

Mastering Machine Learning with R

Second Edition

Advanced prediction, algorithms, and learning methods with R 3.x



Packt

www.packt.com

By Cory Lesmeister

Mastering Machine Learning with R

Second Edition

Advanced prediction, algorithms, and learning methods with
R 3.x

Cory Lesmeister



BIRMINGHAM - MUMBAI

Mastering Machine Learning with R

Second Edition

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2015

Second Edition: April 2017

Production reference: 1140417

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78728-747-1

www.packtpub.com

Credits

Author

Cory Lesmeister

Copy Editor

Manisha Sinha

Reviewers

Doug Ortiz

Miroslav Kopecky

Project Coordinator

Nidhi Joshi

Commissioning Editor

Veena Pagare

Proofreader

Safis Editing

Acquisition Editor

Tushar Gupta

Indexer

Mariammal Chettiyar

Content Development Editors

Manthan Raja

Jagruti Babaria

Graphics

Tania Dutta

Technical Editor

Dharmendra Yadav

Production Coordinator

Shraddha Falebhai

About the Author

Cory Lesmeister has over a dozen years of quantitative experience and is currently a Senior Quantitative Manager in the banking industry, responsible for building marketing and regulatory models. Cory spent 16 years at Eli Lilly and Company in sales, market research, Lean Six Sigma, marketing analytics, and new product forecasting. A former U.S. Army active duty and reserve officer, Cory was in Baghdad, Iraq, in 2009 serving as the strategic advisor to the 29,000-person Iraqi Oil Police, where he supplied equipment to help the country secure and protect its oil infrastructure. An aviation aficionado, Cory has a BBA in aviation administration from the University of North Dakota and a commercial helicopter license.

About the Reviewers

Doug Ortiz is an Independent Consultant who has been architecting, developing, and integrating enterprise solutions throughout his whole career. Organizations that leverage his skillset have been able to rediscover and reuse their underutilized data via existing and emerging technologies, such as Microsoft BI Stack, Hadoop, NoSQL databases, SharePoint, Hadoop, and related toolsets and technologies.

He is the founder of Illustris, LLC, and can be reached at dougortiz@illustris.org.

Interesting aspects of his profession are listed here:

- Has experience integrating multiple platforms and products
- Helps organizations gain a deeper understanding and value of their current investments in data and existing resources, turning them into useful sources of information
- Has improved, salvaged, and architected projects by utilizing unique and innovative techniques

His hobbies include yoga and scuba diving.

Miroslav Kopecky has been a passionate JVM enthusiast since the first moment he joined SUN Microsystems in 2002. He truly believes in distributed system design, concurrency and parallel computing. One of Miro's most favorite hobbies is the development of autonomic systems. He is one of the co-authors and main contributors to the open source Java IoT/Robotics framework Robo4J.

Miro is currently working on the online energy trading platform for enmacc.de as a senior software developer.

I would like to thank my family and my wife Tanja for the big support during reviewing this book.

Packt Upsell

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1787287475>

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface	1
Chapter 1: A Process for Success	8
The process	9
Business understanding	10
Identifying the business objective	11
Assessing the situation	12
Determining the analytical goals	12
Producing a project plan	12
Data understanding	13
Data preparation	13
Modeling	14
Evaluation	15
Deployment	15
Algorithm flowchart	16
Summary	21
Chapter 2: Linear Regression - The Blocking and Tackling of Machine Learning	22
Univariate linear regression	23
Business understanding	26
Multivariate linear regression	32
Business understanding	32
Data understanding and preparation	33
Modeling and evaluation	36
Other linear model considerations	50
Qualitative features	50
Interaction terms	52
Summary	54
Chapter 3: Logistic Regression and Discriminant Analysis	55
Classification methods and linear regression	56
Logistic regression	56
Business understanding	57
Data understanding and preparation	58
Modeling and evaluation	63

The logistic regression model	64
Logistic regression with cross-validation	67
Discriminant analysis overview	70
Discriminant analysis application	73
Multivariate Adaptive Regression Splines (MARS)	76
Model selection	82
Summary	85
Chapter 4: Advanced Feature Selection in Linear Models	86
Regularization in a nutshell	87
Ridge regression	88
LASSO	88
Elastic net	89
Business case	89
Business understanding	89
Data understanding and preparation	90
Modeling and evaluation	96
Best subsets	96
Ridge regression	100
LASSO	105
Elastic net	108
Cross-validation with glmnet	111
Model selection	113
Regularization and classification	114
Logistic regression example	114
Summary	117
Chapter 5: More Classification Techniques - K-Nearest Neighbors and Support Vector Machines	118
K-nearest neighbors	119
Support vector machines	120
Business case	124
Business understanding	124
Data understanding and preparation	125
Modeling and evaluation	131
KNN modeling	131
SVM modeling	136
Model selection	139
Feature selection for SVMs	142
Summary	144
Chapter 6: Classification and Regression Trees	145

An overview of the techniques	146
Understanding the regression trees	146
Classification trees	147
Random forest	148
Gradient boosting	149
Business case	150
Modeling and evaluation	150
Regression tree	150
Classification tree	154
Random forest regression	156
Random forest classification	159
Extreme gradient boosting - classification	163
Model selection	168
Feature Selection with random forests	168
Summary	171
Chapter 7: Neural Networks and Deep Learning	172
Introduction to neural networks	172
Deep learning, a not-so-deep overview	177
Deep learning resources and advanced methods	179
Business understanding	181
Data understanding and preparation	182
Modeling and evaluation	187
An example of deep learning	192
H2O background	193
Data upload to H2O	193
Create train and test datasets	195
Modeling	196
Summary	200
Chapter 8: Cluster Analysis	201
Hierarchical clustering	202
Distance calculations	203
K-means clustering	204
Gower and partitioning around medoids	205
Gower	206
PAM	207
Random forest	207
Business understanding	208
Data understanding and preparation	209
Modeling and evaluation	211

Hierarchical clustering	211
K-means clustering	221
Gower and PAM	225
Random Forest and PAM	227
Summary	229
Chapter 9: Principal Components Analysis	230
An overview of the principal components	231
Rotation	234
Business understanding	236
Data understanding and preparation	237
Modeling and evaluation	239
Component extraction	239
Orthogonal rotation and interpretation	240
Creating factor scores from the components	242
Regression analysis	243
Summary	249
Chapter 10: Market Basket Analysis, Recommendation Engines, and Sequential Analysis	250
An overview of a market basket analysis	251
Business understanding	252
Data understanding and preparation	253
Modeling and evaluation	255
An overview of a recommendation engine	259
User-based collaborative filtering	261
Item-based collaborative filtering	261
Singular value decomposition and principal components analysis	262
Business understanding and recommendations	266
Data understanding, preparation, and recommendations	266
Modeling, evaluation, and recommendations	269
Sequential data analysis	279
Sequential analysis applied	280
Summary	287
Chapter 11: Creating Ensembles and Multiclass Classification	288
Ensembles	289
Business and data understanding	290
Modeling evaluation and selection	291
Multiclass classification	294
Business and data understanding	295

Model evaluation and selection	299
Random forest	300
Ridge regression	302
MLR's ensemble	303
Summary	305
Chapter 12: Time Series and Causality	306
Univariate time series analysis	307
Understanding Granger causality	313
Business understanding	314
Data understanding and preparation	316
Modeling and evaluation	320
Univariate time series forecasting	320
Examining the causality	324
Linear regression	324
Vector autoregression	327
Summary	332
Chapter 13: Text Mining	334
Text mining framework and methods	335
Topic models	337
Other quantitative analyses	338
Business understanding	340
Data understanding and preparation	340
Modeling and evaluation	343
Word frequency and topic models	343
Additional quantitative analysis	348
Summary	357
Chapter 14: R on the Cloud	358
Creating an Amazon Web Services account	359
Launch a virtual machine	361
Start RStudio	365
Summary	367
Appendix A: R Fundamentals	368
Getting R up-and-running	368
Using R	374
Data frames and matrices	379
Creating summary statistics	381
Installing and loading R packages	385
Data manipulation with dplyr	386

Summary	389
Appendix B: Sources	390
Index	391

Preface

"A man deserves a second chance, but keep an eye on him"

-John Wayne

It is not so often in life that you get a second chance. I remember that only days after we stopped editing the first edition, I kept asking myself, *"Why didn't I...?"*, or *"What the heck was I thinking saying it like that?"*, and on and on. In fact, the first project I started working on after it was published had nothing to do with any of the methods in the first edition. I made a mental note that if given the chance, it would go into a second edition.

When I started with the first edition, my goal was to create something different, maybe even create a work that was a pleasure to read, given the constraints of the topic. After all the feedback I received, I think I hit the mark. However, there is always room for improvement, and if you try and be everything to all people, you become nothing to everybody. I'm reminded of one of my favorite Frederick the great quotes, *"He who defends everything, defends nothing"*. So, I've tried to provide enough of the skills and tools, but not all of them, to get a reader up and running with R and machine learning as quickly and painlessly as possible. I think I've added some interesting new techniques that build on what was in the first edition. There will probably always be the detractors who complain it does not offer enough math or does not do this, that, or the other thing, but my answer to that is they already exist! Why duplicate what was already done, and very well, for that matter? Again, I have sought to provide something different, something that would keep the reader's attention and allow them to succeed in this competitive field.

Before I provide a list of the changes/improvements incorporated into the second edition, chapter by chapter, let me explain some universal changes. First of all, I have surrendered in my effort to fight the usage of the assignment operator `<-` versus just using `=`. As I shared more and more code with others, I realized I was out on my own using `=` and not `<-`. The first thing I did when under contract for the second edition was go line by line in the code and change it. The more important part, perhaps, was to clean and standardize the code. This is also important when you have to share code with coworkers and, dare I say, regulators. Using RStudio facilitates this standardization in the most recent versions. What sort of standards! Well, the first thing is to properly space the code. For instance, I would not hesitate in the past to write `c(1,2,3,4,5,6)`. Not anymore! Now, I will write this--`c(1, 2, 3, 4, 5, 6)`--as a space after commas, which makes it easier to read. If you want other ideas, please have a look at Google's R style guide, <https://google.github.io/styleguide/Rguide.xml/>.

I also received a number of e-mails saying that the data I scraped off the Web wasn't available. The National Hockey League decided to launch a completely new version of their statistics, so I had to start from scratch. Problems such as that led me to put data on GitHub.

All in all, I put forth a rather large effort to put the best possible tool in your hands to get you going. On another note, in the month of February '17, there was much attention on the Web on these comments from entrepreneur Mark Cuban:

- "Artificial Intelligence, deep learning, machine learning--whatever you're doing if you don't understand it--learn it. Because otherwise you're going to be a dinosaur within 3 years."
- "I personally think there's going to be a greater demand in 10 years for liberal arts majors than there were for programming majors and maybe even engineering, because when the data is all being spit out for you, options are being spit out for you, you need a different perspective in order to have a different view of the data. And so is having someone who is more of a freer thinker."

Besides the fact that these comments created a bit of a stir on the blogosphere, they also seem to be, at first glance, mutually exclusive. But think about what he is saying here. I think he gets to the core of why I felt compelled to write this book. Here is what I believe, machine learning needs to be embraced and utilized, to some extent, by the *masses*: the tired, the poor, the hungry, the proletariat, and the bourgeoisie. More and more availability of computational power and information will make machine learning something for virtually everyone. However, the flip side of that and what, in my mind, has been and will continue to be a problem is the communication of results. What are you going to do when you describe true positive rate and false positive rate and receive blank stares? How do you quickly tell a story that enlightens your audience? If you think it can't happen, please drop me a note, I'd be more than happy to share my story.

We must have people who can lead these efforts and influence their organization. If a degree in history or music appreciation helps in that endeavor, then so be it. I study history every day, and it has helped me tremendously. Cuban's comments have reinforced my belief that in many ways, the first chapter is the most important in this book. If you are not asking your business partners "what they plan to do differently", you'd better start tomorrow. There are far too many people working far too hard to complete an analysis that is completely irrelevant to the organization and its decisions.

What this book covers

Here is a list of changes from the first edition by chapter:

Chapter 1, *A process for success*, has the flowchart redone to update an unintended typo and add additional methodologies.

Chapter 2, *Linear Regression – the Blocking and Tackling of Machine Learning*, has the code improved, and better charts have been provided; other than that, it remains relatively close to the original.

Chapter 3, *Logistic Regression and Discriminant Analysis*, has the code improved and streamlined. One of my favorite techniques, multivariate adaptive regression splines, has been added; it performs well, handles non-linearity, and is easy to explain. It is my base model, with others becoming "challengers" to try and outperform it.

Chapter 4, *Advanced Feature Selection in Linear Models*, has techniques not only for regression but also for a classification problem included.

Chapter 5, *More Classification Techniques – K-Nearest Neighbors and Support Vector Machines*, has the code streamlined and simplified.

Chapter 6, *Classification and Regression Trees*, has the addition of the very popular techniques provided by the XGBOOST package. Additionally, I added the technique of using random forest as a feature selection tool.

Chapter 7, *Neural Networks and Deep Learning*, has been updated with additional information on deep learning methods and has improved code for the H2O package, including hyper-parameter search.

Chapter 8, *Cluster Analysis*, has the methodology of doing unsupervised learning with random forests added.

Chapter 9, *Principal Components Analysis*, uses a different dataset, and an out-of-sample prediction has been added.

Chapter 10, *Market Basket Analysis, Recommendation Engines, and Sequential Analysis*, has the addition of sequential analysis, which, I'm discovering, is more and more important, especially in marketing.

Chapter 11, *Creating Ensembles and Multiclass Classification*, has completely new content, using several great packages.

Chapter 12, *Time Series and Causality*, has a couple of additional years of climate data added, along with a demonstration of different methods of causality test.

Chapter 13, *Text Mining*, has additional data and improved code.

Chapter 14, *R on the Cloud*, is another chapter of new content, allowing you to get R on the cloud, simply and quickly.

Appendix A, *R Fundamentals*, has additional data manipulation methods.

Appendix B, *Sources*, has a list of sources and references.

What you need for this book

As R is free and open source software, you will only need to download and install it from <https://www.r-project.org/>. Although it is not mandatory, it is highly recommended that you download IDE and RStudio from <https://www.rstudio.com/products/RStudio/>.

Who this book is for

This book is for data science professionals, data analysts, or anyone with working knowledge of machine learning with R, who now want to take their skills to the next level and become an expert in the field.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The data frame is available in the R `MASS` package under the `biopsy` name."

Any command-line input or output is written as follows:

```
> bestglm(Xy = biopsy.cv, IC="CV",  
  CVArgs=list(Method="HTF", K=10,  
  REP=1), family=binomial)
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In order to download new modules, we will go to **Files** | **Settings** | **Project Name** | **Project Interpreter**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.

6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Machine-Learning-with-R-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output.

You can download this file from

https://www.packtpub.com/sites/default/files/downloads/MasteringMachineLearningwithRSecondEdition_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

A Process for Success

"If you don't know where you are going, any road will get you there."

- Robert Carrol

"If you can't describe what you are doing as a process, you don't know what you're doing."

- W. Edwards Deming

At first glance, this chapter may seem to have nothing to do with machine learning, but it has everything to do with machine learning (specifically, its implementation and making change happen). The smartest people, best software, and best algorithms do not guarantee success, no matter how well it is defined.

In most, if not all, projects, the key to successfully solving problems or improving decision-making is not the algorithm, but the softer, more qualitative skills of communication and influence. The problem many of us have with this is that it is hard to quantify how effective one is around these skills. It is probably safe to say that many of us ended up in this position because of a desire to avoid it. After all, the highly successful TV comedy *The Big Bang Theory* was built on this premise. Therefore, the goal of this chapter is to set you up for success. The intent is to provide a process, a flexible process no less, where you can become a **change agent**: a person who can influence and turn their insights into action without positional power. We will focus on **Cross-Industry Standard Process for Data Mining (CRISP-DM)**. It is probably the most well-known and respected of all processes for analytical projects. Even if you use another industry process or something proprietary, there should still be a few gems in this chapter that you can take away.

I will not hesitate to say that this all is easier said than done; without question, I'm guilty of every sin (both commission and omission) that will be discussed in this chapter. With skill and some luck, you can avoid the many physical and emotional scars I've picked up over the last 12 years.

Finally, we will also have a look at a flow chart (a cheat sheet) that you can use to help you identify what methodologies to apply to the problem at hand.

The process

The CRISP-DM process was designed specifically for data mining. However, it is flexible and thorough enough to be applied to any analytical project, whether it is predictive analytics, data science, or machine learning. Don't be intimidated by the numerous lists of tasks as you can apply your judgment to the process and adapt it for any real-world situation. The following figure provides a visual representation of the process and shows the feedback loops that make it so flexible:

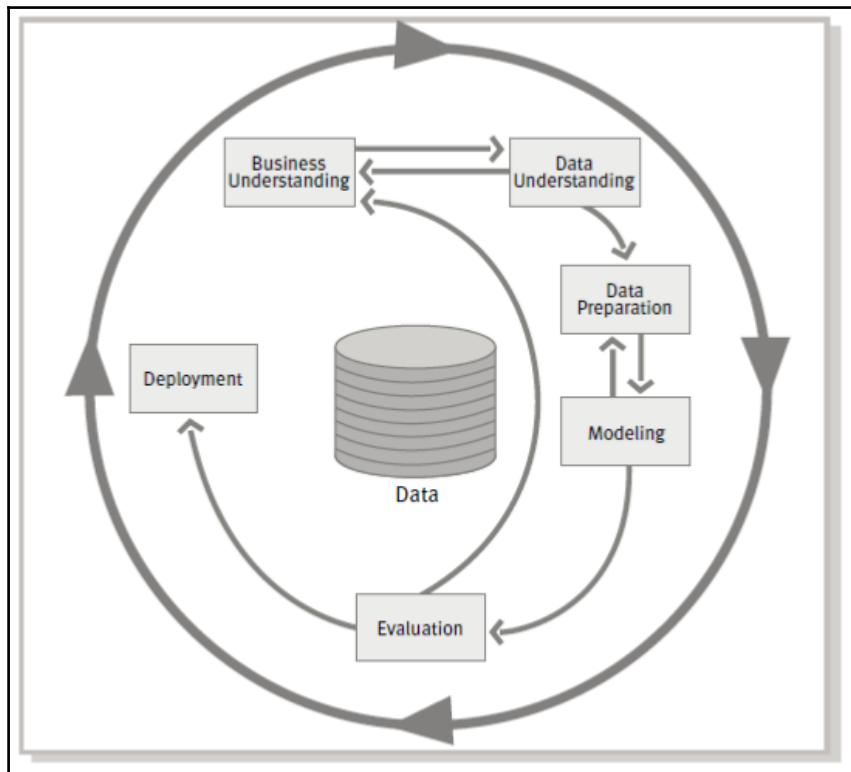


Figure 1: CRISP-DM 1.0, Step-by-step data mining guide

The process has the following six phases:

- Business understanding
- Data understanding
- Data preparation
- Modeling
- Evaluation
- Deployment

For an in-depth review of the entire process with all of its tasks and subtasks, you can examine the paper by SPSS, CRISP-DM 1.0, step-by-step data mining guide, available at <https://the-modeling-agency.com/crisp-dm.pdf>.

I will discuss each of the steps in the process, covering the important tasks. However, it will not be in as detailed as the guide, but more high-level. We will not skip any of the critical details but focus more on the techniques that one can apply to the tasks. Keep in mind that these process steps will be used in later chapters as a framework in the actual application of the machine-learning methods in general and the R code, in particular.

Business understanding

One cannot underestimate how important this first step in the process is in achieving success. It is the foundational step, and failure or success here will likely determine failure or success for the rest of the project. The purpose of this step is to identify the requirements of the business so that you can translate them into analytical objectives. It has the following four tasks:

1. Identifying the business objective.
2. Assessing the situation.
3. Determining analytical goals.
4. Producing a project plan.

Identifying the business objective

The key to this task is to identify the goals of the organization and frame the problem. An effective question to ask is, "What are we going to do different?" This may seem like a benign question, but it can really challenge people to work out what they need from an analytical perspective and it can get to the root of the decision that needs to be made. It can also prevent you from going out and doing a lot of unnecessary work on some kind of "fishing expedition." As such, the key for you is to identify the decision. A working definition of a decision can be put forward to the team as the irrevocable choice to commit or not commit the resources. Additionally, remember that the choice to do nothing different is indeed a decision.

This does not mean that a project should not be launched if the choices are not absolutely clear. There will be times when the problem is not, or cannot be, well defined; to paraphrase former Defense Secretary Donald Rumsfeld, there are known-unknowns. Indeed, there will probably be many times when the problem is ill defined and the project's main goal is to further the understanding of the problem and generate hypotheses; again calling on Secretary Rumsfeld, unknown-unknowns, which means that you don't know what you don't know. However, with ill-defined problems, one could go forward with an understanding of what will happen next in terms of resource commitment based on the various outcomes from hypothesis exploration.

Another thing to consider in this task is the management of expectations. There is no such thing as perfect data, no matter what its depth and breadth are. This is not the time to make guarantees but to communicate what is possible, given your expertise.

I recommend a couple of outputs from this task. The first is a mission statement. This is not the touchy-feely mission statement of an organization, but it is your mission statement or, more importantly, the mission statement approved by the project sponsor. I stole this idea from my years of military experience and I could write volumes on why it is effective, but that is for another day. Let's just say that, in the absence of clear direction or guidance, the mission statement, or whatever you want to call it, becomes the unifying statement for all stakeholders and can help prevent scope creep. It consists of the following points:

- **Who:** This is yourself or the team or project name; everyone likes a cool project name, for example, Project Viper, Project Fusion, and so on
- **What:** This is the task that you will perform, for example, conducting machine learning
- **When:** This is the deadline
- **Where:** This could be geographical, by function, department, initiative, and so on
- **Why:** This is the purpose behind implementing the project, that is, the business goal

The second task is to have as clear a definition of success as possible. Literally, ask "What does success look like?" Help the team/sponsor paint a picture of success that you can understand. Your job then is to translate this into modeling requirements.

Assessing the situation

This task helps you in project planning by gathering information on the resources available, constraints, and assumptions; identifying the risks; and building contingency plans. I would further add that this is also the time to identify the key stakeholders that will be impacted by the decision(s) to be made.

A couple of points here. When examining the resources that are available, do not neglect to scour the records of past and current projects. Odds are someone in the organization has worked, or is working on the same problem and it may be essential to synchronize your work with theirs. Don't forget to enumerate the risks considering **time**, **people**, and **money**. Do everything in your power to create a list of stakeholders, both those that impact your project and those that could be impacted by your project. Identify who these people are and how they can influence/be impacted by the decision. Once this is done, work with the project sponsor to formulate a communication plan with these stakeholders.

Determining the analytical goals

Here, you are looking to translate the business goal into technical requirements. This includes turning the success criterion from the task of creating a business objective to technical success. This might be things such as RMSE or a level of predictive accuracy.

Producing a project plan

The task here is to build an effective project plan with all the information gathered up to this point. Regardless of what technique you use, whether it be a Gantt chart or some other graphic, produce it and make it a part of your communication plan. Make this plan widely available to the stakeholders and update it on a regular basis and as circumstances dictate.

Data understanding

After enduring the all-important pain of the first step, you can now get busy with the data. The tasks in this process consist of the following:

1. Collecting the data.
2. Describing the data.
3. Exploring the data.
4. Verifying the data quality.

This step is the classic case of **Extract, Transform, Load (ETL)**. There are some considerations here. You need to make an initial determination that the data available is adequate to meet your analytical needs. As you explore the data, visually and otherwise, determine whether the variables are sparse and identify the extent to which data may be missing. This may drive the learning method that you use and/or determine whether the imputation of the missing data is necessary and feasible.

Verifying the data quality is critical. Take the time to understand who collects the data, how it is collected, and even why it is collected. It is likely that you may stumble upon incomplete data collection, cases where unintended IT issues led to errors in the data, or planned changes in the business rules. This is critical in time series where often business rules on how the data is classified change over time. Finally, it is a good idea to begin documenting any code at this step. As a part of the documentation process, if a data dictionary is not available, save yourself potential heartache and make one.

Data preparation

Almost there! This step has the following five tasks:

1. Selecting the data.
2. Cleaning the data.
3. Constructing the data.
4. Integrating the data.
5. Formatting the data.

These tasks are relatively self-explanatory. The goal is to get the data ready to input in the algorithms. This includes merging, feature engineering, and transformations. If imputation is needed, then it happens here as well. Additionally, with R, pay attention to how the outcome needs to be labeled. If your outcome/response variable is Yes/No, it may not work in some packages and will require a transformed or no variable with 1/0. At this point, you should also break your data into the various test sets if applicable: train, test, or validate. This step can be an unmitigated burden, but most experienced people will tell you that it is where you can separate yourself from your peers. With this, let's move on to the payoff, where you earn your money.

Modeling

This is where all the work that you've done up to this point can lead to fist-pumping exuberance or fist-pounding exasperation. But hey, if it was that easy, everyone would be doing it. The tasks are as follows:

1. Selecting a modeling technique.
2. Generating a test design.
3. Building a model.
4. Assessing a model.

Oddly, this process step includes the considerations that you have already thought of and prepared for. In the first step, you will need at least some idea about how you will be modeling. Remember that this is a flexible, iterative process and not some strict linear flowchart such as an aircrew checklist.

The cheat sheet included in this chapter should help guide you in the right direction for the modeling techniques. Test design refers to the creation of your test and train datasets and/or the use of cross-validation and this should have been thought of and accounted for in the data preparation.

Model assessment involves comparing the models with the criteria/criterion that you developed in the business understanding, for example, RMSE, Lift, ROC, and so on.

Evaluation

With the evaluation process, the main goal is to confirm that the model selected at this point meets the business objective. Ask yourself and others, "Have we achieved our definition of success?". Let the Netflix prize serve as a cautionary tale here. I'm sure you are aware that Netflix awarded a \$1-million prize to the team that could produce the best recommendation algorithm as defined by the lowest RMSE. However, Netflix did not implement it because the incremental accuracy gained was not worth the engineering effort! Always apply Occam's razor. At any rate, here are the tasks:

1. Evaluating the results.
2. Reviewing the process.
3. Determining the next steps.

In reviewing the process, it may be necessary, as you no doubt determined earlier in the process, to take the results through governance and communicate with the other stakeholders in order to gain their buy-in. As for the next steps, if you want to be a change agent, make sure that you answer the **what**, **so what**, and **now what** in the stakeholders' minds. If you can tie their **now what** into the decision that you made earlier, you have earned your money.

Deployment

If everything is done according to the plan up to this point, it might just come down to flipping a switch and your model goes live. Assuming that this is not the case, here are the tasks for this step:

1. Deploying the plan.
2. Monitoring and maintaining the plan.
3. Producing the final report.
4. Reviewing the project.

After the deployment and monitoring/maintenance and underway, it is crucial for you and those who will walk in your steps to produce a well-written final report. This report should include a white paper and briefing slide. I have to say that I resisted the drive to put my findings in a white paper as I was an indentured servant to the military's passion for PowerPoint slides. However, slides can and will be used against you, cherry-picked or misrepresented by various parties for their benefit. Trust me, that just doesn't happen with a white paper as it becomes an extension of your findings and beliefs. Use PowerPoint to brief stakeholders, but use that the white paper as the document of record and as a preread, should your organization insist on one. It is my standard procedure to create this white paper in R using knitr and LaTeX.

Now for the all-important process review, you may have your own proprietary way of conducting it; but here is what it should cover, whether you conduct it in a formal or informal way:

- What was the plan?
- What actually happened?
- Why did it happen or not happen?
- What should be sustained in future projects?
- What should be improved upon in future projects?
- Create an action plan to ensure sustainment and improvement happen

That concludes the review of the CRISP-DM process, which provides a comprehensive and flexible framework to guarantee the success of your project and make you an agent of change.

Algorithm flowchart

The purpose of this section is to create a tool that will help you not just select possible modeling techniques but also think deeper about the problem. The residual benefit is that it may help you frame the problem with the project sponsor/team. The techniques in the flowchart are certainly not comprehensive but are exhaustive enough to get you started. It also includes techniques not discussed in this book.

The following figure starts the flow of selecting the potential modeling techniques. As you answer the question(s), it will take you to one of the four additional charts:

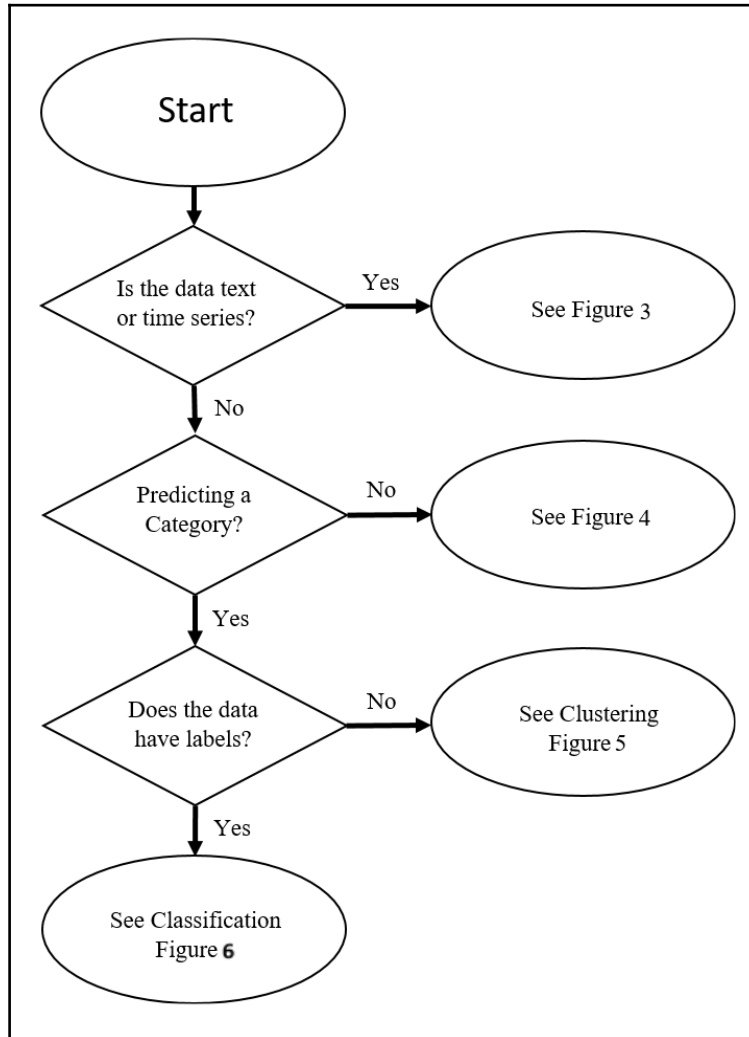


Figure 2

If the data is text or in the time series format, then you will follow the flow in the following figure:

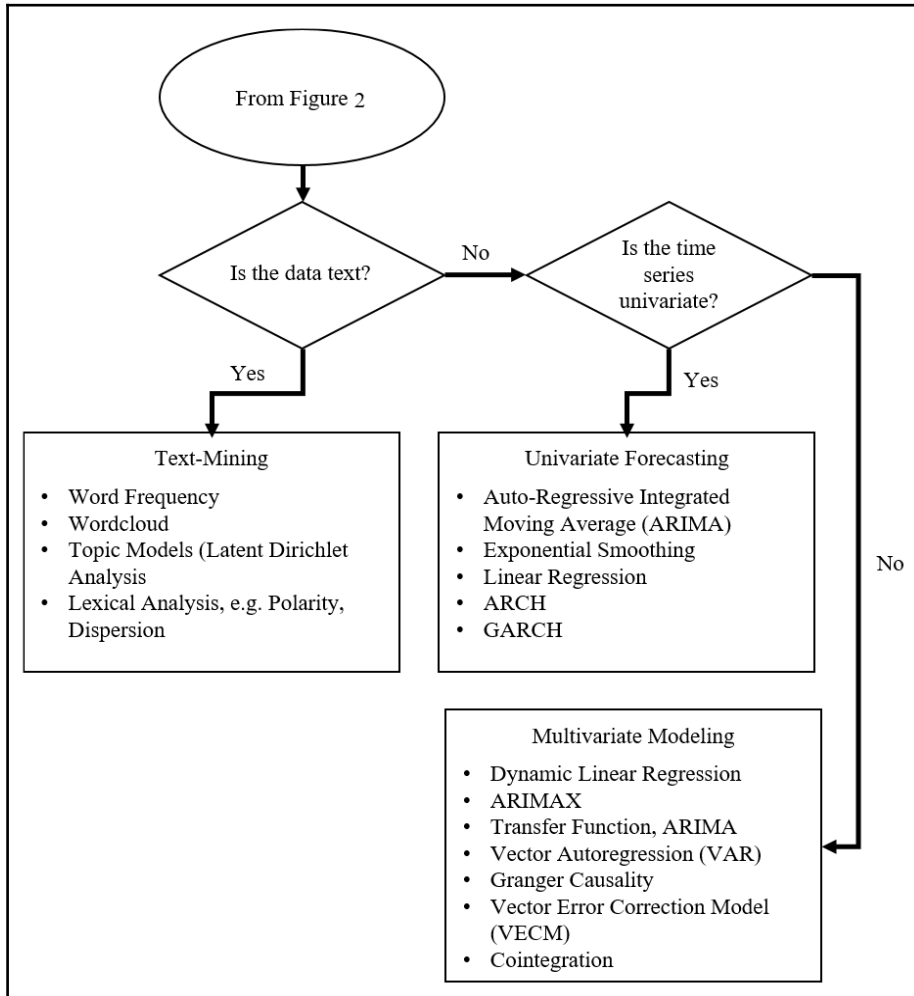


Figure 3

In this branch of the algorithm, you do not have text or time series data. You also do not want to predict a category, so you are looking to make recommendations, understand associations, or predict a quantity:

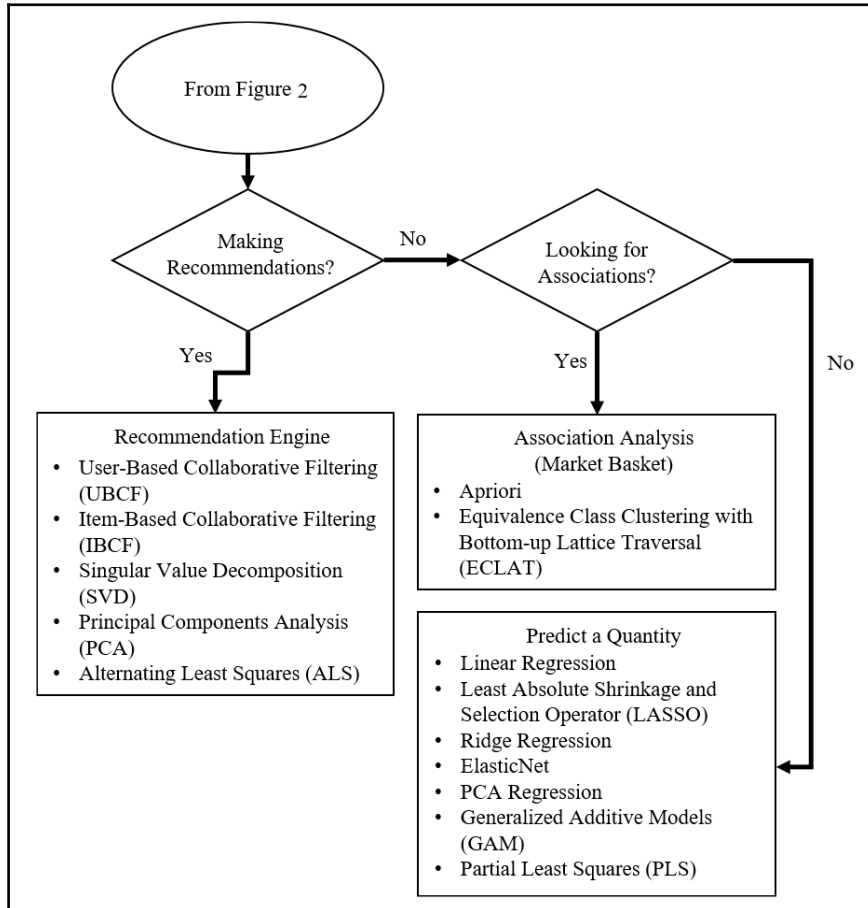


Figure 4

To get to this section, you will have data that is not text or time series. You want to categorize the data, but it does not have an outcome label, which brings us to clustering methods, as follows:

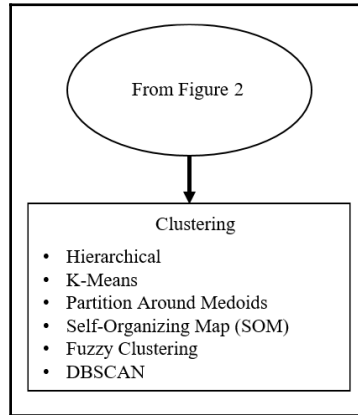


Figure 5

This brings us to the situation where we want to categorize the data and it is labeled, that is, classification:

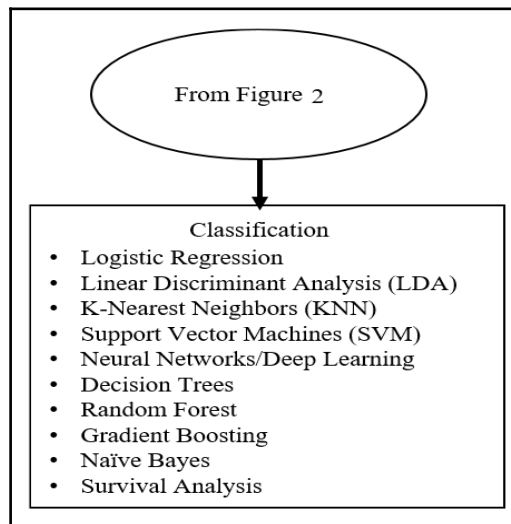


Figure 6

Summary

This chapter was about how to set up you and your team for success in any project that you tackle. The CRISP-DM process is put forward as a flexible and comprehensive framework in order to facilitate the softer skills of communication and influence. Each step of the process and the tasks in each step were enumerated. More than that, the commentary provides some techniques and considerations to with the process execution. By taking heed of the process, you can indeed become an agent of positive change to any organization.

The other item put forth in this chapter was an algorithm flowchart; a cheat sheet to help identify some of the proper techniques to apply in order to solve the business problem. With this foundation in place, we can now move on to applying these techniques to real-world problems.

2

Linear Regression - The Blocking and Tackling of Machine Learning

"Some people try to find things in this game that don't exist, but football is only two things - blocking and tackling."

- Vince Lombardi, Hall of Fame Football Coach

It is important that we get started with a simple, yet extremely effective technique that has been used for a long time: **linear regression**. Albert Einstein is believed to have remarked at one time or another that things should be made as simple as possible, but no simpler. This is sage advice and a good rule of thumb in the development of algorithms for machine learning. Considering the other techniques that we will discuss later, there is no simpler model than tried and tested linear regression, which uses the **least squares approach** to predict a quantitative outcome. In fact, one can consider it to be the foundation of all the methods that we will discuss later, many of which are mere extensions. If you can master the linear regression method, well, then quite frankly, I believe you can master the rest of this book. Therefore, let us consider this a good starting point for our journey towards becoming a machine learning guru.

This chapter covers introductory material, and an expert in this subject can skip ahead to the next topic. Otherwise, ensure that you thoroughly understand this topic before venturing to other, more complex learning methods. I believe you will discover that many of your projects can be addressed by just applying what is discussed in the following section. Linear regression is probably the easiest model to explain to your customers, most of whom will have at least a cursory understanding of **R-squared**. Many of them will have been exposed to it at great depth and thus be comfortable with variable contribution, collinearity, and the like.

Univariate linear regression

We begin by looking at a simple way to predict a quantitative response, Y , with one predictor variable, x , assuming that Y has a linear relationship with x . The model for this can be written as, $Y = B0 + B1x + e$. We can state it as the expected value of Y being a function of the parameters $B0$ (the intercept) plus $B1$ (the slope) times x , plus an error term e . The least squares approach chooses the model parameters that minimize the **Residual Sum of Squares (RSS)** of the predicted y values versus the actual Y values. For a simple example, let's say we have the actual values of $Y1$ and $Y2$ equal to 10 and 20 respectively, along with the predictions of $y1$ and $y2$ as 12 and 18. To calculate RSS, we add the squared differences $RSS = (Y1 - y1)^2 + (Y2 - y2)^2$, which, with simple substitution, yields $(10 - 12)^2 + (20 - 18)^2 = 8$.

I once remarked to a peer during our Lean Six Sigma Black Belt training that it's all about the sum of squares; understand the sum of squares and the rest will flow naturally. Perhaps that is true, at least to some extent.

Before we begin with an application, I want to point out that, if you read the headlines of various research breakthroughs, you should do so with a jaundiced eye and a skeptical mind as the conclusion put forth by the media may not be valid. As we shall see, R, and any other software for that matter, will give us a solution regardless of the inputs. However, just because the math makes sense and a high correlation or R-squared statistic is reported doesn't mean that the conclusion is valid.

To drive this point home, let's have a look at the famous `Anscombe` dataset, which is available in R. The statistician Francis Anscombe produced this set to highlight the importance of data visualization and outliers when analyzing data. It consists of four pairs of X and Y variables that have the same statistical properties but when plotted show something very different. I have used the data to train colleagues and to educate business partners on the hazards of fixating on statistics without exploring the data and checking assumptions. I think this is a good place to start should you have a similar need. It is a brief digression before moving on to serious modeling:

```
> #call up and explore the data

> data(anscombe)

> attach(anscombe)

> anscombe
  x1 x2 x3 x4  y1  y2  y3  y4
1 10 10 10  8  8.04 9.14  7.46  6.58
2   8   8   8  8  6.95 8.14  6.77  5.76
3 13 13 13  8  7.58 8.74 12.74  7.71
```

4	9	9	9	8	8.81	8.77	7.11	8.84
5	11	11	11	8	8.33	9.26	7.81	8.47
6	14	14	14	8	9.96	8.10	8.84	7.04
7	6	6	6	8	7.24	6.13	6.08	5.25
8	4	4	4	19	4.26	3.10	5.39	12.50
9	12	12	12	8	10.84	9.13	8.15	5.56
10	7	7	7	8	4.82	7.26	6.42	7.91
11	5	5	5	8	5.68	4.74	5.73	6.89

As we shall see, each of the pairs has the same correlation coefficient: 0.816. The first two are as follows:

```
> cor(x1, y1) #correlation of x1 and y1
[1] 0.8164205

> cor(x2, y1) #correlation of x2 and y2
[1] 0.8164205
```

The real insight here, as Anscombe intended, is when we plot all the four pairs together, as follows:

```
> par(mfrow = c(2,2)) #create a 2x2 grid for
  plotting

> plot(x1, y1, main = "Plot 1")

> plot(x2, y2, main = "Plot 2")

> plot(x3, y3, main = "Plot 3")

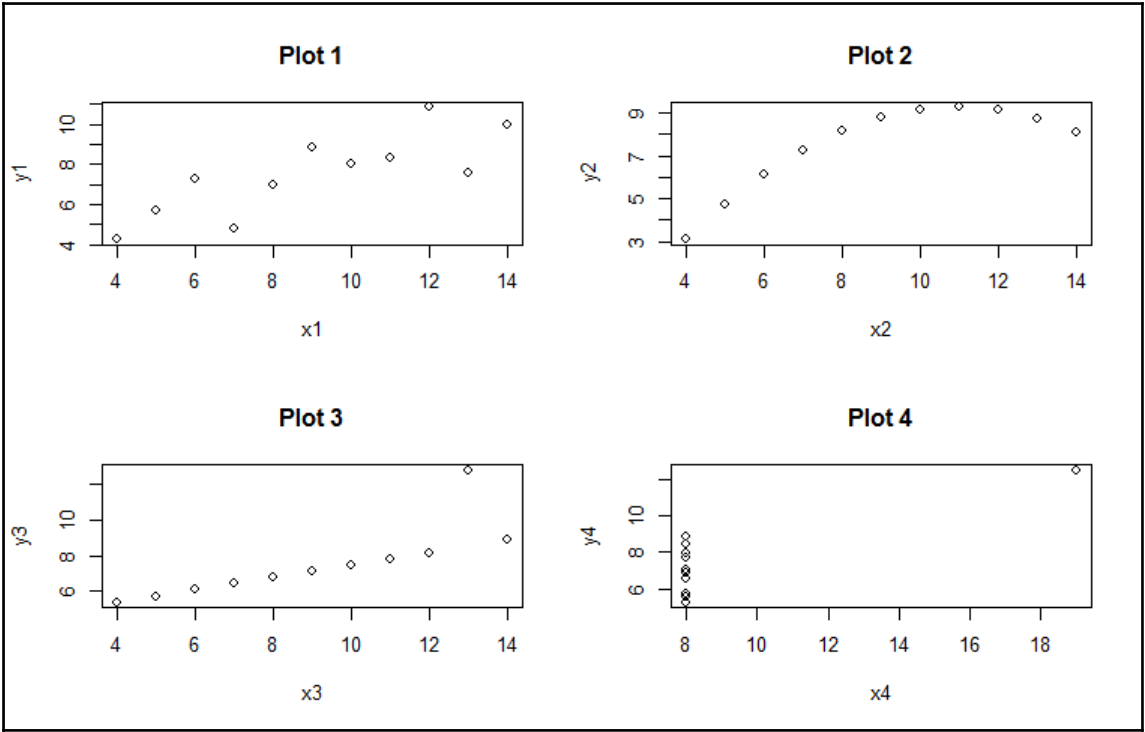
> plot(x4, y4, main = "Plot 4")
```



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

The output of the preceding code is as follows:



As we can see, **Plot 1** appears to have a true linear relationship, **Plot 2** is curvilinear, **Plot 3** has a dangerous outlier, and **Plot 4** is driven by one outlier. There you have it, a cautionary tale about the dangers of solely relying on correlation.

Business understanding

Our first case focuses on the goal of predicting the water yield (in inches) of the Snake River Watershed in Wyoming, USA, as a function of the water content of the year's snowfall. This forecast will be useful in managing the water flow and reservoir levels as the Snake River provides much-needed irrigation water for the farms and ranches of several western states. The `snake` dataset is available in the `alr3` package (note that `alr` stands for applied linear regression):

```
> install.packages("alr3")
> library(alr3)
> data(snake)
> dim(snake)
[1] 17  2
> head(snake)
      X      Y
1 23.1 10.5
2 32.8 16.7
3 31.8 18.2
4 32.0 17.0
5 30.4 16.3
6 24.0 10.5
```

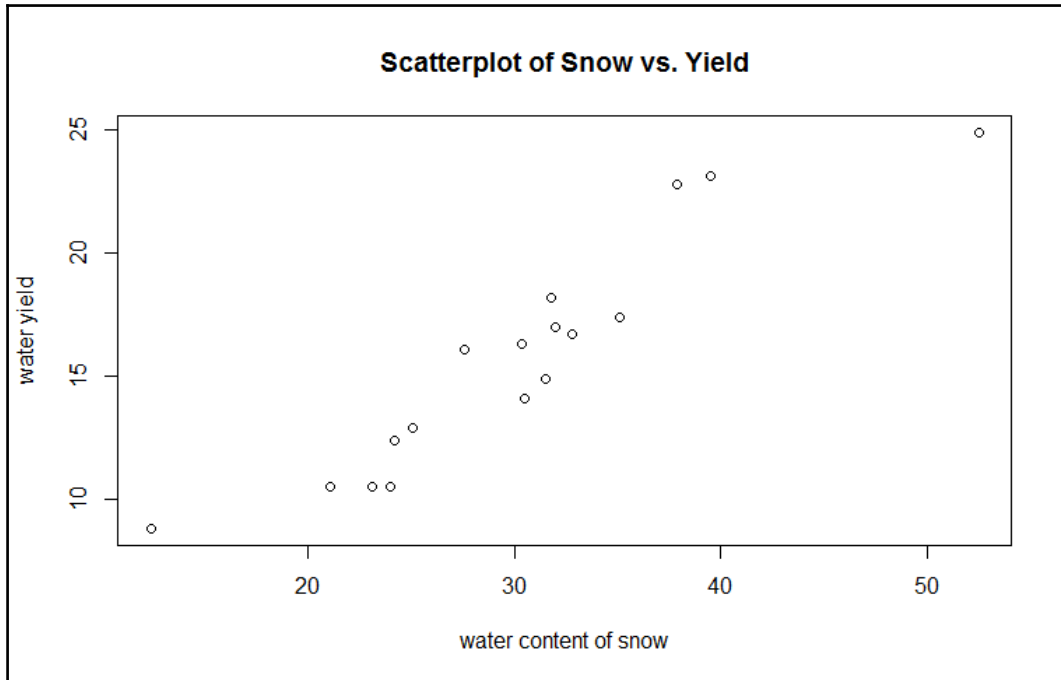
Now that we have 17 observations, data exploration can begin. But first, let's change `x` and `y` to meaningful variable names, as follows:

```
> names(snake) <- c("content", "yield")
> attach(snake) # attach data with new names
> head(snake)

  content yield
1   23.1  10.5
2   32.8  16.7
3   31.8  18.2
4   32.0  17.0
5   30.4  16.3
6   24.0  10.5

> plot(content, yield, xlab = "water content of
      snow", ylab = "water yield")
```


The output of the preceding code is as follows:



This is an interesting plot as the data is linear and has a slight curvilinear shape driven by two potential outliers at both ends of the extreme. As a result, transforming the data or deleting an outlying observation may be warranted.

To perform a linear regression in R, one uses the `lm()` function to create a model in the standard form of $fit = lm(Y \sim X)$. You can then test your assumptions using various functions on your fitted model by using the following code:

```
> yield.fit <- lm(yield ~ content)

> summary(yield.fit)

Call:
lm(formula = yield ~ content)

Residuals:
    Min       1Q   Median       3Q      Max
-2.1793 -1.5149 -0.3624  1.6276  3.1973

Coefficients: Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept)  0.72538    1.54882    0.468    0.646
content      0.49808    0.04952   10.058  4.63e-08
***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05
                 '.' 0.1 ' ' 1

Residual standard error: 1.743 on 15 degrees of
freedom
Multiple R-squared:  0.8709,    Adjusted R-squared:
0.8623
F-statistic: 101.2 on 1 and 15 DF,  p-value:
4.632e-08
```

With the `summary()` function, we can examine a number of items including the model specification, descriptive statistics about the residuals, the coefficients, codes to model significance, and a summary on model error and fit. Right now, let's focus on the parameter coefficient estimates, see if our predictor variable has a significant p-value, and if the overall model F-test has a significant p-value. Looking at the parameter estimates, the model tells us that the `yield` is equal to 0.72538 plus 0.49808 times the `content`. It can be stated that, for every 1 unit change in the `content`, the `yield` will increase by 0.49808 units. The `F-statistic` is used to test the null hypothesis that the model coefficients are all 0.

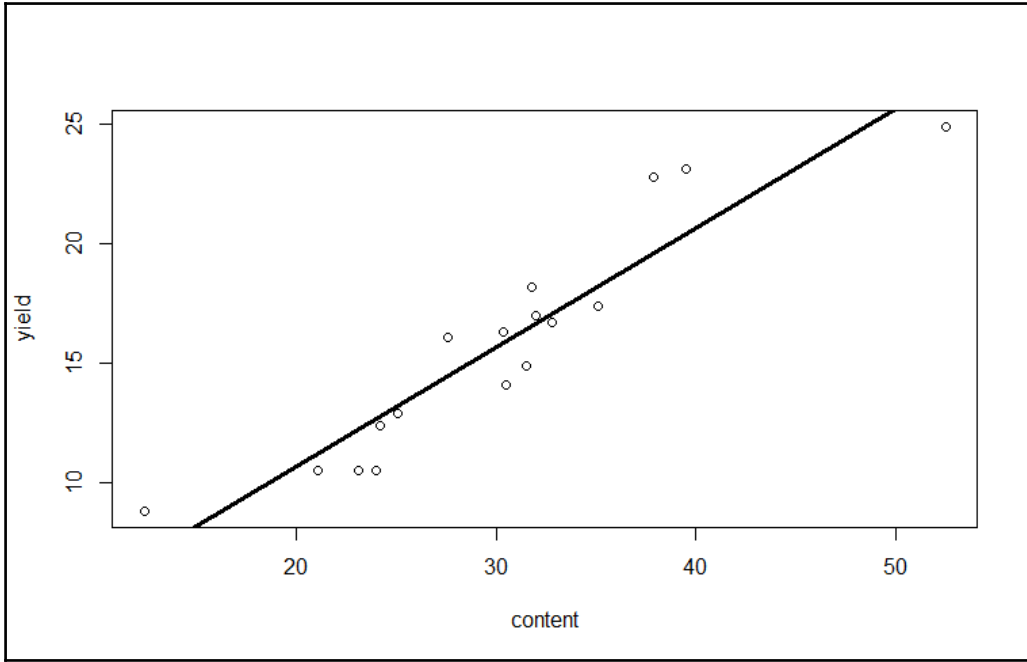
Since the p-value is highly significant, we can reject the null and move on to the t-test for `content`, which tests the null hypothesis that it is 0. Again, we can reject the null. Additionally, we can see `Multiple R-squared` and `Adjusted R-squared` values. `Adjusted R-squared` will be covered under the multivariate regression topic, so let's zero in on `Multiple R-squared`; here we see that it is 0.8709. In theory, it can range from 0 to 1 and is a measure of the strength of the association between `X` and `Y`. The interpretation in this case is that 87 percent of the variation in the **water yield** can be explained by the **water content of snow**. On a side note, `R-squared` is nothing more than the correlation coefficient of `[X, Y]` squared.

We can recall our scatterplot and now add the best fit line produced by our model using the following code:

```
> plot(content, yield)

> abline(yield.fit, lwd=3, col="red")
```

The output of the preceding code is as follows:



A linear regression model is only as good as the validity of its assumptions, which can be summarized as follows:

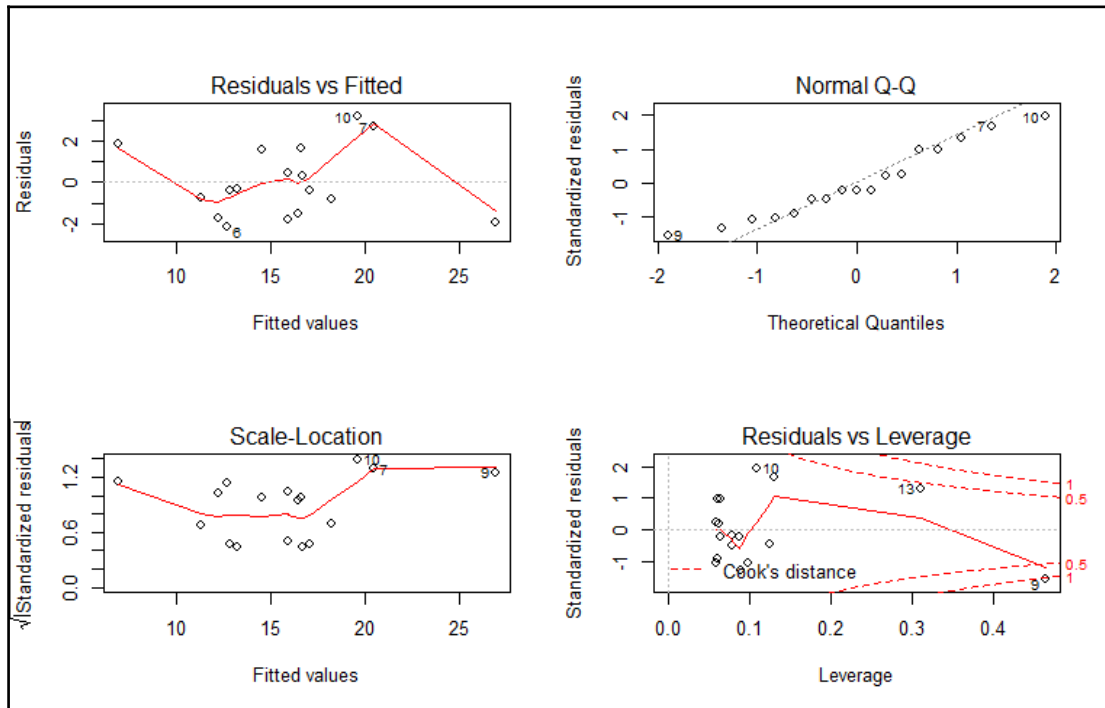
- **Linearity:** This is a linear relationship between the predictor and the response variables. If this relationship is not clearly present, transformations (log, polynomial, exponent, and so on) of X or Y may solve the problem.
- **Non-correlation of errors:** A common problem in time series and panel data where $e_n = \beta_{n-1}$; if the errors are correlated, you run the risk of creating a poorly specified model.
- **Homoscedasticity:** Normally the distributed and constant variance of errors, which means that the variance of errors is constant across different values of inputs. Violations of this assumption can create biased coefficient estimates, leading to statistical tests for significance that can be either too high or too low. This, in turn, leads to a wrong conclusion. This violation is referred to as **heteroscedasticity**.

- **No collinearity:** No linear relationship between two predictor variables, which is to say that there should be no correlation between the features. This, again, can lead to biased estimates.
- **Presence of outliers:** Outliers can severely skew the estimation, and ideally they must be removed prior to fitting a model using linear regression; As we saw in the Anscombe example, this can lead to a biased estimate.

As we are building a univariate model independent of time, we will concern ourselves only with linearity and heteroscedasticity. The other assumptions will become important in the next section. The best way to initially check the assumptions is by producing plots. The `plot()` function, when combined with a linear model fit, will automatically produce four plots allowing you to examine the assumptions. R produces the plots one at a time and you advance through them by hitting the *Enter* key. It is best to examine all four simultaneously and we do it in the following manner:

```
> par(mfrow = c(2,2))  
  
> plot(yield.fit)
```

The output of the preceding code is as follows:



The two plots on the left allow us to examine the homoscedasticity of errors and non-linearity. What we are looking for is some type of pattern or, more importantly, that no pattern exists. Given the sample size of only 17 observations, nothing obvious can be seen. Common heteroscedastic errors will appear to be u-shaped, inverted u-shaped, or clustered close together on the left of the plot. They will become wider as the fitted values increase (a funnel shape). It is safe to conclude that no violation of homoscedasticity is apparent in our model.

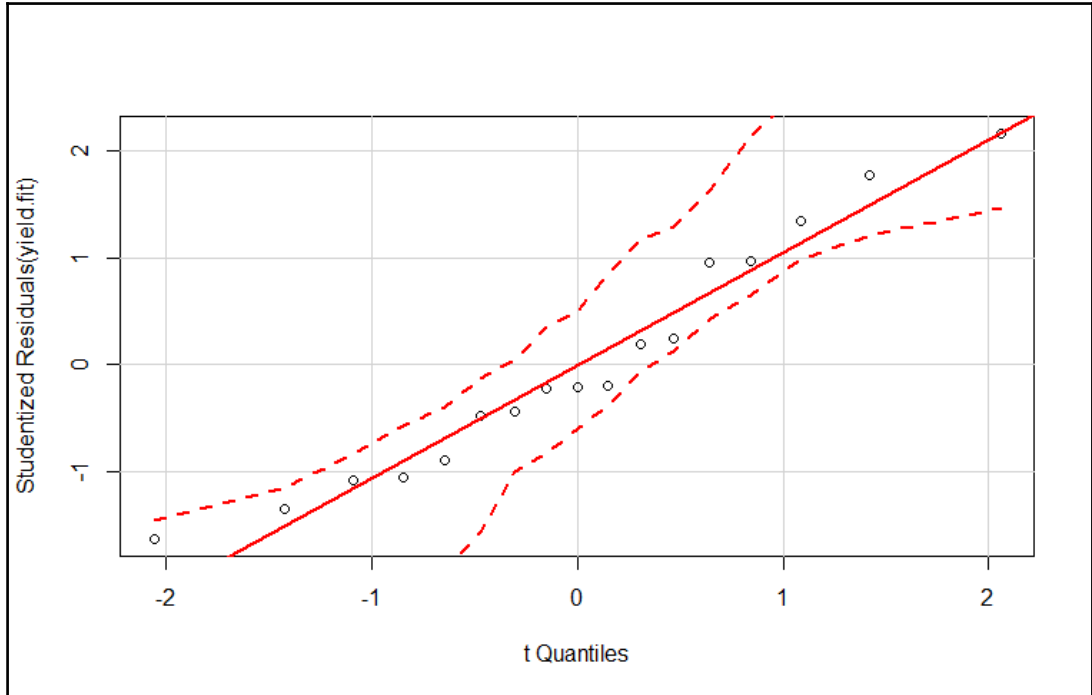
The **Normal Q-Q** plot in the upper-right corner helps us to determine if the residuals are normally distributed. The **Quantile-Quantile (Q-Q)** represents the quantile values of one variable plotted against the quantile values of another. It appears that the outliers (observations 7, 9, and 10), may be causing a violation of the assumption. The **Residuals vs Leverage** plot can tell us what observations, if any, are unduly influencing the model; in other words, if there are any outliers we should be concerned about. The statistic is **Cook's distance** or **Cook's D**, and it is generally accepted that a value greater than 1 should be worthy of further inspection.

What exactly is further inspection? This is where art meets science. The easy way out would be to simply delete the observation, in this case number 9, and redo the model. However, a better option may be to transform the predictor and/or the response variables. If we just delete observation 9, then maybe observations 10 and 13 would fall outside the band for greater than 1. I believe that this is where domain expertise can be critical. More times than I can count, I have found that exploring and understanding outliers can yield valuable insights. When we first examined the previous scatterplot, I pointed out the potential outliers and these happen to be observations number 9 and number 13. As an analyst, it would be critical to discuss with the appropriate subject matter experts to understand why this is the case. Is it a measurement error? Is there a logical explanation for these observations? I certainly don't know, but this is an opportunity to increase the value that you bring to an organization.

Having said that, we can drill down on the current model by examining, in more detail, the **Normal Q-Q** plot. R does not provide confidence intervals to the default Q-Q plot, and given our concerns in looking at the base plot, we should check the confidence intervals. The `qqPlot()` function of the `car` package automatically provides these confidence intervals. Since the `car` package is loaded along with the `alr3` package, I can produce the plot with one line of code:

```
> qqPlot(yield.fit)
```

The output of the preceding code is as follows:



According to the plot, the residuals are normally distributed. I think this can give us the confidence to select the model with all the observations. A clear rationale and judgment would be needed to attempt other models. If we could clearly reject the assumption of normally distributed errors, then we would probably have to examine the variable transformations and/or observation deletion.

Multivariate linear regression

You may be asking yourself whether you will ever have just one predictor variable in the real world. That is indeed a fair question and certainly a very rare case (time series can be a common exception). Most likely, several, if not many, predictor variables or features--as they are affectionately termed in machine learning--will have to be included in your model. And with that, let's move on to multivariate linear regression and a new business case.

Business understanding

In keeping with the water conservation/prediction theme, let's look at another dataset in the `alr3` package, appropriately named `water`. During the writing of the first edition of this book, the severe drought in Southern California caused much alarm. Even the Governor, Jerry Brown, began to take action with a call to citizens to reduce water usage by 20 percent. For this exercise, let's say we have been commissioned by the state of California to predict water availability. The data provided to us contains 43 years of snow precipitation, measured at six different sites in the Owens Valley. It also contains a response variable for water availability as the stream runoff volume near Bishop, California, which feeds into the Owens Valley aqueduct, and eventually the Los Angeles aqueduct. Accurate predictions of the stream runoff will allow engineers, planners, and policy makers to plan conservation measures more effectively. The model we are looking to create will consist of the form $Y = B_0 + B_1x_1 + \dots B_nx_n + e$, where the predictor variables (features) can be from 1 to n .

Data understanding and preparation

To begin, we will load the dataset named `water` and define the structure of the `str()` function as follows:

```
> data(water)

> str(water)
'data.frame':  43 obs. of  8 variables:
 $ Year   : int  1948 1949 1950 1951 1952 1953 1954
             1955 1956 1957 ...
 $ APMAM   : num  9.13 5.28 4.2 4.6 7.15 9.7 5.02 6.7
             10.5 9.1 ...
 $ APSAB   : num  3.58 4.82 3.77 4.46 4.99 5.65 1.45
             7.44 5.85 6.13 ...
 $ APSLAKE: num  3.91 5.2 3.67 3.93 4.88 4.91 1.77
             6.51 3.38 4.08 ...
 $ OPBPC   : num  4.1 7.55 9.52 11.14 16.34 ...
 $ OPRC    : num  7.43 11.11 12.2 15.15 20.05 ...
 $ OPSLAKE: num  6.47 10.26 11.35 11.13 22.81 ...
 $ BSAAM   : int  54235 67567 66161 68094 107080
             67594 65356 67909 92715 70024 ...
```

Here we have eight features and one response variable, BSAAM. The observations start in 1943 and run for 43 consecutive years. Since for this exercise we are not concerned with what year the observations occurred in, it makes sense to create a new data frame excluding the year vector. This is quite easy to do. With one line of code, we can create the new data frame, and then verify that it works with the `head()` function:

```
> socal.water <- water[, -1] #new dataframe with
  the deletion of
  column 1

> head(socal.water)
  APMAM APSAB APSLAKE OPBPC  OPRC OPSLAKE  BSAAM
1  9.13  3.58   3.91  4.10  7.43   6.47  54235
2  5.28  4.82   5.20  7.55 11.11  10.26  67567
3  4.20  3.77   3.67  9.52 12.20  11.35  66161
4  4.60  4.46   3.93 11.14 15.15  11.13  68094
5  7.15  4.99   4.88 16.34 20.05  22.81 107080
6  9.70  5.65   4.91  8.88  8.15   7.41  67594
```

With all the features being quantitative, it makes sense to look at the correlation statistics and then produce a matrix of scatterplots. The correlation coefficient or **Pearson's r** , is a measure of both the strength and direction of the linear relationship between two variables. The statistic will be a number between -1 and 1, where -1 is the total negative correlation and +1 is the total positive correlation. The calculation of the coefficient is the covariance of the two variables divided by the product of their standard deviations. As previously discussed, if you square the correlation coefficient, you will end up with R-squared.

There are a number of ways to produce a matrix of correlation plots. Some prefer to produce **heatmaps**, but I am a big fan of what is produced with the `corrplot` package. It can produce a number of different variations including ellipse, circle, square, number, shade, color, and pie. I like the `ellipse` method, but feel free to experiment with the others. Let's load the `corrplot` package, create a correlation object using the base `cor()` function, and examine the following results:

```
> library(corrplot)

> water.cor <- cor(socal.water)

> water.cor
  APMAM      APSAB      APSLAKE      OPBPC
APMAM  1.0000000  0.82768637  0.81607595  0.12238567
APSAB  0.8276864  1.00000000  0.90030474  0.03954211
APSLAKE 0.8160760  0.90030474  1.00000000  0.09344773
OPBPC  0.1223857  0.03954211  0.09344773  1.00000000
OPRC   0.1544155  0.10563959  0.10638359  0.86470733
```



```

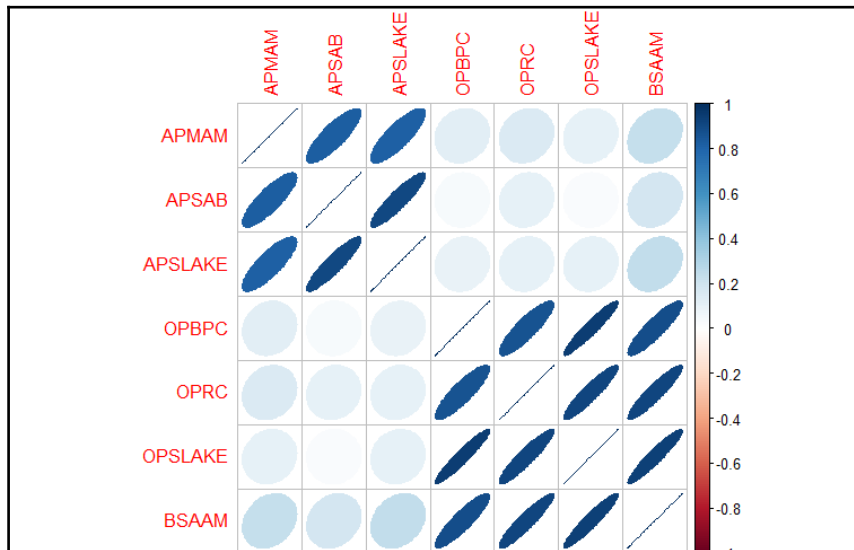
OPSLAKE 0.1075421 0.02961175 0.10058669 0.94334741
BSAAM    0.2385695 0.18329499 0.24934094 0.88574778
      OPRC    OPSLAKE    BSAAM
APMAM    0.1544155 0.10754212 0.2385695
APSAB    0.1056396 0.02961175 0.1832950
APSLAKE  0.1063836 0.10058669 0.2493409
OPBPC    0.8647073 0.94334741 0.8857478
OPRC     1.0000000 0.91914467 0.9196270
OPSLAKE  0.9191447 1.00000000 0.9384360
BSAAM    0.9196270 0.93843604 1.0000000

```

So, what does this tell us? First of all, the response variable is highly and positively correlated with the OP features with OPBPC as 0.8857, OPRC as 0.9196, and OPSLAKE as 0.9384. Also note that the AP features are highly correlated with each other and the OP features as well. The implication is that we may run into the issue of multi-collinearity. The correlation plot matrix provides a nice visual of the correlations as follows:

```
> corplot(water.cor, method = "ellipse")
```

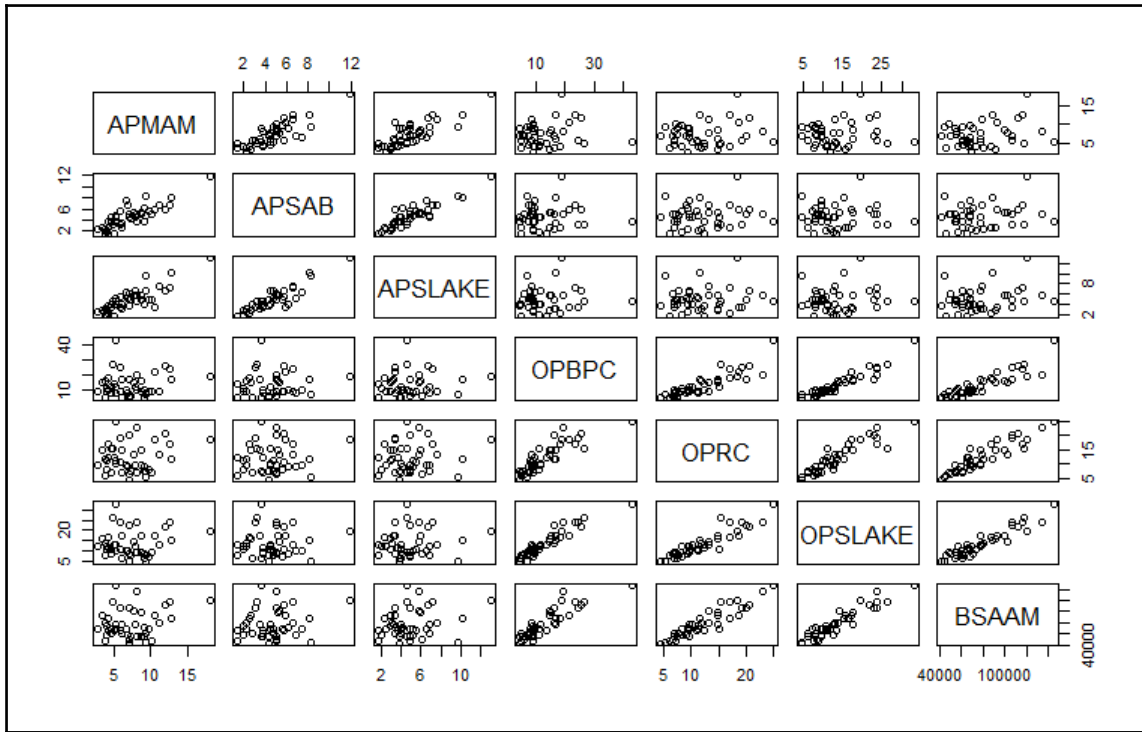
The output of the preceding code snippet is as follows:



Another popular visual is a scatterplot matrix. This can be called with the `pairs()` function. It reinforces what we saw in the correlation plot in the previous output:

```
> pairs(~ ., data = social.water)
```

The output of the preceding code snippet is as follows:



Modeling and evaluation

One of the key elements that we will cover here is the very important task of feature selection. In this chapter, we will discuss the best subsets regression methods stepwise, using the `leaps` package. Later chapters will cover more advanced techniques.

Forward stepwise selection starts with a model that has zero features; it then adds the features one at a time until all the features are added. A selected feature is added in the process that creates a model with the lowest RSS. So in theory, the first feature selected should be the one that explains the response variable better than any of the others, and so on.



It is important to note that adding a feature will always decrease RSS and increase R-squared, but it will not necessarily improve the model `fit` and interpretability.

Backward stepwise regression begins with all the features in the model and removes the least useful, one at a time. A hybrid approach is available where the features are added through forward stepwise regression, but the algorithm then examines if any features that no longer improve the model fit can be removed. Once the model is built, the analyst can examine the output and use various statistics to select the features they believe provide the best fit.

It is important to add here that stepwise techniques can suffer from serious issues. You can perform a forward stepwise on a dataset, then a backward stepwise, and end up with two completely conflicting models. The bottomline is that stepwise can produce biased regression coefficients; in other words, they are too large and the confidence intervals are too narrow (Tibshirani, 1996).

Best subsets regression can be a satisfactory alternative to the stepwise methods for feature selection. In best subsets regression, the algorithm fits a model for all the possible feature combinations; so if you have 3 features, 7 models will be created. As with stepwise regression, the analyst will need to apply judgment or statistical analysis to select the optimal model. Model selection will be the key topic in the discussion that follows. As you might have guessed, if your dataset has many features, this can be quite a task, and the method does not perform well when you have more features than observations (p is greater than n).

Certainly, these limitations for best subsets do not apply to our task at hand. Given its limitations, we will forgo stepwise, but please feel free to give it a try. We will begin by loading the `leaps` package. In order that we may see how feature selection works, we will first build and examine a model with all the features, then drill down with best subsets to select the best fit.

To build a linear model with all the features, we can again use the `lm()` function. It will follow the form: `fit = lm(y ~ x1 + x2 + x3...xn)`. A neat shortcut, if you want to include all the features, is to use a period after the tilde symbol instead of having to type them all in. For starters, let's load the `leaps` package and build a model with all the features for examination as follows:

```
> library(leaps)

> fit <- lm(BSAAM ~ ., data = socal.water)
```

```
> summary(fit)

Call:
lm(formula = BSAAM ~ ., data = socal.water)

Residuals:
    Min       1Q   Median       3Q      Max
-12690  -4936  -1424   4173  18542

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 15944.67    4099.80   3.889 0.000416
***
APMAM         -12.77      708.89  -0.018 0.985725
APSAB        -664.41     1522.89  -0.436 0.665237
APSLAKE       2270.68     1341.29   1.693 0.099112 .
OPBPC          69.70      461.69   0.151 0.880839
OPRC         1916.45      641.36   2.988 0.005031 **
OPSLAKE       2211.58      752.69   2.938 0.005729 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05
                 '.' 0.1 ' ' 1
Residual standard error: 7557 on 36 degrees of
freedom
Multiple R-squared:  0.9248,    Adjusted R-squared:
0.9123
F-statistic: 73.82 on 6 and 36 DF,  p-value: <
2.2e-16
```

Just like univariate regression, we examine the p-value on the F-statistic to verify that at least one of the coefficients is not zero. Indeed, the p-value is highly significant. We should also have significant p-values for the OPRC and OPSLAKE parameters. Interestingly, OPBPC is not significant despite being highly correlated with the response variable. In short, when we control for the other OP features, OPBPC no longer explains any meaningful variation of the predictor, which is to say that the feature OPBPC adds nothing from a statistical standpoint with OPRC and OPSLAKE in the model.

With the first model built, let's move on to best subsets. We create the `sub.fit` object using the `regsubsets()` function of the `leaps` package as follows:

```
> sub.fit <- regsubsets(BSAAM ~ ., data =  
  social.water)
```

Then we create the `best.summary` object to examine the models further. As with all R objects, you can use the `names()` function to list what outputs are available:

```
> best.summary <- summary(sub.fit)  
  
> names(best.summary)  
[1] "which" "rsq" "rss" "adjr2" "cp"  
     "bic" "outmat" "obj"
```

Other valuable functions in model selection include `which.min()` and `which.max()`. These functions will provide the model that has the minimum or maximum value respectively, as shown in following code snippet:

```
> which.min(best.summary$rss)  
[1] 6
```

The code tells us that the model with six features has the smallest RSS, which it should have, as that is the maximum number of inputs and more inputs mean a lower RSS. An important point here is that adding features will always decrease RSS! Furthermore, it will always increase R-squared. We could add a completely irrelevant feature such as the number of wins for the Los Angeles Lakers and RSS would decrease and R-squared would increase. The amount would likely be miniscule, but present nonetheless. As such, we need an effective method to properly select the relevant features.

For feature selection, there are four statistical methods that we will talk about in this chapter: **Aikake's Information Criterion (AIC)**, **Mallow's Cp (Cp)**, **Bayesian Information Criterion (BIC)**, and the adjusted R-squared. With the first three, the goal is to minimize the value of the statistic; with adjusted R-squared, the goal is to maximize the statistics value. The purpose of these statistics is to create as parsimonious a model as possible, in other words, to penalize model complexity.

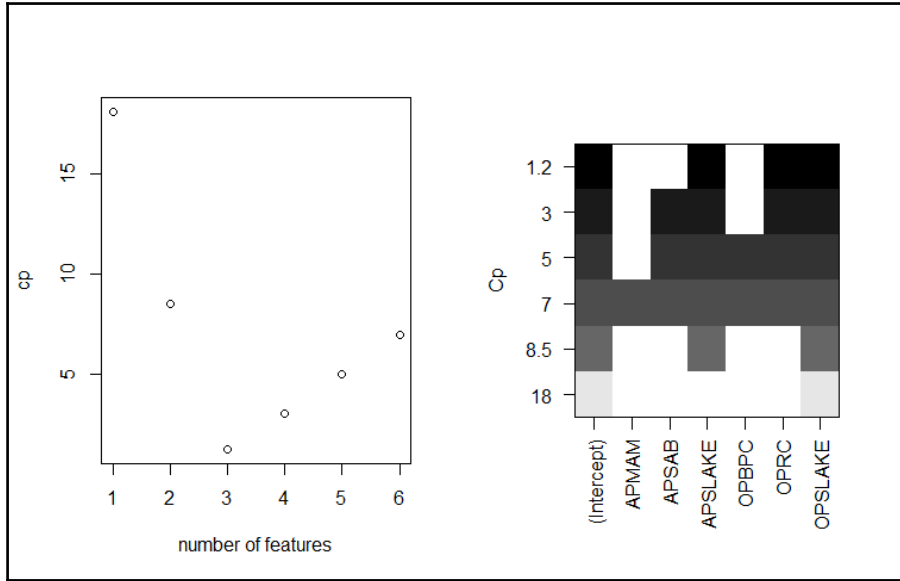
The formulation of these four statistics is as follows:

- $AIC = n * \log\left(\frac{RSS_p}{n}\right) + 2 * p$, where p is the number of features in the model we are testing
- $CP = \frac{RSS_p}{MSE_f} - n + 2 * p$, where p is the number of features in the model we are testing and MSE_f is the mean of the squared error of the model with all features included and n is the sample size
- $BIC = n * \log\left(\frac{RSS_p}{n}\right) + p * \log(n)$, where p is the number of features in the model we are testing and n is the sample size
- $Adjusted\ Rsquared = 1 - \left(\frac{RSS}{n-p-1}\right) / \left(\frac{Rsquared}{n-1}\right)$, where p is the number of features in the model we are testing and n is the sample size

In a linear model, AIC and C_p are proportional to each other, so we will only concern ourselves with C_p , which follows the output available in the `leaps` package. BIC tends to select the models with fewer variables than C_p , so we will compare both. To do so, we can create and analyze two plots side by side. Let's do this for C_p , followed by BIC , with the help of the following code snippet:

```
> par(mfrow = c(1, 2))  
  
> plot(best.summary$cp, xlab = "number of  
  features", ylab = "cp")  
  
> plot(sub.fit, scale = "Cp")
```

The output of the preceding code snippet is as follows:



In the plot on the left-hand side, the model with three features has the lowest **Cp**. The plot on the right-hand side displays those features that provide the lowest **Cp**. The way to read this plot is to select the lowest **Cp** value at the top of the y axis, which is **1.2**. Then, move to the right and look at the colored blocks corresponding to the x axis. Doing this, we see that **APSLAKE**, **OPRC**, and **OPSLAKE** are the features included in this specific model. By using the `which.min()` and `which.max()` functions, we can identify how **Cp** compares to BIC and the adjusted R-squared:

```
> which.min(best.summary$bic)
[1] 3

> which.max(best.summary$adjr2)
[1] 3
```

In this example, BIC and adjusted R-squared match the **C_p** for the optimal model. Now, just as with univariate regression, we need to examine the model and test the assumptions. We'll do this by creating a linear model object and examining the plots much as we did earlier, as follows:

```
> best.fit <- lm(BSAAM ~ APSLAKE + OPRC + OPSLAKE,
  data =
  socal.water)

> summary(best.fit)
Call:
lm(formula = BSAAM ~ APSLAKE + OPRC + OPSLAKE)

Residuals:
    Min       1Q   Median       3Q      Max
-12964  -5140  -1252   4446  18649

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  15424.6      3638.4   4.239 0.000133
***
APSLAKE       1712.5       500.5   3.421 0.001475 **
OPRC          1797.5       567.8   3.166 0.002998 **
OPSLAKE       2389.8       447.1   5.346 4.19e-06
***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05
                 '.' 0.1 ' ' 1

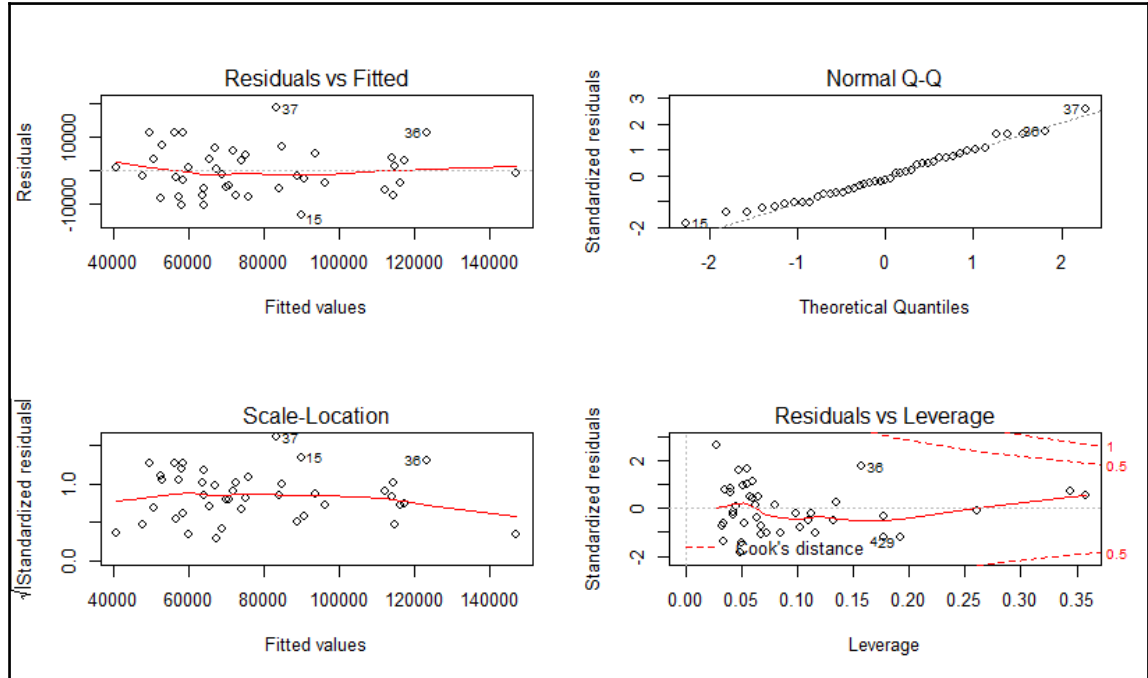
Residual standard error: 7284 on 39 degrees of
freedom
Multiple R-squared:  0.9244,    Adjusted R-squared:
0.9185
F-statistic: 158.9 on 3 and 39 DF,  p-value: <
2.2e-16
```

With the three-feature model, *F*-statistic and all the t-tests have significant p-values. Having passed the first test, we can produce our diagnostic plots:

```
> par(mfrow = c(2,2))

> plot(best.fit)
```


The output of the preceding code snippet is as follows:



Looking at the plots, it seems safe to assume that the residuals have a constant variance and are normally distributed. There is nothing in the leverage plot that would indicate a requirement for further investigation.

To investigate the issue of collinearity, one can call up the **Variance Inflation Factor (VIF)** statistic. VIF is the ratio of the variance of a feature's coefficient, when fitting the full model, divided by the feature's coefficient variance when fitted by itself. The formula is $1 / (1 - R^2_i)$, where R^2_i is the R-squared for our feature of interest, i , being regressed by all the other features. The minimum value that the VIF can take is 1, which means no collinearity at all. There are no hard and fast rules, but in general a VIF value that exceeds 5 (or some say 10) indicates a problematic amount of collinearity (James, p.101, 2013). A precise value is difficult to select, because there is no hard statistical cut-off point for when multi-collinearity makes your model unacceptable.

The `vif()` function in the `car` package is all that is needed to produce the values, as can be seen in the following code snippet:

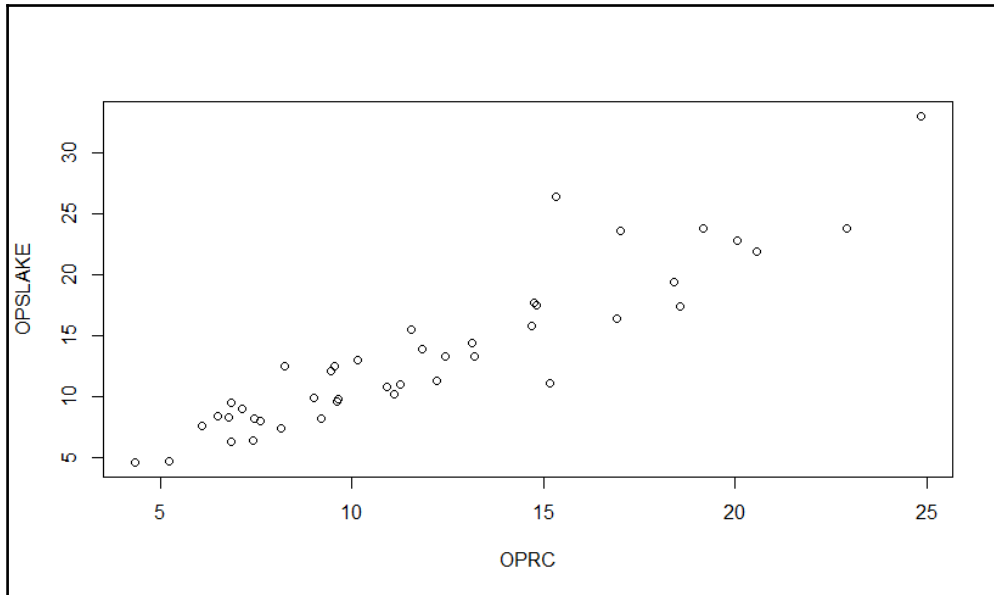
```
> vif(best.fit)

APSLAKE      OPRC  OPSLAKE
1.011499  6.452569  6.444748
```

It shouldn't be surprising that we have a potential collinearity problem with **OPRC** and **OPSLAKE** (values greater than 5) based on the correlation analysis. A plot of the two variables drives the point home, as seen in the following screenshot:

```
> plot(socal.water$OPRC, socal.water$OPSLAKE, xlab
      = "OPRC", ylab = "OPSLAKE")
```

The output of the preceding command is as follows:



The simple solution to address collinearity is to drop the variables to remove the problem without compromising the predictive ability. If we look at the adjusted R-squared from the best subsets, we can see that the two-variable model of APSLAKE and OPSLAKE has produced a value of 0.90, while adding OPRC has only marginally increased it to 0.92:

```
> best.summary$adjr2 #adjusted r-squared values
[1] 0.8777515 0.9001619 0.9185369 0.9168706
    0.9146772 0.9123079
```

Let's have a look at the two-variable model and test its assumptions:

```
> fit.2 <- lm(BSAAM ~ APSLAKE+OPSLAKE, data =
  socal.water)

> summary(fit.2)

Call:
lm(formula = BSAAM ~ APSLAKE + OPSLAKE)

Residuals:
    Min       1Q   Median       3Q      Max
-13335.8  -5893.2  -171.8   4219.5  19500.2

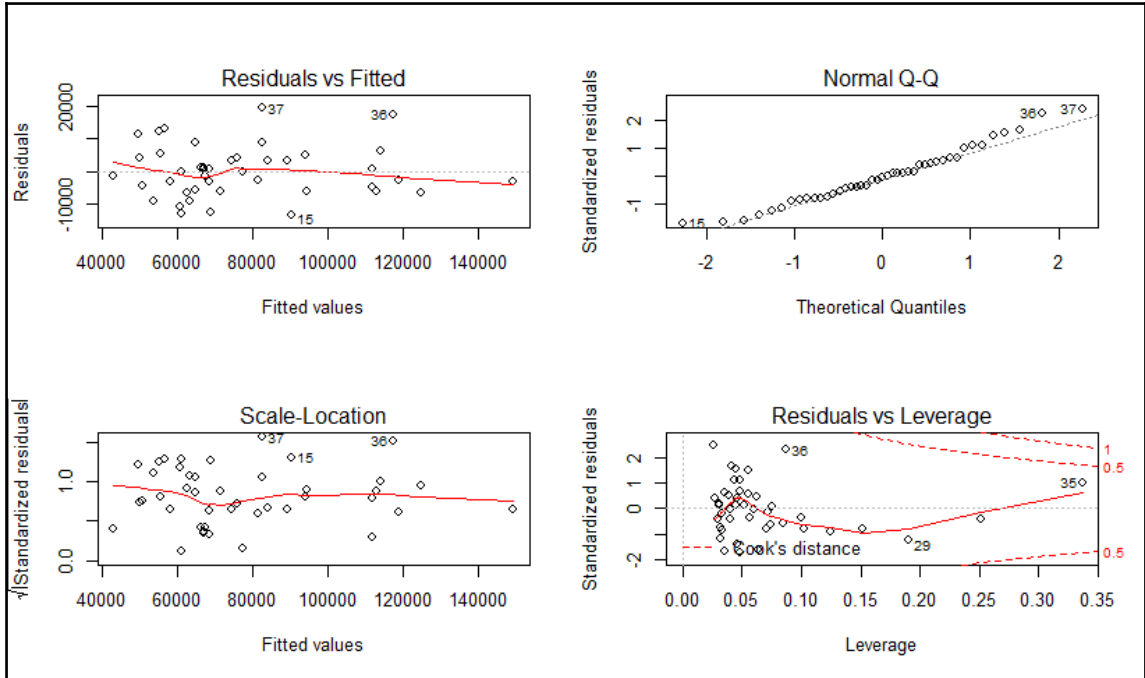
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  19144.9     3812.0   5.022  1.1e-05
***
APSLAKE       1768.8       553.7   3.194  0.00273 **
OPSLAKE       3689.5       196.0  18.829 < 2e-16
***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05
                 '.' 0.1 ' ' 1

Residual standard error: 8063 on 40 degrees of
freedom
Multiple R-squared:  0.9049,    Adjusted R-squared:
 0.9002
F-statistic: 190.3 on 2 and 40 DF,  p-value: <
 2.2e-16

> par(mfrow=c(2,2))

> plot(fit.2)
```

The output of the preceding code snippet is as follows:



The model is significant, and the diagnostics do not seem to be a cause for concern. This should take care of our collinearity problem as well and we can check that using the `vif()` function again:

```
> vif(fit.2)

APSLAKE  OPSLAKE 
1.010221 1.010221
```

As I stated previously, I don't believe the plot of fits versus residuals is of concern, but if you have any doubts you can formally test the assumption of the constant variance of errors in R. This test is known as the **Breusch-Pagan (BP)** test. For this, we need to load the `lmtest` package, and run one line of code. The BP test has the null hypothesis that the error variances are zero versus the alternative of not zero:

```
> library(lmtest)

