

Mathematical Foundations of Machine Learning

Lectures on YouTube:
<https://www.youtube.com/@mathtalent>

Seongjai Kim

Department of Mathematics and Statistics
Mississippi State University
Mississippi State, MS 39762 USA
Email: skim@math.msstate.edu

Updated: April 5, 2023

Seongjai Kim, Professor of Mathematics, Department of Mathematics and Statistics, Mississippi State University, Mississippi State, MS 39762 USA. Email: skim@math.msstate.edu.

Prologue

In organizing this lecture note, I am indebted by the following:

- S. RASCHKA AND V. MIRJALILI, *Python Machine Learning, 3rd Ed.*, 2019 [62].
- (Lecture note) <http://fa.bianp.net/teaching/2018/eecs227at/>, Dr. Fabian Pedregosa, UC Berkeley
- (Lecture note) **Introduction To Machine Learning**, Prof. David Sontag, MIT & NYU
- (Lecture note) **Mathematical Foundations of Machine Learning**, Dr. Justin Romberg, Georgia Tech

This lecture note will grow up as time marches; various core algorithms, useful techniques, and interesting examples would be soon incorporated.

Seongjai Kim
April 5, 2023

Contents

Title	ii
Prologue	iii
Table of Contents	ix
1 Introduction	1
1.1. Why and What in Machine Learning (ML)?	2
1.1.1. Inference problems	2
1.1.2. Modeling	3
1.1.3. Machine learning examples	4
1.2. Three Different Types of ML	5
1.2.1. Supervised learning	6
1.2.2. Unsupervised learning	11
1.3. A Machine Learning Modelcode	12
Exercises for Chapter 1	15
2 Python Basics	17
2.1. Why Python?	18
2.2. Python in 30 Minutes	20
2.2.1. Python essentials	21
2.2.2. Frequently used Python rules	23
2.2.3. Looping and functions	25
2.3. Zeros of Polynomials in Python	26
2.4. Python Classes	30
Exercises for Chapter 2	36
3 Simple Machine Learning Algorithms for Classification	37
3.1. Binary Classifiers – Artificial Neurons	38
3.2. The Perceptron Algorithm	40
3.2.1. The perceptron: A formal definition	40
3.2.2. The perceptron learning rule	41
3.2.3. Problems with the perceptron algorithm	44
3.3. Adaline: ADaptive LInear NEuron	47
3.3.1. Minimizing the cost function with the gradient descent method	48

3.3.2. Convergence and optimization issues with the gradient descent method	49
3.3.3. Feature scaling and stochastic gradient descent	50
Exercises for Chapter 3	53
4 Gradient-based Methods for Optimization	55
4.1. Gradient Descent Method	56
4.1.1. The gradient descent method in 1D	59
4.1.2. The full gradient descent algorithm	61
4.1.3. Surrogate minimization: A unifying principle	65
4.2. Newton's Method	67
4.2.1. Derivation	67
4.2.2. Hessian and principal curvatures	70
4.3. Quasi-Newton Methods	72
4.4. The Stochastic Gradient Method	76
4.5. The Levenberg–Marquardt Algorithm, for Nonlinear Least-Squares Problems	81
4.5.1. The gradient descent method	82
4.5.2. The Gauss-Newton method	83
4.5.3. The Levenberg-Marquardt algorithm	84
Exercises for Chapter 4	86
5 Popular Machine Learning Classifiers	89
5.1. Logistic Sigmoid Function	91
5.1.1. The standard logistic sigmoid function	91
5.1.2. The logit function	93
5.2. Classification via Logistic Regression	95
5.2.1. The logistic cost function	96
5.2.2. Gradient descent learning for logistic regression	99
5.2.3. Regularization: bias-variance tradeoff	100
5.3. Support Vector Machine	102
5.3.1. Linear SVM	102
5.3.2. The method of Lagrange multipliers	104
5.3.3. Karush-Kuhn-Tucker conditions and Complementary slackness	107
5.3.4. The inseparable case: Soft-margin classification	112
5.3.5. Nonlinear SVM and kernel trick	116
5.3.6. Solving the dual problem with SMO	120
5.4. Decision Trees	122
5.4.1. Decision tree objective	123
5.4.2. Random forests: Multiple decision trees	127
5.5. k -Nearest Neighbors	129
Exercises for Chapter 5	131
6 Data Preprocessing in Machine Learning	133

6.1. General Remarks on Data Preprocessing	134
6.2. Dealing with Missing Data & Categorical Data	136
6.2.1. Handling missing data	136
6.2.2. Handling categorical data	137
6.3. Feature Scaling	138
6.4. Feature Selection	140
6.4.1. Selecting meaningful variables	140
6.4.2. Sequential backward selection (SBS)	142
6.4.3. Ridge regression vs. LASSO	143
6.5. Feature Importance	146
Exercises for Chapter 6	148
7 Feature Extraction: Data Compression	149
7.1. Principal Component Analysis	150
7.1.1. Computation of principal components	151
7.1.2. Dimensionality reduction	153
7.1.3. Explained variance	155
7.2. Singular Value Decomposition	156
7.2.1. Interpretation of the SVD	158
7.2.2. Properties of the SVD	162
7.2.3. Computation of the SVD	165
7.2.4. Application of the SVD to image compression	170
7.3. Linear Discriminant Analysis	172
7.3.1. Fisher's LDA (classifier): two classes	173
7.3.2. Fisher's LDA: the optimum projection	175
7.3.3. LDA for multiple classes	178
7.3.4. The LDA: dimensionality reduction	186
7.4. Kernel Principal Component Analysis	189
7.4.1. Principal components of the kernel PCA	190
7.4.2. Computation of the kernel PCA	193
Exercises for Chapter 7	196
8 Cluster Analysis	199
8.1. Basics for Cluster Analysis	200
8.1.1. Quality of clustering	201
8.1.2. Types of clusters	204
8.1.3. Types of clustering and Objective functions	208
8.2. K-Means and K-Medoids Clustering	211
8.2.1. The (basic) K-Means clustering	211
8.2.2. Bisecting K-Means algorithm	217
8.2.3. The K-Medoids algorithm	221
8.2.4. CLARA and CLARANS	222

8.3. Hierarchical Clustering	224
8.3.1. Basics of AGNES and DIANA	224
8.3.2. AGNES: Agglomerative clustering	226
8.4. DBSCAN: Density-based Clustering	231
8.5. Cluster Validation	236
8.5.1. Basics of cluster validation	236
8.5.2. Internal and external measures of cluster validity	240
8.6. Self-Organizing Maps	246
8.6.1. Basics of the SOM	246
8.6.2. Kohonen SOM networks	250
8.6.3. The SOM algorithm and its interpretation	255
Exercises for Chapter 8	259
9 Neural Networks and Deep Learning	261
9.1. Basics for Deep Learning	262
9.2. Neural Networks	266
9.2.1. Sigmoid neural networks	268
9.2.2. A simple network to classify handwritten digits	270
9.2.3. Implementing a network to classify digits [56]	274
9.2.4. Toward deep neural networks	278
9.3. Back-Propagation	281
9.3.1. Notations	281
9.3.2. The cost function	283
9.3.3. The four fundamental equations behind the back-propagation	284
9.4. Deep Learning: Convolutional Neural Networks	288
9.4.1. Introducing convolutional networks	289
9.4.2. CNNs, in practice	295
Exercises for Chapter 9	297
10 Data Mining	299
10.1. Introduction to Data Mining	300
10.2. Vectors and Matrices in Data Mining	304
10.2.1. Examples	304
10.2.2. Data compression: Low rank approximation	308
10.3. Text Mining	312
10.3.1. Vector space model: Preprocessing and query matching	313
10.3.2. Latent Semantic Indexing	319
10.4. Eigenvalue Methods in Data Mining	322
10.4.1. Pagerank	323
10.4.2. The Google matrix	327
10.4.3. Solving the Pagerank equation	329
Exercises for Chapter 10	332

11 Quadratic Programming	333
11.1. Equality Constrained Quadratic Programming	334
11.2. Direct Solution for the KKT System	339
11.2.1. Symmetric factorization	339
11.2.2. Range-space approach	341
11.2.3. Null-space approach	343
11.3. Linear Iterative Methods	344
11.3.1. Convergence theory	345
11.3.2. Graph theory: Estimation of the spectral radius	345
11.3.3. Eigenvalue locus theorem	347
11.3.4. Regular splitting and M-matrices	349
11.4. Iterative Solution of the KKT System	350
11.4.1. Krylov subspace methods	350
11.4.2. The transforming range-space iteration	351
11.5. Active Set Strategies for Convex QP Problems	352
11.5.1. Primal active set strategy	353
11.6. Interior-point Methods	355
11.7. Logarithmic Barriers	357
Exercises for Chapter 11	360
 P Projects	 363
P.1. mCLESS	364
P.1.1. Review: Simple classifiers	365
P.1.2. The mCLESS classifier	366
P.1.3. Feature expansion	370
P.2. Gaussian Sailing to Overcome Local Minima Problems	375
P.3. Quasi-Newton Methods Using Partial Information of the Hessian	377
P.4. Effective Preprocessing Technique for Filling Missing Data	379
 Bibliography	 381
 Index	 387

CHAPTER 1

Introduction

What are we “learning” in **Machine Learning (ML)**?

This is a hard question to which we can only give a somewhat fuzzy answer. But at a high enough level of abstraction, there are two answers:

- **Algorithms**, which solve some kinds of inference problems
- **Models** for datasets.

These answers are so abstract that they are probably completely unsatisfying. But let’s (start to) clear things up, by looking at some particular examples of “inference” and “modeling” problems.

Contents of Chapter 1

1.1. Why and What in Machine Learning (ML)?	2
1.2. Three Different Types of ML	5
1.3. A Machine Learning Modelcode	12
Exercises for Chapter 1	15

1.1. Why and What in Machine Learning (ML)?

1.1.1. Inference problems

Definition 1.1. **Statistical inference** is the process of using data analysis to deduce properties of an underlying probability distribution. Inferential statistical analysis infers properties of a population, for example *by testing hypotheses and deriving estimates*.

Loosely speaking, inference problems take in data, then output some kind of decision or estimate. The output of a statistical inference is a statistical proposition; here are some common forms of statistical proposition.

- a point estimate
- an interval estimate
- a credible interval
- rejection of a hypothesis
- **classification** or **clustering** of the data points into discrete groups

Example 1.2. Inference algorithms can answer the following.

(a) Does this image have a tree in it?



(b) What words are in this picture?



(c) If I give you a recording of somebody speaking, can you produce text of what they are saying?

Remark 1.3. What does a machine learning algorithm do?

Machine learning algorithms are not algorithms for performing inference. Rather, **they are algorithms for building inference algorithms from examples**. An inference algorithm takes a piece of data and outputs a decision (or a probability distribution over the decision space).

1.1.2. Modeling

A second type of problem associated with **machine learning** (ML) might be roughly described as:

*Given a dataset, how can I **succinctly describe it** (in a quantitative, mathematical manner)?*

One example is **regression analysis**. Most models can be broken into two categories:

- **Geometric models**. The general problem is that we have example data points

$$\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N \in \mathbb{R}^D$$

and we want **to find some kind of geometric structure** that (approximately) describes them.

Here is an example: given a set of vectors, what (low dimensional) subspace comes closest to containing them?

- **Probabilistic models**. The basic task here is **to find a probability distribution** that describes the dataset $\{\mathbf{x}_n\}$.

The classical name for this problem is **density estimation** – given samples of a random variable, estimate its probability density function (pdf). This gets extremely tricky in high dimensions (large values of D) or when there are dependencies between the data points. Key to solving these problems is **choosing the right way to describe your probability model**.

Note: In both cases above, **having a concise model** can go a tremendous way towards analyzing the data.

- **As a rule, if you have a simple and accurate model, this is tremendously helping in solving inference problems, because there are fewer parameters to consider and/or estimate.**
- **The categories can either overlap with or complement each other.** It is often the case that the same model can be interpreted as a *geometric* model or a *probabilistic* model.

1.1.3. Machine learning examples

- **Classification:** from data to discrete classes
 - Spam filtering
 - Object detection (e.g., face)
 - Weather prediction (e.g., rain, snow, sun)
- **Regression:** predicting a numeric value
 - Stock market
 - Weather prediction (e.g., Temperature)
- **Ranking:** comparing items
 - Web search (keywords)
 - Given an image, find similar images
- **Collaborative Filtering** (e.g., Recommendation systems)
- **Clustering:** discovering structure in data
 - Clustering points or images
 - Clustering web search results
- **Embedding:** visualizing data
 - Embedding images, words
- **Structured prediction:** from data to discrete classes
 - Speech/image recognition
 - Natural language processing

1.2. Three Different Types of ML

Example 1.4. Three different types of ML:

- Supervised learning: classification, regression
- Unsupervised learning: clustering
- Reinforcement learning: chess engine

Supervised Learning	> Labeled data > Direct feedback > Predict outcome/future
Unsupervised Learning	> No labels > No feedback > Find hidden structure in data
Reinforcement Learning	> Decision process > Reward system > Learn series of actions

Figure 1.1: Three different types of ML (by methods)

1.2.1. Supervised learning

Assumption. Given a data set $\{(x_i, y_i)\}$, $\exists$ a relation $f : X \rightarrow Y$.

Supervised learning:

$$\begin{cases} \text{Given : Training set } \{(x_i, y_i) \mid i = 1, \dots, N\} \\ \text{Find : } \hat{f} : X \rightarrow Y, \text{ a good approximation to } f \end{cases} \quad (1.1)$$

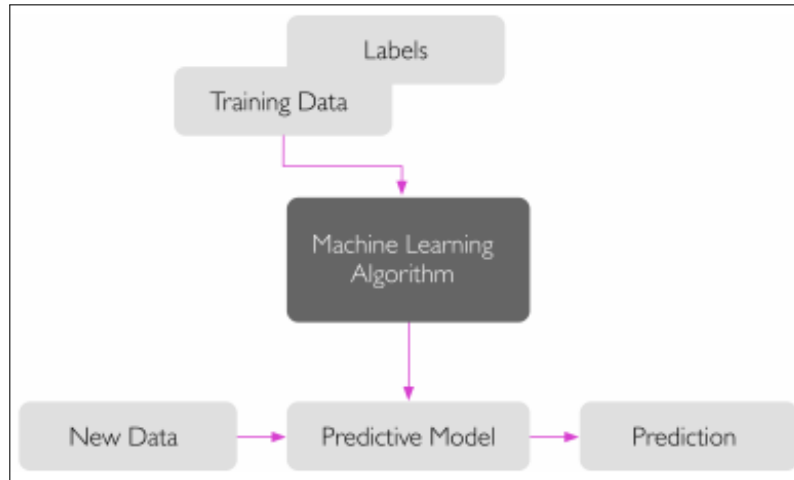


Figure 1.2: Supervised learning and prediction.

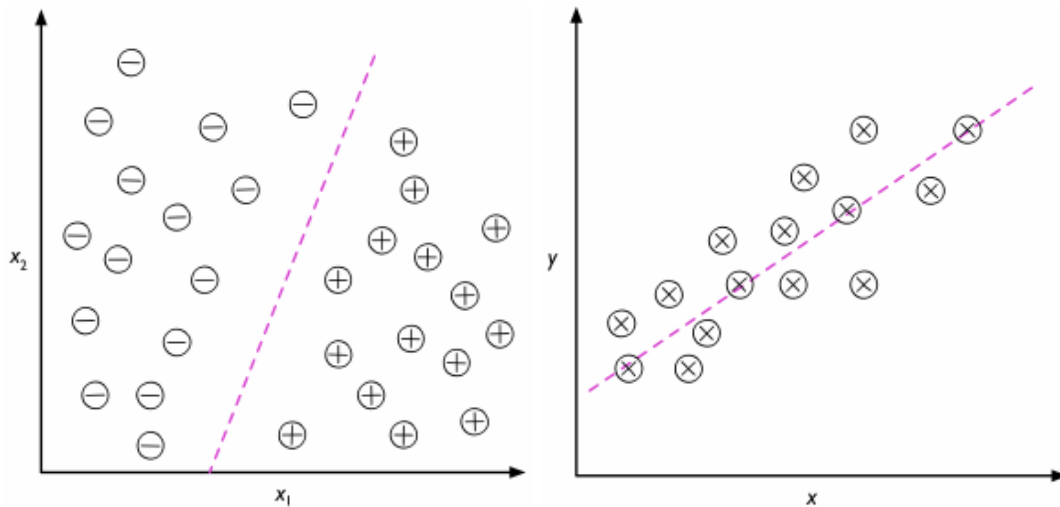
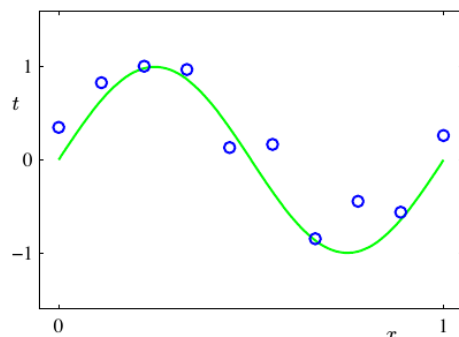


Figure 1.3: Classification and regression.

Example 1.5. Regression

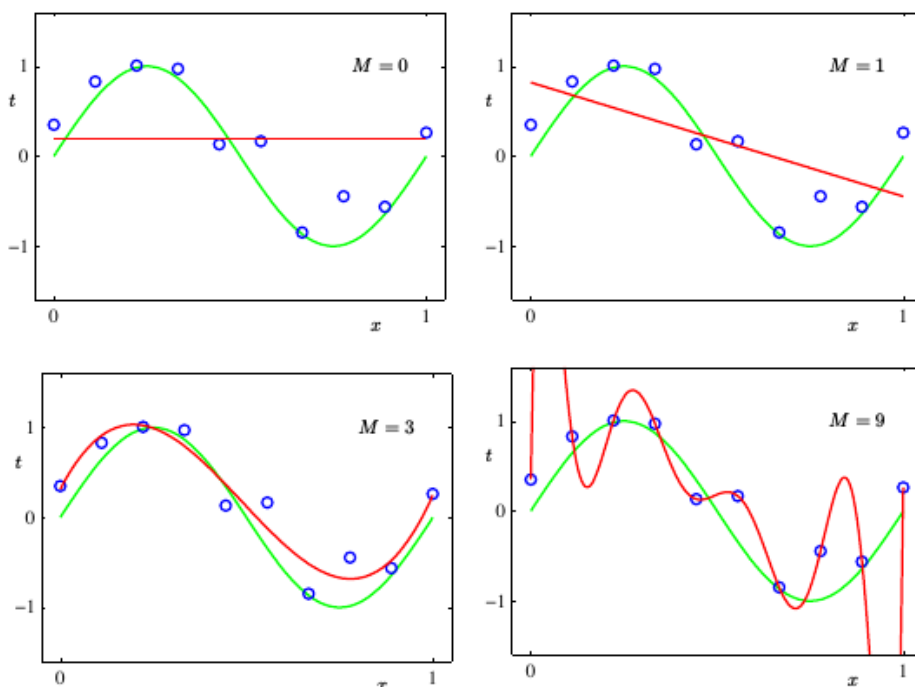
- **Dataset:** Consider a simple dataset: 10 points generated from a sin function, with noise



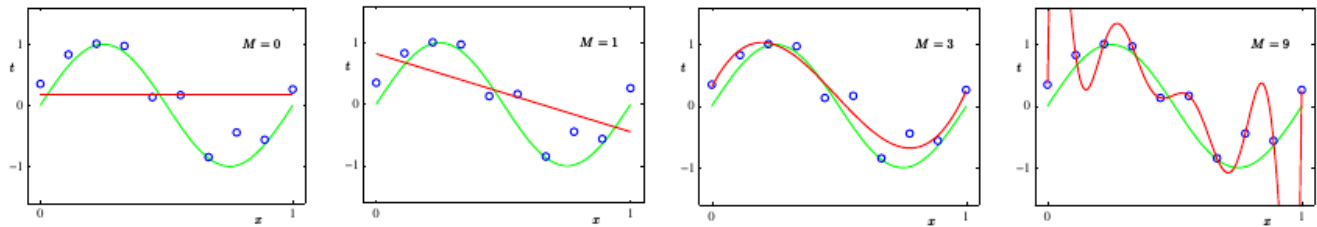
- **Wanted:** Find a regression model

$$f : X = \mathbb{R} \rightarrow Y = \mathbb{R} \quad (1.2)$$

- **Question:** How should we pick the **hypothesis space**, the set of possible functions f ?
 - Often, there exist infinitely many hypotheses
 - Let's select f from $\mathbb{P}_M$, polynomials of degree $\leq M$.
 - Which one is best?



• **Error: misfit**



- We measure the error using a **loss function** $L(y, \hat{y})$.
- For regression, a common choice is squared loss: for each (x_i, y_i) ,

$$L(y_i, f(x_i)) = (y_i - f(x_i))^2.$$

- The **empirical loss** of the function f applied to the training data is then:

$$E_{MS} \stackrel{\text{def}}{=} \frac{1}{N} \sum_{i=1}^N L(y_i, f(x_i)) = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))^2, \quad (1.3)$$

which is the **mean square error**.

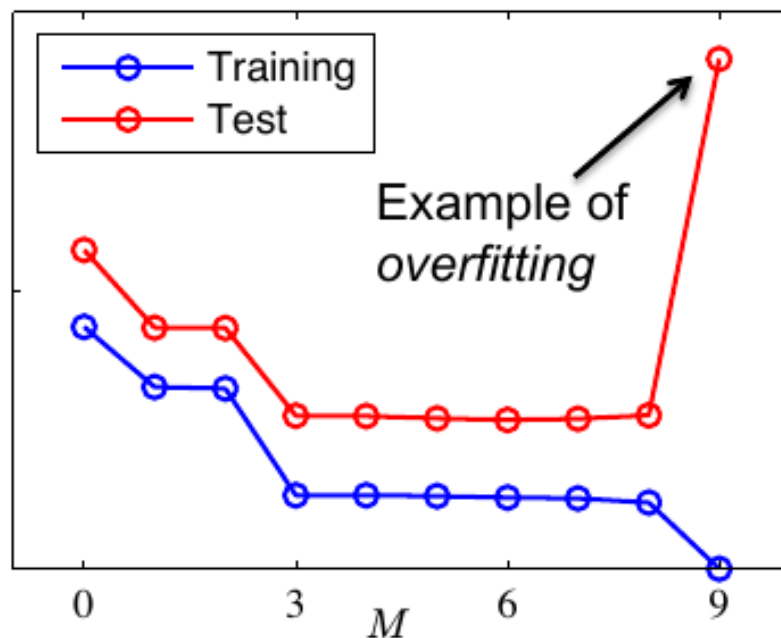


Figure 1.4: Learning curve.

Key Idea 1.6. Training and test performance. Assume that each training and test example–label pair (x, y) is drawn independently at random from the same (but unknown) population of examples and labels. Represent this population as a probability distribution $p(x, y)$, so that:

$$(x_i, y_i) \sim p(x, y).$$

- Then, given a loss function L :
 - Empirical (**training**) loss = $\frac{1}{N} \sum_{i=1}^N L(y_i, f(x_i))$.
(Also called the **empirical risk**, $\hat{R}(f, D_N)$.)
 - Expected (**test**) loss = $E_{(x,y) \sim p} \{L(y, f(x))\}$.
(Also called the **risk** $R(f)$.)
- **Ideally, learning chooses the hypothesis that minimizes the risk.**
 - But this is impossible to compute!
 - The empirical risk is a good (unbiased) estimate of the risk (by linearity of expectation).
- **The principle of empirical risk minimization** reads

$$f^*(D_N) = \arg \min_{f \in \mathcal{F}} \hat{R}(f, D_N). \quad (1.4)$$

Remark 1.7. Key Issues in Machine Learning

- How do we choose **a hypothesis space**?
 - Often we use **prior knowledge** to guide this choice
 - The ability to answer to the next two questions also affects the choice
- How can we **gauge the accuracy of a hypothesis** on unseen testing data?
 - The previous example showed that choosing the hypothesis which simply **minimizes training set error** (i.e., empirical risk minimization) **is not optimal**.
 - This question is **a main topic in learning theory**.
- How do we find **the best hypothesis**?
 - This is an **algorithmic** question, at the intersection of mathematics, computer science, and optimization research.

Proposition 1.8. Occam's Razor Principle (a.k.a. Law of parsimony):

“One should not increase, beyond what is necessary, the number of entities required to explain anything.”

- **William of Occam**: A monk living in the 14-th century, England
- When **many** solutions are available for a given problem, we should select the **simplest** one.
- But what do we mean by **simple**?
- We will use **prior knowledge** of the problem to solve to define what is a simple solution (Example of a prior: smoothness).

1.2.2. Unsupervised learning

Note:

- In supervised learning, we know the right answer beforehand when we train our model, and in reinforcement learning, we define a measure of reward for particular actions by the agent.
- In unsupervised learning, however, we are dealing with **unlabeled data** or **data of unknown structure**. Using unsupervised learning techniques, we are able **to explore the structure of our data** to **extract meaningful information** without the guidance of a known outcome variable or reward function.
- **Clustering** is an exploratory data analysis technique that allows us to organize a pile of information into **meaningful subgroups** (clusters) without having any prior knowledge of their group memberships.

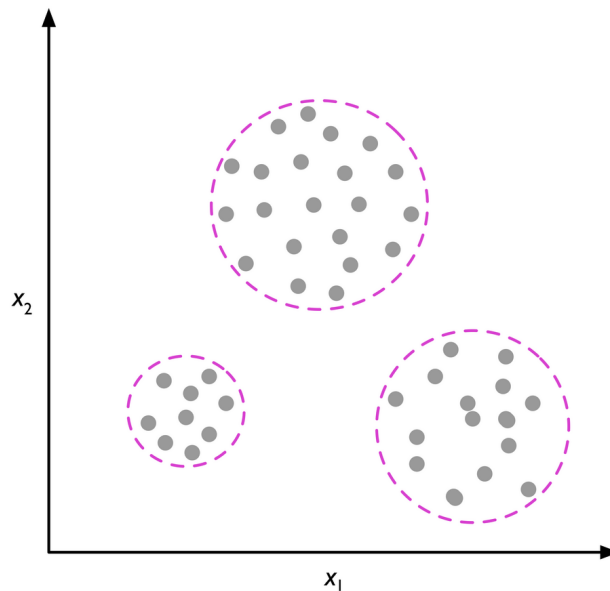


Figure 1.5: Clustering.

1.3. A Machine Learning Modelcode

A code for machine learning can start with the following **machine learning modelcode**. You may copy-and-paste the scripts to run.

```

Machine_Learning_Model.py
1  import numpy as np;      import pandas as pd
2  import seaborn as sbn; import matplotlib.pyplot as plt
3  import time
4  from sklearn.model_selection import train_test_split
5  from sklearn import datasets; #print(dir(datasets))
6  np.set_printoptions(suppress=True)
7
8  #=====
9  # DATA: Read & Preprocessing
10 # load_iris, load_wine, load_breast_cancer, ...
11 #=====
12 data_read = datasets.load_iris(); #print(data_read.keys())
13
14 X = data_read.data
15 y = data_read.target
16 dataname = data_read.filename
17 targets  = data_read.target_names
18 features = data_read.feature_names
19
20 print('X.shape=',X.shape, 'y.shape=',y.shape)
21 #-----
22 # SETTING
23 #-----
24 N,d = X.shape; labelset=set(y)
25 nclass=len(labelset);
26 print('N,d,nclass=',N,d,nclass)
27
28 rtrain = 0.7e0; run = 100
29 rtest  = 1-rtrain
30
31 #=====
32 # CLASSIFICATION
33 #=====
34 btime = time.time()
35 Acc = np.zeros([run,1])
36 ##from sklearn.neighbors import KNeighborsClassifier
37 ##clf = KNeighborsClassifier(5)
38 from myCLF import myCLF    ## My classifier

```

```

39
40 for it in range(run):
41     Xtrain, Xtest, ytrain, ytest = train_test_split(
42         X, y, test_size=rtest, random_state=it, stratify = y)
43     ##clf.fit(Xtrain, ytrain);
44     clf = myCLF(Xtrain,ytrain); clf.fit();    ## My classifier
45     Acc[it] = clf.score(Xtest, ytest)
46
47 #-----
48 # Print: Accuracy && E-time
49 #-----
50 etime = time.time()-btime
51 print('    %s: Acc.(mean,std) = (%.2f,%.2f)%%; Average E-time= %.5f'
52       %(dataname,np.mean(Acc)*100,np.std(Acc)*100,etime/run))
53
54 #=====
55 # Scikit-learn Classifiers, for Comparisions
56 #=====
57 exec(open("sklearn_classifiers.py").read())

```

```

                                sklearn_classifiers.py
1  #=====
2  # Required: X, y, [dataname, run]
3  print('===== Scikit-learn Classifiers, for Comparisions =====')
4  #=====
5  from sklearn.preprocessing import StandardScaler
6  from sklearn.datasets import make_moons, make_circles, make_classification
7  from sklearn.neural_network import MLPClassifier
8  from sklearn.neighbors import KNeighborsClassifier
9  from sklearn.linear_model import LogisticRegression
10 from sklearn.svm import SVC
11 from sklearn.gaussian_process import GaussianProcessClassifier
12 from sklearn.gaussian_process.kernels import RBF
13 from sklearn.tree import DecisionTreeClassifier
14 from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
15 from sklearn.naive_bayes import GaussianNB
16 from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
17 #from sklearn.inspection import DecisionBoundaryDisplay
18
19 #-----
20 classifiers = [
21     LogisticRegression(max_iter = 1000),
22     KNeighborsClassifier(5),
23     SVC(kernel="linear", C=0.5),

```

```

24     SVC(gamma=2, C=1),
25     RandomForestClassifier(max_depth=5, n_estimators=50, max_features=1),
26     MLPClassifier(alpha=1, max_iter=1000),
27     AdaBoostClassifier(),
28     GaussianNB(),
29     QuadraticDiscriminantAnalysis(),
30     GaussianProcessClassifier(),
31 ]
32 names = [
33     "Logistic Regr",
34     "KNeighbors-5 ",
35     "Linear SVM   ",
36     "RBF SVM      ",
37     "Random Forest",
38     "Deep-NN      ",
39     "AdaBoost     ",
40     "Naive Bayes  ",
41     "QDA          ",
42     "Gaussian Proc",
43 ]
44 #-----
45 if dataname is None: dataname = 'No-dataname';
46 if run      is None: run      = 100;
47
48 #=====
49 import os; acc_max=0
50 for name, clf in zip(names, classifiers):
51     Acc = np.zeros([run,1])
52     btime = time.time()
53
54     for it in range(run):
55         Xtrain, Xtest, ytrain, ytest = train_test_split(
56             X, y, test_size=rtest, random_state=it, stratify = y)
57
58         clf.fit(Xtrain, ytrain);
59         Acc[it] = clf.score(Xtest, ytest)
60
61     etime = time.time()-btime
62     accmean = np.mean(Acc)*100
63     print('%s: %s: Acc.(mean,std) = (%.2f,%.2f)%%; E-time= %.5f'
64           %(os.path.basename(dataname),name,accmean,np.std(Acc)*100,etime/run))
65     if accmean>acc_max:
66         acc_max= accmean; alname = name
67 print('sklearn classifiers max: %s= %.2f' %(alname,acc_max))

```

Exercises for Chapter 1

- 1.1. The code in Section 1.3 will run without requiring any other implementation of yours, if (1) uncomment lines 36-37 and 43 and (2) comment lines 38 and 44.
- (a) Save the code written in two files, by copy-and-paste.
 - (b) Install all imported packages to run the code.
 - (c) Report the results.

Installation: If you are on Ubuntu, you may begin with

```
Install-Python-packages
1  sudo apt update
2  sudo apt install python3 -y
3  sudo apt install python3-pip -y
4  rehash
5  sudo pip3 install numpy scipy matplotlib sympy -y
6  sudo pip3 install sklearn seaborn pandas -y
```


CHAPTER 2

Python Basics

Contents of Chapter 2

2.1. Why Python?	18
2.2. Python in 30 Minutes	20
2.3. Zeros of Polynomials in Python	26
2.4. Python Classes	30
Exercises for Chapter 2	36

2.1. Why Python?

Note: A good programming language must be **easy to learn and use** and **flexible and reliable**.

Advantages of Python

Python has the following characteristics.

- Easy to learn and use
- Flexible and reliable
- Extensively used in **Data Science**
- Handy for **Web Development** purposes
- Having **Vast Libraries** support
- Among the **fastest-growing** programming languages in the tech industry

Disadvantage of Python

Python is an interpreted and dynamically-typed language. The line-by-line execution of code, built with a high flexibility, most likely leads to **slow execution**. **Python** is slower than **Matlab** that is slower than **C**.

Remark 2.1. Speed up Python Programs.

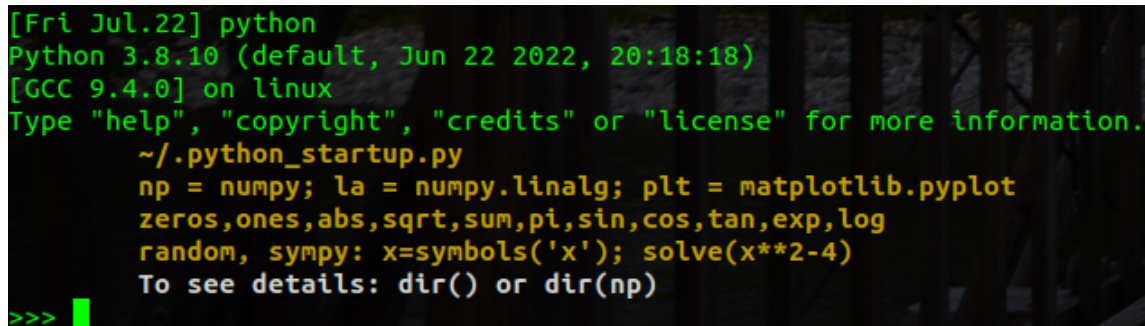
- Use **numpy** and **scipy** for all mathematical operations.
 - Always use a **C library** wherever possible.
-
- You yourself may create and import your own C-module into Python. If you extend Python with pieces of **compiled C-code**, then the resulting code is easily **100× faster than Python**. **Best choice!**
 - **Cython**: It is designed as **a C-extension for Python**, which is developed for users not familiar with C. For Cython implementation, see e.g. <https://www.youtube.com/watch?v=JKMkhARcwdU>, one of Simon Funke's YouTube videos.

- The library `numpy` is designed for a **Matlab-like implementation**.
- Python can be used as a convenient **desktop calculator**.
 - First, set a startup environment
 - Use Python as a desktop calculator

```

~/.python_startup.py
1  #.bashrc: export PYTHONSTARTUP=~/.python_startup.py
2  #.cshrc:  setenv PYTHONSTARTUP ~/.python_startup.py
3  #-----
4  print("\t^[[1;33m~/.python_startup.py")
5
6  import numpy as np
7  import numpy.linalg as la
8  import matplotlib.pyplot as plt
9  print("\tnp = numpy; la = numpy.linalg; plt = matplotlib.pyplot")
10
11 from numpy import zeros,ones,abs,sqrt,sum,pi,sin,cos,tan,exp,log
12 print("\tzeros,ones,abs,sqrt,sum,pi,sin,cos,tan,exp,log")
13
14 import random, sympy
15 from sympy import symbols,solve #x=symbols('x'); solve(x**2-4)
16 print("\trandom, sympy: x=symbols('x'); solve(x**2-4)")
17
18 print("\t^[[1;37mTo see details: dir() or dir(np)^[[m")

```



```

[Fri Jul.22] python
Python 3.8.10 (default, Jun 22 2022, 20:18:18)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
~/.python_startup.py
np = numpy; la = numpy.linalg; plt = matplotlib.pyplot
zeros,ones,abs,sqrt,sum,pi,sin,cos,tan,exp,log
random, sympy: x=symbols('x'); solve(x**2-4)
To see details: dir() or dir(np)
>>>

```

Figure 2.1: Python startup.

2.2. Python in 30 Minutes

Key Features of Python

- Python is a **simple, readable, open source** programming language which is easy to learn.
- It is an **interpreted** language, not a compiled language.
- In Python, **variables are untyped**; i.e., there is no need to define/declare the data type of a variable before using it.
- Python supports the **object-oriented programming** model.
- It is **platform-independent** and easily extensible and embeddable.
- It has a **huge standard library** with lots of modules and packages.
- Python is a **high level language** as it is easy to use because of simple syntax, **powerful** because of its rich libraries and extremely versatile.

Programming Features

- Python does **not support pointers**.
- Python codes are stored with **.py** extension.
- **Indentation**: Python uses indentation to define a block of code.
 - A **code block** (body of a function, loop, etc.) starts with indentation and ends with the first unindented line.
 - The amount of indentation is up to the user, but it must be consistent throughout that block.
- **Comments**:
 - The hash (#) symbol is used to start writing a comment.
 - **Multi-line comments**: Python uses triple quotes, either ''' or """.

2.2.1. Python essentials

- **Sequence datatypes:** list, tuple, string
 - **[list]:** defined using square brackets (and commas)

```
>>> li = ["abc", 14, 4.34, 23]
```
 - **(tuple):** defined using parentheses (and commas)

```
>>> tu = (23, (4,5), 'a', 4.1, -7)
```
 - **"string":** defined using quotes (" , ' , or """)

```
>>> st = 'Hello World'
>>> st = "Hello World"
>>> st = """This is a multi-line string
... that uses triple quotes."""
```
- **Retrieving elements**

```
>>> li[0]
'abc'
>>> tu[1],tu[2],tu[-2]
((4, 5), 'a', 4.1)
>>> st[25:36]
'ng\nthat use'
```
- **Slicing**

```
>>> tu[1:4] # be aware
((4, 5), 'a', 4.1)
```
- **The + and * operators**

```
>>> [1, 2, 3]+[4, 5, 6,7]
[1, 2, 3, 4, 5, 6, 7]
>>> "Hello" + " " + 'World'
Hello World
>>> (1,2,3)*3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

- **Reference semantics**

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```

Be aware with copying lists and numpy arrays!

- **numpy, range, and iteration**

```
>>> range(8)
[0, 1, 2, 3, 4, 5, 6, 7]
>>> import numpy as np
>>> for k in range(np.size(li)):
...     li[k]
... <Enter>
'abc'
14
4.34
23
```

- **numpy array and deepcopy**

```
>>> from copy import deepcopy
>>> A = np.array([1,2,3])
>>> B = A
>>> C = deepcopy(A)
>>> A *= 4
>>> B
array([ 4, 8, 12])
>>> C
array([1, 2, 3])
```

2.2.2. Frequently used Python rules

```
frequently_used_rules.py
1  ## Multi-line statement
2  a = 1 + 2 + 3 + 4 + 5 +\
3      6 + 7 + 8 + 9 + 10
4  b = (1 + 2 + 3 + 4 + 5 +
5      6 + 7 + 8 + 9 + 10) #inside (), [], or {}
6  print(a,b)
7  # Output: 55 55
8
9  ## Multiple statements in a single line using ";"
10 a = 1; b = 2; c = 3
11
12 ## Docstrings in Python
13 def double(num):
14     """Function to double the value"""
15     return 2*num
16 print(double.__doc__)
17 # Output: Function to double the value
18
19 ## Assigning multiple values to multiple variables
20 a, b, c = 1, 2, "Hello"
21 ## Swap
22 b, c = c, b
23 print(a,b,c)
24 # Output: 1 Hello 2
25
26 ## Data types in Python
27 a = 5; b = 2.1
28 print("type of (a,b)", type(a), type(b))
29 # Output: type of (a,b) <class 'int'> <class 'float'>
30
31 ## Python Set: 'set' object is not subscriptable
32 a = {5,2,3,1,4}; b = {1,2,2,3,3,3}
33 print("a=",a,"b=",b)
34 # Output: a= {1, 2, 3, 4, 5} b= {1, 2, 3}
```

```
35
36 ## Python Dictionary
37 d = {'key1':'value1', 'Seth':22, 'Alex':21}
38 print(d['key1'],d['Alex'],d['Seth'])
39 # Output: value1 21 22
40
41 ## Output Formatting
42 x = 5.1; y = 10
43 print('x = %d and y = %d' %(x,y))
44 print('x = %f and y = %d' %(x,y))
45 print('x = {} and y = {}'.format(x,y))
46 print('x = {1} and y = {0}'.format(x,y))
47 # Output: x = 5 and y = 10
48 #           x = 5.100000 and y = 10
49 #           x = 5.1 and y = 10
50 #           x = 10 and y = 5.1
51
52 print("x=",x,"y=",y, sep="#",end="&\n")
53 # Output: x=#5.1#y=#10&
54
55 ## Python Input
56 C = input('Enter any: ')
57 print(C)
58 # Output: Enter any: Starkville
59 #           Starkville
```

2.2.3. Looping and functions

Example 2.2. Compose a Python function which returns cubes of natural numbers.

Solution.

```

1  def get_cubes(num):
2      cubes = []
3      for i in range(1,num+1):
4          value = i**3
5          cubes.append(value)
6      return cubes
7
8  if __name__ == '__main__':
9      num = input('Enter a natural number: ')
10     cubes = get_cubes(int(num))
11     print(cubes)

```

Remark 2.3. `get_cubes.py`

- Lines 8-11 are added for the function to be called directly. That is,
 [Fri Jul.22] python get_cubes.py
 Enter a natural number: 6
 [1, 8, 27, 64, 125, 216]
- When `get_cubes` is called from another function, the last four lines will not be executed.

```

1  from get_cubes import *
2
3  cubes = get_cubes(8)
4  print(cubes)

```

```

1  [Fri Jul.22] python call_get_cubes.py
2  [1, 8, 27, 64, 125, 216, 343, 512]

```

2.3. Zeros of Polynomials in Python

In this section, a code in Python will be implemented to find zeros of polynomials; we will compare the Python code with a Matlab code.

Recall: Let's begin with recalling how to find zeros of polynomials.

- When the **Newton's method** is applied for an approximate zero of $P(x)$, the iteration reads

$$x_{n+1} = x_n - \frac{P(x_n)}{P'(x_n)}. \quad (2.1)$$

Thus both $P(x)$ and $P'(x)$ must be evaluated in each iteration.

- **The derivative $P'(x)$ can be evaluated by using the Horner's method with the same efficiency.** Indeed,
 - The **remainder theorem** says

$$P(x) = (x - x_n)Q(x) + P(x_n). \quad (2.2)$$

- Its derivative reads

$$P'(x) = Q(x) + (x - x_n)Q'(x). \quad (2.3)$$

- Thus

$$P'(x_n) = Q(x_n). \quad (2.4)$$

That is, the evaluation of Q at x_n becomes the desired quantity $P'(x_n)$.

Note: The **Horner's method** is also called the **synthetic division**.

Example 2.4. Let $P(x) = x^4 - 4x^3 + 7x^2 - 5x - 2$. Use the Newton's method and the Horner's method to implement a code and find an approximate zero of P near 3.

Solution. First, let's try to use built-in functions.

```

zeros_of_poly_built_in.py
1  import numpy as np
2
3  coeff = [1, -4, 7, -5, -2]
4  P = np.poly1d(coeff)
5  Pder = np.polyder(P)
6
7  print(P)
8  print(Pder)
9  print(np.roots(P))
10 print(P(3), Pder(3))

```

```

Output
1      4      3      2
2  1 x - 4 x + 7 x - 5 x - 2
3      3      2
4  4 x - 12 x + 14 x - 5
5  [ 2.+0.j  1.1378411+1.52731225j  1.1378411-1.52731225j -0.2756822+0.j ]
6  19 37

```

Now, we implement a code for Newton-Horner method to find an approximate zero of P near 3.

```

Zeros-Polynomials-Newton-Horner.py
1  def horner(A,x0):
2      """ input:  A = [a_n,...,a_1,a_0]
3          output: p,d = P(x0),DP(x0) = horner(A,x0) """
4      n = len(A)
5      p = A[0]; d = 0
6
7      for i in range(1,n):
8          d = p + x0*d
9          p = A[i] +x0*p
10     return p,d
11
12 def newton_horner(A,x0,tol,itmax):
13     """ input:  A = [a_n,...,a_1,a_0]
14         output: x: P(x)=0 """
15     x=x0

```

```

16     for it in range(1,itmax+1):
17         p,d = horner(A,x)
18         h = -p/d;
19         x = x + h;
20         if(abs(h)<tol): break
21     return x,it
22
23 if __name__ == '__main__':
24     coeff = [1, -4, 7, -5, -2]; x0 = 3
25     tol = 10**(-12); itmax = 1000
26     x,it =newton_horner(coeff,x0,tol,itmax)
27     print("newton_horner: x0=%g; x=%g, in %d iterations" %(x0,x,it))

```

Execution

```

1 [Sat Jul.23] python Zeros-Polynomials-Newton-Horner.py
2 newton_horner: x0=3; x=2, in 7 iterations

```

Note: The above Python code must be compared with the Matlab code below.

horner.m

```

1 function [p,d] = horner(A,x0)
2 %   input:  A = [a_0,a_1,...,a_n]
3 %   output: p=P(x0), d=P'(x0)
4
5 n = size(A(:),1);
6 p = A(n); d=0;
7
8 for i = n-1:-1:1
9     d = p + x0*d;
10    p = A(i) +x0*p;
11 end

```

newton_horner.m

```

1 function [x,it] = newton_horner(A,x0,tol,itmax)
2 %   input:  A = [a_0,a_1,...,a_n]; x0: initial for P(x)=0
3 %   outpue: x: P(x)=0
4
5 x = x0;
6 for it=1:itmax
7     [p,d] = horner(A,x);
8     h = -p/d;
9     x = x + h;
10    if(abs(h)<tol), break; end
11 end

```

```
Call_newton_horner.m
1  a = [-2 -5 7 -4 1];
2  x0=3;
3  tol = 10^-12; itmax=1000;
4  [x,it] = newton_horner(a,x0,tol,itmax);
5  fprintf("  newton_horner: x0=%g; x=%g, in %d iterations\n",x0,x,it)
6  Output:  newton_horner: x0=3; x=2, in 7 iterations
```

Observation 2.5.

Python programming is as easy and simple as Matlab programming.

- In particular, numpy is developed for **Matlab-like implementation, with enhanced convenience**.
- Python uses **classes** for object-oriented programming.
- Furthermore, Python is an **open source (free)** programming language, which explains why Python is **fastest-growing in use**.

2.4. Python Classes

Remark 2.6. Classes are a key concept in the so-called **object-oriented programming (OOP)**. **Classes provide a means of bundling data and functionality together.**

- A **class** is a user-defined template or prototype from which real-world objects are created.
- A class tells us what data an object should have, what are the initial/default values of the data, and what methods are associated with the object to take actions on the objects using their data.
- An object is an **instance** of a class, and creating an object from a class is called **instantiation**.

In the following, we would build a simple class, as Dr. Xu did in [83, Appendix B.5]; you will learn how to **initiate, refine, and use classes**.

```
Polynomial_01.py
1 class Polynomial():
2     """A class of polynomials"""
3
4     def __init__(self,coefficient):
5         """Initialize coefficient attribute of a polynomial."""
6         self.coeff = coefficient
7
8     def degree(self):
9         """Find the degree of a polynomial"""
10        return len(self.coeff)-1
11
12 if __name__ == '__main__':
13     p2 = Polynomial([1,2,3])
14     print(p2.coeff)      # a variable; output: [1, 2, 3]
15     print(p2.degree())  # a method;   output: 2
```

- **Lines 1-2:** define a class called Polynomial with a docstring.
 - The parentheses in the class definition are empty because we create this class from scratch.
- **Lines 4-10:** define two functions, `__init__()` and `degree()`. A function in a class is called a **method**.
 - The `__init__()` method is a special method for initialization; it is called the `__init__()` **constructor**.
 - The `self` parameter is required and must come first before the other parameters in each method.
 - Whenever we make an object from the class, we need to provide arguments for parameters, except for `self`.
 - The variable `self.coeff` (prefixed with `self`) is available to every method and is accessible by any object created from the class.
 - Variables prefixed with `self` are called **attributes**.
- **Line 13:** The line `p2 = Polynomial([1,2,3])` creates an object `p2` (a polynomial $x^2 + 2x + 3$), by passing the coefficient list `[1,2,3]`.
 - When Python reads this line, it calls the method `__init__()` in the class `Polynomial` and creates the object named `p2` that represents this particular polynomial $x^2 + 2x + 3$.

Refinement of the Polynomial class

```

_____ Polynomial_02.py _____
1  class Polynomial():
2      """A class of polynomials"""
3
4      count = 0          #a class attribute
5
6      def __init__(self): #constructor
7          """Initialize coefficient attribute of a polynomial."""
8          self.coeff = [1]
9          Polynomial.count += 1
10
11     def __del__(self):  #destructor
12         """Delete a polynomial object"""
13         Polynomial.count -= 1
14         print('Destructor called, a Polynomial object deleted.')
15
16     def degree(self):
17         """Find the degree of a polynomial"""
18         return len(self.coeff)-1
19
20     def evaluate(self,x):
21         """Evaluate a polynomial."""
22
23         n = self.degree()
24         eval = []
25         for xi in x:
26             p = self.coeff[0]      #Horner's method
27             for k in range(1,n+1):
28                 p = self.coeff[k]+ xi*p
29             eval.append(p)
30         return eval
31
32 if __name__ == '__main__':
33     poly1 = Polynomial()
34     print('poly1, default coefficients:', poly1.coeff)
35     poly1.coeff = [1,2,-3]
36     print('poly1, coefficients after reset:', poly1.coeff)
37     print('poly1, degree:', poly1.degree())
38
39     poly2 = Polynomial()
40     poly2.coeff = [1,2,3,4,-5]
41     print('poly2, coefficients after reset:', poly2.coeff)
42     print('poly2, degree:', poly2.degree())

```

```
43     print('number of created polynomials:', Polynomial.count)
44     del poly1
45     print('number of polynomials after a deletion:', Polynomial.count)
46
47     print('poly2.evaluate([-1,0,1,2]):',poly2.evaluate([-1,0,1,2]))
48
```

- **Line 4:** The variable `count` is a class attribute of `Polynomial`.
 - A **class attribute** is a variable that belongs to a class but not a particular object.
 - All objects of the class share this same variable (the class attribute).
- **Line 8:** initializes the class attribute `self.coeff`.
 - Every object or class attribute in a class needs an initial value.
 - One can set a default value for an object attribute in the `__init__()` constructor, and we then do not have to include a parameter for that attribute in the constructor. See Lines 32 and 38.
- **Lines 11-14:** define the `__del__()` method, the **destructor**, for the deletion of objects.
- **Lines 20-30:** define another method called `evaluate`, which uses the *Horner's method*. See Example 2.4, p.27.

Execution

```
1 [Tue Jan.24] python Polynomial_02.py
2 poly1, default coefficients: [1]
3 poly1, coefficients after reset: [1, 2, -3]
4 poly1, degree: 2
5 poly2, coefficients after reset: [1, 2, 3, 4, -5]
6 poly2, degree: 4
7 number of created polynomials: 2
8 Destructor called, a Polynomial object deleted.
9 number of polynomials after a deletion: 1
10 poly2.evaluate([-1,0,1,2]): [-7, -5, 5, 47]
11 Destructor called, a Polynomial object deleted.
```

Inheritance

Note: If we want to write a class that is just a *specialized version of another class*, we do not need to write the class from scratch.

- We call the specialized class a **child class** and the other general class a **parent class**.
- The child class can inherit all the attributes and methods from the parent class; it can also define its own special attributes and methods or even overrides methods of the parent class.

Classes.py

```

1 class Polynomial():
2     """A class of polynomials"""
3
4     def __init__(self,coefficient):
5         """Initialize coefficient attribute of a polynomial."""
6         self.coeff = coefficient
7
8     def degree(self):
9         """Find the degree of a polynomial"""
10        return len(self.coeff)-1
11
12 class Quadratic(Polynomial):
13     """A class of quadratic polynomial"""
14
15     def __init__(self,coefficient):
16         """Initialize the coefficient attributes ."""
17         super().__init__(coefficient)
18         self.power_decrease = 1
19
20     def roots(self):
21         a,b,c = self.coeff
22         if self.power_decrease != 1:
23             a,c = c,a
24         discriminant = b**2-4*a*c
25         r1 = (-b+discriminant**0.5)/(2*a)
26         r2 = (-b-discriminant**0.5)/(2*a)
27         return [r1,r2]
28
29     def degree(self):
30         return 2

```


- **Line 12:** We must include the name of the parent class in the parentheses of the definition of the child class (to indicate the parent-child relation for inheritance).
- **Line 17:** The `super()` function is to give an child object all the attributes defined in the parent class.
- **Line 18:** An additional child class attribute `self.power_decrease` is initialized.
- **Lines 20-27:** define a new method called `roots`.
- **Lines 29-30:** The method `degree()` overrides the parent's method.

```
call_Quadratic.py
1 from Classes import *
2
3 quad1 = Quadratic([2,-3,1])
4 print('quad1, roots:',quad1.roots())
5 quad1.power_decrease = 0
6 print('roots when power_decrease = 0:',quad1.roots())
7 # Output: quad1, roots: [1.0, 0.5]
8 #           roots when power_decrease = 0: [2.0, 1.0]
```

Python Keywords

Keywords are some predefined and reserved words in python.

Ex.: and, or, not, if, else, elif, for, while, break, del, ...

Exercises for Chapter 2

You should use Python for the following problems.

- 2.1. Use nested for loops to assign entries of a 5×5 matrix A such that $A[i, j] = ij$.
- 2.2. The variable d is initially equal to 1. Use a while loop to keep dividing d by 2 until $d < 10^{-6}$.
 - (a) Determine how many divisions are made.
 - (b) Verify your result by algebraic derivation.

Note: A while loop has not been considered in the lecture. However, you can figure it out easily by yourself.

- 2.3. Write a function that takes as input a list of values and returns the largest value. Do this without using the Python `max()` function; you should combine a for loop and an if statement.
 - (a) Produce a random list of size 10-20 to verify your function.
- 2.4. Let $P_4(x) = 2x^4 - 5x^3 - 11x^2 + 20x + 10$. Solve the following.
 - (a) Plot P_4 over the interval $[-3, 4]$.
 - (b) Find all zeros of P_4 , modifying `Zeros-Polynomials-Newton-Horner.py`, p.27.
 - (c) Add markers for the zeros to the plot.
 - (d) Find all roots of $P'_4(x) = 0$.
 - (e) Add markers for the zeros of P'_4 to the plot.

Do **not** use built-in functions, for (b) and (d).

Hint: For plotting, you may import: “import matplotlib.pyplot as plt” then use `plt.plot()`. You will see the Python plotting is quite similar to Matlab plotting.

CHAPTER 3

Simple Machine Learning Algorithms for Classification

In this chapter, we will make use of one of the first algorithmically described machine learning algorithms for classification, the **perceptron** and **adaptive linear neurons** (*adaline*). We will start by implementing a perceptron step by step in Python and training it to classify different flower species in the Iris dataset.

Contents of Chapter 3

3.1. Binary Classifiers – Artificial Neurons	38
3.2. The Perceptron Algorithm	40
3.3. Adaline: ADaptive LInear NEuron	47
Exercises for Chapter 3	53

3.1. Binary Classifiers – Artificial Neurons

Definition 3.1. A **binary classifier** is a function which can decide whether or not an input vector belongs to some specific class (e.g., spam/ham).

- Binary classification often refers to those classification tasks that have two class labels. (**two-class classification**)
- It is a **type of linear classifier**, i.e. a classification algorithm that makes **its predictions based on a linear predictor function** combining a set of weights with the feature vector.
- Linear classifiers are **artificial neurons**.

Remark 3.2. **Neurons** are interconnected nerve cells that are involved in the processing and transmitting of chemical and electrical signals. Such a nerve cell can be described as a simple logic gate with binary outputs;

- multiple signals arrive at the dendrites,
- they are integrated into the cell body,
- and if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

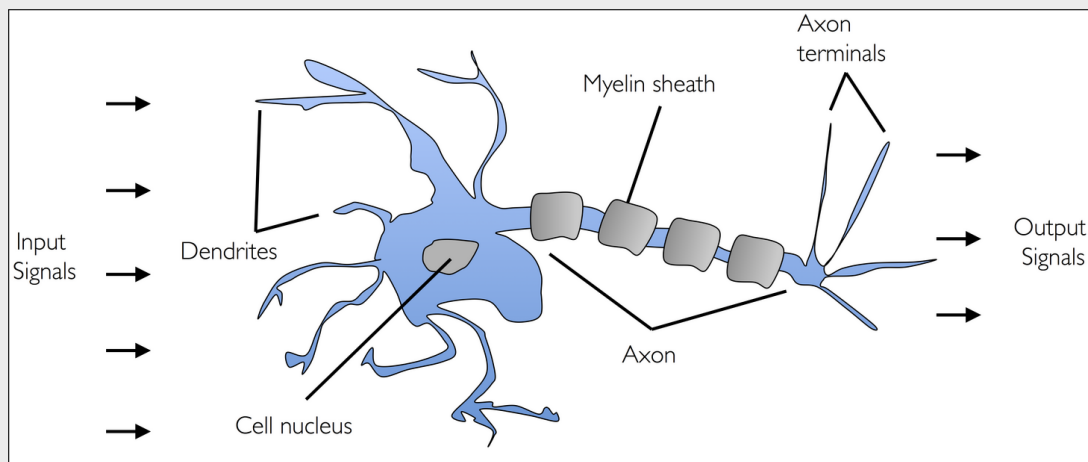


Figure 3.1: A schematic description of a neuron.

Linear classifiers

As artificial neurons, they have the following characteristics:

- Inputs are **feature values**: $\mathbf{x}$
- Each feature has a **weight**: $\mathbf{w}$
- Weighted sum (integration) is the **activation**

$$\text{activation}_{\mathbf{w}}(\mathbf{x}) = \sum_j w_j x_j = \mathbf{w} \cdot \mathbf{x} \quad (3.1)$$

- **Decision/output**: If the activation is

$$\begin{cases} \text{Positive} & \Rightarrow \text{class 1} \\ \text{Negative} & \Rightarrow \text{class 2} \end{cases}$$

Unknowns, in ML:

$$\begin{cases} \text{Training :} & \mathbf{w} \\ \text{Prediction :} & \text{activation}_{\mathbf{w}}(\mathbf{x}) \end{cases}$$

Examples:

- Perceptron
- Adaline (ADaptive LInear NEuron)
- Support Vector Machine (SVM) $\Rightarrow$ nonlinear decision boundaries, too

3.2. The Perceptron Algorithm

The **perceptron** is a binary classifier of supervised learning.

- 1957: Perceptron algorithm is invented by **Frank Rosenblatt**, Cornell Aeronautical Laboratory
 - Built on work of Hebb (1949)
 - Improved by Widrow-Hoff (1960): Adaline
- 1960: Perceptron Mark 1 Computer – hardware implementation
- 1970's: Learning methods for two-layer neural networks

3.2.1. The perceptron: A formal definition

Definition 3.3. We can pose the **perceptron** as a **binary classifier**, in which we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity.

- **Input values:** $\mathbf{x} = (x_1, x_2, \dots, x_m)^T$
- **Weight vector:** $\mathbf{w} = (w_1, w_2, \dots, w_m)^T$
- **Net input:** $z = w_1x_1 + w_2x_2 + \dots + w_mx_m$
- **Activation function:** $\phi(z)$, defined by

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise,} \end{cases} \quad (3.2)$$

where θ is a threshold.

For simplicity, we can bring the threshold θ in (3.2) to the left side of the equation; define a weight-zero as $w_0 = -\theta$ and reformulate as

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise,} \end{cases} \quad z = \mathbf{w}^T \mathbf{x} = w_0 + w_1x_1 + \dots + w_mx_m. \quad (3.3)$$

In the ML literature, the variable w_0 is called the **bias**.

The equation $w_0 + w_1x_1 + \dots + w_mx_m = 0$ represents a hyperplane in $\mathbb{R}^m$, while w_0 decides the intercept.

3.2.2. The perceptron learning rule

The whole idea behind the **Rosenblatt's thresholded perceptron model** is to use a reductionist approach to mimic how a single neuron in the brain works: it either fires or it doesn't.

Summary 3.4. *Thus, Rosenblatt's initial perceptron rule is fairly simple and can be summarized by the following steps:*

1. *Initialize the weights to 0 or small random numbers.*
2. *For each training sample $\mathbf{x}^{(i)}$,*
 - (a) *Compute the output value $\hat{y}^{(i)} (:= \phi(\mathbf{w}^T \mathbf{x}^{(i)}))$.*
 - (b) *Update the weights.*

The update of the weight vector $\mathbf{w}$ can be more formally written as:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \quad \Delta \mathbf{w} = \eta (y^{(i)} - \hat{y}^{(i)}) \mathbf{x}^{(i)}, \quad (3.4)$$

where η is the **learning rate**, $0 < \eta < 1$, $y^{(i)}$ is the true class label of the i -th training sample, and $\hat{y}^{(i)}$ denotes the predicted class label.

Remark 3.5. **A simple thought experiment for the perceptron learning rule:**

- Let the perceptron predict the class label correctly. Then $y^{(i)} - \hat{y}^{(i)} = 0$ so that the weights remain unchanged.
- Let the perceptron make a wrong prediction. Then

$$\Delta w_j = \eta (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)} = \pm 2 \eta x_j^{(i)}$$

so that the weight w_j is pushed towards the direction of the positive or negative target class, respectively.

Note: It is important to note that **convergence of the perceptron** is only guaranteed if the two classes are **linearly separable** and the **learning rate is sufficiently small**. If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (epochs) and/or a threshold for the number of tolerated misclassifications.

Definition 3.6. (Linearly separable dataset). A dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}$ is **linearly separable** if there exist $\hat{\mathbf{w}}$ and γ such that

$$y^{(i)} \hat{\mathbf{w}}^T \mathbf{x}^{(i)} \geq \gamma > 0, \quad \forall i, \quad (3.5)$$

where γ is called the **margin**.

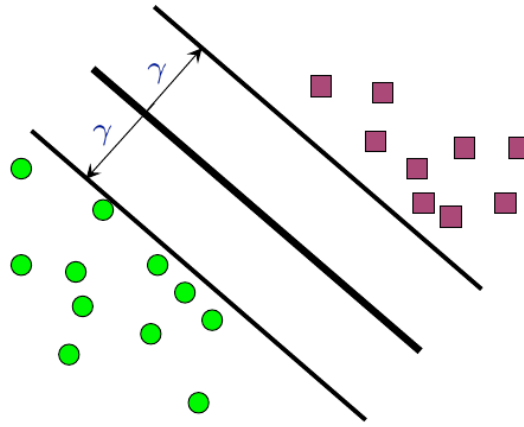


Figure 3.2: Linearly separable dataset.

Definition 3.7. (More formal/traditional definition). Let X and Y be two sets of points in an m -dimensional Euclidean space. Then X and Y are **linearly separable** if there exist $m + 1$ real numbers $w_1, w_2, \dots, w_m, k$ such that every point $\mathbf{x} \in X$ satisfies $\sum_{j=1}^m w_j x_j > k$ and every point $\mathbf{y} \in Y$ satisfies $\sum_{j=1}^m w_j y_j < k$.

Theorem 3.8. Assume the data set $D = \{(\mathbf{x}^{(i)}, y^{(i)})\}$ is linearly separable with margin γ , i.e.,

$$\exists \hat{\mathbf{w}}, \|\hat{\mathbf{w}}\| = 1, \quad y^{(i)} \hat{\mathbf{w}}^T \mathbf{x}^{(i)} \geq \gamma > 0, \quad \forall i. \quad (3.6)$$

Suppose that $\|\mathbf{x}^{(i)}\| \leq R$, $\forall i$, for some $R > 0$. Then, the maximum number of mistakes made by the perceptron algorithm is bounded by R^2/γ^2 .

Proof. Assume the perceptron algorithm makes yet a mistake for $(\mathbf{x}^{(\ell)}, y^{(\ell)})$. Then

$$\begin{aligned} \|\mathbf{w}^{(\ell+1)}\|^2 &= \|\mathbf{w}^{(\ell)} + \eta(y^{(\ell)} - \hat{y}^{(\ell)})\mathbf{x}^{(\ell)}\|^2 \\ &= \|\mathbf{w}^{(\ell)}\|^2 + \|\eta(y^{(\ell)} - \hat{y}^{(\ell)})\mathbf{x}^{(\ell)}\|^2 + 2\eta(y^{(\ell)} - \hat{y}^{(\ell)})\mathbf{w}^{(\ell)T}\mathbf{x}^{(\ell)} \\ &\leq \|\mathbf{w}^{(\ell)}\|^2 + \|\eta(y^{(\ell)} - \hat{y}^{(\ell)})\mathbf{x}^{(\ell)}\|^2 \leq \|\mathbf{w}^{(\ell)}\|^2 + (2\eta R)^2, \end{aligned} \quad (3.7)$$

where we have used

$$(y^{(\ell)} - \hat{y}^{(\ell)})\mathbf{w}^{(\ell)T}\mathbf{x}^{(\ell)} \leq 0. \quad (3.8)$$

(See Exercise 1.) The inequality (3.7) implies

$$\|\mathbf{w}^{(\ell)}\|^2 \leq \ell \cdot (2\eta R)^2. \quad (3.9)$$

(Here we have used $\|\mathbf{w}^{(0)}\| = 0$.) On the other hand,

$$\hat{\mathbf{w}}^T \mathbf{w}^{(\ell+1)} = \hat{\mathbf{w}}^T \mathbf{w}^{(\ell)} + \eta(y^{(\ell)} - \hat{y}^{(\ell)})\hat{\mathbf{w}}^T \mathbf{x}^{(\ell)} \geq \hat{\mathbf{w}}^T \mathbf{w}^{(\ell)} + 2\eta\gamma,$$

which implies

$$\hat{\mathbf{w}}^T \mathbf{w}^{(\ell)} \geq \ell \cdot (2\eta\gamma) \quad (3.10)$$

and therefore

$$\|\mathbf{w}^{(\ell)}\|^2 \geq \ell^2 \cdot (2\eta\gamma)^2. \quad (3.11)$$

It follows from (3.9) and (3.11) that $\ell \leq R^2/\gamma^2$. $\square$

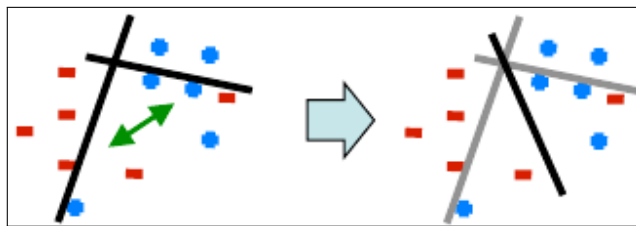
Properties of the perceptron algorithm: For a linearly separable training dataset,

- **Convergence:** The perceptron will converge.
- **Separability:** Some weights get the training set perfectly correct.

3.2.3. Problems with the perceptron algorithm

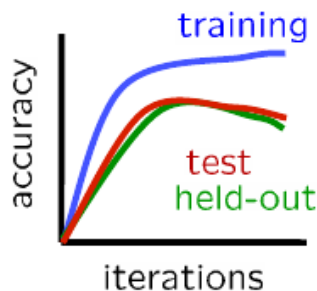
Inseparable Datasets

- If the data is **inseparable** (due to noise, for example), there is no guarantee for convergence or accuracy.
- **Averaged perceptron** is an algorithmic modification that helps with the issue.
 - Average the weight vectors, across all or a last part of iterations



Note: Frequently the training data *is* linearly separable! **Why?**

- For example, when the number of data points is much smaller than the number of features.
 - Perceptron can significantly **overfit** the data.
 - **An averaged perceptron** may help with this issue, too.



Definition 3.9. Hold-out Method: Hold-out is when you split up your dataset into a ‘train’ and ‘test’ set. The training set is what the model is trained on, and the test set is used to see how well that model performs on **unseen data**.

Optimal Separator?

Question. Which of these **linear separators** is optimal?

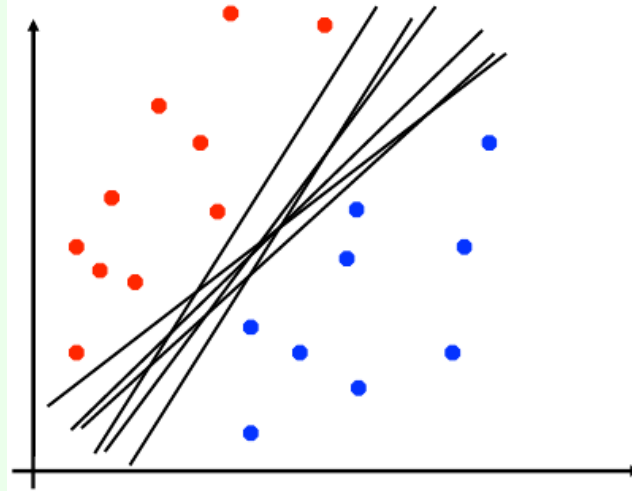


Figure 3.3

Example 3.10. Support Vector Machine (Cortes & Vapnik, 1995) chooses the linear separator with the **largest margin**.

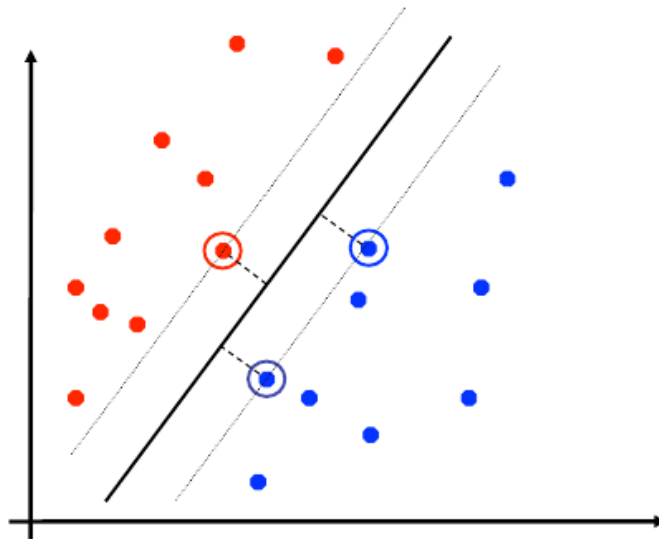


Figure 3.4

We will consider the SVM in Section 5.3.

How Multi-class Classification?

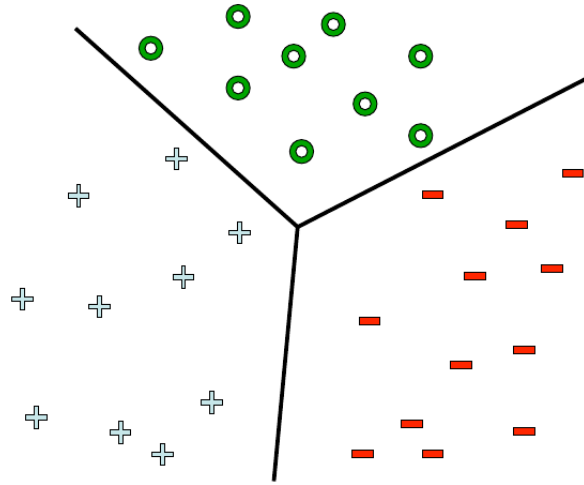
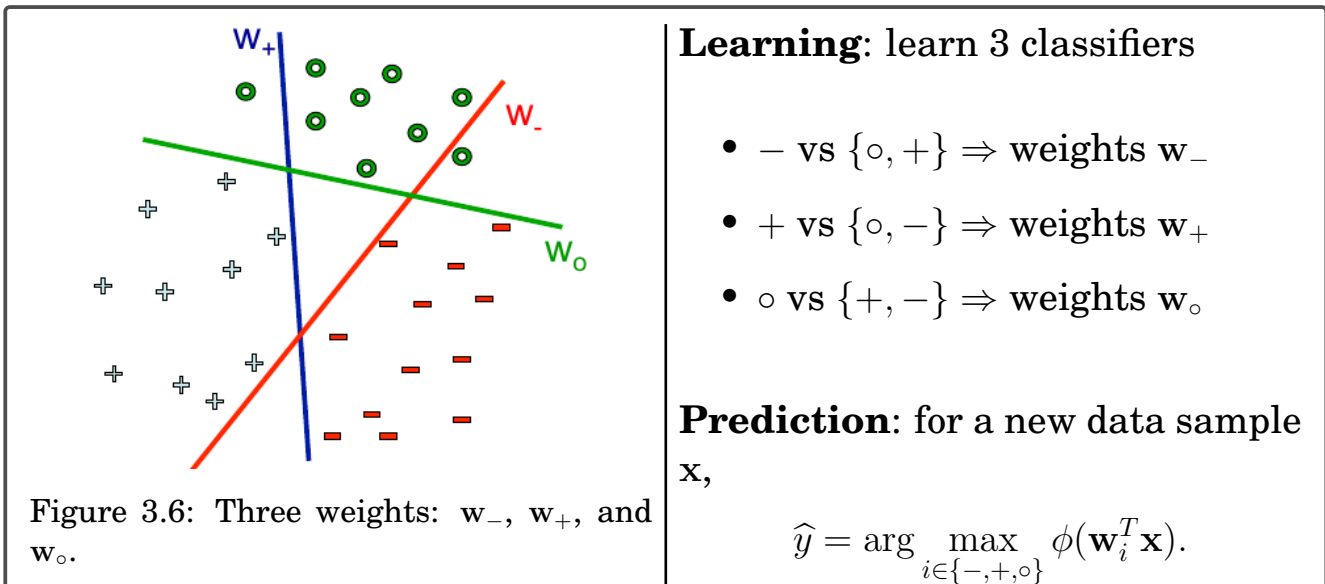


Figure 3.5: Classification for three classes.

One-versus-all (one-versus-all) classification



OVA (OVR) is readily applicable for classification of general n classes, $n \geq 2$.

3.3. Adaline: ADaptive LInear NEuron

- (Widrow & Hoff, 1960)
- Weights are updated based on linear activation: e.g.,

$$\phi(\mathbf{w}^T \mathbf{x}) = \mathbf{w}^T \mathbf{x}.$$

That is, ϕ is the **identity function**.

- Adaline algorithm is particularly interesting because it illustrates the key concept of defining and minimizing **continuous cost functions**, which will **lay the groundwork for understanding more advanced machine learning algorithms** for classification, such as logistic regression and support vector machines, as well as regression models.
- Continuous cost functions allow the ML optimization to incorporate **advanced mathematical techniques** such as **calculus**.

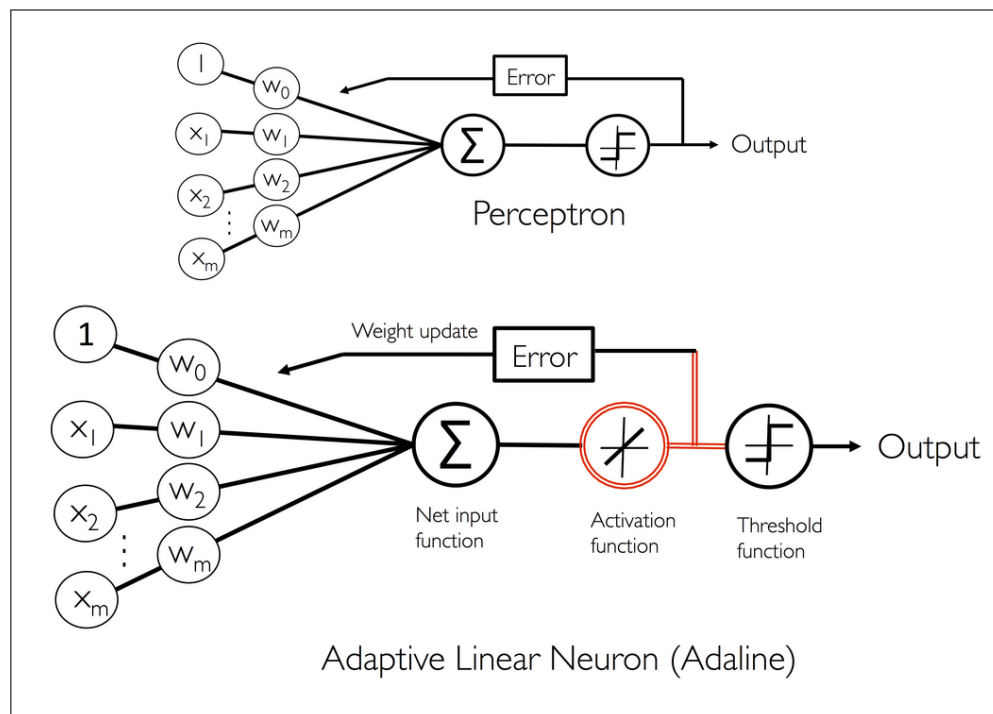


Figure 3.7: Perceptron vs. Adaline

3.3.1. Minimizing the cost function with the gradient descent method

Key Idea 3.11. One of the key ingredients of supervised machine learning algorithms is to define an **objective function** that is to be optimized during the learning process. This objective function is often a **cost function** that we want to minimize.

Definition 3.12. In the case of Adaline, we can define the cost function $\mathcal{J}$ to learn the weights as the **Sum of Squared Errors** (SSE) between the calculated outcomes and the true class labels:

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2} \sum_i \left(y^{(i)} - \phi(\mathbf{w}^T \mathbf{x}^{(i)}) \right)^2. \quad (3.12)$$

Algorithm 3.13. The Gradient Descent Method uses $-\nabla \mathcal{J}(\mathbf{w})$ for the **search direction** (update direction):

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w} = \mathbf{w} - \eta \nabla \mathcal{J}(\mathbf{w}), \quad (3.13)$$

where $\eta > 0$ is the **step length** (learning rate).

Computation of $\nabla \mathcal{J}(\mathbf{w})$ and $\Delta \mathbf{w}$:

The partial derivatives of the cost function $\mathcal{J}$ read

$$\frac{\partial \mathcal{J}(\mathbf{w})}{\partial w_j} = - \sum_i \left(y^{(i)} - \phi(\mathbf{w}^T \mathbf{x}^{(i)}) \right) x_j^{(i)}. \quad (3.14)$$

(Here we have used $\phi = I$.) Thus

$$\Delta \mathbf{w} = -\eta \nabla \mathcal{J}(\mathbf{w}) = \eta \sum_i \left(y^{(i)} - \phi(\mathbf{w}^T \mathbf{x}^{(i)}) \right) \mathbf{x}^{(i)}. \quad (3.15)$$

Note: The **above Adaline rule** is compared with the **perceptron rule** (3.4):

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \quad \Delta \mathbf{w} = \eta (y^{(i)} - \hat{y}^{(i)}) \mathbf{x}^{(i)}.$$

3.3.2. Convergence and optimization issues with the gradient descent method

- Depending on choices of certain **algorithmic parameters**, the gradient descent method may fail to converge to the the global minimizer.
- Data characteristics often determines both successability and speed of convergence; **data preprocessing** operations may improve convergence.
- For large-scale data, the gradient descent method is computationally expensive; a popular alternative is the **stochastic gradient descent method**.

◇ Hyperparameters

Definition 3.14. In ML, a **hyperparameter** is a parameter whose value is set before the learning process begins. Thus it is an **algorithmic parameter**. Examples are

- The learning rate (η)
- The number of maximum epochs/iterations (n_iter)

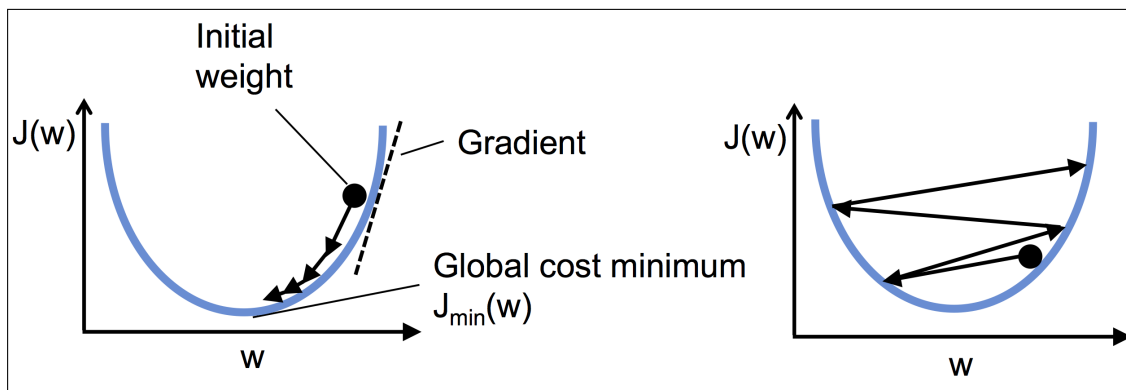


Figure 3.8: Well-chosen learning rate vs. a large learning rate

Hyperparameters must be selected to optimize the learning process:

- to converge **fast** to the global minimizer,
- avoiding overfit.

3.3.3. Feature scaling and stochastic gradient descent

Definition 3.15. Feature Scaling Preprocessing:

The gradient descent is one of the many algorithms that benefit from **feature scaling**. Here, we will consider a feature scaling method called **standardization**, which gives each feature of the data the property of a standard normal distribution.

- For example, to standardize the j -th feature, we simply need to subtract the sample mean μ_j from every training sample and divide it by its standard deviation σ_j :

$$\tilde{x}_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j}. \quad (3.16)$$

Then,

$$\{\tilde{x}_j^{(i)} \mid i = 1, 2, \dots, n\} \sim \mathcal{N}(0, 1). \quad (3.17)$$

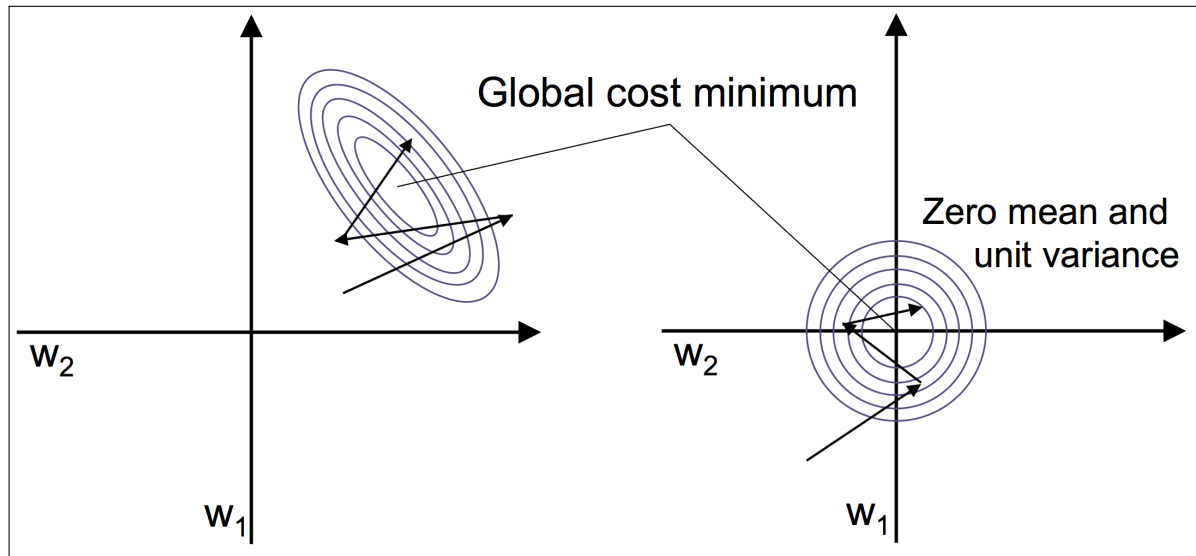


Figure 3.9: Standardization, which is one of **data normalization** techniques.

The gradient descent method has a tendency to converge faster with the standardized data.

Stochastic gradient descent method

Note: Earlier, we learned how to minimize a cost function with negative gradients that are calculated from the **whole training set**; this is why this approach is sometimes also referred to as **batch gradient descent**.

- Now imagine we have a very large dataset with millions of data points.
- Then, running with the gradient descent method can be computationally quite expensive, because we need to reevaluate the whole training dataset each time we take one step towards the global minimum.
- **A popular alternative** to the batch gradient descent algorithm is the **stochastic gradient descent (SGD)**.

Algorithm 3.16. The SGD method updates the weights incrementally for each training sample:

$$\begin{aligned}
 &\text{Given a training set } D = \{(\mathbf{x}^{(i)}, y^{(i)}) \mid i = 1, 2, \dots, n\} \\
 &\quad 1. \text{ For } i = 1, 2, \dots, n \\
 &\quad \quad \mathbf{w} = \mathbf{w} + \eta (y^{(i)} - \phi(\mathbf{w}^T \mathbf{x}^{(i)})) \mathbf{x}^{(i)}; \\
 &\quad 2. \text{ If not convergent, shuffle } D \text{ and goto 1;}
 \end{aligned} \tag{3.18}$$

- The SGD method updates the weights based on a single training example.
- The SGD method typically reaches **convergence much faster** because of the **more frequent weight updates**.
- Since each search direction is calculated based on a single training example, the error surface is **smoother** (not noisier) than in the gradient descent method; **the SGD method can escape shallow local minima more readily**.
- To obtain accurate results via the SGD method, it is important **to present it with data in a random order**, which may prevent cycles with epochs.
- In the SGD method, the learning rate η is often set **adaptively**, decreasing over iteration k . For example, $\eta_k = c_1 / (k + c_2)$.

◇ Mini-batch learning

Definition 3.17. A compromise between the batch gradient descent and the SGD is the so-called **mini-batch learning**. Mini-batch learning can be understood as applying batch gradient descent to smaller subsets of the training data – for example, 32 samples at a time.

The advantage over batch gradient descent is that convergence is reached faster via mini-batches because of the **more frequent weight updates**. Furthermore, mini-batch learning allows us to replace the for-loop over the training samples in stochastic gradient descent by **vectorized operations (vectorization)**, which can further improve the computational efficiency of our learning algorithm.

Exercises for Chapter 3

3.1. Verify (3.8).

Hint: We assumed that the parameter $\mathbf{w}^{(\ell)}$ gave a mistake on $\mathbf{x}^{(\ell)}$. For example, let $\mathbf{w}^{(\ell)T} \mathbf{x}^{(\ell)} \geq 0$. Then we **must** have $(y^{(\ell)} - \hat{y}^{(\ell)}) < 0$. Why?

3.2. Experiment all the examples on pp. 38–51, *Python Machine Learning, 3rd Ed.*. Through the examples, you will learn

- (a) Gradient descent rule for Adaline,
- (b) Feature scaling techniques, and
- (c) Stochastic gradient descent rule for Adaline.

To get the **Iris dataset**, you have to use some lines on as earlier pages from 31.

3.3. Perturb the dataset (X) by a random Gaussian noise G_σ of an observable σ (so as for $G_\sigma(X)$ not to be linearly separable) and do the examples in Exercise 3.2 again.

CHAPTER 4

Gradient-based Methods for Optimization

Optimization is the branch of research-and-development that aims to solve the problem of finding the elements which maximize or minimize a given real-valued function, while respecting constraints. Many problems in engineering and machine learning can be cast as optimization problems, which explains the growing importance of the field. An **optimization problem** is the problem of finding **the best solution** from all **feasible solutions**.

In this chapter, we will discuss details about

- Gradient descent method,
- Newton’s method, and
- Their variants.

Contents of Chapter 4

4.1. Gradient Descent Method	56
4.2. Newton’s Method	67
4.3. Quasi-Newton Methods	72
4.4. The Stochastic Gradient Method	76
4.5. The Levenberg–Marquardt Algorithm, for Nonlinear Least-Squares Problems	81
Exercises for Chapter 4	86

4.1. Gradient Descent Method

The first method that we will describe is one of the oldest methods in optimization: **gradient descent method**, a.k.a **steepest descent method**. The method was suggested by Augustin-Louis Cauchy in 1847 [47]. He was a French mathematician and physicist who made pioneering contributions to mathematical analysis. Motivated by the need to solve “large” quadratic problems (6 variables) that arise in Astronomy, he invented the method of gradient descent. Today, this method is used to comfortably solve problems with thousands of variables.



Figure 4.1: Augustin-Louis Cauchy

Problem 4.1. (Optimization Problem). Let $\Omega \subset \mathbb{R}^d$, $d \geq 1$. Given a real-valued function $f : \Omega \rightarrow \mathbb{R}$, the general problem of finding the value that minimizes f is formulated as follows.

$$\min_{\mathbf{x} \in \Omega} f(\mathbf{x}). \quad (4.1)$$

In this context, f is the **objective function** (sometimes referred to as **loss function** or **cost function**). $\Omega \subset \mathbb{R}^d$ is the **domain** of the function (also known as the **constraint set**).

Example 4.2. (Rosenbrock function). For example, the **Rosenbrock function** in the two-dimensional (2D) space is defined as¹

$$f(x, y) = (1 - x)^2 + 100 (y - x^2)^2. \quad (4.2)$$

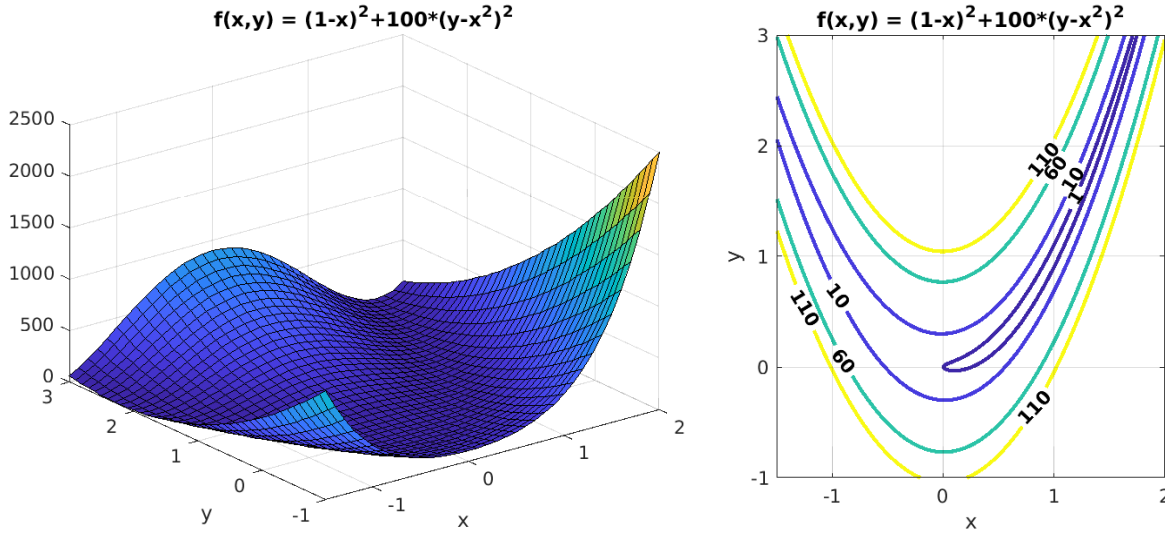


Figure 4.2: Plots of the Rosenbrock function $f(x, y) = (1 - x)^2 + 100 (y - x^2)^2$.

Note: The Rosenbrock function is commonly used when evaluating the performance of an optimization algorithm; because

- its minimizer $\mathbf{x} = \text{np.array}([1., 1.])$ is found in curved valley, and so minimizing the function is non-trivial, and
- the Rosenbrock function is included in the `scipy.optimize` package (as `rosen`), as well as its gradient (`rosen_der`) and its Hessian (`rosen_hess`).

¹The Rosenbrock function in 3D is given as $f(x, y, z) = [(1 - x)^2 + 100 (y - x^2)^2] + [(1 - y)^2 + 100 (z - y^2)^2]$, which has exactly one minimum at $(1, 1, 1)$. Similarly, one can define the Rosenbrock function in general N -dimensional spaces, for $N \geq 4$, by adding one more component for each enlarged dimension.

That is, $f(\mathbf{x}) = \sum_{i=1}^{N-1} [(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2]$, where $\mathbf{x} = [x_1, x_2, \dots, x_N] \in \mathbb{R}^N$. See Wikipedia (https://en.wikipedia.org/wiki/Rosenbrock_function) for details.

Remark 4.3. (Gradient). The gradient ∇f is a vector (a direction to move) that is

- pointing in the **direction of greatest increase** of the function, and
- **zero** ($\nabla f = 0$) at local maxima or local minima.

The goal of the gradient descent method is to address directly the process of minimizing the function f , using the fact that $-\nabla f(\mathbf{x})$ is the direction of **steepest descent** of f at $\mathbf{x}$. Given an initial point $\mathbf{x}_0$, we move it to the direction of $-\nabla f(\mathbf{x}_0)$ so as to get a smaller function value. That is,

$$\mathbf{x}_1 = \mathbf{x}_0 - \gamma \nabla f(\mathbf{x}_0) \Rightarrow f(\mathbf{x}_1) < f(\mathbf{x}_0).$$

We repeat this process till reaching at a desirable minimum. Thus the method is formulated as follows.

Algorithm 4.4. (Gradient descent method). Given an initial point $\mathbf{x}_0$, find iterates $\mathbf{x}_{n+1}$ recursively using

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma \nabla f(\mathbf{x}_n), \quad (4.3)$$

for some $\gamma > 0$. The parameter γ is called the **step length** or the **learning rate**. $\square$

To understand the basics of *gradient descent* (GD) method thoroughly, we start with the algorithm for solving

- unconstrained minimization problems
- defined in the one-dimensional (1D) space.

4.1.1. The gradient descent method in 1D

Problem 4.5. Consider the minimization problem in 1D:

$$\min_x f(x), \quad x \in S, \quad (4.4)$$

where S is a closed interval in $\mathbb{R}$. Then its gradient descent method reads

$$x_{n+1} = x_n - \gamma f'(x_n). \quad (4.5)$$

Picking the step length γ : Assume that the step length was chosen to be independent of n , although one can play with other choices as well. The question is how to select γ in order to make the best gain of the method. To turn the right-hand side of (4.5) into a more manageable form, we invoke Taylor's Theorem:²

$$f(x+t) = f(x) + t f'(x) + \int_x^{x+t} (x+t-s) f''(s) ds. \quad (4.6)$$

Assuming that $|f''(s)| \leq L$, we have

$$f(x+t) \leq f(x) + t f'(x) + \frac{t^2}{2} L.$$

Now, letting $x = x_n$ and $t = -\gamma f'(x_n)$ reads

$$\begin{aligned} f(x_{n+1}) &= f(x_n - \gamma f'(x_n)) \\ &\leq f(x_n) - \gamma f'(x_n) f'(x_n) + \frac{1}{2} L [\gamma f'(x_n)]^2 \\ &= f(x_n) - [f'(x_n)]^2 \left(\gamma - \frac{L}{2} \gamma^2 \right). \end{aligned} \quad (4.7)$$

The gain (learning) from the method occurs when

$$\gamma - \frac{L}{2} \gamma^2 > 0 \quad \Rightarrow \quad 0 < \gamma < \frac{2}{L}, \quad (4.8)$$

and it will be best when $\gamma - \frac{L}{2} \gamma^2$ is maximal. This happens at the point

$$\gamma = \frac{1}{L}. \quad (4.9)$$

² **Taylor's Theorem with integral remainder:** Suppose $f \in C^{n+1}[a, b]$ and $x_0 \in [a, b]$. Then, for every $x \in [a, b]$, $f(x) = \sum_{k=0}^n \frac{f^{(k)}(x_0)}{k!} (x - x_0)^k + R_n(x)$, $R_n(x) = \frac{1}{n!} \int_{x_0}^x (x-s)^n f^{(n+1)}(s) ds$.

Thus an effective **gradient descent method** (4.5) can be written as

$$x_{n+1} = x_n - \gamma f'(x_n) = x_n - \frac{1}{L} f'(x_n) = x_n - \frac{1}{\max |f''(x)|} f'(x_n). \quad (4.10)$$

Furthermore, it follows from (4.7) and (4.9) that

$$f(x_{n+1}) \leq f(x_n) - \frac{1}{2L} [f'(x_n)]^2. \quad (4.11)$$

Remark 4.6. (Convergence of gradient descent method).

Thus it is obvious that the method defines a sequence of points $\{x_n\}$ along which $\{f(x_n)\}$ decreases.

- If f is bounded from below and the level sets of f are bounded, $\{f(x_n)\}$ converges; so does $\{x_n\}$. That is, there is a point $\hat{x}$ such that

$$\lim_{n \rightarrow \infty} x_n = \hat{x}. \quad (4.12)$$

- Now, we can rewrite (4.11) as

$$[f'(x_n)]^2 \leq 2L [f(x_n) - f(x_{n+1})]. \quad (4.13)$$

Since $f(x_n) - f(x_{n+1}) \rightarrow 0$, also $f'(x_n) \rightarrow 0$.

- When f' is continuous, using (4.12) reads

$$f'(\hat{x}) = \lim_{n \rightarrow \infty} f'(x_n) = 0, \quad (4.14)$$

which implies that the limit $\hat{x}$ is a **critical point**.

- The method thus generally finds a critical point but that could still be a local minimum or a saddle point. Which it is cannot be decided at this level of analysis. $\square$

4.1.2. The full gradient descent algorithm

We can implement the *full* gradient descent algorithm as follows. The algorithm has only one free parameter: γ .

Algorithm 4.7. (The Gradient Descent Algorithm).

