** We just ***copy-paste*** and give ***new variable***. All augments are same, we just need to change the file path.

☞ Now its really ***important*** to understand the ***structure***, because we are getting closer to the ***training set*** and the ***test set*** we'll make, to ***train*** our ***model***.

➢ The ***first column*** corresponds to the ***users***. So this ***1*** here that we see, corresponds to the ***first user*** of the ***database***. So all these ***1's*** here correspond to the ***same user***.

➢ Then the ***second column*** corresponds to the ***movies***. And the ***numbers*** (1193, 661 etc) that we see here are the ***movies' IDs***.

➲ That are contained in the ***movies*** DataFrame, and so that's why we imported this DataFrame, it's for you to see which movie IDs corresponds to which movie, just if you want to play or test the recommender system in the end.

RaTings - DataFrame

Index	0	1	2	3
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291
5	1	1197	3	978302268

➢ The third column corresponds, of course, to the ***ratings***. So the ratings go from ***one*** to ***five***, ***1*** means that the user ***didn't like*** the movie, and ***5*** means that user ***absolutely loved*** the movie.

➲ For example, this second line here of index 2, means that, the user number 1, rated the movie number 914, and gave it 3 stars.

➢ The ***fourth column*** we absolutely ***don't care***, these are just the ***timestamps***, that basically specifies ***when each user rated*** the ***movie***. We will ***remove*** this data afterwards when creating the ***training set*** and the ***test set***.

```
# importing the dataset
movies = pd.read_csv("./movie_lens_1m/movies.dat", sep= "::", header=None, engine="python",
encoding="latin-1")

useRs = pd.read_csv("./movie_lens_1m/users.dat", sep= "::", header=None, engine="python",
encoding="latin-1")

RaTings = pd.read_csv("./movie_lens_1m/ratings.dat", sep= "::", header=None, engine="python",
encoding="latin-1")
```

❒ ***Training set*** and the ***test set***: Now we're going to prepare the ***training set*** and the ***test set***.

☞ We are going to take the ***100k ratings*** data set, which is in the ***movie_lens_100k*** folder. It is the five ***train-test-split*** of the whole dataset composed of ***100,000 ratings***.

☞ So as you can see, we have we have ***u1.base*** and ***u1.test***; ***u2.base*** and ***u2.test***; . . . . ; ***u5.base*** and ***u5.test***. Each one of those pairs of sets are actually some separate ***training set*** (base means training set) and ***test set***. These splits will help us to use ***K-fold CV***. In this case we use ***5-fold*** instead of ***10-fold***.

☞ For now we only use ***u1.base*** and ***u1.test*** as *train-set* and *test-set*.

☝ Fist we import data as ***data-frame*** and later we convert it to an ***array***.

☞ The ***separator*** for this U1 based file, in this case ***not a double colon***, but a ***tab***. We need to specify it, because the ***default*** separator is a ***comma***.

```
# preparing the training set and test set
training_set = pd.read_csv("./movie_lens_100k/u1.base", delimiter="\t")
```

☞ The ***delimiter tab*** should rather be taken with this ***delimiter argument*** rather than previously used ***sep argument***.

❒ ***Split-size:*** What is the split? I mean, what is the ***proportion*** of the ***training set*** compared to the ***whole set***?

☞ Remember, the ***original*** dataset contains ***100,000*** **ratings**. And since each observation corresponds to one rating, Here in ***training-set*** we have ***80,000*** **observations**.

☞ That means that we have ***80,000*** ratings. And therefore, the training set is ***80%*** of the original dataset composed of the ***100,000*** ratings. So that will be an ***80%, 20% train-test split***. That's the optimal train-test split to train a model.

☞ The first column (not including Index) corresponds to the users, the ***second column*** corresponds to the ***movies*** and the ***third column*** corresponds to the ***ratings*** and then the ***fourth column*** corresponds to the ***timestamps***.

☞ Take the ***index-4 row***, it is the ***5th*** observation, the ***user*** number ***1*** rated the ***movie*** number ***7***, and gave it ***4 stars***.

training_set - DataFrame

Index	1	1.1	5	874965758
0	1	2	3	876893171
1	1	3	4	878542960
2	1	4	3	876893119
3	1	5	3	889751712
4	1	7	4	875071561
5	1	8	1	875072484

❒ ***NymPy Array:*** Now we have to convert it into an array because ***Pytouch.tensors*** expects the data as array.

☞ Also we ***convert*** all number into ***Integers***, to do that we used ***dtype="int"***

```
train_set = np.array(training_set, dtype="int")
```

❒ ***test_set:*** Now we're gonna do the ***same*** for the ***test set***,

☞ Notice that the ***training set*** and the ***test set*** have ***different ratings***, you know, there is ***no common rating*** of a ***same movie*** by the ***same user*** between the ***training set*** and the ***test set***.

test_set - NumPy object array

	0	1	2	3
0	1	10	3	875693118
1	1	12	5	878542960
2	1	14	5	874965706
3	1	17	3	875073198
4	1	20	4	887431883

☞ However we have the same users, we start with ***user 1*** as in the ***training-set*** and ***test-set***. But for this same ***user 1*** we won't have the ***same movies*** because the ***ratings*** are ***different***.

test_set	Array of int32	(19999, 4)	[[1 10 3 875693118] [1 12 ...
train_set	Array of int32	(79999, 4)	[[1 2 3 876893171] [1 3 ...

```
# preparing the training set and test set
training_set = pd.read_csv("./movie_lens_100k/u1.base", delimiter="\t")
train_set = np.array(training_set, dtype="int")

ts_set = pd.read_csv("./movie_lens_100k/u1.test", delimiter="\t")
test_set = np.array(ts_set, dtype="int")
```

✋ Next, we will get the ***maximum number*** of ***users*** and the ***maximum number*** of ***movies*** in two ***separate variables*** because then we will need these ***two variables*** to prepare our ***RBM***.

14.4.4 Data Preprocessing 2 : max User & max Movies

Now we're gonna get the total number of users and movies. Then we are going to ***convert*** our ***training set*** and ***test set***, into a ***matrix*** where the lines/rows are going to be ***users***, the ***columns*** are going to be ***movies***, and the ***cells*** are going to be the ***ratings***.

❒ We are going to create such a ***matrix*** for the ***training set***, and another one for the ***test set***. And, in ***each*** of these two ***matrices***, we want to include all the ***users*** and all the ***movies*** from the ***original data set***.

☞ In the training-set, if a user ***didn't rate*** a movie, well we'll put a 0, into the corresponding cell of the matrix.

❒ These ***matrices*** will have the ***same number*** of ***users*** and the ***same number*** of ***movies***, so they will have the ***same number*** of ***rows*** and the ***same number*** of ***columns***.

☞ In these two matrices, each cell of indexed **U**, **I**, where **U** is the user and **I** is the movie. Each cell **U**, **I** will get the rating of the movie **I**, by the user **U**. And if, this user **U** didn't rate the movie **I**, we'll put a **0**.

☞ We're gonna make a function to do this.

☞ Since these ***two matrices*** will contain the ***total number*** of ***users*** and the total ***number of movies***, we need to ***find*** those numbers.

❒ ***Finding max-user and max-movies:*** We could ***scroll*** our ***dataset*** and find out ***those numbers***, and manually implement those numbers. But to make our code **more flexible**, we use ***max()*** function, it make us enable to use ***any train-test set*** and finds the corresponding ***max-user*** and ***max-movies*** automatically.

☞ Manually using ***maximum number*** can cause ***problem*** because ***different train-test split*** may have different amount of ***users*** and ***movies***.

☞ So we use the code that finds the maximum from both ***train set*** and ***test set***.

```
# Getting the number of Users and Movies
nb_users = int(max(max(train_set[:, 0]), max(test_set[:, 0])))
nb_movies = int(max(max(train_set[:, 1]), max(test_set[:, 1])))
```

nb_movies	int	1	1682
nb_users	int	1	943

✋ NOTICE:

☝ **max(train_set[:, 0]** finds the ***maximum*** from the first column (user-Id) of ***training-set***. **max(train_set[:, 0]** similarly finds the ***maximum*** from the ***test-set***. Then we pick the *maximum* of these *2 maximum numbers*. Then we convert those to integers.

☝ For ***movies***, **int(max(max(**train_set[:, 1]), **max(**test_set[:, 1])))** finds the ***maximum of maximums*** from 2nd columns (Movie No.) from ***train-set*** and ***test-set***.

☝ Also notice how we changed the ***index no.*** **0** & **1** to select the corresponding columns for ***Users ID*** and ***Movie No***.

🛠 ***Next*** we ***convert*** our ***training set*** and ***test set***, into two ***matrices*** where the ***lines*** are the ***users***, the ***columns*** are the ***movies***, and the ***timestamps*** will be ***removed***.

14.4.5 Data Preprocessing 3 : Creating Matrices

Now we are going to convert our ***training set*** and ***test set*** into an ***array*** with ***users*** in *lines* and ***movies*** in *columns*.

❒ Because we need to make a ***specific structure*** of data that ***RBM expects*** as *inputs*. We will have the observations in lines and the features in columns.

☞ We're just making the ***usual structure*** of data for ***neural networks*** or even for ***machine learning*** in general that is with the ***observations*** in *lines* and the ***features*** in *columns*.

🛠 Since we're gonna do this for both the ***training set*** and the ***test set***, we're gonna create a ***function*** that we will apply on both ***training set*** and the ***test set***.

☞ We're gonna call this **function *convert***, we need to give an arguments to the function, which will be our ***dataset***.

☞ We could use 2-dimensional ***NumPy array***. But we're gonna use ***Torch*** afterwards, we ***won't create a two-dimensional NumPy array***, we will create a ***list of lists***.

☞ Since we have ***943*** users, we'll have ***943 lists***, these will be ***horizontal lists***, each corresponds one ***user***. Each user is a ***list*** of ***observations*** in ***lines***. i.e the ***first*** list will correspond to the ***first user***, the ***second*** list will correspond to the ***second user***.

☞ Each ***list*** contains the ratings of the ***1,682 movies*** by the ***user*** corresponding to the ***list***.

☞ If the ***user didn't rate*** the **movie**, then we'll get a **0** for that.

☝ That's why the new ***converted training set*** and ***test set*** will have the ***same size*** because basically for both the ***training set*** and the ***test set***, we are considering all the **users** and all the **movies**, and we just put a ***0*** when the user ***didn't rate*** the movie.

☝ So, this whole ***list of lists*** will be a list of ***943 lists*** because we have ***943 users***, and each of these ***943 lists*** will be a ***list*** of ***1,682 elements*** because we have ***1,682 movies***.

```
# converting the data into an array with users in lines and movies in column.
def conVert(data):
    new_data = []
    for id_user in range(1, nb_users + 1):
        # use "data[:, 0] == id_user" as condition over movie column "data[:, 1]"
        id_movies = data[:, 1][data[:, 0]== id_user]    # returns a list

        # use "data[:, 0] == id_user" as condition over ratins column "data[:, 2]"
        id_ratings = data[:, 2][data[:, 0]== id_user]

        # vector of zeros
        ratings = np.zeros(nb_movies)
        ratings[id_movies - 1] = id_ratings

        new_data.append(list(ratings))

    return new_data

trn_set_cnvt = conVert(train_set)
tst_set_cnvt = conVert(test_set)
```

🛠 Notice how we get the list of movies that a user rated:

```
id_movies = data[:, 1][data[:, 0]== id_user]    # returns a list
```

☞ Here, ***data[:, 1]*** selects the ***movie column*** (2nd column), then we apply the condition: **"data[:, 0]== id_user"**, it gets the list of movies that are rated by the **id_user**.

🛠 Similarly we get the corresponding ratings, that are given by **id_user**

```
id_ratings = data[:, 2][data[:, 0]== id_user]
```

☞ Here, ***data[:, 2]*** selects the ***ratings column*** (3rd column), then we apply the condition: **"data[:, 0]== id_user"**, it gets the list of movies that are rated by the **id_user**.

☞ **data[:, 0]== id_user** means from the user_id column (first column) select the item matches the **id_user**.

☞ Following creates a ***list of zeros*** named ***ratings*** of size ***nb_movies***.

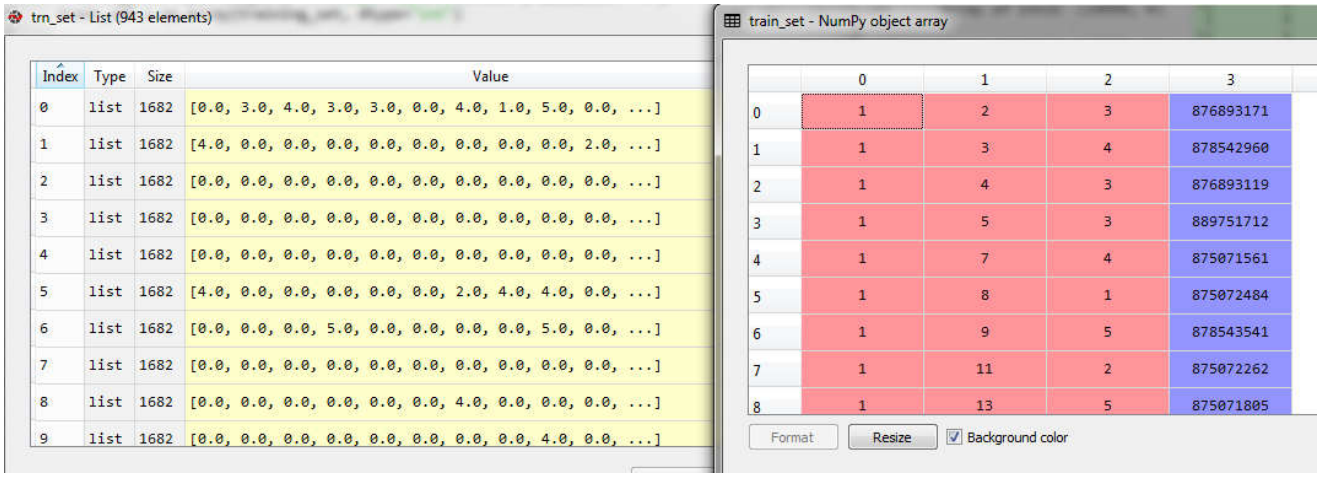
```
ratings = np.zeros(nb_movies)
```

In the ***ratings*** list we put a movie of id "**n**" at the index "**n-1**", since the ids of movies starts from **1**.

🛠 ***List operation:*** Also notice the following list operation, it puts the ratings ***id_ratings*** given by ***id_user*** to the corresponding movie ***id_movies***. Indexes in Python start at zero, and our movies' IDs start at one.

☞ Note that both ***id_movies*** and ***id_ratings*** are ***lists***, so following uses each element of ***id_movies*** as an ***index*** and from ***id_ratings*** uses each element as corresponding ratings.

```
ratings[id_movies - 1] = id_ratings
```



🛠 Notice each line has different ratings now. Since user ***1 didn't rated*** movies **1, 6, 10, 12** and many-others then there are ***0.0*** for these.

🛠 Also we used ***list()*** method, to make sure that ***ratings*** will be a *list*:

```
new_data.append(list(ratings))
```

✋ Note in following, range is because there is ***no user 0*** in our dataset, and for **loop** excludes *"**nb_users + 1**"*, so our users will be ***1*** to ***nb_users***.

```
for id_user in range(1, nb_users + 1):
```

14.4.6 Data Preprocessing 4 : Convert Matrices to torch.Tensor

Now we've converted our ***training set*** and our ***test set*** into the ***arrays*** composed of the ***users*** in *lines* and the *movies* in *columns*.

❒ The *columns* are the ***features*** that are going to be the *input nodes* in the *network*.

☞ For each ***user*** we will have its ***ratings of all the movies***, ***zeros*** included, and these ratings are going to be the *input nodes* for this observation going into the *network*.

❒ Then ***PyTorch*** comes into play, because we will build the architecture with ***PyTorch tensors***.

☞ A tensor is a multi-dimensional matrix but instead of being a ***NumPy array***, this is a ***PyTorch array***.

☞ In fact, we could build a ***NN*** with ***NumPy arrays***, but that would be much less efficient and that's why we're using ***tensors*** using the ***torch.Tensors***.

✋ NOTE: With ***TensorFlow*** we have exactly the same. With ***TensorFlow*** we work with tensors.

☝ Those are another ***kind*** of ***tensor***, another kind of ***multi-dimensional matrix***, and so we could also implement our ***AE/RBM*** from scratch with ***TensorFlow***.

☝ But for ***AE/RBM***, ***PyTorch*** gives *better results*, and also this is much more simple.

```
# Converting the data into Torch Tensosrs
train_set_tensor = torch.FloatTensor(trn_set_cnvt)
test_set_tensor = torch.FloatTensor(tst_set_cnvt)
```

❒ Now ***training set*** and the ***test set*** are ***torch.Tensor***, two, separate *multi-dimensional matrices* based on ***PyTorch***.

☞ We used the **class** ***FloatTensor***, that creates two object of this class: ***train_set_tensor*** and ***test_set_tensor***. These objects will be the ***torch.Tensor*** itself.

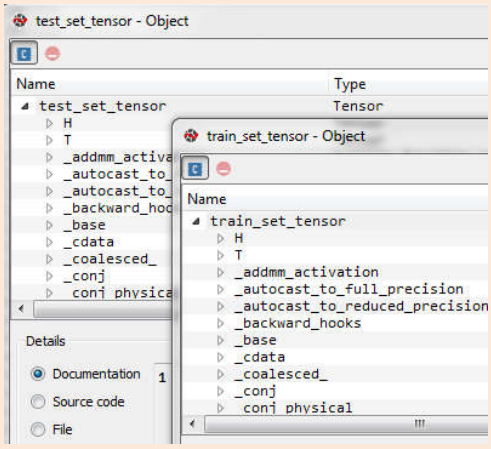
☞ A ***torch.Tensor*** is a multi-dimensional matrix with a *single data-type*. Since we're taking the ***FloatTensor*** class, the data-type will be ***float***.

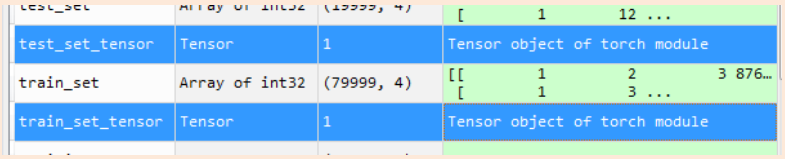
☞ Inside each of those classes, we used *one argument* which has to be a *list of lists*, our ***train-set*** and ***test-set*** (list-of-lists).

➢ The ***FloatTensor*** class expects a ***list of lists***.

This will give the exact, same matrix with the ***users*** in ***lines*** and the ***movies*** in ***columns***, but instead of being a ***NumPy array***, this will be a ***torch.Tensor***.

Now, I have to warn you, the training set and the test set in variable explorer may ***disappear***, because the variable explorer pane in ***Spyder*** may ***not recognize torch.Tensors*** yet.





Now the ***common data pre-processing*** for recommended system is done and now it's time to take care of what is specific to ***RBM***.

Remember, with ***RBM***, we're gonna ***predict*** if a user likes ***yes*** or ***no*** a movie (binary prediction). So we have to convert all the ratings into ***binary ratings***, ***0*** or ***1***. Because these are gonna be the ***inputs*** of our ***RBM***.

All data-preprocessing at once

```
# ----------- RBM : Recommender ---------------

# Importing the libraries
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable

# importing the dataset
movies = pd.read_csv("./movie_lens_1m/movies.dat", sep= "::", header=None, engine="python", encoding="latin-1")
useRs = pd.read_csv("./movie_lens_1m/users.dat", sep= "::", header=None, engine="python", encoding="latin-1")
RaTings = pd.read_csv("./movie_lens_1m/ratings.dat", sep= "::", header=None, engine="python", encoding="latin-1")

# preparing the training set and test set
training_set = pd.read_csv("./movie_lens_100k/u1.base", delimiter="\t")
train_set = np.array(training_set, dtype="int")

ts_set = pd.read_csv("./movie_lens_100k/u1.test", delimiter="\t")
test_set = np.array(ts_set, dtype="int")


# Getting the number of Users and Movies
nb_users = int(max(max(train_set[:, 0]), max(test_set[:, 0])))
nb_movies = int(max(max(train_set[:, 1]), max(test_set[:, 1])))


# converting the data into an array with users in lines and movies in column.
def conVert(data):
    new_data = []
    for id_user in range(1, nb_users + 1):
        # use "data[:, 0] == id_user" as condition over movie column "data[:, 1]"
        id_movies = data[:, 1][data[:, 0]== id_user]    # returns a list

        # use "data[:, 0] == id_user" as condition over ratins column "data[:, 2]"
        id_ratings = data[:, 2][data[:, 0]== id_user]
```

```
        # vector of zeros
        ratings = np.zeros(nb_movies)
        ratings[id_movies - 1] = id_ratings

        new_data.append(list(ratings))

    return new_data

trn_set_cnvt = conVert(train_set)
tst_set_cnvt = conVert(test_set)


# Converting the data into Torch Tensosrs
train_set_tensor = torch.FloatTensor(trn_set_cnvt)
test_set_tensor = torch.FloatTensor(tst_set_cnvt)
```

14.4.7 Prepare Input Data : Convert rating-tensors to binary-rating-tensor

We have to convert these ratings for our RBM. Right now, we have ratings from ***1*** to ***5***,in our ***training set*** and ***test set***.

❒ Now we have to convert these *ratings* into *binary ratings*. Because our ***RBM-recommender system*** will generate *binary-ratings*.

☞ ***1 = liked***, or ***0 = not liked***.

✋ Why do we have to convert these ratings?

☝ Because we want to predict some binary ratings.

☝ We also *need* the *inputs* to have the *binary format* **0** or **1**, because, the ***RBM*** will take the *input vector*, from this vector, ***RBM*** will ***predict*** the ***ratings*** for the *movies* that were ***not originally rated*** by the *user*.

☝ Since these *predicted ratings* are computed, originally, from the *existing ratings* of the *input vector*, well, then the *predicted ratings* in the output must have the *same format* as the existing ratings in the *input*.

➲ Otherwise, things would be inconsistent for the RBM.

❒ Let's convert all these ratings into ***binary ratings***, **1** or **0** for both the ***training set*** and the ***test set***.

☞ We're gonna replace all the **0** in this original training set, by **-1**. Because all the ***zeroes*** in the **original training set**, corresponded to the movies that were ***not rated*** by the ***users***.

➢ Now, minus one (**-1**) will mean that there was *not a rating* for a *specific movie*, given by a *specific user*.

```
train_set_tensor[train_set_tensor == 0] = -1
```

➢ And in **[ ]** brackets, we simply added the *condition* that we want to get these *ratings*. This is a *torch-tensor-operation*, it finds all **0**'s in the ***elements*** of the ***tensor*** and replace these with "**-1**".

❒ We're gonna do the same for the other ratings, that is, the ratings from **1** to **5**.

☞ The ratings that we want to ***convert*** into ***zero***, that is, not liked. Are the movies that were given ***one star*** or ***two stars***.

☝ Unfortunately, the "**or**" doesn't work with ***torch*** objects. There is no option in ***torch.tensor*** object that we can apply compound condition as:

```
(test_set_tensor == 1) or (test_set_tensor == 2)
```

➢ The **or** operator doesn't work like that for PyTorch. That's why we have to do these rating saperately:

```
train_set_tensor[train_set_tensor == 1] = 0
train_set_tensor[train_set_tensor == 2] = 0
```

➢ this *torch-tensor-operations*, finds all **1**,**2**'s in the ***elements*** of the ***tensor*** and replace these with "**0**".

☞ For the movies that the users liked, we make them 1 for those movies that are rated ***3***,***4*** or ***5***.

```
train_set_tensor[train_set_tensor >= 3] = 1
```

➢ In **[ ]** brackets, we simply added the ***condition*** that "**>= 3**", this ***torch-tensor-operation***, finds all **3, 4, 5**'s in the ***elements*** of the ***tensor*** and replace these with "**1**".

➢ The movies that were rated at least three stars were rather liked by the users. So, three stars, four stars, five stars are gonna become one.

❒ Now we do the same for test-set

```
test_set_tensor[test_set_tensor == 0] = -1
test_set_tensor[test_set_tensor == 1] = 0
test_set_tensor[test_set_tensor == 2] = 0
test_set_tensor[test_set_tensor >= 3] = 1
```

☞ Now all the ratings from one to five will be converted into binary ratings in both the training set and the test set. So, we're getting our inputs ready to go into the RBM, and then RBM will returns the ratings (predicts) of the movies that were ***not originally rated*** in the input vector.

```
# Converting the ratings into binary ratings: 1 (liked),  0 (not-liked)
train_set_tensor[train_set_tensor == 0] = -1
        # torch doesn't support combined condition
train_set_tensor[train_set_tensor == 1] = 0
train_set_tensor[train_set_tensor == 2] = 0
train_set_tensor[train_set_tensor >= 3] = 1

test_set_tensor[test_set_tensor == 0] = -1
test_set_tensor[test_set_tensor == 1] = 0
test_set_tensor[test_set_tensor == 2] = 0
test_set_tensor[test_set_tensor >= 3] = 1
```

☝ That's Unsupervised Deep Learning, and that's exactly how it works.

To view our tensors as a NumPy array

🛠 Converting a ***torch.tensor*** object into a **NumPy** array object: You need to call **.detach()** before saving your data e.g. **x.detach().numpy()** if your ***tensors*** have ***grads***...also you might need to call **cpu()**. I think this should work: **x.detach().cpu().numpy()**

☝ Here ***x*** is the ***torch.tensor*** object

```
train_tensor_to_view = train_set_tensor.detach().cpu().numpy()
test_tensor_to_view = test_set_tensor.detach().cpu().numpy()
```

☝ However we can simply use: `train_set_tensor.detach().numpy()` or `train_set_tensor.numpy()`. For our simple dataset, we can use following:

```
train_tensor_to_view = train_set_tensor.numpy()
test_tensor_to_view = test_set_tensor.numpy()
```



Some real life examples converting tensors to numpy array:

Example: Shared storage

PyTorch tensor residing on CPU shares the same storage as numpy array `na`

```
import torch
a = torch.ones((1,2))
print(a)
na = a.numpy()
na[0][0]=10
print(na)
print(a)
```

Output:

```
tensor([[1., 1.]])
[[10.  1.]]
tensor([[10.,  1.]])
```

Example: Eliminate effect of shared storage, copy numpy array first

To avoid the effect of shared storage we need to `copy()` the numpy array `na` to a new numpy array `nac`. Numpy `copy()` method creates the new separate storage.

```
import torch
a = torch.ones((1,2))
print(a)
na = a.numpy()
nac = na.copy()
nac[0][0]=10
print(nac)
print(na)
print(a)
```

Output:

```
tensor([[1., 1.]])
[[10.  1.]]
[[1. 1.]]
tensor([[1., 1.]])
```

Now, just the `nac` numpy array will be altered with the line `nac[0][0]=10`, `na` and `a` will remain as is.

Example: CPU tensor with `requires_grad=True`

```
import torch
a = torch.ones((1,2), requires_grad=True)
print(a)
na = a.detach().numpy()
na[0][0]=10
print(na)
print(a)
```

Output:

```
tensor([[1., 1.]], requires_grad=True)
[[10.  1.]]
tensor([[10.,  1.]], requires_grad=True)
```

In here we call:

```
na = a.numpy()
```

This would cause: *RuntimeError: Can't call numpy() on Tensor that requires grad. Use tensor.detach().numpy() instead.*, because tensors that `require_grad=True` are recorded by PyTorch AD. Note that `tensor.detach()` is the new way for `tensor.data`.

This explains why we need to `detach()` them first before converting using `numpy()`.

Example: CUDA tensor with `requires_grad=False`

```
a = torch.ones((1,2), device='cuda')
print(a)
na = a.to('cpu').numpy()
na[0][0]=10
print(na)
print(a)
```

Output:

```
tensor([[1., 1.]], device='cuda:0')
[[10.  1.]]
tensor([[1., 1.]], device='cuda:0')
```

Example: CUDA tensor with `requires_grad=True`

```
a = torch.ones((1,2), device='cuda', requires_grad=True)
print(a)
na = a.detach().to('cpu').numpy()
na[0][0]=10
print(na)
print(a)
```

Output:

```
tensor([[1., 1.]], device='cuda:0', requires_grad=True)
[[10.  1.]]
tensor([[1., 1.]], device='cuda:0', requires_grad=True)
```

Without `detach()` method the error `RuntimeError: Can't call` numpy() `on Tensor that requires grad. Use tensor.detach().numpy() instead.` will be set.

Without `.to('cpu')` method `TypeError: can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first.` will be set.

You could use `cpu()` but instead of `to('cpu')` but I prefer the newer `to('cpu')`.

14.4.8 RBM-architecture : RBM class - `__init__()`

❒ Now we build the ***architecture*** of our NN i.e, the architecture of the ***RBM***.

☞ We're gonna make a ***class***, which will ***define*** the *architecture of the RBM*.

☞ And then, we simply, create an ***object*** of this ***class***, that object will be the ***RBM model***.

❒ Here we will choose the *number* of *hidden nodes*, and mostly, we will build the newel network just like how it works, that is, we're gonna make this ***Probabilistic Graphical Model***. Because, let's remember, a ***RBM*** is a *probabilistic graphical model*.

❒ The class that we build for RBM, will contain following. These are first parameters that we need to initialize the RBM, using "**__init__**".

- The ***number*** of hidden ***nodes***
- The ***weights***.
- The ***weights*** for the ***probability*** of the ***visible node***, given the ***hidden node***.
- The ***bias*** for the same ***probability***
- The ***bias*** for the ***probability*** of the ***visible node,*** given the ***hidden node***.

We also add some functions, for example a self-driving car will need a lot of functions. Some ***functions*** to recognize ***objects*** on the ***street***. Then some ***functions*** to ***turn right***, to ***turn left***, to move ***forward***, to move ***backward***, or to ***stop*** when there is an ***obstacle*** on the street.

❒ We're gonna make 3 functions.

[1] One function to ***initialize*** the ***RBM object*** that will be created afterwords.

[2] Second function will be ***sample_H***, that will ***sample*** the ***probabilities*** of the ***hidden nodes*** given the ***visible nodes***. $P(h|v)$.

[3] Third function will be ***sample_V***, that will sample the ***probabilities*** of the ***visible nodes*** given the ***hidden nodes***. $P(v|h)$.

[4] 4th function will be ***train***, that will apply the ***Contrastive divergence***.

❒ **__init__: __init__** function is to define the parameters of the object that will be created once the class is made. There are 3 arguments:

☞ ***"self": self*** corresponds to the ***object*** that will be created afterwords. All the variables that are attached to the object will be created by putting a ***self*** before the variable.

☞ **nv**: The second argument is **nv**, and that's the ***number*** of ***visible nodes***.

☞ **nh**: The third argument is **nh**, the ***number*** of ***hidden nodes***.

☞ Then we need to initialize the ***weight*** and the ***bias***.

☞ Inside this **__init__** function there going to be all the ***parameters*** that we will ***optimize*** during the training of the ***RBM***.

i. the ***weights*** and

ii. the ***bias***.

➢ Let's start with the weights, we're gonna call them ***W***.

- Why capital W? Because ***all*** the ***weights*** are going to be ***initialized*** in a ***torch.tensor***.
- These weights are all the ***parameters*** of the ***probabilities*** of the ***visible nodes*** given the ***hidden nodes***. $P(v|h)$.
- According to the theory, they are ***initialized*** in a ***matrix*** of size ***nh***, number of hidden ***nodes***, and ***nv***, the number of visible ***nodes***. Since we're working with ***PyTorch***, well, this matrix is going to be a ***torch.tensor***. A tensor of one single type.

```
self.W = torch.randn(nh, nv)
```

- Since these weights have to be ***initialized randomly*** according to a ***normal distribution***, we need to use this ***rendn()*** of ***PyTorch***. It will initialize all the weights in our tensor.
- This `torch.randn(nh, nv)` initializes a tensor of size $nh \times nv$. According to a ***normal distribution***. This ***normal distribution*** has a ***mean*** of ***0*** and a ***variance*** of ***1***.

  That initializes all the weights for the probabilities of the ***visible nodes*** given the ***hidden nodes***.

➢ Let's initialize the ***bias***.

- There is some ***bias*** for the ***probability*** of the ***hidden node*** given the ***visible node*** $P(h|v)$, and
- some ***bias*** for the ***probability*** of the ***visible node*** given the ***hidden node*** $P(v|h)$.

➢ ***bias*** for $P(h|v)$ the ***probability*** of the ***hidden node*** given the ***visible node*** : We're gonna give the name "**a**".

```
self.a = torch.randn(1, nh)
```

- We're gonna use our ***rendn*** function again to ***initialize*** the ***weights*** according to ***normal distribution*** of ***mean 0*** and ***variance 1***. This will be a Tensor of size $1 \times nh$ (i.e. a ***vector*** of size of the ***hidden nodes***).

- Since there is ***one bias*** for ***each hidden node*** and we have ***nh*** hidden node, that's why we created a vector of ***nh*** element. And initialized to some numbers randomly that follow a normal distribution.
- But we need to create an ***additional dimension corresponding*** to the ***batch***, and therefore, this ***vector*** shouldn't have ***1 dimension***, like a single input vector, it should have ***2 dimensions***.
  - ☑ The ***first dimension*** corresponding to the ***batch*** and the ***second dimension*** corresponding to the ***bias***.
  - ☑ It's because the functions that we're gonna use of ***PyTorch cannot accept a single input vector*** of one dimension as argument, but a ***two dimensional tensor*** with the ***first dimension*** corresponding to the ***batch*** and the ***second dimension*** corresponding to the ***bias***.
  - ☑ So that's why here, we ***cannot*** put directly ***nh***.
- `torch.randn(1, nh)` creates a ***2-D tensor*** with this ***1*** here corresponding to the first dimension that is the ***batch***. And this ***nh*** element here corresponding to the ***bias***.

➢ The ***bias*** for *Probability* of the *visible node* given the *hidden node*, ***p_v_given_h***, $P(v|h)$ ***:*** It is same as above, we give it a name "**b**"

- This time we use **nv** (instead of **nh**), number of the visible node, while we initialize the ***tensor*** we make it a ***2D-tensor*** as we did above.

```
self.b = torch.randn(1, nv)
```

Those will initialize our ***future objects*** of the ***RBM class***.

```
# Creating the architecture of the Neural Netwark
class RBM():
    def __init__(self, nv, nh) :
        self.W = torch.randn(nh, nv)
        self.a = torch.randn(1, nh)
        self.b = torch.randn(1, nv)
```

14.4.9 RBM-architecture : RBM class – `sample_h()`

Now we make the function that will ***sample*** the ***hidden node***. The second function is about ***sampling*** the ***hidden nodes*** according to the probabilities, $P(h|v)$ where ***h*** is a ***hidden*** node and ***v*** is a ***visible*** node

❒ From the intuition section we saw, this probability $P(h|v)$ is nothing else than the ***sigmoid*** *activation function*.

☝ Why do we need this ***sample_h*** function? Because during the training we will *approximate* the *log likelihood gradient* and we will do that through *Gibbs sampling*.

☞ To apply *Gibbs sampling*, we need to compute $P(h|v)$, the *probabilities* of the *hidden nodes* given the *visible nodes*. Once we have this ***probability***, we can ***sample*** the *activations* of the *hidden nodes*.

❒ We're gonna call this function ***sample_h*** because it will return some *samples* of the different *hidden nodes* of our *RBM*.

☞ Suppose we have ***100 hidden nodes*** in our RBM. Well, this function will ***sample*** the *activations* of these *hidden nodes*, that is for ***each*** of these ***100 hidden nodes***, it will ***activate*** them according to a *certain probability* that we will compute in this same function. And for each of this hidden node, this **probability** is $P(h|v)$.

☞ That, is the ***probability*** that this ***hidden node = 1 given v*** that is given the value of **v**, $P(1|v)$. And that is this *probability* that is *equal* to the *activation function*.

☞ This ***sample_h*** function takes two arguments, ***self*** and ***x***. ***x*** will correspond to the **visible** neurons, ***v***, in the probabilities, **p h given v,** i.e. $P(h|v)$.

❒ First we have to compute $P(h|v)$. That is the probability that the *hidden* neuron *equals one* given the values of the *visible neurons*. That is actually our input vector of observations with all the ratings.

☞ It is nothing else than the **Sigmoid Activation Function** applied to $(wx + bias\_a)$.

➢ **wx** is the product of the *vector of weights* "W" with the *vector of visible neurons* "x".

➢ The bias, **a** responds to the *bias* of the *hidden nodes*.

[bias **b** corresponds to the *bias* of the *visible nodes*. We'll apply it to the sample_V function, for the visible nodes]

☯ ***Product wx:*** To compute the product of the ***weights W*** times the ***neurons***, that is x. To make the product of **two *tensors*** in ***Pytorch*** we need to use a function called ***mm*( )**.

```
wx = torch.mm(x, self.W.t())
```

➢ To make things ***mathematically correct***, we actually need to take the ***transpose*** of weights ***W***.

☯ ***Activation-value:*** Now let's compute what is going to be inside the ***sigmoid activation function***.

➲ The value for the activation function is: **(wx + bias)**.

➲ It is same as for every ***Deep-Learning model***, inside the ***activation function*** is a ***linear function*** of the ***neurons*** where the ***coefficients*** are the ***weights***.

➲ Here we have the ***bias = a***. So the value for activation is **(wx + a)**. We'll name this variable ***activation***.

```
activation = wx + self.a.expand_as(wx)
```

- We need to do something more. Remember that ***each input vector*** will ***not be treated individually***, but ***inside batches***. i.e. the new dimension that we created by using ***1's*** in `randn(1, nh)` and `randn(1, nv)`.
- Even if the batch contains one ***input vector*** or you know, one ***vector of bias***, well that ***input vector*** is still in the ***batch***. And in that case we call it a ***mini batch***.
- So when we add the **bias** *(bias of the **hidden nodes** = **a**)*, well we want to make sure that this ***bias*** is ***applied*** to ***each line of the mini batch***.
- To make sure of that, we need to use a ***function*** called `expand_as()` that will again, ***add a new dimension*** for these ***bias*** (bias **a** & **b**) that we're **adding**.
  - ❖ In ***parenthesis***, we need to specify how to expand the ***bias "a"*** by giving the ***tensor*** that we want to ***add with***. Since we are adding ***a*** with ***wx*** we used `expand_as(wx)`.
  - ❖ It is about making "***a***" as the ***similar format*** of "***wx***" before adding. That makes sure that the ***bias*** are ***applied*** to each line of the ***mini batch***.
  - ❖ It is this linear combinations of the the *visible neurons* ***x***, with coefficient ***W*** (*weights*) and added the ***bias***.

☯ ***Sigmoid-Activation-Function:*** Now we can ***compute*** the ***activation function*** that will ***activate*** the ***hidden node***. But remember this ***activation*** function ***represents*** a ***probability***. It will be the ***probability*** that the ***hidden node*** will be ***activated*** according to the ***value*** of the ***visible node***.

⇨ We give it a name ***p_h_given_v*** i.e. $P(h|v)$.

⇨ For example, let's say that we have a ***user*** that ***likes*** only ***drama-movies***.