> bptest(fit.2)

studentized Breusch-Pagan test
```

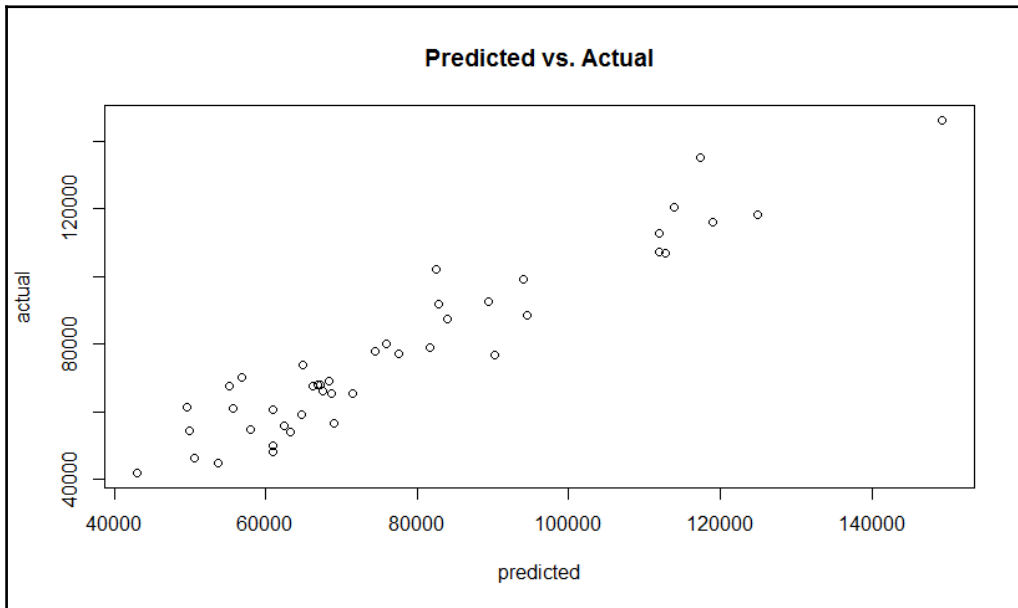
```
data: fit.2  
BP = 0.0046, df = 2, p-value = 0.9977
```

We do not have evidence to reject the null that implies that the error variances are zero because $p\text{-value} = 0.9977$. The $BP = 0.0046$ value in the summary of the test is the chi-squared value.

All things considered, it appears that the best predictive model is with the two features APSLAKE and OPSLAKE. The model can explain 90 percent of the variation in the stream runoff volume. To forecast the runoff, it would be equal to 19,145 (the intercept) plus 1,769 times the measurement at APSLAKE plus 3,690 times the measurement at OPSLAKE. A scatterplot of the Predicted vs. Actual values can be done in base R using the fitted values from the model and the response variable values as follows:

```
> plot(fit.2$fitted.values, socal.water$BSAAM, xlab  
= "predicted", ylab = "actual", main = "Predicted  
vs.Actual")
```

The output of the preceding code snippet is as follows:



Although informative, the base graphics of R are not necessarily ready for a presentation to be made to business partners. However, we can easily spruce up this plot in R. Several packages to improve graphics are available for this example, I will use `ggplot2`. Before producing the plot, we must put the predicted values into our data frame, `socal.water`. I also want to rename `BSAAM` as `Actual` and put in a new vector within the data frame, as shown in the following code snippet:

```
> socal.water["Actual"] = water$BSAAM #create the
  vector Actual

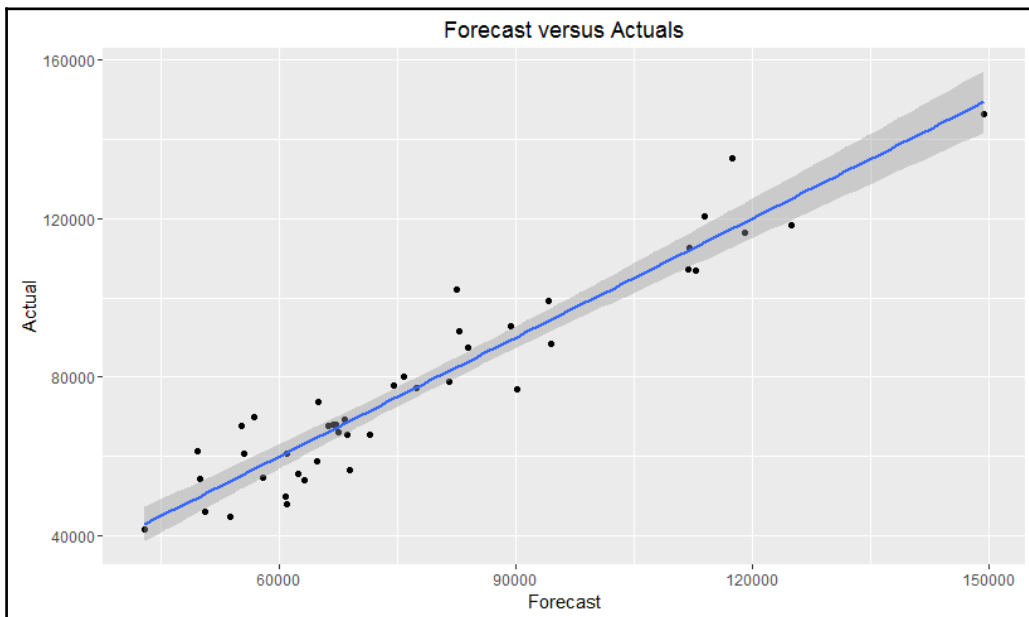
> socal.water$Forecast = predict(fit.2) #populate
  Forecast with the predicted values
```

Next we will load the `ggplot2` package and produce a nicer graphic with just one line of code:

```
> library(ggplot2)

> ggplot(socal.water, aes(x = Forecast, y =
  Actual)) + geom_point() + geom_smooth(method =
  lm) + labs(title = "Forecast versus Actuals")
```

The output is as follows:



Let's examine one final model selection technique before moving on. In the upcoming chapters, we will be discussing cross-validation at some length. Cross-validation is a widely used and effective method of model selection and testing. Why is this necessary at all? It comes down to the bias-variance trade-off. Professor Tarpey of Wright State University has a nice quote on the subject:

"Often we use regression models to predict future observations. We can use our data to fit the model. However, it is cheating to then access how well the model predicts responses using the same data that was used to estimate the model - this will tend to give overly optimistic results in terms of how well a model is able to predict future observations. If we leave out an observation, fit the model and then predict the left out response, then this will give a less biased idea of how well the model predicts."

The cross-validation technique discussed by Professor Tarpey in the preceding quote is known as the **Leave-One-Out Cross-Validation (LOOCV)**. In linear models, you can easily perform an LOOCV by examining the **Prediction Error Sum of Squares (PRESS)** statistic and selecting the model that has the lowest value. The R library `MPV` will calculate the statistic for you, as shown in the following code:

```
> library(MPV)

> PRESS(best.fit)
[1] 2426757258

> PRESS(fit.2)
[1] 2992801411
```

By this statistic alone, we could select our `best.fit` model. However, as described previously, I still believe that the more parsimonious model is better in this case. You can build a simple function to calculate the statistic on your own, taking advantage of some elegant matrix algebra as shown in the following code:

```
> PRESS.best = sum((resid(best.fit)/(1 -
  hatvalues(best.fit)))^2)

> PRESS.fit.2 = sum((resid(fit.2)/(1 -
  hatvalues(fit.2)))^2)

> PRESS.best
[1] 2426757258

> PRESS.fit.2
[1] 2992801411
```

"What are `hatvalues`?" you might ask. Well, if we take our linear model $Y = B_0 + B_1x + e$, we can turn this into a matrix notation: $Y = XB + E$. In this notation, Y remains unchanged, X is the matrix of the input values, B is the coefficient, and E represents the errors. This linear model solves for the value of B . Without going into the painful details of matrix multiplication, the regression process yields what is known as a **Hat Matrix**. This matrix maps, or as some say projects, the calculated values of your model to the actual values; as a result, it captures how influential a specific observation is in your model. So, the sum of the squared residuals divided by 1 minus `hatvalues` is the same as LOOCV.

Other linear model considerations

Before moving on, there are two additional linear model topics that we need to discuss. The first is the inclusion of a qualitative feature, and the second is an interaction term; both are explained in the following sections.

Qualitative features

A qualitative feature, also referred to as a factor, can take on two or more levels such as Male/Female or Bad/Neutral/Good. If we have a feature with two levels, say gender, then we can create what is known as an indicator or dummy feature, arbitrarily assigning one level as 0 and the other as 1. If we create a model with just the indicator, our linear model would still follow the same formulation as before, that is, $Y = B_0 + B_1x + e$. If we code the feature as male being equal to 0 and female equal to 1, then the expectation for male would just be the intercept B_0 , while for female it would be $B_0 + B_1x$. In the situation where you have more than two levels of the feature, you can create $n-1$ indicators; so, for three levels you would have two indicators. If you created as many indicators as levels, you would fall into the dummy variable trap, which results in perfect multi-collinearity.

We can examine a simple example to learn how to interpret the output. Let's load the `ISLR` package and build a model with the `Carseats` dataset using the following code snippet:

```
> library(ISLR)

> data(Carseats)

> str(Carseats)

'data.frame':  400 obs. of  11 variables:
 $ Sales      : num  9.5 11.22 10.06 7.4 4.15 ...
 $ CompPrice  : num  138 111 113 117 141 124 115 136
```



```
      132 132 ...
$ Income      : num  73 48 35 100 64 113 105 81 110
      113 ...
$ Advertising: num  11 16 10 4 3 13 0 15 0 0 ...
$ Population : num  276 260 269 466 340 501 45 425
      108 131 ...
$ Price       : num  120 83 80 97 128 72 108 120 124
      124 ...
$ ShelfLoc    : Factor w/ 3 levels
      "Bad", "Good", "Medium": 1 2 3 3 1
      1 3 2 3 3 ...
$ Age         : num  42 65 59 55 38 78 71 67 76 76
      ...
$ Education   : num  17 10 12 14 13 16 15 10 10 17
      ...
$ Urban       : Factor w/ 2 levels "No", "Yes": 2 2 2
      2 2 1 2 2 1 1
      ...
$ US          : Factor w/ 2 levels "No", "Yes": 2 2 2
      2 1 2 1 2 1 2
      ..
```

For this example, we will predict the sales of `Carseats` using just `Advertising`, a quantitative feature and the qualitative feature `ShelveLoc`, which is a factor of three levels: Bad, Good, and Medium. With factors, R will automatically code the indicators for the analysis. We build and analyze the model as follows:

```
> sales.fit <- lm(Sales ~ Advertising + ShelveLoc,
  data = Carseats)

> summary(sales.fit)

Call:
lm(formula = Sales ~ Advertising + ShelveLoc, data =
Carseats)

Residuals:
    Min       1Q   Median       3Q      Max
-6.6480 -1.6198 -0.0476  1.5308  6.4098

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)    4.89662    0.25207  19.426 < 2e-
16 ***
Advertising     0.10071    0.01692   5.951 5.88e-
09 ***
ShelveLocGood   4.57686    0.33479  13.671 < 2e-
16 ***
```

```
ShelveLocMedium 1.75142    0.27475    6.375 5.11e-
10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05
                '.' 0.1 ' ' 1

Residual standard error: 2.244 on 396 degrees of
freedom
Multiple R-squared:  0.3733,    Adjusted R-squared:
0.3685
F-statistic: 78.62 on 3 and 396 DF,  p-value: <
2.2e-16
```

If the shelving location is good, the estimate of sales is almost double of that when the location is bad, given an intercept of 4.89662. To see how R codes the indicator features, you can use the `contrasts()` function:

```
> contrasts(Carseats$ShelveLoc)

      Good Medium
Bad      0      0
Good     1      0
Medium   0      1
```

Interaction terms

Interaction terms are similarly easy to code in R. Two features interact if the effect on the prediction of one feature depends on the value of the other feature. This would follow the formulation, $Y = B_0 + B_1x + B_2x + B_1B_2x + e$. An example is available in the `MASS` package with the `Boston` dataset. The response is the median home value, which is `medv` in the output. We will use two features: the percentage of homes with a low socioeconomic status, which is termed `lstat`, and the age of the home in years, which is termed `age` in the following output:

```
> library(MASS)

> data(Boston)

> str(Boston)

'data.frame':   506 obs. of  14 variables:
 $ crim      : num  0.00632 0.02731 0.02729 0.03237
               0.06905 ...
 $ zn        : num  18 0 0 0 0 0 12.5 12.5 12.5 12.5
               ...
```

```
$ indus : num 2.31 7.07 7.07 2.18 2.18 2.18 7.87
7.87 7.87 7.87
...
$ chas : int 0 0 0 0 0 0 0 0 0 0 ...
$ nox : num 0.538 0.469 0.469 0.458 0.458 0.458
0.524 0.524
0.524 0.524 ...
$ rm : num 6.58 6.42 7.18 7 7.15 ...
$ age : num 65.2 78.9 61.1 45.8 54.2 58.7 66.6
96.1 100 85.9
...
$ dis : num 4.09 4.97 4.97 6.06 6.06 ...
$ rad : int 1 2 2 3 3 3 5 5 5 5 ...
$ tax : num 296 242 242 222 222 222 311 311 311
311 ...
$ ptratio: num 15.3 17.8 17.8 18.7 18.7 18.7 15.2
15.2 15.2 15.2
...
$ black : num 397 397 393 395 397 ...
$ lstat : num 4.98 9.14 4.03 2.94 5.33 ...
$ medv : num 24 21.6 34.7 33.4 36.2 28.7 22.9
27.1 16.5 18.9 ...
```

Using *feature1*feature2* with the `lm()` function in the code puts both the features as well as their interaction term in the model, as follows:

```
> value.fit <- lm(medv ~ lstat * age, data =
Boston)

> summary(value.fit)

Call:
lm(formula = medv ~ lstat * age, data = Boston)

Residuals:
    Min       1Q   Median       3Q      Max
-15.806  -4.045  -1.333   2.085  27.552

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 36.0885359  1.4698355  24.553  < 2e-16
***
lstat       -1.3921168  0.1674555  -8.313 8.78e-16
***
age         -0.0007209  0.0198792  -0.036  0.9711
lstat:age    0.0041560  0.0018518   2.244  0.0252
*
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05
                '.' 0.1 ' ' 1

Residual standard error: 6.149 on 502 degrees of
freedom
Multiple R-squared:  0.5557,    Adjusted R-squared:
0.5531
F-statistic: 209.3 on 3 and 502 DF,  p-value: <
2.2e-16
```

Examining the output, we can see that, while the socioeconomic status is a highly predictive feature, the age of the home is not. However, the two features have a significant interaction to positively explain the home value.

Summary

In the context of machine learning, we train a model and test it to predict or forecast an outcome. In this chapter, we had an in-depth look at the simple yet extremely effective method of linear regression to predict a quantitative response. Later chapters will cover more advanced techniques, but many of them are mere extensions of what we have learned in this chapter. We discussed the problem of not visually inspecting the dataset and simply relying on the statistics to guide you in model selection.

With just a few lines of code, you can make powerful and insightful predictions to support decision-making. Not only is it simple and effective, but also you can include quantitative variables and interaction terms among the features. Indeed, this is a method that anyone delving into the world of machine learning must master.

3

Logistic Regression and Discriminant Analysis

"The true logic of this world is the calculus of probabilities."

- James Clerk Maxwell, Scottish physicist

In the previous chapter, we took a look at using **Ordinary Least Squares (OLS)** to predict a quantitative outcome, or in other words, linear regression. It is now time to shift gears somewhat and examine how we can develop algorithms to predict qualitative outcomes. Such outcome variables could be binary (male versus female, purchase versus does not purchase, tumor is benign versus malignant) or multinomial categories (education level or eye color). Regardless of whether the outcome of interest is binary or multinomial, the task of the analyst is to predict the probability of an observation belonging to a particular category of the outcome variable. In other words, we develop an algorithm in order to classify the observations.

To begin exploring classification problems, we will discuss why applying the OLS linear regression is not the correct technique and how the algorithms introduced in this chapter can solve these issues. We will then look at a problem of predicting whether or not a biopsied tumor mass is classified as benign or malignant. The dataset is the well-known and widely available **Wisconsin Breast Cancer Data**. To tackle this problem, we will begin by building and interpreting logistic regression models. We will also begin examining methods so as to select features and the most appropriate model. Next, we will discuss both linear and quadratic discriminant analyses and compare and contrast these with logistic regression. Then, building predictive models on the breast cancer data will follow. Finally, we will wrap it up by looking at multivariate regression splines and ways to select the best overall algorithm in order to address the question at hand. These methods (creating `test/train` datasets and cross-validation) will set the stage for more advanced machine learning methods in subsequent chapters.

Classification methods and linear regression

So, why can't we just use the least square regression method that we learned in the previous chapter for a qualitative outcome? Well, as it turns out, you can, but at your own risk. Let's assume for a second that you have an outcome that you are trying to predict and it has three different classes: mild, moderate, and severe. You and your colleagues also assume that the difference between mild and moderate and moderate and severe is an equivalent measure and a linear relationship. You can create a dummy variable where 0 is equal to mild, 1 is equal to moderate, and 2 is equal to severe. If you have reason to believe this, then linear regression might be an acceptable solution. However, qualitative assessments such as the previous ones might lend themselves to a high level of measurement error that can bias the OLS. In most business problems, there is no scientifically acceptable way to convert a qualitative response to one that is quantitative. What if you have a response with two outcomes, say fail and pass? Again, using the dummy variable approach, we could code the fail outcome as 0 and the pass outcome as 1. Using linear regression, we could build a model where the predicted value is the probability of an observation of pass or fail. However, the estimates of $\hat{Y}$ in the model will most likely exceed the probability constraints of $[0, 1]$ and thus be a bit difficult to interpret.

Logistic regression

As previously discussed, our classification problem is best modeled with the probabilities that are bound by 0 and 1. We can do this for all of our observations with a number of different functions, but here we will focus on the logistic function. The logistic function used in logistic regression is as follows:

$$\text{Probability of } Y = e^{B_0 + B_1x} / 1 + e^{B_0 + B_1x}$$

If you have ever placed a friendly wager on horse races or the World Cup, you may understand the concept better as odds. The logistic function can be turned to odds with the formulation of $\text{Probability}(Y) / 1 - \text{Probability}(Y)$. For instance, if the probability of Brazil winning the World Cup is 20 percent, then the odds are $0.2 / 1 - 0.2$, which is equal to 0.25, translating to odds of one in four.

To translate the odds back to probability, take the odds and divide by one plus the odds. The World Cup example is thus $0.25 / 1 + 0.25$, which is equal to 20 percent. Additionally, let's consider the odds ratio. Assume that the odds of Germany winning the Cup are 0.18. We can compare the odds of Brazil and Germany with the odds ratio. In this example, the odds ratio would be the odds of Brazil divided by the odds of Germany. We will end up with an odds ratio equal to $0.25/0.18$, which is equal to 1.39. Here, we will say that Brazil is 1.39 times more likely than Germany to win the World Cup.

One way to look at the relationship of logistic regression with linear regression is to show logistic regression as the log odds or $\log(P(Y)/1 - P(Y))$ is equal to $B_0 + B_1x$. The coefficients are estimated using a maximum likelihood instead of the OLS. The intuition behind the maximum likelihood is that we are calculating the estimates for B_0 and B_1 , which will create a predicted probability for an observation that is as close as possible to the actual observed outcome of Y , a so-called likelihood. The R language does what other software packages do for the maximum likelihood, which is to find the optimal combination of beta values that maximize the likelihood.

With these facts in mind, logistic regression is a very powerful technique to predict the problems involving classification and is often the starting point for model creation in such problems. Therefore, in this chapter, we will attack the upcoming business problem with logistic regression first and foremost.

Business understanding

Dr. William H. Wolberg from the University of Wisconsin commissioned the Wisconsin Breast Cancer Data in 1990. His goal behind collecting the data was to identify whether a tumor biopsy was malignant or benign. His team collected the samples using **Fine Needle Aspiration (FNA)**. If a physician identifies the tumor through examination or imaging an area of abnormal tissue, then the next step is to collect a biopsy. FNA is a relatively safe method of collecting the tissue, and complications are rare. Pathologists examine the biopsy and attempt to determine the diagnosis (malignant or benign). As you can imagine, this is not a trivial conclusion. Benign breast tumors are not dangerous as there is no risk of the abnormal growth spreading to other body parts. If a benign tumor is large enough, surgery might be needed to remove it. On the other hand, a malignant tumor requires medical intervention. The level of treatment depends on a number of factors, but it's most likely that surgery will be required, which can be followed by radiation and/or chemotherapy.

Therefore, the implications of a misdiagnosis can be extensive. A false positive for malignancy can lead to costly and unnecessary treatment, subjecting the patient to a tremendous emotional and physical burden. On the other hand, a false negative can deny a patient the treatment that they need, causing the cancer cells to spread and leading to premature death. Early treatment intervention in breast cancer patients can greatly improve their survival.

Our task then is to develop the best possible diagnostic machine learning algorithm in order to assist the patient's medical team in determining whether the tumor is malignant or not.

Data understanding and preparation

This dataset consists of tissue samples from 699 patients. It is in a data frame with 11 variables, as follows:

- `ID`: Sample code number
- `v1`: Thickness
- `v2`: Uniformity of the cell size
- `v3`: Uniformity of the cell shape
- `v4`: Marginal adhesion
- `v5`: Single epithelial cell size
- `v6`: Bare nucleus (16 observations are missing)
- `v7`: Bland chromatin
- `v8`: Normal nucleolus
- `v9`: Mitosis
- `class`: Whether the tumor diagnosis is benign or malignant; this will be the outcome that we are trying to predict

The medical team has scored and coded each of the nine features on a scale of 1 to 10.

The data frame is available in the R `MASS` package under the `biopsy` name. To prepare this data, we will load the data frame, confirm the structure, rename the variables to something meaningful, and delete the missing observations. At this point, we can begin to explore the data visually. Here is the code that will get us started when we first load the library and then the dataset; using the `str()` function, we will examine the underlying structure of the data:

```
> library(MASS)
> data(biopsy)
```



```
> str(biopsy)
'data.frame': 699 obs. of 11 variables:
 $ ID : chr "1000025" "1002945" "1015425"
       "1016277" ...
 $ V1 : int 5 5 3 6 4 8 1 2 2 4 ...
 $ V2 : int 1 4 1 8 1 10 1 1 1 2 ...
 $ V3 : int 1 4 1 8 1 10 1 2 1 1 ...
 $ V4 : int 1 5 1 1 3 8 1 1 1 1 ...
 $ V5 : int 2 7 2 3 2 7 2 2 2 2 ...
 $ V6 : int 1 10 2 4 1 10 10 1 1 1 ...
 $ V7 : int 3 3 3 3 3 9 3 3 1 2 ...
 $ V8 : int 1 2 1 7 1 7 1 1 1 1 ...
 $ V9 : int 1 1 1 1 1 1 1 1 5 1 ...
 $ class: Factor w/ 2 levels "benign","malignant": 1 1 1 1 1 2 1 1
 1 1 ...
```

An examination of the data structure shows that our features are integers and the outcome is a factor. No transformation of the data to a different structure is needed.

We can now get rid of the ID column, as follows:

```
> biopsy$ID = NULL
```

Next, we will rename the variables and confirm that the code has worked as intended:

```
> names(biopsy) <- c("thick", "u.size", "u.shape",
  "adhsn", "s.size", "nucl", "chrom", "n.nuc",
  "mit", "class")
> names(biopsy)
[1] "thick" "u.size" "u.shape" "adhsn"
    "s.size" "nucl"
    "chrom" "n.nuc"
[9] "mit" "class"
```

Now, we will delete the missing observations. As there are only 16 observations with the missing data, it is safe to get rid of them as they account for only 2 percent of all the observations. A thorough discussion of how to handle the missing data is outside the scope of this chapter and has been included in the [Appendix A, R Fundamentals](#), where I cover data manipulation. In deleting these observations, a new working data frame is created. One line of code does this trick with the `na.omit` function, which deletes all the missing observations:

```
> biopsy.v2 <- na.omit(biopsy)
```

Depending on the package in R that you are using to analyze the data, the outcome needs to be numeric, which is 0 or 1. In order to accommodate that requirement, create the variable `y`, where benign is zero and malignant 1, using the `ifelse()` function as shown here:

```
> y <- ifelse(biopsy$class == "malignant", 1, 0)
```

There are a number of ways in which we can understand the data visually in a classification problem, and I think a lot of it comes down to personal preference. One of the things that I like to do in these situations is examine the **boxplots** of the features that are split by the classification outcome. This is an excellent way to begin understanding which features may be important to the algorithm. Boxplots are a simple way to understand the distribution of the data at a glance. In my experience, it also provides you with an effective way to build the presentation story that you will deliver to your customers. There are a number of ways to do this quickly, and the `lattice` and `ggplot2` packages are quite good at this task. I will use `ggplot2` in this case with the additional package, `reshape2`. After loading the packages, you will need to create a data frame using the `melt()` function. The reason to do this is that melting the features will allow the creation of a matrix of boxplots, allowing us to easily conduct the following visual inspection:

```
> library(reshape2)
> library(ggplot2)
```

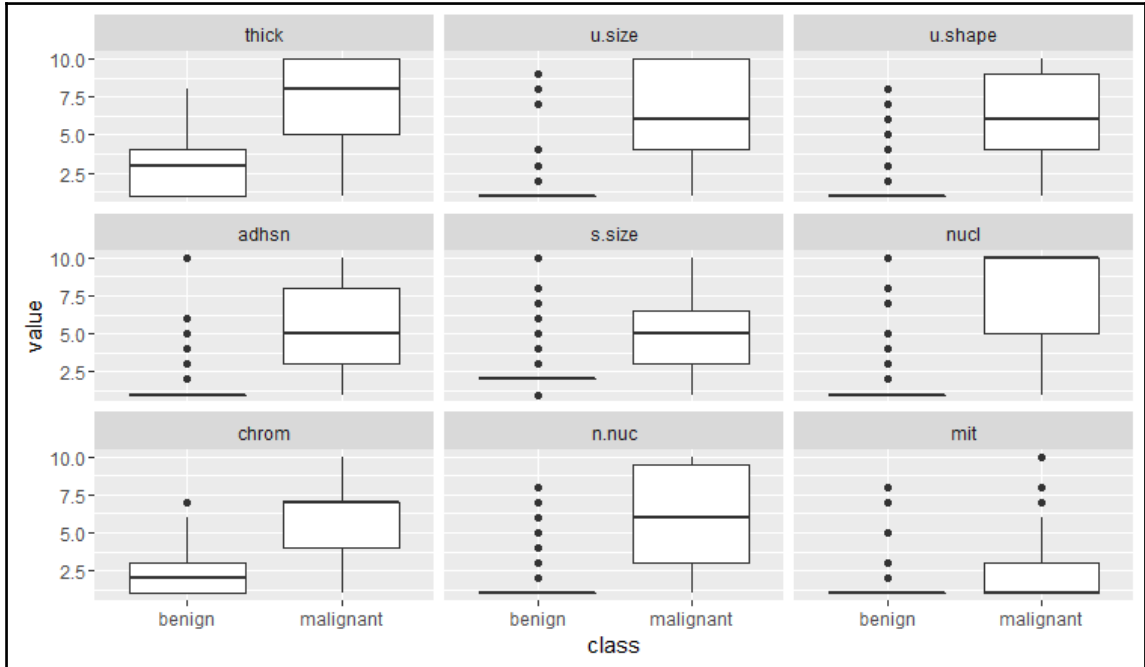
The following code melts the data by their values into one overall feature and groups them by class:

```
> biop.m <- melt(biopsy.v2, id.var = "class")
```

Through the magic of `ggplot2`, we can create a 3x3 boxplot matrix, as follows:

```
> ggplot(data = biop.m, aes(x = class, y = value))
+ geom_boxplot() + facet_wrap(~ variable, ncol = 3)
```

The following is the output of the preceding code:



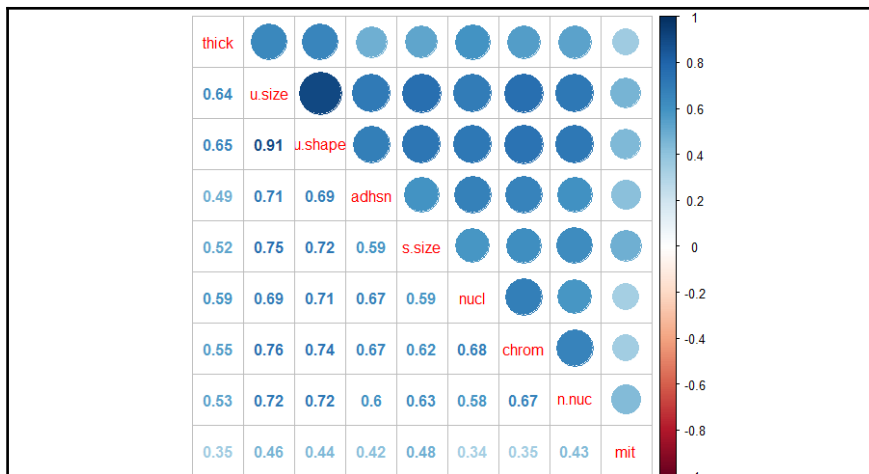
How do we interpret a boxplot? First of all, in the preceding screenshot, the thick white boxes constitute the upper and lower quartiles of the data; in other words, half of all the observations fall in the thick white box area. The dark line cutting across the box is the median value. The lines extending from the boxes are also quartiles, terminating at the maximum and minimum values, outliers notwithstanding. The black dots constitute the outliers.

By inspecting the plots and applying some judgment, it is difficult to determine which features will be important in our classification algorithm. However, I think it is safe to assume that the nuclei feature will be important, given the separation of the median values and corresponding distributions. Conversely, there appears to be little separation of the mitosis feature by class, and it will likely be an irrelevant feature. We shall see!

With all of our features quantitative, we can also do a correlation analysis as we did with linear regression. Collinearity with logistic regression can bias our estimates just as we discussed with linear regression. Let's load the `corrplot` package and examine the correlations as we did in the previous chapter, this time using a different type of correlation matrix, which has both shaded ovals and the correlation coefficients in the same plot, as follows:

```
> library(corrplot)
> bc <- cor(biopsy.v2[, 1:9]) #create an object of
  the features
> corrplot.mixed(bc)
```

The following is the output of the preceding code:



The correlation coefficients are indicating that we may have a problem with collinearity, in particular, the features of uniform shape and uniform size that are present. As part of the logistic regression modeling process, it will be necessary to incorporate the VIF analysis as we did with linear regression. The final task in the data preparation will be the creation of our `train` and `test` datasets. The purpose of creating two different datasets from the original one is to improve our ability so as to accurately predict the previously unused or unseen data.

In essence, in machine learning, we should not be so concerned with how well we can predict the current observations and should be more focused on how well we can predict the observations that were not used in order to create the algorithm. So, we can create and select the best algorithm using the training data that maximizes our predictions on the `test` set. The models that we will build in this chapter will be evaluated by this criterion.

There are a number of ways to proportionally split our data into `train` and `test` sets: 50/50, 60/40, 70/30, 80/20, and so forth. The data split that you select should be based on your experience and judgment. For this exercise, I will use a 70/30 split, as follows:

```
> set.seed(123) #random number generator
> ind <- sample(2, nrow(biopsy.v2), replace = TRUE,
  prob = c(0.7, 0.3))
> train <- biopsy.v2[ind==1, ] #the training data
  set
> test <- biopsy.v2[ind==2, ] #the test data set
> str(test) #confirm it worked
'data.frame':  209 obs. of  10 variables:
 $ thick  : int  5 6 4 2 1 7 6 7 1 3 ...
 $ u.size : int  4 8 1 1 1 4 1 3 1 2 ...
 $ u.shape: int  4 8 1 2 1 6 1 2 1 1 ...
 $ adhsn  : int  5 1 3 1 1 4 1 10 1 1 ...
 $ s.size : int  7 3 2 2 1 6 2 5 2 1 ...
 $ nucl   : int  10 4 1 1 1 1 1 10 1 1 ...
 $ chrom  : int  3 3 3 3 3 4 3 5 3 2 ...
 $ n.nuc  : int  2 7 1 1 1 3 1 4 1 1 ...
 $ mit    : int  1 1 1 1 1 1 1 4 1 1 ...
 $ class  : Factor w/ 2 levels benign,"malignant":
  1 1 1 1 1 2 1
  2 1 1 ...
```

To ensure that we have a well-balanced outcome variable between the two datasets, we will perform the following check:

```
> table(train$class)
benign malignant
 302         172
> table(test$class)
benign malignant
 142         67
```

This is an acceptable ratio of our outcomes in the two datasets; with this, we can begin the modeling and evaluation.

Modeling and evaluation

For this part of the process, we will start with a logistic regression model of all the input variables and then narrow down the features with the best subsets. After this, we will try our hand at **discriminant analysis** and **Multivariate Adaptive Regression Splines (MARS)**.

The logistic regression model

We've already discussed the theory behind logistic regression, so we can begin fitting our models. An R installation comes with the `glm()` function fitting the generalized linear models, which are a class of models that includes logistic regression. The code syntax is similar to the `lm()` function that we used in the previous chapter. One big difference is that we must use the `family = binomial` argument in the function, which tells R to run a logistic regression method instead of the other versions of the generalized linear models. We will start by creating a model that includes all of the features on the `train` set and see how it performs on the `test` set, as follows:

```
> full.fit <- glm(class ~ ., family = binomial,
  data = train)
> summary(full.fit)
Call:
glm(formula = class ~ ., family = binomial, data =
  train)
Deviance Residuals:
    Min       1Q   Median       3Q      Max
-3.3397  -0.1387  -0.0716   0.0321   2.3559
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -9.4293     1.2273  -7.683 1.55e-14
***
thick          0.5252     0.1601   3.280 0.001039 **
u.size        -0.1045     0.2446  -0.427 0.669165
u.shape        0.2798     0.2526   1.108 0.268044
adhsn          0.3086     0.1738   1.776 0.075722 .
s.size         0.2866     0.2074   1.382 0.167021
nucl           0.4057     0.1213   3.344 0.000826
***
chrom          0.2737     0.2174   1.259 0.208006
n.nuc          0.2244     0.1373   1.635 0.102126
mit           0.4296     0.3393   1.266 0.205402
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05
                 '.' 0.1 ' ' 1
(Dispersion parameter for binomial family taken to
  be 1)
    Null deviance: 620.989  on 473  degrees of
    freedom
Residual deviance:  78.373  on 464  degrees of
    freedom
AIC: 98.373
Number of Fisher Scoring iterations: 8
```

The `summary()` function allows us to inspect the coefficients and their p-values. We can see that only two features have p-values less than 0.05 (thickness and nuclei). An examination of the 95 percent confidence intervals can be called on with the `confint()` function, as follows:

```
> confint(full.fit)
              2.5 %      97.5 %
(Intercept) -12.23786660 -7.3421509
thick        0.23250518  0.8712407
u.size       -0.56108960  0.4212527
u.shape      -0.24551513  0.7725505
adhsn        -0.02257952  0.6760586
s.size       -0.11769714  0.7024139
nucl         0.17687420  0.6582354
chrom        -0.13992177  0.7232904
n.nuc        -0.03813490  0.5110293
mit          -0.14099177  1.0142786
```

Note that the two significant features have confidence intervals that do not cross zero. You cannot translate the coefficients in logistic regression as the change in Y is based on a one-unit change in X . This is where the odds ratio can be quite helpful. The beta coefficients from the log function can be converted to odds ratios with an exponent (beta).

In order to produce the odds ratios in R, we will use the following `exp(coef())` syntax:

```
> exp(coef(full.fit))
(Intercept)      thick      u.size      u.shape
adhsn
8.033466e-05 1.690879e+00 9.007478e-01 1.322844e+00
1.361533e+00
s.size      nucl      chrom      n.nuc      mit
1.331940e+00 1.500309e+00 1.314783e+00 1.251551e+00
1.536709e+00
```

The interpretation of an odds ratio is the change in the outcome odds resulting from a unit change in the feature. If the value is greater than 1, it indicates that, as the feature increases, the odds of the outcome increase. Conversely, a value less than 1 would mean that, as the feature increases, the odds of the outcome decrease. In this example, all the features except `u.size` will increase the log odds.

One of the issues pointed out during data exploration was the potential issue of multi-collinearity. It is possible to produce the VIF statistics that we did in linear regression with a logistic model in the following way:

```
> library(car)
> vif(full.fit)
      thick u.size u.shape adhsn   s.size   nucl
      chrom  n.nuc
1.2352 3.2488  2.8303   1.3021  1.6356   1.3729
      1.5234  1.3431
      mit
1.059707
```

None of the values are greater than the VIF rule of thumb statistic of five, so collinearity does not seem to be a problem. Feature selection will be the next task; but, for now, let's produce some code to look at how well this model does on both the `train` and `test` sets.

You will first have to create a vector of the predicted probabilities, as follows:

```
> train.probs <- predict(full.fit, type =
  "response")
> train.probs[1:5] #inspect the first 5 predicted
  probabilities
[1] 0.02052820 0.01087838 0.99992668 0.08987453
     0.01379266
```

Next, we need to evaluate how well the model performed in training and then evaluate how it fits on the test set. A quick way to do this is to produce a confusion matrix. In later chapters, we will examine the version provided by the `caret` package. There is also a version provided in the `InformationValue` package. This is where we will need the outcome as 0's and 1's. The default value by which the function selects either benign or malignant is 0.50, which is to say that any probability at or above 0.50 is classified as malignant:

```
> trainY <- y[ind==1]
> testY <- y[ind==2]
> confusionMatrix(trainY, train.probs)
      0      1
0 294      7
1   8    165
```


The rows signify the predictions, and the columns signify the actual values. The diagonal elements are the correct classifications. The top right value, 7, is the number of false negatives, and the bottom left value, 8, is the number of false positives. We can also take a look at the error rate, as follows:

```
> misClassError(trainY, train.probs)
[1] 0.0316
```

It seems we have done a fairly good job with only a 3.16% error rate on the training set. As we previously discussed, we must be able to accurately predict unseen data, in other words, our test set.

The method to create a confusion matrix for the `test` set is similar to how we did it for the training data:

```
> test.probs <- predict(full.fit, newdata = test,
  type = "response")
> misClassError(testY, test.probs)
[1] 0.0239
> confusionMatrix(testY, test.probs)
      0      1
0 139      2
1   3     65
```

It looks like we have done pretty well in creating a model with all the features. The roughly 98 percent prediction rate is quite impressive. However, we must still check whether there is room for improvement. Imagine that you or your loved one is a patient who has been diagnosed incorrectly. As previously mentioned, the implications can be quite dramatic. With this in mind, is there perhaps a better way to create a classification algorithm? Let's have a look!

Logistic regression with cross-validation

The purpose of cross-validation is to improve our prediction of the `test` set and minimize the chance of overfitting. With the **K-fold cross-validation**, the dataset is split into K equal-sized parts. The algorithm learns by alternatively holding out one of the **K-sets**; it fits a model to the other **K-1 parts**, and obtains predictions for the left-out K-set. The results are then averaged so as to minimize the errors, and appropriate features are selected. You can also perform the **Leave-One-Out-Cross-Validation (LOOCV)** method, where K is equal to 1. Simulations have shown that the LOOCV method can have averaged estimates that have a high variance. As a result, most machine learning experts will recommend that the number of K-folds should be 5 or 10.

An R package that will automatically do CV for logistic regression is the `bestglm` package. This package is dependent on the `leaps` package that we used for linear regression. The syntax and formatting of the data require some care, so let's walk through this in detail:

```
> library(bestglm)
Loading required package: leaps
```

After loading the package, we will need our outcome coded to 0 or 1. If left as a factor, it will not work. The other requirement to utilize the package is that your outcome, or `y`, is the last column and all the extraneous columns have been removed. A new data frame will do the trick for us, via the following code:

```
> X <- train[, 1:9]
> Xy <- data.frame(cbind(X, trainY))
```

Here is the code to run in order to use the CV technique with our data:

```
> bestglm(Xy = biopsy.cv, IC="CV",
  CVArgs=list(Method="HTF", K=10,
    REP=1), family=binomial)
```

The syntax, `Xy = Xy`, points to our properly formatted data frame. `IC = "CV"` tells the package that the information criterion to use is cross-validation. `CVArgs` are the CV arguments that we want to use. The `HTF` method is K-fold, which is followed by the number of folds, `K = 10`, and we are asking it to do only one iteration of the random folds with `REP = 1`. Just as with `glm()`, we will need to use `family = binomial`. On a side note, you can use `bestglm` for linear regression as well by specifying `family = gaussian`. So, after running the analysis, we will end up with the following output, giving us three features for Best Model, such as `thick`, `u.size`, and `nucl`. The statement on Morgan-Tatar search simply means that a simple exhaustive search was done for all the possible subsets, as follows:

```
Morgan-Tatar search since family is non-gaussian.
CV(K = 10, REP = 1)
BICq equivalent for q in (7.16797006619085e-05,
  0.273173435514231)
Best Model:
Estimate Std. Error    z value    Pr(>|z|)
(Intercept) -7.8147191 0.90996494 -8.587934
  8.854687e-18
thick        0.6188466 0.14713075  4.206100
  2.598159e-05
u.size       0.6582015 0.15295415  4.303260
  1.683031e-05
nucl         0.5725902 0.09922549  5.770596
```

7.899178e-09

We can put these features in `glm()` and then see how well the model did on the test set. A `predict()` function will not work with `bestglm`, so this is a required step:

```
> reduce.fit <- glm(class ~ thick + u.size + nucl,  
  family = binomial, data = train)
```

As before, the following code allows us to compare the predicted labels with the actual ones on the test set:

```
> test.cv.probs <- predict(reduce.fit, newdata =  
  test, type = "response")  
> misClassError(testY, test.cv.probs)  
[1] 0.0383  
> confusionMatrix(testY, test.cv.probs)  
      0      1  
0 139      5  
1   3     62
```

The reduced feature model is slightly less accurate than the full feature model, but all is not lost. We can utilize the `bestglm` package again, this time using the best subsets with the information criterion set to BIC:

```
> bestglm(Xy = Xy, IC = "BIC", family = binomial)  
Morgan-Tatar search since family is non-gaussian.  
BIC  
BICq equivalent for q in (0.273173435514231,  
  0.577036596263757)  
Best Model:  
  Estimate Std. Error   z value    Pr(>|z|)  
(Intercept) -8.6169613 1.03155250 -8.353391  
             6.633065e-17  
thick         0.7113613 0.14751510  4.822295  
             1.419160e-06  
adhsn         0.4537948 0.15034294  3.018398  
             2.541153e-03  
nucl          0.5579922 0.09848156  5.665956  
             1.462068e-08  
n.nuc         0.4290854 0.11845720  3.622282  
             2.920152e-04
```

These four features provide the minimum BIC score for all possible subsets. Let's try this and see how it predicts the `test` set, as follows:

```
> bic.fit <- glm(class ~ thick + adhsn + nucl +
  n.nuc, family = binomial, data = train)
> test.bic.probs <- predict(bic.fit, newdata =
  test, type = "response")
> misClassError(testY, test.bic.probs)
[1] 0.0239
> confusionMatrix(testY, test.bic.probs)
      0      1
0 138      1
1   4     66
```

Here we have five errors, just like the full model. The obvious question then is: which one is better? In any normal situation, the rule of thumb is to default to the simplest or the easiest-to-interpret model, given equality of generalization. We could run a completely new analysis with a new randomization and different ratios of the `train` and `test` sets among others. However, let's assume for a moment that we've exhausted the limits of what logistic regression can do for us. We will come back to the full model and the model that we developed on a BIC minimum at the end and discuss the methods of model selection. Now, let's move on to our discriminant analysis methods, which we will also include as possibilities in the final recommendation.

Discriminant analysis overview

Discriminant Analysis (DA), also known as **Fisher Discriminant Analysis (FDA)**, is another popular classification technique. It can be an effective alternative to logistic regression when the classes are well-separated. If you have a classification problem where the outcome classes are well-separated, logistic regression can have unstable estimates, which is to say that the confidence intervals are wide and the estimates themselves likely vary from one sample to another (James, 2013). DA does not suffer from this problem and, as a result, may outperform and be more generalized than logistic regression. Conversely, if there are complex relationships between the features and outcome variables, it may perform poorly on a classification task. For our breast cancer example, logistic regression performed well on the testing and training sets, and the classes were not well-separated. For the purpose of comparison with logistic regression, we will explore DA, both **Linear Discriminant Analysis (LDA)** and **Quadratic Discriminant Analysis (QDA)**.

DA utilizes **Baye's theorem** in order to determine the probability of the class membership for each observation. If you have two classes, for example, benign and malignant, then DA will calculate an observation's probability for both the classes and select the highest probability as the proper class.

Bayes' theorem states that the probability of Y occurring--given that X has occurred--is equal to the probability of both Y and X occurring, divided by the probability of X occurring, which can be written as follows:

$$\text{Probability of } Y|X = \frac{P(X \text{ and } Y)}{P(X)}$$

The numerator in this expression is the likelihood that an observation is from that class level and has these feature values. The denominator is the likelihood of an observation that has these feature values across all the levels. Again, the classification rule says that if you have the joint distribution of X and Y and if X is given, the optimal decision about which class to assign an observation to is by choosing the class with the larger probability (the posterior probability).

The process of attaining posterior probabilities goes through the following steps:

1. Collect data with a known class membership.
2. Calculate the prior probabilities; this represents the proportion of the sample that belongs to each class.
3. Calculate the mean for each feature by their class.
4. Calculate the variance--covariance matrix for each feature; if it is an LDA, then this would be a pooled matrix of all the classes, giving us a linear classifier, and if it is a QDA, then a variance--covariance created for each class.
5. Estimate the normal distribution (Gaussian densities) for each class.
6. Compute the discriminant function that is the rule for the classification of a new object.
7. Assign an observation to a class based on the discriminant function.

This will provide an expanded notation on the determination of the posterior probabilities, as follows:

- $\pi_k = \# \text{ of samples in class } k / \text{ total sample size}$ – This is the prior probability of a randomly chosen observation being in the “kth” class
- $f_k(X) = P(X = x | Y = k)$ is the density function of an observation that come from the kth class; we assume this come from a normal (Gaussian) distribution; with multiple features, the assumption is it comes from a multivariate Gaussian distribution
- Using $p_k(X) = \text{probability of } Y \text{ given } X$, we can adjust Bayes’ Theorem accordingly - $p_x(X) = \pi_k f_k(X) / \sum_{l=1}^k \pi_l f_l(X)$; this is the posterior probability that an observation comes from class k, given the feature values for that observation
- Assuming $k=2$ and the prior probabilities are the same, $\pi_1 = \pi_2$, then an observation is assigned to “class 1” if $2x(\mu_1 - \mu_2) > \mu_1^2 - \mu_2^2$ otherwise it is assigned to “class 2”; this is what is known as the decision boundary; DA creates $k-1$ decision boundaries i.e. with 3 classes ($k=3$), there will be 2 decision boundaries

Even though LDA is elegantly simple, it is limited by the assumption that the observations of each class are said to have a multivariate normal distribution, and there is a common covariance across the classes. QDA still assumes that observations come from a normal distribution, but it also assumes that each class has its own covariance.

Why does this matter? When you relax the common covariance assumption, you now allow quadratic terms into the discriminant score calculations, which was not possible with LDA. The mathematics behind this can be a bit intimidating and are outside the scope of this book. The important part to remember is that QDA is a more flexible technique than logistic regression, but we must keep in mind our **bias-variance** trade-off. With a more flexible technique, you are likely to have a lower bias but potentially a higher variance. Like a lot of flexible techniques, a robust set of training data is needed to mitigate a high classifier variance.

Discriminant analysis application

LDA is performed in the `MASS` package, which we have already loaded so that we can access the biopsy data. The syntax is very similar to the `lm()` and `glm()` functions.

We can now begin fitting our LDA model, which is as follows:

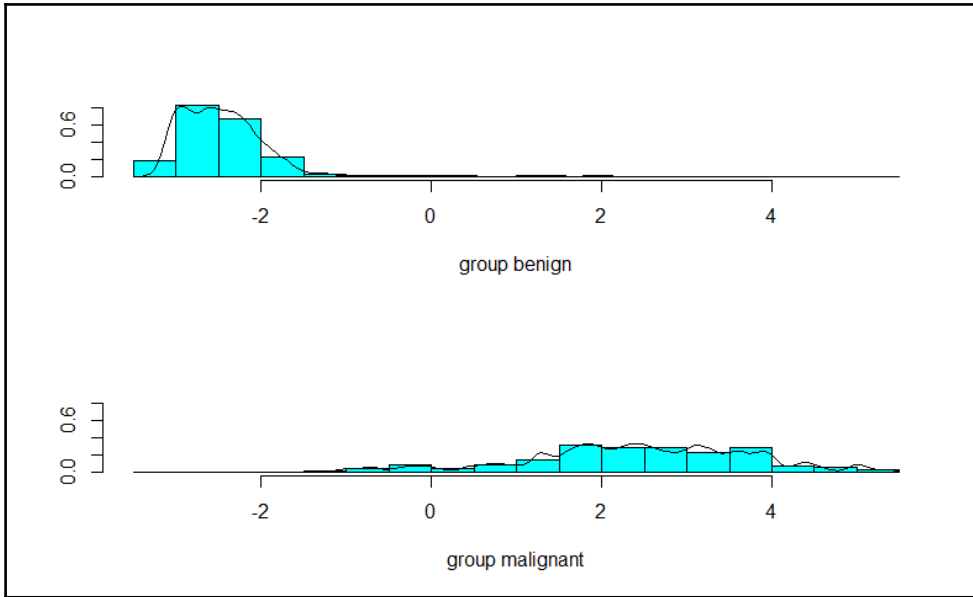
```
> lda.fit <- lda(class ~ ., data = train)
> lda.fit
Call:
lda(class ~ ., data = train)
Prior probabilities of groups:
  benign malignant 
0.6371308 0.3628692 
Group means:
      thick u.size u.shape  adhsn  s.size   nucl
chrom
benign    2.9205 1.30463 1.41390 1.32450 2.11589
      1.39735 2.08278
malignant 7.1918 6.69767 6.68604 5.66860 5.50000
      7.67441 5.95930
      n.nuc   mit
benign    1.22516 1.09271
malignant 5.90697 2.63953
Coefficients of linear discriminants:
      LD1
thick    0.19557291
u.size   0.10555201
u.shape  0.06327200
adhsn    0.04752757
s.size   0.10678521
nucl     0.26196145
chrom    0.08102965
n.nuc    0.11691054
mit      -0.01665454
```

This output shows us that Prior probabilities of groups are approximately 64 percent for benign and 36 percent for malignancy. Next is Group means. This is the average of each feature by their class. Coefficients of linear discriminants are the standardized linear combination of the features that are used to determine an observation's discriminant score. The higher the score, the more likely that the classification is malignant.

The `plot()` function in LDA will provide us with a histogram and/or the densities of the discriminant scores, as follows:

```
> plot(lda.fit, type = "both")
```

The following is the output of the preceding command:



We can see that there is some overlap in the groups, indicating that there will be some incorrectly classified observations.

The `predict()` function available with LDA provides a list of three elements: `class`, `posterior`, and `x`. The `class` element is the prediction of **benign** or **malignant**, the `posterior` is the probability score of `x` being in each class, and `x` is the linear discriminant score. Let's just extract the probability of an observation being malignant:

```
> train.lda.probs <- predict(lda.fit)$posterior[,  
  2]  
> misClassError(trainY, train.lda.probs)  
[1] 0.0401  
> confusionMatrix(trainY, train.lda.probs)  
  0    1  
0 296   13  
1   6  159
```


Well, unfortunately, it appears that our LDA model has performed much worse than the logistic regression models. The primary question is to see how this will perform on the test data:

```
> test.lda.probs <- predict(lda.fit, newdata =  
  test)$posterior[, 2]  
> misClassError(testY, test.lda.probs)  
[1] 0.0383  
> confusionMatrix(testY, test.lda.probs)  
      0      1  
0 140      6  
1   2     61
```

That's actually not as bad as I thought, given the lesser performance on the training data. From a correctly classified perspective, it still did not perform as well as logistic regression (96 percent versus almost 98 percent with logistic regression).

We will now move on to fit a QDA model. In R, QDA is also part of the `MASS` package and the function is `qda()`. Building the model is rather straightforward again, and we will store it in an object called `qda.fit`, as follows:

```
> qda.fit = qda(class ~ ., data = train)  
> qda.fit  
Call:  
qda(class ~ ., data = train)  
Prior probabilities of groups:  
  benign malignant  
0.6371308 0.3628692  
Group means:  
  Thick u.size u.shape adhsn s.size  nucl  chrom  
  n.nuc  
benign    2.9205 1.3046  1.4139 1.3245 2.1158  
          1.3973 2.0827 1.2251  
malignant 7.1918 6.6976  6.6860 5.6686 5.5000  
          7.6744 5.9593 5.9069  
          mit  
benign    1.092715  
malignant 2.639535
```

As with LDA, the output has `Group means` but does not have the coefficients because it is a quadratic function as discussed previously.

The predictions for the `train` and `test` data follow the same flow of code as with LDA:

```
> train.qda.probs <- predict(qda.fit)$posterior[,  
  2]  
> misClassError(trainY, train.qda.probs)  
[1] 0.0422  
> confusionMatrix(trainY, train.qda.probs)  
      0      1  
0 287      5  
1  15 167  
> test.qda.probs <- predict(qda.fit, newdata =  
  test)$posterior[, 2]  
> misClassError(testY, test.qda.probs)  
[1] 0.0526  
> confusionMatrix(testY, test.qda.probs)  
      0      1  
0 132      1  
1  10  66
```

We can quickly tell that QDA has performed the worst on the training data with the confusion matrix, and it has classified the `test` set poorly with 11 incorrect predictions. In particular, it has a high rate of false positives.

Multivariate Adaptive Regression Splines (MARS)

How would you like a modeling technique that provides all of the following?

- Offers the flexibility to build linear and nonlinear models for both regression and classification
- Can support variable interaction terms
- Is simple to understand and explain
- Requires little data preprocessing
- Handles all types of data: numeric, factors, and so on
- Performs well on unseen data, that is, it does well in bias-variance trade-off

If that all sounds appealing, then I cannot recommend the use of MARS models enough. The method was brought to my attention several months ago, and I have found it to perform extremely well. In fact, in a recent case of mine, it outperformed both a random forest and boosted trees on test/validation data. It has quickly become my baseline model and all others are competitors. The other benefit I've seen is that it has negated much of the feature engineering I was doing. Much of that was using **Weight-of-Evidence (WOE)** and **Information Values (IV)** to capture nonlinearity and recode the variables. This WOE/IV technique was something I was planning to write about at length in this second edition. However, I've done a number of tests, and MARS does an excellent job of doing what that technique does (that is, capture non-linearity), so I will not discuss WOE/IV at all.

To understand MARS is quite simple. First, just start with a linear or generalized linear model like we discussed previously. Then, to capture any nonlinear relationship, a `hinge` function is added. These `hinges` are simply points in the input feature that equate to a coefficient change. For example, say we have $Y = 12.5$ (our intercept) + $1.5(\text{variable } 1)$ + $3.3(\text{variable } 2)$; where variables 1 and 2 are on a scale of 1 to 10. Now, let's see how a `hinge` function for variable 2 could come into play:

$$Y = 11 \text{ (new intercept)} + 1.5(\text{variable } 1) + 4.26734(\max(0, \text{variable } 2 - 5.5))$$

Thus, we read the `hinge` function as: we take the maximum of either 0 or variable 2 *minus* 5.50. So, whenever variable 2 has a value greater than 5.5, that value will be multiplied times the coefficient; otherwise, it will be zero. The method will accommodate multiple hinges for each variable and also interaction terms.

The other interesting thing about MARS is automatic variable selection. This can be done with cross-validation, but the default is to build through a forward pass, much like forward step wise regression, then a backward pass to prune the model, which after the forward pass is likely to overfit the data. This backward pass prunes input features and removes hinges based on **Generalized Cross Validation (GCV)**:

$$GCV = RSS / (N * (1 - \text{Effective Number of Parameters} / N)^2)$$

$$\text{Effective Number of Parameters} = \text{Number of Input Features} + \text{Penalty} * (\text{Number of Input Features} - 1) / 2$$

In the `earth` package, $Penalty = 2$ for additive model and 3 for multiplicative model, that is one with interaction terms.

In R, there are quite a few parameters you can tune. I will demonstrate in the example an effective and simple way to implement the methodology. If you so desire, you can learn more about its flexibility in the excellent online *Notes on the earth package*, by Stephen Milborrow, available at this link:

<http://www.milbo.org/doc/earth-notes.pdf>

With that introduction out of the way, let's get started. You can use the MDA package, but I learned on `earth`, so that is what I will present. The code is similar to the earlier examples, where we used `glm()`. However, it is important to specify how you want the model pruned and that it is a binomial response variable. Here, I specify a model selection of a five-fold cross validation (`pmethod = "cv"` and `nfold = 5`), repeated 3 times (`ncross = 3`), as an additive model only with no interactions (`degree = 1`) and only one hinge per input feature (`minspan = -1`). In the data I've been working with, both interaction terms and multiple hinges have led to overfitting. Of course, your results may vary. The code is as follows:

```
> library(earth)
> set.seed(1)
> earth.fit <- earth(class ~ ., data = train,
                    pmethod = "cv",
                    nfold = 5,
                    ncross = 3,
                    degree = 1,
                    minspan = -1,
                    glm=list(family=binomial)
                    )
```

We now examine the model summary. At first, it can seem a little confusing. Of course, we have the model formula and the logistic coefficients, including the `hinge` functions, followed by some commentary and numbers related to generalized R-squared, and so on.

What happens is that a MARS model is built on the data first as a standard linear regression where the response variable is internally coded to 0's and 1's. After feature/variable pruning with final model creation, then it is translated to a GLM. So, just ignore the R-squared values:

```
> summary(earth.fit)
Call: earth(formula=class~., data=train,
  pmethod="cv",
  glm=list(family=binomial), degree=1, ncross=3,
  nfold=5, minspan=-1)
GLM coefficients
```

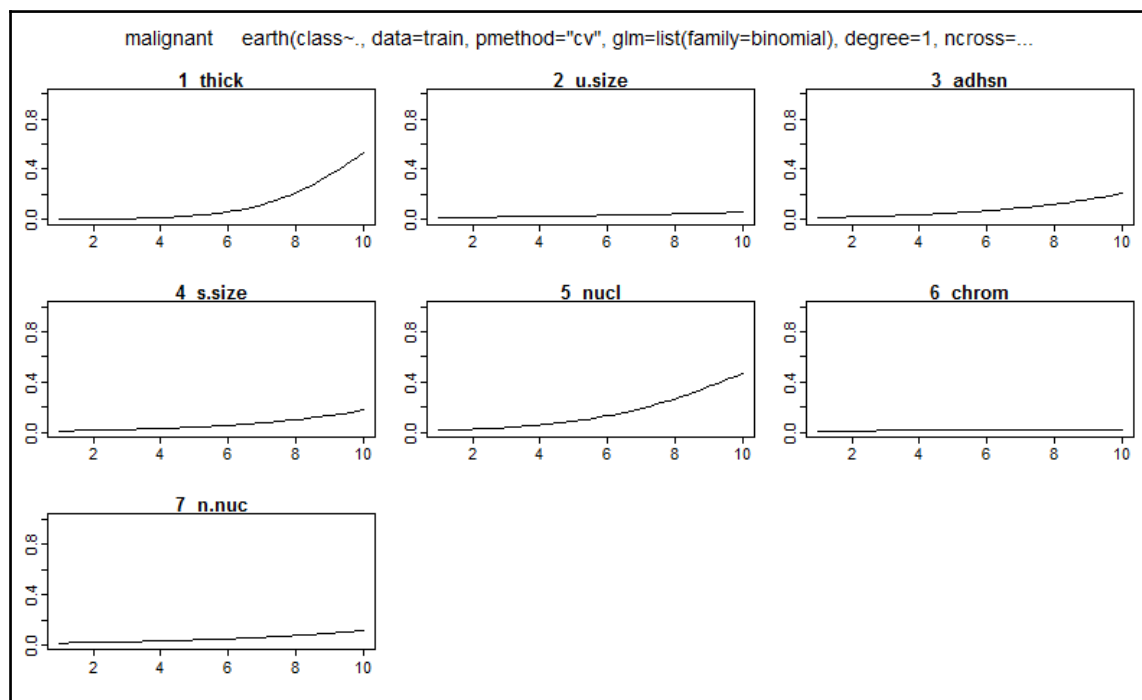
```
      malignant
(Intercept) -6.5746417
u.size 0.1502747
adhsn 0.3058496
s.size 0.3188098
nucl 0.4426061
n.nuc 0.2307595
h(thick-3) 0.7019053
h(3-chrom) -0.6927319
Earth selected 8 of 10 terms, and 7 of 9 predictors
  using pmethod="cv"
Termination condition: RSq changed by less than
  0.001 at 10 terms
Importance: nucl, u.size, thick, n.nuc, chrom,
  s.size, adhsn,
  u.shape-unused, ...
Number of terms at each degree of interaction: 1 7
(additive model)
Earth GRSq 0.8354593 RSq 0.8450554 mean.oof.RSq
  0.8331308 (sd 0.0295)
GLM null.deviance 620.9885 (473 dof) deviance
  81.90976 (466 dof)
  iters 8
pmethod="backward" would have selected the same
model:
8 terms 7 preds, GRSq 0.8354593 RSq 0.8450554
mean.oof.RSq
  0.8331308
```

The model gives us eight terms, including the Intercept and seven predictors. Two of the predictors have `hinge` functions--thickness and chromatin. If the thickness is greater than 3, the coefficient of 0.7019 is multiplied by that value; otherwise, it is 0. For chromatin, if less than 3 then the coefficient is multiplied by the values; otherwise, it is 0.

Plots are available. The first one using the `plotmo()` function produces plots showing the model's response when varying that predictor and holding the others constant. You can clearly see the hinge function at work for thickness:

```
> plotmo(earth.fit)
```

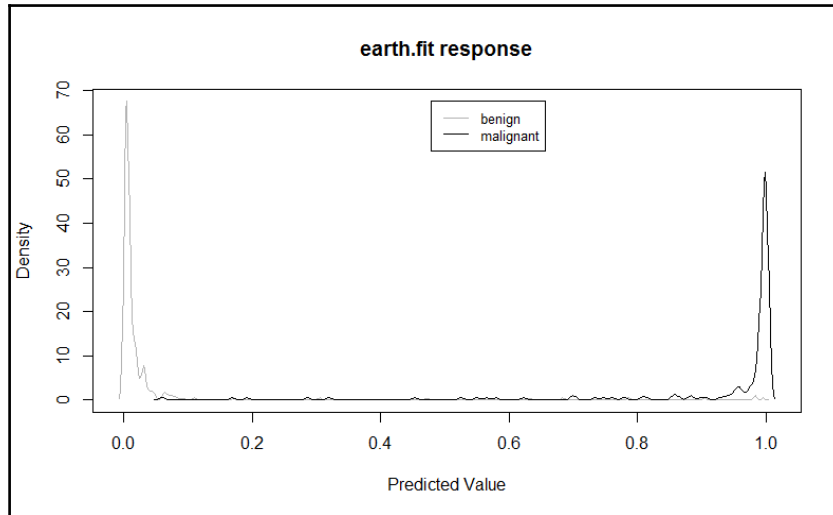
The following is the output of the preceding command:



With `plotd()`, you can see density plots of the predicted probabilities by class label:

```
> plotd(earth.fit)
```

The following is the output of the preceding command:



One can look at relative variable importance. Here we see the variable name, `nsubsets`, which is the number of model subsets that include the variable after the pruning pass, and the `gcv` and `rss` columns show the decrease in the respective value that the variable contributes (`gcv` and `rss` are scaled 0 to 100):

```
> evimp(earth.fit)
      nsubsets  gcv  rss
nucl          7 100.0 100.0
u.size         6  44.2  44.8
thick          5  23.8  25.1
n.nuc          4  15.1  16.8
chrom          3   8.3  10.7
s.size         2   6.0   8.1
adhsn          1   2.3   4.6
```

Let's see how well it did on the test dataset:

```
> test.earth.probs <- predict(earth.fit, newdata =
  test, type = "response")
> misClassError(testY, test.earth.probs)
[1] 0.0287
> confusionMatrix(testY, test.earth.probs)
      0      1
0 138      2
1   4     65
```

This is very comparable to our logistic regression models. We can now compare the models and see what our best choice would be.

Model selection

What are we to conclude from all this work? We have the confusion matrices and error rates from our models to guide us, but we can get a little more sophisticated when it comes to selecting the classification models. An effective tool for a classification model comparison is the **Receiver Operating Characteristic (ROC)** chart. Very simply, ROC is a technique for visualizing, organizing, and selecting classifiers based on their performance (Fawcett, 2006). On the ROC chart, the y-axis is the **True Positive Rate (TPR)** and the x-axis is the **False Positive Rate (FPR)**. The following are the calculations, which are quite simple:

TPR = Positives correctly classified / total positives

FPR = Negatives incorrectly classified / total negatives

Plotting the ROC results will generate a curve, and thus you are able to produce the **Area Under the Curve (AUC)**. The AUC provides you with an effective indicator of performance, and it can be shown that the AUC is equal to the probability that the observer will correctly identify the positive case when presented with a randomly chosen pair of cases in which one case is positive and one case is negative (Hanley JA & McNeil BJ, 1982). In our case, we will just switch the observer with our algorithms and evaluate accordingly.

To create an ROC chart in R, you can use the `ROCR` package. I think this is a great package and allows you to build a chart in just three lines of code. The package also has an excellent companion website (with examples and a presentation) that can be found at the following link:

<http://rocr.bioinf.mpi-sb.mpg.de/>.

What I want to show are three different plots on our ROC chart: the full model, the reduced model using BIC to select the features, the MARS model, and a bad model. This so-called bad model will include just one predictive feature and will provide an effective contrast to our other models. Therefore, let's load the `ROCR` package, build this poorly performing model, and call it `bad.fit` for simplicity, using the `thick` feature:

```
> library(ROCR)
> bad.fit <- glm(class ~ thick, family = binomial,
  data = test)
> test.bad.probs = predict(bad.fit, type =
  "response") #save
probabilities
```


It is now possible to build the ROC chart with three lines of code per model using the `test` dataset. We will first create an object that saves the predicted probabilities with the actual classification. Next, we will use this object to create another object with the calculated TPR and FPR. Then, we will build the chart with the `plot()` function. Let's get started with the model using all of the features or, as I call it, the full model. This was the initial one that we built back in the *Logistic regression model* section of this chapter:

```
> pred.full <- prediction(test.probs, test$class)
```

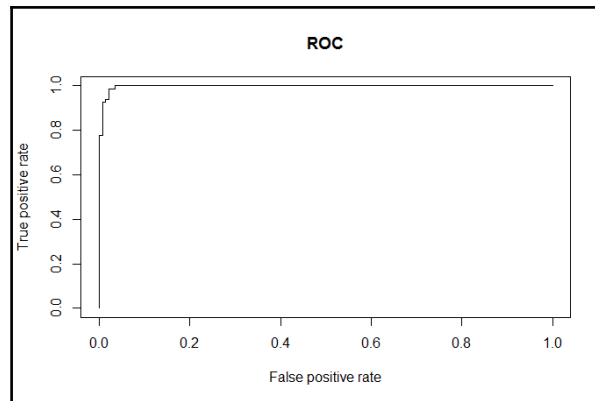
The following is the performance object with the TPR and FPR:

```
> perf.full <- performance(pred.full, "tpr", "fpr")
```

The following `plot` command with the title of ROC and `col=1` will color the line black:

```
> plot(perf.full, main = "ROC", col = 1)
```

The output of the preceding command is as follows:



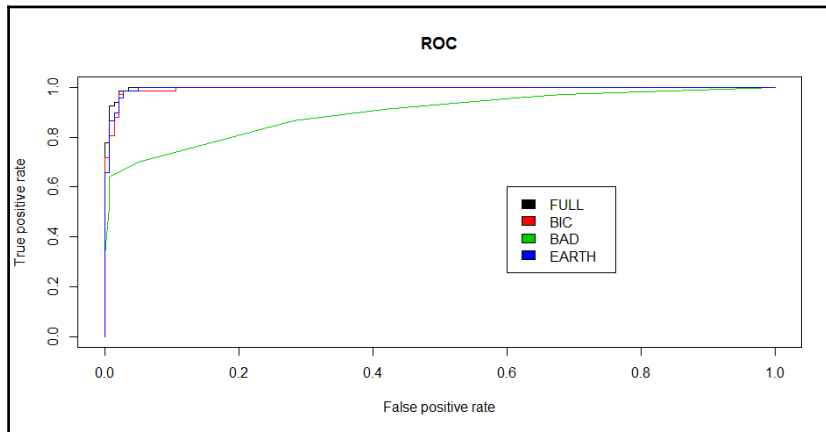
As stated previously, the curve represents TPR on the y-axis and FPR on the x-axis. If you have the perfect classifier with no false positives, then the line will run vertically at 0.0 on the x-axis. If a model is no better than chance, then the line will run diagonally from the lower left corner to the upper right one. As a reminder, the full model missed out on five labels: three false positives and two false negatives. We can now add the other models for comparison using a similar code, starting with the model built using BIC (refer to the *Logistic regression with cross-validation* section of this chapter), as follows:

```
> pred.bic <- prediction(test.bic.probs,  
  test$class)  
> perf.bic <- performance(pred.bic, "tpr", "fpr")  
> plot(perf.bic, col = 2, add = TRUE)
```

The `add=TRUE` parameter in the `plot` command added the line to the existing chart. Finally, we will add the poorly performing model, the MARS model, and include a legend chart, as follows:

```
> pred.bad <- prediction(test.bad.probs,
  test$class)
> perf.bad <- performance(pred.bad, "tpr", "fpr")
> plot(perf.bad, col = 3, add = TRUE)
  > pred.earth <- prediction(test.earth.probs,
    test$class)
  > perf.earth <- performance(pred.earth, "tpr",
    "fpr")
  > plot(perf.earth, col = 4, add = TRUE)
  > legend(0.6, 0.6, c("FULL", "BIC", "BAD",
    "EARTH"), 1:4)
```

The following is the output of the preceding code snippet:



We can see that the `FULL` model, `BIC` model and the `MARS` model are nearly superimposed. It is also quite clear that the `BAD` model performed as poorly as was expected. The final thing that we can do here is compute the AUC. This is again done in the `ROCR` package with the creation of a performance object, except that you have to substitute `auc` for `tpr` and `fpr`. The code and output are as follows:

```
> performance(pred.full, "auc")@y.values
[[1]]
[1] 0.9972672
> performance(pred.bic, "auc")@y.values
[[1]]
[1] 0.9944293
```

```
> performance(pred.bad, "auc")@y.values
[[1]]
[1] 0.8962056
> performance(pred.earth, "auc")@y.values
[[1]]
[1] 0.9952701
```

The highest AUC is for the full model at 0.997. We also see 99.4 percent for the BIC model, 89.6 percent for the bad model and 99.5 for MARS. So, to all intents and purposes, with the exception of the bad model we have no difference in predictive powers between them. What are we to do? A simple solution would be to re-randomize the `train` and `test` sets and try this analysis again, perhaps using a 60/40 split and a different randomization seed. But if we end up with a similar result, then what? I think a statistical purist would recommend selecting the most parsimonious model, while others may be more inclined to include all the variables. It comes down to trade-offs, that is, model accuracy versus interpretability, simplicity, and scalability. In this instance, it seems safe to default to the simpler model, which has the same accuracy. It goes without saying that we won't always get this level of predictability with just GLMs or discriminant analysis. We will tackle these problems in upcoming chapters with more complex techniques and hopefully improve our predictive ability. The beauty of machine learning is that there are several ways to skin the proverbial cat.

Summary

In this chapter, we looked at using probabilistic linear models to predict a qualitative response with three methods: logistic regression, discriminant analysis, and MARS. Additionally, we began the process of using ROC charts in order to explore model selection visually and statistically. We also briefly discussed the model selection and trade-offs that you need to consider. In future chapters, we will revisit the breast cancer dataset to see how more complex techniques perform.

4

Advanced Feature Selection in Linear Models

"I found that math got to be too abstract for my liking and computer science seemed concerned with little details--trying to save a microsecond or a kilobyte in a computation. In statistics I found a subject that combined the beauty of both math and computer science, using them to solve real-world problems."

This was quoted by Rob Tibshirani, Professor, Stanford University at:

https://statweb.stanford.edu/~tibs/research_page.html.

So far, we've examined the usage of linear models for both quantitative and qualitative outcomes with an emphasis on the techniques of feature selection, that is, the methods and techniques to exclude useless or unwanted predictor variables. We saw that the linear models can be quite effective in machine learning problems. However, newer techniques that have been developed and refined in the last couple of decades or so can improve predictive ability and interpretability above and beyond the linear models that we discussed in the preceding chapters. In this day and age, many datasets have numerous features in relation to the number of observations or, as it is called, high-dimensionality. If you've ever worked on a genomics problem, this will quickly become self-evident. Additionally, with the size of the data that we are being asked to work with, a technique like best subsets or stepwise feature selection can take inordinate amounts of time to converge even on high-speed computers. I'm not talking about minutes: in many cases, hours of system time are required to get a best subsets solution.

There is a better way in these cases. In this chapter, we will look at the concept of regularization where the coefficients are constrained or shrunk towards zero. There are a number of methods and permutations to these methods of regularization but we will focus on Ridge regression, **Least Absolute Shrinkage and Selection Operator (LASSO)**, and finally, elastic net, which combines the benefit of both techniques into one.

Regularization in a nutshell

You may recall that our linear model follows the form, $Y = B_0 + B_1x_1 + \dots B_nx_n + e$, and also that the best fit tries to minimize the RSS, which is the sum of the squared errors of the actual minus the estimate, or $e_1^2 + e_2^2 + \dots e_n^2$.

With regularization, we will apply what is known as **shrinkage penalty** in conjunction with the minimization RSS. This penalty consists of a lambda (symbol λ), along with the normalization of the beta coefficients and weights. How these weights are normalized differs in the techniques, and we will discuss them accordingly. Quite simply, in our model, we are minimizing $(RSS + \lambda(normalized\ coefficients))$. We will select λ , which is known as the tuning parameter, in our model building process. Please note that if lambda is equal to 0, then our model is equivalent to OLS, as it cancels out the normalization term.

So what does this do for us and why does it work? First of all, regularization methods are very computationally efficient. In best subsets, we are searching 2^p **models**, and in large datasets, it may not be feasible to attempt. In R, we are only fitting one model to each value of lambda and this is far more efficient. Another reason goes back to our bias-variance trade-off, which was discussed in the preface. In the linear model, where the relationship between the response and the predictors is close to linear, the least squares estimates will have low bias but may have high variance. This means that a small change in the training data can cause a large change in the least squares coefficient estimates (James, 2013).

Regularization through the proper selection of lambda and normalization may help you improve the model fit by optimizing the bias-variance trade-off. Finally, regularization of coefficients works to solve multi collinearity problems.

Ridge regression

Let's begin by exploring what ridge regression is and what it can and cannot do for you. With ridge regression, the normalization term is the sum of the squared weights, referred to as an **L2-norm**. Our model is trying to minimize $RSS + \lambda(\sum B_j^2)$. As lambda increases, the coefficients shrink toward zero but never become zero. The benefit may be an improved predictive accuracy, but as it does not zero out the weights for any of your features, it could lead to issues in the model's interpretation and communication. To help with this problem, we will turn to LASSO.

LASSO

LASSO applies the **L1-norm** instead of the L2-norm as in ridge regression, which is the sum of the absolute value of the feature weights and thus minimizes $RSS + \lambda(\sum |B_j|)$. This shrinkage penalty will indeed force a feature weight to zero. This is a clear advantage over ridge regression, as it may greatly improve the model interpretability.

The mathematics behind the reason that the L1-norm allows the weights/coefficients to become zero, is out of the scope of this book (refer to *Tibsharini, 1996* for further details).

If LASSO is so great, then ridge regression must be clearly obsolete. Not so fast! In a situation of high collinearity or high pairwise correlations, LASSO may force a predictive feature to zero and thus you can lose the predictive ability; that is, say if both feature A and B should be in your model, LASSO may shrink one of their coefficients to zero. The following quote sums up this issue nicely:

"One might expect the lasso to perform better in a setting where a relatively small number of predictors have substantial coefficients, and the remaining predictors have coefficients that are very small or that equal zero. Ridge regression will perform better when the response is a function of many predictors, all with coefficients of roughly equal size."

-(James, 2013)

There is the possibility of achieving the best of both the worlds and that leads us to the next topic, elastic net.

Elastic net

The power of elastic net is that, it performs the feature extraction that ridge regression does not and it will group the features that LASSO fails to do. Again, LASSO will tend to select one feature from a group of correlated ones and ignore the rest. Elastic net does this by including a mixing parameter, α , in conjunction with λ . α will be between 0 and 1 and as before, λ will regulate the size of the penalty. Please note that an α of zero is equal to ridge regression and an α of one is equivalent to LASSO. Essentially, we are blending the L1 and L2 penalties by including a second tuning parameter with a quadratic (squared) term of the beta coefficients. We will end up with the goal of minimizing $(RSS + \lambda[(1-\alpha)(\sum |B_j|^2)/2 + \alpha(\sum |B_j|)])/N$.

Let's put these techniques to test. We will primarily utilize the `leaps`, `glmnet`, and `caret` packages to select the appropriate features and thus the appropriate model in our business case.

Business case

For this chapter, we will stick to cancer--prostate cancer in this case. It is a small dataset of 97 observations and nine variables but allows you to fully grasp what is going on with regularization techniques by allowing a comparison with traditional techniques. We will start by performing best subsets regression to identify the features and use this as a baseline for our comparison.

Business understanding

The Stanford University Medical Center has provided preoperative **Prostate Specific Antigen (PSA)** data on 97 patients who are about to undergo radical prostatectomy (complete prostate removal) for the treatment of prostate cancer. The **American Cancer Society (ACS)** estimates that nearly 30,000 American men died of prostate cancer in 2014 (<http://www.cancer.org/>). PSA is a protein that is produced by the prostate gland and is found in the bloodstream. The goal is to develop a predictive model of PSA among the provided set of clinical measures. PSA can be an effective prognostic indicator, among others, of how well a patient can and should do after surgery. The patient's PSA levels are measured at various intervals after the surgery and used in various formulas to determine if a patient is cancer-free. A preoperative predictive model in conjunction with the postoperative data (not provided here) can possibly improve cancer care for thousands of men each year.

Data understanding and preparation

The data set for the 97 men is in a data frame with 10 variables, as follows:

- `lcavol`: This is the log of the cancer volume
- `lweight`: This is the log of the prostate weight
- `age`: This is the age of the patient in years
- `lbph`: This is the log of the amount of **Benign Prostatic Hyperplasia (BPH)**, which is the non-cancerous enlargement of the prostate
- `svi`: This is the seminal vesicle invasion and an indicator variable of whether or not the cancer cells have invaded the seminal vesicles outside the prostate wall (1 = yes, 0 = no)
- `lcp`: This is the log of capsular penetration and a measure of how much the cancer cells have extended in the covering of the prostate
- `gleason`: This is the patient's Gleason score; a score (2-10) provided by a pathologist after a biopsy about how abnormal the cancer cells appear--the higher the score, the more aggressive the cancer is assumed to be
- `pgg4`: This is the percent of Gleason patterns-four or five (high-grade cancer)
- `lpsa`: This is the log of the PSA; it is the response/outcome
- `train`: This is a logical vector (true or false) that signifies the training or test set

The dataset is contained in the R package `ElemStatLearn`. After loading the required packages and data frame, we can begin to explore the variables and any possible relationships, as follows:

```
> library(ElemStatLearn) #contains the data
> library(car) #package to calculate Variance Inflation Factor
> library(corrplot) #correlation plots
> library(leaps) #best subsets regression
> library(glmnet) #allows ridge regression, LASSO and elastic net
> library(caret) #parameter tuning
```

With the packages loaded, bring up the `prostate` dataset and explore its structure:

```
> data(prostate)
> str(prostate)
'data.frame': 97 obs. of 10 variables:
 $ lcavol : num -0.58 -0.994 -0.511 -1.204 0.751 ...
 $ lweight: num 2.77 3.32 2.69 3.28 3.43 ...
 $ age : int 50 58 74 58 62 50 64 58 47 63 ...
 $ lbph : num -1.39 -1.39 -1.39 -1.39 -1.39 ...
 $ svi : int 0 0 0 0 0 0 0 0 0 0 ...
```



```

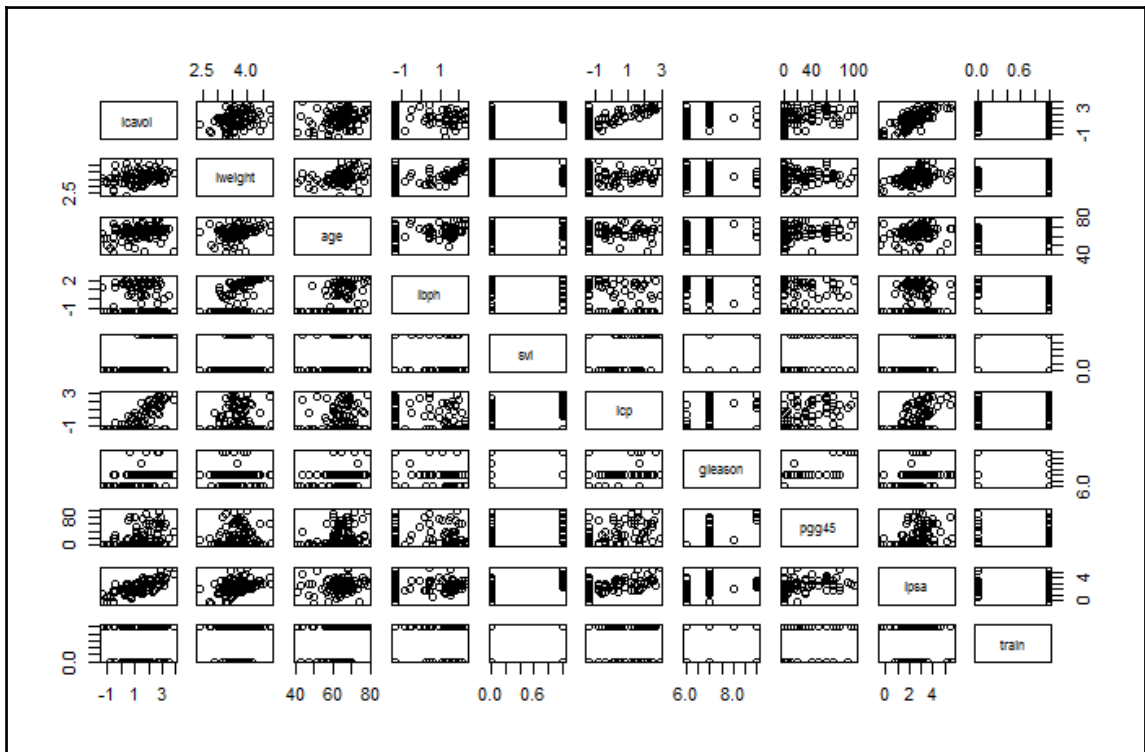
$ lcp      : num  -1.39 -1.39 -1.39 -1.39 -1.39 ...
$ gleason: int    6 6 7 6 6 6 6 6 6 6 ...
$ pgg45    : int    0 0 20 0 0 0 0 0 0 0 ...
$ lpsa     : num  -0.431 -0.163 -0.163 -0.163 0.372 ...
$ train    : logi   TRUE TRUE TRUE TRUE TRUE TRUE ...

```

The examination of the structure should raise a couple of issues that we will need to double-check. If you look at the features, `svi`, `lcp`, `gleason`, and `pgg45` have the same number in the first 10 observations, with the exception of one--the seventh observation in `gleason`. In order to make sure that these are viable as input features, we can use plots and tables so as to understand them. To begin with, use the following `plot()` command and input the entire data frame, which will create a scatterplot matrix:

```
> plot(prostate)
```

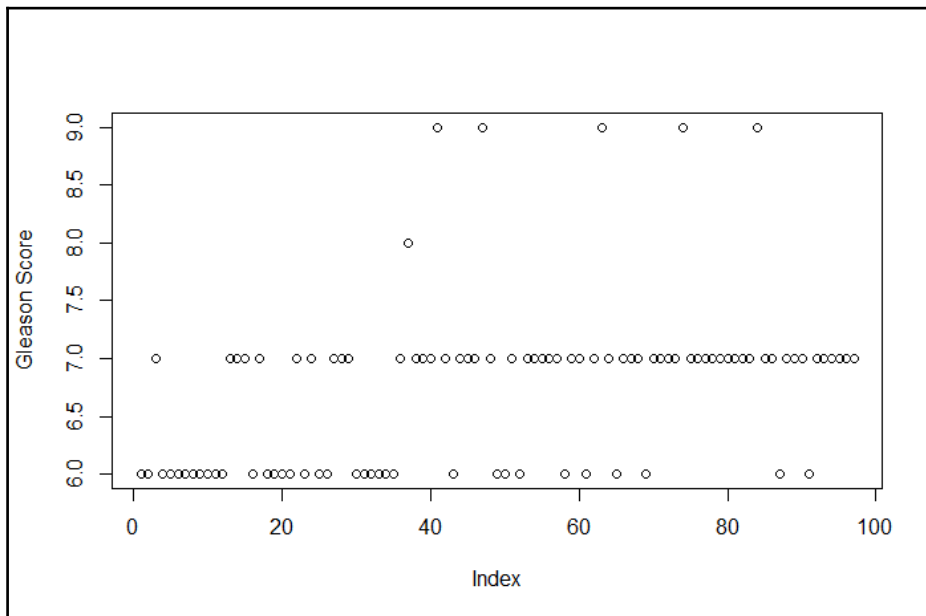
The output of the preceding command is as follows:



With these many variables on one plot, it can get a bit difficult to understand what is going on, so we will drill down further. It does look like there is a clear linear relationship between our outcomes, `lpsa`, and `lcavol`. It also appears that the features mentioned previously have an adequate dispersion and are well-balanced across what will become our train and test sets with the possible exception of the `gleason` score. Note that the `gleason` scores captured in this dataset are of four values only. If you look at the plot where `train` and `gleason` intersect, one of these values is not in either `test` or `train`. This could lead to potential problems in our analysis and may require transformation. So, let's create a plot specifically for that feature, as follows:

```
> plot(prostate$gleason)
```

The following is the output of the preceding command:



We have a problem here. Each dot represents an observation and the *x axis* is the observation number in the data frame. There is only one **Gleason Score** of 8.0 and only five of score 9.0. You can look at the exact counts by producing a table of the features:

```
> table(prostate$gleason)
 6  7  8  9 
35 56  1  5
```

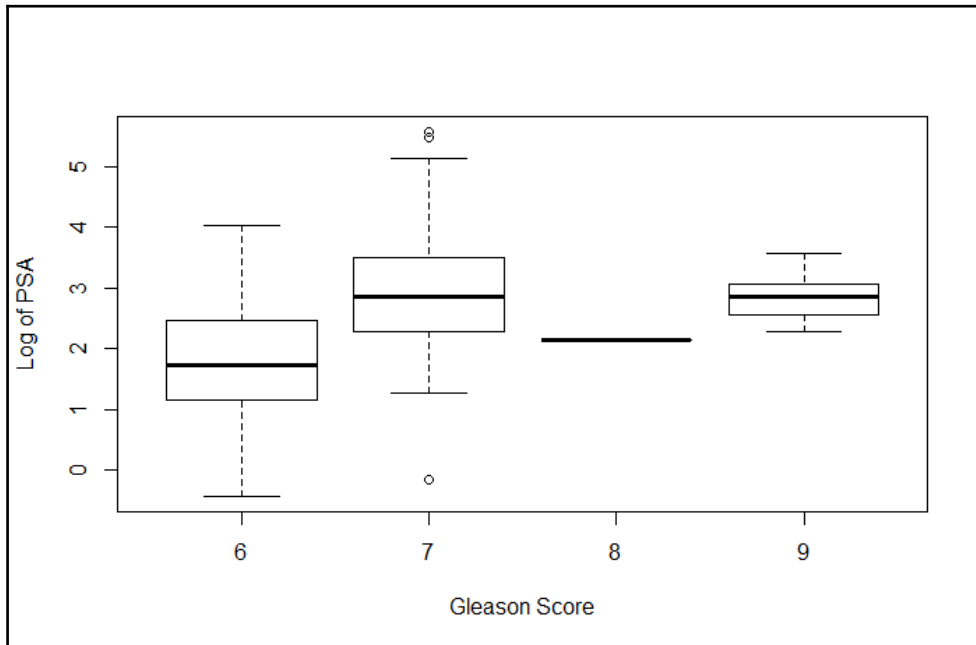
What are our options? We could do any of the following:

- Exclude the feature altogether
- Remove only the scores of 8.0 and 9.0
- Recode this feature, creating an indicator variable

I think it may help if we create a boxplot of Gleason Score versus Log of PSA. We used the `ggplot2` package to create boxplots in a prior chapter, but one can also create it with base R, as follows:

```
> boxplot(prostate$lpsa ~ prostate$gleason, xlab = "Gleason Score",  
          ylab = "Log of PSA")
```

The output of the preceding command is as follows:



Looking at the preceding plot, I think the best option will be to turn this into an indicator variable with 0 being a 6 score and 1 being a 7 or a higher score. Removing the feature may cause a loss of predictive ability. The missing values will also not work with the `glmnet` package that we will use.

You can code an indicator variable with one simple line of code using the `ifelse()` command by specifying the column in the data frame that you want to change. Then follow the logic that, if the observation is number x , then code it y , or else code it z :

```
> prostate$gleason <- ifelse(prostate$gleason == 6, 0, 1)
```

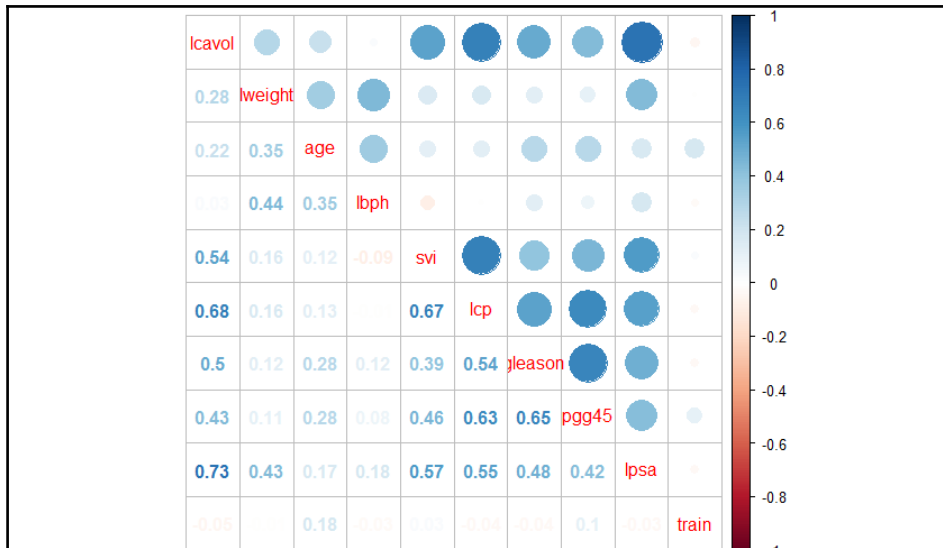
As always, let's verify that the transformation worked as intended by creating a table in the following way:

```
> table(prostate$gleason)
 0  1
35 62
```

That worked to perfection! As the scatterplot matrix was hard to read, let's move on to a correlation plot, which indicates if a relationship/dependency exists between the features. We will create a correlation object using the `cor()` function and then take advantage of the `corrplot` library with `corrplot.mixed()`, as follows:

```
> p.cor = cor(prostate)
> corrplot.mixed(p.cor)
```

The output of the preceding command is:



A couple of things jump out here. First, PSA is highly correlated with the log of cancer volume (`lcavol`); you may recall that in the scatterplot matrix, it appeared to have a highly linear relationship. Second, multicollinearity may become an issue; for example, cancer volume is also correlated with capsular penetration and this is correlated with the seminal vesicle invasion. This should be an interesting learning exercise!

Before the learning can begin, the training and testing sets must be created. As the observations are already coded as being in the `train` set or not, we can use the `subset()` command and set the observations where `train` is coded to `TRUE` as our training set and `FALSE` for our testing set. It is also important to drop `train` as we do not want that as a feature:

```
> train <- subset(prostate, train == TRUE)[, 1:9]
> str(train)
'data.frame': 67 obs. of  9 variables:
 $ lcavol : num  -0.58 -0.994 -0.511 -1.204 0.751 ...
 $ lweight: num   2.77 3.32 2.69 3.28 3.43 ...
 $ age    : int   50 58 74 58 62 50 58 65 63 63 ...
 $ lbph   : num  -1.39 -1.39 -1.39 -1.39 -1.39 ...
 $ svi    : int    0 0 0 0 0 0 0 0 0 0 ...
 $ lcp    : num  -1.39 -1.39 -1.39 -1.39 -1.39 ...
 $ gleason: num    0 0 1 0 0 0 0 0 0 1 ...
 $ pgg45  : int    0 0 20 0 0 0 0 0 0 30 ...
 $ lpsa   : num  -0.431 -0.163 -0.163 -0.163 0.372 ...
> test <- subset(prostate, train == FALSE)[, 1:9]
> str(test)
'data.frame': 30 obs. of  9 variables:
 $ lcavol : num   0.737 -0.777 0.223 1.206 2.059 ...
 $ lweight: num   3.47 3.54 3.24 3.44 3.5 ...
 $ age    : int   64 47 63 57 60 69 68 67 65 54 ...
 $ lbph   : num   0.615 -1.386 -1.386 -1.386 1.475 ...
 $ svi    : int    0 0 0 0 0 0 0 0 0 0 ...
 $ lcp    : num  -1.386 -1.386 -1.386 -0.431 1.348 ...
 $ gleason: num    0 0 0 1 1 0 0 1 0 0 ...
 $ pgg45  : int    0 0 0 5 20 0 0 20 0 0 ...
 $ lpsa   : num   0.765 1.047 1.047 1.399 1.658 ...
```

Modeling and evaluation

With the data prepared, we will begin the modeling process. For comparison purposes, we will create a model using best subsets regression like the previous two chapters and then utilize the regularization techniques.

Best subsets

The following code is, for the most part, a rehash of what we developed in [Chapter 2, Linear Regression - The Blocking and Tackling of Machine Learning](#). We will create the best subset object using the `regsubsets()` command and specify the `train` portion of data. The variables that are selected will then be used in a model on the `test` set, which we will evaluate with a mean squared error calculation.

The model that we are building is written out as `lpsa ~ .` with the tilde and period stating that we want to use all the remaining variables in our data frame, with the exception of the response:

```
> subfit <- regsubsets(lpsa ~ ., data = train)
```

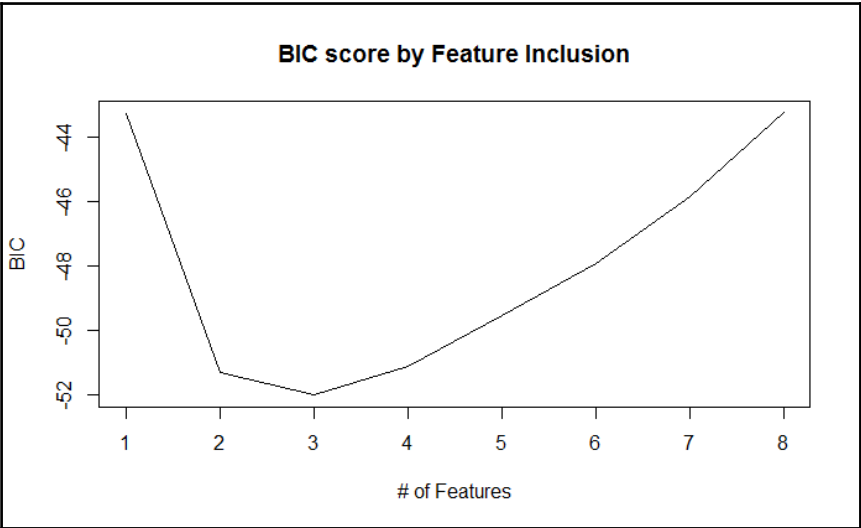
With the model built, you can produce the best subset with two lines of code. The first one turns the `summary` model into an object where we can extract the various subsets and determine the best one with the `which.min()` command. In this instance, I will use BIC, which was discussed in [Chapter 2, Linear Regression - The Blocking and Tackling of Machine Learning](#), which is as follows:

```
> b.sum <- summary(subfit)
> which.min(b.sum$bic)
[1] 3
```

The output is telling us that the model with the 3 features has the lowest `bic` value. A plot can be produced to examine the performance across the subset combinations, as follows:

```
> plot(b.sum$bic, type = "l", xlab = "# of Features", ylab = "BIC",
      main = "BIC score by Feature Inclusion")
```

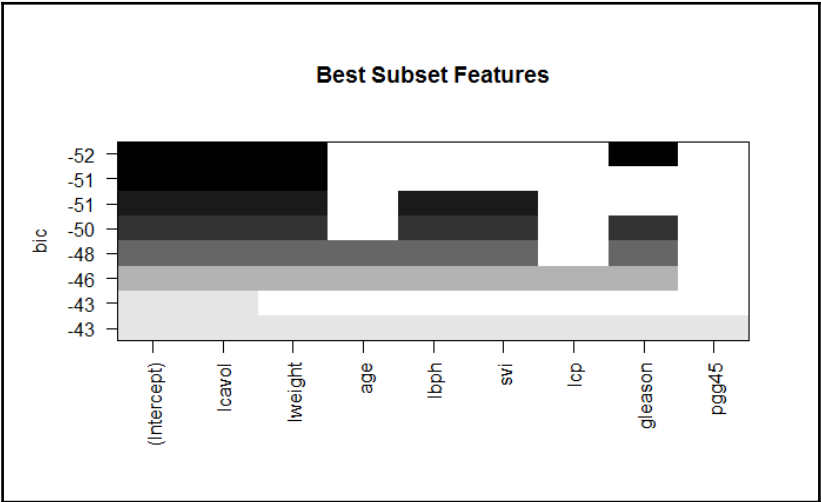
The following is the output of the preceding command:



A more detailed examination is possible by plotting the actual model object, as follows:

```
> plot(subfit, scale = "bic", main = "Best Subset Features")
```

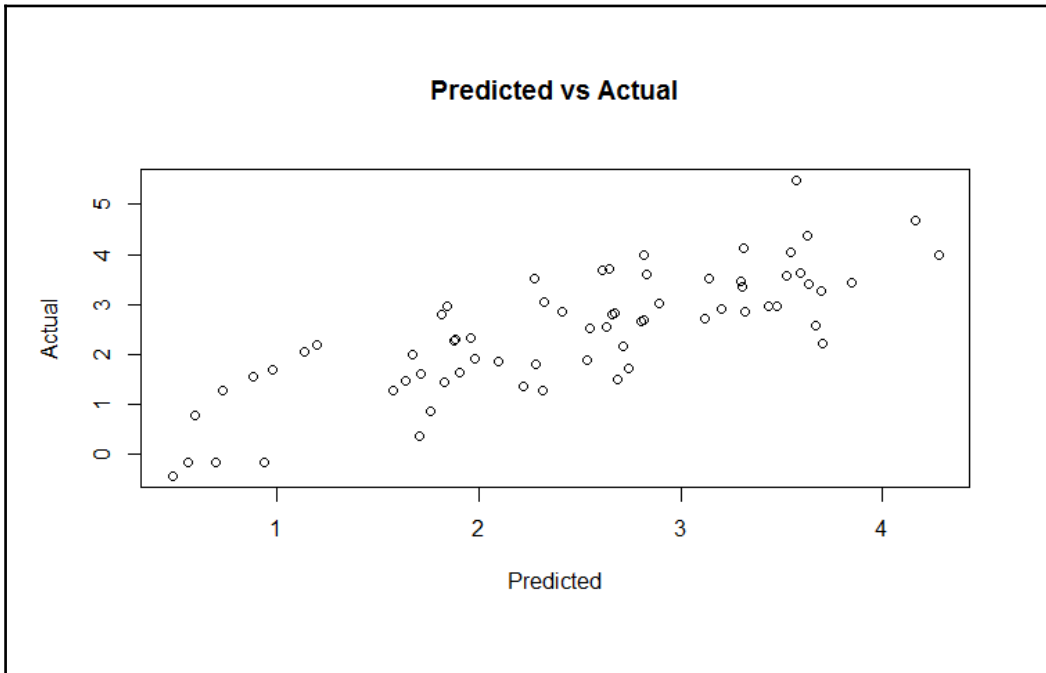
The output of the preceding command is:



So, the previous plot shows us that the three features included in the lowest BIC are `lcavol`, `lweight`, and `gleason`. It is noteworthy that `lcavol` is included in every combination of the models. This is consistent with our earlier exploration of the data. We are now ready to try this model on the `test` portion of the data, but first, we will produce a plot of the fitted values versus the actual values looking for linearity in the solution, and as a check on the constancy of the variance. A linear model will need to be created with just the three features of interest. Let's put this in an object called `ols` for the OLS. Then the fits from `ols` will be compared to the actual in the training set, as follows:

```
> ols <- lm(lpsa ~ lcavol + lweight + gleason, data = train)
> plot(ols$fitted.values, train$lpsa, xlab = "Predicted", ylab =
      "Actual", main = "Predicted vs Actual")
```

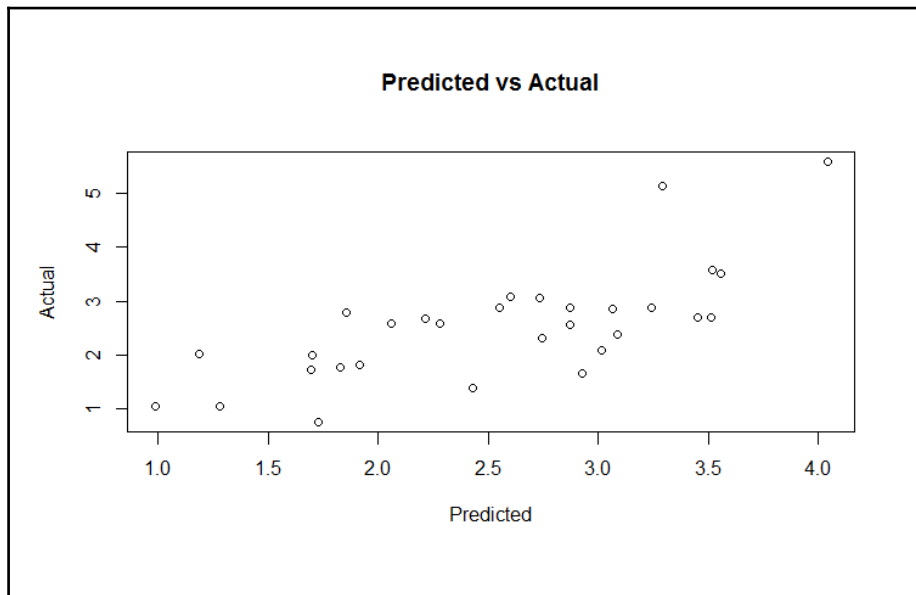
The following is the output of the preceding command:



An inspection of the plot shows that a linear fit should perform well on this data and that the non-constant variance is not a problem. With that, we can see how this performs on the test set data by utilizing the `predict()` function and specifying `newdata=test`, as follows:

```
> pred.subfit <- predict(ols, newdata = test)
> plot(pred.subfit, test$lpsa , xlab = "Predicted", ylab =
      "Actual", main = "Predicted vs Actual")
```

The values in the object can then be used to create a plot of the Predicted vs Actual values, as shown in the following image:



The plot doesn't seem to be too terrible. For the most part, it is a linear fit with the exception of what looks to be two outliers on the high end of the PSA score. Before concluding this section, we will need to calculate **Mean Squared Error (MSE)** to facilitate comparison across the various modeling techniques. This is easy enough where we will just create the residuals and then take the mean of their squared values, as follows:

```
> resid.subfit <- test$lpsa - pred.subfit
> mean(resid.subfit^2)
[1] 0.5084126
```

So, MSE of 0.508 is our benchmark for going forward.

Ridge regression

With ridge regression, we will have all the eight features in the model, so this will be an intriguing comparison with the best subsets model. The package that we will use and is in fact already loaded, is `glmnet`. The package requires that the input features are in a matrix instead of a data frame and for ridge regression, we can follow the command sequence of `glmnet(x = our input matrix, y = our response, family = the distribution, alpha=0)`. The syntax for `alpha` relates to 0 for ridge regression and 1 for doing LASSO.

To get the `train` set ready for use in `glmnet` is actually quite easy by using `as.matrix()` for the inputs and creating a vector for the response, as follows:

```
> x <- as.matrix(train[, 1:8])
> y <- train[, 9]
```

Now, run the ridge regression by placing it in an object called, appropriately I might add, `ridge`. It is important to note here that the `glmnet` package will first standardize the inputs before computing the `lambda` values and then will unstandardize the coefficients. You will need to specify the distribution of the response variable as `gaussian` as it is continuous and `alpha=0` for ridge regression, as follows:

```
> ridge <- glmnet(x, y, family = "gaussian", alpha = 0)
```

The object has all the information that we need in order to evaluate the technique. The first thing to try is the `print()` command, which will show us the number of nonzero coefficients, percent deviance explained, and correspondent value of `Lambda`. The default number in the package of steps in the algorithm is 100. However, the algorithm will stop prior to 100 steps if the percent deviation does not dramatically improve from one `lambda` to another; that is, the algorithm converges to an optimal solution. For the purpose of saving space, I will present only the following first five and last ten `lambda` results:

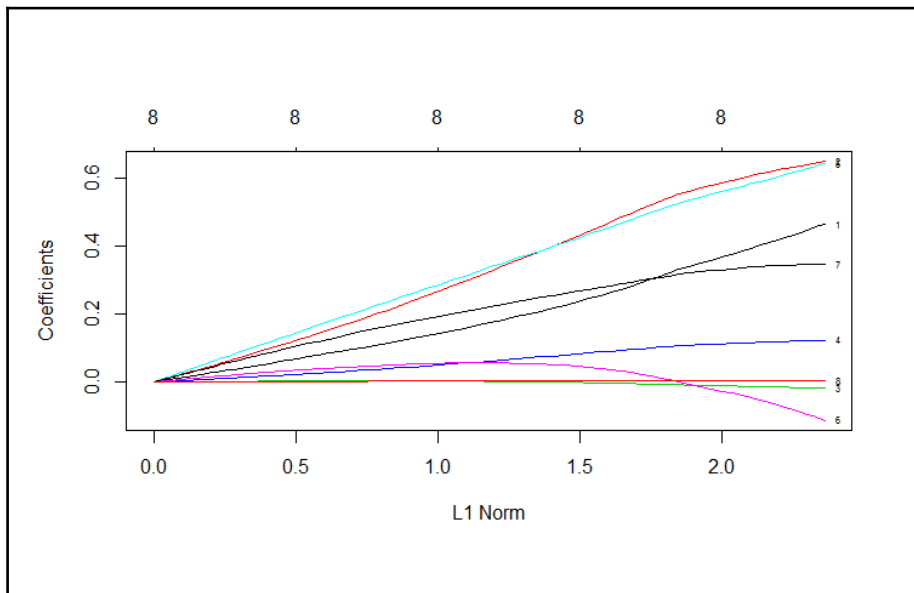
```
> print(ridge)
Call:  glmnet(x = x, y = y, family = "gaussian", alpha = 0)
      Df      %Dev      Lambda
[1,]   8 3.801e-36 878.90000
[2,]   8 5.591e-03 800.80000
[3,]   8 6.132e-03 729.70000
[4,]   8 6.725e-03 664.80000
[5,]   8 7.374e-03 605.80000
.....
[91,]  8 6.859e-01  0.20300
[92,]  8 6.877e-01  0.18500
[93,]  8 6.894e-01  0.16860
[94,]  8 6.909e-01  0.15360
```

```
[95,] 8 6.923e-01 0.13990
[96,] 8 6.935e-01 0.12750
[97,] 8 6.946e-01 0.11620
[98,] 8 6.955e-01 0.10590
[99,] 8 6.964e-01 0.09646
[100,] 8 6.971e-01 0.08789
```

Look at row 100 for example. It shows us that the number of nonzero coefficients or-said another way-the number of features included, is **eight**; please recall that it will always be the same for ridge regression. We also see that the percent of deviance explained is .6971 and the Lambda tuning parameter for this row is 0.08789. Here is where we can decide on which lambda to select for the test set. The lambda of 0.08789 can be used, but let's make it a little simpler, and for the test set, try 0.10. A couple of plots might help here so let's start with the package's default, adding annotations to the curve by adding `label=TRUE` in the following syntax:

```
> plot(ridge, label = TRUE)
```

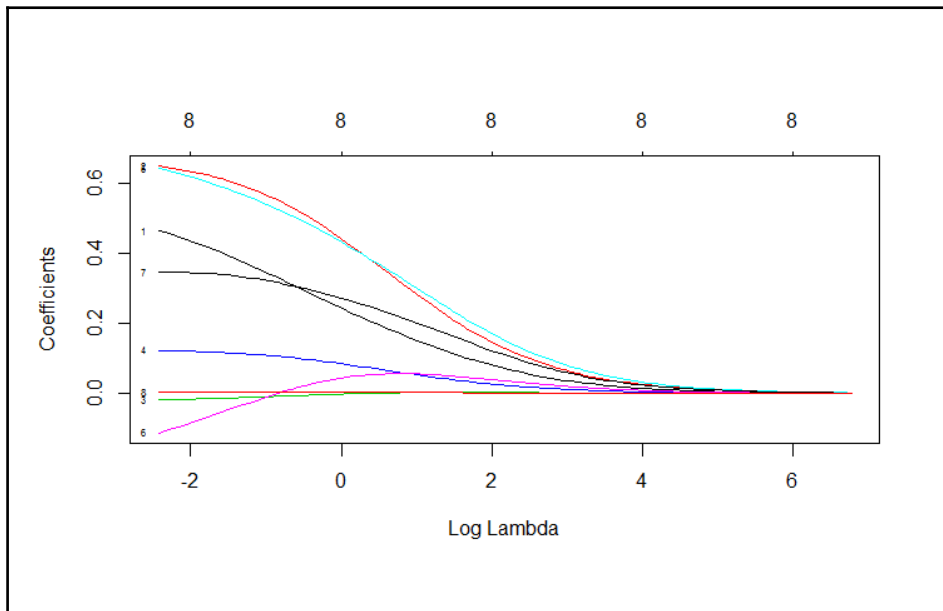
The following is the output of the preceding command:



In the default plot, the y axis is the value of **Coefficients** and the x axis is **L1 Norm**. The plot tells us the coefficient values versus the **L1 Norm**. The top of the plot contains a second x axis, which equates to the number of features in the model. Perhaps a better way to view this is by looking at the coefficient values changing as `lambda` changes. We just need to tweak the code in the following `plot()` command by adding `xvar="lambda"`. The other option is the percent of deviance explained by substituting `lambda` with `dev`:

```
> plot(ridge, xvar = "lambda", label = TRUE)
```

The output of the preceding command is as follows:



This is a worthwhile plot as it shows that as `lambda` decreases, the shrinkage parameter decreases and the absolute values of the coefficients increase. To see the coefficients at a specific `lambda` value, use the `coef()` command. Here, we will specify the `lambda` value that we want to use by specifying `s=0.1`. We will also state that we want `exact=TRUE`, which tells `glmnet` to fit a model with that specific `lambda` value versus interpolating from the values on either side of our `lambda`, as follows:

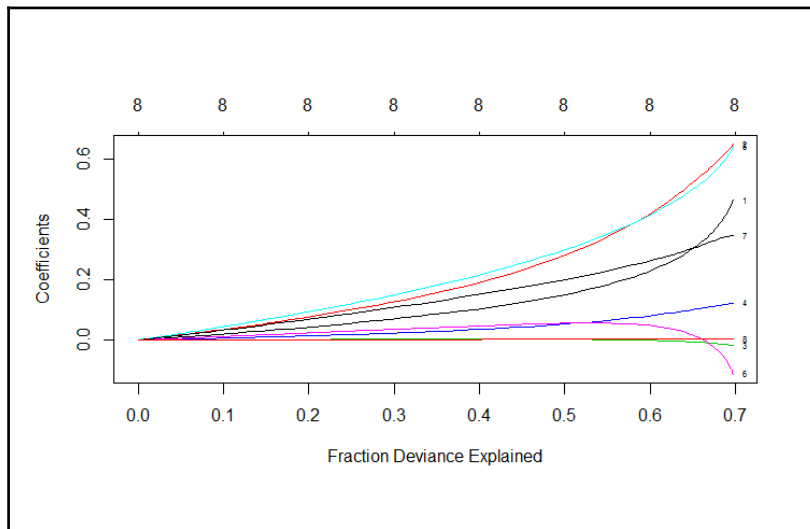
```
> ridge.coef <- coef(ridge, s = 0.1, exact = TRUE)
> ridge.coef
9 x 1 sparse Matrix of class "dgCMatrix"
      1
(Intercept) 0.13062197
```

lcavol	0.45721270
lweight	0.64579061
age	-0.01735672
lbph	0.12249920
svi	0.63664815
lcp	-0.10463486
gleason	0.34612690
pgg45	0.00428580

It is important to note that `age`, `lcp`, and `pgg45` are close to, but not quite, zero. Let's not forget to plot deviance versus coefficients as well:

```
> plot(ridge, xvar = "dev", label = TRUE)
```

The output of the preceding command is as follows:



Comparing the two previous plots, we can see that as `lambda` decreases, the coefficients increase and the percent/fraction of the deviance explained increases. If we were to set `lambda` equal to zero, we would have no shrinkage penalty and our model would equate the OLS.

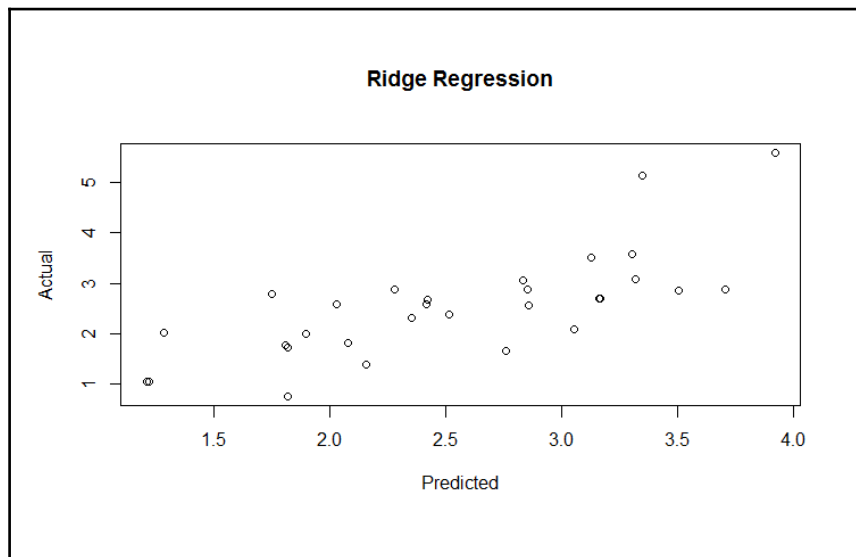
To prove this on the `test` set, we will have to transform the features as we did for the training data:

```
> newx <- as.matrix(test[, 1:8])
```

Then, we use the `predict` function to create an object that we will call `ridge.y` with `type = "response"` and our `lambda` equal to `0.10` and plot the Predicted values versus the Actual values, as follows:

```
> ridge.y <- predict(ridge, newx = newx, type = "response", s =  
  0.1)  
> plot(ridge.y, test$lpsa, xlab = "Predicted", ylab = "Actual", main =  
  "Ridge Regression")
```

The output of the following command is as follows:



The plot of Predicted versus Actual of Ridge Regression seems to be quite similar to best subsets, complete with two interesting outliers at the high end of the PSA measurements. In the real world, it would be advisable to explore these outliers further so as to understand whether they are truly unusual or we are missing something. This is where domain expertise would be invaluable. The MSE comparison to the benchmark may tell a different story. We first calculate the residuals, and then take the mean of those residuals squared:

```
> ridge.resid <- ridge.y - test$lpsa  
> mean(ridge.resid^2)  
[1] 0.4789913
```

Ridge regression has given us a slightly better MSE. It is now time to put LASSO to the test to see if we can decrease our errors even further.

LASSO

To run LASSO next is quite simple and we only have to change one number from our ridge regression model: that is, change $\alpha=0$ to $\alpha=1$ in the `glmnet()` syntax. Let's run this code and also see the output of the model, looking at the first five and last 10 results:

```
> lasso <- glmnet(x, y, family = "gaussian", alpha = 1)
> print(lasso)
Call: glmnet(x = x, y = y, family = "gaussian", alpha = 1)
Df %Dev Lambda
[1,] 0 0.00000 0.878900
[2,] 1 0.09126 0.800800
[3,] 1 0.16700 0.729700
[4,] 1 0.22990 0.664800
[5,] 1 0.28220 0.605800
.....
[60,] 8 0.70170 0.003632
[61,] 8 0.70170 0.003309
[62,] 8 0.70170 0.003015
[63,] 8 0.70170 0.002747
[64,] 8 0.70180 0.002503
[65,] 8 0.70180 0.002281
[66,] 8 0.70180 0.002078
[67,] 8 0.70180 0.001893
[68,] 8 0.70180 0.001725
[69,] 8 0.70180 0.001572
```

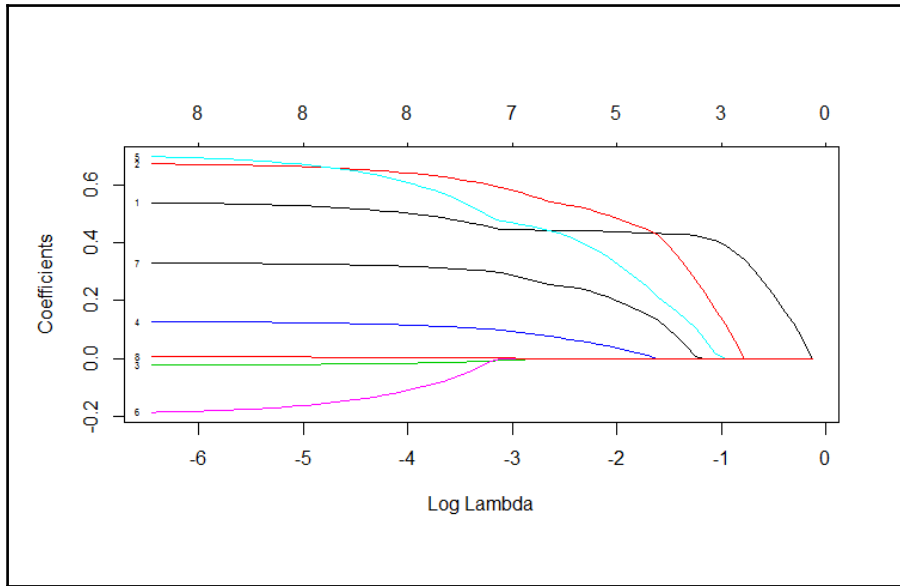
Note that the model building process stopped at step 69 as the deviance explained no longer improved as `lambda` decreased. Also, note that the `Df` column now changes along with `lambda`. At first glance, here it seems that all the eight features should be in the model with a `lambda` of 0.001572. However, let's try and find and test a model with fewer features, around seven, for argument's sake. Looking at the rows, we see that around a `lambda` of 0.045, we end up with 7 features versus 8. Thus, we will plug this `lambda` in for our test set evaluation, as follows:

```
[31,] 7 0.67240 0.053930
[32,] 7 0.67460 0.049140
[33,] 7 0.67650 0.044770
[34,] 8 0.67970 0.040790
[35,] 8 0.68340 0.037170
```

Just as with ridge regression, we can plot the results:

```
> plot(lasso, xvar = "lambda", label = TRUE)
```

The following is the output of the preceding command:



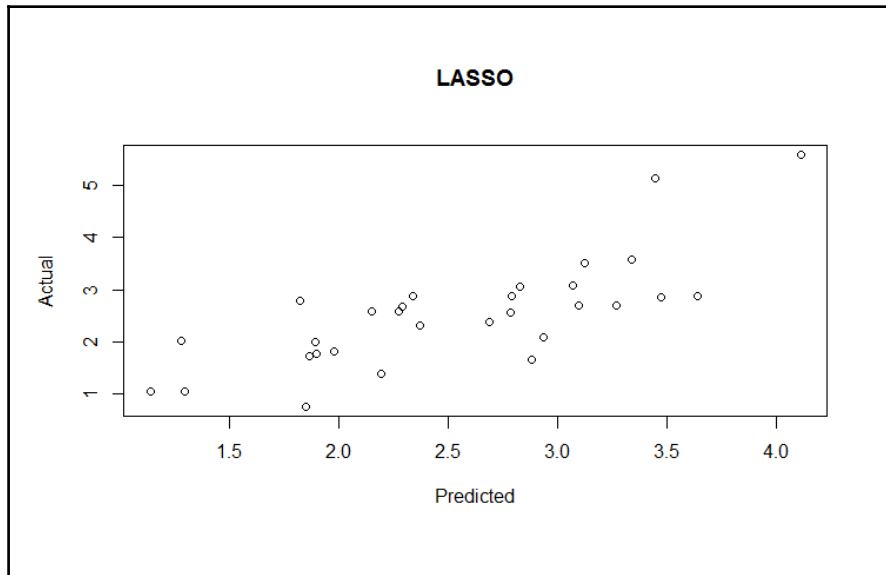
This is an interesting plot and really shows how LASSO works. Notice how the lines labeled 8, 3, and 6 behave, which corresponds to the `pgg45`, `age`, and `lcp` features respectively. It looks as if `lcp` is at or near zero until it is the last feature that is added. We can see the coefficient values of the seven feature model just as we did with ridge regression by plugging it into `coef()`, as follows:

```
> lasso.coef <- coef(lasso, s = 0.045, exact = TRUE)
> lasso.coef
9 x 1 sparse Matrix of class "dgCMatrix"
      1
(Intercept) -0.1305852115
lcavol      0.4479676523
lweight     0.5910362316
age        -0.0073156274
lbph       0.0974129976
svi        0.4746795823
lcp         .
gleason     0.2968395802
pgg45      0.0009790322
```


The LASSO algorithm zeroed out the coefficient for `lcp` at a `lambda` of 0.045. Here is how it performs on the `test` data:

```
> lasso.y <- predict(lasso, newx = newx, type = "response", s =  
  0.045)  
> plot(lasso.y, test$lpsa, xlab = "Predicted", ylab = "Actual",  
  main = "LASSO")
```

The output of the preceding command is as follows:



We calculate MSE as we did before:

```
> lasso.resid <- lasso.y - test$lpsa  
> mean(lasso.resid^2)  
[1] 0.4437209
```

It looks like we have similar plots as before, with only the slightest improvement in MSE. Our last best hope for dramatic improvement is with elastic net. To this end, we will still use the `glmnet` package. The twist will be that, we will solve for `lambda` and for the elastic net parameter known as `alpha`. Recall that `alpha = 0` is the ridge regression penalty and `alpha = 1` is the LASSO penalty. The elastic net parameter will be $0 \leq \alpha \leq 1$. Solving for two different parameters simultaneously can be complicated and frustrating, but we can use our friend in R, the `caret` package, for assistance.

Elastic net

The `caret` package stands for classification and regression training. It has an excellent companion website to help in understanding all of its capabilities:

<http://topepo.github.io/caret/index.html>. The package has many different functions that you can use and we will revisit some of them in the later chapters. For our purpose here, we want to focus on finding the optimal mix of `lambda` and our elastic net mixing parameter, `alpha`. This is done using the following simple three-step process:

1. Use the `expand.grid()` function in base R to create a vector of all the possible combinations of `alpha` and `lambda` that we want to investigate.
2. Use the `trainControl()` function from the `caret` package to determine the resampling method; we will use LOOCV as we did in Chapter 2, *Linear Regression - The Blocking and Tackling of Machine Learning*.
3. Train a model to select our `alpha` and `lambda` parameters using `glmnet()` in `caret`'s `train()` function.

Once we've selected our parameters, we will apply them to the `test` data in the same way as we did with ridge regression and LASSO.



Our grid of combinations should be large enough to capture the best model but not too large that it becomes computationally unfeasible. That won't be a problem with this size dataset, but keep this in mind for future references.

Here are the values of hyperparameters we can try:

- Alpha from 0 to 1 by 0.2 increments; remember that this is bound by 0 and 1
- Lambda from 0.00 to 0.2 in steps of 0.02; the 0.2 lambda should provide a cushion from what we found in ridge regression (`lambda=0.1`) and LASSO (`lambda=0.045`)

You can create this vector by using the `expand.grid()` function and building a sequence of numbers for what the `caret` package will automatically use. The `caret` package will take the values for `alpha` and `lambda` with the following code:

```
> grid <- expand.grid(.alpha = seq(0, 1, by = .2), .lambda =  
  seq(0.00, 0.2, by = 0.02))
```

The `table()` function will show us the complete set of 66 combinations:

```
> table(grid)
      .lambda
.alpha 0 0.02 0.04 0.06 0.08 0.1 0.12 0.14 0.16 0.18 0.2
0      1      1      1      1      1      1      1      1      1      1
0.2    1      1      1      1      1      1      1      1      1      1
0.4    1      1      1      1      1      1      1      1      1      1
0.6    1      1      1      1      1      1      1      1      1      1
0.8    1      1      1      1      1      1      1      1      1      1
1      1      1      1      1      1      1      1      1      1      1
```

We can confirm that this is what we wanted--alpha from 0 to 1 and lambda from 0 to 0.2.



For the resampling method, we will put in the code for `LOOCV` for the method. There are also other resampling alternatives such as bootstrapping or k-fold cross-validation and numerous options that you can use with `trainControl()`, but we will explore these options in future chapters.

You can tell the model selection criteria with `selectionFunction()` in `trainControl()`. For quantitative responses, the algorithm will select based on its default of **Root Mean Square Error (RMSE)**, which is perfect for our purposes:

```
> control <- trainControl(method = "LOOCV")
```

It is now time to use `train()` to determine the optimal elastic net parameters. The function is similar to `lm()`. We will just add the syntax: `method="glmnet", trControl=control` and `tuneGrid=grid`. Let's put this in an object called `enet.train`:

```
> enet.train <- train(lpsa ~ ., data = train, method = "glmnet",
  trControl = control, tuneGrid = grid)
```

Calling the object will tell us the parameters that lead to the lowest RMSE, as follows:

```
> enet.train
glmnet
67 samples
8 predictor
No pre-processing
Resampling:
Summary of sample sizes: 66, 66, 66, 66, 66, 66, ...
Resampling results across tuning parameters:
  alpha  lambda  RMSE  Rsquared
0.0     0.00    0.750  0.609
0.0     0.02    0.750  0.609
0.0     0.04    0.750  0.609
```

0.0	0.06	0.750	0.609
0.0	0.08	0.750	0.609
0.0	0.10	0.751	0.608
.....			
1.0	0.14	0.800	0.564
1.0	0.16	0.809	0.558
1.0	0.18	0.819	0.552
1.0	0.20	0.826	0.549

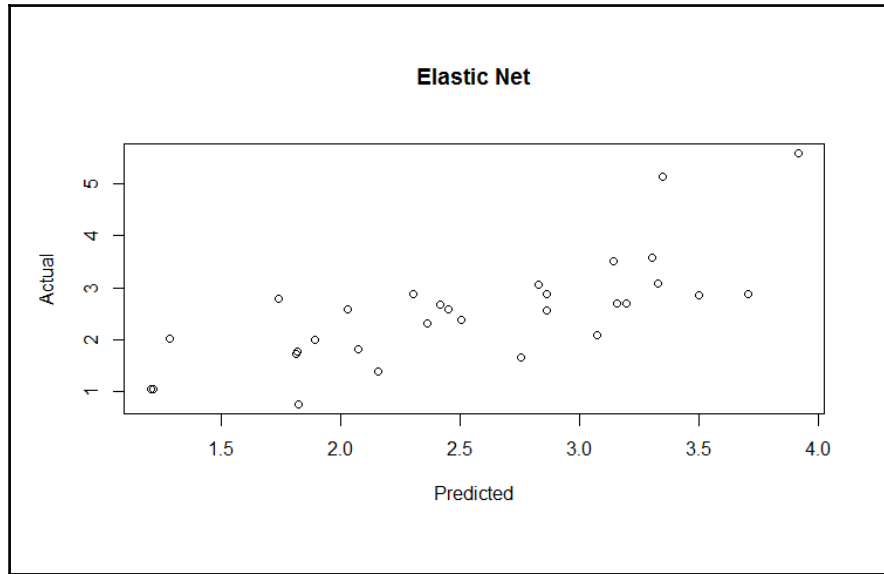
RMSE was used to select the optimal model using the smallest value. The final values used for the model were $\alpha = 0$ and $\lambda = 0.08$.

This experimental design has led to the optimal tuning parameters of $\alpha = 0$ and $\lambda = 0.08$, which is a ridge regression with $s = 0.08$ in `glmnet`, recall that we used 0.10. The R-squared is 61 percent, which is nothing to write home about.

The process for the test set validation is just as before:

```
> enet <- glmnet(x, y, family = "gaussian", alpha = 0, lambda =
.08)
> enet.coef <- coef(enet, s = .08, exact = TRUE)
> enet.coef
9 x 1 sparse Matrix of class "dgCMatrix"
      1
(Intercept) 0.137811097
lcavol      0.470960525
lweight     0.652088157
age         -0.018257308
lbph        0.123608113
svi         0.648209192
lcp         -0.118214386
gleason     0.345480799
pgg45       0.004478267
> enet.y <- predict(enet, newx=newx, type="response", s=.08)
> plot(enet.y, test$lpsa, xlab="Predicted", ylab="Actual",
      main="Elastic Net")
```

The output of the preceding command is as follows:



Calculate MSE as we did before:

```
> enet.resid <- enet.y - test$lpsa
> mean(enet.resid^2)
[1] 0.4795019
```

This model error is similar to the ridge penalty. On the `test` set, our LASSO model did the best in terms of errors. We may be over-fitting! Our best subset model with three features is the easiest to explain, and in terms of errors, is acceptable to the other techniques. We can use cross-validation in the `glmnet` package to possibly identify a better solution.

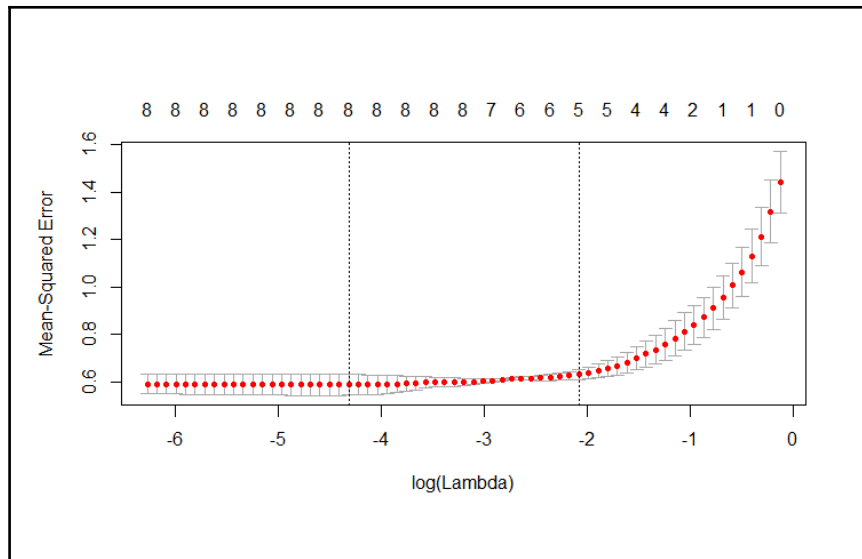
Cross-validation with glmnet

We have used LOOCV with the `caret` package; now we will try k-fold cross-validation. The `glmnet` package defaults to ten folds when estimating `lambda` in `cv.glmnet()`. In k-fold CV, the data is partitioned into an equal number of subsets (folds) and a separate model is built on each k-1 set and then tested on the corresponding holdout set with the results combined (averaged) to determine the final parameters.

In this method, each fold is used as a test set only once. The `glmnet` package makes it very easy to try this and will provide you with an output of the `lambda` values and the corresponding MSE. It defaults to `alpha = 1`, so if you want to try ridge regression or an elastic net mix, you will need to specify it. As we will be trying for as few input features as possible, we will stick to the defaults, but given the size of the training data, use only three folds:

```
> set.seed(317)
> lasso.cv = cv.glmnet(x, y, nfolds = 3)
> plot(lasso.cv)
```

The output of the preceding code is as follows:



The plot for CV is quite different than the other `glmnet` plots, showing **log(Lambda)** versus **Mean-Squared Error** along with the number of features. The two dotted vertical lines signify the minimum of MSE (left line) and one standard error from the minimum (right line). One standard error away from the minimum is a good place to start if you have an over-fitting problem. You can also call the exact values of these two lambdas, as follows:

```
> lasso.cv$lambda.min #minimum
[1] 0.0133582
> lasso.cv$lambda.1se #one standard error away
[1] 0.124579
```

Using `lambda.1se`, we can go through the following process of viewing the coefficients and validating the model on the test data:

```
> coef(lasso.cv, s = "lambda.1se")
9 x 1 sparse Matrix of class "dgCMatrix"
1
(Intercept) -0.13543760
lcavol 0.43892533
lweight 0.49550944
age .
lbph 0.04343678
svi 0.34985691
lcp .
gleason 0.21225934
pgg45 .

> lasso.y.cv = predict(lasso.cv, newx=newx, type = "response",
  s = "lambda.1se")

> lasso.cv.resid = lasso.y.cv - test$lpsa

> mean(lasso.cv.resid^2)
[1] 0.4465453
```

This model achieves an error of 0.45 with just five features, zeroing out age, lcp, and pgg45.

Model selection

We looked at five different models in examining this dataset. The following points were the test set error of these models:

- Best subsets is 0.51
- Ridge regression is 0.48
- LASSO is 0.44
- Elastic net is 0.48
- LASSO with CV is 0.45

On a pure error, LASSO with seven features performed the best. However, does this best address the question that we are trying to answer? Perhaps the more parsimonious model that we found using CV with a lambda of ~ 0.125 is more appropriate. My inclination is to put forth the latter as it is more interpretable.

Having said all this, there is clearly a need for domain-specific knowledge from oncologists, urologists, and pathologists in order to understand what would make the most sense. There is that, but there is also the need for more data. With this sample size, the results can vary greatly just by changing the randomization seeds or creating different `train` and `test` sets (try it and see for yourself.) At the end of the day, these results may likely raise more questions than provide you with answers. But is this bad? I would say no, unless you made the critical mistake of over-promising at the start of the project about what you will be able to provide. This is a fair warning to prudently apply the tools put forth in Chapter 1, *A Process for Success*.

Regularization and classification

The regularization techniques applied above will also work for classification problems, both binomial and multinomial. Therefore, let's not conclude this chapter until we apply some sample code on a logistic regression problem, specifically the breast cancer data from the prior chapter. As in regression with a quantitative response, this can be an important technique to utilize data sets with high dimensionality.

Logistic regression example

Recall that, in the breast cancer data we analyzed, the probability of a tumor being malignant can be denoted as follows in a logistic function:

$$P(\text{malignant}) = 1 / 1 + e^{-(B_0 + B_1X_1 + B_nX_n)}$$

Since we have a linear component in the function, L1 and L2 regularization can be applied. To demonstrate this, let's load and prepare the breast cancer data like we did in the previous chapter:

```
> library(MASS)
> biopsy$ID = NULL
> names(biopsy) = c("thick", "u.size", "u.shape", "adhsn",
  "s.size", "nucl", "chrom", "n.nuc", "mit", "class")
> biopsy.v2 <- na.omit(biopsy)
> set.seed(123)
> ind <- sample(2, nrow(biopsy.v2), replace = TRUE, prob = c(0.7,
```



```
0.3)
> train <- biopsy.v2[ind==1, ]
> test <- biopsy.v2[ind==2, ]
```

Transform the data to an input matrix and the labels:

```
> x <- as.matrix(train[, 1:9])
> y <- train[, 10]
```

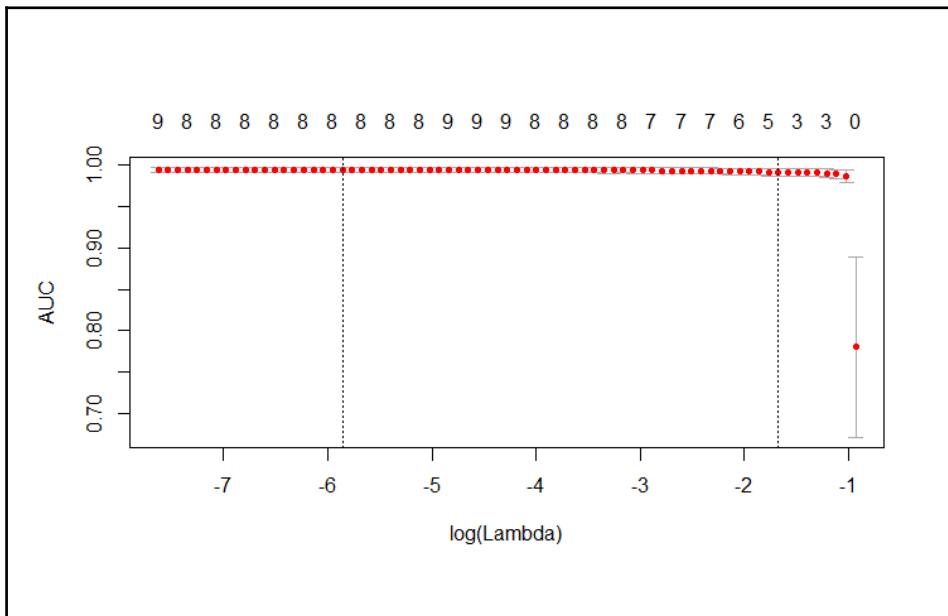
In the function `cv.glmnet`, we will change the family to binomial and the measure to area under the curve, along with five folds:

```
> set.seed(3)
> fitCV <- cv.glmnet(x, y, family = "binomial",
  type.measure = "auc",
  nfolds = 5)
```

Plotting `fitCV` gives us the AUC by lambda:

```
> plot(fitCV)
```

The output from the plot command is as shown:



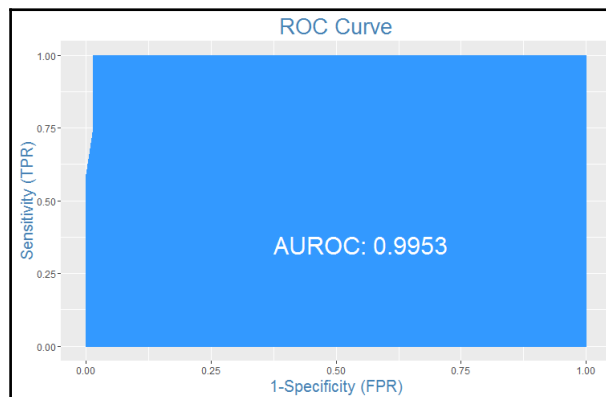
Interesting! Notice the immediate improvement to AUC by adding just one feature. Let's just have a look at the coefficients for one standard error:

```
> fitCV$lambda.1se
[1] 0.1876892
> coef(fitCV, s = "lambda.1se")
10 x 1 sparse Matrix of class "dgCMatrix"
 1
(Intercept) -1.84478214
thick 0.01892397
u.size 0.10102690
u.shape 0.08264828
adhsn .
s.size .
nucl 0.13891750
chrom .
n.nuc .
mit .
```

Here, we see that the four features selected are `thickness`, `u.size`, `u.shape`, and `nucl`. Like we did in the prior chapter, let's look at how it performs on the test set in terms of error and `auc`:

```
> library(InformationValue)
> predCV <- predict(fitCV, newx = as.matrix(test[, 1:9]),
  s = "lambda.1se",
  type = "response")
actuals <- ifelse(test$class == "malignant", 1, 0)
misClassError(actuals, predCV)
[1] 0.0622
> plotROC(actuals, predCV)
```

Output from the previous code:



The results show that it performed comparable to the logistic regression done previously. It doesn't look like using `lambda.1se` was optimal and we should see if we can improve the output of the sample prediction using `lambda.min`:

```
> predCV.min <- predict(fitCV, newx = as.matrix(test[, 1:9]),
  s = "lambda.min",
  type = "response")
> misClassError(actuals, predCV.min)
[1] 0.0239
```

There you have it! An error rate as good as what we did in Chapter 3, *Logistic Regression and Discriminant Analysis*.

Summary

In this chapter, the goal was to use a small dataset to provide an introduction to practically apply an advanced feature selection for linear models. The outcome for our data was quantitative, but the `glmnet` package we used also supports qualitative outcomes (binomial and multinomial classifications). An introduction to regularization and the three techniques that incorporate it were provided and utilized to build and compare models. Regularization is a powerful technique to improve computational efficiency and to possibly extract more meaningful features when compared to the other modeling techniques. Additionally, we started to use the `caret` package to optimize multiple parameters when training a model. Up to this point, we've been purely talking about linear models. In the next couple of chapters, we will begin to use nonlinear models for both classification and regression problems.

5

More Classification Techniques - K-Nearest Neighbors and Support Vector Machines

"Statistical thinking will one day be as necessary for efficient citizenship as the ability to read and write."

- H.G. Wells

In Chapter 3, *Logistic Regression and Discriminant Analysis*, we discussed using logistic regression to determine the probability that a predicted observation belongs to a categorical response what we refer to as a classification problem. Logistic regression was just the beginning of classification methods, with a number of techniques that we can use to improve our predictions.

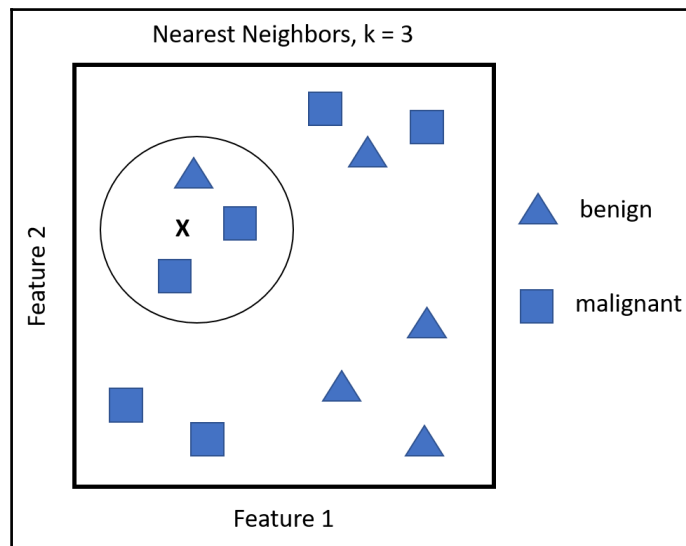
In this chapter, we will delve into two nonlinear techniques: **K-Nearest Neighbors (KNN)** and **Support Vector Machines (SVM)**. These techniques are more sophisticated than what we've discussed earlier because the assumptions on linearity can be relaxed, which means a linear combination of the features in order to define the decision boundary is not needed. Be forewarned though, that this does not always equal superior predictive ability. Additionally, these models can be a bit problematic to interpret for business partners and they can be computationally inefficient. When used wisely, they provide a powerful complement to the other tools and techniques discussed in this book. They can be used for continuous outcomes in addition to classification problems; however, for the purposes of this chapter, we will focus only on the latter.

After a high-level background on the techniques, we will lay out the business case and then put both of them to the test in order to determine the best method of the two, starting with KNN.

K-nearest neighbors

In our previous efforts, we built models that had coefficients or, said another way, parameter estimates for each of our included features. With KNN, we have no parameters as the learning method is the so-called instance-based learning. In short, *The labeled examples (inputs and corresponding output labels) are stored and no action is taken until a new input pattern demands an output value.* (Battiti and Brunato, 2014, p. 11). This method is commonly called **lazy learning**, as no specific model parameters are produced. The `train` instances themselves represent the knowledge. For the prediction of any new instance (a new data point), the `train` data is searched for an instance that most resembles the new instance in question. KNN does this for a classification problem by looking at the closest points—the nearest neighbors to determine the proper class. The k comes into play by determining how many neighbors should be examined by the algorithm, so if $k=5$, it will examine the five nearest points. A weakness of this method is that all five points are given equal weight in the algorithm even if they are less relevant in learning. We will look at the methods using R and try to alleviate this issue.

The best way to understand how this works is with a simple visual example of a binary classification learning problem. In the following figure, we have a plot of whether a tumor is **benign** or **malignant** based on two predictive features. The **X** in the plot indicates a new observation that we would like to predict. If our algorithm considers $K=3$, the circle encompasses the three observations that are nearest to the one that we want to score. As the most commonly occurring classifications are **malignant**, the **X** data point is classified as **malignant**, as shown in the following figure:



Even from this simple example, it is clear that the selection of k for the nearest neighbors is critical. If k is too small, you may have a high variance on the test set observations even though you have a low bias. On the other hand, as k grows, you may decrease your variance but the bias may be unacceptable. Cross-validation is necessary to determine the proper k .

It is also important to point out the calculation of the distance or the nearness of the data points in our feature space. The default distance is **Euclidean Distance**. This is simply the straight-line distance from point A to point B-as the crow flies-or you can utilize the formula that it is equivalent to the square root of the sum of the squared differences between the corresponding points. The formula for Euclidean Distance, given point A and B with coordinates $p_1, p_2, \dots, p_n$ and $q_1, q_2, \dots, q_n$ respectively, would be as follows:

$$\text{Euclidean Distance (A, B)} = \sqrt{\sum_{i=1}^n (p_i - q_i)^2}$$

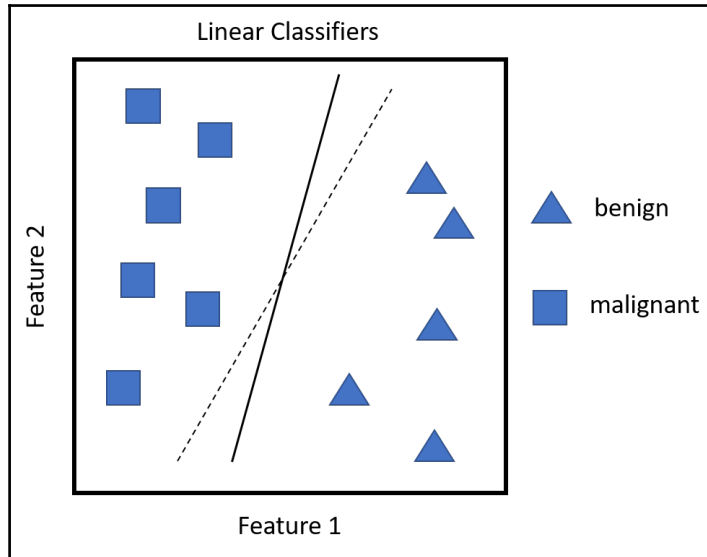
This distance is highly dependent on the scale that the features were measured on, so it is critical to standardize them. Other distance calculations as well as weights, can be used depending on the distance. We will explore this in the upcoming example.

Support vector machines

The first time I heard of support vector machines, I have to admit that I was scratching my head, thinking that this was some form of an academic obfuscation or inside joke. However, my open-minded review of SVM has replaced this natural skepticism with a healthy respect for the technique.

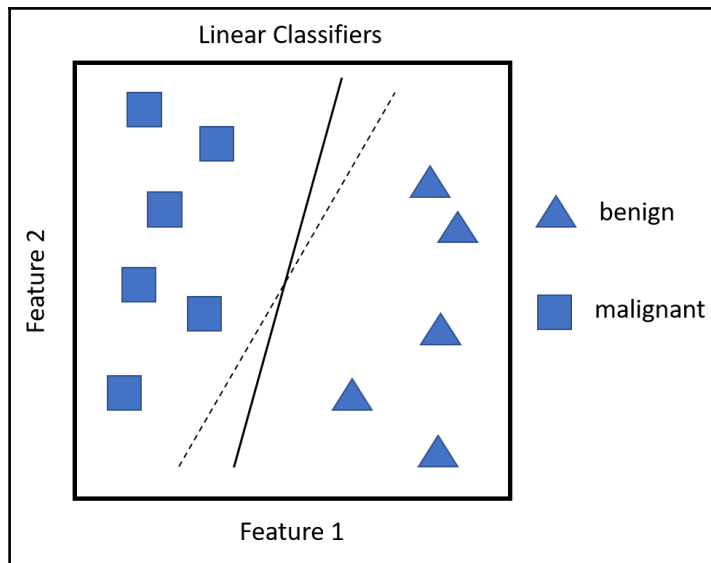
SVMs have been shown to perform well in a variety of settings and are often considered one of the best "out-of-the-box" classifiers (James, G., 2013). To get a practical grasp of the subject, let's look at another simple visual example. In the following figure, you will see that the classification task is linearly separable. However, the dotted line and solid line are just two among an infinite number of possible linear solutions.

You would have separating hyperplanes in a problem that has more than two dimensions:



So many solutions can be problematic for generalization because whatever solution you choose, any new observation to the right of the line will be classified as **benign**, and to the left of the line, it will be classified as **malignant**. Therefore, either line has no bias on the `train` data but may have a widely divergent error on any data to test. This is where the support vectors come into play. The probability that a point falls on the wrong side of the linear separator is higher for the dotted line than the solid line, which means that the solid line has a higher margin of safety for classification. Therefore, as Battiti and Brunato say, *SVMs are linear separators with the largest possible margin and the support vectors the ones touching the safety margin region on both sides.*

The following figure illustrates this idea. The thin solid line is the optimal linear separator to create the aforementioned largest possible margin, thus increasing the probability that a new observation will fall on the correct side of the separator. The thicker black lines correspond to the safety margin, and the shaded data points constitute the support vectors. If the support vectors were to move, then the margin and, subsequently, the decision boundary would change. The distance between the separators is known as the **margin**:



This is all fine and dandy, but the real-world problems are not so clear cut.



In data that is not linearly separable, many observations will fall on the wrong side of the margin (the so-called slack variables), which is a misclassification. The key to building an SVM algorithm is to solve for the optimal number of support vectors via cross-validation. Any observation that lies directly on the wrong side of the margin for its class is known as a **support vector**.

If the tuning parameter for the number of errors is too large, which means that you have many support vectors, you will suffer from a high bias and low variance. On the other hand, if the tuning parameter is too small, the opposite might occur. According to James et al., who refer to the tuning parameter as C , as C decreases, the tolerance for observations being on the wrong side of the margin decreases and the margin narrows. This C , or rather, the cost function, simply allows for observations to be on the wrong side of the margin. If C were set to zero, then we would prohibit a solution where any observation violates the margin.

Another important aspect of SVM is the ability to model nonlinearity with quadratic or higher order polynomials of the input features. In SVMs, this is known as the **kernel trick**. These can be estimated and selected with cross-validation. In the example, we will look at the alternatives.

As with any model, you can expand the number of features using polynomials to various degrees, interaction terms, or other derivations. In large datasets, the possibilities can quickly get out of control. The kernel trick with SVMs allows us to efficiently expand the feature space, with the goal that you achieve an approximate linear separation.

To check out how this is done, first look at the SVM optimization problem and its constraints. We are trying to achieve the following:

- Create weights that maximize the margin
- Subject to the constraints, no (or as few as possible) data points should lie within that margin

Now, unlike linear regression, where each observation is multiplied by a weight, in SVM, the weights are applied to the inner products of just the support vector observations.

What does this mean? Well, an inner product for two vectors is just the sum of the paired observations' product. For example, if vector one is 3, 4, and 2 and vector two is 1, 2, and 3, then you end up with $(3 \times 1) + (4 \times 2) + (2 \times 3)$ or 17. With SVMs, if we take a possibility that an inner product of each observation has an inner product of every other observation, this amounts to the formula that there would be $n(n-1)/2$ combinations, where n is the number of observations. With just 10 observations, we end up with 45 inner products. However, SVM only concerns itself with the support vectors' observations and their corresponding weights. For a linear SVM classifier, the formula is the following:

$$f(x) = \beta_0 + \sum_{i=1}^n \alpha(x, x_i)$$

Here, (x, x_i) are the inner products of the support vectors, as α is non-zero only when an observation is a support vector.

This leads to far fewer terms in the classification algorithm and allows the use of the `kernel` function, commonly referred to as the kernel trick.

The trick in this is that the `kernel` function mathematically summarizes the transformation of the features in higher dimensions instead of creating them explicitly. In a simplistic sense, a kernel function computes a dot product between two vectors. This has the benefit of creating the higher dimensional, nonlinear space, and decision boundary while keeping the optimization problem computationally efficient. The `kernel` functions compute the inner product in a higher dimensional space without transforming them into the higher dimensional space.

The notation for popular kernels is expressed as the inner (dot) product of the features, with x_i and x_j representing vectors, gamma, and c parameters, as follows:

- linear, no transformation, $K(x_i, x_j) = x_i \cdot x_j$
- polynomial, where d = degree of polynomial, $K(x_i, x_j) = (\gamma x_i \cdot x_j + c)^d$
- radial basis function, $K(x_i, x_j) = e(-\gamma |x_i - x_j|^2)$
- sigmoid function, $K(x_i, x_j) = \tanh(\gamma x_i \cdot x_j + c)$

As for the selection of the nonlinear techniques, they require some trial and error, but we will walk through the various selection techniques.

Business case

In the upcoming case study, we will apply KNN and SVM to the same dataset. This will allow us to compare the R code and learning methods on the same problem, starting with KNN. We will also spend some time drilling down into the confusion matrix, comparing a number of statistics to evaluate model accuracy.

Business understanding

The data that we will examine was originally collected by the **National Institute of Diabetes and Digestive and Kidney Diseases (NIDDK)**. It consists of 532 observations and eight input features along with a binary outcome (Yes/No). The patients in this study were of Pima Indian descent from South Central Arizona. The NIDDK data shows that for the last 30 years, research has helped scientists to prove that obesity is a major risk factor in the development of diabetes. The Pima Indians were selected for the study as one-half of the adult Pima Indians have diabetes and 95 per cent of those with diabetes are overweight. The analysis will focus on adult women only. Diabetes was diagnosed according to the WHO criteria and was of the type of diabetes that is known as **type 2**. In this type of diabetes, the pancreas is still able to function and produce insulin and it used to be referred to as non-insulin-dependent diabetes.

Our task is to examine and predict those individuals that have diabetes or the risk factors that could lead to diabetes in this population. Diabetes has become an epidemic in the USA, given the relatively sedentary lifestyle and high-caloric diet. According to the **American Diabetes Association (ADA)**, the disease was the seventh leading cause of death in the USA in 2010, despite being underdiagnosed. Diabetes is also associated with a dramatic increase in comorbidities, such as hypertension, dyslipidemia, stroke, eye diseases, and kidney diseases. The costs of diabetes and its complications are enormous. The ADA estimates that the total cost of the disease in 2012 was approximately \$490 billion. For further background information on the problem, refer to ADA's website at <http://www.diabetes.org/diabetes-basics/statistics/>.

Data understanding and preparation

The dataset for the 532 women is in two separate data frames. The variables of interest are as follows:

- `npreg`: This is the number of pregnancies
- `glu`: This is the plasma glucose concentration in an oral glucose tolerance test
- `bp`: This is the diastolic blood pressure (mm Hg)
- `skin`: This is triceps skin-fold thickness measured in mm
- `bmi`: This is the body mass index
- `ped`: This is the diabetes pedigree function
- `age`: This is the age in years
- `type`: This is diabetic, Yes or No

The datasets are contained in the R package, `MASS`. One data frame is named `Pima.tr` and the other is named `Pima.te`. Instead of using these as separate `train` and `test` sets, we will combine them and create our own in order to discover how to do such a task in R.

To begin, let's load the following packages that we will need for the exercise:

```
> library(class) #k-nearest neighbors
> library(kknn) #weighted k-nearest neighbors
> library(e1071) #SVM
> library(caret) #select tuning parameters
> library(MASS) # contains the data
> library(reshape2) #assist in creating boxplots
> library(ggplot2) #create boxplots
> library(kernlab) #assist with SVM feature selection
```

We will now load the datasets and check their structure, ensuring that they are the same, starting with `Pima.tr`, as follows:

```
> data(Pima.tr)
> str(Pima.tr)
'data.frame':200 obs. of  8 variables:
 $ npreg: int  5 7 5 0 0 5 3 1 3 2 ...
 $ glu  : int  86 195 77 165 107 97 83 193 142 128 ...
 $ bp   : int  68 70 82 76 60 76 58 50 80 78 ...
 $ skin : int  28 33 41 43 25 27 31 16 15 37 ...
 $ bmi  : num  30.2 25.1 35.8 47.9 26.4 35.6 34.3 25.9 32.4 43.3
 ...
 $ ped  : num  0.364 0.163 0.156 0.259 0.133 ...
 $ age  : int  24 55 35 26 23 52 25 24 63 31 ...
 $ type : Factor w/ 2 levels "No","Yes": 1 2 1 1 1 2 1 1 1 2 ...
> data(Pima.te)
> str(Pima.te)
'data.frame':332 obs. of  8 variables:
 $ npreg: int  6 1 1 3 2 5 0 1 3 9 ...
 $ glu  : int  148 85 89 78 197 166 118 103 126 119 ...
 $ bp   : int  72 66 66 50 70 72 84 30 88 80 ...
 $ skin : int  35 29 23 32 45 19 47 38 41 35 ...
 $ bmi  : num  33.6 26.6 28.1 31 30.5 25.8 45.8 43.3 39.3 29 ...
 $ ped  : num  0.627 0.351 0.167 0.248 0.158 0.587 0.551 0.183
           0.704 0.263 ...
 $ age  : int  50 31 21 26 53 51 31 33 27 29 ...
 $ type : Factor w/ 2 levels "No","Yes": 2 1 1 2 2 2 2 1 1 2 ...
```

Looking at the structures, we can be confident that we can combine the data frames into one. This is very easy to do using the `rbind()` function, which stands for row binding and appends the data. If you had the same observations in each frame and wanted to append the features, you would bind them by columns using the `cbind()` function. You will simply name your new data frame and use this syntax: `new data = rbind(data frame1, data frame2)`. Our code thus becomes the following:

```
> pima <- rbind(Pima.tr, Pima.te)
```

As always, double-check the structure. We can see that there are no issues:

```
> str(pima)
'data.frame':532 obs. of  8 variables:
 $ npreg: int  5 7 5 0 0 5 3 1 3 2 ...
 $ glu  : int  86 195 77 165 107 97 83 193 142 128 ...
 $ bp   : int  68 70 82 76 60 76 58 50 80 78 ...
 $ skin : int  28 33 41 43 25 27 31 16 15 37 ...
 $ bmi  : num  30.2 25.1 35.8 47.9 26.4 35.6 34.3 25.9 32.4 43.3
 ...
```

```
$ ped : num  0.364 0.163 0.156 0.259 0.133 ...  
$ age : int   24 55 35 26 23 52 25 24 63 31 ...  
$ type : Factor w/ 2 levels "No","Yes": 1 2 1 1 1 2 1 1 1 2 ...
```

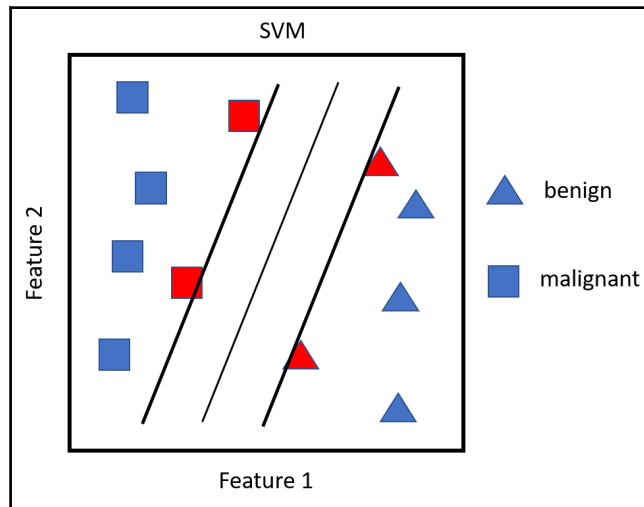
Let's do some exploratory analysis by putting this in boxplots. For this, we want to use the outcome variable, "type", as our ID variable. As we did with logistic regression, the `melt()` function will do this and prepare a data frame that we can use for the boxplots. We will call the new data frame `pima.melt`, as follows:

```
> pima.melt <- melt(pima, id.var = "type")
```

The boxplot layout using the `ggplot2` package is quite effective, so we will use it. In the `ggplot()` function, we will specify the data to use, the `x` and `y` variables, and what type of plot, and create a series of plots with two columns. In the following code, we will put the response variable as `x` and its value as `y` in `aes()`. Then, `geom_boxplot()` creates the boxplots. Finally, we will build the boxplots in two columns with `facet_wrap()`:

```
> ggplot(data = pima.melt, aes(x = type, y = value)) +  
  geom_boxplot() + facet_wrap(~ variable, ncol = 2)
```

The following is the output of the preceding command:



This is an interesting plot because it is difficult to discern any dramatic differences in the plots, probably with the exception of **glucose (glu)**. As you may have suspected, the fasting glucose appears to be significantly higher in the patients currently diagnosed with diabetes. The main problem here is that the plots are all on the same y-axis scale. We can fix this and produce a more meaningful plot by standardizing the values and then re-plotting. R has a built-in function, `scale()`, which will convert the values to a mean of zero and a standard deviation of one. Let's put this in a new data frame called `pima.scale`, converting all of the features and leaving out the `type` response. Additionally, while doing KNN, it is important to have the features on the same scale with a mean of zero and a standard deviation of one. If not, then the distance calculations in the nearest neighbor calculation are flawed. If something is measured on a scale of 1 to 100, it will have a larger effect than another feature that is measured on a scale of 1 to 10. Note that when you scale a data frame, it automatically becomes a matrix. Using the `data.frame()` function, convert it back to a data frame, as follows:

```
> pima.scale <- data.frame(scale(pima[, -8]))
> str(pima.scale)
'data.frame': 532 obs. of  7 variables:
 $ npreg: num  0.448 1.052 0.448 -1.062 -1.062 ...
 $ glu   : num -1.13 2.386 -1.42 1.418 -0.453 ...
 $ bp    : num -0.285 -0.122 0.852 0.365 -0.935 ...
 $ skin  : num -0.112 0.363 1.123 1.313 -0.397 ...
 $ bmi   : num -0.391 -1.132 0.423 2.181 -0.943 ...
 $ ped   : num -0.403 -0.987 -1.007 -0.708 -1.074 ...
 $ age   : num -0.708 2.173 0.315 -0.522 -0.801 ...
```

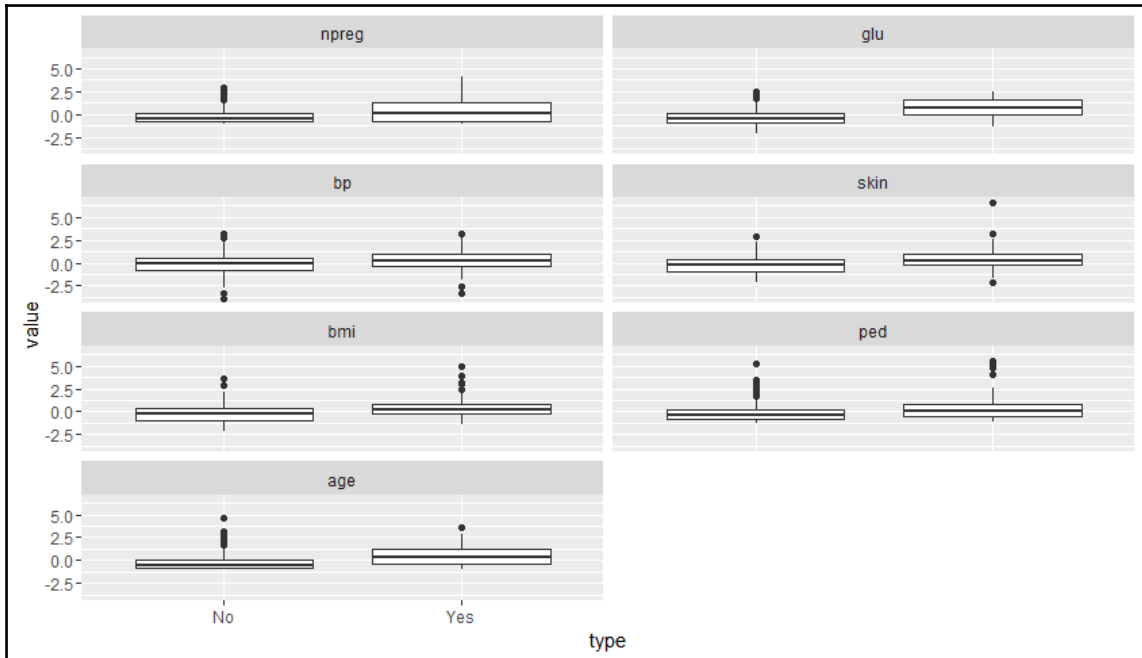
Now, we will need to include the response in the data frame, as follows:

```
> pima.scale$type <- pima$type
```

Let's just repeat the boxplotting process again with `melt()` and `ggplot()`:

```
> pima.scale.melt <- melt(pima.scale, id.var = "type")
> ggplot(data = pima.scale.melt, aes(x = type, y = value)) +
  geom_boxplot() + facet_wrap(~ variable, ncol = 2)
```

The following is the output of the preceding command:



With the features scaled, the plot is easier to read. In addition to glucose, it appears that the other features may differ by `type`, in particular, `age`.

Before splitting this into `train` and `test` sets, let's have a look at the correlation with the R function, `cor()`. This will produce a matrix instead of a plot of the Pearson correlations:

```
> cor(pima.scale[-8])
      npreg      glu      bp      skin
npreg 1.00000000 0.1253296 0.204663421 0.09508511
glu   0.125329647 1.0000000 0.219177950 0.22659042
bp    0.204663421 0.2191779 1.000000000 0.22607244
skin  0.095085114 0.2265904 0.226072440 1.00000000
bmi   0.008576282 0.2470793 0.307356904 0.64742239
ped   0.007435104 0.1658174 0.008047249 0.11863557
age   0.640746866 0.2789071 0.346938723 0.16133614
      bmi      ped      age
npreg 0.008576282 0.007435104 0.64074687
glu   0.247079294 0.165817411 0.27890711
bp    0.307356904 0.008047249 0.34693872
skin  0.647422386 0.118635569 0.16133614
bmi   1.000000000 0.151107136 0.07343826
```

```
ped    0.151107136 1.000000000 0.07165413
age    0.073438257 0.071654133 1.000000000
```

There are a couple of correlations to point out: `npreg/age` and `skin/bmi`. Multicollinearity is generally not a problem with these methods, assuming that they are properly trained and the hyperparameters are tuned.

I think we are now ready to create the `train` and `test` sets, but before we do so, I recommend that you always check the ratio of `Yes` and `No` in our response. It is important to make sure that you will have a balanced split in the data, which may be a problem if one of the outcomes is sparse. This can cause a bias in a classifier between the majority and minority classes. There is no hard and fast rule on what is an improper balance. A good rule of thumb is that you strive for at least a 2:1 ratio in the possible outcomes (He and Wa, 2013):

```
> table(pima.scale$type)
No Yes
355 177
```

The ratio is 2:1 so we can create the `train` and `test` sets with our usual syntax using a 70/30 split in the following way:

```
> set.seed(502)
> ind <- sample(2, nrow(pima.scale), replace = TRUE, prob = c(0.7,
  0.3))
> train <- pima.scale[ind == 1, ]
> test <- pima.scale[ind == 2, ]
> str(train)
'data.frame': 385 obs. of  8 variables:
 $ npreg: num  0.448 0.448 -0.156 -0.76 -0.156 ...
 $ glu  : num  -1.42 -0.775 -1.227 2.322 0.676 ...
 $ bp   : num   0.852 0.365 -1.097 -1.747 0.69 ...
 $ skin : num   1.123 -0.207 0.173 -1.253 -1.348 ...
 $ bmi  : num   0.4229 0.3938 0.2049 -1.0159 -0.0712 ...
 $ ped  : num  -1.007 -0.363 -0.485 0.441 -0.879 ...
 $ age  : num   0.315 1.894 -0.615 -0.708 2.916 ...
 $ type : Factor w/ 2 levels "No","Yes": 1 2 1 1 1 2 2 1 1 1 ...
> str(test)
'data.frame': 147 obs. of  8 variables:
 $ npreg: num  0.448 1.052 -1.062 -1.062 -0.458 ...
 $ glu  : num  -1.13 2.386 1.418 -0.453 0.225 ...
 $ bp   : num  -0.285 -0.122 0.365 -0.935 0.528 ...
 $ skin : num  -0.112 0.363 1.313 -0.397 0.743 ...
 $ bmi  : num  -0.391 -1.132 2.181 -0.943 1.513 ...
 $ ped  : num  -0.403 -0.987 -0.708 -1.074 2.093 ...
 $ age  : num  -0.7076 2.173 -0.5217 -0.8005 -0.0571 ...
 $ type : Factor w/ 2 levels "No","Yes": 1 2 1 1 2 1 2 1 1 1 ...
```


All seems to be in order, so we can move on to building our predictive models and evaluating them, starting with KNN.

Modeling and evaluation

Now we will discuss various aspects pertaining to modeling and evaluation.

KNN modeling

As previously mentioned, it is critical to select the most appropriate parameter (k or K) when using this technique. Let's put the `caret` package to good use again in order to identify k . We will create a grid of inputs for the experiment, with k ranging from 2 to 20 by an increment of 1. This is easily done with the `expand.grid()` and `seq()` functions. The `caret` package parameter that works with the KNN function is simply `.k`:

```
> grid1 <- expand.grid(.k = seq(2, 20, by = 1))
```

We will also incorporate cross-validation in the selection of the parameter, creating an object called `control` and utilizing the `trainControl()` function from the `caret` package, as follows:

```
> control <- trainControl(method = "cv")
```

Now, we can create the object that will show us how to compute the optimal k value with the `train()` function, which is also part of the `caret` package. Remember that while conducting any sort of random sampling, you will need to set the `seed` value as follows:

```
> set.seed(502)
```

The object created by the `train()` function requires the model formula, `train` data name, and an appropriate method. The model formula is the same as we've used before- $y \sim x$. The method designation is simply `knn`. With this in mind, this code will create the object that will show us the optimal k value, as follows:

```
> knn.train <- train(type ~ ., data = train,
  method = "knn",
  trControl = control,
  tuneGrid = grid1)
```

Calling the object provides us with the k parameter that we are seeking, which is $k=17$:

```
> knn.train
k-Nearest Neighbors
385 samples
  7 predictor
  2 classes: 'No', 'Yes'
No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 347, 347, 345, 347, 347, 346, ...
Resampling results across tuning parameters:
  k  Accuracy  Kappa  Accuracy SD  Kappa SD
  2  0.736     0.359  0.0506      0.1273
  3  0.762     0.416  0.0526      0.1313
  4  0.761     0.418  0.0521      0.1276
  5  0.759     0.411  0.0566      0.1295
  6  0.772     0.442  0.0559      0.1474
  7  0.767     0.417  0.0455      0.1227
  8  0.767     0.425  0.0436      0.1122
  9  0.772     0.435  0.0496      0.1316
 10  0.780     0.458  0.0485      0.1170
 11  0.777     0.446  0.0437      0.1120
 12  0.775     0.440  0.0547      0.1443
 13  0.782     0.456  0.0397      0.1084
 14  0.780     0.449  0.0557      0.1349
 15  0.772     0.427  0.0449      0.1061
 16  0.782     0.453  0.0403      0.0954
 17  0.795     0.485  0.0382      0.0978
 18  0.782     0.451  0.0461      0.1205
 19  0.785     0.455  0.0452      0.1197
 20  0.782     0.446  0.0451      0.1124
Accuracy was used to select the optimal model using the largest
value.
The final value used for the model was k = 17.
```

In addition to the results that yield $k=17$, we get the information in the form of a table on the Accuracy and Kappa statistics and their standard deviations from the cross-validation. Accuracy tells us the percentage of observations that the model classified correctly. Kappa refers to what is known as **Cohen's Kappa statistic**. The Kappa statistic is commonly used to provide a measure of how well two evaluators can classify an observation correctly. It provides an insight into this problem by adjusting the accuracy scores, which is done by accounting for the evaluators being totally correct by mere chance. The formula for the statistic is $Kappa = (per\ cent\ of\ agreement - per\ cent\ of\ chance\ agreement) / (1 - per\ cent\ of\ chance\ agreement)$.

The *per cent of agreement* is the rate that the evaluators agreed on for the class (accuracy), and *percent of chance agreement* is the rate that the evaluators randomly agreed on. The higher the statistic, the better they performed with the maximum agreement being one. We will work through an example when we will apply our model on the `test` data.

To do this, we will utilize the `knn()` function from the `class` package. With this function, we will need to specify at least four items. These would be the `train` inputs, the `test` inputs, correct labels from the `train` set, and `k`. We will do this by creating the `knn.test` object and see how it performs:

```
> knn.test <- knn(train[, -8], test[, -8], train[, 8], k = 17)
```

With the object created, let's examine the confusion matrix and calculate the accuracy and kappa:

```
> table(knn.test, test$type)
knn.test No Yes
No      77  26
Yes     16  28
```

The accuracy is done by simply dividing the correctly classified observations by the total observations:

```
> (77 + 28) / 147
[1] 0.7142857
```

The accuracy of 71 per cent is less than that we achieved on the `train` data, which was almost eighty per cent. We can now produce the kappa statistic as follows:

```
> #calculate Kappa
> prob.agree <- (77 + 28) / 147 #accuracy
> prob.chance <- ((77 + 26) / 147) * ((77 + 16) / 147)
> prob.chance
[1] 0.4432875
> kappa <- (prob.agree - prob.chance) / (1 - prob.chance)
> kappa
[1] 0.486783
```

The kappa statistic of 0.49 is what we achieved with the `train` set. Altman(1991) provides a heuristic to assist us in the interpretation of the statistic, which is shown in the following table:

Value of <i>K</i>	Strength of Agreement
<0.20	Poor
0.21-0.40	Fair
0.41-0.60	Moderate
0.61-0.80	Good
0.81-1.00	Very good

With our kappa only moderate and with an accuracy just over 70 per cent on the `test` set, we should see whether we can perform better by utilizing weighted neighbors. A weighting schema increases the influence of neighbors that are closest to an observation versus those that are farther away. The farther away the observation is from a point in space, the more penalized its influence is. For this technique, we will use the `kknn` package and its `train.kknn()` function to select the optimal weighting scheme.

The `train.kknn()` function uses LOOCV that we examined in the prior chapters in order to select the best parameters for the optimal *k* neighbors, one of the two distance measures, and a `kernel` function.

The unweighted *k* neighbors algorithm that we created uses the Euclidian distance, as we discussed previously. With the `kknn` package, there are options available to compare the sum of the absolute differences versus the Euclidian distance. The package refers to the distance calculation used as the `Minkowski` parameter.

As for the weighting of the distances, many different methods are available. For our purpose, the package that we will use has ten different weighting schemas, which includes the unweighted ones. They are rectangular (unweighted), triangular, `epanechnikov`, `biweight`, `triweight`, `cosine`, `inversion`, `gaussian`, `rank`, and `optimal`. A full discussion of these weighting techniques is available in *Hechenbichler K. and Schliep K.P. (2004)*.

For simplicity, let's focus on just two: `triangular` and `epanechnikov`. Prior to having the weights assigned, the algorithm standardizes all the distances so that they are between zero and one. The triangular weighting method multiplies the observation distance by one minus the distance. With `Epanechnikov`, the distance is multiplied by $\frac{3}{4}$ times (one minus the distance two). For our problem, we will incorporate these weighting methods along with the standard unweighted version for comparison purposes.

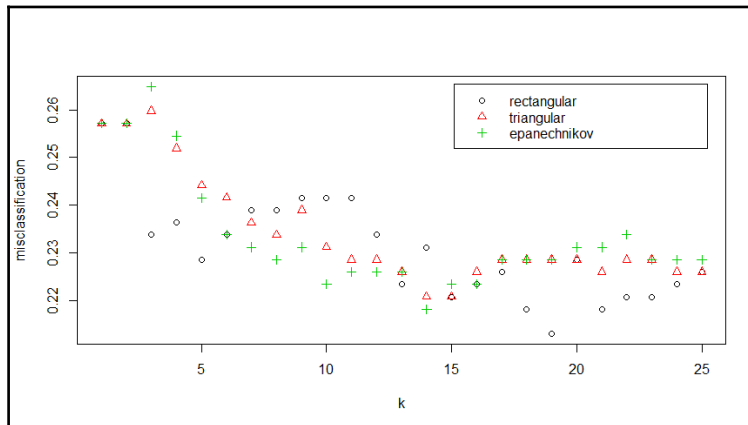
After specifying a random seed, we will create the `train` set object with `kknn()`. This function asks for the maximum number of k values (`kmax`), distance (one is equal to Euclidian and two is equal to absolute), and `kernel`. For this model, `kmax` will be set to 25 and distance will be 2:

```
> set.seed(123)
> kknn.train <- train.kknn(type ~ ., data = train, kmax = 25,
  distance = 2,
  kernel = c("rectangular", "triangular", "epanechnikov"))
```

A nice feature of the package is the ability to plot and compare the results, as follows:

```
> plot(kknn.train)
```

The following is the output of the preceding command:



This plot shows k on the x-axis and the percentage of misclassified observations by `kernel`. To my pleasant surprise, the unweighted (**rectangular**) version at $k: 19$ performs the best. You can also call the object to see what the classification error and the best parameter are in the following way:

```
> kknn.train
Call:
train.kknn(formula = type ~ ., data = train, kmax = 25, distance =
  2, kernel
  = c("rectangular", "triangular", "epanechnikov"))
Type of response variable: nominal
Minimal misclassification: 0.212987
Best kernel: rectangular
Best k: 19
```

So, with this data, weighting the distance does not improve the model accuracy in training and, as we can see here, didn't even do as well on the test set:

```
> kknn.pred <- predict(kknn.train, newdata = test)
> table(kknn.pred, test$type)
kknn.pred No Yes
      No  76  27
      Yes  17  27
```

There are other weights that we could try, but as I tried these other weights, the results that I achieved were not more accurate than these. We don't need to pursue KNN any further. I would encourage you to experiment with various parameters on your own to see how they perform.

SVM modeling

We will use the `e1071` package to build our SVM models. We will start with a linear support vector classifier and then move on to the nonlinear versions. The `e1071` package has a nice function for SVM called `tune.svm()`, which assists in the selection of the tuning parameters/kernel functions. The `tune.svm()` function from the package uses cross-validation to optimize the tuning parameters. Let's create an object called `linear.tune` and call it using the `summary()` function, as follows:

```
> linear.tune <- tune.svm(type ~ ., data = train,
  kernel = "linear",
  cost = c(0.001, 0.01, 0.1, 1, 5, 10))
> summary(linear.tune)
Parameter tuning of 'svm':
- sampling method: 10-fold cross validation
- best parameters:
cost
1
- best performance: 0.2051957
- Detailed performance results:
  cost      error dispersion
1 1e-03 0.3197031 0.06367203
2 1e-02 0.2080297 0.07964313
3 1e-01 0.2077598 0.07084088
4 1e+00 0.2051957 0.06933229
5 5e+00 0.2078273 0.07221619
6 1e+01 0.2078273 0.07221619
```

The optimal `cost` function is one for this data and leads to a misclassification error of roughly 21 per cent. We can make predictions on the `test` data and examine that as well using the `predict()` function and applying `newdata = test`:

```
> best.linear <- linear.tune$best.model
> tune.test <- predict(best.linear, newdata = test)
> table(tune.test, test$type)
tune.test No Yes
      No  82  22
      Yes  13  30
> (82 + 30)/147
[1] 0.7619048
```

The linear support vector classifier has slightly outperformed KNN on both the `train` and `test` sets. The `e1071` package has a nice function for SVM called `tune.svm()` that assists in the selection of the `tuning parameters/kernel` functions. We will now see if nonlinear methods will improve our performance and also use cross-validation to select tuning parameters.

The first `kernel` function that we will try is `polynomial`, and we will be tuning two parameters: a degree of polynomial (`degree`) and kernel coefficient (`coef0`). The `polynomial` order will be 3, 4, and 5 and the coefficient will be in increments from 0.1 to 4, as follows:

```
> set.seed(123)
> poly.tune <- tune.svm(type ~ ., data = train,
  kernel = "polynomial",
  degree = c(3, 4, 5),
  coef0 = c(0.1, 0.5, 1, 2, 3, 4))
> summary(poly.tune)
Parameter tuning of 'svm':
- sampling method: 10-fold cross validation
- best parameters:
  degree coef0
    3     0.1
- best performance: 0.2310391
```

The model has selected `degree` of 3 for the polynomial and coefficient of 0.1. Just as the linear SVM, we can create predictions on the `test` set with these parameters, as follows:

```
> best.poly <- poly.tune$best.model
> poly.test <- predict(best.poly, newdata = test)
> table(poly.test, test$type)
poly.test No Yes
      No  81  28
      Yes  12  26
> (81 + 26) / 147
[1] 0.7278912
```

This did not perform quite as well as the linear model. We will now run the radial basis function. In this instance, the one parameter that we will solve for is `gamma`, which we will examine in increments of 0.1 to 4. If `gamma` is too small, the model will not capture the complexity of the decision boundary; if it is too large, the model will severely overfit:

```
> set.seed(123)
> rbf.tune <- tune.svm(type ~ ., data = train,
  kernel = "radial",
  gamma = c(0.1, 0.5, 1, 2, 3, 4))
> summary(rbf.tune)
Parameter tuning of 'svm':
- sampling method: 10-fold cross validation
- best parameters:
  gamma
    0.5
- best performance: 0.2284076
```

The best `gamma` value is 0.5, and the performance at this setting does not seem to improve much over the other SVM models. We will check for the `test` set as well in the following way:

```
> best.rbf <- rbf.tune$best.model
> rbf.test <- predict(best.rbf, newdata = test)
> table(rbf.test, test$type)
rbf.test No Yes
      No  73  33
      Yes  20  21
> (73+21)/147
[1] 0.6394558
```


The performance is downright abysmal. One last shot to improve here would be with `kernel = "sigmoid"`. We will be solving for two parameters-- `gamma` and the kernel coefficient (`coef0`):

```
> set.seed(123)
> sigmoid.tune <- tune.svm(type ~ ., data = train,
  kernel = "sigmoid",
  gamma = c(0.1, 0.5, 1, 2, 3, 4),
  coef0 = c(0.1, 0.5, 1, 2, 3, 4))
> summary(sigmoid.tune)
Parameter tuning of 'svm':
- sampling method: 10-fold cross validation
- best parameters:
  gamma coef0
    0.1     2
- best performance: 0.2080972
```

This error rate is in line with the linear model. It is now just a matter of whether it performs better on the `test` set or not:

```
> best.sigmoid <- sigmoid.tune$best.model
> sigmoid.test <- predict(best.sigmoid, newdata = test)
> table(sigmoid.test, test$type)
sigmoid.test No Yes
             No  82  19
             Yes  11  35
> (82+35)/147
[1] 0.7959184
```

Lo and behold! We finally have a test performance that is in line with the performance on the `train` data. It appears that we can choose the sigmoid kernel as the best predictor.

So far we've played around with different models. Now, let's evaluate their performance along with the linear model using metrics other than just the accuracy.

Model selection

We've looked at two different types of modeling techniques here, and for all intents and purposes, KNN has fallen short. The best accuracy on the `test` set for KNN was only around 71 per cent. Conversely, with SVM, we could obtain an accuracy close to 80 per cent. Before just simply selecting the most accurate mode, in this case, the SVM with the sigmoid kernel, let's look at how we can compare them with a deep examination of the confusion matrices.

For this exercise, we can turn to our old friend, the `caret` package and utilize the `confusionMatrix()` function. Keep in mind that we previously used the same function from the `InformationValue` package. The `caret` package version provides much more detail and it will produce all of the statistics that we need in order to evaluate and select the best model. Let's start with the last model that we built first, using the same syntax that we used in the base `table()` function with the exception of specifying the `positive` class, as follows:

```
> confusionMatrix(sigmoid.test, test$type, positive = "Yes")
Confusion Matrix and Statistics

      Reference
Prediction No Yes
No      82  19
Yes     11  35

      Accuracy : 0.7959
      95% CI   : (0.7217, 0.8579)
No Information Rate : 0.6327
P-Value [Acc > NIR] : 1.393e-05
      Kappa    : 0.5469
McNemar's Test P-Value : 0.2012
      Sensitivity : 0.6481
      Specificity : 0.8817
      Pos Pred Value : 0.7609
      Neg Pred Value : 0.8119
      Prevalence   : 0.3673
      Detection Rate : 0.2381
      Detection Prevalence : 0.3129
      Balanced Accuracy : 0.7649
      'Positive' Class : Yes
```

The function produces some items that we already covered such as `Accuracy` and `Kappa`. Here are the other statistics that it produces:

- `No Information Rate` is the proportion of the largest class; 63 per cent did not have diabetes.
- `P-Value` is used to test the hypothesis that the accuracy is actually better than `No Information Rate`.
- We will not concern ourselves with `McNemar's Test`, which is used for the analysis of the matched pairs, primarily in epidemiology studies.
- `Sensitivity` is the true positive rate; in this case, the rate of those not having diabetes has been correctly identified as such.
- `Specificity` is the true negative rate or, for our purposes, the rate of a diabetic that has been correctly identified.

- The positive predictive value (Pos Pred Value) is the probability of someone in the population classified as being diabetic and truly has the disease. The following formula is used:

$$PPV = \frac{sensitivity * prevalence}{(sensitivity * prevalence) + (1 - specificity) * (1 - prevalence)}$$

- The negative predictive value (Neg Pred Value) is the probability of someone in the population classified as not being diabetic and truly does not have the disease. The formula for this is as follows:

$$NPV = \frac{specificity * (1 - prevalence)}{((1 - sensitivity) * (prevalence)) + (specificity) * (1 - prevalence)}$$

- Prevalence is the estimated population prevalence of the disease, calculated here as the total of the second column (the Yes column) divided by the total observations.
- Detection Rate is the rate of the true positives that have been identified, in our case, 35, divided by the total observations.
- Detection Prevalence is the predicted prevalence rate, or in our case, the bottom row divided by the total observations.
- Balanced Accuracy is the average accuracy obtained from either class. This measure accounts for a potential bias in the classifier algorithm, thus potentially overpredicting the most frequent class. This is simply *Sensitivity + Specificity divided by 2*.

The sensitivity of our model is not as powerful as we would like and tells us that we are missing some features from our dataset that would improve the rate of finding the true diabetic patients. We will now compare these results with the linear SVM, as follows:

```
> confusionMatrix(tune.test, test$type, positive = "Yes")
      Reference
Prediction No Yes
No      82    24
Yes     11    30
      Accuracy : 0.7619
      95% CI   : (0.6847, 0.8282)
No Information Rate : 0.6327
P-Value [Acc > NIR] : 0.0005615
      Kappa    : 0.4605
```