```

input: initial guess  $\mathbf{x}_0$ , step size  $\gamma > 0$ ;
for  $n = 0, 1, 2, \dots$  do
     $\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma \nabla f(\mathbf{x}_n)$ ;
end for
return  $\mathbf{x}_{n+1}$ ;
  
```

(4.15)

Remark 4.8. In theory, the step length γ can be found as in (4.9):

$$\gamma = \frac{1}{L}, \quad \text{where } L = \max_{\mathbf{x}} \|\nabla^2 f(\mathbf{x})\|. \quad (4.16)$$

Here $\|\cdot\|$ denotes an **induced matrix norm** and $\nabla^2 f(\mathbf{x})$ is the **Hessian** of f defined by

$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \frac{\partial^2 f}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{bmatrix} \in \mathbb{R}^{d \times d}. \quad (4.17)$$

- However, in practice, the computation of the Hessian (and L) can be expensive.

Remark 4.9. Gradient Descent vs. Newton's Method

The **gradient descent method** can be viewed as a simplification of the **Newton's method** (Section 4.2 below), replacing the inverse of Hessian, $(\nabla^2 f)^{-1}$, with a constant γ .

Convergence of Gradient Descent: Constant γ

Here we examine convergence of gradient descent on three examples: a *well-conditioned quadratic*, an *poorly-conditioned quadratic*, and a *non-convex function*, as shown by **Dr. Fabian Pedregosa**, UC Berkeley.

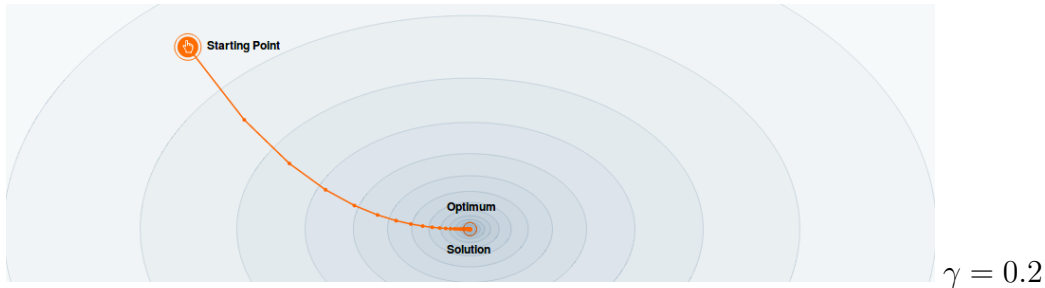


Figure 4.3: On a well-conditioned quadratic function, the gradient descent converges in a few iterations to the optimum

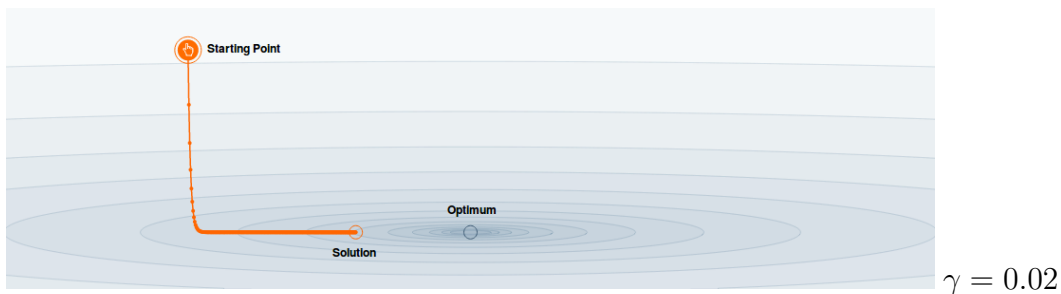


Figure 4.4: On a poorly-conditioned quadratic function, the gradient descent converges and takes many more iterations to converge than on the above well-conditioned problem. This is **partially** because gradient descent requires a ***much smaller step size*** on this problem to converge.

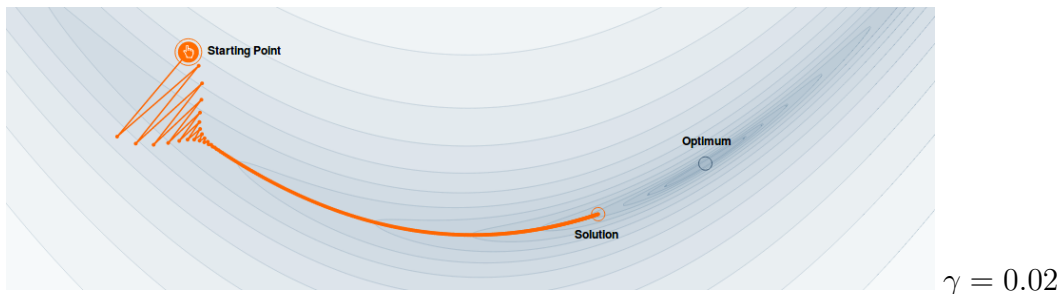


Figure 4.5: Gradient descent also converges on a poorly-conditioned non-convex problem. Convergence is slow in this case.

The Choice of Step Size: Backtracking Line Search

Note: The convergence of the gradient descent method can be extremely **sensitive to the choice of step size**. It often requires to choose the step size **adaptively**: the step size would better be chosen small in regions of large variability of the gradient, while in regions with small variability we would like to take it large.

Strategy 4.10. Backtracking line search procedures allow to select a step size depending on the current iterate and the gradient. In this procedure, we select an initial (optimistic) step size γ_n and evaluate the following inequality (known as **sufficient decrease condition**):

$$f(\mathbf{x}_n - \gamma_n \nabla f(\mathbf{x}_n)) \leq f(\mathbf{x}_n) - \frac{\gamma_n}{2} \|\nabla f(\mathbf{x}_n)\|^2. \quad (4.18)$$

If this inequality is verified, the current step size is kept. If not, the step size is divided by 2 (or any number larger than 1) repeatedly until (4.18) is verified. To get a better understanding, refer to (4.11) on p. 60, with (4.9).

The gradient descent algorithm with backtracking line search then becomes

Algorithm 4.11. (The Gradient Descent Algorithm, with Backtracking Line Search).

```

input: initial guess  $\mathbf{x}_0$ , step size  $\gamma_0 > 0$ ;
for  $n = 0, 1, 2, \dots$  do
    initial step size estimate  $\gamma_n$ ;
    while (TRUE) do
        if  $f(\mathbf{x}_n - \gamma_n \nabla f(\mathbf{x}_n)) \leq f(\mathbf{x}_n) - \frac{\gamma_n}{2} \|\nabla f(\mathbf{x}_n)\|^2$ 
            break;
        else  $\gamma_n = \gamma_n/2$ ;
    end while
     $\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla f(\mathbf{x}_n)$ ;
end for
return  $\mathbf{x}_{n+1}$ ;

```

(4.19)

Convergence of Gradient Descent: Backtracking line search

The following examples show the convergence of gradient descent with the aforementioned backtracking line search strategy for the step size.

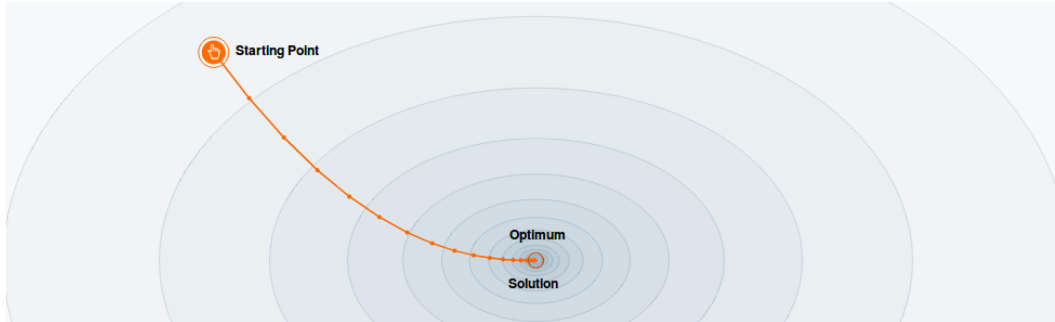


Figure 4.6: On a well-conditioned quadratic function, the gradient descent converges in a few iterations to the optimum. Adding the backtracking line search strategy for the step size does not change much in this case.

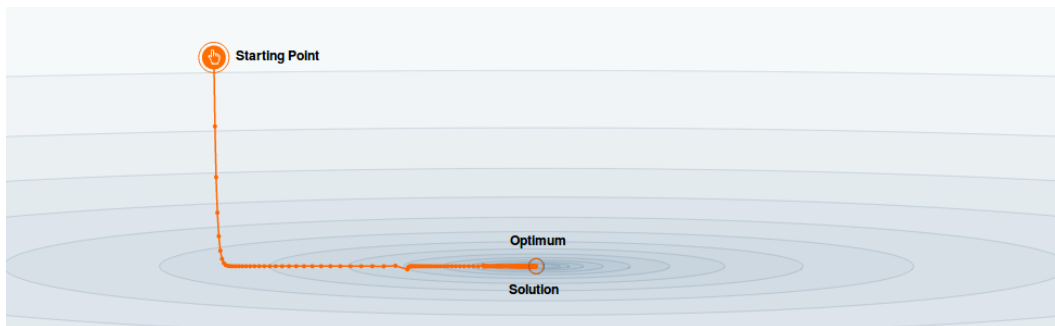


Figure 4.7: In this example we can clearly see the effect of the backtracking line search strategy: once the algorithm is in a region of low curvature, it can take larger step sizes. The final result is a much improved convergence compared with the fixed step-size equivalent.

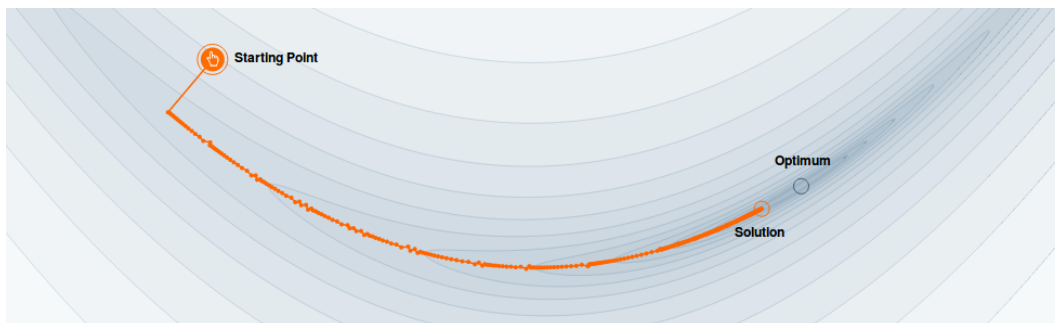


Figure 4.8: The backtracking line search also improves convergence on non-convex problems.

See Exercise 1 on p. 86.

4.1.3. Surrogate minimization: A unifying principle

Now, we aim to solve an optimization problem as in (4.1):

$$\min_{\mathbf{x} \in \Omega} f(\mathbf{x}). \quad (4.20)$$

Key Idea 4.12. Start at an initial estimate $\mathbf{x}_0$ and successively minimize an **approximating function** $\mathcal{Q}_n(\mathbf{x})$ [43]:

$$\mathbf{x}_{n+1} = \arg \min_{\mathbf{x} \in \Omega} \mathcal{Q}_n(\mathbf{x}). \quad (4.21)$$

We will call $\mathcal{Q}_n$ a **surrogate function**. It is also known as a **merit function**. A good surrogate function should be:

- Easy to optimize.
- Flexible enough to approximate a wide range of functions.

Gradient descent method: Approximates the objective function near $\mathbf{x}_n$ with a quadratic surrogate of the form

$$\mathcal{Q}_n(\mathbf{x}) = \mathbf{c}_n + \mathbf{G}_n \cdot (\mathbf{x} - \mathbf{x}_n) + \frac{1}{2\gamma} (\mathbf{x} - \mathbf{x}_n)^T (\mathbf{x} - \mathbf{x}_n), \quad (4.22)$$

which coincides with f in its value and first derivative, i.e.,

$$\begin{aligned} \mathcal{Q}_n(\mathbf{x}_n) = f(\mathbf{x}_n) &\Rightarrow \mathbf{c}_n = f(\mathbf{x}_n), \\ \nabla \mathcal{Q}_n(\mathbf{x}_n) = \nabla f(\mathbf{x}_n) &\Rightarrow \mathbf{G}_n = \nabla f(\mathbf{x}_n). \end{aligned} \quad (4.23)$$

The gradient descent method thus updates its iterates minimizing the following surrogate function:

$$\mathcal{Q}_n(\mathbf{x}) = f(\mathbf{x}_n) + \nabla f(\mathbf{x}_n) \cdot (\mathbf{x} - \mathbf{x}_n) + \frac{1}{2\gamma} \|\mathbf{x} - \mathbf{x}_n\|^2. \quad (4.24)$$

Differentiating the function and equating to zero reads

$$\mathbf{x}_{n+1} = \arg \min_{\mathbf{x}} \mathcal{Q}_n(\mathbf{x}) = \mathbf{x}_n - \gamma \nabla f(\mathbf{x}_n). \quad (4.25)$$

Multiple Local Minima Problem

Remark 4.13. Optimizing Optimization Algorithms

Although you can choose the step size **smartly**, there is no guarantee for your algorithm to converge to the desired solution (the global minimum), particularly when the objective is not convex.

Here, we consider the so-called **Gaussian homotopy continuation** method [53], which may overcome the **local minima problem** for certain classes of optimization problems.

- The method begins by trying to find a convex approximation of an optimization problem, using a technique called **Gaussian smoothing**.
- Gaussian smoothing converts the cost function into a related function, each of whose values is a **weighted average** of all the surrounding values.
- This has the effect of smoothing out any abrupt dips or ascents in the cost function's graph, as shown in Figure 4.9.
- The weights assigned the surrounding values are determined by a Gaussian function, or normal distribution.

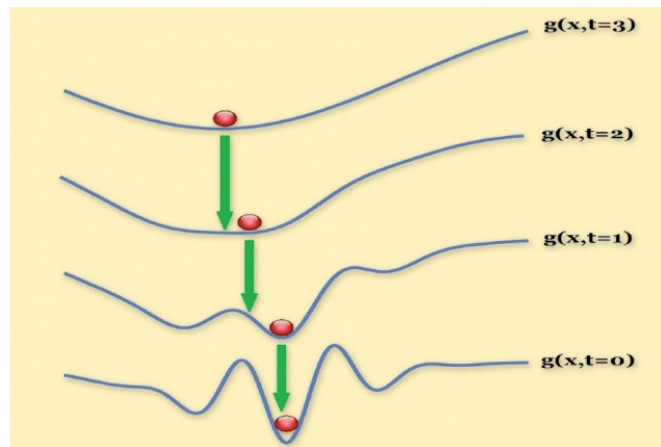


Figure 4.9: Smooth sailing, through a Gaussian smoothing.

However, there will be many ways to incorporate Gaussian smoothing; a realization of the method will be challenging, particularly for ML optimization. See P.2 (p. 375).

4.2. Newton's Method

4.2.1. Derivation

Scheme 4.14. The **Newton's method** is an iterative method to solve the unconstrained optimization problem in (4.1), p. 56, when f is twice differentiable. In Newton's method, we approximate the objective with a **quadratic surrogate** of the form

$$\mathcal{Q}_n(\mathbf{x}) = \mathbf{c}_n + \mathbf{G}_n \cdot (\mathbf{x} - \mathbf{x}_n) + \frac{1}{2\gamma} (\mathbf{x} - \mathbf{x}_n)^T \mathbf{H}_n (\mathbf{x} - \mathbf{x}_n). \quad (4.26)$$

Compared with gradient descent, the quadratic term is not fixed to be the identity but instead incorporates an **invertible matrix** $\mathbf{H}_n$.

- A reasonable condition to impose on this surrogate function is that at $\mathbf{x}_n$ it coincides with f *at least* in **its value** and **first derivatives**, as in (4.23).
- **An extra condition** the method imposes is that

$$\mathbf{H}_n = \nabla^2 f(\mathbf{x}_n), \quad (4.27)$$

where $\nabla^2 f$ is the **Hessian** of f defined as in (4.17).

- Thus the Newton's method updates its iterates **minimizing** the following surrogate function:

$$\mathcal{Q}_n(\mathbf{x}) = f(\mathbf{x}_n) + \nabla f(\mathbf{x}_n) \cdot (\mathbf{x} - \mathbf{x}_n) + \frac{1}{2\gamma} (\mathbf{x} - \mathbf{x}_n)^T \nabla^2 f(\mathbf{x}_n) (\mathbf{x} - \mathbf{x}_n). \quad (4.28)$$

- We can find the **optimum of the function** differentiating and equating to zero. This way we find (assuming the Hessian is invertible)

$$\mathbf{x}_{n+1} = \arg \min_{\mathbf{x}} \mathcal{Q}_n(\mathbf{x}) = \mathbf{x}_n - \gamma [\nabla^2 f(\mathbf{x}_n)]^{-1} \nabla f(\mathbf{x}_n). \quad (4.29)$$

Note: When $\gamma = 1$, $\mathcal{Q}_n(\mathbf{x})$ in (4.28) is the second-order approximation of the objective function near $\mathbf{x}_n$.

Remark 4.15. Where applicable, Newton's method **converges much faster** towards a local maximum or minimum than the gradient descent.

- In fact, every local minimum has a neighborhood such that, if we start within this neighborhood, Newton's method with step size $\gamma = 1$ **converges quadratically** assuming the Hessian is invertible and Lipschitz continuous.

Remark 4.16. The Newton's method can be seen as to find the **critical points** of f , i.e., $\hat{\mathbf{x}}$ such that $\nabla f(\hat{\mathbf{x}}) = 0$. Let

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta \mathbf{x}. \quad (4.30)$$

Then

$$\nabla f(\mathbf{x}_{n+1}) = \nabla f(\mathbf{x}_n + \Delta \mathbf{x}) = \nabla f(\mathbf{x}_n) + \nabla^2 f(\mathbf{x}_n) \Delta \mathbf{x} + \mathcal{O}(|\Delta \mathbf{x}|^2).$$

Truncating high-order terms of $\Delta \mathbf{x}$ and equating the result to zero reads

$$\Delta \mathbf{x} = -(\nabla^2 f(\mathbf{x}_n))^{-1} \nabla f(\mathbf{x}_n). \quad (4.31)$$

Implementation of Newton's Method

Only the difference from the gradient descent algorithm is to compute the Hessian matrix $\nabla^2 f(\mathbf{x}_n)$ to be applied to the gradient.

Algorithm 4.17. (Newton's method).

```

input: initial guess  $\mathbf{x}_0$ , step size  $\gamma > 0$ ;
for  $n = 0, 1, 2, \dots$  do
     $\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma [\nabla^2 f(\mathbf{x}_n)]^{-1} \nabla f(\mathbf{x}_n);$ 
end for
return  $\mathbf{x}_{n+1};$ 
    
```

(4.32)

For the three example functions in Section 4.1.2, the Newton's method performs better as shown in the following.

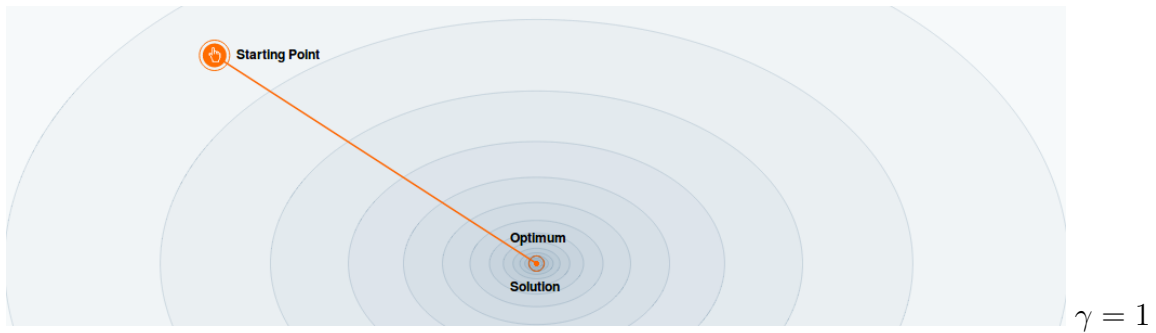


Figure 4.10: In this case the approximation is exact and it converges in a single iteration.

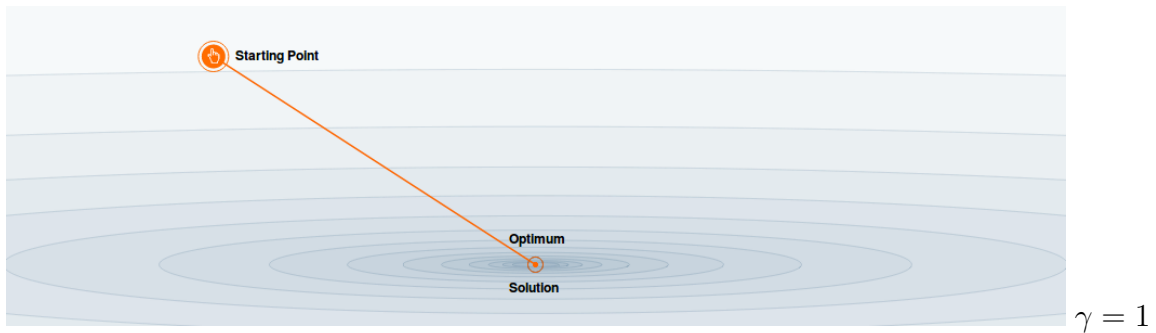


Figure 4.11: Although badly-conditioned, the cost function is quadratic; it converges in a single iteration.

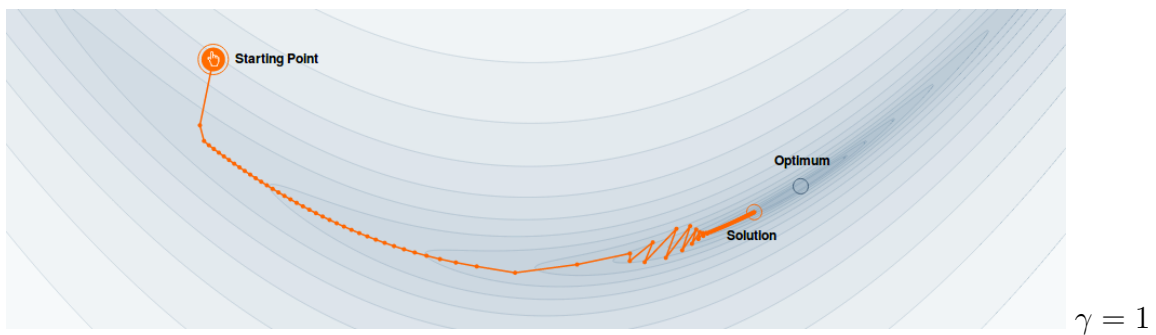


Figure 4.12: When the Hessian is close to singular, there might be some numerical instabilities. However, it is better than the result of the gradient descent method in Figure 4.5.

4.2.2. Hessian and principal curvatures

Claim 4.18. The **Hessian** (or **Hessian matrix**) describes the **local curvature** of a function. The eigenvalues and eigenvectors of the Hessian have geometric meaning:

- The first principal eigenvector (corresponding to the largest eigenvalue in modulus) is the direction of **greatest curvature**.
- The last principal eigenvector (corresponding to the smallest eigenvalue in modulus) is the direction of **least curvature**.
- The corresponding eigenvalues are the respective amounts of these curvatures.

The eigenvectors of the Hessian are called **principal directions**, which are always orthogonal to each other. The eigenvalues of the Hessian are called **principal curvatures** and are invariant under rotation and always real-valued.

Observation 4.19. Let a Hessian matrix $H \in \mathbb{R}^{d \times d}$ be positive definite and its eigenvalue-eigenvector pairs be given as $\{(\lambda_j, \mathbf{u}_j)\}$, $j = 1, 2, \dots, d$.

- Then, given a vector $\mathbf{v} \in \mathbb{R}^d$, it can be expressed as

$$\mathbf{v} = \sum_{j=1}^d \xi_j \mathbf{u}_j,$$

and therefore

$$H^{-1} \mathbf{v} = \sum_{j=1}^d \xi_j \frac{1}{\lambda_j} \mathbf{u}_j, \quad (4.33)$$

where components of $\mathbf{v}$ in leading principal directions of H have been diminished with larger factors.

- Thus the angle measured from $H^{-1} \mathbf{v}$ to **the least principal direction of H** becomes smaller than the angle measured from $\mathbf{v}$.
- It is also true when $\mathbf{v}$ is the gradient vector (in fact, the negation of the gradient vector).

Note: The above observation can be rephrased mathematically as follows. Let $\mathbf{u}_d$ be the least principal direction of H . Then

$$\text{angle}(\mathbf{u}_d, H^{-1}\mathbf{v}) < \text{angle}(\mathbf{u}_d, \mathbf{v}), \quad \forall \mathbf{v}, \quad (4.34)$$

where

$$\text{angle}(\mathbf{a}, \mathbf{b}) = \arccos \left(\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} \right).$$

This implies that by setting $\mathbf{v} = -\nabla f(\mathbf{x}_n)$, the adjusted vector $H^{-1}\mathbf{v}$ is a rotation (and scaling) of the steepest descent vector towards the least curvature direction.

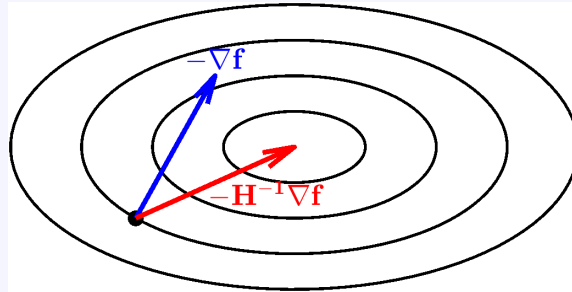


Figure 4.13: The effect of the Hessian inverse H^{-1} .

Claim 4.20. The net effect of H^{-1} is to **rotate and scale** the gradient vector to face towards the minimizer by a certain degree, which may make the Newton's method converge much faster than the gradient descent method.

Example 4.21. One can easily check that at each point (x, y) on the ellipsoid

$$z = f(x, y) = \frac{(x - h)^2}{a^2} + \frac{(y - k)^2}{b^2}, \quad (4.35)$$

the vector $-\left[\nabla^2 f(x, y)\right]^{-1} \nabla f(x, y)$ is always facing towards the minimizer (h, k) . See Exercise 2. $\square$

4.3. Quasi-Newton Methods

Note: The central issue with Newton's method is that we need to be able to compute the **inverse Hessian matrix**.

- For ML applications, the dimensionality of the problem can be of the **order of thousands or millions**; computing the Hessian or its inverse is often impractical.
- Because of these reasons, Newton's method is **rarely used in practice** to optimize functions corresponding to **large problems**.
- Luckily, Newton's method can still work even if the Hessian is replaced by a **good approximation**.

The BFGS Algorithm (1970)

Note: One of the most popular quasi-Newton methods is the BFGS algorithm, which is named after Charles George **Broyden** [9], Roger **Fletcher** [21], Donald **Goldfarb** [24], and David **Shanno** [72].

Key Idea 4.22. As a byproduct of the optimization, we observe many gradients. Can we use these **gradients** to iteratively construct an approximation of the Hessian?

Derivation of BFGS algorithm

- At each iteration of the method, we consider the surrogate function:

$$\mathcal{Q}_n(\mathbf{x}) = \mathbf{c}_n + \mathbf{G}_n \cdot (\mathbf{x} - \mathbf{x}_n) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_n)^T \mathbf{H}_n (\mathbf{x} - \mathbf{x}_n), \quad (4.36)$$

where in this case $\mathbf{H}_n$ is **an approximation to the Hessian matrix**, which is updated iteratively at each stage.

- **A reasonable thing to ask** to this surrogate is that its gradient coincides with ∇f at the last two iterates $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$:

$$\begin{aligned} \nabla \mathcal{Q}_{n+1}(\mathbf{x}_{n+1}) &= \nabla f(\mathbf{x}_{n+1}), \\ \nabla \mathcal{Q}_{n+1}(\mathbf{x}_n) &= \nabla f(\mathbf{x}_n). \end{aligned} \quad (4.37)$$

- From the definition of $\mathcal{Q}_{n+1}$:

$$\mathcal{Q}_{n+1}(\mathbf{x}) = \mathbf{c}_{n+1} + \mathbf{G}_{n+1} \cdot (\mathbf{x} - \mathbf{x}_{n+1}) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_{n+1})^T \mathbf{H}_{n+1} (\mathbf{x} - \mathbf{x}_{n+1}),$$

we have

$$\nabla \mathcal{Q}_{n+1}(\mathbf{x}_{n+1}) - \nabla \mathcal{Q}_{n+1}(\mathbf{x}_n) = \mathbf{G}_{n+1} - \nabla \mathcal{Q}_{n+1}(\mathbf{x}_n) = -\mathbf{H}_{n+1}(\mathbf{x}_n - \mathbf{x}_{n+1}).$$

Thus we reach at the following condition on $\mathbf{H}_{n+1}$:

$$\mathbf{H}_{n+1}(\mathbf{x}_{n+1} - \mathbf{x}_n) = \nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n), \quad (4.38)$$

which is the **secant equation**.

- Let

$$\mathbf{s}_n = \mathbf{x}_{n+1} - \mathbf{x}_n \text{ and } \mathbf{y}_n = \nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n).$$

Then $\mathbf{H}_{n+1}\mathbf{s}_n = \mathbf{y}_n$, which requires to satisfy the **curvature condition**

$$\mathbf{y}_n \cdot \mathbf{s}_n > 0, \quad (4.39)$$

with which $\mathbf{H}_{n+1}$ becomes positive definite. (Pre-multiply $\mathbf{s}_n^T$ to the secant equation to prove it.)

- In order to **maintain the symmetry and positive definiteness** of $\mathbf{H}_{n+1}$, the update formula can be chosen as³

$$\mathbf{H}_{n+1} = \mathbf{H}_n + \alpha \mathbf{u}\mathbf{u}^T + \beta \mathbf{v}\mathbf{v}^T. \quad (4.40)$$

- Imposing the secant condition $\mathbf{H}_{n+1}\mathbf{s}_n = \mathbf{y}_n$ and with (4.40), we get the update equation of $\mathbf{H}_{n+1}$:

$$\mathbf{H}_{n+1} = \mathbf{H}_n + \frac{\mathbf{y}_n \mathbf{y}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n} - \frac{(\mathbf{H}_n \mathbf{s}_n)(\mathbf{H}_n \mathbf{s}_n)^T}{\mathbf{s}_n \cdot \mathbf{H}_n \mathbf{s}_n}. \quad (4.41)$$

- Let $\mathbf{B}_n = \mathbf{H}_n^{-1}$, the inverse of $\mathbf{H}_n$. Then, applying the **Sherman-Morrison formula**, we can update $\mathbf{B}_{n+1} = \mathbf{H}_{n+1}^{-1}$ as follows.

$$\mathbf{B}_{n+1} = \left(\mathbf{I} - \frac{\mathbf{s}_n \mathbf{y}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n} \right) \mathbf{B}_n \left(\mathbf{I} - \frac{\mathbf{y}_n \mathbf{s}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n} \right) + \frac{\mathbf{s}_n \mathbf{s}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n}. \quad (4.42)$$

See Exercise 4.4.

³**Rank-one matrices:** Let A be an $m \times n$ matrix. Then $\text{rank}(A) = 1$ if and only if there exist column vectors $\mathbf{v} \in \mathbb{R}^m$ and $\mathbf{w} \in \mathbb{R}^n$ such that $A = \mathbf{v}\mathbf{w}^T$.

Now, we are ready to summarize the BFGS algorithm.

Algorithm 4.23. (The BFGS algorithm). **The n -th step:**

1. Obtain the search direction: $\mathbf{p}_n = B_n(-\nabla f(\mathbf{x}_n))$.
2. Perform line-search to find an acceptable stepsize γ_n .
3. Set $\mathbf{s}_n = \gamma_n \mathbf{p}_n$ and update $\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{s}_n$.
4. Get $\mathbf{y}_n = \nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n)$.
5. Update $B = H^{-1}$:

$$B_{n+1} = \left(I - \frac{\mathbf{s}_n \mathbf{y}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n} \right) B_n \left(I - \frac{\mathbf{y}_n \mathbf{s}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n} \right) + \frac{\mathbf{s}_n \mathbf{s}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n}.$$

Remark 4.24. The BFGS Algorithm

- The algorithm begins with B_0 , an estimation of H_0^{-1} . It is often better when $B_0 = H_0^{-1}$.
- The resulting algorithm is a method which combines the *low-cost of gradient descent* with the *favorable convergence properties of Newton's method*.

Examples, with the BFGS algorithm

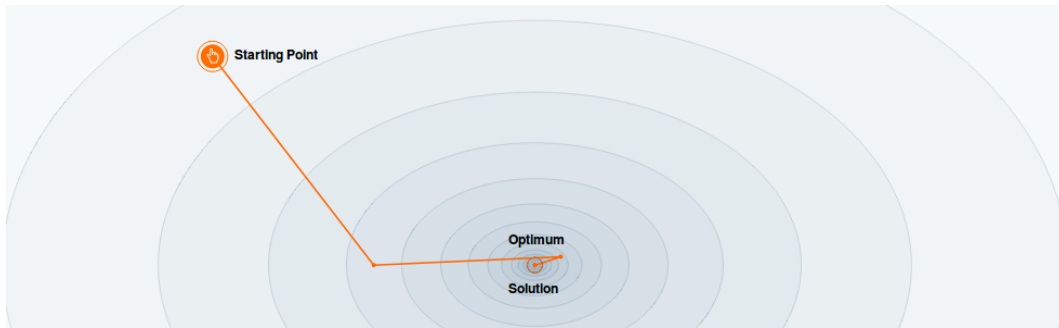


Figure 4.14: BFGS, on the **well-conditioned quadratic** objective function.

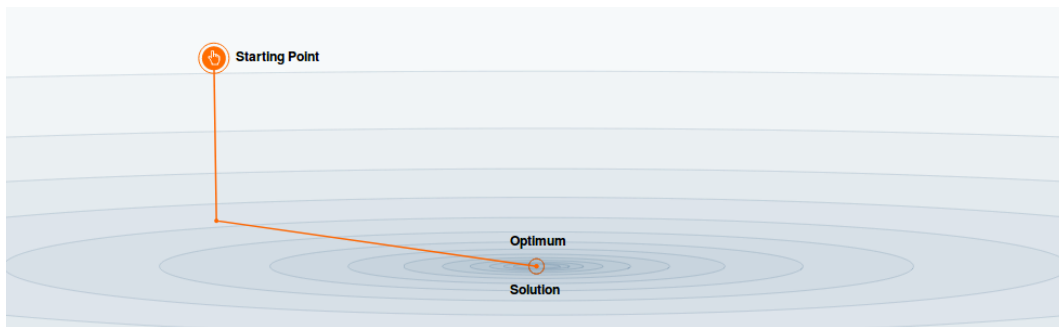


Figure 4.15: On the **poorly-conditioned quadratic** problem, the BFGS algorithm quickly builds a good estimator of the Hessian and is able to converge very fast towards the optimum. Note that this, just like the Newton method (and unlike gradient descent), BFGS does not seem to be affected (much) by a bad conditioning of the problem.

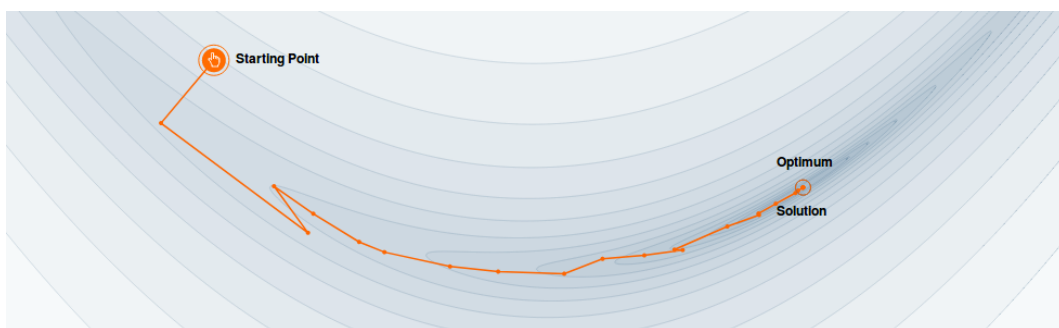


Figure 4.16: Even on the **ill-conditioned nonconvex** problem, the BFGS algorithm also converges extremely fast, with a convergence that is more similar to Newton's method than to gradient descent.

4.4. The Stochastic Gradient Method

The **stochastic gradient method (SGM)**, introduced by Robbins-Monro in 1951 [63], is

- one of the most widely-used methods for large-scale optimization, and
- one of the main methods behind the current AI revolution.

Note: The SGM was considered earlier in Section 3.3.2, as a variant of the gradient descent method for Adaline classification. Here we will discuss it in details for more general optimization problems.

- The stochastic gradient method (a.k.a. **stochastic gradient descent** or **SGD**) can be used to solve optimization problems in which **the objective function is of the form**

$$f(x) = \mathbb{E}[f_i(x)],$$

where the expectation is taken with respect to i .

- **The most common case** is when i can take a finite number of values, in which the problem becomes

$$\min_{\mathbf{x} \in \mathbb{R}^p} f(\mathbf{x}), \quad f(\mathbf{x}) = \frac{1}{m} \sum_{i=1}^m f_i(\mathbf{x}). \quad (4.43)$$

- The SGM can be motivated as an approximation to gradient descent in which at each iteration we approximate the gradient as

$$\nabla f(\mathbf{x}_n) \approx \nabla f_i(\mathbf{x}_n). \quad (4.44)$$

We can write the full stochastic gradient algorithm as follows. The algorithm has only one free parameter: γ .

Algorithm 4.25. (Stochastic Gradient Descent).