- Well, if there is a ***hidden node*** that ***detected*** a specific ***feature*** corresponding to that ***drama genre*** of the movies, those drama-movies that are rated **1** by the user.
- Then the ***probability*** of that ***node*** specific to this ***drama (feature) genre***, given the ***visible node*** of that ***user*** who has ***all the nodes*** of the ***drama movies*** is equal to ***one***.
- This probability: **p_h_given_v** will be very ***high***, because **v = 1** *(user rated 1)* for the ***drama movies*** and **h** corresponds to the ***drama movie genre***. So $P(h|v)$ i.e $P(drama\_genre|1)$ will be very high.

```
p_h_given_v = torch.sigmoid(activation)
```

To use the ***sigmoid activation function*** we used ***sigmoid()*** method of ***PyTorch***. And use the value (linear combination of the neurons) "**activation**" to make it sigmoid.

☯ ***Returning "probability" and "sample of h":*** The final step is to return not only the probability ***p_h_given_v***, but of course a ***sample of h***.

⇨ ***Sample of h*** is the *sample of all the hidden nodes/neurons*, according to this probability ***p_h_given_v***.

⇨ Its important to understand is that we're making a ***Bernoulli RBM***, because we're just predicting a ***binary outcome*** (users like yes or no, a movie).

- Hence we'll return ***Bernoulli samples*** of that ***distribution***. Of that probabilities ***p_h_given_v***.
- ***What does that mean?***
  - ❖ Right now ***p_h_given_v*** is a ***vector*** of **nh** elements.
  - ❖ For example, suppose we have ***100 hidden nodes***, while this ***p_h_given_v*** vector is a vector of ***100 elements***. Each of these ***100 elements*** corresponds to each of the ***100 hidden nodes*** and each of these ***elements*** is the ***probability*** that the ***hidden*** node is ***activated***.

- Let's take the **i'th** element of this *vector*, it is the probability that the **i'th** *hidden node* is *activated*. But remember that's ***given*** the ***values*** of the ***visible nodes***. i.e given the ***ratings*** of the ***user*** we're dealing with.
  - So that's what we have in this ***p_h_given_v*** vector. And the idea is to use these ***probabilities*** to ***sample*** the ***activation*** of each of this ***100 hidden nodes***.
  - That is for each of these ***100 hidden nodes***, while depending on that probability ***p_h_given_v*** that we have for these hidden nodes, we will ***activate yes*** or ***no***, this **hidden neuron**.
  - ***And so how are we going to activate it?***

    Let's suppose that for a ***hidden neuron i***, the probability corresponding to that hidden neuron in this ***p_h_given_v*** vector is ***0.7***, ***70%***.

    In that case we take a ***random number*** between ***zero*** and ***one***, If this *random number* is ***below 0.7***, ***70%*** then we will ***activate*** the neuron. And if this *random number* is ***larger*** than ***0.7*** then we will ***not activate*** the neuron.

    That's how ***Bernoulli sampling*** works. And so we do that for ***each*** of the ***hidden nodes*** of our **100** *nodes*. And then in the end we get a ***vector*** of ***zeros*** and ***ones***.
    - The ***zeros*** correspond to the ***hidden nodes*** that were ***not activated*** *after the sampling*,
    - ***Ones*** correspond to the ***neurons*** that were ***activated*** by the sampling.

- ***How do we return that sampling of the hidden neurons?***
  - We have a torch function called `Bernoulli()`. In this function we have to input the distribution of which we are making that sampling, And i.e. ***p_h_given_v***.
  - So that will return all the ***probabilities*** of the ***hidden neurons***, given the values of the ***visible nodes*** (i.e. the ratings ***0*** or ***1***). and it will return also that ***sampling*** of the ***hidden neurons***.

```
def sample_h(self, x):
    wx = torch.mm(x, self.W.t())
    activation = wx + self.a.expand_as(wx)
    p_h_given_v = torch.sigmoid(activation)
    return p_h_given_v, torch.bernoulli(p_h_given_v)
```

That's the first function we need for ***Gibbs sampling***. But then we need another function, which is ***sample_v*** and basically we will do the same, but this time for the ***visible neurons***.

14.4.9 RBM-architecture : RBM class – `sample_v()`

We just implemented our ***sample_h*** function to ***sample*** the ***hidden node*** according to the probabilities ***p_h_given_v***.

- We're gonna do the same for the ***visible node***, from the values in the hidden nodes, that is whether they were activated or not.
  - We will also estimate the ***probabilities*** of the ***visible nodes***, that is the ***probabilities*** that ***each*** of the ***visible node equals one***.
  - Our goal in the end is to ***output*** the ***predicted ratings***, **1** or **0**, of the movies that were ***not originally rated*** by the ***user***, and these ***new ratings*** that we get in the end are taken from what we ***obtained*** in the ***hidden node***, that is from the ***samples*** of the ***hidden node***.
  - also, it's not the only reason to make sample_v, we also need it for Gibbs sampling when we approximate the ***log likelihood gradients***.

```
def sample_v(self, y):
    wy = torch.mm(y, self.W)
    activation = wy + self.b.expand_as(wy)
    p_v_given_h = torch.sigmoid(activation)
    return p_v_given_h, torch.bernoulli(p_v_given_h)
```

- We just need to copy this ***sample_h()*** function we will replace just one or two things. Our goal is:

  i. To return some ***samples*** of the ***visible node***

  ii. We also return the ***probability*** of ***v_given_h***,

  - We're gonna call our function ***sample_v*** that make some samples of the ***visible node*** according to the probabilities **p_v_given_h** $P(v|h)$.

- **p_v_given_h** means, given the values of the ***hidden nodes*** (i.e. whether the hidden nodes are activated or not)we return the ***probabilities*** that each of the ***visible nodes*** equals ***one***.

☞ As before we return some ***samples*** of the ***visible node*** based on that ***Bernoulli sampling***. It is *vector* of *probabilities* of the ***visible nodes***, since we have ***1,682 movies***, then we have ***1,682 visible nodes***, a vector of 1,682 probabilities, one probability for each of the visible nodes,

- So ***each*** of these ***probabilities*** is the probability that the corresponding ***visible node*** is equal to ***one*** but that remember is ***given*** the ***activations*** of the ***hidden nodes*** (activated or not).

- From the **vector** of probabilities ***p_v_given_h*** we return some ***sampling*** of the *visible nodes*.

- Let's say the **i**-th visible node has a probability of ***0.25***, then we take a random number between ***zero*** and ***one***.
  - ❖ If this number is below 0.25, then this ***visible node*** will get the value ***one***. So that means that we ***predict*** that the ***movie corresponding*** to that ***visible node*** will get a ***like*** by the ***user***,
  - ❖ If this random number is larger than ***0.25***, then this visible node will get the value ***zero***. And, therefore, we predict that the ***movie*** corresponding to that visible node will ***not get a like*** by the ***user***.

❒ We need to change what's inside the ***activation function***,

☞ first, we will replace variable ***x*** by ***y***.

```
def sample_v(self, y):
```

☝ Previously, **x** represented the ***visible node*** because we apply the ***sample_h()*** function to the *visible node* because we want to ***return*** the ***probabilities*** that the `hidden nodes = 1` according to the value of the `visible nodes`.

☞ But ***sample_v()*** function will return the ***probabilities*** of the ***visible nodes*** given the values of the ***hidden nodes***.

- The variable ***y*** is this time the ***values*** of the ***hidden nodes***.

☞ We take the ***torch product*** of ***matrices***, of ***tensors***, with ***y*** and the ***weight w***, the ***product*** of the ***hidden nodes*** with ***weight*** w, for the probabilities ***p_v_given_h***.

- ***No transpose:*** This time since we're making the product of the ***hidden nodes*** and the torch tensor of ***weight*** w, for the probabilities ***p_v_given_h***, well here we must not take the transpose.
  - Before, we had to the take the transpose because we were computing **p_h_given_v**.

- ***The reason is "Matrix-Product":*** W = ***matrix*** of size ***nh*** number of hidden *nodes*, and ***nv*** the number of visible *nodes*.

$$W = nh \times nv$$

x= *vector of visible nodes* ***nv***.

$$x = 1 \times nv$$

So $wx = x.W = (1 \times nv).(nh \times nv)$ is not possible because ***no. of rows of W*** need to same with ***no. of columns of x***.

That's why we need to take transpose,

$$wx = x.W^T = (1 \times nv).(nh \times nv)^T = (1 \times nv).(nv \times nh) = (1 \times nh)$$

⇨ But in the case of **y**, it is *vector of hidden nodes* ***nh***. $y = 1 \times nh$

So $wy = y.W = (1 \times nh).(nh \times nv)$ is possible. And we ***not need to take transpose***.

```
def sample_v(self, y):
        wy = torch.mm(y, self.W)
```

☞ ***Activation-value:***

```
activation = wy + self.b.expand_as(wy)
```

- We use ***b*** for the ***bias*** of the visible nodes. And we use `expand_as(wy)` to make ***b*** as same format of ***wy*** to apply the bias to each line of the mini-batch.

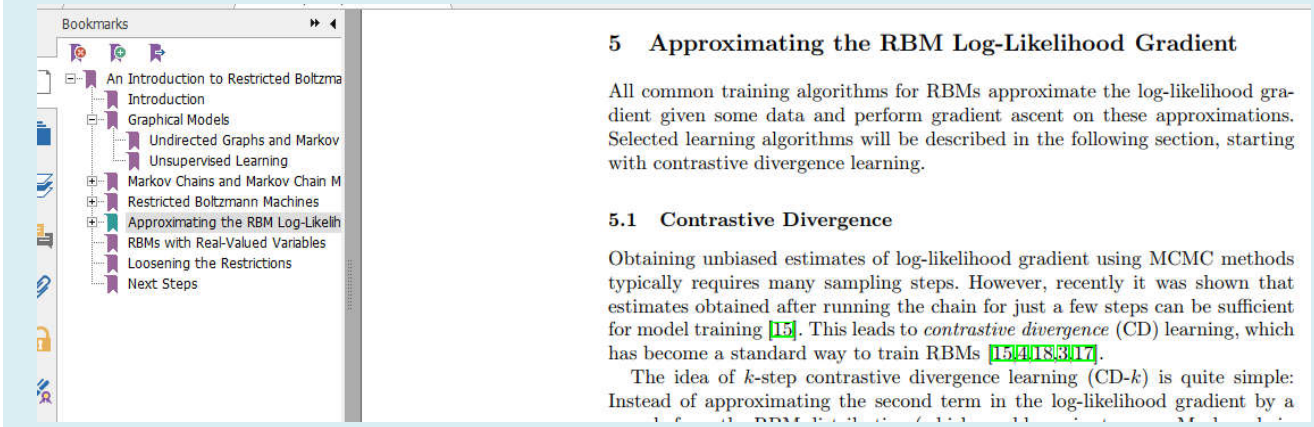
☞ We take the sigmoid activation function, and we return the ***probabilities*** that the ***visible nodes = one*** given the ***activations*** of the ***hidden nodes***. And we also return some ***sample*** using Bernoulli distribution.

```
        p_v_given_h = torch.sigmoid(activation)
        return p_v_given_h, torch.bernoulli(p_v_given_h)
```

14.4.10 RBM-architecture : RBM class – `train()`

The last function of this ***RBM*** class is ***tarin()*** it about ***Contrastive Divergence***,

❒ ***Contrastive Divergence*** used to **approximate** the **likelihood gradient**. Before we implement it lets review the paper:



☞ Contrastive divergence is all about ***approximating*** this ***log-likelihood gradient***.

☞ Remember, the RBM is an ***energy-based model***. We have this ***energy function*** that we are trying to ***minimize***.

➲ This ***energy function*** depends on the ***weight*** of the model, i.e. ***tensor of weights*** in our model.

➲ We need to ***optimize*** these ***weights*** to ***minimize*** the ***energy***.

☝ And not only it can be seen as an ***Energy-Based Model*** but it can also be seen as a ***Probabilistic Graphical Model***.

➧ In that case, the goal is to ***maximize*** the ***log-likelihood*** of the ***training set*** (equivalent to ***minimizing*** the ***energy***).

☝ In general, for any Deep-Learning/Machine-Learning model, we'll ***minimize the energy*** or ***maximize the log-likelihood*** to compute the ***gradient***.

☝ Because the ***direct computations*** of the ***gradient*** are *too heavy* and therefore, *instead of computing it directly*, we are gonna try to ***approximate*** it.

➧ These ***gradient approximations*** helps us to make these ***tiny adjustments*** in the direction of the ***minimum energy***.

✋ It is similar what we did in ***ANN***-part: trying to ***minimize*** a ***loss function*** and through ***Stochastic Gradient Descent (SGD)*** we were ***updating*** the ***weight*** in the ***direction*** of this ***minimum loss***.

☝ But here in ***RBM***, the ***computations*** are ***too heavy*** to compute these ***gradients*** *directly,* and so we need to come up with another solution, which is to ***approximate*** these ***gradients***.

☝ And the **algorithm** that will allow us to this is ***Contrastive Divergence***. This comes with ***Gibbs sampling***.

❒ ***Gibbs sampling:*** ***Gibbs sampling*** consists of creating this ***Gibbs chain*** in ***k-steps*** and this *Gibbs chain* in ***created*** exactly by ***sampling K times*** the ***hidden nodes*** and the ***visible nodes***.

☞ That is, we start with our ***input vector*** $v^{(0)}$, based on the ***probabilities*** $p(h|v^{(0)})$, we ***sample*** the ***first hidden*** nodes (first iteration).

☞ Then we take these ***sampled hidden nodes*** as ***input***. Let's call them $h^{(1)}$ to ***sample*** the ***visible nodes*** with the probabilities $p(v|h^{(1)})$.

☞ Then again we use these sample visible nodes for 2nd iteration, let's call them $v^{(0)}$, to ***sample again*** the ***hidden nodes*** with the ***probabilities*** $p(h|v^{(1)})$ and then again we ***sample the visible nodes*** and we ***sample the hidden nodes*** and we do this ***K times***.

☞ And that's exactly what this ***CD-K algorithm*** (K-step contrastive divergence) is about. The algorithm is below:

Algorithm 1. k-step contrastive divergence

Input: RBM $(V_1, \ldots, V_m, H_1, \ldots, H_n)$, training batch S
Output: gradient approximation Δw_{ij}, Δb_j and Δc_i for $i = 1, \ldots, n$, $j = 1, \ldots, m$

1 init $\Delta w_{ij} = \Delta b_j = \Delta c_i = 0$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$
2 **forall the** $v \in S$ **do**
3 $v^{(0)} \leftarrow v$
4 **for** $t = 0, \ldots, k-1$ **do**
5 **for** $i = 1, \ldots, n$ **do** sample $h_i^{(t)} \sim p(h_i \mid v^{(t)})$
6 **for** $j = 1, \ldots, m$ **do** sample $v_j^{(t+1)} \sim p(v_j \mid h^{(t)})$
7 **for** $i = 1, \ldots, n$, $j = 1, \ldots, m$ **do**
8 $\Delta w_{ij} \leftarrow \Delta w_{ij} + p(H_i = 1 \mid v^{(0)}) \cdot v_j^{(0)} - p(H_i = 1 \mid v^{(k)}) \cdot v_j^{(k)}$
9 $\Delta b_j \leftarrow \Delta b_j + v_j^{(0)} - v_j^{(k)}$
10 $\Delta c_i \leftarrow \Delta c_i + p(H_i = 1 \mid v^{(0)}) - p(H_i = 1 \mid v^{(k)})$

☞ So, in our ***train()*** function of the RBM class, we will simply ***implement*** these three lines ***8, 9, 10***. And we will simply ***update*** our ***tensor*** of ***weights, W***, our ***visible-node-bias b***, and our ***hidden-node-bias a***.

➲ In the paper c_i's are ***hidden-node-bias***, but in our algorithm we call them ***a***. That's the bias of ***p_h_given_v***.

❒ **train():** ***train()*** function has several arguments. (five arguments total).

i. The first one is, ***self*** to use object entities. We also add four more arguments, which are:

ii. ***Input Vector:*** We name it **v0**. That's our ***input vector*** containing the ***ratings*** of all the movies by ***one user***.

iii. ***vk:*** ***vk*** represents ***visible nodes*** obtained ***after K samplings*** (after ***K round trips*** from the ***visible nodes*** to the ***hidden nodes first*** and then way ***back*** from the ***hidden nodes*** to the ***visible nodes***).

❖ So that's the ***visible nodes*** obtained after ***K iterations*** and ***K contrastive divergence***.

iv. **ph0:** ***ph0*** is ***vector*** of ***probabilities*** that at the ***first iteration*** the **hidden nodes = 1** given the values of **v0**, (input vector).

v. ***phk:*** ***phk*** will correspond to the ***probabilities*** of the ***hidden nodes*** after ***K sampling*** given the values of the ***visible nodes***, ***vk***.

❒ ***Inside the function:*** We will first update our ***tensor of weights***, ***W*** then our ***bias b*** and then our ***bias a***.

☞ ***Updating weights:*** We take our ***weights*** and then add the product of ***rating of the movie j*** and the ***probabilities*** that the **hidden nodes = 1** given the values of $\mathbf{v^{(0)}}$.

$$\Delta w_{ij} + p\left(H_i = 1 \mid v^{(0)}\right) \cdot v_j^{(0)} - p\left(H_i = 1 \mid v^{(k)}\right) \cdot v_j^{(k)}$$

And then we subtract the product of ***probabilities*** that the **hidden nodes = 1** given the values of $\mathbf{v^{(k)}}$ after K iterations multiply by ***rating of the movie*** $v_j^{(k)}$ (the ***value*** of the ***visible node*** corresponding to the ***movie*** **j** after **k** ***iterations***.

➲ $p\left(H_i = 1 \mid v^{(0)}\right) \cdot v_j^{(0)}$ = `torch.mm(v0.t(), ph0)`. ***And*** $p\left(H_i = 1 \mid v^{(k)}\right) \cdot v_j^{(k)}$ = `torch.mm(vk.t(), phk))`.

```
def train(self, v0, vk, ph0, phk):
    self.W += (torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk))
```

☞ ***Updating bias b:*** We use the ***torch.sum()*** to calculate the difference between the ***input vector*** of observations and the ***visible nodes*** after ***K samplings***, **vo - vk** .

➲ We also gonna add ***zero*** just to keep the format of **b** as a ***tensor*** of ***two dimensions***.

```
self.b += torch.sum((v0 - vk), 0)
```

☞ ***Updating bias b:*** **a** contains the **bias** of the **probabilities** ***p_h_given_v***.

➲ The probabilities that the ***hidden nodes = 1***, given the values of **v0**, the input vector of observations. $p(H_i = 1 \mid v^{(0)}) =$ **ph0**

➲ The probabilities that the ***hidden nodes = 1***, given the values of **vk**, that is, the ***values*** of the ***visible nodes*** after ***K sampling***. $p(H_i = 1 \mid v^{(k)}) =$ **phk**

➲ We add ***zero*** to keep the format of **a** as a ***tensor*** of ***two dimensions***.

```
self.a += torch.sum((ph0 - phk), 0)
```

```
    def train(self, v0, vk, ph0, phk):
        self.W += (torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk))
        self.b += torch.sum((v0 - vk), 0)
        self.a += torch.sum((ph0 - phk), 0)
```

❒ Finally our ***RBM class*** is over. We now have what's at the ***heart*** of the ***RBM algorithm***:

i. Our sampling functions, ***sample_h()*** and ***sample_v()***

ii. Training function ***train()*** that will to ***Contrastive divergence solution*** with ***Gibbs sampling***.

RBM-class all at once

```
# Creating the architecture of the Neural Network
class RBM():
    def __init__(self, nv, nh) :
        self.W = torch.randn(nh, nv)
        self.a = torch.randn(1, nh)
        self.b = torch.randn(1, nv)

    def sample_h(self, x):
        wx = torch.mm(x, self.W.t())
        activation = wx + self.a.expand_as(wx)
        p_h_given_v = torch.sigmoid(activation)
        return p_h_given_v, torch.bernoulli(p_h_given_v)

    def sample_v(self, y):
        wy = torch.mm(y, self.W)
        activation = wy + self.b.expand_as(wy)
        p_v_given_h = torch.sigmoid(activation)
        return p_v_given_h, torch.bernoulli(p_v_given_h)

    def train(self, v0, vk, ph0, phk):
        self.W += (torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk))
        self.b += torch.sum((v0 - vk), 0)
        self.a += torch.sum((ph0 - phk), 0)
```

Now we have left to do is,

- Create an ***object*** of this ***RBM class***. This will be our ***model*** itself.
- Then we will train the ***model*** over ***several epochs*** so that we find these ***optimal weights*** that will ***predict*** the *ratings of the movies* that were ***not originally rated***.
- Now ***RBM class*** is *ready*, we can create ***object*** of this *class*. Also we can create ***several RBM models*** using this *class*.
- We can ***test*** many of them with ***different configurations***, with several numbers of ***hidden nodes*** because that's basically the main parameter.
- You can add some ***more parameters*** in your class, for example, a ***learning rate*** if you want to improve and tune your model.

Deep Learning

BM project - part 2: Building RBM model

14.5.1 RBM-model

To create our first RBM object we just call the class RBM with the two parameters: *no. of visible node =* ***nv*** and *no. of hidden node =* ***nh***.

- ☞ The object will then initialized (***__init__()*** will invoked), with biases ***a***, ***b*** and weight ***W***.
- ☞ Other functions ***sample_h()***, ***sample_v()*** and ***train()*** will be used during the training, they will not take any action unless they are called.

❒ ***no. of visible node:*** Here ***nv = no. of visible node = is the number of movies***; is a fixed parameter.

- ☞ At the start the ***visible nodes*** are the ***ratings*** of all the movies by a specific user and so we have ***one visible node*** for ***each movie***.

```
nv = len(train_set_tensor[0])
```

train_set_tensor[0] is the length of the ***first column*** of ***train-tensor***, which corresponds to ***movies***. We have 1682 movies so 1682 visible nodes.

❒ ***no. of hidden node:*** Here we can choose any number for ***nh***. The ***hidden nodes*** correspond to some ***features*** that are going to be ***detected*** by the ***RBM model***. These features will be some ***GENRE***, some ***ACTORS***, some ***DIRECTORS*** whether the movie's got an ***OSCAR***, etc.

- ☞ So the ***number of hidden nodes*** corresponds to the ***number of features*** we want to detect.

```
nh = 100
```

So let's say that we want to start by detecting ***100 features***. It's actually hard to say right now about the ***optimal number of features***. Of course we can ***tune*** it ***later***.

❒ ***Batch size:*** We already mentioned this concept of ***batch*** when we add extra dimension during ***creation of batches a & b.*** This ***dimension*** here represented by, **1**. So **1** corresponds to the ***batch***.

```
        self.a = torch.randn(1, nh)
        self.b = torch.randn(1, nv)
```

- ☞ When we train our algorithm, we will ***update*** the ***weights*** after a ***batch*** ***observations*** (instead of each observation). Each batch will contain same number of observations.
- ☞ Also this parameter is tunable. We can adjust it to improve the model.
- ☝ Now, we can set ***batch_size = 1***, in that case you're ***updating*** the ***weights*** after ***each observation*** going to the network. It will be very slow for large amount of data.
- ☞ For fast training we can take a large batch size for example ***batch_size = 100***. Since we have 943 observations the training will go very fast with batch size = 100.

```
batch_size = 100
```

❒ Finally we create the RBM object with parameters, ***nv*** & ***nh***.

```
# creating RBM model
nv = len(train_set_tensor[0])
nh = 100
batch_size = 100

rbm = RBM(nv, nh)
```

14.5.2 Training the RBM-model - part 1

To train the model over each epoch, we use a for loop. This loop will run for specific number of epoch. We just need to include the different functions (that we made in this RBM class) inside this for loop.

❒ ***Epoch:*** We first set the number of epochs. Let's choose for now 10 epochs. Because we have only 943 observations and besides we only have binary value **0** and **1** so the ***convergence*** will be reached ***pretty fast***.

```
# Training the RBM
nb_epoch = 10
for epoch in range(1, nb_epoch + 1):
    train_loss = 0
    s = 0.0
```

☞ For each epoch, ***all our observations*** will go into the network.

☞ We will update the weights after the observations of ***each batch*** passed through the network.

☞ At the end we'll get our ***final visible nodes*** with the ***new ratings*** for the *movies that were not originally rated*.

❒ ***Inside the for loop:*** Note that, for any deep learning algorithm we need a ***loss function*** to ***measure*** the ***error*** between the ***predictions*** and the ***real ratings***.

☞ ***Loss:*** In this training we will compare the predicted ratings to the ratings of the training set.

➲ Basically we will measure the ***difference*** between the ***predicted ratings***, that is either ***0*** or ***1***, and the real ratings, ***0*** or ***1***.

➲ For measuring the loss we can use different methods. Here in RBM we use ***simple difference*** in ***absolute values***. that measures, simply, the ***absolute difference*** between the ***predicted rating*** and the ***real rating***.

➲ But the most common loss measuring method is ***RMSE***, the ***Root Mean Square Error***, which is the ***root*** of the ***mean*** of the ***squared differences*** between the ***predicted ratings*** and the ***real ratings***. We'll use ***RMSE*** in next chapter to build ***Auto-Encoders***.

```
train_loss = 0
```

We will **initialize** it to ***zero***, because before we started training this ***loss*** is equal to ***0***. And then the ***loss*** will increase when we ***find*** some ***errors*** between the ***predicted ratings*** and the ***real ratings***.

☞ ***Counter:*** Here we're gonna need a ***counter***, we name it "***s***". Using this counter ***s***, we're going to **normalize** the *train_loss* (simply divide the ***train_loss*** by ***s***) . We will increment it after each epoch

```
s = 0.0
```

To make ***s*** a ***floating-number*** we use ***0.*** Or ***0.0***, either will work.

❒ ***Nested for loop for bath operation:*** When we made ***sample_h()***, ***sample_v()*** and ***train()*** functions regarding to one user.

☞ But the samplings and the ***contrastive*** of ***divergence*** algorithm have to be done over ***all the users***, but remember over ***all the users*** in the ***batch***.

☞ To get these batches of users, we need ***another for loop*** (a nested for loop).

☞ We loop over all the users and ***increment*** the user by ***batch_size***, It will divide the dataset into batches.

```
for u_id in range(0, nb_users - batch_size, batch_size):
```

Since we increment the loop variable by ***batch_size***, the upper bound must ***nb_users - batch_size***. This loop creates the batches, for user 0 to 99, 100 to 199 etc.

That's how we implement ***batch learning*** from ***scratch***. Everything happens in this loop.

☝ For lop syntax:

```
for i in range(lower_bound, upper_bound, increment):
```

❒ ***Inside the nested loop:***

[1]. We will get separately, our ***input*** and our ***target***.

➢ Our ***input*** is the ***ratings*** of all the ***movies*** by the ***specific user*** we're dealing with right now in the loop.

➢ The target is going to be ***at*** the ***beginning*** the **same as** the **input**.

- Since the ***input*** is gonna go into the ***Gibbs chain*** and will be ***updated*** to get the ***new ratings*** in ***each visible node*** then the ***input*** is going to ***change***, but the ***target*** will keep its ***same initial value***.

☞ ***input vector:*** We name the *input vector* ***vk***, it will go through the ***Gibbs chain*** and will be ***updated*** at each round trip. ***vk*** is going to be the ***output*** of ***Gibbs sampling***, after the ***K steps*** of the random walk.

- But at the start this ***vk*** is actually ***same a***s the ***input batch*** of observations (the ***ratings*** of the ***users***), in the ***batch***. The ratings that ***already existed***.
- To get is all the users in a batch starting from ***u_id*** up to ***u_id + batch_size***, we use following code:

```
vk = train_set_tensor[u_id:(u_id + batch_size)]
```

We selected the users from ***train_set_tensor***, from ***u_id*** up to ***u_id + batch_size***.

☞ ***Target:*** target is the same as the initial value of input at the beginning but it remain unchanged.

- It's the ***batch*** of ***original ratings*** that we want to ***compare***, in the end, to our ***predicted ratings***.
- We need it to ***measure*** the ***error*** between the ***predicted ratings*** and the ***real ratings*** to calculate ***train_loss***.
- So, we're gonna call this target ***v0***, so ***v0*** are our ratings of the movies that were ***already rated*** by the ***100 users*** in this ***batch***.

```
v0 = train_set_tensor[u_id:(u_id + batch_size)]
```

☞ ***Initial Probabilities:*** Before starting ***Contrastive Divergence*** with ***Gibbs Sampling*** we need to specify our *initial probabilities*.

- The **initial probabilities**, is actually ***ph0*** is the ***probabilities*** that the **hidden node** at the start equal **1 given** the **real ratings** $p\left(H_i = 1 \mid v^{(0)}\right) =$ **ph0**
- Now we use our ***sample_h()*** function of ***RBM*** **class**. Since we are getting the probabilities that the **hidden node = 1** given the **visible nodes** at the beginning.
  - ⇨ Because ***sample_h()*** returns ***p_h_given_v***, that is here it will return ***p_h_given_v0***.
  - ⇨ ***Using _ :*** Since this ***sample_h()*** function also returns both *probabilities* and the *samples* (Bernoulli samples) "**return p_h_given_v, torch.bernoulli(p_h_given_v)**", well we have to use a ***Python trick*** to only get the function returned first element, i.e. ***p_h_given_v***, and the ***Python trick*** to do that is to add here a ***comma*** and then an ***underscore***,

```
ph0,_ = rbm.sample_h(v0)
```

  - ⇨ The parameter of ***sample_h()*** is ***x***, and ***x*** corresponds to the *visible node*, because we want to ***sample*** the *hidden nodes* given the *visible nodes*.
  - ⇨ ***Which visible nodes:*** We want the **visible nodes** at the **start**, that is, **v0**. That is the *original ratings* of the movies for all the *users* of our *batch*.

```
# Training the RBM
nb_epoch = 10
for epoch in range(1, nb_epoch + 1):
    train_loss = 0
    s = 0.0
    for u_id in range(0, nb_users - batch_size, batch_size):
        vk = train_set_tensor[u_id:(u_id + batch_size)]
        v0 = train_set_tensor[u_id:(u_id + batch_size)]
        ph0,_ = rbm.sample_h(v0)

        for k in range(10):
```

14.5.3 Training the RBM-model - part 2 : k-step CD

❒ ***Another nested for loop:*** This for loop for the ***K steps of contrastive divergence***. In this for loop that we're gonna make the ***Gibbs chain***, that we're gonna do our ***k steps*** of the ***random walk***. Let's call the looping variable ***k*** in ***range(10)***.

```
for k in range(10):
```

☞ Actually our *k-steps* of the *Random Walk* in *Gibbs sampling* is an ***MCMC*** technique (***Markov Chain Monte Carlo*** technique).

❒ Here we'll explain what's going on with this random walk.

☞ Basically, ***Gibbs sampling*** consists of making ***Gibbs chain***. There are simply several round trips from the ***visible nodes*** to the ***hidden nodes***, and then from the ***hidden nodes*** to the ***visible nodes***.

☞ In each round trip of this ***Gibbs chain*** of ***Gibbs sampling*** well, the ***visible nodes*** are ***updated***. and step after step, we get closer to our good ***predicted ratings***.

☞ At the ***beginning*** we start with our ***input batch*** of observations. That is our ***input ratings*** in ***v0***, in our ***batch*** of ***100 users***.

❒ ***Sampling Hidden Nodes:*** In the first step of ***Gibbs sampling***, from this **batch input vector** ***v0*** of original ratings, we are going to ***sample*** the ***first hidden nodes*** using ***Bernoulli sampling*** using our ***p_h_given_v0*** distribution from ***sample_h()*** function.

☞ So the first step of ***k-steps contrastive divergence*** is to call ***sample_h()*** on the **visible nodes**, to get the ***first sampling*** of the ***first hidden nodes***.

☞ Since we actually want to get this **second element** ***returned*** by ***sample_h()***, we use the similar python trick using '_' as we did before.

☞ In this time, we are going to start with an **underscore** '_' and then **hk**, to be the ***hidden nodes*** obtained at the ***k-th step of contrastive divergence***.

```
_,hk = rbm.sample_h(vk)      # sampling hidden nodes
```

☞ Since we are doing the sampling of the ***first hidden nodes***, given the **values** of the ***first visible nodes***, that is our **original ratings**, the first ***input*** for our ***sample_h()*** function in this first step of Gibbs sampling is ***v0***.

➢ But be careful, ***v0*** is the ***target***, which we ***don't wanna change***. So we have to take this ***vk***, because at the first step ***v0 = vk***. ***vk*** so far is our input batch of observations, and then ***vk*** will be ***updated***.

❒ ***Sampling Visible Nodes:*** We'll update ***vk*** right after we update ***hk*** with the other sampling function which is ***sample_v()***.

☞ So right now $vk = v0$, but in the next step we update ***vk*** so that $vk \neq v0$.

☞ ***vk*** is going to be the ***sampled visible nodes*** after the first step of ***Gibbs sampling***, i.e after getting ***hk***.

☞ We'll get it using ***Bernoulli sampling*** using our ***p_v_given_h*** distribution from ***sample_v()*** function on the *first sample* of our *hidden nodes* i.e. ***hk***.

☞ Again we get this **second element** ***returned*** by ***sample_v()***, we use the similar python trick using ***underscore*** '_' .

```
_,vk = rbm.sample_v(hk)      # sampling visible nodes
```

➢ So we get our ***first update*** of the ***visible nodes*** after the ***first sampling***.

That's the first ***step*** of our ***random walk***, that is the first step of ***Gibbs sampling***.

```
for k in range(10):
    _,hk = rbm.sample_h(vk)      # sampling hidden nodes
    _,vk = rbm.sample_v(hk)      # sampling visible nodes
```

❒ So as the loop continues ***hk*** and ***vk*** update themselves repeatedly.

Algorithm 1. k-step contrastive divergence

Input: RBM $(V_1, \ldots, V_m, H_1, \ldots, H_n)$, training batch S
Output: gradient approximation Δw_{ij}, Δb_j and Δc_i for $i = 1, \ldots, n$, $j = 1, \ldots, m$

1 init $\Delta w_{ij} = \Delta b_j = \Delta c_i = 0$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$
2 **forall the** $\boldsymbol{v} \in S$ **do**
3 $\boldsymbol{v}^{(0)} \leftarrow \boldsymbol{v}$
4 **for** $t = 0, \ldots, k-1$ **do**
5 **for** $i = 1, \ldots, n$ **do** sample $h_i^{(t)} \sim p(h_i \mid \boldsymbol{v}^{(t)})$
6 **for** $j = 1, \ldots, m$ **do** sample $v_j^{(t+1)} \sim p(v_j \mid \boldsymbol{h}^{(t)})$
7 **for** $i = 1, \ldots, n$, $j = 1, \ldots, m$ **do**
8 $\Delta w_{ij} \leftarrow \Delta w_{ij} + p(H_i = 1 \mid \boldsymbol{v}^{(0)}) \cdot v_j^{(0)} - p(H_i = 1 \mid \boldsymbol{v}^{(k)}) \cdot v_j^{(k)}$
9 $\Delta b_j \leftarrow \Delta b_j + v_j^{(0)} - v_j^{(k)}$
10 $\Delta c_i \leftarrow \Delta c_i + p(H_i = 1 \mid \boldsymbol{v}^{(0)}) - p(H_i = 1 \mid \boldsymbol{v}^{(k)})$

☞ In continues until the end of the loop, when we'll get the ***10th sample*** of ***hidden nodes*** and the ***10th sample*** of ***visible nodes***.

❒ Getting ***hk*** and ***vk*** after the ***last step*** of the ***random walk***, we can now ***approximate*** the ***gradients*** using ***8***, ***9***, and ***10th*** steps of ***k-step-Contrastive-divergence algorithm***.

☞ What we did above is the step ***5*** & ***6*** of ***k-step-Contrastive-divergence algorithm***.

☞ We want to get the ***last sample*** after the ***last step*** of the ***random walk***, the ***last sample of* hidden nodes** and the ***last sample of visible nodes***.

☞ We need those last samples to update the ***weight*** and the ***bias*** to approximate the ***gradient***.

✋ Actually here we need the ***last sample of visible nodes***.

☝ But we don't directly use the ***last sample of the* hidden nodes**, that was just to get the ***last sample of the* visible nodes**.

☝ We just got our ***vk***, we can now ***approximate*** the ***gradient*** to update the ***weights*** and the ***bias***.

☝ We use ***tarin()*** function to update the ***weights*** and the ***bias***, from our ***RBM*** class.

❒ ***Ignore unrated movies during weight update:*** But before we apply this ***train()*** function to update the ***weight***, well we need to do something very important:

☞ We don't wanna learn from the ***movies*** which has ***no rating***, that is for the ***cells (movies)*** that have a ***minus one -1*** (no rating).

☞ We'll ignore these cells that contain the ***-1 ratings*** in the ***training process***,

☞ We're going to ***freeze*** these ***visible nodes*** that contain the ***-1 ratings***. So that it ***won't be possible*** to ***update*** them during ***Gibbs sampling***.

☯ ***How can we freeze these visible nodes containing the minus one ratings?*** We need to take our ***vk*** - visible nodes, and inside the for-loop, we keep these -1 values same as follows:

```
for k in range(10):
            _,hk = rbm.sample_h(vk)      # sampling hidden nodes
            _,vk = rbm.sample_v(hk)      # sampling visible nodes
            vk[v0 < 0] = v0[v0 < 0]
```

➢ That is, we use **v0**'s **-1**'s to make **vk**'s unrated movies to ***keep away from updating***.

➢ `vk[v0 < 0]` gonna get the nodes that have a ***-1*** rating in ***vk***, that were ***not originally rated*** by the ***users***.

➢ `v0[v0 < 0]` will replace those nodes by v0's original/unchanged values that were ***not originally rated*** by the ***users***. The ***original minus one ratings*** from the target ***v0***, because it is ***not updated***.

➢ We used `v0 < 0`, to get the ***minus one ratings*** because our ratings are either ***-1***, ***0***, or ***1***.

☝ By doing this, we make sure that the ***training*** is ***not done*** on these ***ratings*** that were ***not actually existed*** (not-rated by users).

So now we can get out of this ***third* `for` loop** and soon enough we will start the training.

14.5.4 Training the RBM-model - part 3 : `train()`

❒ **phk:** We want to apply the ***train()*** function to **update** the ***weight*** and the ***bias***. But, you notice that in this ***train()*** function, we need

```
train(self, v0, vk, ph0, phk):
```

i. target, ***v0***,
ii. our ***sampled visible nodes*** at the ***last step*** of the random walk, ***vk***,
iii. The **initial probabilities**, ***ph0***
iv. And ***k-step probabilities, phk***.

☞ Notice, we don't have any ***phk***. We didn't calculated it because ***vk*** wasn't updated at start.

☞ So before applying the ***train()*** function we compute ***phk***.

☝ We want to get the *first element* returned by the ***sample_h()*** function.

☞ We apply ***sample_h()*** on ***vk***, the last sample of the visible nodes (which we got from the **end** of **3rd for-loop**), the last sample visible nodes, at the 10th step of the ***random walk***, ***Gibbs sampling***.

```
        phk,_ = rbm.sample_h(vk)    # probabilities after k-step
```

Now we got everything we need to apply the ***train()*** function.

❒ ***train():*** Now let's apply the ***train()*** function. It's going to be kid stuff.

☞ Note that, the ***train()*** function doesn't return anything. It just updates the weights according to the steps 8, 9 and 10 of ***k-step-Contrastive-divergence algorithm***.

☞ So we don't need any new variable because it doesn't return anything.