```
McNemar's Test P-Value : 0.0425225
      Sensitivity : 0.5556
      Specificity : 0.8817
      Pos Pred Value : 0.7317
      Neg Pred Value : 0.7736
      Prevalence : 0.3673
      Detection Rate : 0.2041
      Detection Prevalence : 0.2789
      Balanced Accuracy : 0.7186
      'Positive' Class : Yes
```

As we can see by comparing the two models, the linear SVM is inferior across the board. Our clear winner is the sigmoid kernel SVM. However, there is one thing that we are missing here and that is any sort of feature selection. What we have done is just thrown all the variables together as the feature input space and let the blackbox SVM calculations give us a predicted classification. One of the issues with SVMs is that the findings are very difficult to interpret. There are a number of ways to go about this process that I feel are beyond the scope of this chapter; this is something that you should begin to explore and learn on your own as you become comfortable with the basics that have been outlined previously.

Feature selection for SVMs

However, all is not lost on feature selection and I want to take some space to show you a quick way of how to begin exploring this matter. It will require some trial and error on your part. Again, the `caret` package helps out in this matter as it will run a cross-validation on a linear SVM based on the `kernlab` package.

To do this, we will need to set the random seed, specify the cross-validation method in the `caret`'s `rfeControl()` function, perform a recursive feature selection with the `rfe()` function, and then test how the model performs on the test set. In `rfeControl()`, you will need to specify the function based on the model being used. There are several different functions that you can use. Here we will need `lrFuncs`. To see a list of the available functions, your best bet is to explore the documentation with `?rfeControl` and `?caretFuncs`. The code for this example is as follows:

```
> set.seed(123)
> rfeCNTL <- rfeControl(functions = lrFuncs, method = "cv", number
  = 10)
> svm.features <- rfe(train[, 1:7], train[, 8],
  sizes = c(7, 6, 5, 4),
  rfeControl = rfeCNTL,
  method = "svmLinear")
```

To create the `svm.features` object, it was important to specify the inputs and response factor, number of input features via `sizes`, and linear method from `kernlab`, which is the `svmLinear` syntax. Other options are available using this method, such as `svmPoly`. No method for a sigmoid kernel is available. Calling the object allows us to see how the various feature sizes perform, as follows:

```
> svm.features
Recursive feature selection
Outer resampling method: Cross-Validated (10 fold)
Resampling performance over subset size:
  Variables Accuracy  Kappa AccuracySD KappaSD Selected
         4  0.7797 0.4700    0.04969  0.1203
         5  0.7875 0.4865    0.04267  0.1096      *
         6  0.7847 0.4820    0.04760  0.1141
         7  0.7822 0.4768    0.05065  0.1232
The top 5 variables (out of 5):
```

Counter-intuitive as it is, the five variables perform quite well by themselves as well as when `skin` and `bp` are included. Let's try this out on the `test` set, remembering that the accuracy of the full model was 76.2 per cent:

```
> svm.5 <- svm(type ~ glu + ped + npreg + bmi + age,
  data = train,
  kernel = "linear")
> svm.5.predict <- predict(svm.5, newdata = test[c(1, 2, 5, 6, 7)])
> table(svm.5.predict, test$type)
svm.5.predict No Yes
             No  79  21
             Yes 14  33
```

This did not perform as well and we can stick with the full model. You can see through trial and error how this technique can play out in order to determine some simple identification of feature importance. If you want to explore the other techniques and methods that you can apply here, and for blackbox techniques in particular, I recommend that you start by reading the work by Guyon and Elisseeff (2003) on this subject.

Summary

In this chapter, we reviewed two new classification techniques: KNN and SVM. The goal was to discover how these techniques work, and the differences between them, by building and comparing models on a common dataset in order to predict if an individual had diabetes. KNN involved both the unweighted and weighted nearest neighbor algorithms. These did not perform as well as the SVMs in predicting whether an individual had diabetes or not.

We examined how to build and tune both the linear and nonlinear support vector machines using the `e1071` package. We used the extremely versatile `caret` package to compare the predictive ability of a linear and nonlinear support vector machine and saw that the nonlinear support vector machine with a sigmoid kernel performed the best.

Finally, we touched on how you can use the `caret` package to perform a crude feature selection, as this is a difficult challenge with a blackbox technique such as SVM. This can be a major challenge when using these techniques and you will need to consider how viable they are in order to address the business question.

6

Classification and Regression Trees

"The classifiers most likely to be the best are the random forest (RF) versions, the best of which (implemented in R and accessed via caret), achieves 94.1 percent of the maximum accuracy overcoming 90 percent in the 84.3 percent of the data sets."

- Fernández-Delgado et al. (2014)

This quote from *Fernández-Delgado et al.* in the *Journal of Machine Learning Research* is meant to demonstrate that the techniques in this chapter are quite powerful, particularly when used for classification problems. Certainly, they don't always offer the best solution but they do provide a good starting point.

In the previous chapters, we examined the techniques used to predict either a quantity or a label classification. Here, we will apply them to both types of problems. We will also approach the business problem differently than in the previous chapters. Instead of defining a new problem, we will apply the techniques to some of the issues that we already tackled, with an eye to see if we can improve our predictive power. For all intents and purposes, the business case in this chapter is to see if we can improve on the models that we selected before.

The first item of discussion is the basic decision tree, which is both simple to build and to understand. However, the single decision tree method does not perform as well as the other methods that you learned, for example, the support vector machines, or as the ones that we will learn, such as the neural networks. Therefore, we will discuss the creation of multiple, sometimes hundreds, of different trees with their individual results combined, leading to a single overall prediction.

These methods, as the paper referenced at the beginning of this chapter states, perform as well as, or better than, any technique in this book. These methods are known as **random forests** and **gradient boosted trees**. Additionally, we will take a break from a business case and show how employing the random forest method on a dataset can assist in feature elimination/selection.

An overview of the techniques

We will now get to an overview of the techniques, covering the regression and classification trees, random forests, and gradient boosting. This will set the stage for the practical business cases.

Understanding the regression trees

To establish an understanding of tree-based methods, it is probably easier to start with a quantitative outcome and then move on to how it works in a classification problem. The essence of a tree is that the features are partitioned, starting with the first split that improves the RSS the most. These binary splits continue until the termination of the tree. Each subsequent split/partition is not done on the entire dataset but only on the portion of the prior split that it falls under. This top-down process is referred to as recursive partitioning. It is also a process that is **greedy**, a term you may stumble upon in reading about the machine learning methods. Greedy means that, during each split in the process, the algorithm looks for the greatest reduction in the RSS without any regard to how well it will perform on the latter partitions. The result is that you may end up with a full tree of unnecessary branches leading to a low bias but a high variance. To control this effect, you need to appropriately prune the tree to an optimal size after building a full tree.

Figure 6.1, provides a visual of this technique in action. The data is hypothetical with 30 observations, a response ranging from 1 to 10, and two predictor features, both ranging in value from 0 to 10 named **X1** and **X2**. The tree has three splits leading to four terminal nodes. Each split is basically an `if . . . then` statement or uses an R syntax `ifelse()`. The first split is--if **X1** is less than 3.5, then the response is split into four observations with an average value of 2.4 and the remaining 26 observations. This left branch of four observations is a terminal node as any further splits would not substantially improve the RSS. The predicted value for these four observations in that partition of the tree becomes the average. The next split is at **X2** < 4 and finally, **X1** < 7.5.

An advantage of this method is that it can handle highly nonlinear relationships; however, can you see a couple of potential problems? The first issue is that an observation is given the average of the terminal node under which it falls. This can hurt the overall predictive performance (high bias). Conversely, if you keep partitioning the data further and further so as to achieve a low bias, a high variance can become an issue. As with the other methods, you can use cross-validation to select the appropriate tree depth size:

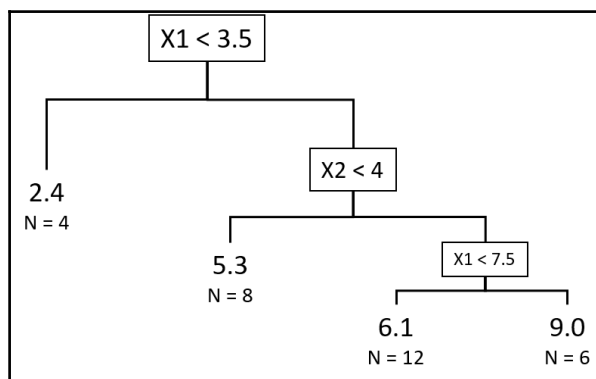


Figure 6.1: Regression Tree with 3 splits and 4 terminal nodes and the corresponding node average and number of observations

Classification trees

Classification trees operate under the same principle as regression trees, except that the splits are not determined by the RSS but an error rate. The error rate used is not what you would expect where the calculation is simply the misclassified observations divided by the total observations. As it turns out, when it comes to tree-splitting, a misclassification rate, by itself, may lead to a situation where you can gain information with a further split but not improve the misclassification rate. Let's look at an example.

Suppose we have a node, let's call it **N0**, where you have seven observations labeled **N0** and three observations labeled **Yes**, and we can say that the misclassified rate is 30 percent. With this in mind, let's calculate a common alternative error measure called the **Gini index**. The formula for a single node Gini index is as follows:

$$Gini = 1 - (\text{probability of Class 1})^2 - (\text{probability of Class 2})^2$$

Then, for **N0**, the Gini is $1 - (.7)^2 - (.3)^2$, which is equal to 0.42, versus the misclassification rate of 30 per cent.

Taking this example further, we will now create node *N1* with three observations from `Class 1` and none from `Class 2`, along with *N2*, which has four observations from `Class 1` and three from `Class 2`. Now, the overall misclassification rate for this branch of the tree is still 30 per cent, but look at how the overall Gini index has improved:

- $Gini(N1) = 1 - (3/3)^2 - (0/3)^2 = 0$
- $Gini(N2) = 1 - (4/7)^2 - (3/7)^2 = 0.49$
- *New Gini index* = (*proportion of N1* $\times$ $Gini(N1)$) + (*proportion of N2* $\times$ $Gini(N2)$), which is equal to $(.3 \times 0) + (.7 \times 0.49)$ or 0.343

By doing a split on a surrogate error rate, we actually improved our model impurity, reducing it from 0.42 to 0.343, whereas the misclassification rate did not change. This is the methodology that is used by the `rpart()` package, which we will be using in this chapter.

Random forest

To greatly improve our model's predictive ability, we can produce numerous trees and combine the results. The random forest technique does this by applying two different tricks in model development. The first is the use of **bootstrap aggregation** or **bagging**, as it is called.

In bagging, an individual tree is built on a random sample of the dataset, roughly two-thirds of the total observations (note that the remaining one-third is referred to as **out-of-bag (oob)**). This is repeated dozens or hundreds of times and the results are averaged. Each of these trees is grown and not pruned based on any error measure, and this means that the variance of each of these individual trees is high. However, by averaging the results, you can reduce the variance without increasing the bias.

The next thing that random forest brings to the table is that concurrently with the random sample of the data--that is, bagging--it also takes a random sample of the input features at each split. In the `randomForest` package, we will use the default random number of the predictors that are sampled, which, for classification problems, is the square root of the total predictors and for regression, it is the total number of the predictors divided by three. The number of predictors the algorithm randomly chooses at each split can be changed via the model tuning process.

By doing this random sample of the features at each split and incorporating it into the methodology, you can mitigate the effect of a highly correlated predictor becoming the main driver in all of your bootstrapped trees, preventing you from reducing the variance that you hoped to achieve with bagging. The subsequent averaging of the trees that are less correlated to each other is more generalizable and robust to outliers than if you only performed bagging.

Gradient boosting

Boosting methods can become extremely complicated to learn and understand, but you should keep in mind what is fundamentally happening behind the curtain. The main idea is to build an initial model of some kind (linear, spline, tree, and so on) called the base learner, examine the residuals, and fit a model based on these residuals around the so-called **loss function**. A loss function is merely the function that measures the discrepancy between the model and desired prediction, for example, a squared error for regression or the logistic function for classification. The process continues until it reaches some specified stopping criterion. This is sort of like the student who takes a practice exam and gets 30 out of 100 questions wrong and, as a result, studies only these 30 questions that were missed. In the next practice exam, they get 10 out of those 30 wrong and so only focus on those 10 questions, and so on. If you would like to explore the theory behind this further, a great resource for you is available in *Frontiers in Neurorobotics*, *Gradient boosting machines, a tutorial*, Natekin A., Knoll A. (2013), at

<http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3885826/>.

As just mentioned, boosting can be applied to many different base learners, but here we will only focus on the specifics of **tree-based learning**. Each tree iteration is small and we will determine how small with one of the tuning parameters referred to as interaction depth. In fact, it may be as small as one split, which is referred to as a stump.

Trees are sequentially fit to the residuals, according to the loss function, up to the number of trees that we specified (our stopping criterion).

There are a number of parameters that require tuning in the model-building process using the Xgboost package, which stands for **eXtreme Gradient Boosting**. This package has become quite popular for online data contests because of its winning performance. There is excellent background material on boosting trees and on Xgboost on the following website:

<http://xgboost.readthedocs.io/en/latest/model.html>

What we will do in the business case is show how to begin to optimize the hyperparameters and produce meaningful output and predictions. These parameters can interact with each other, and if you just tinker with one without considering the other, your model may worsen the performance. The `caret` package will help us in the tuning endeavor.

Business case

The overall business objective in this situation is to see whether we can improve the predictive ability for some of the cases that we already worked on in the previous chapters. For regression, we will revisit the prostate cancer dataset from Chapter 4, *Advanced Feature Selection in Linear Models*. The baseline mean squared error to improve on is 0.444.

For classification purposes, we will utilize both the breast cancer biopsy data from Chapter 3, *Logistic Regression and Discriminant Analysis* and the Pima Indian Diabetes data from Chapter 5, *More Classification Techniques - K-Nearest Neighbors and Support Vector Machines*. In the breast cancer data, we achieved 97.6 per cent predictive accuracy. For the diabetes data, we are seeking to improve on the 79.6 per cent accuracy rate.

Both random forests and boosting will be applied to all three datasets. The simple tree method will only be used on the breast and prostate cancer sets from Chapter 4, *Advanced Feature Selection in Linear Models*.

Modeling and evaluation

To perform the modeling process, we will need to load seven different R packages. Then, we will go through each of the techniques and compare how well they perform on the data analyzed with the prior methods in the previous chapters.

Regression tree

We will jump right into the `prostate` dataset, but let's first load the necessary R packages. As always, please ensure that you have the libraries installed prior to loading the packages:

```
> library(rpart) #classification and regression trees
> library(partykit) #treeplots
> library(MASS) #breast and pima indian data
> library(ElemStatLearn) #prostate data
> library(randomForest) #random forests
> library(xgboost) #gradient boosting
> library(caret) #tune hyper-parameters
```

We will first do regression with the `prostate` data and prepare it, as we did in Chapter 4, *Advanced Feature Selection in Linear Models*. This involves calling the dataset, coding the gleason score as an indicator variable using the `ifelse()` function, and creating the test and train sets. The train set will be `pros.train` and the test set will be `pros.test`, as follows:

```
> data(prostate)
> prostate$gleason <- ifelse(prostate$gleason == 6, 0, 1)
> pros.train <- subset(prostate, train == TRUE)[, 1:9]
> pros.test <- subset(prostate, train == FALSE)[, 1:9]
```

To build a regression tree on the train data, we will use the `rpart()` function from R's `party` package. The syntax is quite similar to what we used in the other modeling techniques:

```
> tree.pros <- rpart(lpsa ~ ., data = pros.train)
```

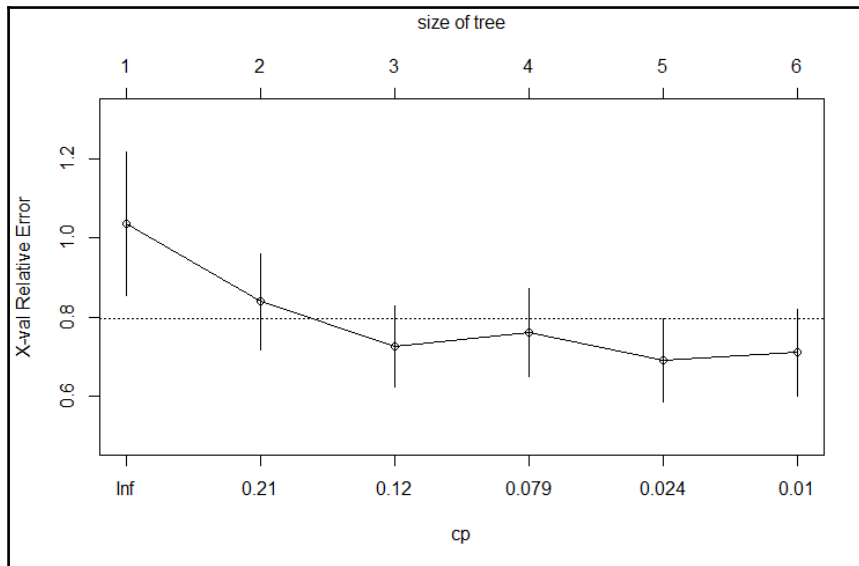
We can call this object and examine the error per number of splits in order to determine the optimal number of splits in the tree:

```
> print(tree.pros$cptable)
  CP nsplit rel error  xerror  xstd
1 0.35852251  0 1.0000000 1.0364016 0.1822698
2 0.12295687  1 0.6414775 0.8395071 0.1214181
3 0.11639953  2 0.5185206 0.7255295 0.1015424
4 0.05350873  3 0.4021211 0.7608289 0.1109777
5 0.01032838  4 0.3486124 0.6911426 0.1061507
6 0.01000000  5 0.3382840 0.7102030 0.1093327
```

This is a very important table to analyze. The first column labeled `CP` is the cost complexity parameter. The second column, `nsplit`, is the number of splits in the tree. The `rel error` column stands for relative error and is the RSS for the number of splits divided by the RSS for no splits $RSS(k)/RSS(0)$. Both `xerror` and `xstd` are based on the ten-fold cross-validation with `xerror` being the average error and `xstd` the standard deviation of the cross-validation process. We can see that while five splits produced the lowest error on the full dataset, four splits produced a slightly less error using cross-validation. You can examine this using `plotcp()`:

```
> plotcp(tree.pros)
```

The output of the preceding command is as follows:



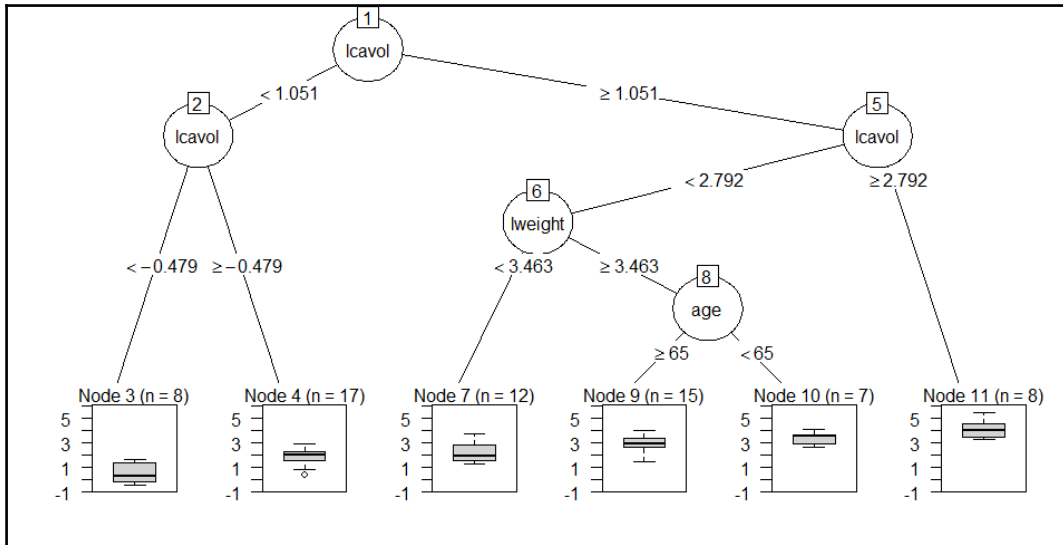
The plot shows us the relative error by the tree size with the corresponding error bars. The horizontal line on the plot is the upper limit of the lowest standard error. Selecting a tree size, 5, which is four splits, we can build a new tree object where `xerror` is minimized by pruning our tree by first creating an object for `cp` associated with the pruned tree from the table. Then the `prune()` function handles the rest:

```
> cp <- min(tree.pros$cpstable[5, ])
> prune.tree.pros <- prune(tree.pros, cp = cp)
```

With this done, you can plot and compare the full and pruned trees. The tree plots produced by the `partykit` package are much better than those produced by the `party` package. You can simply use the `as.party()` function as a wrapper in `plot()`:

```
> plot(as.party(tree.pros))
```

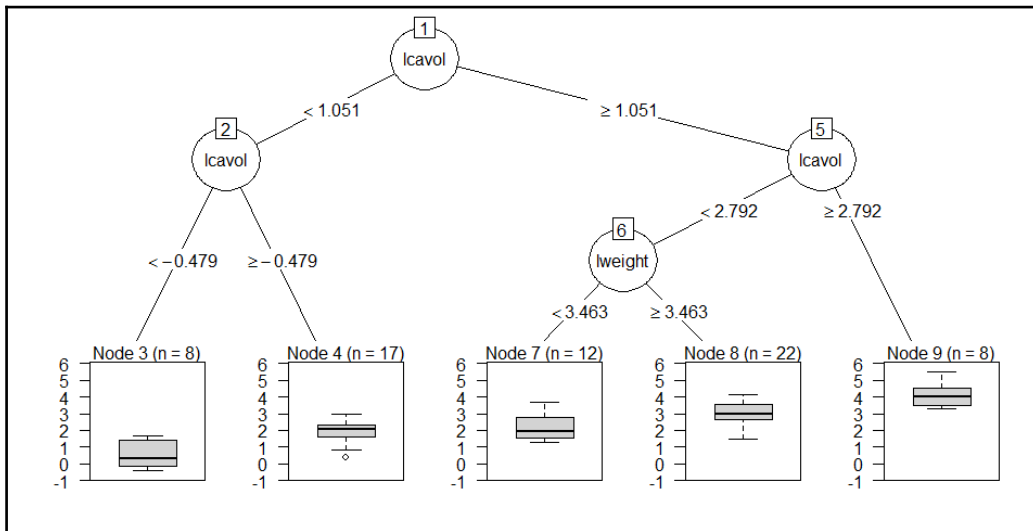
The output of the preceding command is as follows:



Now we will use the `as.party()` function for the pruned tree:

```
> plot(as.party(prune.tree.pros))
```

The output of the preceding command is as follows:



Note that the splits are exactly the same in the two trees with the exception of the last split, which includes the variable `age` for the full tree. Interestingly, both the first and second splits in the tree are related to the log of cancer volume (`lcavol`). These plots are quite informative as they show the splits, nodes, observations per node, and boxplots of the outcome that we are trying to predict.

Let's see how well the pruned tree performs on the `test` data. What we will do is create an object of the predicted values using the `predict()` function and incorporate the `test` data. Then, calculate the errors (the predicted values minus the actual values) and finally, the mean of the squared errors:

```
> party.pros.test <- predict(prune.tree.pros, newdata = pros.test)
> rpart.resid <- party.pros.test - pros.test$lpsa
> mean(rpart.resid^2) #calculate MSE
[1] 0.5267748
```

We have not improved on the predictive value from our work in *Chapter 4, Advanced Feature Selection in Linear Models* where the baseline MSE was 0.44. However, the technique is not without value. One can look at the tree plots that we produced and easily explain what the primary drivers behind the response are. As mentioned in the introduction, the trees are easy to interpret and explain, which may be more important than accuracy in many cases.

Classification tree

For the classification problem, we will prepare the breast cancer data in the same fashion as we did in *Chapter 3, Logistic Regression and Discriminant Analysis*. After loading the data, you will delete the patient ID, rename the features, eliminate the few missing values, and then create the `train/test` datasets in the following way:

```
> data(biopsy)
> biopsy <- biopsy[, -1] #delete ID
> names(biopsy) <- c("thick", "u.size", "u.shape", "adhsn",
  "s.size", "nucl",
  "chrom", "n.nuc", "mit", "class") #change the feature names
> biopsy.v2 <- na.omit(biopsy) #delete the observations with
  missing values
> set.seed(123) #random number generator
> ind <- sample(2, nrow(biopsy.v2), replace = TRUE, prob = c(0.7,
  0.3))
> biop.train <- biopsy.v2[ind == 1, ] #the training data set
> biop.test <- biopsy.v2[ind == 2, ] #the test data set
```


With the data set up appropriately, we will use the same syntax style for a classification problem as we did previously for a regression problem, but before creating a classification tree, we will need to ensure that the outcome is `Factor`, which can be done using the `str()` function:

```
> str(biop.test[, 10])  
Factor w/ 2 levels "benign","malignant": 1 1 1 1 1 2 1 2 1 1 ...
```

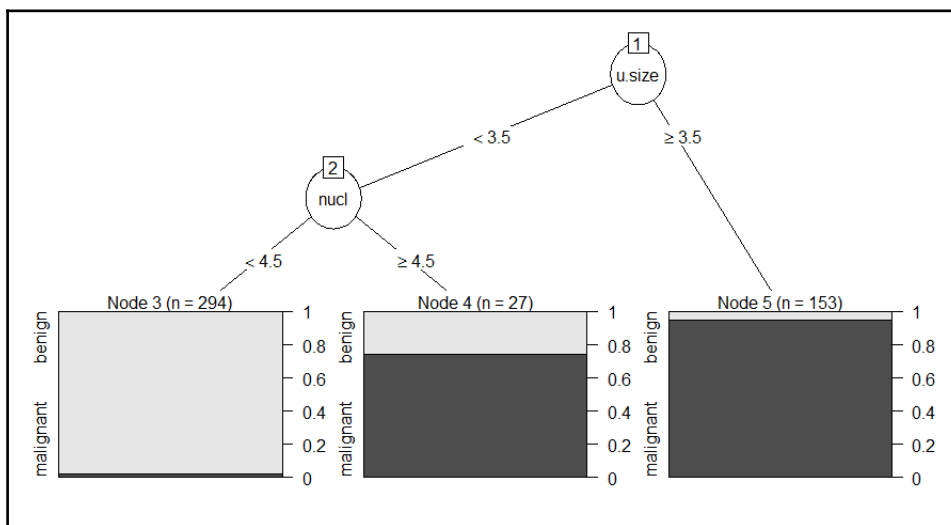
First, create the tree and then examine the table for the optimal number of splits:

```
> set.seed(123)  
> tree.biop <- rpart(class ~ ., data = biop.train)  
> tree.biop$cptable  
CP nsplit rel error xerror xstd  
1 0.79651163 0 1.0000000 1.0000000 0.06086254  
2 0.07558140 1 0.2034884 0.2674419 0.03746996  
3 0.01162791 2 0.1279070 0.1453488 0.02829278  
4 0.01000000 3 0.1162791 0.1744186 0.03082013
```

The cross-validation error is at a minimum with only two splits (row 3). We can now prune the tree, plot the pruned tree, and see how it performs on the `test` set:

```
> cp <- min(tree.biop$cptable[3, ])  
> prune.tree.biop <- prune(tree.biop, cp = cp)  
> plot(as.party(prune.tree.biop))
```

The output of the preceding command is as follows:



An examination of the tree plot shows that the uniformity of the cell size is the first split, then nuclei. The full tree had an additional split at the cell thickness. We can predict the test observations using `type="class"` in the `predict()` function, as follows:

```
> rparty.test <- predict(prune.tree.biop, newdata = biop.test, type
  ="class")
> table(rparty.test, biop.test$class)
rparty.test benign malignant
benign      136      3
malignant    6      64
> (136+64)/209
[1] 0.9569378
```

The basic tree with just two splits gets us almost 96 percent accuracy. This still falls short of 97.6 percent with logistic regression but should encourage us to believe that we can improve on this with the upcoming methods, starting with random forests.

Random forest regression

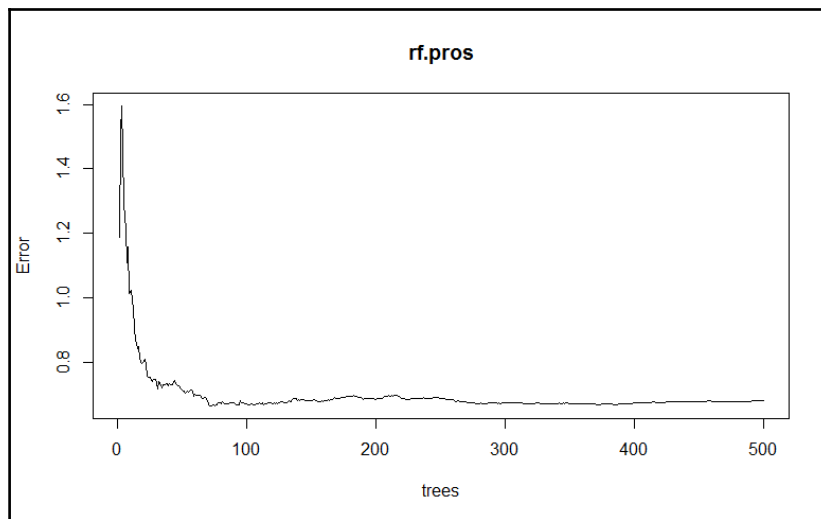
In this section, we will start by focusing on the prostate data again. Before moving on to the breast cancer and Pima Indian sets. We will use the `randomForest` package. The general syntax to create a random forest object is to use the `randomForest()` function and specify the formula and dataset as the two primary arguments. Recall that for regression, the default variable sample per tree iteration is $p/3$, and for classification, it is the square root of p , where p is equal to the number of predictor variables in the data frame. For larger datasets, in terms of p , you can tune the `mtry` parameter, which will determine the number of p sampled at each iteration. If p is less than 10 in these examples, we will forgo this procedure. When you want to optimize `mtry` for larger p datasets, you can utilize the `caret` package or use the `tuneRF()` function in `randomForest`. With this, let's build our forest and examine the results, as follows:

```
> set.seed(123)
> rf.pros <- randomForest(lpsa ~ ., data = pros.train)
> rf.pros
Call:
randomForest(formula = lpsa ~ ., data = pros.train)
Type of random forest: regression
Number of trees: 500
No. of variables tried at each split: 2
Mean of squared residuals: 0.6792314
% Var explained: 52.73
```

The call of the `rf.pros` object shows us that the random forest generated 500 different trees (the default) and sampled two variables at each split. The result is an MSE of 0.68 and nearly 53 percent of the variance explained. Let's see if we can improve on the default number of trees. Too many trees can lead to overfitting; naturally, how many is too many depends on the data. Two things can help out, the first one is a plot of `rf.pros` and the other is to ask for the minimum MSE:

```
> plot(rf.pros)
```

The output of the preceding command is as follows:



This plot shows the MSE by the number of trees in the model. You can see that as the trees are added, significant improvement in MSE occurs early on and then flatlines just before 100 trees are built in the forest.

We can identify the specific and optimal tree with the `which.min()` function, as follows:

```
> which.min(rf.pros$mse)
[1] 75
```

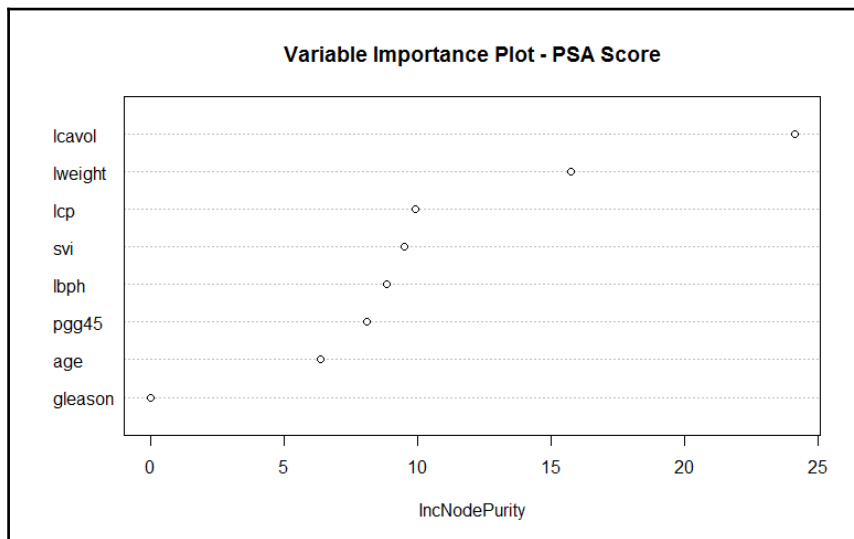
We can try 75 trees in the random forest by just specifying `ntree=75` in the model syntax:

```
> set.seed(123)
> rf.pros.2 <- randomForest(lpsa ~ ., data = pros.train, ntree
=75)
> rf.pros.2
Call:
randomForest(formula = lpsa ~ ., data = pros.train, ntree = 75)
Type of random forest: regression
Number of trees: 75
No. of variables tried at each split: 2
Mean of squared residuals: 0.6632513
% Var explained: 53.85
```

You can see that the MSE and variance explained have both improved slightly. Let's see another plot before testing the model. If we are combining the results of 75 different trees that are built using bootstrapped samples and only two random predictors, we will need a way to determine the drivers of the outcome. One tree alone cannot be used to paint this picture, but you can produce a variable importance plot and corresponding list. The y-axis is a list of variables in descending order of importance and the x-axis is the percentage of improvement in MSE. Note that for the classification problems, this will be an improvement in the Gini index. The function is `varImpPlot()`:

```
> varImpPlot(rf.pros.2, scale = T,
main = "Variable Importance Plot - PSA Score")
```

The output of the preceding command is as follows:



Consistent with the single tree, `lcavol` is the most important variable and `lweight` is the second-most important variable. If you want to examine the raw numbers, use the `importance()` function, as follows:

```
> importance(rf.pros.2)
IncNodePurity
lcavol 24.108641
lweight 15.721079
age 6.363778
lbph 8.842343
svi 9.501436
lcp 9.900339
gleason 0.000000
pgg45 8.088635
```

Now, it is time to see how it did on the test data:

```
> rf.pros.test <- predict(rf.pros.2, newdata = pros.test)
> rf.resid = rf.pros.test - pros.test$lpsa #calculate residual
> mean(rf.resid^2)
[1] 0.5136894
```

The MSE is still higher than our 0.44 that we achieved in Chapter 4, *Advanced Feature Selection in Linear Models* with LASSO and no better than just a single tree.

Random forest classification

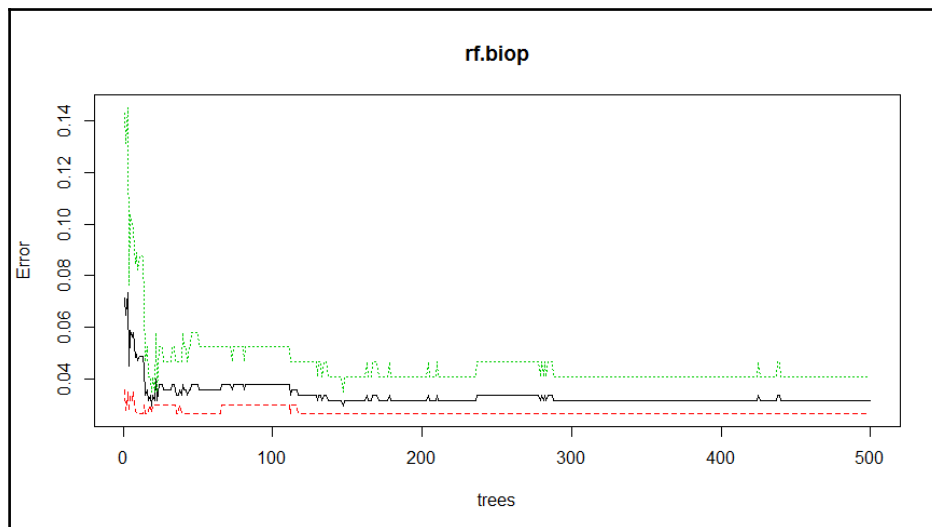
Perhaps you are disappointed with the performance of the random forest regression model, but the true power of the technique is in the classification problems. Let's get started with the breast cancer diagnosis data. The procedure is nearly the same as we did with the regression problem:

```
> set.seed(123)
> rf.biop <- randomForest(class ~ ., data = biop.train)
> rf.biop
Call:
randomForest(formula = class ~ ., data = biop.train)
Type of random forest: classification
Number of trees: 500
No. of variables tried at each split: 3
OOB estimate of error rate: 3.16%
Confusion matrix:
      benign malignant class.error
benign   294         8 0.02649007
malignant   7       165 0.04069767
```

The OOB error rate is 3.16%. Again, this is with all the 500 trees factored into the analysis. Let's plot the **Error by trees**:

```
> plot(rf.biop)
```

The output of the preceding command is as follows:



The plot shows that the minimum error and standard error is the lowest with quite a few trees. Let's now pull the exact number using `which.min()` again. The one difference from before is that we need to specify column 1 to get the error rate. This is the overall error rate and there will be additional columns for each error rate by the class label. We will not need them in this example. Also, `mse` is no longer available but rather `err.rate` is used instead, as follows:

```
> which.min(rf.biop$err.rate[, 1])
[1] 19
```

Only 19 trees are needed to optimize the model accuracy. Let's try this and see how it performs:

```
> set.seed(123)
> rf.biop.2 <- randomForest(class~ ., data = biop.train, ntree =
  19)
> print(rf.biop.2)
Call:
randomForest(formula = class ~ ., data = biop.train, ntree = 19)
Type of random forest: classification
```

```

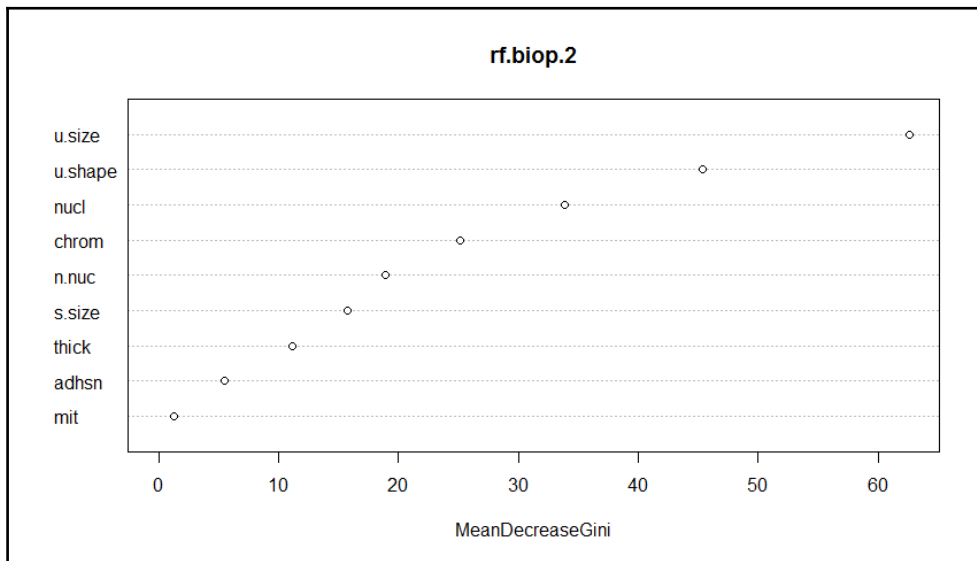
      Number of trees: 19
No. of variables tried at each split: 3
  OOB estimate of error rate: 2.95%
Confusion matrix:
      benign malignant class.error
benign   294         8 0.02649007
malignant    6      166 0.03488372
> rf.biop.test <- predict(rf.biop.2, newdata = biop.test, type =
  "response")
> table(rf.biop.test, biop.test$class)
rf.biop.test benign malignant
  benign    139         0
  malignant     3      67
> (139 + 67) / 209
[1] 0.9856459

```

Well, how about that? The `train` set error is below 3 percent, and the model even performs better on the `test` set where we had only three observations misclassified out of 209 and none were false positives. Recall that the best so far was with logistic regression with 97.6 percent accuracy. So this seems to be our best performer yet on the breast cancer data. Before moving on, let's have a look at the variable importance plot:

```
> varImpPlot(rf.biop.2)
```

The output of the preceding command is as follows:



The importance in the preceding plot is in each variable's contribution to the mean decrease in the Gini index. This is rather different from the splits of the single tree. Remember that the full tree had splits at the size (consistent with random forest), then nuclei, and then thickness. This shows how potentially powerful a technique building random forests can be, not only in the predictive ability, but also in feature selection.

Moving on to the tougher challenge of the Pima Indian diabetes model, we will first need to prepare the data in the following way:

```
> data(Pima.tr)
> data(Pima.te)
> pima <- rbind(Pima.tr, Pima.te)
> set.seed(502)
> ind <- sample(2, nrow(pima), replace = TRUE, prob = c(0.7, 0.3))
> pima.train <- pima[ind == 1, ]
> pima.test <- pima[ind == 2, ]
```

Now we will move on to the building of the model, as follows:

```
> set.seed(321)
> rf.pima = randomForest(type~., data=pima.train)
> rf.pima
Call:
randomForest(formula = type ~ ., data = pima.train)
Type of random forest: classification
Number of trees: 500
No. of variables tried at each split: 2
OOB estimate of error rate: 20%
Confusion matrix:
  No Yes class.error
No 233 29  0.1106870
Yes 48 75  0.3902439
```

We get a 20 per cent misclassification rate error, which is no better than what we've done before on the `train` set. Let's see if optimizing the tree size can improve things dramatically:

```
> which.min(rf.pima$serr.rate[, 1])
[1] 80
> set.seed(321)
> rf.pima.2 = randomForest(type~., data=pima.train, ntree=80)
> print(rf.pima.2)
Call:
randomForest(formula = type ~ ., data = pima.train, ntree = 80)
Type of random forest: classification
Number of trees: 80
No. of variables tried at each split: 2
```



```
OOB estimate of error rate: 19.48%
Confusion matrix:
No Yes class.error
No 230 32 0.1221374
Yes 43 80 0.3495935
```

At 80 trees in the forest, there is minimal improvement in the OOB error. Can random forest live up to the hype on the test data? We will see in the following way:

```
> rf.pima.test <- predict(rf.pima.2, newdata= pima.test,
  type = "response")
> table(rf.pima.test, pima.test$type)
rf.pima.test No Yes
      No 75 21
      Yes 18 33
> (75+33)/147
[1] 0.7346939
```

Well, we get only 73 percent accuracy on the test data, which is inferior to what we achieved using the SVM.

While random forest disappointed on the diabetes data, it proved to be the best classifier so far for the breast cancer diagnosis. Finally, we will move on to gradient boosting.

Extreme gradient boosting - classification

As mentioned previously, we will be using the `xgboost` package in this section, which we have already loaded. Given the method's well-earned reputation, let's try it on the diabetes data.

As stated in the boosting overview, we will be tuning a number of parameters:

- `nrounds`: The maximum number of iterations (number of trees in final model).
- `colsample_bytree`: The number of features, expressed as a ratio, to sample when building a tree. Default is 1 (100% of the features).
- `min_child_weight`: The minimum weight in the trees being boosted. Default is 1.
- `eta`: Learning rate, which is the contribution of each tree to the solution. Default is 0.3.
- `gamma`: Minimum loss reduction required to make another leaf partition in a tree.
- `subsample`: Ratio of data observations. Default is 1 (100%).
- `max_depth`: Maximum depth of the individual trees.



Using the `expand.grid()` function, we will build our experimental grid to run through the training process of the `caret` package. If you do not specify values for all of the preceding parameters, even if it is just a default, you will receive an error message when you execute the function. The following values are based on a number of training iterations I've done previously. I encourage you to try your own tuning values.

Let's build the grid as follows:

```
> grid = expand.grid(  
  nrounds = c(75, 100),  
  colsample_bytree = 1,  
  min_child_weight = 1,  
  eta = c(0.01, 0.1, 0.3), #0.3 is default,  
  gamma = c(0.5, 0.25),  
  subsample = 0.5,  
  max_depth = c(2, 3)  
)
```

This creates a grid of 24 different models that the `caret` package will run so as to determine the best tuning parameters. A note of caution is in order. On a dataset of the size that we will be working with, this process takes only a few seconds. However, in large datasets, this can take hours. As such, you must apply your judgment and experiment with smaller samples of the data in order to identify the tuning parameters, in case the time is of the essence, or you are constrained by the size of your hard drive.

Before using the `train()` function from the `caret` package, I would like to specify the `trainControl` argument by creating an object called `control`. This object will store the method that we want so as to train the tuning parameters. We will use the 5 fold cross-validation, as follows:

```
> cntrl = trainControl(  
  method = "cv",  
  number = 5,  
  verboseIter = TRUE,  
  returnData = FALSE,  
  returnResamp = "final"  
)
```

To utilize the `train.xgb()` function, just specify the formula as we did with the other models: the `train` dataset inputs, labels, method, train control, and experimental grid. Remember to set the random seed:

```
> set.seed(1)
> train.xgb = train(
  x = pima.train[, 1:7],
  y = ,pima.train[, 8],
  trControl = cntrl,
  tuneGrid = grid,
  method = "xgbTree"
)
```

Since in `trControl` I set `verboseIter` to `TRUE`, you should have seen each training iteration within each k-fold.

Calling the object gives us the optimal parameters and the results of each of the parameter settings, as follows (abbreviated for simplicity):

```
> train.xgb
eXtreme Gradient Boosting
No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 308, 308, 309, 308, 307
Resampling results across tuning parameters:
eta max_depth gamma nrounds Accuracy Kappa
0.01 2      0.25  75      0.7924286 0.4857249
0.01 2      0.25 100      0.7898321 0.4837457
0.01 2      0.50  75      0.7976243 0.5005362
.....
0.30 3      0.50  75      0.7870664 0.4949317
0.30 3      0.50 100      0.7481703 0.3936924
Tuning parameter 'colsample_bytree' was held constant at a
value of 1
Tuning parameter 'min_child_weight' was held constant at a
value of 1
Tuning parameter 'subsample' was held constant at a value of 0.5
Accuracy was used to select the optimal model using the largest
value.
The final values used for the model were nrounds = 75, max_depth =
2,
eta = 0.1, gamma = 0.5, colsample_bytree = 1, min_child_weight = 1
and subsample = 0.5.
```

This gives us the best combination of parameters to build a model. The accuracy in the training data was 81% with a Kappa of 0.55. Now it gets a little tricky, but this is what I've seen as best practice. First, create a list of parameters that will be used by the `xgboost` training function, `xgb.train()`. Then, turn the dataframe into a matrix of input features and a list of labeled numeric outcomes (0s and 1s). Then further, turn the features and labels into the input required, as `xgb.Dmatrix`. Try this:

```
> param <- list( objective = "binary:logistic",
  booster = "gbtree",
  eval_metric = "error",
  eta = 0.1,
  max_depth = 2,
  subsample = 0.5,
  colsample_bytree = 1,
  gamma = 0.5
)
> x <- as.matrix(pima.train[, 1:7])
> y <- ifelse(pima.train$type == "Yes", 1, 0)
> train.mat <- xgb.DMatrix(data = x, label = y)
```

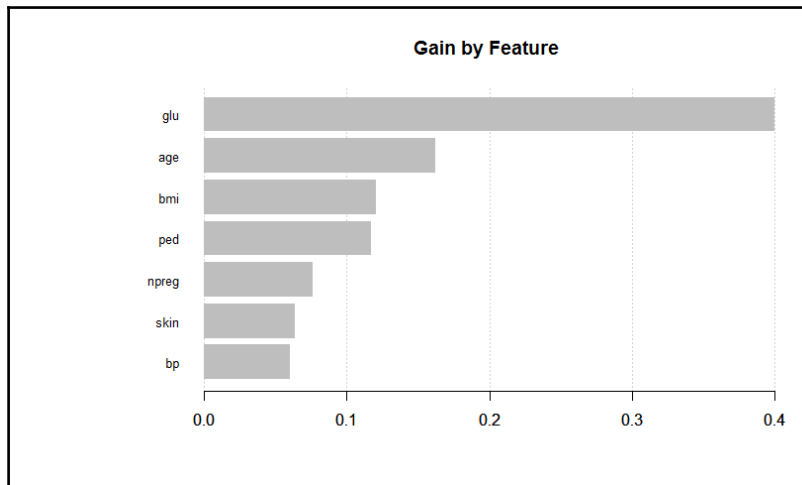
With all of that prepared, just create the model:

```
> set.seed(1)
> xgb.fit <- xgb.train(params = param, data = train.mat, nrounds =
  75)
```

Before seeing how it does on the test set, let's check the variable importance and also plot it. You can examine three items: **gain**, **cover**, and **frequency**. **Gain** is the improvement in accuracy that feature brings to the branches it is on. **Cover** is the relative number of total observations related to this feature. **Frequency** is the per cent of times that feature occurs in all of the trees. The following code produces the desired output:

```
> impMatrix <- xgb.importance(feature_names = dimnames(x)[[2]],
  model = xgb.fit)
> impMatrix
  Feature      Gain      Cover Frequency
1:  glu 0.40000548 0.31701688 0.24509804
2:  age 0.16177609 0.15685050 0.17156863
3:  bmi 0.12074049 0.14691325 0.14705882
4:  ped 0.11717238 0.15400331 0.16666667
5: npreg 0.07642333 0.05920868 0.06862745
6:  skin 0.06389969 0.08682105 0.10294118
7:   bp 0.05998254 0.07918634 0.09803922
> xgb.plot.importance(impMatrix, main = "Gain by Feature")
```

The output of the preceding command is as follows:



How does the feature importance compare to other methods?

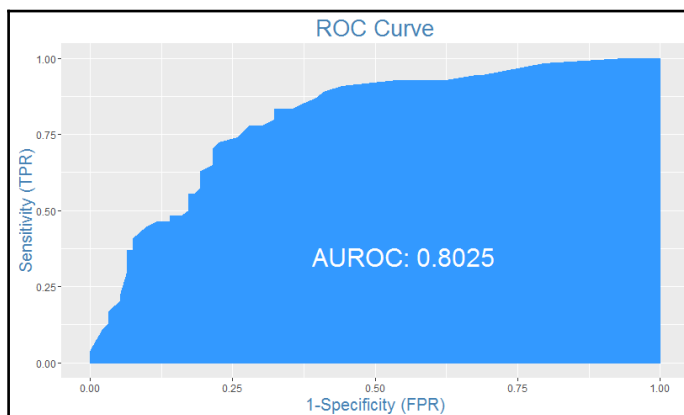
Here is how we see it performed on the test set, which like the training data must be in a matrix. Let's also bring in the tools from the `InformationValue` package to help our efforts. This code loads the library and produces some output to analyze model performance:

```
> library(InformationValue)
> pred <- predict(xgb.fit, x)
> optimalCutoff(y, pred)
[1] 0.3899574
> pima.testMat <- as.matrix(pima.test[, 1:7])
> xgb.pima.test <- predict(xgb.fit, pima.testMat)
> y.test <- ifelse(pima.test$type == "Yes", 1, 0)
> confusionMatrix(y.test, xgb.pima.test, threshold = 0.39)
  0 1
0 72 16
1 20 39
> 1 - misClassError(y.test, xgb.pima.test, threshold = 0.39)
[1] 0.7551
```

Did you notice what I did there with `optimalCutoff()`? Well, that function from `InformationValue` provides the optimal probability threshold to minimize error. By the way, the model error is around 25%. It's still not superior to our SVM model. As an aside, we see the ROC curve and the achievement of an AUC above 0.8. The following code produces the ROC curve:

```
> plotROC(y.test, xgb.pima.test)
```

The output of the code is as follows:



Model selection

Recall that our primary objective in this chapter was to use the tree-based methods to improve the predictive ability of the work done in the prior chapters. What did we learn? First, on the `prostate` data with a quantitative response, we were not able to improve on the linear models that we produced in Chapter 4, *Advanced Feature Selection in Linear Models*. Second, the random forest outperformed logistic regression on the Wisconsin Breast Cancer data of Chapter 3, *Logistic Regression and Discriminant Analysis*. Finally, and I must say disappointingly, we were not able to improve on the SVM model on the Pima Indian diabetes data with boosted trees.

As a result, we can feel comfortable that we have good models for the prostate and breast cancer problems. We will try one more time to improve the model for diabetes in Chapter 7, *Neural Networks and Deep Learning*. Before we bring this chapter to a close, I want to introduce the powerful method of feature elimination using random forest techniques.

Feature Selection with random forests

So far, we've looked at several feature selection techniques, such as regularization, best subsets, and recursive feature elimination. I now want to introduce an effective feature selection method for classification problems with Random Forests using the `Boruta` package. A paper is available that provides details on how it works in providing all relevant features:

Kursa M., Rudnicki W. (2010), *Feature Selection with the Boruta Package*, *Journal of Statistical Software*, 36(11), 1 - 13

What I will do here is provide an overview of the algorithm and then apply it to a wide dataset. This will not serve as a separate business case but as a template to apply the methodology. I have found it to be highly effective, but be advised it can be computationally intensive. That may seem to defeat the purpose, but it effectively eliminates unimportant features, allowing you to focus on building a simpler, more efficient, and more insightful model. It is time well spent.

At a high level, the algorithm creates **shadow attributes** by copying all the inputs and shuffling the order of their observations to decorrelate them. Then, a random forest model is built on all the inputs and a Z-score of the mean accuracy loss for each feature, including the shadow ones. Features with significantly higher Z-scores or significantly lower Z-scores than the shadow attributes are deemed **important** and **unimportant** respectively. The shadow attributes and those features with known importance are removed and the process repeats itself until all features are assigned an importance value. You can also specify the maximum number of random forest iterations. After completion of the algorithm, each of the original features will be labeled as **confirmed**, **tentative**, or **rejected**. You must decide on whether or not to include the tentative features for further modeling. Depending on your situation, you have some options:

- Change the random seed and rerun the methodology multiple (k) times and select only those features that are confirmed in all the k runs
- Divide your data (training data) into k folds, run separate iterations on each fold, and select those features which are confirmed for all the k folds

Note that all of this can be done with just a few lines of code. Let's have a look at the code, applying it to the `Sonar` data from the `mlbench` package. It consists of 208 observations, 60 numerical input features, and one vector of labels for classification. The labels are factors where, `R` if the `sonar` object is a rock and `M` if it is a mine. The first thing to do is load the data and do a quick, very quick, data exploration:

```
> data(Sonar, package="mlbench")
> dim(Sonar)
[1] 208 61
> table(Sonar$Class)
  M  R
111 97
```

To run the algorithm, you just need to load the `Boruta` package and create a formula in the `boruta()` function. Keep in mind that the labels must be and as a factor, or the algorithm will not work. If you want to track the progress of the algorithm, specify `doTrace = 1`. Also, don't forget to set the random seed:

```
> library(Boruta)
> set.seed(1)
> feature.selection <- Boruta(Class ~ ., data = Sonar, doTrace = 1)
```

As mentioned in the previous section, this can be computationally intensive. Here is how long it took on my old-fashioned laptop:

```
> feature.selection$timeTaken
Time difference of 25.78468 secs
```

A simple table will provide the count of the final importance decision. We see that we could safely eliminate half of the features:

```
> table(feature.selection$finalDecision)
Tentative Confirmed Rejected
      12       31       17
```

Using these results, it is simple to create a new dataframe with our selected features. We start out using the `getSelectedAttributes()` function to capture the feature names. In this example, let's only select those that are confirmed. If we wanted to include confirmed and tentative, we just specify `withTentative = TRUE` in the function:

```
> fNameames <- getSelectedAttributes(feature.selection) # withTentative =
TRUE
> fNameames
[1] "V1" "V4" "V5" "V9" "V10" "V11" "V12" "V13" "V15" "V16"
[11] "V17" "V18" "V19" "V20" "V21" "V22" "V23" "V27" "V28" "V31"
[21] "V35" "V36" "V37" "V44" "V45" "V46" "V47" "V48" "V49" "V51"
```



```
[31] "V52"
```

Using the feature names, we create our subset of the Sonar data:

```
> Sonar.features <- Sonar[, fNames]
> dim(Sonar.features)
[1] 208 31
```

There you have it! The `Sonar.features` dataframe includes all the confirmed features from the boruta algorithm. It can now be subjected to further meaningful data exploration and analysis. A few lines of code and some patience as the algorithm does its job can significantly improve your modeling efforts and insight generation.

Summary

In this chapter, you learned both the power and limitations of tree-based learning methods for both classification and regression problems. Single trees, while easy to build and interpret, may not have the necessary predictive power for many of the problems that we are trying to solve. To improve on the predictive ability, we have the tools of random forest and gradient-boosted trees at our disposal. With random forest, hundreds or even thousands of trees are built and the results aggregated for an overall prediction. Each tree of the random forest is built using a sample of the data called bootstrapping as well as a sample of the predictive variables. As for gradient boosting, an initial, and a relatively small, tree is produced. After this initial tree is built, subsequent trees are produced based on the residuals/misclassifications. The intended result of such a technique is to build a series of trees that can improve on the weakness of the prior tree in the process, resulting in decreased bias and variance. We also saw that in R, one can utilize random forests as a feature selection method.

While these methods are extremely powerful, they are not some sort of nostrum in the world of machine learning. Different datasets require judgment on the part of the analyst as to which techniques are applicable. The techniques to be applied to the analysis and the selection of the tuning parameters are equally important. This fine tuning can make all the difference between a good predictive model and a great predictive model.

In the next chapter, we turn our attention to using R to build neural networks and deep learning models.

7

Neural Networks and Deep Learning

"Forget artificial intelligence - in the brave new world of big data, it's artificial idiocy we should be looking out for."

- Tom Chatfield

I recall that at some meeting circa mid-2012, I was part of a group discussing the results of some analysis or other, when one of the people around the table sounded off with a hint of exasperation mixed with a tinge of fright, *this isn't one of those neural networks, is it?* I knew of his past run-ins with and deep-seated anxiety about neural networks, so I assuaged his fears making some sarcastic comment that neural networks have basically gone the way of the dinosaur. No one disagreed! Several months later, I was gobsmacked when I attended a local meeting where the discussion focused on, of all things, neural networks and this mysterious deep learning. Machine learning pioneers such as Ng, Hinton, Salakhutdinov, and Bengio have revived neural networks and improved their performance.

Much media hype revolves around these methods with high-tech companies such as Facebook, Google, and Netflix investing tens, if not hundreds, of millions of dollars. The methods have yielded promising results in voice recognition, image recognition, and automation. If self-driving cars ever stop running off the road and into each other, it will certainly be from the methods discussed here.

In this chapter, we will discuss how the methods work, their benefits, and inherent drawbacks so that you can become conversationally competent about them. We will work through a practical business application of a neural network. Finally, we will apply the deep learning methodology in a cloud-based application.

Introduction to neural networks

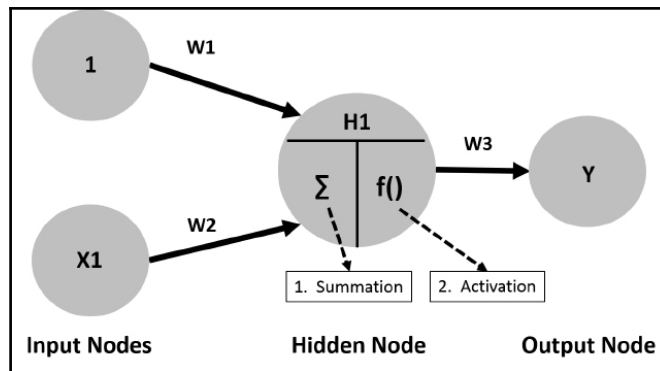
Neural network is a fairly broad term that covers a number of related methods, but in our case, we will focus on a **feed forward** network that trains with **backpropagation**. I'm not going to waste our time discussing how the machine learning methodology is similar or dissimilar to how a biological brain works. We only need to start with a working definition of what a neural network is. I think the Wikipedia entry is a good start.

In machine learning and cognitive science, **Artificial neural networks** (ANNs) are a family of statistical learning models inspired by biological neural networks (the central nervous systems of animals, in particular, the brain) and are used to estimate or approximate functions that can depend on a large number of inputs and are generally unknown. https://en.wikipedia.org/wiki/Artificial_neural_network

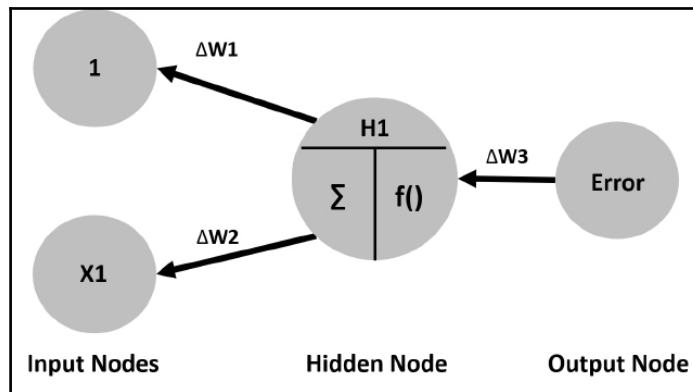
The motivation or benefit of ANNs is that they allow the modeling of highly complex relationships between inputs/features and response variable(s), especially if the relationships are highly nonlinear. No underlying assumptions are required to create and evaluate the model, and it can be used with qualitative and quantitative responses. If this is the yin, then the yang is the common criticism that the results are black box, which means that there is no equation with the coefficients to examine and share with the business partners. In fact, the results are almost not interpretable. The other criticisms revolve around how results can differ by just changing the initial random inputs and that training ANNs is computationally expensive and time-consuming.

The mathematics behind ANNs is not trivial by any measure. However, it is crucial to at least get a working understanding of what is happening. A good way to intuitively develop this understanding is to start a diagram of a simplistic neural network.

In this simple network, the inputs or covariates consist of two nodes or neurons. The neuron labeled **1** represents a constant or more appropriately, the intercept. **X1** represents a quantitative variable. The **W**'s represent the weights that are multiplied by the input node values. These values become **Input Nodes** to **Hidden Node**. You can have multiple hidden nodes, but the principal of what happens in just this one is the same. In the hidden node, **H1**, the *weight * value* computations are summed. As the intercept is notated as **1**, then that input value is simply the weight, **W1**. Now the magic happens. The summed value is then transformed with the **Activation** function, turning the input signal to an output signal. In this example, as it is the only **Hidden Node**, it is multiplied by **W3** and becomes the estimate of **Y**, our response. This is the feed-forward portion of the algorithm:



But wait, there's more! To complete the cycle or epoch, as it is known, backpropagation happens and trains the model based on what was learned. To initiate the backpropagation, an error is determined based on a loss function such as **Sum of Squared Error** or **Cross-Entropy**, among others. As the weights, $W1$ and $W2$, were set to some initial random values between $[-1, 1]$, the initial error may be high. Working backward, the weights are changed to minimize the error in the loss function. The following diagram portrays the backpropagation portion:



This completes one epoch. This process continues, using gradient descent (discussed in Chapter 5, *More Classification Techniques - K-Nearest Neighbors and Support Vector Machines*) until the algorithm converges to the minimum error or prespecified number of epochs. If we assume that our activation function is simply linear, in this example, we would end up with $Y = W3(W1(1) + W2(X1))$.

The networks can get complicated if you add numerous input neurons, multiple neurons in a hidden node, and even multiple hidden nodes. It is important to note that the output from a neuron is connected to all the subsequent neurons and has weights assigned to all these connections. This greatly increases the model complexity. Adding hidden nodes and increasing the number of neurons in the hidden nodes has not improved the performance of ANNs as we had hoped. Thus, the development of deep learning occurs, which in part relaxes the requirement of all these neuron connections.

There are a number of activation functions that one can use/try, including a simple linear function, or for a classification problem, the `sigmoid` function, which is a special case of the logistic function (Chapter 3, *Logistic Regression and Discriminant Analysis*). Other common activation functions are Rectifier, Maxout, and **hyperbolic tangent (tanh)**.

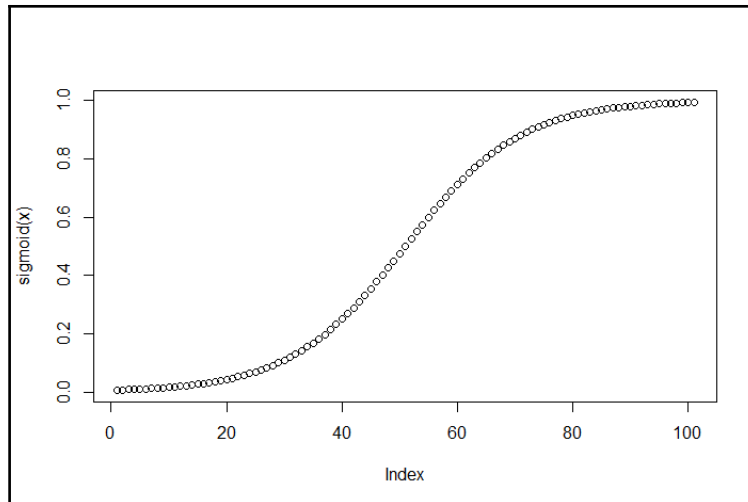
We can plot a `sigmoid` function in R, first creating an R function in order to calculate the `sigmoid` function values:

```
> sigmoid = function(x) {  
  1 / ( 1 + exp(-x) )  
}
```

Then, it is a simple matter to plot the function over a range of values, say -5 to 5:

```
> x <- seq(-5, 5, .1)  
> plot(sigmoid(x))
```

The output of the preceding command is as follows:



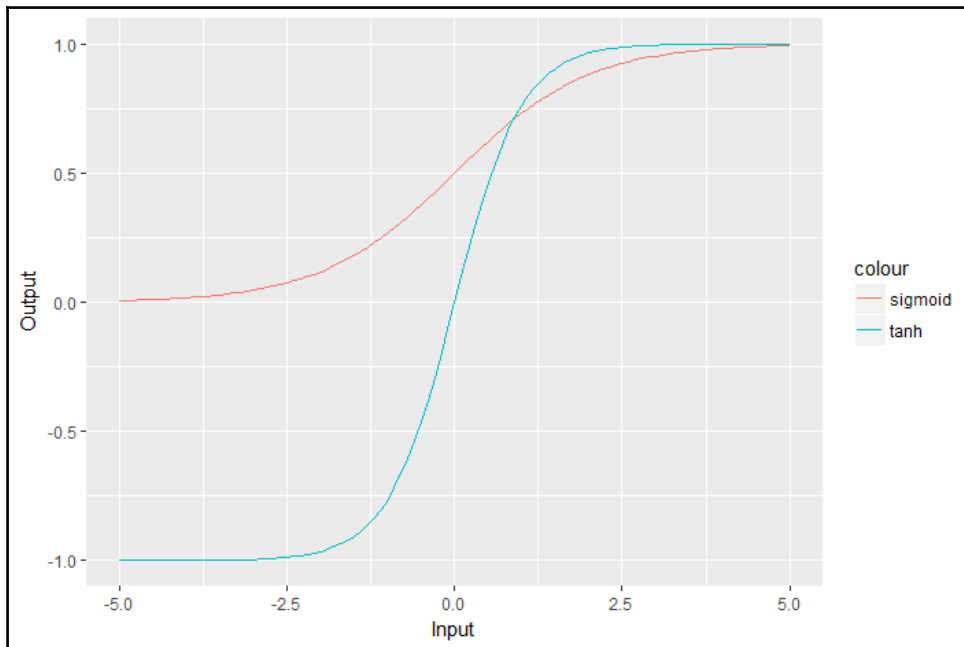
The `tanh` function (hyperbolic tangent) is a rescaling of the logistic `sigmoid` with the output between **-1** and **1**. The `tanh` function relates to `sigmoid` as follows, where **x** is the `sigmoid` function:

$$\tanh(x) = 2 * \text{sigmoid}(2x) - 1$$

Let's plot the `tanh` and `sigmoid` functions for comparison purposes. Let's also use `ggplot`:

```
> library(ggplot2)
> s <- sigmoid(x)
> t <- tanh(x)
> z <- data.frame(cbind(x, s, t))
> ggplot(z, aes(x)) +
  geom_line(aes(y = s, color = "sigmoid")) +
  geom_line(aes(y = t, color = "tanh"))
```

The output of the preceding command is as follows:



So why use the `tanh` function versus `sigmoid`? It seems there are many opinions on the subject; is `tanh` popular in neural networks? In short, assuming you have scaled data with mean 0 and variance 1, the `tanh` function permits weights that are on average close to zero (zero-centered). This helps in avoiding bias and improves convergence. Think about the implications of always having positive weights from an output neuron to an input neuron as in a `sigmoid` function activation. During backpropagation, the weights will become either all positive or all negative between layers. This may cause performance issues. Also, since the gradient at the tails of a `sigmoid` (0 and 1) are almost zero, during backpropagation it can happen that almost no signal will flow between neurons of different layers. A full discussion of the issue is available, LeCun (1998). Keep in mind it is not a foregone conclusion that `tanh` is always better.

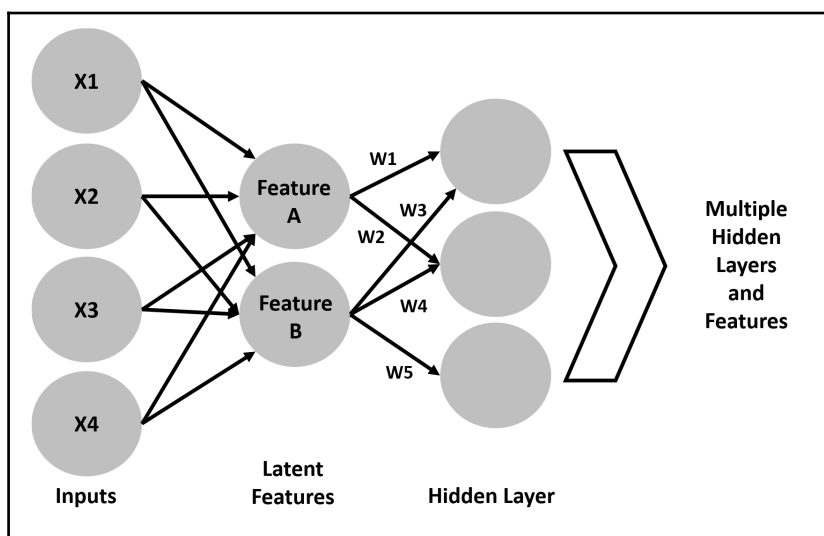
This all sounds fascinating, but the ANN almost went the way of disco as it just did not perform as well as advertised, especially when trying to use deep networks with many hidden layers and neurons. It seems that a slow yet gradual revival came about with the seminal paper by Hinton and Salakhutdinov (2006) in the reformulated and, dare I say, rebranded neural network, deep learning.

Deep learning, a not-so-deep overview

So, what is this deep learning that is grabbing our attention and headlines? Let's turn to Wikipedia again for a working definition: *Deep learning is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using model architectures, with complex structures or otherwise, composed of multiple nonlinear transformations.* That sounds as if a lawyer wrote it. The characteristics of deep learning are that it is based on ANNs where the machine learning techniques, primarily unsupervised learning, are used to create new features from the input variables. We will dig into some unsupervised learning techniques in the next couple of chapters, but one can think of it as finding structure in data where no response variable is available. A simple way to think of it is the **Periodic Table of Elements**, which is a classic case of finding structure where no response is specified. Pull up this table online and you will see that it is organized based on atomic structure, with metals on one side and non-metals on the other. It was created based on latent classification/structure. This identification of latent structure/hierarchy is what separates deep learning from your run-of-the-mill ANN. Deep learning sort of addresses the question whether there is an algorithm that better represents the outcome than just the raw inputs. In other words, can our model learn to classify pictures other than with just the raw pixels as the only input? This can be of great help in a situation where you have a small set of labeled responses but a vast amount of unlabeled input data. You could train your deep learning model using unsupervised learning and then apply this in a supervised fashion to the labeled data, iterating back and forth.

Identification of these latent structures is not trivial mathematically, but one example is the concept of regularization that we looked at in Chapter 4, *Advanced Feature Selection in Linear Models*. In deep learning, one can penalize weights with regularization methods such as *L1* (penalize non-zero weights), *L2* (penalize large weights), and dropout (randomly ignore certain inputs and zero their weight out). In standard ANNs, none of these regularization methods takes place.

Another way is to reduce the dimensionality of the data. One such method is the *autoencoder*. This is a neural network where the inputs are transformed into a set of reduced dimension weights. In the following diagram, notice that **Feature A** is not connected to one of the hidden nodes:



This can be applied recursively and learning can take place over many hidden layers. What you have seen happening in this case is that the network is developing features of features as they are stacked on each other. Deep learning will learn the weights between two layers in sequence first and then only use backpropagation in order to fine-tune these weights. Other feature selection methods include **Restricted Boltzmann Machine** and **Sparse Coding Model**.

The details are beyond our scope, and many resources are available to learn about the specifics. Here are a couple of starting points:

<http://www.cs.toronto.edu/~hinton/>

<http://deeplearning.net/>

Deep learning has performed well on many classification problems, including winning a Kaggle contest or two. It still suffers from the problems of ANNs, especially the black box problem. Try explaining to the uninformed what is happening inside a neural network. However, it is appropriate for problems where an explanation of How is not a problem and the important question is What. After all, do we really care why an autonomous car avoided running into a pedestrian, or do we care about the fact that it did not? Additionally, the Python community has a bit of a head start on the R community in deep learning usage and packages. As we will see in the practical exercise, the gap is closing.

While deep learning is an exciting undertaking, be aware that to achieve the full benefit of its capabilities, you will need a high degree of computational power along with taking the time to train the best model by fine-tuning the hyperparameters. Here is a list of some that you will need to consider:

- An activation function
- Size and number of the hidden layers
- Dimensionality reduction, that is, Restricted Boltzmann versus autoencoder
- The number of epochs
- The gradient descent learning rate
- The loss function
- Regularization

Deep learning resources and advanced methods

One of the more interesting visual tools you can use for both learning and explaining is the interactive widget provided by TensorFlowTM: <http://playground.tensorflow.org/>. This tool allows you to explore, or **tinker**, as the site calls it, the various parameters and how they impact on the response, be it a classification problem or a regression problem. I could spend, well I have spent hours tinkering with it.



Here is an interesting task: create your own experimental design and see how the various parameters affect your prediction.

At this point, it seems that the two fastest growing deep learning open-source tools are TensorFlow™ and MXNet. I still prefer working with the package we will see, `h2o`, but it is important to understand and learn the latest techniques. You can access TensorFlow™ with R, but it requires you to install python first. This series of tutorials will walk you through how to get it up and running:

<https://rstudio.github.io/tensorflow/>.

MXNet does not require the installation of Python and is relatively easy to install and make operational. It also offers a number of pretrained models that allow you to start making predictions quickly. Several R tutorials are available:

<http://mxnet.io/>.

I now want to take the time to enumerate some of the variations of deep neural networks along with the learning tasks where they have performed well.

Convolutional neural networks (CNN) make the assumption that the inputs are images and create features from slices or small portions of the data, which are combined to create a feature map. Think of these small slices as filters or probably more appropriately, kernels that the network learns during training. The activation function for CNN is a **Rectified Linear Unit (ReLU)**. It is simply $f(x) = \max(0, x)$, where x is the input to the neuron. CNNs perform well on image classification, object detection, and even sentence classification.

Recurrent neural networks (RNN) are created to make use of sequential information. In traditional neural networks, the inputs and outputs are independent of each other. With RNN, the output is dependent on the computations of previous layers, permitting information to persist across layers. So, take an output from a neuron (y); it is calculated not only on its input (t) but on all previous layers ($t-1, t-n...$). It is effective at handwriting and speech detection.

Long Short-Term Memory (LSTM) is a special case of RNN. The problem with RNN is that it does not perform well on data with long signals. Thus LSTMs were created to capture complex patterns in data. RNNs combine information during training from previous steps in the same way, regardless of the fact that information in one step is more or less valuable than other steps. LSTMs seek to overcome this limitation by deciding what to remember at each step during training. This multiplication of a weight matrix by the data vector is referred to as a gate, which acts as an information filter. A neuron in LSTM will have two inputs and two outputs. The input from prior outputs and the memory vector passed from the previous gate. Then, it produces the output values and output memory as inputs to the next layer. LSTMs have the limitation of requiring a healthy dose of training data and are computationally intensive. LSTMs have performed well on speech recognition problems.



I recommend you work with the tutorials on MXNet to help you understand how to develop these models for your own use.

With that, let's move on to some practical applications.

Business understanding

It was a calm, clear night on the 20th of April, 1998. I was a student pilot in a Hughes 500D helicopter on a cross-country flight from the St. Paul, MN downtown airport back home to good old Grand Forks, ND. The flight was my final requirement prior to taking the test to achieve a helicopter instrument rating. My log book shows that we were 35 **Distance Measuring Equipment (DME)** or 35 nautical miles from the VOR on Airway Victor 2. This put us somewhere south/southeast of St. Cloud, MN, cruising along at what I recall was 4,500 feet above sea level at approximately 120 knots. Then, it happened...BOOOOM! It is not hyperbole to say that it was a thunderous explosion, followed by a hurricane blast of wind to the face.

It all started when my flight instructor asked a mundane question about our planned instrument approach into Alexandria, MN. We swapped control of the aircraft and I bent over to consult the instrument approach plate on my kneeboard. As I snapped on the red lens flashlight, the explosion happened. Given my face-down orientation, the sound, and ensuing blast of wind, several thoughts crossed my mind: the helicopter is falling apart, I'm plunging to my death, and the Space Shuttle Challenger explosion as an HD quality movie going off in my head. In the 1.359 seconds that it took us to stop screaming, we realized that the Plexiglas windscreen in front of me was essentially gone, but everything else was good to go. After slowing the craft, a cursory inspection revealed that the cockpit was covered in blood, guts, and feathers. We had done the improbable by hitting a Mallard duck over Central Minnesota and in the process, destroyed the windscreen. Had I not been looking at my kneeboard, I would have been covered in pate. We simply declared an emergency and canceled our flight plan with Minneapolis Center and, like the Memphis Belle, limped our way into Alexandria to await rescue from our compatriots at the University of North Dakota (home of the Fighting Sioux).

So what? Well, I wanted to point out how much of a NASA fan and astronaut I am. In a terrifying moment, where for a split second I thought that I was checking out, my mind drifted to the Space Shuttle. Most males my age wanted to shake the hands of George Brett or Wayne Gretzky. I wanted to, and in fact did, shake the hands of Buzz Aldrin. (he was after all on the North Dakota faculty at the time.) Thus, when I found the `shuttle` dataset in the `MASS` package, I had to include it in this tome. By the way, if you ever get the chance to see the Space Shuttle Atlantis display at Kennedy Space Center, do not miss it.

For this problem, we will try and develop a neural network to answer the question of whether or not the shuttle should use the autoland system. The default decision is to let the crew land the craft. However, the autoland capability may be required for situations of crew incapacitation or adverse effects of gravity upon re-entry after extended orbital operations. This data is based on computer simulations, not actual flights. In reality, the autoland system went through some trials and tribulations and, for the most part, the shuttle astronauts were in charge during the landing process. Here are a couple of links for further background information:

<http://www.spaceref.com/news/viewsr.html?pid=10518>

<https://waynehale.wordpress.com/2011/03/11/breaking-through/>

Data understanding and preparation

To start, we will load these four packages. The data is in the `MASS` package:

```
> library(caret)
> library(MASS)
> library(neuralnet)
> library(vcd)
```

The `neuralnet` package will be used for the building of the model and `caret` for the data preparation. The `vcd` package will assist us in data visualization. Let's load the data and examine its structure:

```
> data(shuttle)
> str(shuttle)
'data.frame': 256 obs. of 7 variables:
 $ stability: Factor w/ 2 levels "stab", "xstab": 2 2 2 2 2 2 2
 2 2 2 ...
 $ error    : Factor w/ 4 levels "LX", "MM", "SS", ...: 1 1 1 1 1 1 1
 1 1 ...
 $ sign     : Factor w/ 2 levels "nn", "pp": 2 2 2 2 2 2 1 1 1 1 ...
 $ wind     : Factor w/ 2 levels "head", "tail": 1 1 1 2 2 2 1 1 1 2
 ...
```

```
$ magn      : Factor w/ 4 levels "Light","Medium",...: 1 2 4 1 2 4 1
  2 4 1 ...
$ vis       : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1
  ...
$ use       : Factor w/ 2 levels "auto","noauto": 1 1 1 1 1 1 1 1
  1 ...
```

The data consists of 256 observations and 7 variables. Notice that all of the variables are categorical and the response is `use` with two levels, `auto` and `noauto`. The covariates are as follows:

- `stability`: This is stable positioning or not (`stab/xstab`)
- `error`: This is the size of the error (`MM / SS / LX`)
- `sign`: This is the sign of the error, positive or negative (`pp/nn`)
- `wind`: This is the wind sign (`head / tail`)
- `magn`: This is the wind strength (`Light / Medium / Strong / Out of Range`)
- `vis`: This is the visibility (`yes / no`)

We will build a number of tables to explore the data, starting with the response/outcome:

```
> table(shuttle$use)
auto noauto
145    111
```

Almost 57 per cent of the time, the decision is to use the autolander. There are a number of possibilities to build tables for categorical data. The `table()` function is perfectly adequate to compare one with another, but if you add a third, it can turn into a mess to look at. The `vcd` package offers a number of table and plotting functions. One is `structable()`. This function will take a formula (`column1 + column2 ~ column3`), where `column3` becomes the rows in the table:

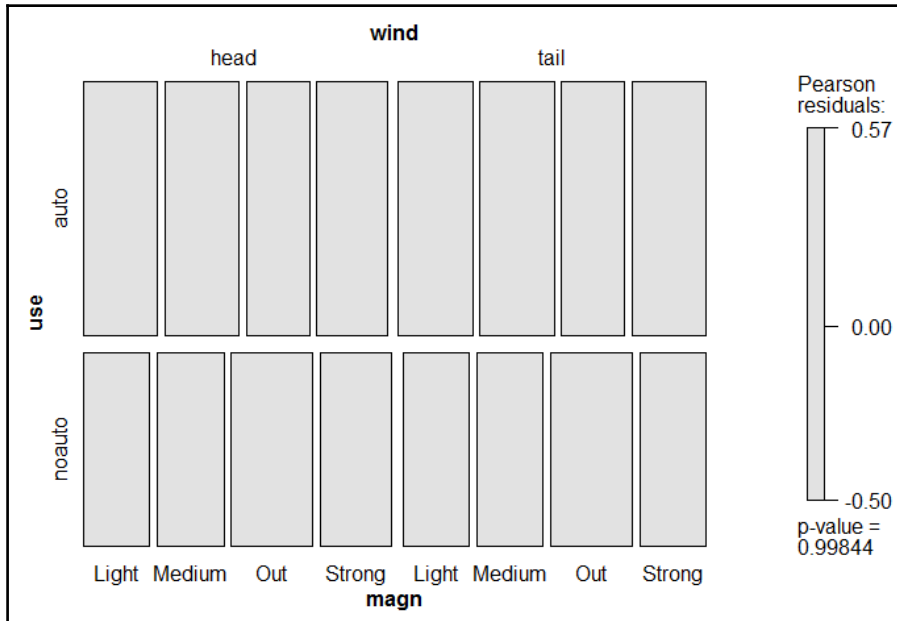
```
> table1 <- structable(wind + magn ~ use, shuttle)
> table1
```

	wind	head				tail				
	magn	Light	Medium	Out	Strong	Light	Medium	Out	Strong	
use										
auto		19	19	16	18	19	19	16	19	
noauto		13	13	16	14	13	13	16	13	

Here, we can see that in the cases of a headwind that was `Light` in magnitude, `auto` occurred 19 times and `noauto`, 13 times. The `vcd` package offers the `mosaic()` function to plot the table created by `structable()` and provide the **p-value** for a chi-squared test:

```
> mosaic(table1, shading = T)
```

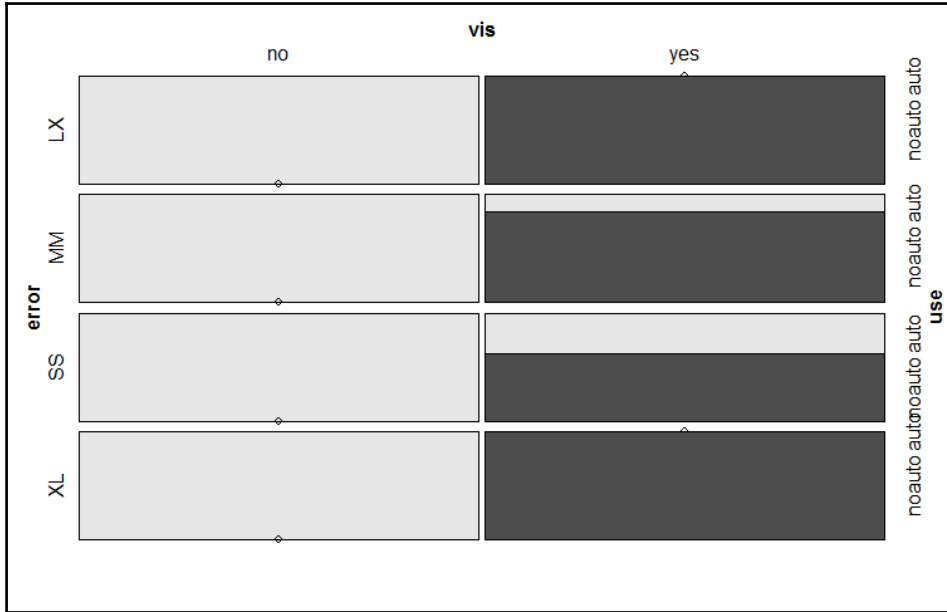
The output of the preceding command is as follows:



The plot tiles correspond to the proportional size of their respective cells in the table, created by recursive splits. You can also see that the **p-value** is not significant, so the variables are independent, which means that knowing the levels of wind and/or **magn** does not help us predict the use of the autolander. You do not need to include a `structable()` object in order to create the plot as it will accept a formula just as well:

```
> mosaic(use ~ error + vis, shuttle)
```

The output of the preceding command is as follows:



Note that the shading of the table has changed, reflecting the rejection of the null hypothesis and dependence in the variables. The plot first takes and splits the visibility. The result is that if the visibility is **no**, then the autolander is used. The next split is horizontal by **error**. If **error** is **SS** or **MM** when **vis** is **no**, then the autolander might be recommended, otherwise it is not. A p-value is not necessary as the gray shading indicates significance.

One can also examine proportional tables with the `prop.table()` function as a wrapper around `table()`:

```
> table(shuttle$use, shuttle$stability)
      stab xstab
auto      81   64
noauto    47   64
> prop.table(table(shuttle$use, shuttle$stability))
      stab      xstab
auto  0.3164062 0.2500000
noauto 0.1835938 0.2500000
```

In case we forget, the chi-squared tests are quite simple:

```
> chisq.test(shuttle$use, shuttle$stability)
Pearson's Chi-squared test with Yates' continuity
correction
data:  shuttle$use and shuttle$stability
X-squared = 4.0718, df = 1, p-value = 0.0436
```

Preparing the data for a neural network is very important as all the covariates and responses need to be numeric. In our case, all of the input features are categorical. However, the `caret` package allows us to quickly create dummy variables as our input features:

```
> dummies <- dummyVars(use ~ .,shuttle, fullRank = T)
> dummies
Dummy Variable Object
Formula: use ~ .
7 variables, 7 factors
Variables and levels will be separated by '.'
A full rank encoding is used
```

To put this into a data frame, we need to predict the `dummies` object to an existing data, either the same or different, in `as.data.frame()`. Of course, the same data is needed here:

```
> shuttle.2 = as.data.frame(predict(dummies, newdata=shuttle))

> names(shuttle.2)
[1] "stability.xstab" "error.MM"      "error.SS"
[4] "error.XL"        "sign.pp"       "wind.tail"
[7] "magn.Medium"     "magn.Out"      "magn.Strong"
[10] "vis.yes"

> head(shuttle.2)
  stability.xstab error.MM error.SS error.XL sign.pp wind.tail
1              1         0         0         0         1         0
2              1         0         0         0         1         0
3              1         0         0         0         1         0
4              1         0         0         0         1         1
5              1         0         0         0         1         1
6              1         0         0         0         1         1
  magn.Medium magn.Out magn.Strong vis.yes
1           0         0           0         0
2           1         0           0         0
3           0         0           1         0
4           0         0           0         0
5           1         0           0         0
6           0         0           1         0
```


We now have an input feature space of ten variables. Stability is now either 0 for `stab` or 1 for `xstab`. The base error is `LX`, and three variables represent the other categories.

The response can be created using the `ifelse()` function:

```
> shuttle.2$use <- ifelse(shuttle$use == "auto", 1, 0)
> table(shuttle.2$use)
 0    1
111 145
```

The `caret` package also provides us with the functionality to create the `train` and `test` sets. The idea is to index each observation as `train` or `test` and then split the data accordingly. Let's do this with a 70/30 `train` to `test` split, as follows:

```
> set.seed(123)
> trainIndex <- createDataPartition(shuttle.2$use, p = .7, list =
  FALSE)
```

The values in `trainIndex` provide us with the row number; in our case, 70 per cent of the total row numbers in `shuttle.2`. It is now a simple case of creating the `train/test` datasets:

```
> shuttleTrain <- shuttle.2[trainIndex, ]
> shuttleTest  <- shuttle.2[-trainIndex, ]
```

Nicely done! We are now ready to begin building the neural networks.

Modeling and evaluation

As mentioned, the package that we will use is `neuralnet`. The function in `neuralnet` will call for the use of a formula as we used elsewhere, such as $y \sim x_1 + x_2 + x_3 + x_4$, `data = df`. In the past, we used `y~`, to specify all the other variables in the data as inputs. However, `neuralnet` does not accommodate this at the time of writing. The way around this limitation is to use the `as.formula()` function. After first creating an object of the variable names, we will use this as an input in order to paste the variables properly on the right side of the equation:

```
> n <- names(shuttleTrain)
> form <- as.formula(paste("use ~", paste(n[!n %in% "use"],
  collapse = " + ")))
> form
use ~ stability.xstab + error.MM + error.SS + error.XL + sign.pp +
  wind.tail
  + magn.Medium + magn.Out + magn.Strong + vis.yes
```

Keep this function in mind for your own use as it may come in quite handy. In the `neuralnet` package, the function that we will use is appropriately named `neuralnet()`. Other than the formula, there are four other critical arguments that we will need to examine:

- `hidden`: This is the number of hidden neurons in each layer, which can be up to three layers; the default is 1
- `act.fct`: This is the activation function with the default logistic and `tanh` available
- `err.fct`: This is the function used to calculate the error with the default `sse`; as we are dealing with binary outcomes, we will use `ce` for cross-entropy
- `linear.output`: This is a logical argument on whether or not to ignore `act.fct` with the default `TRUE`, so for our data, this will need to be `FALSE`

You can also specify the algorithm. The default is resilient with backpropagation and we will use it along with the default of one hidden neuron:

```
> fit <- neuralnet(form, data = shuttleTrain, err.fct = "ce",  
  linear.output = FALSE)
```

Here are the overall results:

```
> fit$result.matrix  
1  
error                      0.009928587504  
reached.threshold          0.009905188403  
steps                      660.000000000000  
Intercept.to.1layhid1      -4.392654985479  
stability.xstab.to.1layhid1 1.957595172393  
error.MM.to.1layhid1       -1.596634090134  
error.SS.to.1layhid1       -2.519372079568  
error.XL.to.1layhid1       -0.371734253789  
sign.pp.to.1layhid1        -0.863963659357  
wind.tail.to.1layhid1      0.102077456260  
magn.Medium.to.1layhid1    -0.018170137582  
magn.Out.to.1layhid1       1.886928834123  
magn.Strong.to.1layhid1    0.140129588700  
vis.yes.to.1layhid1       6.209014123244  
Intercept.to.use           30.721652703205  
1layhid.1.to.use          -65.084168998463
```

We can see that the error is extremely low at 0.0099. The number of steps required for the algorithm to reach the threshold, which is when the absolute partial derivatives of the error function become smaller than this error (default = 0.1). The highest weight of the first neuron is `vis.yes.to.1layhid1` at 6.21.

You can also look at what are known as generalized weights. According to the authors of the `neuralnet` package, the generalized weight is defined as the contribution of the i th covariate to the log-odds:

The generalized weight expresses the effect of each covariate x_i and thus has an analogous interpretation as the i th regression parameter in regression models. However, the generalized weight depends on all other covariates (Gunther and Fritsch, 2010).



The weights can be called and examined. I've abbreviated the output to the first four variables and six observations only. Note that if you sum each row, you will get the same number, which means that the weights are equal for each covariate combination. Please note that your results might be slightly different because of random weight initialization.

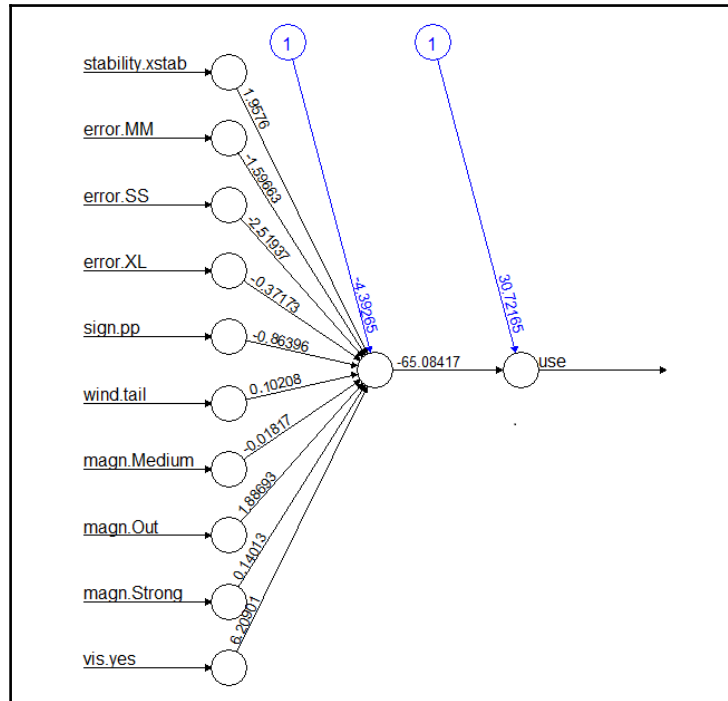
The results are as follows:

```
> head(fit$generalized.weights[[1]])
      [,1]      [,2]      [,3]      [,4]
1 -4.374825405  3.568151106  5.630282059  0.8307501368
2 -4.301565756  3.508399808  5.535998871  0.8168386187
6 -5.466577583  4.458595039  7.035337605  1.0380665866
9 -10.595727733  8.641980909  13.636415225  2.0120579565
10 -10.270199330  8.376476707  13.217468969  1.9502422861
11 -10.117466745  8.251906491  13.020906259  1.9212393878
```

To visualize the neural network, simply use the `plot()` function:

```
> plot(fit)
```

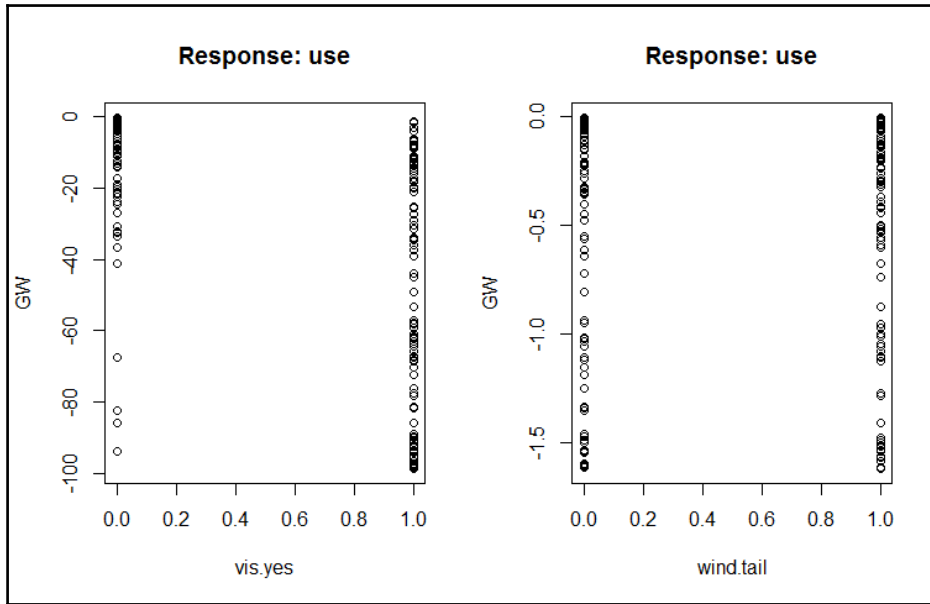
The following is the output of the preceding command:



This plot shows the weights of the variables and intercepts. You can also examine the generalized weights in a plot. Let's look at `vis.yes` versus `wind.tail`, which has a low overall synaptic weight. Notice how `vis.yes` is skewed and `wind.tail` has an even distribution of weights, implying little predictive power:

```
> par(mfrow = c(1, 2))
> gwplot(fit, selected.covariate = "vis.yes")
> gwplot(fit, selected.covariate = "wind.tail")
```

The following is the output of the preceding commands:



We now want to see how well the model performs. This is done with the `compute()` function and specifying the fit model and covariates. This syntax will be the same for the predictions on the `test` and `train` sets. Once computed, a list of the predictions is created with `$net.result`:

```
> resultsTrain <- compute(fit, shuttleTrain[, 1:10])
> predTrain <- resultsTrain$net.result
```

These results are in probabilities, so let's turn them into 0 or 1 and follow this up with a confusion matrix:

```
> predTrain <- ifelse(predTrain >= 0.5, 1, 0)
> table(predTrain, shuttleTrain$use)
predTrain 0 1
0 81 0
1 0 99
```

Lo and behold, the neural network model has achieved 100 per cent accuracy. We will now hold our breath and see how it does on the `test` set:

```
> resultsTest <- compute(fit, shuttleTest[,1:10])
> predTest <- resultsTest$net.result
> predTest <- ifelse(predTest >= 0.5, 1, 0)
> table(predTest, shuttleTest$use)
predTest  0  1
          0 29  0
          1  1 46
```

Only one false positive in the `test` set. If you wanted to identify which one this was, use the `which()` function to single it out, as follows:

```
> which(predTest == 1 & shuttleTest$use == 0)
[1] 62
```

It is row 62 in the `test` set and observation 203 in the full dataset.

I'll leave it to you to see if you can build a neural network that achieves 100% accuracy!

An example of deep learning

Shifting gears away from the Space Shuttle, let's work through a practical example of deep learning, using the `h2o` package. We will do this on data I've modified from the UCI Machine Learning Repository. The original data and its description is available at <https://archive.ics.uci.edu/ml/datasets/Bank+Marketing/>. What I've done is, take the smaller dataset `bank.csv`, scale the numeric variables to mean 0 and variance of 1, create dummies for the character variables/sparse numerics, and eliminate near zero variance variables. The data is available on github <https://github.com/datameister66/data/> named also `bank_DL.csv`. In this section, we will focus on how to load the data in the H2O platform and run the deep learning code to build a classifier to predict whether a customer will respond to a marketing campaign.

H2O background

H2O is an open source predictive analytics platform with prebuilt algorithms, such as k-nearest neighbor, gradient boosted machines, and deep learning. You can upload data to the platform via Hadoop, AWS, Spark, SQL, noSQL, or your hard drive. The great thing about it is that you can utilize the machine learning algorithms in R and, at a much greater scale, on your local machine. If you are interested in learning more, you can visit the site: <http://h2o.ai/product/>.

The process of installing H2O on R is a little different. I put the code here that gave me the latest update (as of February 25, 2017). You can use it to reinstall the latest version or pull it off of the website: <http://h2o-release.s3.amazonaws.com/h2o/rel-lambert/5/docs-website/Ruser/Rinstall.html>. The following is the code to install the latest version:

```
# The following two commands remove any previously installed H2O
# packages for
R.
if ("package:h2o" %in% search()) { detach("package:h2o",
  unload=TRUE) }
if ("h2o" %in% rownames(installed.packages())) {
  remove.packages("h2o") }

# Next, we download packages that H2O depends on.
if (! ("methods" %in% rownames(installed.packages()))) {
  install.packages("methods") }
if (! ("statmod" %in% rownames(installed.packages()))) {
  install.packages("statmod") }
if (! ("stats" %in% rownames(installed.packages()))) {
  install.packages("stats") }
if (! ("graphics" %in% rownames(installed.packages()))) {
  install.packages("graphics") }
if (! ("Rcurl" %in% rownames(installed.packages()))) {
  install.packages("Rcurl") }
if (! ("jsonlite" %in% rownames(installed.packages()))) {
  install.packages("jsonlite") }
if (! ("tools" %in% rownames(installed.packages()))) {
  install.packages("tools") }
if (! ("utils" %in% rownames(installed.packages()))) {
  install.packages("utils") }

# Now we download, install and initialize the H2O package for R.
install.packages("h2o", type="source", repos=c("http://h2o-
release.s3.amazonaws.com/h2o/rel-tverberg/5/R"))
```

Data upload to H2O

Let's assume you have the `bank_DL.csv` file saved in your working directory. Remember, `getwd()` will provide you with the path to it. So, let's load the library and create an object with the file path to the data:

```
> library(h2o)
> path <- "C:/.../bank_DL.csv"
```

We can now connect to H2O and start an instance on the cluster. Specifying `nthreads = -1` requests our instance use all CPUs on the cluster:

```
> localH2O = h2o.init(nthreads = -1)
```

The H2O function, `h2o.uploadFile()`, allows you to upload/import your file to the H2O cloud. The following functions are also available for uploads:

- `h2o.importFolder`
- `h2o.importURL`
- `h2o.importHDFS`

It is quite simple to upload the file and a per cent indicator tracks the status:

```
> bank <- h2o.uploadFile(path = path)
|=====| 100%
```

The data is now in `H2OFrame`, which you can verify with `class()`, as follows:

```
> class(bank)
[1] "H2OFrame"
```

Many of the R commands in H2O may produce a different output than what you are used to seeing. For instance, look at the structure of our data (abbreviated output):

```
> str(bank)
Class 'H2OFrame' <environment: 0x0000000032d02e80>
- attr(*, "op")= chr "Parse"
- attr(*, "id")= chr "bank_DL_sid_95ad_2"
- attr(*, "eval")= logi FALSE
- attr(*, "nrow")= int 4521
- attr(*, "ncol")= int 64
- attr(*, "types")=List of 64
```


We see that it consists of 4,521 observations (nrow) and 64 columns (ncol). By the way, the `head()` and `summary()` functions work exactly the same as in regular R. Before splitting the datasets, let's have a look at our response distribution. It is the column named `y`:

```
> h2o.table(bank$y)
  y Count
1 no  4000
2 yes   521
[2 rows x 2 columns]
```

We see that 521 of the bank's customers responded yes to the offer and 4,000 did not. This response is a bit unbalanced. Techniques that can be used to handle unbalanced response labels are discussed in the chapter on multi-class learning. In this exercise, let's see how deep learning will perform with this lack of label balance.

Create train and test datasets

You can use H2O's functionality to partition the data into train and test sets. The first thing to do is create a vector of random and uniform numbers for the full data:

```
> rand <- h2o.runif(bank, seed = 123)
```

You can then build your partitioned data and assign it with a desired key name, as follows:

```
> train <- bank[rand <= 0.7, ]
> train <- h2o.assign(train, key = "train")
> test  <- bank[rand > 0.7, ]
> test  <- h2o.assign(test, key = "test")
```

With these created, it is probably a good idea that we have a balanced response variable between the `train` and `test` sets. To do this, you can use the `h2o.table()` function and, in our case, it would be column 64:

```
> h2o.table(train[, 64])
  y Count
1 no  2783
2 yes   396
[2 rows x 2 columns]

> h2o.table(test[, 64])
  y Count
1 no  1217
2 yes   125
[2 rows x 2 columns]
```

This appears all well and good, so let's begin the modeling process:

Modeling

As we will see, the deep learning function has quite a few arguments and parameters that you can tune. The thing that I like about the package is the ability to keep it as simple as possible and let the defaults do their thing. If you want to see all the possibilities along with the defaults, see help or run the following command:

```
> args(h2o.deeplearning)
```

Documentation on all the arguments and tuning parameters is available online at <http://h2o.ai/docs/master/model/deep-learning/>.

As on a side note, you can run a demo for the various machine learning methods by just running `demo("method")`. For instance, you can go through the deep learning demo with `demo(h2o.deeplearning)`.

Our next goal is to tune the hyper-parameters using a random search. It takes less time than a full grid search. We will look at `tanh`, with and without dropout, three different hidden layer/neuron combinations, two different dropout ratios, and two different learning rates:

```
> hyper_params <- list(  
  activation = c("Tanh", "TanhWithDropout"),  
  hidden = list(c(20,20),c(30, 30),c(30, 30, 30)),  
  input_dropout_ratio = c(0, 0.05),  
  rate = c(0.01, 0.25)  
)
```

You now help specify the random search criteria in a list. Since we want a random search we will specify `RandomDiscrete`. A full grid search would require `Cartesian`. It is recommended to specify one or more early stopping criterion for a random search such as `max_runtime_secs`, `max_models`. We also specify here that it will stop when the top five models are within 1% error of each other:

```
> search_criteria = list(  
  strategy = "RandomDiscrete", max_runtime_secs = 420,  
  max_models = 100, seed = 123, stopping_rounds = 5,  
  stopping_tolerance = 0.01  
)
```

Now, this is where the magic should happen using the `h2o.grid()` function. We tell it, we want to use the deep learning algorithm, our test data, any validation data (we will use the test set), our input features, and response variable:

```
> randomSearch <- h2o.grid(
  algorithm = "deeplearning",
  grid_id = "randomSearch",
  training_frame = train,
  validation_frame = test,
  x = 1:63,
  y = 64,
  epochs = 1,
  stopping_metric = "misclassification",
  hyper_params = hyper_params,
  search_criteria = search_criteria
)
|=====| 100%
```

An indicator bar tracks the progress, and with this dataset, it should take less than a few seconds.

We now examine the results of the top five models:

```
> grid <- h2o.getGrid("randomSearch", sort_by = "auc", decreasing =
  FALSE)

> grid
H2O Grid Details
=====

Grid ID: randomSearch
Used hyper parameters:
- activation
- hidden
- input_dropout_ratio
- rate
Number of models: 71
Number of failed models: 0

Hyper-Parameter Search Summary: ordered by decreasing auc
      activation      hidden input_dropout_ratio rate
1 TanhWithDropout [30, 30, 30]          0.05 0.25
2 TanhWithDropout [20, 20]              0.05 0.01
3 TanhWithDropout [30, 30, 30]          0.05 0.25
4 TanhWithDropout [40, 40]              0.05 0.01
5 TanhWithDropout [30, 30, 30]          0.0 0.25

      model_ids      auc
1 randomSearch_model_57 0.8636778964667214
```

```
2 randomSearch_model_8    0.8623894823336072
3 randomSearch_model_10   0.856568611339359
4 randomSearch_model_39   0.8565258833196385
5 randomSearch_model_3    0.8544026294165982
```

So the winning model is #57 with activation of `TanhWithDropout`, three hidden layers with 30 neurons each, dropout ratio of 0.05, and learning rate of 0.25, which had an AUC of almost 0.864.

We now have a look at our error rates in the validation/test data with a confusion matrix:

```
> best_model <- h2o.getModel(grid@model_ids[[1]])
> h2o.confusionMatrix(best_model, valid = T)
Confusion Matrix (vertical: actual; across: predicted) for max f1 @
threshold = 0.0953170555399435:
      no yes  Error  Rate
no    1128  89 0.073131 = 89/1217
yes     60  65 0.480000 = 60/125
Totals 1188 154 0.111028 = 149/1342
```

Even though we only have 11% error, we had high errors for the `yes` label with high rates of false positives and false negatives. It possibly indicates that class imbalance may be an issue. We also have just started the hyper-parameter tuning process, so much work could be done to improve the outcome. I'll leave that task to you!

Now let's examine how to build a model using cross-validation. Notice how the hyper-parameters are included in the function `h2o.deeplearning()` with the exception of learning rate, which is specified as `adaptive`. I also included the functionality to up-sample the minority class to achieve balanced labels during training. On another note, the folds are a stratified sample based on the response variable:

```
> dlmodel <- h2o.deeplearning(
  x = 1:63,
  y = 64,
  training_frame = train,
  hidden = c(30, 30, 30),
  epochs = 3,
  nfolds = 5,
  fold_assignment = "Stratified",
  balance_classes = T,
  activation = "TanhWithDropout",
  seed = 123,
  adaptive_rate = F,
  input_dropout_ratio = 0.05,
  stopping_metric = "misclassification",
  variable_importances = T
)
```

If you call the object `dlmodel`, you will receive rather lengthy output. In this instance, let's examine the performance on the holdout folds:

```
> dlmodel
Model Details:
=====
AUC:  0.8571054599
Gini: 0.7142109198

Confusion Matrix (vertical: actual; across: predicted) for F1-optimal
threshold:
      no yes  Error      Rate
no    2492 291 0.104563 = 291/2783
yes    160 236 0.404040 = 160/396
Totals 2652 527 0.141869 = 451/3179
```

Given these results, I think more tuning is in order for the hyper-parameters, particularly with the hidden layers/neurons. Examining out of sample performance is a little different, but is quite comprehensive, utilizing the `h2o.performance()` function:

```
> perf <- h2o.performance(dlmodel, test)
> perf
H2OBinomialMetrics: deeplearning
MSE:                0.07237450145
RMSE:               0.2690250945
LogLoss:            0.2399027004
Mean Per-Class Error: 0.2326113394
AUC:                0.8319605588
Gini:               0.6639211175

Confusion Matrix (vertical: actual; across: predicted) for F1-
optimal
threshold:
      no yes  Error      Rate
no 1050 167 0.137223 = 167/1217
yes  41  84 0.328000 = 41/125
Totals 1091 251 0.154993 = 208/1342

Maximum Metrics: Maximum metrics at their respective thresholds
metric                                threshold  value idx
1 max f1                               0.323529 0.446809 62
2 max f2                               0.297121 0.612245 166
3 max f0point5                         0.323529 0.372011 62
4 max accuracy                         0.342544 0.906110 0
5 max precision                        0.323529 0.334661 62
6 max recall                           0.013764 1.000000 355
7 max specificity                       0.342544 0.999178 0
8 max absolute_mcc                     0.297121 0.411468 166
```

```
9 max min_per_class_accuracy 0.313356 0.799507 131
10 max mean_per_class_accuracy 0.285007 0.819730 176
```

The overall error increased, but we have lower false positive and false negative rates. As before, additional tuning is required.

Finally, the variable importance can be produced. This is calculated based on the so-called Gedeon Method. Keep in mind that these results can be misleading. In the table, we can see the order of the variable importance, but this importance is subject to the sampling variation, and if you change the seed value, the order of the variable importance could change quite a bit. These are the top five variables by importance:

```
> dlmodel$model$variable_importances
Variable Importances:
      variable relative_importance scaled_importance percentage
1 duration                1.000000          1.000000    0.147006
2 poutcome_success        0.806309          0.806309    0.118532
3 month_oct                0.329299          0.329299    0.048409
4 month_mar               0.223847          0.223847    0.032907
5 poutcome_failure        0.199272          0.199272    0.029294
```

With this, we have completed the introduction to deep learning in R using the capabilities of the `H2O` package. It is simple to use while offering plenty of flexibility to tune the hyperparameters and create deep neural networks. Enjoy!

Summary

In this chapter, the goal was to get you up and running in the exciting world of neural networks and deep learning. We examined how the methods work, their benefits, and their inherent drawbacks with applications to two different datasets. These techniques work well where complex, nonlinear relationships exist in the data. However, they are highly complex, potentially require a ton of hyper-parameter tuning, are the quintessential black boxes, and are difficult to interpret. We don't know why the self-driving car made a right on red, we just know that it did so properly. I hope you will apply these methods by themselves or supplement other methods in an ensemble modeling fashion. Good luck and good hunting! We will now shift gears to unsupervised learning, starting with clustering.

8

Cluster Analysis

"Quickly bring me a beaker of wine, so that I may wet my mind and say something clever."

- Aristophanes, Athenian Playwright

In the earlier chapters, we focused on trying to learn the best algorithm in order to solve an outcome or response, for example, a breast cancer diagnosis or level of Prostate Specific Antigen. In all these cases, we had y , and that y is a function of x , or $y = f(x)$. In our data, we had the actual y values and we could train the x accordingly. This is referred to as **supervised learning**. However, there are many situations where we try to learn something from our data and either we do not have the y or we actually choose to ignore it. If so, we enter the world of **unsupervised learning**. In this world, we build and select our algorithm based on how well it addresses our business needs versus how accurate it is.

Why would we try and learn without supervision? First of all, unsupervised learning can help you understand and identify patterns in your data, which may be valuable. Second, you can use it to transform your data in order to improve your supervised learning techniques.

This chapter will focus on the former and the next chapter on the latter.

So, let's begin by tackling a popular and powerful technique known as **cluster analysis**. With cluster analysis, the goal is to group the observations into a number of groups (k -groups), where the members in a group are as similar as possible while the members between groups are as different as possible. There are many examples of how this can help an organization; here are just a few:

- The creation of customer types or segments
- The detection of high-crime areas in a geography
- Image and facial recognition

- Genetic sequencing and transcription
- Petroleum and geological exploration

There are many uses of cluster analysis but there are also many techniques. We will focus on the two most common: **hierarchical** and **k-means**. They are both effective clustering methods, but may not always be appropriate for the large and varied datasets that you may be called upon to analyze. Therefore, we will also examine **Partitioning Around Medoids (PAM)** using a **Gower-based** metric dissimilarity matrix as the input. Finally, we will examine a new methodology I recently learned and applied using **Random Forest** to transform your data. The transformed data can then be used as an input to unsupervised learning.

A final comment before moving on. You may be asked if these techniques are more art than science as the learning is unsupervised. I think the clear answer is, *it depends*. In early 2016, I presented the methods here at a meeting of the Indianapolis, Indiana R-User Group. To a person, we all agreed that it is the judgment of the analysts and the business users that makes unsupervised learning meaningful and determines whether you have, say, three versus four clusters in your final algorithm. This quote sums it up nicely:

"The major obstacle is the difficulty in evaluating a clustering algorithm without taking into account the context: why does the user cluster his data in the first place, and what does he want to do with the clustering afterwards? We argue that clustering should not be treated as an application-independent mathematical problem, but should always be studied in the context of its end-use."

- Luxburg et al. (2012)

Hierarchical clustering

The hierarchical clustering algorithm is based on a dissimilarity measure between observations. A common measure, and what we will use, is **Euclidean distance**. Other distance measures are also available.



Hierarchical clustering is an agglomerative or bottom-up technique. By this, we mean that all observations are their own cluster. From there, the algorithm proceeds iteratively by searching all the pairwise points and finding the two clusters that are the most similar. So, after the first iteration, there are $n-1$ clusters, and after the second iteration, there are $n-2$ clusters, and so forth.

As the iterations continue, it is important to understand that in addition to the distance measure, we need to specify the linkage between the groups of observations. Different types of data will demand that you use different cluster linkages. As you experiment with the linkages, you may find that some may create highly unbalanced numbers of observations in one or more clusters. For example, if you have 30 observations, one technique may create a cluster of just one observation, regardless of how many total clusters that you specify. In this situation, your judgment will likely be needed to select the most appropriate linkage as it relates to the data and business case.

The following table lists the types of common linkages, but note that there are others:

Linkage	Description
Ward	This minimizes the total within-cluster variance as measured by the sum of squared errors from the cluster points to its centroid
Complete	The distance between two clusters is the maximum distance between an observation in one cluster and an observation in the other cluster
Single	The distance between two clusters is the minimum distance between an observation in one cluster and an observation in the other cluster
Average	The distance between two clusters is the mean distance between an observation in one cluster and an observation in the other cluster
Centroid	The distance between two clusters is the distance between the cluster centroids

The output of hierarchical clustering will be a **dendrogram**, which is a tree-like diagram that shows the arrangement of the various clusters.

As we will see, it can often be difficult to identify a clear-cut breakpoint in the selection of the number of clusters. Once again, your decision should be iterative in nature and focused on the context of the business decision.

Distance calculations

As mentioned previously, Euclidean distance is commonly used to build the input for hierarchical clustering. Let's look at a simple example of how to calculate it with two observations and two variables/features.

Let's say that observation *A* costs \$5.00 and weighs 3 pounds. Further, observation *B* costs \$3.00 and weighs 5 pounds. We can place these values in the distance formula: *distance between A and B is equal to the square root of the sum of the squared differences*, which in our example would be as follows:

$d(A, B) = \text{square root}((5 - 3)^2 + (3 - 5)^2)$, which is equal to 2.83

The value of 2.83 is not a meaningful value in and of itself, but is important in the context of the other pairwise distances. This calculation is the default in R for the `dist()` function. You can specify other distance calculations (maximum, manhattan, canberra, binary, and minkowski) in the function. We will avoid going in to detail on why or where you would choose these over Euclidean distance. This can get rather domain-specific, for example, a situation where Euclidean distance may be inadequate is where your data suffers from high-dimensionality, such as in a genomic study. It will take domain knowledge and/or trial and error on your part to determine the proper distance measure.



One final note is to scale your data with a mean of zero and standard deviation of one so that the distance calculations are comparable. If not, any variable with a larger scale will have a larger effect on distances.

K-means clustering

With k-means, we will need to specify the exact number of clusters that we want. The algorithm will then iterate until each observation belongs to just one of the k-clusters. The algorithm's goal is to minimize the within-cluster variation as defined by the squared Euclidean distances. So, the kth-cluster variation is the sum of the squared Euclidean distances for all the pairwise observations divided by the number of observations in the cluster.

Due to the iteration process that is involved, one k-means result can differ greatly from another result even if you specify the same number of clusters. Let's see how this algorithm plays out:

1. **Specify** the exact number of clusters you desire (*k*).
2. **Initialize** *K* observations are randomly selected as the initial *means*.

3. **Iterate:**

- K clusters are created by assigning each observation to its closest cluster center (minimizing within-cluster sum of squares)
- The centroid of each cluster becomes the new *mean*
- This is repeated until convergence, that is the cluster centroids do not change

As you can see, the final result will vary because of the initial assignment in step 1. Therefore, it is important to run multiple initial starts and let the software identify the best solution. In R, this can be a simple process as we will see.

Gower and partitioning around medoids

As you conduct clustering analysis in real life, one of the things that can quickly become apparent is the fact that neither hierarchical nor k-means is specifically designed to handle mixed datasets. By mixed data, I mean both quantitative and qualitative or, more specifically, nominal, ordinal, and interval/ratio data.

The reality of most datasets that you will use is that they will probably contain mixed data. There are a number of ways to handle this, such as doing **Principal Components Analysis (PCA)** first in order to create latent variables, then using them as input in clustering or using different dissimilarity calculations. We will discuss PCA in the next chapter.

With the power and simplicity of R, you can use the **Gower dissimilarity coefficient** to turn mixed data to the proper feature space. In this method, you can even include factors as input variables. Additionally, instead of k-means, I recommend using the **PAM clustering algorithm**.

PAM is very similar to k-means but offers a couple of advantages. They are listed as follows:

- First, PAM accepts a dissimilarity matrix, which allows the inclusion of mixed data
- Second, it is more robust to outliers and skewed data because it minimizes a sum of dissimilarities instead of a sum of squared Euclidean distances (Reynolds, 1992)

This is not to say that you must use Gower and PAM together. If you choose, you can use the Gower coefficients with hierarchical and I've seen arguments for and against using it in the context of k-means. Additionally, PAM can accept other linkages. However, when paired they make an effective method to handle mixed data. Let's take a quick look at both of these concepts before moving on.

Gower

The Gower coefficient compares cases pairwise and calculates a dissimilarity between them, which is essentially the weighted mean of the contributions of each variable. It is defined for two cases called i and j as follows:

$$S_{ij} = \text{sum}(W_{ijk} * S_{ijk}) / \text{sum}(W_{ijk})$$

Here, S_{ijk} is the contribution provided by the k_{th} variable, and W_{ijk} is 1 if the k_{th} variable is valid, or else 0.

For ordinal and continuous variables, $S_{ijk} = 1 - (\text{absolute value of } x_{ij} - x_{ik}) / r_k$, where r_k is the range of values for the k_{th} variable.

For nominal variables, $S_{ijk} = 1$ if $x_{ij} = x_{jk}$, or else 0.

For binary variables, S_{ijk} is calculated based on whether an attribute is present (+) or not present (-), as shown in the following table:

Variables	Value of attribute k			
Case i	+	+	-	-
Case j	+	-	+	-
S_{ijk}	1	0	0	0
W_{ijk}	1	1	1	0

PAM

For **Partitioning Around Medoids**, let's first define a **medoid**.



A medoid is an observation of a cluster that minimizes the dissimilarity (in our case, calculated using the Gower metric) between the other observations in that cluster. So, similar to k-means, if you specify five clusters, you will have five partitions of the data.

With the objective of minimizing the dissimilarity of all the observations to the nearest medoid, the PAM algorithm iterates over the following steps:

1. Randomly select k observations as the initial medoid.
2. Assign each observation to the closest medoid.
3. Swap each medoid and non-medoid observation, computing the dissimilarity cost.
4. Select the configuration that minimizes the total dissimilarity.
5. Repeat steps 2 through 4 until there is no change in the medoids.

Both Gower and PAM can be called using the `cluster` package in R. For Gower, we will use the `daisy()` function in order to calculate the dissimilarity matrix and the `pam()` function for the actual partitioning. With this, let's get started with putting these methods to the test.

Random forest

Like our motivation with the use of the Gower metric in handling mixed, in fact, *messy* data, we can apply random forest in an unsupervised fashion. Selection of this method has some advantages:

- Robust against outliers and highly skewed variables
- No need to transform or scale the data
- Handles mixed data (numeric and factors)
- Can accommodate missing data
- Can be used on data with a large number of variables, in fact, it can be used to eliminate useless features by examining variable importance
- The dissimilarity matrix produced serves as an input to the other techniques discussed earlier (hierarchical, k-means, and PAM)

A couple words of caution. It may take some trial and error to properly tune the Random Forest with respect to the number of variables sampled at each tree split (`mtry = ?` in the function) and the number of trees grown. Studies done show that the more trees grown, up to a point, provide better results, and a good starting point is to grow 2,000 trees (Shi, T. & Horvath, S., 2006).

This is how the algorithm works, given a data set with no labels:

- The current observed data is labeled as class 1
- A second (synthetic) set of observations are created of the same size as the observed data; this is created by randomly sampling from each of the features from the observed data, so if you have 20 observed features, you will have 20 synthetic features
- The synthetic portion of the data is labeled as class 2, which facilitates using Random Forest as an artificial classification problem
- Create a Random Forest model to distinguish between the two classes
- Turn the model's proximity measures of just the observed data (the synthetic data is now discarded) into a dissimilarity matrix
- Utilize the dissimilarity matrix as the clustering input features

So what exactly are these proximity measures?



Proximity measure is a pairwise measure between all the observations. If two observations end up in the same terminal node of a tree, their proximity score is equal to one, otherwise zero.

At the termination of the Random Forest run, the proximity scores for the observed data are normalized by dividing by the total number of trees. The resulting $N \times N$ matrix contains scores between zero and one, naturally with the diagonal values all being one. That's all there is to it. An effective technique that I believe is underutilized and one that I wish I had learned years ago.

Business understanding

Until a couple of weeks ago, I was unaware that there were less than 300 certified Master Sommeliers in the entire world. The exam, administered by the Court of Master Sommeliers, is notorious for its demands and high failure rate.

The trials, tribulations, and rewards of several individuals pursuing the certification are detailed in the critically-acclaimed documentary, **Somm**. So, for this exercise, we will try and help a hypothetical individual struggling to become a Master Sommelier find a latent structure in Italian wines.

Data understanding and preparation

Let's start with loading the R packages that we will need for this chapter. As always, make sure that you have installed them first:

```
> library(cluster) #conduct cluster analysis
> library(compareGroups) #build descriptive statistic tables
> library(HDclassif) #contains the dataset
> library(NbClust) #cluster validity measures
> library(sparcl) #colored dendrogram
```

The dataset is in the `HDclassif` package, which we installed. So, we can load the data and examine the structure with the `str()` function:

```
> data(wine)

> str(wine)
'data.frame':178 obs. of 14 variables:
 $ class: int 1 1 1 1 1 1 1 1 1 1 ...
 $ V1 : num 14.2 13.2 13.2 14.4 13.2 ...
 $ V2 : num 1.71 1.78 2.36 1.95 2.59 1.76 1.87 2.15 1.64 1.35
 ...
 $ V3 : num 2.43 2.14 2.67 2.5 2.87 2.45 2.45 2.61 2.17 2.27 ...
 $ V4 : num 15.6 11.2 18.6 16.8 21 15.2 14.6 17.6 14 16 ...
 $ V5 : int 127 100 101 113 118 112 96 121 97 98 ...
 $ V6 : num 2.8 2.65 2.8 3.85 2.8 3.27 2.5 2.6 2.8 2.98 ...
 $ V7 : num 3.06 2.76 3.24 3.49 2.69 3.39 2.52 2.51 2.98 3.15
 ...
 $ V8 : num 0.28 0.26 0.3 0.24 0.39 0.34 0.3 0.31 0.29 0.22 ...
 $ V9 : num 2.29 1.28 2.81 2.18 1.82 1.97 1.98 1.25 1.98 1.85
 ...
 $ V10 : num 5.64 4.38 5.68 7.8 4.32 6.75 5.25 5.05 5.2 7.22 ...
 $ V11 : num 1.04 1.05 1.03 0.86 1.04 1.05 1.02 1.06 1.08 1.01
 ...
 $ V12 : num 3.92 3.4 3.17 3.45 2.93 2.85 3.58 3.58 2.85 3.55 ...
 $ V13 : int 1065 1050 1185 1480 735 1450 1290 1295 1045 1045 ...
```

The data consists of 178 wines with 13 variables of the chemical composition and one variable `Class`, the label, for the cultivar or plant variety. We won't use this in the clustering but as a test of model performance. The variables, `v1` through `v13`, are the measures of the chemical composition as follows:

- `v1`: alcohol
- `v2`: malic acid
- `v3`: ash
- `v4`: alkalinity of ash
- `v5`: magnesium
- `v6`: total phenols
- `v7`: flavonoids
- `v8`: non-flavonoid phenols
- `v9`: proanthocyanins
- `v10`: color intensity
- `v11`: hue
- `v12`: OD280/OD315
- `v13`: proline

The variables are all quantitative. We should rename them to something meaningful for our analysis. This is easily done with the `names()` function:

```
> names(wine) <- c("Class", "Alcohol", "MalicAcid", "Ash",
  "Alk_ash", "magnesium", "T_phenols", "Flavanoids", "Non_flav",
  "Proantho", "C_Intensity", "Hue", "OD280_315", "Proline")

> names(wine)
[1] "Class"      "Alcohol"    "MalicAcid"  "Ash"
[5] "Alk_ash"    "magnesium"  "T_phenols"  "Flavanoids"
[9] "Non_flav"   "Proantho"   "C_Intensity" "Hue"
[13] "OD280_315"  "Proline"
```

As the variables are not scaled, we will need to do this using the `scale()` function. This will first center the data where the column mean is subtracted from each individual in the column. Then the centered values will be divided by the corresponding column's standard deviation. We can also use this transformation to make sure that we only include columns 2 through 14, dropping class and putting it in a data frame. This can all be done with one line of code:

```
> df <- as.data.frame(scale(wine[, -1]))
```


Now, check the structure to make sure that it all worked according to plan:

```
> str(df)
'data.frame': 178 obs. of  13 variables:
 $ Alcohol      : num  1.514 0.246 0.196 1.687 0.295 ...
 $ MalicAcid    : num  -0.5607 -0.498 0.0212 -0.3458 0.2271 ...
 $ Ash          : num   0.231 -0.826 1.106 0.487 1.835 ...
 $ Alk_ash      : num  -1.166 -2.484 -0.268 -0.807 0.451 ...
 $ magnesium    : num   1.9085 0.0181 0.0881 0.9283 1.2784 ...
 $ T_phenols    : num   0.807 0.567 0.807 2.484 0.807 ...
 $ Flavanoids   : num   1.032 0.732 1.212 1.462 0.661 ...
 $ Non_flav     : num  -0.658 -0.818 -0.497 -0.979 0.226 ...
 $ Proantho     : num   1.221 -0.543 2.13 1.029 0.4 ...
 $ C_Intensity:  : num   0.251 -0.292 0.268 1.183 -0.318 ...
 $ Hue          : num   0.361 0.405 0.317 -0.426 0.361 ...
 $ OD280_315    : num   1.843 1.11 0.786 1.181 0.448 ...
 $ Proline      : num   1.0102 0.9625 1.3912 2.328 -0.0378 ...
```

Before moving on, let's do a quick table to see the distribution of the cultivars or `Class`:

```
> table(wine$Class)

 1  2  3 
59 71 48
```

We can now move on to the modeling step of the process.

Modeling and evaluation

Having created our data frame, `df`, we can begin to develop the clustering algorithms. We will start with hierarchical and then try our hand at k-means. After this, we will need to manipulate our data a little bit to demonstrate how to incorporate mixed data with Gower and Random Forest.

Hierarchical clustering

To build a hierarchical cluster model in R, you can utilize the `hclust()` function in the base `stats` package. The two primary inputs needed for the function are a distance matrix and the clustering method. The distance matrix is easily done with the `dist()` function. For the distance, we will use Euclidean distance. A number of clustering methods are available, and the default for `hclust()` is the complete linkage.

We will try this, but I also recommend Ward's linkage method. Ward's method tends to produce clusters with a similar number of observations.

The complete linkage method results in the distance between any two clusters that is the maximum distance between any one observation in a cluster and any one observation in the other cluster. Ward's linkage method seeks to cluster the observations in order to minimize the within-cluster sum of squares.

It is noteworthy that the R method `ward.D2` uses the squared Euclidean distance, which is indeed Ward's linkage method. In R, `ward.D` is available but requires your distance matrix to be squared values. As we will be building a distance matrix of non-squared values, we will require `ward.D2`.

Now, the big question is how many clusters should we create? As stated in the introduction, the short, and probably not very satisfying answer is that it depends. Even though there are cluster validity measures to help with this dilemma--which we will look at--it really requires an intimate knowledge of the business context, underlying data, and, quite frankly, trial and error. As our sommelier partner is fictional, we will have to rely on the validity measures. However, that is no panacea to selecting the numbers of clusters as there are several dozen validity measures.

As exploring the positives and negatives of the vast array of cluster validity measures is way outside the scope of this chapter, we can turn to a couple of papers and even R itself to simplify this problem for us. A paper by Miligan and Cooper, 1985, explored the performance of 30 different measures/indices on simulated data. The top five performers were CH index, Duda Index, Cindex, Gamma, and Beale Index. Another well-known method to determine the number of clusters is the **gap statistic** (Tibshirani, Walther, and Hastie, 2001). These are two good papers for you to explore if your cluster validity curiosity gets the better of you.

With R, one can use the `NbClust()` function in the `NbClust` package to pull results on 23 indices, including the top five from Miligan and Cooper and the gap statistic. You can see a list of all the available indices in the help file for the package. There are two ways to approach this process: one is to pick your favorite index or indices and call them with R, the other way is to include all of them in the analysis and go with the majority rules method, which the function summarizes for you nicely. The function will also produce a couple of plots as well.

With the stage set, let's walk through the example of using the complete linkage method. When using the function, you will need to specify the minimum and maximum number of clusters, distance measures, and indices in addition to the linkage. As you can see in the following code, we will create an object called `numComplete`. The function specifications are for Euclidean distance, minimum number of clusters two, maximum number of clusters six, complete linkage, and all indices. When you run the command, the function will automatically produce an output similar to what you can see here--a discussion on both the graphical methods and majority rules conclusion:

```
> numComplete <- NbClust(df, distance = "euclidean", min.nc = 2,
  max.nc=6, method = "complete", index = "all")
*** : The Hubert index is a graphical method of determining the
      number of clusters.
In the plot of Hubert index, we seek a significant knee that
      corresponds to a significant increase of the value of the
      measure that is the significant peak in Hubert index second
      differences plot.

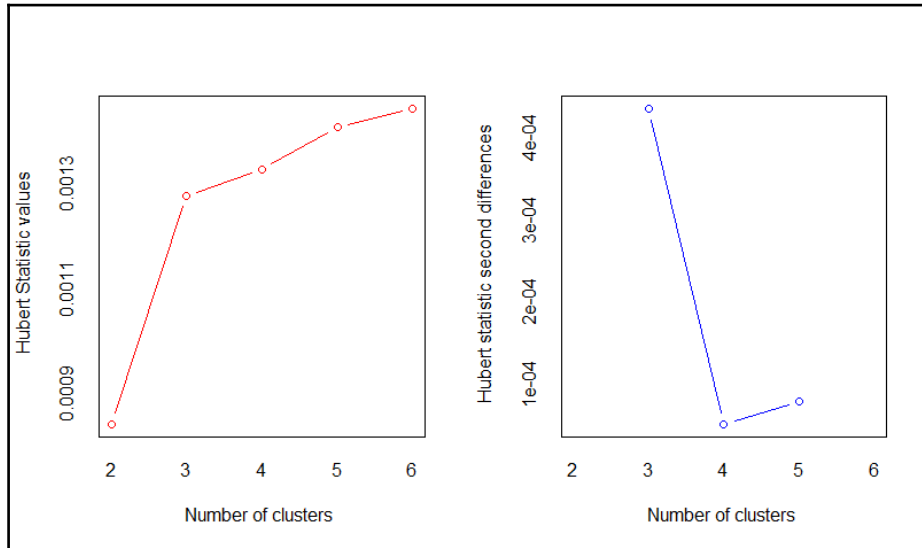
*** : The D index is a graphical method of determining the number
      of clusters.
In the plot of D index, we seek a significant knee (the significant peak in
Dindex second differences plot) that corresponds to a significant increase
of the value of the measure.

*****
* Among all indices:
* 1 proposed 2 as the best number of clusters
* 11 proposed 3 as the best number of clusters
* 6 proposed 5 as the best number of clusters
* 5 proposed 6 as the best number of clusters

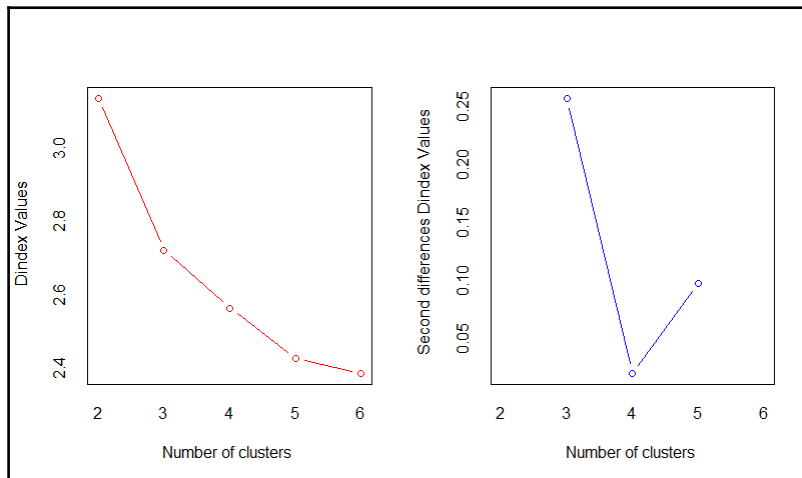
***** Conclusion *****

* According to the majority rule, the best number of clusters is 3
```

Going with the majority rules method, we would select three clusters as the optimal solution, at least for hierarchical clustering. The two plots that are produced contain two graphs each. As the preceding output states, you are looking for a significant knee in the plot (the graph on the left-hand side) and the peak of the graph on the right-hand side. This is the **Hubert Index** plot:



You can see that the bend or knee is at three clusters in the graph on the left-hand side. Additionally, the graph on the right-hand side has its peak at three clusters. The following **Dindex plot** provides the same information:



There are a number of values that you can call with the function and there is one that I would like to show. This output is the best number of clusters for each index and the index value for that corresponding number of clusters. This is done with `$Best.nc`. I've abbreviated the output to the first nine indices:

```
> numComplete$Best.nc
      KL      CH Hartigan    CCC      Scott
Number_clusters 5.0000 3.0000 3.0000 5.000 3.0000
Value_Index    14.2227 48.9898 27.8971 1.148 340.9634
      Marriot  TrCovW  TraceW Friedman
Number_clusters 3.000000e+00 3.00 3.0000 3.0000
Value_Index    6.872632e+25 22389.83 256.4861 10.6941
```

You can see that the first index, (KL), has the optimal number of clusters as five and the next index, (CH), has it as three.

With three clusters as the recommended selection, we will now compute the distance matrix and build our hierarchical cluster object. This code will build the distance matrix:

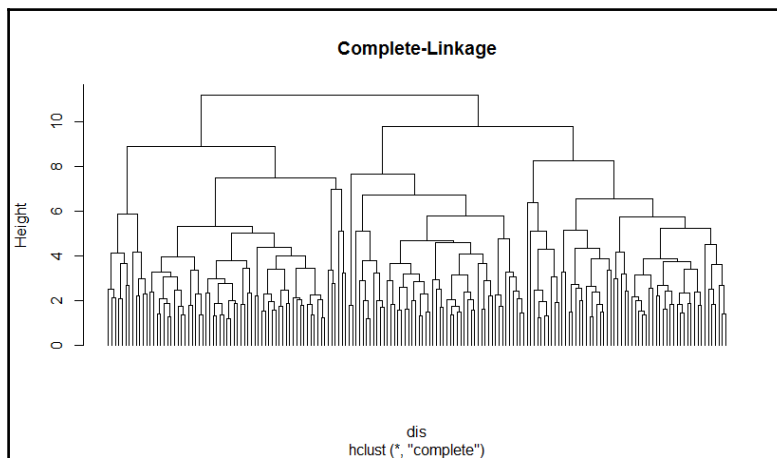
```
> dis <- dist(df, method = "euclidean")
```

Then, we will use this matrix as the input for the actual clustering with `hclust()`:

```
> hc <- hclust(dis, method = "complete")
```

The common way to visualize hierarchical clustering is to plot a **dendrogram**. We will do this with the `plot` function. Note that `hang = -1` puts the observations across the bottom of the diagram:

```
> plot(hc, hang = -1, labels = FALSE, main = "Complete-Linkage")
```



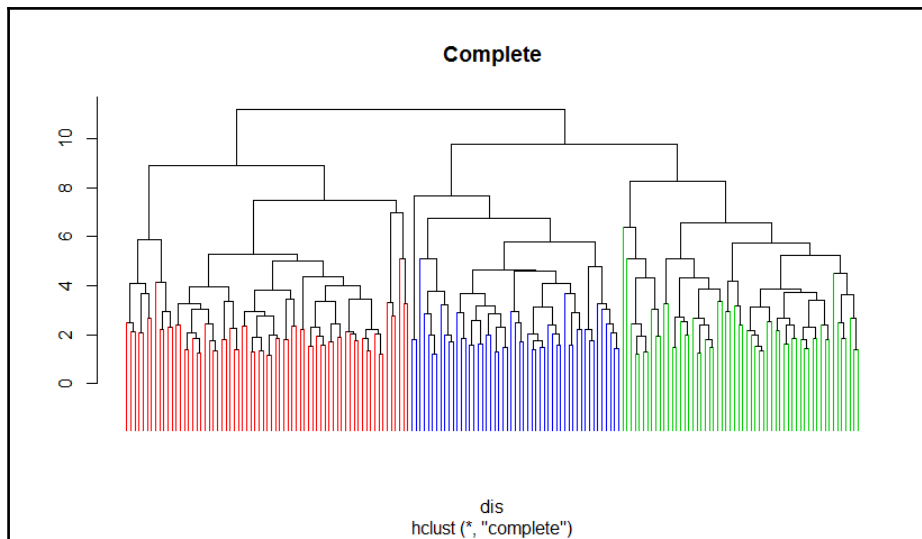
The dendrogram is a tree diagram that shows you how the individual observations are clustered together. The arrangement of the connections (branches, if you will) tells us which observations are similar. The height of the branches indicates how much the observations are similar or dissimilar to each other from the distance matrix. Note that I specified `labels = FALSE`. This was done to aid in the interpretation because of the number of observations. In a smaller dataset of, say, no more than 40 observations, the row names can be displayed.

To aid in visualizing the clusters, you can produce a colored dendrogram using the `sparcl` package. To color the appropriate number of clusters, you need to cut the dendrogram tree to the proper number of clusters using the `cutree()` function. This will also create the cluster label for each of the observations:

```
> comp3 <- cutree(hc, 3)
```

Now, the `comp3` object is used in the function to build the colored dendrogram:

```
> ColorDendrogram(hc, y = comp3, main = "Complete", branchlength = 50)
```



Note that I used `branchlength = 50`. This value will vary based on your own data. As we have the cluster labels, let's build a table that shows the count per cluster:

```
> table(comp3)
comp3
 1  2  3
69 58 51
```

Out of curiosity, let's go ahead and compare how this clustering algorithm compares to the **cultivar** labels:

```
> table(comp3, wine$Class)

comp3  1  2  3
1  51 18  0
2   8 50  0
3   0  3 48
```

In this table, the rows are the clusters and columns are the cultivars. This method matched the cultivar labels at an 84 percent rate. Note that we are not trying to use the clusters to predict a cultivar, and in this example, we have no a priori reason to match clusters to the cultivars.

We will now try Ward's linkage. This is the same code as before; it first starts with trying to identify the number of clusters, which means that we will need to change the method to Ward.D2:

```
> numWard <- NbClust(df, diss = NULL, distance = "euclidean",
  min.nc = 2, max.nc = 6, method = "ward.D2", index = "all")

*** : The Hubert index is a graphical method of determining the number of
clusters.
In the plot of Hubert index, we seek a significant knee that corresponds to
a significant increase of the value of the measure i.e the significant peak
in Hubert index second differences plot.

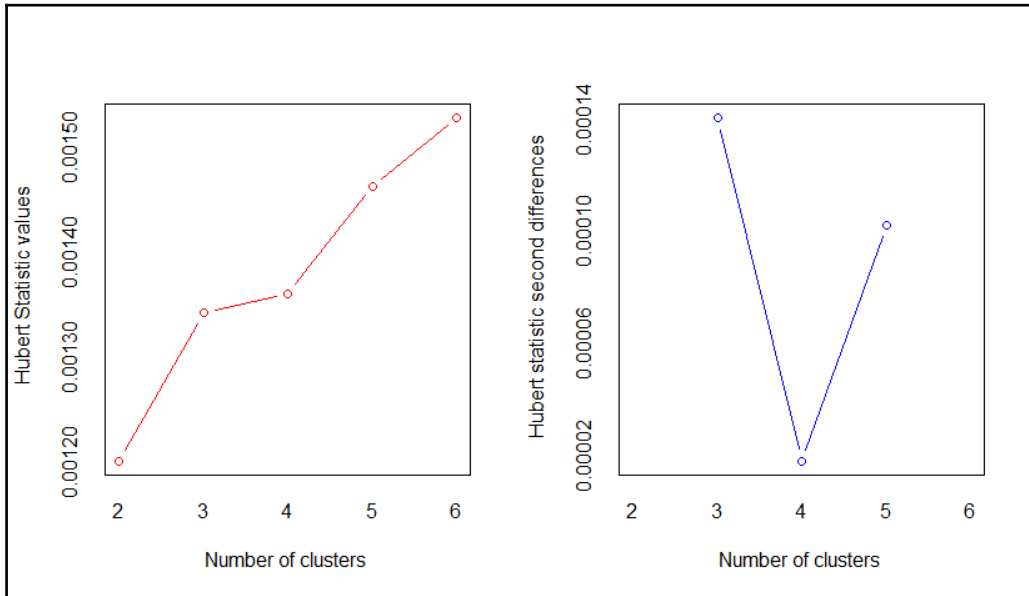
*** : The D index is a graphical method of determining the number of
clusters.
In the plot of D index, we seek a significant knee (the significant peak in
Dindex second differences plot) that corresponds to a significant increase
of the value of the measure.

*****
* Among all indices:
* 2 proposed 2 as the best number of clusters
* 18 proposed 3 as the best number of clusters
* 2 proposed 6 as the best number of clusters

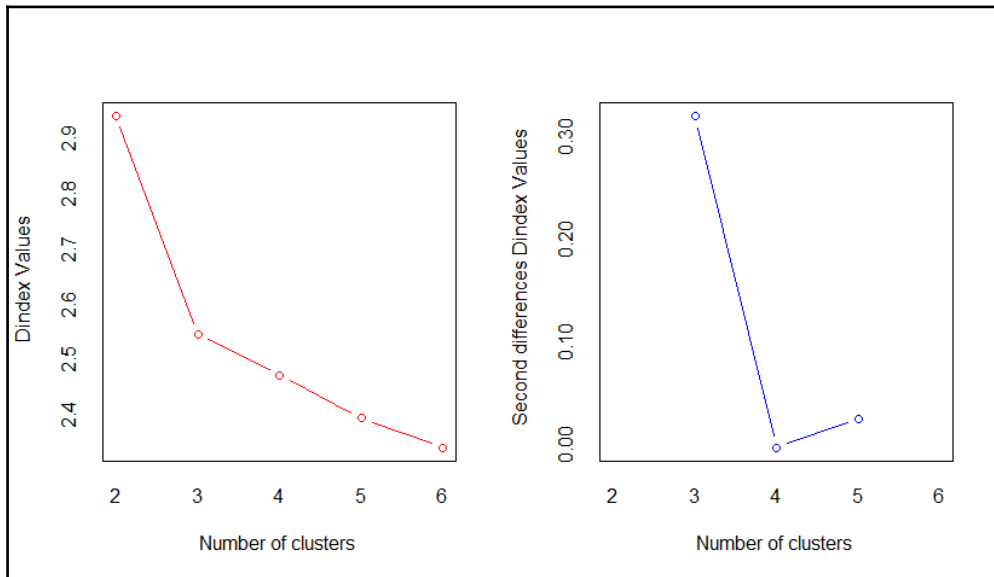
***** Conclusion *****

* According to the majority rule, the best number of clusters is 3
```

This time around also, the majority rules were for a three cluster solution. Looking at the Hubert Index, the best solution is a three cluster as well:

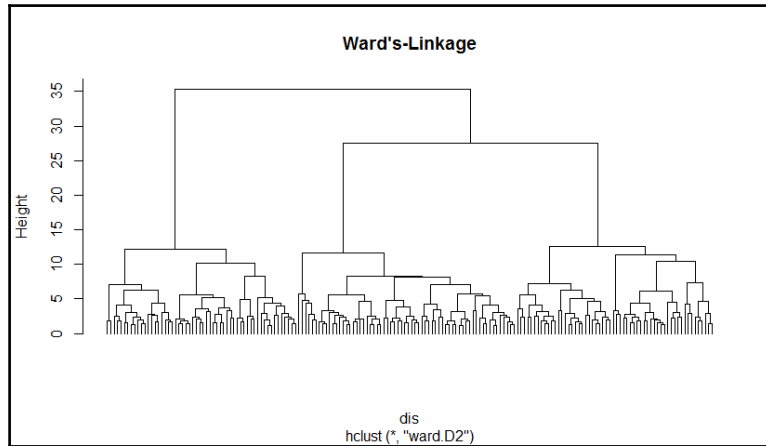


The Dindex adds further support to the three cluster solution:



Let's move on to the actual clustering and production of the dendrogram for Ward's linkage:

```
> hcWard <- hclust(dis, method = "ward.D2")  
  
> plot(hcWard, labels = FALSE, main = "Ward's-Linkage")
```



The plot shows three pretty distinct clusters that are roughly equal in size. Let's get a count of the cluster size and show it in relation to the cultivar labels:

```
> ward3 <- cutree(hcWard, 3)  
> table(ward3, wine$Class)
```

ward3	1	2	3
1	59	5	0
2	0	58	0
3	0	8	48

So, cluster one has 64 observations, cluster two has 58, and cluster three has 56. This method matches the cultivar categories closer than using complete linkage.

With another table, we can compare how the two methods match observations:

```
> table(comp3, ward3)  
ward3  
comp3  1  2  3  
1  53 11  5  
2  11 47  0  
3   0  0 51
```

While cluster three for each method is pretty close, the other two are not. The question now is how do we identify what the differences are for the interpretation? In many examples, the datasets are very small and you can look at the labels for each cluster. In the real world, this is often impossible. A good way to compare is to use the `aggregate()` function, summarizing on a statistic such as the `mean` or `median`. Additionally, instead of doing it on the scaled data, let's try it on the original data. In the function, you will need to specify the dataset, what you are aggregating it by, and the summary statistic:

```
> aggregate(wine[, -1], list(comp3), mean)
  Group.1 Alcohol MalicAcid      Ash Alk_ash magnesium T_phenols
1      1  13.40609  1.898986 2.305797 16.77246 105.00000  2.643913
2      2  12.41517  1.989828 2.381379 21.11724  93.84483  2.424828
3      3  13.11784  3.322157 2.431765 21.33333  99.33333  1.675686
  Flavanoids Non_flav Proantho C_Intensity      Hue OD280_315  Proline
1  2.6689855  0.2966667  1.832899    4.990725  1.0696522  2.970000  984.6957
2  2.3398276  0.3668966  1.678103    3.280345  1.0579310  2.978448  573.3793
3  0.8105882  0.4443137  1.164314    7.170980  0.6913725  1.709804  622.4902
```

This gives us the `mean` by the cluster for each of the 13 variables in the data. With complete linkage done, let's give Ward a try:

```
> aggregate(wine[, -1], list(ward3), mean)
  Group.1 Alcohol MalicAcid      Ash Alk_ash magnesium T_phenols
1      1  13.66922  1.970000 2.463125 17.52812 106.15625  2.850000
2      2  12.20397  1.938966 2.215172 20.20862  92.55172  2.262931
3      3  13.06161  3.166607 2.412857 21.00357  99.85714  1.694286
  Flavanoids Non_flav Proantho C_Intensity      Hue OD280_315  Proline
1  3.0096875  0.2910937  1.908125    5.450000  1.071406  3.158437 1076.0469
2  2.0881034  0.3553448  1.686552    2.895345  1.060000  2.862241  501.4310
3  0.8478571  0.4494643  1.129286    6.850179  0.721000  1.727321  624.9464
```

The numbers are very close. The cluster one for Ward's method does have slightly higher values for all the variables. For cluster two of Ward's method, the mean values are smaller except for Hue. This would be something to share with someone who has the domain expertise to assist in the interpretation. We can help this effort by plotting the values for the variables by the cluster for the two methods.

A nice plot to compare distributions is the **boxplot**. The boxplot will show us the minimum, first quartile, median, third quartile, maximum, and potential outliers.

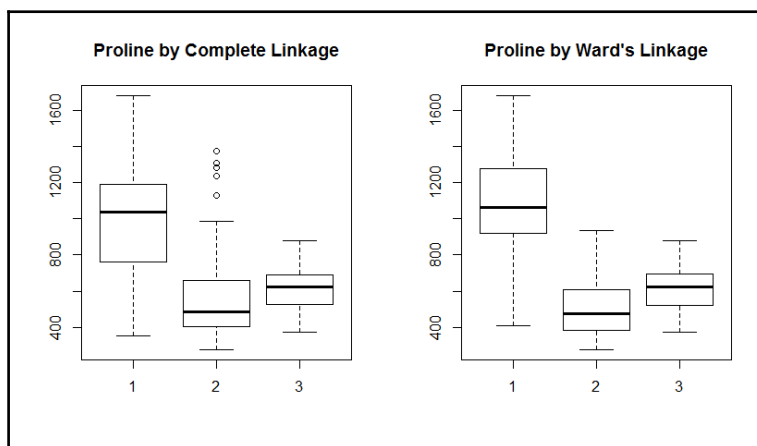
Let's build a comparison plot with two boxplot graphs with the assumption that we are curious about the `Proline` values for each clustering method. The first thing to do is to prepare our plot area in order to display the graphs side by side. This is done with the `par()` function:

```
> par(mfrow = c (1, 2))
```

Here, we specified that we wanted one row and two columns with `mfrow = c(1, 2)`. If you want it as two rows and one column, then it would have been `mfrow = c(2, 1)`. In the `boxplot()` function, we will need to specify that the y axis values are a function of the x axis values with the tilde `~` symbol:

```
> boxplot(wine$Proline ~ comp3, data = wine,
          main="Proline by Complete Linkage")

> boxplot(wine$Proline ~ ward3, data = wine,
          main = "Proline by Ward's Linkage")
```



Looking at the boxplot, the thick boxes represent the first quartile, median (the thick horizontal line in the box), and the third quartile, which is the **interquartile range**. The ends of the dotted lines, commonly referred to as **whiskers** represent the minimum and maximum values. You can see that cluster two in complete linkage has five small circles above the maximum. These are known as **suspected outliers** and are calculated as greater than plus or minus 1.5 times the interquartile range.

Any value that is greater than plus or minus three times the interquartile range are deemed outliers and are represented as solid black circles. For what it's worth, clusters one and two of Ward's linkage have tighter interquartile ranges with no suspected outliers.



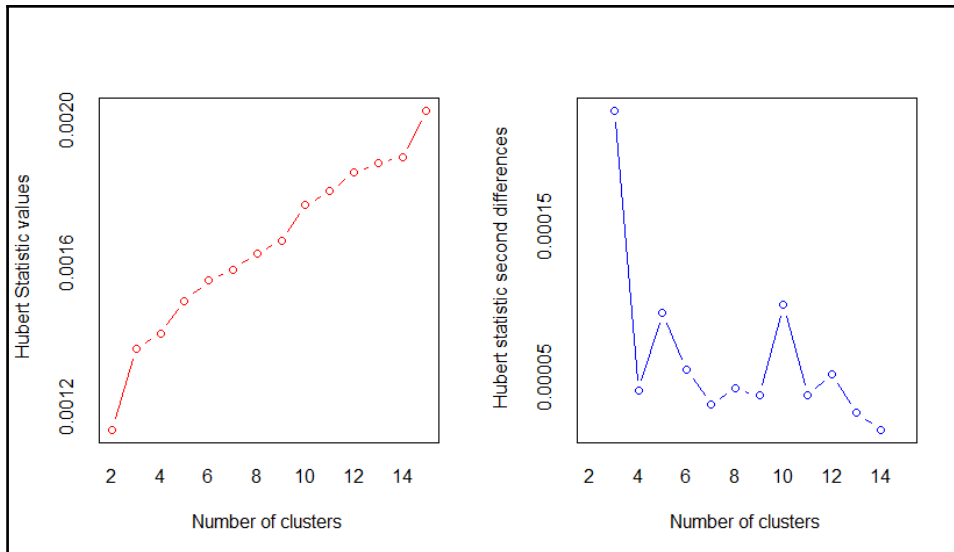
Looking at the boxplots for each of the variables could help you, and a domain expert can determine the best hierarchical clustering method to accept. With this in mind, let's move on to k-means clustering.

K-means clustering

As we did with hierarchical clustering, we can also use `NbClust()` to determine the optimum number of clusters for k-means. All you need to do is specify `kmeans` as the method in the function. Let's also loosen up the maximum number of clusters to 15. I've abbreviated the following output to just the majority rules portion:

```
> numKMeans <- NbClust(df, min.nc = 2, max.nc = 15, method =  
  "kmeans")  
* Among all indices:  
* 4 proposed 2 as the best number of clusters  
* 15 proposed 3 as the best number of clusters  
* 1 proposed 10 as the best number of clusters  
* 1 proposed 12 as the best number of clusters  
* 1 proposed 14 as the best number of clusters  
* 1 proposed 15 as the best number of clusters  
  
***** Conclusion *****  
  
* According to the majority rule, the best number of clusters is 3
```

Once again, three clusters appear to be the optimum solution. Here is the Hubert plot, which confirms this:



In R, we can use the `kmeans()` function to do this analysis. In addition to the input data, we have to specify the number of clusters we are solving for and a value for random assignments, the `nstart` argument. We will also need to specify a random seed:

```
> set.seed(1234)

> km <- kmeans(df, 3, nstart = 25)
```

Creating a table of the clusters gives us a sense of the distribution of the observations between them:

```
> table(km$cluster)

 1  2  3
62 65 51
```

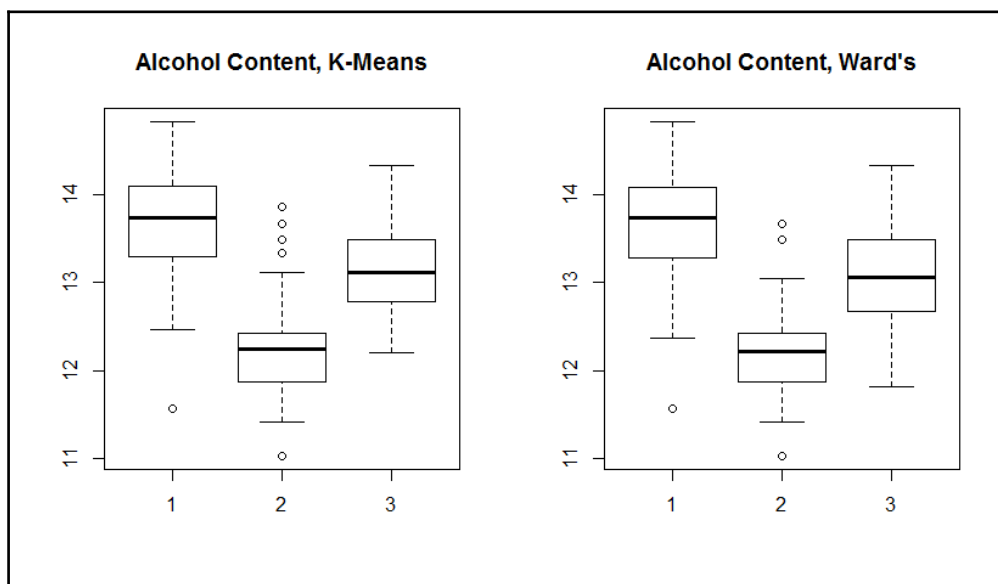
The number of observations per cluster is well-balanced. I have seen on a number of occasions with larger datasets and many more variables that no number of k-means yields a promising and compelling result. Another way to analyze the clustering is to look at a matrix of the cluster centers for each variable in each cluster:

```
> km$centers
  Alcohol  MalicAcid      Ash  Alk_ash  magnesium  T_phenols
1  0.8328826 -0.3029551  0.3636801 -0.6084749  0.57596208  0.88274724
2 -0.9234669 -0.3929331 -0.4931257  0.1701220 -0.49032869 -0.07576891
3  0.1644436  0.8690954  0.1863726  0.5228924 -0.07526047 -0.97657548
 Flavanoids  Non_flav  Proantho  C_Intensity      Hue  OD280_315
1  0.97506900 -0.56050853  0.57865427  0.1705823  0.4726504  0.7770551
2  0.02075402 -0.03343924  0.05810161 -0.8993770  0.4605046  0.2700025
3 -1.21182921  0.72402116 -0.77751312  0.9388902 -1.1615122 -1.2887761
  Proline
1  1.1220202
2 -0.7517257
3 -0.4059428
```

Note that cluster one has, on average, a higher alcohol content. Let's produce a boxplot to look at the distribution of alcohol content in the same manner as we did before and also compare it to Ward's:

```
> boxplot(wine$Alcohol ~ km$cluster, data = wine,
          main = "Alcohol Content, K-Means")

> boxplot(wine$Alcohol ~ ward3, data = wine,
          main = "Alcohol Content, Ward's")
```



The alcohol content for each cluster is almost exactly the same. On the surface, this tells me that three clusters is the proper latent structure for the wines and there is little difference between using k-means or hierarchical clustering. Finally, let's do the comparison of the k-means clusters versus the cultivars:

```
> table(km$cluster, wine$Class)

  1  2  3
1 59  3  0
2  0 65  0
3  0  3 48
```

This is very similar to the distribution produced by Ward's method, and either one would probably be acceptable to our hypothetical sommelier.

However, to demonstrate how you can cluster on data with both numeric and non-numeric values, let's work through some more examples.

Gower and PAM

To begin this step, we will need to wrangle our data a little bit. As this method can take variables that are factors, we will convert alcohol to either high or low content. It also takes only one line of code utilizing the `ifelse()` function to change the variable to a factor. What this will accomplish is if alcohol is greater than zero, it will be `High`, otherwise, it will be `Low`:

```
> wine$Alcohol <- as.factor(ifelse(df$Alcohol > 0, "High", "Low"))
```

We are now ready to create the dissimilarity matrix using the `daisy()` function from the `cluster` package and specifying the method as `gower`:

```
> disMatrix <- daisy(wine[, -1], metric = "gower")
```

The creation of the cluster object--let's call it `pamFit`--is done with the `pam()` function, which is a part of the `cluster` package. We will create three clusters in this example and create a table of the cluster size:

```
> set.seed(123)

> pamFit <- pam(disMatrix, k = 3)

> table(pamFit$clustering)

 1  2  3
63 67 48
```

Now, let's see how it does compared to the cultivar labels:

```
> table(pamFit$clustering, wine$Class)

   1  2  3
1 57  6  0
2  2 64  1
3  0  1 47
```

Let's take this solution and build a descriptive statistics table using the power of the `compareGroups` package. In base R, creating presentation-worthy tables can be quite difficult and this package offers an excellent solution. The first step is to create an object of the descriptive statistics by the cluster with the `compareGroups()` function of the package.

Then, using `createTable()`, we will turn the statistics to an easy-to-export table, which we will do as a .csv. If you want, you can also export the table as a PDF, HTML, or the LaTeX format:

```
> wine$cluster <- pamFit$clustering

> group <- compareGroups(cluster ~ ., data = wine)

> clustab <- createTable(group)

> clustab

-----Summary descriptives table by 'cluster'-----
```

	1 N=63	2 N=67	3 N=48	p.overall
Class	1.10 (0.30)	1.99 (0.21)	2.98 (0.14)	<0.001
Alcohol:				<0.001
High	63 (100%)	1 (1.49%)	28 (58.3%)	
Low	0 (0.00%)	66 (98.5%)	20 (41.7%)	
MalicAcid	1.98 (0.83)	1.92 (0.90)	3.39 (1.05)	<0.001
Ash	2.42 (0.27)	2.27 (0.31)	2.44 (0.18)	0.001
Alk_ash	17.2 (2.73)	20.2 (3.28)	21.5 (2.21)	<0.001
magnesium	105 (11.6)	95.6 (17.2)	98.5 (10.6)	0.001
T_phenols	2.82 (0.36)	2.24 (0.55)	1.68 (0.36)	<0.001
Flavanoids	2.94 (0.47)	2.07 (0.70)	0.79 (0.31)	<0.001
Non_flav	0.29 (0.08)	0.36 (0.12)	0.46 (0.12)	<0.001
Proantho	1.86 (0.47)	1.64 (0.59)	1.17 (0.41)	<0.001
C_Intensity	5.41 (1.31)	3.05 (0.89)	7.41 (2.29)	<0.001
Hue	1.07 (0.13)	1.05 (0.20)	0.68 (0.12)	<0.001
OD280_315	3.10 (0.39)	2.80 (0.53)	1.70 (0.27)	<0.001
Proline	1065 (280)	533 (171)	628 (116)	<0.001
comp_cluster	1.16 (0.37)	1.81 (0.50)	3.00 (0.00)	<0.001

This table shows the proportion of the factor levels by the cluster, and for the numeric variables, the mean and standard deviation are displayed in parentheses. To export the table to a .csv file, just use the `export2csv()` function:

```
> export2csv(clustab, file = "wine_clusters.csv")
```


If you open this file, you will get this table, which is conducive to further analysis and can be easily manipulated for presentation purposes:

	1	2	3	p.overall
	N=60	N=69	N=49	
Alcohol:				<0.001
High	58 (96.7%)	6 (8.70%)	28 (57.1%)	
Low	2 (3.33%)	63 (91.3%)	21 (42.9%)	
MalicAcid	-0.31 (0.62)	-0.37 (0.89)	0.90 (0.97)	<0.001
Ash	0.28 (0.89)	-0.42 (1.14)	0.25 (0.67)	<0.001
Alk_ash	-0.75 (0.76)	0.24 (1.00)	0.58 (0.67)	<0.001
magnesium	0.43 (0.77)	-0.34 (1.18)	-0.05 (0.77)	<0.001
T_phenols	0.87 (0.54)	-0.06 (0.86)	-0.99 (0.56)	<0.001
Flavanoids	0.96 (0.40)	0.04 (0.70)	-1.23 (0.31)	<0.001
Non_flav	-0.58 (0.56)	0.00 (0.98)	0.71 (1.00)	<0.001
Proantho	0.55 (0.72)	0.05 (1.06)	-0.75 (0.72)	<0.001
C_Intensity	0.20 (0.53)	-0.87 (0.38)	0.99 (1.00)	<0.001
Hue	0.46 (0.51)	0.44 (0.89)	-1.19 (0.51)	<0.001
OD280_315	0.77 (0.50)	0.25 (0.69)	-1.30 (0.38)	<0.001
Proline	1.14 (0.74)	-0.72 (0.51)	-0.38 (0.37)	<0.001
comp_cluster	-0.94 (0.42)	-0.14 (0.59)	1.35 (0.00)	<0.001
ward_cluster	-1.16 (0.00)	0.11 (0.49)	1.27 (0.00)	<0.001
km_cluster	-1.16 (0.16)	0.06 (0.34)	1.33 (0.00)	<0.001
class:				<0.001
1	59 (98.3%)	0 (0.00%)	0 (0.00%)	
2	1 (1.67%)	69 (100%)	1 (2.04%)	
3	0 (0.00%)	0 (0.00%)	48 (98.0%)	

Finally, we'll create a dissimilarity matrix with Random Forest and create three clusters with PAM.

Random Forest and PAM

To perform this method in R, you can use the `randomForest()` function. After seeding the random seed, simply create the model object. In the following code, I specify the number of trees as 2000 and set proximity measure to TRUE:

```
> set.seed(1)

> rf <- randomForest(x = wine[, -1], ntree = 2000, proximity = T)

> rf

Call:
randomForest(x = wine[, -1], ntree = 2000, proximity = T)
```

```
      Type of random forest: unsupervised
      Number of trees: 2000
      No. of variables tried at each split: 3
```

As you can see, placing a call to `rf` did not provide any meaningful output other than the variables sampled at each split (`mtry`). Let's examine the first five rows and first five columns of the $N \times N$ matrix:

```
> dim(rf$proximity)
[1] 178 178

> rf$proximity[1:5, 1:5]
      1      2      3      4      5
1 1.0000000 0.2593985 0.2953586 0.36013986 0.17054264
2 0.2593985 1.0000000 0.1307420 0.16438356 0.11029412
3 0.2953586 0.1307420 1.0000000 0.29692833 0.23735409
4 0.3601399 0.1643836 0.2969283 1.00000000 0.08076923
5 0.1705426 0.1102941 0.2373541 0.08076923 1.00000000
```

One way to think of the values is that they are the percentage of times those two observations show up in the same terminal nodes! Looking at variable importance we see that the transformed Alcohol input could be dropped. We will keep it for simplicity:

```
> importance(rf)
      MeanDecreaseGini
Alcohol      0.5614071
MalicAcid    6.8422540
Ash          6.4693717
Alk_ash      5.9103567
magnesium    5.9426505
T_phenols    6.2928709
Flavanoids   6.2902370
Non_flav     5.7312940
Proantho     6.2657613
C_Intensity  6.5375605
Hue          6.3297808
OD280_315    6.4894731
Proline      6.6105274
```

It is now just a matter of creating the dissimilarity matrix, which transforms the proximity values (*square root*($1 - \text{proximity}$)) as follows:

```
> dissMat <- sqrt(1 - rf$proximity)

> dissMat[1:2, 1:2]
      1      2
1 0.0000000 0.8605821
2 0.8605821 0.0000000
```

We now have our input features, so let's run a PAM clustering as we did earlier:

```
> set.seed(123)

> pamRF <- pam(dissMat, k = 3)

> table(pamRF$clustering)

 1  2  3
62 68 48

> table(pamRF$clustering, wine$Class)

 1 2 3
1 57 5 0
2  2 64 2
3  0  2 46
```

These results are comparable to the other techniques applied. Can you improve the results by tuning the Random Forest?

If you have messy data for a clustering problem, consider using Random Forest.

Summary

In this chapter, we started exploring unsupervised learning techniques. We focused on cluster analysis to both provide data reduction and data understanding of the observations.

Four methods were introduced: the traditional hierarchical and k-means clustering algorithms, along with PAM, incorporating two different inputs (Gower and Random Forest). We applied these four methods to find a structure in Italian wines coming from three different cultivars and examined the results.

In the next chapter, we will continue exploring unsupervised learning, but instead of finding structure among the observations, we will focus on finding structure among the variables in order to create new features that can be used in a supervised learning problem.

9

Principal Components Analysis

"Some people skate to the puck. I skate to where the puck is going to be."

- Wayne Gretzky

This chapter is the second one where we will focus on unsupervised learning techniques. In the previous chapter, we covered cluster analysis, which provides us with the groupings of similar observations. In this chapter, we will see how to reduce the dimensionality and improve the understanding of our data by grouping the correlated variables with **Principal Components Analysis (PCA)**. Then, we will use the principal components in supervised learning.

In many datasets, particularly in the social sciences, you will see many variables highly correlated with each other. They may additionally suffer from high dimensionality or, as it is better known, the **curse of dimensionality**. This is a problem because the number of samples needed to estimate a function grows exponentially with the number of input features. In such datasets, there may be the case that some variables are redundant as they end up measuring the same constructs, for example, income and poverty or depression and anxiety. The goal then is to use PCA in order to create a smaller set of variables that capture most of the information from the original set of variables, thus simplifying the dataset and often leading to hidden insights. These new variables (principal components) are highly uncorrelated with each other. In addition to supervised learning, it is also very common to use these components to perform data visualization.

From over a decade of either doing or supporting analytics using PCA, it has been my experience that it is widely used but poorly understood, especially among people who don't do the analysis but consume the results. It is intuitive to understand that you are creating a new variable from the other correlated variables. However, the technique itself is shrouded in potentially misunderstood terminology and mathematical concepts that often bewilder the layperson. The intention here is to provide a good foundation on what it is and how to use it by covering the following:

- Preparing a dataset for PCA
- Conducting PCA
- Selecting our principal components
- Building a predictive model using principal components
- Making out of sample predictions using the predictive model

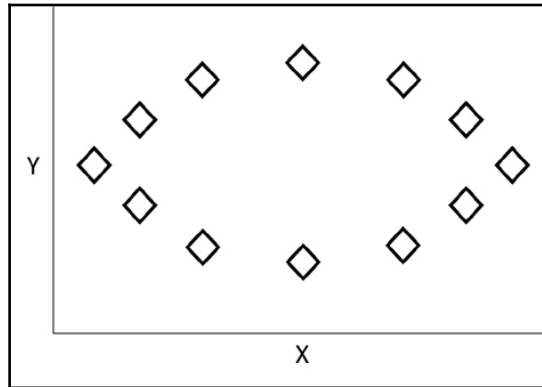
An overview of the principal components

PCA is the process of finding the principal components. What exactly are these?

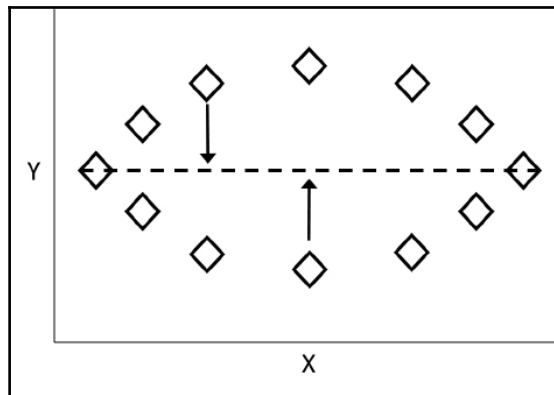
We can consider that a component is a normalized linear combination of the features (James, 2012). The first principal component in a dataset is the linear combination that captures the maximum variance in the data. A second component is created by selecting another linear combination that maximizes the variance with the constraint that its direction is perpendicular to the first component. The subsequent components (equal to the number of variables) would follow this same rule.

A couple of things here. This definition describes the **linear combination**, which is one of the key assumptions in PCA. If you ever try and apply PCA to a dataset of variables having a low correlation, you will likely end up with a meaningless analysis. Another key assumption is that the mean and variance for a variable are sufficient statistics. What this tells us is that the data should fit a normal distribution so that the covariance matrix fully describes our dataset, that is, **multivariate normality**. PCA is fairly robust to non-normally distributed data and is even used in conjunction with binary variables, so the results are still interpretable.

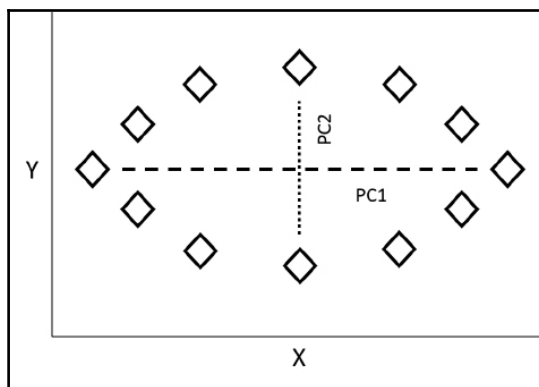
Now, what is this direction described here and how is the linear combination determined? The best way to grasp this subject is with a visualization. Let's take a small dataset with two variables and plot it. PCA is sensitive to scale, so the data has been scaled with a mean of zero and standard deviation of one. You can see in the following figure that this data happens to form the shape of an oval with the diamonds representing each observation:



Looking at the plot, the data has the most variance along the x axis, so we can draw a dashed horizontal line to represent our **first principal component** as shown in the following image. This component is the linear combination of our two variables or $PC1 = \alpha_{11}X_1 + \alpha_{12}X_2$, where the coefficient weights are the variable loadings on the principal component. They form the basis of the direction along which the data varies the most. This equation is constrained by 1 in order to prevent the selection of arbitrarily high values. Another way to look at this is that the dashed line minimizes the distance between itself and the data points. This distance is shown for a couple of points as arrows, as follows:



The **second principal component** is then calculated in the same way, but it is uncorrelated with the first, that is, its direction is at a right angle or orthogonal to the first principal component. The following plot shows the second principal component added as a dotted line:



With the principal component loadings calculated for each variable, the algorithm will then provide us with the principal component scores. The scores are calculated for each principal component for each observation. For **PC1** and the first observation, this would equate to the formula: $Z_{11} = \alpha_{11} * (X_{11} - \text{average of } X_1) + \alpha_{12} * (X_{12} - \text{average of } X_2)$. For **PC2** and the first observation, the equation would be $Z_{12} = \alpha_{21} * (X_{11} - \text{average of } X_1) + \alpha_{22} * (X_{12} - \text{average of } X_2)$. These principal component scores are now the new feature space to be used in whatever analysis you will undertake.

Recall that the algorithm will create as many principal components as there are variables, accounting for 100 percent of the possible variance. So, how do we narrow down the components to achieve the original objective in the first place? There are some heuristics that one can use, and in the upcoming modeling process, we will look at the specifics; but a common method to select a principal component is if its **eigenvalue** is greater than one. While the algebra behind the estimation of eigenvalues and **eigenvectors** is outside the scope of this book, it is important to discuss what they are and how they are used in PCA.



The optimized linear weights are determined using linear algebra in order to create what is referred to as an eigenvector. They are optimal because no other possible combination of weights could explain variation better than they do. The eigenvalue for a principal component then is the total amount of variation that it explains in the entire dataset.

Recall that the equation for the first principal component is $PC1 = \alpha_{11}X_1 + \alpha_{12}X_2$.

As the first principal component accounts for the largest amount of variation, it will have the largest eigenvalue. The second component will have the second highest eigenvalue and so forth. So, an eigenvalue greater than one indicates that the principal component accounts for more variance than any of the original variables does by itself. If you standardize the sum of all the eigenvalues to one, you will have the percentage of the total variance that each component explains. This will also aid you in determining a proper cut-off point.

The eigenvalue criterion is certainly not a hard-and-fast rule and must be balanced with your knowledge of the data and business problem at hand. Once you have selected the number of principal components, you can rotate them in order to simplify their interpretation.

Rotation

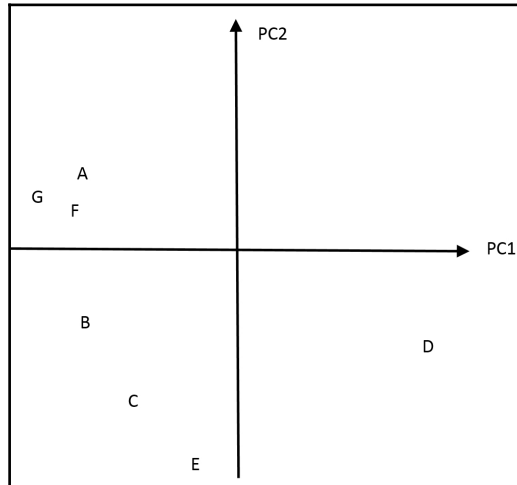
Should you rotate or not? As stated previously, rotation helps in the interpretation of the principal components by modifying the loadings of each variable. The overall variation explained by the rotated number of components will not change, but the contributions to the total variance explained by each component will change. What you will find by rotation is that the loading values will either move farther or closer to zero, theoretically aiding in identifying those variables that are important to each principal component. This is an attempt to associate a variable to only one principal component. Remember that this is unsupervised learning, so you are trying to understand your data, not test some hypothesis. In short, rotation aids you in this endeavor.

The most common form of principal component rotation is known as **varimax**. There are other forms such as **quartimax** and **equimax**, but we will focus on varimax rotation. In my experience, I've never seen the other methods provide better solutions. Trial and error on your part may be the best way to decide the issue.

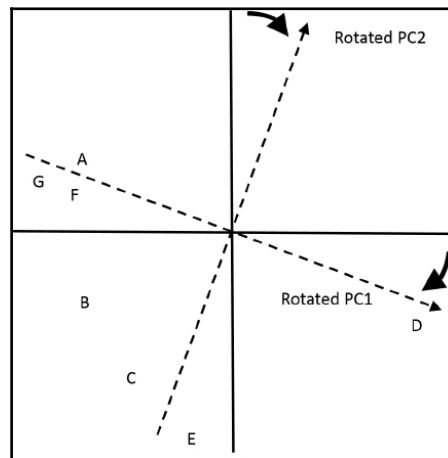


With varimax, we are maximizing the sum of the variances of the squared loadings. The varimax procedure rotates the axis of the feature space and their coordinates without changing the locations of the data points.

Perhaps, the best way to demonstrate this is via another simple illustration. Let's assume that we have a dataset of variables **A** through **G** and we have two principal components. Plotting this data, we will end up with the following illustration:



For the sake of argument, let's say that variable A's loadings are -0.4 on **PC1** and 0.1 on **PC2**. Now, let's say that variable D's loadings are 0.4 on **PC1** and -0.3 on **PC2**. For point E, the loadings are -0.05 and -0.7, respectively. Note that the loadings will follow the direction of the principal component. After running a varimax procedure, the rotated components will look as follows:



The following are the new loadings on **PC1** and **PC2** after rotation:

- Variable **A**: -0.5 and 0.02
- Variable **D**: 0.5 and -0.3
- Variable **E**: 0.15 and -0.75

The loadings have changed but the data points have not. With this simple illustration, we can't say that we have simplified the interpretation, but this should help you understand what is happening during the rotation of the principal components.

Business understanding

For this example, we will delve into the world of sports; in particular, the **National Hockey League (NHL)**. Much work has been done on baseball (think of the book and movie, *Moneyball*) and football; both are American and games that people around the world play with their feet. For my money, there is no better spectator sport than hockey. Perhaps that is an artifact of growing up on the frozen prairie of North Dakota. Nonetheless, we can consider this analysis as our effort to start a MoneyPuck movement.

In this analysis, we will look at the statistics for 30 NHL teams in a data set I've compiled from www.nhl.com and www.puckalytics.com. The goal is to build a model that predicts the total points for a team from an input feature space developed using PCA in order to provide us with some insight on what it takes to be a top professional team. We will learn a model from the 2015-16 season, which saw the Pittsburgh Penguins crowned as champions, and then test its performance on the current season's results as of February 15, 2017. The files are `nhlTrain.csv` and `nhlTest.csv` on <https://github.com/datameister66/data/>.

NHL standings are based on a points system, so our outcome will be team points per game. It is important to understand how the NHL awards points to the teams. Unlike football or baseball where only wins and losses count, professional hockey uses the following point system for each game:

- The winner gets two points whether that is in regulation, overtime, or as a result of the post-overtime shootout
- A regulation loser receives no points
- An overtime or shootout loser receives one point; the so-called **loser point**

The NHL started this point system in 2005 and it is not without controversy, but it hasn't detracted from the game's elegant and graceful violence.

Data understanding and preparation

To begin with, we will load the necessary packages in order to download the data and conduct the analysis. Please ensure that you have these packages installed prior to loading:

```
> library(ggplot2) #support scatterplot  
  
> library(psych) #PCA package
```

Let's also assume you've put the two .csv files into your working directory, so read the training data using the `read.csv()` function:

```
> train <- read.csv("NHLtrain.csv")
```

Examine the data using the structure function, `str()`. For brevity, I've included only the first few lines of the output of the command:

```
> str(train)  
'data.frame': 30 obs. of 15 variables:  
 $ Team : Factor w/ 30 levels "Anaheim","Arizona",...: 1 2 3 4 5 6 7  
   8 9 10 ...  
 $ ppg : num 1.26 0.95 1.13 0.99 0.94 1.05 1.26 1 0.93 1.33 ...  
 $ Goals_For : num 2.62 2.54 2.88 2.43 2.79 2.39 2.85 2.59 2.6 3.23  
   ...  
 $ Goals_Against: num 2.29 2.98 2.78 2.62 3.13 2.7 2.52 2.93 3.02  
   2.78 ...
```

The next thing that we will need to do is look at the variable names.