```

input: initial guess  $\mathbf{x}_0$ , step size sequence  $\gamma_n > 0$ ;
for  $n = 0, 1, 2, \dots$  do
    Choose  $i \in \{1, 2, \dots, m\}$  uniformly at random;
     $\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla f_i(\mathbf{x}_n);$ 
end for
return  $\mathbf{x}_{n+1};$ 

```

(4.45)

The SGD can be much more efficient than gradient descent in the case in which **the objective consists of a large sum**, because at each iteration we only need to evaluate a partial gradient and not the full gradient.

Example 4.26. A least-squares problem can be written in the form acceptable by SGD since

$$\frac{1}{m} \|A\mathbf{x} - \mathbf{b}\|^2 = \frac{1}{m} \sum_{i=1}^m (A_i\mathbf{x} - \mathbf{b}_i)^2, \quad (4.46)$$

where A_i is the i -th row of A .

Step Size for the SGD

- The choice of step size is one of the most delicate aspects of the SGD. For the SGD, **the backtracking line search is not an option** since it would involve to evaluate the objective function at each iteration, which destroys the computational advantage of this method.
- **Two popular step size strategies exist for the SGD**: constant step size and decreasing step size.

(a) **Constant step size**: In the constant step size strategy,

$$\gamma_n = \gamma$$

for some pre-determined constant γ .

The method converges very fast to neighborhood of a local minimum and then **bounces around**. The radius of this neighborhood will depend on the step size γ [44, 51].

(b) **Decreasing step size**: One can guarantee convergence to a local minimizer choosing a step size sequence that satisfies

$$\sum_{n=1}^{\infty} \gamma_n = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \gamma_n^2 < \infty. \quad (4.47)$$

The most popular sequence to verify this is

$$\gamma_n = \frac{C}{n}, \quad (4.48)$$

for some constant C . This is often referred to as a **decreasing step-size sequence**, although in fact the sequence does not need to be monotonically decreasing.

Examples, with SGD



Figure 4.17: For the well-conditioned convex problem, stochastic gradient with constant step size converges quickly to a neighborhood of the optimum, but then bounces around.

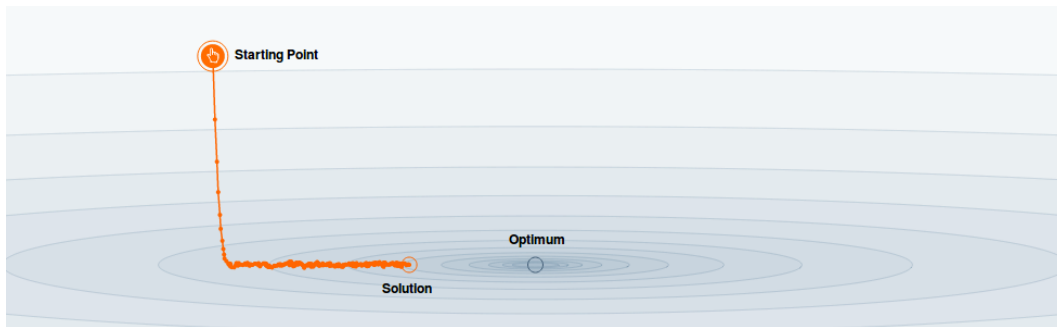


Figure 4.18: Stochastic Gradient **with decreasing step sizes** is quite robust to the choice of step size. On one hand there is really no good way to set the step size (e.g., no equivalent of line search for Gradient Descent) but on the other hand it converges for a wide range of step sizes.

Convergence of the SGD

Question. Why does the SGD converge, despite its update being a very rough estimate of the gradient?

To answer this question, we must first understand the **unbiasedness property** of its update.

Proposition 4.27. (Unbiasedness of the SGD update). Let $\mathbb{E}_n$ denote the expectation with respect to the choice of random sample (i) at iteration n . Then since the index i is chosen *uniformly* at random, we have

$$\begin{aligned}\mathbb{E}_n[\nabla f_{i_n}(\mathbf{x}_n)] &= \sum_{i=1}^m \nabla f_i(\mathbf{x}_n) P(i_n = i) \\ &= \frac{1}{m} \sum_{i=1}^m \nabla f_i(\mathbf{x}_n) = \nabla f(\mathbf{x}_n)\end{aligned}\tag{4.49}$$

This is the crucial property that makes SGD work. For a full proof, see e.g. [7].

4.5. The Levenberg–Marquardt Algorithm, for Nonlinear Least-Squares Problems

The **Levenberg–Marquardt algorithm** (LMA), a.k.a. the **damped least-squares (DLS)** method, is used for the solution of **nonlinear least-squares problems** which arise especially in curve fitting.

- In fitting a function $\hat{y}(x; \mathbf{p})$ of an independent variable x and a parameter vector $\mathbf{p} \in \mathbb{R}^n$ to a set of m data points (x_i, y_i) , it is customary and convenient to minimize the **sum of the weighted squares of the errors** (or **weighted residuals**) between the measured data y_i and the curve-fit function $\hat{y}(x_i; \mathbf{p})$.

$$\begin{aligned} f(\mathbf{p}) &= \sum_{i=1}^m \left[\frac{y_i - \hat{y}(x_i; \mathbf{p})}{\eta_i} \right]^2 \\ &= (\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p}))^T \mathbf{W} (\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p})) \end{aligned} \quad (4.50)$$

where η_i is the measurement error for y_i and the weighting matrix $\mathbf{W}$ is defined as

$$\mathbf{W} = \text{diag}\{1/\eta_i^2\} \in \mathbb{R}^{m \times m}.$$

- However, more formally, $\mathbf{W}$ can be set to the inverse of the measurement error covariance matrix; more generally, the weights can be set to pursue other curve-fitting goals.

Definition 4.28. The **measurement error** (also called the **observational error**) is the difference between a measured quantity and its true value. It includes **random error** and **systematic error** (caused by a mis-calibrated instrument that affects all measurements).

Note: The **goodness-of-fit measure** in (4.50) is called the **chi-squared error criterion** because the sum of squares of normally-distributed variables is distributed as the χ -squared distribution.

If the function $\hat{y}(x; \mathbf{p})$ is **nonlinear** in the model parameters $\mathbf{p}$, then **the minimization** of the χ -squared function f with respect to the parameters must be carried out **iteratively**:

$$\mathbf{p} := \mathbf{p} + \Delta \mathbf{p}. \quad (4.51)$$

The goal of each iteration is to find the parameter update $\Delta \mathbf{p}$ that reduces f . We will begin with the gradient descent method and the Gauss-Newton method.

4.5.1. The gradient descent method

Recall: The gradient descent method is a general minimization method which updates parameter values in the “steepest downhill” direction: the direction opposite to the gradient of the objective function.

- The gradient descent method converges well for problems with simple objective functions.
- For problems with thousands of parameters, gradient descent methods are **sometimes the only workable choice**.

The gradient of the objective function with respect to the parameters is

$$\begin{aligned} \frac{\partial}{\partial \mathbf{p}} f &= 2(\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p}))^T \mathbf{W} \frac{\partial}{\partial \mathbf{p}} (\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p})) \\ &= -2(\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p}))^T \mathbf{W} \left[\frac{\partial \hat{\mathbf{y}}(\mathbf{p})}{\partial \mathbf{p}} \right] \\ &= -2(\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p}))^T \mathbf{W} \mathbf{J}, \end{aligned} \quad (4.52)$$

where $\mathbf{J} = \frac{\partial \hat{\mathbf{y}}(\mathbf{p})}{\partial \mathbf{p}} \in \mathbb{R}^{m \times n}$ is the **Jacobian matrix**. The parameter update $\Delta \mathbf{p}$ that moves the parameters in the direction of steepest descent is given by

$$\Delta \mathbf{p}_{\text{gd}} = \gamma \mathbf{J}^T \mathbf{W} (\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p})), \quad (4.53)$$

where $\gamma > 0$ is the step length.

4.5.2. The Gauss-Newton method

The **Gauss-Newton method** is a method for minimizing a **sum-of-squares objective function**.

- It assumes that the objective function is **approximately quadratic** near the minimizer [6], and utilizes an **approximate Hessian**.
- For moderately-sized problems, the Gauss-Newton method typically converges much faster than gradient-descent methods [52].

Algorithm Derivation

- The function evaluated with perturbed model parameters may be locally approximated through a **first-order Taylor series expansion**.

$$\hat{\mathbf{y}}(\mathbf{p} + \Delta\mathbf{p}) \approx \hat{\mathbf{y}}(\mathbf{p}) + \left[\frac{\partial \hat{\mathbf{y}}(\mathbf{p})}{\partial \mathbf{p}} \right] \Delta\mathbf{p} = \hat{\mathbf{y}}(\mathbf{p}) + \mathbf{J} \Delta\mathbf{p}. \quad (4.54)$$

- Substituting the approximation into (4.50), p. 81, we have

$$\begin{aligned} f(\mathbf{p} + \Delta\mathbf{p}) \approx & \mathbf{y}^T \mathbf{W} \mathbf{y} - 2\mathbf{y}^T \mathbf{W} \hat{\mathbf{y}}(\mathbf{p}) + \hat{\mathbf{y}}(\mathbf{p})^T \mathbf{W} \hat{\mathbf{y}}(\mathbf{p}) \\ & - 2(\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p}))^T \mathbf{W} \mathbf{J} \Delta\mathbf{p} + (\mathbf{J} \Delta\mathbf{p})^T \mathbf{W} \mathbf{J} \Delta\mathbf{p}. \end{aligned} \quad (4.55)$$

Note: The above approximation for $f(\mathbf{p} + \Delta\mathbf{p})$ is **quadratic** in the parameter perturbation $\Delta\mathbf{p}$.

- The parameter update $\Delta\mathbf{p}$ can be found from $\partial f / \partial \Delta\mathbf{p} = 0$:

$$\frac{\partial}{\partial \Delta\mathbf{p}} f(\mathbf{p} + \Delta\mathbf{p}) \approx -2(\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p}))^T \mathbf{W} \mathbf{J} + 2(\mathbf{J} \Delta\mathbf{p})^T \mathbf{W} \mathbf{J} = 0, \quad (4.56)$$

and therefore the resulting normal equation for the Gauss-Newton update reads

$$[\mathbf{J}^T \mathbf{W} \mathbf{J}] \Delta\mathbf{p}_{\text{gn}} = \mathbf{J}^T \mathbf{W} (\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p})). \quad (4.57)$$

Note: The matrix $\mathbf{J}^T \mathbf{W} \mathbf{J} \in \mathbb{R}^{n \times n}$ is an **approximate Hessian** of the objective function. Here, we require $m \geq n$; otherwise, the approximate Hessian must be singular.

4.5.3. The Levenberg-Marquardt algorithm

The **Levenberg-Marquardt algorithm** *adaptively* varies the parameter updates **between the gradient descent and the Gauss-Newton methods**:

$$[J^T W J + \lambda I] \Delta p_{\text{lm}} = J^T W (y - \hat{y}(p)), \quad (4.58)$$

where $\lambda \geq 0$ is the **damping parameter**. Small values of λ result in a Gauss-Newton update and large values of it result in a gradient descent update.

Remark 4.29. Implementation of the Levenberg-Marquardt Algorithm.

- The damping parameter λ is **often initialized to be large** so that first updates are small steps in the steepest-descent direction.
- **As the solution improves, λ is decreased**; the Levenberg-Marquardt method approaches the Gauss-Newton method, and the solution typically accelerates to the local minimum [49, 52].
- If any iteration happens to result in a **bad approximation**, e.g.,

$$f(p + \Delta p_{\text{lm}}) > f(p),$$

then λ is increased.

Acceptance of the Step

There have been many variations of the Levenberg-Marquardt method, particularly for **acceptance criteria**.

Example 4.30. Acceptance Criterion. Recall (4.50) and (4.54):

$$f(\mathbf{p}) = (\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p}))^T \mathbf{W} (\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p})), \quad (4.50)$$

$$\hat{\mathbf{y}}(\mathbf{p} + \Delta \mathbf{p}_{\text{lm}}) \approx \hat{\mathbf{y}}(\mathbf{p}) + \mathbf{J} \Delta \mathbf{p}_{\text{lm}}. \quad (4.54)$$

Then the **Sum of Squared Error (SSE)**, $f(\mathbf{p} + \Delta \mathbf{p}_{\text{lm}})$, can be approximated by

$$\begin{aligned} f(\mathbf{p} + \Delta \mathbf{p}_{\text{lm}}) &= (\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p} + \Delta \mathbf{p}_{\text{lm}}))^T \mathbf{W} (\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p} + \Delta \mathbf{p}_{\text{lm}})) \\ &\approx (\mathbf{y} - [\hat{\mathbf{y}}(\mathbf{p}) + \mathbf{J} \Delta \mathbf{p}_{\text{lm}}])^T \mathbf{W} (\mathbf{y} - [\hat{\mathbf{y}}(\mathbf{p}) + \mathbf{J} \Delta \mathbf{p}_{\text{lm}}]). \end{aligned} \quad (4.59)$$

- At the k -th step, we first compute

$$\begin{aligned} \rho_k(\Delta \mathbf{p}_{\text{lm}}) &= \frac{f(\mathbf{p}) - f(\mathbf{p} + \Delta \mathbf{p}_{\text{lm}})}{f(\mathbf{p}) - (\mathbf{y} - \hat{\mathbf{y}} - \mathbf{J} \Delta \mathbf{p}_{\text{lm}})^T \mathbf{W} (\mathbf{y} - \hat{\mathbf{y}} - \mathbf{J} \Delta \mathbf{p}_{\text{lm}})} \\ &= \frac{f(\mathbf{p}) - f(\mathbf{p} + \Delta \mathbf{p}_{\text{lm}})}{\Delta \mathbf{p}_{\text{lm}}^T (\lambda_k \Delta \mathbf{p}_{\text{lm}} + \mathbf{J}^T \mathbf{W} (\mathbf{y} - \hat{\mathbf{y}}(\mathbf{p})))}. \quad [\Leftarrow (4.58)] \end{aligned} \quad (4.60)$$

- Then the step is accepted when $\rho_k(\Delta \mathbf{p}_{\text{lm}}) > \varepsilon_0$, for a threshold $\varepsilon_0 > 0$.

An example implementation reads

$$\left[\begin{array}{l} \text{Initialize } \mathbf{p}_0, \lambda_0, \text{ and } \varepsilon_0; \text{ (e.g. } \lambda_0 = 0.01 \text{ \& } \varepsilon_0 = 0.1) \\ \text{Compute } \Delta \mathbf{p}_{\text{lm}} \text{ from (4.58);} \\ \text{Evaluate } \rho_k \text{ from (4.60);} \\ \text{If } \rho_k > \varepsilon_0: \\ \quad \mathbf{p}_{k+1} = \mathbf{p}_k + \Delta \mathbf{p}_{\text{lm}}; \lambda_{k+1} = \lambda_k \cdot \max[1/3, 1 - (2\rho_k)^3]; \nu_k = 2; \\ \text{otherwise: } \lambda_{k+1} = \lambda_k \nu_k; \nu_{k+1} = 2\nu_k; \end{array} \right. \quad (4.61)$$

Exercises for Chapter 4

- 4.1. (**Gradient descent method**). Implement the gradient descent algorithm (4.15) and the gradient descent algorithm with backtracking line search (4.19).
- (a) Compare their performances with the Rosenbrock function in 2D (4.2).
 - (b) Find an effective strategy for initial step size estimate for (4.19).
- 4.2. (**Net effect of the inverse Hessian matrix**). Verify the claim in Example 4.21.
- 4.3. (**Newton's method**). Implement a line search version of the Newton's method (4.32) with the Rosenbrock function in 2D.
- (a) Recall the results in Exercise 1. With the backtracking line search, is the Newton's method better than the gradient descent method?
 - (b) Now, we will approximate the Hessian matrix by its diagonal. That is,

$$\mathcal{D}_n = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & 0 \\ 0 & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}(\mathbf{x}_n) \approx \nabla^2 f(\mathbf{x}_n) \stackrel{\text{def}}{=} \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}(\mathbf{x}_n). \quad (4.62)$$

How does the Newton's method perform when the Hessian matrix is replaced by $\mathcal{D}_n$?

- 4.4. (**BFGS update**). Consider $\mathbf{H}_{n+1}$ and $\mathbf{B}_{n+1}$ in (4.41) and (4.42), respectively:

$$\begin{aligned} \mathbf{H}_{n+1} &= \mathbf{H}_n + \frac{\mathbf{y}_n \mathbf{y}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n} - \frac{(\mathbf{H}_n \mathbf{s}_n)(\mathbf{H}_n \mathbf{s}_n)^T}{\mathbf{s}_n \cdot \mathbf{H}_n \mathbf{s}_n}, \\ \mathbf{B}_{n+1} &= \left(\mathbf{I} - \frac{\mathbf{s}_n \mathbf{y}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n} \right) \mathbf{B}_n \left(\mathbf{I} - \frac{\mathbf{y}_n \mathbf{s}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n} \right) + \frac{\mathbf{s}_n \mathbf{s}_n^T}{\mathbf{y}_n \cdot \mathbf{s}_n}. \end{aligned}$$

- (a) Verify the secant condition $\mathbf{H}_{n+1} \mathbf{s}_n = \mathbf{y}_n$.
- (b) Verify $\mathbf{H}_{n+1} \mathbf{B}_{n+1} = \mathbf{I}$, assuming that $\mathbf{H}_n \mathbf{B}_n = \mathbf{I}$.

Continued on the next page $\Rightarrow$

4.5. **(Curve fitting; Optional for undergraduates).** Consider a set of data consisting of four points

	1	2	3	4
x_i	0.0	1.0	2.0	3.0
y_i	1.1	2.6	7.2	21.1

Fit the data with a fitting function of the form

$$\hat{y}(x, \mathbf{p}) = a e^{bx}, \quad \text{where } \mathbf{p} = [a, b], \quad (4.63)$$

by minimizing the sum of the square-errors:

- Implement the three algorithms introduced in Section 4.5: the gradient descent method, the Gauss-Newton method, and the Levenberg-Marquardt method.
- Ignore the weight vector $\mathbf{W}$, i.e., set $\mathbf{W} = \mathbf{I}$.
- For each method, set $\mathbf{p}_0 = [a_0, b_0] = [1.0, 0.8]$.
- Discuss how to choose γ for the gradient descent and λ for the Levenberg-Marquardt.

Hint: The Jacobian for this example must be in $\mathbb{R}^{4 \times 2}$; more precisely,

$$\mathbf{J} = \frac{\partial}{\partial \mathbf{p}} \hat{\mathbf{y}}(x, \mathbf{p}) = \begin{bmatrix} 1 & 0 \\ e^b & a e^b \\ e^{2b} & 2a e^{2b} \\ e^{3b} & 3a e^{3b} \end{bmatrix},$$

because we have $\hat{\mathbf{y}}(x, \mathbf{p}) = [a, a e^b, a e^{2b}, a e^{3b}]^T$ from (4.63) and $\{x_i\}$.

CHAPTER 5

Popular Machine Learning Classifiers

In this chapter, we will study a selection of popular and powerful machine learning algorithms, which are commonly used in academia as well as in the industry. While learning about the differences between several supervised learning algorithms for classification, we will also develop an intuitive appreciation of their individual strengths and weaknesses.

The topics that we will learn about throughout this chapter are as follows:

- Introduction to the concepts of popular classification algorithms such as **logistic regression**, **support vector machine** (SVM), **decision trees**, and **k -nearest neighbors**.
- Questions to ask when selecting a machine learning algorithm
- Discussions about the strengths and weaknesses of classifiers with linear and nonlinear decision boundaries

Contents of Chapter 5

5.1. Logistic Sigmoid Function	91
5.2. Classification via Logistic Regression	95
5.3. Support Vector Machine	102
5.4. Decision Trees	122
5.5. k -Nearest Neighbors	129
Exercises for Chapter 5	131

Choosing a classification algorithm

Choosing an appropriate classification algorithm for a particular problem task requires practice:

- Each algorithm has **its own quirks/characteristics** and is based on certain assumptions.
- **No Free Lunch theorem**: No single classifier works best across all possible scenarios.
- In practice, it is recommended that you **compare the performance of at least a handful of different learning algorithms** to select **the best model** for the particular problem.

Eventually, the performance of a classifier, computational power as well as predictive power, depends heavily on the underlying data that are available for learning. The **five main steps** that are involved in training a machine learning algorithm can be summarized as follows:

1. Selection of features.
2. Choosing a performance metric.
3. Choosing a classifier and optimization algorithm.
4. Evaluating the performance of the model.
5. Tuning the algorithm.

5.1. Logistic Sigmoid Function

A **logistic sigmoid function** (or **logistic curve**) is a common “S” shape curve with equation:

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}, \quad (5.1)$$

where L denotes the curve’s maximum value, x_0 is the sigmoid’s midpoint, and k is the logistic growth rate or steepness of the curve.

In statistics, the **logistic model** is a widely used statistical model that uses a logistic function to model a binary dependent variable; many more complex extensions exist.

5.1.1. The standard logistic sigmoid function

Setting $L = 1$, $k = 1$, and $x_0 = 0$ gives the **standard logistic sigmoid function**:

$$s(x) = \frac{1}{1 + e^{-x}}. \quad (5.2)$$

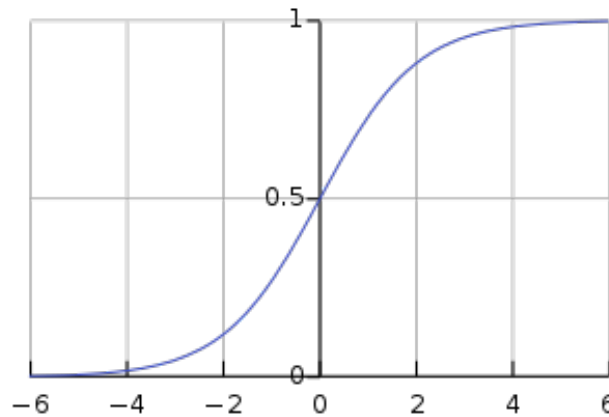


Figure 5.1: Standard logistic **sigmoid function** $s(x) = 1/(1 + e^{-x})$.

Remark 5.1. (The standard logistic sigmoid function):

$$s(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}$$

- The standard logistic function is the solution of the simple first-order non-linear ordinary differential equation

$$\frac{d}{dx}y = y(1 - y), \quad y(0) = \frac{1}{2}. \quad (5.3)$$

It can be verified easily as

$$s'(x) = \frac{e^x(1 + e^x) - e^x \cdot e^x}{(1 + e^x)^2} = \frac{e^x}{(1 + e^x)^2} = s(x)(1 - s(x)). \quad (5.4)$$

- s' is even: $s'(-x) = s'(x)$.
- **Rotational symmetry** about $(0, 1/2)$:

$$s(x) + s(-x) = \frac{1}{1 + e^{-x}} + \frac{1}{1 + e^x} = \frac{2 + e^x + e^{-x}}{2 + e^x + e^{-x}} \equiv 1. \quad (5.5)$$

- $\int s(x) dx = \int \frac{e^x}{1 + e^x} dx = \ln(1 + e^x)$, which is known as the **softplus function** in **artificial neural networks**. It is a smooth approximation of the the **rectifier** (an activation function) defined as

$$f(x) = x^+ = \max(x, 0). \quad (5.6)$$

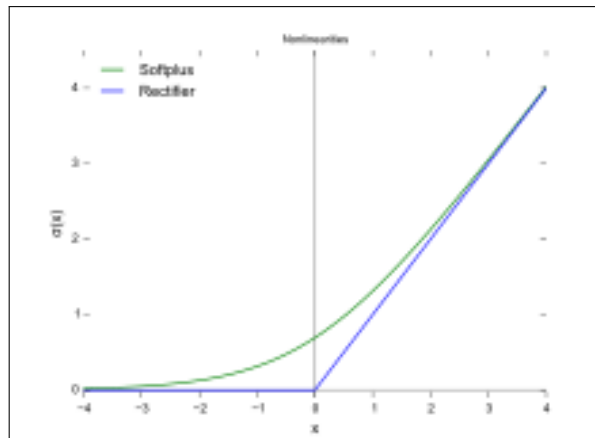


Figure 5.2: The rectifier and its smooth approximation, softplus function $\ln(1 + e^x)$.

5.1.2. The logit function

Logistic regression uses the sigmoid function for activation. We first wish **to explain the idea behind logistic regression as a probabilistic model.**

- Let p be the probability of a particular event (having class label $y = 1$).
- Then the **odds ratio** of the particular event is defined as

$$\frac{p}{1-p}.$$

- We can then define the **logit** function, which is simply the logarithm of the odds ratio (**log-odds**):

$$\text{logit}(p) = \ln \frac{p}{1-p}. \quad (5.7)$$

- The logit function takes input values in $(0, 1)$ and transforms them to values over the entire real line,

which we can use **to express a linear relationship between feature values and the log-odds:**

$$\text{logit}(p(y=1|\mathbf{x})) = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \mathbf{w}^T \mathbf{x}, \quad (5.8)$$

where $p(y=1|\mathbf{x})$ is the conditional probability that a particular sample (given its features $\mathbf{x}$) belongs to class 1.

Remark 5.2. What we are actually interested in is

predicting the probability

that a certain sample belongs to a particular class, which is the inverse form of the logit function:

$$p(y=1|\mathbf{x}) = \text{logit}^{-1}(\mathbf{w}^T \mathbf{x}). \quad (5.9)$$

Quesiton. What is the inverse of the logit function?

Example 5.3. Find the inverse of the logit function

$$\text{logit}(p) = \ln \frac{p}{1-p}.$$

Solution.

Ans: $\text{logit}^{-1}(z) = \frac{1}{1 + e^{-z}}$, the standard logistic sigmoid function.

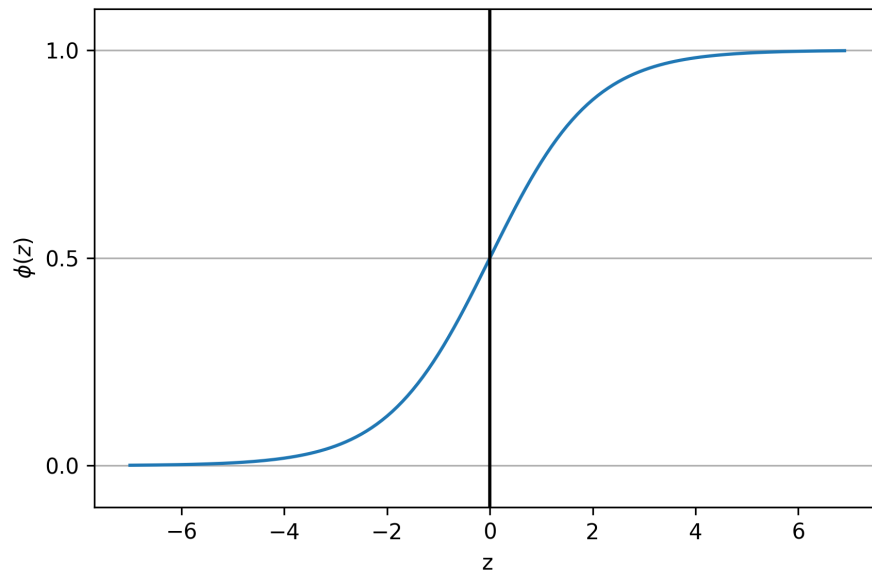


Figure 5.3: The standard logistic sigmoid function, again.

Note: The Sigmoid Function as an Activation Function

- When the standard logistic sigmoid function is adopted as an activation function, the prediction may be considered as the **probability** that a certain sample belongs to a particular class.
- This explains why the logistic sigmoid function is one of most popular activation functions.

5.2. Classification via Logistic Regression

Logistic regression is a probabilistic model.

- **Logistic regression maximizes the likelihood of the parameter w** ; in realization, it is similar to **Adaline**.
- Only the difference is the **activation function** (the sigmoid function), as illustrated in the figure:

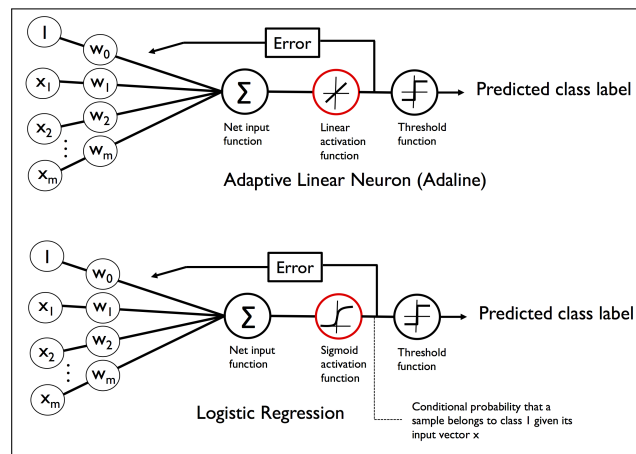


Figure 5.4: Adaline vs. Logistic regression.

- **The prediction** (the output of the sigmoid function) is interpreted as the **probability** of a particular sample belonging to class 1,

$$\phi(z) = p(y = 1 | \mathbf{x}; \mathbf{w}), \quad (5.10)$$

given its features $\mathbf{x}$ parameterized by the weights $\mathbf{w}$, $z = \mathbf{w}^T \mathbf{x}$.

Remark 5.4. Logistic Regression can be applied not only for **classification** (class labels) but also for **class-membership probability**.

- For example, logistic regression is used in **weather forecasting** (to predict the chance of rain).
- Similarly, it can be used to predict the **probability** that a patient has a particular disease given certain symptoms.
 - This is why logistic regression enjoys great popularity in the field of **medicine**.

5.2.1. The logistic cost function

Logistic regression incorporates a cost function in which **the likelihood is maximized**.¹

Definition 5.5. The **binomial distribution** with parameters n and $p \in [0, 1]$ is the discrete probability distribution of the number of successes in a sequence of n **independent** experiments, each asking a success–failure question, with probability of success being p .

- The **probability** of getting exactly k successes in n trials is given by the probability mass function

$$f(k, n, p) = P(k; n, p) = {}_nC_k p^k (1 - p)^{n-k}. \quad (5.11)$$

Definition 5.6. (Likelihood). Let $X_1, X_2, \dots, X_n$ have a joint density function $f(X_1, X_2, \dots, X_n | \theta)$. Given $X_1 = x_1, X_2 = x_2, \dots, X_n = x_n$ observed, the function of θ defined by

$$L(\theta) = L(\theta | x_1, x_2, \dots, x_n) = f(x_1, x_2, \dots, x_n | \theta) \quad (5.12)$$

is the **likelihood function**, or simply the **likelihood**.

Note: The **likelihood** describes the joint probability of the observed data, as a function of the parameters of the chosen statistical model.

- The **likelihood function** indicates which parameter values are more **likely** than others, in the sense that they would make the observed data more probable.
- The **maximum likelihood estimator** selects the parameter values ($\theta = \mathbf{w}$) that give the observed data the largest possible probability.

¹Note that the Adaline minimizes the sum-squared-error (SSE) cost function defined as $\mathcal{J}(\mathbf{w}) = \frac{1}{2} \sum_i \left(\phi(z^{(i)}) - y^{(i)} \right)^2$, where $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$, using the gradient descent method; see Section 3.3.1.

Derivation of the Logistic Cost Function

- Assume that the individual samples in our dataset are **independent** of one another. Then we can define the **likelihood** L as

$$\begin{aligned} L(\mathbf{w}) &= P(\mathbf{y}|\mathbf{x}; \mathbf{w}) = \prod_{i=1}^n P(y^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}) \\ &= \prod_{i=1}^n \left(\phi(z^{(i)}) \right)^{y^{(i)}} \left(1 - \phi(z^{(i)}) \right)^{1-y^{(i)}}, \end{aligned} \quad (5.13)$$

where $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$.

- In practice, it is easier to maximize the (natural) log of this equation, which is called the **log-likelihood function**:

$$\ell(\mathbf{w}) = \ln(L(\mathbf{w})) = \sum_{i=1}^n \left[y^{(i)} \ln \left(\phi(z^{(i)}) \right) + (1 - y^{(i)}) \ln \left(1 - \phi(z^{(i)}) \right) \right]. \quad (5.14)$$

Remark 5.7. Log-Likelihood

- Firstly, applying the log function reduces the potential for **numerical underflow**, which can occur if the likelihoods are very small.
- Secondly, we can convert the product of factors into a summation of factors, which makes it easier to obtain the **derivative** of this function via the addition trick, as you may remember from calculus.
- We can adopt **the negation of the log-likelihood** as **a cost function** $\mathcal{J}$ that can be minimized using gradient descent.

Now, we define the **logistic cost function** to be minimized:

$$\mathcal{J}(\mathbf{w}) = \sum_{i=1}^n \left[-\mathbf{y}^{(i)} \ln \left(\phi(\mathbf{z}^{(i)}) \right) - (1 - \mathbf{y}^{(i)}) \ln \left(1 - \phi(\mathbf{z}^{(i)}) \right) \right], \quad (5.15)$$

where $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$.

Note: Looking at the equation, we can see that the first term becomes zero if $\mathbf{y}^{(i)} = 0$, and the second term becomes zero if $\mathbf{y}^{(i)} = 1$.

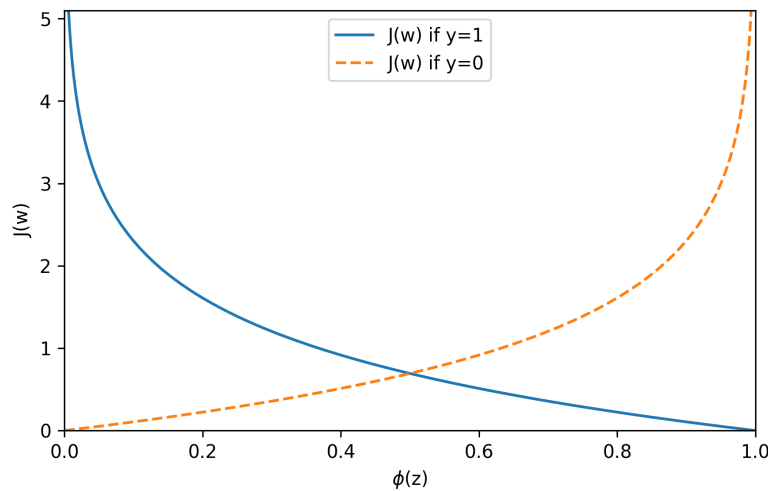


Figure 5.5: Plot of $\mathcal{J}(\mathbf{w})$, when $n = 1$ (one single-sample):

$$\mathcal{J}(\mathbf{w}) = \begin{cases} -\ln(\phi(\mathbf{z})), & \text{if } y = 1, \\ -\ln(1 - \phi(\mathbf{z})), & \text{if } y = 0. \end{cases}$$

Observation 5.8. We can see that

- **(Solid curve, in blue).** If we correctly predict that a sample belongs to class 1, the cost approaches 0.
- **(Dashed curve, in orange).** If we correctly predict $y = 0$, the cost also approaches 0.
- However, if the prediction is **wrong**, the cost goes towards **infinity**.
- Here, the main point is that we **penalize wrong predictions** with an increasingly larger cost, which will enforce the model to fit the sample.
- For general $n \geq 1$, it would try to fit all the samples in the training dataset.

5.2.2. Gradient descent learning for logistic regression

Let's start by calculating the partial derivative of the logistic cost function (5.15) with respect to the j -th weight, w_j :

$$\frac{\partial \mathcal{J}(\mathbf{w})}{\partial w_j} = \sum_{i=1}^n \left[-y^{(i)} \frac{1}{\phi(z^{(i)})} + (1 - y^{(i)}) \frac{1}{1 - \phi(z^{(i)})} \right] \frac{\partial \phi(z^{(i)})}{\partial w_j}, \quad (5.16)$$

where, using $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$ and (5.4),

$$\frac{\partial \phi(z^{(i)})}{\partial w_j} = \phi'(z^{(i)}) \frac{\partial z^{(i)}}{\partial w_j} = \phi(z^{(i)}) (1 - \phi(z^{(i)})) x_j^{(i)}.$$

Thus, it follows from the above and (5.16) that

$$\begin{aligned} \frac{\partial \mathcal{J}(\mathbf{w})}{\partial w_j} &= \sum_{i=1}^n \left[-y^{(i)} (1 - \phi(z^{(i)})) + (1 - y^{(i)}) \phi(z^{(i)}) \right] x_j^{(i)} \\ &= - \sum_{i=1}^n \left[y^{(i)} - \phi(z^{(i)}) \right] x_j^{(i)} \end{aligned}$$

and therefore

$$\nabla \mathcal{J}(\mathbf{w}) = - \sum_{i=1}^n \left[y^{(i)} - \phi(z^{(i)}) \right] \mathbf{x}^{(i)}. \quad (5.17)$$

Algorithm 5.9. Gradient descent learning for logistic regression is formulated as

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \quad (5.18)$$

where $\eta > 0$ is the **step length** (learning rate) and

$$\Delta \mathbf{w} = -\eta \nabla \mathcal{J}(\mathbf{w}) = \eta \sum_{i=1}^n \left[y^{(i)} - \phi(z^{(i)}) \right] \mathbf{x}^{(i)}. \quad (5.19)$$

Note: The above gradient descent rule for logistic regression has the same shape as that of Adaline; see (3.15) on p. 48. Only the difference is the activation function $\phi(z)$.

5.2.3. Regularization: bias-variance tradeoff

- **Overfitting** is a common problem in ML.
 - If a model performs well on the training data but does not generalize well to unseen (test) data, then it is most likely **the sign of overfitting**.
 - Due to a **high variance**, from **randomness** (noise) in the training data.
 - **Variance** measures the consistency (or variability) of the model prediction for a particular sample instance.
- Similarly, our model can also suffer from **underfitting**.
 - Our model is **not complex enough** to capture the pattern in the training data well, and therefore also suffers from low performance on unseen data.
 - Due to a **high bias**.
 - **Bias** is the measure of the **systematic error** that is not due to randomness.

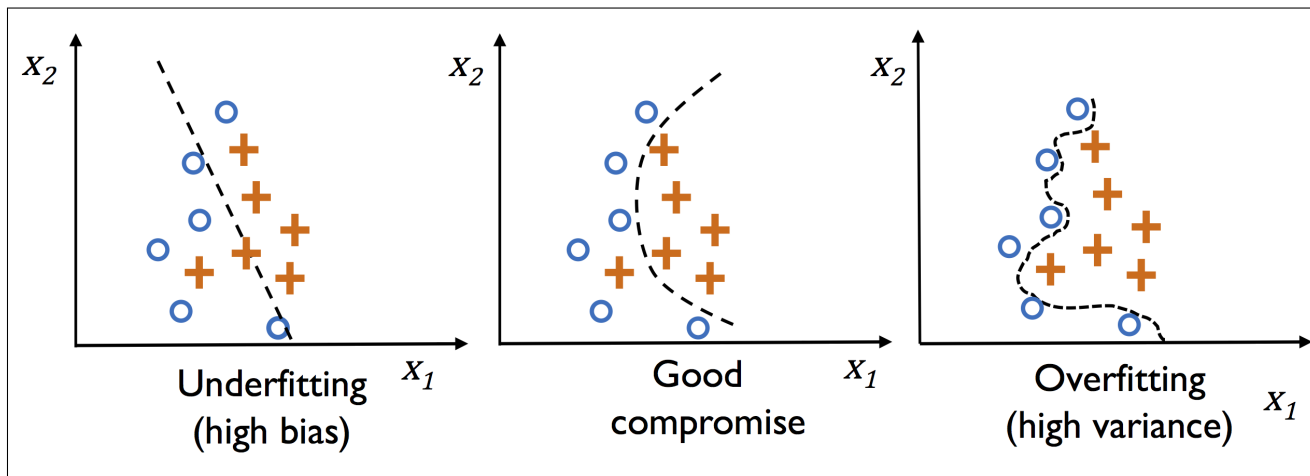


Figure 5.6

Regularization

- One way of finding a good **bias-variance tradeoff**.
- It is useful to prevent overfitting, also handling
 - collinearity (high correlation among features)
 - filter-out noise from data
 - **multiple local minima problem**
- The concept behind **regularization** is to **introduce additional information (bias)** to penalize extreme parameter (weight) values.
- The most common form of regularization is so-called L^2 **regularization** (sometimes also called L^2 **shrinkage** or **weight decay**):

$$\frac{\lambda}{2} \|\mathbf{w}\|^2 = \frac{\lambda}{2} \sum_{j=1}^m w_j^2, \quad (5.20)$$

where λ is the regularization parameter.

The cost function for logistic regression can be regularized by adding a simple regularization term, which will *shrink the weights* during model training: for $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$,

$$\mathcal{J}(\mathbf{w}) = \sum_{i=1}^n \left[-y^{(i)} \ln \left(\phi(z^{(i)}) \right) - (1 - y^{(i)}) \ln \left(1 - \phi(z^{(i)}) \right) \right] + \frac{\lambda}{2} \|\mathbf{w}\|^2. \quad (5.21)$$

Note: Regularization

- Regularization is another reason why **feature scaling** such as **standardization** is important.
- For regularization to work properly, we need to ensure that **all our features are on comparable scales**.
- Then, via the regularization parameter λ , we can control how well we fit the training data while keeping the weights small. By increasing the value of λ , we increase the **regularization strength**.
- See § 6.3 for details on feature scaling.

5.3. Support Vector Machine

- **Support vector machine** (SVM), developed in 1995 by Cortes-Vapnik [12], can be considered as an extension of the Perceptron/Adaline, *which maximizes the margin*.
- The **rationale** behind having decision boundaries with large margins is that they tend to have a **lower generalization error**, whereas **models with small margins are more prone to overfitting**.

5.3.1. Linear SVM

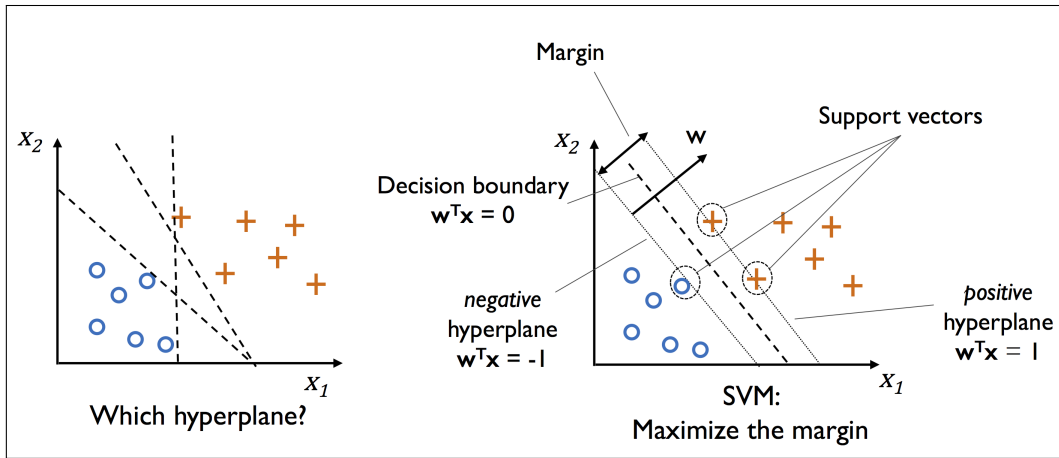


Figure 5.7: Linear support vector machine.

To find an optimal hyperplane that maximizes the margin, let's begin with considering the **positive** and **negative** hyperplanes that are parallel to the decision boundary:

$$\begin{aligned} w_0 + \mathbf{w}^T \mathbf{x}_+ &= 1, \\ w_0 + \mathbf{w}^T \mathbf{x}_- &= -1. \end{aligned} \quad (5.22)$$

where $\mathbf{w} = [w_1, w_2, \dots, w_d]^T$. If we subtract those two linear equations from each other, then we have

$$\mathbf{w} \cdot (\mathbf{x}_+ - \mathbf{x}_-) = 2$$

and therefore

$$\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot (\mathbf{x}_+ - \mathbf{x}_-) = \frac{2}{\|\mathbf{w}\|}. \quad (5.23)$$

Note: $\mathbf{w} = [w_1, w_2, \dots, w_d]^T$ is a **normal vector**^a to the decision boundary (a hyperplane) so that the left side of (5.23) is the distance between the positive and negative hyperplanes.

^aSee Exercise 5.1.

Maximizing the distance (margin) is equivalent to minimizing its reciprocal $\frac{1}{2}\|\mathbf{w}\|$, or minimizing $\frac{1}{2}\|\mathbf{w}\|^2$.

Problem 5.10. The **linear SVM** is formulated as

$$\begin{aligned} \min_{\mathbf{w}, w_0} \quad & \frac{1}{2}\|\mathbf{w}\|^2, \quad \text{subject to} \\ & \begin{cases} w_0 + \mathbf{w}^T \mathbf{x}^{(i)} \geq 1 & \text{if } y^{(i)} = 1, \\ w_0 + \mathbf{w}^T \mathbf{x}^{(i)} \leq -1 & \text{if } y^{(i)} = -1. \end{cases} \end{aligned} \quad (5.24)$$

Remark 5.11. The constraints in Problem 5.10 can be written as

$$y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) - 1 \geq 0, \quad \forall i. \quad (5.25)$$

- The beauty of linear SVM is that if the data is linearly separable, there is a unique global minimum value.
- An ideal SVM analysis should produce a hyperplane that completely separates the vectors (cases) into two non-overlapping classes.
- However, perfect separation may not be possible, or it may result in a model with so many cases that the model does not classify correctly.

Note: Constrained optimization problems such as (5.24) are typically solved using the method of Lagrange multipliers.

5.3.2. The method of Lagrange multipliers

In this subsection, we briefly consider Lagrange's method to solve the problem of the form

$$\min / \max_{\mathbf{x}} f(\mathbf{x}) \quad \text{subj.to} \quad g(\mathbf{x}) = c. \quad (5.26)$$

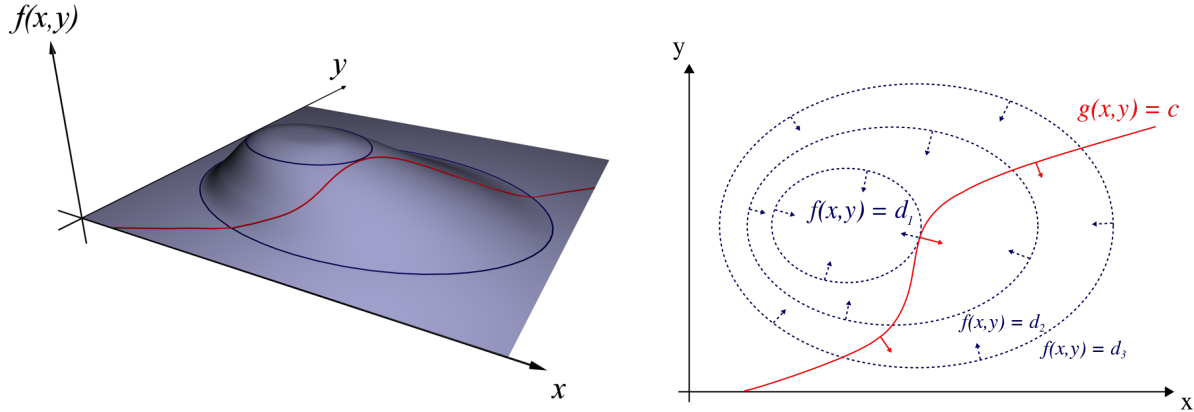


Figure 5.8: The method of Lagrange multipliers in $\mathbb{R}^2$: $\nabla f \parallel \nabla g$, at optimum.

Strategy 5.12. (Method of Lagrange multipliers). For the maximum and minimum values of $f(\mathbf{x})$ **subject to** $g(\mathbf{x}) = c$,

(a) Find $\mathbf{x}$ and λ such that

$$\nabla f(\mathbf{x}) = \lambda \nabla g(\mathbf{x}) \quad \text{and} \quad g(\mathbf{x}) = c.$$

(b) Evaluate f at all these points, to find the maximum and minimum.

Self-study 5.13. Use the method of Lagrange multipliers to find the extreme values of $f(x, y) = x^2 + 2y^2$ on the circle $x^2 + y^2 = 1$.

Hint: $\nabla f = \lambda \nabla g \implies \begin{bmatrix} 2x \\ 4y \end{bmatrix} = \lambda \begin{bmatrix} 2x \\ 2y \end{bmatrix}$. Therefore,
$$\begin{cases} 2x = 2x \lambda & \textcircled{1} \\ 4y = 2y \lambda & \textcircled{2} \\ x^2 + y^2 = 1 & \textcircled{3} \end{cases}$$

From $\textcircled{1}$, $x = 0$ or $\lambda = 1$.

Ans: min: $f(\pm 1, 0) = 1$; max: $f(0, \pm 1) = 2$

Lagrange multipliers – Dual variables

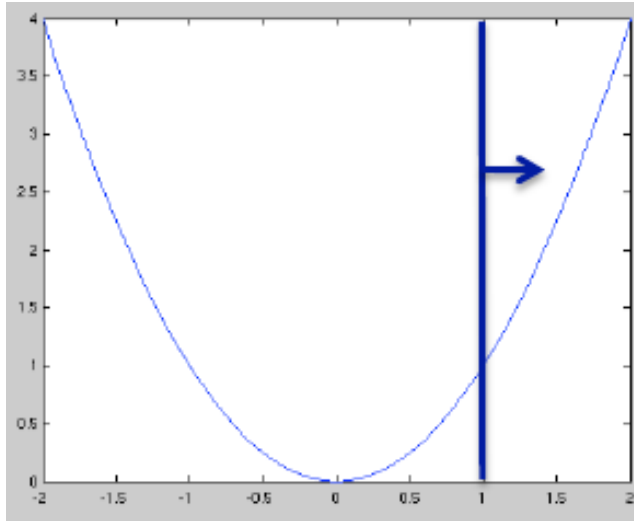


Figure 5.9: $\min_x x^2$ subj.to $x \geq 1$.

For simplicity, consider

$$\min_x x^2 \quad \text{subj.to } x \geq 1. \quad (5.27)$$

Rewriting the constraint

$$x - 1 \geq 0,$$

introduce **Lagrangian (objective)**:

$$\mathcal{L}(x, \alpha) = x^2 - \alpha(x - 1). \quad (5.28)$$

Now, consider

$$\min_x \max_{\alpha} \mathcal{L}(x, \alpha) \quad \text{subj.to } \alpha \geq 0. \quad (5.29)$$

Claim 5.14. The minimization problem (5.27) is equivalent to the min-max problem (5.29).

Proof. ① Let $x > 1$. $\Rightarrow \max_{\alpha \geq 0} \{-\alpha(x - 1)\} = 0$ and $\alpha^* = 0$. Thus,

$$\mathcal{L}(x, \alpha) = x^2. \quad (\text{original objective})$$

② Let $x = 1$. $\Rightarrow \max_{\alpha \geq 0} \{-\alpha(x - 1)\} = 0$ and α is arbitrary. Thus, again,

$$\mathcal{L}(x, \alpha) = x^2. \quad (\text{original objective})$$

③ Let $x < 1$. $\Rightarrow \max_{\alpha \geq 0} \{-\alpha(x - 1)\} = \infty$. However, $\min_x$ won't make this happen! ($\min_x$ is fighting $\max_{\alpha}$) That is, when $x < 1$, the objective $\mathcal{L}(x, \alpha)$ becomes huge as α grows; then, $\min_x$ will push $x \nearrow 1$ or increase it to become $x \geq 1$. In other words, $\min_x$ **forces** $\max_{\alpha}$ **to behave, so constraints will be satisfied.** $\square$

Now, the goal is **to solve (5.29)**. In the following, we will define the **dual problem** of (5.29), which is equivalent to the **primal problem**.

Recall: The min-max problem in (5.29), which is equivalent to the (original) primal problem:

$$\min_x \max_{\alpha} \mathcal{L}(x, \alpha) \quad \text{subj.to} \quad \alpha \geq 0, \quad (\text{Primal}) \quad (5.30)$$

where

$$\mathcal{L}(x, \alpha) = x^2 - \alpha(x - 1).$$

Definition 5.15. The **dual problem** of (5.30) is formulated by swapping $\min_x$ and $\max_{\alpha}$ as follows:

$$\max_{\alpha} \min_x \mathcal{L}(x, \alpha) \quad \text{subj.to} \quad \alpha \geq 0, \quad (\text{Dual}) \quad (5.31)$$

The term $\min_x \mathcal{L}(x, \alpha)$ is called the **Lagrange dual function** and the Lagrange multiplier α is also called the **dual variable**.

How to solve it. For the Lagrange dual function $\min_x \mathcal{L}(x, \alpha)$, the minimum occurs where the gradient is equal to zero.

$$\frac{d}{dx} \mathcal{L}(x, \alpha) = 2x - \alpha = 0 \Rightarrow x = \frac{\alpha}{2}. \quad (5.32)$$

Plugging this to $\mathcal{L}(x, \alpha)$, we have

$$\mathcal{L}(x, \alpha) = \left(\frac{\alpha}{2}\right)^2 - \alpha\left(\frac{\alpha}{2} - 1\right) = \alpha - \frac{\alpha^2}{4}.$$

We can rewrite the dual problem (5.31) as

$$\max_{\alpha \geq 0} \left[\alpha - \frac{\alpha^2}{4} \right]. \quad (\text{Dual}) \quad (5.33)$$

$\Rightarrow$ **the maximum is 1 when $\alpha^* = 2$ (for the dual problem).**

Plugging $\alpha = \alpha^*$ into (5.32) to get $x^* = 1$. Or, using the Lagrangian objective, we have

$$\mathcal{L}(x, \alpha) = x^2 - 2(x - 1) = (x - 1)^2 + 1. \quad (5.34)$$

$\Rightarrow$ **the minimum is 1 when $x^* = 1$ (for the primal problem).** $\square$

5.3.3. Karush-Kuhn-Tucker conditions and Complementary slackness

Allowing **inequality constraints**, the **KKT approach** generalizes the **method of Lagrange multipliers** which allows only equality constraints.

Recall: The **linear SVM** formulated in Problem 5.10:

$$\min_{\mathbf{w}, w_0} \frac{1}{2} \|\mathbf{w}\|^2, \quad \text{subj.to} \quad \quad \quad \text{(Primal)} \quad (5.35)$$

$$y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) - 1 \geq 0, \quad \forall i = 1, 2, \dots, N.$$

To solve the problem, let's begin with its **Lagrangian**:

$$\mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \alpha_i [y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) - 1], \quad (5.36)$$

where $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_N]^T$, the **dual variable (Lagrange multipliers)**.

- The primal problem of the SVM is formulated equivalently as

$$\min_{\mathbf{w}, w_0} \max_{\boldsymbol{\alpha}} \mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) \quad \text{subj.to} \quad \alpha \geq 0, \quad \text{(Primal)} \quad (5.37)$$

while its dual problem reads

$$\max_{\boldsymbol{\alpha}} \min_{\mathbf{w}, w_0} \mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) \quad \text{subj.to} \quad \alpha \geq 0. \quad \text{(Dual)} \quad (5.38)$$

- Solve the “min” problem of (5.38) first, using calculus techniques.

Definition 5.16. **Karush-Kuhn-Tucker (KKT) conditions**

In optimization, the **KKT conditions** [36, 42] are **first derivative tests** for a solution in nonlinear programming to be optimized. It is also called the **first-order necessary conditions**.

Writing the **KKT conditions**, starting with Lagrangian stationarity, where we need to find the first-order derivatives w.r.t. $\mathbf{w}$ and w_0 :

$$\begin{aligned}
\nabla_{\mathbf{w}} \mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) &= \mathbf{w} - \sum_{i=1}^N \alpha_i y^{(i)} \mathbf{x}^{(i)} = 0 \Rightarrow \mathbf{w} = \sum_{i=1}^N \alpha_i y^{(i)} \mathbf{x}^{(i)}, \\
\frac{\partial}{\partial w_0} \mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) &= - \sum_{i=1}^N \alpha_i y^{(i)} = 0 \Rightarrow \sum_{i=1}^N \alpha_i y^{(i)} = 0, \\
\alpha_i &\geq 0, & (\text{dual feasibility}) \\
\alpha_i [y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) - 1] &= 0, & (\text{complementary slackness}) \\
y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) - 1 &\geq 0. & (\text{primal feasibility})
\end{aligned} \tag{5.39}$$

Complementary slackness will be discussed in detail on page 111.

Using the KKT conditions (5.39), we can simplify the Lagrangian:

$$\begin{aligned}
\mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\alpha}) &= \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \alpha_i y^{(i)} w_0 - \sum_{i=1}^N \alpha_i y^{(i)} \mathbf{w}^T \mathbf{x}^{(i)} + \sum_{i=1}^N \alpha_i \\
&= \frac{1}{2} \|\mathbf{w}\|^2 - 0 - \mathbf{w}^T \mathbf{w} + \sum_{i=1}^N \alpha_i \\
&= -\frac{1}{2} \|\mathbf{w}\|^2 + \sum_{i=1}^N \alpha_i.
\end{aligned} \tag{5.40}$$

Again using the first KKT condition, we can rewrite the first term.

$$\begin{aligned}
-\frac{1}{2} \|\mathbf{w}\|^2 &= -\frac{1}{2} \left(\sum_{i=1}^N \alpha_i y^{(i)} \mathbf{x}^{(i)} \right) \cdot \left(\sum_{j=1}^N \alpha_j y^{(j)} \mathbf{x}^{(j)} \right) \\
&= -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}.
\end{aligned} \tag{5.41}$$

Plugging (5.41) into the (simplified) Lagrangian (5.40), we see that the Lagrangian now depends on $\boldsymbol{\alpha}$ only.

Problem 5.17. The **dual problem** of (5.35) is formulated as

$$\begin{aligned} \max_{\alpha} \left[\sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right], \quad \text{subj.to} \\ \left[\begin{array}{l} \alpha_i \geq 0, \quad \forall i, \\ \sum_{i=1}^N \alpha_i y^{(i)} = 0. \end{array} \right. \end{aligned} \quad (5.42)$$

Remark 5.18. (Solving the dual problem).

- We can solve the dual problem (5.42), by using either a generic quadratic programming solver or the **Sequential Minimal Optimization (SMO)**, which we will discuss in § 5.3.6, p. 120.
- For now, **assume** that we solved it to have $\alpha^* = [\alpha_1^*, \dots, \alpha_n^*]^T$.
- Then we can plug it into the first KKT condition to get

$$\mathbf{w}^* = \sum_{i=1}^N \alpha_i^* y^{(i)} \mathbf{x}^{(i)}. \quad (5.43)$$

- We still need to get w_0^* .

Remark 5.19. The objective function $\mathcal{L}(\alpha)$ in (5.42) is a linear combination of the dot products of data samples $\{\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}\}$, which will be used when we generalize the SVM for nonlinear decision boundaries; see § 5.3.5.

Support vectors

Assume momentarily that we have w_0^* . Consider the **complementary slackness KKT condition** along with the primal and dual feasibility conditions:

$$\begin{aligned} \alpha_i^* [y^{(i)}(w_0^* + \mathbf{w}^{*T} \mathbf{x}^{(i)}) - 1] &= 0 \\ \Rightarrow \begin{cases} \alpha_i^* > 0 \Rightarrow y^{(i)}(w_0^* + \mathbf{w}^{*T} \mathbf{x}^{(i)}) = 1 \\ \alpha_i^* < 0 \quad (\text{can't happen}) \\ y^{(i)}(w_0^* + \mathbf{w}^{*T} \mathbf{x}^{(i)}) - 1 > 0 \Rightarrow \alpha_i^* = 0 \\ y^{(i)}(w_0^* + \mathbf{w}^{*T} \mathbf{x}^{(i)}) - 1 < 0 \quad (\text{can't happen}). \end{cases} \end{aligned} \quad (5.44)$$

We define the **optimal (scaled) scoring function**:

$$f^*(\mathbf{x}^{(i)}) = w_0^* + \mathbf{w}^{*T} \mathbf{x}^{(i)}. \quad (5.45)$$

Then

$$\begin{cases} \alpha_i^* > 0 & \Rightarrow y^{(i)} f^*(\mathbf{x}^{(i)}) = \text{scaled margin} = 1, \\ y^{(i)} f^*(\mathbf{x}^{(i)}) > 1 & \Rightarrow \alpha_i^* = 0. \end{cases} \quad (5.46)$$

Definition 5.20. The examples in the first category, for which the scaled margin is 1 and the constraints are active, are called **support vectors**. They are the closest to the decision boundary.

Finding the optimal value of w_0

To get w_0^* , use the primal feasibility condition:

$$y^{(i)}(w_0^* + \mathbf{w}^{*T} \mathbf{x}^{(i)}) \geq 1 \quad \text{and} \quad \min_i y^{(i)}(w_0^* + \mathbf{w}^{*T} \mathbf{x}^{(i)}) = 1.$$

If you take a positive support vector ($y^{(i)} = 1$), then

$$w_0^* = 1 - \min_{i: y^{(i)}=1} \mathbf{w}^{*T} \mathbf{x}^{(i)}. \quad (5.47)$$

Here, you'd better refer to **Summary of SVM** in Algorithm 5.27, p. 115.

Complementary Slackness

Definition 5.21. Types of Constraints

- A **binding constraint** is one where some optimal solution is on the hyperplane for the constraint.
- A **non-binding constraint** is one where no optimal solution is on the line for the constraint.
- A **redundant constraint** is one whose removal would not change the feasible region.

Theorem 5.22. Complementary Slackness

Assume the primal problem (P) has a solution w^* and the dual problem (D) has a solution α^* .

- (a) If $w_j^* > 0$, then the j -th constraint in (D) is binding.
- (b) If $\alpha_i^* > 0$, then the i -th constraint in (P) is binding.

The term **complementary slackness** refers to a relationship between the slackness in a primal constraint and the slackness (positivity) of the associated dual variable.

- Notice that the number of variables in the dual is the same as the number of constraints in the primal, and the number of constraints in the dual is equal to the number of variables in the primal.
- This correspondence suggests that variables in one problem are complementary to constraints in the other.
- We say that **a constraint has slack if it is not binding**.

Example 5.23. The **contrapositive** statement of Theorem 5.22 (b):

If the i -th constraint in (P) is not binding, then $\alpha_i^* = 0$.

or, equivalently,

If the i -th constraint in (P) has slack, then $\alpha_i^* = 0$.

See (5.46).

5.3.4. The inseparable case: Soft-margin classification

When the dataset is inseparable, there would be no separating hyperplane; there is no feasible solution to the linear SVM.

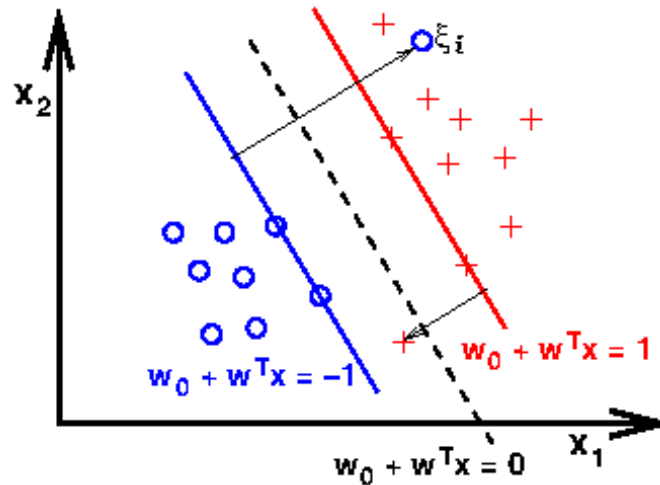


Figure 5.10: Slack variable: ξ_i .

Let's fix our SVM so it can accommodate the inseparable case.

- The new formulation involves the **slack variable**; it allows some instances to fall off the margin, but penalize them.
- So we are allowed to make mistakes now, but we pay a price.

Remark 5.24. The motivation for introducing the slack variable ξ is:

1. **The linear constraints need to be relaxed** for inseparable data.
2. Allow the optimization to converge
 - **under appropriate cost penalization,**
 - **in the presence of misclassifications.**

Such strategy of the SVM is called the **soft-margin classification**.

Recall: The **linear SVM** formulated in Problem 5.10:

$$\begin{aligned} \min_{\mathbf{w}, w_0} \quad & \frac{1}{2} \|\mathbf{w}\|^2, \quad \text{subj.to} \\ & y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) - 1 \geq 0, \quad \forall i. \end{aligned} \quad (\text{Primal}) \quad (5.48)$$

Let's change it to this new primal problem:

Problem 5.25. (Soft-margin classification). The SVM with the slack variable is formulated as

$$\begin{aligned} \min_{\mathbf{w}, w_0, \xi} \quad & \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^N \xi_i \quad \text{subj.to} \\ & \begin{cases} y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) \geq 1 - \xi_i, \\ \xi_i \geq 0. \end{cases} \end{aligned} \quad (\text{Primal}) \quad (5.49)$$

Via the variable C , we can then control the penalty for misclassification. **Large values of C** correspond to large error penalties, whereas we are less strict about misclassification errors if we choose smaller values for C . We can then use the C parameter to control the width of the margin and therefore tune the **bias-**

variance trade-off, as illustrated in the following figure:

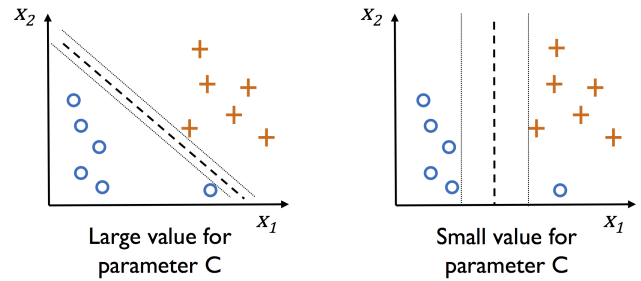


Figure 5.11: Bias-variance trade-off, via C .

The constraints allow some slack of size ξ_i , but we pay a price for it in the objective. That is,

if $y^{(i)} f(\mathbf{x}^{(i)}) \geq 1$, then $\xi_i = 0$ and penalty is 0. Otherwise, $y^{(i)} f(\mathbf{x}^{(i)}) = 1 - \xi_i$ and we pay price $\xi_i > 0$

The Dual for soft-margin classification

Form the Lagrangian of (5.49):

$$\begin{aligned} \mathcal{L}([\mathbf{w}, w_0], \boldsymbol{\xi}, \boldsymbol{\alpha}, \mathbf{r}) = & \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^N \xi_i - \sum_{i=1}^N r_i \xi_i \\ & - \sum_{i=1}^N \alpha_i [y^{(i)}(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) - 1 + \xi_i], \end{aligned} \quad (5.50)$$

where α_i 's and r_i 's are Lagrange multipliers (constrained to be ≥ 0).

After some work, the dual turns out to be

Problem 5.26. The dual problem of (5.48) is formulated as

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \left[\sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right], \quad \text{subj.to} \\ \left[\begin{array}{l} 0 \leq \alpha_i \leq C, \quad \forall i, \\ \sum_{i=1}^N \alpha_i y^{(i)} = 0. \end{array} \right. \end{aligned} \quad (5.51)$$

So the only difference from the original problem's dual, (5.42), is that

$\alpha_i \geq 0$ is changed to $0 \leq \alpha_i \leq C$. Neat!

See § 5.3.6, p. 120, for the solution of (5.51), using the SMO algorithm.

Algebraic expression for the dual problem:

Let

$$Z = \begin{bmatrix} y^{(1)} \mathbf{x}^{(1)} \\ y^{(2)} \mathbf{x}^{(2)} \\ \vdots \\ y^{(N)} \mathbf{x}^{(N)} \end{bmatrix} \in \mathbb{R}^{N \times m}, \quad \mathbf{1} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^N.$$

Then **dual problem (5.51)** can be written as

$$\max_{0 \leq \alpha \leq C} \left[\alpha \cdot \mathbf{1} - \frac{1}{2} \alpha^T Z Z^T \alpha \right] \quad \text{subj.to} \quad \alpha \cdot \mathbf{y} = 0. \quad (5.52)$$

Note:

- $G = Z Z^T \in \mathbb{R}^{N \times N}$ is called the **Gram matrix**. That is,

$$G_{ij} = y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}. \quad (5.53)$$

- The optimization problem (5.52) is a typical **quadratic programming** (QP) problem.
- It admits a **unique solution**.

Algorithm 5.27. (Summary of SVM)• **Training**

- Compute Gram matrix: $G_{ij} = y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}$
- Solve QP to get α^* (Chapter 11, or § 5.3.6)
- Compute the weights: $\mathbf{w}^* = \sum_{i=1}^N \alpha_i^* y^{(i)} \mathbf{x}^{(i)}$ (5.43)
- Compute the intercept: $w_0^* = 1 - \min_{i: y^{(i)}=1} \mathbf{w}^{*T} \mathbf{x}^{(i)}$ (5.47)

• **Classification** (for a new sample $\mathbf{x}$)

- Compute $k_i = \mathbf{x} \cdot \mathbf{x}^{(i)}$ for support vectors $\mathbf{x}^{(i)}$
- Compute $f(\mathbf{x}) = w_0^* + \sum_i \alpha_i^* y^{(i)} k_i$ ($:= w_0^* + \mathbf{w}^{*T} \mathbf{x}$) (5.24)
- Test $\text{sign}(f(\mathbf{x}))$.

5.3.5. Nonlinear SVM and kernel trick

Note: A reason why the SVM is popular is that

- It can be **easily kernelized** to solve nonlinear classification problems incorporating **linearly inseparable data**.

The basic idea behind **kernel methods** is

- To transform the data to **a higher-dimensional space where the data becomes linearly separable**.

For example, for the inseparable data set in Figure 5.12, we define

$$\phi(x_1, x_2) = (x_1, x_2, x_1^2 + x_2^2).$$

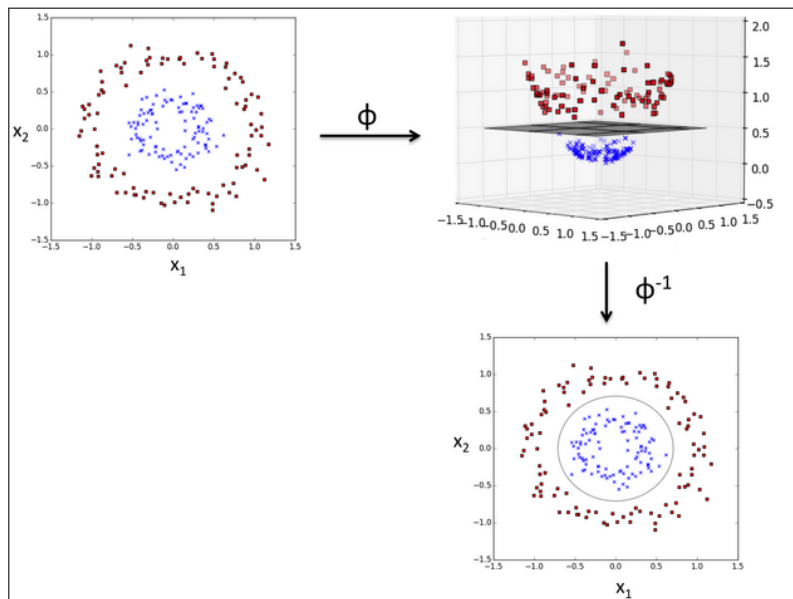


Figure 5.12: Inseparable dataset, feature expansion, and kernel SVM.

To solve a nonlinear problem using an SVM, we would

- ① **Transform the training data to a higher-dimensional space**, via a mapping ϕ , and ② **train a linear SVM model**.
- Then, **for new unseen data**, **classify using** ① **the same mapping ϕ to transform** and ② **the same linear SVM model**.

Kernel Trick

Recall: the **dual problem** to the soft-margin SVM given in (5.51):

$$\begin{aligned} \max_{\alpha} \left[\sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right], \quad \text{subj.to} \\ \left[\begin{array}{l} 0 \leq \alpha_i \leq C, \quad \forall i, \\ \sum_{i=1}^N \alpha_i y^{(i)} = 0. \end{array} \right. \end{aligned} \quad (5.54)$$

Observation 5.28. The objective is a linear combination of dot products $\{\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}\}$. Thus,

- If the kernel SVM transforms the data samples through ϕ , the dot product $\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}$ must be replaced by $\phi(\mathbf{x}^{(i)}) \cdot \phi(\mathbf{x}^{(j)})$.
- The dot product $\phi(\mathbf{x}^{(i)}) \cdot \phi(\mathbf{x}^{(j)})$ is performed in a higher-dimension, which may be costly.

Definition 5.29. In order **to save the expensive step of explicit computation** of this dot product (in a higher-dimension), we define a so-called **kernel function**:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \approx \phi(\mathbf{x}^{(i)}) \cdot \phi(\mathbf{x}^{(j)}). \quad (5.55)$$

One of the most widely used kernels is the **Radial Basis Function (RBF)** kernel or simply called the **Gaussian kernel**:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp \left(- \frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2} \right) = \exp \left(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2 \right), \quad (5.56)$$

where $\gamma = 1/(2\sigma^2)$. Occasionally, the parameter γ plays an important role in controlling overfitting.

Note: Roughly speaking, the term **kernel** can be interpreted as a **similarity function** between a pair of samples.

This is the big picture behind the kernel trick.

Kernel SVM: It can also be summarized as in **Algorithm 5.27**, p. 115; **only the difference** is that dot products $\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}$ and $\mathbf{x} \cdot \mathbf{x}^{(i)}$ are replaced by $\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ and $\mathcal{K}(\mathbf{x}, \mathbf{x}^{(i)})$, respectively.

Common Kernels

- Polynomial of degree exactly k (e.g. $k = 2$):

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = (\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)})^k \quad (5.57)$$

- Polynomial of degree up to k : for some $c > 0$,

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = (c + \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)})^k \quad (5.58)$$

- Sigmoid:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \tanh(a \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} + b) \quad (5.59)$$

- Gaussian RBF:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2}\right) \quad (5.60)$$

- And many others: Fisher kernel, graph kernel, string kernel, ...
very active area of research!

Example 5.30. (Quadratic kernels). Let $\mathcal{K}(\mathbf{x}, \mathbf{z}) = (c + \mathbf{x} \cdot \mathbf{z})^2$:

$$\begin{aligned} (c + \mathbf{x} \cdot \mathbf{z})^2 &= \left(c + \sum_{j=1}^m x_j z_j\right) \left(c + \sum_{\ell=1}^m x_\ell z_\ell\right) \\ &= c^2 + 2c \sum_{j=1}^m x_j z_j + \sum_{j=1}^m \sum_{\ell=1}^m x_j z_j x_\ell z_\ell \\ &= c^2 + \sum_{j=1}^m (\sqrt{2c} x_j)(\sqrt{2c} z_j) + \sum_{j,\ell=1}^m (x_j x_\ell)(z_j z_\ell). \end{aligned} \quad (5.61)$$

Define a **feature expansion** as

$$\phi([x_1, \dots, x_m]) = [x_1^2, x_1 x_2, \dots, x_m x_{m-1}, x_m^2, \sqrt{2c} x_1, \dots, \sqrt{2c} x_m, c], \quad (5.62)$$

which is in $\mathbb{R}^{m^2+m+1}$. Then $\phi(\mathbf{x}) \cdot \phi(\mathbf{z}) = \mathcal{K}(\mathbf{x}, \mathbf{z}) = (c + \mathbf{x} \cdot \mathbf{z})^2$. $\square$

Note: Kernel Functions

- Kernels may **not** be expressed as $\phi(\mathbf{x}) \cdot \phi(\mathbf{z})$.
- The mapping ϕ may transform $\mathbf{x}$ to **infinite dimensions**.

Summary 5.31. Linear Classifiers

- They are a simple and popular way to learn a classifier
- They suffer from inefficient use of data, overfitting, or lack of expressiveness
- **SVM**
 - It can fix these problems using ① **maximum margins** and ② **feature expansion** (mapping to a higher-dimension).
 - In order to make feature expansion **computationally feasible**, we need the ③ **kernel trick**, which avoids writing out high-dimensional feature vectors explicitly.

Remark 5.32. Kernel Trick

- There is no explicit feature expansion.
 - The kernel $\mathcal{K}(\mathbf{x}, \mathbf{z})$ must be formulated **meaningfully**.
-
- The **kernel function $\mathcal{K}$** must be considered as **a nonlinear measure for the data to become separable.**

5.3.6. Solving the dual problem with SMO

SMO (Sequential Minimal Optimization) is

- a type of coordinate ascent algorithm,
- but adapted to the SVM so that the solution always stays within the feasible region.

Recall: The **dual problem** of the soft-margin SVM, formulated in (5.51):

$$\begin{aligned} \max_{\alpha} \left[\sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right], \quad \text{subj.to} \\ \left[\begin{array}{l} 0 \leq \alpha_i \leq C, \quad \forall i, \\ \sum_{i=1}^N \alpha_i y^{(i)} = 0. \end{array} \right. \end{aligned} \quad (5.63)$$

Quesiton. Start with (5.63). Let's say you want to hold $\alpha_2, \dots, \alpha_N$ fixed and take a coordinate step in the first direction. That is, change α_1 to maximize the objective in (5.63). Can we make any progress? Can we get a better feasible solution by doing this?

Turns out, no. Let's see why. Look at the constraint in (5.63), $\sum_{i=1}^N \alpha_i y^{(i)} = 0$. This means

$$\alpha_1 y^{(1)} = - \sum_{i=2}^N \alpha_i y^{(i)} \quad \Rightarrow \quad \alpha_1 = -y^{(1)} \sum_{i=2}^N \alpha_i y^{(i)}.$$

So, since $\alpha_2, \dots, \alpha_N$ are fixed, α_1 is also fixed.

Thus, if we want to update any of the α_i 's, **we need to update at least 2 of them simultaneously** to keep the solution feasible (i.e., to keep the constraints satisfied).

- Start with a feasible vector α .
- Let's **update α_1 and α_2** , holding $\alpha_3, \dots, \alpha_N$ fixed.

Question: What values of α_1 and α_2 are we allowed to choose?

- The constraint is: $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = - \sum_{i=3}^N \alpha_i y^{(i)} =: \xi$.

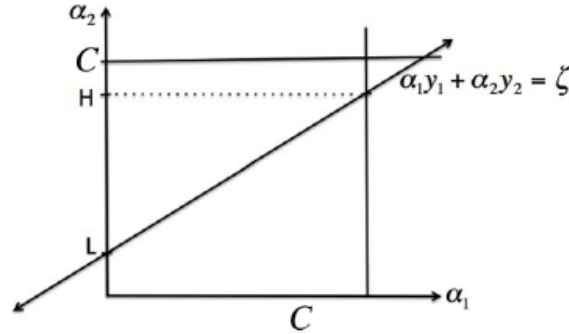


Figure 5.13

We are only allowed to choose α_1 and α_2 **on the line**.

- **When we pick α_2** , we can get α_1 from

$$\alpha_1 = \frac{1}{y^{(1)}} (\xi - \alpha_2 y^{(2)}) = y^{(1)} (\xi - \alpha_2 y^{(2)}). \quad (5.64)$$

- **Optimization for α_2 :** The other constraints in (5.63) says $0 \leq \alpha_1, \alpha_2 \leq C$. Thus, α_2 needs to be within $[L, H]$ on the figure ($\because \alpha_1 \in [0, C]$). To do the coordinate ascent step, we will optimize the objective over α_2 , keeping it within $[L, H]$. Using (5.64), (5.63) becomes

$$\max_{\alpha_2 \in [L, H]} \left[y^{(1)} (\xi - \alpha_2 y^{(2)}) + \alpha_2 + \sum_{i=3}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y^{(i)} y^{(j)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right], \quad (5.65)$$

of which the objective is quadratic in α_2 . This means we can just set its derivative to 0 to optimize it $\Rightarrow$ **get α_2** .

- After updating α_1 using (5.64), move to the next iteration of SMO.

Note: There are heuristics to choose the order of α_i 's chosen to update.

5.4. Decision Trees

Decision tree classifiers are attractive models if we care about **interpretability**. As the name decision tree suggests, we can think of this model as breaking down our data by making decision based on **asking a series of questions**. Decision tree was invented by a British researcher, William Belson, in 1959 [1].

Note: Decision trees are commonly used in **operations research**, specifically in **decision analysis**, to help identify a strategy most likely to reach a goal, but are also a popular tool in ML.

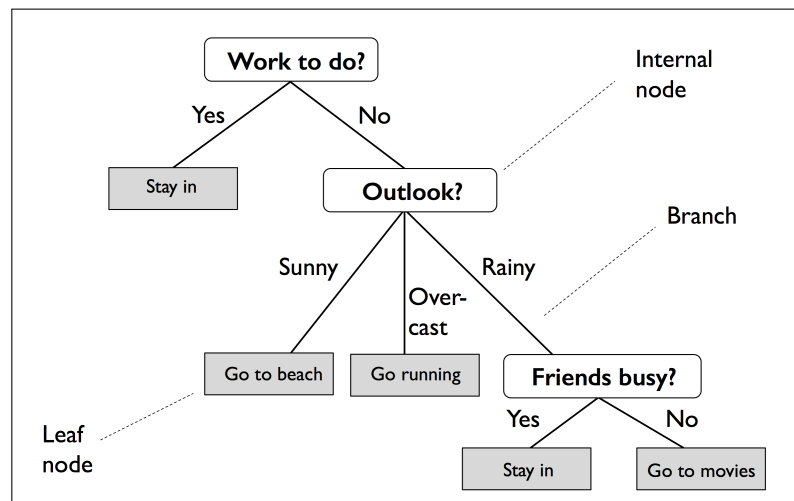


Figure 5.14: A decision tree to decide upon an activity on a particular day.

Key Idea 5.33. (Decision tree).

- **Start** at the tree root
- **Split** the data so as to result in the largest **Information Gain** (IG)
- **Repeat** the splitting at each child node until the leaves are pure
(This means the samples at each node all belong to the same class)
- **In practice**, this can result in a very **deep tree** with many nodes, which can easily lead to **overfitting**
(We typically set a limit for the maximal depth of the tree)

5.4.1. Decision tree objective

- Decision tree also needs to incorporate an **objective function**, to be optimized via the tree learning algorithm.
- Here, the objective function is to **maximize the information gain at each split**, which we define as follows:

$$IG(D_P, f) = I(D_P) - \sum_{j=1}^m \frac{N_j}{N_P} I(D_j), \quad (5.66)$$

where

- f : the feature to perform the split
- D_P : the parent dataset
- D_j : the dataset of the j -th child node
- I : the **impurity measure**
- N_P : the total number of samples at the parent node
- N_j : the number of samples in the j -th child node

- The **information gain** is simply the difference between the impurity of the parent node and the average of the child node impurities
 - The lower the impurity of the child nodes, the larger the information gain.
- However, for simplicity and to reduce the combinatorial search space, most libraries implement **binary decision trees**, where each parent node is split into two child nodes, D_L and D_R :

$$IG(D_P, f) = I(D_P) - \frac{N_L}{N_P} I(D_L) - \frac{N_R}{N_P} I(D_R). \quad (5.67)$$

Impurity measure?

Commonly used in binary decision trees:

- **Entropy**

$$I_H(t) = - \sum_{i=1}^c p(i|t) \log_2 p(i|t) \quad (5.68)$$

- **Gini impurity**

$$I_G(t) = \sum_{i=1}^c p(i|t) (1 - p(i|t)) = 1 - \sum_{i=1}^c p(i|t)^2 \quad (5.69)$$

- **Classification error**

$$I_E(t) = 1 - \max_i \{p(i|t)\} \quad (5.70)$$

where $p(i|t)$ denotes the proportion of the samples that belong to class i for a particular node t .

Mind simulation: When $c = 2$

- **Entropy**: It is maximal, if we have a uniform class distribution; it is 0, if all samples at the node t belong to the same class.

$$I_H(t) = 0, \text{ if } p(i = 1|t) = 1 \text{ or } p(i = 2|t) = 0$$

$$I_H(t) = 1, \text{ if } p(i = 1|t) = p(i = 2|t) = 0.5$$

⇒ We can say that the entropy criterion attempts to maximize the mutual information in the tree.

- **Gini impurity**: Intuitively, it can be understood as a criterion to minimize the probability of misclassification. The Gini impurity is maximal, if the classes are perfectly mixed.

$$I_G(t) = 1 - \sum_{i=1}^c 0.5^2 = 0.5$$

⇒ In practice, both Gini impurity and entropy yield very similar results.

- **Classification error**: It is less sensitive to changes in the class probabilities of the nodes.

⇒ The classification error is a useful criterion for pruning, but not recommended for growing a decision tree.

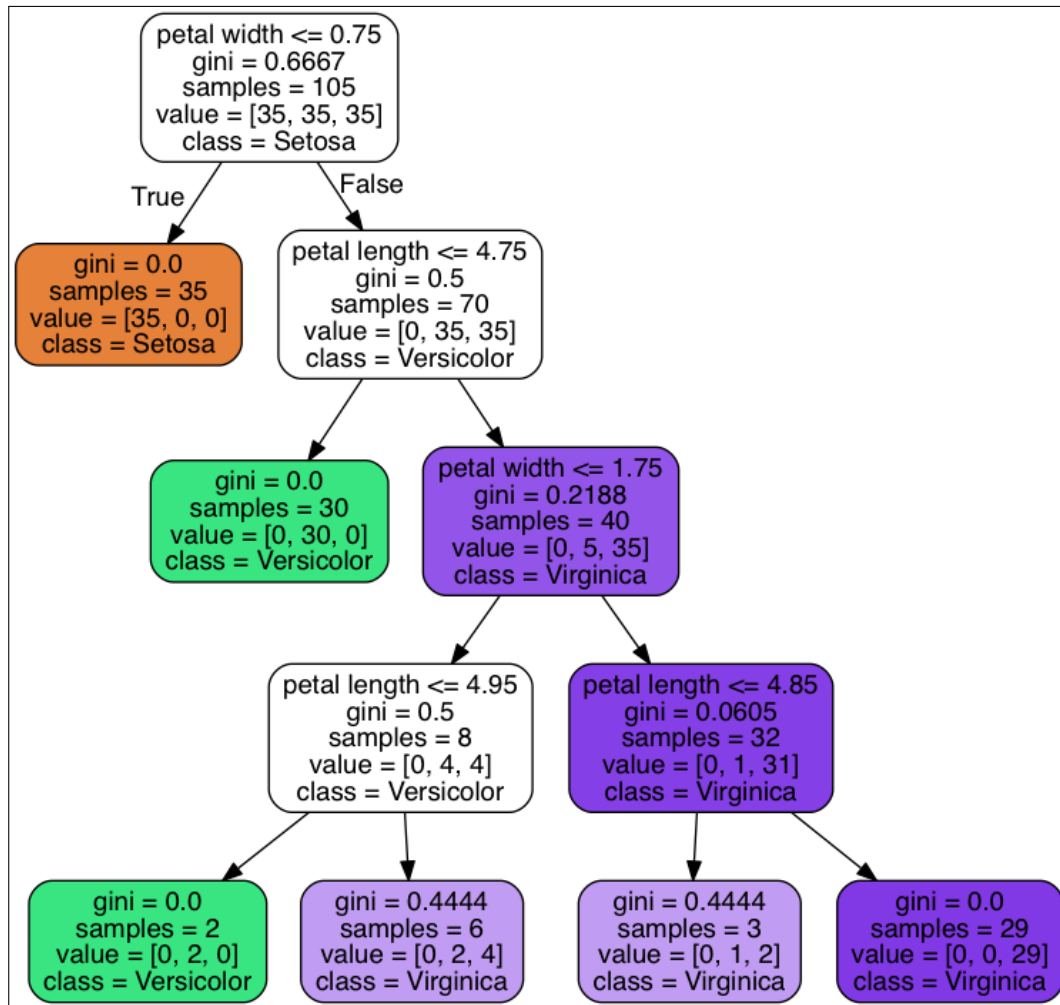


Figure 5.15: A decision tree result with Gini impurity measure, for three classes with two features (petal length, petal width). Page 99, *Python Machine Learning, 3rd Ed.*

Quesiton. How can the decision tree find questions such as

`petal width <= 0.75` `petal length <= 4.75` ... ?

Algorithm 5.34. (Decision tree split rule).

1. For each and every feature in D_P , $f_j^{(i)}$:

- make a question to split D_P into D_L and D_R
(e.g. $f_j^{(k)} \leq f_j^{(i)}$, for which k 's?)
- compute the impurities: $I(D_L)$ and $I(D_R)$
- compute the information gain:

$$IG(D_P, f_j^{(i)}) = I(D_P) - \frac{N_L}{N_P} I(D_L) - \frac{N_R}{N_P} I(D_R).$$

2. Let

$$f_q^{(p)} = \arg \max_{i,j} IG(D_P, f_j^{(i)}). \quad (5.71)$$

3. Then, the **best split** question (at the current node) is

$$f_q^{(k)} \leq f_q^{(p)}, \text{ for which } k\text{'s?} \quad (5.72)$$

The maximum in (5.71) often happens when one of the child impurities is zero or very small.

5.4.2. Random forests: Multiple decision trees

Random forests (or **random decision forests**) are an **ensemble learning** method for classification, regression, and other tasks that operates by constructing **multiple decision trees** at training time and outputting the class that is the **mode of the predicted classes** (classification) or **mean prediction** (regression) of the individual trees [32].

- Random forests have gained huge popularity in applications of ML **during the last decade** due to their good classification performance, scalability, and ease of use.
- The idea behind a random forest is **to average multiple (deep) decision trees** that individually suffer from high variance, to build a more robust model that has a better generalization performance and is less susceptible to overfitting.

Algorithm 5.35. Random Forest.

The algorithm can be summarized in four simple steps:

1. Draw a random **bootstrap sample** of size n (Randomly choose n samples from the training set *with replacement*).
2. Grow a decision tree from the bootstrap sample.
3. Repeat Steps 1-2 k times.
4. Aggregate the prediction by each tree to assign the class label by **majority vote**.

Note: In Step 2, when we are training the individual decision tree:

- instead of evaluating all features to determine the best split at each node,
- we can consider a random (without replacement) subset of those (of size d).

Remark 5.36. A big advantage of random forests is that

we don't have to worry so much about choosing good hyperparameter values.

- A smaller n increases randomness of the random forest; the bigger n is, the larger the degree of overfitting becomes.

Default $n = \text{size}(\text{the original training set})$, in most implementations

- Default $d = \sqrt{M}$, where M is the number of features in the training set
- The only parameter that we really need to care about in practice is

the number of trees k (Step 3).

Typically, the larger the number of trees, the better the performance of the random forest classifier at the expense of an increased computational cost.

5.5. k -Nearest Neighbors

The k -nearest neighbor (k -NN) classifier is a typical example of a **lazy learner**.

- It is called lazy not because of its apparent simplicity, but because it **doesn't learn a discriminative function** from the training data, but memorizes the training dataset instead.
- Analysis of the training data is **delayed until a query is made** to the system.

Algorithm 5.37. (k -NN algorithm). The algorithm itself is fairly straightforward and can be summarized by the following steps:

1. Choose the number k and a distance metric.
2. For the new sample, find the k -nearest neighbors.
3. Assign the class label by majority vote.

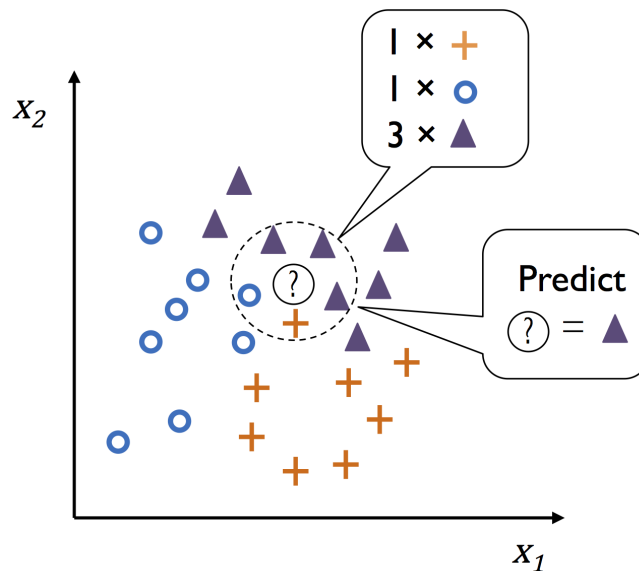


Figure 5.16: Illustration for how a new data point (?) is assigned the triangle class label, based on majority voting, when $k = 5$.

***k*-NN: pros and cons**

- Since it is memory-based, the classifier **immediately adapts** as we collect new training data.
- The **computational complexity** for classifying new samples grows linearly with the number of samples in the training dataset in the worst-case scenario.^a
- Furthermore, we can't discard training samples since *no training step* is involved. Thus, **storage space** can become a challenge if we are working with large datasets.

^aJ. H. Friedman, J. L. Bentley, and R.A. Finkel (1977). *An Algorithm for Finding Best Matches in Logarithmic Expected Time*, ACM transactions on Mathematical Software (TOMS), **3**, no. 3, pp. 209–226. The algorithm in the article is called the **KD-tree**.

***k*-NN: what to choose *k* and a distance metric?**

- The **right choice of *k* is crucial** to find a good balance between overfitting and underfitting.
(For `sklearn.neighbors.KNeighborsClassifier`, default `n_neighbors = 5`.)
- We also choose a distance metric that is appropriate for the features in the dataset. (e.g., the simple Euclidean distance, along with data standardization)
- Alternatively, we can choose the **Minkowski distance**:

$$d(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|_p \stackrel{\text{def}}{=} \left(\sum_{i=1}^m |x_i - z_i|^p \right)^{1/p}. \quad (5.73)$$

(For `sklearn.neighbors.KNeighborsClassifier`, default `p = 2`.)

Remark 5.38. The *k*-NN algorithm is very susceptible (wide open) to **overfitting** due to the **curse of dimensionality**.^a

Since regularization is not applicable for *k*-NN, we can use **feature selection** and **dimensionality reduction** techniques to help us avoid the curse of dimensionality and avoid overfitting. This will be discussed in more details later.

^aThe **curse of dimensionality** describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset. Intuitively, we can think of even the closest neighbors being too far away in a high-dimensional space to give a good estimate.

Exercises for Chapter 5

- 5.1. The equation $c_1x_1 + c_2x_2 + \cdots + c_nx_n = d$ determines a hyperplane in $\mathbb{R}^n$. Prove that the vector $[c_1, c_2, \dots, c_n]$ is a normal vector of the hyperplane.
- 5.2. For this problem, you would modify the code used for Problem 3.2 in Chapter 3. For the standardized data (X_{SD}) ,
 - (a) Apply the logistic regression gradient descent (Algorithm 5.9).
 - (b) Compare the results with that of Adaline descent gradient.
- 5.3. (*Continuation of Problem 5.2*). Perturb the standardized data (X_{SD}) by a random Gaussian noise G_σ of an observable σ (so as for $G_\sigma(X_{SD})$ not to be linearly separable).
 - (a) Apply the logistic regression gradient descent (Algorithm 5.9) for the noisy data $G_\sigma(X_{SD})$.
 - (b) Modify the code for the logistic regression with regularization (5.21) and apply the resulting algorithm for $G_\sigma(X_{SD})$.
 - (c) Compare their performances
- 5.4. (**Optional for Undergraduate Students**) Verify the formulation in (5.51), which is dual to the minimization of (5.50).
- 5.5. Experiment examples on pp. 84–91, *Python Machine Learning, 3rd Ed.*, in order to optimize the performance of kernel SVM by finding a best kernel and optimal hyperparameters (gamma and C).

Choose one of Exercises 6 and 7 below to implement and experiment. The experiment will guide you to understand how the LM software has been composed from scratch. You may use the example codes thankfully shared by Dr. Jason Brownlee, who is the founder of machinelearningmastery.com.

- 5.6. Implement a **decision tree** algorithm that incorporates the Gini impurity measure, from scratch, to run for the data used on page 96, *Python Machine Learning, 3rd Ed.*. Compare your results with the figure on page 97 of the book. You may refer to <https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/>
- 5.7. Implement a k -**NN** algorithm, from scratch, to run for the data used on page 106, *Python Machine Learning, 3rd Ed.*. Compare your results with the figure on page 103 of the book. You may refer to <https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/>

CHAPTER 6

Data Preprocessing in Machine Learning

Data preprocessing (or, **data preparation**) is a data mining technique, which is **the most time consuming** (*often, the most important*) step in machine learning.

Contents of Chapter 6

6.1. General Remarks on Data Preprocessing	134
6.2. Dealing with Missing Data & Categorical Data	136
6.3. Feature Scaling	138
6.4. Feature Selection	140
6.5. Feature Importance	146
Exercises for Chapter 6	148

6.1. General Remarks on Data Preprocessing

Data preprocessing is a data mining technique.

- It involves transforming raw data into a **understandable and more tractable** format.
- Real-world data is often **incomplete, redundant, inconsistent**, and/or **lacking in certain behaviors or trends**, and is likely to contain many errors.
- Data preprocessing is a proven method of resolving such issues.
- Often, data preprocessing is the **most important phase** of a machine learning project, especially in computational biology.

Summary 6.1. Different steps involved for data preprocessing can be summarized as follows.

1. **Data Cleaning:** In this first step, the primary focus is on handling missing data, noisy data, detection and removal of outliers, and minimizing duplication and computed biases within the data.
2. **Data Integration:** This process is used when data is gathered from various data sources and data are combined to form consistent data.
3. **Data Transformation:** This step is used to convert the raw data into a specified format according to the need of the model.
 - (a) *Normalization* – Numerical data is converted into the specified range (e.g., **feature scaling** $\rightarrow \sim \mathcal{N}(0, 1)$).
 - (b) *Aggregation* – This method combines some features into one.
4. **Data Reduction:** Redundancy within the data can be removed and efficiently organize the data.

The more disciplined you are in your handling of data, the more consistent and better results you are likely to achieve.

Remark 6.2. **Data preparation** is **difficult** because the process is *not objective*, and it is **important** because ML algorithms *learn from data*. Consider the following.

- Preparing data for analysis is one of the most **important** steps in any data-mining project – and traditionally, one of the most **time consuming**.
- Often, it takes up to 80% of the time.
- Data preparation is **not a once-off process**; that is, it is iterative as you understand the problem deeper on each successive pass.
- It is critical that you **feed the algorithms with the right data** for the problem you want to solve. Even if you have a good dataset, you need to make sure that it is in a useful scale and format and that meaningful features are included.

Questions in ML, in practice

- What would reduce the **generalization error**?
- What is the **best form of the data** to describe the problem? (It is difficult to answer, because it is not objective.)
- Can we design effective methods and/or smart algorithms for **automated data preparation**?

6.2. Dealing with Missing Data & Categorical Data

6.2.1. Handling missing data

Software: `[pandas.DataFrame].isnull().sum() > 1`

For missing values, three different steps can be executed.

- **Removal of samples (rows) or features (columns):**
 - It is **the simplest and efficient method** for handling the missing data.
 - However, we may end up removing too many samples or features.

Software: `pandas.dropna`

- **Filling the missing values manually:**
 - This is **one of the best-chosen methods**.
 - But there is one limitation that when there are large data set, and missing values are significant.
- **Imputing missing values using computed values:**
 - The missing values can also be occupied by computing **mean, median, or mode** of the observed given values.
 - Another method could be the predictive values that are computed by using any ML or Deep Learning algorithms.
 - But one drawback of this approach is that **it can generate bias** within the data as the calculated values are not accurate concerning the observed values.

Software: `from sklearn.preprocessing import Imputer`

6.2.2. Handling categorical data

It is common that real-world datasets contain one or more categorical feature columns. These categorical features must be effectively handled to fit in *numerical computing libraries*.

When we are talking about categorical data, we should further distinguish between **ordinal features** and **nominal features**.

- Mapping ordinal features: e.g.,

$$\text{size} : \begin{bmatrix} \text{M} \\ \text{L} \\ \text{XL} \end{bmatrix} \longleftrightarrow \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}. \quad (6.1)$$

- This is called an **ordinal encoding** or an **integer encoding**.
- The integer values have a natural ordered relationship between each other; machine learning algorithms may understand and harness this relationship.

- Encoding nominal features: **one-hot encoding**, e.g.,

$$\text{color} : \begin{bmatrix} \text{blue} \\ \text{green} \\ \text{red} \end{bmatrix} \longleftrightarrow \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \longleftrightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (6.2)$$

Software: `from sklearn.preprocessing import OneHotEncoder`

Remark 6.3. For categorical variables where no ordinal relationship exists, the integer encoding is not enough.

- In fact, assuming a natural ordering between categories and using the integer encoding may result in poor performance or unexpected results.
- The **one-hot encoding** can be used, although ordinal relationship exists.

6.3. Feature Scaling

Note: **Feature scaling** is a method used **to standardize the range of independent variables or features** of the data.

- It is one of **data normalization** methods^a in a broad sense.
- It is generally performed during the data preprocessing step.
- There are some scale-invariant algorithms such as decision trees and random forests.
- Most of other algorithms (we have learned) perform better with feature scaling.

^aIn a broad sense, **data normalization** is a process of reorganizing data, by cleaning and adjusting data values measured on different scales to a **notionally common scale**; its intention is to bring the entire probability distributions of adjusted values into alignment.

There are two common approaches to bring different features onto the same scale:

- **min-max scaling (normalization):**

$$x_{j,\text{norm}}^{(i)} = \frac{x_j^{(i)} - x_{j,\min}}{x_{j,\max} - x_{j,\min}} \in [0, 1], \quad (6.3)$$

where $x_{j,\min}$ and $x_{j,\max}$ are the minimum and maximum of the j -th feature column (in the training dataset), respectively.

- **standardization:**

$$x_{j,\text{std}}^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j}, \quad (6.4)$$

where μ_j is the sample mean of the j -th feature column and σ_j is the corresponding standard deviation.

- The standardized data has the **standard normal distribution**.

Remark 6.4. **Standardization** is more practical than the **min-max scaling** for many ML methods, especially for **optimization algorithms** such as the gradient descent method.

- **Reason ①:** For many linear models such as the logistic regression and the SVM, we can **easily initialize the weights** to 0 or small random values close to 0.
 - ⇐ Standardization possibly results in w^* *small*.
- **Reason ②:** It makes **regularization** perform more effectively; see Sections 5.2.3 and 6.4.3 for regularization.
 - ⇐ The minimizer of the penalty term is **0**.

6.4. Feature Selection

6.4.1. Selecting meaningful variables

Remark 6.5. Overfitting. If we observe that a model performs **much better on a training dataset** than on the test dataset, it is a strong indicator of **overfitting**.

- Overfitting means **the model fits the parameters too closely** with regard to the particular observations in **the training dataset**, but does not generalize well to new data. (The model has a high variance.)
- The reason for the overfitting is that our model is **too complex for the given training data**.

Common solutions to reduce the **generalization error** (via bias-variance tradeoff) are listed as follows:

- **Collect more training data** (often, not applicable)
- **Introduce regularization** (penalty for complexity)
- **Choose a simpler model** (fewer parameters)
- **Reduce the dimensionality** (feature selection)

Feature Selection (a.k.a. Variable Selection)

Its objective is four-fold:

- enhancing generalization by reducing overfitting/variance,
- providing faster and more cost-effective predictors,
- reducing training time, and
- providing a better understanding of the underlying process that generated the data.

Recall: Curse of Dimensionality. It describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset.

Methods for “automatic” feature selection

- **Filter methods:** Filter methods suppress the least interesting features, after assigning a scoring to **each feature** and ranking the features. The methods consider the feature *independently*, or with regard to the dependent variable.

Examples: Chi-squared test & correlation coefficient scores.

- **Wrapper methods:** Wrapper methods evaluate **subsets of features** which allows, unlike filter approaches, **to detect the possible interactions between features**. They prepare **various combinations of features**, to evaluate and compare with other combinations. The two main disadvantages of these methods are:

- Increasing overfitting risk, when the data size is not enough.
- Significant computation time, for a large number of variables.

Example: The recursive feature elimination algorithm

- **Embedded methods:** Embedded methods have been recently proposed that try **to combine the advantages of both previous methods**. They learn which features contribute the best to the accuracy of the model while the model is being created. The most common types of embedded feature selection methods are **regularization methods**.

Examples: ridge regression^a, LASSO^b, & elastic net regularization^c

^aThe **ridge regression** (a.k.a. **Tikhonov regularization**) is the most commonly used method of regularization of ill-posed problems. In machine learning, ridge regression is basically a regularized linear regression model: $\min_{\mathbf{w}} Q(X, \mathbf{y}; \mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$, in which the regularization parameter λ should be learned as well, using a method called cross validation. It is related to the Levenberg-Marquardt algorithm for non-linear least-squares problems.

^b**LASSO** (least absolute shrinkage and selection operator) is a regression analysis method that performs both variable selection and regularization in order to enhance the prediction accuracy and interpretability of the statistical model it produces. It includes an L^1 penalty term: $\min_{\mathbf{w}} Q(X, \mathbf{y}; \mathbf{w}) + \lambda \|\mathbf{w}\|_1$. It was originally developed in Geophysics [68, (Santosa-Symes, 1986)], and later independently rediscovered and popularized in 1996 by Robert Tibshirani [75], who coined the term and provided further insights into the observed performance.

^cThe **elastic net regularization** is a regularized regression method that linearly combines the L^1 and L^2 penalties of the LASSO and ridge methods, particularly in the fitting of linear or logistic regression models.

We will see how L^1 -regularization can reduce overfitting (serving as a feature selection method).

6.4.2. Sequential backward selection (SBS)

The idea behind the **sequential backward selection** (SBS) algorithm is quite simple:

- **The SBS sequentially removes features one-by-one** until the new feature subspace contains the desired number of features.
- In order to determine which feature is to be removed at each stage, we need to define the **criterion function** $\mathcal{C}$, e.g., performance of the classifier after the removal of a particular feature.
- Then, the feature to be removed at each stage can simply be defined as **the feature that maximizes this criterion**; or in more intuitive terms, at each stage we eliminate the feature that causes the **least performance loss after removal**.

Algorithm 6.6. Sequential Backward Selection

We can outline the algorithm in four simple steps:

1. Initialize the algorithm with $k = d$, where d is the dimensionality of the full feature space F_d .

2. Determine the feature $\hat{f}$ such that

$$\hat{f} = \arg \max_{f \in F_k} \mathcal{C}(F_k - f).$$

3. Remove the feature $\hat{f}$ from the feature set:

$$F_{k-1} = F_k - \hat{f}; \quad k = k - 1;$$

4. Terminate if k equals the number of desired features; otherwise, go to step 2.

6.4.3. Ridge regression vs. LASSO

Observation 6.7. When an L^p -penalty term is involved ($p = 1, 2$), the minimization problem can be written as follows:

$$\mathbf{w}^* = \arg \min_{\mathbf{w}} \mathcal{Q}(X, \mathbf{y}; \mathbf{w}) + \lambda \mathcal{R}_p(\mathbf{w}), \quad (6.5)$$

where

$$\mathcal{R}_p(\mathbf{w}) := \frac{1}{p} \|\mathbf{w}\|_p^p, \quad p = 1, 2. \quad (6.6)$$

- **Regularization** can be considered as adding a penalty term to the cost function to **encourage smaller weights**; or in other words, we penalize large weights.
- Thus, by **increasing the regularization strength** ($\lambda \uparrow$),
 - we can **shrink the weights** towards zero, and
 - **decrease the dependence** of our model on the training data.
- The **minimizer** $\mathbf{w}^*$ must be the point where the L^p -ball **intersects** with the minimum-valued contour of the unpenalized cost function.
 - The variable λ in (6.5) is a kind of **Lagrange multiplier**.

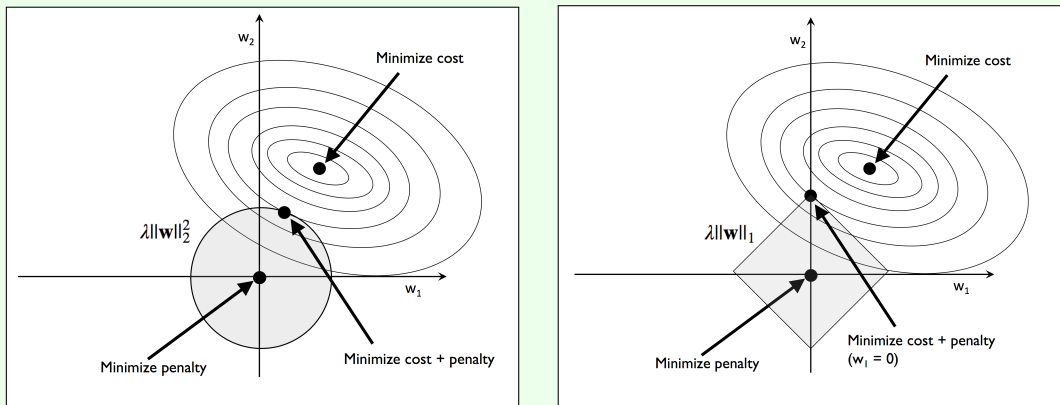


Figure 6.1: L^2 -regularization ($\|\mathbf{w}\|_2^2 = \sum_{i=1}^m w_i^2$) and L^1 -regularization ($\|\mathbf{w}\|_1 = \sum_{i=1}^m |w_i|$).

LASSO (L^1 -regularization). In the right figure, the L^1 -ball touches the minimum-valued contour of the cost function at $w_1 = 0$; the **optimum** is *more likely* located **on the axes**, which **encourages sparsity** (zero entries in w^*).

Remark 6.8. LASSO (L^1 -regularization)

- We can **enforce sparsity** (more zero entries) by increasing the regularization strength λ .
- A **sparse model** is a model where many of the weights are 0 or close to 0. Therefore L^1 -**regularization** is more suitable to create desired 0-weights, particularly for sparse models.

Remark 6.9. Regularization

In general, **regularization** can be understood as **adding bias** and preferring a **simpler model to reduce the variance (overfitting)**, *in the absence of sufficient training data, in particular.*

- L^1 -regularization **encourages sparsity**.
- We can **enforce sparsity** (more zero entries) by increasing the regularization strength λ .
- Thus it can **reduce overfitting**, serving as a **feature selection** method.
- L^1 -regularization may introduce **oscillation**, particularly when the regularization strength λ is large.
- A **post-processing operation** may be needed to take into account oscillatory behavior of L^1 -regularization.

Example 6.10. Consider a model consisting of the weights $\mathbf{w} = (w_1, \dots, w_m)^T$ and

$$\mathcal{R}_1(\mathbf{w}) = \sum_{i=1}^m |w_i|, \quad \mathcal{R}_2(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^m w_i^2. \quad (6.7)$$

Let us minimize $\mathcal{R}_p(\mathbf{w})$, $p = 1, 2$, using gradient descent.

Solution. The gradients read

$$\nabla_{\mathbf{w}} \mathcal{R}_1(\mathbf{w}) = \text{sign}(\mathbf{w}), \quad \nabla_{\mathbf{w}} \mathcal{R}_2(\mathbf{w}) = \mathbf{w}, \quad (6.8)$$

where

$$\text{sign}(w_i) = \begin{cases} 1, & \text{if } w_i > 0 \\ -1, & \text{if } w_i < 0 \\ 0, & \text{if } w_i = 0. \end{cases}$$

Thus the gradient descent becomes

$$\begin{aligned} \mathcal{R}_1 &: \mathbf{w}_{k+1} = \mathbf{w}_k - \lambda \text{sign}(\mathbf{w}_k), \\ \mathcal{R}_2 &: \mathbf{w}_{k+1} = \mathbf{w}_k - \lambda \mathbf{w}_k = (1 - \lambda) \mathbf{w}_k = (1 - \lambda)^{k+1} \mathbf{w}_0. \quad \square \end{aligned} \quad (6.9)$$

- **The L^2 -gradient** is linearly decreasing towards 0 as the weight goes towards 0. Thus **L^2 -regularization** will move any weight towards 0, but it will take smaller and smaller steps as a weight approaches 0. (The model never reaches a weight of 0.)
- In contrast, **L^1 -regularization** will move any weight towards 0 with the same step size λ , regardless the weight's value.
 - The iterates for minimizing $\mathcal{R}_1$ **may oscillate endlessly** near 0. (e.g., $w_0 = 0.2$ and $\lambda = 0.5$
 $\Rightarrow w_1 = -0.3 \Rightarrow w_2 = 0.2 \Rightarrow w_3 = -0.3 \Rightarrow w_4 = 0.2 \Rightarrow \dots$)
 - The oscillatory phenomenon may not be severe for real-world problems where $\mathcal{R}_1$ is used as a penalty term for a cost function.
 - However, we may need a **post-processing** to take account of oscillation, when λ is set large.

6.5. Feature Importance

The concept of **feature importance** is straightforward: it is *the increase in the model's prediction error after we permuted the feature's values*, which breaks the relationship between the feature and the true outcome.

- A feature is “important” if shuffling its values increases the **model error**, because in this case the model relied on the feature for the prediction.
- A feature is “unimportant” if shuffling its values leaves the model error unchanged, because in this case the model ignored the feature for the prediction.

- The **permutation feature importance** measurement was introduced by Breiman (2001) [8] for random forests.
- Based on this idea, Fisher, Rudin, and Dominici (2018) [19] proposed a **model-agnostic version** of the feature importance and called it **model reliance**.

Algorithm 6.11. Permutation feature importance (FI)

input: Trained model f , feature matrix X , target vector y ,
error measure $\mathcal{L}(f, X, y)$;

1. Estimate the original model error $\epsilon^{\text{orig}} = \mathcal{L}(f, X, y)$;
2. For each feature $j = 1, 2, \dots, d$; do:

Permute feature j in the data X to get $X^{(j)}$;
 Estimate error $\epsilon^{(j)} = \mathcal{L}(f, X^{(j)}, y)$;
 Calculate permutation FI: $FI^{(j)} = \epsilon^{(j)} / \epsilon^{\text{orig}}$ (or, $\epsilon^{(j)} - \epsilon^{\text{orig}}$);
3. Sort features by descending FI ;

Should we compute FI on training or test data?

To answer the question, you need to decide whether

- you want to know **how much the model relies on each feature** for making predictions (→ training data) or
- **how much the feature contributes** to the performance of the model on **unseen data** (→ test data).

There is no research addressing the question of training vs. test data; more research and more experience are needed to gain a better understanding.

Exercises for Chapter 6

First, read pp. 135-143, *Python Machine Learning, 3rd Ed.*.

- 6.1. On pp. 135-143, the **sequential backward selection** (SBS) is implemented as a feature selection method and experimented with a **k -NN classifier** (`n_neighbors=5`), using the wine dataset.
 - (a) Perform the same experiment with the k -NN classifier replaced by the **support vector machine** (soft-margin SVM classification).
 - (b) In particular, analyze **accuracy of the soft-margin SVM** and plot the result as in the figure on p. 139.
- 6.2. On pp. 141-143, the **permutation feature importance** is assessed from the **random forest** classifier, using the wine dataset.
 - (a) Discuss whether or not you can derive feature importance for a **k -NN classifier**.
 - (b) Assess feature importance with the **logistic regression** classifier, using the same dataset.
 - (c) Based on the computed feature importance, analyze and plot **accuracy of the logistic regression** classifier for `k_features = 1, 2, ..., 13`.

CHAPTER 7

Feature Extraction: Data Compression

There are *two main categories* of **dimensionality reduction** methods:

- **Feature selection**: Select a subset of the original features.
- **Feature extraction**: Construct a new feature subspace.

Feature Extraction

- It can be understood as an approach to **dimensionality reduction** and **data compression**.
 - with the goal of maintaining most of the relevant information
- In practice, **feature extraction** is used
 - to improve storage space or the computational efficiency
 - **to improve the predictive performance** by reducing the curse of dimensionality

In this chapter, we will study **three fundamental techniques** for dimensionality reduction:

- **Principal component analysis (PCA)**
- **Linear discriminant analysis (LDA)**, *maximizing class separability*
- **Kernel principal component analysis**, for nonlinear PCA

Contents of Chapter 7

7.1. Principal Component Analysis	150
7.2. Singular Value Decomposition	156
7.3. Linear Discriminant Analysis	172
7.4. Kernel Principal Component Analysis	189
Exercises for Chapter 7	196

7.1. Principal Component Analysis

- **Principal component analysis (PCA)** (a.k.a. **orthogonal linear transformation**) was invented in 1901 by K. Pearson [58], as an analogue of the principal axis theorem in mechanics; it was later independently developed and named by H. Hotelling in the 1930s [33, 34].
- The PCA is a statistical procedure that **uses an orthogonal transformation** to convert a set of observations (*of possibly correlated variables*) to **a set of linearly uncorrelated variables** called the **principal components**.
- The **orthogonal axes** of the new subspace can be interpreted as the **directions of maximum variance** given the constraint that the new feature axes are orthogonal to each other:

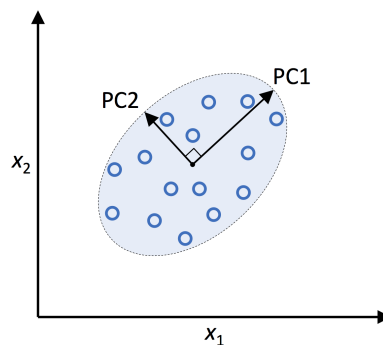


Figure 7.1: Principal components.

- As an **unsupervised^a** linear transformation technique, the PCA is widely used across **various fields** – in ML, most prominently for feature extraction and dimensionality reduction.
- The PCA identifies **patterns in data** based on the **correlation between features**.
- The PCA directions are **highly sensitive to data scaling**, and we **need to standardize the features** prior to PCA.

^aThe PCA is a unsupervised technique, because it does not use any class label information.

7.1.1. Computation of principal components

- Consider a **data matrix** $X \in \mathbb{R}^{N \times d}$:
 - each of the N rows represents a different data point,
 - each of the d columns gives a particular kind of feature, and
 - each column has zero empirical mean (e.g., after standardization).

- **The goal** is to find an **orthogonal weight matrix** $W \in \mathbb{R}^{d \times d}$ such that

$$Z = XW \quad (7.1)$$

maximizes the variance ($\Rightarrow$ minimizes the reconstruction error).

- Here $Z \in \mathbb{R}^{N \times d}$ is called the **score matrix**, of which columns represent **principal components** of X .

First weight vector w_1 : the first column of W :

In order to maximize variance of z_1 , the first weight vector w_1 should satisfy

$$\begin{aligned} w_1 &= \arg \max_{\|w\|=1} \|z_1\|^2 = \arg \max_{\|w\|=1} \|Xw\|^2 \\ &= \arg \max_{\|w\|=1} w^T X^T X w = \arg \max_{w \neq 0} \frac{w^T X^T X w}{w^T w}, \end{aligned} \quad (7.2)$$

where the quantity to be maximized can be recognized as a **Rayleigh quotient**.

Theorem 7.1. For a **positive semidefinite matrix** (such as $X^T X$), the maximum of the Rayleigh quotient is the same as the largest eigenvalue of the matrix, which occurs when w is the corresponding eigenvector, i.e.,

$$w_1 = \arg \max_{w \neq 0} \frac{w^T X^T X w}{w^T w} = \frac{v_1}{\|v_1\|}, \quad (X^T X)v_1 = \lambda_1 v_1, \quad (7.3)$$

where λ_1 is the largest eigenvalue of $X^T X \in \mathbb{R}^{d \times d}$.

Example 7.2. With w_1 found, the **first principal component** of a data vector $x^{(i)}$ can then be given as a score $z_1^{(i)} = x^{(i)} \cdot w_1$.

Further weight vectors $\mathbf{w}_k$:

The k -th weight vector can be found by ① subtracting the first $(k - 1)$ principal components from X :

$$\hat{X}_k := X - \sum_{i=1}^{k-1} X \mathbf{w}_i \mathbf{w}_i^T, \quad (7.4)$$

and then ② finding the weight vector which **extracts the maximum variance** from this new data matrix $\hat{X}_k$:

$$\mathbf{w}_k = \arg \max_{\|\mathbf{w}\|=1} \|\hat{X}_k \mathbf{w}\|^2. \quad (7.5)$$

Claim 7.3. The above turns out to give the (normalized) eigenvectors of $X^T X$. That is, the **transformation matrix** W is the stack of eigenvectors of $X^T X$:

$$W = [\mathbf{w}_1 | \mathbf{w}_2 | \cdots | \mathbf{w}_d], \quad (X^T X) \mathbf{w}_j = \lambda_j \mathbf{w}_j, \quad \mathbf{w}_i^T \mathbf{w}_j = \delta_{ij}, \quad (7.6)$$

where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$.

With W found, a **data vector** $\mathbf{x}$ is transformed to a d -dimensional row vector of principal components

$$\mathbf{z} = \mathbf{x}W, \quad (7.7)$$

of which components $z_j, j = 1, 2, \dots, d$, are decorrelated.

Remark 7.4. From **Singular Value Decomposition**:

While the weight matrix $W \in \mathbb{R}^{d \times d}$ is the collection of eigenvectors of $X^T X$, the score matrix $Z \in \mathbb{R}^{N \times d}$ is the stack of eigenvectors of XX^T , scaled by the square-root of eigenvalues:

$$Z = [\sqrt{\lambda_1} \mathbf{u}_1 | \sqrt{\lambda_2} \mathbf{u}_2 | \cdots | \sqrt{\lambda_d} \mathbf{u}_d], \quad (XX^T) \mathbf{u}_j = \lambda_j \mathbf{u}_j, \quad \mathbf{u}_i^T \mathbf{u}_j = \delta_{ij}. \quad (7.8)$$

See (7.14) and § 7.2.

7.1.2. Dimensionality reduction

The transformation $Z = XW$ maps data points in $\mathbb{R}^d$ to a new d -dimensional space of principal components. **Keeping only the first k principal components** ($k < d$) gives a **truncated transformation**:

$$Z_k = X W_k : \mathbf{x}^{(i)} \in \mathbb{R}^d \mapsto \mathbf{z}^{(i)} \in \mathbb{R}^k, \quad (7.9)$$

where $Z_k \in \mathbb{R}^{N \times k}$ and $W_k \in \mathbb{R}^{d \times k}$. Define the **truncated data** as

$$X_k := Z_k W_k^T = X W_k W_k^T. \quad (7.10)$$

Questions. How can we choose k ?
Is the difference $\|X - X_k\|$ small?

Remark 7.5. The principal components transformation can also be associated with the **singular value decomposition** (SVD) of X :

$$X = U \Sigma V^T, \quad (7.11)$$

where

- U : $n \times d$ orthogonal (the **left singular vectors** of X .)
- Σ : $d \times d$ diagonal (the **singular values** of X .)
- V : $d \times d$ orthogonal (the **right singular vectors** of X .)

- The matrix Σ explicitly reads

$$\Sigma = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_d), \quad (7.12)$$

where $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_d \geq 0$.

- In terms of this factorization, the matrix $X^T X$ reads

$$X^T X = (U \Sigma V^T)^T U \Sigma V^T = V \Sigma U^T U \Sigma V^T = V \Sigma^2 V^T. \quad (7.13)$$

- Comparing with the **eigenvector factorization** of $X^T X$, we have
 - the right singular vectors $V \cong$ the eigenvectors of $X^T X \Rightarrow V \cong W$
 - the square of singular values of X are equal to the eigenvalues of $X^T X$
 $\Rightarrow \sigma_j^2 = \lambda_j, j = 1, 2, \dots, d$.

Further considerations for the SVD

- Using the SVD, the **score matrix** Z reads

$$Z = XW = U \Sigma V^T W = U \Sigma, \quad (7.14)$$

and therefore each column of Z is given by one of the left singular vectors of X multiplied by the corresponding singular value. This form is also the **polar decomposition** of Z . See (7.8) on p. 152.

- As with the eigen-decomposition, the SVD, the **truncated score matrix** $Z_k \in \mathbb{R}^{N \times k}$ can be obtained by considering only the first k largest singular values and their singular vectors:

$$Z_k = XW_k = U \Sigma V^T W_k = U \Sigma_k, \quad (7.15)$$

where

$$\Sigma_k := \text{diag}(\sigma_1, \dots, \sigma_k, 0, \dots, 0). \quad (7.16)$$

- Now, using (7.15), the truncated data matrix reads

$$X_k = Z_k W_k^T = U \Sigma_k W_k^T = U \Sigma_k W^T = U \Sigma_k V^T. \quad (7.17)$$

Claim 7.6. It follows from (7.11) and (7.17) that

$$\begin{aligned} \|X - X_k\|_2 &= \|U \Sigma V^T - U \Sigma_k V^T\|_2 \\ &= \|U(\Sigma - \Sigma_k) V^T\|_2 \\ &= \|\Sigma - \Sigma_k\|_2 = \sigma_{k+1}, \end{aligned} \quad (7.18)$$

where $\|\cdot\|_2$ is the induced matrix L^2 -norm.

Remark 7.7. Efficient algorithms exist to calculate the SVD of X without having to form the matrix $X^T X$. **Computing the SVD is now the standard way to carry out the PCA.** See [27, 80].

7.1.3. Explained variance

Note: Since we want to reduce the dimensionality of our dataset by compressing it onto a new feature subspace, we only select the subset of the eigenvectors (principal components) that contains **most of the information (variance)**. **The eigenvalues define the magnitude of the eigenvectors**, so we have to sort the eigenvalues by decreasing magnitude; we are interested in the top k eigenvectors based on the values of their corresponding eigenvalues.

Definition 7.8. Let $\lambda_j (= \sigma_j^2)$ be eigenvalues of $X^T X$: $(X^T X)\mathbf{v}_j = \lambda_j \mathbf{v}_j$. Define the **explained variance ratio** of each eigenvalue as

$$evr(\lambda_i) = \frac{\lambda_i}{\sum_{j=1}^d \lambda_j}, \quad i = 1, 2, \dots, d, \quad (7.19)$$

and **cumulative explained variance** as

$$cev(\lambda_k) = \sum_{i=1}^k evr(\lambda_i) = \sum_{i=1}^k \lambda_i / \sum_{j=1}^d \lambda_j, \quad k = 1, 2, \dots, d. \quad (7.20)$$

Then, we may choose k satisfying

$$cev(\lambda_{k-1}) < \varepsilon \quad \text{and} \quad cev(\lambda_k) \geq \varepsilon, \quad (7.21)$$

for a tolerance ε . (**The smallest k such that $cev(\lambda_k) \geq \varepsilon$.**)

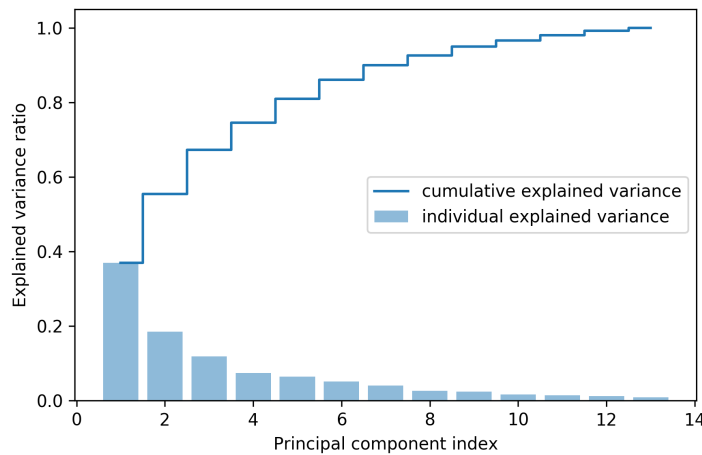


Figure 7.2: evr and cev for the wine dataset.

7.2. Singular Value Decomposition

Here we will deal with the SVD in detail.

Theorem 7.9. (SVD Theorem). *Let $A \in \mathbb{R}^{m \times n}$ with $m \geq n$. Then we can write*

$$A = U \Sigma V^T, \quad (7.22)$$

where $U \in \mathbb{R}^{m \times n}$ and satisfies $U^T U = I$, $V \in \mathbb{R}^{n \times n}$ and satisfies $V^T V = I$, and $\Sigma = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_n)$, where

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n \geq 0.$$

Remark 7.10. The matrices are illustrated pictorially as

$$\begin{bmatrix} A \end{bmatrix} = \begin{bmatrix} U \end{bmatrix} \begin{bmatrix} \Sigma \end{bmatrix} \begin{bmatrix} V^T \end{bmatrix}, \quad (7.23)$$

where

U : $m \times n$ orthogonal (the **left singular vectors** of A .)

Σ : $n \times n$ diagonal (the **singular values** of A .)

V : $n \times n$ orthogonal (the **right singular vectors** of A .)

- For some $r \leq n$, the singular values may satisfy

$$\underbrace{\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r}_{\text{nonzero singular values}} > \sigma_{r+1} = \dots = \sigma_n = 0. \quad (7.24)$$

In this case, $\text{rank}(A) = r$.

- If $m < n$, the **SVD** is defined by considering A^T .

Proof. (of Theorem 7.9) Use induction on m and n : we assume that the SVD exists for $(m-1) \times (n-1)$ matrices, and prove it for $m \times n$. We assume $A \neq 0$; otherwise we can take $\Sigma = 0$ and let U and V be arbitrary orthogonal matrices.

- The basic step occurs when $n = 1$ ($m \geq n$). We let $A = U\Sigma V^T$ with $U = A/\|A\|_2$, $\Sigma = \|A\|_2$, $V = 1$.
- For the induction step, choose $\mathbf{v}$ so that

$$\|\mathbf{v}\|_2 = 1 \quad \text{and} \quad \|A\|_2 = \|A\mathbf{v}\|_2 > 0.$$

- Let $\mathbf{u} = \frac{A\mathbf{v}}{\|A\mathbf{v}\|_2}$, which is a unit vector. Choose $\tilde{U}, \tilde{V}$ such that

$$U = [\mathbf{u} \quad \tilde{U}] \in \mathbb{R}^{m \times n} \quad \text{and} \quad V = [\mathbf{v} \quad \tilde{V}] \in \mathbb{R}^{n \times n}$$

are orthogonal.

- Now, we write

$$U^T A V = \begin{bmatrix} \mathbf{u}^T \\ \tilde{U}^T \end{bmatrix} \cdot A \cdot [\mathbf{v} \quad \tilde{V}] = \begin{bmatrix} \mathbf{u}^T A \mathbf{v} & \mathbf{u}^T A \tilde{V} \\ \tilde{U}^T A \mathbf{v} & \tilde{U}^T A \tilde{V} \end{bmatrix}$$

Since

$$\begin{aligned} \mathbf{u}^T A \mathbf{v} &= \frac{(A\mathbf{v})^T (A\mathbf{v})}{\|A\mathbf{v}\|_2} = \frac{\|A\mathbf{v}\|_2^2}{\|A\mathbf{v}\|_2} = \|A\mathbf{v}\|_2 = \|A\|_2 \equiv \sigma, \\ \tilde{U}^T A \mathbf{v} &= \tilde{U}^T \mathbf{u} \|A\mathbf{v}\|_2 = 0, \end{aligned}$$

we have

$$U^T A V = \begin{bmatrix} \sigma & 0 \\ 0 & U_1 \Sigma_1 V_1^T \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & U_1 \end{bmatrix} \begin{bmatrix} \sigma & 0 \\ 0 & \Sigma_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & V_1 \end{bmatrix}^T,$$

or equivalently

$$A = \left(U \begin{bmatrix} 1 & 0 \\ 0 & U_1 \end{bmatrix} \right) \begin{bmatrix} \sigma & 0 \\ 0 & \Sigma_1 \end{bmatrix} \left(V \begin{bmatrix} 1 & 0 \\ 0 & V_1 \end{bmatrix} \right)^T. \quad (7.25)$$

Equation (7.25) is our desired decomposition. $\square$

7.2.1. Interpretation of the SVD

Algebraic interpretation of the SVD

Let $\text{rank}(A) = r$. let the SVD of A be $A = U \Sigma V^T$, with

$$\begin{aligned} U &= [\mathbf{u}_1 \quad \mathbf{u}_2 \quad \cdots \quad \mathbf{u}_n], \\ \Sigma &= \text{diag}(\sigma_1, \sigma_2, \cdots, \sigma_n), \\ V &= [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_n], \end{aligned}$$

and σ_r be the **smallest** positive singular value. Since

$$A = U \Sigma V^T \iff AV = U \Sigma V^T V = U \Sigma,$$

we have

$$\begin{aligned} AV &= A[\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_n] = [A\mathbf{v}_1 \quad A\mathbf{v}_2 \quad \cdots \quad A\mathbf{v}_n] \\ &= [\mathbf{u}_1 \quad \cdots \quad \mathbf{u}_r \quad \cdots \quad \mathbf{u}_n] \begin{bmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_r & & \\ & & & \ddots & \\ & & & & 0 \end{bmatrix} \\ &= [\sigma_1 \mathbf{u}_1 \quad \cdots \quad \sigma_r \mathbf{u}_r \quad 0 \quad \cdots \quad 0]. \end{aligned} \tag{7.26}$$

Therefore,

$$A = U \Sigma V^T \iff \begin{cases} A\mathbf{v}_j = \sigma_j \mathbf{u}_j, & j = 1, 2, \cdots, r \\ A\mathbf{v}_j = \mathbf{0}, & j = r + 1, \cdots, n \end{cases} \tag{7.27}$$

Similarly, starting from $A^T = V \Sigma U^T$,

$$A^T = V \Sigma U^T \iff \begin{cases} A^T \mathbf{u}_j = \sigma_j \mathbf{v}_j, & j = 1, 2, \cdots, r \\ A^T \mathbf{u}_j = \mathbf{0}, & j = r + 1, \cdots, n \end{cases} \tag{7.28}$$

Summary 7.11. It follows from (7.27) and (7.28) that

- $(\mathbf{v}_j, \sigma_j^2)$, $j = 1, 2, \dots, r$, are eigenvector-eigenvalue pairs of $A^T A$.

$$A^T A \mathbf{v}_j = A^T (\sigma_j \mathbf{u}_j) = \sigma_j^2 \mathbf{v}_j, \quad j = 1, 2, \dots, r. \quad (7.29)$$

So, the singular values play the role of eigenvalues.

- Similarly, we have

$$A A^T \mathbf{u}_j = A (\sigma_j \mathbf{v}_j) = \sigma_j^2 \mathbf{u}_j, \quad j = 1, 2, \dots, r. \quad (7.30)$$

- Equation (7.29) gives how to find the **singular values** $\{\sigma_j\}$ and the **right singular vectors** V , while (7.27) shows a way to compute the **left singular vectors** U .
- **(Dyadic decomposition)** The matrix $A \in \mathbb{R}^{m \times n}$ can be expressed as

$$A = \sum_{j=1}^n \sigma_j \mathbf{u}_j \mathbf{v}_j^T. \quad (7.31)$$

When $\text{rank}(A) = r \leq n$,

$$A = \sum_{j=1}^r \sigma_j \mathbf{u}_j \mathbf{v}_j^T. \quad (7.32)$$

This property has been utilized for various approximations and applications, e.g., by dropping singular vectors corresponding to *small* singular values.

Geometric interpretation of the SVD

The matrix A maps an **orthonormal basis**

$$\mathcal{B}_1 = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$$

of $\mathbb{R}^n$ onto a new “scaled” **orthogonal basis**

$$\mathcal{B}_2 = \{\sigma_1 \mathbf{u}_1, \sigma_2 \mathbf{u}_2, \dots, \sigma_r \mathbf{u}_r\}$$

for a subspace of $\mathbb{R}^m$:

$$\mathcal{B}_1 = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\} \xrightarrow{A} \mathcal{B}_2 = \{\sigma_1 \mathbf{u}_1, \sigma_2 \mathbf{u}_2, \dots, \sigma_r \mathbf{u}_r\} \quad (7.33)$$

Consider a unit sphere $\mathcal{S}^{n-1}$ in $\mathbb{R}^n$:

$$\mathcal{S}^{n-1} = \left\{ \mathbf{x} \mid \sum_{j=1}^n x_j^2 = 1 \right\}.$$

Then, $\forall \mathbf{x} \in \mathcal{S}^{n-1}$,

$$\begin{aligned} \mathbf{x} &= x_1 \mathbf{v}_1 + x_2 \mathbf{v}_2 + \dots + x_n \mathbf{v}_n \\ A\mathbf{x} &= \sigma_1 x_1 \mathbf{u}_1 + \sigma_2 x_2 \mathbf{u}_2 + \dots + \sigma_r x_r \mathbf{u}_r \\ &= y_1 \mathbf{u}_1 + y_2 \mathbf{u}_2 + \dots + y_r \mathbf{u}_r, \quad (y_j = \sigma_j x_j) \end{aligned} \quad (7.34)$$

So, we have

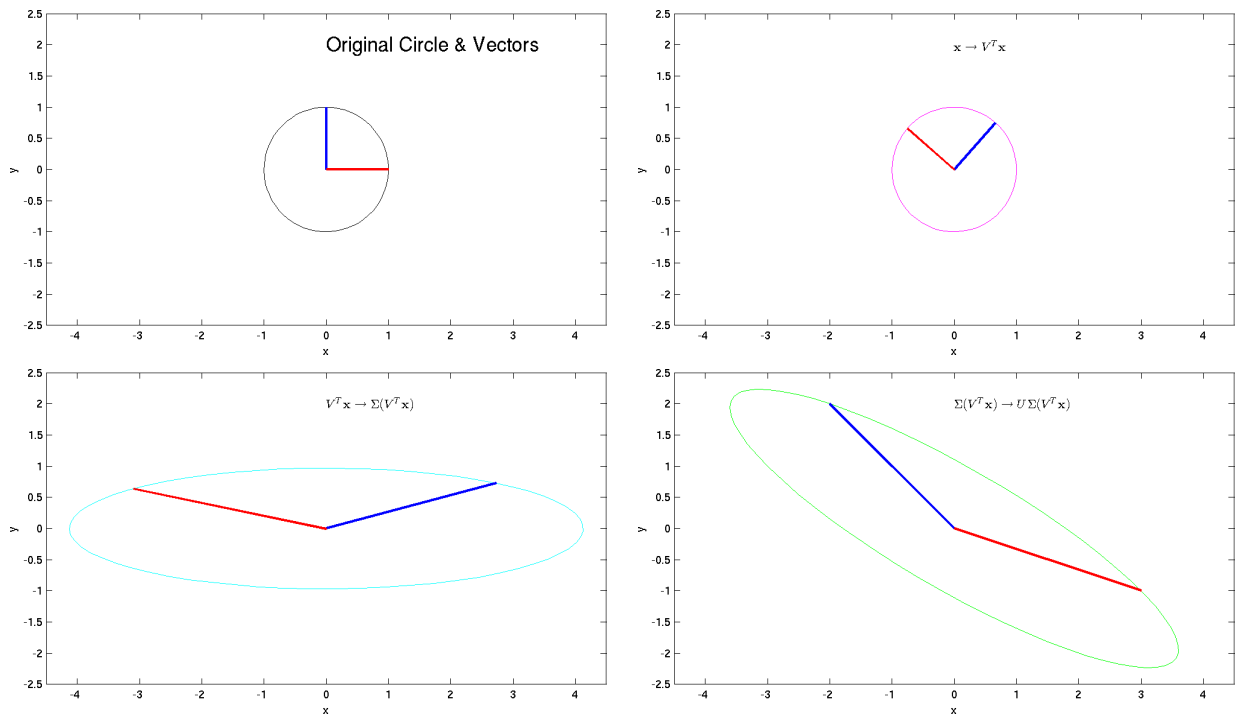
$$\begin{aligned} y_j = \sigma_j x_j &\iff x_j = \frac{y_j}{\sigma_j} \\ \sum_{j=1}^n x_j^2 = 1 \text{ (sphere)} &\iff \sum_{j=1}^r \frac{y_j^2}{\sigma_j^2} = \alpha \leq 1 \text{ (ellipsoid)} \end{aligned} \quad (7.35)$$

Example 7.12. We build the set $A(S^{n-1})$ by multiplying one factor of $A = U\Sigma V^T$ at a time. Assume for simplicity that $A \in \mathbb{R}^{2 \times 2}$ and nonsingular. Let

$$\begin{aligned} A &= \begin{bmatrix} 3 & -2 \\ -1 & 2 \end{bmatrix} = U\Sigma V^T \\ &= \begin{bmatrix} -0.8649 & 0.5019 \\ 0.5019 & 0.8649 \end{bmatrix} \begin{bmatrix} 4.1306 & 0 \\ 0 & 0.9684 \end{bmatrix} \begin{bmatrix} -0.7497 & 0.6618 \\ 0.6618 & 0.7497 \end{bmatrix} \end{aligned}$$

Then, for $\mathbf{x} \in S^1$,

$$A\mathbf{x} = U\Sigma V^T \mathbf{x} = U(\Sigma(V^T \mathbf{x}))$$



In general,

- $V^T : S^{n-1} \rightarrow S^{n-1}$ (rotation in $\mathbb{R}^n$)
- $\Sigma : \mathbf{e}_j \mapsto \sigma_j \mathbf{e}_j$ (scaling from S^{n-1} to $\mathbb{R}^n$)
- $U : \mathbb{R}^n \rightarrow \mathbb{R}^m$ (rotation)

7.2.2. Properties of the SVD

Theorem 7.13. Let $A \in \mathbb{R}^{m \times n}$ with $m \geq n$. Let $A = U\Sigma V^T$ be the SVD of A , with

$$\sigma_1 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_n = 0.$$

Then,

$$\begin{cases} \text{rank}(A) &= r \\ \text{Null}(A) &= \text{Span}\{\mathbf{v}_{r+1}, \dots, \mathbf{v}_n\} \\ \text{Range}(A) &= \text{Span}\{\mathbf{u}_1, \dots, \mathbf{u}_r\} \\ A &= \sum_{i=1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^T \end{cases} \quad (7.36)$$

and

$$\begin{cases} \|A\|_2 &= \sigma_1 \quad (\text{See Exercise 2.}) \\ \|A\|_F^2 &= \sigma_1^2 + \cdots + \sigma_r^2 \quad (\text{See Exercise 3.}) \\ \min_{\mathbf{x} \neq 0} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} &= \sigma_n \quad (m \geq n) \\ \kappa_2(A) &= \|A\|_2 \cdot \|A^{-1}\|_2 = \frac{\sigma_1}{\sigma_n} \\ &\quad (\text{when } m = n, \& \exists A^{-1}) \end{cases} \quad (7.37)$$

Theorem 7.14. Let $A \in \mathbb{R}^{m \times n}$, $m \geq n$, $\text{rank}(A) = n$, with singular values

$$\sigma_1 \geq \sigma_2 \geq \cdots \sigma_n > 0.$$

Then

$$\begin{aligned} \|(A^T A)^{-1}\|_2 &= \sigma_n^{-2}, \\ \|(A^T A)^{-1} A^T\|_2 &= \sigma_n^{-1}, \\ \|A(A^T A)^{-1}\|_2 &= \sigma_n^{-1}, \\ \|A(A^T A)^{-1} A^T\|_2 &= 1. \end{aligned} \quad (7.38)$$

Definition 7.15. $(A^T A)^{-1} A^T$ is called the **pseudoinverse** of A , while $A(A^T A)^{-1}$ is called the **pseudoinverse** of A^T . Let $A = U\Sigma V^T$ be the SVD of A . Then

$$(A^T A)^{-1} A^T = V\Sigma^{-1}U^T \stackrel{\text{def}}{=} A^+. \quad (7.39)$$

Theorem 7.16. Let $A \in \mathbb{R}^{m \times n}$ with $\text{rank}(A) = r > 0$. Let $A = U\Sigma V^T$ be the SVD of A , with singular values

$$\sigma_1 \geq \cdots \geq \sigma_r > 0.$$

Define, for $k = 1, \dots, r-1$,

$$A_k = \sum_{j=1}^k \sigma_j \mathbf{u}_j \mathbf{v}_j^T \quad (\text{sum of rank-1 matrices}).$$

Then, $\text{rank}(A_k) = k$ and

$$\begin{aligned} \|A - A_k\|_2 &= \min\{\|A - B\|_2 \mid \text{rank}(B) \leq k\} \\ &= \sigma_{k+1}, \\ \|A - A_k\|_F^2 &= \min\{\|A - B\|_F^2 \mid \text{rank}(B) \leq k\} \\ &= \sigma_{k+1}^2 + \cdots + \sigma_r^2. \end{aligned} \tag{7.40}$$

That is, of all matrices of rank $\leq k$, A_k is closest to A .

Note: The matrix A_k can be written as

$$A_k = U \Sigma_k V^T, \tag{7.41}$$

where $\Sigma_k = \text{diag}(\sigma_1, \dots, \sigma_k, 0, \dots, 0)$. The **pseudoinverse of A_k** reads

$$A_k^+ = V \Sigma_k^+ U^T, \tag{7.42}$$

where

$$\Sigma_k^+ = \text{diag}(1/\sigma_1, 1/\sigma_2, \dots, 1/\sigma_k, 0, \dots, 0). \tag{7.43}$$

Corollary 7.17. Suppose $A \in \mathbb{R}^{m \times n}$ has full rank; $\text{rank}(A) = n$. Let $\sigma_1 \geq \cdots \geq \sigma_n$ be the singular values of A . Let $B \in \mathbb{R}^{m \times n}$ satisfy

$$\|A - B\|_2 < \sigma_n.$$

Then B also has full rank.

Full SVD

- For $A \in \mathbb{R}^{m \times n}$,

$$A = U\Sigma V^T \iff U^T A V = \Sigma,$$

where $U \in \mathbb{R}^{m \times n}$ and $\Sigma, V \in \mathbb{R}^{n \times n}$.

- Expand

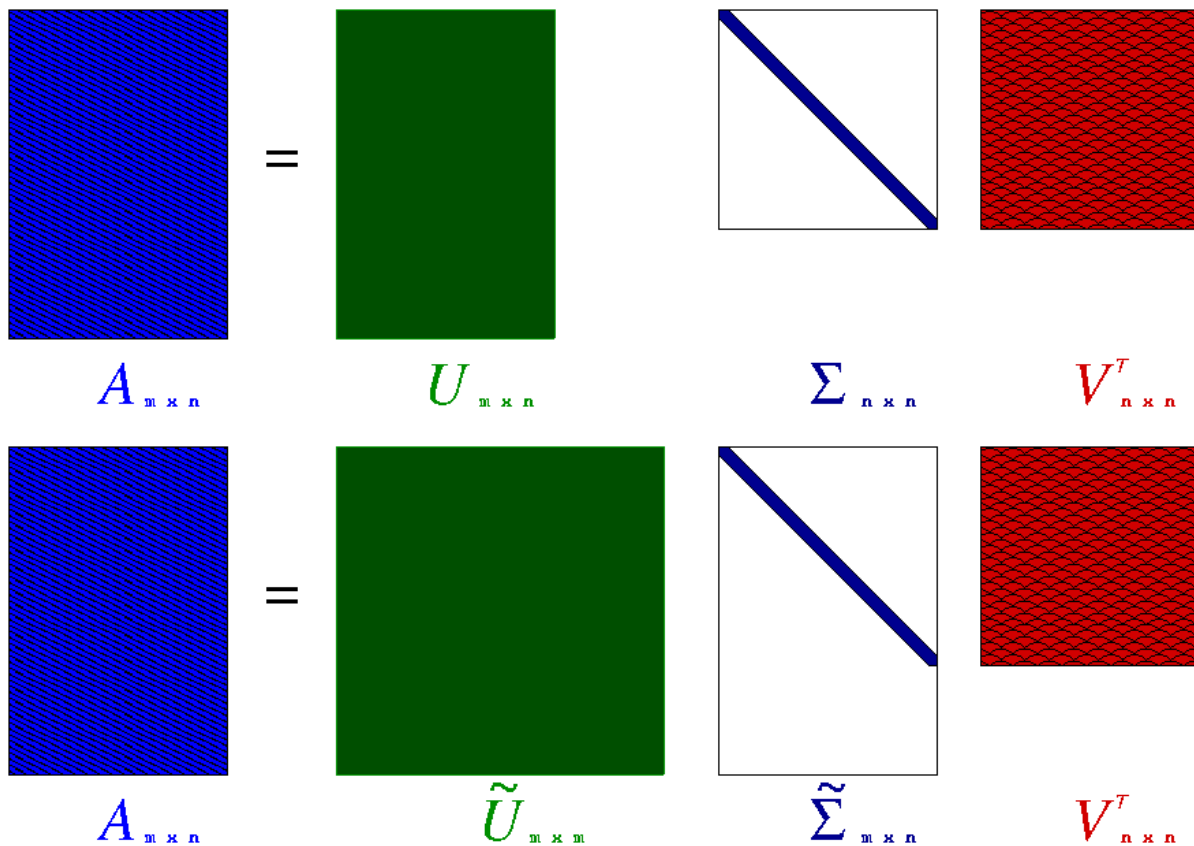
$$U \rightarrow \tilde{U} = [U \ U_2] \in \mathbb{R}^{m \times m}, \quad (\text{orthogonal})$$

$$\Sigma \rightarrow \tilde{\Sigma} = \begin{bmatrix} \Sigma \\ O \end{bmatrix} \in \mathbb{R}^{m \times n},$$

where O is an $(m - n) \times n$ zero matrix.

- Then,

$$\tilde{U} \tilde{\Sigma} V^T = [U \ U_2] \begin{bmatrix} \Sigma \\ O \end{bmatrix} V^T = U \Sigma V^T = A \quad (7.44)$$



7.2.3. Computation of the SVD

For $A \in \mathbb{R}^{m \times n}$, the procedure is as follows.

1. Form $A^T A$ ($A^T A$ – **covariance matrix** of A).
2. Find the eigen-decomposition of $A^T A$ by orthogonalization process, i.e., $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$,

$$A^T A = V \Lambda V^T,$$

where $V = [\mathbf{v}_1 \ \cdots \ \mathbf{v}_n]$ is orthogonal, i.e., $V^T V = I$.

3. Sort the eigenvalues according to their magnitude and let

$$\sigma_j = \sqrt{\lambda_j}, \quad j = 1, 2, \dots, n.$$

4. Form the U matrix as follows,

$$\mathbf{u}_j = \frac{1}{\sigma_j} A \mathbf{v}_j, \quad j = 1, 2, \dots, r.$$

If necessary, pick up the remaining columns of U so it is orthogonal. (These additional columns must be in $\text{Null}(AA^T)$.)

$$5. A = U \Sigma V^T = [\mathbf{u}_1 \ \cdots \ \mathbf{u}_r \ \cdots \ \mathbf{u}_n] \text{diag}(\sigma_1, \dots, \sigma_r, 0, \dots, 0) \begin{bmatrix} \mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_n^T \end{bmatrix}$$

Lemma 7.18. *Let $A \in \mathbb{R}^{n \times n}$ be symmetric. Then (a) all the eigenvalues of A are real and (b) eigenvectors corresponding to distinct eigenvalues are orthogonal.*

Proof. See Exercise 4. $\square$

Example 7.19. Find the SVD for $A = \begin{bmatrix} 1 & 2 \\ -2 & 1 \\ 3 & 2 \end{bmatrix}$.

Solution.

$$1. A^T A = \begin{bmatrix} 14 & 6 \\ 6 & 9 \end{bmatrix}.$$

2. Solving $\det(A^T A - \lambda I) = 0$ gives the eigenvalues of $A^T A$

$$\lambda_1 = 18 \text{ and } \lambda_2 = 5,$$

of which corresponding eigenvectors are

$$\tilde{\mathbf{v}}_1 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}, \tilde{\mathbf{v}}_2 = \begin{bmatrix} -2 \\ 3 \end{bmatrix} \implies V = \begin{bmatrix} \frac{3}{\sqrt{13}} & -\frac{2}{\sqrt{13}} \\ \frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix}$$

3. $\sigma_1 = \sqrt{\lambda_1} = \sqrt{18} = 3\sqrt{2}$, $\sigma_2 = \sqrt{\lambda_2} = \sqrt{5}$. So

$$\Sigma = \begin{bmatrix} \sqrt{18} & 0 \\ 0 & \sqrt{5} \end{bmatrix}$$

$$4. \mathbf{u}_1 = \frac{1}{\sigma_1} A \mathbf{v}_1 = \frac{1}{\sqrt{18}} A \begin{bmatrix} \frac{3}{\sqrt{13}} \\ \frac{2}{\sqrt{13}} \end{bmatrix} = \frac{1}{\sqrt{18}} \frac{1}{\sqrt{13}} \begin{bmatrix} 7 \\ -4 \\ 13 \end{bmatrix} = \begin{bmatrix} \frac{7}{\sqrt{234}} \\ -\frac{4}{\sqrt{234}} \\ \frac{13}{\sqrt{234}} \end{bmatrix}$$

$$\mathbf{u}_2 = \frac{1}{\sigma_2} A \mathbf{v}_2 = \frac{1}{\sqrt{5}} A \begin{bmatrix} \frac{-2}{\sqrt{13}} \\ \frac{3}{\sqrt{13}} \end{bmatrix} = \frac{1}{\sqrt{5}} \frac{1}{\sqrt{13}} \begin{bmatrix} 4 \\ 7 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{4}{\sqrt{65}} \\ \frac{7}{\sqrt{65}} \\ 0 \end{bmatrix}.$$

$$5. A = U \Sigma V^T = \begin{bmatrix} \frac{7}{\sqrt{234}} & \frac{4}{\sqrt{65}} \\ -\frac{4}{\sqrt{234}} & \frac{7}{\sqrt{65}} \\ \frac{13}{\sqrt{234}} & 0 \end{bmatrix} \begin{bmatrix} \sqrt{18} & 0 \\ 0 & \sqrt{5} \end{bmatrix} \begin{bmatrix} \frac{3}{\sqrt{13}} & \frac{2}{\sqrt{13}} \\ -\frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix}$$

Example 7.20. Find the **pseudoinverse** of A ,

$$A^+ = (A^T A)^{-1} A^T = V \Sigma^{-1} U^T,$$

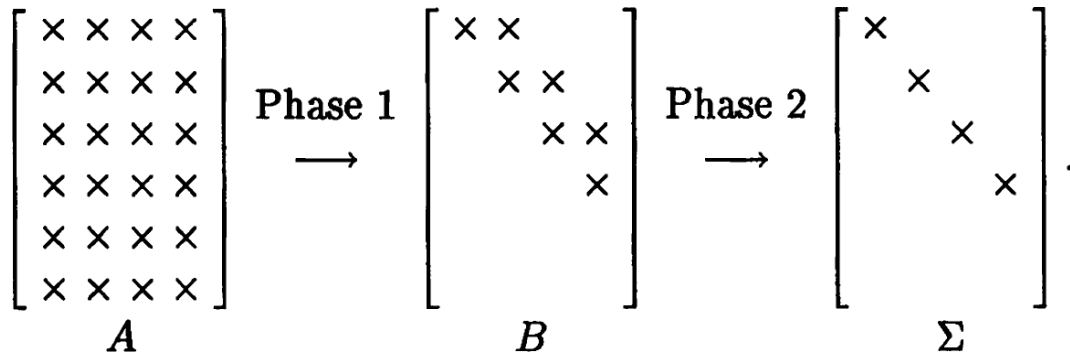
when $A = \begin{bmatrix} 1 & 2 \\ -2 & 1 \\ 3 & 2 \end{bmatrix}.$

Solution. From Example 7.19, we have

$$A = U \Sigma V^T = \begin{bmatrix} \frac{7}{\sqrt{234}} & \frac{4}{\sqrt{65}} \\ -\frac{4}{\sqrt{234}} & \frac{7}{\sqrt{65}} \\ \frac{13}{\sqrt{234}} & 0 \end{bmatrix} \begin{bmatrix} \sqrt{18} & 0 \\ 0 & \sqrt{5} \end{bmatrix} \begin{bmatrix} \frac{3}{\sqrt{13}} & \frac{2}{\sqrt{13}} \\ -\frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix}$$

Thus,

$$\begin{aligned} A^+ &= V \Sigma^{-1} U^T = \begin{bmatrix} \frac{3}{\sqrt{13}} & -\frac{2}{\sqrt{13}} \\ \frac{2}{\sqrt{13}} & \frac{3}{\sqrt{13}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{18}} & 0 \\ 0 & \frac{1}{\sqrt{5}} \end{bmatrix} \begin{bmatrix} \frac{7}{\sqrt{234}} & -\frac{4}{\sqrt{234}} & \frac{13}{\sqrt{234}} \\ \frac{4}{\sqrt{65}} & \frac{7}{\sqrt{65}} & 0 \end{bmatrix} \\ &= \begin{bmatrix} -\frac{1}{\frac{30}{11}} & -\frac{4}{\frac{15}{13}} & \frac{1}{6} \\ \frac{1}{45} & \frac{1}{45} & \frac{1}{9} \end{bmatrix} \end{aligned}$$

Computer implementation [25]Figure 7.3: A two-phase procedure for the SVD: $A = U\Sigma V^T$.

Algorithm 7.21. (Golub and Reinsch, 1970) [26]. Let $A \in \mathbb{R}^{m \times n}$.

- **Phase 1:** It **constructs two finite sequences** of **Householder transformations** to find an **upper bidiagonal matrix**:

$$P_n \cdots P_1 A Q_1 \cdots Q_{n-2} = B \quad (7.45)$$

- **Phase 2:** It is to **iteratively diagonalize B** using the **QR method**.

Golub-Reinsch SVD algorithm

- It is extremely stable.
- Computational complexity:
 - Computation of U , V , and Σ : $4m^2n + 8mn^2 + 9n^3$.
 - Computation of V and Σ : $4mn^2 + 8n^3$.
- Phases 1 & 2 take $\mathcal{O}(mn^2)$ and $\mathcal{O}(n^2)$ flops, respectively. (when Phase 2 is done with $\mathcal{O}(n)$ iterations)
- Python: `U,S,V = numpy.linalg.svd(A)`
- Matlab/Maple: `[U,S,V] = svd(A)`
- Mathematica: `{U,S,V} = SingularValueDecomposition[A]`

Numerical rank

In the absence of round-off errors and uncertainties in the data, the SVD reveals the rank of the matrix. Unfortunately the presence of errors makes rank determination problematic. For example, consider

$$A = \begin{bmatrix} 1/3 & 1/3 & 2/3 \\ 2/3 & 2/3 & 4/3 \\ 1/3 & 2/3 & 3/3 \\ 2/5 & 2/5 & 4/5 \\ 3/5 & 1/5 & 4/5 \end{bmatrix} \quad (7.46)$$

- Obviously A is of rank 2, as its third column is the sum of the first two.
- Matlab “svd” (with IEEE double precision) produces

$$\sigma_1 = 2.5987, \quad \sigma_2 = 0.3682, \quad \text{and} \quad \sigma_3 = 8.6614 \times 10^{-17}.$$

- What is the rank of A , 2 or 3? What if σ_3 is in $\mathcal{O}(10^{-13})$?
- For this reason we must introduce a **threshold** T . Then we say that A has **numerical rank** r if A has r singular values larger than T , that is,

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > T \geq \sigma_{r+1} \geq \cdots \quad (7.47)$$

In Matlab

- Matlab has a “rank” command, which computes the numerical rank of the matrix with a default threshold

$$T = 2 \max\{m, n\} \epsilon \|A\|_2 \quad (7.48)$$

where ϵ is the unit round-off error.

- In Matlab, the unit round-off error can be found from the parameter “eps”

$$\text{eps} = 2^{-52} = 2.2204 \times 10^{-16}.$$

- For the matrix A in (7.46),

$$T = 2 \cdot 5 \cdot \text{eps} \cdot 2.5987 = 5.7702 \times 10^{-15}$$

and therefore $\text{rank}(A)=2$.

See Exercise 5.

7.2.4. Application of the SVD to image compression

- $A \in \mathbb{R}^{m \times n}$ is a sum of rank-1 matrices (dyadic decomposition):

$$\begin{aligned} V &= [\mathbf{v}_1, \dots, \mathbf{v}_n], \quad U = [\mathbf{u}_1, \dots, \mathbf{u}_n], \\ A &= U \Sigma V^T = \sum_{i=1}^n \sigma_i \mathbf{u}_i \mathbf{v}_i^T, \quad \mathbf{u}_i \in \mathbb{R}^m, \quad \mathbf{v}_i \in \mathbb{R}^n. \end{aligned} \quad (7.49)$$

- The approximation

$$A_k = U \Sigma_k V^T = \sum_{i=1}^k \sigma_i \mathbf{u}_i \mathbf{v}_i^T \quad (7.50)$$

is closest to A among matrices of rank $\leq k$, and

$$\|A - A_k\|_2 = \sigma_{k+1}. \quad (7.51)$$

- It only takes $(m + n) \cdot k$ words to store $\mathbf{u}_1$ through $\mathbf{u}_k$, and $\sigma_1 \mathbf{v}_1$ through $\sigma_k \mathbf{v}_k$, from which we can reconstruct A_k .

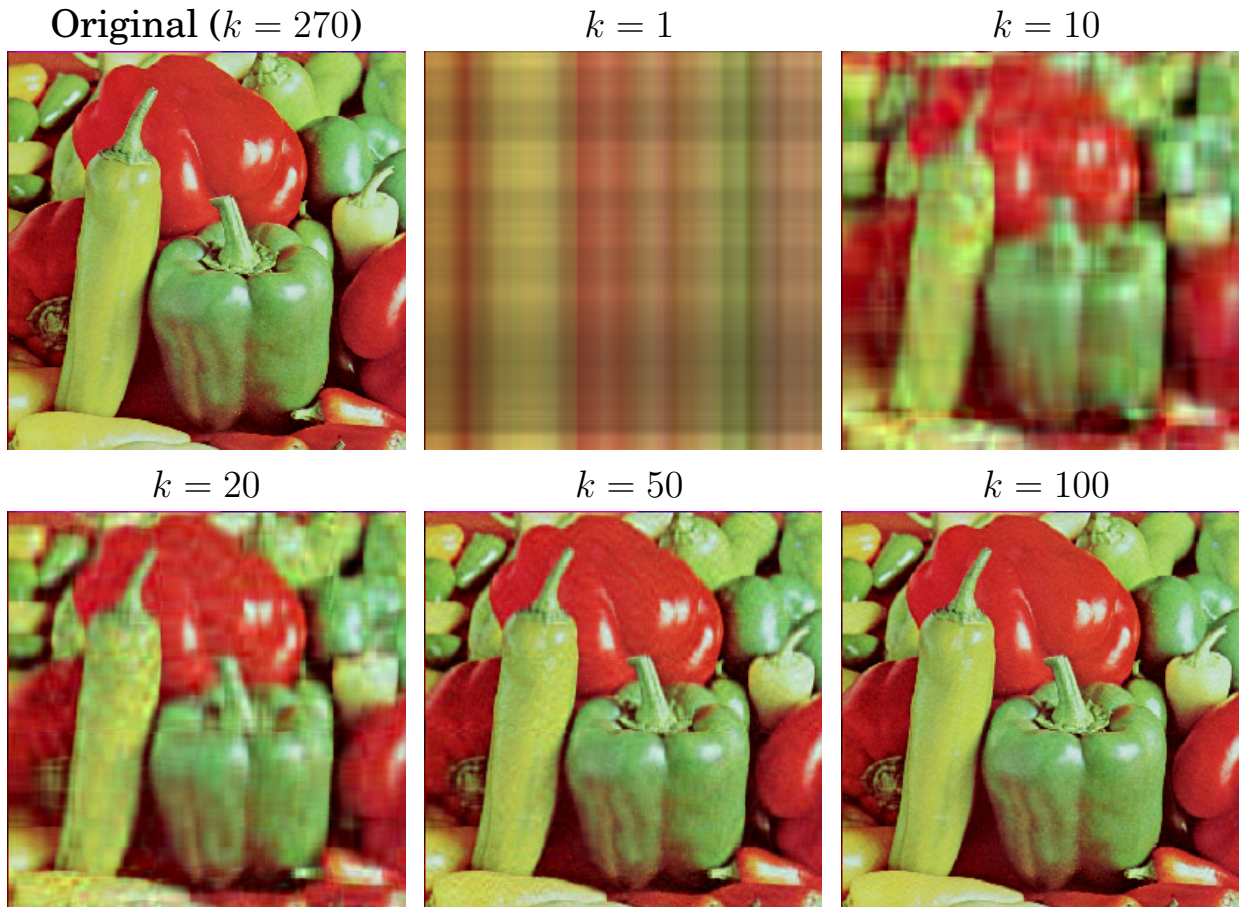
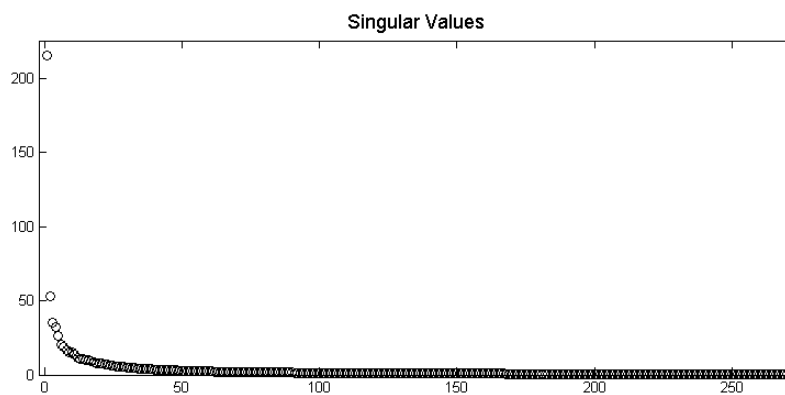
Image compression using k singular values