☞ We just take our ***rbm*** object of the ***RBM*** class and apply ***train()***, using ***v0***, ***vk***, ***ph0*** and ***phk*** values for our four arguments.

```
# creating RBM model
nv = len(train_set_tensor[0])
nh = 100
batch_size = 100

rbm = RBM(nv, nh)


# Training the RBM
nb_epoch = 10
for epoch in range(1, nb_epoch + 1):
    train_loss = 0
    s = 0.0
    for u_id in range(0, nb_users - batch_size, batch_size):
        vk = train_set_tensor[u_id:(u_id + batch_size)]     # input vector
        v0 = train_set_tensor[u_id:(u_id + batch_size)]     # target vector

        ph0,_ = rbm.sample_h(v0)    # initial probabilities

            # applying k-step contrastive divergence
        for k in range(10):
            _,hk = rbm.sample_h(vk)     # sampling hidden nodes
            _,vk = rbm.sample_v(hk)     # sampling visible nodes
            vk[v0 < 0] = v0[v0 < 0]     # preventing updates to unrated nodes

        phk,_ = rbm.sample_h(vk)    # probabilities after k-step
        rbm.train(v0, vk, ph0, phk) # train RBM model
```

☝ Now the ***training*** is going to happen. The ***weight*** and the ***bias*** are going to be ***updated towards*** the direction of ***maximum likelihood***.

☝ And therefore, all our ***probabilities*** $p(v|h)$, of ***visible node*** given the states of the ***hidden nodes*** will be more and ***more relevant***,

☝ It will get the ***largest weights*** for the ***probabilities*** that are the ***most significant*** and eventually that will ***lead us*** to some ***predicted ratings*** that are going to be ***close*** to the ***real ratings***.

14.5.5 Training the RBM-model - part 4 : measuring error- `train_loss`

We now calculate ***train_loss*** to measure how ***close*** the ***predicted ratings*** to the ***real ratings***.

❒ So we now update the ***train_loss***, which previously set to ***0***. Because right now we actually have our ***predictions***, after updating the ***weights*** using ***train()***.

☞ Here `torch.abs(vk[v0 >= 0] - v0[v0 >= 0])` finds the ***tensor of absolute values*** of the difference **vk – v0**.

➢ We only take the **non-negative-values** of both **vk** and **v0**, using condition **"v0 >= 0"**

➢ Finally we use `torch.mean()` to find the ***simple difference*** in the ***tensor of absolute values***.

☝ Here in RBM we use ***simple difference*** in ***absolute values***. that measures, simply, the ***absolute difference*** between the ***predicted rating*** and the ***real rating***.

But the most common loss measuring method is ***RMSE***, the ***Root Mean Square Error***, which we'll use in next chapter to build ***Auto-Encoders***.

Also we increment counter **s** and ***train_loss*** inside the 2nd ***for loop***.

Outside the 2nd ***for loop*** we calculate the **normalized** *train_loss*. And ***print*** this value and epoch no.

```
        # Error rate
        train_loss += torch.mean(torch.abs(vk[v0 >= 0] - v0[v0 >= 0]))
        s += 1.0

    loSS = train_loss/s
    print(f"Epoch no. {epoch}.\t loss = {loSS}")
```

At the last steps, the ***weights*** are going ***close*** to the ***optimal weights***. The ***optimal sample visible*** nodes after the ***10 steps*** of ***Gibbs sampling*** contains our best ***predicted ratings***.

We compared the *non-negative* values of ***vk*** (the last visible nodes after the last batch of users that went through the network; after the last step of contrastive divergence) and ***v0*** (the target vector), take their ***absolute values*** and calculated the **simple mean-distance** of those **absolute values**. The ***simple distance*** in ***absolute values*** between the ***predictedn*** and the ***real rating***.

Remember in the training we didn't include the ratings that were actually non-existent, the -1 ratings were ignored. Hence we considered the non-negative values here.

When we compute this difference between the ***real original ratings*** and the ***predictions***, well, we have to take them for the ones that ***actually exist***.

```
# Training the RBM
nb_epoch = 10
for epoch in range(1, nb_epoch + 1):
    train_loss = 0
    s = 0.0
    for u_id in range(0, nb_users - batch_size, batch_size):
        vk = train_set_tensor[u_id:(u_id + batch_size)]     # input vector
        v0 = train_set_tensor[u_id:(u_id + batch_size)]     # target vector

        ph0,_ = rbm.sample_h(v0)    # initial probabilities

            # applying k-step contrastive divergence
        for k in range(10):
            _,hk = rbm.sample_h(vk)     # sampling hidden nodes
            _,vk = rbm.sample_v(hk)     # sampling visible nodes
            vk[v0 < 0] = v0[v0 < 0]     # preventing updates to unrated nodes

        phk,_ = rbm.sample_h(vk)    # probabilities after k-step
        rbm.train(v0, vk, ph0, phk) # train RBM model

        # Error rate
        train_loss += torch.mean(torch.abs(vk[v0 >= 0] - v0[v0 >= 0]))
        s += 1.0

    loSS = train_loss/s
    print(f"Epoch no. {epoch}.\t loss = {loSS}")
```

One Fix before executing the model: For the latest update of **PyTorch** we found the following error:

```
RuntimeError: The expanded size of the tensor (1682) must match the existing size (100) at non-
singleton dimension 1

     19         return p_v_given_h, torch.bernoulli(p_v_given_h)
     20     def train(self, v0, vk, ph0, phk):
---> 21         self.W += torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk)
     22         self.b += torch.sum((v0 - vk), 0)
     23         self.a += torch.sum((ph0 - phk), 0)
```

Now, **print(rbm.W.size())** will show you **torch.Size([100, 1682])**

print((torch.mm(v0.t(), ph0)-torch.mm(vk.t(), phk)).size()) will show you **torch.Size([1682, 100])**

- So it looks like we should take transpose of `(torch.mm(v0.t(), ph0)-torch.mm(vk.t(), phk))`, so it should be something like `(torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk)).t()`: we simply take transpose using **`t()`**.
- So inside our ***RBM class*** we update this code:

```
class RBM():
    def __init__(self, nv, nh) :
 . . . .
 . . . .
 . . . .
    def train(self, v0, vk, ph0, phk):
        self.W += (torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk)).t()
```

```
# -------- Creating the architecture of the Neural Netwark --------
class RBM():
    def __init__(self, nv, nh) :
        self.W = torch.randn(nh, nv)
        self.a = torch.randn(1, nh)
        self.b = torch.randn(1, nv)

    def sample_h(self, x):
        wx = torch.mm(x, self.W.t())
        activation = wx + self.a.expand_as(wx)
        p_h_given_v = torch.sigmoid(activation)
        return p_h_given_v, torch.bernoulli(p_h_given_v)

    def sample_v(self, y):
        wy = torch.mm(y, self.W)
        activation = wy + self.b.expand_as(wy)
        p_v_given_h = torch.sigmoid(activation)
        return p_v_given_h, torch.bernoulli(p_v_given_h)

    def train(self, v0, vk, ph0, phk):
        self.W += (torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk)).t()
        self.b += torch.sum((v0 - vk), 0)
        self.a += torch.sum((ph0 - phk), 0)



# creating RBM model
nv = len(train_set_tensor[0])
nh = 100
batch_size = 100

rbm = RBM(nv, nh)


# Training the RBM
nb_epoch = 10
for epoch in range(1, nb_epoch + 1):
    train_loss = 0
    s = 0.0
    for u_id in range(0, nb_users - batch_size, batch_size):
        vk = train_set_tensor[u_id:(u_id + batch_size)]     # input vector
        v0 = train_set_tensor[u_id:(u_id + batch_size)]     # target vector

        ph0,_ = rbm.sample_h(v0)    # initial probabilities

            # applying k-step contrastive divergence
        for k in range(10):
            _,hk = rbm.sample_h(vk)     # sampling hidden nodes
            _,vk = rbm.sample_v(hk)     # sampling visible nodes
            vk[v0 < 0] = v0[v0 < 0]     # preventing updates to unrated nodes

        phk,_ = rbm.sample_h(vk)    # probabilities after k-step
        rbm.train(v0, vk, ph0, phk) # train RBM model

        # Error rate
        train_loss += torch.mean(torch.abs(vk[v0 >= 0] - v0[v0 >= 0]))
        s += 1.0

    loSS = train_loss/s
    print(f"Epoch no. {epoch}.\t loss = {loSS}")
```

Training the model: After executing the above codes we get following result:

```
Epoch no. 1.     loss = 0.3290286362171173
Epoch no. 2.     loss = 0.25205469131469727
Epoch no. 3.     loss = 0.25061896443367004
Epoch no. 4.     loss = 0.24644401669502258
Epoch no. 5.     loss = 0.2497491091489792
Epoch no. 6.     loss = 0.2500344514846802
Epoch no. 7.     loss = 0.24852003157138824
Epoch no. 8.     loss = 0.24873429536819458
Epoch no. 9.     loss = 0.24498504400253296
Epoch no. 10.    loss = 0.24745795130729675
```

- And we end up with a ***train loss*** of ***0.25*** which is pretty good because that means that in the ***training set***, well we get the correct predictive rating, ***three*** times out of ***four***.

Next to apply on test-set: Now, we need to ***evaluate*** our ***model*** on new observations and that is what the ***test_set*** is for.

- Next, we will make our **predictions** on the test set ***without*** doing any ***training***, of course.
- And also we will compute a ***test_loss***, which will be the same *mean of the absolute distance between* the *predictions* and *the ratings*.
- We're hoping to get a ***test_loss*** that is around this ***0.25*** value. If it is around it, that means that even on *new observations* well we predict *correctly*, *three ratings* out of *four*. So that would be amazing (no-overfitting).

14.5.6 RBM-model : Evaluating on Test-set

Now we test our RBM on ***test_set***. We will see if the results are *close* to the *training set results*. That is if even on *new observations*, we can predict *three correct* ratings out of *four*.

- These are binary ratings, and we would definitely succeed at making a recommended system.

Getting the ***test_set*** results is going to be quite similar as getting the ***train_set*** results. The only difference is that there is ***not*** gonna be a ***training***.

- So we will *remove* at least one *for loop*. And we also use *MCMC techniques*. **MCMC-Markov chain Monte Carlo techniques** is the essential, the *crucial point to understand* here.

Now we copy above training-code and we will make the required change.

- There is no training, so no epochs and first for-loop. We re-align everything.
- We rename ***train_loss*** to ***test_loss***,
- Then we will rename the counter to ***cnt*** we initialize at *zero*, and we will increment it by one at each step.
- Now we loop over all the users in ***test_set***.
- We do not need a ***batch_size***. Because the batch size is just a technique specific to the training. During training, *batch size* is a parameter that you *tune* to get *more* or *less* better *performance* on *results* on the *training set*, and therefore, on the test set.
  - So we remove the ***batch_size*** step of the for-loop and we only take our users, up to the last user.
  - Our *model* will make some *predictions* for *each* of the *users* one by one.
  - And therefore, we can also remove the **0** from **range()** of the for-loop

Inside the for-loop:

- We rename ***vk*** to ***v*** and ***v0*** to ***vt***. So ***v*** will be out *input vector* and ***vt*** as *target vector*. ***v*** is the ***input*** on which we will make the prediction.
- Also we change `[u_id:(u_id + batch_size)]` to `[u_id:(u_id + 1)]`, notice we replace `batch_size` with **1**. Since we are gonna make some predictions for each of the users one by one, we will replace this ***batch_size*** by ***1***.

☞ ***Keep the train-set for input vector & test-set for target vector:*** Note that (important), we use the ***test_set*** for the ***target vector*** but for the ***input vector***, we use our ***train_set***.

```
# Evaluating the RBM on Test-set
test_loss = 0
cnt = 0.0
for u_id in range(nb_users):
    v = train_set_tensor[u_id:(u_id + 1)]     # input vector
    vt = test_set_tensor[u_id:(u_id + 1)]     # target vector
```

✋ Since we are dealing with the ***test set*** so we are trying to ***predict*** the ***ratings*** in the ***test set***. So it would make sense to use ***test_set*** for both ***v*** & ***vt***.

☠ But that would be ***wrong***.

☝ ***Target vt:*** Indeed, we need ***test_set*** for ***vt*** because we want to compare the ***real ratings*** of the ***test set*** to our ***predictions***. Because ***vt*** contains the ***original ratings*** of the ***test set***, so that is what we will ***compare*** to our ***predictions*** in the end.

☝ ***Input v:*** But here for ***v***, the input, we actually need to keep the ***training set***. It's because, the ***training set*** is the ***input*** that will be ***used*** to ***activate*** the ***hidden neurons*** to get the ***output***.

⇨ *We need to understand this first crucial point:* We are using the ***inputs*** of the ***training set*** to ***activate*** the ***neurons*** of the ***RBM*** to get the ***predicted ratings*** of the ***test set***.

⇨ Right now the ***training set*** contains the ***ratings*** of the ***training set*** and it *doesn't contain the answers* of the ***test set***.

⇨ But, by using the ***inputs*** of the ***training set*** we will ***activate*** the ***neurons*** of our ***RBM*** to ***predict*** the ***ratings*** of the movies that were ***not rated yet***, and those ***will be the ratings*** of the ***test set.***

➢ So we need this **training-set**, ***train_set_tensor*** as ***input*** to get the ***predicted ratings*** of the ***test set***. Because ***we*** are ***getting*** these ***predicted ratings*** from the ***inputs*** of the ***training set*** that are used to *activate the neurons of our RBM*.

☞ `ph0,_ = rbm.sample_h(v0)` computes ***probabilities*** that the **hidden node** equal **1 given** the **real ratings**

$$p\left(H_i = 1 \mid v^{(0)}\right) = \textbf{ph0}$$

⇨ ***ph0*** was needed to ***train*** the ***model***, therefore, we don't need it for the ***test set***.

☞ ***MCMC & Blind walk:*** The *MCMC* techniques, *Markov chains Monte Carlo* techniques is related to the ***random walk*** and more precisely the ***blind walk***.

☝ To get our **predictions** of the ***test set ratings***, do we need to apply ***again*** the ***k step contrastive divergence***?

☝ More precisely, do we need to make ***k steps*** of the ***random walk*** (i.e. 10 step for-loop), that is 10 steps of the random walk?

⇨ Actually we need one step of the ***random walk (blind walk)***, hence we don't need the ***for-loop*** for ***k-step*** (i.e 10 step) to get our ***final prediction***. So we only need *only one step of contrastive divergence*.

⇨ Also this is ***not exactly*** the ***random walk*** because in the ***random walk*** the ***probabilities*** are the ***same***. Here, even if it's a ***Markov chain*** the ***probabilities*** are ***not the same*** so it's *not a random walk*, so it's rather a ***Blind Walk***.

☝ The ***Principle Of The Blind Walk*** is that, imagine you were ***blindfolded*** and you had to make ***100 steps*** on a ***straight line*** without getting ***out*** of the straight line.

☝ You will be ***trained*** with ***Gibbs sampling*** to make ***100 steps*** by ***staying*** on the ***straight line*** but you're ***blindfolded*** so you know it's not easy to make some steps and always stay on the ***straight line***, so, you may go a little bit on the left, on the right and after 100 steps it's hard for you to be on the straight line.

➢ But you were ***trained*** to make these ***100 steps***, staying on the ***straight line*** being ***blindfolded***. And that is by doing some ***random steps***.

➢ It is close to the random walk technique and the MCMC but the difference is that in the random walk the ***probabilities*** are the ***same*** and here the ***probabilities*** are ***different***. And that's the thing you are ***trained*** to make ***100 steps*** by ***staying*** on a ***straight line***.

☝ And so the ***principle*** of all this is that you are ***trained*** to do this for ***100 steps*** so that when you ***make one step***.

➢ When you have to take the **challenge** to make ***only one step*** and ***still*** be on that **straight line**, well you will have ***High Chance of SUCCESS***.

⇨ Same thing goes here, our ***RBM-recommender-model*** is ***trained*** for ***10-step (k-step)*** and now, for ***test-set*** it have to take ***only-one step***.

- ➢ That's the whole principle of the blind walk technique from MCMC, Markov chain Monte Carlo. That's close to the random walk but keep in mind that the probabilities are not the same, it is blind-walk with different probabilities.

⇨ So our prediction will be ***directly*** the ***result*** of ***one round trip*** of ***Gibbs sampling***. One iteration, ***one step*** of the ***blind walk***. Then we'll get all our ***predictions*** of the ***test set*** in one shot.

⇨ In this step we're gonna start with an ***if*** condition to ignore/filter the ***unrated-movies*** from the ***test-set*** (the ***-1*** are ratings that just never happened). And it's the same id we are dealing with for the train-set & the test-set.

- ➢ We use ***len()*** function to get the ***real-ratings*** (***original ratings*** of the ***test set***) that are existent from the **target *vt***,
- ➢ We take all the ***ratings*** that are ***existent*** using the condition "`vt>=0`".
- ➢ `len(vt[vt>=0]) > 0` Specifies the number of the visible nodes containing the non-negative-ratings must be larger than zero, and in that condition we can make some predictions.

```
if len(vt[vt>=0]) > 0:
```

⇨ To make the predictions we use `sample_h()` and `sample_v()` functions. And in this step we use ***h*** and ***v*** instead of ***hk*** and ***vk***.

```
_,h = rbm.sample_h(v)      # sampling hidden nodes
_,v = rbm.sample_v(h)      # sampling visible nodes
```

- ➢ For the ***input visible*** **nodes** we'll get only ***one hidden node*** because there is ***one step only***. i.e. one ***vector of hidden nodes*** to get our ***final vector of predicted ratings***. And that's all for the only step of the ***blind walk***.
- ➢ We don't need " `vk[v0 < 0] = v0[v0 < 0]`" here. We also do not use ***phk***, or ***train()***, so we get rid of those lines.

⇨ NOTE that, now we have to update the ***test_loss*** and the **counter *cnt*** inside this "***if-block***". These needs to be in the ***if-block*** because we are still computing the ***test_loss*** only for the ***existent ratings***.

- ➢ We are still taking the ***absolute distance*** between the ***prediction*** and the ***target***, so we used ***torch.mean()*** and ***torch.abs()***.

```
test_loss += torch.mean(torch.abs(vt[vt >= 0] - v[vt >= 0]))
```

- ➢ So, the ***target*** is ***vt*** and the ***prediction*** is ***v***. We take the existent ratings using the condition "`vt >= 0`" (to get the indexes of the cells that have the existent ratings).

```
    if len(vt[vt>=0]) > 0:
        _,h = rbm.sample_h(v)      # sampling hidden nodes
        _,v = rbm.sample_v(h)      # sampling visible nodes

        # Error rate
        test_loss += torch.mean(torch.abs(vt[vt >= 0] - v[vt >= 0]))
        cnt += 1.0
```

☞ At the end we calculate the normalized ***test_loss*** using ***test_loss/cnt***. And print the result.

```
eval_loSS = test_loss/cnt
print(f"Evaluation or Test loss = {eval_loSS}")
```

```
# Evaluating the RBM on Test-set
test_loss = 0
cnt = 0.0
for u_id in range(nb_users):
    v = train_set_tensor[u_id:(u_id + 1)]     # input vector from training-set
    vt = test_set_tensor[u_id:(u_id + 1)]     # target vector from test-set

    if len(vt[vt>=0]) > 0:
        _,h = rbm.sample_h(v)      # sampling hidden nodes
        _,v = rbm.sample_v(h)      # sampling visible nodes

        # Error rate
        test_loss += torch.mean(torch.abs(vt[vt >= 0] - v[vt >= 0]))
        cnt += 1.0

eval_loSS = test_loss/cnt
print(f"Evaluation or Test loss = {eval_loSS}")
```

In summery:

[1]. We start with a ***test loss*** of ***zero*** then the ***counter*** to ***zero***.

[2]. We loop over all our users.

[3]. Then for all the ratings that are ***existent*** in the ***test set***, we ***sample*** the ***hidden nodes*** first

[4]. then we use these ***hidden nodes*** as input to ***sample the visible nodes***.

[5]. We update ***test_loss*** and counter ***cnt*** inside ***if-block***.

[6]. We did this for only one round trip according to the MCMC theory.

```
In [2]:
   ...: test_loss = 0
   ...: cnt = 0.0
   ...: for u_id in range(nb_users):
   ...:     v = train_set_tensor[u_id:(u_id + 1)]     # input vector from training-set
   ...:     vt = test_set_tensor[u_id:(u_id + 1)]     # target vector from test-set
   ...:
   ...:     if len(vt[vt>=0]) > 0:
   ...:         _,h = rbm.sample_h(v)     # sampling hidden nodes
   ...:         _,v = rbm.sample_v(h)     # sampling visible nodes
   ...:
   ...:         # Error rate
   ...:         test_loss += torch.mean(torch.abs(vt[vt >= 0] - v[vt >= 0]))
   ...:         cnt += 1.0
   ...:
   ...: eval_loSS = test_loss/cnt
   ...: print(f"Evaluation or Test loss = {eval_loSS}")
Evaluation or Test loss = 0.24375128746032715
```

After executing above code, we get a ***test loss*** of ***0.24***. Which is definitely excellent, because for ***new observations*** for ***new movies*** we managed to ***predict*** some ***correct ratings*** three times out of four and even better than that. Because we are slightly below ***25%***.

Here we definitely managed to make a ***robust recommended system***, but, remember this was the easy one. Predicting binary ratings ***0*** and ***1***.

In the next chapter we'll take it to the next level because we will be predicting some ratings from ***1*** to ***5***, using the AutoEncoders.

- Obviously, working with some continuous values will increase the complexity of the problem. But it will help us to understand to build ***two different recommended systems*** by applying ***two different deep learning models***.
- But, the good news is ***Auto Encoders – AE***, is actually a much more simple model than ***Boltzmann machine***.
- With AE we will get some amazing predictions even for ratings between ***1*** to ***5***.

All Code at once (practiced) : RBM-Recommender model

```
# ---------- RBM : Recommender ---------------

# Importing the libraries
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable

# importing the dataset
movies = pd.read_csv("./movie_lens_1m/movies.dat", sep= "::", header=None, engine="python", encoding="latin-1")
useRs = pd.read_csv("./movie_lens_1m/users.dat", sep= "::", header=None, engine="python", encoding="latin-1")
RaTings = pd.read_csv("./movie_lens_1m/ratings.dat", sep= "::", header=None, engine="python", encoding="latin-1")
```

```
# preparing the training set and test set
training_set = pd.read_csv("./movie_lens_100k/u1.base", delimiter="\t")
train_set = np.array(training_set, dtype="int")

ts_set = pd.read_csv("./movie_lens_100k/u1.test", delimiter="\t")
test_set = np.array(ts_set, dtype="int")


# Getting the number of Users and Movies
nb_users = int(max(max(train_set[:, 0]), max(test_set[:, 0])))
nb_movies = int(max(max(train_set[:, 1]), max(test_set[:, 1])))


# converting the data into an array with users in lines and movies in column.
def conVert(data):
    new_data = []
    for id_user in range(1, nb_users + 1):
        # use "data[:, 0] == id_user" as condition over movie column "data[:, 1]"
        id_movies = data[:, 1][data[:, 0]== id_user]    # returns a list

        # use "data[:, 0] == id_user" as condition over ratins column "data[:, 2]"
        id_ratings = data[:, 2][data[:, 0]== id_user]

        # vector of zeros
        ratings = np.zeros(nb_movies)
        ratings[id_movies - 1] = id_ratings

        new_data.append(list(ratings))

    return new_data

trn_set_cnvt = conVert(train_set)
tst_set_cnvt = conVert(test_set)


# Converting the data into Torch Tensosrs. Following are the Tensors of ratings
train_set_tensor = torch.FloatTensor(trn_set_cnvt)
test_set_tensor = torch.FloatTensor(tst_set_cnvt)



# Converting the ratings into binary ratings: 1 (liked),  0 (not-liked)
train_set_tensor[train_set_tensor == 0] = -1
        # torch doesn't support combined condition
train_set_tensor[train_set_tensor == 1] = 0
train_set_tensor[train_set_tensor == 2] = 0
train_set_tensor[train_set_tensor >= 3] = 1

test_set_tensor[test_set_tensor == 0] = -1
test_set_tensor[test_set_tensor == 1] = 0
test_set_tensor[test_set_tensor == 2] = 0
test_set_tensor[test_set_tensor >= 3] = 1

# train_tensor_to_view = train_set_tensor.detach().cpu().numpy()
# test_tensor_to_view = test_set_tensor.detach().cpu().numpy()
train_tensor_to_view = train_set_tensor.numpy()
test_tensor_to_view = test_set_tensor.numpy()


# -------- Creating the architecture of the Neural Netwark --------
class RBM():
    def __init__(self, nv, nh) :
        self.W = torch.randn(nh, nv)
        self.a = torch.randn(1, nh)
        self.b = torch.randn(1, nv)

    def sample_h(self, x):
        wx = torch.mm(x, self.W.t())
        activation = wx + self.a.expand_as(wx)
        p_h_given_v = torch.sigmoid(activation)
        return p_h_given_v, torch.bernoulli(p_h_given_v)

    def sample_v(self, y):
        wy = torch.mm(y, self.W)
        activation = wy + self.b.expand_as(wy)
        p_v_given_h = torch.sigmoid(activation)
        return p_v_given_h, torch.bernoulli(p_v_given_h)

    def train(self, v0, vk, ph0, phk):
        self.W += (torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk)).t()
        self.b += torch.sum((v0 - vk), 0)
        self.a += torch.sum((ph0 - phk), 0)
```

```
# creating RBM model
nv = len(train_set_tensor[0])
nh = 100
batch_size = 100

rbm = RBM(nv, nh)


# Training the RBM
nb_epoch = 10
for epoch in range(1, nb_epoch + 1):
    train_loss = 0
    s = 0.0
    for u_id in range(0, nb_users - batch_size, batch_size):
        vk = train_set_tensor[u_id:(u_id + batch_size)]     # input vector
        v0 = train_set_tensor[u_id:(u_id + batch_size)]     # target vector

        ph0,_ = rbm.sample_h(v0)    # initial probabilities

            # applying k-step contrastive divergence
        for k in range(10):
            _,hk = rbm.sample_h(vk)     # sampling hidden nodes
            _,vk = rbm.sample_v(hk)     # sampling visible nodes
            vk[v0 < 0] = v0[v0 < 0]     # preventing updates to unrated nodes

        phk,_ = rbm.sample_h(vk)    # probabilities after k-step
        rbm.train(v0, vk, ph0, phk) # train RBM model

        # Error rate
        train_loss += torch.mean(torch.abs(vk[v0 >= 0] - v0[v0 >= 0]))
        s += 1.0

    loSS = train_loss/s
    print(f"Epoch no. {epoch}.\t loss = {loSS}")



# Evaluating the RBM on Test-set
test_set_ratings_list = []
pridicted_rating_list = []

test_loss = 0
cnt = 0.0
for u_id in range(nb_users):
    v = train_set_tensor[u_id:(u_id + 1)]     # input vector from training-set
    vt = test_set_tensor[u_id:(u_id + 1)]     # target vector from test-set

    if len(vt[vt>=0]) > 0:
        _,h = rbm.sample_h(v)     # sampling hidden nodes
        _,v = rbm.sample_v(h)     # sampling visible nodes

        # Error rate
        test_loss += torch.mean(torch.abs(vt[vt >= 0] - v[vt >= 0]))
        cnt += 1.0

    # creating list of original & predicted ratings
    original_test_set_ratings = vt.numpy()
    test_set_ratings_list.append(original_test_set_ratings)
    predicted_ratings = v.numpy()
    pridicted_rating_list.append(predicted_ratings)

eval_loSS = test_loss/cnt
print(f"Evaluation or Test loss = {eval_loSS}")

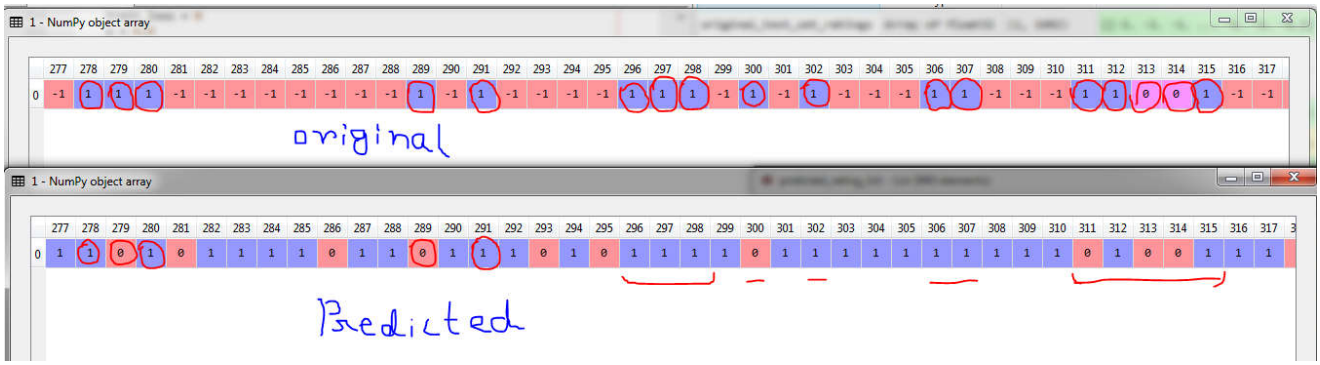

# python prctc_RBM.py
```

Result

```
In [1]: runfile('D:/1_Development_2.0/ML_phase_3_ML_Intro/ml_p3_ch14_dp_5_BLTZ/
prctc_RBM.py', wdir='D:/1_Development_2.0/ML_phase_3_ML_Intro/ml_p3_ch14_dp_5_BLTZ')
Epoch no. 1.      loss = 0.37458986043930054
Epoch no. 2.      loss = 0.24945561587810516
Epoch no. 3.      loss = 0.25079256296157837
Epoch no. 4.      loss = 0.24438756704330444
Epoch no. 5.      loss = 0.25012412667274475
Epoch no. 6.      loss = 0.24456749856472015
Epoch no. 7.      loss = 0.24766132235527039
Epoch no. 8.      loss = 0.25108692049980164
Epoch no. 9.      loss = 0.24655379354953766
Epoch no. 10.     loss = 0.2484351545572281
Evaluation or Test loss = 0.243212029337883
```

We can see that in ***test_set*** prediction we have ***Test-Loss 0.2432***, i.e. more than ***75% correct predictions***.

<u>Comparison between predicted and real results</u>



From this comparison we can see that in the ***predicted*** result from ***RBM***, we have some ***ratings*** that are ***not originally rated***. Also we get ***75% correct predictions*** for the ***movies*** that are ***originally rated*** (marked red).

Chapter 15 : Part 1

Deep Learning

Auto Encoders (AE)

`Introduction`

15.1.1 What we will learn in this chapter

[1]. ***Auto Encoders:*** What are ***`Auto Encoders`***, what is the architecture and how do they work.
[2]. ***Training of an Auto Encoder:*** We'll talk about **Training** of an **Auto Encoder** and the steps that go into that.
[3]. ***Overcomplete Hidden Layers:*** We'll talk about a situation where we have ***`Overcomplete hidden layers`*** in an ***`auto encoder`***.

We'll And then we'll talk about ***three regularization*** techniques:

[4]. ***Sparse Autoencoders***
[5]. ***Denoising Autoencoders***
[6]. ***Contractive Autoencoders***

[7]. ***Stacked Autoencoders***
[8]. ***Deep Autoencoders***

NOTE: Starting from over ***Complete Hidden Layers*** to ***Deep Auto Encoders***, are more of a high level overview. We'll go through a quick introduction to each one of these variations, at least we will know what is the reference to when we hear them.

15.1.2 Auto Encoder (AE)

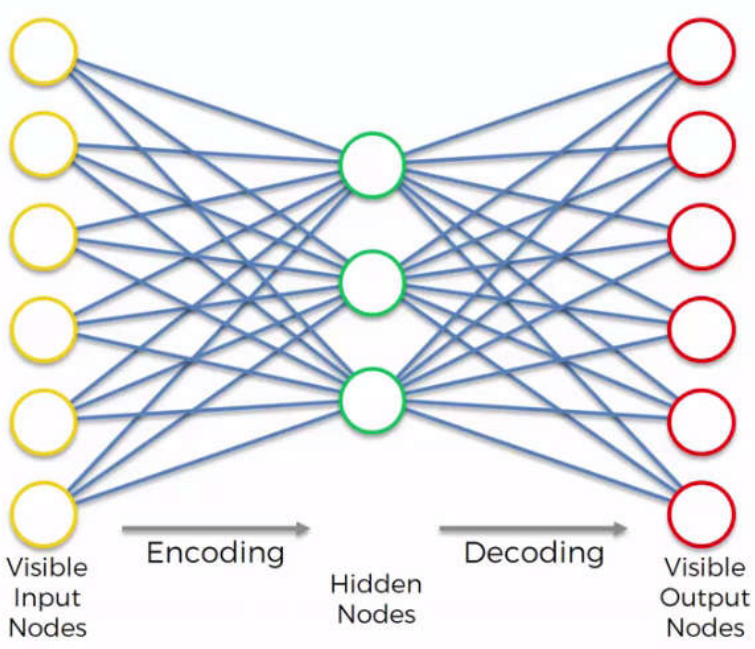
Right-side diagram is what an ***Auto Encoder*** looks like. This is a ***directed*** types of ***neural networks***.

☞ The ***blue lines*** don't have ***arrows*** on the ***ends***, but we'll consider it as a ***directed type*** of ***network*** and everything's ***moving*** from ***left*** to ***right***.

❒ ***How Auto Encoder work:*** An **Auto Encoder** ***encodes itself***, that's the whole philosophy behind the Auto Encoder.

☞ It takes some sort of ***inputs***, put some through a ***hidden layer***, and then it gets ***outputs***. The point is: it aims for the ***outputs*** to be ***identical*** to the ***inputs***.

❒ ***Training an Auto Encoders:*** We set Auto Encoders in such a way that on the output you get values which are equivalent to your inputs.



❒ ***Auto Encoders are self-supervised:*** We can say that, ***Auto Encoders*** are not a ***pure*** type of ***Unsupervised Deep Learning algorithm***. They are actually a ***Self-Supervised Deep Learning Algorithm***, because they are comparing to something on the end.

💣 In ***BMs*** we *didn't even have outputs*. We didn't have to *compare* to any kind of *labels or anything*.
💣 In ***SOMs***, we *didn't have anything to compare to,* we're just looking for *features*.

☞ However, in Auto Encoders we are looking for ***hidden layer*** (also called the ***coding layer***, or the ***bottleneck***). We're looking for how to structure the ***hidden layer***, but at the same time, we are ***comparing*** the ***outputs*** to the ***inputs***.

☞ ***In short ward:*** You have ***inputs***, they get ***encoded***, and then they get ***decoded*** and then ***compared*** to ***inputs***, that's how the training happens.

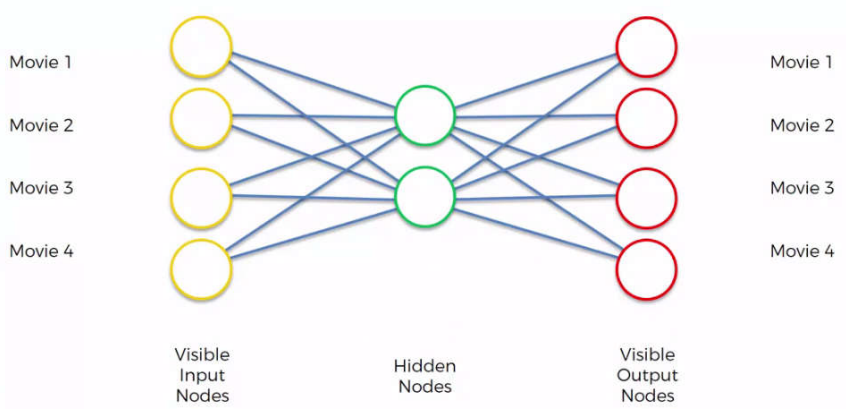
You can already imagine how information is ***propagated forward***, and then you have ***gradient descent*** then it ***propagate backward***. We'll talk about all those things next.

Usage of Auto Encoders: Well, there's a couple of things that they can be used for.

- ***They can be used for feature detection:*** Once you've ***encoded*** your data, these ***hidden-nodes*** will represent ***certain features*** which are ***important*** in your ***data***, and then you can just look at them and get those ***features*** out of them, or use those ***features*** in the ***future***.
- They can also be used to build ***powerful recommender systems***.
- ***They can be used for encoding:*** Actually ***Auto Encoders*** are designed to ***encode*** data. You feed it with lots and lots of values, ***encoded*** into a ***smaller representation*** and then all you'll have to carry around the ***decoder part*** and ***encoded data*** it would take up ***less*** space.

Example (walk through): Now let's have a look at an example of how they actually work, so we can understand them better on an ***intuitive level***.

- Here is our simplified Auto Encoder with four input nodes, and two nodes in the hidden layer.
- As we can see, we've got four movies at left and four movies at right.
- These are the movies that a person has watched, and we're going to encode the rating for those movies. **1** = liked that movie, and **0** = didn't like that movie.



- This example actually comes from this blog, **"PROBABLY DANCE"**. It's a great blog, it's by a person who's actually a ***programmer*** who isn't a ***Deep Learning scientist***, but he really broke it down into good steps.
- Now let's have a look at how this information can be encoded through the ***Auto Encoder***.


PROBABLY DANCE

I am program and like games

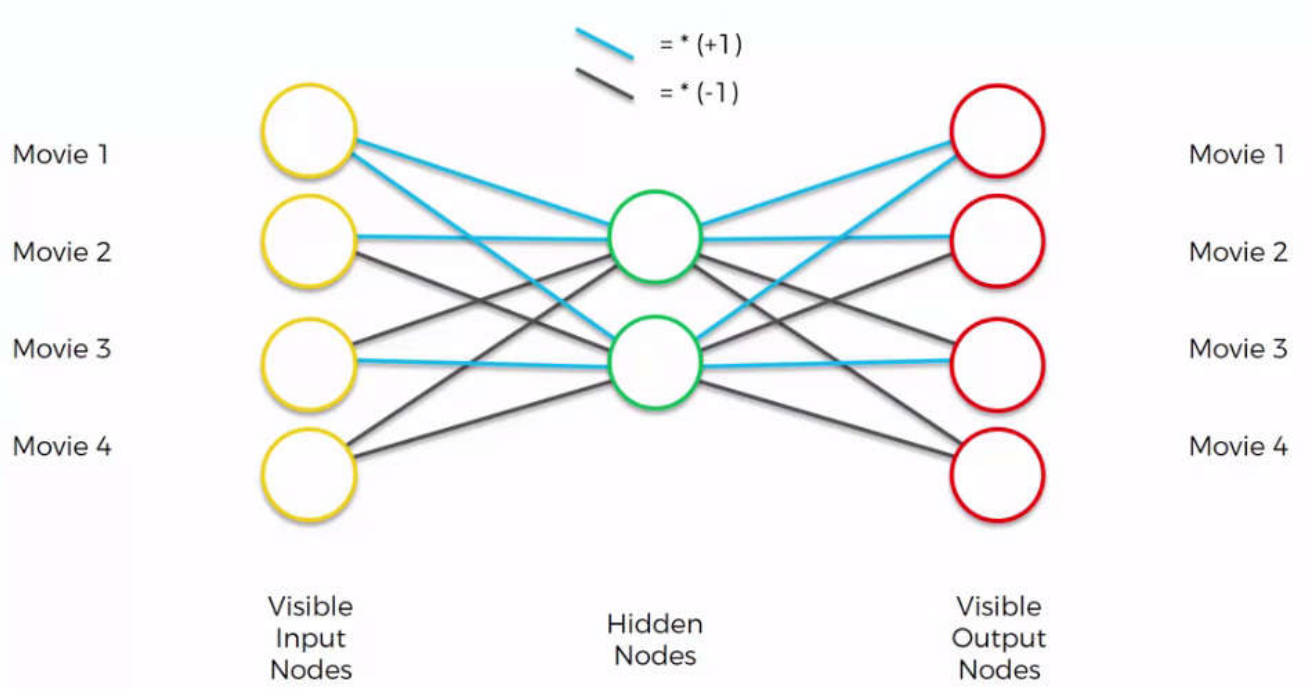
Neural Networks Are Impressively Good At Compression

by Mallw Shavuplw

I'm trying to get into neural networks. There have been a couple big breakthroughs in the field in recent years and suddenly my side project of messing around with programming languages seemed short sighted. It almost seems like we'll have real AI soon and I want to be working on that. While making my first couple steps into the field it's hard to keep that enthusiasm. A lot of the field is still kinda handwavy where when you want to find out why something is used the way it's used, the only answer you can get is "because it works like this and it doesn't work if we change it."