```
> names(train)  
[1] "Team" "ppg" "Goals_For" "Goals_Against" "Shots_For"  
[6] "Shots_Against" "PP_perc" "PK_perc" "CF60_pp" "CA60_sh"  
[11] "OZFOperc_pp" "Give" "Take" "hits" "blks"
```

Let's go over what they mean:

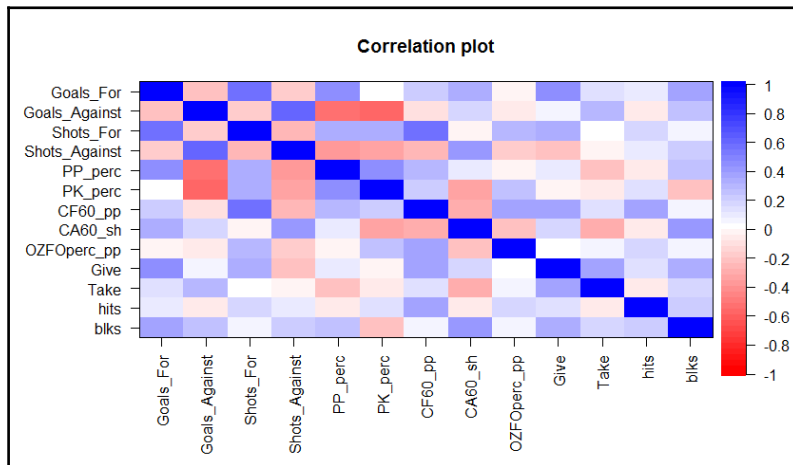
- Team: This is the team's city
- ppg: The average points per game per the point calculation discussed earlier
- Goals_For: The average goals the team scores per game
- Goals_Against: The goals allowed per game
- Shots_For: Shots on goal per game
- Shots_Against: Opponent shots on goal per game
- PP_perc: Percent of power play opportunities the team scores a goal

- PK_perc: Percent of time the team does not allow a goal when their opponent is on the power play
- CF60_pp: The team's Corsi Score per 60 minutes of power play time; Corsi Score is the sum of shots for (Shots_For), shot attempts that miss the net and shots blocked by the opponent
- CA60_sh: The opponents Corsi Score per 60 minutes of opponent power play time i.e. the team is shorthanded
- OZFOperc_pp: The percentage of face offs that took place in the offensive zone while the team was on the power play
- Give: The average number per game that the team gives away the puck
- Take: The average number per game that the team gains control of the puck
- hits: The average number of the team's bodychecks per game
- blks: The average number per game of the team's blocking an opponent's shot on goal

We'll need to have the data standardized with mean 0 and standard deviation of 1. Once we do that we can create and plot the correlations of the input features using the `cor.plot()` function available in the `psych` package:

```
> train.scale <- scale(train[, -1:-2])  
  
> nhl.cor <- cor(train.scale)  
  
> cor.plot(nhl.cor)
```

The following is the output of the preceding command:



A couple of things are of interest. Notice that `Shots_For` is correlated with `Goals_For` and conversely, `Shots_Against` with `Goals_Against`. There also is some negative correlation with `PP_perc` and `PK_perc` with `Goals_Against`.

As such, this should be an adequate dataset to extract several principal components.

Please note that these are features/variables that I've selected based on my interest. There are a bunch of different statistics you can gather on your own and see if you can improve the predictive power.

Modeling and evaluation

For the modeling process, we will follow the following steps:

1. Extract the components and determine the number to retain.
2. Rotate the retained components.
3. Interpret the rotated solution.
4. Create the factor scores.
5. Use the scores as input variables for regression analysis and evaluate the performance on the test data.

There are many different ways and packages to conduct PCA in R, including what seems to be the most commonly used `prcomp()` and `princomp()` functions in base R. However, for my money, it seems that the `psych` package is the most flexible with the best options.

Component extraction

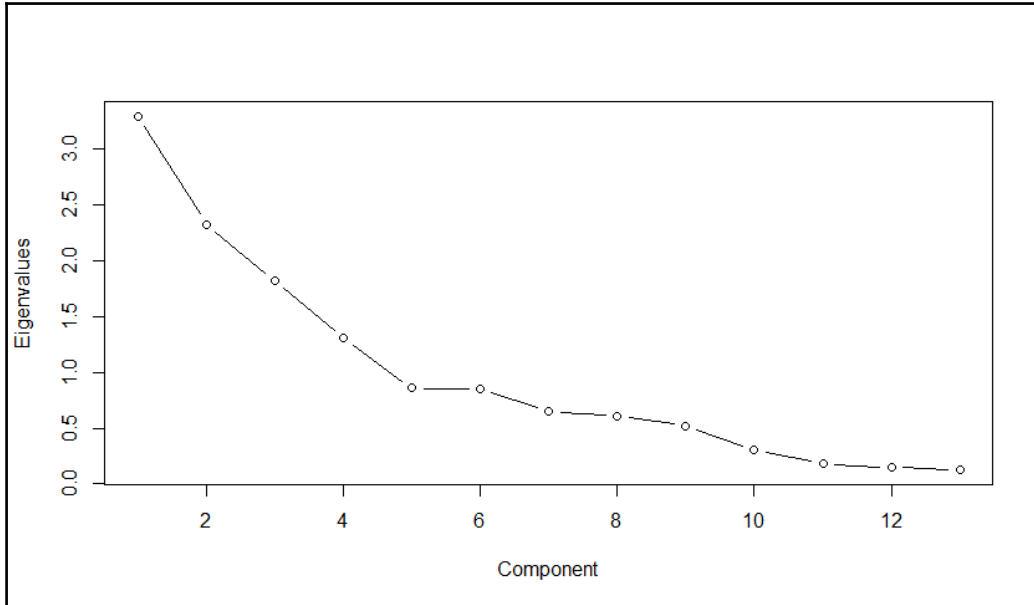
To extract the components with the `psych` package, you will use the `principal()` function. The syntax will include the data and whether or not we want to rotate the components at this time:

```
> pca <- principal(train.scale, rotate="none")
```

You can examine the components by calling the `pca` object that we created. However, my primary intent is to determine what should be the number of components to retain. For that, a scree plot will suffice. A scree plot can aid you in assessing the components that explain the most variance in the data. It shows the `Component` number on the *x*-axis and their associated `Eigenvalues` on the *y*-axis:

```
> plot(pca$values, type="b", ylab="Eigenvalues", xlab="Component")
```

The following is the output of the preceding command:



What you are looking for is a point in the scree plot where the rate of change decreases. This will be what is commonly called an elbow or bend in the plot. That elbow point in the plot captures the fact that additional variance explained by a component does not differ greatly from one component to the next. In other words, it is the break point where the plot flattens out. In this plot, five components look pretty compelling.

Another rule I've learned over the years is that you should capture about 70% of the total variance, which means that the cumulative variance explained by each of the selected components accounts for 70 percent of the variance explained by all the components.

Orthogonal rotation and interpretation

As we discussed previously, the point behind rotation is to maximize the loadings of the variables on a specific component, which helps in simplifying the interpretation by reducing/eliminating the correlation among these components. The method to conduct orthogonal rotation is known as "varimax". There are other non-orthogonal rotation methods that allow correlation across factors/components. The choice of the rotation methodology that you will use in your profession should be based on the pertinent literature, which exceeds the scope of this chapter. Feel free to experiment with this dataset.

I think that when in doubt, the starting point for any PCA should be orthogonal rotation.

For this process, we will simply turn back to the `principal()` function, slightly changing the syntax to account for 5 components and orthogonal rotation, as follows:

```
> pca.rotate <- principal(train.scale, nfactors = 5, rotate =  
  "varimax")  
  
> pca.rotate  
Principal Components Analysis  
Call: principal(r = train.scale, nfactors = 5, rotate = "varimax")  
Standardized loadings (pattern matrix) based upon correlation  
matrix
```

	RC1	RC2	RC5	RC3	RC4	h2	u2	com
Goals_For	-0.21	0.82	0.21	0.05	-0.11	0.78	0.22	1.3
Goals_Against	0.88	-0.02	-0.05	0.21	0.00	0.82	0.18	1.1
Shots_For	-0.22	0.43	0.76	-0.02	-0.10	0.81	0.19	1.8
Shots_Against	0.73	-0.02	-0.20	-0.29	0.20	0.70	0.30	1.7
PP_perc	-0.73	0.46	-0.04	-0.15	0.04	0.77	0.23	1.8
PK_perc	-0.73	-0.21	0.22	-0.03	0.10	0.64	0.36	1.4
CF60_pp	-0.20	0.12	0.71	0.24	0.29	0.69	0.31	1.9
CA60_sh	0.35	0.66	-0.25	-0.48	-0.03	0.85	0.15	2.8
OZFOperc_pp	-0.02	-0.18	0.70	-0.01	0.11	0.53	0.47	1.2
Give	-0.02	0.58	0.17	0.52	0.10	0.65	0.35	2.2
Take	0.16	0.02	0.01	0.90	-0.05	0.83	0.17	1.1
hits	-0.02	-0.01	0.27	-0.06	0.87	0.83	0.17	1.2
blks	0.19	0.63	-0.18	0.14	0.47	0.70	0.30	2.4

	RC1	RC2	RC5	RC3	RC4
SS loadings	2.69	2.33	1.89	1.55	1.16
Proportion Var	0.21	0.18	0.15	0.12	0.09
Cumulative Var	0.21	0.39	0.53	0.65	0.74
Proportion Explained	0.28	0.24	0.20	0.16	0.12
Cumulative Proportion	0.28	0.52	0.72	0.88	1.00

There are two important things to digest here in the output. The first is the variable loadings for each of the five components that are labeled RC1 through RC5. We see with component one that `Goals_Against` and `Shots_Against` have high positive loadings, while `PP_perc` and `PK_perc` have high negative loadings. The high loading for component two is `Goals_For`. Component five has high loadings with `Shots_For`, `ff`, and `OZFOperc_pp`. Component three seems to be only about the variables `take` while component four is about hits. Next, we will move on to the second part for examination: the table starting with the sum of square, `SS` loadings. Here, the numbers are the eigenvalues for each component. When they are normalized, you will end up with the `Proportion Explained` row, which as you may have guessed, stands for the proportion of the variance explained by each component. You can see that component one explains 28 percent of all the variance explained by the five rotated components. Remember above I mentioned the heuristic rule that your selected components should account for a minimum of about 70 of the total variation. Well, if you look at the `Cumulative Var` row, you see that these five rotated components account for 74% of the total and we can feel confident we have the right number to go forward with our modeling.

Creating factor scores from the components

We will now need to capture the rotated component loadings as the factor scores for each individual team. These scores indicate how each observation (in our case, the NHL team) relates to a rotated component. Let's do this and capture the scores in a data frame as we will need to use it for our regression analysis:

```
> pca.scores <- data.frame(pca.rotate$scores)

> head(pca.scores)
      RC1      RC2      RC5      RC3      RC4
1 -2.21526408 0.002821488 0.3161588 -0.1572320 1.5278033
2  0.88147630 -0.569239044 -1.2361419 -0.2703150 -0.0113224
3  0.10321189 0.481754024 1.8135052 -0.1606672 0.7346531
4 -0.06630166 -0.630676083 -0.2121434 -1.3086231 0.1541255
5  1.49662977 1.156905747 -0.3222194  0.9647145 -0.6564827
6 -0.48902169 -2.119952370 1.0456190  2.7375097 -1.3735777
```

We now have the scores for each component for each team. These are simply the variables for each observation multiplied by the loadings on each component and then summed. We now can bring in the response (`ppg`) as a column in the data.

```
> pca.scores$ppg <- train$ppg
```

With this done, we will now move on to the predictive model.

Regression analysis

To do this part of the process, we will repeat the steps and code from Chapter 2, *Linear Regression - The Blocking and Tackling of Machine Learning*. If you haven't done so, please look at Chapter 2, *Linear Regression - The Blocking and Tackling of Machine Learning* for some insight on how to interpret the following output.

We will use the following `lm()` function to create our linear model with all the factors as inputs and then summarize the results:

```
> nhl.lm <- lm(ppg ~ ., data = pca.scores)

> summary(nhl.lm)

Call:
lm(formula = ppg ~ ., data = pca.scores)

Residuals:
    Min       1Q   Median       3Q      Max
-0.163274 -0.048189  0.003718  0.038723  0.165905

Coefficients:
            Estimate      Std. Error t value Pr(>|t|)
(Intercept)  1.111333     0.015752  70.551  < 2e-16 ***
RC1          -0.112201     0.016022  -7.003  3.06e-07 ***
RC2           0.070991     0.016022   4.431  0.000177 ***
RC5           0.022945     0.016022   1.432  0.164996
RC3          -0.017782     0.016022  -1.110  0.278044
RC4          -0.005314     0.016022  -0.332  0.743003
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.08628 on 24 degrees of freedom
Multiple R-squared:  0.7502, Adjusted R-squared:  0.6981
F-statistic: 14.41 on 5 and 24 DF, p-value: 1.446e-06
```

The good news is that our overall model is highly significant statistically, with p-value of $1.446e-06$ and Adjusted R-squared is almost 70 percent. The bad news is that three components are not significant. We could simply choose to keep them in our model, but let's see what happens if we exclude them, just keeping **RC1** and **RC2**:

```
> nhl.lm2 <- lm(ppg ~ RC1 + RC2, data = pca.scores)

> summary(nhl.lm2)

Call:
lm(formula = ppg ~ RC1 + RC2, data = pca.scores)
```

```
Residuals:
    Min       1Q   Median       3Q      Max
-0.18914 -0.04430  0.01438  0.05645  0.16469

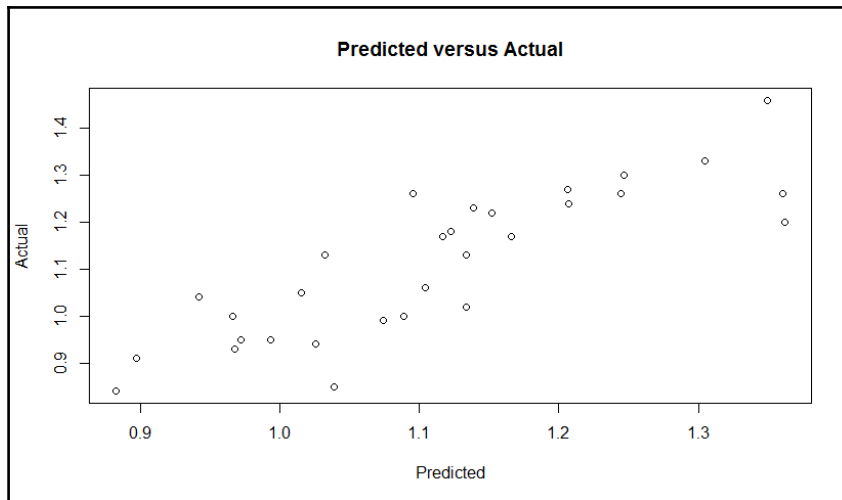
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.11133     0.01587  70.043  < 2e-16 ***
RC1          -0.11220     0.01614  -6.953  1.8e-07 ***
RC2           0.07099     0.01614   4.399  0.000153 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.0869 on 27 degrees of freedom
Multiple R-squared:  0.7149, Adjusted R-squared:  0.6937
F-statistic: 33.85 on 2 and 27 DF, p-value: 4.397e-08
```

This model still achieves roughly the same Adjusted R-squared value (93.07 percent) with statistically significant factor coefficients. I will spare you the details of running the diagnostic tests. Instead, let's look at some plots in order to examine our analysis better. We can do a scatterplot of the predicted and actual values with the base R graphics, as follows:

```
> plot(nhl.lm2$fitted.values, train$ppg,
      main="Predicted versus Actual",
      xlab="Predicted", ylab="Actual")
```

The following is the output of the preceding command:



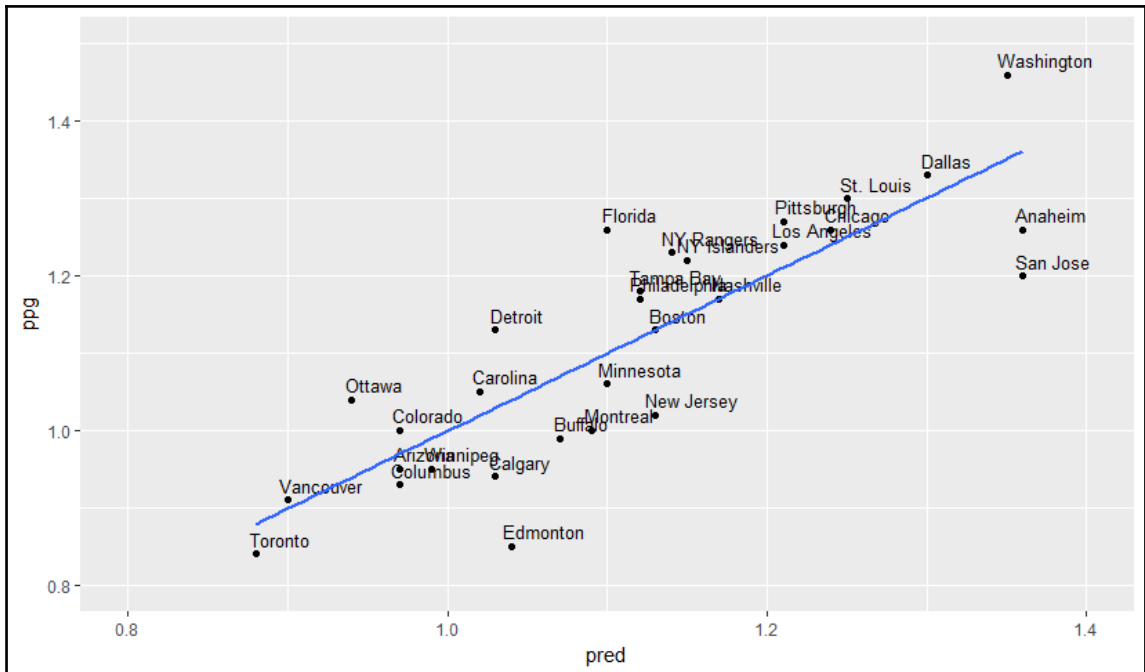
This confirms that our model does a good job of using two components to predict the team's success and also highlights the strong linear relationship between the principal components and team points per game. Let's kick it up a notch by doing a scatterplot using the `ggplot2` package and include the team names in it. The only problem is that it is a very powerful function with many options. There are numerous online resources to help you navigate the `ggplot()` maze, but this code should help you on your way. Let's first create our baseline plot and assign it to an object called `p` then add various plot functionality.

```
> train$pred <- round(nhl.lm2$fitted.values, digits = 2)

> p <- ggplot(train, aes(x = pred,
  y = ppg,
  label = Team))

> p + geom_point() +
  geom_text(size = 3.5, hjust = 0.1, vjust = -0.5, angle = 0) +
  xlim(0.8, 1.4) + ylim(0.8, 1.5) +
  stat_smooth(method = "lm", se = FALSE)
```

The following is the output of the preceding commands:



The syntax to create `p` is very simple. We just specified the data frame and put in `aes()` what we want our `x` and `y` to be along with the variable that we want to use as labels. We then just add layers of neat stuff such as data points. Add whatever you want to the plot by including `+` in the syntax, as follows:

```
> p + geom_point() +
```

We specified how we wanted our `team` labels to appear. It takes quite a bit of trial and error to get the font size and position in order:

```
geom_text() +
```

Now, specify the `x` and `y` axis limits, otherwise the plot will cut out any observations that fall outside them, as follows:

```
xlim() + ylim() +
```

Finally, we added a best fit line with no standard error shading:

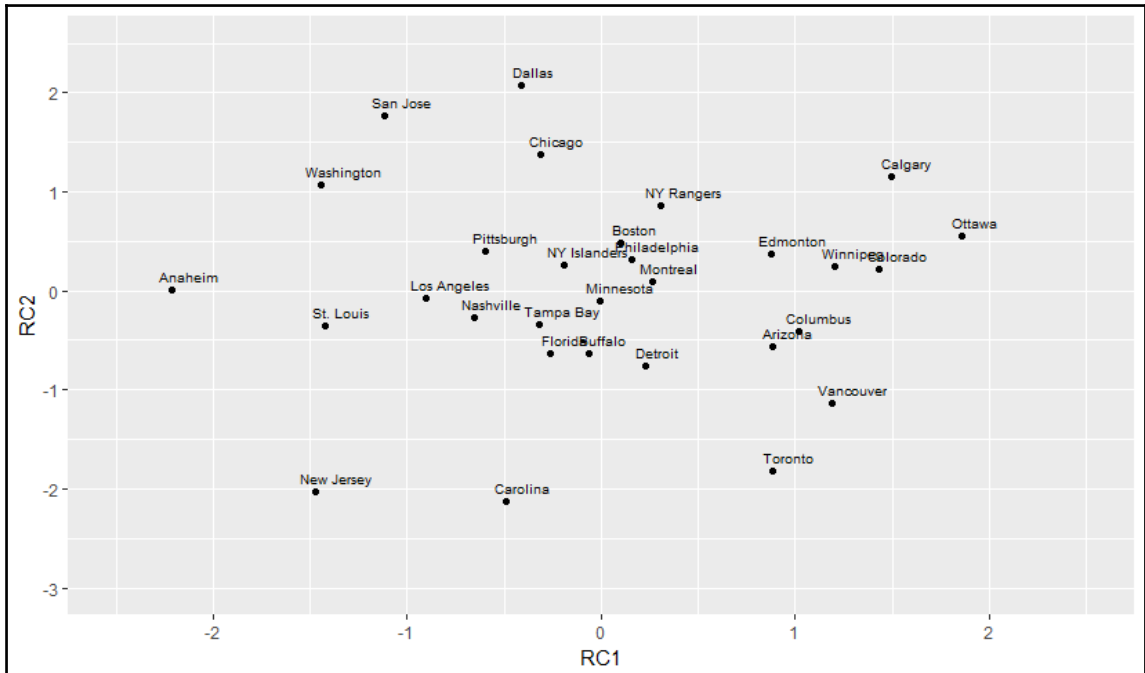
```
stat_smooth(method = "lm", se = FALSE)
```

I guess one way to think about this plot is that the teams below the line underachieved, while those above it overachieved.

Another bit of analysis will be to plot the teams in relationship to their factor scores, what is referred to as a **biplot**. Once again, `ggplot()` facilitates this analysis. Using the preceding code as our guide, let's update it and see what the result is:

```
> pca.scores$Team <- train$Team  
  
> p2 <- ggplot(pca.scores, aes(x = RC1, y = RC2, label = Team))  
  
> p2 + geom_point() +  
  geom_text(size = 2.75, hjust = .2, vjust = -0.75, angle = 0) +  
  xlim(-2.5, 2.5) + ylim(-3.0, 2.5)
```

The output of the preceding command is as follows:



As you can see, the x axis are the team scores on **RC1** and the y axis are the scores on **RC2**. Look at the **Anaheim** ducks with the lowest score on **RC1** and an average score for **RC2**. Now think about the impact of this. With the negative loadings on **RC1** for the power play and penalty kill, along with the positive loading of **Goals_Against**, it would indicate that the team performed well defensively, and was effective shorthanded. By the way, **Pittsburgh** was the eventual winner of the Stanley Cup. Their scores are solid, but nothing noteworthy. Keep in mind that the team had a horrible start to the season and fired the coach they started the season with. It would be interesting to compare how they did on this analysis in the first half of the season versus the latter half. You can evaluate the model error as well, like we did previously. Let's look at **Root Means Squared Error (RMSE)**:

```
> sqrt(mean(nhl.lm2$residuals^2))  
[1] 0.08244449
```

With that done, we need to see how it performs out of sample. We are going to load the test data, predict the team scores on the components, then make our predictions based on the linear model. The `predict` function from the `psych` package will automatically scale the test data:

```
> test <- read.csv("NHLtest.csv")

> test.scores <- data.frame(predict(pca.rotate, test[, c(-1:-2)]))

> test.scores$pred <- predict(nhl.lm2, test.scores)
```

I think we should plot the results as we did above, showing team names. Let's get this all in a data frame:

```
> test.scores$ppg <- test$ppg

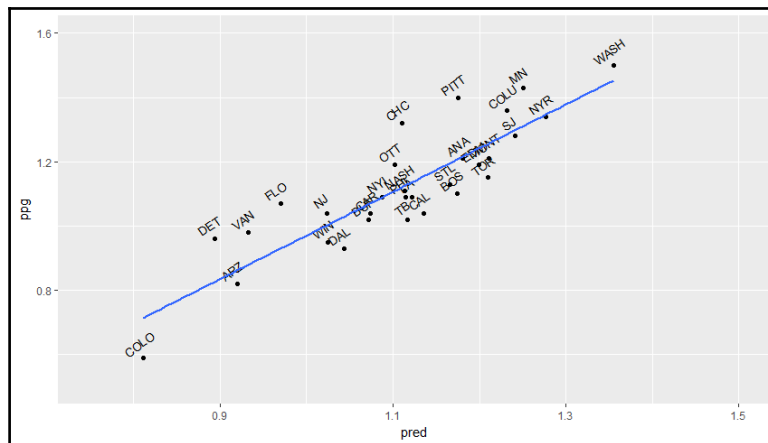
> test.scores$Team <- test$Team
```

Then, utilize the power of `ggplot()`:

```
> p <- ggplot(test.scores, aes(x = pred,
                               y = ppg,
                               label = Team))

> p + geom_point() +
  geom_text(size=3.5, hjust=0.4, vjust = -0.9, angle = 35) +
  xlim(0.75, 1.5) + ylim(0.5, 1.6) +
  stat_smooth(method="lm", se=FALSE)
```

The output of the preceding command is as follows:



I abbreviated the team names in the test data to make it easier to understand. Our points per game leader is the Washington Capitals and the worst team is the Colorado Avalanche. In fact, when I pulled this data, Colorado had lost five straight games. They did break that losing streak as I watched them beat Carolina in overtime.

Finally, let's check the RMSE.

```
> resid <- test.scores$ppg - test.scores$pred  
  
> sqrt(mean(resid^2))  
[1] 0.1011561
```

That is not bad with an output of sample error of 0.1 versus in sample of 0.08. I think we can declare this a valid model. However, there are still a ton of team statistics we could add here to improve predictive power and reduce error. I'll keep working on it, and I hope you do as well.

Summary

In this chapter, we took a second stab at unsupervised learning techniques by exploring PCA, examining what it is, and applying it in a practical fashion. We explored how it can be used to reduce the dimensionality and improve the understanding of the dataset when confronted with numerous highly correlated variables. Then, we applied it to real data from the National Hockey League, using the resulting principal components in a regression analysis to predict total team points. Additionally, we explored ways to visualize the data and the principal components.

As an unsupervised learning technique, it requires some judgment along with trial and error to arrive at an optimal solution that is acceptable to business partners. Nevertheless, it is a powerful tool to extract latent insights and to support supervised learning.

We will next look at using unsupervised learning to develop market basket analyses and recommendation engines in which PCA can play an important role.

10

Market Basket Analysis, Recommendation Engines, and Sequential Analysis

It's much easier to double your business by doubling your conversion rate than by doubling your traffic.

- Jeff Eisenberg, CEO of BuyerLegends.com

I don't see smiles on the faces of people at Whole Foods.

- Warren Buffett

One would have to live on the dark side of the moon in order to not observe each and every day the results of the techniques that we are about to discuss in this chapter. If you visit www.amazon.com, watch movies on www.netflix.com, or visit any retail website, you will be exposed to terms such as "related products", "because you watched...", "customers who bought *x* also bought *y*", or "recommended for you", at every twist and turn. With large volumes of historical real-time or near real-time information, retailers utilize the algorithms discussed here to attempt to increase both the buyer's quantity and value of their purchases.

The techniques to do this can be broken down into two categories: association rules and recommendation engines. Association rule analysis is commonly referred to as market basket analysis as one is trying to understand what items are purchased together. With recommendation engines, the goal is to provide a customer with other items that they will enjoy based on how they have rated previously viewed or purchased items.

Another technique a business can use is to understand the sequence in which you purchase or use their products and services. This is called sequential analysis. A very common implementation of this methodology is to understand how customers click through various webpages and/or links.

In the examples coming up, we will endeavor to explore how R can be used to develop such algorithms. We will not cover their implementation, as that is outside the scope of this book. We will begin with a market basket analysis of purchasing habits at a grocery store, then dig into building a recommendation engine on website reviews, and finally, analyze the sequence of web pages.

An overview of a market basket analysis

Market basket analysis is a data mining technique that has the purpose of finding the optimal combination of products or services and allows marketers to exploit this knowledge to provide recommendations, optimize product placement, or develop marketing programs that take advantage of cross-selling. In short, the idea is to identify which items go well together, and profit from it.

You can think of the results of the analysis as an `if . . . then` statement. If a customer buys an airplane ticket, then there is a 46 percent probability that they will buy a hotel room, and if they go on to buy a hotel room, then there is a 33 percent probability that they will rent a car.

However, it is not just for sales and marketing. It is also being used in fraud detection and healthcare; for example, if a patient undergoes treatment A, then there is a 26 percent probability that they might exhibit symptom X. Before going into the details, we should have a look at some terminology, as it will be used in the example:

- **Itemset:** This is a collection of one or more items in the dataset.
- **Support:** This is the proportion of the transactions in the data that contain an itemset of interest.
- **Confidence:** This is the conditional probability that if a person purchases or does *x*, they will purchase or do *y*; the act of doing *x* is referred to as the *antecedent* or Left-Hand Side (LHS), and *y* is the *consequence* or Right-Hand Side (RHS).

- **Lift:** This is the ratio of the support of x occurring together with y divided by the probability that x and y occur if they are independent. It is the **confidence** divided by the probability of x times the probability of y ; for example, say that we have the probability of x and y occurring together as 10 percent and the probability of x is 20 percent and y is 30 percent, then the lift would be 10 percent (20 percent times 30 percent) or 16.67 percent.

The package in R that you can use to perform a market basket analysis is **arules: Mining Association Rules and Frequent Itemsets**. The package offers two different methods of finding rules. Why would one have different methods? Quite simply, if you have massive datasets, it can become computationally expensive to examine all the possible combinations of the products. The algorithms that the package supports are **apriori** and **ECLAT**. There are other algorithms to conduct a market basket analysis, but **apriori** is used most frequently, and so, that will be our focus.

With **apriori**, the principle is that, if an itemset is frequent, then all of its subsets must also be frequent. A minimum frequency (support) is determined by the analyst prior to executing the algorithm, and once established, the algorithm will run as follows:

- Let $k=1$ (the number of items)
- Generate itemsets of a length that are equal to or greater than the specified support
- Iterate $k + (1...n)$, pruning those that are infrequent (less than the support)
- Stop the iteration when no new frequent itemsets are identified

Once you have an ordered summary of the most frequent itemsets, you can continue the analysis process by examining the confidence and lift in order to identify the associations of interest.

Business understanding

For our business case, we will focus on identifying the association rules for a grocery store. The dataset will be from the **arules** package and is called **Groceries**. This dataset consists of actual transactions over a 30-day period from a real-world grocery store and consists of 9,835 different purchases. All the items purchased are put into one of 169 categories, for example, bread, wine, meat, and so on.

Let's say that we are a start-up microbrewery trying to make a headway in this grocery outlet and want to develop an understanding of what potential customers will purchase along with beer. This knowledge may just help us in identifying the right product placement within the store or support a cross-selling campaign.

Data understanding and preparation

For this analysis, we will only need to load two packages, as well as the `Groceries` dataset:

```
> library(arules)

> library(arulesViz)

> data(Groceries)

> head(Groceries)

transactions in sparse format with
9835 transactions (rows) and
169 items (columns)

> str(Groceries)
Formal class 'transactions' [package "arules"] with 3 slots
 ..@ data :Formal class 'ngCMatrix' [package "Matrix"] with 5
   slots
 .. .. ..@ i : int [1:43367] 13 60 69 78 14 29 98 24 15 29 ...
 .. .. ..@ p : int [1:9836] 0 4 7 8 12 16 21 22 27 28 ...
 .. .. ..@ Dim : int [1:2] 169 9835
 .. .. ..@ Dimnames:List of 2
 .. .. .. ..$ : NULL
 .. .. .. ..$ : NULL
 .. .. ..@ factors : list()
 ..@ itemInfo : 'data.frame': 169 obs. of 3 variables:
 .. ..$ labels: chr [1:169] "frankfurter" "sausage" "liver loaf"
   "ham" ...
 .. ..$ level2: Factor w/ 55 levels "baby food","bags",...: 44 44
 44 44 44 44
 44 42 42 41 ...
 .. ..$ level1: Factor w/ 10 levels "canned food",...: 6 6 6 6 6 6
 6 6 6 6
 ...
 ..@ itemsetInfo: 'data.frame': 0 obs. of 0 variables
```

This dataset is structured as a sparse matrix object, known as the `transaction` class.

So, once the structure is that of the class `transaction`, our standard exploration techniques will not work, but the `arules` package offers us other techniques to explore the data. On a side note, if you have a data frame or matrix and want to convert it to the `transaction` class, you can do this with a simple syntax, using the `as()` function.



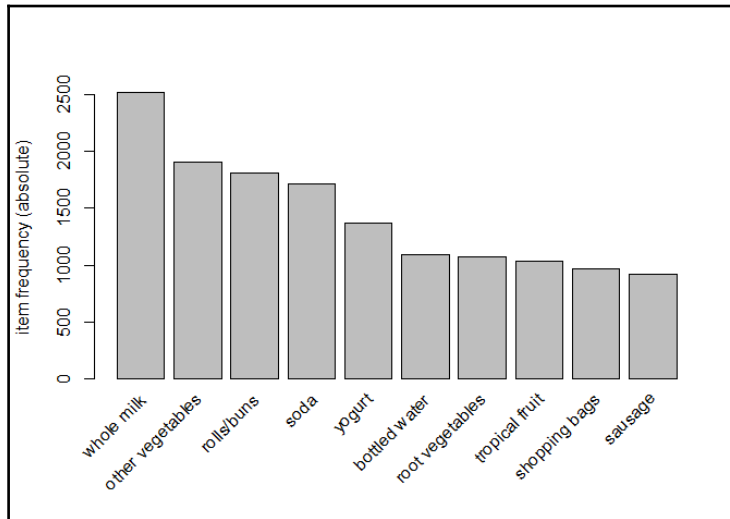
The following code is for illustrative purposes only, so do not run it:

```
> # transaction.class.name <-  
as(current.data.frame, "transactions").
```

The best way to explore this data is with an item frequency plot using the `itemFrequencyPlot()` function in the `arules` package. You will need to specify the transaction dataset, the number of items with the highest frequency to plot, and whether or not you want the relative or absolute frequency of the items. Let's first look at the absolute frequency and the top 10 items only:

```
> itemFrequencyPlot(Groceries, topN = 10, type = "absolute")
```

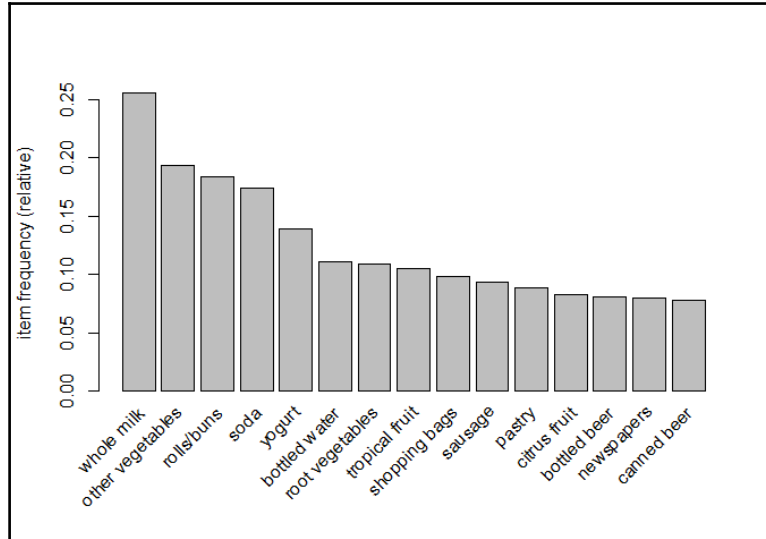
The output of the preceding command is as follows:



The top item purchased was **whole milk** with roughly **2,500** of the 9,836 transactions in the basket. For a relative distribution of the top 15 items, let's run the following code:

```
> itemFrequencyPlot(Groceries, topN = 15)
```

The following is the output of the preceding command:



Alas, here we see that beer shows up as the 13th and 15th most purchased item at this store. Just under 10 percent of the transactions had purchases of **bottled beer** and/or **canned beer**.

For the purpose of this exercise, this is all we really need to do, therefore, we can move right on to the modeling and evaluation.

Modeling and evaluation

We will start by mining the data for the overall association rules before moving on to our rules for beer specifically. Throughout the modeling process, we will use the `apriori()` function in the `arules` package. The two main things that we will need to specify in the function is the dataset and parameters. As for the parameters, you will need to apply judgment when specifying the minimum support, confidence, and the minimum and/or maximum length of basket items in an itemset. Using the item frequency plots, along with trial and error, let's set the minimum support at 1 in 1,000 transactions and minimum confidence at 90 percent. Additionally, let's establish the maximum number of items to be associated as four. The following is the code to create the object that we will call `rules`:

```
> rules <- apriori(Groceries, parameter = list(supp = 0.001, conf = 0.9, maxlen=4))
```

Calling the object shows how many rules the algorithm produced:

```
> rules
set of 67 rules
```

There are a number of ways to examine the rules. The first thing that I recommend is to set the number of displayed digits to only two, with the `options()` function in base R. Then, sort and inspect the top five rules based on the lift that they provide, as follows:

```
> options(digits = 2)

> rules <- sort(rules, by = "lift", decreasing = TRUE)

> inspect(rules[1:5])
  lhs                rhs                support confidence lift
1 {liquor, red/blush wine} => {bottled beer}      0.0019
  0.90 11.2
2 {root vegetables, butter, cream cheese }      => {yogurt}
  0.0010      0.91 6.5
3 {citrus fruit, root vegetables, soft cheese}=> {other vegetables}
  0.0010      1.00 5.2
4 {pip fruit, whipped/sour cream, brown bread}=> {other vegetables}
  0.0011      1.00 5.2
5 {butter,whipped/sour cream, soda}      => {other vegetables}
  0.0013      0.93 4.8
```

Lo and behold, the rule that provides the best overall lift is the purchase of liquor and red wine on the probability of purchasing bottled beer. I have to admit that this is pure chance and not intended on my part. As I always say, it is better to be lucky than good. Although, it is still not a very common transaction with a support of only 1.9 per 1,000.

You can also sort by the support and confidence, so let's have a look at the first 5 rules by="confidence" in descending order, as follows:

```
> rules <- sort(rules, by = "confidence", decreasing = TRUE)

> inspect(rules[1:5])
  lhs                rhs                support confidence lift
1 {citrus fruit, root vegetables, soft cheese}=> {other vegetables}
  0.0010      1 5.2
2 {pip fruit, whipped/sour cream, brown bread}=> {other vegetables}
  0.0011      1 5.2
3 {rice, sugar}      => {whole milk}      0.0012      1 3.9
4 {canned fish, hygiene articles} => {whole milk} 0.0011 1 3.9
5 {root vegetables, butter, rice} => {whole milk} 0.0010 1 3.9
```

You can see in the table that `confidence` for these transactions is 100 percent. Moving on to our specific study of beer, we can utilize a function in `arules` to develop cross tabulations--the `crossTable()` function--and then examine whatever suits our needs. The first step is to create a table with our dataset:

```
> tab <- crossTable(Groceries)
```

With `tab` created, we can now examine the joint occurrences between the items. Here, we will look at just the first three rows and columns:

```
> tab[1:3, 1:3]
      frankfurter sausage liver loaf
frankfurter      580      99       7
sausage           99     924      10
liver loaf        7      10      50
```

As you might imagine, shoppers only selected liver loaf 50 times out of the 9,835 transactions. Additionally, of the 924 times, people gravitated toward `sausage`, 10 times they felt compelled to grab `liver loaf`. (Desperate times call for desperate measures!) If you want to look at a specific example, you can either specify the row and column number or just spell that item out:

```
> table["bottled beer", "bottled beer"]
[1] 792
```

This tells us that there were 792 transactions of `bottled beer`. Let's see what the joint occurrence between `bottled beer` and `canned beer` is:

```
> table["bottled beer", "canned beer"]
[1] 26
```

I would expect this to be low as it supports my idea that people lean toward drinking beer from either a bottle or a can. I strongly prefer a bottle. It also makes a handy weapon to protect oneself from all these ruffian protesters like Occupy Wallstreet and the like.

We can now move on and derive specific rules for `bottled beer`. We will again use the `apriori()` function, but this time, we will add a syntax around `appearance`. This means that we will specify in the syntax that we want the left-hand side to be items that increase the probability of a purchase of `bottled beer`, which will be on the right-hand side. In the following code, notice that I've adjusted the `support` and `confidence` numbers. Feel free to experiment with your own settings:

```
> beer.rules <- apriori(data = Groceries, parameter = list(support
= 0.0015, confidence = 0.3), appearance = list(default = "lhs",
rhs = "bottled beer"))
```

```
> beer.rules
set of 4 rules
```

We find ourselves with only 4 association rules. We have seen one of them already; now let's bring in the other three rules in descending order by lift:

```
> beer.rules <- sort(beer.rules, decreasing = TRUE, by = "lift")

> inspect(beer.rules)
  lhs                rhs                support confidence lift
1 {liquor, red/blush wine} => {bottled beer}  0.0019  0.90 11.2
2 {liquor}              => {bottled beer}    0.0047  0.42  5.2
3 {soda, red/blush wine} => {bottled beer}    0.0016  0.36  4.4
4 {other vegetables, red/blush wine} => {bottled beer} 0.0015 0.31
  3.8
```

In all of the instances, the purchase of `bottled beer` is associated with booze, either `liquor` and/or `red wine`, which is no surprise to anyone. What is interesting is that `white wine` is not in the mix here. Let's take a closer look at this and compare the joint occurrences of `bottled beer` and types of wine:

```
> tab["bottled beer", "red/blush wine"]
[1] 48

> tab["red/blush wine", "red/blush wine"]
[1] 189

> 48/189
[1] 0.25

> tab["white wine", "white wine"]
[1] 187

> tab["bottled beer", "white wine"]
[1] 22

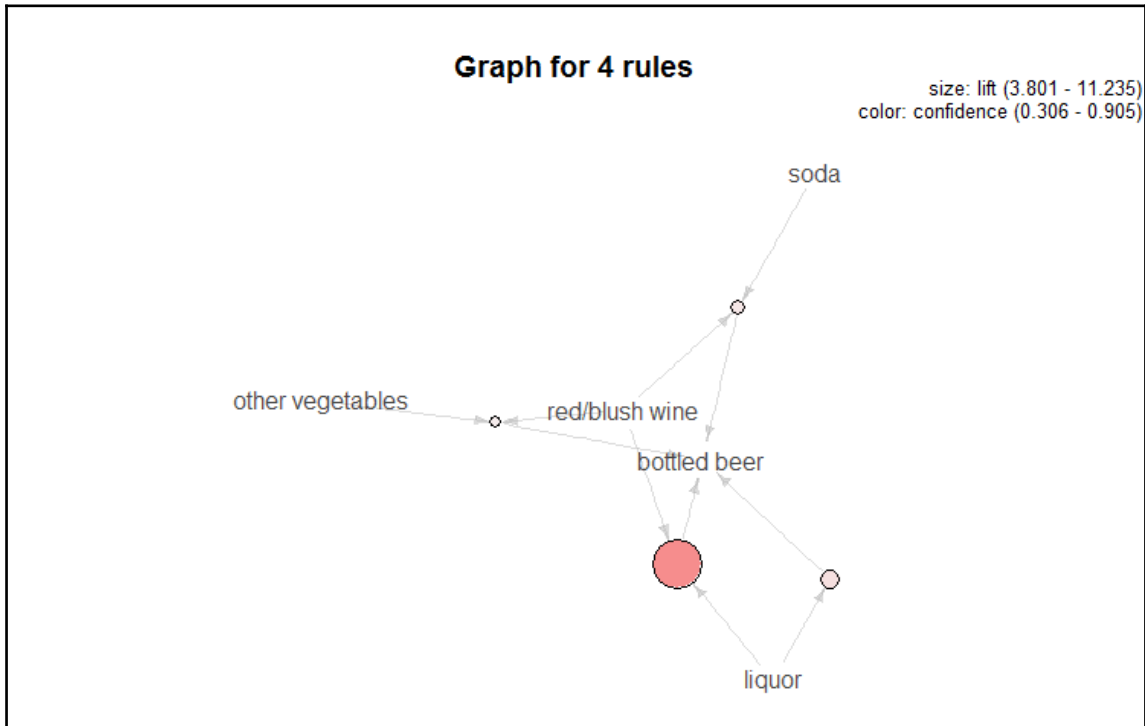
> 22/187
[1] 0.12
```

It's interesting that 25 percent of the time, when someone purchased `red wine`, they also purchased `bottled beer`; but with `white wine`, a joint purchase only happened in 12 percent of the instances. We certainly don't know why in this analysis, but this could potentially help us to determine how we should position our product in this grocery store. Another thing before we move on is to look at a plot of the rules. This is done with the `plot()` function in the `arulesViz` package.

There are many graphic options available. For this example, let's specify that we want a graph, showing lift, and the rules provided and shaded by confidence. The following syntax will provide this accordingly:

```
> plot(beer.rules, method = "graph", measure = "lift", shading =  
"confidence")
```

The following is the output of the preceding command:



This graph shows that **liquor/red wine** provides the best **lift** and the highest level of **confidence** with both the **size** of the circle and its shading.

What we've just done in this simple exercise is show how easy it is with R to conduct a market basket analysis. It doesn't take much imagination to figure out the analytical possibilities that one can include with this technique, for example, in corporate customer segmentation, longitudinal purchase history, and so on, as well as how to use it in ad displays, co-promotions, and so on. Now let's move on to a situation where customers rate items, and learn how to build and test recommendation engines.

An overview of a recommendation engine

We will now focus on situations where users have provided rankings or ratings on previously viewed or purchased items. There are two primary categories of designing recommendation systems: *collaborative filtering* and *content-based* (Ansari, Essegaier, and Kohli, 2000). The former category is what we will concentrate on, as this is the focus of the `recommenderlab` R package that we will be using.

For content-based approaches, the concept is to link user preferences with item attributes. These attributes may be things such as the genre, cast, or storyline for a movie or TV show recommendation. As such, recommendations are based entirely on what the user provides as ratings; there is no linkage to what anyone else recommends. This has the advantage over content-based approaches in that when a new item is added, it can be recommended to a user if it matches their profile, instead of relying on other users to rate it first (the so-called "first rater problem"). However, content-based methods can suffer when limited content is available, either because of the domain or when a new user enters the system. This can result in non-unique recommendations, that is, poor recommendations (Lops, Gemmis, and Semeraro, 2011).

In collaborative filtering, the recommendations are based on the many ratings provided by some or all of the individuals in the database. Essentially, it tries to capture the wisdom of the crowd.

For collaborative filtering, we will focus on the following four methods:

- **User-based collaborative filtering (UBCF)**
- **Item-based collaborative filtering (IBCF)**
- **Singular value decomposition (SVD)**
- **Principal components analysis (PCA)**

We will look at these methods briefly before moving on to the business case. It is also important to understand that `recommenderlab` was not designed to be used as a real-world implementation tool, but rather as a laboratory tool in order to research algorithms provided in the package as well as algorithms that you wish to experiment with on your own.

User-based collaborative filtering

In UBCF, the algorithm finds *missing ratings for a user by first finding a neighborhood of similar users and then aggregating the ratings of these users to form a prediction* (Hahsler, 2011). The neighborhood is determined by selecting either the KNN that is the most similar to the user we are making predictions for or by some similarity measure with a minimum threshold. The two similarity measures available in `recommenderlab` are **pearson correlation coefficient** and **cosine similarity**. I will skip the formulas for these measures as they are readily available in the package documentation.

Once the neighborhood method is decided on, the algorithm identifies the neighbors by calculating the similarity measure between the individual of interest and their neighbors on only those items that were rated by both. Through a scoring scheme, say, a simple average, the ratings are aggregated in order to make a predicted score for the individual and item of interest.

Let's look at a simple example. In the following matrix, there are six individuals with ratings on four movies, with the exception of my rating for *Mad Max*. Using $k=1$, the nearest neighbor is **Homer**, with **Bart** a close second; even though **Flanders** hated the **Avengers** as much as I did. So, using Homer's rating for **Mad Max**, which is **4**, the predicted rating for me would also be a **4**:

	Avengers	American Sniper	Les Miserable	Mad Max
Homer	3	5	3	4
Marge	5	2	5	3
Bart	5	5	1	4
Lisa	5	1	5	2
Flanders	1	1	4	1
Me	1	5	2	?

There are a number of ways to weigh the data and/or control the bias. For instance, **Flanders** is quite likely to have lower ratings than the other users, so normalizing the data where the new rating score is equal to the user rating for an item minus the average for that user for all the items is likely to improve the rating accuracy.

The weakness of UBCF is that, to calculate the similarity measure for all the possible users, the entire database must be kept in memory, which can be quite computationally expensive and time-consuming.

Item-based collaborative filtering

As you might have guessed, IBCF uses the similarity between the items and not users to make a recommendation. *The assumption behind this approach is that users will prefer items that are similar to other items they like* (Hahsler, 2011). The model is built by calculating a pairwise similarity matrix of all the items. The popular similarity measures are Pearson correlation and cosine similarity. To reduce the size of the similarity matrix, one can specify to retain only the k -most similar items. However, limiting the size of the neighborhood may significantly reduce the accuracy, leading to poorer performance versus UCBF.

Continuing with our simplified example, if we examine the following matrix, with $k=1$ the item most similar to **Mad Max** is **American Sniper**, and we can thus take that rating as the prediction for **Mad Max**, as follows:

	Avengers	American Sniper	Les Miserable	Mad Max
Homer	3	5	3	4
Marge	5	2	5	3
Bart	5	5	1	4
Lisa	5	1	5	2
Flanders	1	1	4	1
Me	1	5	2	?

Singular value decomposition and principal components analysis

It is quite common to have a dataset where the number of users and items number in the millions. Even if the rating matrix is not that large, it may be beneficial to reduce the dimensionality by creating a smaller (lower-rank) matrix that captures most of the information in the higher-dimension matrix. This may potentially allow you to capture important latent factors and their corresponding weights in the data. Such factors could lead to important insights, such as the movie genre or book topics in the rating matrix. Even if you are unable to discern meaningful factors, the techniques may filter out the noise in the data.

One issue with large datasets is that you will likely end up with a sparse matrix that has many ratings missing. One weakness of these methods is that they will not work on a matrix with missing values, which must be imputed. As with any data imputation task, there are a number of techniques that one can try and experiment with, such as using the mean, median, or code as zeroes. The default for `recommenderlab` is to use the median.

So, what is SVD? It is simply a method for matrix factorization, and can help transform a set of correlated features to a set of uncorrelated features. Say that you have a matrix called **A**. This matrix will factor into three matrices: **U**, **D**, and **V^T**. **U** is an orthogonal matrix, **D** is a non-negative, diagonal matrix, and **V^T** is a transpose of an orthogonal matrix. Now, let's look at our rating matrix and walk through an example using R.

The first thing that we will do is recreate the rating matrix (think of it as matrix **A**, as shown in the following code):

```
> ratings <- c(3, 5, 5, 5, 1, 1, 5, 2, 5, 1, 1, 5, 3, 5, 1, 5, 4
, 2, 4, 3, 4, 2, 1, 4)

> ratingMat <- matrix(ratings, nrow = 6)

> rownames(ratingMat) <- c("Homer", "Marge", "Bart", "Lisa",
"Flinders", "Me")

> colnames(ratingMat) <- c("Avengers", "American Sniper", "Les
Miserable", "Mad Max")

> ratingMat
Avengers  American Sniper  Les Miserable  Mad  Max
Homer      3              5              3      4
Marge      5              2              5      3
Bart       5              5              1      4
Lisa       5              1              5      2
Flinders   1              1              4      1
Me         1              5              2      4
```

Now, we will use the `svd()` function in base R to create the three matrices described above, which R calls `$d`, `$u`, and `$v`. You can think of the `$u` values as an individual's loadings on that factor and `$v` as a movie's loadings on that dimension. For example, `Mad Max` loads on dimension one at -0.116 (1st row, 4th column):

```
> svd <- svd(ratingMat)

> svd
$d
[1] 16.1204848  6.1300650  3.3664409  0.4683445

$u
      [,1]      [,2]      [,3]      [,4]
[1,] -0.4630576  0.2731330  0.2010738 -0.27437700
[2,] -0.4678975 -0.3986762 -0.0789907  0.53908884
[3,] -0.4697552  0.3760415 -0.6172940 -0.31895450
[4,] -0.4075589 -0.5547074 -0.1547602 -0.04159102
[5,] -0.2142482 -0.3017006  0.5619506 -0.57340176
```

```
[6,] -0.3660235  0.4757362  0.4822227  0.44927622

$V
      [,1]      [,2]      [,3]      [,4]
[1,] -0.5394070 -0.3088509 -0.77465479 -0.1164526
[2,] -0.4994752  0.6477571  0.17205756 -0.5489367
[3,] -0.4854227 -0.6242687  0.60283871 -0.1060138
[4,] -0.4732118  0.3087241  0.08301592  0.8208949
```

It is easy to explore how much variation is explained by reducing the dimensionality. Let's sum the diagonal numbers of `$d`, then look at how much of the variation we can explain with just two factors, as follows:

```
> sum(svd$d)
[1] 26.08534

> var <- sum(svd$d[1:2])

> var
[1] 22.25055

> var/sum(svd$d)
[1] 0.8529908
```

With two of the four factors, we are able to capture just over 85 percent of the total variation in the full matrix. You can see the scores that the reduced dimensions would produce. To do this, we will create a function. (Many thanks to the www.stackoverflow.com respondents who helped me put this function together.) This function will allow us to specify the number of factors that are to be included for a prediction. It calculates a rating value by multiplying the `$u` matrix times the `$v` matrix times the `$d` matrix:

```
> f1 <- function(x) {
score = 0
for(i in 1:n )
score <- score + svd$u[,i] %*% t(svd$v[,i]) * svd$d[i]
return(score)}
}
```

By specifying `n=4` and calling the function, we can recreate the original rating matrix:

```
> n = 4

> f1(svd)
      [,1] [,2] [,3] [,4]
[1,]    3    5    3    4
[2,]    5    2    5    3
[3,]    5    5    1    4
[4,]    5    1    5    2
[5,]    1    1    4    1
[6,]    1    5    2    4
```

Alternatively, we can specify `n=2` and examine the resulting matrix:

```
> n = 2

> f1(svd)
      [,1]      [,2]      [,3]      [,4]
[1,] 3.509402 4.8129937 2.578313 4.049294
[2,] 4.823408 2.1843483 5.187072 2.814816
[3,] 3.372807 5.2755495 2.236913 4.295140
[4,] 4.594143 1.0789477 5.312009 2.059241
[5,] 2.434198 0.5270894 2.831096 1.063404
[6,] 2.282058 4.8361913 1.043674 3.692505
```

So, with SVD, you can reduce the dimensionality and possibly identify the meaningful latent factors.

If you went through the prior chapter, you will see the similarities with PCA. In fact, the two are closely related and often used interchangeably as they both utilize matrix factorization. You may be asking what is the difference? In short, PCA is based on the covariance matrix, which is symmetric. This means that you start with the data, compute the covariance matrix of the centered data, diagonalize it, and create the components.

Let's apply a portion of the PCA code from the prior chapter to our data in order to see how the difference manifests itself:

```
> library(psych)

> pca <- principal(ratingMat, nfactors = 2, rotate = "none")

> pca
Principal Components Analysis
Call: principal(r = ratingMat, nfactors = 2, rotate =
"none")
Standardized loadings (pattern matrix) based upon correlation
```

```
matrix
      PC1  PC2  h2  u2
Avengers -0.09 0.98 0.98 0.022
American Sniper 0.99 -0.01 0.99 0.015
Les Miserable -0.90 0.18 0.85 0.150
Mad Max 0.92 0.29 0.93 0.071

      PC1  PC2
SS loadings 2.65 1.09
Proportion Var 0.66 0.27
Cumulative Var 0.66 0.94
Proportion Explained 0.71 0.29
Cumulative Proportion 0.71 1.00
```

You can see that PCA is easier to interpret. Notice how `American Sniper` and `Mad Max` have high loadings on the first component, while only `Avengers` has a high loading on the second component. Additionally, these two components account for 94 percent of the total variance in the data. It is noteworthy to include that, in the time between the first and second editions of this book, PCA has become unavailable.

Having applied a simplistic rating matrix to the techniques of collaborative filtering, let's move on to a more complex example using real-world data.

Business understanding and recommendations

This business case is a joke, literally. Maybe it is more appropriate to say a bunch of jokes, as we will use the `Jester5k` data from the `recommenderlab` package. This data consists of 5,000 ratings on 100 jokes sampled from the Jester Online Joke Recommender System. It was collected between April 1999 and May 2003, and all the users have rated at least 36 jokes (Goldberg, Roeder, Gupta, and Perkins, 2001). Our goal is to compare the recommendation algorithms and select the best one.

As such, I believe it is important to lead off with a statistical joke to put one in the proper frame of mind. I'm not sure of how to properly provide attribution for this one, but it is popular all over the Internet.

A statistician's wife had twins. He was delighted. He rang the minister who was also delighted. "Bring them to church on Sunday and we'll baptize them", said the minister. "No", replied the statistician. "Baptize one. We'll keep the other as a control."

Data understanding, preparation, and recommendations

The one library that we will need for this exercise is `recommenderlab`. The package was developed by the Southern Methodist University's Lyle Engineering Lab, and they have an excellent website with supporting documentation at <https://lyle.smu.edu/IDA/recommenderlab/>:

```
> library(recommenderlab)

> data(Jester5k)

> Jester5k
5000 x 100 rating matrix of class 'realRatingMatrix' with
362106 ratings.
```

The rating matrix contains 362106 total ratings. It is quite easy to get a list of a user's ratings. Let's look at user number 10. The following output is abbreviated for the first five jokes:

```
> as(Jester5k[10,], "list")
$u12843
  j1    j2    j3    j4    j5 ...
-1.99 -6.89  2.09 -4.42 -4.90 ...
```

You can also look at the mean rating for a user (user 10) and/or the mean rating for a specific joke (joke 1), as follows:

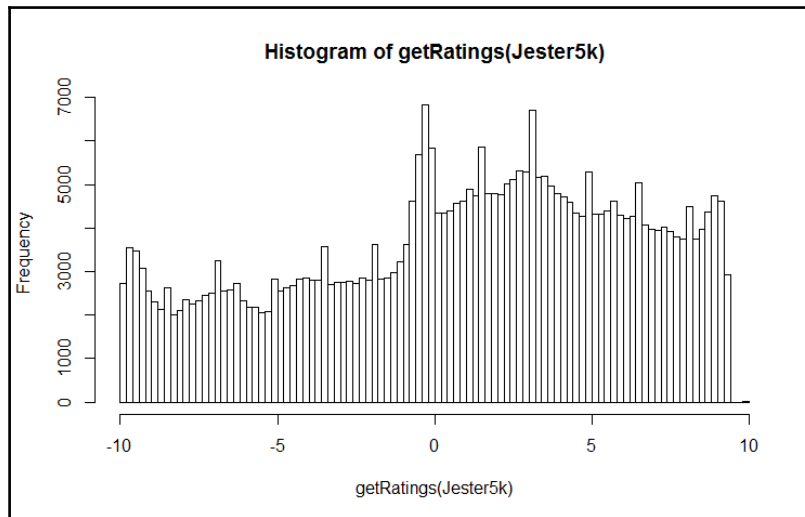
```
> rowMeans(Jester5k[10,])
u12843
-1.6

> colMeans(Jester5k[,1])
j1
0.92
```

One method to get a better understanding of the data is to plot the ratings as a histogram, both the raw data and after normalization. We will do this with the `getRating()` function from `recommenderlab`:

```
> hist(getRatings(Jester5k), breaks=100)
```

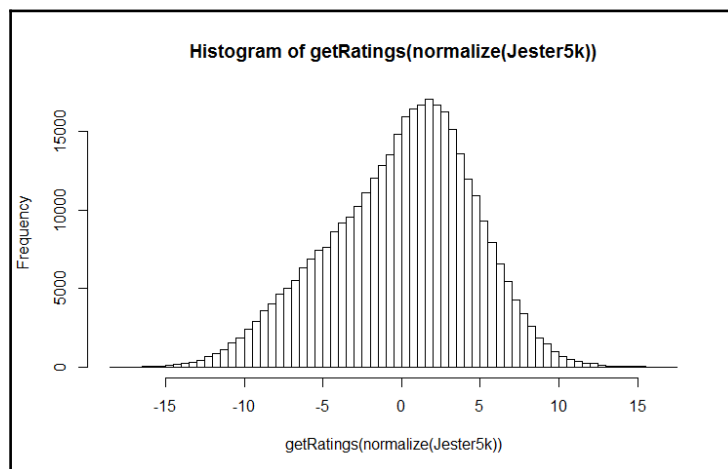
The output of the preceding command is as follows:



The `normalize()` function in the package centers the data by subtracting the mean of the ratings of the joke from that joke's rating. As the preceding distribution is slightly biased towards the positive ratings, normalizing the data can account for this, thus yielding a more normal distribution but still showing a slight skew towards the positive ratings, as follows:

```
> hist(getRatings(normalize(Jester5k)), breaks = 100)
```

The following is the output of the preceding command:



Before modeling and evaluation, it is quite easy to create the `train` and `test` datasets with the `recommenderlab` package with the `evaluationScheme()` function. Let's do an 80/20 split of the data for the `train` and `test` sets. You can also choose k-fold cross-validation and bootstrap resampling if you desire. We will also specify that for the `test` set, the algorithm will be given 15 ratings. This means that the other rating items will be used to compute the error. Additionally, we will specify what the threshold is for a good rating; in our case, greater than or equal to 5:

```
> set.seed(123)

> e <- evaluationScheme(Jester5k, method="split",
  train=0.8, given=15, goodRating=5)

> e
Evaluation scheme with 15 items given
Method: 'split' with 1 run(s).
Training set proportion: 0.800
Good ratings: >=5.000000
Data set: 5000 x 100 rating matrix of class
'realRatingMatrix' with 362106
ratings.
```

With the `train` and `test` data established, we will now begin to model and evaluate the different recommenders: user-based, item-based, popular, SVD, PCA, and random.

Modeling, evaluation, and recommendations

In order to build and test our recommendation engines, we can use the same function, `Recommender()`, merely changing the specification for each technique. In order to see what the package can do and explore the parameters available for all six techniques, you can examine the registry. Looking at the following IBCF, we can see that the default is to find 30 neighbors using the cosine method with the centered data while the missing data is not coded as a zero:

```
> recommenderRegistry$get_entries(dataType =
"realRatingMatrix")

$ALS_realRatingMatrix
Recommender method: ALS for realRatingMatrix
Description: Recommender for explicit ratings based on latent
  factors, calculated by alternating least squares algorithm.
Reference: Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, Rong
  Pan (2008).
Large-Scale Parallel Collaborative Filtering for the Netflix Prize,
```

4th Int'l
Conf. Algorithmic Aspects in Information and Management, LNCS 5034.
Parameters:
normalize lambda n_factors n_iterations min_item_nr seed
1 NULL 0.1 10 10 1 NULL

\$ALS_implicit_realRatingMatrix
Recommender method: ALS_implicit for realRatingMatrix
Description: Recommender for implicit data based on latent factors,
calculated by alternating least squares algorithm.
Reference: Yifan Hu, Yehuda Koren, Chris Volinsky (2008).
Collaborative
Filtering for Implicit Feedback Datasets, ICDM '08 Proceedings of
the 2008
Eighth IEEE International Conference on Data Mining, pages 263-272.
Parameters:
lambda alpha n_factors n_iterations min_item_nr seed
1 0.1 10 10 10 1 NULL

\$IBCF_realRatingMatrix
Recommender method: IBCF for realRatingMatrix
Description: Recommender based on item-based collaborative
filtering.
Reference: NA
Parameters:
k method normalize normalize_sim_matrix alpha na_as_zero
1 30 "Cosine" "center" FALSE 0.5 FALSE

\$POPULAR_realRatingMatrix
Recommender method: POPULAR for realRatingMatrix
Description: Recommender based on item popularity.
Reference: NA
Parameters:
normalize aggregationRatings aggregationPopularity
1 "center" new("standardGeneric" new("standardGeneric"

\$RANDOM_realRatingMatrix
Recommender method: RANDOM for realRatingMatrix
Description: Produce random recommendations (real ratings).
Reference: NA
Parameters: None

\$RERECOMMEND_realRatingMatrix
Recommender method: RERECOMMEND for realRatingMatrix
Description: Re-recommends highly rated items (real ratings).
Reference: NA
Parameters:
randomize minRating

```
1 1 NA

$SVD_realRatingMatrix
Recommender method: SVD for realRatingMatrix
Description: Recommender based on SVD approximation with column-mean
imputation.
Reference: NA
Parameters:
  k maxiter normalize
  1 10 100 "center"

$SVDF_realRatingMatrix
Recommender method: SVDF for realRatingMatrix
Description: Recommender based on Funk SVD with gradient descend.
Reference: NA
Parameters:
  k gamma lambda min_epochs max_epochs min_improvement normalize
  1 10 0.015 0.001 50 200 1e-06 "center"
verbose
1 FALSE

$UBCF_realRatingMatrix
Recommender method: UBCF for realRatingMatrix
Description: Recommender based on user-based collaborative
filtering.
Reference: NA
Parameters:
  method nn sample normalize
  1 "cosine" 25 FALSE "center"
```

Here is how you can put together the algorithms based on the `train` data. For simplicity, let's use the default algorithm settings. You can adjust the parameter settings by simply including your changes in the function as a list:

```
> ubcf <- Recommender(getData(e, "train"), "UBCF")

> ibcf <- Recommender(getData(e, "train"), "IBCF")

> svd <- Recommender(getData(e, "train"), "SVD")

> popular <- Recommender(getData(e, "train"), "POPULAR")

> pca <- Recommender(getData(e, "train"), "PCA")

> random <- Recommender(getData(e, "train"), "RANDOM")
```

Now, using the `predict()` and `getData()` functions, we will get the predicted ratings for the 15 items of the test data for each of the algorithms, as follows:

```
> user_pred <- predict(ubcf, getData(e, "known"), type = "ratings")
> item_pred <- predict(ibcf, getData(e, "known"), type = "ratings")
> svd_pred <- predict(svd, getData(e, "known"), type = "ratings")
> pop_pred <- predict(popular, getData(e, "known"), type =
  "ratings")
> rand_pred <- predict(random, getData(e, "known"), type =
  "ratings")
```

We will examine the error between the predictions and unknown portion of the test data using the `calcPredictionAccuracy()` function. The output will consist of RMSE, MSE, and MAE for all the methods. We'll examine UBCF by itself. After creating the objects for all five methods, we can build a table by creating an object with the `rbind()` function and giving names to the rows with the `rownames()` function:

```
> P1 <- calcPredictionAccuracy(user_pred, getData(e,
  "unknown"))
> P1
RMSE MSE MAE
4.5 19.9 3.5
> P2 <- calcPredictionAccuracy(item_pred, getData(e, "unknown"))
> P3 <- calcPredictionAccuracy(svd_pred, getData(e, "unknown"))
> P4 <- calcPredictionAccuracy(pop_pred, getData(e, "unknown"))
> P5 <- calcPredictionAccuracy(rand_pred, getData(e, "unknown"))
> error <- rbind(P1, P2, P3, P4, P5)
> rownames(error) <- c("UBCF", "IBCF", "SVD", "Popular", "Random")
> error
      RMSE MSE MAE
UBCF    4.5  20  3.5
IBCF    4.6  22  3.5
SVD     4.6  21  3.7
Popular 4.5  20  3.5
Random  6.3  40  4.9
```

We can see in the output that the user-based and popular algorithms slightly outperform IBCF and SVD and all outperform random predictions.

There is another way to compare methods using the `evaluate()` function. Making comparisons with `evaluate()` allows one to examine additional performance metrics as well as performance graphs. As the UBCF and Popular algorithms performed the best, we will look at them along with IBCF.

The first task in this process is to create a list of the algorithms that we want to compare, as follows:

```
> algorithms <- list(POPULAR = list(name = "POPULAR"),
  UBCF = list(name = "UBCF"), IBCF = list(name = "IBCF"))

> algorithms
$POPULAR
$POPULAR$name
[1] "POPULAR"

$UBCF
$UBCF$name
[1] "UBCF"

$IBCF
$IBCF$name
[1] "IBCF"
```

For this example, let's compare the top 5, 10, and 15 joke recommendations:

```
> evlist <- evaluate(e, algorithms, n = c(5, 10, 15))
POPULAR run
 1 [0.07sec/4.7sec]
UBCF run
 1 [0.04sec/8.9sec]
IBCF run
 1 [0.45sec/0.32sec] 3
```

Note that by executing the command, you will receive an output on how long it took to run the algorithm. We can now examine the performance using the `avg()` function:

```
> set.seed(1)

> avg(evlist)
$POPULAR
      TP    FP    FN    TN  precision  recall  TPR    FPR
 5  2.07  2.93  12.9  67.1      0.414   0.182 0.182 0.0398
10  3.92  6.08  11.1  63.9      0.393   0.331 0.331 0.0828
```

15	5.40	9.60	9.6	60.4	0.360	0.433	0.433	0.1314
----	------	------	-----	------	-------	-------	-------	--------

\$UBCF

	TP	FP	FN	TN	precision	recall	TPR	FPR
5	2.07	2.93	12.93	67.1	0.414	0.179	0.179	0.0398
10	3.88	6.12	11.11	63.9	0.389	0.326	0.326	0.0835
15	5.41	9.59	9.59	60.4	0.360	0.427	0.427	0.1312

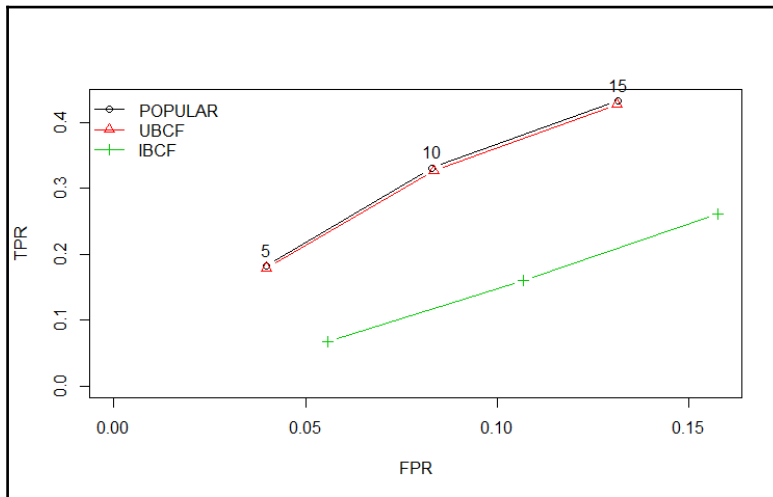
\$IBCF

	TP	FP	FN	TN	precision	recall	TPR	FPR
5	1.02	3.98	14.0	66.0	0.205	0.0674	0.0674	0.0558
10	2.35	7.65	12.6	62.4	0.235	0.1606	0.1606	0.1069
15	3.72	11.28	11.3	58.7	0.248	0.2617	0.2617	0.1575

Note that the performance metrics for POPULAR and UBCF are nearly the same. One could say that the simpler-to-implement popular-based algorithm is probably the better choice for a model selection. We can plot and compare the results as **Receiver Operating Characteristic Curves (ROC)**, comparing TPR and FPR or precision/recall, as follows:

```
> plot(evlist, legend = "topleft", annotate = TRUE)
```

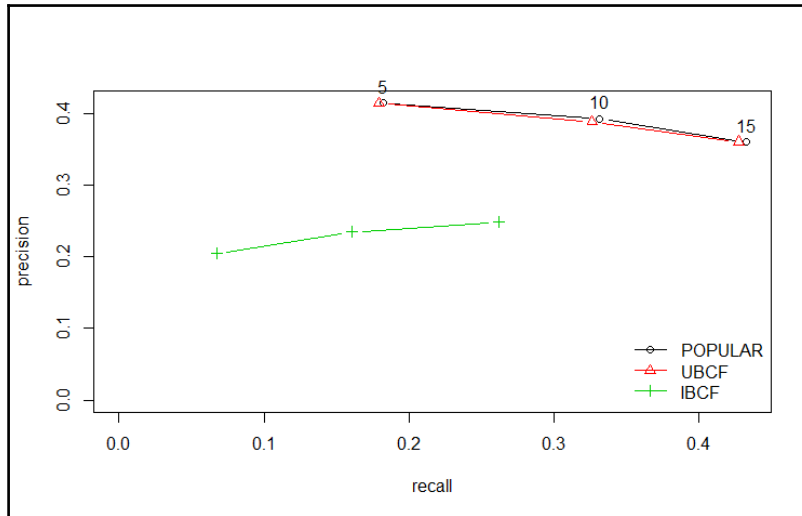
The following is the output of the preceding command:



To get the precision/recall curve plot you only need to specify "prec" in the plot function:

```
> plot(evlist, "prec", legend = "bottomright", annotate = TRUE)
```


The output of the preceding command is as follows:



You can clearly see in the plots that the popular-based and user-based algorithms are almost identical and outperform the item-based one. The `annotate=TRUE` parameter provides numbers next to the point that corresponds to the number of recommendations that we called for in our evaluation.

This was simple, but what are the actual recommendations from a model for a specific individual? This is quite easy to code as well. First, let's build a "popular" recommendation engine on the full dataset. Then, we will find the top five recommendations for the first two raters. We will use the `Recommend()` function and apply it to the whole dataset, as follows:

```
> R1 <- Recommender(Jester5k, method = "POPULAR")

> R1
Recommender of type 'POPULAR' for 'realRatingMatrix'
learned using 5000 users.
```

Now, we just need to get the top five recommendations for the first two raters and produce them as a list:

```
> recommend <- predict(R1, Jester5k[1:2], n = 5)

> as(recommend, "list")
$u2841
[1] "j89" "j72" "j76" "j88" "j83"

$u15547
[1] "j89" "j93" "j76" "j88" "j91"
```

It is also possible to see a rater's specific rating score for each of the jokes by specifying this in the `predict()` syntax and then putting it in a matrix for review. Let's do this for ten individuals (raters 300 through 309) and three jokes (71 through 73):

```
> rating <- predict(R1, Jester5k[300:309], type = "ratings")

> rating
10 x 100 rating matrix of class 'realRatingMatrix' with 322
ratings.

> as(rating, "matrix")[, 71:73]
      j71 j72 j73
u7628 -2.042 1.50 -0.2911
u8714      NA  NA      NA
u24213 -2.935  NA -1.1837
u13301  2.391 5.93  4.1419
u10959      NA  NA      NA
u23430 -0.432 3.11      NA
u11167 -1.718 1.82  0.0333
u4705  -1.199 2.34  0.5519
u24469 -1.583 1.96  0.1686
u13534 -1.545 2.00      NA
```

The numbers in the matrix indicate the predicted rating scores for the jokes that the individual rated, while the NAs indicate those that the user did not rate.

Our final effort on this data will show how to build recommendations for those situations where the ratings are binary, that is, good or bad or 1 or 0. We will need to turn the ratings into this binary format with 5 or greater as a 1 and less than 5 as 0. This is quite easy to do with `Recommenderlab` using the `binarize()` function and specifying `minRating=5`:

```
> Jester.bin <- binarize(Jester5k, minRating = 5)
```

Now, we will need to have our data reflect the number of ratings equal to one in order to match what we need the algorithm to use for the training. For argument's sake, let's go with greater than 10. The code to create the subset of the necessary data is shown in the following lines:

```
> Jester.bin <- Jester.bin[rowCounts(Jester.bin) > 10]

> Jester.bin
3054 x 100 rating matrix of class 'binaryRatingMatrix' with 84722
ratings.
```

You will need to create `evaluationScheme`. In this instance, we will go with `cross-validation`. The default k-fold in the function is 10, but we can also safely go with `k=5`, which will reduce our computation time:

```
> set.seed(456)

> e.bin <- evaluationScheme(Jester.bin, method = "cross-
validation", k = 5, given = 10)
```

For comparison purposes, the algorithms under evaluation will include `random`, `popular`, and `UBCF`:

```
> algorithms.bin <- list("random" = list(name = "RANDOM", param =
NULL), "popular" = list(name = "POPULAR", param = NULL), "UBCF" =
list(name = "UBCF"))
```

It is now time to build our model, as follows:

```
> results.bin <- evaluate(e.bin, algorithms.bin, n = c(5, 10, 15))

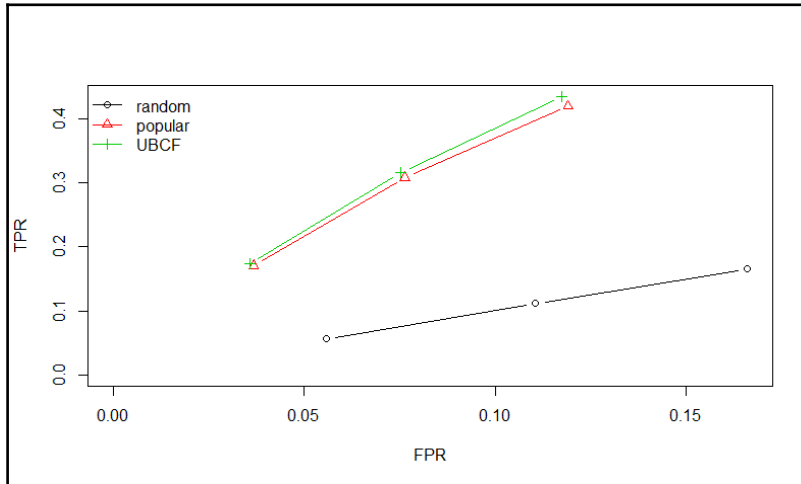
RANDOM run
1 [0sec/0.41sec]
2 [0.01sec/0.39sec]
3 [0sec/0.39sec]
4 [0sec/0.41sec]
5 [0sec/0.4sec]
POPULAR run
1 [0.01sec/3.79sec]
2 [0sec/3.81sec]
3 [0sec/3.82sec]
4 [0sec/3.92sec]
5 [0.02sec/3.78sec]
UBCF run
1 [0sec/5.94sec]
2 [0sec/5.92sec]
3 [0sec/6.05sec]
4 [0sec/5.86sec]
```

5 [0sec/6.09sec]

Forgoing the table of performance metrics, let's take a look at the plots:

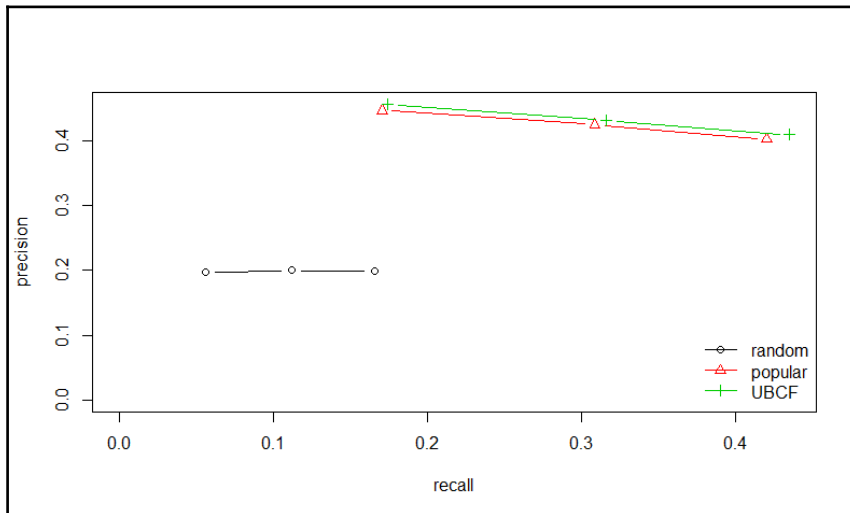
```
> plot(results.bin, legend = "topleft")
```

The output of the preceding command is as follows:



```
> plot(results.bin, "prec", legend = "bottomright")
```

The output of the preceding command is as follows:



The user-based algorithm slightly outperforms the popular-based one, but you can clearly see that they are both superior to any random recommendation. In our business case, it will come down to the judgment of the decision-making team as to which algorithm to implement.

Sequential data analysis

There are known knowns. These are things we know that we know. There are known unknowns. That is to say, there are things that we know we don't know. But there are also unknown unknowns. There are things we don't know we don't know.

- Donald Rumsfeld, Former Secretary of Defense

The very first business question I came across after the 1st edition was published revolved around product sequential analysis. The team worked on complicated Excel spreadsheets and pivot tables, along with a bunch of SAS code, to produce insights. After coming across this problem, I explored what could be done with R and was pleasantly surprised to stumble into the `TraMineR` package, specifically designed for just such a task. I believe the application of R to the problem would have greatly simplified the analysis.

The package was designed for the social sciences, but it can be used in just about every situation where you want to mine and learn how observation's states evolve over discrete periods or events (longitudinal data). A classic use would be as in the case mentioned above where you want to understand the order in which customers purchase products. This would facilitate a recommendation engine of sorts where you can create the probability of the next purchase, as I've heard it being referred to as a next logical product offer. Another example could be in healthcare, examining the order that a patient receives treatments and/or medications, or even physician prescribing habits. I've worked on such tasks, creating simple and complex Markov chains to build models and create forecasts. Indeed, `TraMineR` allows the creation of Markov chain transition matrices to support such models.

The code we will examine does the hard work of creating, counting, and plotting the various combinations of transitions over time, also incorporating covariates. That will be our focus, but keep in mind that one can also build a dissimilarity matrix for clustering. The core features covered in the practical exercise will consist of the following:

- Transition rates
- duration within each state
- Sequence frequency

Let's get started.

Sequential analysis applied

For this exercise, I've created an artificial dataset; to follow along, you can download it from GitHub:

<https://github.com/datameister66/data/blob/master/sequential.csv>

There are also datasets available with the package and tutorials are available. My intent was to create something new that mirrored situations I have encountered. I developed it completely from random (with some supervision), so it does not match any real world data. It consists of 5,000 observations, with each observation, the history of a customer and nine variables:

- `Cust_segment`--a factor variable indicating the customer's assigned segment (see Chapter 8, *Cluster Analysis*)
- Eight discrete purchase events named `Purchase1` through `Purchase8`; remember, these are events and not time-based, which is to say that a customer could have purchased all eight products at the same time, but in a specific order

Within each purchase variable are the generic names of the product, seven possible products to be exact. They are named `Product_A` through `Product_G`. What are these products? Doesn't matter! Use your imagination or apply it to your own situation. If a customer only purchased one product, then `Purchase1` would contain the name of that product and the other variables would be `NULL`.

Here we load the file as a dataframe. The structure output is abbreviated for clarity:

```
> df <- read.csv("sequential.csv")

> str(df)
'data.frame': 5000 obs. of 9 variables:
 $ Cust_Segment: Factor w/ 4 levels "Segment1","Segment2",...: 1 1 1
 1 1 1 1 1 1 1 ...
 $ Purchase1 : Factor w/ 7 levels "Product_A","Product_B",...: 1 2 7
 3 1 4 1 4 4 4 ...
```

Time for some exploration of the data, starting with a table of the customer segment counts and a count of the first product purchased:

```
> table(df$Cust_Segment)

Segment1 Segment2 Segment3 Segment4
  2900      572      554      974

> table(df$Purchase1)

Product_A Product_B Product_C Product_D Product_E Product_F
Product_G
  1451      765      659      1060      364      372
329
```

Segment1 is the largest segment, and the most purchased initial product is Product A. However, is it the most purchased product overall? This code will provide the answer:

```
> table(unlist(df[, -1]))

Product_A Product_B Product_C Product_D Product_E Product_F
Product_G
 3855      3193      3564      3122      1688      1273      915
22390
```

Yes, ProductA is the most purchased. The count of NULL values is 22,390.

Now you may be wondering if we can just build some summaries without much trouble, and that is surely the case. Here, I put the `count()` and `arrange()` functions from the `dplyr` package to good use to examine the frequency of sequences between the first and second purchase:

```
> dfCount <- count(df, Purchase1, Purchase2)

> dfCount <- arrange(dfCount, desc(n))

> dim(dfCount)
[1] 56 3

> head(dfCount)
Source: local data frame [6 x 3]
Groups: Purchase1 [4]

  Purchase1 Purchase2      n
  <fctr>    <fctr> <int>
1 Product_A Product_A   548
2 Product_D           548
```

3	Product_B	346
4	Product_C Product_C	345
5	Product_B Product_B	291
6	Product_D Product_D	281

We see that the most frequent sequences are the purchase of `ProductA` followed by another purchase of `ProductA`, along with the purchase of `ProductD` followed by no additional purchases. What is interesting is the frequency of similar product purchases.

We can now begin further examination using the `TraMineR` package. To begin, the data needs to be put into an object of class `sequence` with the `seqdef()` function. This should consist of only the sequences and not any covariates. Also, you can specify the distance of tick marks in plotting functions with `xstep = n`. In our case, we will have a tick mark for every event:

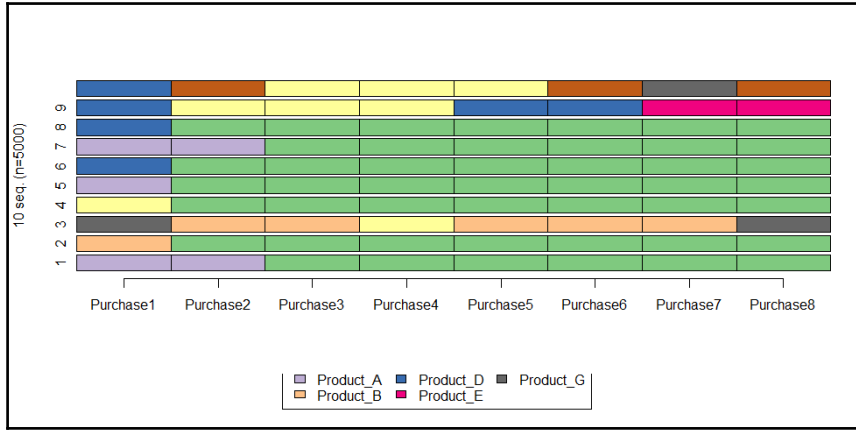
```
> seq <- seqdef(df[, -1], xstep = 1)

> head(seq)
Sequence
1 Product_A-Product_A-----
2 Product_B-----
3 Product_G-Product_B-Product_B-Product_C-Product_B-Product_B-
Product_B-
Product_G
4 Product_C-----
5 Product_A-----
6 Product_D-----
```

We can now explore the data further. Let's look at the index plot, which produces the sequences of the first 10 observations. You can use indices with the data to examine as many observations and event periods as you wish:

```
> seqiplot(seq)
```

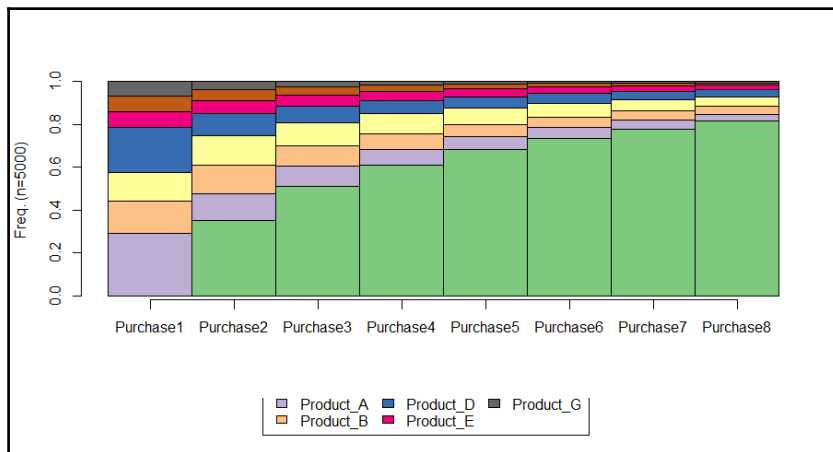

The output of the preceding command is as follows:



One can plot all observations with `seqIplot()`, but given the size of the data, it doesn't produce anything meaningful. A plot of distribution by state is more meaningful:

```
> seqdplot(seq)
```

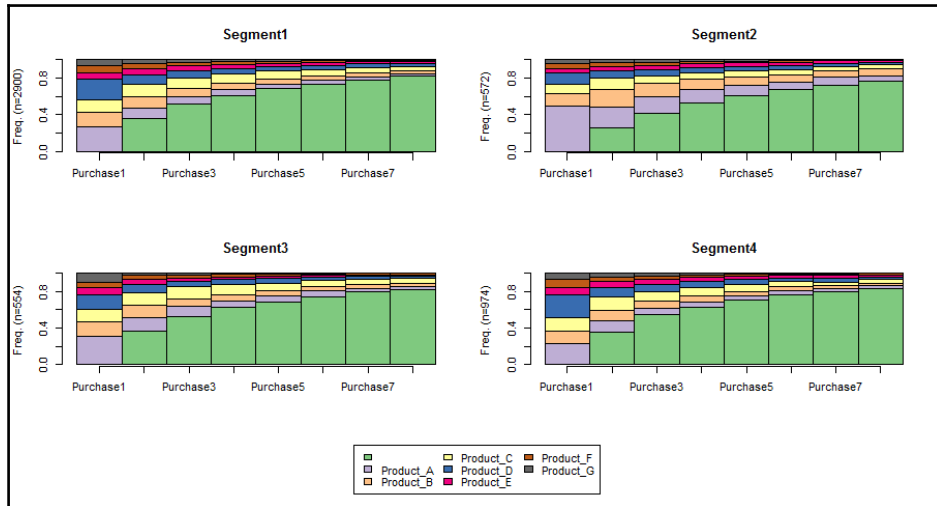
The output of the preceding command is as follows:



With this plot, it is easy to see the distribution of product purchases by state. We can also group this plot by segments and determine whether there are differences:

```
> seqdplot(seq, group = df$Cust_Segment)
```

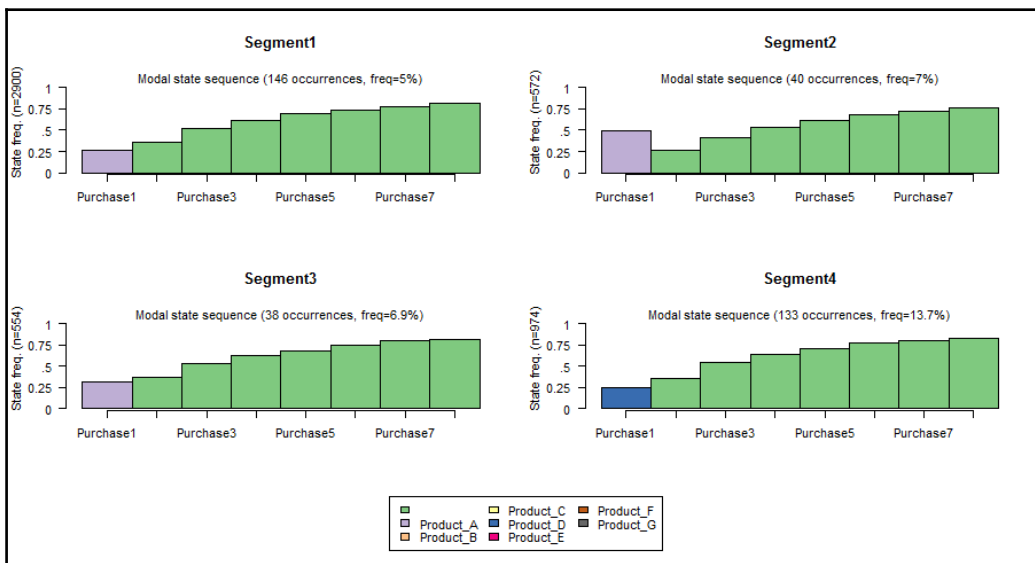
The output of the preceding command is as follows:



Here, we clearly see that `Segment2` has a higher proportion of `ProductA` purchases than the other segments. Another way to see that insight is with the modal plot:

```
> seqmsplot(seq, group = df$Cust_Segment)
```

The output of the preceding command is as follows:



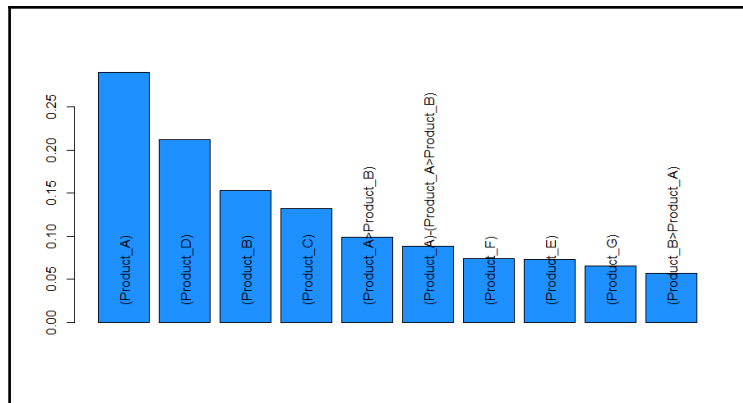
This is interesting. Around 50% of `Segment2` purchased `ProductA` first, while segment 4's most frequent initial purchase was `ProductD`. Another plot that may be of interest, but I believe not in this case, is the mean time plot. It plots the average "time" spent in each state. Since we are not time-based, it doesn't make sense, but I include for your consideration:

```
> seqmplot(seq, group = df$Cust_Segment)
```

Let's supplement our preceding code and look further at the transition of sequences. This code creates an object of sequences, then narrows that down to those sequences with an occurrence of at least 5%, then plots the top 10 sequences:

```
> seqE <- seqcreate(seq)
> subSeq <- seqefsub(seqE, pMinSupport = 0.05)
> plot(subSeq[1:10], col = "dodgerblue")
```

The output of the preceding command is as follows:



Note that the plot shows the percentage frequency of the sequences through the eight transition states. If you want to narrow that down to, say, the first two transitions, you would do that in the `seqcreate()` function using indices.

Finally, let's see how you can use the data to create a transition matrix. This matrix shows the probability of transitioning from one state to the next. In our case, it provides the probability of purchasing the next product. As I mentioned before, this can also be used in a Markov chain simulation to develop a forecast. That is outside the scope of this chapter, but if you are interested I recommend having a look at the `markovchain` package in R and its tutorial on how to implement the procedure. Two possible transition matrices are available.

One that incorporates the overall probability through all states and another that develops a transition matrix from one state to the next, that is, time-varying matrices. This code shows how to develop the former. To produce the latter, just specify "time.varying = TRUE" in the function:

```
> seqMat <- seqtrate(seq)
[>] computing transition rates for states

/Product_A/Product_B/Product_C/Product_D/
  Product_E/Product_F/Product_G ...