```

1  img = imread('Peppers.png'); [m,n,d]=size(img);
2  [U,S,V] = svd(reshape(im2double(img),m,[]));
3  %%---- select k <= p=min(m,n)
4  k = 20;
5  img_k = U(:,1:k)*S(1:k,1:k)*V(:,1:k)';
6  img_k = reshape(img_k,m,n,d);
7  figure, imshow(img_k)

```

The “Peppers” image is in $[270, 270, 3] \in \mathbb{R}^{270 \times 810}$.

**Peppers: Singular values**

Peppers: Storage: It requires $(m + n) \cdot k$ words. For example, when $k = 50$,

$$(m + n) \cdot k = (270 + 810) \cdot 50 = \boxed{54,000}, \quad (7.52)$$

which is approximately **a quarter** the full storage space

$$270 \times 270 \times 3 = \boxed{218,700}.$$

7.3. Linear Discriminant Analysis

Linear discriminant analysis is a method to find a **linear combination of features** that **characterizes or separates** two or more classes of objects or events.

- The LDA is sometimes also called **Fisher's LDA**. Fisher initially formulated the LDA for two-class classification problems in 1936 [20], and later generalized for multi-class problems by C. Radhakrishna Rao under the assumption of **equal class covariances** and **normally distributed classes** in 1948 [61].
- The LDA may be used as a **linear classifier**, or, more commonly, for **dimensionality reduction** (§ 7.3.4) for a later classification.
- The general concept behind the LDA is very similar to PCA.¹

LDA objective

- The LDA objective is to perform **dimensionality reduction**.
 - So what? PCA does that, too! 😐
- However, we want to preserve as much of the **class discriminatory information** as possible.
 - OK, this is new! 😊

LDA

- Consider a pattern classification problem, where we have c classes.
- Suppose each class has N_k samples in $\mathbb{R}^d$, where $k = 1, 2, \dots, c$.
- Let $\mathcal{X}_k = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(N_k)}\}$ be the set of d -dimensional samples for class k .
- Let $X \in \mathbb{R}^{d \times N}$ be the data matrix, stacking all the samples from all classes, such that each **column** represents a sample, where $N = \sum_k N_k$.
- The LDA seeks to obtain a transformation of X to Z through projecting the samples in X onto a **hyperplane** with dimension $c - 1$.

¹In PCA, the main idea is to re-express the available dataset to extract the relevant information by **reducing the redundancy** and to **minimize the noise**. While (unsupervised) PCA attempts to *find the orthogonal component axes of maximum variance* in a dataset, the goal in the (**supervised**) LDA is to find the feature subspace that **optimizes class separability**.

7.3.1. Fisher's LDA (classifier): two classes

Let us define a transformation of samples $\mathbf{x}$ onto a line $[(c - 1)\text{-space}]$, for $c = 2$]:

$$z = \mathbf{w}^T \mathbf{x} = \mathbf{w} \cdot \mathbf{x}, \quad (7.53)$$

where $\mathbf{w} \in \mathbb{R}^d$ is a **projection vector**.

Of all the possible lines, we would like to select the one that maximizes the separability of the scalars $\{z\}$.

- In order to find a good projection vector, we need to define a measure of separation between the projections.
- The mean vector of each class in $\mathbf{x}$ and z feature space is

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{\mathbf{x} \in \mathcal{X}_k} \mathbf{x}, \quad \tilde{\mu}_k = \frac{1}{N_k} \sum_{\mathbf{x} \in \mathcal{X}_k} z = \frac{1}{N_k} \sum_{\mathbf{x} \in \mathcal{X}_k} \mathbf{w}^T \mathbf{x} = \mathbf{w}^T \boldsymbol{\mu}_k, \quad (7.54)$$

i.e., projecting $\mathbf{x}$ to z will lead to projecting the mean of $\mathbf{x}$ to the mean of z .

- We could then choose the **distance between the projected means** as our objective function:

$$\hat{\mathcal{J}}(\mathbf{w}) = |\tilde{\mu}_1 - \tilde{\mu}_2| = |\mathbf{w}^T (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)|. \quad (7.55)$$

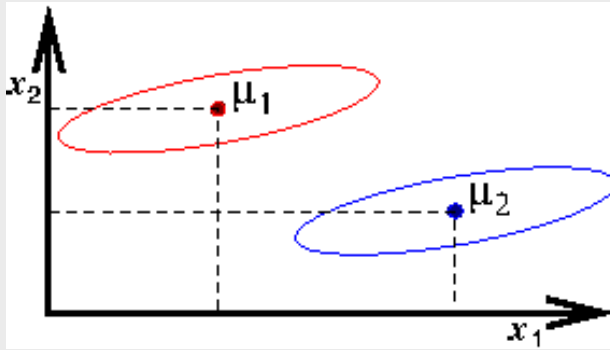


Figure 7.4: The x_1 -axis has a larger distance between means, while the x_2 -axis yields a better class separability.

- However, the distance between

the projected means is **not a very good measure**, since it does not take into account the sample distribution within the classes.

- The maximizer $\mathbf{w}^*$ of (7.55) must be parallel to $(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$:

$$\mathbf{w}^* \parallel (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2);$$

the projection to a parallel line of $(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$ is not an optimal transformation.

Fisher's LDA: The Key Idea

The solution proposed by Fisher is **to maximize a function that represents the difference between the means**, normalized by a measure of the **within-class variability** (called the **scatter**).

- For each class k , we define the **scatter** (an equivalent of the variance) as

$$\tilde{s}_k^2 = \sum_{\mathbf{x} \in \mathcal{X}_k} (z - \tilde{\mu}_k)^2, \quad z = \mathbf{w}^T \mathbf{x}. \quad (7.56)$$

- The quantity $\tilde{s}_k^2$ measures the variability within class $\mathcal{X}_k$ after projecting it on the z -axis.
- Thus, $\tilde{s}_1^2 + \tilde{s}_2^2$ measures the variability within the two classes at hand after projection; it is called the **within-class scatter** of the projected samples.
- **Fisher's linear discriminant** is defined as the linear function $\mathbf{w}^T \mathbf{x}$ that maximizes the objective function:

$$\mathbf{w}^* = \arg \max_{\mathbf{w}} \mathcal{J}(\mathbf{w}), \quad \text{where } \mathcal{J}(\mathbf{w}) = \frac{(\tilde{\mu}_1 - \tilde{\mu}_2)^2}{\tilde{s}_1^2 + \tilde{s}_2^2}. \quad (7.57)$$

- Therefore, Fisher's LDA searches for a projection where samples from the same class are projected very close to each other; at the same time, the projected means are as farther apart as possible.

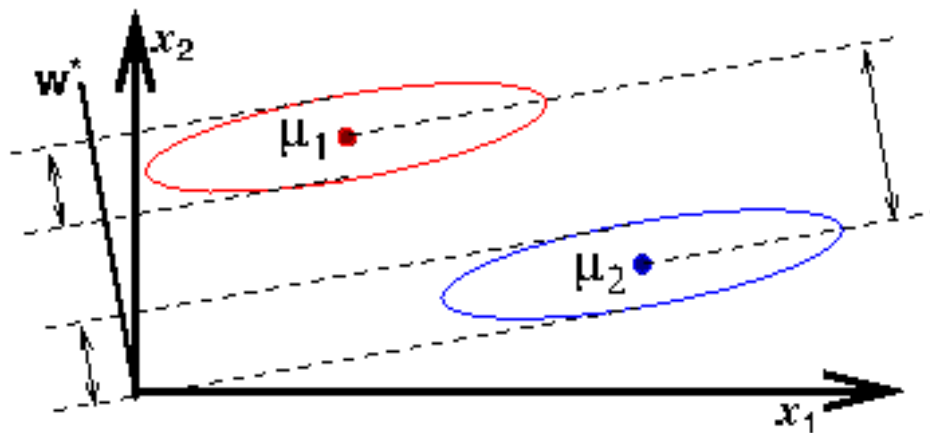


Figure 7.5: Fisher's LDA.

7.3.2. Fisher's LDA: the optimum projection

Rewrite the Fisher's objective function:

$$\mathcal{J}(\mathbf{w}) = \frac{(\tilde{\mu}_1 - \tilde{\mu}_2)^2}{\tilde{s}_1^2 + \tilde{s}_2^2}, \quad (7.58)$$

where

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{\mathbf{x} \in \mathcal{X}_k} \mathbf{x}, \quad \tilde{\mu}_k = \mathbf{w}^T \boldsymbol{\mu}_k, \quad \tilde{s}_k^2 = \sum_{\mathbf{x} \in \mathcal{X}_k} (z - \tilde{\mu}_k)^2.$$

- In order to express $\mathcal{J}(\mathbf{w})$ as an explicit function of $\mathbf{w}$, we first define a measure of the scatter in the feature space $\mathbf{x}$:

$$S_w = S_1 + S_2, \quad \text{for } S_k = \sum_{\mathbf{x} \in \mathcal{X}_k} (\mathbf{x} - \boldsymbol{\mu}_k)(\mathbf{x} - \boldsymbol{\mu}_k)^T, \quad (7.59)$$

where $S_w \in \mathbb{R}^{d \times d}$ is called the **within-class scatter matrix** of samples $\mathbf{x}$, while S_k is the **covariance matrix** of class $\mathcal{X}_k$.

Then, the scatter of the projection z can then be expressed as

$$\begin{aligned} \tilde{s}_k^2 &= \sum_{\mathbf{x} \in \mathcal{X}_k} (z - \tilde{\mu}_k)^2 = \sum_{\mathbf{x} \in \mathcal{X}_k} (\mathbf{w}^T \mathbf{x} - \mathbf{w}^T \boldsymbol{\mu}_k)^2 \\ &= \sum_{\mathbf{x} \in \mathcal{X}_k} \mathbf{w}^T (\mathbf{x} - \boldsymbol{\mu}_k)(\mathbf{x} - \boldsymbol{\mu}_k)^T \mathbf{w} \\ &= \mathbf{w}^T \left(\sum_{\mathbf{x} \in \mathcal{X}_k} (\mathbf{x} - \boldsymbol{\mu}_k)(\mathbf{x} - \boldsymbol{\mu}_k)^T \right) \mathbf{w} = \mathbf{w}^T S_k \mathbf{w}. \end{aligned} \quad (7.60)$$

Thus, the denominator of the objective function gives

$$\tilde{s}_1^2 + \tilde{s}_2^2 = \mathbf{w}^T S_1 \mathbf{w} + \mathbf{w}^T S_2 \mathbf{w} = \mathbf{w}^T S_w \mathbf{w} =: \tilde{S}_w, \quad (7.61)$$

where $\tilde{S}_w$ is the **within-class scatter** of projected samples z .

- Similarly, the difference between the projected means (in z -space) can be expressed in terms of the means in the original feature space ($\mathbf{x}$ -space).

$$\begin{aligned} (\tilde{\mu}_1 - \tilde{\mu}_2)^2 &= (\mathbf{w}^T \boldsymbol{\mu}_1 - \mathbf{w}^T \boldsymbol{\mu}_2)^2 = \mathbf{w}^T \underbrace{(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T}_{=: S_b} \mathbf{w} \\ &= \mathbf{w}^T S_b \mathbf{w} =: \tilde{S}_b, \end{aligned} \quad (7.62)$$

where the rank-one matrix $S_b \in \mathbb{R}^{d \times d}$ is called the **between-class scatter matrix** of the original samples $\mathbf{x}$, while $\tilde{S}_b$ is the **between-class scatter** of the projected samples $\mathbf{z}$.

- Since S_b is the outer product of two vectors, $\text{rank}(S_b) \leq 1$.

We can finally express the Fisher criterion in terms of S_w and S_b as

$$\mathcal{J}(\mathbf{w}) = \frac{(\tilde{\mu}_1 - \tilde{\mu}_2)^2}{\tilde{s}_1^2 + \tilde{s}_2^2} = \frac{\mathbf{w}^T S_b \mathbf{w}}{\mathbf{w}^T S_w \mathbf{w}}. \quad (7.63)$$

Hence, $\mathcal{J}(\mathbf{w})$ is a measure of the difference between class means (encoded in the between-class scatter matrix), normalized by a measure of the within-class scatter matrix.

- To find the maximum of $\mathcal{J}(\mathbf{w})$, we differentiate it with respect to $\mathbf{w}$ and equate to zero. Applying some algebra leads (Exercise 6)

$$S_w^{-1} S_b \mathbf{w} = \mathcal{J}(\mathbf{w}) \mathbf{w}. \quad (7.64)$$

Note that $S_w^{-1} S_b$ is a **rank-one matrix**.

Equation (7.64) is a **generalized eigenvalue problem**:

$$S_w^{-1} S_b \mathbf{w} = \lambda \mathbf{w} \iff S_b \mathbf{w} = \lambda S_w \mathbf{w}; \quad (7.65)$$

the maximizer $\mathbf{w}^*$ of $\mathcal{J}(\mathbf{w})$ is the eigenvector associated with the **nonzero eigenvalue** $\lambda^* = \mathcal{J}(\mathbf{w})$.

Summary 7.22. Finding the eigenvector of $S_w^{-1} S_b$ associated with the largest eigenvalue yields

$$\mathbf{w}^* = \arg \max_{\mathbf{w}} \mathcal{J}(\mathbf{w}) = \arg \max_{\mathbf{w}} \frac{\mathbf{w}^T S_b \mathbf{w}}{\mathbf{w}^T S_w \mathbf{w}}. \quad (7.66)$$

This is known as Fisher's Linear Discriminant, although it is not a discriminant but a specific choice of direction for the projection of the data down to one dimension.

Fisher's LDA: an example

We will compute the Linear Discriminant projection for the following two-dimensional dataset of two classes ($c = 2$).

```

1  m=2; n=5;
2
3  X1=[2,3; 4,3; 2,1; 3,4; 5,4];
4  X2=[7,4; 6,8; 7,6; 8,9; 10,9];
5
6  Mu1 = mean(X1)';           % Mu1 = [3.2,3.0]
7  Mu2 = mean(X2)';           % Mu2 = [7.6,7.2]
8
9  S1 = cov(X1,0)*n;
10 S2 = cov(X2,0)*n;
11 Sw = S1+S2;                 % Sw = [20,13; 13,31]
12
13 Sb = (Mu1-Mu2)*(Mu1-Mu2)'; % Sb = [19.36,18.48; 18.48,17.64]
14
15 invSw_Sb = inv(Sw)*Sb;
16 [V,L] = eig(invSw_Sb);       % V1 = [ 0.9503,0.3113]; L1 = 1.0476
17                               % V2 = [-0.6905,0.7234]; L2 = 0.0000

```

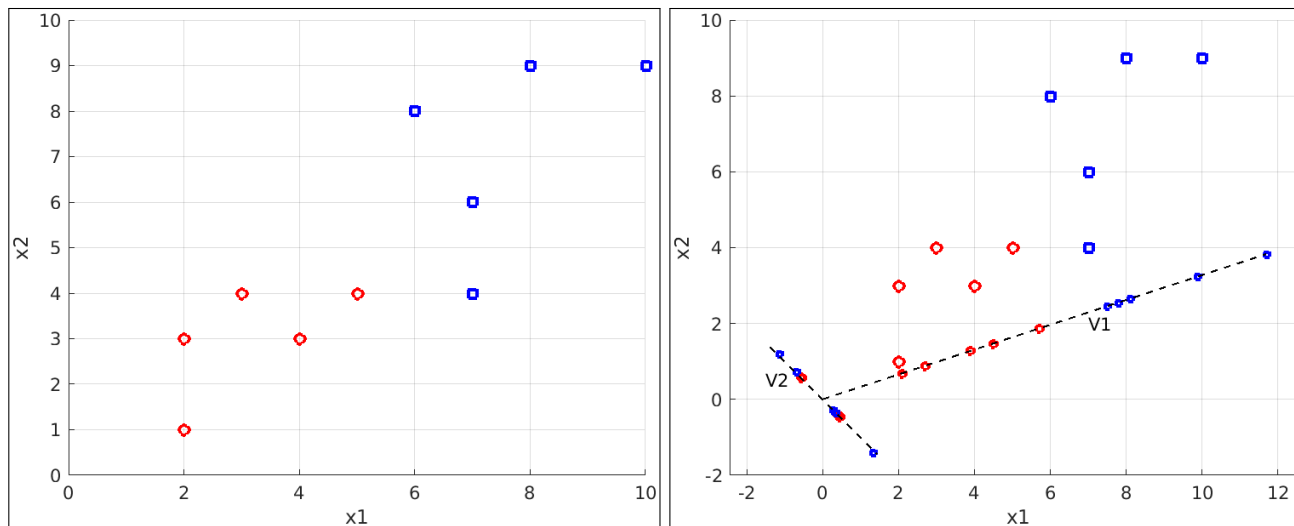


Figure 7.6: A synthetic dataset and Fisher's LDA projection.

7.3.3. LDA for multiple classes

- Now, we have c -classes instead of just two.
- We are now seeking $(c-1)$ projections $[z_1, z_2, \dots, z_{c-1}]$ by means of $(c-1)$ projection vectors $\mathbf{w}_k \in \mathbb{R}^d$.
- Let $W = [\mathbf{w}_1 | \mathbf{w}_2 | \dots | \mathbf{w}_{c-1}]$, a collection of column vectors, such that

$$z_k = \mathbf{w}_k^T \mathbf{x} \implies \mathbf{z} = W^T \mathbf{x} \in \mathbb{R}^{c-1}. \quad (7.67)$$

- If we have N sample (column) vectors, we can stack them into one matrix as follows.

$$Z = W^T X, \quad (7.68)$$

where $X \in \mathbb{R}^{d \times N}$, $W \in \mathbb{R}^{d \times (c-1)}$, and $Z \in \mathbb{R}^{(c-1) \times N}$.

Recall: For the two classes case, the **within-class scatter matrix** was computed as

$$S_w = S_1 + S_2.$$

This can be generalized in the c -classes case as:

$$S_w = \sum_{k=1}^c S_k, \quad S_k = \sum_{\mathbf{x} \in \mathcal{X}_k} (\mathbf{x} - \boldsymbol{\mu}_k)(\mathbf{x} - \boldsymbol{\mu}_k)^T, \quad (7.69)$$

where $\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{\mathbf{x} \in \mathcal{X}_k} \mathbf{x}$, where N_k is the number of data samples in class $\mathcal{X}_k$, and $S_w \in \mathbb{R}^{d \times d}$.

Recall: For the two classes case, the **between-class scatter matrix** was computed as

$$S_b = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T.$$

For c -classes case, we will measure the **between-class scatter matrix** with respect to the mean of all classes as follows:

$$S_b = \sum_{k=1}^c N_k (\boldsymbol{\mu}_k - \boldsymbol{\mu})(\boldsymbol{\mu}_k - \boldsymbol{\mu})^T, \quad \boldsymbol{\mu} = \frac{1}{N} \sum_{\forall \mathbf{x}} \mathbf{x}, \quad (7.70)$$

where $\text{rank}(S_b) = c - 1$.

Definition 7.23. As an analogue to (7.66), we may define the LDA optimization, for c classes case, as follows.

$$W^* = \arg \max_W \mathcal{J}(W) = \arg \max_W \frac{W^T S_b W}{W^T S_w W}. \quad (7.71)$$

Recall: For two-classes case, when we set $\frac{\partial \mathcal{J}(\mathbf{w})}{\partial \mathbf{w}} = 0$, the optimization problem is reduced to the eigenvalue problem

$$S_w^{-1} S_b \mathbf{w}^* = \lambda^* \mathbf{w}^*, \quad \text{where } \lambda^* = \mathcal{J}(\mathbf{w}^*).$$

For c -classes case, we have $(c - 1)$ projection vectors. Hence the eigenvalue problem can be generalized to the c -classes case:

$$S_w^{-1} S_b \mathbf{w}_k^* = \lambda_k^* \mathbf{w}_k^*, \quad \lambda_k^* = \mathcal{J}(\mathbf{w}_k^*), \quad k = 1, 2, \dots, c - 1. \quad (7.72)$$

Thus, it can be shown that the optimal projection matrix

$$W^* = [\mathbf{w}_1^* | \mathbf{w}_2^* | \dots | \mathbf{w}_{c-1}^*] \in \mathbb{R}^{d \times (c-1)} \quad (7.73)$$

is the one whose columns are the eigenvectors corresponding to the **eigenvalues** of the following generalized eigenvalue problem:

$$S_w^{-1} S_b W^* = \boldsymbol{\lambda}^* \cdot W^*, \quad \boldsymbol{\lambda}^* = [\lambda_1^*, \dots, \lambda_{c-1}^*], \quad (7.74)$$

where $S_w^{-1} S_b \in \mathbb{R}^{d \times d}$ and $(\cdot)$ denotes the pointwise product.

Illustration – 3 classes

- Let us generate a dataset for each class to illustrate the LDA transformation.
- For each class:
 - Use the random number generator to generate a uniform stream of 500 samples that follows $\mathcal{U}(0, 1)$.
 - Using the Box-Muller approach, convert the generated uniform stream to $\mathcal{N}(0, 1)$.
 - Then use the method of eigenvalues and eigenvectors to manipulate the standard normal to have the required mean vector and covariance matrix .
 - Estimate the mean and covariance matrix of the resulted dataset.

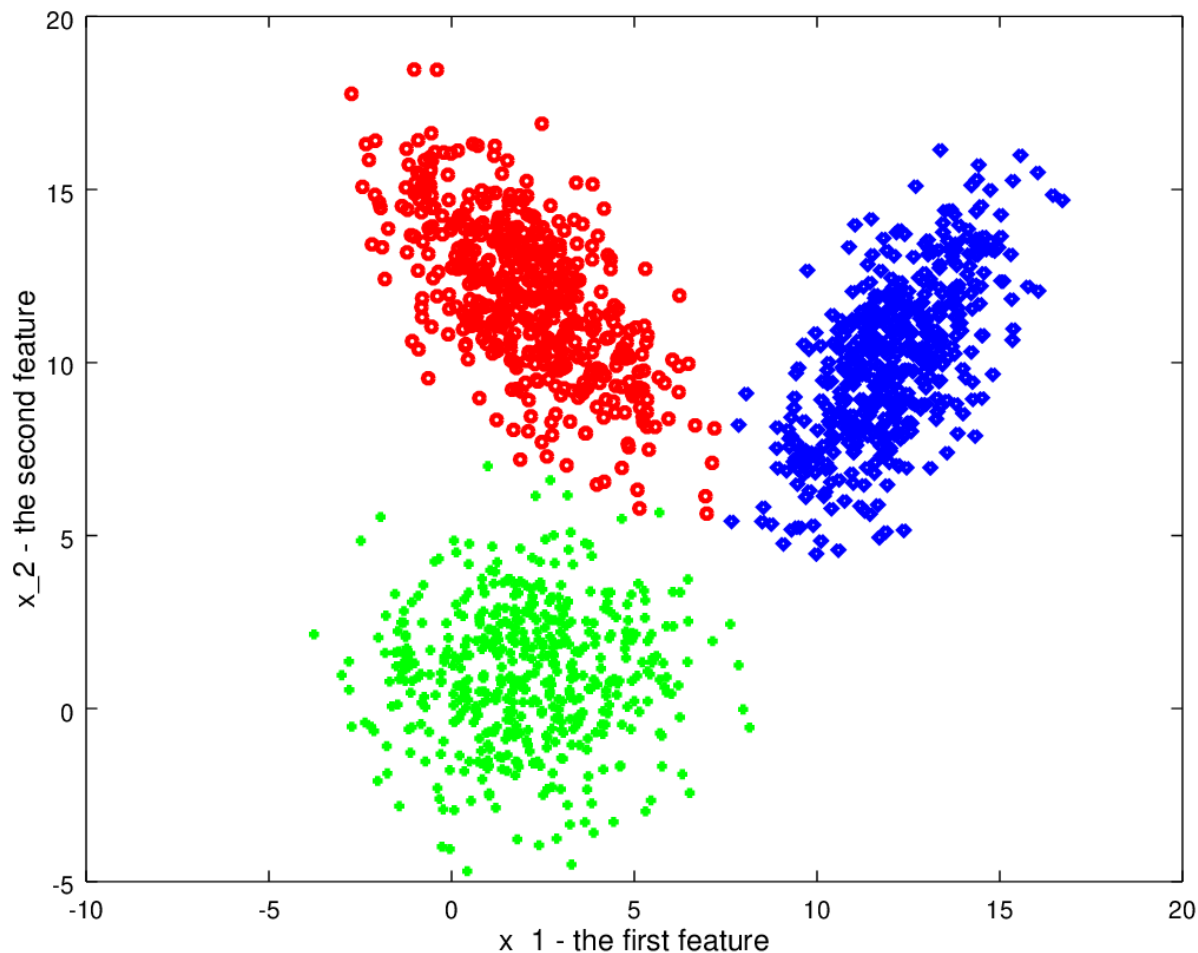


Figure 7.7: Generated and manipulated dataset, for 3 classes.

```

1      close all;
2      try, pkg load statistics; end % for octave
3
4      %% uniform stream
5      U = rand(2,1000); u1 = U(:,1:2:end); u2 = U(:,2:2:end);
6
7      %% Box-Muller method to convert to N(0,1)
8      X = sqrt((-2).*log(u1)).*(cos(2*pi.*u2)); % 2 x 500
9      clear u1 u2 U;
10
11     %% manipulate for required Mean and Cov
12     Mu = [5;5];
13
14     Mu1= Mu +[-3;7]; Cov1 =[5 -1; -3 3];
15     X1 = denormalize(X,Mu1,Cov1);
16     Mu2= Mu +[-3;-4]; Cov2 =[4 0; 0 4];
17     X2 = denormalize(X,Mu2,Cov2);
18     Mu3= Mu +[7; 5]; Cov3 =[4 1; 3 3];
19     X3 = denormalize(X,Mu3,Cov3);
20
21     %%Begin the computation of the LDA Projection Vectors
22     % estimate mean and covariance
23     N1 = size(X1,2); N2 = size(X2,2); N3 = size(X3,2);
24     Mu1 = mean(X1')'; Mu2 = mean(X2')'; Mu3 = mean(X3')';
25     Mu = (Mu1+Mu2+Mu3)/3.;
26
27     % within-class scatter matrix
28     S1 = cov(X1'); S2 = cov(X2'); S3 = cov(X3');
29     Sw = S1+S2+S3;
30
31     % between-class scatter matrix
32     Sb1 = N1 * (Mu1-Mu)*(Mu1-Mu)'; Sb2 = N2 * (Mu2-Mu)*(Mu2-Mu)';
33     Sb3 = N3 * (Mu3-Mu)*(Mu3-Mu)';
34     Sb = Sb1+Sb2+Sb3;
35
36     % computing the LDA projection
37     invSw_Sb = inv(Sw)*Sb; [V,D] = eig(invSw_Sb);
38     w1 = V(:,1); w2 = V(:,2);
39     if D(1,1)<D(2,2), w1 = V(:,2); w2 = V(:,1); end
40
41     lda_c3_visualize;

```

Figure 7.8: lda_c3.m

```

1 function Xnew = denormalize(X,Mu,Cov)
2 % it manipulates data samples in N(0,1) to something else.
3
4 [V,D] = eig(Cov); VsD = V*sqrt(D);
5
6 Xnew = zeros(size(X));
7 for j=1:size(X,2)
8     Xnew(:,j)= VsD * X(:,j);
9 end
10
11 %Now, add "replicated and tiled Mu"
12 Xnew = Xnew + repmat(Mu,1,size(Xnew,2));

```

```

1 figure, hold on; axis([-10 20 -5 20]);
2 xlabel('x_1 - the first feature','fontsize',12);
3 ylabel('x_2 - the second feature','fontsize',12);
4 plot(X1(1,:),X1(2:),'ro','markersize',4,"linewidth",2)
5 plot(X2(1,:),X2(2:),'g+','markersize',4,"linewidth",2)
6 plot(X3(1,:),X3(2:),'bd','markersize',4,"linewidth",2)
7 hold off
8 print -dpng 'LDA_c3_Data.png'
9
10 figure, hold on; axis([-10 20 -5 20]);
11 xlabel('x_1 - the first feature','fontsize',12);
12 ylabel('x_2 - the second feature','fontsize',12);
13 plot(X1(1,:),X1(2:),'ro','markersize',4,"linewidth",2)
14 plot(X2(1,:),X2(2:),'g+','markersize',4,"linewidth",2)
15 plot(X3(1,:),X3(2:),'bd','markersize',4,"linewidth",2)
16
17 plot(Mu1(1),Mu1(2),'c.','markersize',20)
18 plot(Mu2(1),Mu2(2),'m.','markersize',20)
19 plot(Mu3(1),Mu3(2),'r.','markersize',20)
20 plot(Mu(1),Mu(2),'k*','markersize',15,"linewidth",3)
21 text(Mu(1)+0.5,Mu(2)-0.5,'\mu','fontsize',18)
22
23 t = -5:20; line1_x = t*w1(1); line1_y = t*w1(2);
24 plot(line1_x,line1_y,'k-',"linewidth",3);
25 t = -5:10; line2_x = t*w2(1); line2_y = t*w2(2);
26 plot(line2_x,line2_y,'m--',"linewidth",3);
27 hold off
28 print -dpng 'LDA_c3_Data_projection.png'
29
30 %Project the samples through w1
31 wk = w1;
32 z1_wk = wk'*X1; z2_wk = wk'*X2; z3_wk = wk'*X3;

```

```

33
34 z1_wk_Mu = mean(z1_wk); z1_wk_sigma = std(z1_wk);
35 z1_wk_pdf = mvnpdf(z1_wk',z1_wk_Mu,z1_wk_sigma);
36
37 z2_wk_Mu = mean(z2_wk); z2_wk_sigma = std(z2_wk);
38 z2_wk_pdf = mvnpdf(z2_wk',z2_wk_Mu,z2_wk_sigma);
39
40 z3_wk_Mu = mean(z3_wk); z3_wk_sigma = std(z3_wk);
41 z3_wk_pdf = mvnpdf(z3_wk',z3_wk_Mu,z3_wk_sigma);
42
43 figure, plot(z1_wk,z1_wk_pdf,'ro',z2_wk,z2_wk_pdf,'g+',...
44             z3_wk,z3_wk_pdf,'bd')
45 xlabel('z','fontsize',12); ylabel('p(z|w1)','fontsize',12);
46 print -dpng 'LDA_c3_Xw1_pdf.png'
47
48 %Project the samples through w2
49 wk = w2;
50 z1_wk = wk'*X1; z2_wk = wk'*X2; z3_wk = wk'*X3;
51
52 z1_wk_Mu = mean(z1_wk); z1_wk_sigma = std(z1_wk);
53 z1_wk_pdf = mvnpdf(z1_wk',z1_wk_Mu,z1_wk_sigma);
54
55 z2_wk_Mu = mean(z2_wk); z2_wk_sigma = std(z2_wk);
56 z2_wk_pdf = mvnpdf(z2_wk',z2_wk_Mu,z2_wk_sigma);
57
58 z3_wk_Mu = mean(z3_wk); z3_wk_sigma = std(z3_wk);
59 z3_wk_pdf = mvnpdf(z3_wk',z3_wk_Mu,z3_wk_sigma);
60
61 figure, plot(z1_wk,z1_wk_pdf,'ro',z2_wk,z2_wk_pdf,'g+',...
62             z3_wk,z3_wk_pdf,'bd')
63 xlabel('z','fontsize',12); ylabel('p(z|w2)','fontsize',12);
64 print -dpng 'LDA_c3_Xw2_pdf.png'

```

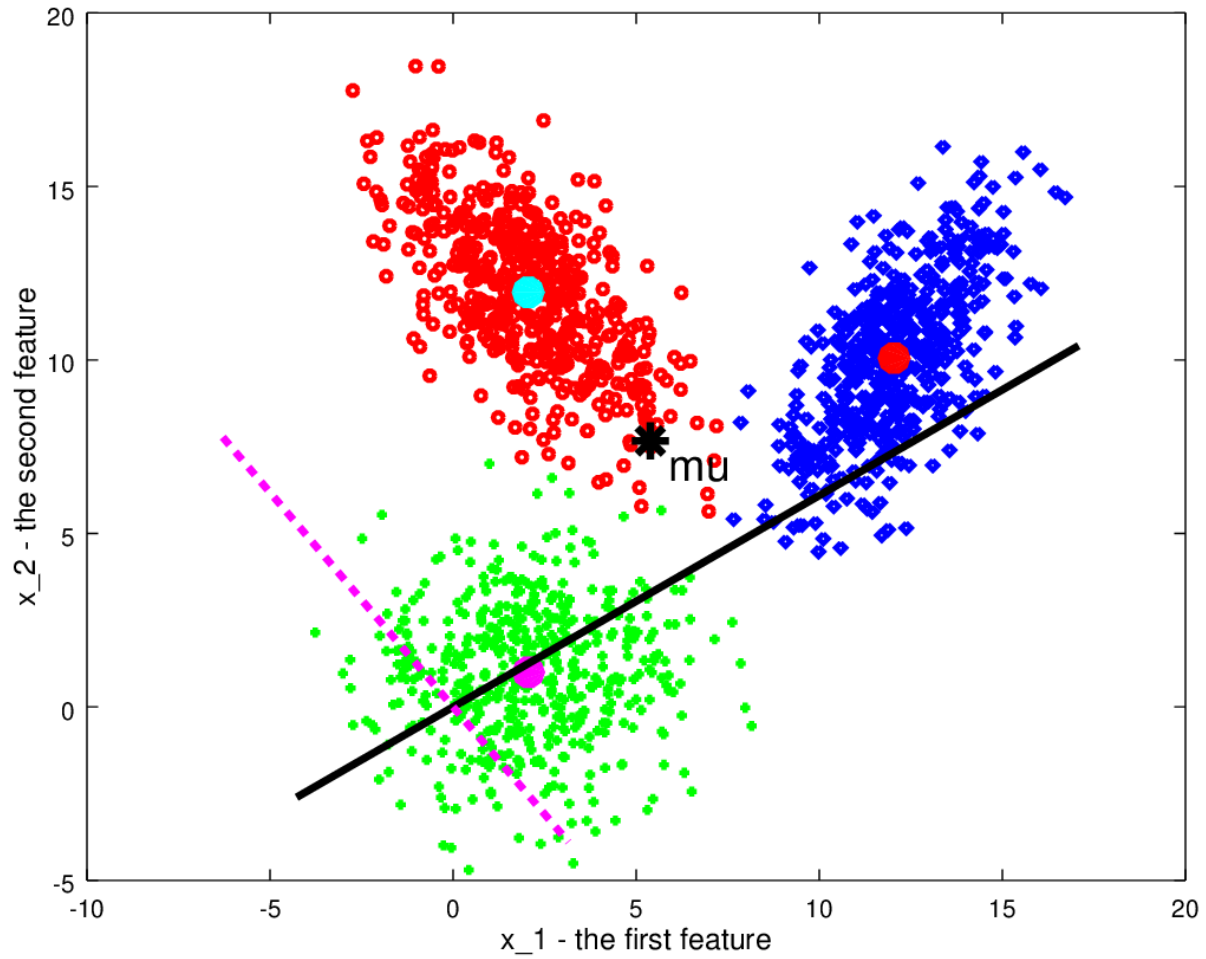


Figure 7.9: w_1^* (solid line in black) and w_2^* (dashed line in magenta).

- $w_1^* = [0.85395, 0.52036]^T$, $w_2^* = [-0.62899, 0.77742]^T$.
- Corresponding eigenvalues read

$$\lambda_1 = 3991.2, \quad \lambda_2 = 1727.7.$$

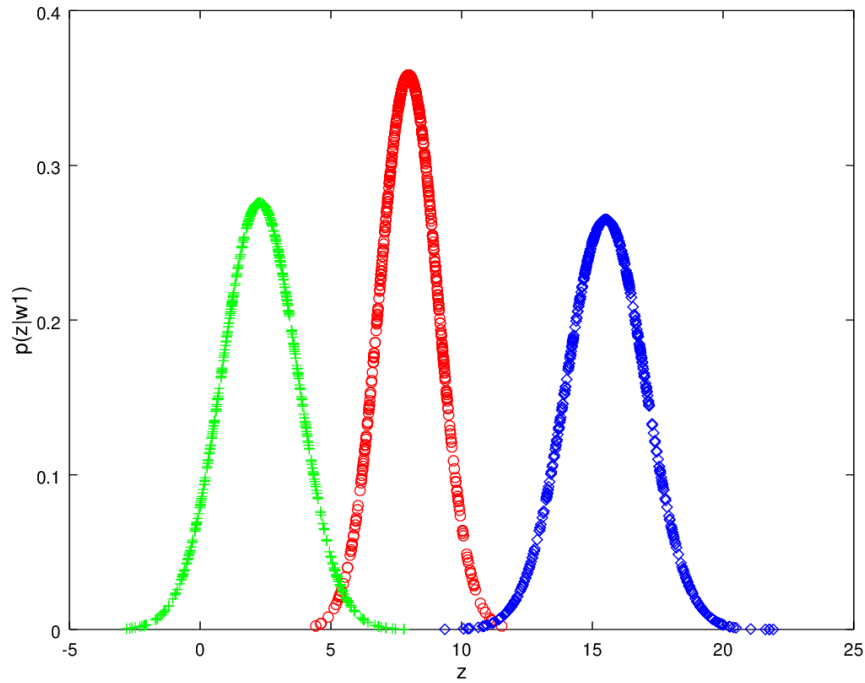


Figure 7.10: Classes PDF, along the first projection vector w_1^* ; $\lambda_1 = 3991.2$.

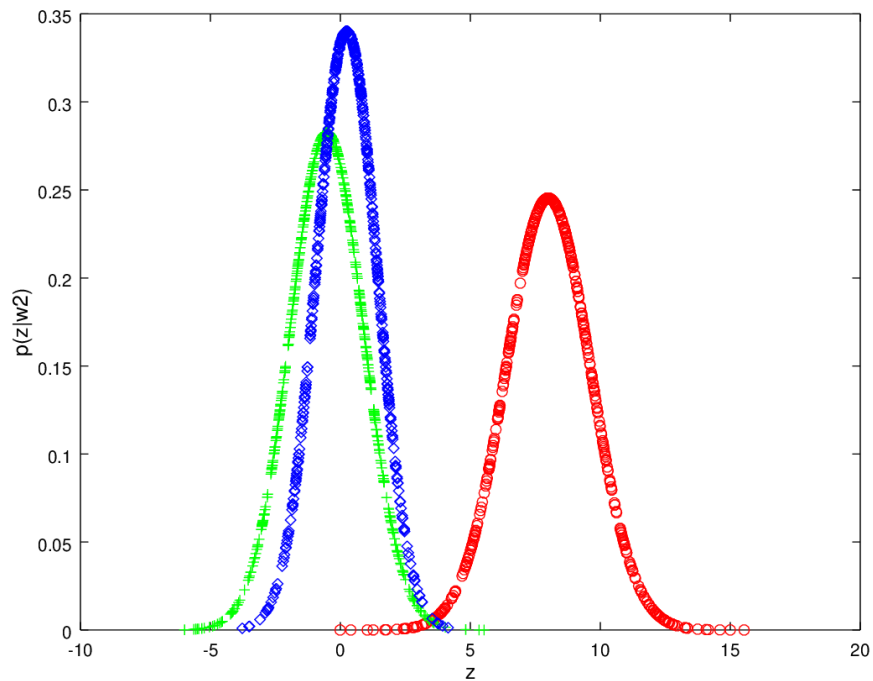


Figure 7.11: Classes PDF, along the second projection vector w_2^* ; $\lambda_2 = 1727.7$.

Apparently, the projection vector that has the highest eigenvalue provides higher discrimination power between classes.

7.3.4. The LDA: dimensionality reduction

Let $X \in \mathbb{R}^{N \times d}$ be the data matrix, in which each **row** represents a sample.

We summarize the main steps that are required to perform the LDA for dimensionality reduction.

1. **Standardize** the d -dimensional dataset (d is the number of features).
2. For each class j , compute the d -dimensional **mean vector** μ_j .
3. Construct the **within-class scatter matrix** S_w (7.69) and the **between-class scatter matrix** S_b (7.70).
4. Compute the **eigenvectors** and corresponding **eigenvalues** of the matrix $S_w^{-1}S_b$ (7.72).
5. Sort the eigenvalues by **decreasing order** to rank the corresponding eigenvectors.
6. Choose the k eigenvectors that correspond to the k largest eigenvalues to **construct a transformation matrix**

$$W = [\mathbf{w}_1 | \mathbf{w}_2 | \cdots | \mathbf{w}_k] \in \mathbb{R}^{d \times k}; \quad (7.75)$$

the eigenvectors are the columns of this matrix.

7. **Project the samples** onto a new feature subspace: $X \rightarrow Z := XW$.

Remark 7.24.

- $\text{rank}(S_w^{-1}S_b) \leq c - 1$; we must have $k \leq c - 1$.
- The projected feature Z_{ij} is $\mathbf{x}^{(i)} \cdot \mathbf{w}_j$ in the projected coordinates and $(\mathbf{x}^{(i)} \cdot \mathbf{w}_j) \mathbf{w}_j$ in the original coordinates.

Limitations of the LDA (classifier)

- The LDA produces **at most** $(c - 1)$ **feature projections**.
 - If the classification error estimates establish that more features are needed, some other method must be employed to provide those additional features.
- The LDA is a parametric method, since it **assumes unimodal Gaussian likelihoods**.
 - If the distributions are significantly non-Gaussian, the LDA projections will not be able to preserve any complex structure of the data, which may be needed for classification.
- The LDA will **fail** when the discriminatory information is **not in the mean** but rather **in the variance of the data**.

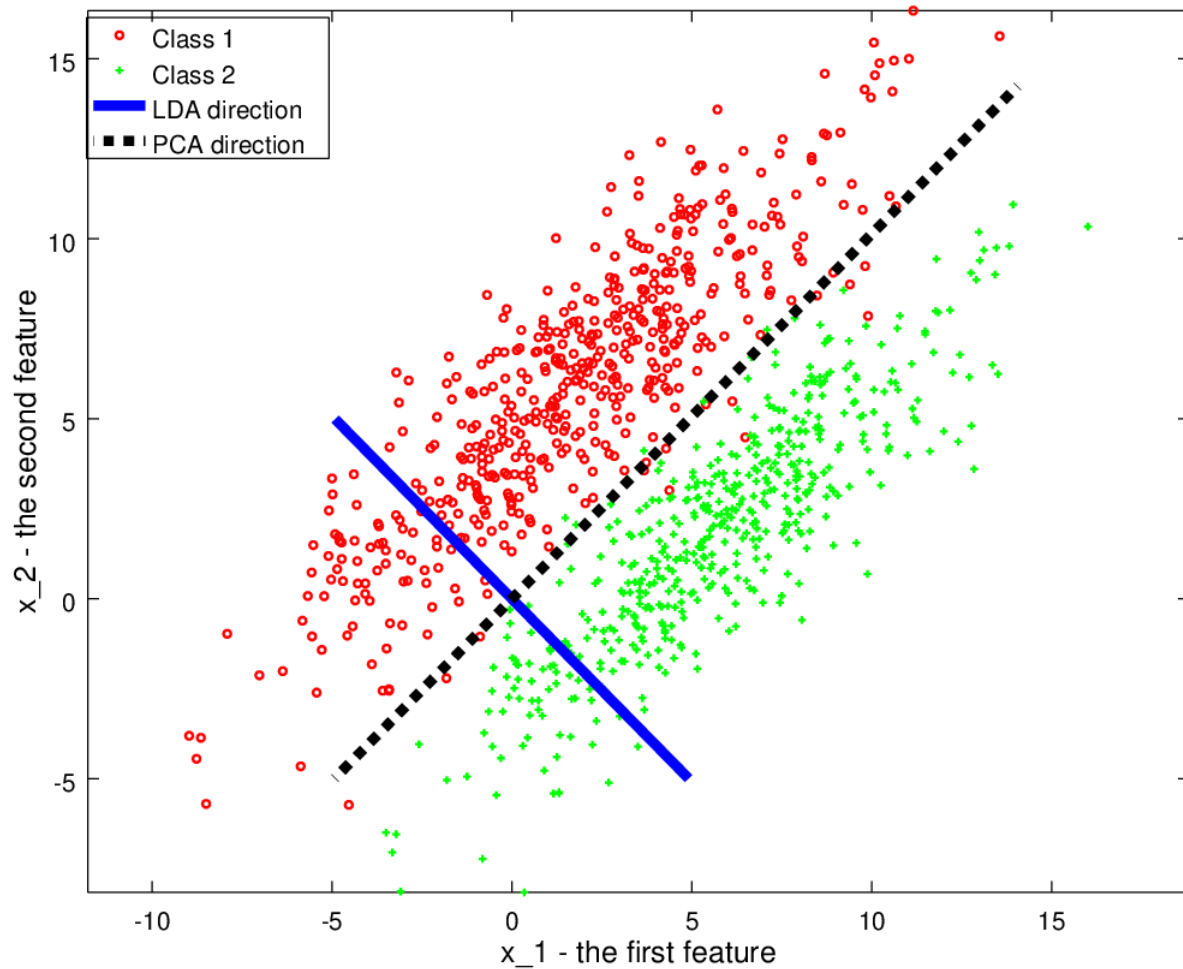
LDA vs. PCA

Figure 7.12: PCA vs. LDA.

😊 The (supervised) **LDA classifier must work better** than the (unsupervised) PCA, for datasets in Figures 7.9 and 7.12.

Recall: Fisher's LDA was generalized under the assumption of **equal class covariances** and **normally distributed classes**.

😊 However, even if one or more of those assumptions are (slightly) violated, the LDA for dimensionality reduction can still work reasonably well.

7.4. Kernel Principal Component Analysis

The **kernel principal component analysis** (kernel PCA) [70] is **an extension of the PCA using kernel techniques** and performing the originally linear operations of the PCA in a **kernel Hilbert space**.

Recall: (PCA). Consider a **data matrix** $X \in \mathbb{R}^{N \times d}$:

- each of the N rows represents a different data point,
- each of the d columns gives a particular kind of feature, and
- each column has zero empirical mean (e.g., after standardization).

- The goal of the standard PCA is to find an **orthogonal** weight matrix $W_k \in \mathbb{R}^{d \times k}$ such that

$$Z_k = X W_k, \quad k \leq d, \quad (7.76)$$

where $Z_k \in \mathbb{R}^{N \times k}$ is call the **truncated score matrix** and $Z_d = Z$. Columns of Z represent the **principal components** of X .

- (Claim 7.3, p. 152). The transformation matrix W_k turns out to be the collection of normalized eigenvectors of $X^T X$:

$$W_k = [\mathbf{w}_1 | \mathbf{w}_2 | \cdots | \mathbf{w}_k], \quad (X^T X) \mathbf{w}_j = \lambda_j \mathbf{w}_j, \quad \mathbf{w}_i^T \mathbf{w}_j = \delta_{ij}, \quad (7.77)$$

where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k \geq 0$.

- (Remark 7.4, p. 152). The matrix $Z_k \in \mathbb{R}^{N \times k}$ is scaled eigenvectors of XX^T :

$$Z_k = [\sqrt{\lambda_1} \mathbf{u}_1 | \sqrt{\lambda_2} \mathbf{u}_2 | \cdots | \sqrt{\lambda_k} \mathbf{u}_k], \quad (XX^T) \mathbf{u}_j = \lambda_j \mathbf{u}_j, \quad \mathbf{u}_i^T \mathbf{u}_j = \delta_{ij}. \quad (7.78)$$

- A **data (row) vector** $\mathbf{x}$ (**new or old**) is transformed to a k -dimensional row vector of principal components

$$\mathbf{z} = \mathbf{x} W_k \in \mathbb{R}^{1 \times k}. \quad (7.79)$$

- (Remark 7.5, p. 153). Let $\mathbf{X} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^T$ be the **SVD** of X , where

$$\mathbf{\Sigma} = \text{diag}(\sigma_1, \sigma_2, \cdots, \sigma_d), \quad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_d \geq 0.$$

Then,

$$\begin{aligned} V &\cong W; \quad \sigma_j^2 = \lambda_j, \quad j = 1, 2, \cdots, d, \\ Z_k &= [\sigma_1 \mathbf{u}_1 | \sigma_2 \mathbf{u}_2 | \cdots | \sigma_k \mathbf{u}_k]. \end{aligned} \quad (7.80)$$

7.4.1. Principal components of the kernel PCA

Note: Let $C = \frac{1}{N} X^T X$, the covariance matrix of X . Then,

$$C = \frac{1}{N} \sum_{i=1}^N \mathbf{x}^{(i)} \mathbf{x}^{(i)T} \in \mathbb{R}^{d \times d}, \quad C_{jk} = \frac{1}{N} \sum_{i=1}^N x_j^{(i)} x_k^{(i)}. \quad (7.81)$$

Here, we consider $\mathbf{x}^{(i)}$ as a column vector (when standing alone), while it lies in X as a row.

- The kernel PCA is a generalization of the PCA, where the dataset X is **transformed into a higher dimensional space** (by creating non-linear combinations of the original features):

$$\phi : X \in \mathbb{R}^{N \times d} \rightarrow \phi(X) \in \mathbb{R}^{N \times p}, \quad d < p, \quad (7.82)$$

and the **covariance matrix** is computed via outer products between such expanded samples:

$$C = \frac{1}{N} \sum_{i=1}^N \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T = \frac{1}{N} \phi(X)^T \phi(X) \in \mathbb{R}^{p \times p}. \quad (7.83)$$

- To obtain the eigenvectors – **the principal components** – from the covariance matrix, we should solve the eigenvalue problem:

$$C\mathbf{v} = \lambda\mathbf{v}. \quad (7.84)$$

- Assume (7.84) is solved.

– Let, for $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_k \geq \dots \geq \lambda_p \geq 0$,

$$V_k = [\mathbf{v}_1 | \mathbf{v}_2 | \dots | \mathbf{v}_k] \in \mathbb{R}^{p \times k}, \quad C\mathbf{v}_j = \lambda_j \mathbf{v}_j, \quad \mathbf{v}_i^T \mathbf{v}_j = \delta_{ij}. \quad (7.85)$$

– Then, the **score matrix Z_k (principal components)** for the kernel PCA reads

$$Z_k = \phi(X) V_k \in \mathbb{R}^{N \times k}, \quad (7.86)$$

which is an analogue to (7.76).

However, it is computationally expensive or impossible to solve the eigenvalue problem (7.84), when p is large or infinity.

An Alternative to the Computation of the Score Matrix

Claim 7.25. Let $\mathbf{v}$ be an eigenvector of C as in (7.84). Then it can be expressed as linear combination of data points:

$$\mathbf{v} = \sum_{i=1}^N \alpha_i \phi(\mathbf{x}^{(i)}). \quad (7.87)$$

Proof. Since $C\mathbf{v} = \lambda\mathbf{v}$, we get

$$\frac{1}{N} \sum_{i=1}^N \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T \mathbf{v} = \lambda \mathbf{v}$$

and therefore

$$\mathbf{v} = \frac{1}{\lambda N} \sum_{i=1}^N \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T \mathbf{v} = \frac{1}{\lambda N} \sum_{i=1}^N [\phi(\mathbf{x}^{(i)}) \cdot \mathbf{v}] \phi(\mathbf{x}^{(i)}), \quad (7.88)$$

where $\phi(\mathbf{x}^{(i)}) \cdot \mathbf{v}$ is a scalar and $\alpha_i := (\phi(\mathbf{x}^{(i)}) \cdot \mathbf{v})/(\lambda N)$. $\square$

Note:

- The above claim means that all eigenvectors $\mathbf{v}$ with $\lambda \neq 0$ lie in the span of $\phi(\mathbf{x}^{(1)}), \dots, \phi(\mathbf{x}^{(N)})$.
- Thus, finding the eigenvectors in (7.84) is equivalent to finding the coefficients $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_N)^T$.

How to find α

- Let $C\mathbf{v}_j = \lambda_j \mathbf{v}_j$ with $\lambda_j \neq 0$. Then, (7.87) can be written as

$$\mathbf{v}_j = \sum_{\ell=1}^N \alpha_{\ell j} \phi(\mathbf{x}^{(\ell)}) = \phi(X)^T \boldsymbol{\alpha}_j. \quad (7.89)$$

- By substituting this back into the equation and using (7.83), we get

$$C\mathbf{v}_j = \lambda_j \mathbf{v}_j \Rightarrow \frac{1}{N} \phi(X)^T \phi(X) \phi(X)^T \boldsymbol{\alpha}_j = \lambda_j \phi(X)^T \boldsymbol{\alpha}_j. \quad (7.90)$$

and therefore

$$\frac{1}{N} \phi(X) \phi(X)^T \phi(X) \phi(X)^T \boldsymbol{\alpha}_j = \lambda_j \phi(X) \phi(X)^T \boldsymbol{\alpha}_j. \quad (7.91)$$

- Let K be the **similarity (kernel) matrix**:

$$K \stackrel{\text{def}}{=} \phi(X) \phi(X)^T \in \mathbb{R}^{N \times N}. \quad (7.92)$$

- Then, (7.91) can be rewritten as

$$K^2 \boldsymbol{\alpha}_j = (N \lambda_j) K \boldsymbol{\alpha}_j. \quad (7.93)$$

- We can remove a factor of K from both sides of the above equation:^a

$$K \boldsymbol{\alpha}_j = \mu_j \boldsymbol{\alpha}_j, \quad \mu_j = N \lambda_j. \quad (7.94)$$

which implies that $\boldsymbol{\alpha}_j$ are **eigenvectors of K** .

- It should be noticed that $\boldsymbol{\alpha}_j$ are **analogues of $\mathbf{u}_j$, where $X = U \Sigma V^T$** .

^aThis will only affects the eigenvectors with zero eigenvalues, which will not be a principle component anyway.

Note: There is a **normalization condition** for the $\boldsymbol{\alpha}_j$ vectors:

$$\|\mathbf{v}_j\| = 1 \iff \|\boldsymbol{\alpha}_j\| = 1/\sqrt{\mu_j}.$$

$$\begin{aligned} \therefore \begin{cases} 1 &= \mathbf{v}_j^T \mathbf{v}_j = (\phi(X)^T \boldsymbol{\alpha}_j)^T \phi(X)^T \boldsymbol{\alpha}_j = \boldsymbol{\alpha}_j^T \phi(X) \phi(X)^T \boldsymbol{\alpha}_j & \Leftarrow (7.89) \\ &= \boldsymbol{\alpha}_j^T K \boldsymbol{\alpha}_j = \boldsymbol{\alpha}_j^T (\mu_j \boldsymbol{\alpha}_j) = \mu_j \|\boldsymbol{\alpha}_j\|^2 & \Leftarrow (7.94) \end{cases} \\ & \hspace{15em} (7.95) \end{aligned}$$

7.4.2. Computation of the kernel PCA

Remark 7.26. Let the eigenvalue-eigenvector pairs of the kernel matrix K be given as

$$K\alpha_j = \mu_j\alpha_j, \quad j = 1, 2, \dots, N; \quad \alpha_i^T \alpha_j = \delta_{ij}. \quad (7.96)$$

- Then, referring (7.78) derived for the standard PCA, we may conclude that the **k principal components for the kernel PCA** are

$$\mathcal{A}_k = [\sqrt{\mu_1}\alpha_1 | \sqrt{\mu_2}\alpha_2 | \dots | \sqrt{\mu_k}\alpha_k] \in \mathbb{R}^{N \times k}. \quad (7.97)$$

- It follows from (7.86), (7.89), and (7.95)-(7.96) that for a **new point $\mathbf{x}$** , its projection onto the principal components is:

$$\begin{aligned} z_j &= \phi(\mathbf{x})^T \mathbf{v}_j = \frac{1}{\sqrt{\mu_j}} \phi(\mathbf{x})^T \sum_{\ell=1}^N \alpha_{\ell j} \phi(\mathbf{x}^{(\ell)}) = \frac{1}{\sqrt{\mu_j}} \sum_{\ell=1}^N \alpha_{\ell j} \phi(\mathbf{x})^T \phi(\mathbf{x}^{(\ell)}) \\ &= \frac{1}{\sqrt{\mu_j}} \sum_{\ell=1}^N \alpha_{\ell j} \mathcal{K}(\mathbf{x}, \mathbf{x}^{(\ell)}) = \frac{1}{\sqrt{\mu_j}} \mathcal{K}(\mathbf{x}, X)^T \alpha_j. \end{aligned} \quad (7.98)$$

That is, due to (7.95) and (7.96), when $\mathbf{v}_j$ is expressed in terms of α_j , it must be scaled by $1/\sqrt{\mu_j}$.

Construction of the kernel matrix K

- The **kernel trick** is to avoid calculating the pairwise dot products of the transformed samples $\phi(\mathbf{x})$ explicitly by using a kernel function.
- For a selected kernel function $\mathcal{K}$,

$$K = \begin{bmatrix} \mathcal{K}(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & \mathcal{K}(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) & \dots & \mathcal{K}(\mathbf{x}^{(1)}, \mathbf{x}^{(N)}) \\ \mathcal{K}(\mathbf{x}^{(2)}, \mathbf{x}^{(1)}) & \mathcal{K}(\mathbf{x}^{(2)}, \mathbf{x}^{(2)}) & \dots & \mathcal{K}(\mathbf{x}^{(2)}, \mathbf{x}^{(N)}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{K}(\mathbf{x}^{(N)}, \mathbf{x}^{(1)}) & \mathcal{K}(\mathbf{x}^{(N)}, \mathbf{x}^{(2)}) & \dots & \mathcal{K}(\mathbf{x}^{(N)}, \mathbf{x}^{(N)}) \end{bmatrix} \in \mathbb{R}^{N \times N}. \quad (7.99)$$

where $\mathcal{K}$ is called the **kernel function**.^a

^aAs for nonlinear SVM, the most commonly used kernels are the polynomial kernel, the hyperbolic tangent (sigmoid) kernel, and the Gaussian Radial Basis Function (RBF) kernel. See (5.57)-(5.60), p. 118.

Normalizing the feature space

- In general, $\phi(\mathbf{x}^{(i)})$ may not be zero mean.
- Thus $K = \phi(X)\phi(X)^T$ would better be normalized before start finding its eigenvectors and eigenvalues.
- Centered features:

$$\tilde{\phi}(\mathbf{x}^{(i)}) = \phi(\mathbf{x}^{(i)}) - \frac{1}{N} \sum_{k=1}^N \phi(\mathbf{x}^{(k)}), \quad \forall i. \quad (7.100)$$

- The corresponding kernel is

$$\begin{aligned} \tilde{\mathcal{K}}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= \tilde{\phi}(\mathbf{x}^{(i)})^T \tilde{\phi}(\mathbf{x}^{(j)}) \\ &= \left(\phi(\mathbf{x}^{(i)}) - \frac{1}{N} \sum_{k=1}^N \phi(\mathbf{x}^{(k)}) \right)^T \left(\phi(\mathbf{x}^{(j)}) - \frac{1}{N} \sum_{k=1}^N \phi(\mathbf{x}^{(k)}) \right) \\ &= \mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) - \frac{1}{N} \sum_{k=1}^N \mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(k)}) - \frac{1}{N} \sum_{k=1}^N \mathcal{K}(\mathbf{x}^{(k)}, \mathbf{x}^{(j)}) \\ &\quad + \frac{1}{N^2} \sum_{k, \ell=1}^N \mathcal{K}(\mathbf{x}^{(k)}, \mathbf{x}^{(\ell)}). \end{aligned} \quad (7.101)$$

- In a matrix form

$$\tilde{K} = K - K\mathbf{1}_{1/N} - \mathbf{1}_{1/N}K + \mathbf{1}_{1/N}K\mathbf{1}_{1/N}, \quad (7.102)$$

where $\mathbf{1}_{1/N}$ is an $N \times N$ matrix where all entries are equal to $1/N$.

Summary 7.27. (Summary of the Kernel PCA).

- Pick a kernel function $\mathcal{K}$.
- For data $X \in \mathbb{R}^{N \times d}$, construct the kernel matrix

$$K = [\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})] \in \mathbb{R}^{N \times N}. \quad (7.103)$$

- Normalize the kernel matrix K :

$$\tilde{K} = K - K\mathbf{1}_{1/N} - \mathbf{1}_{1/N}K + \mathbf{1}_{1/N}K\mathbf{1}_{1/N}. \quad (7.104)$$

- Solve an eigenvalue problem:

$$\tilde{K}\boldsymbol{\alpha}_j = \mu_j\boldsymbol{\alpha}_j, \quad \boldsymbol{\alpha}_i^T\boldsymbol{\alpha}_j = \delta_{ij}. \quad (7.105)$$

- Then, the **k principal components for the kernel PCA** are

$$\mathcal{A}_k = [\mu_1\boldsymbol{\alpha}_1 | \mu_2\boldsymbol{\alpha}_2 | \cdots | \mu_k\boldsymbol{\alpha}_k] \in \mathbb{R}^{N \times k}, \quad k \leq N. \quad (7.106)$$

- For **a data point $\mathbf{x}$ (new or old)**, we can represent it as

$$z_j = \phi(\mathbf{x})^T \mathbf{v}_j = \phi(\mathbf{x})^T \sum_{\ell=1}^N \alpha_{\ell j} \phi(\mathbf{x}^{(\ell)}) = \sum_{\ell=1}^N \alpha_{\ell j} \mathcal{K}(\mathbf{x}, \mathbf{x}^{(\ell)}), \quad j = 1, 2, \dots, k. \quad (7.107)$$

Note: Formulas in (7.106)-(7.107) are alternatives of (7.97)-(7.98).

Properties of the KPCA

- With an appropriate choice of kernel function, the kernel PCA can give a good re-encoding of the data that lies along a nonlinear manifold.
- The kernel matrix is in $(N \times N)$ -dimensions, so the kernel PCA will have difficulties when we have lots of data points.

Exercises for Chapter 7

7.1. Read pp. 145–158, *Python Machine Learning, 3rd Ed.*, about the PCA.

- (a) Find the optimal number of components k^* which produces the best classification accuracy (for logistic regression), by experimenting the example code with `n_components = 1, 2, ..., 13`.
- (b) What is the corresponding **cumulative explained variance**?

7.2. Let $A \in \mathbb{R}^{m \times n}$. Prove that $\|A\|_2 = \sigma_1$, the largest singular value of A . **Hint:** Use the following

$$\frac{\|A\mathbf{v}_1\|_2}{\|\mathbf{v}_1\|_2} = \frac{\sigma_1 \|\mathbf{u}_1\|_2}{\|\mathbf{v}_1\|_2} = \sigma_1 \implies \|A\|_2 \geq \sigma_1$$

and arguments around Equations (7.34) and (7.35) for the opposite directional inequality.

7.3. Recall that the Frobenius matrix norm is defined by

$$\|A\|_F = \left(\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2 \right)^{1/2}, \quad A \in \mathbb{R}^{m \times n}.$$

Show that $\|A\|_F = (\sigma_1^2 + \dots + \sigma_k^2)^{1/2}$, where σ_j are nonzero singular values of A . **Hint:** You may use the norm-preserving property of orthogonal matrices. That is, if U is orthogonal, then $\|UB\|_2 = \|B\|_2$ and $\|UB\|_F = \|B\|_F$.

7.4. Prove Lemma 7.18. **Hint:** For (b), let $A\mathbf{v}_i = \lambda_i \mathbf{v}_i$, $i = 1, 2$, and $\lambda_1 \neq \lambda_2$. Then

$$(\lambda_1 \mathbf{v}_1) \cdot \mathbf{v}_2 = \underbrace{(A\mathbf{v}_1) \cdot \mathbf{v}_2 = \mathbf{v}_1 \cdot (A\mathbf{v}_2)}_{\because A \text{ is symmetric}} = \mathbf{v}_1 \cdot (\lambda_2 \mathbf{v}_2).$$

For (a), you may use a similar argument, but with the dot product being defined for complex values, i.e.,

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^T \bar{\mathbf{v}},$$

where $\bar{\mathbf{v}}$ is the complex conjugate of $\mathbf{v}$.

7.5. Use Matlab to generate a random matrix $A \in \mathbb{R}^{8 \times 6}$ with rank 4. For example,