- Firstly, we establish ***connections*** with certain ***weights***. To prove that it is possible to take ***four values*** and ***encode*** them into actually ***two values***, and carry your data around, ***save space*** and ***extract*** those ***features***.
- For now we're ***not gonna worry*** about the ***training***.

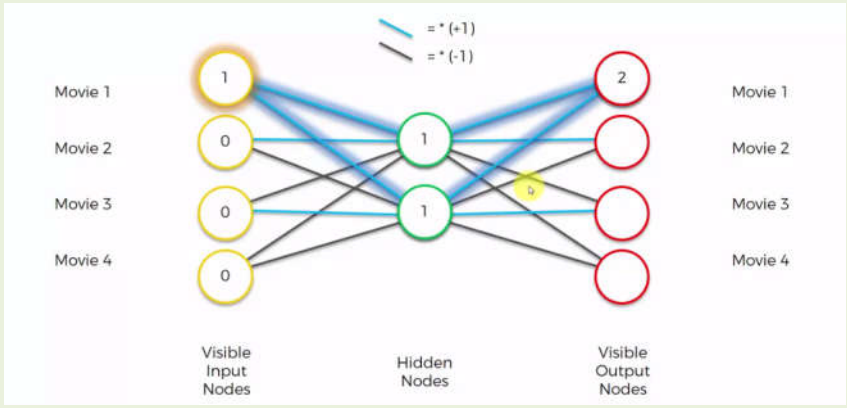
- We're going to color our synapses in two different colors, ***blue*** and ***black***.
  - ***Blue*** is basically a multiplication by "**+1**". Weight is plus one.
  - ***Black*** is a multiplication by "**-1**". Weight is minus one.
  - Also in **Auto Encoders**, we normally use the **Hyperbolic Tangent Activation** function (ranges between **[1, -1]**).

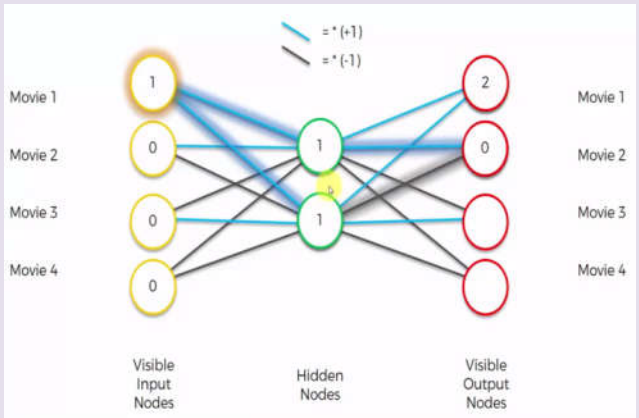


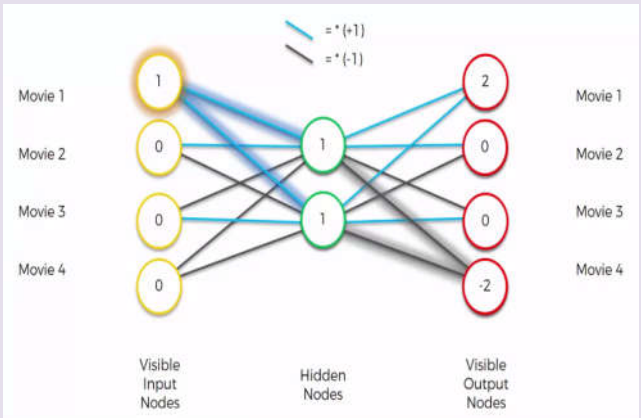
☐ Let's have a look at an input. Let's say as an input, we've got **1** for first input, and **0** for others. Means that the person just like ***Movie 1***, and dislikes the rest of the movies.

☐ In that case, hidden nodes will be, "**+1**" and "**+1**" because **blue synapse** is **multiplying** by "**+1**". And the **0**'s in the ***input nodes***, they will always just ***add zero***, and ***not*** going to ***contribute*** to the ***hidden nodes***.

For the **first output-node**:
1*(blue-synapse) + 1*(blue-synapse) =
`(1*1) + (1*1) = 2`
For the **2nd and 3rd output-node**:
1*(blue-synapse) + 1*(black-synapse) =
`(1*1) + (1*-1) = 0`
For the **4th output-node**:
1*(black-synapse)+ 1*(black-synapse) =
`(1*-1) + (1*-1) = -2`

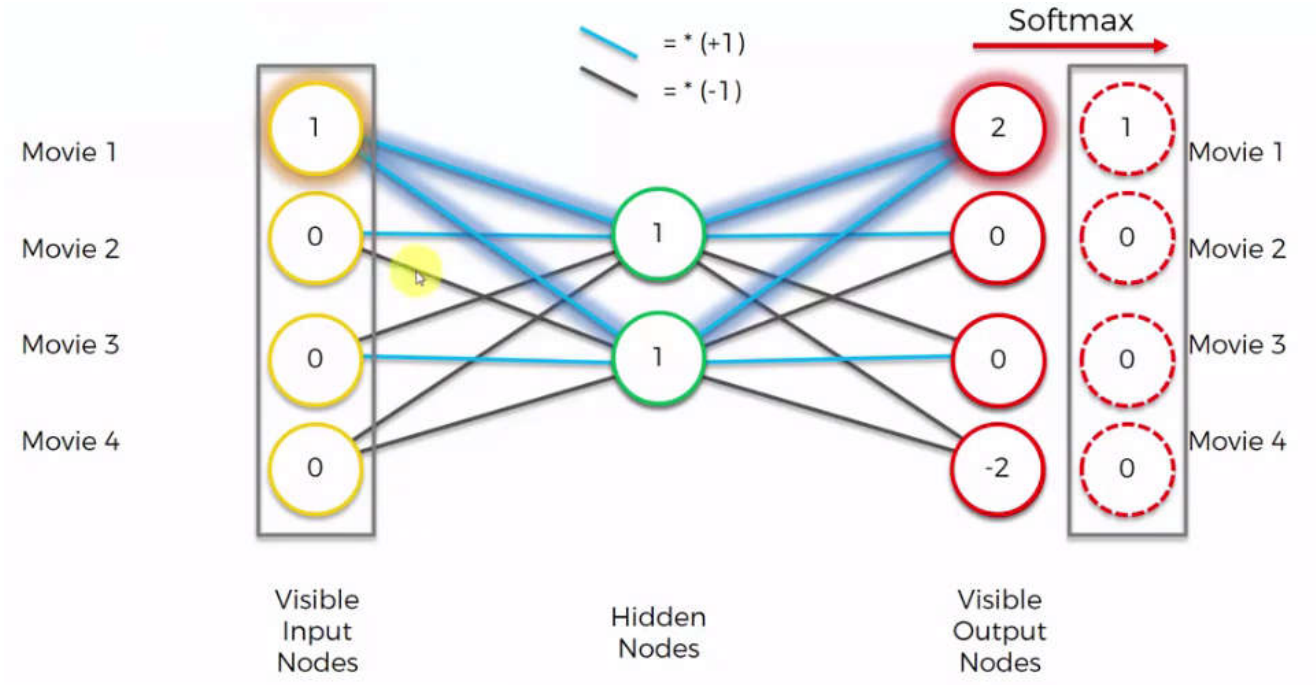




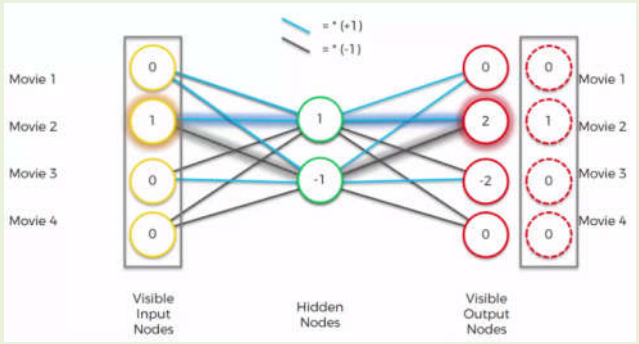


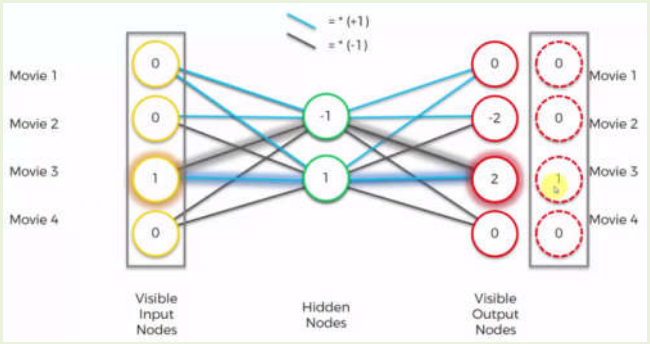
☯ So those are your ***outputs*** but those are actually ***preliminary outputs***. In Auto Encoder we also have a ***SOFTMAX function*** on the ***end***. As you can see, ***final output result*** is indeed identical to our ***input***.

☯ ***Softmax function*** takes the highest value, so in this case it is "**2**" and it turns that into "**1**" and everything else into "**0**".



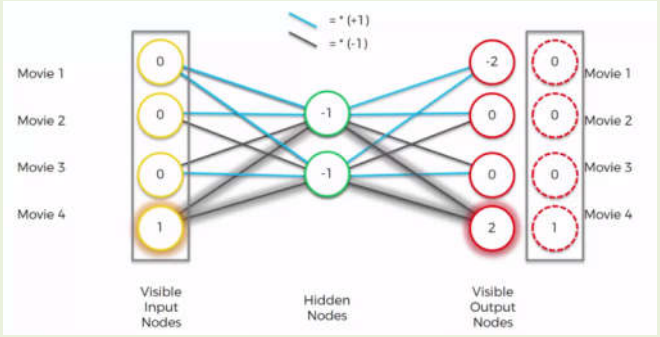
☐ Let's have a look at some other cases. We don't go to the details.





☝ Notice how the values inside hidden nodes changes from "**1**" to "**-1**".

☝ You can see, in our data set, we've got rows with three **0**'s and an **1** using ***4 nodes***. Then we can encode them into a small format where we just have ***2 hidden nodes***.



☐ Every state, is represented by a hidden layer using these hidden nodes `{1, 1}` or `{1, -1}` or `{-1, 1}` or `{-1, -1}`.

➲ Then you just need these ***weights*** and ***Softmax function*** to **reconstruct** your ***output***.

✋ It is a very ***simplified*** example, but this gives a overview of how ***Auto Encoders*** work. Of course ***Auto Encoders*** are much more complex than that.

❒ ***Additional reading:*** For additional reading this is the same blog that we already mentioned it's called, ***"Neural Networks Are Impressively Are Good At Compression"*** by ***Malte Skarupke*** and will include it in the additional resources.

☞ Nicely written very ***easy introduction*** into ***Auto Encoders***. Here you can see a much more ***sophisticated example***.

☞ It's not even actually mentioned that it's Auto Encoders from the very start, just ***Neural Networks***, and then in the comments you can read that indeed they were talking about **Auto Encoders**.

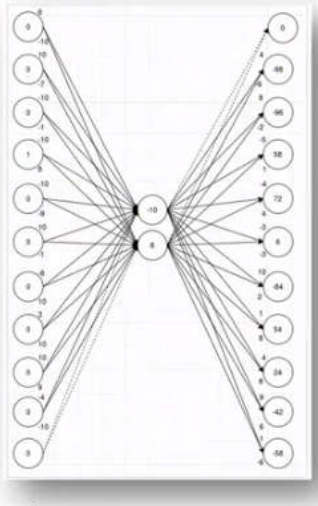
Additional Reading:

Neural Networks Are Impressively Good At Compression

By Malte Skarupke (2016)

Link:

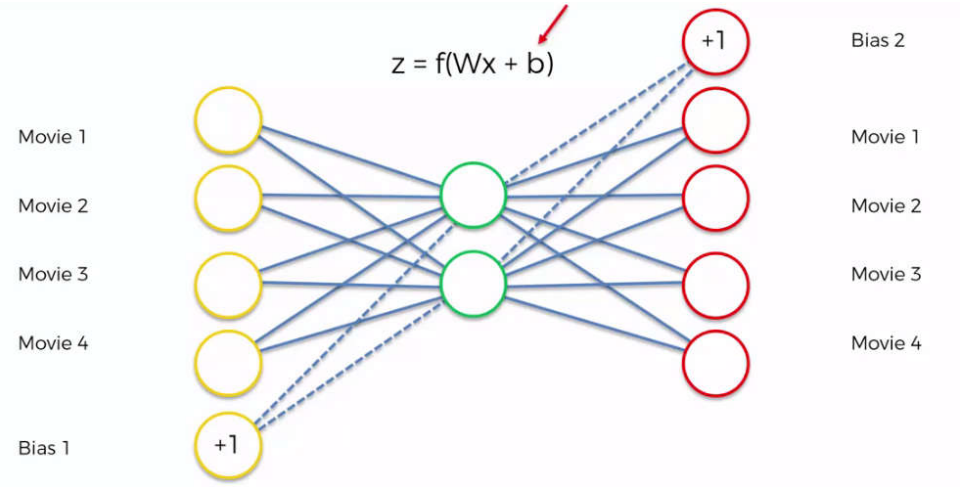
https://probablydance.com/2016/04/30/neural-networks-are-impressively-good-at-compression/



15.1.3 Biases in Auto Encoder (AE)

We've got our ***simplified version*** of ***Auto Encoders*** and sometimes you'll see a bit of a ***different representation*** as follows. The "**+1**" on both side with the ***dashed connection*** are just ***biases***.

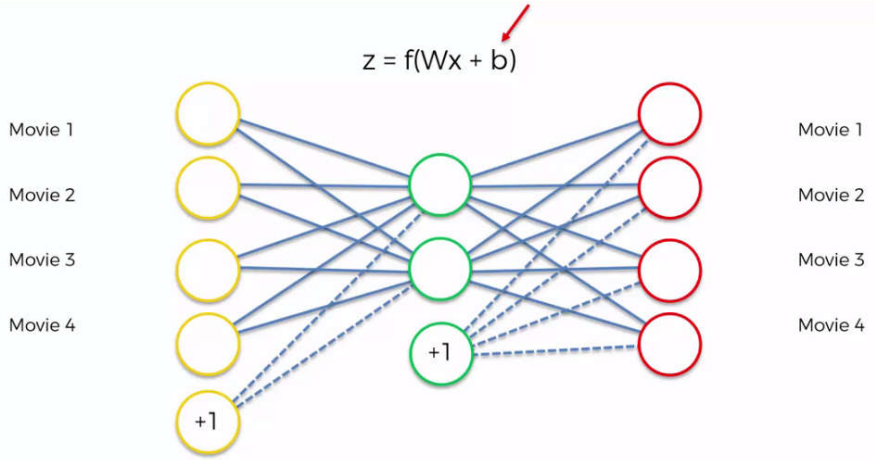
❒ ***Biases*** means that in your ***activation function*** you've got a constant **b** in $z = f(Wx + b), W =$ weights, $b =$ bias



❒ The more ***correct representation*** is given in right. This is how it should be represented if people are showing biases.

☞ ***Bias*** is just added there because ***bias affects*** the ***layer ahead***, the layer that's in front.

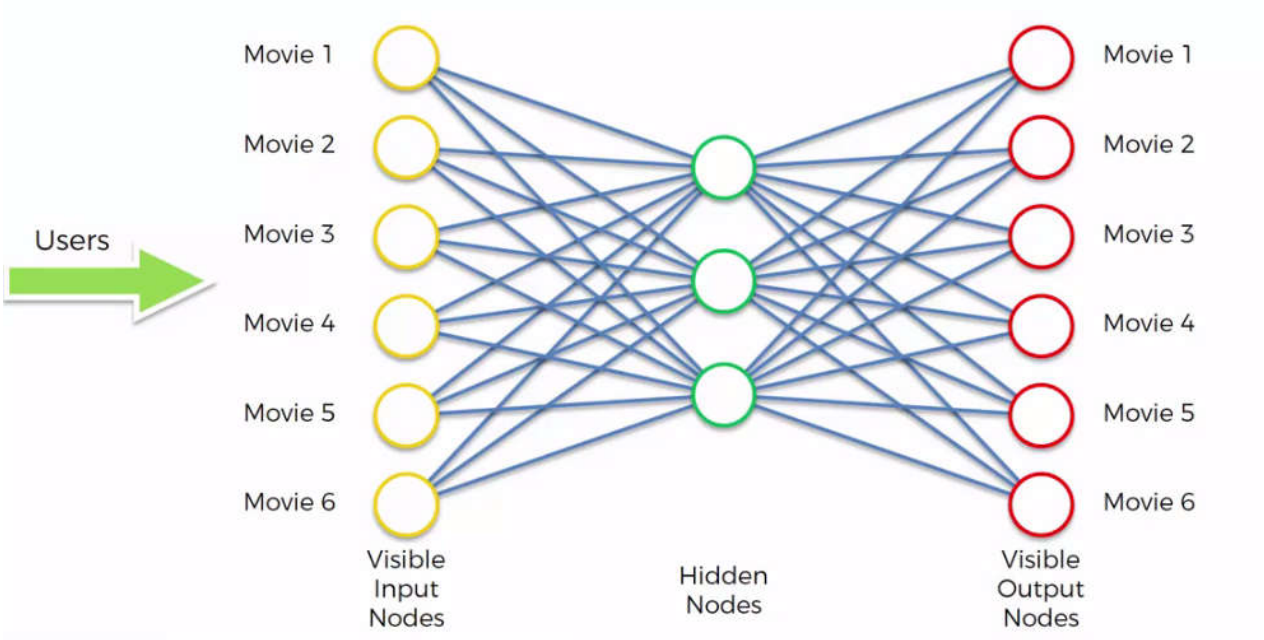
☞ If you see that just remember it's just basically a ***bias*** in it. It's just a ***constant*** in that ***equation***.



15.1.4 Training an AE

Here we walk through the steps to training an Auto Encoder. Here we've got an Auto Encoder, where we've got some inputs: the ***ratings*** of lots and ***lots of users*** for these ***six movies***, and then we'll get some outputs.

❒ Based on those **ratings**, ***Auto Encoder*** will come up with a way to ***compress*** that ***data*** and ***adjust*** the ***weights*** so that it set itself up to be able to ***encode*** *the data and* ***decode*** *the data in the future.*



Steps for Training an Auto Encoder

➢ ***STEP 1:*** We start with an ***array*** where the ***lines/rows*** (the observations) correspond to the ***users*** and the ***columns*** (the ***features***) correspond to the ***movies***. Each cell **(u, i)** contains the **rating** (from ***1*** to ***5***, ***0*** if **no rating**) of the ***movie i*** by the ***user u***.

☞ Here every single row in your data set is a unique user who rated those movies.

➢ ***STEP 2:*** The ***first user*** (first row) goes into the network. The ***input vector*** $x = (r_1, r_2, \dots, r_m)$ contains all its ***ratings*** for ***all*** the ***movies***. i.e all the ratings of that user for all of the movies.

➢ ***STEP 3:*** The ***input*** *vector* x is ***encoded*** into a *vector* z of ***lower dimensions*** by a ***mapping function*** f (e.g: **sigmoid function** or a **hyperbolic tangent**):

$$z = f(Wx + b)$$ where W is the ***vector*** of ***input weights*** and b the ***bias***

➢ ***STEP 4:*** z is then ***decoded*** into the ***output*** *vector* y of ***same dimensions*** as ***input-vector*** x, aiming to ***replicate*** the input ***vector*** x.

➢ ***STEP 5:*** The **reconstruction error** $d(x, y) = ||x - y||$ (which is how ***different*** is the ***output*** compared to the ***input***) is ***computed***. The goal is to **minimize** $d(x, y)$.

➢ ***STEP 6:*** **Back-Propagation:** from ***right*** to ***left***, the **error** is **back-propagated**. The ***weights*** are ***updated*** according to how much they are ***responsible*** for the ***error*** (so you've got a ***Gradient Decent process*** happening there) The ***learning rate*** decides by *how much* we ***update*** the ***weights*** (that's a parameter you can tune during decoding and you will see that in the practical example).

➢ ***STEP 7:*** Repeat ***Steps 1*** to ***6*** and *update the weights* **after each observation** (***Reinforcement*** Learning).

Or

Repeat Steps 1 to 6 but *update the weights* only **after a batch of observations** (***Batch*** Learning).

➢ ***STEP 8:*** When the whole ***training*** set passed ***through*** the ***ANN***. that makes an ***epoch***. Redo more ***epochs***.

Training an Auto Encoder

STEP 1: We start with an array where the lines (the observations) correspond to the users and the columns (the features) correspond to the movies. Each cell (u, i) contains the rating (from 1 to 5, 0 if no rating) of the movie i by the user u.

⬇

STEP 2: The first user goes into the network. The input vector $x = (r_1, r_2, ..., r_m)$ contains all its ratings for all the movies.

⬇

STEP 3: The input vector x is encoded into a vector z of lower dimensions by a mapping function f (e.g: sigmoid function):

$z = f(Wx + b)$ where W is the vector of input weights and b the bias

⬇

STEP 4: z is then decoded into the output vector y of same dimensions as x, aiming to replicate the input vector x.

⬇

STEP 5: The reconstruction error $d(x, y) = ||x-y||$ is computed. The goal is to minimize it.

⬇

STEP 6: Back-Propagation: from right to left, the error is back-propagated. The weights are updated according to how much they are responsible for the error. The learning rate decides by how much we update the weights.

⬇

STEP 7: Repeat Steps 1 to 6 and update the weights after each observation (Reinforcement Learning). Or: Repeat Steps 1 to 6 but update the weights only after a batch of observations (Batch Learning).

⬇

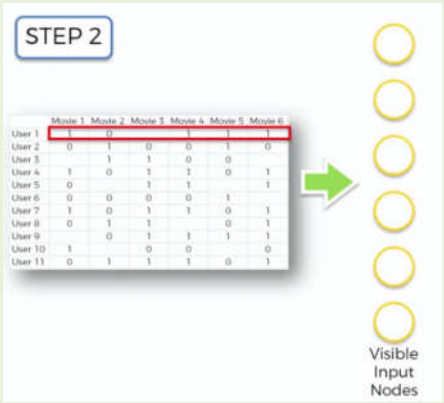
STEP 8: When the whole training set passed through the ANN, that makes an epoch. Redo more epochs.

❒ ***Steps walkthrough an example:*** Here we have ratings ***0*** and ***1*** but we can have **ratings** from ***1*** to ***5*** and ***0*** for **empty** (it will be a more powerful recommender system). In this case we've got 1, 0 and empty, we can easily change them to, "**+1**", "**−1**" for ratings and **0** for the empty cell.

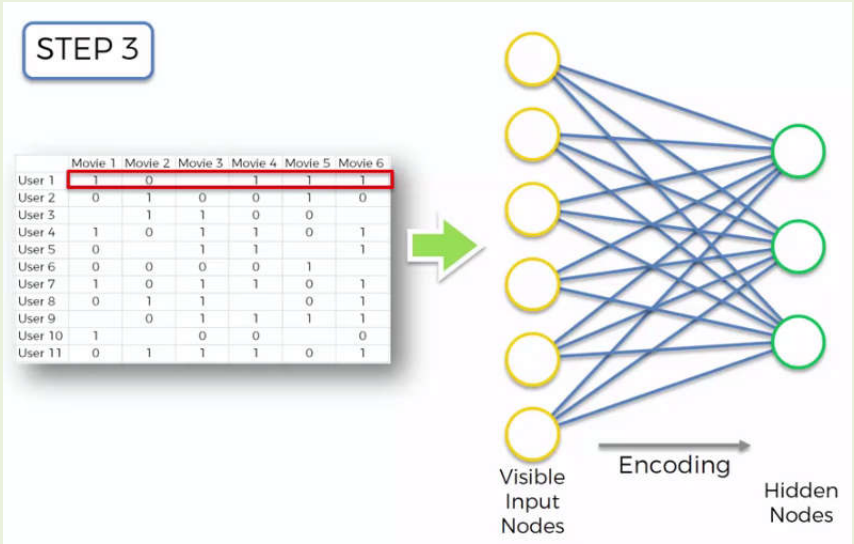
[1]. That's our ***input***.



[2]. We take the ***first row***, we put it into our ***Auto Encoder***,

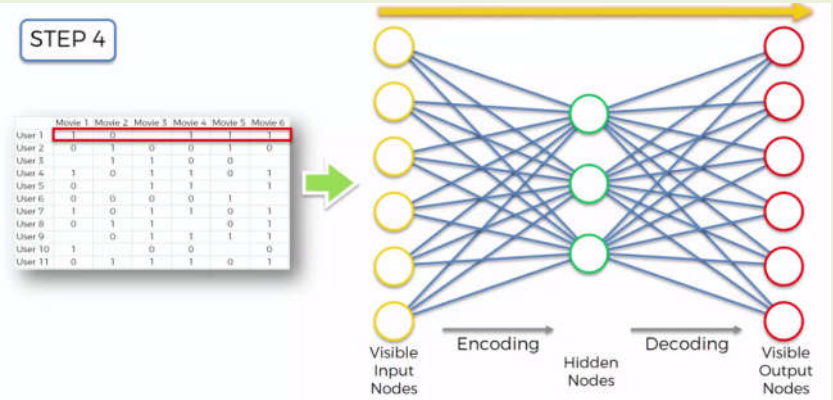


[3]. We calculate the ***hidden nodes***. At the very start you'll have some ***randomly initialized weights***. This process is called ***encoding***.

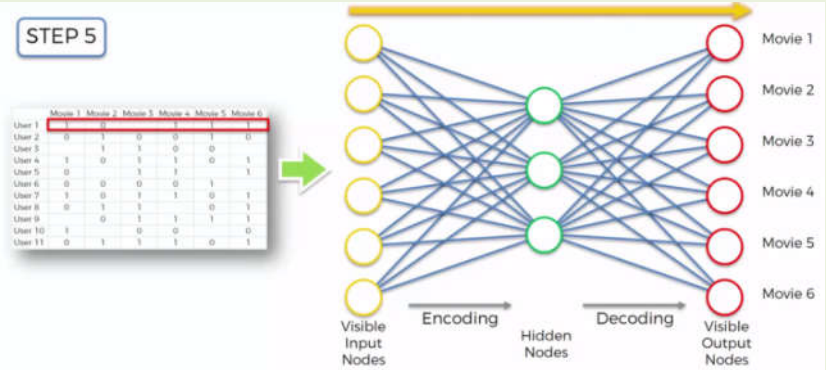


[4]. We're going to calculate our ***visible output nodes***, again initially it will be some ***random*** starting ***weights***. This process is called decoding.

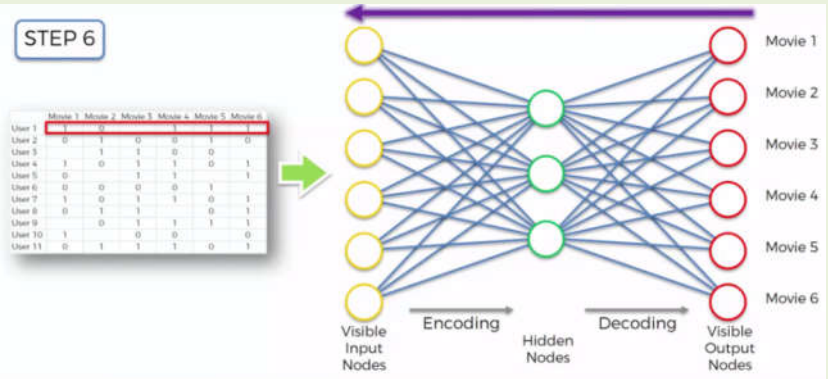
That's the ***forward propagation***. We've just put the information, the data through our ***Auto Encoder*** from ***left*** to ***right***.



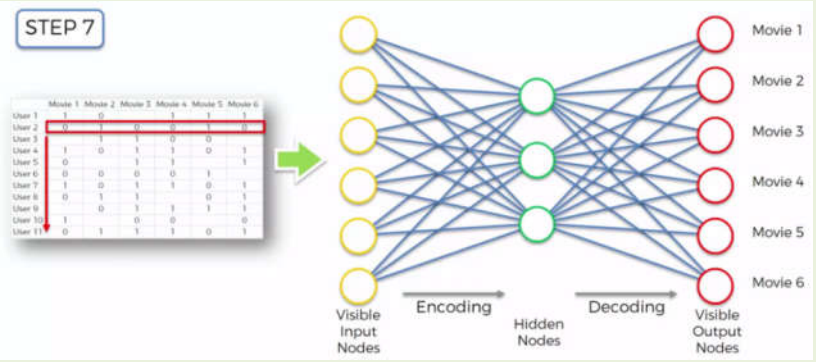
[5]. We're going to ***compare*** the results to the ***actual ratings*** for those ***six movies***. And then we'll calculate the ***error***.



[6]. And then we will ***propagate*** it ***back*** through the network, we'll ***adjust*** the ***weights*** accordingly.



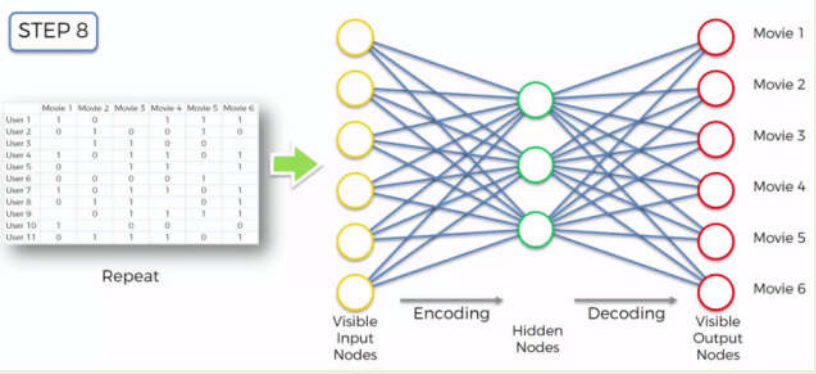
[7]. We will take the ***next row*** in our data set. And we'll do the same thing and so on. We'll ***continue*** through all the ***rows***.



[8]. It means we've finished a whole ***epoch***.

And then we just ***repeat*** these ***epochs***.

Hopefully, our training ***converges*** to some sort of ***value***.



❒ ***Additional reading:*** Here is a blog by ***Francois Chollet***, the creator of ***Keras***, it's called ***Building Autoencoders In Keras***, check it out, it's got some actual practical applications so even if you want to get some hands on experience before you start to Practice.

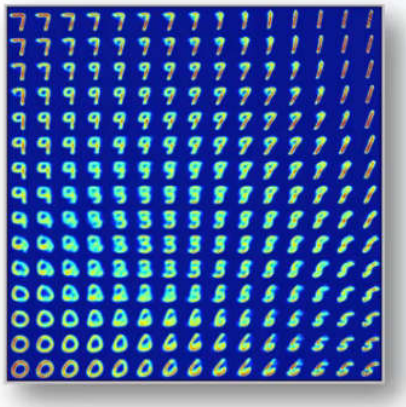
Additional Reading:

Building Autoencoders in Keras

By Francois Chollet (2016)

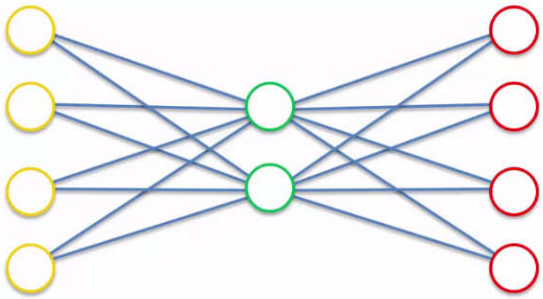
Link:

https://blog.keras.io/building-autoencoders-in-keras.html



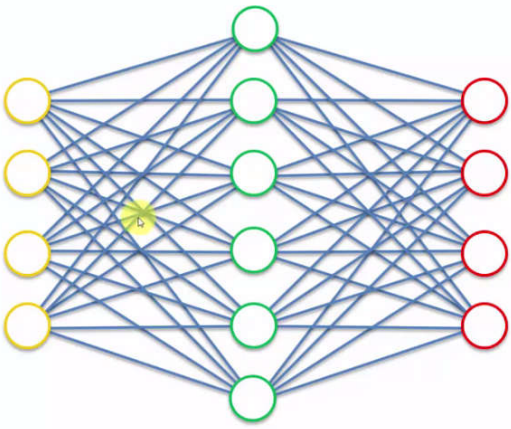
15.1.5 Overcomplete hidden layers in AE

Overcomplete hidden layers is a underlying concept in most of the variations of Autoencoders. So here we've got an ***autoencoder***, four ***input values***, two ***nodes*** in the ***hidden layer***, and four ***nodes*** in the ***output layer***.



- What if we wanted to ***increase*** the number of ***nodes*** in the ***hidden layer***?
- What if we wanted actually to have ***more nodes*** in the ***hidden layer*** than in the ***input layer***?

Why would we do that? We said that an ***auto encoder*** can be used as a ***feature extraction*** tool, what if we want ***more features***.



Number of nodes in hidden layer in ANN doesn't matter: Remember in ANN it was very easy for us to do, we had a ***certain*** number of ***inputs***, then we could have a whole ***layer*** of ***hidden nodes*** that could have ***more nodes*** than ***input layer*** (It could be six, it could be 10, it could be 100, doesn't matter). We could add more layers and so on.

- We weren't restricted to how many nodes we would have in the hidden layer. And that allowed us to extract more features in ANN.

Increasing nodes in hidden layer in Auto-Encoder can cause problem: If we increase nodes in hidden layer in ***auto encoder***, obviously the ***auto-encoder*** can ***cheat***.

- In the ***auto-encoder***, the goal is to get the ***outputs*** to equate to the ***inputs***.
- As soon as you give it a ***hidden layer*** which is the ***same size*** or ***greater*** than the ***input layer*** (in this example four or more hidden nodes), makes ***auto-encoder*** able to cheat.
- The information is just gonna ***fly through*** the ***hidden nodes*** , and no encoding-decoding happens, also ***extra hidden-nodes*** remain ***unused***.
- After the training and is going to be just useless it's not going to extract any valuable information, any valuable features for us through that process.

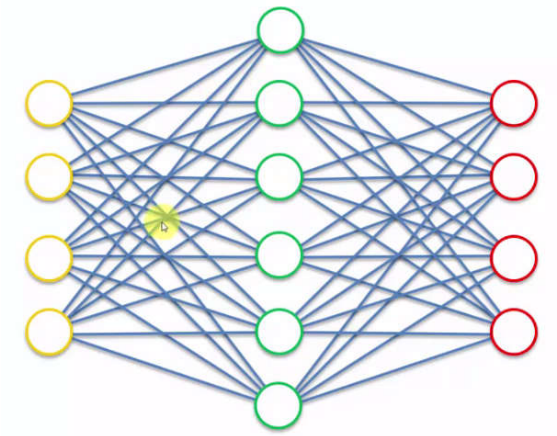
Next we're going to look at three different approaches that are used to solve that problem.

Deep Learning

Auto-Encoders (AE) : Regularization & Other types

3 Regularization techniques & Other types of Auto-Encoders

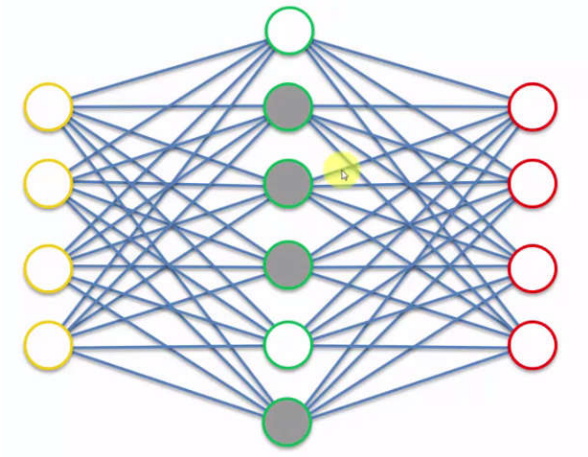
15.2.1 Sparse Auto-Encoders: Regularization technique 1



❒ A **sparse AE** is an **auto-encoder** which looks like this diagram, where the ***hidden layer*** is ***greater*** than the ***input layer***. But a ***regularization*** technique which introduces ***sparsity*** has been applied.

- ☞ A ***regularization technique*** basically means something that helps ***prevent over-fitting*** or stabilizes the algorithm.
- ☞ In this case, if it was just sending the values through the hidden layer, it would be over-fitting in a way.

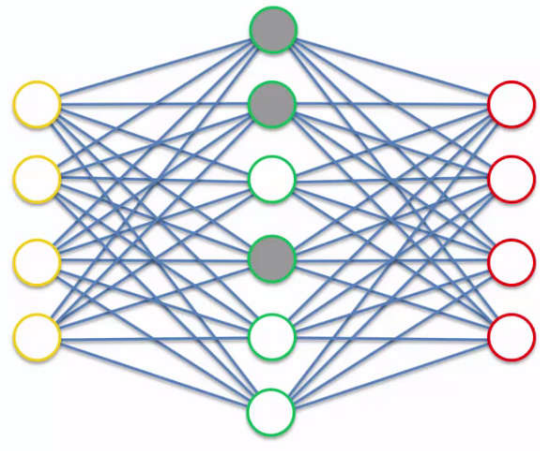
Sparse AE is one of regularization techniques that prevent over-fitting.

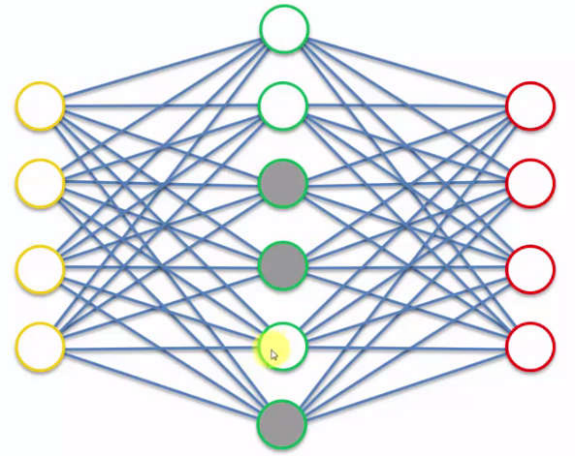


❒ Basically ***Sparse auto-encoder*** introduces a ***constraint*** on the ***loss function***, or a ***penalty*** of a ***loss function***, which doesn't allow the auto-encoder to use all of its ***hidden layer*** every ***single time***.