> options(digits = 2) # make output easier to read

> seqMat[2:4, 1:3]
      [-> ] [-> Product_A] [-> Product_B]
[Product_A ->]  0.19          0.417          0.166
[Product_B ->]  0.26          0.113          0.475
[Product_C ->]  0.19          0.058          0.041
```

The output shows rows 2 through 4 and columns 1 through 3. The matrix shows us that the probability of having Product A and the next purchase being ProductA is almost 42%, while it is 19% to not purchase another product, and 17% to purchase ProductB. The final output we will examine is the probability of not purchasing another product for each prior purchase:

```
> seqMat[, 1] [ ->] [Product_A ->] [Product_B ->] [Product_C ->]
  [Product_D ->]
1.00          0.19          0.26          0.19          0.33
[Product_E ->] [Product_F ->] [Product_G ->]
      0.18          0.25          0.41
```

Of course, the matrix shows that the probability of not purchasing another product after not purchasing is 100%. Also notice that the probability of not purchasing after acquiring Product D is 33%. Implications for Segment4? Perhaps.

What is fascinating is that this analysis was done with only a few lines of code and didn't require the use of Excel or some expensive visualization software. Have longitudinal data? Give sequential analysis a try!

Summary

In this chapter, the goal was to provide an introduction to how to use R in order to build and test association rule mining (market basket analysis) and recommendation engines. Market basket analysis is trying to understand what items are purchased together. With recommendation engines, the goal is to provide a customer with other items that they will enjoy based on how they have rated previously viewed or purchased items. It is important to understand the R package that we used (`recommenderlab`) for recommendation is not designed for implementation, but to develop and test algorithms. The other thing examined here was longitudinal data and mining it to learn valuable insights, in our case, the order in which customers purchased our products. Such an analysis has numerous applications, from marketing campaigns to healthcare.

We are now going to shift gears back to supervised learning. In the next chapter, we are going to cover some of the most exciting and important methods in practical machine learning, that is multi-class classification and creating ensemble models, something that is very easy to do in R with recent package releases.

11

Creating Ensembles and Multiclass Classification

"This is how you win ML competitions: you take other people's work and ensemble them together."

- Vitaly Kuznetsov, NIPS2014

You may have already realized that we have discussed ensemble learning. It is defined by www.scholarpedia.org as "the process by which multiple models, such as classifiers or experts, are strategically generated and combined to solve a particular computational intelligence problem". In random forest and gradient boosting, we combined the "votes" of hundreds or thousands of trees to make a prediction. Thus, by definition, those models are ensembles. This methodology can be extended to any learner to create ensembles, which some refer to as meta-ensembles or meta-learners. We will look at one of these methods referred to as "stacking". In this methodology, we will produce a number of classifiers and use their predicted class probabilities as input features to another classifier. This method *can* result in improved predictive accuracy. In the previous chapters, we focused on classification problems focused on binary outcomes. We will now look at methods to predict those situations where the data consists of more than two outcomes, a very common situation in real-world data sets. I have to confess that the application of these methods in R is some of the most interesting and enjoyable applications I have come across.

Ensembles

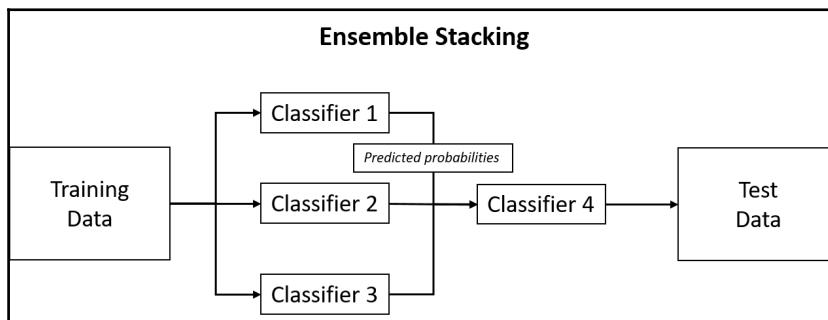
The quote at the beginning of this chapter mentions using ensembles to win machine learning competitions. However, they do have practical applications. I've provided a definition of what ensemble modeling is, but why does it work? To demonstrate this, I've co-opted an example, from the following blog, which goes into depth at a number of ensemble methods:

<http://mlwave.com/kaggle-ensembling-guide/>.

As I write this chapter, we are only a couple of days away from Super Bowl 51, the Atlanta Falcons versus the New England Patriots. Let's say we want to review our probability of winning a friendly wager where we want to take the Patriots minus the points (3 points as of this writing). Assume that we have been following three expert prognosticators that all have the same probability of predicting that the Patriots will cover the spread (60%). Now, if we favor any one of the so-called experts, it is clear we have a 60% chance to win. However, let's see what creating an ensemble of their predictions can do to increase our chances of profiting and humiliating friends and family.

Start by calculating the probability of each possible outcome for the experts picking New England. If all three pick New England, we have $0.6 \times 0.6 \times 0.6$, or a 21.6% chance, that all three are correct. If any two of the three pick New England then we have $(0.6 \times 0.6 \times 0.3) \times 3$ for a total of 43.2%. By using majority voting, if at least two of the three pick New England, then our probability of winning becomes almost 65%.

This is a rather simplistic example but representative nonetheless. In machine learning, it can manifest itself by incorporating the predictions from several average or even weak learners to improve overall accuracy. The diagram that follows shows how this can be accomplished:



In this graphic, we build three different classifiers and use their predicted probabilities as inputs to a fourth and different classifier in order to make predictions on the test data. Let's see how to apply this with R.

Business and data understanding

We are going to visit our old nemesis the Pima Diabetes data once again. It has proved to be quite a challenge with most classifiers producing accuracy rates in the mid-70s. We've looked at this data in [Chapter 5, More Classification Techniques - K-Nearest Neighbors and Support Vector Machines](#) and [Chapter 6, Classification and Regression Trees](#) so we can skip over the details. There are a number of R packages to build ensembles, and it is not that difficult to build your own code. In this iteration, we are going to attack the problem with the `caret` and `caretEnsemble` packages. Let's get the packages loaded and the data prepared, including creating the train and test sets using the `createDataPartition()` function from `caret`:

```
> library(MASS)

> library(caretEnsemble)

> library(caTools)

> pima <- rbind(Pima.tr, Pima.te)

> set.seed(502)

> split <- createDataPartition(y = pima$type, p = 0.75, list = F)

> train <- pima[split, ]

> test <- pima[-split, ]
```


Modeling evaluation and selection

As we've done in prior chapters, the first recommended task when utilizing caret functions is to build the object that specifies how model training is going to happen. This is done with the `trainControl()` function. We are going to create a five-fold cross-validation and save the final predictions (the probabilities). It is recommended that you also index the resamples so that each base model trains on the same folds. Also, notice in the function that I specified upsampling. Why? Well, notice that the ratio of "Yes" versus "No" is 2 to 1:

```
> table(train$type)
```

```
   No  Yes  
267 133
```

This ratio is not necessarily imbalanced, but I want to demonstrate something here. In many data sets, the outcome of interest is a rare event. As such, you can end up with a classifier that is highly accurate but does a horrible job at predicting the outcome of interest, which is to say it doesn't predict any true positives. To balance the response, you can upsample the minority class, downsample the majority class, or create "synthetic data". We will focus on synthetic data in the next exercise, but here, let's try upsampling. In upsampling, for each cross-validation fold, the minority class is randomly sampled with replacement to match the number of observations in the majority class. Here is our function:

```
> control <- trainControl(method = "cv",  
  number = 5,  
  savePredictions = "final",  
  classProbs = T,  
  index=createResample(train$type, 5),  
  sampling = "up",  
  summaryFunction = twoClassSummary)
```

We can now train our models using the `caretList()` function. You can use any model in the function that is supported by the caret package. A list of models is available with their corresponding hyperparameters here:

<https://rdrr.io/cran/caret/man/models.html>

In this example, we will train three models:

- Classification tree - "rpart"
- Multivariate Adaptive Regression Splines - "earth"
- K-Nearest Neighbors - "knn"

Let's put this all together:

```
> set.seed(2)

> models <- caretList(
  type ~ ., data = train,
  trControl = control,
  metric = "ROC",
  methodList = c("rpart", "earth", "knn")
)
```

Not only are the models built, but the parameters for each model are tuned according to caret's rules. You can create your own tune grids for each model by incorporating the `caretModelSpec()` function, but for demonstration purposes, we will let the function do that for us. You can examine the results by calling the model object. This is the abbreviated output:

```
> models
...
Resampling results across tuning parameters:
```

	cp	ROC	Sens	Spec
	0.03007519	0.7882347	0.8190343	0.6781714
	0.04010025	0.7814718	0.7935024	0.6888857
	0.36090226	0.7360166	0.8646440	0.6073893

A trick to effective ensembles is that the base models are not highly correlated. This is a subjective statement, and there is no hard rule of correlated predictions. One should experiment with the results and substitute a model if deemed necessary. Let's look at our results:

```
> modelCor(resamples(models))
```

	rpart	earth	knn
rpart	1.0000000	0.9589931	0.7191618
earth	0.9589931	1.0000000	0.8834022
knn	0.7191618	0.8834022	1.0000000

The classification tree and earth models are highly correlated. This may be an issue, but let's progress by creating our the fourth classifier, the stacking model, and examining the results. To do this, we will capture the predicted probabilities for "Yes" on the test set in a dataframe:

```
> model_preds <- lapply(models, predict, newdata=test, type="prob")  
  
> model_preds <- lapply(model_preds, function(x) x[, "Yes"])  
  
> model_preds <- data.frame(model_preds)
```

We now stack these models for a final prediction with `caretStack()`. This will be a simple logistic regression based on five bootstrapped samples:

```
> stack <- caretStack(models, method = "glm",  
  metric = "ROC",  
  trControl = trainControl(  
    method = "boot",  
    number = 5,  
    savePredictions = "final",  
    classProbs = TRUE,  
    summaryFunction = twoClassSummary  
  ))
```

You can examine the final model as such:

```
> summary(stack)
```

Call:
NULL

Deviance Residuals:

	Min	1Q	Median	3Q	Max
	-2.1029	-0.6268	-0.3584	0.5926	2.3714

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	2.2212	0.2120	10.476	< 2e-16 ***
rpart	-0.8529	0.3947	-2.161	0.03071 *
earth	-3.0984	0.4250	-7.290	3.1e-13 ***
knn	-1.2626	0.3524	-3.583	0.00034 ***

Even though `rpart` and `earth` were highly correlated, their coefficients are both significant and we can probably keep both in the analysis. We can now compare the individual model results with the `ensembled` learner:

```
> prob <- 1-predict(stack, newdata = test, type = "prob")  
  
> model_preds$ensemble <- prob  
  
> colAUC(model_preds, test$type)  
           rpart      earth      knn  ensemble  
No vs. Yes 0.7413481 0.7892562 0.7652376 0.8001033
```

What we see with the `colAUC()` function is the individual model AUCs and the AUC of the stacked/ensemble. The ensemble has led to a slight improvement over only using MARS from the `earth` package. So in this example, we see how creating an ensemble via model stacking can indeed increase predictive power. Can you build a better ensemble given this data? What other sampling or classifiers would you try? With that, let's move on to multi-class problems.

Multiclass classification

There are a number of approaches to learning in multiclass problems. Techniques such as random forest and discriminant analysis will deal with multiclass while some techniques and/or packages will not, for example, generalized linear models, `glm()`, in base R. As of this writing, the `caretEnsemble` package, unfortunately, will not work with multiclass. However, the Machine Learning in R (`mlr`) package does support multiple classes and also ensemble methods. If you are familiar with `sci-kit Learn` for Python, one could say that `mlr` endeavors to provide the same functionality for R. The `mlr` and the `caret`-based packages are quickly turning into my favorites for almost any business problem. I intend to demonstrate how powerful the package is on a multiclass problem, then conclude by showing how to do an ensemble on the `Pima` data.

For the multiclass problem, we will look at how to tune a random forest and then examine how to take a GLM and turn it into a multiclass learner using the "one versus rest" technique. This is where we build a binary probability prediction for each class versus all the others, then ensemble them together to predict an observation's final class. The technique allows you to extend any classifier method to multiclass problems, and it can often outperform multiclass learners.

One quick note: don't confuse the terminology of multiclass and multilabel. In the former, an observation can be assigned to one and only one class, while in the latter, it can be assigned to multiple classes. An example of this is text that could be labeled both politics and humor. We will not cover multilabel problems in this chapter.

Business and data understanding

We are once again going to visit our wine data set that we used in [Chapter 8](#), *Cluster Analysis*. If you recall, it consists of 13 numeric features and a response of three possible classes of wine. Our task is to predict those classes. I will include one interesting twist and that is to artificially increase the number of observations. The reasons are twofold. First, I want to fully demonstrate the resampling capabilities of the `mlr` package, and second, I wish to cover a synthetic sampling technique. We utilized upsampling in the prior section, so synthetic is in order.

Our first task is to load the package libraries and bring the data:

```
> library(mlr)
> library(ggplot2)
> library(HDclassif)
> library(DMwR)
> library(reshape2)
> library(corrplot)
> data(wine)
> table(wine$class)

 1  2  3
59 71 48
```

We have 178 observations, plus the response labels are numeric (1, 2 and 3). Let's more than double the size of our data. The algorithm used in this example is **Synthetic Minority Over-Sampling Technique (SMOTE)**. In the prior example, we used upsampling where the minority class was sampled WITH REPLACEMENT until the class size matched the majority. With `SMOTE`, take a random sample of the minority class and compute/identify the k-nearest neighbors for each observation and randomly generate data based on those neighbors. The default nearest neighbors in the `SMOTE()` function from the `DMwR` package is 5 ($k = 5$). The other thing you need to consider is the percentage of minority oversampling. For instance, if we want to create a minority class double its current size, we would specify `"percent.over = 100"` in the function. The number of new samples for each case added to the current minority class is `percent over/100`, or one new sample for each observation. There is another parameter for `percent over`, and that controls the number of majority classes randomly selected for the new dataset.

Here is the application of the technique, first starting by structuring the classes to a factor, otherwise the function will not work:

```
> wine$class <- as.factor(wine$class)

> set.seed(11)

> df <- SMOTE(class ~ ., wine, perc.over = 300, perc.under = 300)

> table(df$class)

  1    2    3 
195 237 192
```

Voila! We have created a dataset of 624 observations. Our next endeavor will involve a visualization of the number of features by class. I am a big fan of boxplots, so let's create boxplots for the first four inputs by class. They have different scales, so putting them into a dataframe with mean 0 and standard deviation of 1 will aid the comparison:

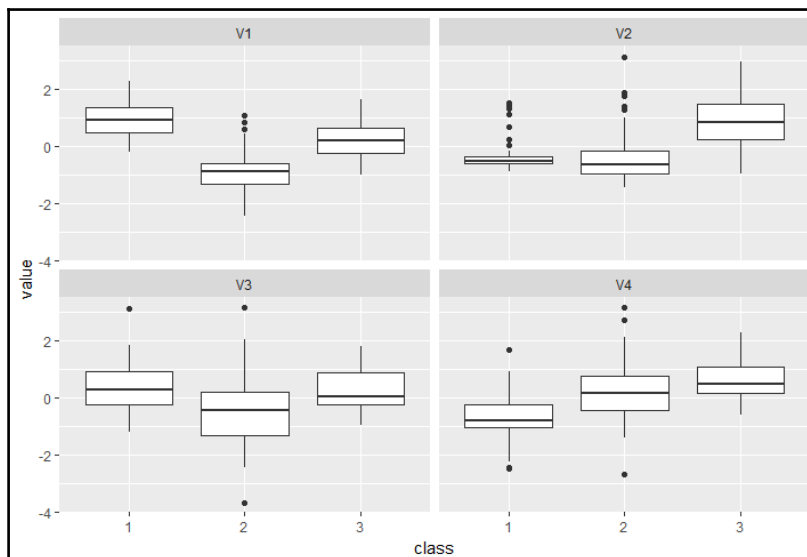
```
> wine.scale <- data.frame(scale(wine[, 2:5]))

> wine.scale$class <- wine$class

> wine.melt <- melt(wine.scale, id.var="class")

> ggplot(data = wine.melt, aes(x = class, y = value)) +
  geom_boxplot() +
  facet_wrap(~ variable, ncol = 2)
```

The output of the preceding command is as follows:



Recall from Chapter 3, *Logistic Regression and Discriminant Analysis* that a dot on the boxplot is considered an outlier. So, what should we do with them? There are a number of things to do:

- Nothing--doing nothing is always an option
- Delete the outlying observations
- Truncate the observations either within the current feature or create a separate feature of truncated values
- Create an indicator variable per feature that captures whether an observation is an outlier

I've always found outliers interesting and usually look at them closely to determine why they occur and what to do with them. We don't have that kind of time here, so let me propose a simple solution and code around truncating the outliers. Let's create a function to identify each outlier and reassign a high value (> 99 th percentile) to the 75th percentile and a low value (< 1 percentile) to the 25th percentile. You could do median or whatever, but I've found this to work well.



You could put these code excerpts into the same function, but I've done in this fashion for simplification and understanding.

These are our outlier functions:

```
> outHigh <- function(x) {  
  x[x > quantile(x, 0.99)] <- quantile(x, 0.75)  
  x  
}  
  
> outLow <- function(x) {  
  x[x < quantile(x, 0.01)] <- quantile(x, 0.25)  
  x  
}
```

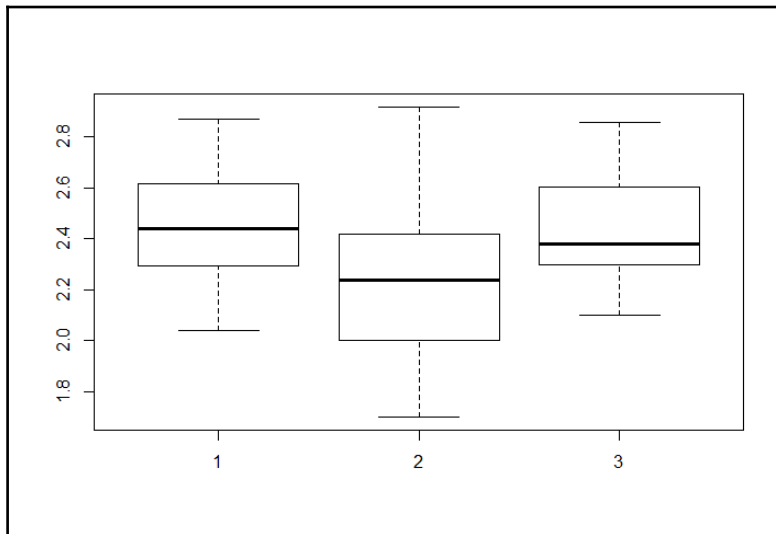
Now we execute the function on the original data and create a new dataframe:

```
> wine.trunc <- data.frame(lapply(wine[, -1], outHigh))  
  
> wine.trunc <- data.frame(lapply(wine.trunc, outLow))  
  
> wine.trunc$class <- wine$class
```

A simple comparison of a truncated feature versus the original is in order. Let's try that with V3:

```
> boxplot(wine.trunc$V3 ~ wine.trunc$class)
```

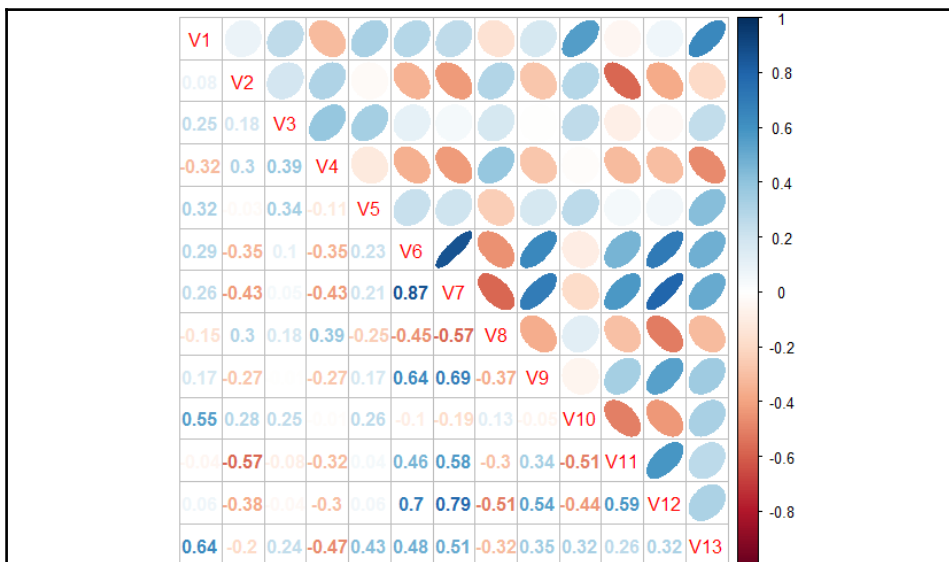
The output of the preceding command is as follows:



So that worked out well. Now it's time to look at correlations:

```
> c <- cor(wine.trunc[, -14])
> corrplot.mixed(c, upper = "ellipse")
```

The output of the preceding command is as follows:



We see that **V6** and **V7** are the highest correlated features, and we see a number above 0.5. In general, this is not a problem with non-linear based learning methods, but we will account for this in our GLM by incorporating an L2 penalty (ridge regression).

Model evaluation and selection

We will begin by creating our training and testing sets, then create a random forest classifier as our base model. After evaluating its performance, we will move on and try the one-versus-rest classification method and see how it performs. We split our data 70/30. Also, one of the unique things about the `mlr` package is its requirement to put your training data into a "task" structure, specifically a classification task. Optionally, you can place your test set in a task as well.

A full list of models is available [here](https://mlr-org.github.io/mlr-tutorial/release/html/integrated_learners/index.html), plus you can also utilize your own:

https://mlr-org.github.io/mlr-tutorial/release/html/integrated_learners/index.html

```
> library(caret) #if not already loaded

> set.seed(502)

> split <- createDataPartition(y = df$class, p = 0.7, list = F)

> train <- df[split, ]

> test <- df[-split, ]

> wine.task <- makeClassifTask(id = "wine", data = train, target =
  "class")
```

Random forest

With our training data task created, you have a number of functions to explore it. Here is the abbreviated output that looks at its structure:

```
> str(getTaskData(wine.task))
'data.frame': 438 obs. of 14 variables:
 $ class: Factor w/ 3 levels "1","2","3": 1 2 1 2 2 1 2 1 1 2 ...
 $ V1 : num 13.6 11.8 14.4 11.8 13.1 ...
```

There are many ways to use `mlr` in your analysis, but I recommend creating your resample object. Here we create a resampling object to help us in tuning the number of trees for our random forest, consisting of three subsamples:

```
> rdesc <- makeResampleDesc("Subsample", iters = 3)
```

The next object establishes the grid of trees for tuning with the minimum number of trees, 750, and the maximum of 2000. You can also establish a number of multiple parameters like we did with the `caret` package. Your options can be explored using calling help for the function with `makeParamSet`:

```
> param <- makeParamSet(
  makeDiscreteParam("ntree", values = c(750, 1000, 1250, 1500,
    1750, 2000))
)
```

Next, create a control object, establishing a numeric grid:

```
> ctrl <- makeTuneControlGrid()
```

Now, go ahead and tune the hyperparameter for the optimal number of trees. Then, call up both the optimal number of trees and the associated out-of-sample error:

```
> tuning <- tuneParams("classif.randomForest", task = wine.task,
  resampling = rdesc, par.set = param,
  control = ctrl)

> tuning$x
$ntree
[1] 1250

> tuning$y
mmce.test.mean
0.01141553
```

The optimal number of trees is 1,250 with a mean misclassification error of 0.01 percent, almost perfect classification. It is now a simple matter of setting this parameter for training as a wrapper around the `makeLearner()` function. Notice that I set the predict type to probability as the default is the predicted class:

```
> rf <- setHyperPars(makeLearner("classif.randomForest",
  predict.type = "prob"), par.vals = tuning$x)
```

Now we train the model:

```
> fitRF <- train(rf, wine.task)
```

You can see the confusion matrix on the train data:

```
> fitRF$learner.model
      OOB estimate of error rate: 0%
Confusion matrix:
   1   2   3 class.error
1 72   0   0           0
2  0  97   0           0
3  0   0 101           0
```

Then, evaluate its performance on the test set, both error and accuracy (1 - error). With no test task, you specify `newdata = test`, otherwise if you did create a test task, just use `test.task`:

```
> predRF <- predict(fitRF, newdata = test)

> getConfMatrix(predRF)
      predicted
true   1  2  3 -SUM-
  1    58  0  0     0
  2     0 71  0     0
  3     0  0 57     0
-SUM-  0  0  0     0

> performance(predRF, measures = list(mmce, acc))
mmce acc
  0    1
```

Well, that was just too easy as we are able to predict each class with no error.

Ridge regression

For demonstration purposes, let's still try our ridge regression on a one-versus-rest method. To do this, create a `MulticlassWrapper` for a binary classification method. The `classif.penalized.ridge` method is from the `penalized` package, so make sure you have it installed:

```
> ovr <- makeMulticlassWrapper("classif.penalized.ridge",
  mcw.method = "onevsrest")
```

Now let's go ahead and create a wrapper for our classifier that creates a bagging resample of 10 iterations (it is the default) with replacement, sampling 70% of the observations and all of the input features:

```
> bag.ovr = makeBaggingWrapper(ovr, bw.iters = 10, #default of 10
  bw.replace = TRUE, #default
  bw.size = 0.7,
  bw.feats = 1)
```

This can now be used to train our algorithm. Notice in the code I put `mlr::` before `train()`. The reason is that `caret` also has a `train()` function, so we are specifying we want `mlr` train function and not `caret`'s. Sometimes, if this is not done when both packages are loaded, you will end up with an egregious error:

```
> set.seed(317)
> fitOVR <- mlr::train(bag.ovr, wine.task)
> predOVR <- predict(fitOVR, newdata = test)
```

Let's see how it did:

```
> head(data.frame(predOVR))
  truth response
60    2         2
78    2         2
79    2         2
49    1         1
19    1         1
69    2         2

> getConfMatrix(predOVR)
      predicted
true   1  2  3 -SUM-
  1    58  0  0     0
  2     0 71  0     0
  3     0  0 57     0
-SUM-  0  0  0     0
```

Again, it is just too easy. However, don't focus on the accuracy as much as the methodology of creating your classifier, tuning any parameters, and implementing a resampling strategy.

MLR's ensemble

Here is something we haven't found too easy: the `Pima` diabetes classification. Like `caret`, you can build ensemble models, so let's give that a try. I will also show how to incorporate `SMOTE` into the learning process instead of creating a separate dataset.

First, make sure you run the code from the beginning of this chapter to create the train and test sets. I'll pause here and let you take care of that.

Great, now let's create the training task as before:

```
> pima.task <- makeClassifTask(id = "pima", data = train, target =
  "type")
```

The `smote()` function here is a little different from what we did before. You just have to specify the rate of minority oversample and the k-nearest neighbors. We will double our minority class (Yes) based on the three nearest neighbors:

```
> pima.smote <- smote(pima.task, rate = 2, nn = 3)

> str(getTaskData(pima.smote))
'data.frame': 533 obs. of 8 variables:
```

We now have 533 observations instead of the 400 originally in train. To accomplish our ensemble stacking, we will create three base models (random forest, quadratic discriminant analysis, and L1 penalized GLM). This code puts them together as the base models, learners if you will, and ensures we have probabilities created for use as input features:

```
> base <- c("classif.randomForest", "classif.qda", "classif.glmnet")

> learns <- lapply(base, makeLearner)

> learns <- lapply(learns, setPredictType, "prob")
```

The stacking model will simply be a GLM, with coefficients tuned by cross-validation. The package default is five folds:

```
> sl <- makeStackedLearner(base.learners = learns,
  super.learner = "classif.logreg",
  predict.type = "prob",
  method = "stack.cv")
```

We can now train the base and stacked models. You can choose to incorporate resampling and tuning wrappers as you see fit, just like we did in the prior sections. In this case, we will just stick with the defaults. Training and prediction on the test set works the same way also:

```
> slFit <- mlr::train(sl, pima.smote)

> predFit <- predict(slFit, newdata = test)

> getConfMatrix(predFit)
      predicted
true      No Yes -SUM-
No         70  18   18
Yes        15  29   15
-SUM-      15  18   33

> performance(predFit, measures = list(mmce, acc, auc))
      mmce      acc      auc
0.25    0.75    0.7874483
```

All that effort and we just achieved 75% accuracy and a slightly inferior AUC to the ensemble built using `caretEnsemble`, granted we used different base learners. So, that begs the question as before, can you improve on these results? Please let me know your results.

Summary

In this chapter, we looked at the very important machine learning methods of creating an ensemble model by stacking and then multiclass classification. In stacking, we used base models (learners) to create predicted probabilities that were used on input features to another model (super learner) to make our final predictions. Indeed, the stacked method showed slight improvement over the individual base models. As for multiclass methods, we worked on using a multiclass classifier as well as taking a binary classification method and applying it to a multiclass problem using the one-versus-all technique. As a side task, we also incorporated two sampling techniques (upsampling and Synthetic Minority Oversampling Technique) to balance the classes. Also significant was the utilization of two very powerful R packages, `caretEnsemble` and `mlr`. These methods and packages are powerful additions to an R machine learning practitioner.

Up next, we are going to delve into the world of time series and causality. In my opinion, time series analysis is one of the most misunderstood and neglected areas of machine learning. The next chapter should get you on your way to help our profession close that gap.

12

Time Series and Causality

"An economist is an expert who will know tomorrow why the things he predicted yesterday didn't happen today."

- Laurence J. Peter

A univariate time series is where the measurements are collected over a standard measure of time, which could be by the minute, hour, day, week, or month. What makes the time series problematic over the other data is that the order of the observations probably matters. This dependency of order can cause the standard analysis methods to produce an unnecessarily high bias or variance.

It seems that there is a paucity of literature on machine learning and time series data. This is unfortunate as so much of real-world data involves a time component. Furthermore, time series analysis can be quite complicated and tricky. I would say that if you haven't seen a time series analysis done incorrectly, you haven't been looking close enough.

Another aspect involving time series that is often neglected is causality. Yes, we don't want to confuse correlation with causation, but in time series analysis, one can apply the technique of Granger causality in order to determine if causality, statistically speaking, exists.

In this chapter, we will apply time series/econometric techniques to identify univariate forecast models, vector autoregression models, and finally, Granger causality. After completing the chapter, you may not be a complete master of the time series analysis, but you will know enough to perform an effective analysis and understand the fundamental issues to consider when building time series models and creating predictive models (forecasts).

Univariate time series analysis

We will focus on two methods to analyze and forecast a single time series: **exponential smoothing** and **autoregressive integrated moving average (ARIMA)** models. We will start by looking at exponential smoothing models.

Like moving average models, exponential smoothing models use weights for past observations. But unlike moving average models, the more recent the observation the more weight it is given relative to the later ones. There are three possible smoothing parameters to estimate: the overall smoothing parameter, a trend parameter, and smoothing parameter. If no trend or seasonality is present, then these parameters become null.

The smoothing parameter produces a forecast with the following equation:

$$Y_{t+1} = \alpha(Y_t) + (1 - \alpha)Y_{t-1} + (1 - \alpha)Y_{t-2} + \dots, \text{ where } 0 < \alpha \leq 1$$

In this equation, Y_t is the value at the time T , and alpha (α) is the smoothing parameter. Algorithms optimize the alpha (and other parameters) by minimizing the errors, for example, **sum of squared error (SSE)** or **mean squared error (MSE)**.

The forecast equation along with trend and seasonality equations, if applicable, will be as follows:

- The forecast, where A is the preceding smoothing equation and h is the number of forecast periods, $Y_{t+h} = A + hB_t + S_t$
- The trend equation, $B_t = \beta(A_t - A_{t-1}) + (1 - \beta)B_{t-1}$
- The seasonality, where m is the number of seasonal periods, $S_t = \Omega(Y_t - A_{t-1} - B_{t-1}) + (1 - \Omega)S_{t-m}$

This equation is referred to as the **Holt-Winters Method**. The forecast equation is additive in nature with the trend as linear. The method also allows the inclusion of a dampened trend and multiplicative seasonality, where the seasonality proportionally increases or decreases over time. In my experience, the Holt-Winter's Method provides the best forecasts, even better than the ARIMA models. I have come to this conclusion on having to update long-term forecasts for hundreds of time series based on monthly data, and in roughly 90 percent of the cases, Holt-Winters produced minimal forecast error. Additionally, you don't have to worry about the assumption of stationarity as in an ARIMA model. Stationarity is where the time series has a constant mean, variance, and correlation between all the time periods. Having said this, it is still important to understand the ARIMA models as there will be situations where they have the best performance.

Starting with the autoregressive model, the value of Y at time T is a linear function of the prior values of Y . The formula for an autoregressive lag-1 model $AR(1)$, is $Y_t = constant + \Phi Y_{t-1} + E_t$. The critical assumptions for the model are as follows:

- E_t denotes the errors that are identically and independently distributed with a mean zero and constant variance
- The errors are independent of Y_t
- $Y_t, Y_{t-1}, Y_{t-n}...$ is stationary, which means that the absolute value of Φ is less than one

With a stationary time series, you can examine **Autocorrelation Function (ACF)**. The ACF of a stationary series gives correlations between Y_t and Y_{t-h} for $h = 1, 2...n$. Let's use R to create an $AR(1)$ series and plot it. In doing so, we will also look at the capabilities of the `ggfortify` package, which acts as a wrapper around `ggplot2` functions:

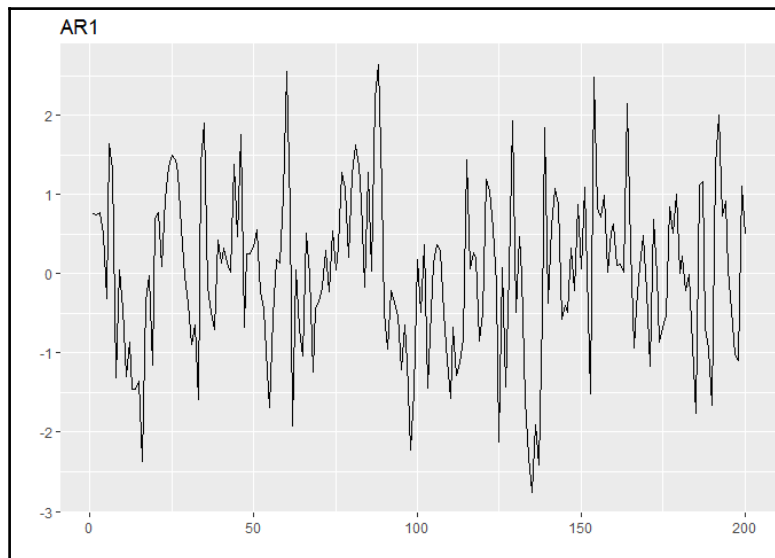
```
> library(ggfortify)

> set.seed(123)

> ar1 <- arima.sim(list(order = c(1, 0, 0), ar = 0.5), n = 200)

> autoplot(ar1, main = "AR1")
```

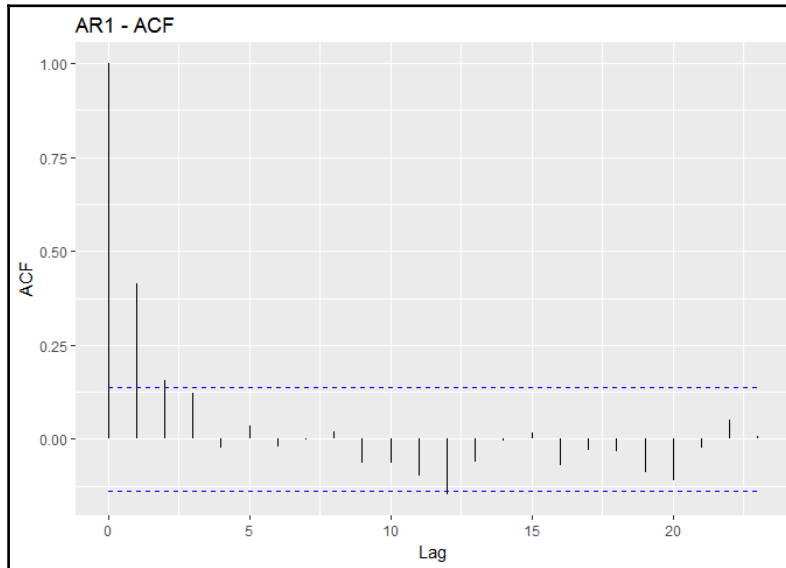
The following is the output of the preceding command:



Now, let's examine ACF:

```
> autoplot(acf(ar1, plot = F), main = "AR1 - ACF")
```

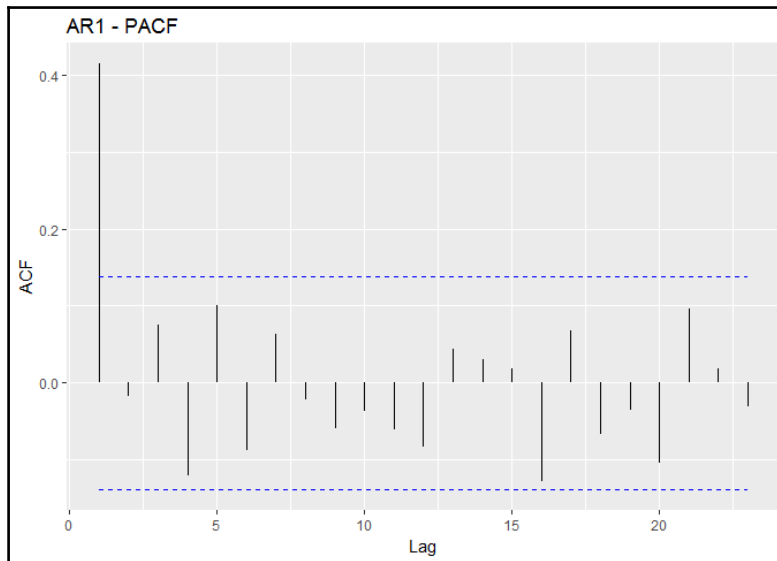
The output of the preceding command is as follows:



The **ACF** plot shows the correlations exponentially decreasing as the **Lag** increases. The dotted blue lines indicate the confidence bands of a significant correlation. Any line that extends above the high or below the low band is considered significant. In addition to ACF, one should also examine **Partial Autocorrelation Function (PACF)**. PACF is a conditional correlation, which means that the correlation between Y_t and Y_{t-h} is conditional on the observations that come between the two. One way to intuitively understand this is to think of a linear regression model and its coefficients. Let's assume that you have $Y = B_0 + B_1X_1$ versus $Y = B_0 + B_1X_1 + B_2X_2$. The relationship of X to Y in the first model is linear with a coefficient, but in the second model, the coefficient will be different because of the relationship between Y and X_2 now being accounted for as well. Note that in the following **PACF** plot, the partial autocorrelation value at lag-1 is identical to the autocorrelation value at lag-1, as this is not a conditional correlation:

```
> autoplot(pacf(ar1, plot = F), main = "AR1 - PACF")
```

The following is the output of the preceding command:



We can safely make the assumption that the series is stationary from the appearance of the preceding time series plot. We'll look at a couple of statistical tests in the practical exercise to ensure that the data is stationary, but most of the time, the eyeball test is sufficient. If the data is not stationary, then it is possible to detrend the data by taking its differences. This is the Integrated (I) in ARIMA. After differencing, the new series becomes $\Delta Y_t = Y_t - Y_{t-1}$. One should expect a first-order difference to achieve stationarity, but on some occasions, a second-order difference may be necessary. An ARIMA model with $AR(1)$ and $I(1)$ would be annotated as $(1,1,0)$.

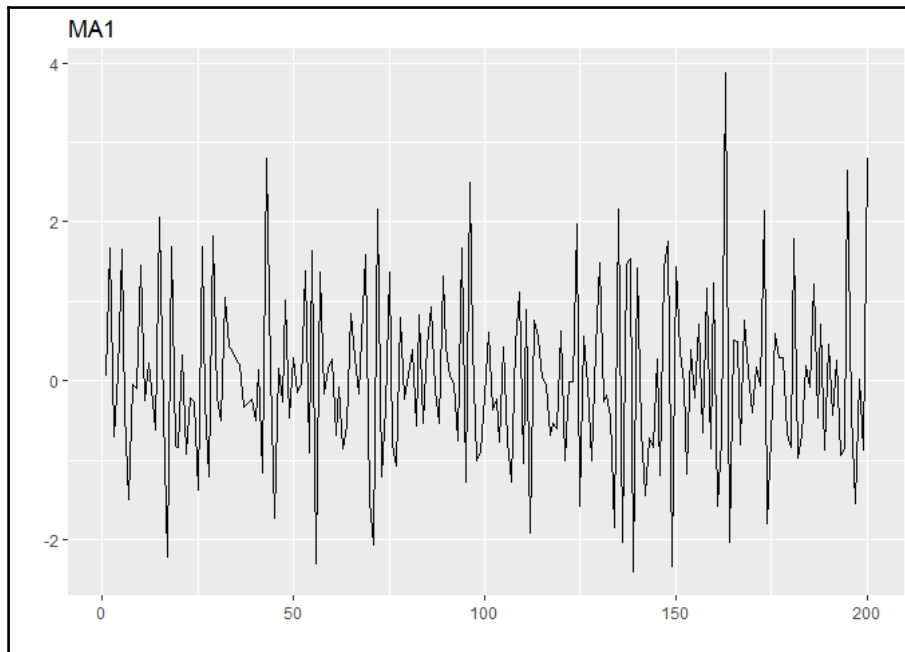
The MA stands for moving average. This is not the simple moving average as the 50-day moving average of a stock price, it's rather a coefficient that is applied to the errors. The errors are, of course, identically and independently distributed with a mean zero and constant variance. The formula for an $MA(1)$ model is $Y_t = constant + E_t + \Theta E_{t-1}$. As we did with the $AR(1)$ model, we can build an $MA(1)$ in R, as follows:

```
> set.seed(123)

> ma1 <- arima.sim(list(order = c(0, 0, 1), ma = -0.5), n = 200)

> autoplot(ma1, main = "MA1")
```

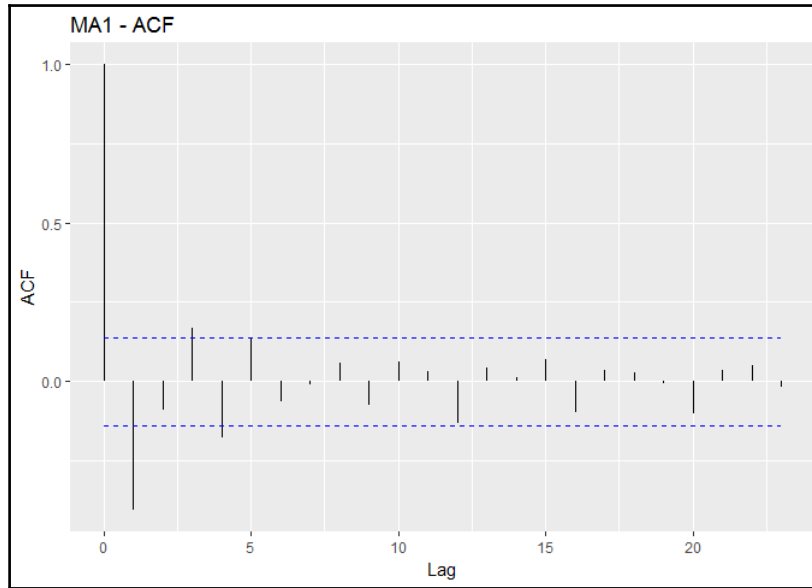
The following is the output of the preceding command:



The ACF and PACF plots are a bit different from the $AR(1)$ model. Note that there are some rules of thumb while looking at the plots in order to determine whether the model has AR and/or MA terms. They can be a bit subjective; so I will leave it to you to learn these heuristics, but trust R to identify the proper model. In the following plots, we will see a significant correlation at lag-1 and two significant partial correlations at lag-1 and lag-2:

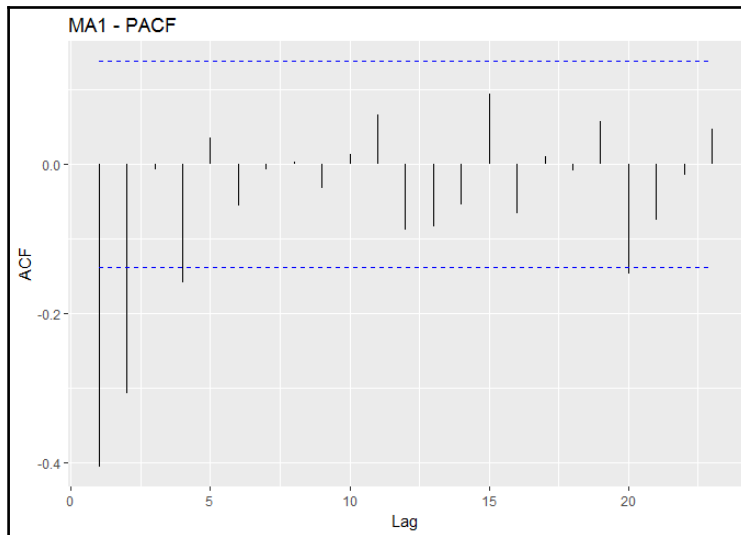
```
> autoplot(acf(ma1, plot = F), main = "MA1 - ACF")
```

The output of the preceding command is as follows:



The preceding figure is the ACF plot, and now, we will see the PACF plot:

```
> autoplot(pacf(ma1, plot = F), main = "MA1 - PACF")
```



With the ARIMA models, it is possible to incorporate seasonality, including the autoregressive, integrated, and moving average terms. The nonseasonal ARIMA model notation is commonly (p,d,q) . With seasonal ARIMA, assume that the data is monthly, then the notation would be $(p,d,q) \times (P,D,Q)12$, with the '12' in the notation taking the monthly seasonality into account. In the packages that we will use, R will automatically identify whether the seasonality should be included; if so, the optimal terms will be included as well.

Understanding Granger causality

Imagine you are asked a question such as, "What is the relationship between the amount of new prescriptions and total prescriptions for medicine X?". You know that these are measured monthly, so what could you do to understand that relationship, given that people believe that new scripts will drive up total scripts. Or, how about testing the hypothesis that commodity prices, in particular copper, is a leading indicator of stock market prices in the US? Well, with two sets of time series data, x and y , Granger causality is a method that attempts to determine whether one series is likely to influence a change in the other. This is done by taking different lags of one series and using this to model the change in the second series. To accomplish this, we will create two models that will predict y , one with only the past values of y (Ω) and the other with the past values of y and x (Π). The models are as follows, where k is the number of lags in the time series:

$$\begin{aligned} \text{Let } \Omega = y_t &= \beta_0 + \beta_1 y_{t-1} + \dots + \beta_k y_{t-k} + \epsilon \\ \text{and let } \Pi = y_t &= \beta_0 + \beta_1 y_{t-1} + \dots + \beta_k y_{t-k} + \alpha_1 x_{t-1} + \dots + \alpha_k x_{t-k} + \epsilon \end{aligned}$$

The RSS are then compared and F -test is used to determine whether the nested model (Ω) is adequate enough to explain the future values of y or whether the full model (Π) is better. F -test is used to test the following null and alternate hypotheses:

- H_0 : $\alpha_i = 0$ for each $i \in [1, k]$, no Granger causality
- H_1 : $\alpha_i \neq 0$ for at least one $i \in [1, k]$, Granger causality

Essentially, we are trying to determine whether we can say that statistically, x provides more information about the future values of y than the past values of y alone. In this definition, it is clear that we are not trying to prove actual causation; only that the two values are related by some phenomenon. Along these lines, we must also run this model in reverse in order to verify that y does not provide information about the future values of x . If we find that this is the case, it is likely that there is some exogenous variable, say Z , that needs to be controlled or would possibly be a better candidate for the Granger causation. To avoid spurious results, the method should be applied to a stationary time series. Note that research papers are available that discuss the techniques nonlinear models use, but this is outside of the scope for this book; however, we will examine it from a non-stationary standpoint. There is an excellent introductory paper that revolves around the age-old conundrum of the chicken and the egg (Thurman, 1988).

There are a couple of different ways to identify the proper lag structure. Naturally, one can use brute force and ignorance to test all the reasonable lags, one at a time. One may have a rational intuition based on domain expertise or perhaps prior research that exists to guide the lag selection. If not, then **Vector Autoregression (VAR)** can be applied to identify the lag structure with the lowest information criterion, such as **Aikake's Information Criterion (AIC)** or **Final Prediction Error (FPE)**. For simplicity, here is the notation for the VAR models with two variables, and this incorporates only one lag for each variable. This notation can be extended for as many variables and lags as appropriate.

- $Y = \text{constant}_1 + B_{11}Y_{t-1} + B_{12}Y_{t-1} + e_1$
- $X = \text{constant}_1 + B_{21}Y_{t-1} + B_{22}Y_{t-1} + e_2$

In R, this process is quite simple to implement as we will see in the following practical problem.

Business understanding

The planet isn't going anywhere. We are! We're goin' away.

- Philosopher and comedian, George Carlin

Climate change is happening. It always has and will, but the big question, at least from a political and economic standpoint, is I the climate change man-made? I will use this chapter to put econometric time series modeling to the test to try and learn whether carbon emissions cause, statistically speaking, climate change, and in particular, rising temperatures. Personally, I would like to take a neutral stance on the issue, always keeping in mind the tenets that Mr. Carlin left for us in his teachings on the subject.

The first order of business is to find and gather the data. For temperature, I chose the **HadCRUT4** annual median temperature time series, which is probably the gold standard. This data is compiled by a cooperative effort of the Climate Research Unit of the University of East Anglia and the Hadley Centre at the UK's Meteorological Office. A full discussion of how the data is compiled and modeled is available at

<http://www.metoffice.gov.uk/hadobs/index.html>.

The data that we will use is provided as an annual anomaly, which is calculated as the difference of the median annual surface temperature for a given time period versus the average of the reference years (1961-1990). The annual surface temperature is an ensemble of the temperatures collected globally and blended from the **CRUTEM4** surface air temperature and **HadSST3** sea-surface datasets. This data has come under attack as biased and unreliable:

<http://www.telegraph.co.uk/comment/11561629/Top-scientists-start-to-examine-fiddled-global-warming-figures.html>. This is way outside of our scope of effort here, so we must accept and utilize this data as it is. I've pulled the data from 1919 March, 1958 through 2013 to match our CO2 data.

Global CO2 emission estimates can be found at the **Carbon Dioxide Information Analysis Center (CDIAC)** of the US Department of Energy at the following website: <http://cdiac.ornl.gov/>

I've placed the data in a .csv file (`climate.csv`) for you to download and store in your working directory: <https://github.com/datameister66/data/>

Let's load it and examine the structure:

```
> climate <- read.csv("climate.csv", stringsAsFactors = F)

> str(climate)
'data.frame': 95 obs. of 3 variables:
 $ Year: int 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928 ...
 $ CO2 : int 806 932 803 845 970 963 975 983 1062 1065 ...
 $ Temp: num -0.272 -0.241 -0.187 -0.301 -0.272 -0.292 -0.214
        -0.105 -0.208 -0.206 ...
```

Finally, we will put this in a time series structure, specifying the start and end years:

```
> climate <- ts(climate[, 2:3], frequency = 12,
  start = 1919, end = 2013)

> head(climate)
      CO2    Temp
[1,] 806 -0.272
[2,] 932 -0.241
[3,] 803 -0.187
```

```
[4,] 845 -0.301
[5,] 970 -0.272
[6,] 963 -0.292
```

With our data loaded and put in time series structures, we can now begin to understand and further prepare it for analysis.

Data understanding and preparation

Two packages are required for this effort, so ensure they are installed on your system:

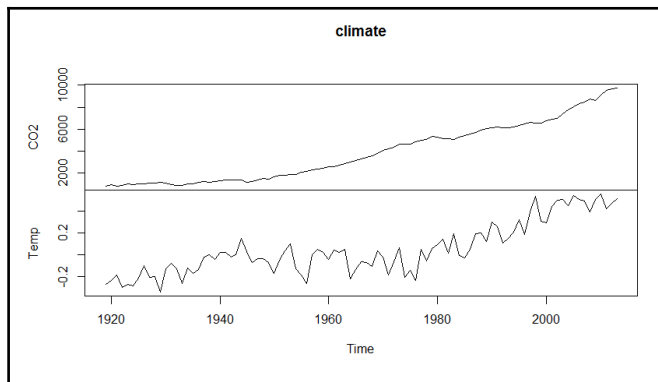
```
> library(forecast)

> library(tseries)
```

Let's start out with plots of the two time series:

```
> autoplot(climate)
```

The output of the preceding command is as follows:



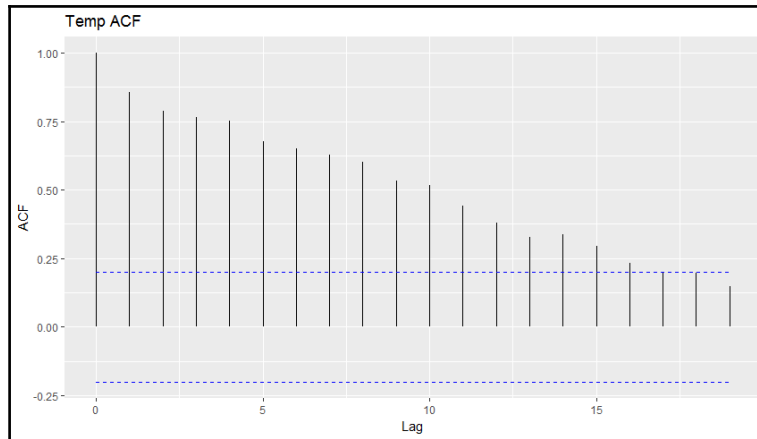
It appears that CO2 levels really started to increase after World War II and a rapid rise in temperature anomalies in the mid-1970s. There does not appear to be any obvious outliers, and variation over time appears constant. Using the standard procedure, we can see that the two series are highly correlated, as follows:

```
> cor(climate)
      CO2      Temp
CO2  1.0000000 0.8404215
Temp 0.8404215 1.0000000
```

As discussed earlier, this is nothing to jump for joy about as it proves absolutely nothing. We will look for the structure by plotting ACF and PACF for both series:

```
> autoplot(acf(climate[, 2], plot = F), main="Temp ACF")
```

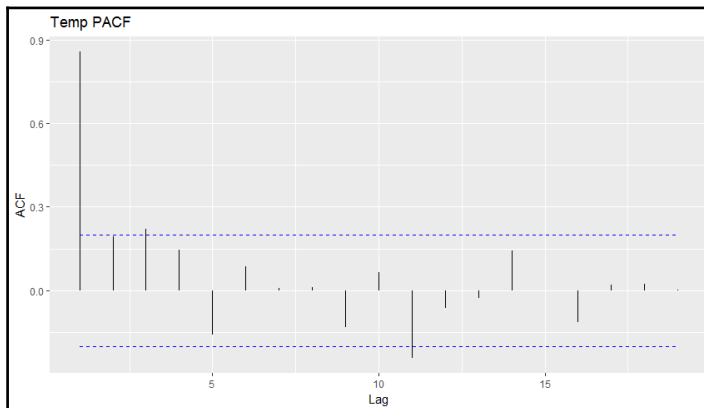
The output of the preceding code snippet is as follows:



This code gives us the PACF plot for temperature:

```
> autoplot(pacf(climate[, 2], plot = F), main = "Temp PACF")
```

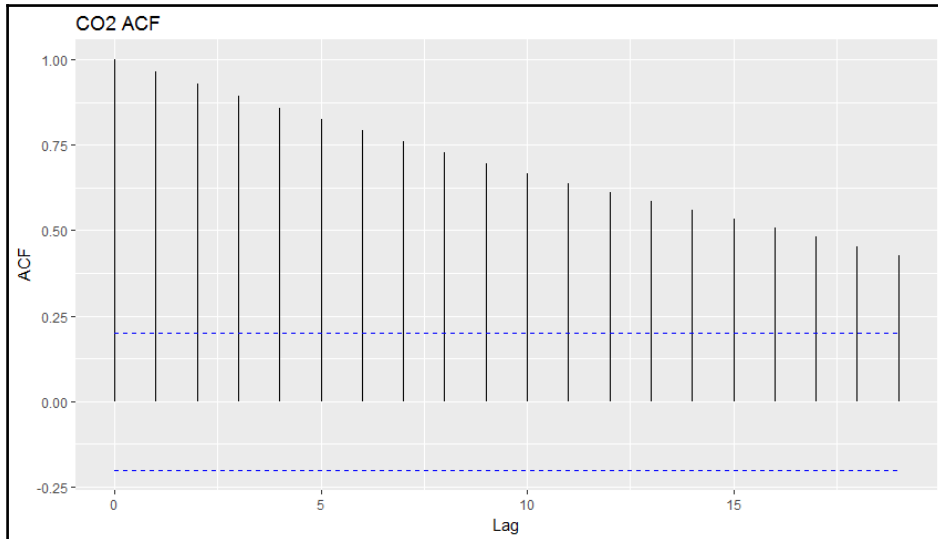
The output of the preceding code snippet is as follows:



This code gives us the ACF plot for CO2:

```
> autoplot(acf(climate[, 1], plot = F), main = "CO2 ACF")
```

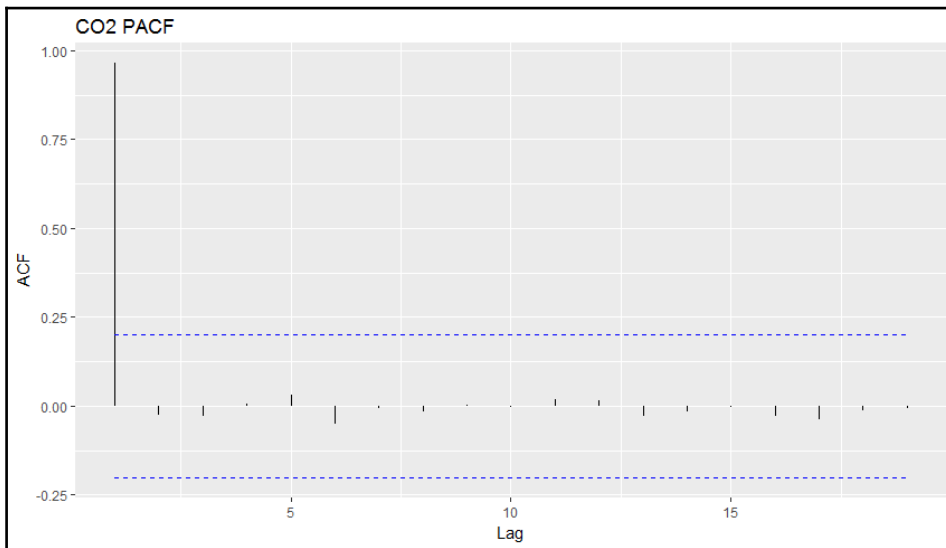
The output of the preceding code snippet is as follows:



This code gives us the PACF plot for CO2:

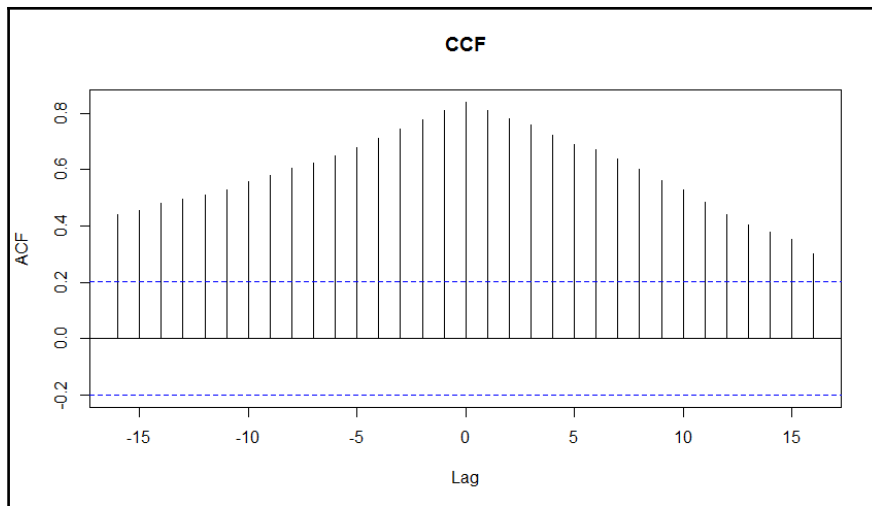
```
> autoplot(pacf(climate[, 1], plot = F), main = "CO2 PACF")
```

The output of the preceding code snippet is as follows:



With the slowly decaying ACF patterns and rapidly decaying PACF patterns, we can assume that these series are both autoregressive, although temp appears to have some significant MA terms. Next, let's have a look at **Cross Correlation Function (CCF)**. Note that we put our x before our y in the function:

```
> ccf(climate[, 1], climate[, 2], main = "CCF")
```



CCF shows us the correlation between the temperature and lags of CO2. If the negative lags of the x variable have a high correlation, we can say that x leads y . If the positive lags of x have a high correlation, we say that x lags y . Here, we can see that CO2 is both a leading and lagging variable. For our analysis, it is encouraging that we see the former, but odd that we see the latter. We will see during the VAR and Granger causality analysis whether this will matter or not.

Additionally, we need to test whether the data is stationary. We can prove this with the **Augmented Dickey-Fuller (ADF)** test available in the `tseries` package, using the `adf.test()` function, as follows:

```
> adf.test(climate[, 1])
```

Augmented Dickey-Fuller Test

```
data: climate[, 1]
Dickey-Fuller = -1.1519, Lag order = 4, p-value =
0.9101
alternative hypothesis: stationary
```

```
> adf.test(climate[, 2])

Augmented Dickey-Fuller Test

data: climate[, 2]
Dickey-Fuller = -1.8106, Lag order = 4, p-value =
0.6546
alternative hypothesis: stationary
```

For both series, we have insignificant *p*-values, so we cannot reject the null and conclude that they are not stationary.

Having explored the data, let's begin the modeling process, starting with the application of univariate techniques to the temperature anomalies.

Modeling and evaluation

For the modeling and evaluation step, we will focus on three tasks. The first is to produce a univariate forecast model applied to just the surface temperature. The second is to develop a regression model of the surface temperature based on itself and CO2 levels, using that output to inform our work on whether CO2 levels Granger cause the surface temperature anomalies.

Univariate time series forecasting

With this task, the objective is to produce a univariate forecast for the surface temperature, focusing on choosing either a Holt linear trend model or an ARIMA model. We will train the models and determine their predictive accuracy on an out-of-time test set, just like we've done in other learning endeavors. The following code creates the temperature subset and then the train and test sets, starting after WWII:

```
> temp <- climate[, 2]

> temp <- climate[, 2]

> train <- window(temp, start = 1946, end = 2003)

> test <- window(temp, start = 2004)
```

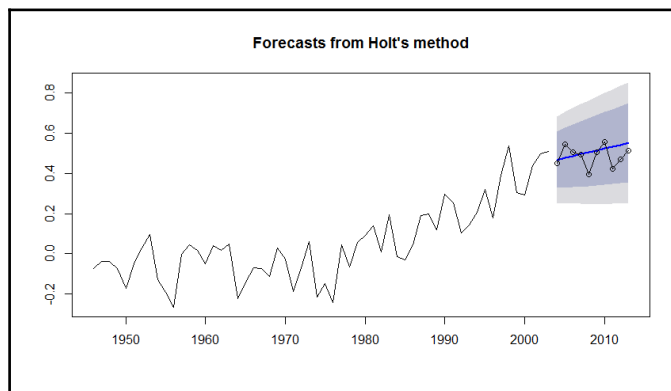
To build our smoothing model, we will use the `holt()` function found in the `forecast` package. We will build two models, one with and one without a damped trend. In this function, we will need to specify the time series, number of forecast periods as $h = \dots$, method to select the initial state values, either "optimal" or "simple", and whether we want a damped trend. Specifying "optimal", the algorithm will find optimal initial starting values along with the smoothing parameters, while "simple" calculates starting values using the first few observations. Now, in the `forecast` package, you can use the `ets()` function, which will find all the optimal parameters. However, in our case, let's stick with `holt()` so that we can compare methods. Let's try the `holt` model without a damped trend, as follows:

```
> fit.holt <- holt(train, h = 10, initial = "optimal")
```

Plot the `forecast` and see how well it performed out of sample with the following code:

```
> plot(forecast(fit.holt))  
  
> lines(test, type = "o")
```

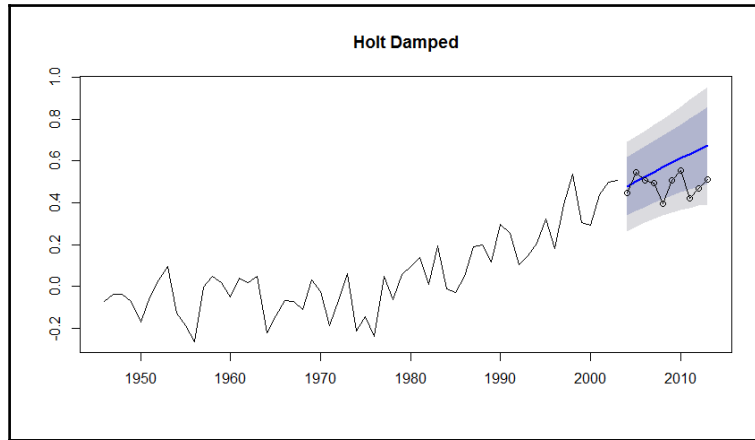
The output of the preceding code is as follows:



Looking at the plot, it seems that this forecast is showing a slight linear uptrend. Let's have a go by including the damped trend, as follows:

```
> fit.holtd <- holt(train, h = 10, initial = "optimal", damped =  
  TRUE)  
  
> plot(forecast(fit.holtd), main = "Holt Damped")  
  
> lines(test, type = "o")
```

The output of the preceding code is as follows:



Lastly, in univariate analysis, we develop an ARIMA model, using `auto.arima()`, which is also from the `forecast` package. There are many options that you can specify in the function, or you can just include your time series data and it will find the best ARIMA fit:

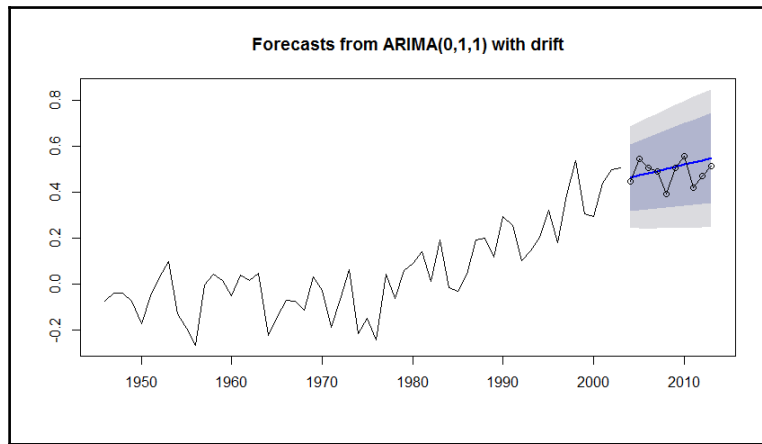
```
> fit.arima <- auto.arima(train)
> summary(fit.arima)
Series: train
ARIMA(0,1,1) with drift

Coefficients:
      ma1  drift
-0.6949  0.0094
s.e.  0.1041  0.0047
```

The abbreviated output shows that the model selected is an $MA = 1$, $I = 1$, or $ARIMA(0, 1, 1)$ with drift (equivalent to an intercept term). We can examine the plot of its performance on the `test` data in the same fashion as before:

```
> plot(forecast(fit.arima, h = 10))
> lines(test, type="o")
```


The output of the preceding code is as follows:



This is very similar to the `holt` method with no damped trend. We can score each model to find the one that provides the lowest error, mean absolute percentage error (MAPE), with the following code:

```
> mapeHOLT <- sum(abs((test - fit.holt$mean)/test))/10

> mapeHOLT
[1] 0.105813

> mapeHOLTD <- sum(abs((test - fit.holtd$mean)/test))/10

> mapeHOLTD
[1] 0.2220256

> mapeARIMA <- sum(abs((test - forecast(fit.arma, h =
  10)$mean)/test))/10

> mapeARIMA
[1] 0.1034813
```

The forecast error is slightly less with the ARIMA 0,1,1 versus the `holt` methods, and clearly, the damped trend model performed the worst.

With the statistical and visual evidence, it seems that the best choice for a univariate forecast model is the ARIMA model. Interestingly, in the first edition using annual data, the Holt method with a damped trend had the best accuracy.

With this, we've completed the building of a univariate forecast model for the surface temperature anomalies, and now we will move on to the next task of seeing if CO2 levels cause these anomalies.

Examining the causality

For this chapter, this is where I think the rubber meets the road and we will separate causality from mere correlation, well, statistically speaking anyway. This is not the first time that this technique has been applied to the problem. Triacca (2005) found no evidence to suggest that atmospheric CO2 Granger caused the surface temperature anomalies. On the other hand, Kodra (2010) concluded that there is a causal relationship, but put forth the caveat that their data was not stationary even after a second-order differencing. While this effort will not settle the debate, it will hopefully inspire you to apply the methodology in your personal endeavors. The topic at hand certainly provides an effective training ground to demonstrate the Granger causality.

Our plan here is to first demonstrate spurious linear regression where the residuals suffer from autocorrelation, also known as serial correlation. Then, we will examine two different approaches to Granger causality. The first will be the traditional methods, where both series are stationary. Then, we will look at the method demonstrated by Toda and Yamamoto (1995), which applies the methodology to the raw data or, as it is sometimes called, the "levels".

Linear regression

Let's get started with the spurious regression then, which I have seen implemented in the real world far too often. Here we simply build a linear model and examine the results:

```
> fit.lm <- lm(Temp ~ CO2, data = climate)

> summary(fit.lm)

Call:
lm(formula = Temp ~ CO2, data = climate)

Residuals:
    Min       1Q   Median       3Q      Max
-0.36411 -0.08986  0.00011  0.09475  0.28763

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -2.430e-01  2.357e-02  -10.31  <2e-16 ***
```

```
CO2          7.548e-05   5.047e-06   14.96   <2e-16 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

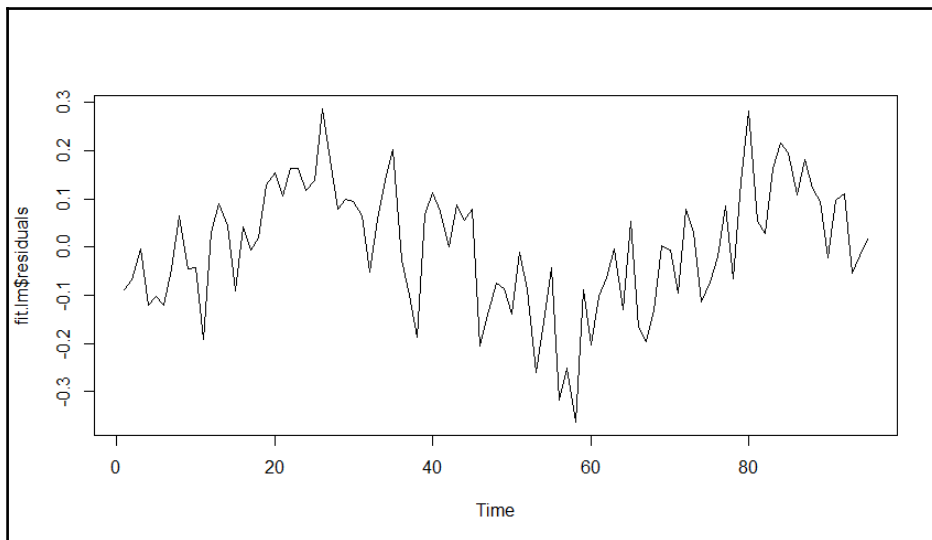
Residual standard error: 0.1299 on 93 degrees of freedom
Multiple R-squared:  0.7063, Adjusted R-squared:  0.7032
F-statistic: 223.7 on 1 and 93 DF, p-value: < 2.2e-16
```

Notice how everything is significant, and we have an adjusted R-squared of 0.7. OK, they are highly correlated but this is all meaningless as discussed by Granger and Newbold (1974). Again, I have seen results like these presented in meetings with many people with advanced degrees, and I had to be the bad guy and challenge the results.

We can plot the serial correlation, starting with a time series plot of the residuals, which produce a clear pattern:

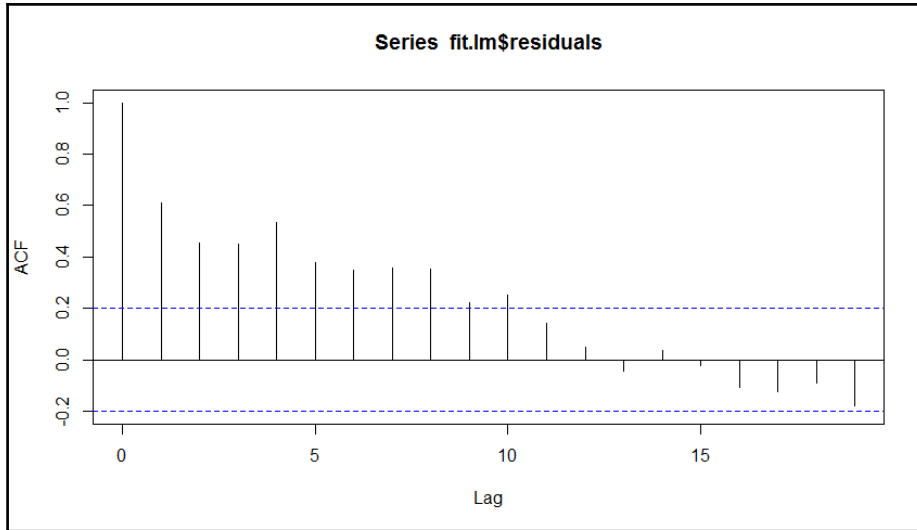
```
> plot.ts(fit.lm$residuals)
```

The output of the preceding code is as follows:



Then we create an ACF plot showing significant autocorrelation out to lag 10:

```
> acf(fit.lm$residuals)
```



You can test for autocorrelation by performing the Durbin-Watson test. The null hypothesis in the test is no autocorrelation exists:

```
> dwtest(fit.lm)
```

Durbin-Watson test

data: fit.lm

DW = 0.77425, **p-value** = 4.468e-12

alternative hypothesis: true autocorrelation is greater than 0

From examining the plots, it comes as no surprise that we can safely reject the null hypothesis of no autocorrelation. The simple way to deal with autocorrelation is to incorporate lagged variables of the dependent time series and/or to make all the data stationary. We will do that next using vector autoregression to identify the appropriate lag structure to incorporate in our causality efforts.

Vector autoregression

We've seen in the preceding section that temperature is stationary and CO2 requires a first order difference. Another simple way to show this is with the `forecast` package's `ndiffs()` function. It provides an output that spells out the minimum number of differences needed to make the data stationary. In the function, you can specify which test out of the three available ones you would like to use: **Kwiatkowski, Philips, Schmidt & Shin (KPSS)**, **Augmented Dickey Fuller (ADF)**, or **Philips-Peron (PP)**. I will use ADF in the following code, which has a null hypothesis that the data is not stationary:

```
> ndiffs(climate[, 1], test = "adf")
[1] 1

> ndiffs(climate[, 2], test = "adf")
[1] 1
```

We see that both require a first order difference to become stationary. To get started, we will create the difference. Then, we will complete the traditional approach, where both series are stationary. Let's also load our packages for this exercise:

```
> library(vars)

> library(aod)

> climateDiff <- diff(climate)

> climateDiff <- window(climateDiff, start = 1946)

> head(climateDiff)
      CO2      Temp
[1,]   78   -0.099
[2,]  154    0.034
[3,]   77    0.001
[4,]  -50   -0.035
[5,]  211   -0.100
[6,]  137    0.121
```

It is now a matter of determining the optimal lag structure based on the information criteria using vector autoregression. This is done with the `VARselect` function in the `vars` package. You only need to specify the data and number of lags in the model using `lag.max = x` in the function. Let's use a maximum of 12 lags:

```
> lag.select <- VARselect(climateDiff, lag.max = 12)

> lag.select$selection
AIC(n) HQ(n) SC(n) FPE(n)
    5     1     1     5
```

We called the information criteria using `lag$selection`. Four different criteria are provided including **AIC**, **Hannan-Quinn Criterion (HQ)**, **Schwarz-Bayes Criterion (SC)**, and **FPE**. Note that AIC and SC are covered in Chapter 2, *Linear Regression - The Blocking and Tackling of Machine Learning*, so I will not go over the criterion formulas or differences here. If you want to see the actual results for each lag, you can use `lag$criteria`. We can see that AIC and FPE have selected lag 5 and HQ and SC lag 1 as the optimal structure to a VAR model. It seems to make sense that the 5-year lag is the one to use. We will create that model using the `var()` function. I'll let you try it with lag 1:

```
> fit1 <- VAR(climateDiff, p = 5)
```

The summary results are quite lengthy as it builds two separate models and would take up probably two whole pages. What I provide is the abbreviated output showing the results with temperature as the prediction:

```
> summary(fit1)
Residual standard error: 0.1006 on 52 degrees of freedom
Multiple R-Squared: 0.4509, Adjusted R-squared: 0.3453
F-statistic: 4.27 on 10 and 52 DF, p-value: 0.0002326
```

The model was significant with a resulting adjusted R-square of 0.35.

As we did in the previous section, we should check for serial correlation. Here, the `VAR` package provides the `serial.test()` function for multivariate autocorrelation. It offers several different tests, but let's focus on `Portmanteau Test`, and also please note that the `DW` test is for univariate series only. The null hypothesis is that autocorrelations are zero and the alternate is that they are not zero:

```
> serial.test(fit1, type = "PT.asymptotic")

Portmanteau Test (asymptotic)

data: Residuals of VAR object fit1
Chi-squared = 35.912, df = 44, p-value = 0.8021
```

With `p-value` at 0.3481, we do not have evidence to reject the null and can say that the residuals are not autocorrelated. What does the test say with 1 lag?

To do the Granger causality tests in R, you can use either the `lmtest` package and the `Grangertest()` function or the `causality()` function in the `vars` package. I'll demonstrate the technique using `causality()`. It is very easy as you just need to create two objects, one for `x` causing `y` and one for `y` causing `x`, utilizing the `fit1` object previously created:

```
> x2y <- causality(fit1, cause = "CO2")

> y2x <- causality(fit1, cause = "Temp")
```

It is now just a simple matter to call the Granger test results:

```
> x2y$Granger

Granger causality H0: CO2_diff do not Granger-cause
climate2.temp

data: VAR object fit1
F-Test = 2.2069, df1 = 5, df2 = 104, p-value = 0.05908

> y2x$Granger

Granger causality H0: climate2.temp do not Granger-cause
CO2_diff

data: VAR object fit1
F-Test = 0.66783, df1 = 5, df2 = 104, p-value = 0.6487
```

The `p-value` value for CO2 differences of Granger causing temperature is 0.05908 and not significant in the other direction. So what does all this mean? The first thing we can say is that Y does not cause X. As for X causing Y, we cannot reject the null at the 0.05 significance level and therefore conclude that X does not Granger cause Y. However, is that the relevant conclusion here? Remember, the p-value evaluates how likely the effect is if the null hypothesis is true. Also, remember that the test was never designed to be some binary Yay or Nay. If this was a controlled experiment, it would be unlikely for us to hesitate to say we had insufficient evidence to reject the null, like the **Food and Drug Administration (FDA)** would do for a phase 3 clinical trial. Since this study is based on observational data, I believe we can say that it is highly probable that "CO2 emissions Granger cause Surface Temperature Anomalies". But, there is a lot of room for criticism on that conclusion. I mentioned upfront the controversy around the quality of the data. The thing that still concerns me is what year to start the analysis. I chose 1945 because it looked about right; you could say I applied *proc eyeball* in SAS terminology. What year is chosen has a dramatic impact on the analysis, changing the lag structure and also leading to insignificant p-values.

However, we still need to model the original CO2 levels using the alternative Granger causality technique. The process to find the correct number of lags is the same as before, except we do not need to make the data stationary:

```
> climateLevels <- window(climate, start = 1946)

> level.select <- VARselect(climateLevels, lag.max = 12)

> level.select$selection
AIC(n) HQ(n) SC(n) FPE(n)
    10     1     1     6
```

Let's try the lag 6 structure and see whether we can achieve significance, remembering to add one extra lag to account for the integrated series. A discussion on the technique and why it needs to be done is available at <http://davegiles.blogspot.de/2011/04/testing-for-granger-causality.html>:

```
fit2 <- VAR(climateLevels, p = 7)
> serial.test(fit2, type = "PT.asymptotic")

Portmanteau Test (asymptotic)

data: Residuals of VAR object fit2
Chi-squared = 35.161, df = 36, p-value = 0.5083
```


Now, to determine Granger causality for X causing Y, you conduct a Wald test, where the coefficients of X and only X are 0 in the equation to predict Y, remembering to not include the extra coefficients that account for integration in the test.

The Wald test in R is available in the `aod` package we've already loaded. We need to specify the coefficients of the full model, its variance-covariance matrix, and the coefficients of the causative variable.



The coefficients for Temp that we need to test in the VAR object consist of a range of even numbers from 2 to 12, while the coefficients for CO2 are odd from 1 to 11. Instead of using `c(2, 4, 6, and so on)` in our function, let's create an object with base R's `seq()` function.

First, let's see how CO2 does Granger causing temperature:

```
> CO2terms <- seq(1, 11, 2)
> Tempterms <- seq(2, 12, 2)
```

We are now ready to run the `wald` test, described in the following code:

```
> wald.test(b = coef(fit2$varresult$Temp),
  Sigma = vcov(fit2$varresult$Temp),
  Terms = c(CO2terms))
Wald test:
-----

Chi-squared test:
X2 = 11.5, df = 6, P(> X2) = 0.074
```

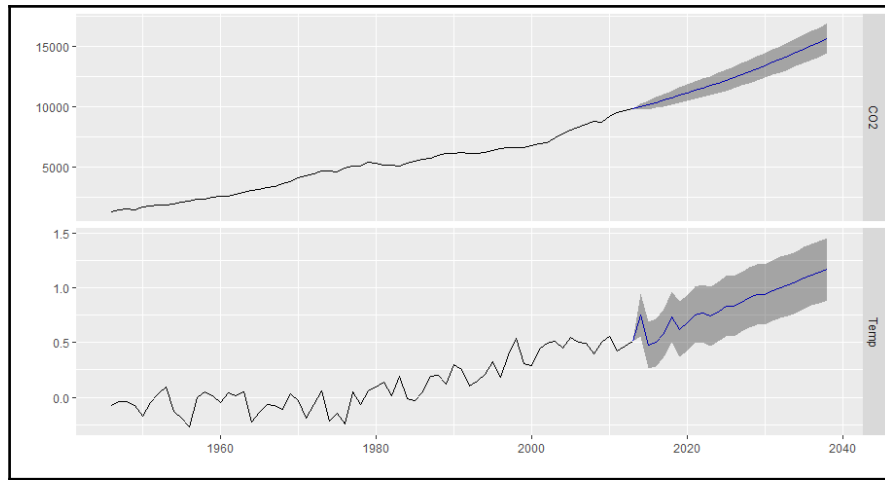
How about that? We are close to the magical 0.05 p-value. Let's test the other direction causality with the following code:

```
> wald.test(b = coef(fit2$varresult$CO2),
  Sigma = vcov(fit2$varresult$CO2),
  Terms = c(Tempterms))
Wald test:
-----

Chi-squared test:
X2 = 3.9, df = 6, P(> X2) = 0.69
```

The last thing to show here is how to use a vector autoregression to produce a forecast. A `predict` function is available, so let's `autoplot()` it for a 25-year period and see what happens:

```
> autoplot(predict(fit2, n.ahead = 25, ci = 0.95))
```



It seems dark days lie ahead, perhaps, to coin a phrase, "Winter is Coming" from the popular TV series *Game of Thrones*. Fine by me as my investment and savings plan for a long time has consisted of canned goods and ammunition. What else am I supposed to do, ride a horse to work? I'll do that the day Al Gore does. In the meantime, I am going to work on my suntan.

If nothing else, I hope it has stimulated your thinking on how to apply the technique to your own real-world problems or maybe even to examine the climate change data in more detail. There should be a high bar when it comes to demonstrating causality, and Granger causality is a great tool for assisting in that endeavor.

Summary

In this chapter, the goal was to discuss how important the element of time is in the field of machine learning and analytics, to identify the common traps when analyzing the time series, and demonstrate the techniques and methods to work around these traps. We explored both the univariate and bivariate time series analyses for global temperature anomalies and human carbon dioxide emissions. Additionally, we looked at Granger causality to determine whether we can say, statistically speaking, that atmospheric CO₂ levels cause surface temperature anomalies. We discovered that the p-values are higher than 0.05 but less than 0.10 for Granger causality from CO₂ to temperature. It does show that Granger causality is an effective tool in investigating causality in machine learning problems. In the next chapter, we will shift gears and take a look at how to apply learning methods to textual data.

Additionally, keep in mind that in time series analysis, we just skimmed the surface. I encourage you to explore other techniques around changepoint detection, decomposition of time series, nonlinear forecasting, and many others. Although not usually considered part of the machine learning toolbox, I believe you will find it an invaluable addition to yours.

13

Text Mining

"I think it's much more interesting to live not knowing than to have answers which might be wrong."

- Richard Feynman

The world is awash with textual data. If you Google, Bing, or Yahoo how much of that data is unstructured, that is, in a textual format, estimates would range from 80 to 90 percent. The real number doesn't matter. What does matter is that a large proportion of the data is in a text format. The implication is that anyone seeking to find insights in that data must develop the capability to process and analyze text.

When I first started out as a market researcher, I used to manually pore through page after page of moderator-led focus groups and interviews with the hope of capturing some qualitative insight an Aha! moment if you will-and then haggle with fellow team members over whether they had the same insight or not. Then, you would always have that one individual in a project who would swoop in and listen to two interviews-out of the 30 or 40 on the schedule and, alas, they had their mind made up on what was really happening in the world. Contrast that with the techniques being used now, where an analyst can quickly distill data into meaningful quantitative results, support qualitative understanding, and maybe even sway the swooper.

Over the last few years, I've applied the techniques discussed here to mine physician-patient interactions, understand FDA fears on prescription drug advertising, and capture patient concerns about a rare cancer, to name just a few. Using R and the methods in this chapter, you too can extract the powerful information in textual data.

Text mining framework and methods

There are many different methods to use in text mining. The goal here is to provide a basic framework to apply to such an endeavor. This framework is not all-inclusive of the possible methods but will cover those that are probably the most important for the vast majority of projects that you will work on. Additionally, I will discuss the modeling methods in as succinct and clear a manner as possible, because they can get quite complicated. Gathering and compiling text data is a topic that could take up several chapters. Therefore, let's begin with the assumption that our data is available from Twitter, a customer call center, scraped off the web, or whatever, and is contained in some sort of text file or files.

The first task is to put the text files in one structured file referred to as a **corpus**. The number of documents could be just one, dozens, hundreds, or even thousands. R can handle a number of raw text files, including RSS feeds, PDF files, and MS Word documents. With the corpus created, the data preparation can begin with the text transformation.

The following list is comprised of probably some of the most common and useful transformations for text files:

- Change capital letters to lowercase
- Remove numbers
- Remove punctuation
- Remove stop words
- Remove excess whitespace
- Word stemming
- Word replacement

In transforming the corpus, you are creating not only a more compact dataset, but also simplifying the structure in order to facilitate relationships among the words, thereby leading to an increased understanding. However, keep in mind that not all of these transformations are necessary all the time and judgment must be applied, or you can iterate to find the transformations that make the most sense.

By changing words to lowercase, you can prevent the improper counting of words. Say that you have a count for hockey three times and Hockey once, where it is the first word in a sentence. R will not give you a count of hockey=4, but hockey=3 and Hockey=1.

Removing punctuation also achieves the same purpose, but as we will see in the business case, punctuation is important if you want to split your documents by sentences.

In removing stop words, you are getting rid of the common words that have no value; in fact, they are detrimental to the analysis, as their frequency masks important words. Examples of stop words *are, and, is, the, not, and to*. Removing whitespace makes a more compact corpus by getting rid of things such as tabs, paragraph breaks, double-spacing, and so on.

The stemming of words can get a bit tricky and might add to your confusion because it deletes word suffixes, creating the base word, or what is known as the **radical**. I personally am not a big fan of stemming and the analysts I've worked with agree with that sentiment. However, you can use the stemming algorithm included in the R package, `tm`, where the function calls the **porter stemming algorithm** from the `SnowballC` package. An example of stemming would be where your corpus has family and families. Recall that R would count this as two separate words. By running the stemming algorithm, the stemmed word for the two instances would become *famili*. This would prevent the incorrect count, but in some cases, it can be odd to interpret and is not very visually appealing in a wordcloud for presentation purposes. In some cases, it may make sense to run your analysis with both stemmed and unstemmed words in order to see which one makes sense.

Probably the most optional of the transformations is to replace the words. The goal of replacement is to combine the words with a similar meaning, for example, management and leadership. You can also use it in lieu of stemming. I once examined the outcome of stemmed and unstemmed words and concluded that I could achieve a more meaningful result by replacing about a dozen words instead of stemming.

With the transformation of the corpus completed, the next step is to create either a **Document-Term Matrix (DTM)** or **Term-Document Matrix (TDM)**. What either of these matrices does is create a matrix of word counts for each individual document in the matrix. A DTM would have the documents as rows and the words as columns, while in a TDM, the reverse is true. Text mining can be performed on either matrix.

With a matrix, you can begin to analyze the text by examining word counts and producing visualizations such as `wordclouds`. One can also find word associations by producing correlation lists for specific words. It also serves as a necessary data structure in order to build topic models.

Topic models

Topic models are a powerful method to group documents by their main topics. Topic models allow probabilistic modeling of term frequency occurrences in documents. The fitted model can be used to estimate the similarity between documents, as well as between a set of specified keywords using an additional layer of latent variables, which are referred to as topics (Grun and Hornik, 2011). In essence, a document is assigned to a topic based on the distribution of the words in that document, and the other documents in that topic will have roughly the same frequency of words.

The algorithm that we will focus on is **Latent Dirichlet Allocation (LDA)** with Gibbs sampling, which is probably the most commonly used sampling algorithm. In building topic models, the number of topics must be determined before running the algorithm (k -dimensions). If no apriori reason for the number of topics exists, then you can build several and apply judgment and knowledge to the final selection. LDA with Gibbs sampling is quite complicated mathematically, but my intent is to provide an introduction so that you are at least able to describe how the algorithm learns to assign a document to a topic in layman terms. If you are interested in mastering the math, block out a couple of hours on your calendar and have a go at it. Excellent background material is available at <http://www.cs.columbia.edu/~blei/papers/Blei2012.pdf>.

LDA is a generative process, and so, the following will iterate to a steady state:

1. For each document (j), there are 1 to j documents. We will randomly assign it a multinomial distribution (**dirichlet distribution**) to the topics (k) with 1 to k topics, for example, document A is 25 percent topic one, 25 percent topic two, and 50 percent topic three.
2. Probabilistically, for each word (i), there are 1 to i words to a topic (k); for example, the word *mean* has a probability of 0.25 for the topic statistics.
3. For each word(i) in document(j) and topic(k), calculate the proportion of words in that document assigned to that topic; note it as the probability of topic(k) with document(j), $p(k|j)$, and the proportion of word(i) in topic(k) from all the documents containing the word. Note it as the probability of word(i) with topic(k), $p(i|k)$.
4. Resample, that is, assign w a new t based on the probability that t contains w , which is based on $p(k|j)$ times $p(i|k)$.
5. Rinse and repeat; over numerous iterations, the algorithm finally converges and a document is assigned a topic based on the proportion of words assigned to a topic in that document.

The LDA that we will be doing assumes that the order of words and documents does not matter. There has been work done to relax these assumptions in order to build models of language generation and sequence models over time (known as **dynamic topic modelling**).

Other quantitative analyses

We will now shift gears to analyze text semantically based on sentences and the tagging of words based on the parts of speech, such as noun, verb, pronoun, adjective, adverb, preposition, singular, plural, and so on. Often, just examining the frequency and latent topics in the text will suffice for your analysis. However, you may find occasions when a deeper understanding of the style is required in order to compare the speakers or writers.

There are many methods to accomplish this task, but we will focus on the following five:

- Polarity (sentiment analysis)
- Automated readability index (complexity)
- Formality
- Diversity
- Dispersion

Polarity is often referred to as sentiment analysis, which tells you how positive or negative the text is. By analyzing polarity in R with the `qdap` package, a score will be assigned to each sentence and you can analyze the average and standard deviation of polarity by groups such as different authors, text, or topics. Different polarity dictionaries are available and `qdap` defaults to one created by Hu and Liu, 2004. You can alter or change this dictionary according to your requirements.

The algorithm works by first tagging the words with a positive, negative, or neutral sentiment based on the dictionary. The tagged words are then clustered based on the four words prior and two words after a tagged word, and these clusters are tagged with what are known as **valence shifters** (neutral, negator, amplifier, and de-amplifier). A series of weights based on their number and position are applied to both the words and clusters. This is then summed and divided by the square root of the number of words in that sentence.

Automated readability index is a measure of the text complexity and a reader's ability to understand. A specific formula is used to calculate this index: $4.71(\# \text{ of characters} / \# \text{ of words}) + 0.5(\# \text{ of words} / \# \text{ of sentences}) - 21.43$.

The index produces a number, which is a rough estimate of a student's grade level to fully comprehend. If the number is 9, then a high school freshman, aged 13 to 15, should be able to grasp the meaning of the text.

The formality measure provides an understanding of how a text relates to the reader or speech relates to a listener. I like to think of it as a way to understand how comfortable the person producing the text is with the audience, or an understanding of the setting where this communication takes place. If you want to experience formal text, attend a medical conference or read a legal document. Informal text is said to be contextual in nature.

The formality measure is called **F-Measure**. This measure is calculated as follows:

- Formal words (f) are nouns, adjectives, prepositions, and articles
- Contextual words (c) are pronouns, verbs, adverbs, and interjections
- $N = \text{sum of } (f + c + \text{conjunctions})$
- $\text{Formality Index} = 50((\text{sum of } f - \text{sum of } c / N) + 1)$

This is totally irrelevant, but when I was in Iraq, one of the army generals—who shall remain nameless. I had to brief and write situation reports for was absolutely adamant that adverbs were not to be used, ever, or there would be wrath. The idea was that you can't quantify words such as highly or mostly because they mean different things to different people. Five years later, I still scour my business e-mails and PowerPoint presentations for unnecessary adverbs. Formality writ large!

Diversity, as it relates to text mining, refers to the number of different words used in relation to the total number of words used. This can also mean the expanse of the text producer's vocabulary or lexicon richness. The `qdap` package provides five—that's right, five—different measures of diversity: Simpson, Shannon, Collision, Bergen Parker, and Brillouin. I won't cover these five in detail but will only say that the algorithms are used not only for communication and information science retrieval, but also for biodiversity in nature.

Finally, dispersion, or lexical dispersion, is a useful tool in order to understand how words are spread throughout a document and serves as an excellent way to explore a text and identify patterns. The analysis is conducted by calling the specific word or words of interest, which are then produced in a plot showing when the word or words occurred in the text over time. As we will see, the `qdap` package has a built-in plotting function to analyze the text dispersion.

We covered a framework on text mining about how to prepare the text, count words, and create topic models and, finally, dived deep into other lexical measures. Now, let's apply all this and do some real-world text mining.

Business understanding

For this case study, we will take a look at president Obama's State of the Union speeches. I have no agenda here; just curious as to what can be uncovered in particular and if and how his message changed over time. Perhaps this will serve as a blueprint to analyze any politician's speech in order to prepare an opposing candidate in a debate or speech of their own. If not, so be it.

The two main analytical goals are to build topic models on the six State of the Union speeches and then compare the first speech in 2010 and the last in January, 2016 for sentence-based textual measures, such as sentiment and dispersion.

Data understanding and preparation

The primary package that we will use is `tm`, the text mining package. We will also need `SnowballC` for the stemming of the words, `RColorBrewer` for the color palettes in wordclouds, and the `wordcloud` package. Please ensure that you have these packages installed before attempting to load them:

```
> library(tm)

> library(wordcloud)

> library(RColorBrewer)
```

The data files are available for download in <https://github.com/datameister66/data>. Please ensure you put the text files into a separate directory because it will all go into our corpus for analysis.

Download the seven `.txt` files, for example `sou2012.txt`, into your working R directory. You can identify your current working directory and set it with these functions:

```
> getwd()

> setwd("../data")
```

We can now begin to create the corpus by first creating an object with the path to the speeches and then seeing how many files are in this directory and what they are named:

```
> name <- file.path("../text")

> length(dir(name))
[1] 7

> dir(name)
[1] "sou2010.txt" "sou2011.txt" "sou2012.txt" "sou2013.txt"
[5] "sou2014.txt" "sou2015.txt" "sou2016.txt"
```

We will name our corpus `docs` and create it with the `Corpus()` function, wrapped around the directory source function, `DirSource()`, which is also part of the `tm` package:

```
> docs <- Corpus(DirSource(name))

> docs
<<VCorpus>>
Metadata: corpus specific: 0, document level (indexed): 0
Content: documents: 7
```



Note that there is no `corpus` or `document level` metadata. There are functions in the `tm` package to apply things such as author's names and timestamp information, among others, at both `document level` and `corpus`. We will not utilize this for our purposes.

We can now begin the text transformations using the `tm_map()` function from the `tm` package. These will be the transformations that we discussed previously--lowercase letters, remove numbers, remove punctuation, remove stop words, strip out the whitespace, and stem the words:

```
> docs <- tm_map(docs, tolower)

> docs <- tm_map(docs, removeNumbers)

> docs <- tm_map(docs, removePunctuation)

> docs <- tm_map(docs, removeWords, stopwords("english"))

> docs <- tm_map(docs, stripWhitespace)
```

At this point, it is a good idea to eliminate unnecessary words. For example, during the speeches, when Congress applauds a statement, you will find (Applause) in the text. This must be removed:

```
> docs <- tm_map(docs, removeWords, c("applause", "can", "cant",  
  "will",  
  "that", "weve", "dont", "wont", "youll", "youre"))
```

After completing the transformations and removal of other words, make sure that your documents are plain text, put it in a document-term matrix, and check the dimensions:

```
> docs = tm_map(docs, PlainTextDocument)  
  
> dtm = DocumentTermMatrix(docs)  
  
> dim(dtm)  
[1]    7 4738
```

The six speeches contain 4738 words. It is optional, but one can remove the sparse terms with the `removeSparseTerms()` function. You will need to specify a number between zero and one where the higher the number, the higher the percentage of sparsity in the matrix. Sparsity is the relative frequency of a term in the documents. So, if your sparsity threshold is 0.75, only terms with sparsity greater than 0.75 are removed. For us that would be $(1 - 0.75) * 7$, which is equal to 1.75. Therefore, any term in fewer than two documents would be removed:

```
> dtm <- removeSparseTerms(dtm, 0.75)  
  
> dim(dtm)  
[1]    7 2254
```

As we don't have the metadata on the documents, it is important to name the rows of the matrix so that we know which document is which:

```
> rownames(dtm) <- c("2010", "2011", "2012", "2013", "2014",  
  "2015", "2016")
```

Using the `inspect()` function, you can examine the matrix. Here, we will look at the seven rows and the first five columns:

```
> inspect(dtm[1:7, 1:5])
```

	Terms				
Docs	abandon	ability	able	abroad	absolutely
2010	0	1	1	2	2
2011	1	0	4	3	0
2012	0	0	3	1	1
2013	0	3	3	2	1
2014	0	0	1	4	0
2015	1	0	1	1	0
2016	0	0	1	0	0

It appears that our data is ready for analysis, starting with looking at the word frequency counts. Let me point out that the output demonstrates why I've been trained to not favor wholesale stemming. You may be thinking that 'ability' and 'able' could be combined. If you stemmed the document you would end up with 'abl'. How does that help the analysis? I think you lose context, at least in the initial analysis. Again, I recommend applying stemming thoughtfully and judiciously.

Modeling and evaluation

Modeling will be broken into two distinct parts. The first will focus on word frequency and correlation and culminate in the building of a topic model. In the next portion, we will examine many different quantitative techniques by utilizing the power of the `qdap` package in order to compare two different speeches.

Word frequency and topic models

As we have everything set up in the document-term matrix, we can move on to exploring word frequencies by creating an object with the column sums, sorted in descending order. It is necessary to use `as.matrix()` in the code to sum the columns. The default order is ascending, so putting `-` in front of `freq` will change it to descending:

```
> freq <- colSums(as.matrix(dtm))  
  
> ord <- order(-freq)
```

We will examine the `head` and `tail` of the object with the following code:

```
> freq[head(ord)]
new  america  people  jobs  now  years
    193     174    168   163   157   148

> freq[tail(ord)]
wright written yearold youngest youngstown zero
         2         2         2         2         2         2
```

The most frequent word is `new` and, as you might expect, the president mentions `america` frequently. Also notice how important employment is with the frequency of `jobs`. I find it interesting that he mentions Youngstown, for Youngstown, OH, a couple of times.

To look at the frequency of the word frequency, you can create tables, as follows:

```
> head(table(freq))
freq
  2   3   4   5   6   7
596 354 230 141 137 89

> tail(table(freq))
freq
148 157 163 168 174 193
  1   1   1   1   1   1
```

What these tables show is the number of words with that specific frequency. So 354 words occurred three times; and one word, `new` in our case, occurred 193 times.

Using `findFreqTerms()`, we can see which words occurred at least 125 times:

```
> findFreqTerms(dtm, 125)
[1] "america" "american" "americans" "jobs" "make" "new"
[7] "now"     "people"   "work"     "year" "years"
```

You can find associations with words by correlation with the `findAssocs()` function. Let's look at `jobs` as two examples using 0.85 as the correlation cutoff:

```
> findAssocs(dtm, "jobs", corlimit = 0.85)
$jobs
colleges serve market shouldnt defense put tax came
    0.97  0.91  0.89      0.88    0.87 0.87 0.87 0.86
```

For visual portrayal, we can produce `wordclouds` and a bar chart. We will do two `wordclouds` to show the different ways to produce them: one with a minimum frequency and the other by specifying the maximum number of words to include. The first one with minimum frequency, also includes code to specify the color. The scale syntax determines the minimum and maximum word size by frequency; in this case, the minimum frequency is 70:

```
> wordcloud(names(freq), freq, min.freq = 70, scale = c(3, .5),  
  colors = brewer.pal(6, "Dark2"))
```

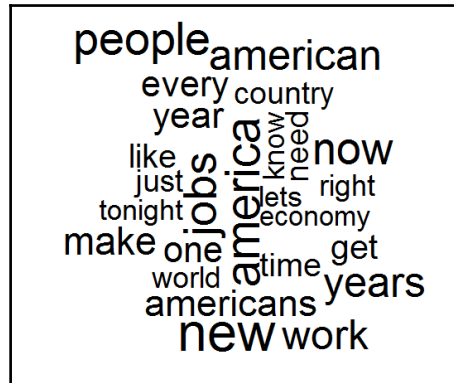
The output of the preceding command is as follows:



One can forgo all the fancy graphics, as we will in the following image, capturing the 25 most frequent words:

```
> wordcloud(names(freq), freq, max.words = 25)
```

The output of the preceding command is as follows:



To produce a bar chart, the code can get a bit complicated, whether you use base R, `ggplot2`, or `lattice`. The following code will show you how to produce a bar chart for the 10 most frequent words in base R:

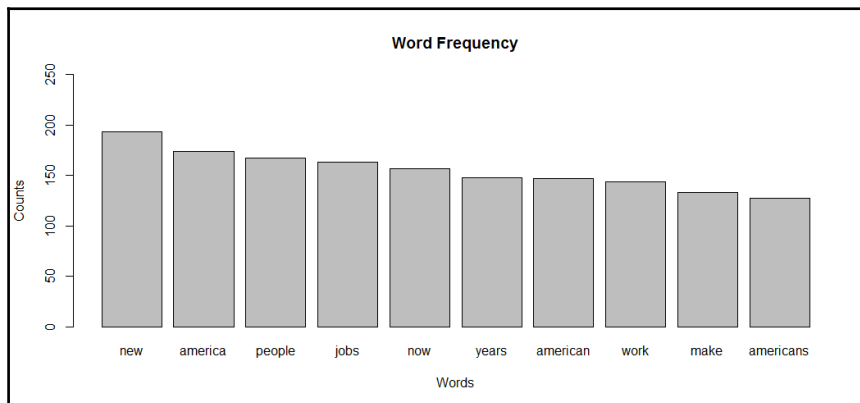
```
> freq <- sort(colSums(as.matrix(dtm)), decreasing = TRUE)

> wf <- data.frame(word = names(freq), freq = freq)

> wf <- wf[1:10, ]

> barplot(wf$freq, names = wf$word, main = "Word Frequency",
  xlab = "Words", ylab = "Counts", ylim = c(0, 250))
```

The output of the preceding command is as follows:



We will now move on to building topic models using the `topicmodels` package, which offers the `LDA()` function. The question now is how many topics to create. It seems logical to solve for three topics ($k=3$). Certainly, I encourage you to try other numbers of topics:

```
> library(topicmodels)

> set.seed(123)

> lda3 <- LDA(dtm, k = 3, method = "Gibbs")

> topics(lda3)
2010 2011 2012 2013 2014 2015 2016
     2     1     1     1     3     3     2
```

We can see an interesting transition over time. The first and last addresses have the same topic grouping, almost as if he opened and closed his tenure with the same themes.

Using the `terms()` function produces a list of an ordered word frequency for each topic. The list of words is specified in the function, so let's look at the top 20 per topic:

```
> terms(lda3, 25)
      Topic 1      Topic 2      Topic 3
[1,] "jobs"      "people"    "america"
[2,] "now"       "one"      "new"
[3,] "get"       "work"     "every"
[4,] "tonight"   "just"     "years"
[5,] "last"      "year"     "like"
[6,] "energy"    "know"     "make"
[7,] "tax"       "economy"  "time"
[8,] "right"     "americans" "need"
[9,] "also"      "businesses" "american"
[10,] "government" "even"     "world"
[11,] "home"     "give"     "help"
[12,] "well"     "many"     "lets"
[13,] "american" "security" "want"
[14,] "two"      "better"   "states"
[15,] "congress" "come"     "first"
[16,] "country"  "still"    "country"
[17,] "reform"   "workers"  "together"
[18,] "must"     "change"   "keep"
[19,] "deficit"  "take"     "back"
[20,] "support"  "health"   "americans"
[21,] "business" "care"     "way"
[22,] "education" "families" "hard"
[23,] "companies" "made"     "today"
[24,] "million"  "future"   "working"
[25,] "nation"   "small"    "good"
```

Topic 2 covers the first and last speeches. Nothing really stands out as compelling in that topic like the others. It will be interesting to see how the next analysis can yield insights into those speeches.

Topic 1 covers the next three speeches. Here, the message transitions to "jobs", "energy", "reform", and the "deficit", not to mention the comments about "education" and as we saw above, the correlation of "jobs" and "colleges".

Topic 3 brings us to the next two speeches. The focus seems to really shift on to the economy and business with mentions to "security" and healthcare.

In the next section, we can dig into the exact speech content further, along with comparing and contrasting the first and last State of the Union addresses.

Additional quantitative analysis

This portion of the analysis will focus on the power of the `qdap` package. It allows you to compare multiple documents over a wide array of measures. Our effort will be on comparing the 2010 and 2016 speeches. For starters, we will need to turn the text into data frames, perform sentence splitting, and then combine them to one data frame with a variable created that specifies the year of the speech. We will use this as our grouping variable in the analyses. Dealing with text data, even in R, can be tricky. The code that follows seemed to work the best in this case to get the data loaded and ready for analysis. We first load the `qdap` package. Then, to bring in the data from a text file, we will use the `readLines()` function from base R, collapsing the results to eliminate unnecessary whitespace. I also recommend putting your text encoding to ASCII, otherwise you may run into some bizarre text that will mess up your analysis. That is done with the `iconv()` function:

```
> library(qdap)

> speech16 <- paste(readLines("sou2016.txt"), collapse=" ")
Warning message:
In readLines("sou2016.txt") : incomplete final line found on
'sou2016.txt'

> speech16 <- iconv(speech16, "latin1", "ASCII", "")
```

The warning message is not an issue as it is just telling us that the final line of text is not the same length as the other lines in the `.txt` file. We now apply the `qprep()` function from `qdap`.

This function is a wrapper for a number of other replacement functions and using it will speed pre-processing, but it should be used with caution if more detailed analysis is required. The functions it passes through are as follows:

- `bracketX()`: apply bracket removal
- `replace_abbreviation()`: replaces abbreviations
- `replace_number()`: numbers to words, for example '100' becomes 'one hundred'
- `replace_symbol()`: symbols become words, for example '@' becomes 'at'

```
> prep16 <- qprep(speech16)
```

The other pre-processing we should do is to replace contractions (can't to cannot), remove stopwords, in our case the top 100, and remove unwanted characters, with the exception of periods and question marks. They will come in handy shortly:

```
> prep16 <- replace_contraction(prepare16)

> prep16 <- rm_stopwords(prepare16, Top100Words, separate = F)

> prep16 <- strip(prepare16, char.keep = c("?", "."))
```

Critical to this analysis is to now split it into sentences and add what will be the grouping variable, the year of the speech. This also creates the `tot` variable, which stands for Turn of Talk, serving as an indicator of sentence order. This is especially helpful in a situation where you are analyzing dialogue, say in a debate or question and answer session:

```
> sent16 <- data.frame(speech = prepare16)

> sent16 <- sentSplit(sent16, "speech")

> sent16$year <- "2016"
```

Repeat the steps for the 2010 speech:

```
> speech10 <- paste(readLines("sou2010.txt"), collapse=" ")

> speech10 <- iconv(speech10, "latin1", "ASCII", "")

> speech10 <- gsub("(Applause.)", "", speech10)

> prep10 <- qprep(speech10)

> prep10 <- replace_contraction(prepare10)

> prep10 <- rm_stopwords(prepare10, Top100Words, separate = F)
```

```
> prep10 <- strip(prepare10, char.keep = c("?", "."))  
  