```
A = randn(8,4);
A(:,5:6) = A(:,1:2)+A(:,3:4);
[Q,R] = qr(randn(6));
A = A*Q;
```

- (a) Print out A on your computer screen. Can you tell by looking if it has (numerical) rank 4?
- (b) Use Matlab's "svd" command to obtain the singular values of A . How many are "large?" How many are "tiny?" (You may use the command "format short e" to get a more accurate view of the singular values.)
- (c) Use Matlab's "rank" command to confirm that the numerical rank is 4.

- (d) Use the “rank” command with a small enough threshold that it returns the value 6. (Type “help rank” for information about how to do this.)
- 7.6. Verify (7.64). **Hint:** Use the quotient rule for $\frac{\partial \mathcal{J}(\mathbf{w})}{\partial \mathbf{w}}$ and equate the numerator to zero.
- 7.7. Try to understand the kernel PCA more deeply by experimenting pp. 175–188, *Python Machine Learning, 3rd Ed.*. Its implementation is slightly different from (but equivalent to) Summary 7.27.
- (a) Modify the code, following Summary 7.27, and test if it works as expected as in *Python Machine Learning, 3rd Ed.*.
- (b) The datasets considered are transformed via the Gaussian radial basis function (RBF) kernel only. What happens if you use the following kernels?

$$\begin{aligned}\mathcal{K}_1(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= (a_1 + b_1 \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)})^2 && \text{(polynomial of degree up to 2)} \\ \mathcal{K}_2(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= \tanh(a_2 + b_2 \mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}) && \text{(sigmoid)}\end{aligned}$$

Can you find a_i and b_i , $i = 1, 2$, appropriately?

CHAPTER 8

Cluster Analysis

Cluster analysis or **clustering** is **the task of finding groups of objects** such that the objects in a group will be similar (or related) to one another and different from (or unrelated to) the objects in other groups. It is a main task of **exploratory data mining**, and a common technique for **statistical data analysis**, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, bioinformatics, data compression, and computer graphics.

History. Cluster analysis was originated in **anthropology** by Driver and Kroeber in 1932 [16], introduced to **psychology** by Zubin in 1938 [84] and Robert Tryon in 1939 [76], and famously used by Cattell beginning in 1943 [11] for trait theory classification in **personality psychology**.

Contents of Chapter 8

8.1. Basics for Cluster Analysis	200
8.2. K-Means and K-Medoids Clustering	211
8.3. Hierarchical Clustering	224
8.4. DBSCAN: Density-based Clustering	231
8.5. Cluster Validation	236
8.6. Self-Organizing Maps	246
Exercises for Chapter 8	259

8.1. Basics for Cluster Analysis

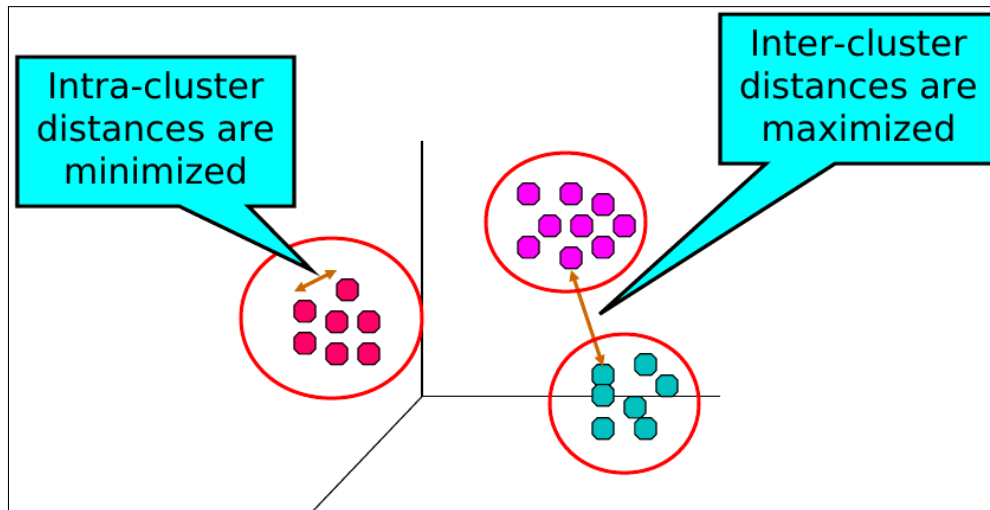


Figure 8.1: Intra-cluster distance vs. inter-cluster distance.

Applications of Cluster Analysis

- **Understanding**
 - group related documents or browsings
 - group genes/proteins that have similar functionality, or
 - group stocks with similar price fluctuations
- **Summarization**
 - reduce the size of large data sets

Not Cluster Analysis

- Supervised classification – Uses class label information
- Simple segmentation – Dividing students into different registration groups alphabetically, by last name
- Results of a query – Groupings are a result of an external specification

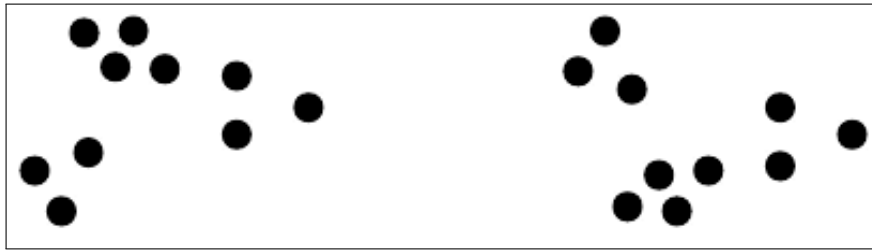
Clustering uses **only the data** (unsupervised learning):
to discover hidden structures in data

8.1.1. Quality of clustering

- A **good clustering** method will produce high quality clusters with
 - **high intra-class similarity**
 - **low inter-class similarity**
- The quality of a clustering result depends on both **the similarity measure** and **its implementation**
- The quality of a clustering method is also measured by its ability to discover some or all of the **hidden patterns**

Measuring the Quality of Clustering

- **Dissimilarity/Similarity/Proximity metric:** Similarity is expressed in terms of a distance function $d(i, j)$
- The definitions of **distance functions** are usually very different for interval-scaled, boolean, categorical, ordinal ratio, and vector variables.
- There is a separate “quality” function that measures the “goodness” of a cluster.
- *Weighted measures:* Weights should be associated with different variables based on applications and data semantics.
- It is hard to define “**similar enough**” or “**good enough**”
 - **the answer is typically highly subjective**

Notion of a Cluster can be AmbiguousFigure 8.2: **How many clusters?**

The answer could be:



Figure 8.3: Two clusters.



Figure 8.4: Four clusters.

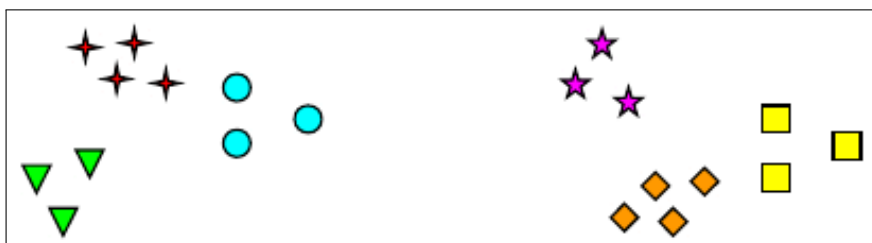


Figure 8.5: Six clusters.

Similarity and Dissimilarity Between Objects

- **Distances** are normally used to measure the similarity or dissimilarity between two data objects
- Some popular ones include: **Minkowski distance**

$$d(i, j) = \|\mathbf{x}_i - \mathbf{x}_j\|_p = (|x_{i1} - x_{j1}|^p + |x_{i2} - x_{j2}|^p \cdots |x_{id} - x_{jd}|^p)^{1/p}, \quad (8.1)$$

where $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^d$, two d -dimensional data objects.

- When $p = 1$, it is **Manhattan distance**
- When $p = 2$, it is **Euclidean distance**
- *Other Distances*: Also, one can use weighted distance, parametric Pearson product moment correlation, or other dissimilarity measures
- Various similarity measures have been studied for
 - Binary variables
 - Nominal variables & ordinal variables
 - Ratio-scaled variables
 - Variables of mixed types
 - Vector objects

8.1.2. Types of clusters

- Center-based clusters
- Contiguity/connectivity-based clusters
- Density-based clusters
- Conceptual clusters

Note: (Well-separated clusters). A cluster is a set of objects such that an object in a cluster is closer (more similar) to **every/some of points** in the cluster, than any points not in the cluster.

Center-based Clusters

- The center of a cluster is often
 - a **centroid**, the average of all the points in the cluster, or
 - a **medoid**, the most representative point of a cluster.
- *A cluster is a set of objects such that an object in a cluster is closer (more similar) to **the “center” of a cluster**, than to the center of any other clusters.*



Figure 8.6: Well-separated, 4 center-based clusters.

Contiguity-based Clusters

- Contiguous cluster (nearest neighbor or transitive)
- *A cluster is a set of points such that a point in a cluster is closer (or more similar) to **one or more other points** in the cluster, than to any points not in the cluster.*



Figure 8.7: 8 contiguous clusters.

Density-based Clusters

- *A cluster is a dense region of points, which is separated by low-density regions, from other regions of high density.*
- Used when the clusters are irregular or intertwined, and when noise and outliers are present.

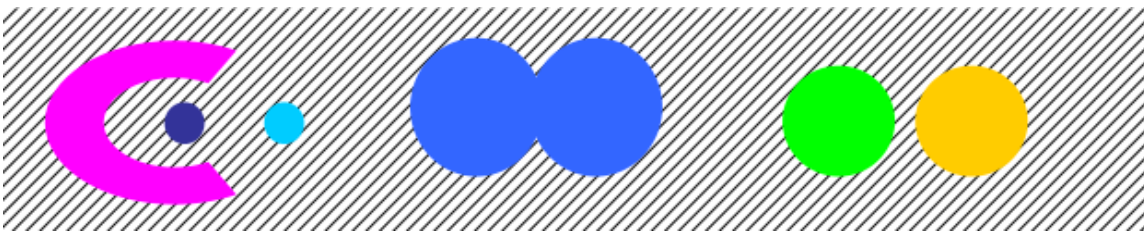


Figure 8.8: 6 density-based clusters.

Conceptual Clusters

- Points in a cluster share some general property.
 - Conceptual clusters are hard to detect, because they are often none of the center-based, contiguity-based, or density-based.
 - Points in the intersection of the circles belong to both.

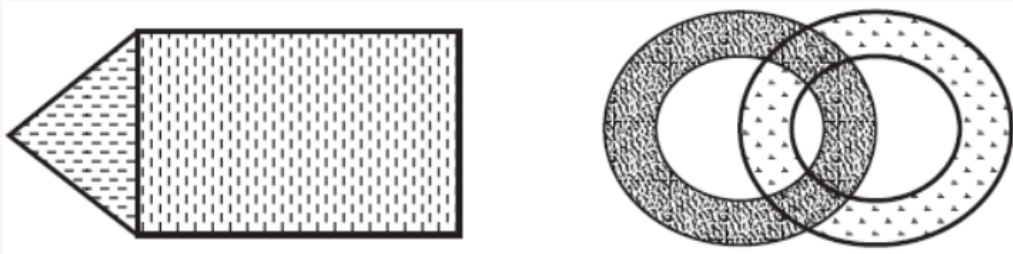


Figure 8.9: Conceptual clusters

Clusters Defined by an Objective Function

- Find clusters that minimize or maximize an objective function.
- Enumerate all possible ways of dividing the points into clusters and evaluate the “goodness” of each potential set of clusters by using the given objective function. (**NP-Hard**)
- Can have global or local objectives. *Typically*,
 - **Partitional clustering algorithms** have global objectives
 - **Hierarchical clustering algorithms** have local objectives

Computational Complexity Theory

Problem Types

- **P (Polynomial Time)**: Problems which are **solvable in polynomial time** (when running on a deterministic Turing machine^a).
- **NP (Non-deterministic Polynomial Time)**: Decision problems which can be **verified in polynomial time**.
- **NP-Hard**: These are at least as hard as the hardest problems in NP, **in both solution and verification**.
- **NP-Complete**: These are the problems which are both NP and NP-Hard.

^aA **Turing machine** is a theoretical machine that manipulates symbols on a strip of tape according to a table of rules. A deterministic Turing machine is a theoretical machine, used in thought experiments to examine the abilities and limitations of algorithms. In a deterministic Turing machine, the set of rules impose at most one action to be performed for any given situation. In a nondeterministic Turing machine, it may have a set of rules that prescribes more than one action for a given situation [13].

Problem Type	Verifiable in P-time	Solvable in P-time
P	Yes	Yes
NP	Yes	Yes or No
NP-Complete	Yes	Unknown
NP-Hard	Yes or No	Unknown

Question. **P = NP?** (P versus NP problem)

- This one is the most famous problem in computer science, and one of the most important outstanding questions in the mathematical sciences.
- In fact, the **Clay Institute** is offering one million dollars for a solution to the problem.
 - It's clear that P is a subset of NP.
 - The open question is whether or not NP problems have deterministic polynomial time solutions.

8.1.3. Types of clustering and Objective functions

- Partitional clustering
- Hierarchical clustering (agglomerative; divisive)
- Density-based clustering (DBSCAN)

Partitional Clustering

Divide data objects into **non-overlapping subsets** (clusters) such that each data object is in exactly one subset

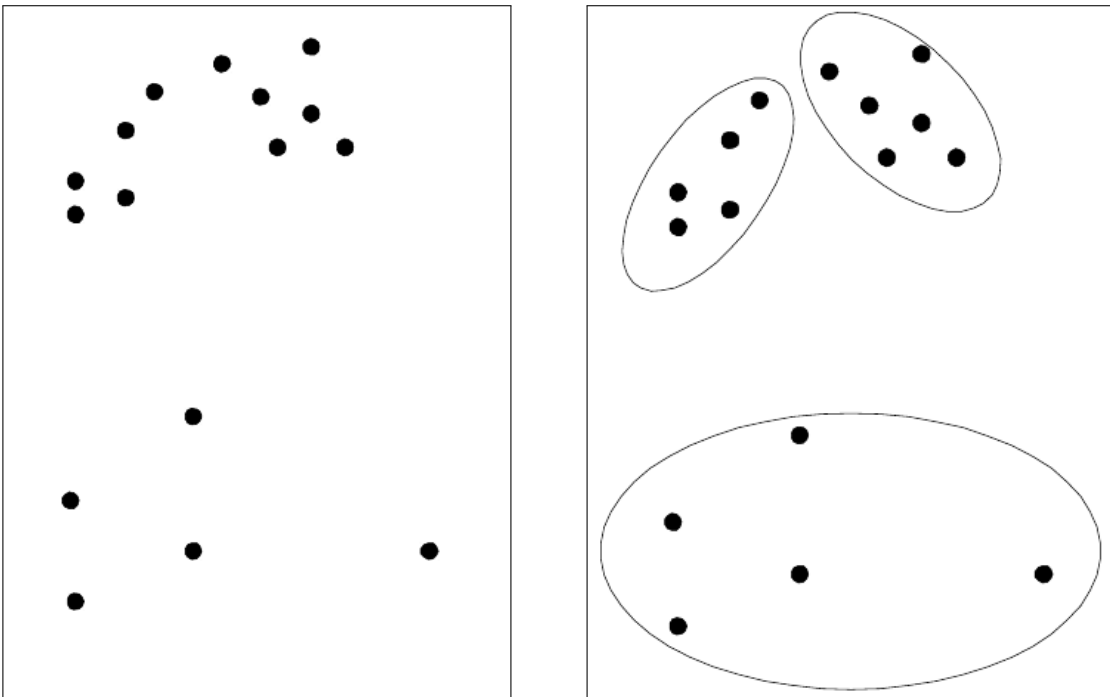


Figure 8.10: Original points & A partitional clustering

Examples are

- K-Means, Bisecting K-Means
- K-Medoids (PAM: partitioning around medoids)
- CLARA, CLARANS (Sampling-based PAMs)

Hierarchical Clustering

A set of **nested clusters**, organized as a hierarchical tree

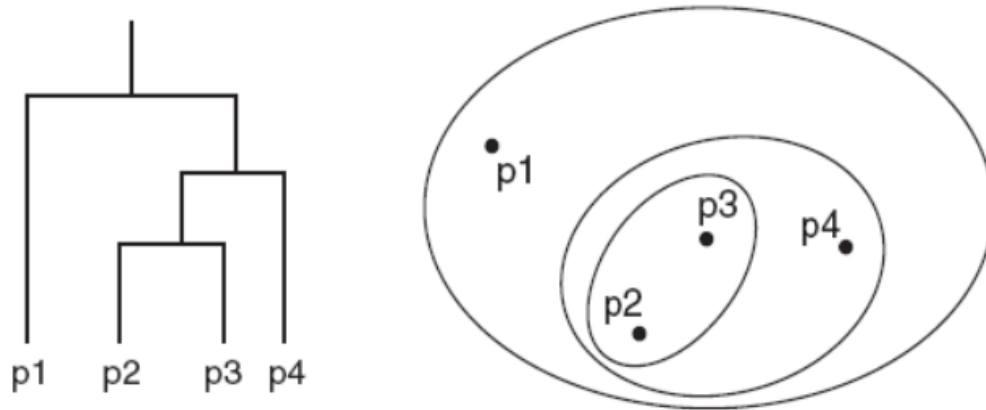


Figure 8.11: Dendrogram and Nested cluster diagram.

Other Distinctions Between Sets of Clusters

- **Exclusive (hard) vs. Non-exclusive (soft)**
 - In non-exclusive clusterings, points may belong to multiple clusters.
- **Fuzzy vs. Non-fuzzy**
 - In fuzzy clustering, a point belongs to every cluster with some **membership weight** between 0 and 1
 - Membership weights must sum to 1
 - **Probabilistic clustering** has similar characteristics
- **Partial vs. Complete**
 - In some cases, we only want to cluster some of the data
- **Homogeneous vs. Heterogeneous**
 - Cluster of widely different sizes, shapes, and densities

Objective Functions

Global objective function

- Typically used in partitional clustering
 - K-Means minimizes the **Sum of Squared Errors (SSE)**:

$$SSE = \sum_{i=1}^K \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|_2^2, \quad (8.2)$$

where $\mathbf{x}$ is a data point in cluster C_i and $\boldsymbol{\mu}_i$ is the center for cluster C_i as the mean of all points in the cluster.

- **Mixture models:** assume that the dataset is a “mixture” of a number of parametric statistical distributions (e.g., Gaussian mixture models).

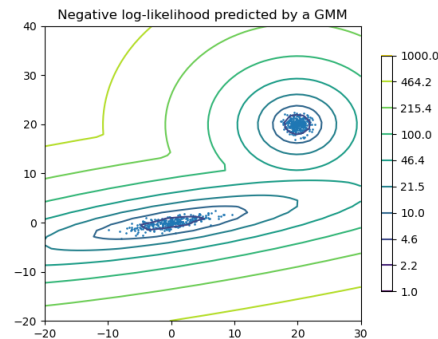


Figure 8.12: A two-component Gaussian mixture model: data points, and equi-probability surfaces of the model.

Local objective function

- **Hierarchical clustering** algorithms typically have local objectives
- **Density-based clustering** is based on local density estimates
- **Graph based approaches:** Graph partitioning and shared nearest neighbors

We will consider the objective functions when we talk about individual clustering algorithms.

8.2. K-Means and K-Medoids Clustering

- Given
 - X , a dataset of N objects
 - K , the number of clusters to form
- Organize the objects into K partitions ($K \leq N$), where each partition represents a cluster
- The clusters are formed to optimize an **objective partitioning criterion**:
 - Objects within a cluster are similar
 - Objects of different clusters are dissimilar

8.2.1. The (basic) K-Means clustering

- **Partitional clustering** approach
- Each cluster is associated with a **centroid** (mean)
- Each point is assigned to the cluster **with the closest centroid**
- Number of clusters, K , must be specified

Algorithm 8.1. Lloyd's algorithm (a.k.a. **Voronoi iteration**):
(Lloyd, 1957) [48]

1. Select K points as the initial centroids;
2. **repeat**
3. Form K clusters by assigning all points to the closest centroid;
4. Recompute the centroid of each cluster;
5. **until** (the centroids don't change)

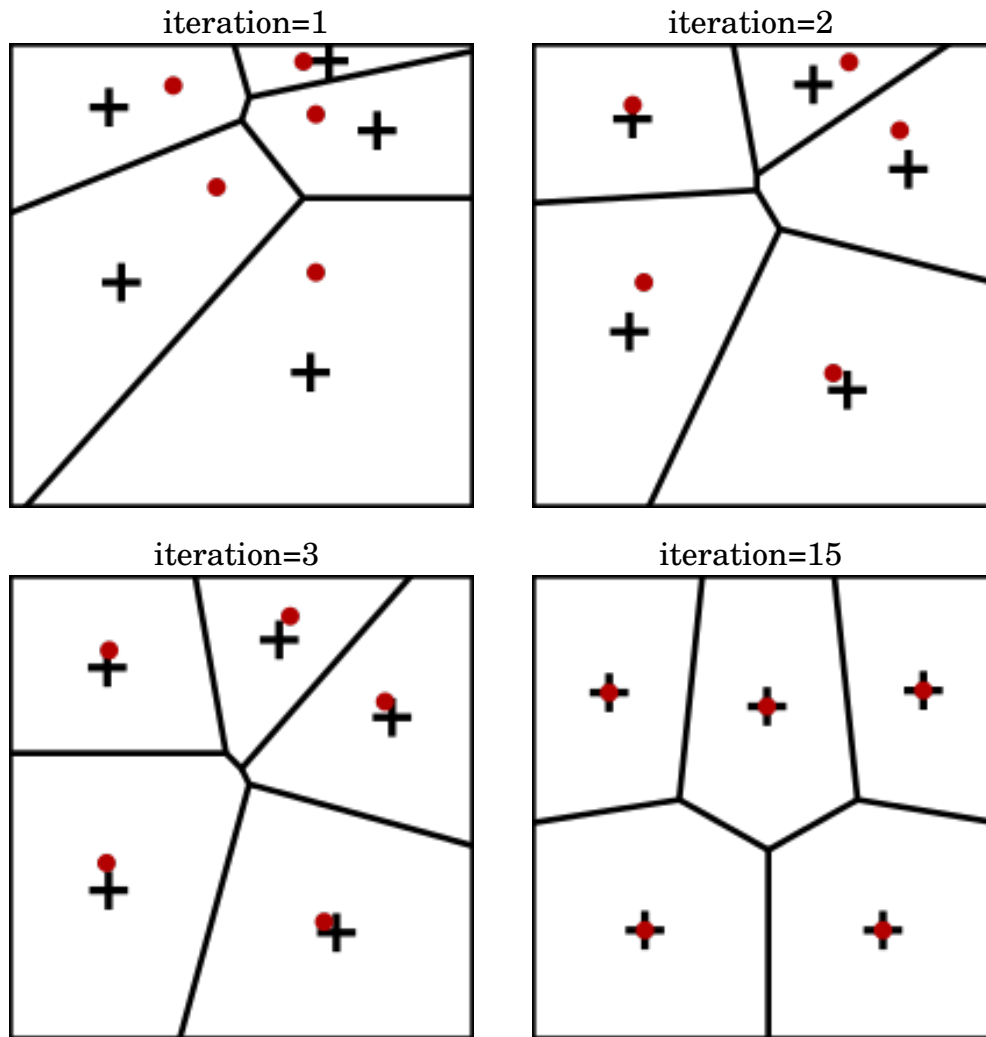


Figure 8.13: Lloyd's algorithm: The Voronoi diagrams, the given centroids ($\bullet$), and the updated centroids ($\times$), for iteration = 1, 2, 3, and 15.

The K-Means Clustering – Details

- Initial centroids are often chosen randomly.
 - Clusters produced vary from one run to another.
- The centroid is (typically) the mean of the points in the cluster.
- “Closeness” is measured by **Euclidean distance**, cosine similarity, correlation, etc..
- The K-Means will converge typically in the first few iterations.
 - Often the stopping condition is changed to “**until** (relatively few points change clusters)” or some measure of clustering doesn’t change.
- Complexity is $\mathcal{O}(N * d * K * I)$, where
 - N : the number of points
 - d : the number of attributes
 - K : the number of clusters
 - I : the number of iterations

Evaluating the K-Means Clusters

- Most common measure is Sum of Squared Error (SSE):

$$SSE = \sum_{i=1}^K \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|_2^2, \quad (8.3)$$

where $\mathbf{x}$ is a data point in cluster C_i and $\boldsymbol{\mu}_i$ is the center for cluster C_i .

- **Multiple runs:** Given sets of clusters, we can choose the one with the smallest error.
- One easy way to reduce SSE is to increase K , the number of clusters.
 - A good clustering with smaller K can have a lower SSE than a poor clustering with higher K .

The K-Means is **heuristic** to minimize the SSE.

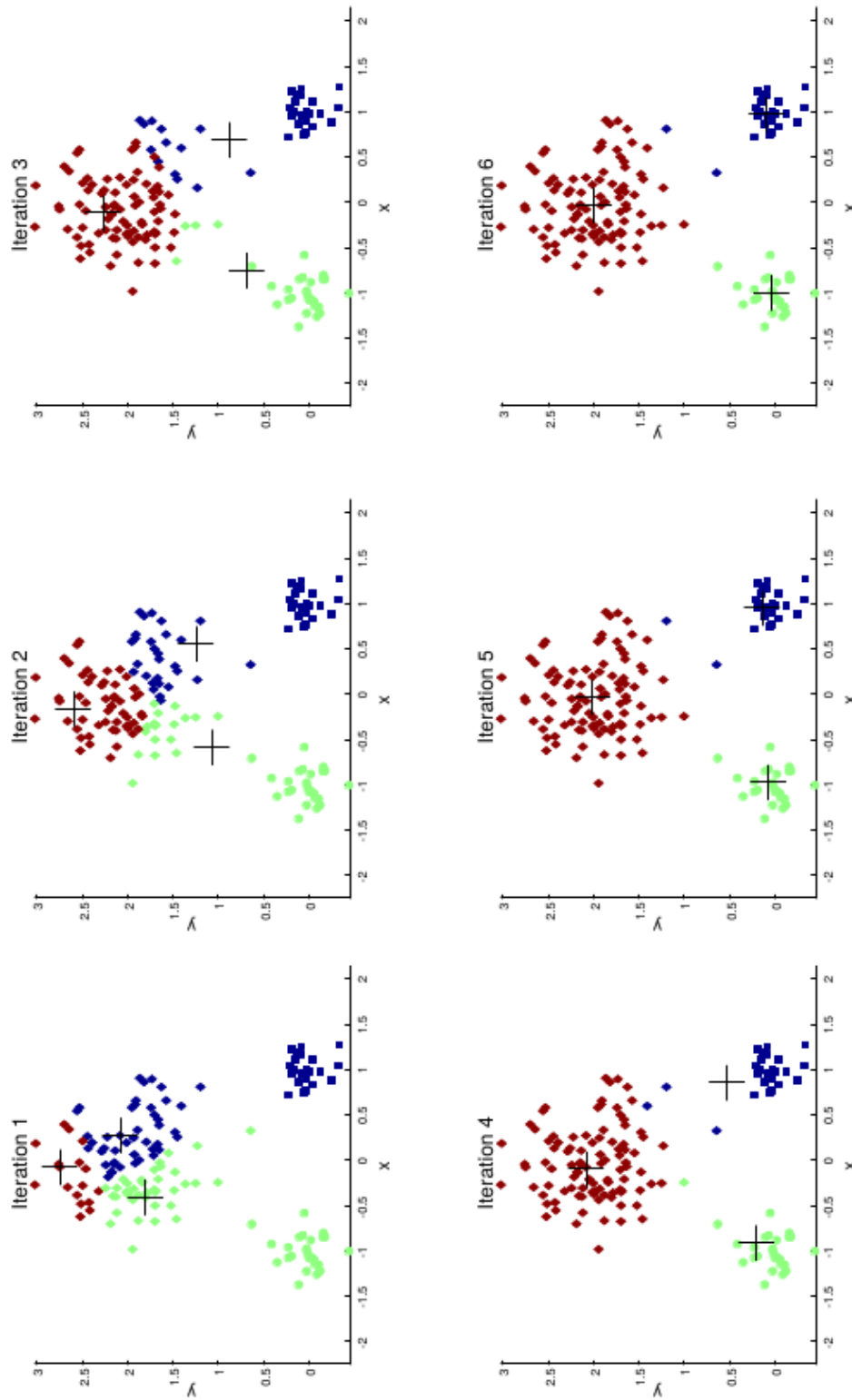


Figure 8.14: K-Means clustering example.

Problems with Selecting Initial Points

The chance of **selecting one centroid from each cluster** is small.

- Chance is relatively small when K is large
- If clusters are the same size, n , then

$$P = \frac{\text{\# of ways to select a centroid from each cluster}}{\text{\# of ways to select } K \text{ centroids}}$$

$$= \frac{K!n^K}{(Kn)^K} = \frac{K!}{K^K}.$$

- For example, if $K = 5$ or 10 , then probability is:

$$5!/5^5 = 0.0384, \quad 10!/10^{10} = 0.00036.$$

- Sometimes the initial centroids will readjust themselves in “right” way, and sometimes they don’t.

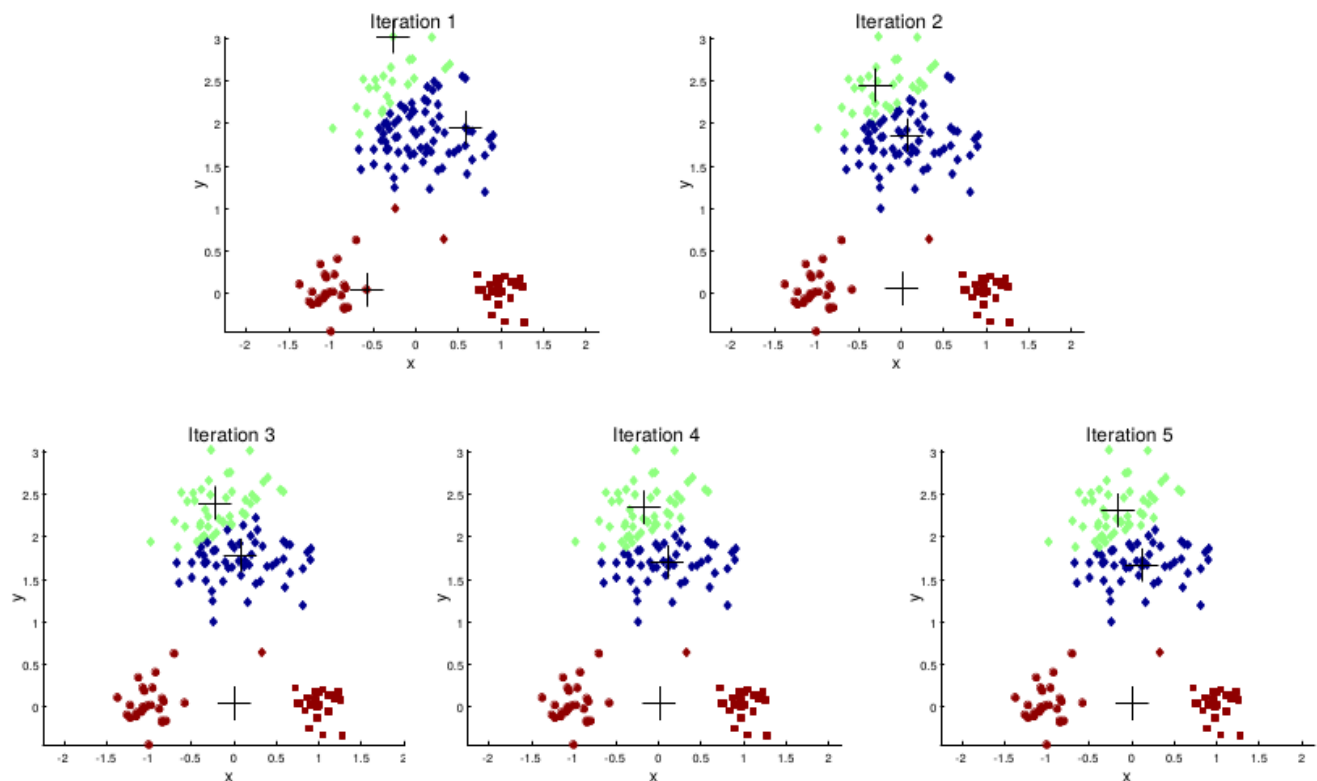


Figure 8.15: Importance of choosing initial centroids.

Solutions to Initial Centroids Problem

- **Multiple runs**
 - Helps, but probability is not on your side
- **Sample and use hierarchical clustering** to determine initial centroids
- Select **more than** K initial centroids and then, among these initial centroids
 - Select most widely separated
- **Post-processing**
- **Bisecting K-Means**
 - Not as susceptible to initialization issues

Pre-processing and Post-processing

- **Pre-processing**
 - Normalize the data
 - Eliminate outliers
- **Post-processing**
 - **Eliminate** small clusters that may represent outliers
 - **Split** “loose” clusters, i.e., clusters with relatively high SSE
 - **Merge** clusters that are “close” and that have relatively low SSE
 - * Can use these steps during the clustering process – ISODATA

8.2.2. Bisecting K-Means algorithm

A variant of the K-Means that can produce a partitional or a hierarchical clustering

1. **Initialize** (a list of clusters), containing all points.
2. **Repeat**
 - (a) Select a cluster from the list of clusters
 - (b) **for** $i = 1$ to $iter_runs$ **do**
 Bisect the selected cluster using the basic K-Means
end for
 - (c) Add the **two clusters from the bisection with the lowest SSE** to the list of clusters.
- until** (the list of clusters contains K clusters)

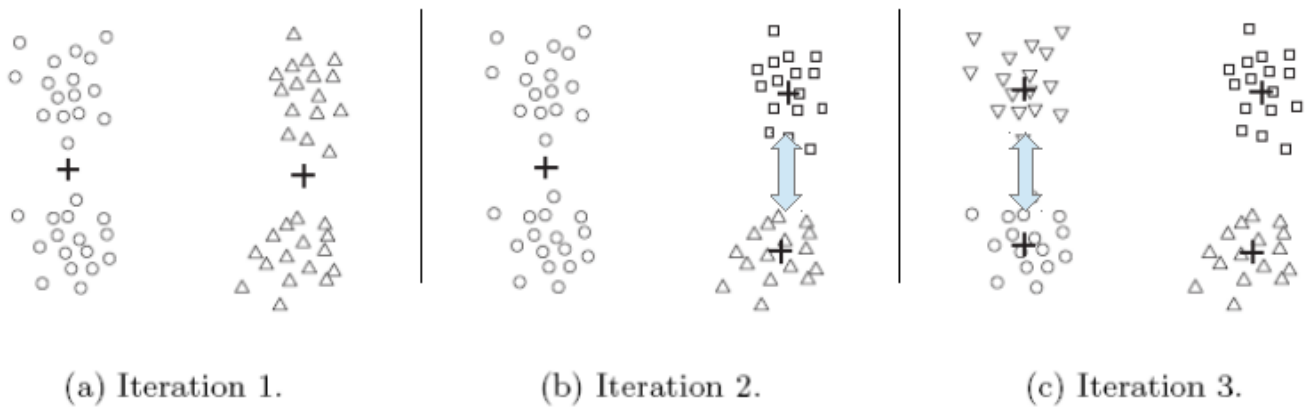


Figure 8.16: Bisecting K-Means algorithm, with $K = 4$.

Note: The bisecting K-Means algorithm is not as susceptible to initialization issues as the basic K-Means clustering.

Limitations of K-Means Algorithms

- The K-Means have problems when clusters are of differing
– **sizes, densities, and non-globular shapes**
- The K-Means have problems when the data contains **outliers**

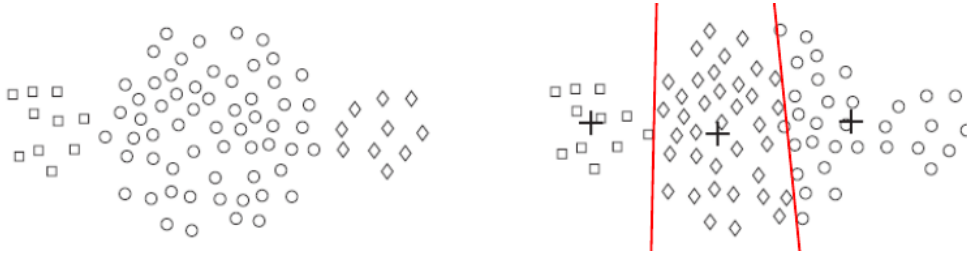


Figure 8.17: The K-Means with 3 clusters of different sizes.

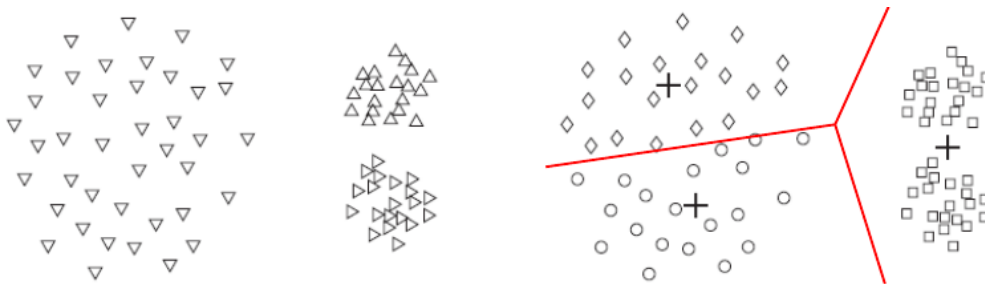


Figure 8.18: The K-Means with 3 clusters of different densities.

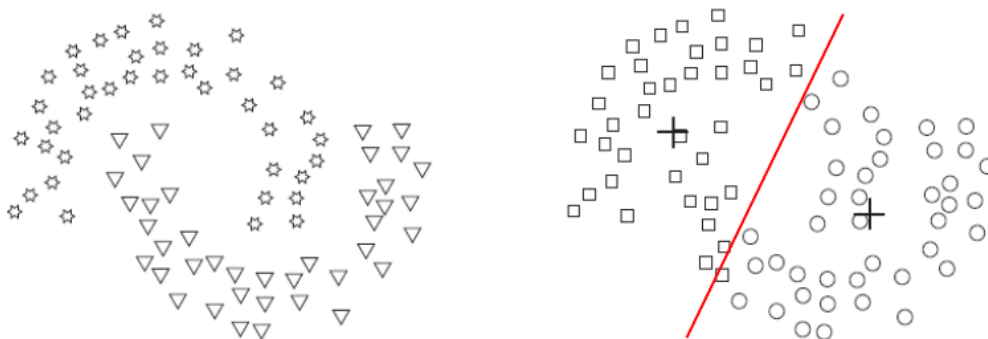


Figure 8.19: The K-Means with 2 non-globular clusters.

Overcoming K-Means Limitations

- Use a larger number of clusters
- **Several clusters represent a true cluster**

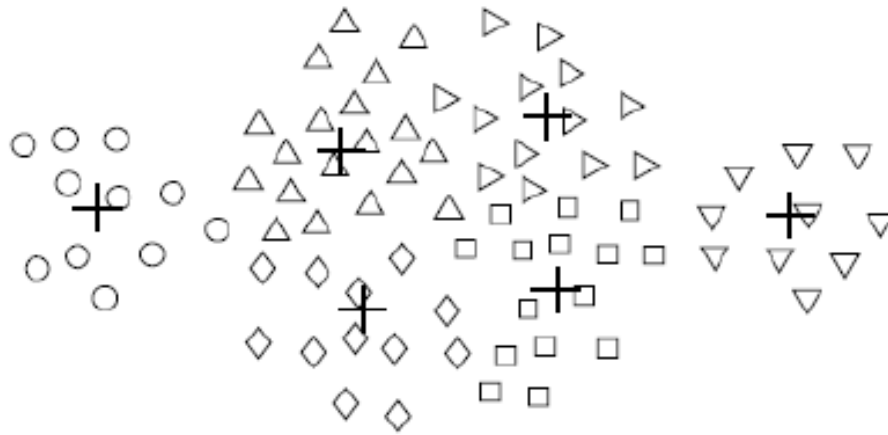


Figure 8.20: Unequal-sizes.

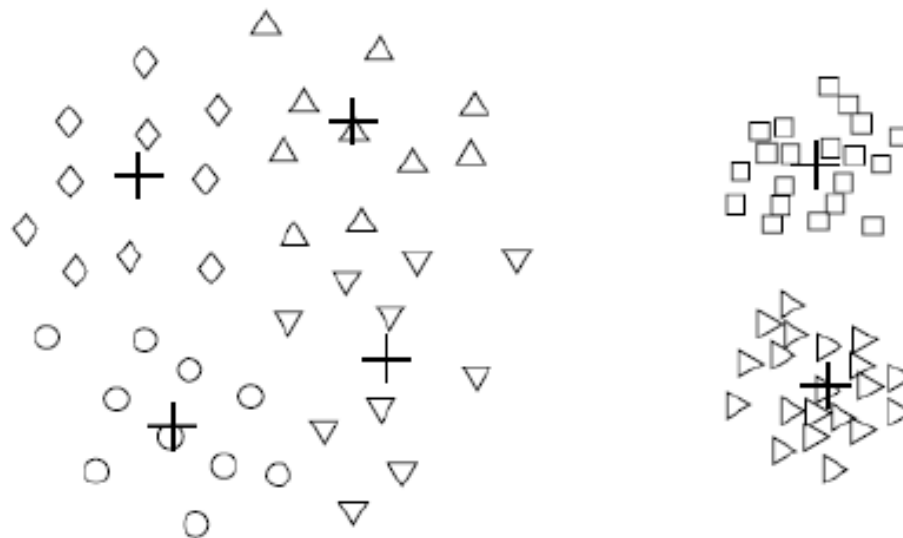


Figure 8.21: Unequal-densities.

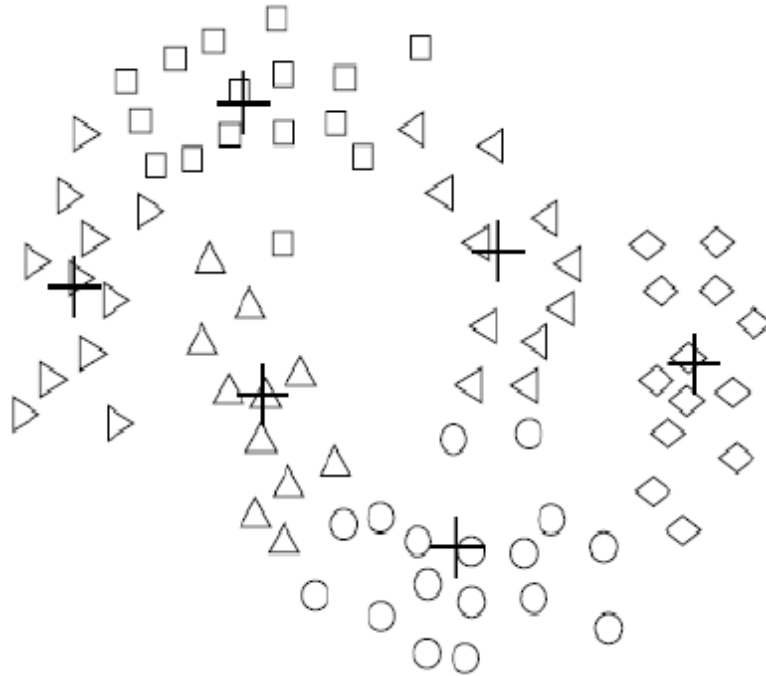


Figure 8.22: Non-spherical shapes.

Overcoming the K-Means Outlier Problem

- The K-Means algorithms are **sensitive to outliers**.
 - Since an object with an extremely large value may substantially distort the distribution of the data.
- **Solutions:**
 - (a) Instead of taking the mean value of the object in a cluster as a reference point, **medoids** can be used, which is the **most centrally located object** in a cluster.
 - (b) **Develop an effective outlier removal algorithm.** We will do it as a project which combines clustering and supervised learning for classification.

8.2.3. The K-Medoids algorithm

The **K-Medoids algorithm** (or **PAM algorithm**) is a clustering algorithm similar to the K-Means algorithm. Both the K-Means and K-Medoids algorithms are **partitional** (breaking the dataset up into groups) and both attempt to minimize the distance between points labeled to be in a cluster and a point designated as the center of that cluster. The K-Medoids chooses **data points as centers (medoids)** and can be used **with arbitrary distances**, while the K-Means only minimizes the squared Euclidean distances from cluster means. The PAM method was proposed by (Kaufman & Rousseeuw, 1987) [37] for the work **with L^1 -norm and other distances**.

- Find representative objects, called **medoids**, in clusters.
- The **PAM** (partitioning around medoids) starts from an initial set of medoids and **iteratively replaces** one of the medoids by one of the non-medoids **if it improves the total distance** of the resulting clustering.
- The PAM works effectively for **small datasets**, but **does not scale well** for large data sets.
- **CLARA** (Clustering LARge Applications): sampling-based method (Kaufmann & Rousseeuw, 1990) [38]
- **CLARANS**: CLARA with randomized search (Ng & Han, 1994) [54]

PAM (Partitioning Around Medoids): Use real objects to represent the clusters (called medoids).

Initialization: select K representative objects;

Associate each data point to the closest medoid;

while (the cost of the configuration decreases) :

```

[ For each medoid  $m$  and each non-medoid data point  $o$  :
    swap  $m$  and  $o$ ;
    associate each data point to the closest medoid;
    recompute the cost (sum of distances of points to their medoid);
    If the total cost of the configuration increased, undo the swap;
```

Pros and cons of the PAM

- The PAM is **more robust** than the K-Means in the presence of noise and outliers, because a medoid is less influenced by outliers or other extreme values than a mean.
 - The PAM works efficiently for small datasets but does **not scale well** for large data sets.
 - The run-time complexity of the PAM is $\mathcal{O}(K(N - K)^2)$ for each iteration, where N is the number of data points and K is the number of clusters.
- ⇒ **CLARA** (Clustering LARge Applications): sampling-based method (Kaufmann & Rousseeuw, 1990) [38]

The PAM finds the best K-medoids among a given data, and the CLARA finds the best K-medoids among the selected samples.

8.2.4. CLARA and CLARANS

CLARA (Clustering LARge Applications)

- Sampling-based PAM (Kaufmann & Rousseeuw, 1990) [38]
 - It draws **multiple samples** of the dataset, applies the PAM on each sample, and gives the best clustering as the output.
- ⊕ **Strength:** deals with larger data sets than the PAM.
- ⊖ **Weakness:**
- Efficiency depends on the sample size.
 - A good clustering based on samples will not necessarily represent a good clustering of the whole data set, if the sample is biased.
- **Medoids are chosen from the sample:**
 - ⊖ The algorithm cannot find the best solution if one of the best K-Medoids is not among the selected samples.

CLARANS (“Randomized” CLARA)

- **CLARANS** (CLARA with Randomized Search) (Ng & Han; 1994,2002) [54, 55]
 - The CLARANS draws **sample of neighbors dynamically**.
 - CLARANS draws **a sample of neighbors** in each step of a search, while CLARA draws a sample of nodes at the beginning of a search.
 - The clustering process can be presented as **searching a graph** where every node is a potential solution, that is, **a set of K medoids**.
 - If a local optimum is found, **the CLARANS starts with new randomly selected node** in **search for a new local optimum**.
 - Finds several local optimums and output the clustering with the **best local optimum**.
- ⊕ It is more efficient and scalable than both the PAM and the CLARA; handles outliers.
- ⊕ Focusing techniques and **spatial access structures** may further improve its performance; see (Ng & Han, 2002) [55] and (Schubert & Rousseeuw, 2018) [71].
- ⊖ Yet, the computational complexity of the CLARANS is $\mathcal{O}(N^2)$, where N is the number of objects.
- ⊖ The clustering quality depends on the sampling method.

8.3. Hierarchical Clustering

8.3.1. Basics of AGNES and DIANA

Hierarchical clustering can be divided into two main types: **agglomerative** and **divisive**.

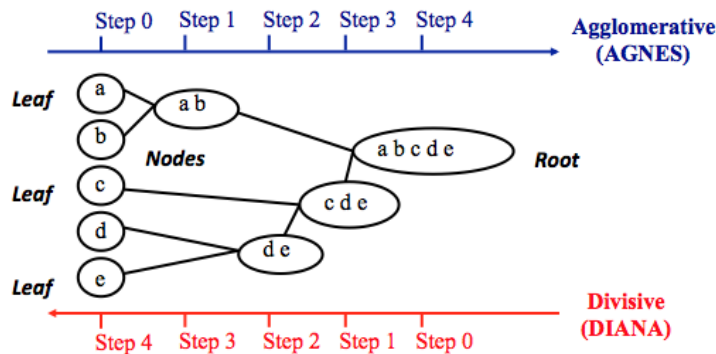


Figure 8.23: AGNES and DIANA

Agglomerative hierarchical clustering

(a.k.a. **AGNES**: Agglomerative Nesting). It works in a **bottom-up manner**.

- Each object is initially considered as its own singleton cluster (leaf).
- At each iteration, the **two closest clusters** are merged into a new bigger cluster (nodes).
- This procedure is iterated **until** all points are merged into a single cluster (root).
- The result is a tree which can be plotted as a dendrogram.

Divisive hierarchical clustering

(a.k.a. **DIANA**: Divisive Analysis). It works in a **top-down manner**; the algorithm is an inverse order of the AGNES.

- It begins with the root, where all objects are included in a single cluster.
- **Repeat**: the **most heterogeneous cluster** is divided into two.
- **Until**: all objects are in their own cluster.

Note: Agglomerative clustering is good at identifying small clusters, while divisive hierarchical clustering is good for large clusters.

Complexity

- The optimum cost is $\mathcal{O}(N^2)$, because it uses the proximity matrix. (N is the number of points)
- In practice, $\mathcal{O}(N^3)$ in many cases.
 - There are $\mathcal{O}(N)$ steps and at each step the proximity matrix of size $\mathcal{O}(N^2)$ must be updated and searched.
 - Complexity can be reduced to $\mathcal{O}(N^2 \log(N))$ for some approaches.

Limitations

- **Greedy:** Once a decision is made to combine two clusters, it cannot be undone.
- **No global objective function** is directly minimized.
- Most algorithms have problems with one or more of:
 - Sensitivity to noise and outliers
 - Difficulty handling different sized clusters and non-convex shapes
 - Chaining, breaking large clusters

Hierarchical Clustering vs. K-Means

- Recall that K-Means or K-Medoids requires
 - The number of clusters K
 - An initial assignment of data to clusters
 - A distance measure between data $d(\mathbf{x}_i, \mathbf{x}_j)$
- Hierarchical clustering requires only a **similarity measure** between groups/clusters of data points.

8.3.2. AGNES: Agglomerative clustering

Quesiton. How do we measure the similarity (or dissimilarity) between two groups of observations?

A number of different **cluster agglomeration methods** (i.e, linkage methods) have been developed to answer to the question. The most popular choices are:

- Single linkage
- Complete linkage
- Group linkage
- Centroid linkage
- Ward's minimum variance

1. **Single linkage**: the similarity of the closest pair

$$d_{SL}(G, H) = \min_{i \in G, j \in H} d(i, j). \quad (8.4)$$

- Single linkage can produce “**chaining**”, where a sequence of close observations in different groups cause early merges of those groups
- It tends to produce **long “loose” clusters**.

2. **Complete linkage**: the similarity of the furthest pair

$$d_{CL}(G, H) = \max_{i \in G, j \in H} d(i, j). \quad (8.5)$$

- Complete linkage has the opposite problem; it might not merge close groups because of **outlier members** that are far apart.
- It tends to produce **more compact clusters**.

3. **Group average**: the average similarity between groups

$$d_{GA}(G, H) = \frac{1}{N_G N_H} \sum_{i \in G} \sum_{j \in H} d(i, j). \quad (8.6)$$

- Group average represents a **natural compromise**, but depends on the scale of the similarities. Applying a monotone transformation to the similarities can change the results.
4. **Centroid linkage**: It computes the dissimilarity between the centroid for group G (a mean vector of length d variables) and the centroid for group H .
5. **Ward's minimum variance**: It minimizes the total within-cluster variance. More precisely, at each step, the method finds **the pair of clusters** that leads to **minimum increase in total within-cluster variance** after merging. It uses the squared error (as an objective function).

Interpretable Visualization of AGNES

- Each level of the resulting tree is a segmentation of data
- The algorithm results in a **sequence** of groupings
- It is **up to the user to choose a natural clustering** from this sequence

Dendrogram

- Agglomerative clustering is monotonic
 - The **similarity** between merged clusters is **monotone decreasing** with the level of the merge.
- **Dendrogram**: Plot each merge at the dissimilarity between the two merged groups
 - Provides an **interpretable visualization** of the algorithm and data
 - **Useful summarization tool**, part of why hierarchical clustering is popular

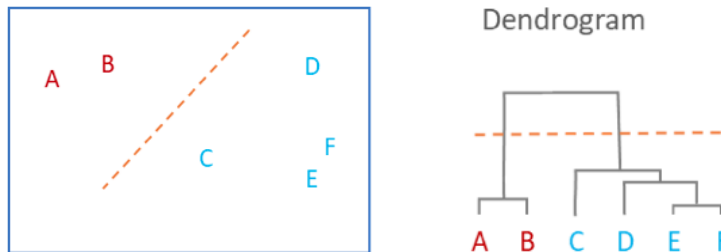


Figure 8.24: Six observations and a dendrogram showing their hierarchical clustering.

Remark 8.2.

- The height of the dendrogram indicates **the order** in which the clusters were joined; it reflects **the distance between the clusters**.
- The greater the difference in height, the more **dissimilarity**.
- Observations are allocated to clusters by drawing a **horizontal line** through the dendrogram. Observations that are joined together below the line are in the same clusters.

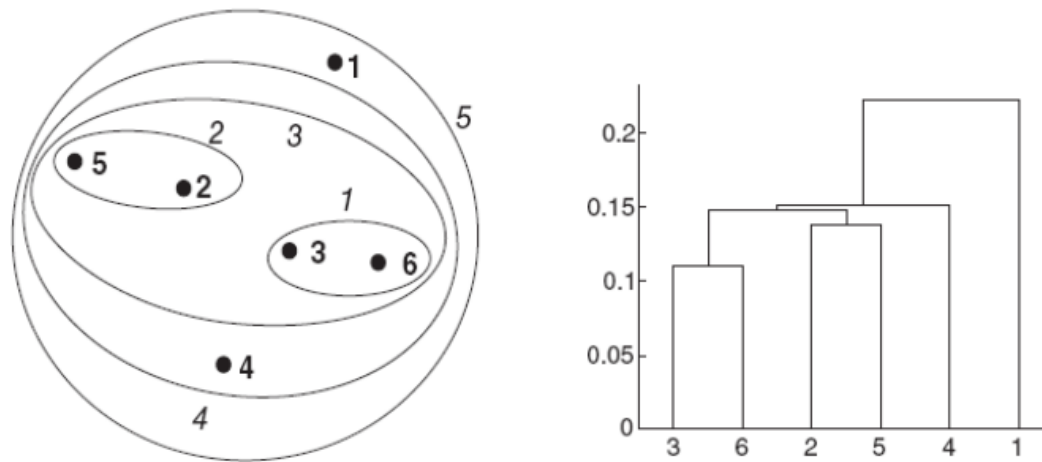
Single Link

Figure 8.25: Single link clustering of six points.

- **Pros:** Non-spherical, non-convex clusters
- **Cons:** Chaining

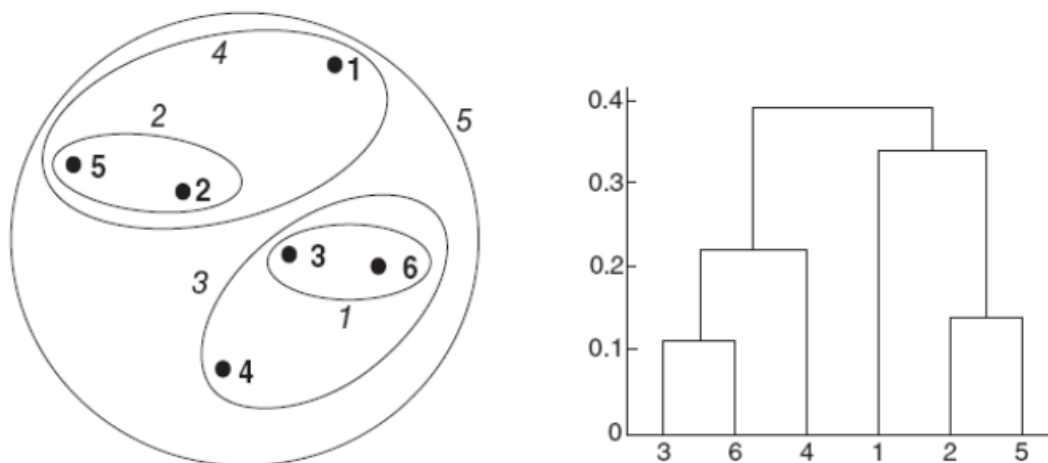
Complete Link

Figure 8.26: Complete link clustering of six points.

- **Pros:** more robust against noise (no chaining)
- **Cons:** Tends to break large clusters; biased towards globular clusters

Average Link

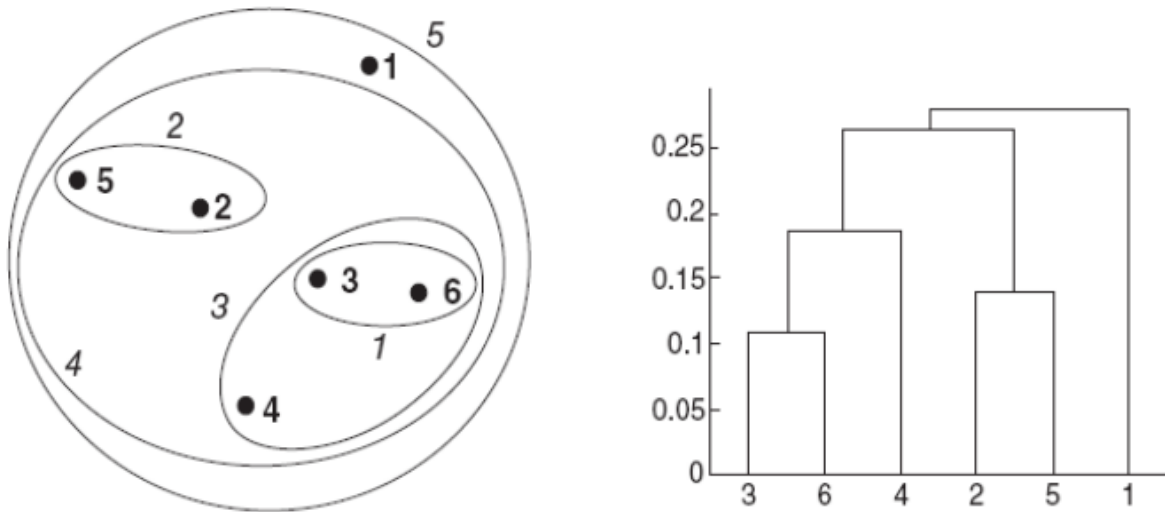


Figure 8.27: Average link clustering of six points.

- Compromise between single and complete links

Ward's Minimum Variance Method

- Similarity of two clusters is based on the increase in *squared error* when two clusters are merged
- **Less susceptible to noise and outliers**
- Biased towards *globular clusters*.
- Hierarchical analogue of the K-Means; it can be used **to initialize the K-Means**. (Note that the K-Means works with a global objective function.)

8.4. DBSCAN: Density-based Clustering

In **density-based clustering**:

- Clusters are defined as **areas of higher density** than the remainder of the data set. (**core points**)
- Objects in sparse areas are usually considered to be **noise** and **border points**.
- The most popular density-based clustering method is
 - **DBSCAN^a** (Ester, Kriegel, Sander, & Xu, 1996) [18].

^aDensity-Based Spatial Clustering of Applications with Noise (DBSCAN).

DBSCAN

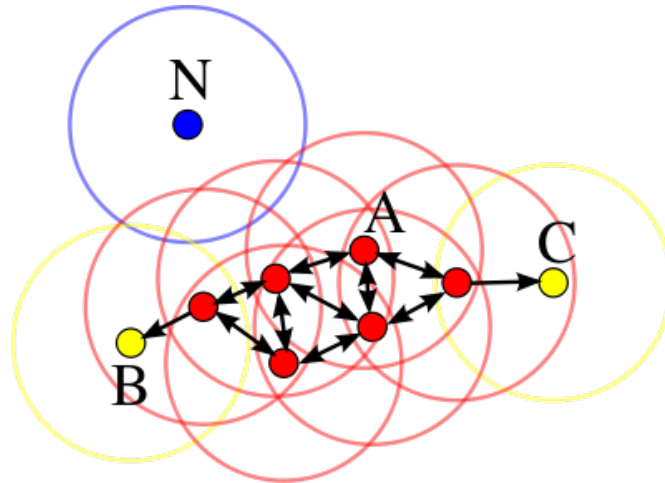
- Given a set of points, it groups points that are closely packed together (points with many nearby neighbors),
 - marking as **outliers points** that lie alone in low-density regions (whose nearest neighbors are too far away).
- It is **one of the most common clustering algorithms** and also most cited in scientific literature. (Citation #: 27905, as of Apr. 04, 2023)
- In 2014, the algorithm was awarded **the test of time award^a** at the leading data mining conference,
KDD 2014: <https://www.kdd.org/kdd2014/>.

^aThe test of time award is an award given to algorithms which have received substantial attention in theory and practice.

Preliminary for DBSCAN

- Consider a set of points in some space to be clustered.
- Let ε be a parameter specifying the **radius** of a neighborhood with respect to some point.
- For the purpose of DBSCAN clustering, the points are classified as **core points, reachable points, and outliers**, as follows:
 - A point p is a **core point** if at least minPts points are within distance ε of it (including p itself).
 - A point q is **directly reachable from p** if point q is within distance ε from the core point p .
(Points are only said to be directly reachable from core points.)
 - A point q is **reachable from p** if there is a path $p_1, \dots, p_n$ with $p_1 = p$ and $p_n = q$, where each p_{i+1} is directly reachable from p_i .
(Note that this implies that all points on the path must be core points, with the possible exception of q .)
 - All points not reachable from any other points are **outliers** or **noise points**.
- Now, a core point forms a cluster together with all points (core or non-core) that are **reachable** from it.
- Each cluster contains **at least one core point**; **non-core points** can be part of a cluster, but they form its “**edge**”, since they cannot be used to reach more points.

A reachable point is also called a **border point**.

Illustration of the DBSCANFigure 8.28: Illustration of the DBSCAN, with $\text{minPts} = 4$.

- Point *A* and 5 other red points are **core points**. They are all reachable from one another, so they form a **single cluster**.
- Points *B* and *C* are not core points, but are **reachable from A** (via other core points) and **thus belong to the cluster** as well.
- Point *N* is a **noise point** that is neither a core point nor directly-reachable.

Note: Reachability is not a symmetric relation since, by definition, no point may be reachable from a non-core point, regardless of distance (so a non-core point may be reachable, but nothing can be reached from it).

Definition 8.3. Two points p and q are **density-connected** if there is a point c such that both p and q are reachable from c . Density-connectedness is symmetric.

A DBSCAN cluster satisfies two properties:

1. All points within the cluster are mutually density-connected.
2. If a point is density-reachable from any point of the cluster, then it is part of the cluster as well.

DBSCAN: Pseudocode