- ☞ So that the ***Auto-Encoder*** can only use a certain number of nodes from it's ***hidden layer***.
- ☞ For instance, in this image, ***Auto-Encoder*** can use **two nodes** in this case.
- ☞ When the values ***go*** through these ***disable-nodes***, ***output*** will be very ***small***.

❒ In following two passes the disable nodes aren't participating on encoding.





- ☞ While you training this whole hidden layer, you are extracting features from each one of these hidden-nodes, but at the same time, you're not using all of these nodes.
- ☞ So the ***auto-encoder*** cannot ***cheat*** because even though it has ***more*** nodes in the ***hidden layer*** than in the ***input layer***, but it is ***not able to use*** all of them at any **given pass**. And that's how the **sparse auto-encoder** works.
- ☞ In reality, it is still compressing the information but just every time is using different nodes.

15.2.2 Additional Reading

❒ We've got a interesting tutorial here by ***Chris McCormick*** is called ***Deep Learning tutorial Sparse Autoencoder***. And this is a ***MATLAB*** tutorial. He will walk you through how to do it in ***MATLAB***.

☞ There's some ***introductory mathematics***, not too complex, but some basic ***level formulas*** which allow you to gently get introduced to the ***mathematics*** behind the ***sparse Auto-Encoders***.

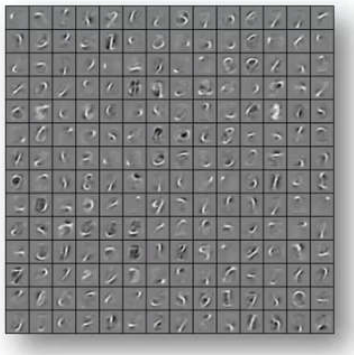
Additional Reading:

Deep Learning Tutorial - Sparse Autoencoder

By Chris McCormick (2014)

Link:

http://mccormickml.com/2014/05/30/deep-learning-tutorial-sparse-autoencoder/



☞ If you want to understand how the equations work which don't allow the auto-encoders to switch on all of its nodes at the same time, at any given point, at any given pass.

❒ The next one is called, ***Deep Learning Sparse Auto-encoders*** by ***Eric Wilkinson***. It's a very short blog post on the essence of sparse auto-encoders.

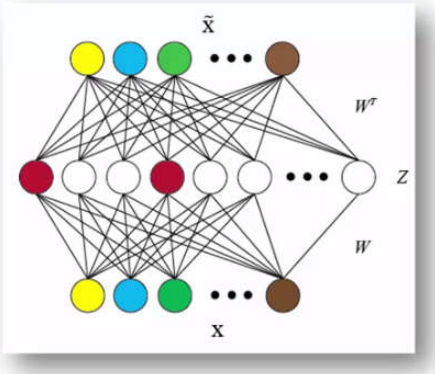
Additional Reading:

Deep Learning: Sparse Autoencoders

By Eric Wilkinson (2014)

Link:

http://www.ericlwilkinson.com/blog/2014/11/19/deep-learning-sparse-autoencoders



❒ A very strong powerful, heavy artillery paper on sparse Auto-Encoders. It's called, ***K-Sparse Auto-encoders*** by ***Alireza Makhzani***, 2014. You might find this paper interesting if you want to learn more about sparse encoders.

☞ ***K-Sparse Auto-Encoders*** used all over the place and it's constantly mentioned.

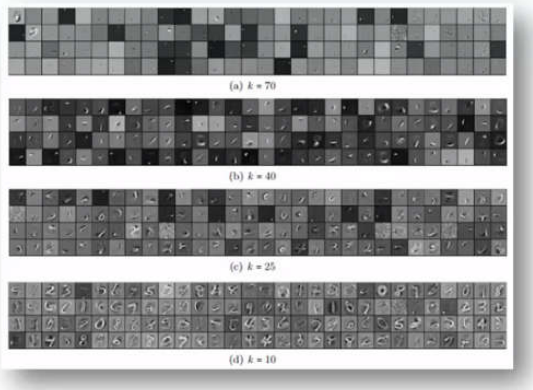
Additional Reading:

k-Sparse Autoencoders

By Alireza Makhzani et al. (2014)

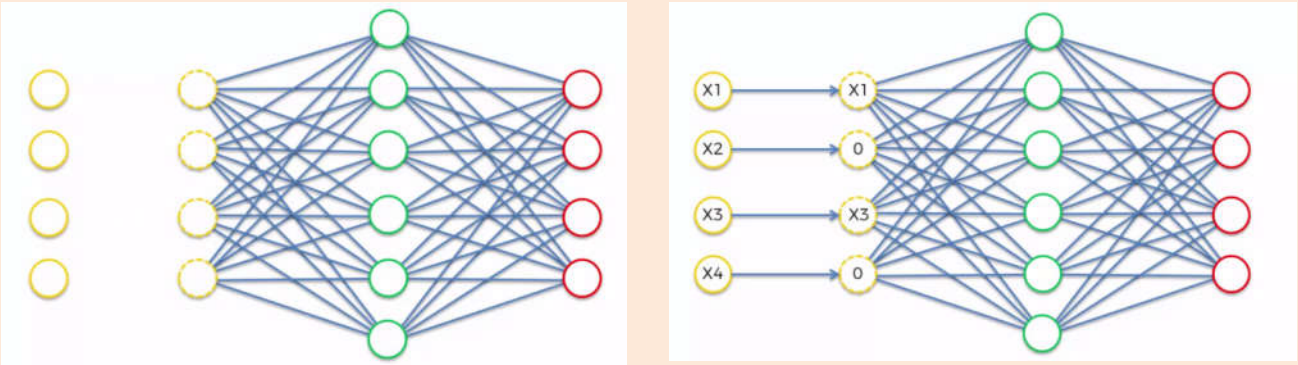
Link:

https://arxiv.org/pdf/1312.5663.pdf



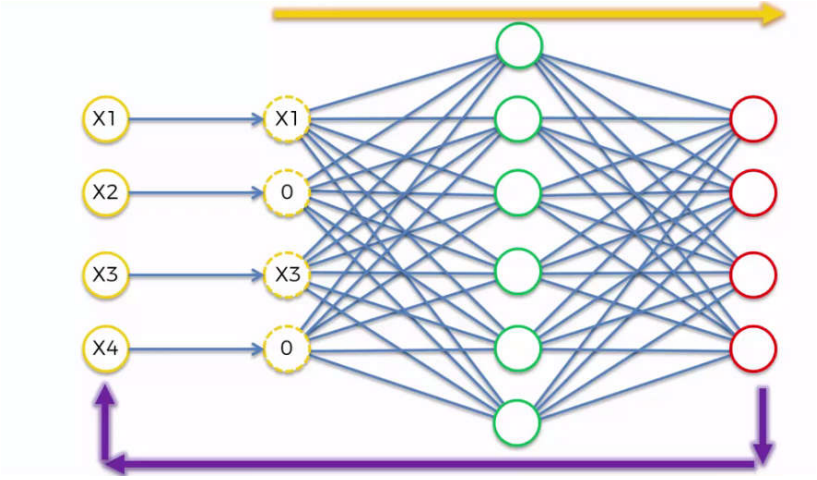
15.2.3 Denoising Auto-Encoders: Regularization technique 2

Denoising Autoencoders is another **regularization** technique, a quite a popular technique actually, which resolves the problem when ***"we have more nodes in the hidden-layer than in the input-layer and the Auto-encoder simply copy input values to output-values without finding any meaningful features"***.



- Here we are going replace the ***actual input-values*** with ***noised-input-values***.
  - ***Noised-input-values*** are the ***input-values*** where we randomly make some of the ***input-values*** equal to ***zero***.
  - ***Noised-input-values*** are modified version of our ***input values***. So, let's say we have input values ***{X1, X2, X3, X4}***. Now we're going to take these inputs randomly and turn some of them into zeros, ***{X1, 0, X3, 0}***.
  - You can specify the number of **0**'s in the ***setup*** of your ***Auto-encoder***, it can be, for instance, ***half*** of your ***inputs*** that you have are turned into ***zeros*** every ***single time*** and it happens ***randomly***. At every single pass it can be different variables.
  - Also it is important to note that because this happens randomly, this type of ***Auto-Encoder*** is a ***Stochastic Auto-Encoder***.

- Once you put this data through your Auto-Encoder then in the end you compare the output, not with the ***modified/noised-input-values***, but with their ***original input-values***.
  - And that ***prevents*** the ***Auto-encoder*** from simply just ***copy*** inputs all the way through to the ***outputs*** because it's actually comparing the ***output***, not with the ***noisy*** but with the ***original inputs*** and that helps ***resolve*** the problem that we are facing.

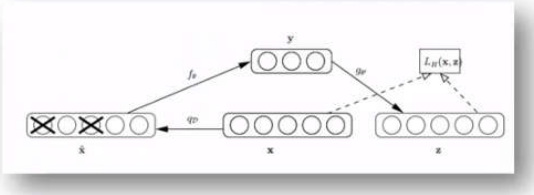


- ***Additional Reading:*** In terms of additional reading, here is a great paper by **Pascal Vincent** and others, 2008, it's called ***Extracting and Composing Robust Features with Denoising Autoencoders***.

Additional Reading:

Extracting and Composing Robust Features with Denoising Autoencoders

By Pascal Vincent et al. (2008)

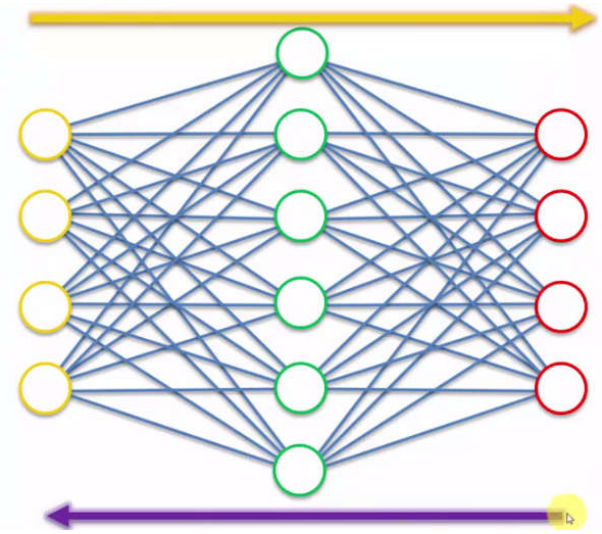


Link:

http://www.cs.toronto.edu/~larocheh/publications/icml-2008-denoising-autoencoders.pdf

15.2.4 Contractive Auto-Encoders: Regularization technique 3

Contractive Auto-Encoders is another ***regularization technique*** just like ***sparse Auto-Encoders*** and ***denoising Auto-Encoders***. Which is designed to ***resolve*** the same problem of ***over-complete hidden layer*** in the Auto-Encoders.



- We are not going too much detail, the ***Contractive Auto-Encoders*** add a ***penalty*** into the ***loss-function*** during the ***back propagation***. It doesn't allow the Auto-Encoders to just simply copy these values across.

- ***Additional Reading:*** A wonderful article by **Salah Rifai**, it's called ***"Contractive Auto-Encoders: Explicit Invariance During Feature Extraction"***. This article has quite a lot of authors and one of them is Yoshua Benjio, so quite an in-depth study into the topic.

  ☞ Based on this article they say that they can even get ***better results*** than **Denoising Auto-Encoders** on some certain data sets.

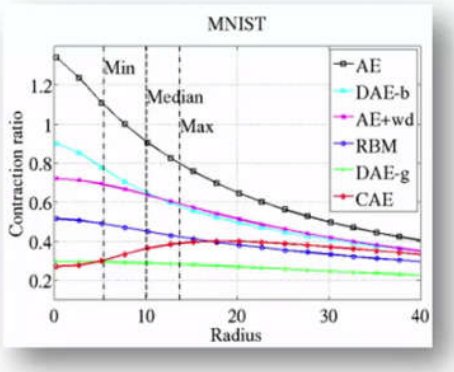
Additional Reading:

Contractive Auto-Encoders: Explicit Invariance During Feature Extraction

By Salah Rifai et al. (2011)

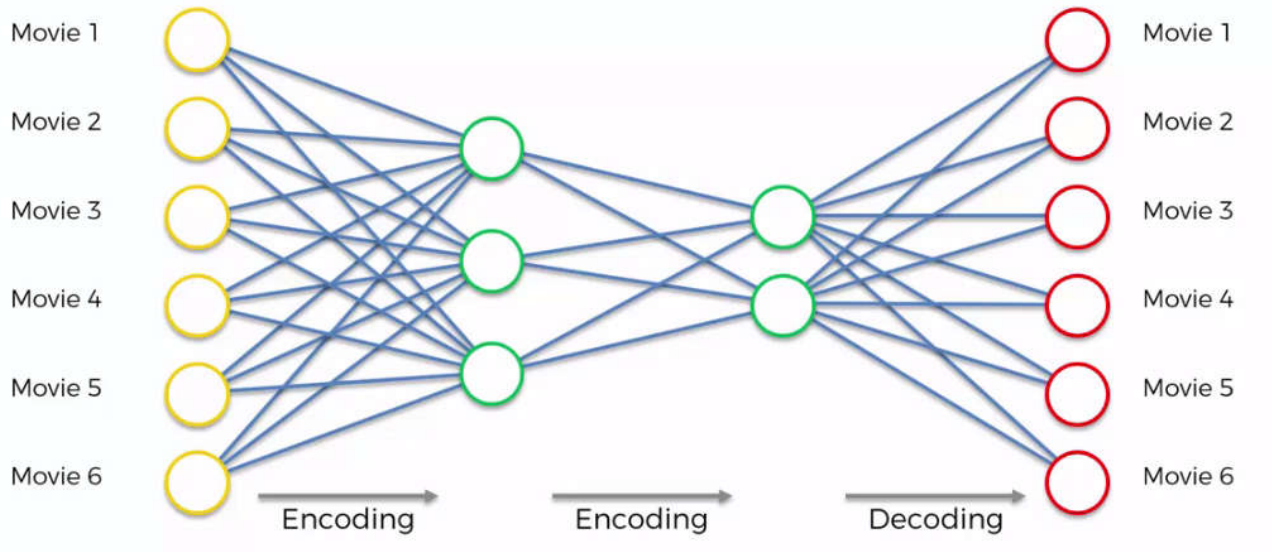
Link:

http://machinelearning.wustl.edu/mlpapers/paper_files/ICML2011Rifai_455.pdf



15.2.5 Stacked Auto-Encoders (AE)

Here is a ***Stacked AE***. A ***stacked AE*** is an ***AE*** with ***another hidden layer***. So we have ***two*** stages of ***encoding*** and ***one*** stage of ***decoding*** here. And what we get is a very, very powerful algorithm.



- In fact it's been shown that sometimes ***stacked AE*** can supersede the results that are achieved by DBN. Where **DBN** is **undirected NN** and is a **Boltzmann Machines**. But ***stacked AE*** is a directed NN.

- ***Additional reading:*** A great paper to look into on this topic is by ***Pascal Vincent*** and others is called ***Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion***. It's quite a large paper.
  - It also has **Yoshua Bengio** one of the co-authors on this paper as well, and it builds upon their **2008 paper** which we have **already referenced** before.

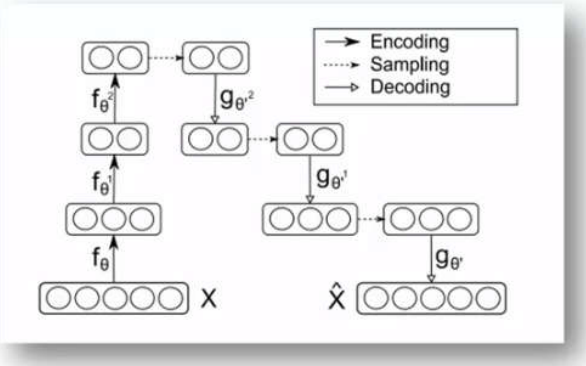
Additional Reading:

Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion

By Pascal Vincent et al. (2010)

Link:

http://www.jmlr.org/papers/volume11/vincent10a/vincent10a.pdf

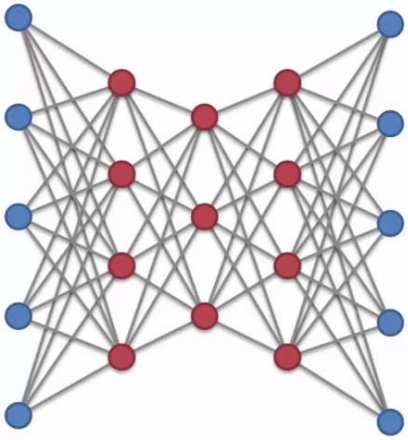


15.2.6 Deep AE

Note that ***Stacked AE*** are not the same thing as ***Deep AE***.

Stacked AE <> Deep AE

- Right side image is a ***Deep AE***. These are actually **stacked *pre-trained***, **layer by layer *RBMs***,
  - ***Deep AE*** basically are RBMs that are stacked. They're pre-trained layer by layer. Then they're unrolled. Then they're fine tuned with back propagation.
  - Here you get directionality in your network and then you have back propagation. But in essence a Deep AE, comes from RBMs.



NOTE :

- ***Stacked AE*** are just ***normal AE*** that are ***stacked***.
- A ***deep AE*** is ***RBMs stacked*** on top of each other, and then certain things are done with them in order to achieve a ***AE mechanism***.

- ***Additional reading:*** If you'd like to learn more about ***deep AE***, a great paper by ***Geoffrey Hinton*** and others called, ***Reducing the Dimensionality of Data with Neural Networks***.

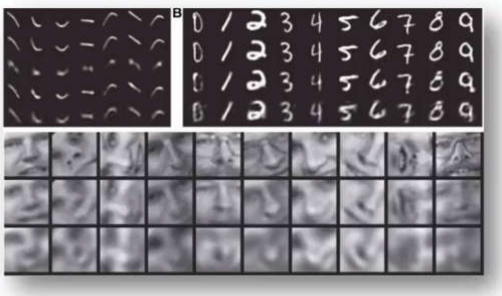
Additional Reading:

Reducing the Dimensionality of Data with Neural Networks

By Geoffrey Hinton et al. (2006)

Link:

https://www.cs.toronto.edu/~hinton/science.pdf



Deep Learning

AutoEncoder : project

15.3.1 Data Preprocessing

❒ We will *train* our *auto-encoders- AE* on this *training set*, and so it will try to identify some *patterns* to find some *groups of movies* that are liked by *similar segments of users*,

- ☞ ***AE*** will find these *patterns*, it will find some *specific features* of the *movies* which will be the *hidden nodes* in the *auto-encoders*, and these *specific features* can be some *genres*, some *actors* that are in the *movies*, or some *directors*.
- ☞ For example, *one hidden node* can be a *specific director*, like *Tarantino*, for example, or it can identify *some movies* with a *great actor* that is *liked by the same group of people* because they gave the *same high rating*.

☝ So basically it will **identify** some **patterns**, it will identify **some features**, based on which, in the **future**, the ***AE-recommender-model*** will be able to *predict* the *rating* of a movie that *one user hasn't seen yet*, and it will be able to *predict* that *rating* based on the *features* that the *auto-encoder's detected*, and based on the *history* of this *user*.

☝ The ***AE*** will take the ***features*** that it ***detected***, and it will take also the *ratings* of that one same user to *predict* the *rating* of the *new movie* that the same *user hasn't seen*, and therefore has *not rated yet*.

❒ There is *no common rating* of the *same movie* by the *same user* between the ***training sets*** and the ***test set***, however, we have the *same users*, for example in train-set we start with *user one*, as in the *training set*. But, for this same *user one*, we won't have the *same movies* because the *ratings* are *different in the test-set*.

✋ ***It is not a supervised technique:*** *Test set* is just to get the *real results* on one's side, and so what will happen is that we will make our *predictions* using *our auto-encoder's model* to predict the rating, for example, ***AE*** could *predict* rating `4` of `movie 14` by `user-1`, but in test-set the real rating is `5`.

☝ Then, we will measure the *error* between the *real rating, 5*, and the *prediction, 4*, and that will allow us to ***measure*** the ***MSE*** that is the ***Mean Squared Error***.

❒ We are going to *convert* our *training sets* and *test set* into *2 matrices*, where the `lines/rows` are going to be the `users`, the `columns` are going to be the `movies`, and the `cells` are going to be the `ratings`.

- ☞ In each of these *two matrices*, we want to include *all the users* and *all the movies* from the *original data set*.
- ☞ And if in the *training set*, a *user didn't rate* a *movie*, well we'll *put* a *zero* into the *cell* of the *matrix* that corresponds to this *user* and *that movie*.
- ☞ These *matrices* will have the *same number of users* and the *same number of movies* so they will have the *same number of lines/rows* and the *same number of columns*.
- ☞ And in these *two matrices*, each `cell` of indexes `U, I`, where `U` is the `user` and `I` is the `movie`, each `cell UI` will get a `rating` of the `movie I` by the `user U` and if this **user U didn't rate** the **movie I**, we'll put a `0`.

For this reason we get first *total number of users* and *total number of movies*.

❒ We need to make those *matrices* for a *specific structure of data* that will fit to the *structure* of the *AE* and the *AEs* are like *neural networks*, where you have some *input nodes* that are the *features* and you have some *observations* going one by one into the *NN* starting with the *input nodes*.

- ☞ So we created ***list of lists*** will be a list of *943 lists* because we have *943 users* and each of these *943 lists* will be a *list of 1,682 elements* because we have *1,682 movies* (exactly same as RBM-model).

❒ We use *tensors* instead of *NumPy array*. A *tensor* is a *multidimensional matrix*, but instead of being a ***NumPy array***, this is a ***PyTorch array***. And in fact, we could build a neural network with *NumPy arrays*, but that would be much *less efficient* and that's why we're using *tensors*, as what we're about to do with the ***torch.tensors***.

- ☞ So, the `training_set` is going to be one `torch.tensor` and the `test_set` is going to be another `torch.tensor`. Two separate multidimensional matrices based on ***PyTorch***.

Note that, with ***TensorFlow***, we work with ***tensors***, that are ***TensorFlow Tensors***.

- Those are another kind of ***tensor***, another kind of ***multidimensional matrix***.
- We could also ***implement*** our ***AE*** from ***scratch*** with ***TensorFlow***. But it turned out that for ***AE***, ***PyTorch*** give ***better results***, and also it is much more simple.

We copy all code from our **RBM-recommender**. We use the code ***before*** the ***binary-rating conversion***. Because in our AE-model we actually predict ratings from ***1*** to ***5***.

```
# ----------- AE : Recommender ---------------

# Importing the libraries
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable


# ---------- importing the dataset -----------

# preparing the training set and test set
training_set = pd.read_csv("./movie_lens_100k/u1.base", delimiter="\t")
train_set = np.array(training_set, dtype="int")

ts_set = pd.read_csv("./movie_lens_100k/u1.test", delimiter="\t")
test_set = np.array(ts_set, dtype="int")


# Getting the number of Users and Movies
nb_users = int(max(max(train_set[:, 0]), max(test_set[:, 0])))
nb_movies = int(max(max(train_set[:, 1]), max(test_set[:, 1])))


# converting the data into an array with users in lines and movies in column.
def conVert(data):
    new_data = []
    for id_user in range(1, nb_users + 1):
        # use "data[:, 0] == id_user" as condition over movie column "data[:, 1]"
        id_movies = data[:, 1][data[:, 0]== id_user]    # returns a list

        # use "data[:, 0] == id_user" as condition over ratins column "data[:, 2]"
        id_ratings = data[:, 2][data[:, 0]== id_user]

        # vector of zeros
        ratings = np.zeros(nb_movies)
        ratings[id_movies - 1] = id_ratings

        new_data.append(list(ratings))

    return new_data

trn_set_cnvt = conVert(train_set)
tst_set_cnvt = conVert(test_set)


# Converting the data into Torch Tensosrs. Following are the Tensors of ratings
train_set_tensor = torch.FloatTensor(trn_set_cnvt)
test_set_tensor = torch.FloatTensor(tst_set_cnvt)
```

Next, we create the ***architecture*** of our ***NN***, that is our ***AE-class***. We'll make an ***AutoEncoder-class***. And we'll use this ***class*** to build our ***AE-recommender-model***. Inside this class we'll define 2 functions, one is ***__init__*** and other is ***forward()***.

- We can use it afterwards to change the architecture of the AE to try other AE-architectures.

15.3.2 AE-architecture : StackedAutoEncoders class - `__init__()`

The class that we're going to make will be the model that will contain the instructions on how to build the auto-encoder.

❒ We're making this class, because to make an auto-encoder, we need to define multiple things:

- ❖ How many layers we want to have,
- ❖ How many nodes in the layers,
- ❖ We also need an activation function,
- ❖ A criterion, and
- ❖ An optimizer function.

☞ To make an auto-encoder, we not only need some ***variables***, that's will get, for example, the ***info of the layers***, and ***some functions***, for the ***activation*** and the ***optimizer***. And to get all this in one same recipe, well, we can only use a ***class***, or, at least, that's the ***simple solution***.

☝ We could also a ***module*** contains ***several classes***,

☝ Or even some ***libraries***, which contain ***several modules***.

☝ But the ***simplest solution*** is to make a ***class***. And a *simple function wouldn't be enough*.

❒ So the ***first reason*** to make this class is: a ***`class`*** can gather all these features, variables and functions to make the ***`auto-encoder`***.

❒ But then there is a ***second reason***, and this is very important, because this is related to ***PyTorch***.

☞ ***Inheritance:*** We're going to create a class that's we're gonna call ***`StackedAutoEncoders`*** (SAE in short).

☞ That is actually going to be the ***child class*** of an existing ***`parent`*** class in ***PyTorch***. This parent class is called ***`Module`***.

☞ And this *class* ***`Module`*** is taken from the ***`nn`*** *module* that we imported here: ***`torch.nn as nn`***.

☝ We are doing this so that we can use ***`all`*** the ***`variables`*** and ***`functions`*** from the parent class ***`Module`***, and that's what ***inheritance*** is all about.

☝ Because this parent class ***Module*** contains all the tools to make an ***auto-encoder***. Basically, it contains everything we need to make an ***auto-encoder***.

- ❖ It contains an ***optimizer*** function,
- ❖ It contains a ***criterion***,
- ❖ It contains tools to make ***full connections*** between the layers.

❒ Our auto-encoder is a stacked auto-encoder, because we will have ***several hidden layers***, so we will have ***several encodings*** of the ***input vector features***.

☞ We're gonna call this child class ***`StackedAutoEncoders`***, with capital S-A-E, because we use capitals for classes.

☞ In the parentheses of ***`StackedAutoEncoders()`***, we're going to input the parent class which is Module.

```
class StackedAutoEncoders(nn.Module):
```

❒ ***`__init__():`*** We now define the ***`__init__()`*** function because we need it to ***initialize*** the ***objects*** that are created from this SAE-class. We define it as:

```
def __init__(self, ):
```

➢ We add a comma, and then just nothing **`(self, ):`**, because this will just **consider the variables** of the ***`Module`*** class, because we are doing inheritance (it is something like ***variable arguments***).

☞ Now using the ***`super`*** function we get the inherited methods from the ***`Module`*** class.

```
def __init__(self, ):
        super(StackedAutoEncoders, self).__init__()
```

`super(StackedAutoEncoders, self).__init__()` will make sure we get all the **inherited** classes and methods of the **parent** class ***`nn.Module`***.

➢ We're gonna use the ***`nn.Linear()`*** class, to make the different ***`full connections`*** between the ***`layers`***.

❒ ***Full Connection - ENCODING (first hidden layer):*** Now we start creating the ***architecture*** of the ***neural network***, by choosing the ***number of layers*** and the ***hidden neurons*** in each of the ***hidden layers***.

☞ The first part of the ***neural network*** that is the ***full connection*** between the ***input vector*** of features (ratings of all the movies for one specific user) and ***first hidden layer***.

☞ The *first hidden layer* in ***AutoEncoders*** is a *shorter vector* than the ***input vector***, that's the technique of ***auto-encoders***.

➢ We're encoding the ***input vector*** into a ***shorter vector***. That's will take place in the ***first hidden layer***.

☞ Now we're going to create an ***object*** of the ***class*** that is inherited from the ***nn*** module. This object will represent the ***full connection*** between this ***first input vector features*** and the ***first encoded vector***.

➢ We're gonna call this **first full connection** ***fc1***, and it is associated to our object so we used ***self.fc1***,

➢ Then we use the inherited class ***Linear()*** from the ***nn*** module. So, ***self.fc1 = nn. Linear()***

```
self.fc1 = nn.Linear(nb_movies, 20)
```

➢ Parameters of ***nn.Linear()***:

i. The first input is the number of features in the input vector, i.e. number of movies ***nb_movies***. Since the ***features*** are actually ***movies***, because ***one observation (one-user)*** contains all the ***ratings*** of all the ***movies***.

ii. The second input is going to be the ***number of nodes/neurons*** in the ***first hidden layer***. i.e the ***number of elements*** in the ***first encoded vector***.

- We tried several values for this vector. The value we're gonna use is based on ***experiments***, and the value is 20. But this is still not tuned. To get a better score we can tune it later.
- ***20 nodes*** in ***first hidden layer*** led us to a pretty good score. It means that our first ***encoded vector*** is a vector of ***20 elements***.

If we want to make the parallel with the problem, well, ***what would represent this first hidden layer composed of 20 neurons?***

☝ Remember that the ***neurons*** in the ***auto-encoder*** represent actually ***some features*** that, through ***unsupervised learning***, the auto-encoder detects.

- So, these ***20 features*** that are in the ***first hidden layer*** can represent some ***features of movies*** that are liked by ***similar people***.
- For example, ***one*** of these ***20 nodes*** could be a ***specific genre*** of a movie. One of the **detected feature** could be the ***horror*** movie ***genre***.
- In that case, when a ***new user*** comes in the ***system*** then, if this new user gave some ***good ratings*** to ***all*** the ***horror movies*** of the ***database***, then this will ***activate*** this ***horror genre neuron/node*** and therefore a ***big weight*** will be **attributed** to this **neuron/node** in the *final prediction* to predict the *rating* of a *horror movie*.

  Of course this is a very simple example and of course, the **features** can be much **more complex** that even would be **difficult to define**, but *that's how it works*.

☝ With this ***number 20*** here, we're trying to detect ***20 features***.

❒ ***Second full connection - ENCODING (Second layer):*** Since we're making some ***stacked auto-encoders***, we're gonna make ***another layer***. We have to make a *second full connection*.

```
self.fc2 = nn.Linear(20, 10)
```

☞ It is same as previous layer, we just need to give this layer a name "**fc2**" i.e. full-connection-2.

☞ Again we're gonna use the ***Linear*** class that will make this ***full connection***.

☞ To make the full connection between the ***first hidden layer***, composed of the ***20 neurons*** and this second hidden layer, then we need to specify the number of ***nodes of previous layer*** (which is 20) and since we are still ***encoding***, we choose ***smaller number*** of nodes for this ***layer***, by experiment this number is ***10***.

☞ In this ***second hidden layer*** with *10 neurons*. it will ***detect*** even more ***features***, ***based*** on the ***previous features*** that were detected.

❒ ***Third full connection - DECODING (3rd layer):*** Since we're doing deep learning, let's add a third layer. But now, we start DECODING.

```
self.fc3 = nn.Linear(10, 20)
```

☞ We name this third hidden layer ***fc3***, the third ***full connection*** between the ***second hidden layer*** and the ***third hidden layer***.

☞ Since we are now ***Decoding***, the number of ***nodes*** will be ***higher*** than the ***previous layer***. We specify ***10 neurons*** of ***previous*** layer and ***20*** for this ***third layer***.

☞ Here we're just starting to ***reconstruct*** the ***original input vector***. So, let's make things ***symmetrical***.

❒ ***Fourth full connection - DECODING (4th layer):*** This is the last part of the ***decoding***, the last full connection we have to make. We name the *fourth full connection* as ***fc4***. Here we input ***20***, because we had ***20 nodes*** in the third layer.

```
self.fc4 = nn.Linear(20, nb_movies)
```

☞ Now this layer is the ***last layer***, and the number of ***neurons*** in the ***output layer*** must be ***same*** as ***input layer***, because, in ***auto-encoders***, we are ***re-constructing*** the ***input vector***.

☞ Hence the ***output vector*** should have the ***same dimension*** as the ***input vector***.

☞ Therefore, the ***number of neurons*** in the ***output layer*** is also going to be ***nb_movies***.

❒ ***Activation function:*** Now we have to specify an ***activation*** function that will, ***activate the neurons*** when the ***observation enters*** into the ***network***.

➢ For example, if someone gives some ***good ratings*** for ***horror movies***, well, this will ***activate*** the ***specific feature*** for the ***horror movie genre***. And this activation will be done with a certain function,

☞ There are *different activation functions*, we tried the ***Rectifier Activation*** function and also, the ***Sigmoid Activation*** function, and it turns out that we got ***better results*** with a ***Sigmoid Activation*** function between the ***four full connections***.

```
self.activation = nn.Sigmoid()
```

☞ So, to get the sigmoid activation function, we use ***nn.Sigmoid()***, from the ***nn*** module, we don't need any arguments for now.

☞ Later we can try some combinations of the ***Rectifier Activation*** function and the ***Sigmoid Activation*** function, to get better result.

```
class StackedAutoEncoders(nn.Module):
    def __init__(self, ):
        super(StackedAutoEncoders, self).__init__()
        self.fc1 = nn.Linear(nb_movies, 20)
        self.fc2 = nn.Linear(20, 10)
        self.fc3 = nn.Linear(10, 20)
        self.fc4 = nn.Linear(20, nb_movies)
        self.activation = nn.Sigmoid()
```

So that's all for the **__init__()** function. That's the ***architecture*** of our ***stacked auto-encoder***. You're totally welcome to try to change it.

☝ Technically speaking, we just created ***five objects*** of ***two different classes***, ***four objects*** of the ***Linear*** class, and ***one object*** of this ***Sigmoid*** class.

15.3.3 AE-architecture : StackedAutoEncoders class - `forward()`

Now, we're gonna make another function that is required to build our auto-encoders. This second function going to make the ***action that takes place*** in ***auto-encoder***. This action is basically the ***encoding***, and the ***decoding***.

➢ We're going to name this function ***forward()***. And that will proceed to the different ***ENCODINGS*** and ***DECODINGS*** when the observation is ***forwarded*** into the network. We spell it correctly because PyTorch will use it (so ***foRWard()*** may not work it has to be ***forward()***).

☞ It will not only do the action of encoding and decoding, but also will apply to ***different activation functions*** inside the full connections.

☞ Also, the main purpose of making this function is that it will ***return*** in the end the ***vector of predicted ratings*** (output-vector) that we will ***compare*** to the ***vector of real ratings*** (input-vector).

☝ We call this function ***forward***, like ***forward propagation***, because during the forward propagation the ***encoding*** and ***decoding*** take place.

❒ ***Arguments:*** In this function, we have to input **two arguments**, ***self*** and ***x*** *(x is our input vector: the features with all the ratings for the movies in one specific user)*,

☞ then we'll ***transform*** this input vector ***x***, by ***encoding*** it ***twice*** and then ***decoding*** it ***twice*** again to get the ***final output vector*** i.e. the decoded vector that was reconstructed.

```
def forward(self, x):
```

❒ ***First-Encoding:*** First, we are about to encode our input vector ***features x*** into a first ***shorter vector*** composed of ***20 elements*** in our ***first hidden layer***.

```
    def forward(self, x):
        x = self.activation(self.fc1(x))
```

☞ we're going to take our AE-object (i.e. ***self***) and then we'll use our ***activation*** object that we created using the ***Sigmoid*** class, because this ***sigmoid activation function*** will activate the *neurons* of this first *encoded vector of 20 elements*.

☞ The ***parameter*** of this ***activation*** function will **not** be **directly** the input vector **x**. We've used ***fc1(x)***, because we are ***encoding***, and to do the ***encoding*** we have to apply the ***activation*** function on the ***first full connection*** **fc1**. Hence to include that information of the ***first full connection*** we don't input **x** ***directly***.

➢ Since ***fc1*** is an object of the ***Linear*** class it will apply the full-connection over ***x***. And it becomes the first-full-connection.

☝ Since ***forward()*** will return the ***output vector*** in the end using ***several encoding-decoding***, and we'll *compare* it to the ***real rating***.

➢ For this reason, we're going to ***modify x*** after each ***encoding*** or ***decoding***, hence we used the variable ***x*** again so that it will be ***modified*** by the ***activation*** function.

➢ The ***first encoded vector*** in the first hidden layer will be the new modified ***x***.

```
x = self.activation(self.fc1(x))
```

➢ This ***x*** in `self.fc1(x)` is the input vector features of the ***first full connection***

➢ and this ***x*** in "`x = self.activation(`" will be the ***new first encoded vector*** resulting from this ***first encoding*** that happens here with the ***activation function*** in the ***first full connection***.

Now basically we need to do the ***same*** for the ***other full connections*** and so this is going to be ***exactly*** the ***same***.

❒ ***Second-Encoding:*** We ***update x*** of the ***first hidden layer***, then on this vector in the first hidden layer, we made the ***second full connection*** which will ***encode*** this vector of ***20*** elements into a ***shorter vector*** of ***10*** elements.

```
x = self.activation(self.fc2(x))
```

☞ At the same time, we apply the ***sigmoid*** activation function to ***activate*** the ***neurons*** and then eventually ***x*** becomes this ***new encoded*** vector of ***10*** elements in this ***second hidden layer***.

❒ ***first-Decoding:*** We do the same for the third full connection represented by ***fc3***. Now we are DECODING.

```
x = self.activation(self.fc3(x))
```

☞ The third full connection ***fc3*** corresponds to the decoding from an ***input vector*** of ***10 elements*** in the second hidden layer to a ***larger output vector composed of 20 elements***.

❒ ***Final Decoding:*** To get our final output vector we use ***fc4*** for 2nd (final) DECODING. It will be our reconstructed output vector, similar to input vector.

```
x = self.fc4(x)
```

☞ But we ***don't apply*** the ***activation*** function because this is the final part of the decoding.

☞ We only have to use our **fourth full connection**, ***fc4 without*** the ***activation*** function.

❒ From the ***forward()*** we ***return*** the reconstructed output ***x***. Then ***x*** is our vector of ***predicted ratings***.

```
def forward(self, x):
    x = self.activation(self.fc1(x))
    x = self.activation(self.fc2(x))
    x = self.activation(self.fc3(x))
    x = self.fc4(x)
    return x
```

Finally ***StackedAutoEncoders*** class is done. It was composed of two functions,

[1]. the ***__init__()*** defines the ***architecture*** of our ***auto encoders*** and

[2]. the ***forward()*** does the action of establishing different full connections by applying at the ***activation functions*** to ***activate*** the ***right neurons*** in the network.

➢ In ***forward()***, we did ***two encodings*** and then ***two decodings*** to get our **reconstructed output vector** in the output layer.

Next we'll **compare *output*** to the ***real*** ratings, to measure the ***loss*** so that we can then obtain the ***weight*** to *reduce* this *loss*.