```

1  DBSCAN(D, eps, MinPts)
2      C=0                                # Cluster counter
3      for each unvisited point P in dataset D
4          mark P as visited
5          NP = regionQuery(P, eps)        # Find neighbors of P
6          if size(NP) < MinPts
7              mark P as NOISE
8          else
9              C = C + 1
10             expandCluster(P, NP, C, eps, MinPts)
11
12 expandCluster(P, NP, C, eps, MinPts)
13     add P to cluster C
14     for each point Q in NP
15         if Q is not visited
16             mark Q as visited
17             NQ = regionQuery(Q, eps)
18             if size(NQ) >= MinPts
19                 NP = NP joined with NQ
20     if Q is not yet member of any cluster
21         add Q to cluster C

```

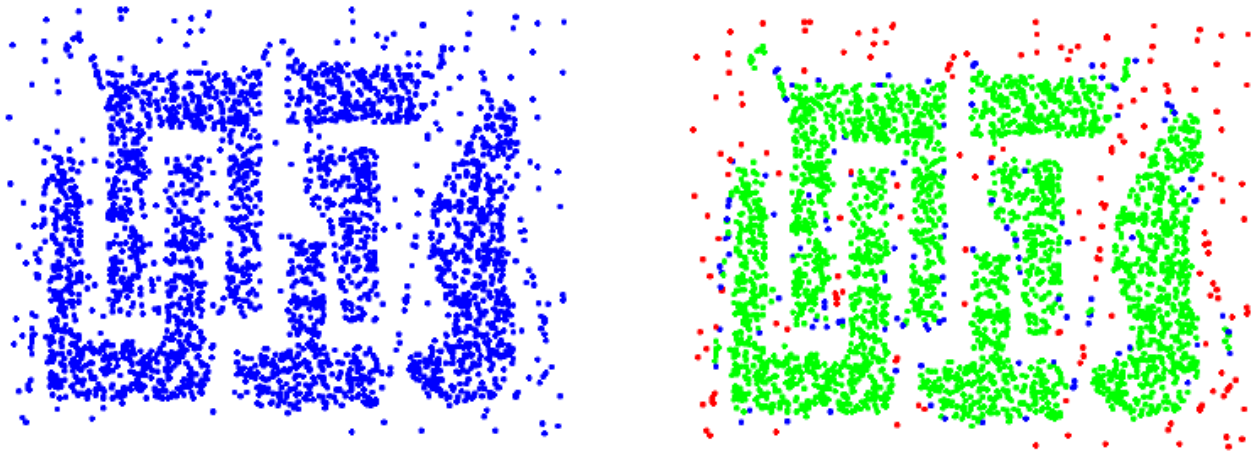


Figure 8.29: Original points (left) and point types of the DBSCAN clustering with $\text{eps}=10$ and $\text{MinPts}=4$ (right): **core** (green), **border** (blue), and **noise** (red).

Properties of DBSCAN Clustering

- Resistant to Noise
- Can handle clusters of different shapes and sizes
- Eps and MinPts depend on each other and **can be hard to specify**

When the DBSCAN does NOT work well

- Varying densities
- High-dimensional data

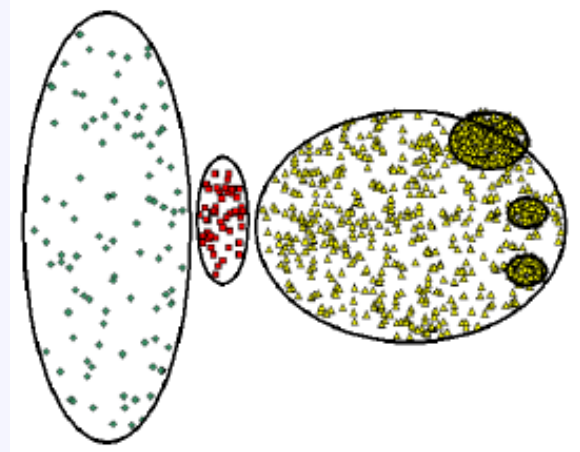
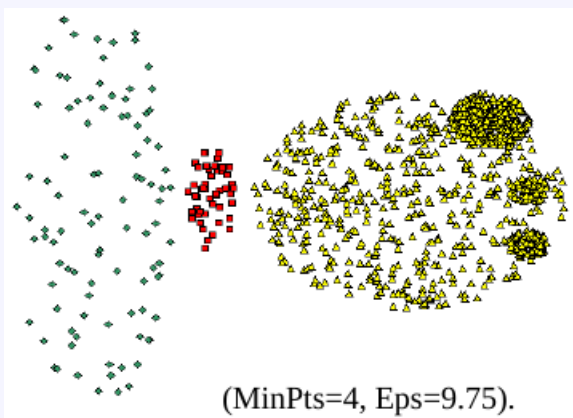
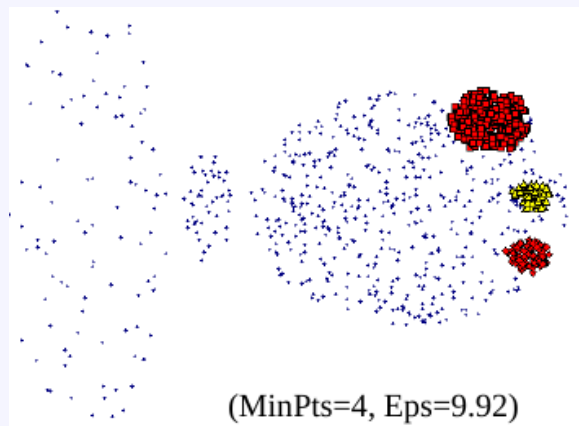


Figure 8.30: Original points.



(MinPts=4, Eps=9.75).



(MinPts=4, Eps=9.92)

Figure 8.31: The DBSCAN clustering. For both cases, it results in 3 clusters.

8.5. Cluster Validation

8.5.1. Basics of cluster validation

- For supervised classification (= using class label), we have a variety of measures to evaluate how good our model is.
 - Accuracy, precision, recall
- For cluster analysis (= unsupervised), the analogous question is:
How to evaluate the “goodness” of the resulting clusters?
- But “clusters are in the eye of the beholder”!
- Then, why do we want to evaluate them?
 - To avoid finding patterns in **noise**
 - To compare **clustering algorithms**
 - To compare **two sets of clusters**
 - To compare **two clusters**

Aspects of Cluster Validation

1. Determining the **clustering tendency** of a set of data, i.e., distinguishing whether non-random structure actually exists in the data (e.g., to avoid overfitting).
2. **External Validation:** Compare the results of a cluster analysis to externally known class labels (ground truth).
3. **Internal Validation:** Evaluating how well the results of a cluster analysis fit the data without reference to external information – use only the data.
4. **Compare clusterings** to determine which is better.
5. Determining the **“correct” number of clusters**.

For 2, 3, and 4, we can further distinguish whether we want to evaluate the entire clustering or just individual clusters.

Measures of Cluster Validity

Numerical measures for judging various aspects of cluster validity are classified into the following three types.

- **External Index:** Used to measure the extent to which cluster labels match **externally supplied class labels**.
 - Entropy, Purity, Rand index
- **Internal Index:** Used to measure the **goodness** of a clustering structure **without respect to external information**.
 - Sum of Squared Error (SSE), Silhouette coefficient
- **Relative Index:** Used to compare 2 different clusterings or clusters.
 - Often an external or internal index is used for this function, e.g., SSE or entropy

Sometimes these are referred to as **criteria** instead of **indices**

- However, for some researchers, a **criterion** is the general strategy and an **index** is the numerical measure itself that implements the criterion.

Definition 8.4. The **correlation** coefficient $\rho_{X,Y}$ between two random variables X and Y with expected values μ_X and μ_Y and standard deviations σ_X and σ_Y is defined as

$$\rho_{X,Y} = \text{corr}(X, Y) = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y} = \frac{\mathbb{E}[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y} \in [-1, 1], \quad (8.7)$$

where if X and Y are independent, $\rho_{X,Y} = 0$. (The reverse may not be true.)

Note: The correlation between two vectors $\mathbf{u}$ and $\mathbf{v}$ is the cosine of the angle between the two vectors.

$$\text{corr}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}. \quad (8.8)$$

Measuring Cluster Validity Via Correlation

- Two matrices
 - **Proximity matrix**^a ($P \in \mathbb{R}^{N \times N}$)
 - **Incidence matrix** ($I \in \mathbb{R}^{N \times N}$)
 - * One row and one column for each data point
 - * An entry is 1 if the associated pair of points belong to the same cluster
 - * An entry is 0 if the associated pair of points belongs to different clusters
- **Compute the correlation between the two matrices**
 - Since the matrices are symmetric, only the correlation between $N(N-1)/2$ entries needs to be calculated.
- **High correlation** indicates that points that belong to the same cluster are close to each other.

Example: For K-Means clusterings of two data sets, the correlation coefficient are:

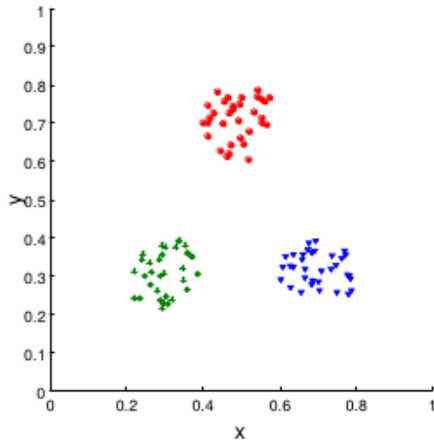


Figure 8.32: $\rho_{P,I} = -0.924$.

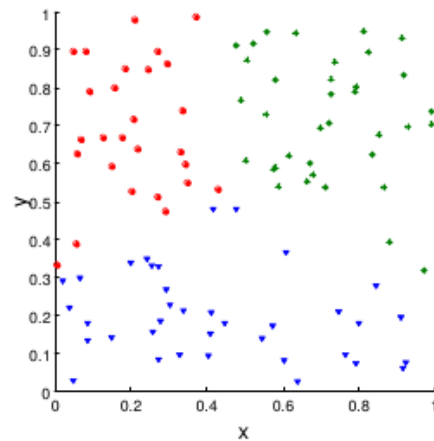


Figure 8.33: $\rho_{P,I} = -0.581$.

- **Not a good measure for some density- or contiguity-based clusters** (e.g., single link HC).

^aA **proximity matrix** is a square matrix in which the entry in cell (i, j) is some measure of the similarity (or distance) between the items to which row i and column j correspond.

Using Similarity Matrix for Cluster Validation

Order the **similarity matrix** with respect to cluster labels and inspect visually.

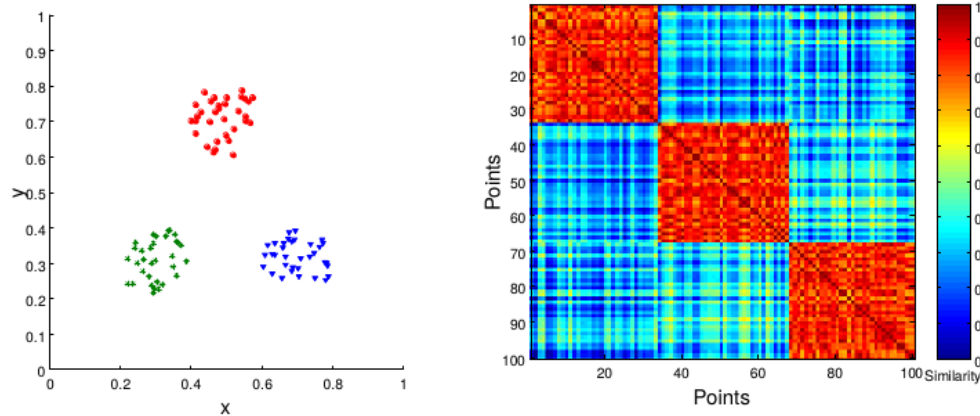


Figure 8.34: **Clusters are so crisp!**

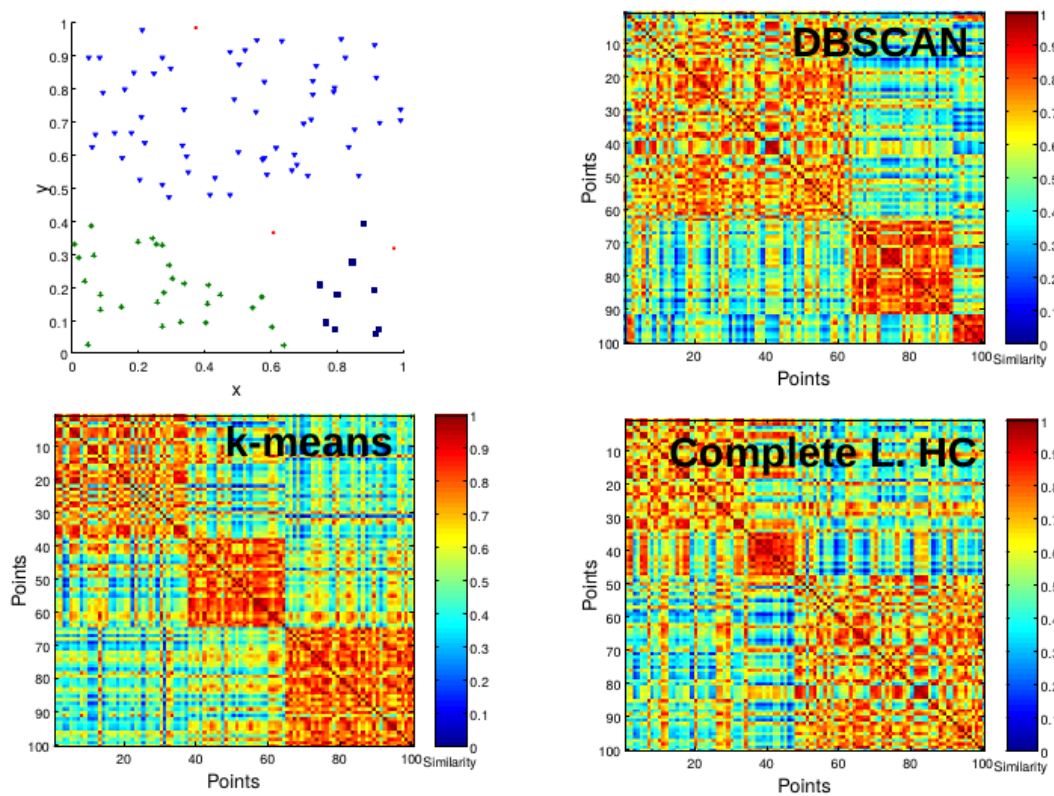


Figure 8.35: Clusters in random data are not so crisp.

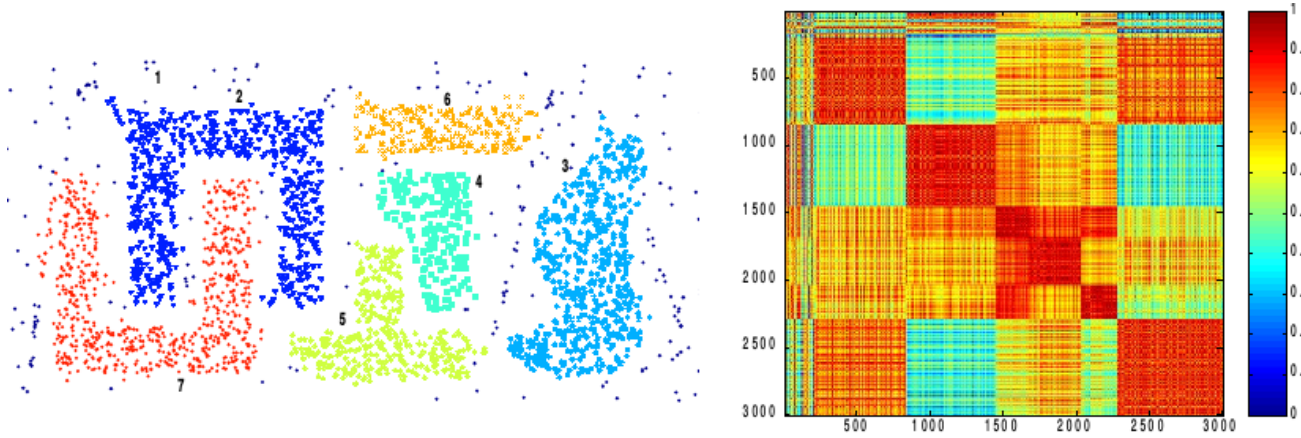


Figure 8.36: Similarity matrix for cluster validation, for DBSCAN.

8.5.2. Internal and external measures of cluster validity

Internal Measures

- (Average) SSE is good for comparing two clusterings or two clusters.

$$SSE = \sum_{i=1}^K \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2. \quad (8.9)$$

- It can also be used **to estimate the number of clusters**

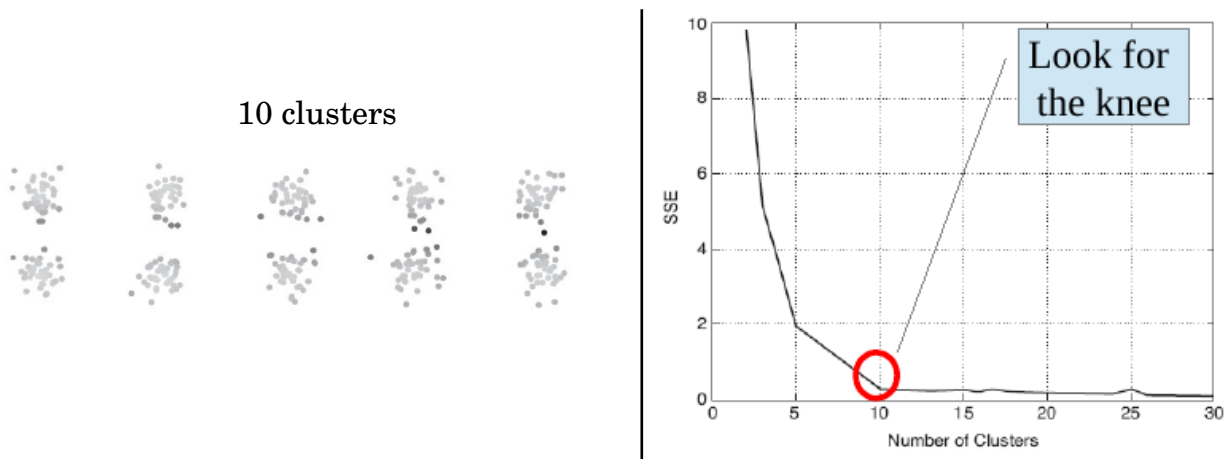


Figure 8.37: Estimation for the number of clusters.

Cohesion and Separation

- **Cluster cohesion:** Measure how closely related are objects in a cluster
 - Example: Within-cluster sum of squares (WSS=SSE)

$$WSS = \sum_{i=1}^K \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2. \quad (8.10)$$

- **Cluster separation:** Measures how distinct or well-separated a cluster is from other clusters
 - Example: Between-cluster sum of squares (BSS)

$$BSS = \sum_{i=1}^K |C_i| \|\boldsymbol{\mu} - \boldsymbol{\mu}_i\|^2. \quad (8.11)$$

- **Total sum of squares:** $TSS = WSS + BSS$
 - TSS is a constant for a given data set (independently of the number of clusters)
 - Example: a cluster $\{1, 2, 4, 5\}$ can be separated into two clusters $\{1, 2\} \cup \{4, 5\}$. It is easy to check the following.
 - * 1 cluster: $TSS = WSS + BSS = 10 + 0 = 10$.
 - * 2 clusters: $TSS = WSS + BSS = 1 + 9 = 10$.

Silhouette Coefficient

- **Silhouette coefficient** combines ideas of both cohesion and separation, but for individual points. For an individual point i :
 - Calculate $a(i)$ = average distance of i to all other points in its cluster
 - Calculate $b(i)$ = $\min \{\text{average distance of } i \text{ to points in another cluster}\}$
 - The silhouette coefficient for the point i is then given by

$$s(i) = 1 - a(i)/b(i). \quad (8.12)$$
 - Typically, $s(i) \in [-1, 1]$.
 - The closer to 1, the better.
- Can calculate **the average silhouette width** for a cluster or a clustering

Selecting K with Silhouette Analysis on K-Means Clustering

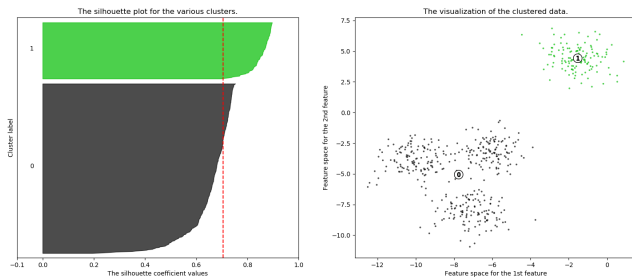


Figure 8.38: $n_clusters = 2$;
average silhouette score = 0.705.

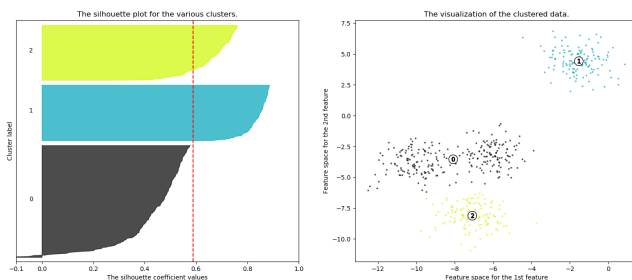


Figure 8.39: $n_clusters = 3$;
average silhouette score = 0.588.

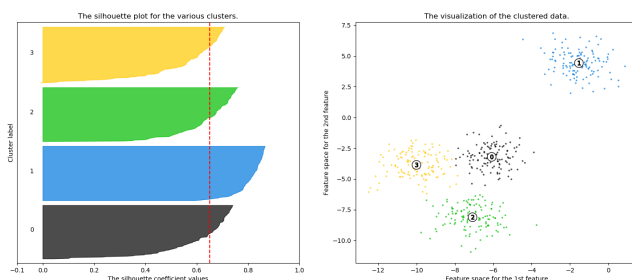


Figure 8.40: $n_clusters = 4$;
average silhouette score = 0.651.

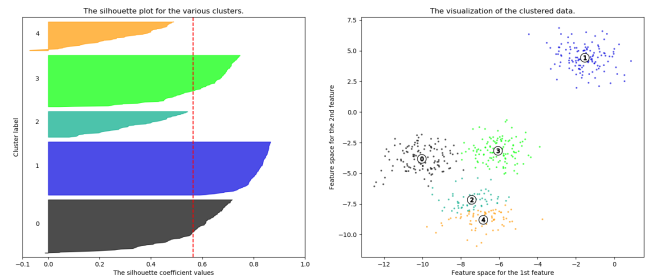


Figure 8.41: $n_clusters = 5$;
average silhouette score = 0.564.

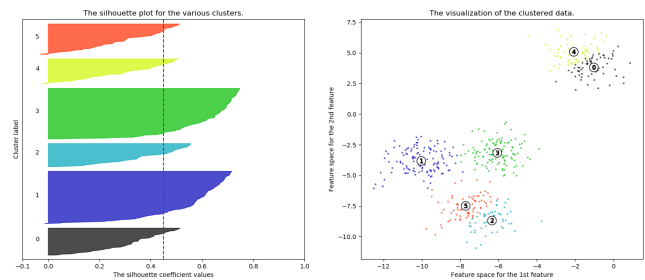


Figure 8.42: $n_clusters = 6$;
average silhouette score = 0.450.

- The silhouette plot shows that ($n_clusters = 3, 5,$ and 6) are **bad picks** for the data, due to
 - the presence of clusters with below average silhouette scores
 - wide fluctuations in the size of the silhouette plots
- Silhouette analysis is ambivalent in deciding between **2** and **4**.
- When $n_clusters = 4$, all the silhouette subplots are more or less of similar thickness.

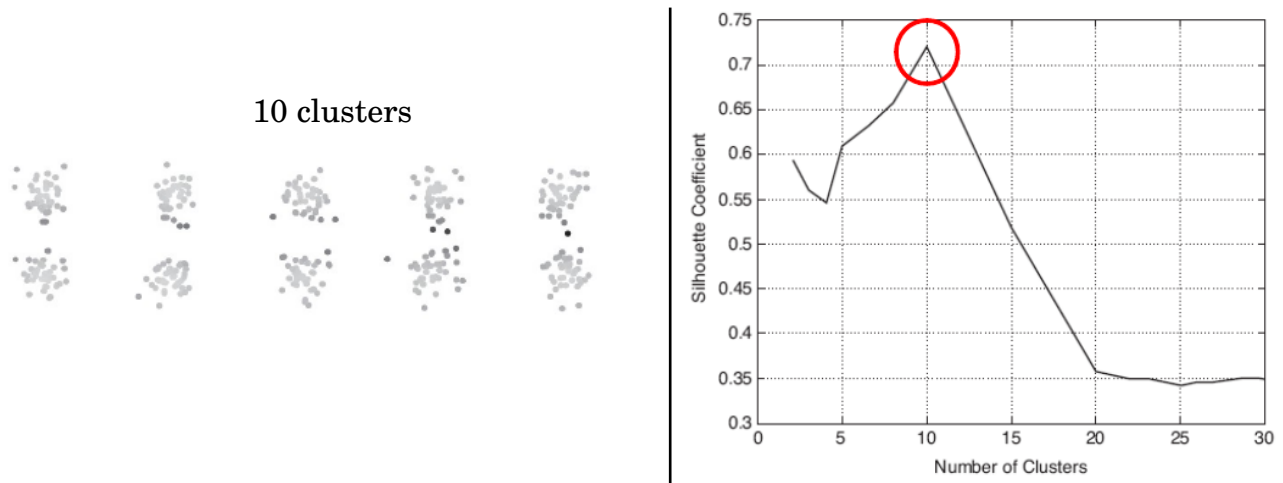
Another way of Picking K with Silhouette Analysis

Figure 8.43: Average silhouette coefficient vs. number of clusters.

External Measures of Cluster Validity

• Entropy^a

- For cluster j , let p_{ij} be the probability that a member of cluster j belongs to class i , defined as

$$p_{ij} = n_{ij}/N_j, \quad (8.13)$$

where N_j is the number of points in cluster j and n_{ij} is the number of points of class i in cluster j .

- **The entropy of each cluster j** is defined as

$$e_j = - \sum_{i=1}^L p_{ij} \log_2 p_{ij}, \quad (8.14)$$

where L is the number of classes and

- The total entropy is calculated as the sum of entropies of each cluster weighted by the size of each cluster: for $N = \sum_{j=1}^K N_j$,

$$e = \frac{1}{N} \sum_{j=1}^K N_j e_j. \quad (8.15)$$

• Purity

- The purity of cluster j is given by

$$\text{purity}_j = \max_i p_{ij}. \quad (8.16)$$

- The overall purity of a clustering is

$$\text{purity} = \frac{1}{N} \sum_{j=1}^K N_j \text{purity}_j. \quad (8.17)$$

^aThe concept of **entropy** was introduced earlier in §5.4.1. *Decision tree objectives*, when we defined impurity measures. See (5.68) on p. 124.

Other measures: Precision, Recall, F-measure, Rand

Final Comment on Cluster Validity

The following is a claim in an old book by (Jain & Dubes, 1988) [35].

However, today, the claim is yet true.

“The validation of clustering structures is the most difficult and frustrating part of cluster analysis. Without a strong effort in this direction, cluster analysis will remain a black art accessible only to those true believers who have experience and great courage.”

8.6. Self-Organizing Maps

8.6.1. Basics of the SOM

- **Self-Organizing Map (SOM)** refers to a process in which *the internal organization of a system increases automatically* without being guided or managed by an outside source.
 - This process is due to **local interaction** with **simple rules**.
 - Local interaction gives rise to **global structure**.
- **Why SOM?**

A high-dimensional dataset is represented as an one/two-dimensional discretized pattern using self-organizing maps or Kohonen maps.
- **Advantage of SOM?**

The primary benefit of employing an SOM is that the data is simple to read and comprehend. Grid clustering and the decrease of dimensionality make it simple to spot patterns in the data.

We can interpret emerging **global structures** as **learned structures**, which in turn appear as **clusters** of similar objects.

Note: The SOM acts as a **unsupervised clustering algorithm** and a powerful **visualization tool** as well.

- It considers a **neighborhood structure** among the clusters.
- ⊕ The SOM is **widely used** in many application domains, such as economy, industry, management, sociology, geography, text mining, etc..
- ⊕ **Many variants** have been suggested to adapt the SOM to the processing of complex data, such as time series, categorical data, nominal data, dissimilarity or kernel data.
- ⊖ However, the SOM has suffered from **a lack of rigorous results** on its **convergence** and **stability**.

Game of Life: – Most famous example of self-organization.

Simple local rules (en.wikipedia.org/wiki/Conway's_Game_of_Life):

- **Any live cell with fewer than two live neighbors** dies, as if caused by under-population.
- **Any live cell with two or three live neighbors** lives on to the next generation.
- **Any live cell with more than three live neighbors** dies, as if by overcrowding.
- **Any dead cell with exactly three live neighbors** becomes a live cell, as if by reproduction.

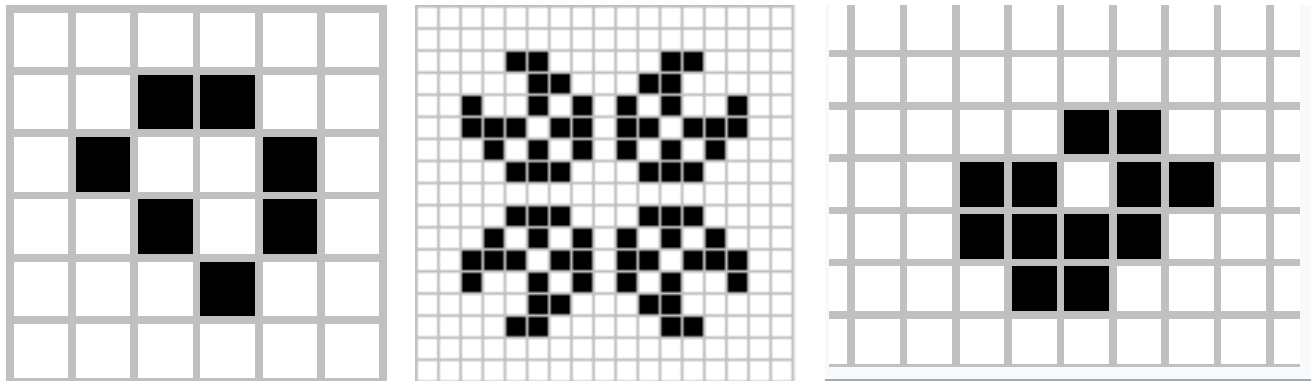


Figure 8.44: Still life, oscillator, and spaceship.

SOM Architecture

- A **feed-forward neural network**^a architecture based on **competitive learning**^b, invented by **Teuvo Kohonen** in 1982 [39].
 - Does not depend on *a priori* selection of number of clusters to search for
 - Will find the appropriate number of clusters, given a set of instances.
- Neurobiological studies indicate that different sensory inputs (motor^c, visual, auditory, etc.) are mapped onto corresponding areas of the cerebral cortex in an **orderly fashion**.
- Our interest is in building **artificial topographic maps** that learn through self-organization in a neurobiologically inspired manner.
- We shall follow the **principle of topographic map formation**:
 “The spatial location of an output neuron in a topographic map corresponds to a particular domain or feature drawn from the input space”.

^aA **feed-forward neural network** is an artificial neural network wherein connections between the nodes do not form a cycle. It is the first and simplest type of artificial neural network devised. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes.

^bOne particularly interesting class of unsupervised system is based on **competitive learning**, in which the output neurons compete amongst themselves to be activated, with the result that only one is activated at any one time. This activated neuron is called a **winner-takes-all neuron** or simply the **winning neuron**. Such competition can be induced/implemented by having **lateral inhibition connections** (negative feedback paths) between the neurons. The result is that the neurons are forced to organize themselves.

^c**Motor output** is a response to the stimuli received by the nervous system.

Some More Details on the SOM

- **The principal goal of an SOM** is to **transform** an incoming signal pattern of arbitrary dimension **into a one/two-dimensional discrete map**, and to perform this transformation adaptively in a topologically ordered fashion.
 - We therefore set up our SOM by placing neurons at the nodes of a one/two-dimensional lattice.
 - Higher dimensional maps are also possible, but not so common.
- The neurons become **selectively tuned** to various input patterns (stimuli) or classes of input patterns during the course of the competitive learning.
- The locations of the neurons so tuned (i.e. the winning neurons) become ordered, and a **meaningful coordinate system** for the input features is created on the lattice.
 - The SOM thus forms the required topographic map of the input patterns.
- We can view this as **a non-linear generalization of principal component analysis (PCA)**.

Versions of the SOM

- **Basic version:** a stochastic process
- **Deterministic version:**
 - For **industrial applications**, it can be more convenient to use a deterministic version of the SOM, in order to get **the same results** at each run of the algorithm when the initial conditions and the data remain unchanged (**repeatable!**).
 - To address this issue, T. Kohonen has introduced the batch SOM in 1995 [40].

8.6.2. Kohonen SOM networks

We shall concentrate on the particular kind of the SOM known as a **Kohonen SOM network** or **Kohonen network**.

- The Kohonen SOM network has a **feed-forward structure** with a **single computational layer** arranged in rows and columns.
- Each neuron is **fully connected** to all the source nodes in the input layer

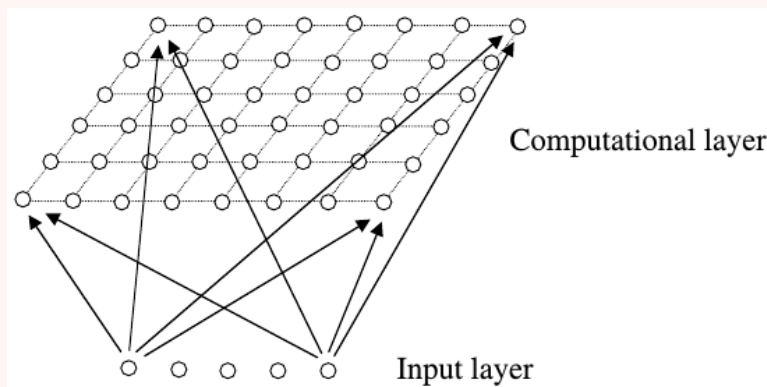


Figure 8.45: Kohonen network.

Data Types for the Kohonen SOM

Originally, the SOM algorithm was defined for data described by numerical vectors which belong to a subset X of an Euclidean space (typically $\mathbb{R}^d$). For some results, we need to assume that the subset is bounded and convex. Two different settings have to be considered from theoretical and practical points of view:

- **Continuous setting:** the input space $X \subset \mathbb{R}^d$ is modeled by a probability distribution with a density function f ,
- **Discrete setting:** the input space X comprises N data points

$$\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N \in \mathbb{R}^d.$$

Here the discrete setting means a finite subset of the input space.

Components of Self-Organization

Preliminary 8.5. The self-organization process involves four major components:

1. **Initialization:** All the connection weights are initialized with small random values.
2. **Competition:** For each input pattern, the neurons compute their respective values of a **discriminant function** which provides the basis for competition.
 - The particular neuron with **the smallest value of the discriminant function** is declared **the winner**.
3. **Cooperation:** The winning neuron determines the spatial location of a **topological neighborhood** of excited neurons, thereby providing the basis for cooperation among neighboring neurons.
(smoothing the neighborhood of the winning neuron)
4. **Adaptation:** The excited neurons decrease their individual values of the discriminant function in relation to the input pattern through suitable adjustment of the associated connection weights, such that the response of the winning neuron to the subsequent application of a similar input pattern is enhanced.
(making the winning neuron look more like the observation)

Remark 8.6. The SOM is an unsupervised system based on **competitive learning**.

- The output neurons compete amongst themselves to be activated, with the result that only one is activated at any one time.
- This activated neuron is called a **winner-takes-all neuron** or simply the **winning neuron**.
- Such competition can be implemented by having lateral inhibition connections (**negative feedback paths**) between the neurons.
- **The result:** The neurons are forced to organize themselves.

The Competitive Process

- **The input space** is d -dimensional (i.e. there are d input units).
- **The connection weights** between **the d input units** and **the k th output neuron** (in the computational layer) can be written

$$\mathbf{w}_k = \begin{bmatrix} w_{1k} \\ w_{2k} \\ \vdots \\ w_{dk} \end{bmatrix}, \quad k = 1, \dots, K, \quad (8.18)$$

where K is the total number of neurons in the computational layer.

Definition 8.7. We can define the **discriminant function** to be the squared Euclidean distance between the input vector $\mathbf{x}$ and the weight vector $\mathbf{w}_k$ for each neuron k :

$$d_k(\mathbf{x}) = \|\mathbf{x} - \mathbf{w}_k\|^2 = \sum_{i=1}^d (x_i - w_{ik})^2. \quad (8.19)$$

Thus, the neuron whose weight vector comes closest to the input vector (i.e. is most similar to it) is declared the winner; see Preliminary 8.5, item 2.

The Cooperative Process

In neurobiological studies, it is found that there is **lateral interaction** between a set of excited neurons.

- When one neuron fires, its closest neighbors tend to get excited more than those further away.
- There is a **topological neighbourhood** that decays with distance.

Definition 8.8. **Neighbourhood Function**

- Let us take K units on a regular lattice (string-like for 1D, or grid-like for 2D).
- If $\mathcal{K} = \{1, 2, \dots, K\}$ and t is the time, a **neighborhood function** $h(t)$ is defined on $\mathcal{K} \times \mathcal{K}$. It has to satisfy the following properties:
 - (a) h is symmetric with $h_{kk} = 1$,
 - (b) $h_{k\ell}$ depends only on the distance $\text{dist}(k, \ell)$ between units k and ℓ on the lattice, and
 - (c) h decreases with increasing distance.
- **Several choices are possible for h .**
 - The most classical is the **step function**; equal to 1 if the distance between k and ℓ is less than a specific radius (this radius can decrease with time), and 0 otherwise.
 - Another very classical choice is a **Gaussian-shaped function**

$$h_{k\ell}(t) = \exp\left(-\frac{\text{dist}^2(k, \ell)}{2\sigma^2(t)}\right), \quad (8.20)$$

where $\sigma^2(t)$ **can decrease over time** to reduce the intensity and the scope of the neighborhood relations. A popular time dependence is an exponential decay:

$$\sigma(t) = \sigma_0 \exp(-t/\tau_\sigma). \quad (8.21)$$

The Adaptive Process

The SOM must involve an **adaptive (learning) process** by which

- the outputs become self-organized and
- the **feature map** between inputs and outputs is formed.

Learning Rule in the Adaptive Process

- The point of the **topographic neighborhood** is twofold:
 - The winning neuron gets its weights updated.
 - Its neighbors will have their weights updated as well, although by not as much as the winner itself.

- An appropriate **weight update rule** is formulated as

$$\Delta \mathbf{w}_k = \eta(t) \cdot h_{I(\mathbf{x}),k}(t) \cdot (\mathbf{x} - \mathbf{w}_k), \quad \forall k \in \mathcal{K}, \quad (8.22)$$

where $I(\mathbf{x})$ is the index of the winning neuron and $\eta(t)$ is a **learning rate** ($0 < \eta(t) < 1$, constant or decreasing).

- **The effect of each learning weight update** is to move the weight vectors $\mathbf{w}_k$ of the winning neuron and its neighbors towards the input vector $\mathbf{x}$.
 - Repeated presentations of the training data thus leads to topological ordering.

Remark 8.9. The learning rule (8.22) has several important properties:

- Maximal at the winning neuron.
- Symmetric about that neuron.
- Decreases monotonically to zero as the distance goes to infinity.
- Translation invariant (i.e., independent of the location of the winning neuron).

8.6.3. The SOM algorithm and its interpretation

The stages of the SOM can be summarized as follows.

Algorithm 8.10. The Stochastic SOM

- **Initialization:** A connection weight $\mathbf{w}_k \in \mathbb{R}^d$ is attached to each unit k , whose initial values are chosen at random and denoted by

$$W(0) = [\mathbf{w}_1(0), \mathbf{w}_2(0), \dots, \mathbf{w}_K(0)].$$

- **For** $t = 0, 1, 2, \dots$

(a) **Sampling:** A data point $\mathbf{x}$ is **randomly drawn** (according to the density function f or from the finite set X)

(b) **Matching:** The **best matching unit** is defined by

$$I(\mathbf{x}) = \arg \min_{k \in \mathcal{K}} \|\mathbf{x} - \mathbf{w}_k(t)\|^2 \quad (8.23)$$

(c) **Updating:** All the weights are updated via

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \Delta \mathbf{w}_k, \quad \forall k \in \mathcal{K}, \quad (8.24)$$

where, as defined in (8.22),

$$\Delta \mathbf{w}_k = \eta(t) \cdot h_{I(\mathbf{x}),k}(t) \cdot (\mathbf{x} - \mathbf{w}_k).$$

(d) **Continuation:** Keep returning to the **sampling** step until the feature map stops changing.

Results of the SOM

- After learning, cluster C_k can be defined as the set of inputs closer to $\mathbf{w}_k$ than to any other one.
- The **Kohonen map** is the representation of
 - the weights or
 - the cluster contents,
 displayed according to the **neighborhood structure**.

Example: Data approximation

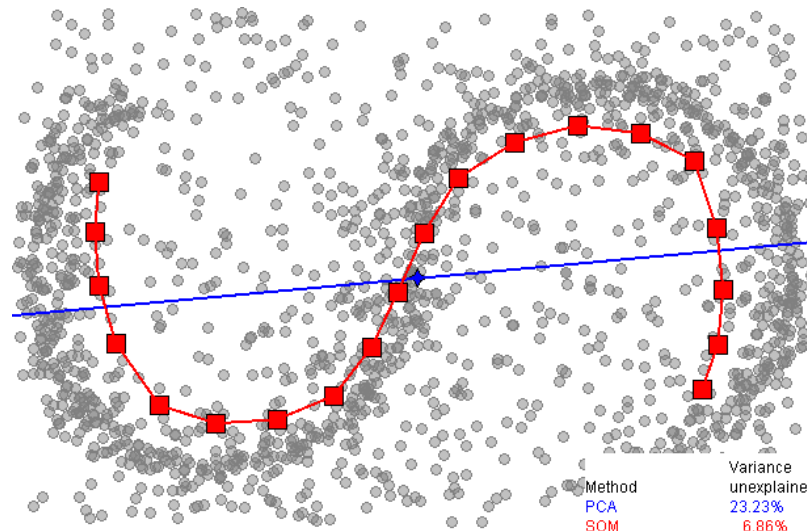


Figure 8.46: **Data approximation:** One-dimensional **SOM** vs. **PCA**.

SOM is a red broken line with squares, 20 nodes. The first principal component is presented by a blue line. Data points are the small gray circles. The fraction of **variance unexplained** in this example is **6.86% for SOM** and **23.23% for PCA**.

(“Self-organizing map”, Wikipedia)

Properties of the Kohonen Maps

- **The quantization property:** the weights represent the data space as accurately as possible, as do other quantization algorithms.
 - To get a better quantization, the learning rate $\eta(t)$ decreases with time as well as the scope of the neighborhood function h .
- **The self-organization property**, that means that the weights preserve the topology of the data:
 - **close inputs** belong to the **same cluster** (as do any clustering algorithms) or to **neighboring clusters**.

Theoretical Issues

- The algorithm is easy to define and to use, and a lot of practical studies confirm that it works.
 - However, the theoretical study of its convergence when t tends to ∞ remains without complete proof and provides open problems.
 - The main question is to know if the solution obtained from a finite sample converges to the true solution that might be obtained from the true data distribution.
- When t tends to ∞ , the $\mathbb{R}^d$ -valued stochastic processes $[\mathbf{w}_k(t)]_{k=1,2,\dots,K}$ can present oscillations, explosion to infinity, convergence in distribution to an equilibrium process, convergence in distribution or almost sure to a finite set of points in $\mathbb{R}^d$, etc.. Some of the open questions are:
 - Is the algorithm convergent in distribution or almost surely, when t tends to ∞ ?
 - What happens when $\eta(t)$ is constant? (when it decreases?)
 - If a limit state exists, is it stable?
 - How to characterize the organization?

In Practice: Ordering and Convergence

Note: The SOM algorithm may start from an initial state of complete disorder, and it will gradually lead to an organized representation of activation patterns drawn from the input space.

There are two identifiable phases of **the adaptive process**:

1. **Ordering or Self-organizing phase** – during which the topological ordering of the weight vectors takes place.
 - Typically this will take as many as 1000 iterations of the SOM algorithm.
 - Careful consideration needs to be given to the choice of neighbourhood and learning rate parameters.
2. **Convergence phase** – during which the feature map is fine tuned and comes to provide an accurate statistical quantification of the input space.
 - Typically the number of iterations in this phase will be at least 500 times the number of neurons in the network.
 - Again, the parameters must be chosen carefully.

Exercises for Chapter 8

- 8.1. We will experiment the K-Means algorithm following the first section of Chapter 11, *Python Machine Learning, 3rd Ed.*, in a little bit different fashion.
- (a) Make a dataset of 4 clusters (modifying the code on pp. 354–355).
 - (b) For $K = 1, 2, \dots, 10$, run the K-Means clustering algorithm with the initialization `init='k-means++'`.
 - (c) For each K , compute the within-cluster SSE (distortion) for an **elbow analysis** to select an appropriate K . **Note:** Rather than using `inertia_` attribute, **implement a function** for the computation of distortion.
 - (d) Produce **silhouette plots** for $K = 3, 4, 5, 6$.
- 8.2. Now, let's experiment DBSCAN, following *Python Machine Learning, 3rd Ed.*, pp. 376–381.
- (a) Produce a dataset having **three** half-moon-shaped structures each of which consists of 100 samples.
 - (b) Compare performances of K-Means, AGNES, and DBSCAN.
(Set `n_clusters=3` for K-Means and AGNES.)
 - (c) For K-Means and AGNES, what if you choose `n_clusters` much larger than 3 (for example, 9, 12, 15)?
 - (d) Again, for K-Means and AGNES, perform an **elbow analysis** to select an appropriate K .

CHAPTER 9

Neural Networks and Deep Learning

Deep learning is a family of machine learning methods based on **learning data representations**, as opposed to task-specific algorithms. Learning can be supervised, semi-supervised, or unsupervised [3, 5, 69].

- Deep learning allows computational models that are composed of **multiple processing layers to learn representations of data** with multiple levels of abstraction.
- These methods have **dramatically improved the state-of-the-art** in **speech recognition, visual object recognition**, and many other domains such as **drug discovery** and **genomics**.
- Deep learning **discovers complex structures in large datasets** by using powerful algorithms such as the **convolutional mapping** and the **back-propagation**.
- **Convolutional neural nets** have brought about breakthroughs in processing images, video, speech and audio, whereas **recurrent neural nets** have shone light on sequential data such as text and speech.

Contents of Chapter 9

9.1. Basics for Deep Learning	262
9.2. Neural Networks	266
9.3. Back-Propagation	281
9.4. Deep Learning: Convolutional Neural Networks	288
Exercises for Chapter 9	297

9.1. Basics for Deep Learning

Conventional Machine Learning

- Limited in their ability to process data in their raw form
- **Feature!!**
 - Coming up with features is difficult, time-consuming, requires expert knowledge.
 - When working applications of learning, we spend a lot of time tuning the features.



Examples of features: Histogram of oriented gradients (HOG), the **scale-invariant feature transform (SIFT)** (Lowe, 1999) [50], etc.

Representation Learning

- A machine is fed with raw data
⇒ discover representations, automatically
- **Deep Learning** methods are representation-learning methods with multiple levels of **representation/abstraction**
 - Simple non-linear modules ⇒ higher and abstract representation
 - With the composition of enough such transformations, very complex functions can be learned.
- **Key Aspects**
 - **Layers of features** are not designed by human engineers.
 - **Learn features** from data using a general-purpose learning procedure.

Advances in Deep Learning

- Image recognition [23, 29, 41]
- Speech recognition [29, 66]
- Natural language understanding [23, 73, 82]
 - Machine translation
 - Image 2 text
 - Sentiment analysis
 - **Question-answering (QA) machine:**
IBM's Watson, 2011, defeated legendary Jeopardy champions Brad Rutter and Ken Jennings, winning the first place prize of \$1 million
- Many other domains
 - Predicting the activity of potential drug molecules
 - Analyzing particle accelerator data
 - Reconstructing brain circuits
 - Predicting the effects of mutations in non-coding DNA on gene expression and disease
- **Image-based Classifications:** Deep learning has provided breakthrough results in **speech recognition** and **image classification**.

Why/What about Deep Learning?

- **Why is it generally better than other methods** on image, speech, and certain other types of data? Short answers:
 - Deep learning means using a **neural network** with **several layers of nodes** between input and output
 - The series of layers between input & output do **feature identification and processing** in a series of stages, just as our brains seem to.
- Multi-layer neural networks have been more than 30 years (Rina Dechter, 1986) [14]. **What is actually new?**
 - We have always had good algorithms for learning the weights in networks **with 1 hidden layer**.
But these algorithms are not good at learning the weights for networks with more hidden layers
 - **The New** are: **methods for training many-layer networks**

Terms: AI vs. ML vs. Deep Learning

- **Artificial intelligence (AI)**: Intelligence exhibited by machines
- **Machine learning (ML)**: An approach to achieve AI
- **Deep learning (DL)**: A technique for implementing ML
 - Feature/Representation-learning
 - Multi-layer neural networks (NN)
 - Back-propagation

(In the 1980s and 1990s, researchers did not have much luck, except for a few special architectures.)

 - **New ideas** enable learning in deep NNs, since 2006

Back-propagation to Train Multi-layer Architectures

- Nothing more than a **practical application of the chain rule**, for derivatives
- Feed-forward neural networks
 - Non-linear function: $\max(z, 0)$ (ReLU), $\tanh(z)$, $1/(1 + e^{-z})$
- Forsaken because poor local minima
- Revived around 2006 by unsupervised learning procedures with unlabeled data
 - Local minima are rarely a problem, in practice
 - **CIFAR** (Canadian Institute for Advanced Research): [4, 30, 31, 46]
 - Recognizing handwritten digits or detecting pedestrians
 - Speech recognition by **GPUs**, with 10 or 20 times faster (Raina *et al.*, 2009) [60] and (Bengio, 2013) [2].
- **Convolutional neural network (CNN)**
 - Widely adopted by computer-vision community (LeCun *et al.*, 1989) [45]

Machine Learning Challenges We've Yet to Overcome

- **Interpretability**: Although ML has come very far, researchers still **don't know exactly how deep nets training work**.
 - If we don't know how training nets actually work, how do we make any real progress?
- **One-Shot Learning**: We still haven't been able to achieve one-shot learning. **Traditional gradient-based networks need a huge amount of data**, and are often in the form of **extensive iterative training**.
 - Instead, we should find a way to enable neural networks to learn, using just a few examples.

9.2. Neural Networks

Recall: In 1957, Frank Rosenblatt invented the **perceptron** algorithm:

- For input values: $\mathbf{x} = (x_1, x_2, \dots, x_d)^T$,
- Learn weight vector: $\mathbf{w} = (w_1, w_2, \dots, w_d)^T$
- Get the net input $z = w_1x_1 + w_2x_2 + \dots + w_dx_d = \mathbf{w} \cdot \mathbf{x}$
- Classify, using the activation function

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise,} \end{cases} \quad z = \mathbf{w} \cdot \mathbf{x}, \quad (9.1)$$

or, equivalently,

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise,} \end{cases} \quad z = b + \mathbf{w} \cdot \mathbf{x}, \quad (9.2)$$

where $b = -\theta$ is the **bias**. (See (3.2) and (3.3), p. 40.)

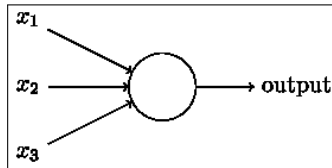


Figure 9.1: Perceptron: The simplest artificial neuron.

Example: Going to a cheese festival. Suppose the weekend is coming up, and you’ve heard that there will be a cheese festival. You like cheese, and are trying to decide whether or not to go to the festival. You might make your decision by weighing up three factors:

- Is the weather good? ($x_1 \in [0, 1]$)
- Does your boyfriend or girlfriend want to accompany you? ($x_2 \in \{0, 1\}$)
- Is the festival near public transit? ($x_3 \in [0, 1]$)

For example, choose $w_1 = 6$, $w_2 = 2$, $w_3 = 2$, and $b = -\theta = -5$.

With these choices, the perceptron implements the desired decision-making model. By varying the weights and the threshold, we can get different models of decision-making.

Perceptron is the simplest artificial neuron that makes decisions by **weighting up** evidence.

Complex Network of Perceptrons

- Obviously, the perceptron is not a complete model of human decision-making!
- What the example illustrates is how a perceptron can weigh up different kinds of evidence in order to make decisions.
- It should seem plausible that a complex network of perceptrons could make quite subtle decisions:

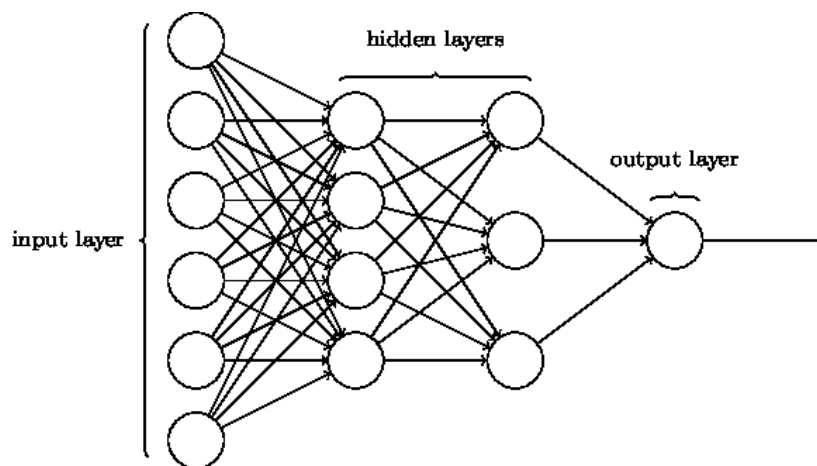


Figure 9.2: A complex network of perceptrons.

An Issue on Perceptron Networks

- A small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from -1 to 1 .
- That flip may then cause the behavior of the rest of the network to completely change in some very complicated way.
- We can overcome this problem by introducing a new type of artificial neuron called a **sigmoid neuron**.

9.2.1. Sigmoid neural networks

Recall: The **logistic sigmoid function** is defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \quad (9.3)$$

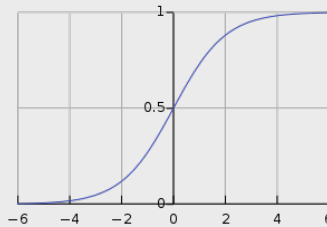


Figure 9.3: The standard logistic sigmoid function $\sigma(z) = 1/(1 + e^{-z})$.

Sigmoid Neural Networks

- They are built with **sigmoid neurons**.
- The output of a sigmoid neuron with inputs $\mathbf{x}$, weights $\mathbf{w}$, and bias b is

$$\sigma(z) = \frac{1}{1 + \exp(-b - \mathbf{w} \cdot \mathbf{x})}, \quad (9.4)$$

which we considered as the **logistic regression** model in Section 5.2.

- Advantages of the sigmoid activation:
 - It allows **calculus** to design learning rules. ($\sigma' = \sigma(1 - \sigma)$)
 - Small changes in weights and bias produce a **corresponding small change** in the output.

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b. \quad (9.5)$$

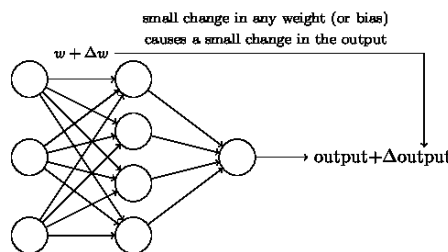


Figure 9.4: Δoutput is a linear combination of Δw_j and Δb .

The architecture of (sigmoid) neural networks

- The leftmost layer is called the **input layer**, and the neurons within the layer are called **input neurons**.
- The rightmost layer is the **output layer**.
- The middle layers are called **hidden layers**.
- The design of the input and output layers in a network is often straightforward. For example, for the classification of handwritten digits:
 - If the images are in 28×28 grayscale pixels, then we'd have 784 ($= 28 \times 28$) input neurons.
 - It is heuristic to set 10 neurons in the output layer. (rather than 4, where $2^4 = 16 \geq 10$)
- There can be **quite an art** to the design of the hidden layers. In particular, it is not possible to sum up the design process for the hidden layers with a few simple rules of thumb. Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behavior they want out of their nets. For example, such heuristics can be used to help determine how to trade off the number of hidden layers against the time required to train the network. We'll meet several such design heuristics later in this chapter.

9.2.2. A simple network to classify handwritten digits

- The problem of recognizing handwritten digits has two components: segmentation and classification.

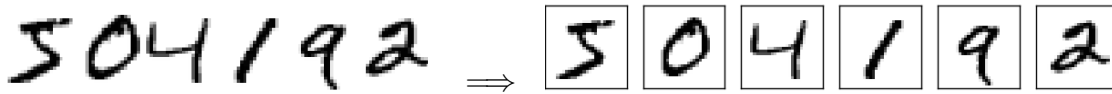


Figure 9.5: Segmentation.

- We'll focus on algorithmic components for the classification of individual digits.

MNIST data set:

A modified subset of two data sets collected by NIST (US National Institute of Standards and Technology):

- Its first part contains 60,000 images (for training)
- The second part is 10,000 images (for test), each of which is in 28×28 grayscale pixels

A Simple Feed-forward Network

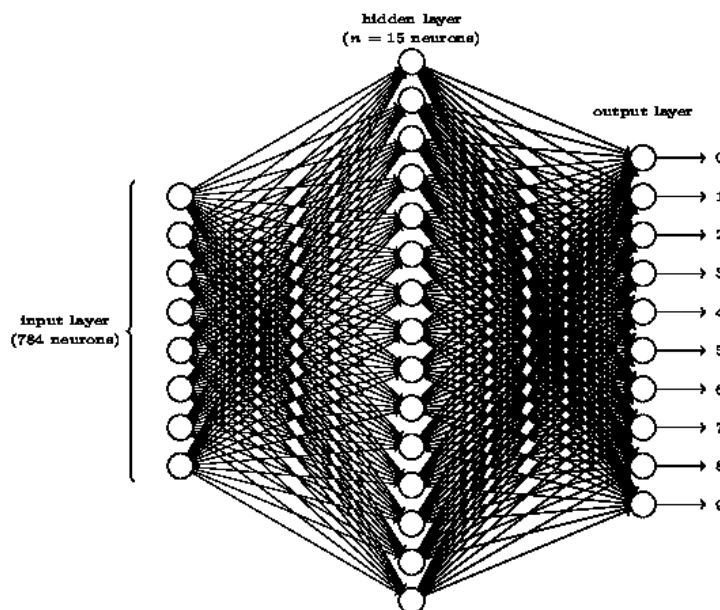


Figure 9.6: A sigmoid network having a hidden layer.

What the Neural Network Will Do

- Let's concentrate on **the first output neuron**, the one that is trying to decide whether or not the input digit is a **0**.
- It does this by weighing up evidence from the hidden layer of neurons.
- **What are those hidden neurons doing?**
- Let's suppose **for the sake of argument** that **the first neuron** in the hidden layer may detect whether or not an image like the following is present



It can do this by heavily weighting input pixels which overlap with the image, and only lightly weighting the other inputs.

- Similarly, let's suppose that **the second, third, and fourth neurons** in the hidden layer detect whether or not the following images are present



- As you may have guessed, these four images together make up the 0 image that we saw in the line of digits shown in Figure 9.5:



- So if all four of these hidden neurons are firing, then we can conclude that the digit is a 0.

Learning with Gradient Descent

- **Data set** $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}, i = 1, 2, \dots, N$
(e.g., if an image $\mathbf{x}^{(k)}$ depicts a 2, then $\mathbf{y}^{(k)} = (0, 0, 1, 0, 0, 0, 0, 0, 0)^T$.)
- **Cost function**

$$C(\mathbf{W}, B) = \frac{1}{2N} \sum_i \|\mathbf{y}^{(i)} - \mathbf{a}(\mathbf{x}^{(i)})\|^2, \quad (9.6)$$

where $\mathbf{W}$ denotes the collection of all weights in the network, B all the biases, and $\mathbf{a}(\mathbf{x}^{(i)})$ is the vector of outputs from the network when $\mathbf{x}^{(i)}$ is input.

- **Gradient descent method**

$$\begin{bmatrix} \mathbf{W} \\ B \end{bmatrix} \leftarrow \begin{bmatrix} \mathbf{W} \\ B \end{bmatrix} + \begin{bmatrix} \Delta \mathbf{W} \\ \Delta B \end{bmatrix}, \quad (9.7)$$

where

$$\begin{bmatrix} \Delta \mathbf{W} \\ \Delta B \end{bmatrix} = -\eta \begin{bmatrix} \nabla_{\mathbf{W}} C \\ \nabla_B C \end{bmatrix}.$$

Note: To compute the gradient ∇C , we need to compute the gradients $\nabla C_{\mathbf{x}^{(i)}}$ separately for each training input, $\mathbf{x}^{(i)}$, and then average them:

$$\nabla C = \frac{1}{N} \sum_i \nabla C_{\mathbf{x}^{(i)}}. \quad (9.8)$$

Unfortunately, when the number of training inputs is very large, it can take a long time, and learning thus occurs slowly. An idea called **stochastic gradient descent** can be used to speed up learning.

Stochastic Gradient Descent

The idea is to estimate the gradient ∇C by computing $\nabla C_{\mathbf{x}^{(i)}}$ for a **small sample of randomly chosen training inputs**. By averaging over this small sample, it turns out that we can quickly get a good estimate of the true gradient ∇C ; this helps speed up gradient descent, and thus learning.

- Pick out a small number of randomly chosen training inputs ($m \ll N$):

$$\tilde{\mathbf{x}}^{(1)}, \tilde{\mathbf{x}}^{(2)}, \dots, \tilde{\mathbf{x}}^{(m)},$$

which we refer to as a **mini-batch**.

- Average $\nabla C_{\tilde{\mathbf{x}}^{(k)}}$ to approximate the gradient ∇C . That is,

$$\frac{1}{m} \sum_{k=1}^m \nabla C_{\tilde{\mathbf{x}}^{(k)}} \approx \nabla C \stackrel{\text{def}}{=} \frac{1}{N} \sum_i \nabla C_{\mathbf{x}^{(i)}}. \quad (9.9)$$

- For classification of handwritten digits for the MNIST data set, you may choose: `batch_size = 10`.

Note: In practice, you can implement the stochastic gradient descent as follows. **For an epoch**,

- Shuffle the data set
- For each m samples (selected from the beginning), update (W, B) using the approximate gradient (9.9).

9.2.3. Implementing a network to classify digits [56]

network.py

```

1  """
2  network.py          (by Michael Nielsen)
3  ~~~~~
4  A module to implement the stochastic gradient descent learning
5  algorithm for a feedforward neural network. Gradients are calculated
6  using backpropagation. """
7  ##### Libraries
8  # Standard library
9  import random
10 # Third-party libraries
11 import numpy as np
12
13 class Network(object):
14     def __init__(self, sizes):
15         """The list ``sizes`` contains the number of neurons in the
16         respective layers of the network. For example, if the list
17         was [2, 3, 1] then it would be a three-layer network, with the
18         first layer containing 2 neurons, the second layer 3 neurons,
19         and the third layer 1 neuron. """
20
21         self.num_layers = len(sizes)
22         self.sizes = sizes
23         self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
24         self.weights = [np.random.randn(y, x)
25                         for x, y in zip(sizes[:-1], sizes[1:])]
26
27     def feedforward(self, a):
28         """Return the output of the network if ``a`` is input."""
29         for b, w in zip(self.biases, self.weights):
30             a = sigmoid(np.dot(w, a)+b)
31         return a
32
33     def SGD(self, training_data, epochs, mini_batch_size, eta,
34            test_data=None):
35         """Train the neural network using mini-batch stochastic
36         gradient descent. The ``training_data`` is a list of tuples
37         ``(x, y)`` representing the training inputs and the desired
38         outputs. """
39
40         if test_data: n_test = len(test_data)
41         n = len(training_data)
42         for j in xrange(epochs):
43             random.shuffle(training_data)
44             mini_batches = [
45                 training_data[k:k+mini_batch_size]
46                 for k in xrange(0, n, mini_batch_size)]

```



```

47         for mini_batch in mini_batches:
48             self.update_mini_batch(mini_batch, eta)
49         if test_data:
50             print "Epoch {0}: {1} / {2}".format(
51                 j, self.evaluate(test_data), n_test)
52         else:
53             print "Epoch {0} complete".format(j)
54
55     def update_mini_batch(self, mini_batch, eta):
56         """Update the network's weights and biases by applying
57         gradient descent using backpropagation to a single mini batch.
58         The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
59         is the learning rate."""
60         nabla_b = [np.zeros(b.shape) for b in self.biases]
61         nabla_w = [np.zeros(w.shape) for w in self.weights]
62         for x, y in mini_batch:
63             delta_nabla_b, delta_nabla_w = self.backprop(x, y)
64             nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
65             nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
66         self.weights = [w-(eta/len(mini_batch))*nw
67                         for w, nw in zip(self.weights, nabla_w)]
68         self.biases = [b-(eta/len(mini_batch))*nb
69                        for b, nb in zip(self.biases, nabla_b)]
70
71     def backprop(self, x, y):
72         """Return a tuple ``(nabla_b, nabla_w)`` representing the
73         gradient for the cost function C_x. ``nabla_b`` and
74         ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
75         to ``self.biases`` and ``self.weights``."""
76         nabla_b = [np.zeros(b.shape) for b in self.biases]
77         nabla_w = [np.zeros(w.shape) for w in self.weights]
78         # feedforward
79         activation = x
80         activations = [x] #list to store all the activations, layer by layer
81         zs = [] # list to store all the z vectors, layer by layer
82         for b, w in zip(self.biases, self.weights):
83             z = np.dot(w, activation)+b
84             zs.append(z)
85             activation = sigmoid(z)
86             activations.append(activation)
87         # backward pass
88         delta = self.cost_derivative(activations[-1], y) * \
89             sigmoid_prime(zs[-1])
90         nabla_b[-1] = delta
91         nabla_w[-1] = np.dot(delta, activations[-2].transpose())
92
93         for l in xrange(2, self.num_layers):

```

```

94         z = zs[-1]
95         sp = sigmoid_prime(z)
96         delta = np.dot(self.weights[-1+1].transpose(), delta) * sp
97         nabla_b[-1] = delta
98         nabla_w[-1] = np.dot(delta, activations[-1-1].transpose())
99         return (nabla_b, nabla_w)
100
101     def evaluate(self, test_data):
102         test_results = [(np.argmax(self.feedforward(x)), y)
103                         for (x, y) in test_data]
104         return sum(int(x == y) for (x, y) in test_results)
105
106     def cost_derivative(self, output_activations, y):
107         """Return the vector of partial derivatives \partial C_x /
108         \partial a for the output activations."""
109         return (output_activations-y)
110
111     ##### Miscellaneous functions
112     def sigmoid(z):
113         return 1.0/(1.0+np.exp(-z))
114
115     def sigmoid_prime(z):
116         return sigmoid(z)*(1-sigmoid(z))

```

The code is executed using

Run_network.py

```

1  import mnist_loader
2  training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
3
4  import network
5  n_neurons = 20
6  net = network.Network([784 , n_neurons, 10])
7
8  n_epochs, batch_size, eta = 30, 10, 3.0
9  net.SGD(training_data , n_epochs, batch_size, eta, test_data = test_data)

```

```
len(training_data)=50000, len(validation_data)=10000, len(test_data)=10000
```

Validation Accuracy

```

1  Epoch 0: 9006 / 10000
2  Epoch 1: 9128 / 10000
3  Epoch 2: 9202 / 10000
4  Epoch 3: 9188 / 10000
5  Epoch 4: 9249 / 10000
6  ...
7  Epoch 25: 9356 / 10000
8  Epoch 26: 9388 / 10000
9  Epoch 27: 9407 / 10000
10 Epoch 28: 9410 / 10000
11 Epoch 29: 9428 / 10000

```

Accuracy Comparisons

- scikit-learn's SVM classifier using the default settings: 9435/10000
- A well-tuned SVM: $\approx 98.5\%$
- Well-designed (convolutional) NN: 9979/10000 (**only 21 missed!**)

Note: For **well-designed neural networks**, the performance is close to **human-equivalent**, and is **arguably better**, since quite a few of the MNIST images are difficult even for humans to recognize with confidence, e.g.,

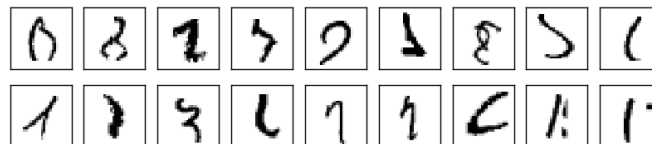


Figure 9.7: MNIST images difficult even for humans to recognize.

Moral of the Neural Networks

- Let all the complexity be learned, automatically, from data
 - Simple algorithms can perform well for some problems:
- (sophisticated algorithm) $\leq$ (simple learning algorithm + good training data)**

9.2.4. Toward deep neural networks

Interpretability issue: While the neural network gives impressive performance, that performance is somewhat mysterious. The weights and biases in the network were discovered automatically. And that means we don't immediately have an explanation of how the network does and what it does.

Intuition for Deep Neural Networks

- Assume we want to determine whether an image shows a human face or not.
- Suppose we are going to design a network **by hand**, choosing appropriate weights and biases (not using a learning algorithm).
- Then, **a heuristic** we could use is to **decompose the problem into sub-problems**:
 1. Does the image have an eye in the top left?
 2. Does it have an eye in the top right?
 3. Does it have a nose in the middle?
 4. Does it have a mouth in the bottom middle?
 5. Is there hair on top? ...
- Of course, this is just a rough heuristic, and it suffers from many deficiencies.
 - Maybe the person is bald, so they have no hair.
 - Maybe we can only see part of the face, or the face is at an angle, so some of the facial features are obscured.
- Still, the heuristic suggests that if we can solve the sub-problems using neural networks, then we can build a neural network for face-detection, by combining the networks for the sub-problems.

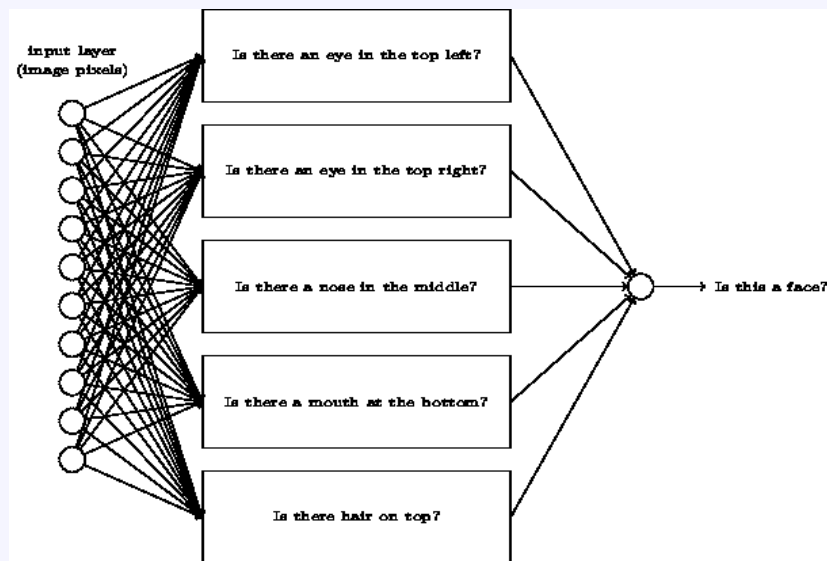


Figure 9.8: A possible architecture for face-detection.

- It is also plausible that the sub-problems/sub-networks can be decomposed. For example, for the question: *Is there an eye in the top left?*
 1. Is there an eyebrow?
 2. Are there eyelashes?
 3. Is there an iris? ...
- Thus, the sub-network can now be decomposed.

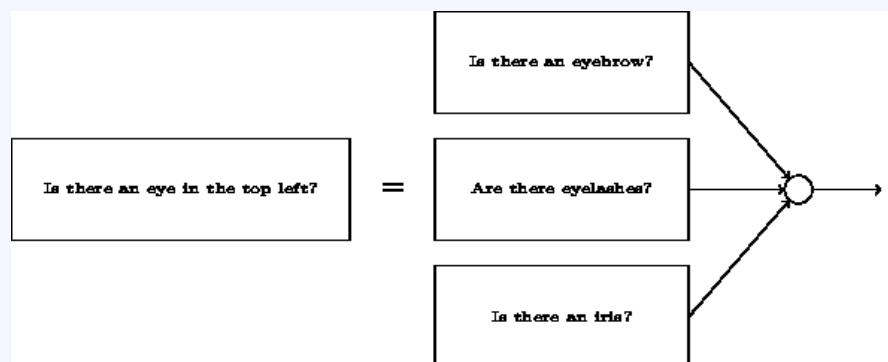


Figure 9.9: A sub-network is decomposed.

- Those questions also can be broken down, further and further, through multiple layers.
 - Ultimately, we'll be working with sub-networks that answer **questions so simple** they can easily be answered at the level of single pixels.
 - Those questions might, for example, be about the presence or absence of very simple shapes at particular points in the image.
 - Such questions can be answered by single neurons connected to the raw pixels in the image.

Summary: Deep Neural Networks

- A network breaks down **a complicated question** into **a series of simple questions** answerable at the level of single pixels.
- It does this through **a series of many layers**, with **early layers** answering very simple and specific questions about the input image, and **later layers** building up a hierarchy of ever more complex and abstract concepts.
- Networks with this kind of many-layer structure – two or more hidden layers - are called **deep neural networks**.

Final Remarks

- Researchers in the 1980s and 1990s tried using stochastic gradient descent and back-propagation to train deep networks.
- Unfortunately, except for a few special architectures, they didn't have much luck.
- The networks would learn, but very slowly, and in practice often too slowly to be useful.
- **Since 2006, a set of new techniques** has been developed that **enable learning in deep neural networks**.

9.3. Back-Propagation

- **In the previous section**, we saw an example of neural networks that could learn their weights and biases using the stochastic gradient descent algorithm.
- **In this section**, we will see **how to compute the gradient**, more precisely, the derivatives of the cost function with respect to weights and biases in all layers.
- ***The back-propagation is a practical application of the chain rule*** for the computation of derivatives.
- The back-propagation algorithm was originally introduced in the 1970s, but its importance was not fully appreciated until a famous 1986 paper by Rumelhart-Hinton-Williams [65], in *Nature*.

9.3.1. Notations

- Let's begin with notations which let us refer to weights, biases, and activations in the network in an unambiguous way.

w_{jk}^ℓ : the **weight** for the connection from the k -th neuron in the $(\ell - 1)$ -th layer to the j -th neuron in the ℓ -th layer

b_j^ℓ : the **bias** of the j -th neuron in the ℓ -th layer

a_j^ℓ : the **activation** of the j -th neuron in the ℓ -th layer

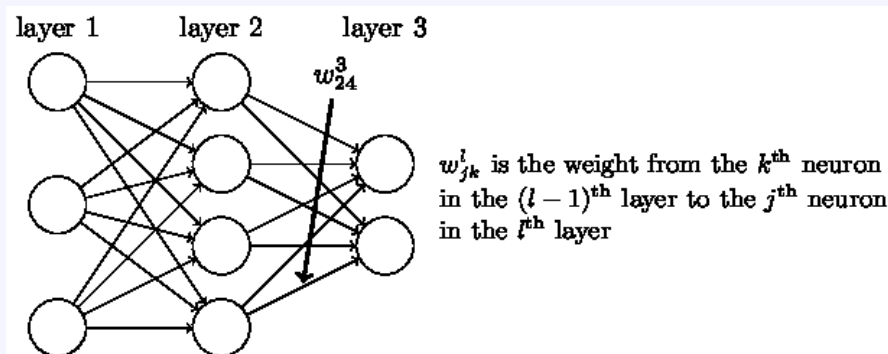


Figure 9.10: The weight w_{jk}^ℓ .

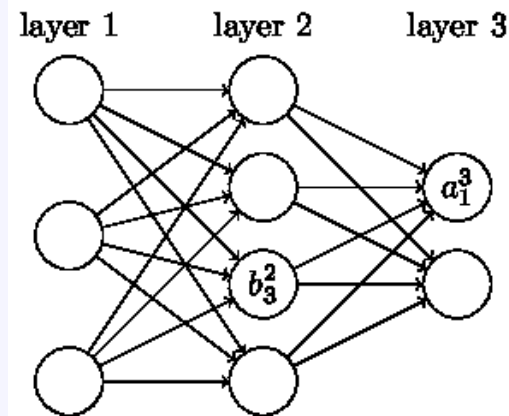


Figure 9.11: The bias b_j^ℓ and activation a_j^ℓ .

- With these notations, the activation a_j^ℓ reads

$$a_j^\ell = \sigma \left(\sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell \right), \quad (9.10)$$

where the sum is over all neurons k in the $(\ell - 1)$ -th layer. Denote the **weighted input** by

$$z_j^\ell := \sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell. \quad (9.11)$$

- Now, define

$$\begin{aligned} W^\ell = [w_{jk}^\ell] &: \text{the weight matrix for layer } \ell \\ \mathbf{b}^\ell = [b_j^\ell] &: \text{the bias vector for layer } \ell \\ \mathbf{z}^\ell = [z_j^\ell] &: \text{the weighted input vector for layer } \ell \\ \mathbf{a}^\ell = [a_j^\ell] &: \text{the activation vector for layer } \ell \end{aligned} \quad (9.12)$$

- Then, (9.10) can be rewritten (in a vector form) as

$$\mathbf{a}^\ell = \sigma(\mathbf{z}^\ell) = \sigma(W^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell). \quad (9.13)$$

9.3.2. The cost function

The Cost Function: With the notations, the quadratic cost function (9.6) has the form

$$C = \frac{1}{2N} \sum_{\mathbf{x}} \|\mathbf{y}(\mathbf{x}) - \mathbf{a}^L(\mathbf{x})\|^2, \quad (9.14)$$

where N is the total number of training examples, $\mathbf{y}(\mathbf{x})$ is the corresponding desired output for the training example $\mathbf{x}$, and L denotes the number of layers in the network.

Two Assumptions for the Cost Function

1. The cost function can be written as an average

$$C = \frac{1}{N} \sum_{\mathbf{x}} C_{\mathbf{x}}, \quad (9.15)$$

over cost functions $C_{\mathbf{x}}$ for individual training examples $\mathbf{x}$.

2. The cost function can be written as a function of the outputs from the neural network ($\mathbf{a}^L$).

Remark 9.1. Thus the cost function in (9.14) satisfies the assumptions, with

$$C_{\mathbf{x}} = \frac{1}{2} \|\mathbf{y}(\mathbf{x}) - \mathbf{a}^L(\mathbf{x})\|^2 = \frac{1}{2} \sum_j (y_j(\mathbf{x}) - a_j^L(\mathbf{x}))^2. \quad (9.16)$$

- The reason we need the first assumption is because what the back-propagation actually lets us do is **compute the partial derivatives** $\partial C_{\mathbf{x}} / \partial w_{jk}^\ell$ and $\partial C_{\mathbf{x}} / \partial b_j^\ell$ for a single training example.
- We then **can recover** $\partial C / \partial w_{jk}^\ell$ **and** $\partial C / \partial b_j^\ell$ **by averaging over training examples.**
- With this assumption in mind, we may focus on computing the partial derivatives for a single example (fixed), as given in (9.16).

The Hadamard product / Schur product

Definition 9.2. A frequently used algebraic operation is the **element-wise product** of two vectors/matrices, which is called the **Hadamard product** or the **Schur product**, and defined as

$$(\mathbf{c} \odot \mathbf{d})_j = c_j d_j. \quad (9.17)$$

- For example,

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 3 \\ 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}. \quad (9.18)$$

- In Numpy, $A*B$ denotes the Hadamard product of A and B , while $A \cdot \text{dot}(B)$ or $A@B$ produces the regular matrix-matrix multiplication.

9.3.3. The four fundamental equations behind the back-propagation

The back-propagation is about understanding *how changing the weights and biases in a network changes the cost function*, which means computing the partial derivatives $\partial C / \partial w_{jk}^\ell$ and $\partial C / \partial b_j^\ell$.

Definition 9.3. Define the **learning error** (or, **error**) of neuron j in layer ℓ by

$$\delta_j^\ell \stackrel{\text{def}}{=} \frac{\partial C}{\partial z_j^\ell}. \quad (9.19)$$

The back-propagation will give us a way of computing $\delta^\ell = [\delta_j^\ell]$ for every layer ℓ , and then relating those errors to the quantities of real interest, $\partial C / \partial w_{jk}^\ell$ and $\partial C / \partial b_j^\ell$.

Theorem 9.4. Suppose that the cost function C satisfies the two assumptions in Section 9.3.2 so that it represents the cost for a single training example. Assume the network contains L layers, of which the feed-forward model is given as in (9.10):

$$a_j^\ell = \sigma(z_j^\ell), \quad z_j^\ell = \sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell; \quad \ell = 2, 3, \dots, L.$$

Then,

$$\begin{aligned} \text{(a)} \quad & \delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L), \\ \text{(b)} \quad & \delta_j^\ell = \sum_k w_{kj}^{\ell+1} \delta_k^{\ell+1} \sigma'(z_j^\ell), \quad \ell = L-1, \dots, 2, \\ \text{(c)} \quad & \frac{\partial C}{\partial b_j^\ell} = \delta_j^\ell, \quad \ell = 2, \dots, L, \\ \text{(d)} \quad & \frac{\partial C}{\partial w_{jk}^\ell} = a_k^{\ell-1} \delta_j^\ell, \quad \ell = 2, \dots, L. \end{aligned} \tag{9.20}$$

Proof. Here, we will prove (b) only; see Exercise 1 for the others. Using the definition (9.19) and the chain rule, we have

$$\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell} = \sum_k \frac{\partial C}{\partial z_k^{\ell+1}} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_k \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} \delta_k^{\ell+1} \tag{9.21}$$

Note

$$z_k^{\ell+1} = \sum_i w_{ki}^{\ell+1} a_i^\ell + b_k^{\ell+1} = \sum_i w_{ki}^{\ell+1} \sigma(z_i^\ell) + b_k^{\ell+1}. \tag{9.22}$$

Differentiating it, we obtain

$$\frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_i w_{ki}^{\ell+1} \frac{\partial \sigma(z_i^\ell)}{\partial z_j^\ell} = w_{kj}^{\ell+1} \sigma'(z_j^\ell). \tag{9.23}$$

Substituting back into (9.21), we complete the proof. $\square$

Remark 9.5. (9.20) can be written in a vector form as

$$\begin{aligned} \text{(a)} \quad & \delta^L = \nabla_{\mathbf{a}^L} C \odot \sigma'(\mathbf{z}^L), \\ \text{(b)} \quad & \delta^\ell = ((W^{\ell+1})^T \delta^{\ell+1}) \odot \sigma'(\mathbf{z}^\ell), \quad \ell = L-1, \dots, 2, \\ \text{(c)} \quad & \nabla_{\mathbf{b}^\ell} C = \delta^\ell, \quad \ell = 2, \dots, L, \\ \text{(d)} \quad & \nabla_{W^\ell} C = \delta^\ell (\mathbf{a}^{\ell-1})^T, \quad \ell = 2, \dots, L. \end{aligned} \tag{9.24}$$

Remarks 9.6. Saturated Neurons

- The four fundamental equations satisfy independently of choices of **the cost function C and the activation σ** .
 - A consequence of (9.24.d) is that if $a^{\ell-1}$ is small (in modulus), gradient term $\partial C / \partial W^\ell$ will also tend to be small. In this case, we'll say the weight **learns slowly**, meaning that it's not changing much during gradient descent.
 - In other words, a consequence of (9.24.d) is that **weights output from low-activation neurons learn slowly**.
 - The sigmoid function σ becomes very flat when $\sigma(z_j^L)$ is approximately 0 or 1. When this occurs we will have $\sigma'(z_j^L) \approx 0$. So, a weight in the final layer will learn slowly if the output neuron is either low activation (≈ 0) or high activation (≈ 1). In this case, we usually say **the output neuron has saturated** and, as a result, the weight is learning slowly (or stopped).
 - Similar remarks hold also in other layers and for the biases as well.
-
- **Summing up**, weights and biases will learn slowly if
 - either **the in-neurons (upwind) are in low-activation**
 - or **the out-neurons (downwind) have saturated**.

Designing Activation Functions

The four fundamental equations can be used to design activation functions which have **particular desired learning properties**.

- For example, suppose we were to choose a (non-sigmoid) activation function σ so that σ' **is always positive, and never gets close to zero**.
- That would **prevent the slow-down of learning** that occurs when ordinary sigmoid neurons saturate.
- **Learning accuracy and efficiency** can be improved by finding **more effective cost and activation functions**.

The Back-Propagation Algorithm

Algorithm 9.7. Let's summarize the **back-propagation algorithm**:

1. **Input $\mathbf{x}$:** Set the corresponding activation $\mathbf{a}^1$ for the input layer.
2. **Feed-forward:**

$$\mathbf{z}^\ell = W^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell; \quad \mathbf{a}^\ell = \sigma(\mathbf{z}^\ell); \quad \ell = 2, 3, \dots, L$$

3. **Output error δ^L :**

$$\delta^L = \nabla_{\mathbf{a}^L} C \odot \sigma'(\mathbf{z}^L);$$

4. **Back-propagate the error:**

$$\delta^\ell = ((W^{\ell+1})^T \delta^{\ell+1}) \odot \sigma'(\mathbf{z}^\ell); \quad \ell = L-1, \dots, 2$$

5. **The gradient of the cost function:**

$$\nabla_{\mathbf{b}^\ell} C = \delta^\ell; \quad \nabla_{W^\ell} C = \delta^\ell (\mathbf{a}^{\ell-1})^T; \quad \ell = 2, \dots, L$$

An SDG Learning Step, Based on a Mini-batch

1. Input a set of training examples of size m ;
2. Initialize: $\Delta W = 0$ and $\Delta B = 0$;
3. For each training example $\mathbf{x}$:
 - (a) Apply the back-propagation algorithm to find

$$\nabla_{\mathbf{b}^\ell} C_{\mathbf{x}} = \delta^{\mathbf{x},\ell}; \quad \nabla_{W^\ell} C_{\mathbf{x}} = \delta^{\mathbf{x},\ell} (\mathbf{a}^{\mathbf{x},\ell-1})^T; \quad \ell = 2, \dots, L$$

- (b) Update the gradient:

$$\begin{aligned} \Delta B &= \Delta B + [\nabla_{\mathbf{b}^2} C_{\mathbf{x}} | \dots | \nabla_{\mathbf{b}^L} C_{\mathbf{x}}]; \\ \Delta W &= \Delta W + [\nabla_{W^2} C_{\mathbf{x}} | \dots | \nabla_{W^L} C_{\mathbf{x}}]; \end{aligned}$$

4. Gradient descent: Update the biases and weights

$$B = B - \frac{\eta}{m} \Delta B; \quad W = W - \frac{\eta}{m} \Delta W;$$

See lines 55–69 in `network.py` shown in Section 9.2.3, p. 274.

9.4. Deep Learning: Convolutional Neural Networks

In this section, we will consider **deep neural networks**; the focus is on understanding core fundamental principles behind them, and applying those principles in the simple, easy-to-understand context of the MNIST problem.

Example 9.8. Consider neural networks for the classification of handwritten digits, as shown in the following images:

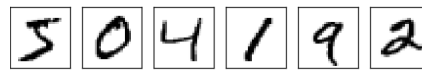


Figure 9.12: A few images in the MNIST data set.

A neural network can be built, with three hidden layers, as follows:

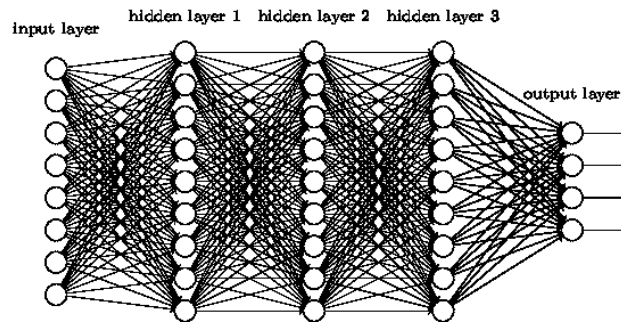


Figure 9.13

- Let each hidden layers have 30 neurons:
 - $n_weights = 28^2 \cdot 30 + 30 \cdot 30 + 30 \cdot 30 + 30 \cdot 10 = 25,620$
 - $n_biases = 30 + 30 + 30 + 10 = 100$
- Optimization is difficult
 - The number of parameters to teach is huge (**low efficiency**)
 - Multiple local minima problem (**low solvability**)
 - Adding hidden layers is **not necessarily improving** accuracy

In **fully-connected networks**, **deep** neural networks have been **hardly practical**, except for some special applications.

9.4.1. Introducing convolutional networks

Remarks 9.9. The neural network exemplified in Figure 9.13 can produce a **classification accuracy better than 98%**, for the MNIST handwritten digit data set. **But upon reflection, it's strange to use networks with fully-connected layers to classify images.**

- The network architecture does not take into account the **spatial structure of the images**.
- For instance, it treats **input pixels** which are far apart and close together, **on exactly the same footing**.

What if, instead of starting with a network architecture which is tabula rasa (blank mind), we used an architecture which tries to **take advantage of the spatial structure**? Could it be **better than 99%**?

Here, we will introduce **convolutional neural networks (CNN)**, which use a special architecture which is *particularly well-adapted to classify images*.

- The architecture makes convolutional networks **fast to train**.
 - This, in turn, **helps train deep, many-layer networks**.
 - Today, deep CNNs or some close variants are used in most neural networks for **image recognition**.
-
- CNNs use three basic ideas:
 - (a) **local receptive fields**,
 - (b) **shared weights and biases**, &
 - (c) **pooling**.

(a) Local Receptive Fields

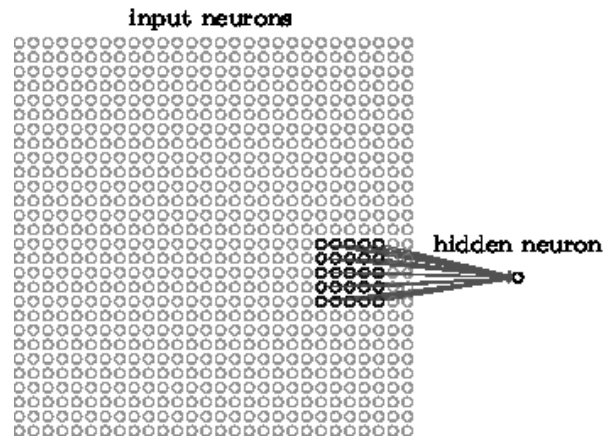


Figure 9.14: An illustration for local receptive fields.

- In CNNs, **the geometry of neurons (units) in the input layer** is exactly the same as that of images (e.g., 28×28). (rather than a vertical line of neurons as in fully-connected networks)
- As per usual, we'll **connect** the input neurons (pixels) to a layer of hidden neurons.
 - But we will **not connect fully** from every input pixel to every hidden neuron.
 - Instead, we only make connections in **small, localized regions** of the input image.
 - **For example:** Each neuron in the first hidden layer will be connected to a small region of the input neurons, say, a 5×5 region (Figure 9.14).
- That region in the input image is called the **local receptive field** for the hidden neuron.

- We slide the local receptive field across the entire input image.
 - For each local receptive field, there is a different hidden neuron in the first hidden layer.

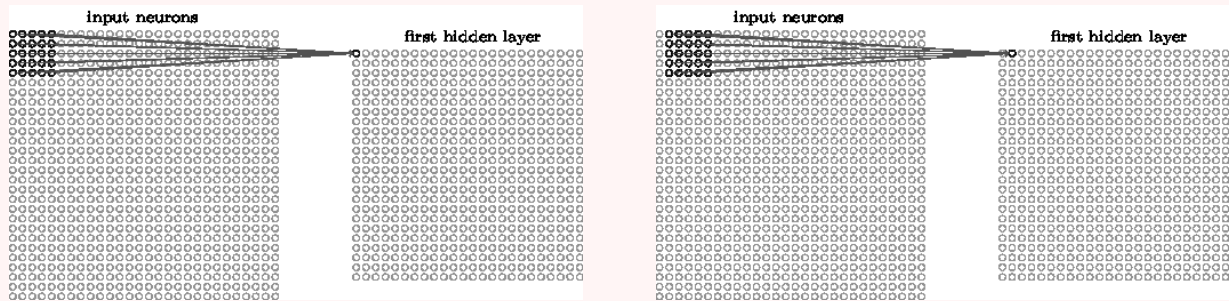


Figure 9.15: Two of local receptive fields, starting from the top-left corner.
(Geometry of neurons in the first hidden layer is 24×24 .)

Note: We have seen that the local receptive field is moved by one pixel at a time (`stride_length=1`).

- In fact, sometimes a different **stride length** is used.
 - For instance, we might move the local receptive field 2 pixels to the right (or down).
 - Most software gives a hyperparameter for the user to set the stride length.

(b) Shared Weights and Biases

Recall that each hidden neuron has a **bias** and 5×5 **weights** connected to its corresponding local receptive field.

- In CNNs, we use **the same weights and bias** for each of the 24×24 hidden neurons. In other words, for the (j, k) -th hidden neuron, the output is:

$$\sigma\left(b + \sum_{p=0}^4 \sum_{q=0}^4 w_{p,q} a_{j+p, k+q}\right), \quad (9.25)$$

where σ is the neural activation function (e.g., the sigmoid function), b is the shared value for the bias, and $w_{p,q}$ is a 5×5 array of shared weights.

- The weighting in (9.25) is just a form of **convolution**; we may rewrite it as

$$\mathbf{a}^1 = \sigma(b + \mathbf{w} * \mathbf{a}^0). \quad (9.26)$$

- So the network is called a **convolutional network**.
- We sometimes call the map, from the input layer to the hidden layer, a **feature map**.
 - Suppose **the weights and bias** are such that the hidden neuron can **pick out a feature** (e.g., a vertical edge) in a particular local receptive field.
 - That ability is also likely to be useful at other places in the image.
 - And therefore it is useful to apply **the same feature detector** everywhere in the image.
- We call the weights and bias defining the feature map the **shared weights** and the **shared bias**, respectively.
- A set of the shared weights and bias defines clearly a **kernel or filter**.

To put it in slightly more abstract terms, CNNs are well adapted to the **translation invariance** of images.^a

^aMove a picture of a cat a little ways, and it's still an image of a cat.

Remark 9.10. Multiple feature maps.

- The network structure we have considered so far can detect just **a single localized feature**.
- To do **more effective image recognition**, we'll need more than one **feature map**.
- Thus, a complete convolutional layer consists of **several different feature maps**:

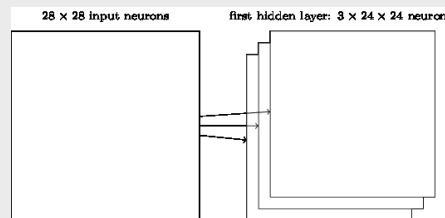
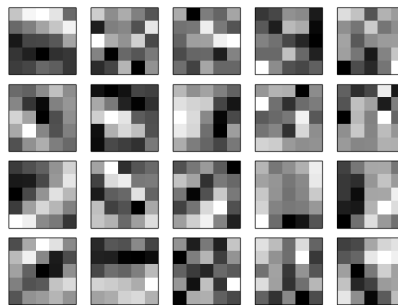


Figure 9.16: A convolutional network, consisting of 3 feature maps.

Modern CNNs are often built with 10 to 50 feature maps, each associated to a $r \times r$ local receptive field: $r = 3 \sim 9$.

Figure 9.17: The 20 images corresponding to 20 different feature maps, which are actually learned when classifying the MNIST data set ($r = 5$).**The number of parameters to learn**

A big advantage of sharing weights and biases is that it greatly reduces the number of parameters involved in a convolutional network.

- Convolutional networks: $(5 \times 5 + 1) * 20 = 520$
- Fully-connected networks: $(28 \times 28 + 1) * 20 = 15,700$

(c) Pooling

- CNNs also contain **pooling layers**, in addition to the convolutional layers just mentioned.
- Pooling layers are usually used **right after convolutional layers**.
- What they do is **to simplify the information** in the output from the convolutional layer.

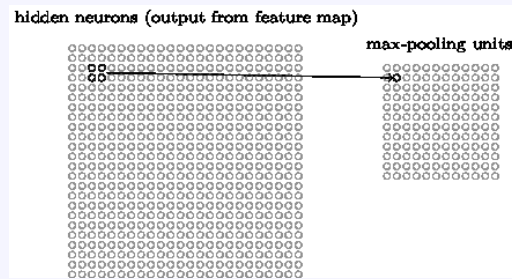


Figure 9.18: Pooling: summarizing a region of 2×2 neurons in the convolutional layer.

From Figure 9.16: Since we have 24×24 neurons output from the convolutional layer, after pooling we will have 12×12 neurons for each feature map:

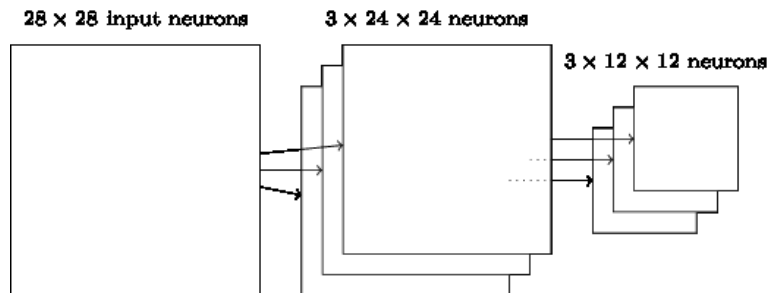


Figure 9.19: A convolutional network, consisting of 3 feature maps and pooling.

Types of pooling

1. **max-pooling**: simply outputs the maximum activation in the 2×2 input neurons.
2. **L^2 -pooling**: outputs the L^2 -average of the 2×2 input neurons.
- ...

9.4.2. CNNs, in practice

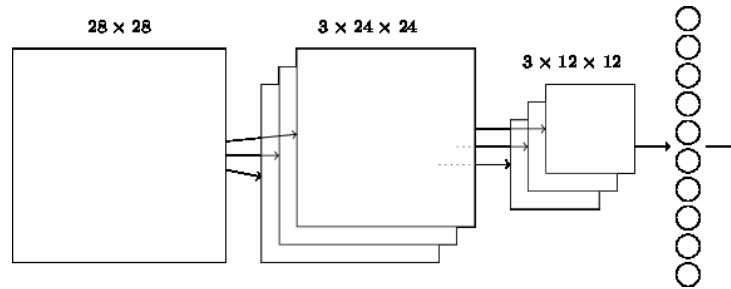


Figure 9.20: A *simple* CNN of three feature maps, to classify MNIST digits.

- To form a complete CNN by putting all these ideas, we need to **add some extra layers**, below the convolution-pooling layers.
- Figure 9.20 shows a CNN that involves an extra layer of 10 output neurons, for the 10 possible values for MNIST digits ('0', '1', '2', etc.).
- The final layer of connections is a fully-connected layer. For example:
 - Let $\text{filter_shape} = (20, 1, 5, 5)$, $\text{poolsize} = (2, 2)$
(20 feature maps; $1 \times 5 \times 5$ kernel; 2×2 pooling)
 - Then, the number of parameters to teach:
 $(5^2 + 1) \cdot 20 + (20 \cdot 12^2) \cdot 10 = 29,300$.
 - Classification accuracy for the MNIST data set $\lesssim 99\%$
- Add a **second convolution-pooling layer**:
 - Its input is the output of the first convolution-pooling layer.
 - Let $\text{filter_shape} = (40, 20, 5, 5)$, $\text{poolsize} = (2, 2)$
 - The output of the second convolution-pooling layer: $40 \times 5 \times 5$
 - Then, the number of parameters to teach:
 $(5^2 + 1) \cdot 20 + (5^2 + 1) \cdot 40 + (40 \cdot 5^2) \cdot 10 = 11,560$
 - Classification accuracy for the MNIST data set $\gtrsim 99\%$
- Add a **fully-connected layer** (up the output layer):
 - Let choose 40 neurons: $\implies 41,960$ parameters
 - Classification accuracy $\approx 99.5\%$
- Use an **ensemble of networks**
 - Using 5 CNNs, classification accuracy = **99.67% (33 missed!)**

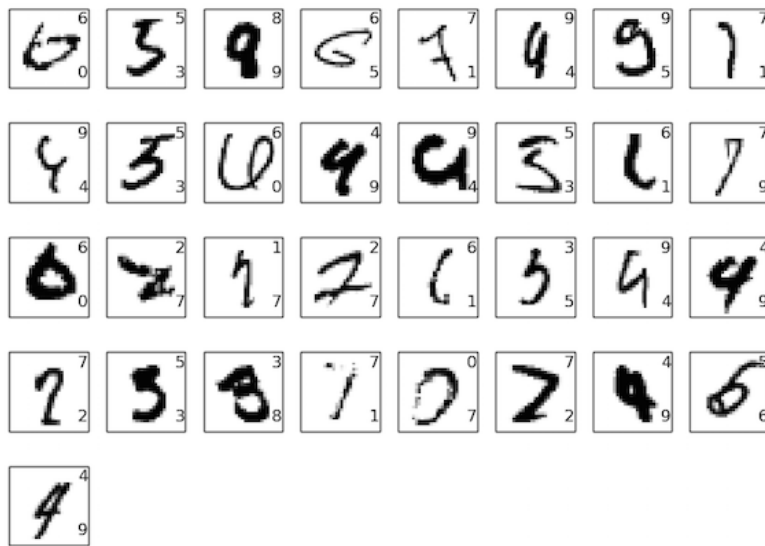


Figure 9.21: The images missed by an ensemble of 5 CNNs. The label in the top right is the correct classification, while in the bottom right is the label classified output.

Remarks 9.11. Intuitively speaking:

- **(Better representation).** The use of **translation invariance** by the convolutional layer will reduce the number of parameters it needs to get the same performance as the fully-connected model.
 - **(Convolution kernels).** The filters try to detect **localized features**, producing **feature maps**.
 - **(Efficiency).** **Pooling** simplifies the information in the output from the convolutional layer.
 - That, in turn, will result in **faster training** for the convolutional model, and, ultimately, will help us **build deep networks** using convolutional layers.
-
- **Fully-connected hidden layers** try to collect information for **more widely formed features**.

Exercises for Chapter 9

- 9.1. Complete proof of Theorem 9.4. **Hint:** The four fundamental equations in (9.20) can be obtained by simple applications of the chain rule.
- 9.2. The core equations of back-propagation in a network with fully-connected layers are given in (9.20). Suppose we have a network containing a convolutional layer, a max-pooling layer, and a fully-connected output layer, as in the network shown in Figure 9.20. How are the core equations of back-propagation modified?
- 9.3. **(Designing a deep network).** First, download a CNN code (including the MNIST data set) by accessing to

<https://github.com/mnielsen/neural-networks-and-deep-learning.git>

or ‘git clone’ it. In the ‘src’ directory, there are 8 python source files:

```
conv.py  mnist_average_darkness.py  mnist_svm.py  network2.py
expand_mnist.py  mnist_loader.py      network.py  network3.py
```

On lines 16–22 in Figure 9.22 below, I put a design of a CNN model, which involved 2 hidden layers, one for a convolution-pooling layer and the other for a fully-connected layer. Its test accuracy becomes approximately 98.8% in 30 epochs.

- (a) Set ‘GPU = False’ in network3.py, if you are NOT using a GPU.
(Default: ‘GPU = True’, set on line 50-some of network3.py)
- (b) Modify Run_network3.py appropriately to design a CNN model as accurate as possible. Can your network achieve an accuracy better than 99.5%? **Hint:** You may keep using the SoftmaxLayer for the final layer. **ReLU** (Rectified Linear Units) seems comparable with the sigmoid function (default) for activation. The default p_dropout=0.0. You should add some more layers, meaningful, and tune well all of hyperparameters: the number of feature maps for convolutional layers, the number of fully-connected neurons, η , p_dropout, etc..
- (c) Design an ensemble of 5 such networks to further improve the accuracy. **Hint:** Explore the function ‘ensemble’ defined in conv.py.

```
Run_network3.py
1  """ Run_network3.py:
2      -----
3      A CNN model, for the MNIST data set,
4      which uses network3.py written by Michael Nielsen.
5      The source code can be downloaded from
6          https://github.com/mnielsen/neural-networks-and-deep-learning.git
7      or 'git clone' it
8  """
9  import network3
10 from network3 import Network, ReLU
11 from network3 import ConvPoolLayer, FullyConnectedLayer, SoftmaxLayer
12
13 training_data, validation_data, test_data = network3.load_data_shared()
14
15 mini_batch_size = 10
16 net = Network([
17     ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
18                   filter_shape=(20, 1, 5, 5),
19                   poolsize=(2, 2), activation_fn=ReLU),
20     FullyConnectedLayer(
21         n_in=20*12*12, n_out=100, activation_fn=ReLU, p_dropout=0.0),
22     SoftmaxLayer(n_in=100, n_out=10, p_dropout=0.5)], mini_batch_size)
23
24 n_epochs, eta = 30, 0.1
25 net.SGD(training_data, n_epochs, mini_batch_size, eta, \
26         validation_data, test_data)
```

Figure 9.22: Run_network3.py

CHAPTER 10

Data Mining

Contents of Chapter 10

10.1. Introduction to Data Mining	300
10.2. Vectors and Matrices in Data Mining	304
10.3. Text Mining	312
10.4. Eigenvalue Methods in Data Mining	322
Exercises for Chapter 10	332

10.1. Introduction to Data Mining

Why Mine Data?

Commercial Viewpoint

- Lots of data is being collected and warehoused.
 - Web data, e-commerce
 - Purchases at department/grocery stores
 - Bank/Credit Card transactions
- Computers have become cheaper and more powerful.
- Competitive pressure is strong.
 - Provide better, customized services for an edge (e.g. in Customer Relationship Management)

Scientific Viewpoint

- Data collected and stored at enormous speeds (GB/hour)
 - Remote sensors on a satellite
 - Telescopes scanning the skies
 - Microarrays generating gene expression data
 - Scientific simulations generating terabytes of data
- Traditional techniques infeasible for raw data
- Data mining may help scientists
 - in classifying and segmenting data
 - in **Hypothesis Formation**

Mining Large Data Sets - Motivation

- There is often information “hidden” in the data that is not readily evident.
- Human analysts may take weeks to discover useful information.
- Much of the data is never analyzed at all.
 - **Data gap** becomes larger and larger.

What is Data Mining?

- **Data mining** is a process to turn raw data into useful information/patterns.
 - Non-trivial extraction of implicit, previously unknown and potentially useful information from data.
 - Exploration & analysis, by automatic or semi-automatic means, of large quantities of data in order to discover meaningful patterns.
 - **Patterns must be: valid, novel, understandable, and potentially useful.**

Note: Data mining is also called **Knowledge Discovery in Data (KDD)**.

Origins of Data Mining

- Draws ideas from machine learning/AI, pattern recognition, statistics, and database systems.
- Traditional Techniques may be unsuitable, due to
 - Enormity of data
 - High dimensionality of data
 - Variety: Heterogeneous, distributed nature of data

Data Mining Methods

- **Prediction Methods**
 - Use some variables to predict unknown or future values of other variables.
- **Description Methods**
 - Find human-interpretable patterns that describe the data.

Data Mining Tasks

- **Classification** [Predictive]
 - Given a collection of records (training set), find a **model** for class attribute as a function of the values of other attributes.
- **Regression** [Predictive]
 - Predict a value of a given continuous valued variable based on the values of other variables, assuming a linear or nonlinear model of dependency.
- **Clustering** [Descriptive]
 - Given a set of data points and a similarity measure among them, find clusters such that data points in one cluster are more similar to one another than points in other clusters.
- **Association Rule Discovery** [Descriptive]
 - Given a set of records each of which contain some number of items from a given collection, produce dependency rules which will predict occurrence of an item based on occurrences of other items.
- **Sequential Pattern Discovery** [Descriptive]
 - Given is a set of objects, with each object associated with its own timeline of events, find rules that predict strong sequential dependencies among different events.
- **Deviation/Anomaly Detection** [Predictive]
 - Detect significant deviations from normal behavior.

Challenges in Data Mining

- Scalability
- Dimensionality
- Complex and Heterogeneous Data
 - Spatial and temporal data
 - Point and interval data
 - Categorical data
 - Graph data
 - semi/un-structured Data
- Data Quality
- Data Ownership and Distribution
- Privacy Preservation
- Streaming Data

Related Fields

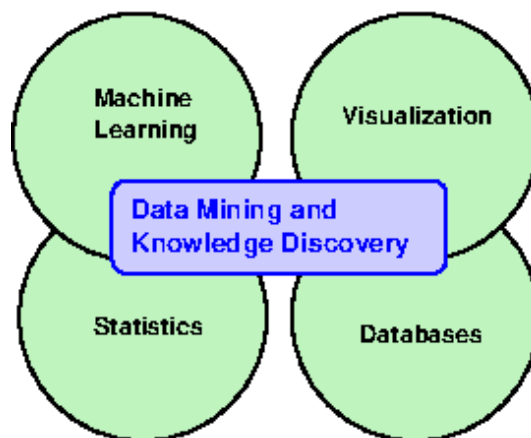


Figure 10.1: Related fields.

10.2. Vectors and Matrices in Data Mining

Note: Often the data are numerical, and the data points can be thought of as belonging to a high-dimensional vector space. Ensembles of data points can then be organized as matrices. In such cases it is natural to use concepts and techniques from linear algebra. Here, we present *Numerical Linear Algebra in Data Mining*, following and modifying (Eldén, 2006) [17].

10.2.1. Examples

Example 10.1. Term-document matrices are used in information retrieval. Consider the following set of five documents. Key words, referred to as **terms**, are marked in boldface.

-
- Document 1: The **Google matrix** P is a model of the **Internet**.
 Document 2: P_{ij} is nonzero if there is a **link** from **web page** j to i .
 Document 3: The **Google matrix** is used to **rank** all **web pages**.
 Document 4: The **ranking** is done by solving a **matrix eigenvalue** problem.
 Document 5: **England** dropped out of the top 10 in the **FIFA ranking**.
-

- Counting the frequency of terms in each document we get the following result.

Term	Doc 1	Doc 2	Doc 3	Doc 4	Doc 5
eigenvalue	0	0	0	1	0
England	0	0	0	0	1
FIFA	0	0	0	0	1
Google	1	0	1	0	0
Internet	1	0	0	0	0
link	0	1	0	0	0
matrix	1	0	1	1	0
page	0	1	1	0	0
rank	0	0	1	1	1
web	0	1	1	0	0

The set of terms is called the **dictionary**.

- Each document is represented by a vector in $\mathbb{R}^{10}$, and we can organize the data as a **term-document matrix**,

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{10 \times 5}. \quad (10.1)$$

- Assume that we want to find all documents that are relevant with respect to the query “ranking of web pages”. This is represented by a query vector, constructed in an analogous way as the term-document matrix, using the same dictionary.

$$\mathbf{q} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \in \mathbb{R}^{10}. \quad (10.2)$$

Thus the query itself is considered as a document.

- Now, the information retrieval task can be formulated as a mathematical problem: *find the columns of A that are close to the vector $\mathbf{q}$.*
 - To solve this problem we use some distance measure in $\mathbb{R}^{10}$.

Remark 10.2. Information Retrieval, with term-document matrix $A \in \mathbb{R}^{m \times n}$.

- It is common that **m is large**, of the order 10^6 , say.
- **The matrix A is sparse**, because most of the documents only contain a small fraction of the terms in the dictionary.
- In some methods for information retrieval, linear algebra techniques, such as **singular value decomposition (SVD)**, are used for **data compression** and **retrieval enhancement**.

Note: The very idea of data mining is to extract useful information from **large and often unstructured datasets**. Therefore

- **the methods must be efficient** and often specially designed for large problems.

Example 10.3. Google Pagerank algorithm. The task of extracting information from all the web pages available on the Internet, is performed by **search engines**.

- The core of the **Google search engine** is a matrix computation, probably the largest that is performed routinely.
- The **Google matrix** P is assumed to be of dimension of **the order billions (2005)**, and it is used as a model of (all) the web pages on the Internet.
- In the **Google Pagerank algorithm**, the problem of assigning ranks to all the web pages is formulated as a **matrix eigenvalue problem**.

The Google Pagerank algorithm

- Let all web pages be ordered from 1 to n , and let i be a particular web page.
- Then O_i will denote the set of pages that i is linked to, the **outlinks**. The number of outlinks is denoted $N_i = |O_i|$.
- The set of inlinks, denoted I_i , are the pages that have an outlink to i .
- Now define Q to be a square matrix of dimension n , and let

$$Q_{ij} = \begin{cases} 1/N_j, & \text{if there is a link from } j \text{ to } i, \\ 0, & \text{otherwise.} \end{cases} \quad (10.3)$$

- This definition means that row i has nonzero elements in those positions that correspond to inlinks of i .
 - Similarly, column j has nonzero elements equal to $1/N_j$ in those positions that correspond to the outlinks of j .
 - Thus, the sum of each column is either 0 or 1.
- The following **link graph** illustrates a set of web pages with outlinks and inlinks.

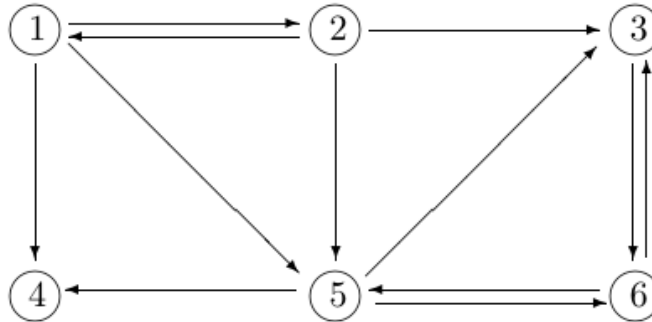


Figure 10.2: A link graph, for six web pages.

The corresponding **link matrix** becomes

$$Q = \begin{bmatrix} 0 & 1/3 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 & 1/3 & 1/2 \\ 1/3 & 0 & 0 & 0 & 1/3 & 0 \\ 1/3 & 1/3 & 0 & 0 & 0 & 1/2 \\ 0 & 0 & 1 & 0 & 1/3 & 0 \end{bmatrix} \quad (10.4)$$

- Define a **pagerank vector** r , which holds the ranks of all pages.
- Then, the vector r can be found as the eigenvector corresponding to the eigenvalue $\lambda = 1$ of Q :

$$Qr = \lambda r. \quad (10.5)$$

We discuss numerical aspects of the Pagerank computation in Section 10.4.

10.2.2. Data compression: Low rank approximation

Note: Rank Reduction.

- One way of measuring the information contents in a data matrix is to compute its rank.
- Obviously, linearly dependent column or row vectors are redundant.
- Therefore, one natural procedure for extracting information from a data matrix is **to systematically determine a sequence of linearly independent vectors**, and deflate the matrix by subtracting rank one matrices, one at a time.
- It turns out that this **rank reduction procedure** is closely related to **matrix factorization, data compression, dimensionality reduction, and feature selection/extraction**.
- The key link between the concepts is the **Wedderburn rank reduction theorem**.

Theorem 10.4. (Wedderburn, 1934) [78]. Suppose $A \in \mathbb{R}^{m \times n}$, $f \in \mathbb{R}^n$, and $g \in \mathbb{R}^m$. Then

$$\text{rank}\left(A - \frac{Afg^T A}{\omega}\right) = \text{rank}(A) - 1 \iff \omega = g^T A f \neq 0. \quad (10.6)$$

Algorithm 10.5. Wedderburn rank-reduction process.

Based on Wedderburn rank reduction theorem, a stepwise rank reduction procedure can be defined.

- Let $A^{(0)} = A$.
- Define a sequence of matrices $\{A^{(i)}\}$:

$$A^{(i+1)} = A^{(i)} - \frac{A^{(i)} f^{(i)} g^{(i)T} A^{(i)}}{\omega_i}, \quad (10.7)$$

where $f^{(i)} \in \mathbb{R}^n$ and $g^{(i)} \in \mathbb{R}^m$ such that

$$\omega_i = g^{(i)T} A^{(i)} f^{(i)} \neq 0. \quad (10.8)$$

- The sequence defined in (10.7) terminates when $r = \text{rank}(A^{(i+1)})$, since each time the rank of the matrix decreases by one. The matrices $A^{(i)}$ are called **Wedderburn matrices**.

Remark 10.6. The Wedderburn rank-reduction process gives a matrix decomposition called the **rank-reduction decomposition**.

$$A = \widehat{F} \Omega^{-1} \widehat{G}, \quad (10.9)$$

where

$$\begin{aligned} \widehat{F} &= (\mathbf{f}_1, \dots, \mathbf{f}_r) \in \mathbb{R}^{m \times r}, \quad \mathbf{f}_i = A^{(i)} f^{(i)}, \\ \Omega &= \text{diag}(\omega_1, \dots, \omega_r) \in \mathbb{R}^{r \times r}, \\ \widehat{G} &= (\mathbf{g}_1, \dots, \mathbf{g}_r) \in \mathbb{R}^{n \times r}, \quad \mathbf{g}_i = A^{(i)T} g^{(i)}. \end{aligned} \quad (10.10)$$

Theorem 10.4 can be generalized to the case where the reduction of rank is larger than one, as shown in the next theorem.

Theorem 10.7. (Guttman, 1957) [28]. Suppose $A \in \mathbb{R}^{m \times n}$, $F \in \mathbb{R}^{n \times k}$, and $G \in \mathbb{R}^{m \times k}$. Then

$$\begin{aligned} \text{rank}(A - AFR^{-1}G^T A) &= \text{rank}(A) - \text{rank}(AFR^{-1}G^T A) \\ &\iff R = G^T AF \text{ is nonsingular.} \end{aligned} \quad (10.11)$$

Note: There are many choices of F and G that satisfy the condition (10.11).

- Therefore, various rank-reduction decompositions are possible.
- It is known that several standard matrix factorizations in numerical linear algebra are instances of the Wedderburn formula:
 - Gram-Schmidt orthogonalization,
 - singular value decomposition,
 - QR and Cholesky decomposition, and
 - the Lanczos procedure.

Relation between the truncated SVD and the Wedderburn rank reduction process

- Recall the truncated SVD (7.17), page 154.
- In the rank reduction formula (10.11), define the error matrix E as

$$E = A - AFR^{-1}G^T A = A - AF(G^T AF)^{-1}G^T A, \quad (10.12)$$

where $F \in \mathbb{R}^{n \times k}$ and $G \in \mathbb{R}^{m \times k}$.

- Assume that $k \leq \text{rank}(A) = r$, and consider the problem

$$\min \|E\| = \min_{F \in \mathbb{R}^{n \times k}, G \in \mathbb{R}^{m \times k}} \|A - AF(G^T AF)^{-1}G^T A\|, \quad (10.13)$$

where the norm is **orthogonally invariant** such as the L^2 -norm and the Frobenius norm.

- According to Theorem 7.16, p.163, the minimum error is obtained when

$$(AF)(G^T AF)^{-1}(G^T A) = U\Sigma_k V^T = U_k \Sigma_k V_k^T, \quad (10.14)$$

which is equivalent to choosing

$$F = V_k \quad G = U_k. \quad (10.15)$$

10.3. Text Mining

Definition 10.8. **Text mining** is methods that extract useful information from large and often unstructured collections of texts.

- A related term is **information retrieval**.
- A typical application is search in data bases of abstract of scientific papers.
 - For instance, in medical applications one may want to find all the abstracts in the data base that deal with a particular syndrome.
 - So one puts together a search phrase, a query, with key words that are relevant for the syndrome.
 - Then the retrieval system is used to match the query to the documents in the data base, and present to the user all the documents that are relevant, preferably ranked according to relevance.

Example 10.9. The following is a typical query (Eldén, 2006) [17].

9. the use of induced hypothermia in heart surgery, neurosurgery, head injuries and infectious diseases. (10.16)

We will refer to this query as **Q9** in the sequel.

Note: Another well-known area of text mining is **web search engines**.

- There the search phrase is usually very short.
- Often there are so many relevant documents that it is out of the question to present them all to the user.
- In that application the **ranking of the search result** is critical for the efficiency of the search engine.
- We will come back to this problem in Section 10.4.

Public Domain Text Mining Software

A number of public domain software are available.

- **R**
 - **textmineR**
- **Python**
 - **nlTK** (natural language toolkit)
 - **spaCy** (written in Cython)

In this section

We will review one of the most common methods for text mining, namely the **vector space model** (Salton *et al.*, 1975) [67].

- In **Example 10.1**, we demonstrated the basic ideas of the construction of a **term-document matrix** in the vector space model.
- Below we first give a very brief overview of the **preprocessing** that is usually done before the actual term-document matrix is set up.
- Then we describe a variant of the vector space model: **Latent Semantic Indexing** (LSI) (Deerwester *et al.*, 1990) [15], which is based on the SVD of the term-document matrix.

10.3.1. Vector space model: Preprocessing and query matching

Note: In information retrieval, **keywords** that carry information about the contents of a document are called **terms**.

- A basic task is to create a list of all the terms in alphabetic order, a so called **index**.
- But before the index is made, two preprocessing steps should be done:
 - (a) removal of stop words
 - (b) stemming

Removal of Stop Words

- **Stop words** are words that one can find in virtually any document.
- The occurrence of such a word in a document does not distinguish this document from other documents.
- The following is the beginning of one stop list
a, able, about, above, according, accordingly, across, actually, after, afterwards, again, against, ain't, all, allow, allows, almost, alone, along, already, also, although, always, am, among, amongst, an, and, ...
- Various sets of stop words are available on the Internet, e.g.
<https://countwordsfree.com/stopwords>.

Stemming

- **Stemming** is the process of reducing each word that is conjugated or has a suffix to its stem.
- Clearly, from the point of view of information retrieval, no information is lost in the following reduction.

computable	}	$\Rightarrow$ comput
computation		
computing		
computed		
computational		

- Public domain stemming algorithms are available on the Internet, e.g. the **Porter Stemming Algorithm**
<https://tartarus.org/martin/PorterStemmer/>.

The Term-Document Matrix

- The **term-document matrix** $A \in \mathbb{R}^{m \times n}$, where
 - m = the number of terms in the dictionary
 - n = the number of documents
- It is common not only to **count the occurrence of terms in documents** but also to **apply a term weighting scheme**.
- Similarly, **document weighting** is usually done.

Example 10.10. For example, one can define the elements in A by

$$a_{ij} = f_{ij} \log(n/n_i), \quad (10.17)$$

where

- f_{ij} is **term frequency**,
the number of times term i appears in document j ,
- n_i is the number of documents that contain term i
(*inverse document frequency*).

If a term occurs frequently in only a few documents, then both factors are large. In this case the term discriminates well between different groups of documents, and it gets a large weight in the documents where it appears.

Normally, the term-document matrix is *sparse*: most of the matrix elements are equal to zero.

Example 10.11. For the stemmed Medline collection in Example 10.9, p.312, the matrix is 4163×1063 , with 48263 non-zero elements, i.e. approximately 1%. (It includes 30 query columns.) The first 500 rows and columns of the matrix are illustrated in Figure 10.3.

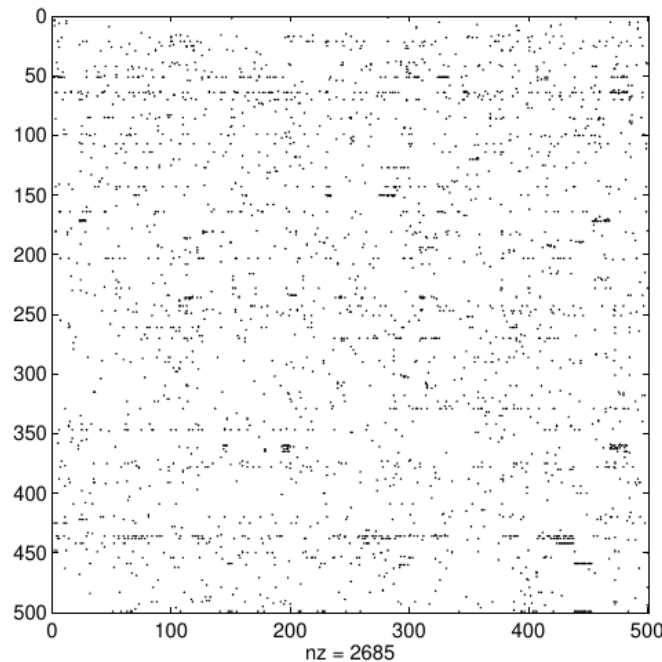


Figure 10.3: The first 500 rows and columns of the Medline matrix. Each dot represents a non-zero element.

Query Matching

- The query is parsed using the same dictionary as the documents, giving a vector $\mathbf{q} \in \mathbb{R}^m$.
- **Query matching** is the process of finding all documents that are considered relevant to a particular query $\mathbf{q}$.
- This is often done using the **cosine distance measure**: All documents $\{\mathbf{a}_j\}$ are returned for which

$$\frac{\mathbf{q} \cdot \mathbf{a}_j}{\|\mathbf{q}\| \|\mathbf{a}_j\|} \geq \text{tol}, \quad (10.18)$$

where tol is user-defined tolerance.

Example 10.12. Query matching is performed for query **Q9** in the stemmed Medline collection. With $\text{tol} = 0.19$ only a single document is considered relevant. When the tolerance was lowered to 0.17, then three documents are retrieved.

- Irrelevant documents may be returned.
 - For a high value of the tolerance, the retrieved documents are likely to be relevant.
 - When the cosine tolerance is lowered, irrelevant documents may be returned *relatively more*.

Definition 10.13. In performance modelling for information retrieval, we define the following measures:

$$P = \frac{T_r}{T_t} \text{ (precision)} \quad R = \frac{T_r}{B_r} \text{ (recall)}, \quad (10.19)$$

where

T_r = the number of relevant documents retrieved

T_t = the total number of documents retrieved

B_r = the total number of relevant documents in the database

Note: With the cosine measure:

- We see that with a large value of `tol`, we have high precision, but low recall.
- For a small value of `tol`, we have high recall, but low precision.

Example 10.14. Query matching is performed for query **Q9** in the Medline collection using the cosine measure, in order to obtain recall and precision as illustrated in Figure 10.4.

- In the comparison of different methods, it is more illustrative to draw the **recall versus precision diagram**.
- Ideally a method has high recall at the same time as the precision is high. Thus, the closer the curve is to the upper right corner, the better the method is.

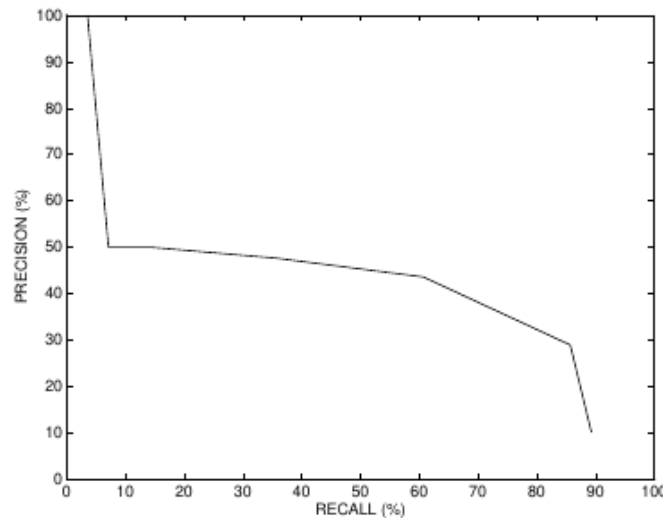


Figure 10.4: Recall versus precision diagram for query matching for Q9, using the vector space method.

10.3.2. Latent Semantic Indexing

Latent Semantic Indexing (LSI) is based on the assumption

- that there is **some underlying latent semantic structure** in the data that is corrupted by the wide variety of words used
- and that this semantic structure can be enhanced by projecting the data onto a lower-dimensional space using the **singular value decomposition**.

Algorithm 10.15. Latent Semantic Indexing (LSI)

- Let $A = U\Sigma V^T$ be the SVD of the term-document matrix.
- Let A_k be its approximation of rank k :

$$A_k = U_k \Sigma_k V_k^T = U_k (\Sigma_k V_k^T) =: U_k D_k, \quad (10.20)$$

where $V_k \in \mathbb{R}^{n \times k}$ so that $D_k \in \mathbb{R}^{k \times n}$.

- The columns of U_k live in the document space and are an orthogonal basis that we use to approximate all the documents.
- Column j of D_k holds the coordinates of document j in terms of the orthogonal basis.
- Note that

$$\mathbf{q}^T A_k = \mathbf{q}^T U_k D_k = (U_k^T \mathbf{q})^T D_k \in \mathbb{R}^{1 \times n}. \quad (10.21)$$

- Thus, in query matching, we compute the coordinates of the query in terms of the new document basis and compute the cosines from

$$\cos \theta_j = \frac{\hat{\mathbf{q}}_k \cdot (D_k \mathbf{e}_j)}{\|\hat{\mathbf{q}}_k\| \|D_k \mathbf{e}_j\|}, \quad \hat{\mathbf{q}}_k = U_k^T \mathbf{q}. \quad (10.22)$$

- This means that the query-matching is performed in a k -dimensional space.

Example 10.16. Query matching is carried out for **Q9** in the Medline collection, approximating the matrix using the truncated SVD with of rank 100 ($k = 100$). The recall-precision curve is given in Figure 10.5. It is seen that for this query, the LSI improves the retrieval performance.

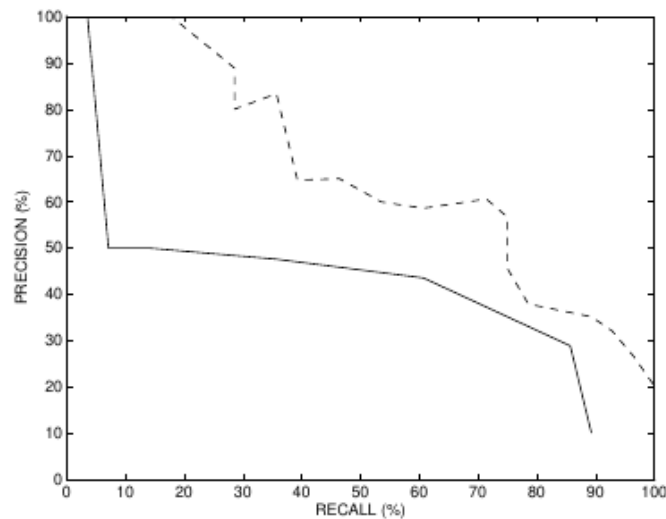


Figure 10.5: Recall versus precision diagram for query matching for Q9, using the full vector space method (solid curve) and the rank 100 approximation (dashed).

Example 10.17. Recall Example 10.1. Consider the term-document matrix $A \in \mathbb{R}^{10 \times 5}$ and the query vector $\mathbf{q} \in \mathbb{R}^{10}$, of which the query is “**ranking of web pages**”. See pages 304–305 for details.

Document 1: The **Google matrix** P is a model of the **Internet**.
 Document 2: P_{ij} is nonzero if there is a **link** from **web page** j to i .
 Document 3: The **Google matrix** is used to **rank** all **web pages**.
 Document 4: The **ranking** is done by solving a **matrix eigenvalue** problem.
 Document 5: **England** dropped out of the top 10 in the **FIFA ranking**.

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{10 \times 5}, \quad \mathbf{q} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \in \mathbb{R}^{10}.$$

- (Eldén, 2006) [17]

“**Obviously, Documents 1-4 are relevant with respect to the query, while Document 5 is totally irrelevant.** However, we obtain the following cosines for query and the original data

$$(0 \quad 0.6667 \quad 0.7746 \quad 0.3333 \quad 0.3333)$$

We then compute the SVD of the term-document matrix, and use a rank 2 approximation. After projection to the two-dimensional subspace the cosines, computed according to (10.22), are

$$(0.7857 \quad 0.8332 \quad 0.9670 \quad 0.4873 \quad 0.1819)$$

It turns out that Document 1, which was deemed totally irrelevant for the query in the original representation, is now highly relevant. In addition, the scores for the relevant Documents 2-4 have been reinforced. At the same time, the score for Document 5 has been significantly reduced.”

- **However, I view it as a warning.**

10.4. Eigenvalue Methods in Data Mining

Note: An **Internet search** performs two major operations, using a search engine.

(a) **Traditional text processing.** The aim is to find all the web pages containing the words of the query.

(b) **Sorting out.**

- Due to the massive size of the Web, the number of hits is likely to be much too large to be handled by the user.
- Therefore, some measure of quality is needed to sort out the pages that are likely to be most relevant to the particular query.

When one uses a web search engine, then typically the search phrase is under-specified.

Example 10.18. A **Google search** conducted on October 21, 2022, using the search phrase “**university**”:

- The result: links to universities, including *Mississippi State University*, *University of Arizona*, *University of Washington - Seattle*, *University of Wisconsin–Madison*, *The University of Texas at Austin*, and *University of Southern California - Los Angeles*.
- The total number of web pages relevant to the search phrase was more than 7 billions.

Remark 10.19. Google uses an algorithm (**Pagerank**) for ranking all the web pages that agrees rather well with a common-sense quality measure.

- Google assigns a high rank to a web page, if it has **inlinks** from other pages that have a high rank.
- We will see that this “**self-referencing**” statement can be formulated mathematically as an eigenvalue problem.

10.4.1. Pagerank

Note: Google uses the concept of **Pagerank** as a quality measure of web pages. It is based on the assumption that

the number of links to and from a page give information about the importance of a page.

- Let all web pages be ordered from 1 to n , and let i be a particular web page.
- Then O_i will denote the set of pages that i is linked to, the **outlinks**. The number of outlinks is denoted $N_i = |O_i|$.
- The set of **inlinks**, denoted I_i , are the pages that have an outlink to i .

Note: In general, a page i can be considered as *more important the more inlinks it has*.

- However, a ranking system based *only on the number of inlinks* is easy to manipulate.
 - When you design a web page i that you would like to be seen by as many as possible, you could simply create a large number of (information-less and unimportant) pages that have outlinks to i .
- In order to discourage this, one may define **the rank of i** in such a way that if **a highly ranked page j** , has an outlink to i , this should add to the importance of i .
- Here the manner is:

the rank of page i is a weighted sum of the ranks of the pages that have outlinks to i .

Definition 10.20. The preliminary definition of **Pagerank** is

$$r_i = \sum_{j \in I_i} \frac{r_j}{N_j}, \quad i = 1, 2, \dots, n. \quad (10.23)$$

That is, the weighting is such that the rank of a page j is **divided evenly** among its outlinks.

Remark 10.21. Pagerank may not be solvable.

- As in (10.3)-(10.4), p. 307, let Q be a square matrix of dimension n , and let

$$Q_{ij} = \begin{cases} 1/N_j, & \text{if there is a link from } j \text{ to } i, \\ 0, & \text{otherwise,} \end{cases} \quad (10.24)$$

where Q is sometimes called the **normalized web matrix**.

- Then, (10.23) can be written as

$$\lambda \mathbf{r} = Q\mathbf{r}, \quad \lambda = 1, \quad (10.25)$$

i.e., $\mathbf{r}$ is an eigenvector of Q with eigenvalue $\lambda = 1$.

- However, it is not clear that Pagerank is well-defined, because we do not know if there exists an eigenvalue equal to 1.

Reformulation of (10.23)

Modify the matrix Q to have an eigenvalue $\lambda = 1$.

- Assume that a surfer visiting a web page, always chooses the next page among the outlinks with equal probability.
- Assume that the random surfer never get stuck.
 - In other words, there should be no web pages without outlinks (such a page corresponds to a zero column in Q).
- Therefore the model is modified so that zero columns are replaced by a constant value in each position.

- Define the vectors

$$\mathbf{e} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^n, \quad \mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}, \quad d_j = \begin{cases} 1, & \text{if } N_j = 0, \\ 0, & \text{otherwise.} \end{cases} \quad (10.26)$$

- Then the modified matrix is defined

$$P = Q + \frac{1}{n} \mathbf{e} \mathbf{d}^T. \quad (10.27)$$

- Then P is a **column-stochastic matrix**, of which columns are probability vectors. That is, it has non-negative elements ($P \geq 0$) and the sum of each column is 1.
- Furthermore,

$$\mathbf{e}^T P = \mathbf{e}^T Q + \frac{1}{n} \mathbf{e}^T \mathbf{e} \mathbf{d}^T = \mathbf{e}^T Q + \mathbf{d}^T = \mathbf{e}^T, \quad (10.28)$$

which implies that $\lambda = 1$ is a **left eigenvalue** and therefore a **right eigenvalue**. Note that

$$\begin{aligned} \mathbf{e}^T P = \mathbf{e}^T &\iff P^T \mathbf{e} = \mathbf{e}, \\ \det(A - \lambda I) &= \det(A^T - \lambda I). \end{aligned} \quad (10.29)$$

-
- Now, we define the **Pagerank vector** $\mathbf{r}$ as a unique eigenvector of P with eigenvalue $\lambda = 1$,

$$P \mathbf{r} = \mathbf{r}. \quad (10.30)$$

- However, uniqueness is still not guaranteed.
 - To ensure this, the directed graph corresponding to the matrix must be **strongly connected**
 - Equivalently, in matrix terms, P must be **irreducible**.
 - Equivalently, there must not exist any subgraph, which has no outlinks.

The uniqueness of the eigenvalue is guaranteed by the **Perron-Frobenius theorem**.

Theorem 10.22. (Perron-Frobenius) *If $A \in \mathbb{R}^{n \times n}$ is nonnegative, then*

- $\rho(A)$ is an eigenvalue of A .
- There is a nonnegative eigenvector x such that $Ax = \rho(A)x$.

Theorem 10.23. (Perron-Frobenius) *If $A \in \mathbb{R}^{n \times n}$ is nonnegative and irreducible, then*

- $\rho(A)$ is an eigenvalue of A .
- $\rho(A) > 0$.
- There is a positive eigenvector x such that $Ax = \rho(A)x$.
- $\rho(A)$ is a simple eigenvalue.

Theorem 10.24. (Perron) *If $A \in \mathbb{R}^{n \times n}$ is positive, then*

- Theorem 10.23 holds, and in addition,
- $|\lambda| < \rho(A)$ for any eigenvalue λ with $\lambda \neq \rho(A)$.

Corollary 10.25. *Let A be an irreducible column-stochastic matrix. Then*

- The largest eigenvalue in magnitude is equal to 1.
- There is a unique corresponding eigenvector r satisfying $r > 0$ and $\|r\|_1 = 1$; this is the only eigenvector that is non-negative.
- If $A > 0$, then $|\lambda_i| < 1, i = 2, 3, \dots, n$.

Remark 10.26. Given the size of the Internet and reasonable assumptions about its structure,

- it is highly probable that the **link graph** is not strongly connected,
- which means that **the Pagerank eigenvector of P may not be well-defined**.

10.4.2. The Google matrix

To ensure connectedness, i.e., to make it impossible for the random walker to get trapped in a subgraph, **one can add, artificially, a link from every web page to all the other**. In matrix terms, this can be made by taking a convex combination of P and a rank one matrix.

One billion dollar idea, by **Sergey Brin** and **Lawrence Page** in 1996

- The **Google matrix** is the matrix

$$G = \alpha P + (1 - \alpha) \frac{1}{n} \mathbf{e} \mathbf{e}^T, \quad (10.31)$$

for some α satisfying $0 < \alpha < 1$, called the **damping factor**.

- Obviously G is irreducible (since $G > 0$) and column-stochastic.^a
- Furthermore,

$$\mathbf{e}^T G = \alpha \mathbf{e}^T P + (1 - \alpha) \mathbf{e}^T \frac{1}{n} \mathbf{e} \mathbf{e}^T = \alpha \mathbf{e}^T + (1 - \alpha) \mathbf{e}^T = \mathbf{e}^T. \quad (10.32)$$

- The **pagerank equation** reads

$$G \mathbf{r} = \left[\alpha P + (1 - \alpha) \frac{1}{n} \mathbf{e} \mathbf{e}^T \right] \mathbf{r} = \mathbf{r}. \quad (10.33)$$

^aA $n \times n$ matrix is called a **Markov matrix** if all entries are nonnegative and the sum of each column vector is equal to 1. A Markov matrix are also called a **stochastic matrix**.

Note: The random walk interpretation of the additional rank one term is that each time step a page is visited, the surfer will jump to any page in the whole web with probability $1 - \alpha$ (sometimes referred to as **teleportation**).

- Recall (10.27): $P = Q + \frac{1}{n}\mathbf{e}\mathbf{d}^T$, which can be interpreted as follows.

When a random surfer visits a web page of no outlinks, the surfer will jump to any page with an equal probability $1/n$.

- The convex combination in (10.31): $G = \alpha P + (1 - \alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T$.

Although there are outlinks, the surfer will jump to any page with an equal probability $(1 - \alpha)/n$.

Proposition 10.27. Let the eigenvalues of the column-stochastic matrix P be $\{1, \lambda_2, \lambda_3, \dots, \lambda_n\}$. Then, the eigenvalues of $G = \alpha P + (1 - \alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T$ are $\{1, \alpha\lambda_2, \alpha\lambda_3, \dots, \alpha\lambda_n\}$.

- This means that even if P has a multiple eigenvalue equal to 1, the second largest eigenvalue in magnitude of G is equal to α .

Remark 10.28. The vector $\frac{1}{n}\mathbf{e}$ in (10.31) can be replaced by a non-negative vector $\mathbf{v}$ with $\|\mathbf{v}\|_1 = 1$, which can be chosen in order to make the search biased towards certain kinds of web pages. Therefore, it is referred to as a **personalization vector**.

10.4.3. Solving the Pagerank equation

Now, we should solve the Pagerank equation, an eigenvalue problem

$$G\mathbf{r} = \mathbf{r}, \quad (10.34)$$

where $\mathbf{r} \geq 0$ with $\|\mathbf{r}\|_1 = 1$.

Observation 10.29. The Google matrix $G \in \mathbb{R}^{n \times n}$

- G is a full matrix, although it is not necessary to construct it explicitly.
- n represents the number of all web pages, which is order of billions.
- It is **impossible** to use **sparse eigenvalue algorithms** that require the storage of more than very few vectors.

The only viable method so far for Pagerank computations on the whole web seems to be the **power method**.

- The rate of convergence of the power method depends on the ratio of the second largest and the largest eigenvalue in magnitude.
- Here, we have

$$|1 - \lambda^{(k)}| = \mathcal{O}(\alpha^k), \quad (10.35)$$

due to Proposition 10.27.

- In view of the huge dimension of the Google matrix, it is non-trivial to compute the matrix-vector product. We will consider some details.

The power method: matrix-vector product

Recall: It follows from (10.24), (10.27), and (10.31) that the **Google matrix** is formulated as

$$G = \alpha P + (1 - \alpha) \frac{1}{n} \mathbf{e} \mathbf{e}^T, \quad (10.36)$$

where

$$P = Q + \frac{1}{n} \mathbf{e} \mathbf{d}^T.$$

Here Q is the **link matrix** and $\mathbf{e}$ and $\mathbf{d}$ are defined as in (10.26).

Derivation 10.30. Let $\mathbf{z} = G\mathbf{y}$.

- **Normalization-free:** Since G is column-stochastic ($\mathbf{e}^T G = \mathbf{e}^T$),

$$\|\mathbf{z}\|_1 = \mathbf{e}^T \mathbf{z} = \mathbf{e}^T G\mathbf{y} = \mathbf{e}^T \mathbf{y} = \|\mathbf{y}\|_1. \quad (10.37)$$

Thus, when the power method begins with $\mathbf{y}^{(0)}$ with $\|\mathbf{y}^{(0)}\|_1 = 1$, the normalization step in the power method is unnecessary.

- Let us look at the multiplication in some detail:

$$\mathbf{z} = \left[\alpha P + (1 - \alpha) \frac{1}{n} \mathbf{e} \mathbf{e}^T \right] \mathbf{y} = \alpha Q\mathbf{y} + \beta \frac{\mathbf{e}}{n}, \quad (10.38)$$

where

$$\beta = \alpha \mathbf{d}^T \mathbf{y} + (1 - \alpha) \mathbf{e}^T \mathbf{y}. \quad (10.39)$$

- Apparently we need to know which pages lack outlinks ($\mathbf{d}$), in order to find β . However, in reality, we do not need to define $\mathbf{d}$. It follows from (10.37) and (10.38) that

$$\beta = 1 - \alpha \mathbf{e}^T Q\mathbf{y} = 1 - \|\alpha Q\mathbf{y}\|_1. \quad (10.40)$$

Algorithm 10.31. The following Matlab code implements the matrix vector multiplication: $z = Gy$.

```
zhat = alpha*Q*y;  
beta = 1-norm(zhat,1);  
z = zhat + beta*v;  
residual = norm(y-z,1);
```

Here v is $(1/n)\mathbf{e}$ or a **personalization vector**; see Remark 10.28.

Note:

- From Proposition 10.27, we know that the second eigenvalue of the Google matrix is $\alpha\lambda_2$.
- A typical value of $\alpha = 0.85$.
- Approximately $k = 57$ iterations are needed to reach $0.85^k < 10^{-4}$.
- This is reported to be close the number of iterations used by Google.

Exercises for Chapter 10

- 10.1. Consider Example 10.17, p.321. Compute vectors of cosines, for each subspace approximations, i.e., with A_k where $k = 1, 2, \dots, 5$.
- 10.2. Verify equations in Derivation 10.30, p.330, particularly (10.38), (10.39), and (10.40).
- 10.3. Consider the link matrix Q in (10.4) and its corresponding link graph in Figure 10.2. Find the pagerank vector $\mathbf{r}$ by solving the Google pagerank equation.
- You may initialize the power method with any vector $\mathbf{r}^{(0)}$ satisfying $\|\mathbf{r}^{(0)}\|_1 = 1$.
 - Set $\alpha = 0.85$.
 - Let the iteration stop, when $\text{residual} < 10^{-4}$.
- 10.4. Now, consider a **modified** link matrix $\tilde{Q}$, by adding an outlink from page ④ to ⑤ in Figure 10.2. Find the pagerank vector $\tilde{\mathbf{r}}$, by setting parameters and initialization the same way as for the previous problem.
- Compare $\mathbf{r}$ with $\tilde{\mathbf{r}}$.
 - Compare the number of iterations for convergence.

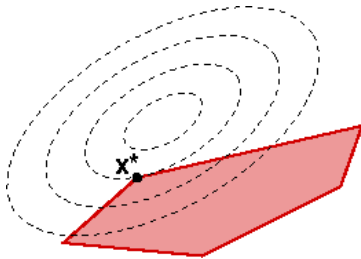
CHAPTER 11

Quadratic Programming

Quadratic programming (QP) is the process of solving a constrained quadratic optimization problem. That is, the objective f is quadratic and the constraints are linear in several variables $\mathbf{x} \in \mathbb{R}^n$. Quadratic programming is a particular type of nonlinear programming. Its **general form** is

$$\begin{aligned} \min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}) &:= \frac{1}{2} \mathbf{x}^T A \mathbf{x} - \mathbf{x}^T \mathbf{b}, \quad \text{subj.to} \\ C\mathbf{x} &= \mathbf{c}, \\ D\mathbf{x} &\leq \mathbf{d}, \end{aligned} \tag{11.1}$$

where $A \in \mathbb{R}^{n \times n}$ is symmetric, $C \in \mathbb{R}^{m \times n}$, $D \in \mathbb{R}^{p \times n}$, $\mathbf{b} \in \mathbb{R}^n$, $\mathbf{c} \in \mathbb{R}^m$, and $\mathbf{d} \in \mathbb{R}^p$.



In this chapter, we will study various methods for solving the QP problem (11.1), involving

- the method of Lagrange multipliers,
- direct solution methods, and
- iterative solution methods.

Contents of Chapter 11

11.1. Equality Constrained Quadratic Programming	334
11.2. Direct Solution for the KKT System	339
11.3. Linear Iterative Methods	344
11.4. Iterative Solution of the KKT System	350
11.5. Active Set Strategies for Convex QP Problems	352
11.6. Interior-point Methods	355
11.7. Logarithmic Barriers	357
Exercises for Chapter 11	360

11.1. Equality Constrained Quadratic Programming

If only **equality constraints** are imposed, the QP (11.1) reduces to

$$\begin{aligned} \min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}) &:= \frac{1}{2} \mathbf{x}^T A \mathbf{x} - \mathbf{x}^T \mathbf{b}, \quad \text{subj.to} \\ C \mathbf{x} &= \mathbf{c}, \end{aligned} \quad (11.2)$$

where $A \in \mathbb{R}^{n \times n}$, $C \in \mathbb{R}^{m \times n}$, $m \leq n$. For the time being we assume that C has full row rank m and $m < n$.

To solve the problem, let's begin with its **Lagrangian**:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{2} \mathbf{x}^T A \mathbf{x} - \mathbf{x}^T \mathbf{b} - \boldsymbol{\lambda}(\mathbf{c} - C \mathbf{x}), \quad (11.3)$$

where $\boldsymbol{\lambda} \in \mathbb{R}^m$ is the the associated Lagrange multiplier. Then, we have

$$\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = A \mathbf{x} - \mathbf{b} + C^T \boldsymbol{\lambda}. \quad (11.4)$$

The **KKT conditions** (first-derivative tests) for the solution $\mathbf{x} \in \mathbb{R}^n$ of (11.2) give rise to the following linear system

$$\underbrace{\begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix}}_{:=K} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{c} \end{bmatrix}, \quad (11.5)$$

where the second row is the primal feasibility.

Let $Z \in \mathbb{R}^{n \times (n-m)}$ be a matrix whose columns span the null space of C , $\mathcal{N}(C)$, i.e.,

$$CZ = 0 \quad \text{or} \quad \text{Span}(Z) = \mathcal{N}(C). \quad (11.6)$$

Definition 11.1. The matrix K in (11.5) is called the **KKT matrix** and the matrix $Z^T A Z$ is referred to as the **reduced Hessian**.

Note: Now, a question is: "Is the KKT matrix nonsingular?"

Definition 11.2. Let A be a symmetric matrix. We say that A is **(positive) semidefinite**, and we write $A \succeq 0$, if all eigenvalues of A are nonnegative. We say that A is **(positive) definite**, and write $A \succ 0$, if all eigenvalues of A are positive.

Lemma 11.3. Assume that $A \in \mathbb{R}^{n \times n}$ is symmetric and (positive) semidefinite ($A \succeq 0$) and $C \in \mathbb{R}^{m \times n}$ has full row rank $m \leq n$. Then the following are equivalent.

- (a) $\mathcal{N}(A) \cap \mathcal{N}(C) = \{0\}$.
- (b) $Cx = 0, x \neq 0 \Rightarrow x^T Ax > 0$.
- (c) $Z^T AZ$ is positive definite ($\succ 0$), where $Z \in \mathbb{R}^{n \times (n-m)}$ is a matrix for which $\text{Span}(Z) = \mathcal{N}(C)$.
- (d) $A + C^T QC \succ 0$ for some $Q \succeq 0$.

Proof. See Exercise 1. $\square$

Proposition 11.4. For a symmetric matrix A , the following are equivalent.

- 1. $A \succeq 0$.
- 2. $A = U^T U$ for some U .
- 3. $x^T Ax \geq 0$ for all $x \in \mathbb{R}^n$.
- 4. All principal minors of A are nonnegative.
- 5. There exist $x_1, x_2, \dots, x_k \in \mathbb{R}^n$ such that

$$A = \sum_{i=1}^k x_i x_i^T. \quad (11.7)$$

Definition 11.5. For $A, B \in \mathbb{R}^{n \times n}$, we define the **dot product of matrices** as

$$A \cdot B = \sum_{i=1}^n \sum_{j=1}^n A_{ij} B_{ij} = \text{tr}(A^T B). \quad (11.8)$$

Proposition 11.6.

- If $A, B \succeq 0$, then $A \cdot B \geq 0$, and $A \cdot B = 0$ implies $AB = 0$.
- A symmetric matrix A is semidefinite if $A \cdot B \geq 0$ for every $B \succeq 0$.

Theorem 11.7. (Existence and uniqueness). Assume that $C \in \mathbb{R}^{m \times n}$ has full row rank $m \leq n$ and that the reduced Hessian $Z^T A Z$ is positive definite. Then, the KKT matrix K is **nonsingular** and therefore the KKT system (11.5) admits a **unique solution** $(\mathbf{x}^*, \lambda^*)$.

Proof. Suppose that $\mathbf{x} \in \mathcal{N}(A) \cap \mathcal{N}(C)$, $\mathbf{x} \neq 0$. $\Rightarrow A\mathbf{x} = C\mathbf{x} = 0$ and therefore

$$K \begin{bmatrix} \mathbf{x} \\ 0 \end{bmatrix} = \begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 0 \end{bmatrix} = 0, \quad (11.9)$$

which implies the KKT matrix K is singular.

Now, **we assume that the KKT matrix is singular.** That is, there are $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{z} \in \mathbb{R}^m$, not both zero, such that

$$K \begin{bmatrix} \mathbf{x} \\ \mathbf{z} \end{bmatrix} = \begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{z} \end{bmatrix} = 0,$$

which implies

$$A\mathbf{x} + C^T \mathbf{z} = 0 \quad \text{and} \quad C\mathbf{x} = 0. \quad (11.10)$$

It follows from the above equations that

$$0 = \mathbf{x}^T A \mathbf{x} + \mathbf{x}^T C^T \mathbf{z} = \mathbf{x}^T A \mathbf{x},$$

which contradicts (b) in Lemma 11.3, unless $\mathbf{x} = 0$.

In the case (i.e., $\mathbf{x} = 0$), we must have $\mathbf{z} \neq 0$. But then $C^T \mathbf{z} = 0$ contradicts the assumption that C has full row rank. $\square$

Note: More generally, the nonsingularity of the KKT matrix is equivalent to each of statements in Lemma 11.3.

Theorem 11.8. (Global minimizer of (11.2)). *Let the assumptions in Theorem 11.7 be satisfied and $(\mathbf{x}^*, \boldsymbol{\lambda}^*)$ the unique solution of the KKT system (11.5). Then $\mathbf{x}^*$ is the **unique global solution** of the QP (11.2).*

Proof. When $m = n$, the theorem is trivial; we may assume $m < n$.

Let $\mathcal{F} = \{\mathbf{x} \in \mathbb{R}^n \mid C\mathbf{x} = \mathbf{c}\}$, the feasible set.

Clearly, $\mathbf{x}^*$ is a solution of (11.2), i.e., $\mathbf{x}^* \in \mathcal{F}$.

Let $\mathbf{x} \in \mathcal{F}$ be **another feasible point** and $\mathbf{p} := \mathbf{x}^* - \mathbf{x} \neq 0$.

Then $\mathbf{x} = \mathbf{x}^* - \mathbf{p}$ and

$$\begin{aligned} f(\mathbf{x}) &= \frac{1}{2}(\mathbf{x}^* - \mathbf{p})^T A(\mathbf{x}^* - \mathbf{p}) - (\mathbf{x}^* - \mathbf{p})^T \mathbf{b} \\ &= \frac{1}{2}\mathbf{p}^T A\mathbf{p} - \mathbf{p}^T A\mathbf{x}^* + \mathbf{p}^T \mathbf{b} + f(\mathbf{x}^*). \end{aligned} \quad (11.11)$$

Now, (11.5) implies that $A\mathbf{x}^* = \mathbf{b} - C^T \boldsymbol{\lambda}^*$ and thus

$$\mathbf{p}^T A\mathbf{x}^* = \mathbf{p}^T (\mathbf{b} - C^T \boldsymbol{\lambda}^*) = \mathbf{p}^T \mathbf{b} - \mathbf{p}^T C^T \boldsymbol{\lambda}^* = \mathbf{p}^T \mathbf{b},$$

where we have used $C\mathbf{p} = C(\mathbf{x}^* - \mathbf{x}) = \mathbf{c} - \mathbf{c} = 0$. Hence, (11.11) reduces to

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{p}^T A\mathbf{p} + f(\mathbf{x}^*). \quad (11.12)$$

Since $\mathbf{p} \in \mathcal{N}(C)$, we can write $\mathbf{p} = Z\mathbf{y}$, for some nonzero $\mathbf{y} \in \mathbb{R}^{n-m}$, and therefore

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{y}^T Z^T A Z \mathbf{y} + f(\mathbf{x}^*). \quad (11.13)$$

Since $Z^T A Z \succ 0$, we deduce $f(\mathbf{x}) > f(\mathbf{x}^*)$; consequently, $\mathbf{x}^*$ is the unique global minimizer of (11.2). $\square$

Theorem 11.9. *Let the assumptions in Theorem 11.7 be satisfied. Then the KKT matrix K has exactly n positive and m negative eigenvalues.*

Proof. From Lemma 11.3, $A + C^T C \succ 0$; also see (11.89), p. 360. Therefore there exists a nonsingular matrix $R \in \mathbb{R}^{n \times n}$ such that

$$R^T(A + C^T C)R = I. \quad (11.14)$$

Let $CR = U\Sigma V_1^T$ be the **singular value decomposition** of CR , where $U \in \mathbb{R}^{m \times m}$, $\Sigma = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_m)$, and $V_1 \in \mathbb{R}^{n \times m}$. Let $V_2 \in \mathbb{R}^{n \times (n-m)}$ such that

$$V = [V_1 \ V_2]$$

is orthogonal, and define

$$S = [\Sigma \ 0] \in \mathbb{R}^{m \times n}.$$

Then, we have

$$CR = USV^T \quad (11.15)$$

and therefore

$$V^T R^T (A + C^T C) R V = V^T R^T A R V + (C R V)^T C R V = I.$$

It follows from (11.15) that $S = U^T C R V$ and therefore

$$S^T S = (U^T C R V)^T U^T C R V = (C R V)^T C R V.$$

Thus we have $\Lambda := V^T R^T A R V = I - S^T S$ is diagonal; we can write

$$\Lambda = V^T R^T A R V = \text{diag}(1 - \sigma_1^2, 1 - \sigma_2^2, \dots, 1 - \sigma_m^2, 1, \dots, 1). \quad (11.16)$$

Now, applying a congruence transformation to the KKT matrix gives

$$\begin{bmatrix} V^T R^T & 0 \\ 0 & U^T \end{bmatrix} \begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} R V & 0 \\ 0 & U \end{bmatrix} = \begin{bmatrix} \Lambda & S^T \\ S & 0 \end{bmatrix} \quad (11.17)$$

and the **inertia** of the KKT matrix is equal to the inertia of the matrix on the right.¹ Applying a permutation to the matrix on the right of (11.17) gives a block diagonal matrix with n diagonal blocks

$$\begin{bmatrix} \lambda_i & \sigma_i \\ \sigma_i & 0 \end{bmatrix}, \quad i = 1, 2, \dots, m; \quad [\lambda_i], \quad i = m + 1, \dots, n, \quad (11.18)$$

where λ_i are as in (11.16). The eigenvalues of the 2×2 -blocks are

$$\frac{\lambda_i \pm \sqrt{\lambda_i^2 + 4\sigma_i^2}}{2}, \quad i = 1, 2, \dots, m,$$

i.e., one eigenvalue is positive and one is negative. So we can conclude that there are $m + (n - m) = n$ positive eigenvalues and m negative eigenvalues. $\square$

¹**Sylvester's law of inertia** is a theorem in matrix algebra about certain properties of the coefficient matrix of a real quadratic form that remain invariant under a change of basis. Namely, if A is the symmetric matrix that defines the quadratic form, and S is any invertible matrix such that $D = SAS^T$ is diagonal, then the number of negative elements in the diagonal of D is always the same, for all such S ; and the same goes for the number of positive elements.

11.2. Direct Solution for the KKT System

Recall: In (11.5), the KKT system for the solution $\mathbf{x} \in \mathbb{R}^n$ of (11.2) reads

$$\underbrace{\begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix}}_{:=K} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{c} \end{bmatrix}, \quad (11.19)$$

where the second row is the primal feasibility.

For direct solutions of the KKT system (11.19), this section considers symmetric factorization, the range-space approach, and the null-space approach.

11.2.1. Symmetric factorization

A method to solve the KKT system (11.19) is to provide a **symmetric factorization** of the KKT matrix:

$$PKP^T = LDL^T, \quad (11.20)$$

where P is a permutation matrix (appropriately chosen), L is lower triangular with $\text{diag}(L) = I$, and D is block diagonal. Based on (11.20), we rewrite the KKT system (11.19) as

$$P \begin{bmatrix} \mathbf{b} \\ \mathbf{c} \end{bmatrix} = PK \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} = PKP^T \left(P \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} \right) = LDL^T \left(P \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} \right).$$

Thus it can be solved as follows.

$$\begin{aligned} &\text{solve } Ly_1 = P \begin{bmatrix} \mathbf{b} \\ \mathbf{c} \end{bmatrix} \\ &\text{solve } Dy_2 = y_1 \\ &\text{solve } L^T y_3 = y_2 \\ &\text{set } \begin{bmatrix} \mathbf{x}^* \\ \boldsymbol{\lambda}^* \end{bmatrix} = P^T y_3 \end{aligned} \quad (11.21)$$

In python, `scipy.linalg.ldl` is available.

```

1  import numpy as np
2  from scipy.linalg import ldl
3
4  A = np.array([[ 2, -1,  0],
5                [-1,  0, -1],
6                [ 0, -1,  4]])
7  L0,D,P = ldl(A, lower=1) # Use the default: the lower matrix
8
9  print('L0=\n',L0)
10 print('D=\n',D)
11 print('P=\n',P)
12 print('L0*D*L0^T=\n',L0.dot(D).dot(L0.T),'\n#-----')
13
14 P_L0 = L0[P,:] # the row permutation of 'L0'
15 print('P*L0=\n',P_L0)

```