15.3.4 SAE model : criterion & optimizer for SAE architecture

Now we create an object of ***StackedAutoEncoders*** class. We name this object "***sae***". Since we didn't specify any arguments in the ***__init__()*** function of this ***StackedAutoEncoders*** class, we don't have to input any arguments here.

```
sae = StackedAutoEncoders()
```

❒ ***Criterion:*** Which we'll need this criterion to train the model.

```
criterion = nn.MSELoss()
```

☞ It's basically the ***criterion*** for the ***loss function***, and the ***loss function*** is going to be the ***mean squared error*** so we used `nn.MSELoss()` class from ***nn*** module.

❒ ***Optimizer:*** Now the last thing that we need is an ***optimizer***. It is much more like in ***Keras***. The optimizer will apply ***Stochastic Gradient Decent*** to update the ***different weights*** in order to reduce the ***error*** at each ***epoch***.

☞ We'll use ***torch.optim*** module to import our optimizer. We can use ***ADAM-optimizer*** or ***RMSProp-optimizer***. By experiment ***RMSProp*** gives better result.

```
optimizer = optim.RMSprop(sae.parameters(), lr = 0.01, weight_decay= 0.5)
```

☞ ***Arguments:*** We will actually input three things.

i. We have to input all the parameters of our ***AutoEncoders***, that is the parameters to build our ***StackedAutoEncoders*** architecture.
   - ➢ We don't have to rewrite all that again. The easy way is to use ***sae.parameters()*** that gets all the parameters from our ***sae*** object.
   - ➢ **lr**: Is the learning rate. It depends on experiments, a good value I found was ***0.01***. Of course we can tune it to get better result.
   - ➢ **weight_decay**: Specifies the decay. Decay is used to ***reduce*** the ***learning rate*** after every few ***epochs*** and that's in order to ***regulate*** the ***convergence***. We can also tune it. Based on my experimenting a good value I found was ***0.5***.

✋ NOTE: We apply ***parameters*** attribute on ***sae*** object , not the class ***StackedAutoEncoders***.

```
# -------- Creating the architecture of the Neural Netwark for SAE --------
class StackedAutoEncoders(nn.Module):
    def __init__(self, ):
        super(StackedAutoEncoders, self).__init__()
        self.fc1 = nn.Linear(nb_movies, 20)
        self.fc2 = nn.Linear(20, 10)
        self.fc3 = nn.Linear(10, 20)
        self.fc4 = nn.Linear(20, nb_movies)
        self.activation = nn.Sigmoid()

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.activation(self.fc3(x))
        x = self.fc4(x)
        return x

sae = StackedAutoEncoders()
criterion = nn.MSELoss()
optimizer = optim.RMSprop(sae.parameters(), lr = 0.01, weight_decay= 0.5)
```

Now that's the end of the ***architecture*** of our ***neural networks***. Next we're gonna implement the training,

15.3.5 Train the SAE model : Nested-for-loop & input

Our code will be bit technical to optimize our code so that you can use it on high dimensional datasets. Here we'll use a dataset with 200,000 ratings. If you want to use this code for the 1 million ratings dataset or even a larger dataset, you need an optimized code that saves up the memory as much as possible.

We won't go in too much details about the technique and besides this is some technique related to PyTorch. The most important thing that matters is that you understand

- How the architecture of the **Neural Network** works.
- How to manipulate our ***StackedAutoEncoders*** architecture by changing the *number of layers* and the *number of neurons* in the *hidden layers* in the defined `StackedAutoEncoders` class.
- You can try *different combinations* of the *activation functions* by manipulating the *objects* related to the *activation functions*.

- We're gonna use some pretty *advanced techniques* in ***PyTorch***.

- ***No. of Epochs:*** The first step is to define a number of EPOCH, it is based on experimenting. We're gonna train our ***StackedAutoEncoders*** on ***200*** EPOCHs.

```
nb_epoch = 200
```

- ***Outer-loop, train_loss:*** Second step is to make a ***nested-for-loop***, **outer for loop** runs over the **epochs** and the *inner for-loop* will loop over all our *observations* in each EPOCH (i.e. all our users because each observation corresponds to the ratings of a user).
  - Before we introduce the second loop, we need to do is initialize ***train_loss*** and counter ***s***. (As we did in RBM. We also will make the ***test_loss*** for the loss of the ***test set***.)
  - ***Counter s:*** Counter ***s*** will count the number of users that *rated* at least *one movie*. The purpose is to optimize the memory, we won't do the computations for the *users* who *didn't rate any movies*.
    - To do this we need to **keep track** of the **number** of **users** who *rated at least one movie*. Hence, we use **s**. We use ***0.*** or ***0.0*** to make is *floating-point-data-type*.
    - Since we're gonna use ***s*** to compute the ***RMSE*** in the end, therefore ***floating-point*** is used to make sure that ***int-type*** and ***float-type*** do not cause any issue.
    - Since the *root-mean-squared error* is a *float*, then all the *elements* to compute the root-mean-squared error should be a *float*. It's not compulsory, but that's just to avoid a warning.

- The next step is to start the *second loop*. It will loop over *all the observations*, i.e *all the users*. So this ***inner-for loop*** will *introduce* all the *actions* that will take *place* in a single *EPOCH*.
  - ***Inside 2nd for loop:*** We're gonna get our predicted rating using our ***StackedAutoEncoders*** class with ***sae*** object that we created earlier.
    - We're gonna compute the *loss error* on one EPOCH. To observe how the *training evolving*. Because we want to *optimize* this *loss*, so we keep track if it's *decreasing* over the EPOCHs.
    - We will apply our optimizer `optim.RMSprop` to apply ***Stochastic Gradient Descent*** to *update* the *weights* and lead to the *convergence*.

- The loop variable we call ***id_user***. We set the ***range(nb_users)***, without upper/lower bound, it just *loop for-each user*.
  - Because actually it needs to be the *indexes of the observations* of our *training set* and the indexes of the observations of our training set don't go from ***1*** to ***943***. They go from ***0*** to ***942***. ***range(nb_users)*** will make this ***range*** from ***0*** to ***942***.

```
for id_user in range(nb_users):
```

  - Now we apply the *input vector*, that is, the *input vector* of *features* that contains *all* the *ratings* of all the *movies* given by this *particular user* inside the loop.

```
input = Variable(train_set_tensor[id_user]).unsqueeze(0)
```

    - ***train_set_tensor[id_user]*** will get the ratings for all movies that are given by ***id_user***. ***id_user*** is the user with which we are dealing right now in the **loop**.
    - But ***train_set_tensor[id_user]*** is a ***vector*** and a *network* in *PyTorch* or even on ***Keras*** can not accept a ***single vector*** of ***one dimension*** they accepts a ***Batch of input vectors***.

      For example, our ***forward()*** function will not take a ***simple vectors*** of ***one dimension*** as ***input***. So we need to add an ***extra fake dimension*** which will correspond to a ***batch*** (we also did it in ***RBM***, but in different way).

Also remember, we did it already in ***CNN***, when we used the ***predict*** function to ***predict*** if the ***image*** contains a ***cat*** or a ***dog***. we had to add ***one dimension*** and that ***additional dimension*** was ***for*** the ***batch***. We put our input image of a ***cat*** or a ***dog*** into a ***batch*** and that added ***one dimension*** that then would make the whole thing accepted by the predict method.

- So we'll create a ***batch***. This ***batch*** will ***contain one vector***, but we will be into a ***new dimension*** corresponding to the ***batches***.
  - We now have to use the ***Variable module*** that we imported at the beginning from ***torch.autograd***. It's just a **PyTorch** technique.
    i. We put ***train_set_tensor[id_user]*** inside ***Variable()***.
    ii. To create this additional dimension, we use the function ***unsqueeze()*** on ***Variable()***, like this.
    iii. To specify the ***index*** of this ***new dimension***. We use ***unsqueeze(0)***. Because we are gonna put this new dimension in first position (as we did in ***Keras***). And so this dimension will have ***index zero***.

This will create a batch of a single input vector.

```
# ---- Training the SAE model ----
nb_epoch = 200
for epoch in range(1, nb_epoch+1):
    train_loss = 0
    s = 0.0
    for id_user in range(nb_users):
        input = Variable(train_set_tensor[id_user]).unsqueeze(0)
```

NOTE: The ***batch*** can have ***several input vectors***. Remember we called this ***Batch Learning***.

But ***here*** we're gonna do ***Online Learning***. That means that ***we're going to update*** the ***weights*** after ***each observation*** going to the ***network***. And therefore, we are creating a ***batch*** of ***one input vector***. But we have to create this batch, otherwise it won't work.

15.3.6 Train the SAE model : target-vector

Target-vector: As we did in ***RBM***, we need a ***target vector***, which is a ***copy*** of the ***input vector***. Since we're going to ***modify*** the ***input***, and ***target*** remain the ***same***, so that we can ***compare*** the ***modified input*** and the original data in ***target***.

We use ***clone()*** function to make the copy of original-input to ***target***.

Filter the user: In this step we're gonna introduce an ***if condition*** to find the ***users*** who rated ***at least one movie***. So if an observation ***contains only zeros***, which means that the user ***didn't rate*** any ***movies***, then we ***won't care*** of this ***observation***.

```
# ---- Training the SAE model ----
nb_epoch = 200
for epoch in range(1, nb_epoch+1):
    train_loss = 0
    s = 0.0
    for id_user in range(nb_users):
        input = Variable(train_set_tensor[id_user]).unsqueeze(0)
        target = input.clone()
        if torch.sum(target.data > 0) > 0:
            output = sae(input)
```

Here ***target.data*** will take all the values of ***target***, ***target.data*** will be all the ratings of this user here at the loop right now, it's just all the ratings.

target.data > 0, consider all the ratings that are ***larger than zero***. We wanna check if `torch.sum(target.data > 0)` is larger than zero.

And if that's the case, `torch.sum(target.data > 0) > 0` means that the observation contains ***at least one rating*** that is ***not zero***. In that way we consider the users that ***rated at least one movie***.

❒ ***Output:*** To get our vector of predicted ratings, i.e. our output at the very right of the network. We're gonna introduce a new variable named ***output***.

☛ We have to use our ***sae*** object. Because this object is an object of the ***StackedAutoEncoders*** class. In this ***StackedAutoEncoders*** class, the action of ***forwarding*** the ***input vector*** into the ***network*** takes place.

☛ ***forward()*** function returns the output of the network, i.e. the ***vector*** of ***predicted ratings***. So since this ***forward()*** function is part of the ***StackedAutoEncoders*** class, and we did use ***variable-arguments*** in `def __init__(self, ):`, we can use a parameter with ***sae*** object i.e ***sae(x)***, this ***x*** will then passed to ***forward(x)***, and ***forward()*** function will be applied and will **return modified *x***.

☛ In the ***forward()*** function ***encodings*** and ***decoding*** will take place with the ***input***. And this will return eventually to the ***vector*** of ***predicted ratings***.

```
output = sae(input)
```

☛ ***For simplicity we rewrite the if condition:***

```
for epoch in range(1, nb_epoch+1):
    train_loss = 0
    s = 0.0
    for id_user in range(nb_users):
        input = Variable(train_set_tensor[id_user]).unsqueeze(0)
        target = input.clone()

        non_zero_ratings = torch.sum(target.data > 0)
        if non_zero_ratings > 0:
            output = sae(input)
```

15.3.7 Train the SAE model : loss & mean_corrector

Now we have our **Real Ratings** in ***target*** and our **Predicted Ratings** in ***output***. In this step we optimize the memory and the computations.

❒ When we apply ***Stochastic Gradient Descent***, we make sure that the gradient is computed only with respect to the ***input*** and not the ***target-vector***.

```
        non_zero_ratings = torch.sum(target.data > 0)
        if non_zero_ratings > 0:
            output = sae(input)
            target.require_grad = False
            output[target == 0] = 0
            loss = criterion(output, target)
            mean_corrector = nb_movies/float(non_zero_ratings + 1e-10)
```

☞ `"target.require_grad = False"` Here ***require_grad*** will make sure that we ***don't compute*** the ***gradient*** with respect to the ***target*** and that will save a lot of computations and that optimizes the code.

☞ `"output[target == 0] = 0"` Considers ***only non-zero values***. We don't wanna deal with the ***movies*** that the user didn't rate, where the ***ratings*** are ***equal to zero***, but that is only for the ***output*** vector.

➢ We take the values of our output vector, those ratings that are ***0*** in the ***target***-vector, and we reset those to 0.

➢ We're just taking the same indexes of the ***ratings = 0*** in the ***target*** vector (original-rating). And for these indexes of the output vector, we will set the values corresponding to these indexes to zero. (Notice, we did it in previous chapter for our ***RBM*** model).

[The ratings which were 0 at the start are changed after encoding & decoding. Se here we are resetting them to 0 by using ***target-vector***.]

➢ The reason is, we don't want to ***update*** the ***ratings*** that are ***initially 0***, not-rated by the user. So that these they ***won't have impact*** on the ***updates*** of the different ***weights*** right ***after measuring*** the ***error***.

- After we've measured the ***error***, the ***weights*** will be ***updated*** by the ***RMSprop Optimizer*** and updating these ***weights*** require some computations and in these computations, these ***0-ratings*** here don't count.
- So, after updating, even if they're ***not equal to zero***, they not being ***counted***, and so, to save up ***some memory***, again, we set them to ***zero***.

☞ `"loss = criterion(output, target) "` Computes the loss error using our ***criterion*** object. We inputted two arguments ***target*** - the vector of real ratings and ***output*** - the vector of predicted ratings. *First argument* is ***output***, *second argument* is ***target***.

☞ `"mean_corrector = nb_movies/float(non_zero_ratings + 1e-10) "` It's just a ***ratio*** where ***nb_movies*** as numerator and ***non_zero_ratings*** as denominator.

➢ Notice we added a small number `1e-10` to the *denominator* to avoid *"undefined"* mathematical error, so that *denominator* always be a ***non-zero number***. Because ***non_zero_ratings*** could be ***0*** in case the *user not-rated any movies*.

➢ ***non_zero_ratings*** means we're considering all the movies that have *non-zero ratings* by the *current user* in the *loop*.

✋ Why do we need to create this ***mean_corrector***?

☝ This actually represents the ***average of the error***, but by only considering the ***movies*** that were ***rated***.

☝ We need to do this ***because*** we only considered here the ***movies*** that have *non-zero ratings*. When we will compute ***mean***, this ***mean*** has to be *computed* only on the ***movies*** that we consider. That is, the ***movies*** that got *non-zero ratings*.

This ***mean_corrector*** variable is just to adapt to this consideration of the movies that got non-zero ratings. We need to do this because this will then be mathematically *relevant* to compute the *mean* of the *errors*.

15.3.8 Train the SAE model : backward, train-loss, train

❒ **backward():** Now we're gonna call the ***backward()*** method for the ***loss***. This method comes from ***criterion*** object. By calling the ***backward*** method we just tell in *which direction* we need to *update* the *different weights*.

☞ To specify *increase* or *decrease* the *weight*. We still inside the ***if***-condition:

```
loss.backward()
```

❒ ***train_loss:*** We now compute the RMSE to update the ***train_loss***.

☛ ***loss.data[0]*** gonna get the part of this loss object that contains the error. We excess to the ***data*** in the ***loss*** object and then we need to take the *index* of the *data* that contains this *train loss* which is ***0***.

```
            # train_loss += np.sqrt(loss.data[0] * mean_corrector)
            train_loss += np.sqrt(loss.data * mean_corrector)
```

☛ However in **newer version of PyTorch** we use ***loss.data*** without index. To avoid runtime error:

```
IndexError: invalid index of a 0-dim tensor. Use `tensor.item()` in Python or `tensor.item<T>()` in C++ to convert a
0-dim tensor to a number

That's because in PyTorch>=0.5, the index of 0-dim tensor is invalid. The master branch is designed for PyTorch
0.4.1, loss_val.data[0] works well.

Try to change

        total_loss += loss_val.data[0]
        loss_values = [v.data[0] for v in losses]

to

        total_loss += loss_val.data
        loss_values = [v.data for v in losses]


might fix the problem.
```

☛ We multiply ***loss.data*** with our adjustment factor ***mean_corrector***. We're just adjusting ***loss*** with ***mean_corrector*** factor to compute the relevant ***mean***.

☛ Since this ***loss.data*** is the ***Squared*** error and we want to get the ***Square-Root*** of this error. i.e one degree loss, we will take the root of this loss.data.mean_corrector.

```
np.sqrt(loss.data * mean_corrector)
```

❒ We also ***increment*** the ***counter s*** here that corresponds to the ***number of users*** who rated at ***least one movie*** (***s*** just filtering the users who gives rating from the total users.). We used 1.0 because we want ***s*** to be a float-number.

```
s += 1.0
```

❒ Now we apply the ***optimizer*** that we defined before, which is `optim.RMSprop()`. We apply it here to ***update*** the ***weight***.

☞ We need to use ***step()*** method of the ***optimizer*** object from the ***RMSprop*** to apply the ***optimizer*** to update the ***weights***.

```
optimizer.step()
```

☞ ***optimizer.step()*** was the last step of both the ***if*** condition and the inner ***for loop***. so we're done dealing with our observation, taking care of all the actions that happened into the network.

✋ ***The difference between backward and optimizer:***

☝ ***backward()*** decides the ***direction*** to which-way the ***weight*** will be ***updated*** (i.e. increased or decreased).

☝ And `optimizer.step()` decides ***intensity*** of the ***updates***. That is, the ***amount*** by which the ***weights*** will be ***updated***.

So, ***backward()*** decides the **direction** of weight-update. and `optimizer.step()` decides the **intensity/amount** weight-update.

❒ ***Print:*** Finally, we print the ***epoch*** and ***train loss***, inside the ***outer for-loop***. We first calculate the ***normalized train-loss*** by dividing ***train_loss*** by ***s***.

```
train_loss_normalized = train_loss/s
print(f"Epoch : {epoch}, Loss = {train_loss_normalized}")
```

☝ Our model is now ready to train!!! If you wanna build a ***different AutoEncoder model***, just *change the architecture* in the ***StackedAutoEncoders*** class ***definition***. You can try some other ***combinations*** of ***activation functions*** there.

☝ To tune the model, also try different number of ***epochs***, with different ***activation functions***.

```
# ---- Training the SAE model ----
nb_epoch = 200
for epoch in range(1, nb_epoch+1):
    train_loss = 0
    s = 0.0
    for id_user in range(nb_users):
        input = Variable(train_set_tensor[id_user]).unsqueeze(0)
        target = input.clone()

        non_zero_ratings = torch.sum(target.data > 0)
        if non_zero_ratings > 0:
            output = sae(input)
            target.require_grad = False
            output[target == 0] = 0
            loss = criterion(output, target)
            mean_corrector = nb_movies/float(non_zero_ratings + 1e-10)

            loss.backward()
            train_loss += np.sqrt((loss.data)*mean_corrector)
            s += 1.0
            optimizer.step()

    train_loss_normalized = train_loss/s
    print(f"Epoch : {epoch}, Loss = {train_loss_normalized}")
```

👽 Before we execute our code, let's set some ***expectations*** of what we would like to get.

➴ First thing is, the ***loss*** represents the ***average of the differences*** between the ***real rating*** and the ***predicted rating***, on the ***training set***.

➢ Which means that, for example, if we get a ***loss*** of ***1*** at the **last epoch**, that will mean that the ***average difference*** between the ***real ratings*** of the *movies* by the *users* and the ***predicted ratings*** will be one.

- And that's not too bad because it means: when we predict if a user is going to like a movie, on average we will ***make*** an ***error of 1 star*** out of ***5***.
  - We were hoping to get a loss that would be ***less*** than ***one star*** so that the **average difference** between the ***real rating*** and the ***predicted rating*** is ***less than 1***. Therefore, on average, we will make better predictions of whether a user is going to like a movie or not.
  - We are at least expecting to get a ***loss*** of less than ***1*** *(loss < 1 star)*,

Second thing is we're now calculating the ***loss*** on the ***training sets***, also we need to calculate the ***test_loss*** on the ***test set***. Then we can see is there a ***high over-fitting*** or not.

Result: at the ***last epoch***, we might expect a ***final loss*** of ***0.91***,

- So that's not ***too bad***. Besides, we were ***training*** on ***100,000*** ratings and you will definitely get a ***better loss error*** if you train this on more ratings. For example ***1 – million dataset***. Our code is ***optimized*** so that the training doesn't take ***too much time***.
- Now let's hope that we will get around the same error on the test set.

Increasing epochs may lead to Overfitting: Of course, by having ***more epochs*** you can get a ***lower loss*** on the ***training set***, but then you might end up with a ***larger difference*** of the ***loss*** between the ***training set*** and the ***test set*** which leads to ***high-overfitting***, and we want to avoid that, so ***200*** **epoch** is enough for our current dataset.

NotImplementedError: Module [StackedAutoEncoders] is missing the required "forward " function.

This error is shown if we misspell the ***forward()*** function in our SAE-architecture the ***StackedAutoEncoders*** class.

All code at once (train only)

```
# ---------- AE : Recommender. SAE Stacked-Auto-Encoder ---------------

# Importing the libraries
from turtle import clone
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable


# ---------- importing the dataset -----------

# preparing the training set and test set
training_set = pd.read_csv("./movie_lens_100k/u1.base", delimiter="\t")
train_set = np.array(training_set, dtype="int")

ts_set = pd.read_csv("./movie_lens_100k/u1.test", delimiter="\t")
test_set = np.array(ts_set, dtype="int")


# Getting the number of Users and Movies
nb_users = int(max(max(train_set[:, 0]), max(test_set[:, 0])))
nb_movies = int(max(max(train_set[:, 1]), max(test_set[:, 1])))


# converting the data into an array with users in lines and movies in column.
def conVert(data):
    new_data = []
    for id_user in range(1, nb_users + 1):
        # use "data[:, 0] == id_user" as condition over movie column "data[:, 1]"
        id_movies = data[:, 1][data[:, 0]== id_user]    # returns a list
```

```
        # use "data[:, 0] == id_user" as condition over ratins column "data[:, 2]"
        id_ratings = data[:, 2][data[:, 0]== id_user]

        # vector of zeros
        ratings = np.zeros(nb_movies)
        ratings[id_movies - 1] = id_ratings

        new_data.append(list(ratings))

    return new_data

trn_set_cnvt = conVert(train_set)
tst_set_cnvt = conVert(test_set)


# Converting the data into Torch Tensosrs. Following are the Tensors of ratings
train_set_tensor = torch.FloatTensor(trn_set_cnvt)
test_set_tensor = torch.FloatTensor(tst_set_cnvt)




# -------- Creating the architecture of the Neural Netwark for SAE --------
class StackedAutoEncoders(nn.Module):
    def __init__(self, ):
        super(StackedAutoEncoders, self).__init__()
        self.fc1 = nn.Linear(nb_movies, 20)
        self.fc2 = nn.Linear(20, 10)
        self.fc3 = nn.Linear(10, 20)
        self.fc4 = nn.Linear(20, nb_movies)
        self.activation = nn.Sigmoid()

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.activation(self.fc3(x))
        x = self.fc4(x)
        return x

sae = StackedAutoEncoders()
criterion = nn.MSELoss()
optimizer = optim.RMSprop(sae.parameters(), lr = 0.01, weight_decay= 0.5)


# ---- Training the SAE model ----
nb_epoch = 200
for epoch in range(1, nb_epoch+1):
    train_loss = 0
    s = 0.0
    for id_user in range(nb_users):
        input = Variable(train_set_tensor[id_user]).unsqueeze(0)
        target = input.clone()

        non_zero_ratings = torch.sum(target.data > 0)
        if non_zero_ratings > 0:
            output = sae(input)
            target.require_grad = False
            output[target == 0] = 0
            loss = criterion(output, target)
            mean_corrector = nb_movies/float(non_zero_ratings + 1e-10)

            loss.backward()
            train_loss += np.sqrt((loss.data)*mean_corrector)
            s += 1.0
            optimizer.step()

    train_loss_normalized = train_loss/s
    print(f"Epoch : {epoch}, Loss = {train_loss_normalized}")

# python prctc_SAE.py
```

15.3.9 Test the model : use test-set to calculate "test_loss"

Here we try to compute the ***test-set loss*** and we hope we can make it below 1 (1 star error of ratings).

❒ Similar to our ***RBM*** model, we ***don't need*** a ***nested-for-loop***, because we are ***not training*** the ***model***.

☞ We don't need ***200 epochs*** to measure the ***test-set*** performance to measure the ***test-set loss***. We of course need only ***one epoch***. Because we're gonna measure the ***global loss***, one time. We only need ***inner for loop*** here because this for loop loops over ***all*** the ***users*** of the data set.

❒ For the **target-*vector,*** we now choose **test-*set*** and for the **input-*vector*** we use the **train-*set***. Because

☞ ***test_loss*** compute the ***loss*** on ***test-set***. We initialized it to ***0***.

☞ Here ***cnt*** is the ***counter*** it is the number of ***users*** that ***rated at least one movie*** in the ***test-set***.

```
test_loss = 0
cnt = 0.0
for u_id in range(nb_users):
    v = Variable(train_set_tensor[u_id]).unsqueeze(0)     # input vector from training-set
    vt = Variable(test_set_tensor[u_id]).unsqueeze(0)     # target vector from "test-set"
```

☞ The ***input*** corresponding to the user is of course, all the ratings of movies this user watched. We need the ***training set*** in ***input-vector***. Because:

➢ We put this ***input vector*** into the ***AE-network***, then the ***AE*** will look at the ***ratings*** of the ***movies*** and especially the ***positive ratings***, and based on these ratings, it will ***predict*** the ***ratings*** of the ***movies*** that the user ***hasn't watched*** yet (i.e. ratings that are not present in training-set). That's the ***prediction***.

➢ Here the point of using the ***test-set*** as ***target-vector*** is that, it contains the ***real-ratings*** of the movies that were ***not present*** in ***training-set*** (we used AE to generate those), so that we can ***compare predicted & real ratings*** and measure the ***test_loss***.

➢ ***test_loss*** will indicate how our model performs on new data.

☯ For example, if in our ***input vector*** our user gave ***five star*** ratings to all the ***action movies*** he watched, then when we feed this input vector into the network, well, the ***neurons*** corresponding to the ***specific features*** related to ***action movies***, will be ***activated*** with a ***large weight*** to ***predict*** high ratings for the other ***action*** movies that the user ***hasn't watched yet***.

❖ And then what we'll do, is we will compare this ***predicted*** ratings to the ratings of the ***test-set*** because the ***test set*** contains these ratings that were not ***part*** of the ***training set***. That is, these action movies that the user ***hasn't watched yet***, in the ***training set***.

```
test_loss = 0
cnt = 0.0
for u_id in range(nb_users):
    v = Variable(train_set_tensor[u_id]).unsqueeze(0)     # input vector from training-set
    vt = Variable(test_set_tensor[u_id]).unsqueeze(0)     # target vector from "test-set"

    non_zero_ratings_tst = torch.sum(vt.data > 0)
    if non_zero_ratings_tst > 0:
        tst_output = sae(v)
        vt.require_grad = False

        tst_output[vt == 0] = 0
        loss_tst = criterion(tst_output, vt)

        mean_corrector_tst = nb_movies/float(non_zero_ratings_tst + 1e-10)
        test_loss += np.sqrt((loss_tst.data)*mean_corrector_tst)
        cnt += 1.0

eval_loSS = test_loss/cnt
print(f"Evaluation or Test loss = {eval_loSS}")
```

❒ The predictions happen right here: `tst_output = sae(v)` . *The SAE model* ***sae*** *just making* ***one step forward*** *after training.*

☞ Our ***forward*** function that returns the ***vector*** of ***predicted ratings*** and therefore by ***calling*** our object on the ***input*** here, we will get our ***vector*** of ***predicted ratings*** for the movies that the user ***hasn't watched yet***, and this will go into ***tst_output***.

☞ We use " `vt.require_grad = False` " to override the ***computations*** of the ***gradient*** with respect to the ***target*** because we don't need them.

✋ We don't use `loss.backward()` or `optimizer.step()` either. Because those are for update weight and for training purpose.

☞ We use `tst_output[vt == 0] = 0` to avoid the movies that are ***not rated*** by the user in the ***test-set***.

❒ To calculate ***loss***, between ***real-ratings*** in ***test-set*** and ***predicted-ratings*** from ***train-set***:

```
loss_tst = criterion(tst_output, vt)
```

❒ We also need the ***mean_cottector*** for test set:

```
mean_corrector_tst = nb_movies/float(non_zero_ratings_tst + 1e-10)
```

❒ Finally we compute the ***test_loss*** and increment our counter ***cnt***:

```
test_loss += np.sqrt((loss_tst.data)*mean_corrector_tst)
cnt += 1.0
```

❒ After the loop, we calculate the ***normalized-loss*** and print the result:

```
eval_loSS = test_loss/cnt
print(f"Evaluation or Test loss = {eval_loSS}")
```

Result: Our test-loss is ***0.953***. That's the error between the real & predicted ratings. On average, our model is going to ***predict*** a ***rating*** that will be ***different*** from the ***real rating*** by less than ***one star***. If we manage to do this, that means that our recommended system will be pretty powerful because it can predict for new movies that you are going to like it or not.

We built a robust recommended system. The ***test loss*** is ***0.95*** **stars**, that is less than one star. So for example if you're applying this recommended system for the movie you're gonna watch tonight, and let's say that after watching the movie you give the rating ***four stars***, then this ***recommended system*** would ***predict*** that you would give ***between three and five stars*** to this movie.

```
Epoch : 183, Loss = 0.9141837358474731
Epoch : 184, Loss = 0.9146118760108948
Epoch : 185, Loss = 0.913851797580719
Epoch : 186, Loss = 0.9144442677497864
Epoch : 187, Loss = 0.9136586785316467
Epoch : 188, Loss = 0.9141697287559509
Epoch : 189, Loss = 0.9134584665298462
Epoch : 190, Loss = 0.913621187210083
Epoch : 191, Loss = 0.9131093621253967
Epoch : 192, Loss = 0.9135682582855225
Epoch : 193, Loss = 0.9126824736595154
Epoch : 194, Loss = 0.9133999943733215
Epoch : 195, Loss = 0.9126423001289368
Epoch : 196, Loss = 0.9129593372344971
Epoch : 197, Loss = 0.9126214981079102
Epoch : 198, Loss = 0.9125758409500122
Epoch : 199, Loss = 0.9122977256774902
Epoch : 200, Loss = 0.9126603603363037
Evaluation or Test loss = 0.9528855681419373
```

All code at once (practiced)

```
# ---------- AE : Recommender. SAE Stacked-Auto-Encoder --------------

# Importing the libraries
from turtle import clone
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable


# ---------- importing the dataset ----------

# preparing the training set and test set
training_set = pd.read_csv("./movie_lens_100k/u1.base", delimiter="\t")
train_set = np.array(training_set, dtype="int")

ts_set = pd.read_csv("./movie_lens_100k/u1.test", delimiter="\t")
test_set = np.array(ts_set, dtype="int")


# Getting the number of Users and Movies
nb_users = int(max(max(train_set[:, 0]), max(test_set[:, 0])))
```

```
nb_movies = int(max(max(train_set[:, 1]), max(test_set[:, 1])))


# converting the data into an array with users in lines and movies in column.
def conVert(data):
    new_data = []
    for id_user in range(1, nb_users + 1):
        # use "data[:, 0] == id_user" as condition over movie column "data[:, 1]"
        id_movies = data[:, 1][data[:, 0]== id_user]    # returns a list

        # use "data[:, 0] == id_user" as condition over ratins column "data[:, 2]"
        id_ratings = data[:, 2][data[:, 0]== id_user]

        # vector of zeros
        ratings = np.zeros(nb_movies)
        ratings[id_movies - 1] = id_ratings

        new_data.append(list(ratings))

    return new_data

trn_set_cnvt = conVert(train_set)
tst_set_cnvt = conVert(test_set)


# Converting the data into Torch Tensosrs. Following are the Tensors of ratings
train_set_tensor = torch.FloatTensor(trn_set_cnvt)
test_set_tensor = torch.FloatTensor(tst_set_cnvt)




# -------- Creating the architecture of the Neural Netwark for SAE --------
class StackedAutoEncoders(nn.Module):
    def __init__(self, ):
        super(StackedAutoEncoders, self).__init__()
        self.fc1 = nn.Linear(nb_movies, 20)
        self.fc2 = nn.Linear(20, 10)
        self.fc3 = nn.Linear(10, 20)
        self.fc4 = nn.Linear(20, nb_movies)
        self.activation = nn.Sigmoid()

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.activation(self.fc3(x))
        x = self.fc4(x)
        return x

sae = StackedAutoEncoders()
criterion = nn.MSELoss()
optimizer = optim.RMSprop(sae.parameters(), lr = 0.01, weight_decay= 0.5)


# ---- Training the SAE model ----
nb_epoch = 200
for epoch in range(1, nb_epoch+1):
    train_loss = 0
    s = 0.0
    for id_user in range(nb_users):
        input = Variable(train_set_tensor[id_user]).unsqueeze(0)
        target = input.clone()

        non_zero_ratings = torch.sum(target.data > 0)
        if non_zero_ratings > 0:
            output = sae(input)
            target.require_grad = False
            output[target == 0] = 0
            loss = criterion(output, target)
            mean_corrector = nb_movies/float(non_zero_ratings + 1e-10)

            loss.backward()
            train_loss += np.sqrt((loss.data)*mean_corrector)
            s += 1.0
            optimizer.step()
```

```
    train_loss_normalized = train_loss/s
    print(f"Epoch : {epoch}, Loss = {train_loss_normalized}")



# ---- Testing the SAE model on Test-set ----
# Evaluating the SAE on Test-set
test_set_ratings_list = []
pridicted_rating_list = []

test_loss = 0
cnt = 0.0
for u_id in range(nb_users):
    v = Variable(train_set_tensor[u_id]).unsqueeze(0)      # input vector from training-set
    vt = Variable(test_set_tensor[u_id]).unsqueeze(0)      # target vector from "test-set"

    non_zero_ratings_tst = torch.sum(vt.data > 0)
    if non_zero_ratings_tst > 0:
        tst_output = sae(v)
        vt.require_grad = False

        tst_output[vt == 0] = 0     # avoiding unrated movies in test-set
        loss_tst = criterion(tst_output, vt)    # comparing

        mean_corrector_tst = nb_movies/float(non_zero_ratings_tst + 1e-10)
        test_loss += np.sqrt((loss_tst.data)*mean_corrector_tst)
        cnt += 1.0

    # creating list of original & predicted ratings : Tensor to NumPy-array conversion
        """ Now we have to use detach() to convert tensor to Numpy-array
                because our tensor requires "grad" now.
            We did it in RBM already but without detach() """

    original_test_set_ratings = vt.detach().numpy()
    test_set_ratings_list.append(original_test_set_ratings)
    predicted_ratings = tst_output.detach().numpy()
    pridicted_rating_list.append(predicted_ratings)

eval_loSS = test_loss/cnt
print(f"Evaluation or Test loss = {eval_loSS}")



# python prctc_SAE.py
```

☯ We compared the rating in the test-set & predicted-output for user no. 4.



Another version from Github

```
# Stacked AutoEncoders: SAE

# Importing the libraries
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable

# Importing the dataset
# movies = pd.read_csv('ml-1m/movies.dat', sep='::', header=None, engine='python', encoding='latin-1')
# users = pd.read_csv('ml-1m/users.dat', sep='::', header=None, engine='python', encoding='latin-1')
# ratings = pd.read_csv('ml-1m/ratings.dat', sep='::', header=None, engine='python', encoding='latin-1')

# Preparing the training set and the test set
training_set = pd.read_csv('movie_lens_100k/u1.base', delimiter='\t')
training_set = np.array(training_set, dtype='int')
test_set = pd.read_csv('movie_lens_100k/u1.test', delimiter='\t')
test_set = np.array(test_set, dtype='int')

# Getting the number of users and movies
nb_users = int(max(max(training_set[:, 0]), max(test_set[:, 0])))
nb_movies = int(max(max(training_set[:, 1]), max(test_set[:, 1])))


# Converting the data into an array with users in lines and movies in columns
def convert(data):
    new_data = []
    for id_users in range(1, nb_users + 1):
        id_movies = data[:, 1][data[:, 0] == id_users]
        id_ratings = data[:, 2][data[:, 0] == id_users]
        ratings = np.zeros(nb_movies)
        ratings[id_movies - 1] = id_ratings
        new_data.append(list(ratings))
    return new_data


# Array with users in lines and movies in columns
# [user_1, user_2, ..., user_943]
# [[movie_1_rating] [movie_2_rating] ... [movie_1682_rating]]
training_set = convert(training_set)
test_set = convert(test_set)

# Converting the data into Torch tensors
training_set = torch.FloatTensor(training_set)#.cuda()
test_set = torch.FloatTensor(test_set)#.cuda()


# Creating the architecture of the Neural Network
class SAE(nn.Module):
    def __init__(self, ):
        super(SAE, self).__init__()
        self.fc1 = nn.Linear(nb_movies, 20)
        self.fc2 = nn.Linear(20, 10)
        self.fc3 = nn.Linear(10, 20)
        self.fc4 = nn.Linear(20, nb_movies)
        self.activation = nn.Sigmoid()

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.activation(self.fc3(x))
        x = self.fc4(x)
        return x


sae = SAE()
criterion = nn.MSELoss()
optimizer = optim.RMSprop(sae.parameters(), lr=0.01, weight_decay=0.5)
```

```python
# Training the SAE
nb_epoch = 200
for epoch in range(1, nb_epoch + 1):
    train_loss = 0
    s = 0.  # Number of users who rated at least 1 movie
    for id_user in range(nb_users):
        # Creates a 2D array instead of 1D, Pytorch only accept batch of data
        input = Variable(training_set[id_user]).unsqueeze(0)
        target = input.clone()  # Outputs should be the same as inputs
        if torch.sum(target.data) > 0:  # User rated at least 1 movie
            output = sae.forward(input)
            target.require_grad = False  # For code optimization
            output[target == 0] = 0
            loss = criterion(output, target)
            loss.backward()  # Perform backpropagation
            optimizer.step()  # Define the intensity of backward pass

            # 1e-10 so that we never divide by 0
            # mean_corrector corresponds to the average of the error of the rated movies only
            # it's not used in the backprop calculation, just for metrics
            mean_corrector = nb_movies / float(torch.sum(target.data > 0) + 1e-10)
            # train_loss += np.sqrt(loss.data[0] * mean_corrector)
            train_loss += np.sqrt(loss.data * mean_corrector)
            s += 1.  # Increment number of users who rated at least 1 movie
    print('epoch: ' + str(epoch) + ' loss: ' + str(train_loss / s))

# Testing the SAE
test_loss = 0
s = 0.
for id_user in range(nb_users):
    # The training set contains movies that the user has not yet watched
    input = Variable(training_set[id_user]).unsqueeze(0)
    # The test set contains the movies that the user watched
    target = Variable(test_set[id_user])
    if torch.sum(target.data) > 0:
        output = sae.forward(input)
        target.require_grad = False
        # output[target == 0] = 0
        output[(target == 0).unsqueeze(0)]= 0
        loss = criterion(output, target)

        mean_corrector = nb_movies / float(torch.sum(target.data > 0) + 1e-10)
        # test_loss += np.sqrt(loss.data[0] * mean_corrector)
        test_loss += np.sqrt(loss.data * mean_corrector)
        s += 1.
print('test loss: ' + str(test_loss / s))
```

```
""" IndexError: invalid index of a 0-dim tensor. Use `tensor.item()` in Python or `tensor.item<T>()` in C++ to convert
a 0-dim tensor to a number

That's because in PyTorch>=0.5, the index of 0-dim tensor is invalid. The master branch is designed for PyTorch 0.4.1,
loss_val.data[0] works well.

Try to change
                total_loss += loss_val.data[0]
                loss_values = [v.data[0] for v in losses]

to
                total_loss += loss_val.data
                loss_values = [v.data for v in losses]

might fix the problem."""


""" IndexError: The shape of the mask [1682] at index 0 does not match the shape of the indexed tensor [1, 1682] at
index 0.
Change:
                output[target == 0] = 0      # I get error at this line

To:
                output[(target == 0).unsqueeze(0)] = 0

Reason: The torch.Tensor returned by target == 0 is of the shape [1682].
        (target == 0).unsqueeze(0) will convert it to [1, 1682]          """
```

Homework Challenge - Coding Exercise

So far our training and test sets have the following format:

```
Column1: User

Column 2: Movie

Column 3: Rating

Column 4: Timestamp
```

Define a function that will convert this format into a list of horizontal lists, where each horizontal list corresponds to a user and includes all its ratings of the movies. In each list should also be included the movies that the user didn't rate and for these movies, just put a zero. So what you should get in the end is a huge list of 943 horizontal lists (because there are 943 users):

```
List of User 1: [Ratings of all the movies by User 1]

List of User 2: [Ratings of all the movies by User 2]

.....................................................................

List of User 943: [Ratings of all the movies by User 943]
```

Why doing this ? Because we want to create a new structure of data, having the shape of a 2d array where:

```
the rows are the users,
the columns are the movies,
the cells are the ratings.
```

This coding exercise will be excellent practice for you because you will work with four important techniques in Python:

```
functions
lists and arrays
for loops
handling indexes
```

Try to complete this Homework as hard as you can, the more you try, the more you will progress.