```

1  L0=
2  [[ 1.  0.  0. ]
3   [-0.5 -0.25  1. ]
4   [ 0.  1.  0. ]]
5  D=
6  [[ 2.  0.  0. ]
7   [ 0.  4.  0. ]
8   [ 0.  0. -0.75]]
9  P=
10 [0 2 1]
11 L0*D*L0^T=
12 [[ 2. -1.  0.]
13  [-1.  0. -1.]
14  [ 0. -1.  4.]]
15 #-----
16 P*L0=
17 [[ 1.  0.  0. ]
18  [ 0.  1.  0. ]
19  [-0.5 -0.25  1. ]]

```

- As one can see from the result, $P*L0$ is a lower triangular matrix. The output $L0$ of Python function `ldl` is permuted as

$$L0 = P^T L. \quad (11.22)$$

- **Reference:** [10] J.R. Bunch and L. Kaufman, *Some stable methods for calculating inertia and solving symmetric linear systems*, **Math. Comput.** 31, 1977, pp.163-179.

11.2.2. Range-space approach

Recall: The KKT system for the solution $\mathbf{x} \in \mathbb{R}^n$ of (11.2) is given by

$$\underbrace{\begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix}}_{:=K} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{c} \end{bmatrix}, \quad (11.23)$$

where the second row is the primal feasibility.

The **range-space approach** applies, when $A \in \mathbb{R}^{n \times n}$ is *symmetric positive definite*. Block Gauss elimination of the primal variable $\mathbf{x}$ leads to the **Schur complement system**

$$CA^{-1}C^T \boldsymbol{\lambda} = CA^{-1}\mathbf{b} - \mathbf{c}, \quad (11.24)$$

where $S := CA^{-1}C^T \in \mathbb{R}^{m \times m}$ is the **Schur complement**. See Exercise 2.

Note: Once the optimal Lagrange multipliers $\boldsymbol{\lambda}^*$ is determined from (11.24), the minimizer $\mathbf{x}^*$ can be obtained by solving the first equation of the KKT system

$$A\mathbf{x} = \mathbf{b} - C^T \boldsymbol{\lambda}^*. \quad (11.25)$$

Theorem 11.10. Suppose that $A \in \mathbb{R}^{n \times n}$ is symmetric positive definite, $C \in \mathbb{R}^{m \times n}$, $m \leq n$, such that $\text{rank}(C) = m$. Let $S = CA^{-1}C^T$ be the **Schur complement** associated with the KKT-matrix. Then S is symmetric positive definite on $\mathbb{R}^m$.

Proof. See Exercise 3. $\square$

Remark 11.11. The range-space approach is particularly effective, when

- The matrix A is well conditioned and efficiently invertible. (e.g., diagonal or block-diagonal)
- Its inverse A^{-1} is known explicitly. (e.g., by means of a quasi-Newton updating formula)
- The number of equality constraints (m) is small.

Note that $C \in \mathbb{R}^{m \times n}$ and it can be considered as a map $C : \mathbb{R}^n \rightarrow \mathbb{R}^m$.

11.2.3. Null-space approach

The null-space approach does not require regularity of A and thus has a wider range of applicability than the range-space approach. The method begins with some assumptions.

1. Assume that $C \in \mathbb{R}^{m \times n}$ has full row rank m .
2. Assume that $Z^T A Z \succ 0$, where $Z \in \mathbb{R}^{n \times (n-m)}$ is a matrix for which $\text{Span}(Z) = \mathcal{N}(C)$ and $CZ = 0$; see (11.6).

3. Let $Y \in \mathbb{R}^{n \times m}$ be a matrix such that $[Y \ Z] \in \mathbb{R}^{n \times n}$ is nonsingular.

4. Partition the vector $\mathbf{x} \in \mathbb{R}^n$ according to

$$\mathbf{x} = Y\mathbf{w}_Y + Z\mathbf{w}_Z, \quad (11.26)$$

where $\mathbf{w}_Y \in \mathbb{R}^m$, $\mathbf{w}_Z \in \mathbb{R}^{n-m}$.

5. Substitute (11.26) into the **second equation** of (11.19) to have

$$C\mathbf{x} = CY\mathbf{w}_Y + \underbrace{CZ}_{=0}\mathbf{w}_Z = \mathbf{c} \Rightarrow CY\mathbf{w}_Y = \mathbf{c}, \quad (11.27)$$

i.e. $Y\mathbf{w}_Y$ is a **particular solution** of $C\mathbf{x} = \mathbf{c}$.

6. Furthermore, $\mathbf{w}_Y$ is well determined by (11.27). $\because$ Since $C \in \mathbb{R}^{m \times n}$ has full row rank m and $[Y \ Z] \in \mathbb{R}^{n \times n}$ is nonsingular, the product $C[Y \ Z] = [CY \ 0] \in \mathbb{R}^{m \times n}$ has full row rank m and therefore $CY \in \mathbb{R}^{m \times m}$ is nonsingular. $\square$

7. On the other hand, substituting (11.26) into the **first equation** of (11.19), we get

$$A\mathbf{x} + C^T \boldsymbol{\lambda} = AY\mathbf{w}_Y + AZ\mathbf{w}_Z + C^T \boldsymbol{\lambda} = \mathbf{b}. \quad (11.28)$$

Multiplying Z^T and observing $Z^T C^T = (CZ)^T = 0$ yield

$$Z^T A Z \mathbf{w}_Z = Z^T \mathbf{b} - Z^T A Y \mathbf{w}_Y. \quad (11.29)$$

The **reduced KKT system** (11.29) can be solved easily e.g. by a Cholesky factorization of the reduced Hessian $Z^T A Z \in \mathbb{R}^{(n-m) \times (n-m)}$.

8. Once $\mathbf{w}_Y$ and $\mathbf{w}_Z$ have been computed as solutions of (11.27) and (11.29), respectively, $\mathbf{x}^*$ is obtained from (11.26).
9. When Lagrange multipliers $\boldsymbol{\lambda}^*$ is to be computed, we multiply (11.28) by Y^T and solve the resulting equation:

$$(CY)^T \boldsymbol{\lambda}^* = Y^T \mathbf{b} - Y^T A \mathbf{x}^*. \quad (11.30)$$

11.3. Linear Iterative Methods

Consider a **linear algebraic system**

$$A\mathbf{x} = \mathbf{b}, \quad (11.31)$$

for which we assume that $A \in \mathbb{R}^{n \times n}$ is invertible.

Key Idea 11.12. Iterative methods for solving (11.31):

- Linear iterative methods begin with splitting the matrix A by

$$A = M - N, \quad (11.32)$$

for some invertible matrix M .

- Then, the linear system equivalently reads

$$M\mathbf{x} = N\mathbf{x} + \mathbf{b}. \quad (11.33)$$

- Associated with the splitting is an iterative method

$$M\mathbf{x}^k = N\mathbf{x}^{k-1} + \mathbf{b}, \quad (11.34)$$

or, equivalently,

$$\mathbf{x}^k = M^{-1}(N\mathbf{x}^{k-1} + \mathbf{b}) = \mathbf{x}^{k-1} + M^{-1}(\mathbf{b} - A\mathbf{x}^{k-1}), \quad (11.35)$$

for an initial value $\mathbf{x}^0$.

Note: Methods differ for different choices of M .

- M must be easy to invert (efficiency), and
- $M^{-1} \approx A^{-1}$ (convergence).

11.3.1. Convergence theory

- Let

$$\mathbf{e}^k = \mathbf{x} - \mathbf{x}^k.$$

- It follows from (11.33) and (11.34) that the error equation reads

$$M \mathbf{e}^k = N \mathbf{e}^{k-1} \quad (11.36)$$

or, equivalently,

$$\mathbf{e}^k = M^{-1} N \mathbf{e}^{k-1}. \quad (11.37)$$

- Since

$$\begin{aligned} \|\mathbf{e}^k\| &\leq \|M^{-1}N\| \cdot \|\mathbf{e}^{k-1}\| \leq \|M^{-1}N\|^2 \cdot \|\mathbf{e}^{k-2}\| \\ &\leq \dots \leq \|M^{-1}N\|^k \cdot \|\mathbf{e}^0\|, \end{aligned} \quad (11.38)$$

a sufficient condition for the convergence is

$$\|M^{-1}N\| < 1. \quad (11.39)$$

Definition 11.13. Let $\sigma(B)$ be the **spectrum**, the set of eigenvalues of the matrix B , and $\rho(B)$ denote the **spectral radius** defined by

$$\rho(B) = \max_{\lambda_i \in \sigma(B)} |\lambda_i|.$$

Theorem 11.14. *The iteration converges if and only if*

$$\rho(M^{-1}N) < 1. \quad (11.40)$$

11.3.2. Graph theory: Estimation of the spectral radius

Definition 11.15. A **permutation matrix** is a square matrix in which each row and each column has one entry of unity, all others zero.

Definition 11.16. For $n \geq 2$, an $n \times n$ complex-valued matrix A is **reducible** if there is a permutation matrix P such that

$$PAP^T = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix},$$

where A_{11} and A_{22} are respectively $r \times r$ and $(n - r) \times (n - r)$ submatrices, $0 < r < n$. If no such permutation matrix exists, then A is **irreducible**.

The geometrical interpretation of the concept of the irreducibility by means of graph theory is useful.

Geometrical interpretation of irreducibility

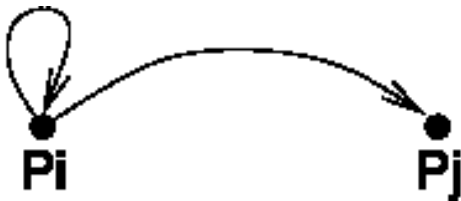


Figure 11.1: The directed paths for nonzero a_{ii} and a_{ij} .

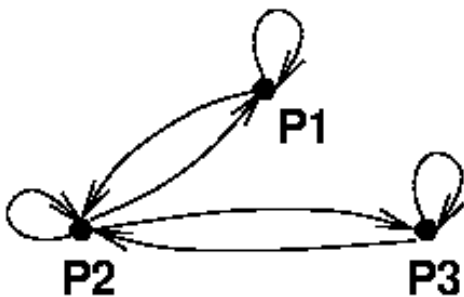


Figure 11.2: The directed graph $G(A)$ for A in (11.41).

- Given $A = [a_{ij}] \in \mathbb{C}^{n \times n}$, consider n distinct points

$$P_1, P_2, \dots, P_n$$

in the plane, which we will call **nodes** or **nodal points**.

- For any nonzero entry a_{ij} of A , we connect P_i to P_j by a path $\overrightarrow{P_i P_j}$, directed from the node P_i to the node P_j ; a nonzero a_{ii} is joined to itself by a directed loop, as shown in Figure 11.1.
- In this way, every $n \times n$ matrix A can be associated a **directed graph** $G(A)$. For example, the matrix

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} \quad (11.41)$$

has a directed graph shown in Figure 11.2.

Definition 11.17. A directed graph is **strongly connected** if, for any ordered pair of nodes (P_i, P_j) , there is a directed path of a finite length

$$\overrightarrow{P_i P_{k_1}}, \overrightarrow{P_{k_1} P_{k_2}}, \dots, \overrightarrow{P_{k_{r-1}} P_{k_r=j}},$$

connecting from P_i to P_j .

The theorems to be presented in this subsection can be found in [77] along with their proofs.

Theorem 11.18. *An $n \times n$ complex-valued matrix A is irreducible if and only if its directed graph $G(A)$ is strongly connected.*

11.3.3. Eigenvalue locus theorem

For $A = [a_{ij}] \in \mathbb{C}^{n \times n}$, let

$$\Lambda_i := \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}| \quad (11.42)$$

Theorem 11.19. (Eigenvalue locus theorem) *Let $A = [a_{ij}]$ be an irreducible $n \times n$ complex matrix. Then,*

1. **(Gerschgorin [22])** *All eigenvalues of A lie in the union of the disks in the complex plane*

$$|z - a_{ii}| \leq \Lambda_i, \quad 1 \leq i \leq n. \quad (11.43)$$

2. **(Taussky [74])** *In addition, assume that λ , an eigenvalue of A , is a boundary point of the union of the disks $|z - a_{ii}| \leq \Lambda_i$. Then, all the n circles $|z - a_{ii}| = \Lambda_i$ must pass through the point λ , i.e., $|\lambda - a_{ii}| = \Lambda_i$ for all $1 \leq i \leq n$.*

Example 11.20. For example, for

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

$\Lambda_1 = 1$, $\Lambda_2 = 2$, and $\Lambda_3 = 1$. Since $a_{ii} = 2$, for $i = 1, 2, 3$,

$$|\lambda - 2| < 2$$

for all eigenvalues λ of A . $\square$

Positiveness

Definition 11.21. An $n \times n$ complex-valued matrix $A = [a_{ij}]$ is **diagonally dominant** if

$$|a_{ii}| \geq \Lambda_i := \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|, \quad (11.44)$$

for all $1 \leq i \leq n$. An $n \times n$ matrix A is *irreducibly diagonally dominant* if A is irreducible and diagonally dominant, with strict inequality holding in (11.44) for at least one i .

Theorem 11.22. Let A be an $n \times n$ strictly or irreducibly diagonally dominant complex-valued matrix. Then, A is nonsingular. If all the diagonal entries of A are in addition positive real, then the real parts of all eigenvalues of A are positive.

Corollary 11.23. A Hermitian matrix satisfying the conditions in Theorem 11.22 is positive definite.

11.3.4. Regular splitting and M-matrices

Definition 11.24. For $n \times n$ real matrices, A , M , and N , $A = M - N$ is a **regular splitting** of A if M is nonsingular with $M^{-1} \geq 0$, and $N \geq 0$.

Theorem 11.25. If $A = M - N$ is a regular splitting of A and $A^{-1} \geq 0$, then

$$\rho(M^{-1}N) = \frac{\rho(A^{-1}N)}{1 + \rho(A^{-1}N)} < 1. \quad (11.45)$$

Thus, the matrix $M^{-1}N$ is convergent and the iterative method of (11.34) converges for any initial value x^0 .

Definition 11.26. An $n \times n$ real matrix $A = [a_{ij}]$ with $a_{ij} \leq 0$ for all $i \neq j$ is an **M-matrix** if A is nonsingular and $A^{-1} \geq 0$.

Theorem 11.27. Let $A = (a_{ij})$ be an $n \times n$ M-matrix. If M is any $n \times n$ matrix obtained by setting certain off-diagonal entries of A to zero, then $A = M - N$ is a regular splitting of A and $\rho(M^{-1}N) < 1$.

Theorem 11.28. Let A be an $n \times n$ real matrix with $A^{-1} > 0$, and $A = M_1 - N_1 = M_2 - N_2$ be two regular splittings of A . If $N_2 \geq N_1 \geq 0$, where neither $N_2 - N_1$ nor N_1 is null, then

$$1 > \rho(M_2^{-1}N_2) > \rho(M_1^{-1}N_1) > 0. \quad (11.46)$$

11.4. Iterative Solution of the KKT System

Recall: The KKT system for the solution $\mathbf{x} \in \mathbb{R}^n$ of (11.2) is given by

$$\underbrace{\begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix}}_{:=K} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{c} \end{bmatrix}, \quad (11.47)$$

where the second row is the primal feasibility.

When the direct solution of the KKT system (11.47) is **computationally too costly**, the alternative is to use an iterative method. An iterative solver can be applied

- either to the **entire KKT system**
- or to the **special structure of the KKT matrix**, as in the range-space and null-space approach, and based on **regular splitting**^a of specifically transformed matrices of K .

^aFor $n \times n$ real matrices, A , M , and N , $A = M - N$ is a **regular splitting** of A if M is nonsingular with $M^{-1} \geq 0$, and $N \geq 0$.

The **transforming null-space iteration** does not require regularity of A and therefore has a wider range of applicability than the **transforming range-space iteration**. Here we will deal with the transforming range-space iteration *only* for simplicity.

11.4.1. Krylov subspace methods

The KKT matrix $K \in \mathbb{R}^{(n+m) \times (n+m)}$ is **indefinite**; if C has full row rank m , then K has exactly n positive and m negative eigenvalues, as shown in Theorem 11.9. Therefore, for iterative methods for the solution of **entire KKT system**, appropriate candidates are Krylov subspace methods like

- **GMRES** (Generalized Minimum RESidual) and
- **QMR** (Quasi Minimum Residual).

11.4.2. The transforming range-space iteration

Assumption. The matrix $A \in \mathbb{R}^{n \times n}$ is *symmetric positive definite (SPD)* and A has an *easily invertible SPD approximation* $\hat{A}$ such that $\hat{A}^{-1}A \sim I$.

1. We choose $L \in \mathbb{R}^{(n+m) \times (n+m)}$ as a lower triangular block matrix

$$L = \begin{bmatrix} I & 0 \\ -C\hat{A}^{-1} & I \end{bmatrix}, \quad (11.48)$$

which gives rise to the **regular splitting** of LK :

$$LK = \begin{bmatrix} \hat{A} & C^T \\ 0 & \hat{S} \end{bmatrix} - \begin{bmatrix} \hat{A}(I - \hat{A}^{-1}A) & 0 \\ C(I - \hat{A}^{-1}A) & 0 \end{bmatrix} =: M_1 - M_2, \quad (11.49)$$

where $\hat{S} = -C\hat{A}^{-1}C^T \in \mathbb{R}^{m \times m}$. (Note $M_2 = M_1 - LK \sim 0$.)

2. Let

$$\psi := (\mathbf{x}, \boldsymbol{\lambda})^T, \quad \beta := (\mathbf{b}, \mathbf{c})^T.$$

Then the KKT system (11.47) gives $LK\psi = (M_1 - M_2)\psi = L\beta$ so that

$$\begin{aligned} M_1\psi &= M_2\psi + L\beta \\ &= (M_1 - LK)\psi + L\beta = M_1\psi + L(\beta - K\psi). \end{aligned} \quad (11.50)$$

3. Given an initialization $\psi_0 \in \mathbb{R}^{(n+m) \times (n+m)}$, we compute ψ_{k+1} by means of the **transforming range-space iteration**

$$\begin{aligned} \psi_{k+1} &= (I - M_1^{-1}LK)\psi_k + M_1^{-1}L\beta \\ &= \psi_k + M_1^{-1}L(\beta - K\psi_k), \quad k \geq 0. \end{aligned} \quad (11.51)$$

Implementation of (11.51):

$$\begin{aligned} \text{compute } \mathbf{r}_k &= (\mathbf{r}_k^{(1)}, \mathbf{r}_k^{(2)})^T := \beta - K\psi_k; \\ \text{compute } L\mathbf{r}_k &= \begin{bmatrix} \mathbf{r}_k^{(1)} \\ -C\hat{A}^{-1}\mathbf{r}_k^{(1)} + \mathbf{r}_k^{(2)} \end{bmatrix}; \\ \text{solve } M_1\Delta\psi_k &= L\mathbf{r}_k; \\ \text{set } \psi_{k+1} &= \psi_k + \Delta\psi_k; \end{aligned} \quad (11.52)$$

11.5. Active Set Strategies for Convex QP Problems

Recall: Quadratic programming formulated in a **general form** (11.1):

$$\begin{aligned} \min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}) &:= \frac{1}{2} \mathbf{x}^T A \mathbf{x} - \mathbf{x}^T \mathbf{b}, \quad \text{subj.to} \\ C\mathbf{x} &= \mathbf{c}, \\ D\mathbf{x} &\leq \mathbf{d}, \end{aligned} \tag{11.53}$$

where $A \in \mathbb{R}^{n \times n}$, $C \in \mathbb{R}^{m \times n}$, $D \in \mathbb{R}^{p \times n}$, and $\mathbf{b} \in \mathbb{R}^n$, $\mathbf{c} \in \mathbb{R}^m$, $\mathbf{d} \in \mathbb{R}^p$. Here, we assume A is SPD.

Definition 11.29. The inequality constraint in (11.53) can be written as

$$g_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, p.$$

Given a point $\mathbf{x}$ in the feasible region, a **constraint** $g_i(\mathbf{x}) \leq 0$ is called **active** at $\mathbf{x}$ if $g_i(\mathbf{x}) = 0$ and **inactive** if $g_i(\mathbf{x}) < 0$. The **active set** at $\mathbf{x}$ is made up of those constraints that are active at the current point. (Equality constraints are always active.)

Note: The active set is particularly important in optimization theory, because it determines which constraints will influence the final result of optimization. For example,

- For linear programming problem, the active set gives the hyperplanes that intersect at the solution point.
- **In quadratic programming**, the solution is not necessarily on one of the edges of the bounding polygon; an estimation of the active set gives us a subset of inequalities to watch while searching the solution, which **reduces the complexity of the search** [57].

11.5.1. Primal active set strategy

We rewrite the matrices C and D in the form

$$C = \begin{bmatrix} C_1 \\ \vdots \\ C_m \end{bmatrix}, \quad C_i \in \mathbb{R}^n; \quad D = \begin{bmatrix} D_1 \\ \vdots \\ D_p \end{bmatrix}, \quad D_i \in \mathbb{R}^n. \quad (11.54)$$

Then the inequality constraints in (11.53) can be equivalently stated as

$$D_i^T \mathbf{x} \leq d_i, \quad i = 1, 2, \dots, p. \quad (11.55)$$

(C_i and D_i are row vectors; we will deal with them like column vectors.)

The **primal active set strategy** is an iterative procedure:

- Given a feasible iterate $\mathbf{x}_k$, $k \geq 0$, we determine its active set

$$\mathcal{I}_{ac}(\mathbf{x}_k) \subset \{1, 2, \dots, p\} \quad (11.56)$$

and consider the corresponding constraints as equality constraints, whereas the remaining inequality constraints are disregarded.

- Setting

$$\mathbf{p} = \mathbf{x}_k - \mathbf{x}, \quad \mathbf{r}_k = A\mathbf{x}_k - \mathbf{b}, \quad (11.57)$$

we find

$$f(\mathbf{x}) = f(\mathbf{x}_k - \mathbf{p}) = \frac{1}{2} \mathbf{p}^T A \mathbf{p} - \mathbf{r}_k^T \mathbf{p} + \mathbf{g}, \quad (11.58)$$

where $\mathbf{g} = \frac{1}{2} \mathbf{x}_k^T A \mathbf{x}_k - \mathbf{b}^T \mathbf{x}_k$.

- Then the equality constrained QP problem to be solved for the $(k+1)$ -st iteration step is:

$$\begin{aligned} \mathbf{p}_k = \arg \min_{\mathbf{p} \in \mathbb{R}^n} & \left(\frac{1}{2} \mathbf{p}^T A \mathbf{p} - \mathbf{r}_k^T \mathbf{p} \right), \quad \text{subj.to} \\ & C \mathbf{p} = 0 \\ & D_i^T \mathbf{p} = 0, \quad i \in \mathcal{I}_{ac}(\mathbf{x}_k). \end{aligned} \quad (11.59)$$

- The new iterate is then obtained according to

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{p}_k, \quad (11.60)$$

where α_k is chosen such that $\mathbf{x}_{k+1}$ stays feasible.

Remark 11.30. (Determination of α_k). The parameter can be determined as follows.

- For each $i \in \mathcal{I}_{ac}(\mathbf{x}_k)$, we have

$$\mathbf{D}_i^T \mathbf{x}_{k+1} = \mathbf{D}_i^T \mathbf{x}_k - \alpha_k \mathbf{D}_i^T \mathbf{p}_k = \mathbf{D}_i^T \mathbf{x}_k \leq d_i. \quad (11.61)$$

- If $\mathbf{D}_i^T \mathbf{p}_k \geq 0$ for some $i \notin \mathcal{I}_{ac}(\mathbf{x}_k)$, it follows that

$$\mathbf{D}_i^T \mathbf{x}_{k+1} = \mathbf{D}_i^T \mathbf{x}_k - \alpha_k \mathbf{D}_i^T \mathbf{p}_k \leq \mathbf{D}_i^T \mathbf{x}_k \leq d_i. \quad (11.62)$$

- On the other hand, if $\mathbf{D}_i^T \mathbf{p}_k < 0$ for some $i \notin \mathcal{I}_{ac}(\mathbf{x}_k)$, we have

$$\mathbf{D}_i^T \mathbf{x}_{k+1} = \mathbf{D}_i^T \mathbf{x}_k - \alpha_k \mathbf{D}_i^T \mathbf{p}_k \leq d_i \iff \alpha_k \leq \frac{d_i - \mathbf{D}_i^T \mathbf{x}_k}{-\mathbf{D}_i^T \mathbf{p}_k}. \quad (11.63)$$

- Consequently, in order to guarantee feasibility, we choose

$$\alpha_k := \min(1, \hat{\alpha}_k), \text{ where } \hat{\alpha}_k := \min_{i \notin \mathcal{I}_{ac}(\mathbf{x}_k); \mathbf{D}_i^T \mathbf{p}_k < 0} \frac{\mathbf{D}_i^T \mathbf{x}_k - d_i}{\mathbf{D}_i^T \mathbf{p}_k}. \quad (11.64)$$

Remark 11.31. (Update for $\mathcal{I}_{ac}(\mathbf{x}_{k+1})$). Let's begin with defining the set of blocking constraints:

$$\mathcal{I}_{bl}(\mathbf{p}_k) \stackrel{\text{def}}{=} \left\{ i \notin \mathcal{I}_{ac}(\mathbf{x}_k) \mid \mathbf{D}_i^T \mathbf{p}_k < 0, \frac{\mathbf{D}_i^T \mathbf{x}_k - d_i}{\mathbf{D}_i^T \mathbf{p}_k} \leq 1 \right\}. \quad (11.65)$$

Then we specify $\mathcal{I}_{ac}(\mathbf{x}_{k+1})$ by adding the most restrictive blocking constraint to $\mathcal{I}_{ac}(\mathbf{x}_k)$:

$$\mathcal{I}_{ac}(\mathbf{x}_{k+1}) \stackrel{\text{def}}{=} \mathcal{I}_{ac}(\mathbf{x}_k) \cup \left\{ j \in \mathcal{I}_{bl}(\mathbf{p}_k) \mid \frac{\mathbf{D}_j^T \mathbf{x}_k - d_j}{\mathbf{D}_j^T \mathbf{p}_k} = \hat{\alpha}_k \right\}. \quad (11.66)$$

For such a j (the index of the newly added constraint), we clearly have

$$\mathbf{D}_j^T \mathbf{x}_{k+1} = \mathbf{D}_j^T \mathbf{x}_k - \alpha_k \mathbf{D}_j^T \mathbf{p}_k = \mathbf{D}_j^T \mathbf{x}_k - \hat{\alpha}_k \mathbf{D}_j^T \mathbf{p}_k = d_j, \quad (11.67)$$

and therefore $\mathcal{I}_{ac}(\mathbf{x}_{k+1})$ contains active constraints only.

11.6. Interior-point Methods

Interior-point methods are iterative schemes where the iterates approach the optimal solution *from the interior of the feasible set*. For simplicity, we consider **inequality-constrained quadratic programming** problems of the form

$$\min_{\mathbf{x} \in \mathbb{R}^n} Q(\mathbf{x}) := \frac{1}{2} \mathbf{x}^T A \mathbf{x} - \mathbf{x}^T \mathbf{b}, \quad \text{subj.to} \quad (11.68)$$

$$D\mathbf{x} \leq \mathbf{d},$$

where $A \in \mathbb{R}^{n \times n}$ is SPD, $D \in \mathbb{R}^{p \times n}$, and $\mathbf{b} \in \mathbb{R}^n$ and $\mathbf{d} \in \mathbb{R}^p$.

Note: Its Lagrangian reads

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}) \stackrel{\text{def}}{=} Q(\mathbf{x}) + \sum_{i=1}^p \mu_i (D_i^T \mathbf{x} - d_i), \quad (11.69)$$

where $\boldsymbol{\mu} = (\mu_1, \mu_2, \dots, \mu_p)^T$ and $D = [D_1, \dots, D_p]^T$, and therefore

$$\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\mu}) = \nabla_{\mathbf{x}} Q(\mathbf{x}) + D^T \boldsymbol{\mu} = A\mathbf{x} - \mathbf{b} + D^T \boldsymbol{\mu}. \quad (11.70)$$

Thus the **KKT conditions** for (11.68) are stated as

$$\begin{aligned} A\mathbf{x} + D^T \boldsymbol{\mu} - \mathbf{b} &= 0, \\ D\mathbf{x} - \mathbf{d} &\leq 0, \\ \mu_i (D\mathbf{x} - \mathbf{d})_i &= 0, \quad i = 1, 2, \dots, p, \\ \mu_i &\geq 0, \quad i = 1, 2, \dots, p. \end{aligned} \quad (11.71)$$

$\{\mu_i (D\mathbf{x} - \mathbf{d})_i = 0\}$ is the **complementary slackness**.

By introducing a **slack variable** $\mathbf{z} = \mathbf{d} - D\mathbf{x}$, the above conditions can be equivalently formulated as follows:

$$\begin{aligned} A\mathbf{x} + D^T \boldsymbol{\mu} - \mathbf{b} &= 0, \\ D\mathbf{x} + \mathbf{z} - \mathbf{d} &= 0, \\ \mu_i z_i &= 0, \quad i = 1, 2, \dots, p, \\ \mu_i, z_i &\geq 0, \quad i = 1, 2, \dots, p. \end{aligned} \quad (11.72)$$

The **interior-point method** begins with replacing $\mu_i z_i = 0$ by $\mu_i z_i = \theta > 0$ for all $i = 1, 2, \dots, p$, and enforces $\theta \searrow 0$.

Equation (11.72) can be rewritten as a constrained system of nonlinear equations. We define the nonlinear map

$$F(\mathbf{x}, \boldsymbol{\mu}, \mathbf{z}) \stackrel{\text{def}}{=} \begin{bmatrix} A\mathbf{x} + D^T \boldsymbol{\mu} - \mathbf{b} \\ D\mathbf{x} + \mathbf{z} - \mathbf{d} \\ \mathcal{Z}\mathcal{M}\mathbf{e} \end{bmatrix}, \quad (11.73)$$

where

$$\mathcal{Z} = \text{diag}(z_1, \dots, z_p), \quad \mathcal{M} = \text{diag}(\mu_1, \dots, \mu_p), \quad \mathbf{e} = (1, \dots, 1)^T.$$

Definition 11.32. (Central path). The set of points $(\mathbf{x}_\tau, \boldsymbol{\mu}_\tau, \mathbf{z}_\tau)$, $\tau > 0$, satisfying

$$F(\mathbf{x}_\tau, \boldsymbol{\mu}_\tau, \mathbf{z}_\tau) = \begin{bmatrix} 0 \\ 0 \\ \tau \mathbf{e} \end{bmatrix}, \quad \mathbf{z}, \boldsymbol{\mu} \geq 0, \quad (11.74)$$

is called the **central path**.

Newton's method:

- Given a feasible iterate $(\mathbf{x}, \boldsymbol{\mu}, \mathbf{z}) = (\mathbf{x}_k, \boldsymbol{\mu}_k, \mathbf{z}_k)$, we introduce a duality measure θ :

$$\theta := \frac{1}{p} \sum_{i=1}^p z_i \mu_i = \frac{\mathbf{z}^T \boldsymbol{\mu}}{p}. \quad (11.75)$$

- The idea is to apply **Newton's method** to (11.73) to compute $(\mathbf{x}_{\sigma\theta}, \boldsymbol{\mu}_{\sigma\theta}, \mathbf{z}_{\sigma\theta})$ on the central path, where $\sigma \in [0, 1]$ is an algorithm parameter.
- The Newton increments $(\Delta\mathbf{x}, \Delta\boldsymbol{\mu}, \Delta\mathbf{z})$ solve the linear system

$$\nabla F(\mathbf{x}, \boldsymbol{\mu}, \mathbf{z}) \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\boldsymbol{\mu} \\ \Delta\mathbf{z} \end{bmatrix} = -F(\mathbf{x}, \boldsymbol{\mu}, \mathbf{z}) + \begin{bmatrix} 0 \\ 0 \\ \sigma\theta \mathbf{e} \end{bmatrix}, \quad (11.76)$$

where

$$\nabla F(\mathbf{x}, \boldsymbol{\mu}, \mathbf{z}) \stackrel{\text{def}}{=} \begin{bmatrix} A & D^T & 0 \\ D & 0 & I \\ 0 & \mathcal{Z} & \mathcal{M} \end{bmatrix}.$$

- The new iterate $(\mathbf{x}_{k+1}, \boldsymbol{\mu}_{k+1}, \mathbf{z}_{k+1})$ is then determined by means of

$$(\mathbf{x}_{k+1}, \boldsymbol{\mu}_{k+1}, \mathbf{z}_{k+1}) = (\mathbf{x}_k, \boldsymbol{\mu}_k, \mathbf{z}_k) + \alpha(\Delta\mathbf{x}, \Delta\boldsymbol{\mu}, \Delta\mathbf{z}), \quad (11.77)$$

with α chosen such that the new iterate stays **feasible**.

11.7. Logarithmic Barriers

Recall: The **inequality-constrained quadratic programming** problem in (11.68):

$$\begin{aligned} \min_{\mathbf{x} \in \mathbb{R}^n} Q(\mathbf{x}) &:= \frac{1}{2} \mathbf{x}^T A \mathbf{x} - \mathbf{x}^T \mathbf{b}, \quad \text{subj.to} \\ D\mathbf{x} &\leq \mathbf{d}, \end{aligned} \quad (11.78)$$

where $A \in \mathbb{R}^{n \times n}$ is SPD, $D \in \mathbb{R}^{p \times n}$, and $\mathbf{b} \in \mathbb{R}^n$ and $\mathbf{d} \in \mathbb{R}^p$.

Algorithms based on barrier functions are iterative methods where the iterates are forced to stay within the **interior** of the feasible set:

$$\mathcal{F}^{\text{int}} \stackrel{\text{def}}{=} \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{D}_i^T \mathbf{x} - d_i < 0, \ 1 \leq i \leq p\}. \quad (11.79)$$

Barrier functions commonly have the following properties:

- They are smooth within $\mathcal{F}^{\text{int}}$.
- They approach ∞ as $\mathbf{x}$ approaches the boundary of $\mathcal{F}^{\text{int}}$.
- They are infinite outside $\mathcal{F}^{\text{int}}$.

Definition 11.33. (Logarithmic barrier function). For the QP problem (11.78), the objective functional

$$B_\beta(\mathbf{x}) \stackrel{\text{def}}{=} Q(\mathbf{x}) - \beta \sum_{i=1}^p \log(d_i - \mathbf{D}_i^T \mathbf{x}), \quad \beta > 0, \quad (11.80)$$

is called the **logarithmic barrier function**.^a The parameter β is referred to as the **barrier parameter**.

^aThe function $\log$ stands for the natural logarithm ($\ln$), as commonly accepted in the literature of computational algorithms.

Theorem 11.34. (Properties of the logarithmic barrier function)

[81, (Wright,1992)]. Assume that the set $\mathcal{S}$ of solutions of (11.78) is nonempty and bounded and that the interior $\mathcal{F}^{\text{int}}$ of the feasible set is nonempty. Let $\{\beta_k\}$ be a decreasing sequence of barrier parameters with

$$\beta_k \rightarrow 0, \text{ as } k \rightarrow \infty. \quad (11.81)$$

Then there holds:

1. For any $\beta > 0$, the logarithmic barrier function $B_\beta(\mathbf{x})$ **is convex** in $\mathcal{F}^{\text{int}}$ and attains a minimizer $\mathbf{x}_\beta \in \mathcal{F}^{\text{int}}$.
2. There is a **unique minimizer**; any local minimizer $\mathbf{x}_\beta$ is also a global minimizer of $B_\beta(\mathbf{x})$.
3. If $\{\mathbf{x}_{\beta_k} \mid k \in \mathbb{N}\}$ is a sequence of minimizers, then there exists $\mathbb{N}' \subset \mathbb{N}$ such that

$$\mathbf{x}_{\beta_k} \rightarrow \mathbf{x}^* \in \mathcal{S}, \quad k \in \mathbb{N}'.$$

4. If Q^* is the optimal value of the objective functional Q in (11.78), then for any sequence $\{\mathbf{x}_{\beta_k}\}$ of minimizers,

$$Q(\mathbf{x}_{\beta_k}) \rightarrow Q^*, \quad B_{\beta_k}(\mathbf{x}_{\beta_k}) \rightarrow Q^*, \text{ as } k \rightarrow \infty. \quad (11.82)$$

Objective: In the following:

We will have a closer look at the relation between a **minimizer of $B_\beta(\mathbf{x})$** and **the solution of the KKT system**, a point $(\mathbf{x}, \mu)$ satisfying the KKT conditions for (11.78).

Recall: The KKT conditions for (11.78) are given in (11.71), p. 355:

$$\begin{aligned}\nabla_{\mathbf{x}}Q(\mathbf{x}) + D^T\boldsymbol{\mu} &= A\mathbf{x} - \mathbf{b} + D^T\boldsymbol{\mu} = 0, \\ D\mathbf{x} - \mathbf{d} &\leq 0, \\ \mu_i(D\mathbf{x} - \mathbf{d})_i &= 0, \quad i = 1, 2, \dots, p, \\ \mu_i &\geq 0, \quad i = 1, 2, \dots, p.\end{aligned}\tag{11.83}$$

If $\mathbf{x}_\beta$ is a minimizer of $B_\beta(\mathbf{x})$, we obviously have

$$\nabla_{\mathbf{x}}B_\beta(\mathbf{x}_\beta) = \nabla_{\mathbf{x}}Q(\mathbf{x}_\beta) + \sum_{i=1}^p \frac{\beta}{d_i - \mathbf{D}_i^T \mathbf{x}_\beta} \mathbf{D}_i = 0. \tag{11.84}$$

Definition 11.35. Perturbed (or, approximate) complementarity is the vector $\mathbf{z}_\beta \in \mathbb{R}^p$ having its components

$$(\mathbf{z}_\beta)_i = z_{\beta,i} := \frac{\beta}{d_i - \mathbf{D}_i^T \mathbf{x}_\beta}, \quad 1 \leq i \leq p. \tag{11.85}$$

In terms of the perturbed complementarity, (11.84) can be stated as

$$\nabla_{\mathbf{x}}Q(\mathbf{x}_\beta) + \sum_{i=1}^p z_{\beta,i} \mathbf{D}_i = 0. \tag{11.86}$$

Rewrite the first of the KKT conditions (11.83) as

$$\nabla_{\mathbf{x}}Q(\mathbf{x}) + \sum_{i=1}^p \mu_i \mathbf{D}_i = 0. \tag{11.87}$$

- Obviously, (11.87) looks the same as (11.86).
- Apparently, the 2nd and the 4th KKT conditions in (11.83) are satisfied by $(\mathbf{x}, \boldsymbol{\mu}) = (\mathbf{x}_\beta, \mathbf{z}_\beta)$.
- However, the 3rd KKT condition does not hold true, because it follows readily from (11.85) that

$$z_{\beta,i}(d_i - \mathbf{D}_i^T \mathbf{x}_\beta) = \beta > 0, \quad 1 \leq i \leq p. \tag{11.88}$$

As $\beta \rightarrow 0$, the minimizer $\mathbf{x}_\beta$ and the associated $\mathbf{z}_\beta$ come closer and closer to satisfying the 3rd KKT condition. This is why $\mathbf{z}_\beta$ is called **perturbed (approximate) complementarity**.

Exercises for Chapter 11

11.1. Recall the KKT matrix

$$K = \begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix}.$$

Here, we assume that $A \in \mathbb{R}^{n \times n}$ is symmetric and (positive) semidefinite ($A \succeq 0$) and $C \in \mathbb{R}^{m \times n}$ has full row rank $m \leq n$. Show the following are equivalent.

- (a) $\mathcal{N}(A) \cap \mathcal{N}(C) = \{0\}$.
- (b) $Cx = 0, x \neq 0 \Rightarrow x^T Ax > 0$.
- (c) $Z^T AZ$ is positive definite ($\succ 0$), where $Z \in \mathbb{R}^{n \times (n-m)}$ is a matrix for which $\text{Span}(Z) = \mathcal{N}(C)$.
- (d) $A + C^T QC \succ 0$ for some $Q \succeq 0$.

When (c) is considered, you may assume $m < n$.

Hint: (a) $\Leftrightarrow$ (b): Use contradictions. For example, $\mathcal{N}(A) \cap \mathcal{N}(C) \neq \{0\} \Rightarrow \exists x \in \mathcal{N}(A) \cap \mathcal{N}(C), x \neq 0 \Rightarrow Ax = 0$ and $Cx = 0$ and therefore $x^T Ax = 0$, which is a contradiction; this implies (a) $\Leftarrow$ (b).

(b) $\Leftarrow$ (c): Let $Cx = 0, x \neq 0 \Rightarrow x$ must have the form $x = Zy$ for some $y \neq 0$. (why?) $\Rightarrow x^T Ax = y^T Z^T P Z y > 0$.

(b) $\Leftrightarrow$ (d): If (b) holds, then

$$x^T (A + C^T C) x = x^T Ax + \|Cx\|^2 > 0. \quad (11.89)$$

$\Rightarrow$ (d) holds with $Q = I$. On the other hand, if (d) holds for some $Q \succeq 0$, then

$$x^T (A + C^T QC) x = x^T Ax + x^T C^T QC x > 0, \quad x \neq 0. \quad (11.90)$$

When $Cx = 0, x^T Ax$ must be positive.

Now, you should fill out missing gaps: (a) $\Rightarrow$ (b) and (b) $\Rightarrow$ (c).

11.2. Derive the Schur complement system (11.24) from the KKT system (11.19).

11.3. Suppose that $A \in \mathbb{R}^{n \times n}$ is symmetric positive definite, $C \in \mathbb{R}^{m \times n}, m \leq n$, such that $\text{rank}(C) = m$. Let $S = CA^{-1}C^T$ be the **Schur complement** associated with the KKT-matrix, as in (11.24). Show that S is symmetric positive definite on $\mathbb{R}^m$.

Hint: You may begin with claiming that $C^T x \neq 0$ for every nonzero $x \in \mathbb{R}^m$. (Figure out why)

11.4. Verify the splitting (11.49).

11.5. Recall the dual problem of linear SVM in (5.52):

$$\max_{0 \leq \alpha \leq C} [\alpha \cdot 1 - \frac{1}{2} \alpha^T G \alpha] \quad \text{subj.to} \quad \alpha \cdot y = 0, \quad (11.91)$$

where $G = ZZ^T$ and $G_{ij} = y^{(i)} y^{(j)} x^{(i)} \cdot x^{(j)}$.

- (a) Ignoring momentarily the inequality constraints on the dual problem, $0 \leq \alpha \leq C$, prove that the problem has a unique solution. **Hint:** Formulate the KKT system for the problem and check if it satisfies a statement in Lemma 11.3. You may use the fact that $G = ZZ^T \succeq 0$.

- (b) Now, considering the inequality constraints, discuss why the problem is yet admitting a unique solution α^* .

11.6. Consider the following equality-constrained QP problem

$$\begin{aligned} \min_{(x,y,z) \in \mathbb{R}^3} \quad & 3x^2 + 2y^2 + 2z^2 - 2yz - 8x - 3y - 3z, \quad \text{subj.to} \\ & x + z = 1, \\ & y + z = 3. \end{aligned} \tag{11.92}$$

Begin with pencil-and-paper:

- (a) Formulate the KKT system for (11.92) of the form (11.5) by identifying $A \in \mathbb{R}^{3 \times 3}$, $C \in \mathbb{R}^{2 \times 3}$, and vectors $\mathbf{b} \in \mathbb{R}^3$ and $\mathbf{c} \in \mathbb{R}^2$.
- (b) Solve the QP problem (11.92) by the **null-space approach**. In particular, specify the matrices Y and Z and compute $\mathbf{w}_Y$ and $\mathbf{w}_Z$.

Now, use your computer:

- (c) Implement the null-space algorithm presented in Section 11.2.3 to find the minimizer of (11.92) numerically.
- (d) Implement the range-space (Schur complement) method presented in Section 11.2.2 to find the minimizer of (11.92) numerically.

11.7. Now, consider the following inequality-constrained QP problem

$$\begin{aligned} \min_{\mathbf{x} \in \mathbb{R}^3} \quad & Q(\mathbf{x}) := (x_1 - 1)^2 + 2(x_2 - 2)^2 + 2(x_3 - 2)^2 - 2x_1x_2, \quad \text{subj.to} \\ & x_1 + x_2 - 3x_3 \leq 0, \\ & 4x_1 - x_2 + x_3 \leq 1. \end{aligned} \tag{11.93}$$

Implement the **interior-point method** with the Newton's iterative update, presented in Section 11.6, to solve the QP problem (11.93).

- (a) As in Exercise 6, you should first formulate the KKT system for (11.93) by identifying $A \in \mathbb{R}^{3 \times 3}$, $D \in \mathbb{R}^{2 \times 3}$, and vectors $\mathbf{b} \in \mathbb{R}^3$ and $\mathbf{d} \in \mathbb{R}^2$.
- (b) Choose a *feasible* initial value $(\mathbf{x}_0, \boldsymbol{\mu}_0, \mathbf{z}_0)$.
- (c) Select the algorithm parameter $\sigma \in [0, 1]$ *appropriately* for Newton's method.
- (d) Discuss how to determine α in (11.77) in order for the new iterate $(\mathbf{x}_{k+1}, \boldsymbol{\mu}_{k+1}, \mathbf{z}_{k+1})$ to stay feasible.

APPENDIX P

Projects

Finally we add projects.

Contents of Projects

P.1. mCLESS	364
P.2. Gaussian Sailing to Overcome Local Minima Problems	375
P.3. Quasi-Newton Methods Using Partial Information of the Hessian	377
P.4. Effective Preprocessing Technique for Filling Missing Data	379

P.1. mCLESS

Note: Some **machine learning** algorithms are considered as **black boxes**, because

- the models are sufficiently complex and
- they are not straightforwardly interpretable to humans.

Lack of interpretability in predictive models can **undermine trust** in those models, especially in **health care**, in which so many decisions are – literally – life and death issues [59].

Remark P.1. Why is Interpretability Important?

1. Understand the strengths and weaknesses of the model
2. Better feedback
3. Enhanced probability of adoption and success
4. Discover insights

Project Objectives

- Develop an **interpretable machine learning algorithm**.
 - Formulated with the least-squares error.
 - We call it the *Multi-Class Least-Error-Square Sum* (**mCLESS**).
- Compare it with traditional methods, for various datasets.

P.1.1. Review: Simple classifiers

The **Perceptron** [64] (or Adaline) is the simplest artificial neuron that makes decisions for datasets of two classes by *weighting up evidence*.

- Inputs: feature values $\mathbf{x} = [x_1, x_2, \dots, x_d]$
- Weight vector and bias: $\mathbf{w} = [w_1, w_2, \dots, w_d]^T, w_0$
- Net input:

$$z = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_d x_d \quad (\text{P.1.1})$$

- *Activation*:

$$\phi(z) = \begin{cases} 1, & \text{if } z \geq \theta \\ 0, & \text{otherwise,} \end{cases} \quad (\text{P.1.2})$$

where θ is a threshold. When the logistic sigmoid function is chosen for the **activation function**, i.e., $\phi(z) = 1/(1 + e^{-z})$, the resulting classifier is called the **Logistic Regression**.

Remark P.2. Note that the net input in (P.1.1) represents a **hyper-plane** in $\mathbb{R}^d$.

- More complex neural networks can be built, stacking the simple artificial neurons as building blocks.
- Machine learning (ML) is to train weights from datasets of an arbitrary number of classes.
 - The weights must be trained in such a way that *data points in a class are heavily weighted by the corresponding part of weights*.
- The **activation function** is incorporated in order
 - (a) **to keep the net input restricted to a certain limit** as per our requirement and, more importantly,
 - (b) **to add nonlinearity** to the network.

P.1.2. The mCLESS classifier

Here we present a new classifier which is based on a least-squares formulation and able to classify datasets having arbitrary numbers of classes. Its nonlinear expansion will also be suggested.

Two-layer Neural Networks

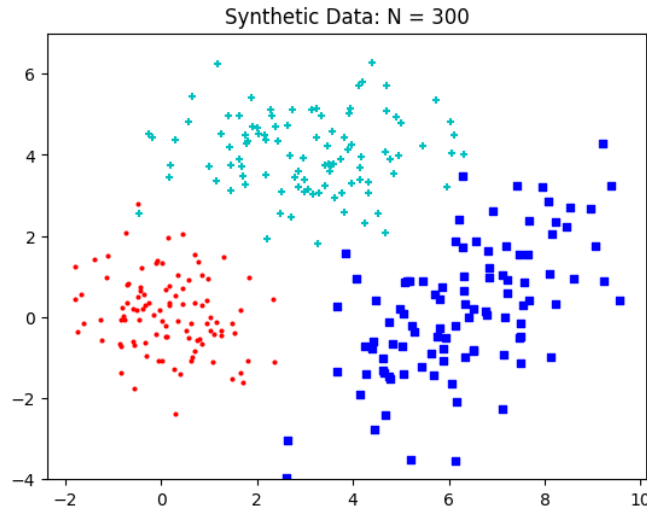


Figure P.1: A synthetic data of three classes.

- In order to describe the proposed algorithm effectively, we exemplify a synthetic data of three classes, as shown in Figure P.1, in which each class has 100 points.
- A point in the c -th class is expressed as

$$\mathbf{x}^{(c)} = [x_1^{(c)}, x_2^{(c)}] = [x_1, x_2, c] \quad c = 0, 1, 2,$$

where the number in $()$ in the superscript denotes the class that the point belongs to.

- Let's consider an artificial neural network of the identity activation and no hidden layer, for simplicity.

A set of weights can be trained in a way that **points in a class are heavily weighted by the corresponding part of weights**, i.e.,

$$w_0^{(j)} + w_1^{(j)} x_1^{(i)} + w_2^{(j)} x_2^{(i)} = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \quad (\text{P.1.3})$$

where δ_{ij} is called the Kronecker delta and $w_0^{(j)}$ is a bias for the class j .

- The weights can be determined by the least-squares method.
- We will call the algorithm the **Multi-Class Least-Error-Square Sum (mCLESS)**.

mCLESS: Algebraic Formulation

Training

- **Dataset:** We express the dataset $\{X, \mathbf{y}\}$ used for Figure P.1 by

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ \vdots & \vdots \\ x_{N1} & x_{N2} \end{bmatrix} \in \mathbb{R}^{N \times 2}, \quad \mathbf{y} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix}, \quad (\text{P.1.4})$$

where $c_i \in \{0, 1, 2\}$, the class number.

- **The algebraic system:** It can be formulated using (P.1.3).
 - Define the **information matrix**:

$$A = \begin{bmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ \vdots & \vdots & \vdots \\ 1 & x_{N1} & x_{N2} \end{bmatrix} \in \mathbb{R}^{N \times 3}. \quad (\text{P.1.5})$$

Note. The information matrix can be made using

```
A = np.column_stack((np.ones([N,1]),X))
```

- The **weight matrix** to be learned is:

$$W = [\mathbf{w}^{(0)}, \mathbf{w}^{(1)}, \mathbf{w}^{(2)}] = \begin{bmatrix} w_0^{(0)} & w_0^{(1)} & w_0^{(2)} \\ w_1^{(0)} & w_1^{(1)} & w_1^{(2)} \\ w_2^{(0)} & w_2^{(1)} & w_2^{(2)} \end{bmatrix}, \quad (\text{P.1.6})$$

where the j -th column weights heavily points in the j -th class.

– Define the **source matrix**:

$$B = [\delta_{c_i,j}] \in \mathbb{R}^{N \times 3}. \quad (\text{P.1.7})$$

For example, if the i -th point is in Class 0, then the i -th row of B is $[1, 0, 0]$.

- Then the **multi-column least-squares** (MC-LS) problem reads

$$\widehat{W} = \arg \min_W \|AW - B\|^2, \quad (\text{P.1.8})$$

which can be solved by the **method of normal equations**:

$$(A^T A) \widehat{W} = A^T B, \quad A^T A \in \mathbb{R}^{3 \times 3}. \quad (\text{P.1.9})$$

- **The output of training:** The weight matrix $\widehat{W}$.

Note: The normal matrix $A^T A$ is occasionally singular, particularly for small datasets. In the case, the MC-LS problem can be solved using the **singular value decomposition (SVD)**.

Prediction

The prediction step in the mCLESS is quite simple:

- Let $[x_1, x_2]$ be a new point.
- Compute

$$[1, x_1, x_2] \widehat{W} = [p_0, p_1, p_2], \quad \widehat{W} \in \mathbb{R}^{3 \times 3}. \quad (\text{P.1.10})$$

Note. Ideally, if the point $[x_1, x_2]$ is in class j , then p_j is near 1, while others would be near 0. Thus p_j is the largest.

- Decide the class c :

$$c = \text{np.argmax}([p_0, p_1, p_2], \text{axis} = 1). \quad (\text{P.1.11})$$

Experiment P.3. mCLESS, with a Synthetic Dataset

- As a data preprocessing, the dataset X is scaled column-wisely so that **the maximum value in each column is 1 in modulus**.
- The training is carried out with randomly selected 70% the dataset.
- The output of training, $\widehat{W}$, represents three sets of parallel lines.
 - Let $[w_0^{(j)}, w_1^{(j)}, w_2^{(j)}]^T$ be the j -th column of $\widehat{W}$. Define $L_j(x_1, x_2)$ as

$$L_j(x_1, x_2) = w_0^{(j)} + w_1^{(j)}x_1 + w_2^{(j)}x_2, \quad j = 0, 1, 2. \quad (\text{P.1.12})$$
 - Figure P.2 depicts $L_j(x_1, x_2) = 0$ and $L_j(x_1, x_2) = 1$ superposed on the training set.
- It follows from (P.1.11) that the mCLESS can be viewed as an **one-versus-rest (OVR)** classifier; see Section 3.2.3.

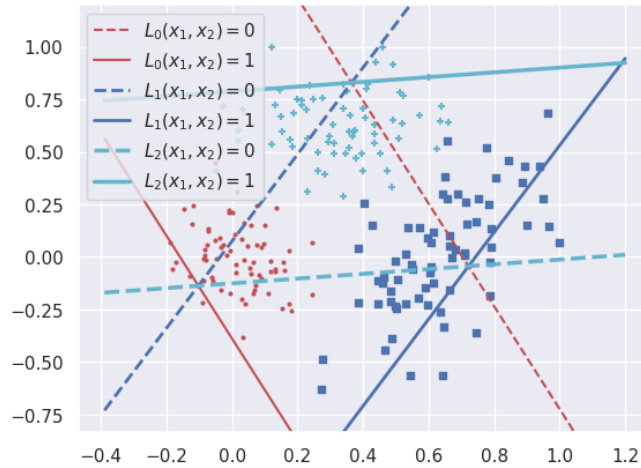


Figure P.2: Lines represented by the weight vectors. mCLESS is interpretable!

The whole algorithm (training-prediction) is run 100 times, with randomly splitting the dataset into 70:30 parts respectively for training and prediction; which results in **97.87% and 0.00171 sec** for the average accuracy and e-time. The used is a laptop of an Intel Core i7-10750H CPU at 2.60GHz.

P.1.3. Feature expansion

Remark P.4. **Nonlinear mCLESS**

- The mCLESS so far is a **linear classifier**.
- As for other classifiers, its **nonlinear expansion** begins with a data transformation, more precisely, **feature expansion**.
- For example, the **Support Vector Machine (SVM)** replaces the dot product of feature vectors (point) with the result of a kernel function applied to the feature vectors, in the construction of the Gram matrix:

$$\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) \approx \sigma(\mathbf{x}_i) \cdot \sigma(\mathbf{x}_j),$$

where σ is a function for feature expansion.

- Thus, without an explicit expansion of feature vectors, the SVM can incorporate the effect of data transformation effectively. Such a technique is called the **kernel trick**. See Section 5.3.5.
- However, the **mCLESS** does not incorporate dot products between points.
 - As a result, we must **perform feature expansion without a kernel trick**, which results in an augmented normal matrix, expanded in both column and row directions.

Feature Expansion for mCLESS

- A feature expansion is expressed as

$$\begin{cases} \mathbf{x} = [x_1, x_2, \dots, x_d] \\ \mathbf{w} = [w_0, w_1, \dots, w_d]^T \end{cases} \Rightarrow \begin{cases} \tilde{\mathbf{x}} = [x_1, x_2, \dots, x_d, \sigma(\mathbf{x})] \\ \tilde{\mathbf{w}} = [w_0, w_1, \dots, w_d, w_{d+1}]^T \end{cases} \quad (\text{P.1.13})$$

where $\sigma()$ is a **feature function** of $\mathbf{x}$.

- Then, the expanded weights must be trained to satisfy

$$[1, \tilde{\mathbf{x}}^{(i)}] \tilde{\mathbf{w}}^{(j)} = w_0^{(j)} + w_1^{(j)} x_1^{(i)} + \dots + w_d^{(j)} x_d^{(i)} + w_{d+1}^{(j)} \sigma(\mathbf{x}^{(i)}) = \delta_{ij}, \quad (\text{P.1.14})$$

for all points in the dataset. Compare the equation with (P.1.3).

- The corresponding expanded information and weight matrices read

$$\tilde{A} = \left[\begin{array}{ccccc|c} 1 & x_{11} & x_{12} & \cdots & x_{1d} & \sigma(\mathbf{x}_1) \\ 1 & x_{21} & x_{22} & \cdots & x_{2d} & \sigma(\mathbf{x}_2) \\ & \vdots & & \ddots & & \vdots \\ 1 & x_{N1} & x_{N2} & \cdots & x_{Nd} & \sigma(\mathbf{x}_N) \end{array} \right], \quad \tilde{W} = \left[\begin{array}{cccc} w_0^{(0)} & w_0^{(1)} & \cdots & w_0^{(C-1)} \\ w_1^{(0)} & w_1^{(1)} & \cdots & w_1^{(C-1)} \\ \vdots & & \ddots & \vdots \\ w_d^{(0)} & w_d^{(1)} & \cdots & w_d^{(C-1)} \\ w_{d+1}^{(0)} & w_{d+1}^{(1)} & \cdots & w_{d+1}^{(C-1)} \end{array} \right], \quad (\text{P.1.15})$$

where $\tilde{A} \in \mathbb{R}^{N \times (d+2)}$, $\tilde{W} \in \mathbb{R}^{(d+2) \times C}$, and C is the number of classes.

- Feature expansion can be performed multiple times. When α features are added, the optimal weight matrix $\widehat{W} \in \mathbb{R}^{(d+1+\alpha) \times C}$ is the least-squares solution of

$$(\tilde{A}^T \tilde{A}) \widehat{W} = \tilde{A}^T B, \quad (\text{P.1.16})$$

where $\tilde{A}^T \tilde{A} \in \mathbb{R}^{(d+1+\alpha) \times (d+1+\alpha)}$ and B is the same as in (P.1.7).

Remark P.5. Various feature functions $\sigma()$ can be considered. Here we will focus on the **feature function** of the form

$$\sigma(\mathbf{x}) = \|\mathbf{x} - \mathbf{p}\|, \quad (\text{P.1.17})$$

the Euclidean distance between $\mathbf{x}$ and a prescribed point $\mathbf{p}$.

Now, the question is: “How can we find $\mathbf{p}$?”

Generation of the Synthetic Data

synthetic_data.py

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  from GLOBAL_VARIABLES import *
4
5  def generate_data(n,scale,theta):
6      # Normally distributed around the origin
7      x = np.random.normal(0,1, n); y = np.random.normal(0,1, n)
8      P = np.vstack((x, y)).T
9      # Transform
10     sx,sy = scale
11     S = np.array([[sx,0],[0,sy]])
12     c,s = np.cos(theta), np.sin(theta)
13     R = np.array([[c,-s],[s,c]]).T #T, due to right multiplication
14     return P.dot(S).dot(R)
15
16 def synthetic_data():
17     N=0
18     plt.figure()
19     for i in range(N_CLASS):
20         scale = SCALE[i]; theta = THETA[i]; N+=N_D1
21         D1 = generate_data(N_D1,scale,theta) +TRANS[i]
22         D1 = np.column_stack((D1,i*np.ones([N_D1,1])))
23         if i==0: DATA = D1
24         else:     DATA = np.row_stack((DATA,D1))
25         plt.scatter(D1[:,0],D1[:,1],s=15,c=COLOR[i],marker=MARKER[i])
26
27     np.savetxt(DAT_FILENAME,DATA,delimiter=',',fmt=FORMAT)
28     print('    saved: %s' %(DAT_FILENAME))
29
30     #xmin,xmax = np.min(DATA[:,0]), np.max(DATA[:,0])
31     ymin,ymax = np.min(DATA[:,1]), np.max(DATA[:,1])
32     plt.ylim([int(ymin)-1,int(ymax)+1])
33
34     plt.title('Synthetic Data: N = '+str(N))
35     myfigsave(FIG_FILENAME)
36     if __name__ == '__main__':
37         plt.show(block=False); plt.pause(5)
38
39 if __name__ == '__main__':
40     synthetic_data()

```

```
GLOBAL_VARIABLES.py
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  N_D1 = 100
5  FORMAT = '%.3f', '%.3f', '%d'
6
7  SCALE = [[1,1],[1,2],[1.5,1]]; TRANS = [[0,0],[6,0],[3,4]]
8  #SCALE = [[1,1],[1,1],[1,1]]; TRANS = [[0,0],[4,0],[8,0]]
9  THETA = [0,-0.25*np.pi, 0]
10 COLOR = ['r','b','c']
11 MARKER = ['.','s','+','*']
12 LINESTYLE = [['r--','r-'],['b--','b-'],['c--','c-']]
13
14 N_CLASS = len(SCALE)
15
16 DAT_FILENAME = 'synthetic.data'
17 FIG_FILENAME = 'synthetic-data.png'
18 FIG_INTERPRET = 'synthetic-data-interpret.png'
19
20 def myfigsave(figname):
21     plt.savefig(figname, bbox_inches='tight')
22     print('    saved: %s' %(figname))
```

What to do

1. Implement mCLESS:

- **Training.** You should implement modules for each of (P.1.5) and (P.1.7). Then use `Xtrain` and `ytrain` to get A and B .
- **Test.** Use the same module (implemented for A) to get A_{test} from `Xtest`. Then perform $P = (A_{\text{test}}) * \widehat{W}$ as in (P.1.10). Now, you can get the prediction using

```
prediction = np.argmax(P,axis=1);
```

which may be compared with `ytest` to obtain accuracy.

2. Use following datasets:

- **Synthetic datasets.** Generate two different synthetic datasets:

- ① Use Line 7 in `GLOBAL_VARIABLES.py`
- ② Use Line 8 in `GLOBAL_VARIABLES.py`

- **Real datasets.** Use public datasets such as `iris` and `wine`.

To get the public datasets, you may use:

```
from sklearn import datasets
data_read1 = datasets.load_iris()
data_read2 = datasets.load_wine()
```

3. Compare the performance of mCLESS with

- `LogisticRegression(max_iter = 1000)`
- `KNeighborsClassifier(5)`
- `SVC(gamma=2, C=1)`
- `RandomForestClassifier(max_depth=5, n_estimators=50, max_features=1)`

See Section 1.3.

4. (Optional for Undergraduate Students) Add modules for feature expansion, as described on page 371.

- For this, try to an **interpretable strategy** to find an effective point p such that the feature expansion with (P.1.17) improves accuracy.
- Experiment Steps 1-3.

5. Report your experiments with the code and results.

You may start with the **machine learning modelcode** in Section 1.3; add your own modules.

P.2. Gaussian Sailing to Overcome Local Minima Problems

- A **Gaussian smoothing** for a 2D function $f(x, y)$ can be achieved by storing the function to a 2D-array A , and applying a built-in function in `scipy`:

`scipy.ndimage.filters.gaussian_filter`

which requires a parameter σ (standard deviation for Gaussian kernel).

- Alternatively, one can employ an **averaging operator**; for example, apply a few iterations of the following convolution

$$\mathcal{S} * A, \quad \mathcal{S} \stackrel{\text{def}}{=} \frac{1}{(2 + c_w)^2} \begin{bmatrix} 1 & c_w & 1 \\ c_w & c_w^2 & c_w \\ 1 & c_w & 1 \end{bmatrix}, \quad (\text{P.2.1})$$

for $c_w \geq 0$. Since

$$\begin{bmatrix} 1 & c_w & 1 \\ c_w & c_w^2 & c_w \\ 1 & c_w & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ c_w \\ 1 \end{bmatrix} [1 \ c_w \ 1],$$

the convolution smoothing can be implemented easily and conveniently as

$$\mathcal{S} * A = \frac{1}{(2 + c_w)^2} \left(\begin{bmatrix} 1 \\ c_w \\ 1 \end{bmatrix} * ([1 \ c_w \ 1] * A) \right),$$

which is a horizontal filtering followed by a vertical filtering and finally a scaling with the factor $1/(2 + c_w)^2$.

- In this project, you will explore the gradient descent method with line search (4.19) for the computation of the global minimizer of multiple local minima problems.

Tasks to do

1. Go to <http://www.sfu.ca/~ssurjano/optimization.html> (Virtual Library of Simulation Experiments) and select **three functions** of your interests having multiple local minima. (e.g., two of them are the Ackley function and Griewank function.)
2. Store each of the functions in a 2D-array A which has dimensions large enough.
3. Compute

$$A_\sigma = \text{gaussian_filter}(A, \sigma), \text{ or } A_t = \underbrace{\mathcal{S} * \mathcal{S} * \cdots * \mathcal{S}}_{t\text{-times}} * A,$$

which can be considered as a convex/smooth approximation of the original function. You can use it for the **estimation** of f and its derivatives at $\mathbf{x}_n$.

4. Design a set of σ/t -values $\mathcal{T}$ (including “0” as the last entry) so that given an initial point $\mathbf{x}_0$, the Gaussian homotopy continuation method discussed in Remark 4.13 can locate the global minimum, while the algorithm (4.19) can find only a local minimum, for each of the functions.

Report. Submit hard copies of your experiences.

- Attach a “summary” or “conclusion” page at the beginning of report.
- Your work process.
- Your code.
- Figures; for each of the functions, include a figure that shows movement of the minimizers for all σ ’s or t ’s in $\mathcal{T}$.
- Discuss pros and cons of the Gaussian sailing strategy.

You may work in a group of two people; however, you must report individually.

P.3. Quasi-Newton Methods Using Partial Information of the Hessian

Consider a unconstrained optimization problem of the form

$$\min_{\mathbf{x} \in \mathbb{R}^d} f(\mathbf{x}), \quad (\text{P.3.1})$$

and Newton's method

$$\mathbf{x}_{n+1} = \arg \min_{\mathbf{x}} \mathcal{Q}_n(\mathbf{x}) = \mathbf{x}_n - \gamma [\nabla^2 f(\mathbf{x}_n)]^{-1} \nabla f(\mathbf{x}_n), \quad (\text{P.3.2})$$

where

$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \frac{\partial^2 f}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{bmatrix} \in \mathbb{R}^{d \times d}. \quad (\text{P.3.3})$$

Known. (Remark 4.15). Where applicable, **Newton's method converges much faster** towards a local extremum than gradient descent. Every local minimum has a neighborhood such that, if we start within this neighborhood, Newton's method with step size $\gamma = 1$ converges quadratically assuming the Hessian is invertible and Lipschitz continuous.

Issues. The central issue with Newton's method is that we need to be able to compute the **inverse Hessian matrix**.

- For ML applications, the dimensionality of the problem can be of the **order of thousands or millions**; computing the Hessian or its inverse is often impractical.
- Because of these reasons, Newton's method is **rarely used in practice** to optimize functions corresponding to **large problems**.
- Luckily, the above algorithm can still work even if the Hessian is replaced by a **good approximation**.
- Various "**quasi-Newton**" methods have been developed so as to approximate and update the Hessian matrix (without evaluating the second derivatives).

Objectives. You are required to perform tasks including the following.

• **Algorithm Development:**

- Note the Newton's search direction is $-H^{-1}\nabla f(\mathbf{x}_n)$, where $H = \nabla^2 f(\mathbf{x}_n)$.
- Select a set of k components ($k \ll d$) from $\nabla f \in \mathbb{R}^d$ which **would dominate** the search direction. Then, for some permutation P ,

$$P \nabla f(\mathbf{x}_n) = \begin{bmatrix} \widehat{\nabla f}(\mathbf{x}_n) \\ \widetilde{\nabla f}(\mathbf{x}_n) \end{bmatrix}. \quad (\text{P.3.4})$$

- Construct $\widehat{H} = \widehat{H}(\mathbf{x}_n) \in \mathbb{R}^{k \times k}$ (using finite differences) to solve

$$\widehat{H} \widehat{\mathbf{q}} = \widehat{\nabla f}(\mathbf{x}_n). \quad (\text{P.3.5})$$

- Find a scaling factor $\sigma > 0$ such that

$$-P^T \begin{bmatrix} \sigma \widehat{\mathbf{q}} \\ \widetilde{\nabla f}(\mathbf{x}_n) \end{bmatrix} \quad (\text{P.3.6})$$

is the **final search direction**.

- *Suggestions:* For $d \geq 10$, $k = 2 \sim 5$ and

$$\sigma \cong \frac{\|\widehat{\nabla f}(\mathbf{x}_n)\|}{\|\widehat{\mathbf{q}}\|}. \quad (\text{P.3.7})$$

• **Comparisons:**

- Implement (or download codes for) the original Newton's method and one of quasi-Newton methods (e.g., BFGS).
- Let's call our method the **partial Hessian (PH)-Newton** method. Compare the PH-Newton with those known methods for: the number of iterations, the total elapsed time, convergence behavior, and stability/robustness.
- Test with e.g. the Rosenbrock function defined on $\mathbb{R}^d$, $d \geq 10$, with various initial points $\mathbf{x}_0$.

P.4. Effective Preprocessing Technique for Filling Missing Data

(From Remark 6.2). **Data preparation** is **difficult** because the process is *not objective*, and it is **important** because ML algorithms *learn from data*. Consider the following.

- Preparing data for analysis is one of the most **important** steps in any data-mining project – and traditionally, one of the most **time consuming**.
- Often, it takes **up to 80% of the time**.
- Data preparation is **not a once-off process**; that is, it is iterative as you understand the problem deeper on each successive pass.

Known. For missing values, three different steps can be executed.

- **Removal of samples (rows) or features (columns):**
It is the simplest and efficient method for handling the missing data.
 However, we may end up removing too many samples or features.
- **Filling the missing values manually:**
This is **one of the best-chosen methods**.
 But there is one limitation that when there are large data set, and missing values are significant.
- **Imputing missing values using computed values:**
The missing values can also be occupied by computing **mean, median, or mode** of the observed given values. Another method could be the predictive values that are computed by using any ML or Deep Learning algorithm.
 But one drawback of this approach is that it can generate bias within the data as the calculated values are not accurate concerning the observed values.

Objectives.

- **Algorithm Development:**

- Think **a good strategy or two** for filling the missing values, if it is to be done **manually**.
- What information available from the dataset is useful and help us build a good strategy?
- *Suggestions:* Use **near-values** to interpolate; try to employ the concept of **feature importance**, if available.

- **Comparisons:**

- For the Wine dataset, for example, erase $r\%$ data values in random; $r = 5, 10, 20$.
- Compare your new filling strategy with ① the simple **sample removal** method and ② the **imputation strategy** using mean, median, or mode.
- Perform **accuracy analysis** for various classifiers, e.g., logistic regression, support vector machine, and random forests.

Bibliography

- [1] W. BELSON, *Matching and prediction on the principle of biological classification*, JRSS, Series C, Applied Statistics, 8 (1959), pp. 65–75.
- [2] Y. BENGIO, *Deep learning of representations: Looking forward*, in Statistical Language and Speech Processing, A. H. Dediu, C. Martín-Vide, R. Mitkov, and B. Truthe, eds., vol. 7978 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2013, pp. 1–37.
- [3] Y. BENGIO, A. COURVILLE, AND P. VINCENT, *Representation learning: A review and new perspectives*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 35 (2013), pp. 1798–1828.
- [4] Y. BENGIO, P. LAMBLIN, D. POPOVICI, AND H. LAROCHELLE, *Greedy layer-wise training of deep networks*, in NIPS'06 Proceedings of the 19th International Conference on Neural Information Processing Systems, NIPS, 2006, pp. 153–160.
- [5] Y. BENGIO, Y. LECUN, AND G. HINTON, *Deep learning*, Nature, 521 (2015), pp. 436–444.
- [6] A. BJÖRCK, *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, 1996.
- [7] L. BOTTOU, F. E. CURTIS, AND J. NOCEDAL, *Optimization methods for large-scale machine learning*, SIAM Rev., 60 (2018), pp. 223–311.
- [8] L. BREIMAN, *Random forests*, Machine Learning, 45 (2001), pp. 5–32.
- [9] C. G. BROYDEN, *The convergence of a class of double-rank minimization algorithms 1. General considerations*, IMA Journal of Applied Mathematics, 6 (1970), pp. 76–90.
- [10] J. BUNCH AND L. KAUFMAN, *Some stable methods for calculating inertia and solving symmetric linear systems*, Math. Comput., 31 (1977), pp. 163–179.
- [11] R. B. CATTELL, *The description of personality: Basic traits resolved into clusters*, Journal of Abnormal and Social Psychology, 38 (1943), pp. 476–506.
- [12] C. CORTES AND V. N. VAPNIK, *Support-vector networks*, Machine Learning, 20 (1995), pp. 273–297.
- [13] G. G. DANIEL, *Deterministic and Nondeterministic Turing Machine*, Springer Netherlands, Dordrecht, 2013, pp. 624–624.

- [14] R. DECHTER, *Learning while searching in constraint-satisfaction-problems*, in AAAI, T. Kehler, ed., Morgan Kaufmann, 1986, pp. 178–185.
- [15] S. DEERWESTER, S. DUMAIS, G. FURNAS, T. LANDAUER, AND R. HARSHMAN, *Indexing by latent semantic analysis.*, Journal of the American Society for Information Science, (1990), pp. 391–407.
- [16] H. E. DRIVER AND A. L. KROEBER, *Quantitative expression of cultural relationships*, in University of California Publications in American Archaeology and Ethnology, vol. Quantitative Expression of Cultural Relationships, 1932, pp. 211–256.
- [17] L. ELDÉN, *Numerical linear algebra in data mining*, Acta Numerica, 15 (2006), pp. 327 – 384.
- [18] M. ESTER, H.-P. KRIEGEL, J. SANDER, AND X. XU, *A density-based algorithm for discovering clusters in large spatial databases with noise*, in Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, KDD-96, AAAI Press, 1996, pp. 226–231.
- [19] A. FISHER, C. RUDIN, AND F. DOMINICI, *Model class reliance: Variable importance measures for any machine learning model class, from the "rashomon" perspective*, (2018).
- [20] R. A. FISHER, *The use of multiple measurements in taxonomic problems*, Annals of Eugenics, 7 (1936), pp. 179–188.
- [21] R. FLETCHER, *A new approach to variable metric algorithms*, The Computer Journal, 13 (1970), pp. 317–322.
- [22] S. GERSCHGORIN, *Über die abgrenzung der eigenwerte einer matrix*, Izv. Akad. Nauk SSSR Ser. Mat., 7 (1931), pp. 746–754.
- [23] X. GLOROT, A. BORDES, AND Y. BENGIO, *Deep sparse rectifier neural networks*, in Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, G. Gordon, D. Dunson, and M. Dudík, eds., vol. 15 of Proceedings of Machine Learning Research, Fort Lauderdale, FL, USA, 11–13 Apr 2011, PMLR, pp. 315–323.
- [24] D. GOLDFARB, *A family of variable metric methods derived by variational means*, Mathematics of Computation, 24 (1970), pp. 23–26.
- [25] F. GOLUB AND C. V. LOAN, *Matrix Computations, 3rd Ed.*, The Johns Hopkins University Press, Baltimore, 1996.
- [26] G. H. GOLUB AND C. REINSCH, *Singular value decomposition and least squares solutions*, Numer. Math., 14 (1970), pp. 403–420.
- [27] B. GROSSER AND B. LANG, *An $\mathcal{O}(n^2)$ algorithm for the bidiagonal svd*, Lin. Alg. Appl., 358 (2003), pp. 45–70.

- [28] L. GUTTMAN, *A necessary and sufficient formula for matrix factoring*, Psychometrika, 22 (1957), pp. 79–81.
- [29] G. HINTON, , , G. DAHL, A.-R. MOHAMED, N. JAITLEY, A. SENIOR, V. VANHOUCHE, B. KINGSBURY, AND T. SAINATH, *Deep neural networks for acoustic modeling in speech recognition*, IEEE Signal Processing Magazine, 29 (2012), pp. 82–97.
- [30] G. E. HINTON, S. OSINDERO, AND Y.-W. TEH, *A fast learning algorithm for deep belief nets*, Neural Comput., 18 (2006), pp. 1527–1554.
- [31] G. E. HINTON AND R. SALAKHUTDINOV, *Reducing the dimensionality of data with neural networks*, Science, 313 (2006), pp. 504–507.
- [32] T. K. HO, *Random decision forests*, in Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, 1995, pp. 278–282.
- [33] H. HOTELLING, *Analysis of a complex of statistical variables into principal components*, Journal of Educational Psychology, 24 (1933), pp. 417–441 and 498–520.
- [34] ———, *Relations between two sets of variates*, Biometrika, 28 (1936), pp. 321–377.
- [35] A. K. JAIN AND R. C. DUBES, *Algorithms for Clustering Data*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
- [36] W. KARUSH, *Minima of functions of several variables with inequalities as side constraints*, M.Sc. Dissertation, Department of Mathematics, Univ. of Chicago, Chicago, Illinois, 1939.
- [37] L. KAUFMAN AND P. ROUSSEEUV, *Clustering by means of medoids*, in Statistical Data Analysis Based on the L^1 -Norm and Related Methods, Y. Dodge, ed., North-Holland, 1987, pp. 405–416.
- [38] L. KAUFMAN AND P. J. ROUSSEEUV, *Finding Groups in Data: An Introduction to Cluster Analysis*, John Wiley & Sons, Inc., 1990.
- [39] T. KOHONEN, *Self-organized formation of topologically correct feature maps*, Biological Cybernetics, 43 (1982), pp. 59–69.
- [40] T. KOHONEN, *Self-Organizing Maps*, Springer series in information sciences, 30, Springer, Berlin, 3rd ed., Dec. 2001.
- [41] A. KRIZHEVSKY, I. SUTSKEVER, AND G. E. HINTON, *Imagenet classification with deep convolutional neural networks*, Commun. ACM, 60 (2017), pp. 84–90.
- [42] H. W. KUHN AND A. W. TUCKER, *Nonlinear programming*, in Proceedings of 2nd Berkeley Symposium, Berkeley, CA, USA, 1951, University of California Press., pp. 481–492.
- [43] K. LANGE, D. R. HUNTER, AND I. YANG, *Optimization transfer using surrogate objective functions*, Journal of Computational and Graphical Statistics, 9 (2000), pp. 1–20.

- [44] R. LEBLOND, F. PEDREGOSA, AND S. LACOSTE-JULIEN, *Improved asynchronous parallel optimization analysis for stochastic incremental methods*, CoRR, abs/1801.03749 (2018).
- [45] Y. LECUN, B. E. BOSER, J. S. DENKER, D. HENDERSON, R. E. HOWARD, W. E. HUBBARD, AND L. D. JACKEL, *Handwritten digit recognition with a back-propagation network*, in Advances in Neural Information Processing Systems 2, D. S. Touretzky, ed., Morgan-Kaufmann, 1990, pp. 396–404.
- [46] Y. LECUN, S. CHOPRA, R. HADSELL, F. J. HUANG, AND ET AL., *A tutorial on energy-based learning*, in PREDICTING STRUCTURED DATA, MIT Press, 2006.
- [47] C. LEMARECHAL, *Cauchy and the gradient method*, Documenta Mathematica, Extra Volume, (2012), pp. 251–254.
- [48] S. P. LLOYD, *Least square quantization in pcm*, Bell Telephone Laboratories Report, 1957. Published in journal: S. P. Lloyd (1982). *Least squares quantization in PCM*. IEEE Transactions on Information Theory. 28 (2): pp. 129–137.
- [49] M. LOURAKIS, *A brief description of the Levenberg-Marquardt algorithm implemented by levmar*, Technical Report, Institute of Computer Science, Foundation for Research and Technology – Hellas, 2005.
- [50] D. G. LOWE, *Object recognition from local scale-invariant features*, in Proceedings of the Seventh IEEE International Conference on Computer Vision, IEEE, 1999, pp. 1150–1157.
- [51] S. MANDT, M. D. HOFFMAN, AND D. M. BLEI, *Stochastic gradient descent as approximate Bayesian inference*, Journal of Machine Learning Research, 18 (2017), pp. 1–35.
- [52] D. MARQUARDT, *An algorithm for least-squares estimation of nonlinear parameters*, Journal of the Society for Industrial and Applied Mathematics, 11 (1963), pp. 431–441.
- [53] H. MOBAHI AND J. W. FISHER III, *On the link between gaussian homotopy continuation and convex envelopes*, in Energy Minimization Methods in Computer Vision and Pattern Recognition. Lecture Notes in Computer Science, vol. 8932, X.-C. Tai, E. Bae, T. F. Chan, and M. Lysaker, eds., Hong Kong, China, 2015, Springer, pp. 43–56.
- [54] R. T. NG AND J. HAN, *Efficient and effective clustering methods for spatial data mining*, in Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, San Francisco, CA, USA, 1994, Morgan Kaufmann Publishers Inc., pp. 144–155.
- [55] —, *CLARANS: A method for clustering objects for spatial data mining*, IEEE Transactions on Knowledge and Data Engineering, 14 (2002), pp. 1003–1016.
- [56] M. NIELSEN, *Neural networks and deep learning*. (The online book can be found at <http://neuralnetworksanddeeplearning.com>), 2013.

- [57] J. NOCEDAL AND S. J. WRIGHT, *Numerical Optimization (2nd Ed.)*, Springer-Verlag, Berlin, New York, 2006.
- [58] K. PEARSON, *On lines and planes of closest fit to systems of points in space*, Philosophical Magazine, 2 (1901), pp. 559–572.
- [59] J. PETCH, S. DI, AND W. NELSON, *Opening the black box: The promise and limitations of explainable machine learning in cardiology*, Canadian Journal of Cardiology, 38 (2022), pp. 204–213.
- [60] R. RAINA, A. MADHAVAN, AND A. Y. NG, *Large-scale deep unsupervised learning using graphics processors*, in Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09, New York, NY, USA, 2009, ACM, pp. 873–880.
- [61] C. R. RAO, *The utilization of multiple measurements in problems of biological classification*, Journal of the Royal Statistical Society. Series B: Statistical Methodology, 10 (1948), pp. 159–203.
- [62] S. RASCHKA AND V. MIRJALILI, *Python Machine Learning, 3rd Ed.*, Packt Publishing Ltd., Birmingham, UK, 2019.
- [63] H. ROBBINS AND S. MONRO, *A stochastic approximation method*, Ann. Math. Statist., 22 (1951), pp. 400–407.
- [64] F. ROSENBLATT, *The Perceptron, a Perceiving and Recognizing Automaton Project Para*, Report: Cornell Aeronautical Laboratory, Cornell Aeronautical Laboratory, 1957.
- [65] D. E. RUMELHART, G. E. HINTON, AND R. J. WILLIAMS, *Learning representations by back-propagating errors*, Nature, 323 (1986), pp. 533–536.
- [66] T. N. SAINATH, B. KINGSBURY, G. SAON, H. SOLTAU, M. A. R., G. DAHL, AND B. RAMABHADHAN, *Deep convolutional neural networks for large-scale speech tasks*, Neural Netw., 64 (2015), pp. 39–48.
- [67] G. SALTON, A. WONG, AND C.-S. YANG, *A vector space model for automatic indexing*, Communications of the ACM, 18 (1975), pp. 613–620.
- [68] F. SANTOSA AND W. W. SYMES, *Linear inversion of band-limited reflection seismograms*, SIAM Journal on Scientific Computing, 7 (1986), pp. 1307–1330.
- [69] J. SCHMIDHUBER, *Deep learning in neural networks: An overview*, Neural Networks, 61 (2015), pp. 85–117.
- [70] B. SCHÖLKOPF, A. SMOLA, AND K.-R. MÜLLER, *Kernel principal component analysis*, in Artificial Neural Networks — ICANN'97, EDITORS, ed., vol. 1327 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1997, pp. 583–588.
- [71] E. SCHUBERT AND P. J. ROUSSEEUW, *Faster k -medoids clustering: Improving the PAM, CLARA, and CLARANS algorithms*, CoRR, abs/1810.05691 (2018).

- [72] D. F. SHANNO, *Conditioning of quasi-Newton methods for function minimization*, Mathematics of Computation, 24 (1970), pp. 647–656.
- [73] I. SUTSKEVER, O. VINYALS, AND Q. V. LE, *Sequence to sequence learning with neural networks*, in Advances in Neural Information Processing Systems 27, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds., Curran Associates, Inc., 2014, pp. 3104–3112.
- [74] O. TAUSSKY, *Bounds for characteristic roots of matrices*, Duke Math. J., 15 (1948), pp. 1043–1044.
- [75] R. TIBSHIRANI, *Regression shrinkage and selection via the LASSO*, Journal of the Royal Statistical Society. Series B (methodological), 58 (1996), pp. 267–288.
- [76] R. C. TRYON, *Cluster Analysis: Correlation Profile and Orthometric (factor) Analysis for the Isolation of Unities in Mind and Personality*, Edwards Brothers, 1939.
- [77] R. VARGA, *Matrix Iterative Analysis, 2nd Ed.*, Springer-Verlag, Berlin, Heidelberg, 2000.
- [78] J. H. M. WEDDERBURN, *Lectures on Matrices*, Amer. Math. Soc., New York, 1934.
- [79] WIKIPEDIA, *Self-organizing map*.
- [80] P. R. WILLEMS, B. LANG, AND C. VÖMEL, *Computing the bidiagonal SVD using multiple relatively robust representations*, SIAM Journal on Matrix Analysis and Applications, 28 (2006), pp. 907–926.
- [81] M. H. WRIGHT, *Interior methods for constrained optimization*, Acta Numerica, 1 (1992), pp. 341–407.
- [82] K. XU, J. L. BA, R. KIROS, K. CHO, A. COURVILLE, R. SALAKHUTDINOV, R. S. ZEMEL, AND Y. BENGIO, *Show, attend and tell: Neural image caption generation with visual attention*, in Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15, 2015, pp. 2048–2057.
- [83] S. XU, *An Introduction to Scientific Computing with MATLAB and Python Tutorials*, CRC Press, Boca Raton, FL, 2022.
- [84] J. ZUBIN, *A technique for measuring like-mindedness*, The Journal of Abnormal and Social Psychology, 33 (1938), pp. 508–516.

Index

- `;`, Python slicing, 21
- `__init__()` constructor, 31
- acceptance criterion, 85
- activation function, 365
- active set, 352
- Adaline, 47, 95, 365
- adaline, 37
- adaptive linear neurons, 37
- adaptive step size, 63
- agglomerative clustering, 224, 226
- AGNES, 224, 228
- algorithmic parameter, 49
- approximate complementarity, 359
- approximate Hessian, 83
- argmax, numpy, 368
- artificial neural networks, 92
- artificial neurons, 38
- artificial topographic maps, 248
- association rule discovery, 302
- attributes, 31
- averaged perceptron, 44
- averaging operator, 375
- back-propagation, 281
- back-propagation algorithm, 287
- backtracking line search, 63
- barrier functions, 357
- barrier parameter, 357
- batch gradient descent, 51
- best split, 126
- between-class scatter, 176
- between-class scatter matrix, 176, 178
- BFGS algorithm, 72
- bias, 40, 100, 266
- bias-variance tradeoff, 101
- binary classifier, 38
- binary decision tree, 123
- binding constraint, 111
- binomial distribution, 96
- bisecting K-Means algorithm, 217
- black boxes, 364
- bootstrap sample, 127
- border point, 232
- `call_get_cubes.py`, 25
- categorical data, 137
- Cauchy, Augustin-Louis, 56
- central path, 356
- centroid, 204
- centroid linkage, 227
- chain rule, 265
- chi-squared error criterion, 81
- child class, 34
- CIFAR, 265
- CLARA, 221, 222
- CLARANS, 223
- class, 29, 30
- class attribute, 33
- class-membership probability, 95
- `Classes.py`, 34
- classification, 2, 302
- classification error, 124
- Clay Institute, 207
- cluster analysis, 199
- cluster cohesion, 241
- cluster separation, 241
- cluster validation, 236
- clustering, 2, 11, 199, 302
- CNN, 265, 289
- code block, 20
- column-stochastic matrix, 325
- `column_stack`, numpy, 367
- competitive learning, 248, 251
- complementary slackness, 108, 111, 355
- complete linkage, 226

- computational complexity theory, 207
- constraint set, 56
- contrapositive, 111
- convergence, iterative methods, 345
- convolution, 292
- convolutional network, 292
- convolutional neural network, 265
- convolutional neural networks, 289
- core points, 232
- correlation, 237
- cosine distance measure, 317
- cost function, 48
- covariance matrix, 165, 175
- criterion function, 142
- critical point, 60
- critical points, 68
- cumulative explained variance, 155
- curse of dimensionality, 130, 140
- curvature condition, 73
- cython, 18

- damped least-squares, 81
- damping factor, 327
- damping parameter, 84
- data compression, 149, 308
- data gap, 301
- data matrix, 151, 189
- data mining, 301
- data normalization, 50, 138
- data preparation, 133, 135, 379
- data preprocessing, 49, 133, 134
- DBSCAN, 231
- decision tree, 122
- decreasing step-size sequence, 78
- deep learning, 261, 288
- deep neural network, 278, 280
- deep neural networks, 288
- deepcopy, 22
- definite, 335
- dendrogram, 228
- denormalize.m, 182
- density estimation, 3
- density-based clustering, 231
- density-connected, 233
- description methods, 302

- desktop calculator, 19
- destructor, 33
- deviation/anomaly detection, 302
- diagonal dominance, 348
- DIANA, 224
- dictionary, 304
- dimensionality reduction, 149, 308
- directed graph, 346
- discriminant function, 252
- distance functions, 201
- divisive clustering, 224
- DLS, 81
- document weighting, 315
- domain, 56
- dot product of matrices, 335
- dual feasibility, 108
- dual problem, 106, 109, 114, 117, 120
- dual variable, 106, 107
- dyadic decomposition, 159

- eigenvalue locus theorem, 347
- elastic net regularization, 141
- elbow analysis, 259
- element-wise product, 284
- embedded methods, 141
- empirical loss, 8
- empirical risk, 9
- ensemble learning, 127
- entropy, 124, 244
- error, 284
- Euclidean distance, 203
- explained variance ratio, 155

- feasible solution, 55
- feature expansion, 119, 370
- feature extraction, 149
- feature function, 371
- feature importance, 146
- feature map, 254, 292, 293
- feature scaling, 50, 101, 134, 138
- feature selection, 140, 144, 149
- feature selection, automatic, 141
- feature selection/extraction, 308
- feed-forward neural network, 248
- filter methods, 141

- first-order necessary conditions, 107
- Fisher's LDA, 172
- frequently_used_rules.py, 23
- fully-connected networks, 288

- Gauss-Newton method, 83
- Gaussian homotopy continuation, 66
- Gaussian kernel, 117
- Gaussian sailing, 375
- Gaussian smoothing, 66, 375
- generalization error, 135, 140
- generalized eigenvalue problem, 176
- get_cubes.py, 25
- Gini impurity, 124
- GLOBAL_VARIABLES.py, 373
- GMRES, 350
- Golub-Reinsch SVD algorithm, 168
- goodness-of-fit measure, 81
- Google, 322
- Google matrix, 306, 327, 330
- Google Pagerank algorithm, 306
- Google search, 322
- Google search engine, 306
- gradient descent algorithm, 61, 63
- gradient descent method, 48, 56, 60, 61, 272
- Gram matrix, 115
- graph theory, 345
- greatest curvature, 70
- group average, 227

- Hadamard product, 284
- Hessian, 61, 67, 70
- Hessian matrix, 70, 377
- hidden layers, 269
- hierarchical clustering, 209, 224
- high bias, 100
- high variance, 100
- hold-out method, 44
- Horner's method, 26, 33
- horner, in Python, 27
- horner.m, 28
- Householder transformation, 168
- hyperparameter, 49
- hyperplane, 40, 365

- hypothesis formation, 300
- hypothesis space, 7

- identity function, 47
- image compression, 170
- impurity measure, 123
- incidence matrix, 238
- indentation, 20
- index, 313
- induced matrix norm, 61
- inertia of matrix, 338
- inference, statistical, 2
- information gain, 122, 123
- information matrix, 367
- information retrieval, 306, 312
- inheritance, 34
- initial centroids problem, 216
- inlinks, 323
- input layer, 269
- input neurons, 269
- inseparable, 44
- install Python packages, 15
- instance, 30
- instantiation, 30
- integer encoding, 137
- intercept, 40
- interior-point method, 355, 361
- internal measures, 240
- Internet search, 322
- interpretability, 122, 265, 278, 364
- inverse document frequency, 315
- iris, 374
- Iris dataset, 53
- irreducible, 325
- irreducible matrix, 346

- Jacobian matrix, 82

- K-Means clustering, 211
- K-Medoids algorithm, 221
- k-nearest neighbor, 129
- Karush-Kuhn-Tucker conditions, 107
- KD-tree, 130
- kernel function, 117, 193
- kernel matrix, 192
- kernel method, 116

- kernel PCA, 189
- kernel PCA, summary, 195
- kernel principal component analysis, 149, 189
- kernel SVM, 116
- kernel trick, 117, 119, 193, 370
- keywords, 35
- KKT approach, 107
- KKT conditions, 107, 334, 355
- KKT matrix, 334
- Knowledge Discovery in Data, 301
- Kohonen map, 255
- Kohonen network, 250
- Kohonen SOM network, 250
- Kohonen, Teuvo, 248

- L2 regularization, 101
- L2 shrinkage, 101
- L2-pooling, 294
- Lagrange dual function, 106
- Lagrange multiplier, 143
- Lagrange multipliers, 104, 107
- Lagrangian, 107, 334
- Lagrangian (objective), 105
- largest margin, 45
- LASSO, 141
- latent semantic indexing, 313, 319
- lateral inhibition connections, 248
- lateral interaction, 253
- law of parsimony, 10
- Lawrence Page, 327
- lazy learner, 129
- lda_c3.m, 181
- lda_c3_visualize.m, 182
- lda_Fisher.m, 177
- ldl_test.py, 340
- learning error, 284
- learning rate, 41, 58, 254
- least curvature, 70
- left eigenvalue, 325
- left singular vectors, 153, 156
- Levenberg-Marquardt algorithm, 81, 84
- likelihood, 96, 97
- likelihood function, 96
- linear algebraic system, 344
- linear classifier, 39, 370
- linear discriminant analysis, 149, 172
- linear iterative method, 344
- linear separators, 45
- linear SVM, 103, 107, 113
- linearly separable, 42
- link graph, 307, 326
- link matrix, 307, 330
- list, in Python, 21
- Lloyd's algorithm, 211
- local minima problem, 66, 375
- local receptive field, 290
- log-likelihood function, 97
- logarithmic barrier function, 357
- logistic cost function, 97
- logistic curve, 91
- logistic model, 91
- Logistic Regression, 365
- logistic regression, 95, 268
- logistic sigmoid function, 91, 268
- logit, 93
- loss function, 8
- LSI, 319

- M-matrix, 349
- machine learning, 364
- machine learning algorithm, 3
- machine learning challenges, 265
- machine learning modelcode, 12, 374
- Machine_Learning_Model.py, 12
- Manhattan distance, 203
- margin, 42
- Markov matrix, 327
- matrix eigenvalue problem, 306
- matrix factorization, 308
- max-pooling, 294
- maximum likelihood estimator, 96
- mCLESS, 364, 367
- mean square error, 8
- measurement error, 81
- medicine, 95
- medoid, 204
- membership weight, 209
- merit function, 65
- method of Lagrange multipliers, 104, 107

- method of normal equations, 368
- method, in Python class, 31
- min-max scaling, 138
- mini-batch, 273
- mini-batch learning, 52
- Minkowski distance, 130, 203
- missing data, 136
- mixture model, 210
- model, 302
- model reliance, 146
- motor output, 248
- multi-class least-error-square sum, 367
- multi-column least-squares, 368
- multi-line comments, 20
- multiple local minima problem, 66, 101

- negative feedback paths, 251
- neighborhood function, 253
- neighbourhood function, 253
- network.py, 274
- neural network, 264
- neuron, 38
- Newton's method, 26, 61, 67, 356, 377
- newton_horner, in Python, 27
- newton_horner.m, 28
- nlTK, 313
- no free lunch theorem, 90
- nodal point, 346
- nominal features, 137
- non-binding constraint, 111
- nonlinear least-squares problems, 81
- nonlinear SVM, 116
- normal vector, 103
- normalization, 134, 138
- normalization condition, 192
- normalized web matrix, 324
- NP, 207
- NP-Complete, 207
- NP-Hard, 206, 207
- null-space approach, 343, 361
- numerical rank, 169
- numerical underflow, 97
- numpy, 18, 29

- object-oriented programming, 30

- objective function, 48, 56
- observational error, 81
- Occam's razor principle, 10
- odds ratio, 93
- one-hot encoding, 137
- one-shot learning, 265
- one-versus-all, 46
- one-versus-rest, 369
- OOP, 30
- optimization, 55
- optimization problem, 55, 56
- ordinal encoding, 137
- ordinal features, 137
- orthogonal linear transformation, 150
- orthogonally invariant, 311
- outlier problem, K-Means, 220
- outlier removal algorithm, 220
- outliers, 232
- outlinks, 307, 323
- output layer, 269
- OVA, 46
- overfit, 44
- overfitting, 100, 140
- OVR, 46, 369

- P, 207
- P versus NP problem, 207
- Pagerank, 322–324
- pagerank equation, 327
- Pagerank vector, 325
- pagerank vector, 308
- PAM algorithm, 221
- parent class, 34
- partitional clustering, 208
- PCA, 150
- Peppers image, 170
- peppers_SVD.m, 170
- Perceptron, 365
- perceptron, 37, 40, 266
- permutation feature importance, 146, 148
- permutation matrix, 345
- Perron-Frobenius theorem, 326
- personalization vector, 328, 331
- perturbed complementarity, 359
- polar decomposition, 154

- Polynomial_01.py, 31
- Polynomial_02.py, 32
- pooling, 294
- pooling layers, 294
- Porter Stemming Algorithm, 314
- positive definite, 335
- positive semidefinite, 335
- positive semidefinite matrix, 151
- power method, 329
- precision, 317
- prediction methods, 302
- preprocessing, 313
- primal active set strategy, 353
- primal feasibility, 108
- principal component analysis, 149, 150
- principal components, 150, 189
- principal curvatures, 70
- principal directions, 70
- principle of topographic map formation, 248
- probabilistic clustering, 209
- probabilistic model, 95
- projection vector, 173
- proximity matrix, 238
- pseudoinverse, 162, 163, 167
- purity, 244
- Python, 18
- Python essentials, 21
- python_startup.py, 19
- Q9, 312
- QMR, 350
- QR method, 168
- quadratic programming, 115, 333
- quadratic programming, equality constrained, 334
- quadratic programming, general form, 333, 352
- quadratic programming, inequality-constrained, 355, 357
- quasi-Newton method, 377
- quasi-Newton methods, 72
- query matching, 317
- radial basis function, 117
- random decision forests, 127
- random error, 81
- random forest, 127
- random forests, 127
- randomness, 100
- range, in Python, 22
- range-space approach, 341
- rank reduction procedure, 308
- rank-one matrix, 73, 176
- rank-reduction decomposition, 309
- Rayleigh quotient, 151
- reachable points, 232
- recall, 317
- recall versus precision diagram, 318
- rectified linear units, 297
- rectifier, 92
- reduced Hessian, 334
- reduced KKT system, 343
- reducible matrix, 346
- redundant constraint, 111
- reference semantics, in Python, 22
- regression, 302
- regression analysis, 3
- regression example, 7
- regular splitting, 349–351
- regularization, 101, 143, 144
- regularization methods, 141
- regularization strength, 101
- ReLU, 297
- remainder theorem, 26
- representation learning, 262
- retrieving elements, in Python, 21
- ridge regression, 141
- right eigenvalue, 325
- right singular vectors, 153, 156
- risk, 9
- Rosenbrock function, 57
- rotational symmetry, 92
- Run_network.py, 276
- Run_network3.py, 298
- scale-invariant feature transform, 262
- scatter, 174
- Schur complement, 341, 360
- Schur product, 284

- scipy, 18
- score matrix, 151
- search direction, 48
- search engine, 306
- secant condition, 86
- secant equation, 73
- self, 31
- self-organizing map, 246
- self-referencing, 322
- semidefinite, 335
- sequential backward selection, 142
- Sequential Minimal Optimization, 109
- sequential minimal optimization, 120
- sequential pattern discovery, 302
- Sergey Brin, 327
- set of blocking constraints, 354
- SGD, 51, 76
- SGM, 76
- shared bias, 292
- shared weights, 292
- Sherman-Morrison formula, 73
- SIFT, 262
- sigmoid function, 91, 268
- sigmoid neural networks, 268
- sigmoid neuron, 267
- sigmoid neurons, 268
- silhouette coefficient, 241
- silhouette plots, 259
- similarity function, 117
- similarity matrix, 192, 239
- single linkage, 226
- singular value decomposition, 152, 153, 156, 306, 319, 338, 368
- singular values, 153, 156
- sklearn_classifiers.py, 13
- slack variable, 112, 355
- slicing, in Python, 21
- SMO, 109, 120
- soft-margin classification, 112
- SoftmaxLayer, 297
- softplus function, 92
- SOM, 246
- source matrix, 368
- spaCy, 313
- sparse, 316
- sparse eigenvalue algorithms, 329
- sparse model, 144
- sparsity, 144
- SPD, 351
- spectral radius, 345
- spectrum, 345
- Speed up Python Programs, 18
- SSE, 85, 210
- standard logistic sigmoid function, 91
- standard normal distribution, 138
- standardization, 50, 101, 138
- statistical inference, 2
- steepest descent, 58
- steepest descent method, 56
- stemming, 314
- step length, 48, 58, 99
- stochastic gradient descent, 49, 51, 76, 77, 273
- stochastic gradient method, 76
- stochastic matrix, 327
- stochastic SOM, 255
- stop words, 314
- stride length, 291
- string, in Python, 21
- strongly connected, 325, 347
- sufficient decrease condition, 63
- sum of squared error, 85
- sum of squared errors, 48, 210
- sum-of-squares objective function, 83
- sum-squared-error, 96
- summary of SVM, 115
- supervised learning, 6
- support vector machine, 45, 102, 370
- support vectors, 110
- surrogate function, 65
- SVD, 306, 368
- SVD theorem, 156
- SVD, algebraic interpretation, 158
- SVD, geometric interpretation, 160
- SVM, 370
- SVM summary, 115
- SVM, nonlinear, 116
- Sylvester's law of inertia, 338
- symmetric factorization, 339
- symmetric positive definite, 351

- synthetic division, 26
- synthetic_data.py, 372
- systematic error, 81, 100
- Taylor series expansion, 83
- Taylor's Theorem, with integral remainder, 59
- teleportation, 328
- term, 304
- term frequency, 315
- term weighting scheme, 315
- term-document matrix, 304, 305, 313, 315
- terms, 313
- text mining, 312
- textmineR, 313
- the test of time award, 231
- Tikhonov regularization, 141
- topographic neighborhood, 254
- topological neighbourhood, 253
- transformation matrix, 152
- transforming null-space iteration, 350
- transforming range-space iteration, 350, 351
- translation invariance, 292, 296
- truncated data, 153
- truncated score matrix, 154, 189
- truncated transformation, 153
- tuple, in Python, 21
- Turing machine, 207
- two-class classification, 38
- unbiasedness property, 80
- underfitting, 100
- unit round-off error, 169
- unsupervised learning, 11, 200
- upper bidiagonal matrix, 168
- variable selection, 140
- variance, 100
- vector space model, 313
- vectorization, 52
- Voronoi iteration, 211
- Ward's method, 230
- Ward's minimum variance, 227
- weather forecasting, 95
- web search engines, 312
- Wedderburn matrices, 309
- Wedderburn rank reduction theorem, 308
- Wedderburn rank-reduction process, 309
- weight decay, 101
- weight matrix, 367
- William of Occam, 10
- wine, 374
- winner-takes-all neuron, 248, 251
- winning neuron, 248, 251
- within-class scatter, 174, 175
- within-class scatter matrix, 175, 178
- within-class variability, 174
- wrapper methods, 141
- Zeros-Polynomials-Newton-Horner.py, 27
- zeros_of_poly_built_in.py, 27