The solution is in the next tutorial.

Good luck!

An Introduction to Restricted Boltzmann Machines

Asja Fischer[1,2] and Christian Igel[2]

[1] Institut für Neuroinformatik, Ruhr-Universität Bochum, Germany
[2] Department of Computer Science, University of Copenhagen, Denmark

Abstract. Restricted Boltzmann machines (RBMs) are probabilistic graphical models that can be interpreted as stochastic neural networks. The increase in computational power and the development of faster learning algorithms have made them applicable to relevant machine learning problems. They attracted much attention recently after being proposed as building blocks of multi-layer learning systems called deep belief networks. This tutorial introduces RBMs as undirected graphical models. The basic concepts of graphical models are introduced first, however, basic knowledge in statistics is presumed. Different learning algorithms for RBMs are discussed. As most of them are based on Markov chain Monte Carlo (MCMC) methods, an introduction to Markov chains and the required MCMC techniques is provided.

1 Introduction

Boltzmann machines (BMs) have been introduced as bidirectionally connected networks of stochastic processing units, which can be interpreted as neural network models [1,16]. A BM can be used to learn important aspects of an unknown probability distribution based on samples from this distribution. In general, this learning process is difficult and time-consuming. However, the learning problem can be simplified by imposing restrictions on the network topology, which leads us to *restricted Boltzmann machines* (RBMs, [34]), the topic of this tutorial.

A (restricted) BM is a parameterized generative model representing a probability distribution. Given some observations, the training data, learning a BM means adjusting the BM parameters such that the probability distribution represented by the BM fits the training data as well as possible. Boltzmann machines consist of two types of units, so called visible and hidden neurons, which can be thought of as being arranged in two layers. The visible units constitute the first layer and correspond to the components of an observation (e.g., one visible unit for each pixel of a digital input image). The hidden units model dependencies between the components of observations (e.g., dependencies between pixels in images). They can be viewed as non-linear feature detectors [16].

Boltzmann machines can also be regarded as particular graphical models [22], more precisely undirected graphical models also known as Markov random fields. The embedding of BMs into the framework of probabilistic graphical models provides immediate access to a wealth of theoretical results and well-developed

L. Alvarez et al. (Eds.): CIARP 2012, LNCS 7441, pp. 14–36, 2012.
© Springer-Verlag Berlin Heidelberg 2012

algorithms. Therefore, our tutorial introduces RBMs from this perspective. Computing the likelihood of an undirected model or its gradient for inference is in general computationally intensive, and this also holds for RBMs. Thus, sampling based methods are employed to approximate the likelihood and its gradient. Sampling from an undirected graphical model is in general not straightforward, but for RBMs Markov chain Monte Carlo (MCMC) methods are easily applicable in the form of Gibbs sampling, which will be introduced in this tutorial along with basic concepts of Markov chain theory.

After successful learning, an RBM provides a closed-form representation of the distribution underlying the observations. It can be used to compare the probabilities of (unseen) observations and to sample from the learnt distribution (e.g., to generate image textures [25,21]), in particular from marginal distributions of interest. For example, we can fix some visible units corresponding to a partial observation and sample the remaining visible units for completing the observation (e.g., to solve an image inpainting task [21]).

Boltzmann machines have been proposed in the 1980s [1,34]. Compared to the times when they were first introduced, RBMs can now be applied to more interesting problems due to the increase in computational power and the development of new learning strategies [15]. Restricted Boltzmann machines have received a lot of attention recently after being proposed as building blocks of multi-layer learning architectures called deep belief networks (DBNs, [19,17]). The idea is that the hidden neurons extract relevant features from the observations. These features can serve as input to another RBM. By stacking RBMs in this way, one can learn features from features in the hope of arriving at a high level representation.

It is an important property that single as well as stacked RBMs can be reinterpreted as deterministic feed-forward neural networks. Than they are used as functions from the domain of the observations to the expectations of the latent variables in the top layer. Such a function maps the observations to learnt features, which can, for example, serve as input to a supervised learning system. Further, the neural network corresponding to a trained RBM or DBN can be augmented by an output layer, where units in the new added output layer represent labels corresponding to observations. Then the model corresponds to a standard neural network for classification or regression that can be further trained by standard supervised learning algorithms [31]. It has been argued that this initialization (or unsupervised pretraining) of the feed-forward neural network weights based on a generative model helps to overcome problems observed when training multi-layer neural networks [19].

This introduction to RBMs is meant to supplement existing tutorials, such as the highly recommended review by Bengio [2], by providing more background information on Markov random fields and MCMC methods in Section 2 and Section 3, respectively. However, basic knowledge in statistics is presumed. We put an emphasis on topics that are – based on our experience – sometimes not familiar to people starting with RBMs. Restricted Boltzmann machines will be presented in Section 4. Section 5 will consider RBM training algorithms based

on approximations of the log-likelihood gradient. This includes a discussion of contrastive divergence learning [15] as well as parallel tempering [10]. We will close by hinting at generalizations of RBMs in sections 6 and 7.

2 Graphical Models

Probabilistic graphical models describe probability distributions by mapping conditional dependence and independence properties between random variables on a graph structure (two sets of random variables $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ are conditionally independent given a set of random variables $\boldsymbol{X}_3$ if $p(\boldsymbol{X}_1, \boldsymbol{X}_2|\boldsymbol{X}_3) = p(\boldsymbol{X}_1|\boldsymbol{X}_3)p(\boldsymbol{X}_2|\boldsymbol{X}_3)$). Visualization by graphs can help to develop, understand and motivate probabilistic models. Furthermore, complex computations (e.g., marginalization) can be derived efficiently by using algorithms exploiting the graph structure.

There exist graphical models associated with different kind of graph structures, for example *factor graphs*, *Bayesian networks* associated with directed graphs, and *Markov random fields*, which are also called *Markov networks* or undirected graphical models. This tutorial focuses on the latter. A general introduction to graphical models for machine learning can, for example be found in [5]. The most comprehensive resource on graphical models is the textbook by Koller and Friedman [22].

2.1 Undirected Graphs and Markov Random Fields

First, we will summarize some fundamental concepts from graph theory. An *undirected graph* is a tuple $G = (V, E)$, where V is a finite set of nodes and E is a set of undirected edges. An edge consists out of a pair of nodes from V. If there exists an edge between two nodes v and w, i.e. $\{v, w\} \in E$, w belongs to the neighborhood of v and vice versa. The *neighborhood* $\mathcal{N}_v := \{w \in V : \{w, v\} \in E\}$ of v is defined by the set of nodes connected to v. A *clique* is a subset of V in which all nodes are pairwise connected. A clique is called *maximal* if no node can be added such that the resulting set is still a clique. In the following we will denote by $\mathcal{C}$ the set of all maximal cliques of an undirected graph. We call a sequence of nodes $v_1, v_2, \ldots, v_m \in V$, with $\{v_i, v_{i+1}\} \in E$ for $i = 1, \ldots, m-1$ a *path* from v_1 to v_m. A set $\mathcal{V} \subset V$ *separates* two nodes $v \notin \mathcal{V}$ and $w \notin \mathcal{V}$, if every path from v to w contains a node from $\mathcal{V}$.

We now associate a random variable X_v taking values in a state space Λ_v with each node v in an undirected graph $G = (V, E)$. To ease the notation, we assume $\Lambda_v = \Lambda$ for all $v \in V$. The random variables $\boldsymbol{X} = (X_v)_{v \in V}$ are called *Markov random field* (MRF) if the joint probability distribution p fulfills the *(global) Markov property* w.r.t. the graph: For all disjunct subsets $\mathcal{A}, \mathcal{B}, \mathcal{S} \subset V$, where all nodes in $\mathcal{A}$ and $\mathcal{B}$ are separated by $\mathcal{S}$ the variables $(X_a)_{a \in \mathcal{A}}$ and $(X_b)_{b \in \mathcal{B}}$ are conditional independent given $(X_s)_{s \in \mathcal{S}}$, i.e. for all $\boldsymbol{x} \in \Lambda^{|V|}$ it holds $p\left((x_a)_{a \in \mathcal{A}}|(x_t)_{t \in \mathcal{S} \cup \mathcal{B}}\right) = p\left((x_a)_{a \in \mathcal{A}}|(x_t)_{t \in \mathcal{S}}\right)$. A set of nodes $\mathrm{MB}(v)$ is called the *Markov blanket* of node v, if for any set of nodes $\mathcal{B}$ with $v \notin \mathcal{B}$ we have

$p(v \mid \mathrm{MB}(v), \mathcal{B}) = p(v \mid \mathrm{MB}(v))$. This means that v is conditional independent from any other variables given $\mathrm{MB}(v)$. In an MRF, the Markov blanket $\mathrm{MB}(v)$ is given by the neighborhood $\mathcal{N}_v$ of v, a fact that is also referred to as *local* Markov property.

Since conditional independence of random variables and the factorization properties of the joint probability distribution are closely related, one can ask if there exists a general factorization form of the distributions of MRFs. An answer to this question is given by the *Hammersley-Clifford Theorem* (for rigorous formulations and proofs we refer to [23,22]). The theorem states that a strictly positive distribution p satisfies the Markov property w.r.t. an undirected graph G if and only if p factorizes over G. A distribution is said to factorize about an undirected graph G with maximal cliques $\mathcal{C}$ if there exists a set of non-negative functions $\{\psi_C\}_{C \subset \mathcal{C}}$, called *potential functions*, with

$$\forall \boldsymbol{x}, \hat{\boldsymbol{x}} \in \Lambda^{|V|} : (x_c)_{c \in C} = (\hat{x}_c)_{c \in C} \Rightarrow \psi_C(\boldsymbol{x}) = \psi_C(\hat{\boldsymbol{x}}) \tag{1}$$

and

$$p(\boldsymbol{x}) = \frac{1}{Z} \prod_{C \in \mathcal{C}} \psi_C(\boldsymbol{x}). \tag{2}$$

The normalization constant $Z = \sum_{\boldsymbol{x}} \prod_{C \in \mathcal{C}} \psi_C(\boldsymbol{x}_C)$ is called *partition function*.

If p is strictly positive, the same holds for the potential functions. Thus we can write

$$p(\boldsymbol{x}) = \frac{1}{Z} \prod_{C \in \mathcal{C}} \psi_C(\boldsymbol{x}_C) = \frac{1}{Z} e^{\sum_{C \in \mathcal{C}} \ln \psi_C(\boldsymbol{x}_C)} = \frac{1}{Z} e^{-E(\boldsymbol{x})} \ , \tag{3}$$

where we call $E := \sum_{C \in \mathcal{C}} \ln \psi_C(\boldsymbol{x}_C)$ the *energy function*. Thus, the probability distribution of every MRF can be expressed in the form given by (3), which is also called *Gibbs distribution*.

2.2 Unsupervised Learning

Unsupervised learning means learning (important aspects of) an unknown distribution q based on sample data. This includes finding new representations of data that foster learning, generalization, and communication. If we assume that the structure of the graphical model is known and the energy function belongs to a known family of functions parameterized by $\boldsymbol{\theta}$, unsupervised learning of a data distribution with an MRF means adjusting the parameters $\boldsymbol{\theta}$. We write $p(\boldsymbol{x}|\boldsymbol{\theta})$ when we want to emphasize the dependency of a distribution on its parameters.

We consider training data $S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\ell\}$. The data samples are assumed to be independent and identically distributed (i.i.d.). That is, they are drawn independently from some unknown distribution q. A standard way of estimating the parameters of a statistical model is maximum-likelihood estimation. Applied to MRFs, this corresponds to finding the MRF parameters that maximize the probability of S under the MRF distribution, i.e. training corresponds to finding the parameters $\boldsymbol{\theta}$ that maximize the likelihood given the training data. The

likelihood $\mathcal{L} : \Theta \to \mathbb{R}$ of an MRF given the data set S maps parameters $\boldsymbol{\theta}$ from a parameter space Θ to $\mathcal{L}(\boldsymbol{\theta} \,|\, S) = \prod_{i=1}^{\ell} p(\boldsymbol{x}_i|\boldsymbol{\theta})$. Maximizing the likelihood is the same as maximizing the log-likelihood given by

$$\ln \mathcal{L}(\boldsymbol{\theta} \,|\, S) = \ln \prod_{i=1}^{\ell} p(\boldsymbol{x}_i|\boldsymbol{\theta}) = \sum_{i=1}^{\ell} \ln p(\boldsymbol{x}_i|\boldsymbol{\theta}) \ . \tag{4}$$

For the Gibbs distribution of an MRF it is in general not possible to find the maximum likelihood parameters analytically. Thus, numerical approximation methods have to be used, for example gradient ascent which is described below.

Maximizing the likelihood corresponds to minimizing the distance between the unknown distribution q underlying S and the distribution p of the MRF in terms of the *Kullback-Leibler-divergence* (KL-divergence), which for a finite state space Ω is given by:

$$\mathrm{KL}(q||p) = \sum_{\boldsymbol{x} \in \Omega} q(\boldsymbol{x}) \ln \frac{q(\boldsymbol{x})}{p(\boldsymbol{x})} = \sum_{\boldsymbol{x} \in \Omega} q(\boldsymbol{x}) \ln q(\boldsymbol{x}) - \sum_{\boldsymbol{x} \in \Omega} q(\boldsymbol{x}) \ln p(\boldsymbol{x}) \tag{5}$$

The KL-divergence is a (non-symmetric) measure of the difference between two distributions. It is always positive and zero if and only if the distributions are the same. As becomes clear by equation (5) the KL-divergence can be expressed as the difference between the entropy of q and a second term. Only the latter depends on the parameters subject to optimization. Approximating the expectation over q in this term by the training samples from q results in the log-likelihood. Therefore, maximizing the log-likelihood corresponds to minimizing the KL-divergence.

Optimization by Gradient Ascent. If it is not possible to find parameters maximizing the likelihood analytically, the usual way to find them is gradient ascent on the log-likelihood. This corresponds to iteratively updating the parameters $\boldsymbol{\theta}^{(t)}$ to $\boldsymbol{\theta}^{(t+1)}$ based on the gradient of the log-likelihood. Let us consider the following update rule:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \underbrace{\eta \frac{\partial}{\partial \boldsymbol{\theta}^{(t)}} \left(\sum_{i=1}^{N} \ln \mathcal{L}(\boldsymbol{\theta}^{(t)}|\boldsymbol{x}_i) \right) - \lambda \boldsymbol{\theta}^{(t)} + \nu \Delta \boldsymbol{\theta}^{(t-1)}}_{:= \Delta \boldsymbol{\theta}^{(t)}} \tag{6}$$

If the constants $\lambda \in \mathbb{R}_0^+$ and $\nu \in \mathbb{R}_0^+$ are set to zero, we have vanilla gradient ascent. The constant $\eta \in \mathbb{R}^+$ is the learning rate. As we will see later, it can be desirable to strive for models with weights having small absolute values. To achieve this, we can optimize an objective function consisting of the log-likelihood minus half of the norm of the parameters $\|\theta\|^2/2$ weighted by λ. This method called *weight decay* penalizes weights with large magnitude. It leads to the $-\lambda \boldsymbol{\theta}^{(t)}$ term in our update rule (6). In a Bayesian framework, weight decay

can be interpreted as assuming a zero-mean Gaussian prior on the parameters. The update rule can be further extended by a *momentum* term $\Delta\boldsymbol{\theta}^{(t-1)}$, weighted by the parameter ν. Using a momentum term helps against oscillations in the iterative update procedure and can speed-up the learning process as known from feed-forward neural network training [31].

Introducing Latent Variables. Suppose we want to model an m-dimensional unknown probability distribution q (e.g., each component of a sample corresponds to one of m pixels of an image). Typically, not all variables $\boldsymbol{X} = (X_v)_{v \in V}$ in an MRF need to correspond to some observed component, and the number of nodes is larger than m. We split $\boldsymbol{X}$ into *visible* (or *observed*) variables $\boldsymbol{V} = (V_1, \ldots, V_m)$ corresponding to the components of the observations and *latent* (or *hidden*) variables $\boldsymbol{H} = (H_1, \ldots, H_n)$ given by the remaining $n = |V| - m$ variables. Using latent variables allows to describe complex distributions over the visible variables by means of simple (conditional) distributions. In this case the Gibbs distribution of an MRF describes the joint probability distribution of $(\boldsymbol{V}, \boldsymbol{H})$ and one is usually interested in the marginal distribution of $\boldsymbol{V}$ which is given by

$$p(\boldsymbol{v}) = \sum_{\boldsymbol{h}} p(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \ , \tag{7}$$

where $Z = \sum_{\boldsymbol{v}, \boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})}$. While the visible variables correspond to the components of an observation, the latent variables introduce dependencies between the visible variables (e.g., between pixels of an input image).

Log-Likelihood Gradient of MRFs with Latent Variables. Restricted Boltzmann machines are MRFs with hidden variables and RBM learning algorithms are based on gradient ascent on the log-likelihood. For a model of the form (7) with parameters $\boldsymbol{\theta}$, the log-likelihood given a single training example $\boldsymbol{v}$ is

$$\ln \mathcal{L}(\boldsymbol{\theta} \,|\, \boldsymbol{v}) = \ln p(\boldsymbol{v} \,|\, \boldsymbol{\theta}) = \ln \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})} = \ln \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})} - \ln \sum_{\boldsymbol{v}, \boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \tag{8}$$

and for the gradient we get:

$$\begin{aligned} \frac{\partial \ln \mathcal{L}(\boldsymbol{\theta} \,|\, \boldsymbol{v})}{\partial \boldsymbol{\theta}} &= \frac{\partial}{\partial \boldsymbol{\theta}} \left(\ln \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \right) - \frac{\partial}{\partial \boldsymbol{\theta}} \left(\ln \sum_{\boldsymbol{v}, \boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \right) \\ &= -\frac{1}{\sum\limits_{\boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})}} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial \boldsymbol{\theta}} + \frac{1}{\sum\limits_{\boldsymbol{v}, \boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})}} \sum_{\boldsymbol{v}, \boldsymbol{h}} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial \boldsymbol{\theta}} \\ &= -\sum_{\boldsymbol{h}} p(\boldsymbol{h} \,|\, \boldsymbol{v}) \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial \boldsymbol{\theta}} + \sum_{\boldsymbol{v}, \boldsymbol{h}} p(\boldsymbol{v}, \boldsymbol{h}) \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial \boldsymbol{\theta}} \end{aligned} \tag{9}$$

In the last step we used that the conditional probability can be written in the following way:

$$p(\boldsymbol{h} \,|\, \boldsymbol{v}) = \frac{p(\boldsymbol{v}, \boldsymbol{h})}{p(\boldsymbol{v})} = \frac{\frac{1}{Z} e^{-E(\boldsymbol{v},\boldsymbol{h})}}{\frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})}} = \frac{e^{-E(\boldsymbol{v},\boldsymbol{h})}}{\sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})}} \tag{10}$$

Note that the last expression of equality (9) is the difference of two expectations: the expected values of the energy function under the model distribution and under the conditional distribution of the hidden variables given the training example. Directly calculating this sums, which run over all values of the respective variables, leads to a computational complexity which is in general exponential in the number of variables of the MRF. To avoid this computational burden, the expectations can be approximated by samples drawn from the corresponding distributions based on MCMC techniques.

3 Markov Chains and Markov Chain Monte Carlo Techniques

Markov chains play an important role in RBM training because they provide a method to draw samples from 'complex' probability distributions like the Gibbs distribution of an MRF. This section will serve as an introduction to some fundamental concepts of Markov chain theory. A detailed introduction can be found, for example, in [6] and the aforementioned textbooks [5,22]. The section will describe Gibbs sampling as an MCMC technique often used for MRF training and in particular for training RBMs.

3.1 Definition of a Markov Chain and Convergence to Stationarity

A *Markov chain* is a time discrete stochastic process for which the *Markov property* holds, that is, a family of random variables $X = \{X^{(k)} | k \in \mathbb{N}_0\}$ which take values in a (in the following considerations finite) set Ω and for which $\forall k \geq 0$ and $\forall j, i, i_0, \ldots, i_{k-1} \in \Omega$ it holds

$$p_{ij}^{(k)} := P\left(X^{(k+1)} = j | X^{(k)} = i, X^{(k-1)} = i_{k-1}, \ldots, X^{(0)} = i_0\right) \tag{11}$$

$$= P\left(X^{(k+1)} = j | X^{(k)} = i\right) \ . \tag{12}$$

This means that the next state of the system depends only on the current state and not on the sequence of events that preceded it. If for all $k \geq 0$ the $p_{ij}^{(k)}$ have the same value p_{ij}, the chain is called *homogeneous* and the matrix $\mathbf{P} = (p_{ij})_{i,j \in \Omega}$ is called *transition matrix* of the homogeneous Markov chain.

If the starting distribution $\mu^{(0)}$ (i.e., the probability distribution of $X^{(0)}$) is given by the probability vector $\boldsymbol{\mu}^{(0)} = (\mu^{(0)}(i))_{i \in \Omega}$, with $\mu^{(0)}(i) = P(X^{(0)} = i)$, the distribution $\boldsymbol{\mu}^{(k)}$ of $X^{(k)}$ is given by $\boldsymbol{\mu}^{(k)\,\mathrm{T}} = \boldsymbol{\mu}^{(0)\,\mathrm{T}} \mathbf{P}^k$.

A distribution π for which it holds $\boldsymbol{\pi}^{\mathrm{T}} = \boldsymbol{\pi}^{\mathrm{T}} \mathbf{P}$ is called *stationary distribution*. If the Markov chain for any time k reaches the stationary distribution $\boldsymbol{\mu}^{(k)} = \boldsymbol{\pi}$ all subsequent states will be distributed accordingly, that is, $\boldsymbol{\mu}^{(k+n)} = \boldsymbol{\pi}$ for

all $n \in \mathbb{N}$. A sufficient (but not necessary) condition for a distribution π to be stationary w.r.t. a Markov chain described by the transition probabilities $p_{ij}, i, j \in \Omega$ is that $\forall i, j \in \Omega$ it holds:

$$\pi(i)p_{ij} = \pi(j)p_{ji} \ . \tag{13}$$

This is called the *detailed balance condition*.

Especially relevant are Markov chains for which it is known that there exists an unique stationary distribution. For finite Ω this is the case if the Markov chain is *irreducible*. A Markov chain is irreducible if one can get from any state in Ω to any other in a finite number of transitions or more formally $\forall i, j \in \Omega$ $\exists k > 0$ with $P(X^{(k)} = j | X^{(0)} = i) > 0$.

A chain is called *aperiodic* if for all $i \in \Omega$ the greatest common divisor of $\{k | P(X^{(k)} = i | X^{(0)} = i) > 0 \wedge k \in \mathbb{N}_0\}$ is 1. One can show that an irreducible and aperiodic Markov chain on a finite state space is guarantied to converge to its stationary distribution (see, e.g., [6]). That is, for an arbitrary starting distribution μ it holds

$$\lim_{k \to \infty} d_V(\boldsymbol{\mu}^T \, \mathrm{P}^k, \boldsymbol{\pi}^T) = 0 \ , \tag{14}$$

where d_V is the *distance of variation*. For two distributions α and β on a finite state space Ω, the distance of variation is defined as

$$d_V(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}|\boldsymbol{\alpha} - \boldsymbol{\beta}| = \frac{1}{2} \sum_{x \in \Omega} |\alpha(x) - \beta(x)| \ . \tag{15}$$

To ease the notation, we allow both row and column probability vectors as arguments of the functions in (15).

Markov chain Monte Carlo methods make use of this convergence theorem for producing samples from certain probability distribution by setting up a Markov chain that converges to the desired distributions. Suppose you want to sample from a distribution q with a finite state space. Then you construct an irreducible and aperiodic Markov chain with stationary distribution $\pi = q$. This is a non-trivial task. If t is large enough, a sample $X^{(t)}$ from the constructed chain is then approximately a sample from π and therefore from q. Gibbs Sampling [13] is such a MCMC method and will be described in the following section.

3.2 Gibbs Sampling

Gibbs Sampling belongs to the class of Metropolis-Hastings algorithms [14]. It is a simple MCMC algorithm for producing samples from the joint probability distribution of multiple random variables. The basic idea is to update each variable subsequently based on its conditional distribution given the state of the others. We will describe it in detail by explaining how Gibbs sampling can be used to simulate the Gibbs distribution of an MRF.

We consider an MRF $\boldsymbol{X} = (X_1, \ldots, X_N)$ w.r.t. a graph $G = (V, E)$, where $V = \{1, \ldots, N\}$ for the sake of clearness of notation. The random variables X_i, $i \in V$ take values in a finite set Λ and $\pi(\boldsymbol{x}) = \frac{1}{Z} e^{-\mathcal{E}(\boldsymbol{x})}$ is the joint probability

distribution of $\boldsymbol{X}$. Furthermore, if we assume that the MRF changes its state during time, we can consider $X = \{\boldsymbol{X}^{(k)} | k \in \mathbb{N}_0\}$ as a Markov chain taking values in $\Omega = \Lambda^N$ where $\boldsymbol{X}^{(k)} = (X_1^{(k)}, \ldots, X_N^{(k)})$ describes the state of the MRF at time $k \geq 0$. At each transition step we now pick a random variable X_i, $i \in V$ with a probability $q(i)$ given by a strictly positive probability distribution q on V and sample a new value for X_i based on its conditional probability distribution given the state $(x_v)_{v \in V \setminus i}$ of all other variables $(X_v)_{v \in V \setminus i}$, i.e. based on $\pi\left(X_i | (x_v)_{v \in V \setminus i}\right) = \pi\left(X_i | (x_w)_{w \in \mathcal{N}_i}\right)$. Therefore, the transition probability p_{xy} for two states $\boldsymbol{x}, \boldsymbol{y}$ of the MRF $\boldsymbol{X}$ with $\boldsymbol{x} \neq \boldsymbol{y}$ is given by:

$$p_{\boldsymbol{xy}} = \begin{cases} q(i)\pi\left(y_i | (x_v)_{v \in V \setminus i}\right), & \text{if } \exists i \in V \text{ so that } \forall v \in V \text{ with } v \neq i\text{: } x_v = y_v \\ 0, & \text{else .} \end{cases} \tag{16}$$

And the probability, that the state of the MRF $\boldsymbol{x}$ stays the same, is given by:

$$p_{\boldsymbol{xx}} = \sum_{i \in V} q(i)\pi\left(x_i | (x_v)_{v \in V \setminus i}\right) \quad . \tag{17}$$

It is easy to see that the joint distribution π of the MRF is the stationary distribution of the Markov chain defined by these transition probabilities by showing that the detailed balance condition (13) holds: For $\boldsymbol{x} = \boldsymbol{y}$ this follows directly. If $\boldsymbol{x}$ and $\boldsymbol{y}$ differ in the value of more than one random variable it follows from the fact that $p_{\boldsymbol{yx}} = p_{\boldsymbol{xy}} = 0$. Assume that $\boldsymbol{x}$ and $\boldsymbol{y}$ differ only in the state of exactly one variable X_i, i.e., $y_j = x_j$ for $j \neq i$ and $y_i \neq x_i$, then it holds:

$$\begin{aligned} \pi(\boldsymbol{x})p_{\boldsymbol{xy}} = \pi(\boldsymbol{x})q(i)\pi\left(y_i | (x_v)_{v \in V \setminus i}\right) &= \pi\left(x_i, (x_v)_{v \in V \setminus i}\right) q(i) \frac{\pi\left(y_i, (x_v)_{v \in V \setminus i}\right)}{\pi\left((x_v)_{v \in V \setminus i}\right)} \\ &= \pi\left(y_i, (x_v)_{v \in V \setminus i}\right) q(i) \frac{\pi\left(x_i, (x_v)_{v \in V \setminus i}\right)}{\pi\left((x_v)_{v \in V \setminus i}\right)} \\ &= \pi(\boldsymbol{y})q(i)\pi\left(x_i | (x_v)_{v \in V \setminus i}\right) = \pi(\boldsymbol{y})p_{\boldsymbol{yx}}. \end{aligned} \tag{18}$$

Since π is strictly positive so are the conditional probability distributions of the single variables. Thus, it follows that every single variable X_i can take every state $x_i \in \Lambda$ in a single transition step and thus every state of the whole MRF can reach any other in Λ^N in a finite number of steps and the Markov chain is irreducible. Furthermore it follows from the positivity of the conditional distributions that $p_{\boldsymbol{xx}} > 0$ for all $\boldsymbol{x} \in \Lambda^N$ and thus that the Markov chain is aperiodic. Aperiodicity and irreducibility guaranty that the chain converges to the stationary distribution π.

In practice the single random variables to be updated are usually not chosen at random based on a distribution q but subsequently in fixed predefined order. The corresponding algorithm is often referred to as *periodic Gibbs Sampler*. If $\boldsymbol{P}$ is the transition matrix of the Gibbs chain, the convergence rate of the periodic

Gibbs sampler to the stationary distribution of the MRF is bounded by the following inequality (see for example [6]):

$$|\boldsymbol{\mu}\mathbf{P}^k - \boldsymbol{\pi}| \leq \frac{1}{2}|\boldsymbol{\mu} - \boldsymbol{\pi}|(1 - e^{-N\triangle})^k, \tag{19}$$

where $\triangle = \sup_{l \in V} \delta_l$ and $\delta_l = \sup\{|\mathcal{E}(\boldsymbol{x}) - \mathcal{E}(\boldsymbol{y})|; x_i = y_i \forall i \in V \text{ with } i \neq l\}$. Here μ is an arbitrary starting distribution and $\frac{1}{2}|\boldsymbol{\mu} - \boldsymbol{\pi}|$ is the distance in variation as defined in (15).

4 Restricted Boltzmann Machines

A RBM (also denoted as Harmonium [34]) is an MRF associated with a bipartite undirected graph as shown in Fig. 1. It consists of m visible units $\boldsymbol{V} = (V_1, ..., V_m)$ to represent observable data and n hidden units $\boldsymbol{H} = (H_1, ..., H_n)$ to capture dependencies between observed variables. In binary RBMs, our focus in this tutorial, the random variables $(\boldsymbol{V}, \boldsymbol{H})$ take values $(\boldsymbol{v}, \boldsymbol{h}) \in \{0, 1\}^{m+n}$ and the joint probability distribution under the model is given by the Gibbs distribution $p(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z} e^{-E(\boldsymbol{v}, \boldsymbol{h})}$ with the energy function

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{i=1}^{n}\sum_{j=1}^{m} w_{ij} h_i v_j - \sum_{j=1}^{m} b_j v_j - \sum_{i=1}^{n} c_i h_i \ . \tag{20}$$

For all $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$, w_{ij} is a real valued weight associated with the edge between units V_j and H_i and b_j and c_i are real valued bias terms associated with the jth visible and the ith hidden variable, respectively.

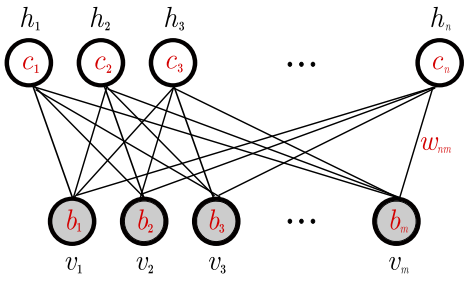


Fig. 1. The undirected graph of an RBM with n hidden and m visible variables

The graph of an RBM has only connections between the layer of hidden and visible variables but not between two variables of the same layer. In terms of probability this means that the hidden variables are independent given the state of the visible variables and vice versa:

$$p(\boldsymbol{h} \,|\, \boldsymbol{v}) = \prod_{i=1}^{n} p(h_i \,|\, \boldsymbol{v}) \text{ and } p(\boldsymbol{v} \,|\, \boldsymbol{h}) = \prod_{i=1}^{m} p(v_i \,|\, \boldsymbol{h}) \ . \tag{21}$$

The absence of connections between hidden variables makes the marginal distribution (7) of the visible variables easy to calculate:

$$\begin{aligned}
p(\boldsymbol{v}) &= \frac{1}{Z} \sum_{\boldsymbol{h}} p(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z} \sum_{\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})} \\
&= \frac{1}{Z} \sum_{h_1} \sum_{h_2} \cdots \sum_{h_n} e^{\sum_{j=1}^{m} b_j v_j} \prod_{i=1}^{n} e^{h_i \left(c_i + \sum_{j=1}^{m} w_{ij} v_j\right)} \\
&= \frac{1}{Z} e^{\sum_{j=1}^{m} b_j v_j} \sum_{h_1} e^{h_1 \left(c_1 + \sum_{j=1}^{m} w_{1j} v_j\right)} \sum_{h_2} e^{h_2 \left(c_2 + \sum_{j=1}^{m} w_{2j} v_j\right)} \cdots \sum_{h_n} e^{h_n \left(c_n + \sum_{j=1}^{m} w_{nj} v_j\right)} \\
&= \frac{1}{Z} e^{\sum_{j=1}^{m} b_j v_j} \prod_{i=1}^{n} \sum_{h_i} e^{h_i \left(c_i + \sum_{j=1}^{m} w_{ij} v_j\right)} \\
&= \frac{1}{Z} \prod_{j=1}^{m} e^{b_j v_j} \prod_{i=1}^{n} \left(1 + e^{c_i + \sum_{j=1}^{m} w_{ij} v_j}\right)
\end{aligned} \tag{22}$$

This equation shows why a (marginalized) RBM can be regarded as a *product of experts* model [15,39], in which a number of "experts" for individual components of the observations are combined multiplicatively.

Any distribution on $\{0,1\}^m$ can be modeled arbitrarily well by an RBM with m visible and $k+1$ hidden units, where k denotes the cardinality of the support set of the target distribution, that is, the number of input elements from $\{0,1\}^m$ that have a non-zero probability of being observed [24]. It has been shown recently that even less units can be sufficient depending on the patterns in the support set [30].

The RBM can be interpreted as a stochastic neural network, where nodes and edges correspond to neurons and synaptic connections, respectively. The conditional probability of a single variable being one can be interpreted as the firing rate of a (stochastic) neuron with sigmoid activation function $\sigma(x) = 1/(1+e^{-x})$, because it holds:

$$p(H_i = 1 \,|\, \boldsymbol{v}) = \sigma\left(\sum_{j=1}^{m} w_{ij} v_j + c_i\right) \tag{23}$$

and

$$p(V_j = 1 \,|\, \boldsymbol{h}) = \sigma\left(\sum_{i=1}^{n} w_{ij} h_i + b_j\right) \ . \tag{24}$$

To see this, let $\boldsymbol{v}_{-l}$ denote the state of all visible units except the lth one and let us define

$$\alpha_l(\boldsymbol{h}) := -\sum_{i=1}^{n} w_{il} h_i - b_l \tag{25}$$

and

$$\beta(\boldsymbol{v}_{-l}, \boldsymbol{h}) := -\sum_{i=1}^{n} \sum_{j=1, j\neq l}^{m} w_{ij} h_i v_j - \sum_{j=1, j\neq l}^{m} b_i v_i - \sum_{i=1}^{n} c_i h_i \ . \tag{26}$$

Then $E(\boldsymbol{v}, \boldsymbol{h}) = \beta(\boldsymbol{v}_{-l}, \boldsymbol{h}) + v_l \alpha_l(\boldsymbol{h})$, where $v_l \alpha_l(\boldsymbol{h})$ collects all terms involving v_l and we can write [2]:

$$\begin{aligned}
p(V_l = 1 \,|\boldsymbol{h}) = p(V_l = 1 \,|\boldsymbol{v}_{-l}, \boldsymbol{h}) &= \frac{p(V_l = 1, \boldsymbol{v}_{-l}, \boldsymbol{h})}{p(\boldsymbol{v}_{-l}, \boldsymbol{h})} \\
= \frac{e^{-E(v_l=1, \boldsymbol{v}_{-l}, \boldsymbol{h})}}{e^{-E(v_l=1, \boldsymbol{v}_{-l}, \boldsymbol{h})} + e^{-E(v_l=0, \boldsymbol{v}_{-l}, \boldsymbol{h})}} &= \frac{e^{-\beta(\boldsymbol{v}_{-l}, \boldsymbol{h}) - 1 \cdot \alpha_l(\boldsymbol{h})}}{e^{-\beta(\boldsymbol{v}_{-l}, \boldsymbol{h}) - 1 \cdot \alpha_l(\boldsymbol{h})} + e^{-\beta(\boldsymbol{v}_{-l}, \boldsymbol{h}) - 0 \cdot \alpha_l(\boldsymbol{h})}} \\
= \frac{e^{-\beta(\boldsymbol{v}_{-l}, \boldsymbol{h})} \cdot e^{-\alpha_l(\boldsymbol{h})}}{e^{-\beta(\boldsymbol{v}_{-l}, \boldsymbol{h})} \cdot e^{-\alpha_l(\boldsymbol{h})} + e^{-\beta(\boldsymbol{v}_{-l}, \boldsymbol{h})}} &= \frac{e^{-\beta(\boldsymbol{v}_{-l}, \boldsymbol{h})} \cdot e^{-\alpha_l(\boldsymbol{h})}}{e^{-\beta(\boldsymbol{v}_{-l}, \boldsymbol{h})} \cdot \left(e^{-\alpha_l(\boldsymbol{v}_{-l}, \boldsymbol{h})} + 1\right)} \\
= \frac{e^{-\alpha_l(\boldsymbol{v}_{-l}, \boldsymbol{h})}}{e^{-\alpha_l(\boldsymbol{v}_{-l}, \boldsymbol{h})} + 1} &= \frac{\frac{1}{e^{\alpha_l(\boldsymbol{h})}}}{\frac{1}{e^{\alpha_l(\boldsymbol{h})}} + 1} = \frac{1}{1 + e^{\alpha_l(\boldsymbol{v}_{-l}, \boldsymbol{h})}} \\
&= \sigma(-\alpha_l(\boldsymbol{h})) = \sigma\left(\sum_{i=1}^{n} w_{il} h_i + b_j\right)
\end{aligned} \tag{27}$$

The independence between the variables in one layer makes Gibbs sampling especially easy: Instead of sampling new values for all variables subsequently, the states of all variables in one layer can be sampled jointly. Thus, Gibbs sampling can be performed in just two sub steps: sampling a new state $\boldsymbol{h}$ for the hidden neurons based on $p(\boldsymbol{h}|\boldsymbol{v})$ and sampling a state $\boldsymbol{v}$ for the visible layer based on $p(\boldsymbol{v}|\boldsymbol{h})$. This is also referred to as *block Gibbs sampling*.

As mentioned in the introduction, an RBM can be reinterpreted as a standard feed-forward neural network with one layer of non-linear processing units. From this perspective the RBM is viewed as a deterministic function $\{0,1\}^m \to \mathbb{R}^n$ that maps an input $\boldsymbol{v} \in \{0,1\}^m$ to $\boldsymbol{y} \in \mathbb{R}^n$ with $y_i = p(H_i = 1|\boldsymbol{v})$. That is, an observation is mapped to the expected value of the hidden neurons given the observation.

4.1 The Gradient of the Log-Likelihood

The log-likelihood gradient of an MRF can be written as the sum of two expectations, see (9). For RBMs the first term of (9) (i.e., the expectation of the energy gradient under the conditional distribution of the hidden variables given a training sample $\boldsymbol{v}$) can be computed efficiently because it factorizes nicely. For example, w.r.t. the parameter w_{ij} we get:

$$\sum_{\boldsymbol{h}} p(\boldsymbol{h} \,|\, \boldsymbol{v}) \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial w_{ij}} = \sum_{\boldsymbol{h}} p(\boldsymbol{h} \,|\, \boldsymbol{v}) h_i v_j$$

$$= \sum_{\boldsymbol{h}} \prod_{k=1}^{n} p(h_k \,|\, \boldsymbol{v}) h_i v_j = \sum_{h_i} \sum_{\boldsymbol{h}_{-i}} p(h_i \,|\, \boldsymbol{v}) p(\boldsymbol{h}_{-i} \,|\, \boldsymbol{v}) h_i v_j$$

$$= \sum_{h_i} p(h_i \,|\, \boldsymbol{v}) h_i v_j \underbrace{\sum_{\boldsymbol{h}_{-i}} p(\boldsymbol{h}_{-i} | \boldsymbol{v})}_{=1} = p(H_i = 1 | \boldsymbol{v}) v_j = \sigma\Bigg(\sum_{j=1}^{m} w_{ij} v_j + c_i \Bigg) v_j \quad (28)$$

Since the second term in (9) can also be written as $\sum_{\boldsymbol{v}} p(\boldsymbol{v}) \sum_{\boldsymbol{h}} p(\boldsymbol{h} \,|\, \boldsymbol{v}) \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial \boldsymbol{\theta}}$ or $\sum_{\boldsymbol{h}} p(\boldsymbol{h}) \sum_{\boldsymbol{v}} p(\boldsymbol{v} \,|\, \boldsymbol{h}) \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial \boldsymbol{\theta}}$ we can also reduce its computational complexity by applying the same kind of factorization to the inner sum, either factorizing over the hidden variables as shown above or factorizing over the visible variables in an analogous way. However, the computation remains intractable for regular sized RBMs because its complexity is still exponential in the size of the smallest layer (the outer sum still runs over either 2^m or 2^n states).

Using the factorization trick (28) the derivative of the log-likelihood of a single training pattern $\boldsymbol{v}$ w.r.t. the weight w_{ij} becomes

$$\frac{\partial \ln \mathcal{L}(\boldsymbol{\theta} \,|\, \boldsymbol{v})}{\partial w_{ij}} = - \sum_{\boldsymbol{h}} p(\boldsymbol{h} \,|\, \boldsymbol{v}) \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial w_{ij}} + \sum_{\boldsymbol{v}, \boldsymbol{h}} p(\boldsymbol{v}, \boldsymbol{h}) \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial w_{ij}}$$

$$= \sum_{\boldsymbol{h}} p(\boldsymbol{h} \,|\, \boldsymbol{v}) h_i v_j - \sum_{\boldsymbol{v}} p(\boldsymbol{v}) \sum_{\boldsymbol{h}} p(\boldsymbol{h} \,|\, \boldsymbol{v}) h_i v_j$$

$$= p(H_i = 1 | \, \boldsymbol{v}) v_j - \sum_{\boldsymbol{v}} p(\boldsymbol{v}) p(H_i = 1 | \, \boldsymbol{v}) v_j \ . \quad (29)$$

For the mean of this derivative over a training set $S = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\ell\}$ often the following notations are used:

$$\frac{1}{\ell} \sum_{\boldsymbol{v} \in S} \frac{\partial \ln \mathcal{L}(\boldsymbol{\theta} \,|\, \boldsymbol{v})}{\partial w_{ij}} = \frac{1}{\ell} \sum_{\boldsymbol{v} \in S} \left[-\mathbb{E}_{p(\boldsymbol{h} \,|\, \boldsymbol{v})} \left[\frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial w_{ij}} \right] + \mathbb{E}_{p(\boldsymbol{h}, \boldsymbol{v})} \left[\frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial w_{ij}} \right] \right]$$

$$= \frac{1}{\ell} \sum_{\boldsymbol{v} \in S} \left[\mathbb{E}_{p(\boldsymbol{h} \,|\, \boldsymbol{v})} \left[v_i h_j \right] - \mathbb{E}_{p(\boldsymbol{h}, \boldsymbol{v})} \left[v_i h_j \right] \right]$$

$$= \left\langle v_i h_j \right\rangle_{p(\boldsymbol{h} \,|\, \boldsymbol{v}) q(\boldsymbol{v})} - \left\langle v_i h_j \right\rangle_{p(\boldsymbol{h}, \boldsymbol{v})} \quad (30)$$

with q denoting the empirical distribution. This gives the often stated rule:

$$\sum_{\boldsymbol{v} \in S} \frac{\partial \ln \mathcal{L}(\boldsymbol{\theta} \,|\, \boldsymbol{v})}{\partial w_{ij}} \propto \left\langle v_i h_j \right\rangle_{\text{data}} - \left\langle v_i h_j \right\rangle_{\text{model}} \quad (31)$$

Analogously to (29) we get the derivatives w.r.t. the bias parameter b_j of the jth visible variable

$$\frac{\partial \ln \mathcal{L}(\boldsymbol{\theta} \,|\, \boldsymbol{v})}{\partial b_j} = v_j - \sum_{\boldsymbol{v}} p(\boldsymbol{v}) v_j \quad (32)$$

and w.r.t. the bias parameter c_i of the ith hidden variable

$$\frac{\partial \ln \mathcal{L}(\boldsymbol{\theta} \,|\, \boldsymbol{v})}{\partial c_i} = p(H_i = 1 |\, \boldsymbol{v}) - \sum_{\boldsymbol{v}} p(\boldsymbol{v}) p(H_i = 1 |\, \boldsymbol{v}) \ . \tag{33}$$

To avoid the exponential complexity of summing over all values of the visible variables (or all values of the hidden if one decides to factorize over the visible variables beforehand) when calculating the second term of the log-likelihood gradient – or the second terms of (29), (32), and (33) – one can approximate this expectation by samples from the model distribution. These samples can be obtained by Gibbs sampling. This requires running the Markov chain "long enough" to ensure convergence to stationarity. Since the computational costs of such an MCMC approach are still too large to yield an efficient learning algorithm common RBM learning techniques – as described in the following section – introduce additional approximations.

5 Approximating the RBM Log-Likelihood Gradient

All common training algorithms for RBMs approximate the log-likelihood gradient given some data and perform gradient ascent on these approximations. Selected learning algorithms will be described in the following section, starting with contrastive divergence learning.

5.1 Contrastive Divergence

Obtaining unbiased estimates of log-likelihood gradient using MCMC methods typically requires many sampling steps. However, recently it was shown that estimates obtained after running the chain for just a few steps can be sufficient for model training [15]. This leads to *contrastive divergence* (CD) learning, which has become a standard way to train RBMs [15,4,18,3,17].

The idea of k-step contrastive divergence learning (CD-k) is quite simple: Instead of approximating the second term in the log-likelihood gradient by a sample from the RBM-distribution (which would require to run a Markov chain until the stationary distribution is reached), a Gibbs chain is run for only k steps (and usually $k = 1$). The Gibbs chain is initialized with a training example $\boldsymbol{v}^{(0)}$ of the training set and yields the sample $\boldsymbol{v}^{(k)}$ after k steps. Each step t consists of sampling $\boldsymbol{h}^{(t)}$ from $p(\boldsymbol{h}|\boldsymbol{v}^{(t)})$ and sampling $\boldsymbol{v}^{(t+1)}$ from $p(\boldsymbol{v}|\boldsymbol{h}^{(t)})$ subsequently. The gradient (equation (9)) w.r.t. $\boldsymbol{\theta}$ of the log-likelihood for one training pattern $\boldsymbol{v}^{(0)}$ is then approximated by

$$\mathrm{CD}_k(\boldsymbol{\theta}, \boldsymbol{v}^{(0)}) = -\sum_{\boldsymbol{h}} p(\boldsymbol{h}|\boldsymbol{v}^{(0)}) \frac{\partial E(\boldsymbol{v}^{(0)}, \boldsymbol{h})}{\partial \boldsymbol{\theta}} + \sum_{\boldsymbol{h}} p(\boldsymbol{h}|\boldsymbol{v}^{(k)}) \frac{\partial E(\boldsymbol{v}^{(k)}, \boldsymbol{h})}{\partial \boldsymbol{\theta}} \ . \tag{34}$$

The derivatives in direction of the single parameters are obtained by "estimating" the expectations over $p(\boldsymbol{v})$ in (29), (32) and (33) by the single sample $\boldsymbol{v}^{(k)}$. A batch version of CD-k can be seen in algorithm 1.

Algorithm 1. k-step contrastive divergence

Input: RBM $(V_1, \ldots, V_m, H_1, \ldots, H_n)$, training batch S
Output: gradient approximation Δw_{ij}, Δb_j and Δc_i for $i = 1, \ldots, n$, $j = 1, \ldots, m$

```
init Δw_ij = Δb_j = Δc_i = 0 for i = 1,...,n, j = 1,...,m
forall the v ∈ S do
    v^(0) ← v
    for t = 0,...,k−1 do
        for i = 1,...,n do sample h_i^(t) ~ p(h_i | v^(t))
        for j = 1,...,m do sample v_j^(t+1) ~ p(v_j | h^(t))
    for i = 1,...,n, j = 1,...,m do
        Δw_ij ← Δw_ij + p(H_i = 1 | v^(0))·v_j^(0) − p(H_i = 1 | v^(k))·v_j^(k)
        Δb_j ← Δb_j + v_j^(0) − v_j^(k)
        Δc_i ← Δc_i + p(H_i = 1 | v^(0)) − p(H_i = 1 | v^(k))
```

Since $\boldsymbol{v}^{(k)}$ is not a sample from the stationary model distribution the approximation (34) is biased. Obviously, the bias vanishes as $k \to \infty$. That CD is a biased approximation becomes also clear by realizing that it does not maximize the likelihood of the data under the model but the difference of two KL-divergences [15]:

$$\mathrm{KL}(q|p) - \mathrm{KL}(p_k|p) \ , \tag{35}$$

where q is the empirical distribution and p_k is the distribution of the visible variables after k steps of the Markov chain. If the chain already reached stationarity it holds $p_k = p$ and thus $\mathrm{KL}(p_k|p) = 0$ and the approximation error of CD vanishes.

The theoretical results from [3] give a good understanding of the CD approximation and the corresponding bias by showing that the log-likelihood gradient can – based on a Markov chain – be expressed as a sum of terms containing the k-th sample:

Theorem 1 (Bengio and Delalleau [3]). *For a converging Gibbs chain*

$$\boldsymbol{v}^{(0)} \Rightarrow \boldsymbol{h}^{(0)} \Rightarrow \boldsymbol{v}^{(1)} \Rightarrow \boldsymbol{h}^{(1)} \ldots$$

starting at data point $\boldsymbol{v}^{(0)}$, the log-likelihood gradient can be written as

$$\begin{aligned} \frac{\partial}{\partial \boldsymbol{\theta}} \ln p(\boldsymbol{v}^{(0)}) = &- \sum_{\boldsymbol{h}} p(\boldsymbol{h}|\boldsymbol{v}^{(0)}) \frac{\partial E(\boldsymbol{v}^{(0)}, \boldsymbol{h})}{\partial \boldsymbol{\theta}} \\ &+ E_{p(\boldsymbol{v}^{(k)}|\boldsymbol{v}^{(0)})} \left[\sum_{\boldsymbol{h}} p(\boldsymbol{h}|\boldsymbol{v}^{(k)}) \frac{\partial E(\boldsymbol{v}^{(k)}, \boldsymbol{h})}{\partial \boldsymbol{\theta}} \right] + E_{p(\boldsymbol{v}^{(k)}|\boldsymbol{v}^{(0)})} \left[\frac{\partial \ln p(\boldsymbol{v}^{(k)})}{\partial \boldsymbol{\theta}} \right] \end{aligned} \tag{36}$$

and the final term converges to zero as k goes to infinity.

The first two terms in equation (36) just correspond to the expectation of the CD approximation (under p_k) and the bias is given by the final term.

The approximation error does not only depend on the value of k but also on the rate of convergence or the mixing rate of the Gibbs chain. The rate describes how fast the Markov chain approaches the stationary distribution. The mixing rate of the Gibbs chain of an RBM is up to the magnitude of the model parameters [15,7,3,12]. This becomes clear by considering that the conditional probabilities $p(v_j|\boldsymbol{h})$ and $p(h_i|\boldsymbol{v})$ are given by thresholding $\sum_{i=1}^{n} w_{ij}h_i + b_j$ and $\sum_{j=1}^{m} w_{ij}v_j + c_i$, respectively. If the absolute values of the parameters are high, the conditional probabilities can get close to one or zero. If this happens, the states get more and more "predictable" and the Markov chain changes its state slowly. An empirical analysis of the dependency between the size of the bias and magnitude of the parameters can be found in [3].

An upper bound on the expectation of the CD approximation error under the empirical distribution is given by the following theorem [12]:

Theorem 2 (Fischer and Igel [12]). *Let p denote the marginal distribution of the visible units of an RBM and let q be the empirical distribution defined by a set of samples $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\ell$. Then an upper bound on the expectation of the error of the CD-k approximation of the log-likelihood derivative w.r.t some RBM parameter θ_a is given by*

$$\left| E_{q(\boldsymbol{v}^{(0)})} \left[E_{p(\boldsymbol{v}^{(k)}|\boldsymbol{v}^{(0)})} \left[\frac{\partial \ln p(\boldsymbol{v}^{(k)})}{\partial \theta_a} \right] \right] \right| \leq \frac{1}{2} \|q - p\| \left(1 - e^{-(m+n)\Delta} \right)^k \tag{37}$$

with

$$\Delta = \max \left\{ \max_{l \in \{1,\ldots,m\}} \vartheta_l, \max_{l \in \{1,\ldots,n\}} \xi_l \right\} ,$$

where

$$\vartheta_l = \max \left\{ \left| \sum_{i=1}^{n} I_{\{w_{il}>0\}} w_{il} + b_l \right| , \left| \sum_{i=1}^{n} I_{\{w_{il}<0\}} w_{il} + b_l \right| \right\}$$

and

$$\xi_l = \max \left\{ \left| \sum_{j=1}^{m} I_{\{w_{lj}>0\}} w_{lj} + c_l \right| , \left| \sum_{j=1}^{m} I_{\{w_{lj}<0\}} w_{lj} + c_l \right| \right\} .$$

The bound (and probably also the bias) depends on the absolute values of the RBM parameters, on the size of the RBM (the number of variables in the graph), and on the distance in variation between the modeled distribution and the starting distribution of the Gibbs chain.

As a consequence of the approximation error CD-learning does not necessarily lead to a maximum likelihood estimate of the model parameters. Yuille [42] specifies conditions under which CD learning is guaranteed to converge to the maximum likelihood solution, which need not hold for RBM training in general. Examples of energy functions and Markov chains for which CD-1 learning does not converge are given in [27]. The empirical comparisons of the CD-approximation and the true gradient for RBMs small enough that the gradient

is still tractable conducted in [7] and [3] shows that the bias can lead to a convergence to parameters that do not reach the maximum likelihood.

The bias, however, can also lead to a distortion of the learning process: After some learning iterations the likelihood can start to diverge (see figure 2) in the sense that the model systematically gets worse if k is not large [11]. This is especially bad because the log-likelihood is not tractable in reasonable sized RBMs, and so the misbehavior can not be displayed and used as a stopping criterion. Because the effect depends on the magnitude of the weights, weight decay can help to prevent it. However, the weight decay parameter λ, see equation (6), is difficult to tune. If it is too small, weight decay has no effect. If it is too large, the learning converges to models with low likelihood [11].

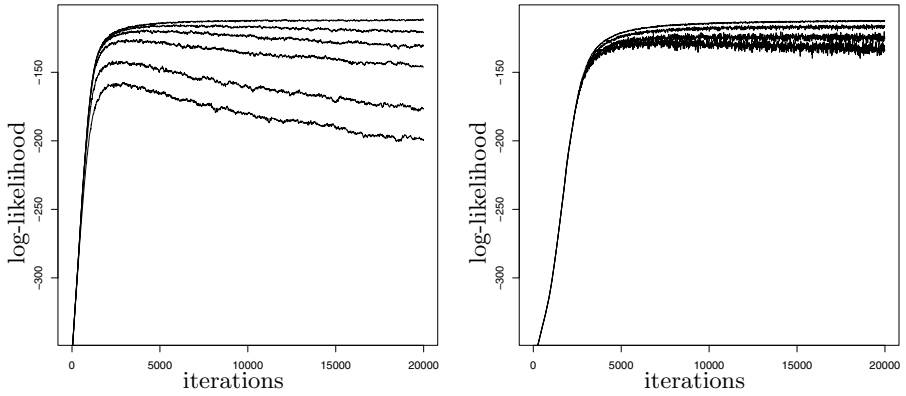


Fig. 2. Evolution of the log-likelihood during batch training of an RBM. In the left plot, CD-k with different values for k (from bottom to top $k = 1, 2, 5, 10, 20, 100$) was used. In the right plot, we employed parallel tempering (PT, see section 5.3) with different numbers M of temperatures (from bottom to top $M = 4, 5, 10, 50$). In PT the inverse temperatures were equally distributed in the interval $[0, 1]$, which may not be the optimal [9]. The training set was given by a 4×4 variant of the Bars-and-Stripes benchmark problem [28]. The learning rate was $\eta = 0.1$ for CD and $\eta = 0.05$ for PT and neither weight decay nor a momentum term were used ($\lambda = \nu = 0$). Shown are medians over 25 runs.

More recently proposed learning algorithms try to yield better approximations of the log-likelihood gradient by sampling from Markov chains with increased mixing rate.

5.2 Persistent Contrastive Divergence

The idea of *persistent contrastive divergence* (PCD, [36]) is described in [41] for log-likelihood maximization of general MRFs and is applied to RBMs in [36]. The PCD approximation is obtained from the CD approximation (34) by replacing the sample $v^{(k)}$ by a sample from a Gibbs chain that is independent

from the sample $v^{(0)}$ of the training distribution. The algorithm corresponds to standard CD learning without reinitializing the visible units of the Markov chain with a training sample each time we want to draw a sample $v^{(k)}$ approximately from the RBM distribution. Instead one keeps "persistent" chains which are run for k Gibbs steps after each parameter update (i.e., the initial state of the current Gibbs chain is equal to $v^{(k)}$ from the previous update step). The fundamental idea underlying PCD is that one could assume that the chains stay close to the stationary distribution if the learning rate is sufficiently small and thus the model changes only slightly between parameter updates [41,36]. The number of persistent chains used for sampling (or the number of samples used to approximate the second term of gradient (9)) is a hyper parameter of the algorithm. In the canonical from, there exists one Markov chain per training example in a batch.

The PCD algorithm was further refined in a variant called *fast persistent contrastive divergence* (FPCD, [37]). Fast PCD tries to reach faster mixing of the Gibbs chain by introducing additional parameters w_{ij}^f, b_j^f, c_i^f (for $i = 1, \ldots, n$ and $j = 1, \ldots, m$) referred to as *fast* parameters. These new set of parameters is only used for sampling and not in the model itself. When calculating the conditional distributions for Gibbs sampling, the regular parameters are replaced by the sum of the regular and the fast parameters, i.e., Gibbs sampling is based on the probabilities $\tilde{p}(H_i = 1 \,|\, \boldsymbol{v}) = \sigma\left(\sum_{j=1}^{m} (w_{ij} + w_{ij}^f) v_j + (c_i + c_i^f)\right)$ and $\tilde{p}(V_j = 1 \,|\, \boldsymbol{h}) = \sigma\left(\sum_{i=1}^{n} (w_{ij} + w_{ij}^f) h_i + (b_j + b_j^f)\right)$ instead of the conditional probabilities given by (23) and (24). The learning update rule for the fast parameters equals the one for the regular parameters, but with an independent, large learning rate leading to faster changes as well as a large weight decay parameter. Weight decay can also be used for the regular parameters, but it was suggested that regularizing just the fast weights is sufficient [37].

Neither PCD nor FPCD seem to enlarge the mixing rate (or decrease the bias of the approximation) sufficiently to avoid the divergence problem as can be seen in the empirical analysis in [11].

5.3 Parallel Tempering

One of the most promising sampling technique used for RBM-training so far is *parallel tempering* (PT, [33,10,8]). It introduces supplementary Gibbs chains that sample form more and more smoothed replicas of the original distribution. This can be formalized in the following way: Given an ordered set of M temperatures $T_1, T_2, \ldots, T_M$ with $1 = T_1 < T_2 < \cdots < T_M$, we define a set of M Markov chains with stationary distributions

$$p_r(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z_r} e^{-\frac{1}{T_r} E(\boldsymbol{v}, \boldsymbol{h})} \tag{38}$$

for $r = 1, \ldots, M$, where $Z_r = \sum_{\boldsymbol{v}, \boldsymbol{h}} e^{-\frac{1}{T_r} E(\boldsymbol{v}, \boldsymbol{h})}$ is the corresponding partition function, and p_1 is exactly the model distribution.

Algorithm 2. k-step parallel tempering with M temperatures

Input: RBM $(V_1, \ldots, V_m, H_1, \ldots, H_n)$, training batch S, current state $\boldsymbol{v}_r$ of Markov chain with stationary distribution p_r for $r = 1, \ldots, M$
Output: gradient approximation Δw_{ij}, Δb_j and Δc_i for $i = 1, \ldots, n$, $j = 1, \ldots, m$

init $\Delta w_{ij} = \Delta b_j = \Delta c_i = 0$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$
forall the $\boldsymbol{v} \in S$ **do**
  for $r = 1, \ldots, M$ **do**
    $\boldsymbol{v}_r^{(0)} \leftarrow \boldsymbol{v}_r$
    for $i = 1, \ldots, n$ **do** sample $h_{r,i}^{(0)} \sim p(h_{r,i} \,|\, \boldsymbol{v}_r^{(0)})$
    for $t = 0, \ldots, k-1$ **do**
      for $j = 1, \ldots, m$ **do** sample $v_{r,j}^{(t+1)} \sim p(v_{r,j} \,|\, \boldsymbol{h}_r^{(t)})$
      for $i = 1, \ldots, n$ **do** sample $h_{r,i}^{(t+1)} \sim p(h_{r,i} \,|\, \boldsymbol{v}_r^{(t+1)})$
    $\boldsymbol{v}_r \leftarrow \boldsymbol{v}_r^{(k)}$
  `/* swapping order below works well in practice [26] */`
  for $r \in \{s \,|\, 2 \leq s \leq M \text{ and } s \bmod 2 = 0\}$ **do**
    swap $(\boldsymbol{v}_r^{(k)}, \boldsymbol{h}_r^{(k)})$ and $(\boldsymbol{v}_{r-1}^{(k)}, \boldsymbol{h}_{r-1}^{(k)})$ with probability given by (40)
  for $r \in \{s \,|\, 3 \leq s \leq M \text{ and } s \bmod 2 = 1\}$ **do**
    swap $(\boldsymbol{v}_r^{k}, \boldsymbol{h}_r^{k})$ and $(\boldsymbol{v}_{r-1}^{k}, \boldsymbol{h}_{r-1}^{k})$ with probability given by (40)
  for $i = 1, \ldots, n$, $j = 1, \ldots, m$ **do**
    $\Delta w_{ij} \leftarrow \Delta w_{ij} + p(H_i = 1 \,|\, \boldsymbol{v}) \cdot v_j - p(H_i = 1 \,|\, \boldsymbol{v}_1^{(k)}) \cdot v_{1,j}^{(k)}$
    $\Delta b_j \leftarrow \Delta b_j + v_j - v_{1,j}^{(k)}$
    $\Delta c_i \leftarrow \Delta c_i + p(H_i = 1 \,|\, \boldsymbol{v}) - p(H_i = 1 \,|\, \boldsymbol{v}_1^{(k)})$

In each step of the algorithm, we run k (usually $k = 1$) Gibbs sampling steps in each tempered Markov chain yielding samples $(\boldsymbol{v}_1, \boldsymbol{h}_1), \ldots, (\boldsymbol{v}_M, \boldsymbol{h}_M)$. After this, two neighboring Gibbs chains with temperatures T_r and T_{r-1} may exchange particles $(\boldsymbol{v}_r, \boldsymbol{h}_r)$ and $(\boldsymbol{v}_{r-1}, \boldsymbol{h}_{r-1})$ with an exchange probability based on the Metropolis ratio,

$$\min\left\{1, \frac{p_r\big(\boldsymbol{v}_{r-1}, \boldsymbol{h}_{r-1}\big) p_{r-1}\big(\boldsymbol{v}_r, \boldsymbol{h}_r\big)}{p_r\big(\boldsymbol{v}_r, \boldsymbol{h}_r\big) p_{r-1}\big(\boldsymbol{v}_{r-1}, \boldsymbol{h}_{r-1}\big)}\right\} , \tag{39}$$

which gives for RBMs

$$\min\left\{1, \exp\left(\left(\frac{1}{T_r} - \frac{1}{T_{r-1}}\right) * \left(E(\boldsymbol{v}_r, \boldsymbol{h}_r) - E(\boldsymbol{v}_{r-1}, \boldsymbol{h}_{r-1})\right)\right)\right\} . \tag{40}$$

After performing these swaps between chains, which enlarge the mixing rate, we take the (eventually exchanged) sample $\boldsymbol{v}_1$ of original chain (with temperature $T_1 = 1$) as a sample from the model distribution. This procedure is repeated L times yielding samples $\boldsymbol{v}_{1,1}, \ldots, \boldsymbol{v}_{1,L}$ used for the approximation of the expectation under the RBM distribution in the log-likelihood gradient (i.e., for the

approximation of the second term in (9)). Usually L is set to the number of samples in the (mini) batch of training data as shown in algorithm 2.

Compared to CD, PT introduces computational overhead, but results in a faster mixing Markov chain and thus a less biased gradient approximation. The evolution of the log-likelihood during training using PT with different values of M can be seen in figure 2.

6 RBMs with Real-Valued Variables

So far, we considered only observations represented by binary vectors, but often one would like to model distributions over continuous data. There are several ways to define RBMs with real-valued visible units. As demonstrated by [18], one can model a continuous distribution with a binary RBM by a simple "trick". The input data is scaled to the interval $[0, 1]$ and modeled by the probability of the visible variables to be one. That is, instead of sampling binary values, the expectation $p(V_j = 1|\boldsymbol{h})$ is regarded as the current state of the variable V_j. Except for the continuous values of the visible variables and the resulting changes in the sampling procedure the learning process remains the same. By keeping the energy function as given in (20) and just replacing the state space $\{0, 1\}^m$ of $\boldsymbol{V}$ by $[0, 1]^m$, the conditional distributions of the visible variables belong to the class of truncated exponential distributions. This can be shown in the same way as the sigmoid function for binary RBMs is derived in (27). Visible neurons with a Gaussian distributed conditional are for example gained by augmenting the energy with quadratical terms $\sum_j d_j v_j^2$ weighted by parameters d_j, $j = 1, \ldots, m$.

In contrast to the universal approximation capabilities of standard RBMs on $\{0, 1\}^m$, the subset of real-valued distributions that can be modeled by an RBM with real-valued visible units is rather constrained [38].

More generally, it is possible to cover continuous valued variables by extending the definition of an RBM to any MRF whose energy function is such that $p(\boldsymbol{h}|\boldsymbol{v}) = \prod_i p(h_i|\boldsymbol{v})$ and $p(\boldsymbol{v}|\boldsymbol{h}) = \prod_j p(v_j|\boldsymbol{h})$. As follows directly from the Hammersley-Clifford theorem and as also discussed in [18], this holds for any energy function of the form

$$E(\boldsymbol{v}, \boldsymbol{h}) = \sum_{i,j} \phi_{i,j}(h_i, v_j) + \sum_j \omega_j(v_j) + \sum_i \nu_i(h_i)) \tag{41}$$

with real-valued functions $\phi_{i,j}$, ω_j, and ν_i ,$i = 1, \ldots, n$ and $j = 1, \ldots, m$, fulfilling the constraint that the partition function Z is finite. Welling et al. [40] come to almost the same generalized form of the energy function in their framework for constructing *exponential family harmoniums* from arbitrary marginal distributions $p(v_j)$ and $p(h_i)$ from the exponential family.

7 Loosening the Restrictions

In this closing section, we will give a very brief outlook on selected extensions of RBMs that loosen the imposed restrictions on the bipartite network topology

by introducing dependencies on further random variables or by allowing for arbitrary connections between nodes in the model.

Conditional RBMs. Several generalizations and extensions of RBMs exist. A notable example are *conditional RBMs* (e.g., example [35,29]). In these models, some of the parameters in the RBM energy are replaced by parametrized functions of some conditioning random variables, see [2] for an introduction.

Boltzmann Machines. Removing the "R" from the RBM brings us back to where everything started, to the general Boltzmann machine [1]. These are MRFs consisting of a set of hidden and visible variables where the energy is given by

$$E(\boldsymbol{v}, \boldsymbol{h}) = \\ -\sum_{i=1}^{n}\sum_{j=1}^{m} h_i w_{ij} v_j - \sum_{k=1}^{m}\sum_{l<k} v_k u_{kl} v_l - \sum_{k=1}^{n}\sum_{l<k} h_k y_{kl} h_l - \sum_{j=1}^{m} b_j v_j - \sum_{i=1}^{n} c_i h_i \ . \tag{42}$$

The graph corresponds to the one of an RBM with additional connections between the variables of one layer. These dependencies make sampling more complex (in Gibbs sampling each variable has to be updated independently) and thus training more difficult. However, specialized learning algorithms for particular "deep" graph structures have been developed [32].

8 Next Steps

The goal of this tutorial was to introduce RBMs from the probabilistic graphical model perspective. The text is meant to supplement existing tutorials, and it is biased in the sense that it focuses on material that we found helpful in our work. We hope that the reader is now equipped to move on to advanced models building on RBMs – in particular to deep learning architectures, where [2] may serve as an excellent starting point.

All experiments in this tutorial can be reproduced using the open source machine learning library Shark [20], which implements most of the models and algorithms that were discussed.

Acknowledgments. The authors acknowledge support from the German Federal Ministry of Education and Research within the National Network Computational Neuroscience under grant number 01GQ0951 (Bernstein Fokus "Learning behavioral models: From human experiment to technical assistance").

References

1. Ackley, D.H., Hinton, G.E., Sejnowski, T.J.: A learning algorithm for Boltzmann machines. Cognitive Science 9, 147–169 (1985)
2. Bengio, Y.: Learning deep architectures for AI. Foundations and Trends in Machine Learning 21(6), 1601–1621 (2009)
3. Bengio, Y., Delalleau, O.: Justifying and generalizing contrastive divergence. Neural Computation 21(6), 1601–1621 (2009)

4. Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H., Montreal, U.: Greedy layer-wise training of deep networks. In: Schölkopf, B., Platt, J., Hoffman, T. (eds.) Advances in Neural Information Processing (NIPS 19), pp. 153–160. MIT Press (2007)
5. Bishop, C.M.: Pattern recognition and machine learning. Springer (2006)
6. Brémaud, P.: Markov chains: Gibbs fields, Monte Carlo simulation, and queues. Springer (1999)
7. Carreira-Perpiñán, M.Á., Hinton, G.E.: On contrastive divergence learning. In: 10th International Workshop on Artificial Intelligence and Statistics (AISTATS 2005), pp. 59–66 (2005)
8. Cho, K., Raiko, T., Ilin, A.: Parallel tempering is efficient for learning restricted Boltzmann machines. In: Proceedings of the International Joint Conference on Neural Networks (IJCNN 2010), pp. 3246–3253. IEEE Press (2010)
9. Desjardins, G., Courville, A., Bengio, Y.: Adaptive parallel tempering for stochastic maximum likelihood learning of RBMs. In: Lee, H., Ranzato, M., Bengio, Y., Hinton, G., LeCun, Y., Ng, A.Y. (eds.) NIPS 2010 Workshop on Deep Learning and Unsupervised Feature Learning (2010)
10. Desjardins, G., Courville, A., Bengio, Y., Vincent, P., Dellaleau, O.: Parallel tempering for training of restricted Boltzmann machines. In: JMLR Workshop and Conference Proceedings: AISTATS 2010, vol. 9, pp. 145–152 (2010)
11. Fischer, A., Igel, C.: Empirical Analysis of the Divergence of Gibbs Sampling Based Learning Algorithms for Restricted Boltzmann Machines. In: Diamantaras, K., Duch, W., Iliadis, L.S. (eds.) ICANN 2010, Part III. LNCS, vol. 6354, pp. 208–217. Springer, Heidelberg (2010)
12. Fischer, A., Igel, C.: Bounding the bias of contrastive divergence learning. Neural Computation 23, 664–673 (2011)
13. Geman, S., Geman, D.: Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. IEEE Transactions on Pattern Analysis and Machine Intelligence 6, 721–741 (1984)
14. Hastings, W.K.: Monte Carlo sampling methods using Markov chains and their applications. Biometrika 57(1), 97–109 (1970)
15. Hinton, G.E.: Training products of experts by minimizing contrastive divergence. Neural Computation 14, 1771–1800 (2002)
16. Hinton, G.E.: Boltzmann machine. Scholarpedia 2(5), 1668 (2007)
17. Hinton, G.E.: Learning multiple layers of representation. Trends in Cognitive Sciences 11(10), 428–434 (2007)
18. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Computation 18(7), 1527–1554 (2006)
19. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science 313(5786), 504–507 (2006)
20. Igel, C., Glasmachers, T., Heidrich-Meisner, V.: Shark. Journal of Machine Learning Research 9, 993–996 (2008)
21. Kivinen, J., Williams, C.: Multiple texture boltzmann machines. In: JMLR Workshop and Conference Proceedings: AISTATS 2012, vol. 22, pp. 638–646 (2012)
22. Koller, D., Friedman, N.: Probabilistic graphical models: Principles and techniques. MIT Press (2009)
23. Lauritzen, S.L.: Graphical models. Oxford University Press (1996)
24. Le Roux, N., Bengio, Y.: Representational power of restricted Boltzmann machines and deep belief networks. Neural Computation 20(6), 1631–1649 (2008)
25. Le Roux, N., Heess, N., Shotton, J., Winn, J.M.: Learning a generative model of images by factoring appearance and shape. Neural Computation 23(3), 593–650 (2011)

26. Lingenheil, M., Denschlag, R., Mathias, G., Tavan, P.: Efficiency of exchange schemes in replica exchange. Chemical Physics Letters 478, 80–84 (2009)
27. MacKay, D.J.C.: Failures of the one-step learning algorithm. Cavendish Laboratory, Madingley Road, Cambridge CB3 0HE, UK (2001), http://www.cs.toronto.edu/~mackay/gbm.pdf
28. MacKay, D.J.C.: Information Theory, Inference & Learning Algorithms. Cambridge University Press (2002)
29. Mnih, V., Larochelle, H., Hinton, G.: Conditional restricted Boltzmann machines for structured output prediction. In: Cozman, F.G., Pfeffer, A. (eds.) Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI 2011), p. 514. AUAI Press (2011)
30. Montufar, G., Ay, N.: Refinements of universal approximation results for deep belief networks and restricted Boltzmann machines. Neural Comput. 23(5), 1306–1319
31. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In: Rumelhart, D.E., McClelland, J.L. (eds.) Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1: Foundations, pp. 318–362. MIT Press (1986)
32. Salakhutdinov, R., Hinton, G.E.: Deep Boltzmann machines. In: JMLR Workshop and Conference Proceedings: AISTATS 2009, vol. 5, pp. 448–455 (2009)
33. Salakhutdinov, R.: Learning in Markov random fields using tempered transitions. In: Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C.K.I., Culotta, A. (eds.) Advances in Neural Information Processing Systems 22, pp. 1598–1606 (2009)
34. Smolensky, P.: Information processing in dynamical systems: Foundations of harmony theory. In: Rumelhart, D.E., McClelland, J.L. (eds.) Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1: Foundations, pp. 194–281. MIT Press (1986)
35. Taylor, G.W., Hinton, G.E., Roweis, S.T.: Modeling human motion using binary latent variables. In: Schölkopf, B., Platt, J., Hoffman, T. (eds.) Advances in Neural Information Processing Systems (NIPS 19), pp. 1345–1352. MIT Press (2007)
36. Tieleman, T.: Training restricted Boltzmann machines using approximations to the likelihood gradient. In: Cohen, W.W., McCallum, A., Roweis, S.T. (eds.) International Conference on Machine learning (ICML), pp. 1064–1071. ACM (2008)
37. Tieleman, T., Hinton, G.E.: Using fast weights to improve persistent contrastive divergence. In: Pohoreckyj Danyluk, A., Bottou, L., Littman, M.L. (eds.) International Conference on Machine Learning (ICML), pp. 1033–1040. ACM (2009)
38. Wang, N., Melchior, J., Wiskott, L.: An analysis of Gaussian-binary restricted Boltzmann machines for natural images. In: Verleysen, M. (ed.) European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN), pp. 287–292. d-side publications, Evere (2012)
39. Welling, M.: Product of experts. Scholarpedia 2(10), 3879 (2007)
40. Welling, M., Rosen-Zvi, M., Hinton, G.: Exponential family harmoniums with an application to information retrieval. In: Saul, L.K., Weiss, Y., Bottou, L. (eds.) Advances in Neural Information Processing Systems (NIPS 17), pp. 1481–1488. MIT Press, Cambridge (2005)
41. Younes, L.: Maximum likelihood estimation of Gibbs fields. In: Possolo, A. (ed.) Proceedings of an AMS-IMS-SIAM Joint Conference on Spacial Statistics and Imaging. Lecture Notes Monograph Series, Institute of Mathematical Statistics, Hayward (1991)
42. Yuille, A.L.: The convergence of contrastive divergence. In: Saul, L., Weiss, Y., Bottou, L. (eds.) Advances in Neural Processing Systems (NIPS 17), pp. 1593–1600. MIT Press (2005)