> sent10 <- data.frame(speech = prep10)  
> sent10 <- sentSplit(sent10, "speech")  
  
> sent10$year <- "2010"
```

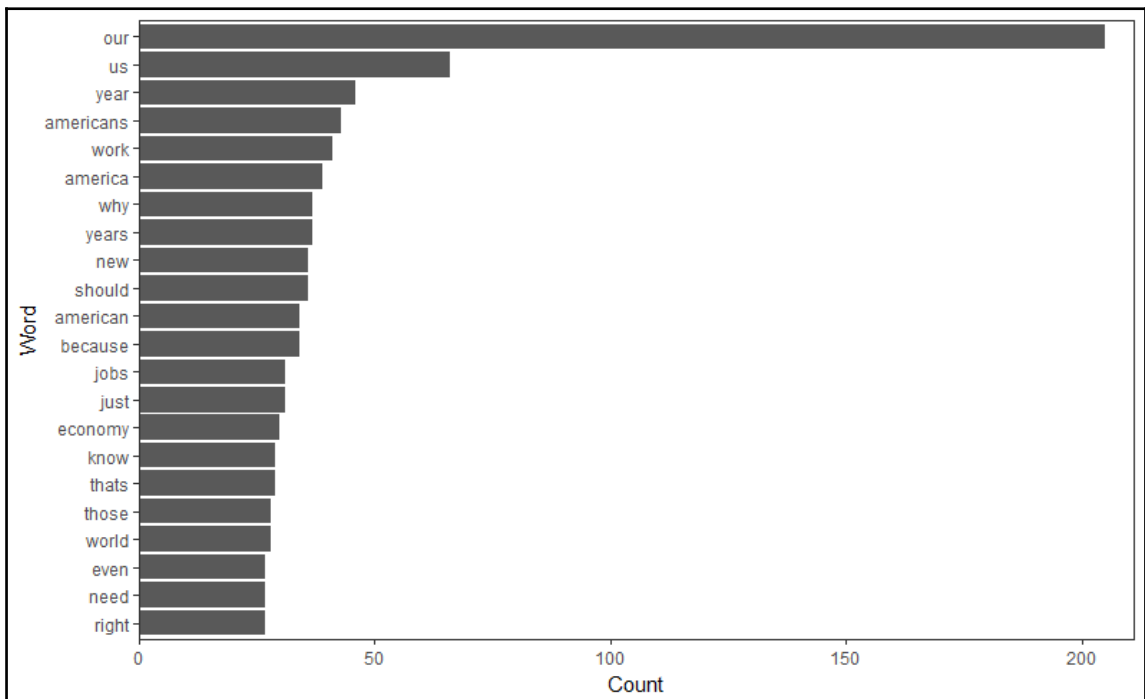
Concatenate the separate years into one dataframe:

```
> sentences <- data.frame(rbind(sent10, sent16))
```

One of the great things about the `qdap` package is that it facilitates basic text exploration, as we did before. Let's see a plot of frequent terms:

```
> plot(freq_terms(sentences$speech))
```

The output of the preceding command is as follows:



You can create a word frequency matrix that provides the counts for each word by speech:

```
> wordMat <- wfm(sentences$speech, sentences$year)

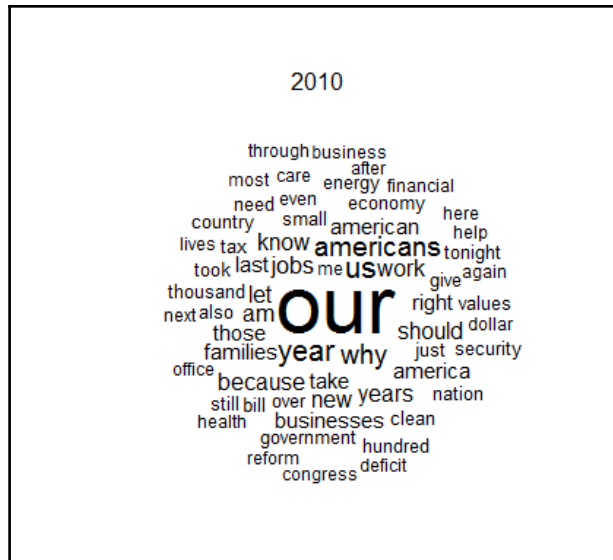
> head(wordMat[order(wordMat[, 1], wordMat[, 2], decreasing =
  TRUE),])
```

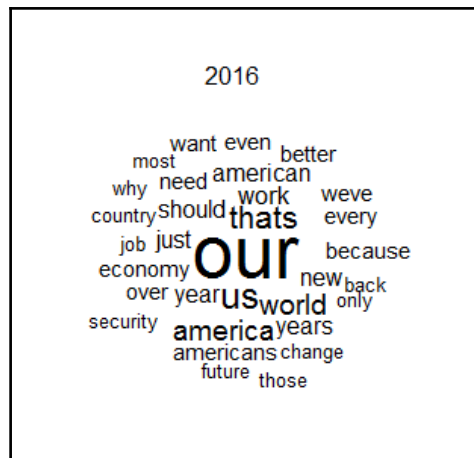
	2010	2016
our	120	85
us	33	33
year	29	17
americans	28	15
why	27	10
jobs	23	8

This can also be converted into a document-term matrix with the function `as.dtm()` should you so desire. Let's next build `wordclouds`, by year with `qdap` functionality:

```
> trans_cloud(sentences$speech, sentences$year, min.freq = 10)
```

The preceding command produces the following two images:



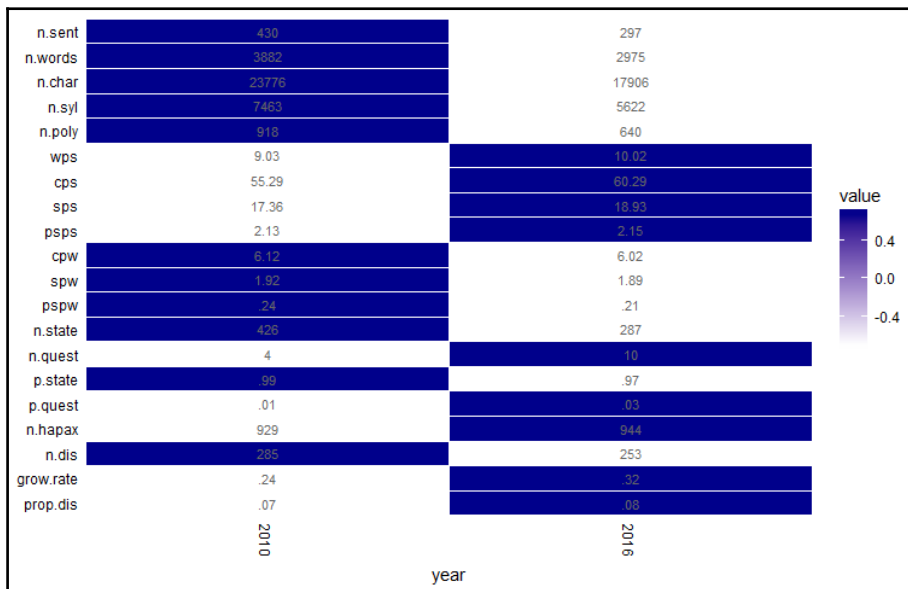


Comprehensive word statistics are available. Here is a plot of the stats available in the package. The plot loses some of its visual appeal with just two speeches, but is revealing nonetheless. A complete explanation of the stats is available under `?word_stats`:

```
> ws <- word_stats(sentences$speech, sentences$year, rm.incomplete = T)

> plot(ws, label = T, lab.digits = 2)
```

The output of the preceding command is as follows:



Notice that the 2016 speech was much shorter, with over a hundred fewer sentences and almost a thousand fewer words. Also, there seems to be the use of asking questions as a rhetorical device in 2016 versus 2010 (n.quest 10 versus n.quest 4).

To compare the polarity (sentiment scores), use the `polarity()` function, specifying the text and grouping variables:

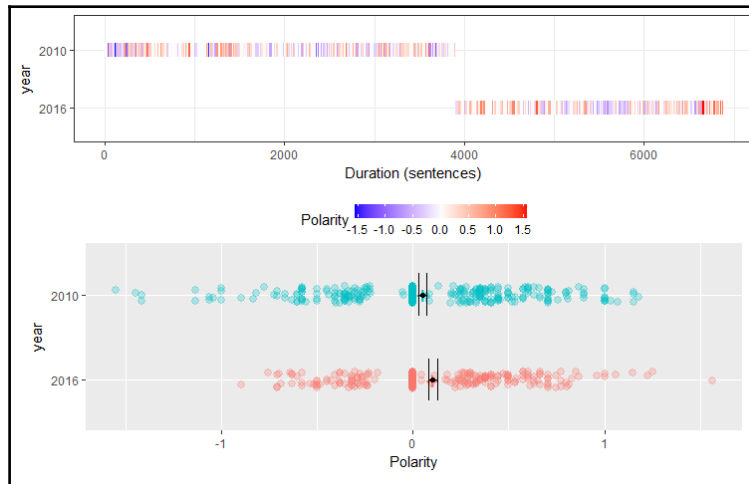
```
> pol = polarity(sentences$speech, sentences$year)

> pol
  year total.sentences total.words ave.polarity sd.polarity
 stan.mean.polarity
1 2010             435       3900         0.052         0.432
 0.121
2 2016             299       2982         0.105         0.395
 0.267
```

The `stan.mean.polarity` value represents the standardized mean polarity, which is the average polarity divided by the standard deviation. We see that 2015 was slightly higher (0.267) than 2010 (0.121). This is in line with what we would expect, wanting to end on a more positive note. You can also plot the data. The plot produces two charts. The first shows the polarity by sentences over time and the second shows the distribution of the polarity:

```
> plot(pol)
```

The output of the preceding command is as follows:



This plot may be a challenge to read in this text, but let me do my best to interpret it. The 2010 speech starts out with a strong negative sentiment and is slightly more negative than 2016. We can identify the most negative sentiment sentence by creating a dataframe of the `pol` object, find the sentence number, and produce it:

```
> pol.df <- pol$all

> which.min(pol.df$polarity)
[1] 12

> pol.df$text.var[12]

[1] "One year ago, I took office amid two wars, an economy rocked
    by a severe recession, a financial system on the verge of
    collapse, and a government deeply in debt."
```

Now that is negative sentiment! Ironically, the government is even more in debt today. We will look at the readability index next:

```
> ari <- automated_readability_index(sentences$speech,
    sentences$year)

> ari$Readability
  year word.count sentence.count character.count
1 2010      3900           435         23859
2 2016      2982           299         17957
Automated_Readability_Index
1              11.86709
2              11.91929
```

I think it is no surprise that they are basically the same. Formality analysis is next. This takes a couple of minutes to run in R:

```
> form <- formality(sentences$speech, sentences$year)

> form
  year word.count formality
1 2016      2983      65.61
2 2010      3900      63.88
```


This looks to be very similar. We can examine the proportion of the parts of the speech. A plot is available, but adds nothing to the analysis, in this instance:

```
> form$form.prop.by
  year word.count  noun   adj  prep articles pronoun
1 2010      3900 44.18 15.95  3.67      0      4.51
2 2016      2982 43.46 17.37  4.49      0      4.96
  verb adverb interj other
1 23.49   7.77   0.05  0.38
2 21.73   7.41   0.00  0.57
```

Now, the diversity measures are produced. Again, they are nearly identical. A plot is also available, (`plot(div)`), but being so similar, it once again adds no value. It is important to note that Obama's speech writer for 2010 was Jon Favreau, and in 2016, it was Cody Keenan:

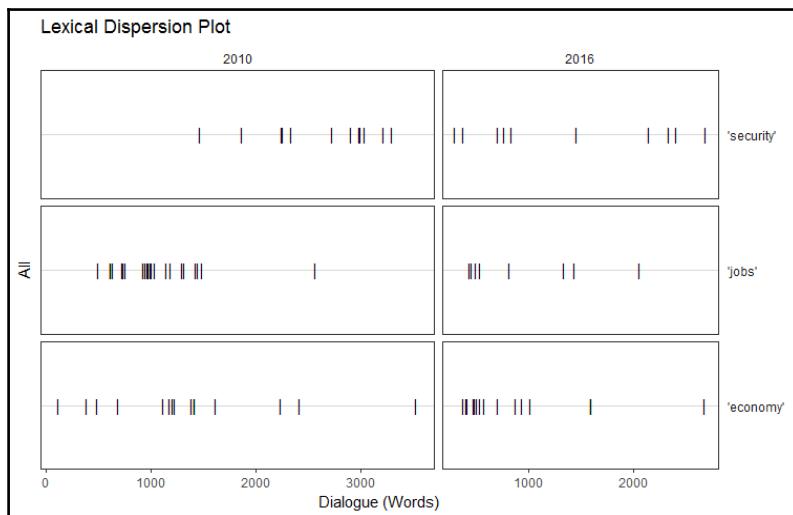
```
> div <- diversity(sentences$speech, sentences$year)

> div
  year   wc simpson shannon collision berger_parker brillouin
1 2010 3900   0.998   6.825    5.970         0.031    6.326
2 2015 2982   0.998   6.824    6.008         0.029    6.248
```

One of my favorite plots is the dispersion plot. This shows the dispersion of a word throughout the text. Let's examine the dispersion of "jobs", "families", and "economy":

```
> dispersion_plot(sentences$speech,
  rm.vars = sentences$year,
  c("security", "jobs", "economy"),
  color = "black", bg.color = "white")
```

The output of the preceding command is as follows:



This is quite interesting as you can visualize how much longer the 2010 speech is. In 2010, the first half of his speech was focused heavily on jobs while in 2016 it appears it was more about the state of the overall economy; no doubt how much of a hand he played in saving it from the brink of disaster. In 2010, security was not brought in until later in the speech versus placed throughout the final address. You can see and understand how text analysis can provide insight into what someone is thinking, what their priorities are, and how they go about communicating them.

This completes our analysis of the two speeches. I must confess that I did not listen to any of these speeches. In fact, I haven't watched a State of the Union address since Reagan was president, probably with the exception of the 2002 address. This provided some insight for me on how the topics and speech formats have changed over time to accommodate political necessity, while the overall style of formality and sentence structure has remained consistent. Keep in mind that this code can be adapted to text for dozens, if not hundreds, of documents and with multiple speakers, for example screenplays, legal proceedings, interviews, social media, and on and on. Indeed, text mining can bring quantitative order to what has been qualitative chaos.

Summary

In this chapter, we looked at how to address the massive volume of textual data that exists through text mining methods. We looked at a useful framework for text mining, including preparation, word frequency counts and visualization, and topic models using LDA with the `tm` package. Included in this framework were other quantitative techniques, such as polarity and formality, in order to provide a deeper lexical understanding, or what one could call style, with the `qdap` package. The framework was then applied to president Obama's seven State of the Union addresses, which showed that, although the speeches had a similar style, the core messages changed over time as the political landscape changed. Despite it not being practical to cover every possible text mining technique, those discussed in this chapter should be adequate for most problems that one might face. In the next chapter, we are going to shift gears away from building models and focus on a technique to get R on the cloud, allowing you to scale your machine learning to whatever problem you may be trying to solve.

14

R on the Cloud

"If someone asks me what cloud computing is, I try not to get bogged down with definitions. I tell them that, simply put, cloud computing is a better way to run your business."

- Marc Benioff, CEO, Salesforce.com

Since I'm not a CEO of a company trying to profit from the cloud, let's get bogged down with a definition. I like the one put forward by Microsoft™ here--<https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/>.

Simply put, cloud computing is the delivery of computing services--servers, storage, databases, networking, software, analytics, and more--over the Internet ("the cloud"). Companies offering these computing services are called cloud providers and typically charge for cloud computing services based on usage, similar to how you're billed for water or electricity at home.

If you are not using the cloud for machine learning, well I guarantee that at some point in the not-too-distant future, you will. I still know some people who fear the idea of losing control of their data, security issues, and so on. However, as one start-up CEO put it to me, I like to ask them if they access their supposedly secure data on a laptop via WiFi, and when they reply sure, they are telling me they are on the cloud and it is just a matter of where the hardware is stored.

There you have it. Do you want your office dungeon to have rows of servers or do you want to let someone else handle that problem with their secure, redundant, and discrete global infrastructure?

Using cloud-based computing with R can facilitate the seamless work across multiple locations and also provide you with tremendous computing power, which can be quickly scaled up or down as needed. This can be a significant cost saving.

There are many ways to get R on the cloud, but I am going to use **Amazon Web Services (AWS)** and their **Elastic Compute Cloud (EC2)** for this demonstration because it is what I first learned and it is what I'm familiar using. That is not to say I endorse it over other products. I don't and won't, unless Jeff Bezos selects me for a manned space mission, then my attitude shall change.

At any rate, the goal here is to get you up and running with R and RStudio on the cloud quickly and without having to write a line of Linux code. Now, to maximize the power of AWS and its bewildering array of tools, you can learn how to apply Linux code through a **Secure Shell (SSH)**. To do this, we will create and launch a virtual computer known as an **instance**. We will then login to RStudio via a web browser and cover some of the functionality. There are many tutorials out on the web about how to do this, but my goal is to get you started in the simplest and quickest manner possible, and to get you using R on the cloud **TODAY**.

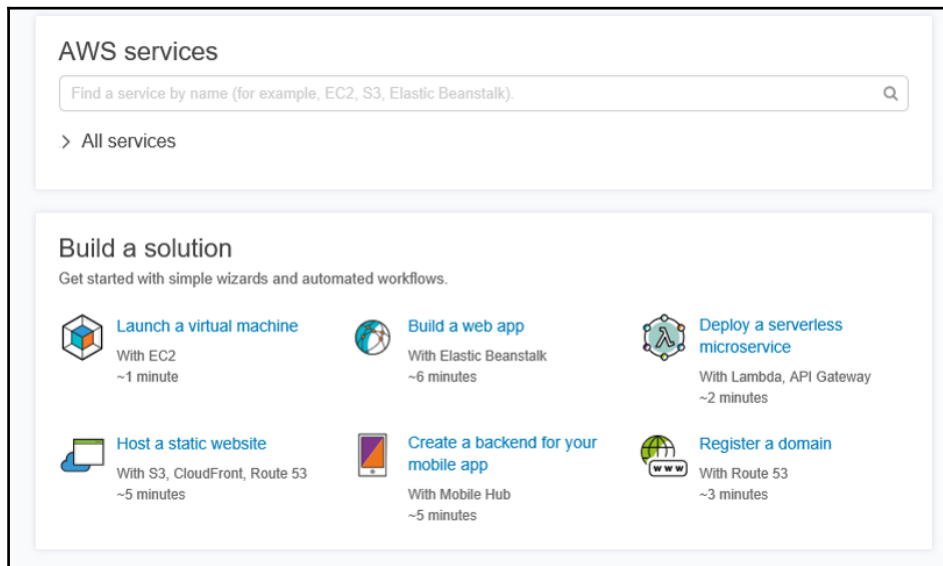
Creating an Amazon Web Services account

The first thing to do is to sign up for an AWS account:

<https://aws.amazon.com/>

This is the only prerequisite for this exercise. The process requires a credit card, but what we will be doing here will not cost a cent as it is done on a free instance. Going forward, you can quickly launch a new instance with greater computing power as needed, then stop or terminate the instance when finished. As you create your account and login, you can choose to create security groups or not. I will demonstrate it by creating a new security group during instance creation. A security group allows you to control who can and how they can access the instance. Also, don't worry about creating a **Pair Key** at this point unless you so desire. We will create that as well.

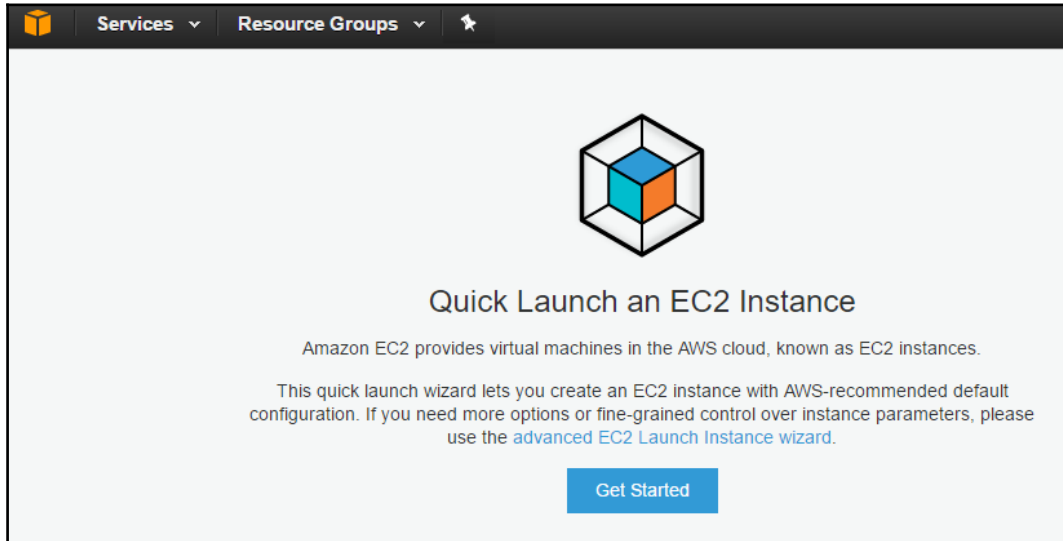
Once that is completed, log in to your AWS console, which should give you a webpage looking like this:



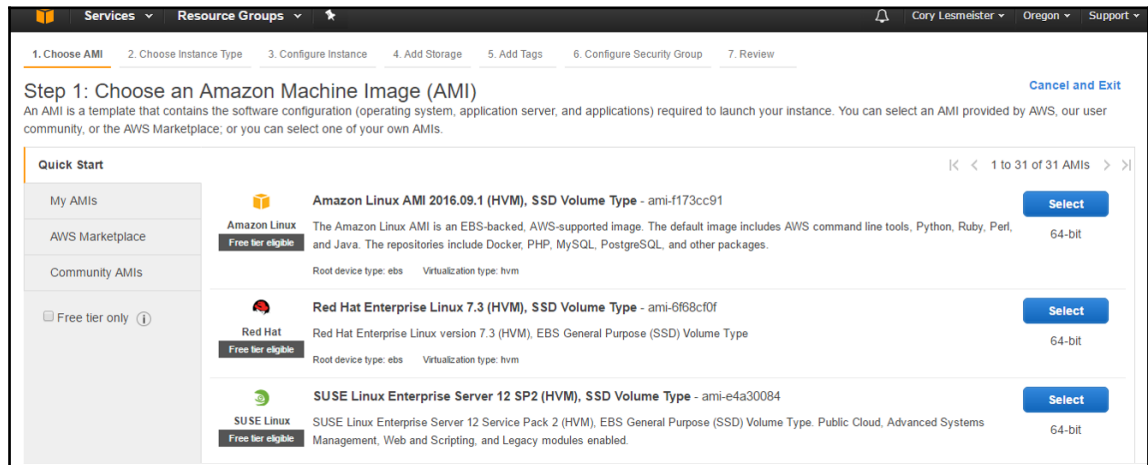
If you are here, it is now time to create and launch a virtual machine with a simple click on the cleverly named hyperlink **Launch a virtual machine**.

Launch a virtual machine

The hyperlink to launch a virtual machine will take you to this page:



Avoid the **Get Started** button and click on **advanced EC2 Launch Instance Wizard**, which takes you to this page:



As you gain experience, you can use the various **Amazon Machine Images (AMI)** and customize how you use R on AWS. However, our goals here are quick and simple. With that in mind, there are several community AMIs created by AWS users that incorporate R and RStudio already. So, under **Quick Start**, click **Community AMIs**. A search box will pop up and I recommend to start using the AMI maintained by Louis Aslett, http://www.louisaslett.com/RStudio_AMI/. This AMI will be displayed by searching for **rstudio aslett**. That will bring the page up, so click on the **Select** button, as shown here:

Step 1: Choose an Amazon Machine Image (AMI) Cancel and Exit

My AMIs
AWS Marketplace
Community AMIs
Operating system

 4 results for "rstudio aslett" on AWS Marketplace
Partner software pre-configured to run on AWS

 **RStudio-0.99.491_R-3.2.3_ubuntu-14.04-LTS-64bit - ami-1d7f657c**
Ready to run RStudio server for statistical computation (www.louisaslett.com). Connect to instance public DNS in web browser (standard port 80), username rstudio and password rstudio

Select
64-bit

This takes you to **Step 2**, where you select the instance type. I've picked the `t2.micro` free tier:

1. Choose AMI 2. Choose Instance Type 3. Configure Instance 4. Add Storage 5. Add Tags 6. Configure Security Group 7. Review

Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. [Learn more](#) about instance types and how they can meet your computing needs.

Filter by: All instance types Current generation [Show/Hide Columns](#)

Currently selected: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only)

	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance	IPv6 Support
☐	General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate	Yes
☒	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.small	1	2	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.large	2	8	EBS only	-	Low to Moderate	Yes

Cancel
Previous
Review and Launch
Next: Configure Instance Details

Once you've picked the instance type you desire, hit **Review and Launch**. Since this is an existing AMI, you can jump to **Step 7**, the **Review tab**. You can launch from here, but let's click on **Step 6, Configure Security Group**:

1. Choose AMI 2. Choose Instance Type 3. Configure Instance 4. Add Storage 5. Add Tags 6. Configure Security Group 7. Review

Step 7: Review Instance Launch

Please review your instance launch details. You can go back to edit changes for each section. Click **Launch** to assign a key pair to your instance and complete the launch process.

Improve your instances' security. Your security group, launch-wizard-2, is open to the world.
Your instances may be accessible from any IP address. We recommend that you update your security group rules to allow access from known IP addresses only.
You can also open additional ports in your security group to facilitate access to the application or service you're running, e.g., HTTP (80) for web servers. [Edit security groups](#)

▼ AMI Details

RStudio-0.99.491_R-3.2.3_ubuntu-14.04-LTS-64bit - ami-1d7f657c
Ready to run RStudio server for statistical computation (www.louisaslett.com). Connect to instance public DNS in web browser (standard port 80), username rstudio and password rstudio
Root Device Type: ebs Virtualization type: hvm

This will bring you to the step in the process where you can either create a security group or use an existing one. Here is an example of creating a **new security group**:

1. Choose AMI 2. Choose Instance Type 3. Configure Instance 4. Add Storage 5. Add Tags 6. Configure Security Group 7. Review

Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. [Learn more](#) about Amazon EC2 security groups.

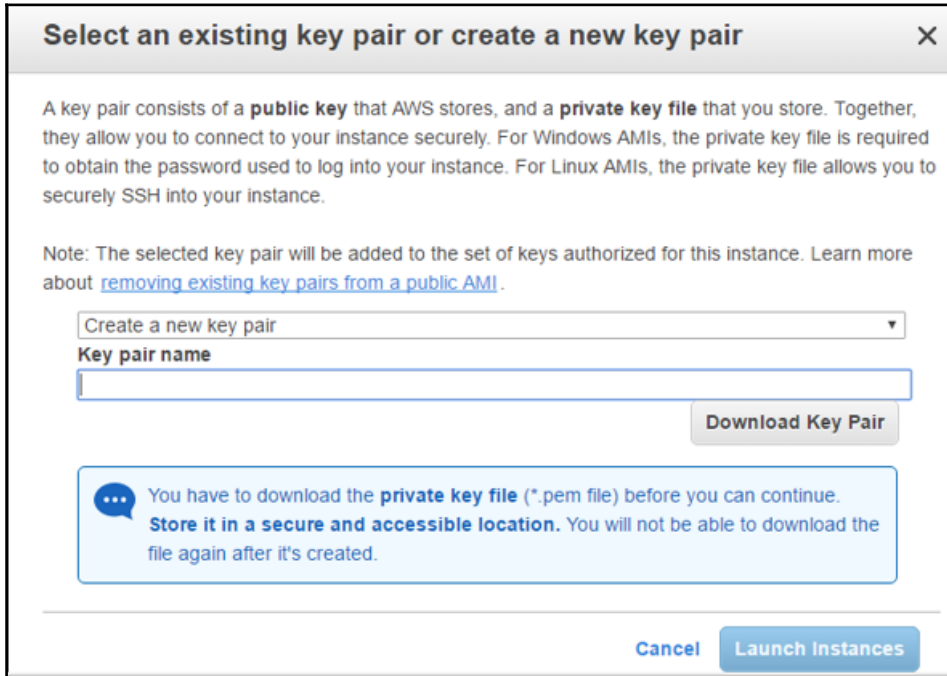
Assign a security group: ☒ Create a new security group ☐ Select an existing security group

Security group name:

Description:

Type	Protocol	Port Range	Source
All traffic	All	0 - 65535	My IP 24.214.32.76/32
Custom TCP Rule	TCP	8787	Anywhere 0.0.0.0/0

Once you are satisfied with that step in the process (you don't have to change a thing), click **Review and Launch**. That takes you back to **Step 7**, where you can simply click on **Launch**. This brings you to the point where you select a new or existing key pair:



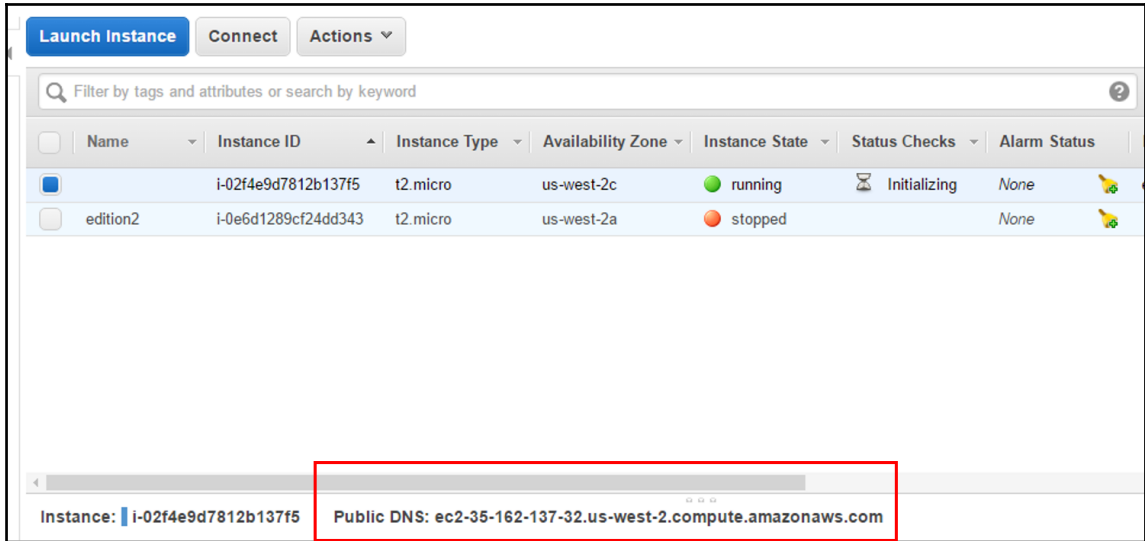
The screenshot shows a modal dialog box titled "Select an existing key pair or create a new key pair" with a close button (X) in the top right corner. The dialog contains the following elements:

- Text:** "A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance."
- Note:** "Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about [removing existing key pairs from a public AMI](#)."
- Form:** A dropdown menu with the text "Create a new key pair" and a downward arrow. Below it is a text input field labeled "Key pair name".
- Button:** A "Download Key Pair" button is located to the right of the "Key pair name" input field.
- Message Box:** A light blue box with a speech bubble icon contains the text: "You have to download the **private key file** (*.pem file) before you can continue. **Store it in a secure and accessible location.** You will not be able to download the file again after it's created."
- Footer:** At the bottom right, there are two buttons: "Cancel" and "Launch Instances".

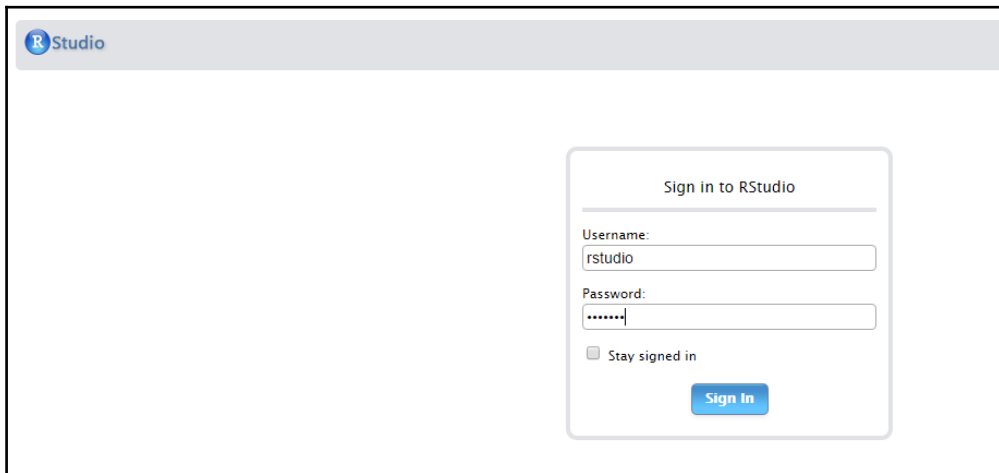
Once completed, click **Launch Instances** and proceed back to your AWS console.

Start RStudio

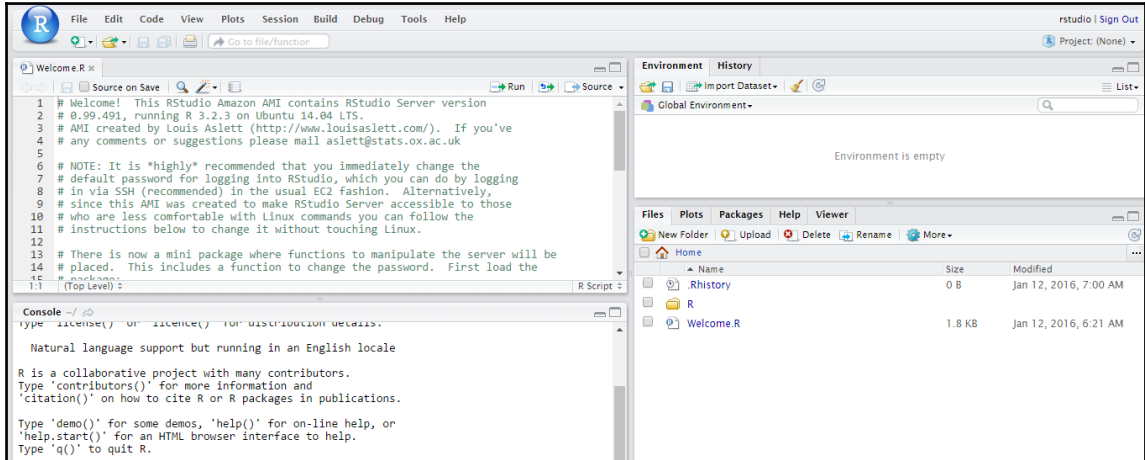
With your instance running, when you return to the AWS console and select that instance you will see something like the following:



Take note of the **Public DNS** on the selected instance. This will be all that you need to start RStudio on your web browser of choice. When you launch that in your browser, you will come to the RStudio login page. The username and password are both `rstudio`:



That's it! You are now running RStudio on a virtual machine. It should look like this:



In the upper-left panel, the **Source Panel**, are instructions on how to change the password as well as functions to link to Dropbox.

To show how to load data from the web, I will load one of the `.csv` files from github that we've used in prior chapters. Let's try `climate.csv` shall we? The first thing is to install and load the `RCurl` package:

```
> install.packages("RCurl")  
  
> library(RCurl)
```

We now need to get the link to the data on github:

```
> url <-  
"https://raw.githubusercontent.com/datameister66/data/master/climate.csv"
```

Then, pull the file into RStudio:

```
> climate <- read.csv(text = getURL(url))
```

And, be sure it worked:

```
> head(climate)
  Year CO2  Temp
1 1919 806 -0.272
2 1920 932 -0.241
3 1921 803 -0.187
4 1922 845 -0.301
5 1923 970 -0.272
6 1924 963 -0.292
```

There you have it. You are now a cloud-based machine learning warrior, able to operate on the virtual machine almost exactly like you would on your own machine.



Please remember that once you are done and quit RStudio, be sure to go back to your console and stop the instance.

Summary

In this final chapter, we went through the process of quickly and simply getting R and RStudio running on the cloud. Utilizing AWS in this exercise, we covered step by step how to create a virtual machine (an instance) on the cloud, configure it, launch it and bring up RStudio on a web browser. Finally, we went over how easy it is to load data, by bringing in the `climate.csv` file from GitHub. With this introduction to cloud computing, you can now perform work anywhere you have an Internet connection, and can quickly scale the power of your instance to meet your needs. That concludes the primary chapters of the book. I hope you've enjoyed it and can implement the methods in here as well as other methods you learn over time. Thank You!

A R Fundamentals

"One of my most productive days was throwing away 1,000 lines of code."
- Ken Thompson

This chapter covers the basic programming syntax functions and capabilities of R. Its intention is to introduce you to R and accelerate your learning. The objectives are as follows:

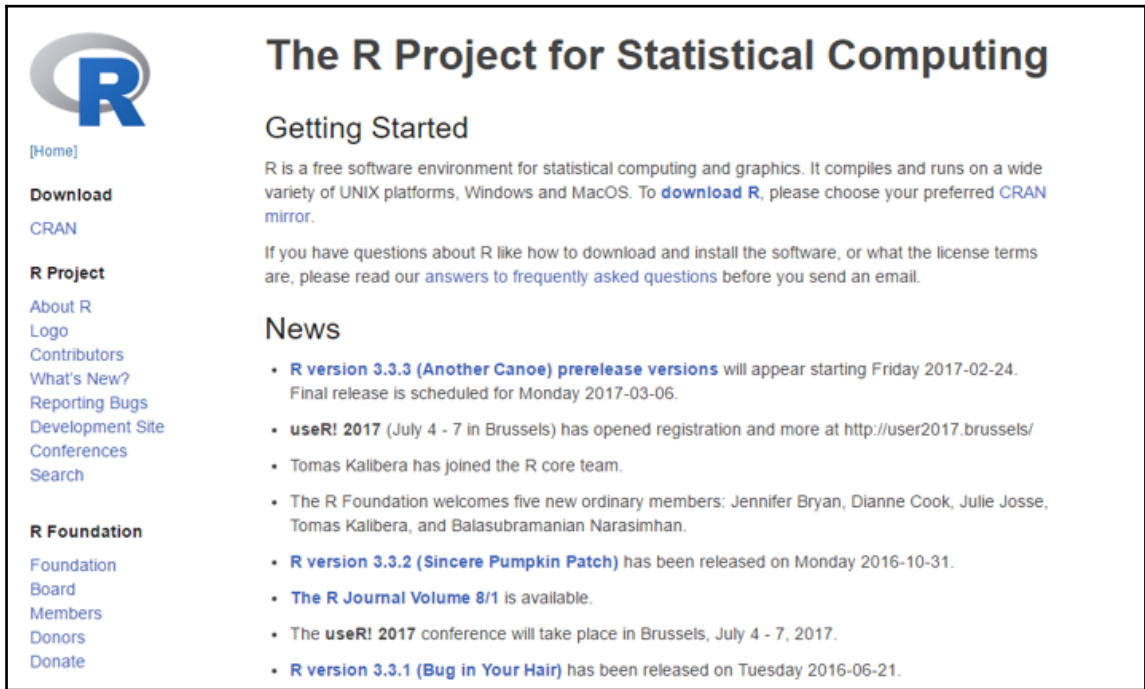
- Installing R and RStudio
- Creating and exploring vectors
- Creating data frames and matrices
- Exploring mathematical and statistical functions
- Building simple plots
- Introducing `dplyr` data manipulation
- Installing and loading packages

All of the examples in this Appendix are covered in one way or another in the preceding chapters. However, if you are completely new to R, this is a great starting point. It may accelerate your understanding of the content in the chapters.

Getting R up-and-running

We want to accomplish two things here: first, install the latest version of R and second, install RStudio, which is an **Integrated Development Environment (IDE)** for R.

Let's start by going to R's homepage at <https://www.r-project.org/>. This page will look similar to the following screenshot:



You can see that there is a link, **download R**, and in the **News** section the latest **R version is 3.3.2 (Sincere Pumpkin Patch)**, which was released on **2016-10-31**. **Version 3.3.3** is scheduled for release in March. Now, click one of the links, either **CRAN** under **Download** or **download R** under **Getting Started**, and you will come to the following screen, which has **CRAN Mirrors**:

CRAN Mirrors	
The Comprehensive R Archive Network is available at the following URLs. please choose a location close to you. Some statistics on the status of the mirrors can be found here: main page . windows release . windows old release .	
0-Cloud	
https://cloud.r-project.org/	Automatic redirection to servers worldwide, currently sponsored by Rstudio
http://cloud.r-project.org/	Automatic redirection to servers worldwide, currently sponsored by Rstudio
Algeria	
https://cran.usthb.dz/	University of Science and Technology Houari Boumediene
http://cran.usthb.dz/	University of Science and Technology Houari Boumediene
Argentina	
http://mirror.fcaglp.unlp.edu.ar/CRAN/	Universidad Nacional de La Plata
Australia	
https://cran.csiro.au/	CSIRO
http://cran.csiro.au/	CSIRO
https://cran.ms.unimelb.edu.au/	University of Melbourne
http://cran.ms.unimelb.edu.au/	University of Melbourne
https://cran.curtin.edu.au/	Curtin University of Technology
Austria	
https://cran.wu.ac.at/	Wirtschaftsuniversität Wien
http://cran.wu.ac.at/	Wirtschaftsuniversität Wien
Belgium	
http://www.freeststatistics.org/cran/	K U Leuven Association
https://lib.ugent.be/CRAN/	Ghent University Library
http://lib.ugent.be/CRAN/	Ghent University Library

These are the links, by country and sorted alphabetically, that will take you to the download page. Being in the USA, I will scroll down and find many links available:

USA	
https://cran.cnr.berkeley.edu/	University of California, Berkeley, CA
http://cran.cnr.berkeley.edu/	University of California, Berkeley, CA
http://cran.stat.ucla.edu/	University of California, Los Angeles, CA
https://mirror.las.iastate.edu/CRAN/	Iowa State University, Ames, IA
http://mirror.las.iastate.edu/CRAN/	Iowa State University, Ames, IA
https://ftp.ussg.iu.edu/CRAN/	Indiana University
http://ftp.ussg.iu.edu/CRAN/	Indiana University
https://rweb.crmda.ku.edu/cran/	University of Kansas, Lawrence, KS
http://rweb.crmda.ku.edu/cran/	University of Kansas, Lawrence, KS
https://cran.mtu.edu/	Michigan Technological University, Houghton, MI
http://cran.mtu.edu/	Michigan Technological University, Houghton, MI
http://cran.wustl.edu/	Washington University, St. Louis, MO
http://archive.linux.duke.edu/cran/	Duke University, Durham, NC
http://cran.case.edu/	Case Western Reserve University, Cleveland, OH
http://iis.stat.wright.edu/CRAN/	Wright State University, Dayton, OH

Once you find a similar link that is close to your location, click on it and you will see this as part of the page that will be loaded:

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Now, click on your appropriate operating system:

What we want now is to install base R for the first time, so click **install R for the first time** and we will come to the following page, which has the link to initiate the download:

R-3.3.2 for Windows (32/64 bit)

[Download R 3.3.2 for Windows](#) (62 megabytes, 32/64 bit)
[Installation and other instructions](#)
[New features in this version](#)

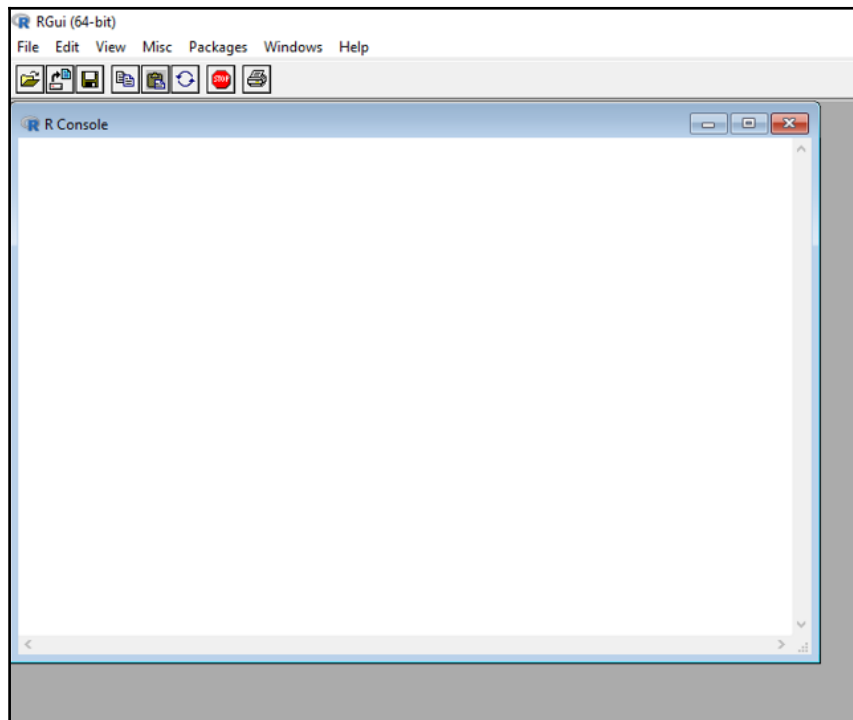
If you want to double-check that the package you have downloaded exactly matches the package distributed by R, you can compare the [md5sum](#) of the .exe to the [true fingerprint](#). You will need a version of md5sum for windows: both [graphical](#) and [command line versions](#) are available.

Frequently asked questions

- [Does R run under my version of Windows?](#)
- [How do I update packages in my previous version of R?](#)
- [Should I run 32-bit or 64-bit R?](#)

Please see the [R FAQ](#) for general information about R and the [R Windows FAQ](#) for Windows-specific information.

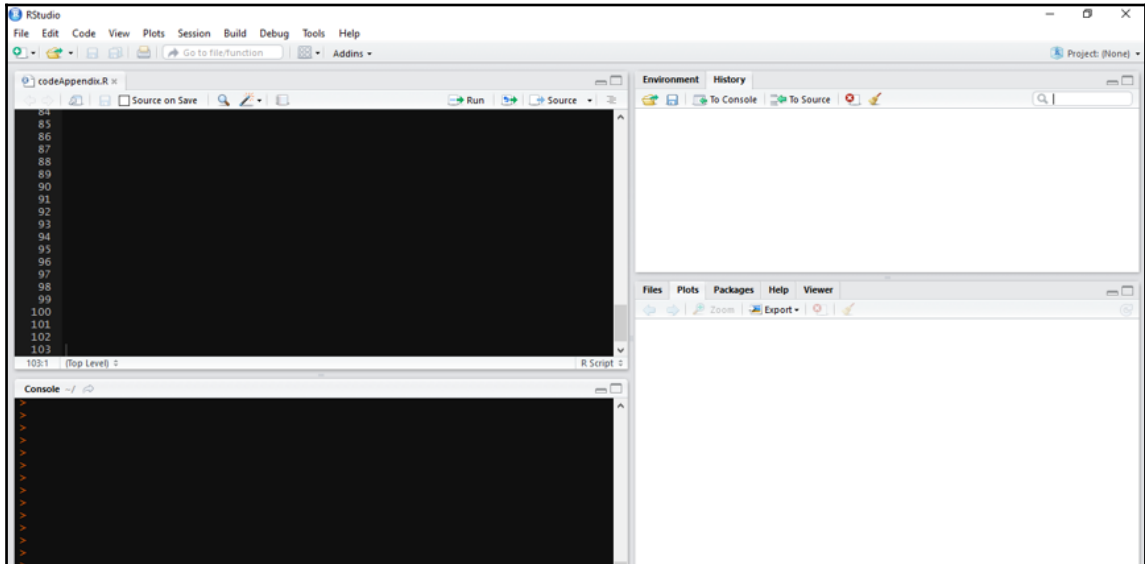
Now, you can just download and install R as any other program. After the installation, run R and you will see the base **Graphical User Interface (GUI)**:



This is all you need to run all of the code in this book. However, it is extremely helpful if you utilize R in the context of RStudio's IDE, which is available for free. This link will direct you to the page where you can download the free version:

<https://www.rstudio.com/products/RStudio/>

On the page, you will find the download for the free and commercial versions. Needless to say, let's stick with the free version, so download and install it. After this is installed and opened for the first time, you will see something as the following. Keep in mind that your screen will be different from what you see here based on the packages that I have loaded and the operating system:



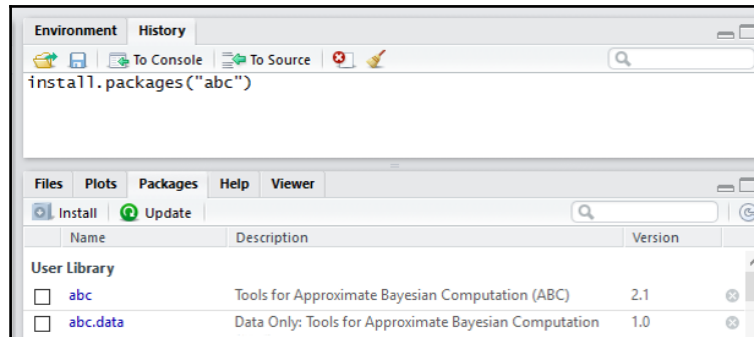
Note that on the left is the same console with the command prompt that you can see in the preceding figure. The IDE improves the experience so that you can manage **Environment** and **History** (to the upper right) and **Files**, **Plots**, **Packages**, and **Help** (to the lower right).

Let's not get distracted here with a full tutorial on what RStudio can do; instead, we'll focus on a couple of important items. One of the great benefits of R is the vast number of high-quality packages for various analyses. Let's look at how the IDE ties it all together by loading a package called `abc`, which stands for approximate Bayesian computation. Go to the command prompt and type the following:

```
> install.packages("abc")
```

After this runs, notice that in the lower right panel (ensure that the **Packages** tab is clicked on) that the `abc` package is now installed as well as the dependent `abc.data` package.

Now, go to the upper right and click on the **History** tab. You should see the command that you executed in order to load the package:



Now, if you click on the **To Console** button, what will be placed in front of the command prompt. If you click **To Source**, a new area will open and will allow you to put your project script together.

The `install.packages()` command has now gone from the history to a source file. As you experiment with your code and get it to where it works as you want it, put it into a source file. You can save it, e-mail it, and so on. All the code for each of the chapters in this book is saved as a source file.

Using R

With all systems ready to launch, let's start our first commands. R will take both the strings in quotes or simple numbers. Here, we will put one command as a string and one command as a number. The output is the same as the input:

```
> "Let's Go Sioux!"
[1] "Let's Go Sioux!"

> 15
[1] 15
```

R can also act as a calculator:

```
> ((22+5)/9)*2
[1] 6
```

Where R starts to shine is in the creation of vectors. Here, we will put the first 10 numbers of the Fibonacci sequence in a vector using the `c()` function, which stands for combining the values to a vector or list (concatenate):

```
> c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34) #Fibonacci sequence
[1] 0 1 1 2 3 5 8 13 21 34
```

Note that in this syntax, I included a comment, `Fibonacci sequence`. In R, anything after the `#` key on the command line is not executed.

Now, let's create an object that contains these numbers of the sequence. You can assign any vector or list to an object. In most R code, you will see the assign symbol as `<-`, which is read as gets. Here, we will create an object, `x`, of the Fibonacci sequence:

```
> x <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
```

To see the contents of the `x` object, just type it in the command prompt:

```
> x
[1] 0 1 1 2 3 5 8 13 21 34
```

You can select subsets of a vector using brackets after an object. This will get you the first three observations of the sequence:

```
> x[1:3]
[1] 0 1 1
```

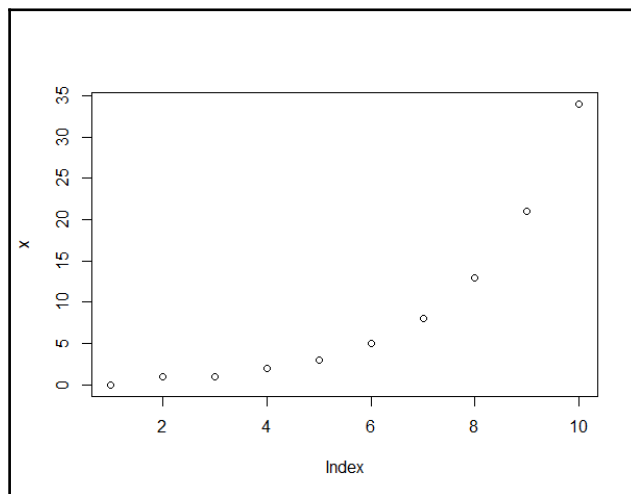
One can use a negative sign in the bracketed numbers in order to exclude the observations:

```
> x[-5:-6]
[1] 0 1 1 2 8 13 21 34
```

To visualize this sequence, we will utilize the `plot()` function:

```
> plot(x)
```

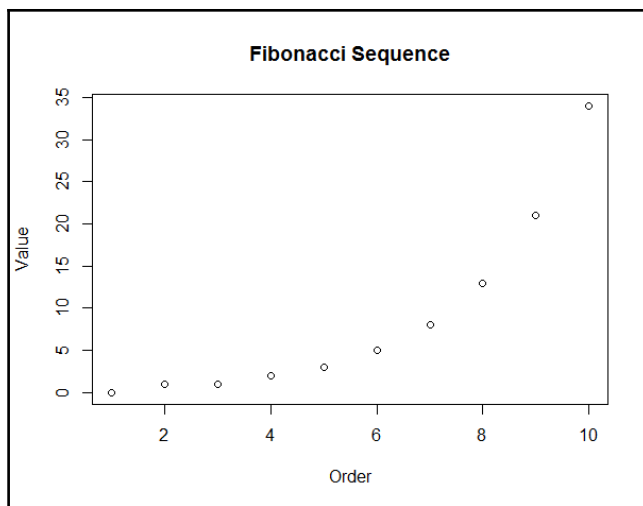
The output of the preceding command is as follows:



Adding a title and axis labels to the plot is easy using `main=...`, `xlab=...`, and `ylab=...`:

```
> plot(x, main = "Fibonacci Sequence", xlab = "Order", ylab = "Value")
```

The output of the preceding command is as follows:



We can transform a vector in R with a plethora of functions. Here, we will create a new object, `y`, that is the square root of `x`:

```
> y <- sqrt(x)

> y
[1] 0.000000 1.000000 1.000000 1.414214 1.732051 2.236068 2.828427
[8] 3.605551 4.582576 5.830952
```

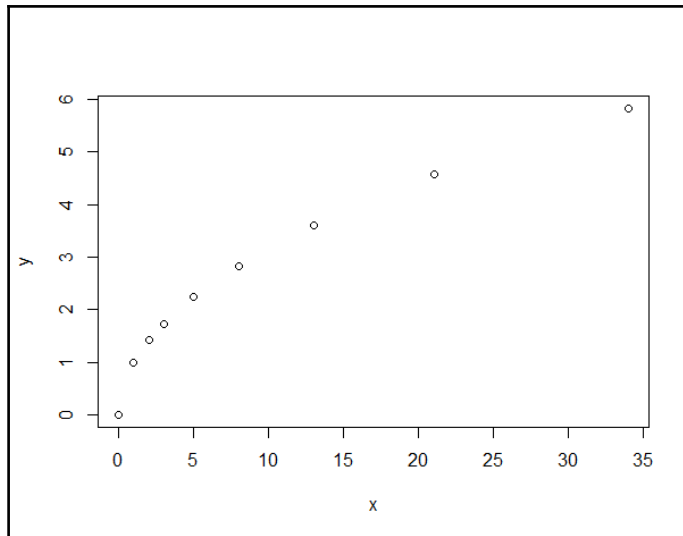
It is important here to point out that, if you are unsure of what syntax can be used in a function, then using `?` in front of it will pull up help on the topic. Try this!

```
> ?sqrt
```

This opens up help for a function. With the creation of `x` and `y`, one can produce a scatter plot:

```
> plot(x, y)
```

The following is the output of the preceding command:



Let's now look at creating another object that is a constant. Then, we will use this object as a scalar and multiply it by the `x` vector, creating a new vector called `x2`:

```
> z <- 3

> x2 <- x * z

> x2
[1] 0 3 3 6 9 15 24 39 63 102
```

R allows you to perform logical tests. For example, let's test whether one value is less than another:

```
> 5 < 6
[1] TRUE

> 6 < 5
[1] FALSE
```

In the first instance, R returned `TRUE` and in the second, `FALSE`. If you wanted to find out if a value is equal to another value, then you would use two equal signs (a test of equality). Remember, the equal symbol assigns a value and does not test for equality. Here is an example where we want to see if any of the values in the Fibonacci sequence that we created are equal to zero:

```
> x == 0
[1] TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

The output provides a list and we can clearly see that the first value of the `x` vector is indeed zero. In short, R's relational operators, `<=`, `<`, `==`, `>`, `>=`, and `!=`, stand for less than or equal, less than, equal, greater than, greater than or equal, and not equal respectively.

A couple of functions that we should address are `rep()` and `seq()`, which are useful in creating your own vectors. For example, `rep(5, 3)` would replicate the value 5 three times. It also works with strings:

```
> rep("North Dakota Hockey, 2016 NCAA Division 1 Champions", times=3)
[1] "North Dakota Hockey, 2016 NCAA Division 1 Champions"
[2] "North Dakota Hockey, 2016 NCAA Division 1 Champions"
[3] "North Dakota Hockey, 2016 NCAA Division 1 Champions"
```

For a demonstration of `seq()`, let's say that we want to create a sequence of numbers from 0 to 10, `by = 2`. Then the code would be as follows:

```
> seq(0, 10, by = 2)
[1] 0 2 4 6 8 10
```


Data frames and matrices

We will now create a data frame, which is a collection of variables (vectors). We will create a vector of 1, 2, and 3 and another vector of 1, 1.5, and 2.0. Once this is done, the `rbind()` function will allow us to combine the rows:

```
> p <- seq(1:3)

> p
[1] 1 2 3

> q = seq(1, 2, by = 0.5)

> q
[1] 1.0 1.5 2.0

> r <- rbind(p, q)

> r
  [,1] [,2] [,3]
p    1  2.0    3
q    1  1.5    2
```

The result is a list of two rows with three values each. You can always determine the structure of your data using the `str()` function, which in this case shows us that we have two lists, one named `p` and the other named `q`:

```
> str(r)
num [1:2, 1:3] 1 1 2 1.5 3 2
- attr(*, "dimnames")=List of 2
..$ : chr [1:2] "p" "q"
..$ : NULL
```

Now, let's put them together as columns using `cbind()`:

```
> s <- cbind(p, q)

> s
  p q
[1,] 1 1.0
[2,] 2 1.5
[3,] 3 2.0
```

To put this in a data frame, use the `data.frame()` function. After that, examine the structure:

```
> s <- data.frame(s)

> str(s)
'data.frame': 3 obs. of  2 variables:
 $ p: num  1 2 3
 $ q: num  1 1.5 2
```

We now have a data frame, `(s)`, that has two variables of three observations each. We can change the names of the variables using `names()`:

```
> names(s) <- c("column 1", "column 2")

> s
  column 1 column 2
1         1      1.0
2         2      1.5
3         3      2.0
```

Let's have a go at putting this into a matrix format with `as.matrix()`. In some packages, R will require the analysis to be done on a data frame, but in others it will require a matrix. You can switch back and forth between a data frame and matrix as you require:

```
> t <- as.matrix(s)

> t
      column 1 column 2
[1,]         1      1.0
[2,]         2      1.5
[3,]         3      2.0
```

One of the things that you can do is check whether a specific value is in a matrix or data frame. For instance, we want to know the value of the first observation and first variable. In this case, we will need to specify the first row and the first column in brackets as follows:

```
> t[1,1]
column 1
1
```

Let's assume that you want to see all the values in the second variable (column). Then, just leave the row blank but remember to use a comma before the column(s) that you want to see:

```
> t[,2]
[1] 1.0 1.5 2.0
```

Conversely, let's say we want to look at the first two rows only. In this case, just use a colon symbol:

```
> t[1:2,]
      column 1 column 2
[1,]         1      1.0
[2,]         2      1.5
```

Assume that you have a data frame or matrix with 100 observations and ten variables and you want to create a subset of the first 70 observations and variables 1, 3, 7, 8, 9, and 10. What would this look like?

Well, using the colon, comma, concatenate function, and brackets, you could simply do the following:

```
> new <- old[1:70, c(1,3,7:10)]
```

Notice how you can easily manipulate what observations and variables you want to include. You can also easily exclude variables. Say that we just want to exclude the first variable; then you could do the following using a negative sign for the first variable:

```
> new <- old[, -1]
```

This syntax is very powerful in R for the fundamental manipulation of data. In the main chapters, we will also bring in more advanced data manipulation techniques.

Creating summary statistics

We will now cover some basic measures of a central tendency, dispersion, and simple plots. The first question that we will address is How does R handle missing values in calculations? To see what happens, create a vector with a missing value (NA in the R language), then sum the values of the vector with `sum()`:

```
> a <- c(1, 2, 3, NA)

> sum(a)
[1] NA
```

Unlike SAS, which would sum the non-missing values, R does not sum the non-missing values, but simply returns `NA`, indicating that at least one value is missing. Now, we could create a new vector with the missing value deleted but you can also include the syntax to exclude any missing values with `na.rm = TRUE`:

```
> sum(a, na.rm = TRUE)
[1] 6
```

Functions exist to identify measures of the central tendency and dispersion of a vector:

```
> data <- c(4, 3, 2, 5.5, 7.8, 9, 14, 20)

> mean(data)
[1] 8.1625

> median(data)
[1] 6.65

> sd(data)
[1] 6.142112

> max(data)
[1] 20

> min(data)
[1] 2

> range(data)
[1] 2 20

> quantile(data)
 0%   25%   50%   75%  100%
2.00  3.75  6.65 10.25 20.00
```

A `summary()` function is available that includes the mean, median, and quartile values:

```
> summary(data)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
2.000   3.750   6.650   8.162  10.250  20.000
```

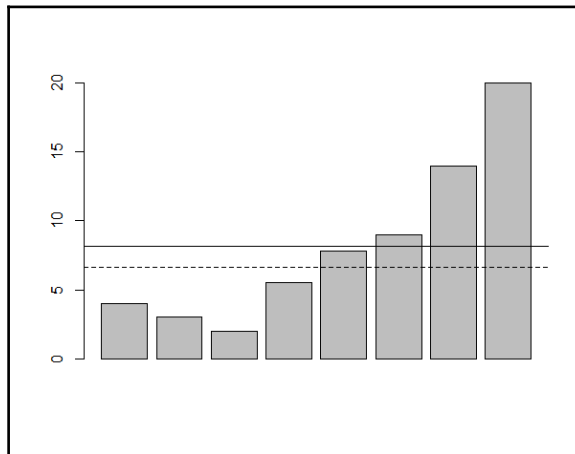
We can use plots to visualize the data. The base plot here will be `barplot()`, then we will use `abline()` to include the mean and median. As the default line is solid, we will create a dotted line for median with `lty = 2` to distinguish it from mean:

```
> barplot(data)

> abline(h = mean(data))

> abline(h = median(data), lty = 2)
```

The output of the preceding command is as follows:



A number of functions are available to generate different data distributions. Here, we can look at one such function for a normal distribution with a mean of zero and a standard deviation of 1, using `rnorm()` to create 100 data points. We will then plot the values and also plot a histogram. Additionally, to duplicate the results, ensure that you use the same random seed with `set.seed()`:

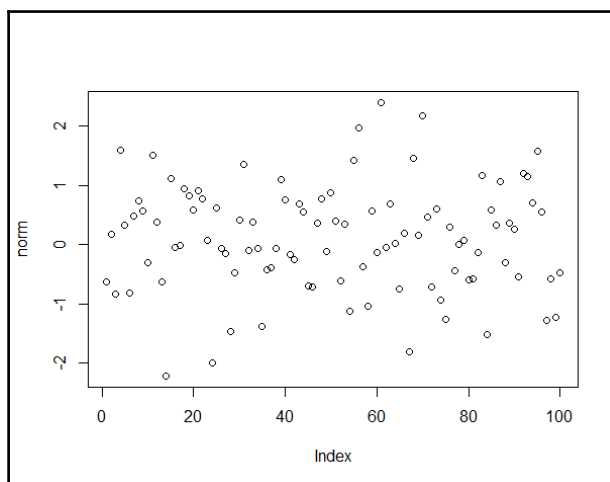
```
> set.seed(1)

> norm = rnorm(100)
```

This is the plot of the 100 data points:

```
> plot(norm)
```

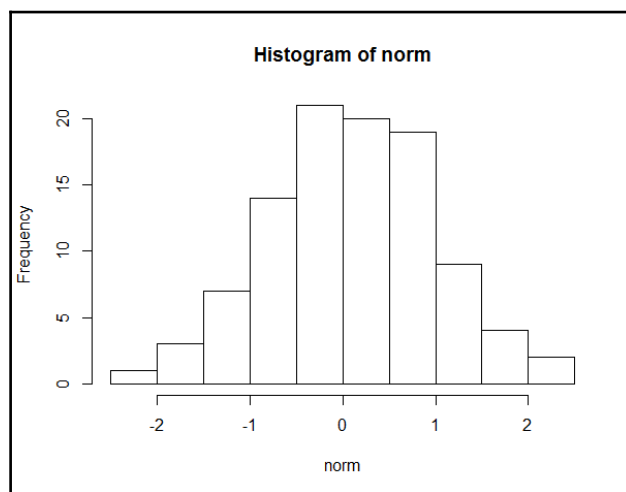
The output of the preceding command is as follows:



Finally, produce a histogram with `hist(norm)`:

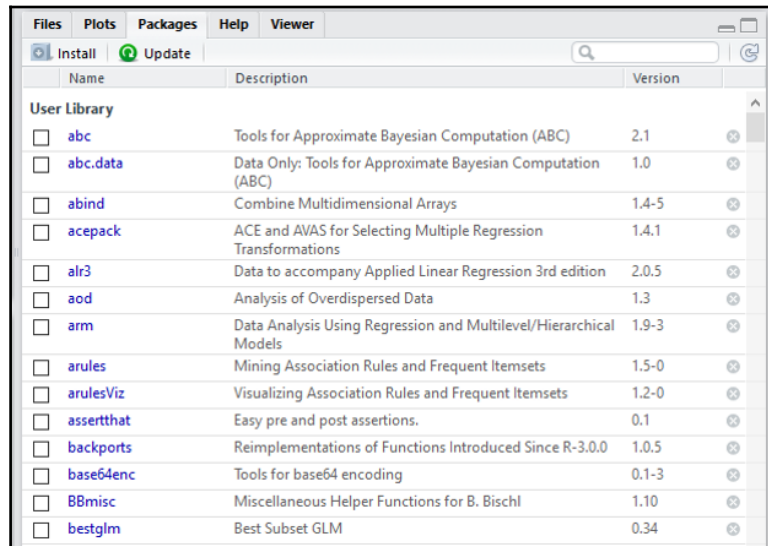
```
> hist(norm)
```

The following is the output of the preceding command:

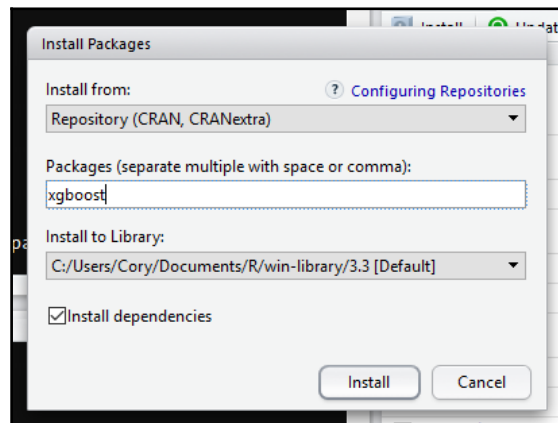


Installing and loading R packages

We discussed earlier how to install an R package using the `install()` function. To use an installed package, you also need to load it to be able to use it. Let's go through this again, first with the installation in RStudio and then loading the package. Look for and click the **Packages** tab. You should see something similar to this:



Now, let's install the R package, `xgboost`. Click on the **Install** icon and type the package name in the **Packages** section of the popup:



Click the **Install** button. Once the package has been fully installed, the command prompt will return. To load the package in order to be able to use it, only the `library()` function is required:

```
> library(xgboost)
```

With this, you are now able to use the functions built into the package.

Data manipulation with dplyr

Over the past couple of years I have been using `dplyr` more and more to manipulate and summarize data. It is faster than using the base functions, allows you to chain functions, and once you are familiar with it has a more user-friendly syntax. In my experience, just a few functions can accomplish the majority of your data manipulation needs. Install the package as described above, then load it into the R environment.

```
> library(dplyr)
```

Let's explore the `iris` dataset available in base R. Two of the most useful functions are `summarize()` and `group_by()`. In the code that follows, we see how to produce a table of the mean of `Sepal.Length` grouped by the `Species`. The variable we put the mean in will be called `average`.

```
> summarize(group_by(iris, Species), average = mean(Sepal.Length))
# A tibble: 3 X 2
  Species average
  <fctr>   <dbl>
1  setosa    5.006
2 versicolor 5.936
3  virginica 6.588
```

There are a number of summary functions: `n` (number), `n_distinct` (number of distinct), `IQR` (interquartile range), `min` (minimum), `max` (maximum), `mean` (mean), and `median` (median).

Another thing that helps you and others read the code is the pipe operator `%>%`. With the pipe operator, you chain your functions together instead of having to wrap them inside each other. You start with the dataframe you want to use, then chain the functions together where the first function values/arguments are passed to the next function and so on. This is how to use the pipe operator to produce the results as we got before.

```
> iris %>% group_by(Species) %>% summarize(average =  
  mean(Sepal.Length))  
# A tibble: 3 X 2  
  Species average  
  <fctr>   <dbl>  
1   setosa    5.006  
2 versicolor  5.936  
3  virginica  6.588
```

The `distinct()` function allows us to see what are the unique values in a variable. Let's see what different values are present in `Species`.

```
> distinct(iris, Species)  
  Species  
1   setosa  
2 versicolor  
3  virginica
```

Using the `count()` function will automatically do a count for each level of the variable.

```
> count(iris, Species)  
# A tibble: 3 X 2  
  Species     n  
  <fctr> <int>  
1   setosa    50  
2 versicolor    50  
3  virginica    50
```

What about selecting certain rows based on a matching condition? For that we have `filter()`. Let's select all rows where `Sepal.Width` is greater than 3.5 and put them in a new dataframe:

```
> df <- filter(iris, Sepal.Width > 3.5)
```

Let's look at this dataframe, but first we want to arrange the values by `Petal.Length` in descending order:

```
> df <- arrange(iris, desc(Petal.Length))

> head(df)
  Sepal.Length Sepal.Width Petal.Length Petal.Width  Species
1          7.7         2.6          6.9         2.3 virginica
2          7.7         3.8          6.7         2.2 virginica
3          7.7         2.8          6.7         2.0 virginica
4          7.6         3.0          6.6         2.1 virginica
5          7.9         3.8          6.4         2.0 virginica
6          7.3         2.9          6.3         1.8 virginica
```

OK, we now want to select variables of interest. This is done with the `select()` function. Next, we are going to create two dataframes, one with the columns starting with `Sepal` and another with the `Petal` columns and the `Species` column--in other words, column names NOT starting with `Se`. This can be done by using those specific names in the function; alternatively, as follows, use the `starts_with` syntax:

```
> iris2 <- select(iris, starts_with("Se"))

> iris3 <- select(iris, -starts_with("Se"))
```

OK, let's bring them back together. Remember `cbind()` earlier? With `dplyr` you can use the `bind_cols()` function, which will put it into a dataframe. Be advised that like `cbind()` it will match rows by position. If you have rownames or some other key, such as customer ID and so on, you can join the data using functions such as `left_join()` and `inner_join()`. Since our rows match, this command will work just fine.

```
> theIris <- bind_cols(iris2, iris3)
  head(theIris)
  head(iris)
```

On your own, use the `head()` function to compare the first six rows of the `iris` and `iris` and you will see they are an exact match. If you want to concatenate your data like we did above with `rbind()`, the `bind_rows()` function is available. What if we want to see how many unique `Sepal.Width` measurements there are? Recall that there are a total of 150 observations in the data set. We've already used `distinct()` and `count()`. This code will give you just the number of distinct values, which is 23:

```
> summarize(iris, n_distinct(Sepal.Width))
  n_distinct(Sepal.Width)
1                23
```

It seems in almost any large amount of data there are duplicate observations, or they are created with complex joins. To dedupe with `dplyr` is quite simple. For instance, let's assume we want to create a dataframe of just the unique values of `Sepal.Width`, and want to keep all of the columns. This will do the trick:

```
> dedupe <- iris %>% distinct(Sepal.Width, .keep_all = T)

> str(dedupe)
'data.frame': 23 obs. of 5 variables:
 $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 4.4 5.4 5.8 ...
 $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 2.9 3.7 4 ...
 $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.4 1.5 1.2 ...
 $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.2 ...
 $ Species : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1
 1 1 1 1 1
```

Notice that I used the pipe operator to chain `iris` to the function and within `distinct()` I specified `.keep_all = T` so that all columns are in the new dataframe; otherwise only `Sepal.Width` would have entered.

There you have it. If you want to become more efficient at data-munging in R, give `dplyr` a try.

Summary

The purpose of this appendix was to allow R novices to learn the basics of the programming language and prepare them for the code in the book. This consisted of learning how to install R and RStudio and creating objects, vectors, and matrices. Then, we explored some of the mathematical and statistical functions. We covered how to install and load a package in R using RStudio. Finally, we explored the power of `dplyr` to efficiently manipulate and summarize data. Throughout the appendix, the plot syntax for the base and examples are included. While this appendix will not make you an expert in R, it will get you up to speed to follow along with the examples in the book.

B Sources

Granger, G.W.J., Newbold, P., (1974), Spurious Regressions in Econometrics, *Journal of Econometrics*, 1974 (2), 111-120

Hechenbichler, K., Schliep, K.P., (2004), Weighted k-Nearest-Neighbors and Ordinal Classification, *Institute for Statistics, Sonderforschungsbereich 386*, Paper 399. <http://epub.ub.uni-muenchen.de/>

Hinton, G.E., Salakhutdinov, R.R., (2006), Reducing the Dimensionality of Data with Neural Networks, *Science*, August 2006, 313(5786):504-7

James, G., Witten, D., Hastie, T., Tibshirani, R. (2013), *An Introduction to Statistical Learning*, 1st ed. New York: Springer

Kodra, E., (2011), Exploring Granger Causality Between Global Average Observed Time Series of Carbon Dioxide and Temperature, *Theoretical and Applied Climatology*, Vol. 104 (3), 325-335

Natekin, A., Knoll, A., (2013), Gradient Boosting Machines, a Tutorial, *Frontiers in Neuroinformatics*, 2013; 7-21. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3885826/>

Tibshirani, R., (1996), Regression Shrinkage and Selection via the LASSO, *Journal of the Royal Statistical Society, Series B.*, 58(1), 267-288

Triacca, U., (2005), Is Granger causality analysis appropriate to investigate the relationship between atmospheric concentration of carbon dioxide and global surface air temperature?, *Theoretical and Applied Climatology*, 81 (3), 133-135

Toda, H., Yamamoto, T., (1995), Statistical Inference in Vector Autoregressions with Possibly Integrated Processes, *Journal of Econometrics*, 1995, (66), issue 1-2, 225-250

Index

A

- Aikake's Information Criterion (AIC) 39, 314, 328
- algorithm flowchart 16, 17, 18, 19, 20
- Amazon Machine Images (AMI)
 - about 362
 - URL 362
- Amazon Web Services (AWS)
 - about 359
 - account, creating 359, 360
 - RStudio, starting 365, 367
 - URL 359
 - virtual machine, launching 361, 362, 364
- apriori 252
- Area Under Curve (AUC) 82
- Artificial neural networks (ANNs)
 - about 173
 - reference link 173
- Augmented Dickey-Fuller (ADF) test 319
- Autocorrelation Function (ACF) 308
- automated readability index 338
- autoregressive integrated moving average (ARIMA)
 - model 307

B

- backpropagation 173
- backward stepwise regression 37
- bank.csv dataset
 - URL 192
- Baye's theorem 71
- Bayesian Information Criterion (BIC) 39
- Benign Prostatic Hyperplasia (BPH) 90
- bias-variance 72
- bootstrap aggregation (bagging) 148
- boxplot 220
- Breusch-Pagan (BP) test 46
- business case

- about 89
- business understanding 89, 90
- data preparation 90
- business understanding
 - about 10, 89
 - analytical goals, determining 12
 - business objective, identifying 11
 - project plan, producing 12
 - situation, assessing 12

C

- Carbon Dioxide Information Analysis Center (CDIAC)
 - URL 315
- caret package
 - about 108
 - models, URL 291
 - reference link 108
- change agent 8
- classification 114
- classification methods 56
- classification trees 147
- climate.csv file
 - URL 315
- cloud computing
 - reference link 358
- cluster analysis
 - about 201
 - business understanding 208
 - data preparation 209, 210
 - data understanding 209, 210
 - hierarchical 202
 - k-means 202
- Cohen's Kappa statistic 132
- collaborative filtering
 - about 260
 - item-based collaborative filtering (IBCF) 262

- principal components analysis (PCA) 262, 266
- singular value decomposition (SVD) 262, 266
- user-based collaborative filtering (UBCF) 261
- convolutional neural networks (CNN) 180
- Cook's distance (Cook's D) 31
- cosine similarity 261
- cover 166
- CRISP-DM 1.0
 - reference link 10
- Cross Correlation Function (CCF) 319
- Cross-Entropy 174
- Cross-Industry Standard Process for Data Mining (CRISP-DM)
 - about 8
 - process 9, 10
- cross-validation
 - with glmnet 111, 112, 113
- cultivar labels 217
- curse of dimensionality 230

D

- data frame
 - creating 379, 380
- data manipulation
 - with dplyr 386, 387, 388, 389
- data preparation 13, 14, 90
- data understanding 13, 90
- decision trees, business case
 - about 150
 - classification tree, using 154, 155
 - evaluation 150
 - extreme gradient boosting, using 163, 165, 166, 167, 168
 - feature selection, with random forests 169, 170
 - model, selecting 168
 - modeling 150
 - random forest classification 159, 163
 - random forest regression 156, 157, 158
 - regression tree, using 150, 152, 153
- deep learning, business case
 - business, understanding 181, 182
 - data, preparing 182, 184, 185, 186, 187
 - data, understanding 182, 184, 185, 186, 187
 - evaluation 187, 189, 190, 191, 192
 - modeling 187, 189, 190, 191, 192

- deep learning
 - about 177
 - advanced methods 179
 - example 192
 - H2O 193
 - resources 179
 - URL 179
- dendrogram 203, 215
- deployment 15, 16
- diabetes dataset
 - URL 125
- Dindex plot 214
- dirichlet distribution 337
- Discriminant Analysis (DA)
 - overview 70, 71, 72
- Distance Measuring Equipment (DME) 181
- Document-Term Matrix (DTM) 336
- dplyr
 - used, for data manipulation 386, 387, 388, 389
- dummy feature 50
- dynamic topic modelling 338

E

- ECLAT 252
- eigenvalues 233
- eigenvectors 233
- Elastic Compute Cloud (EC2) 359
- elastic net 89, 108
- ensembles
 - about 289, 290
 - business understanding 290
 - data, understanding 290
 - model, selection 291, 292, 293, 294
 - modeling, evaluation 291, 293, 294
 - reference link 289
- equimax 234
- Euclidean Distance 120, 202, 203, 204
- evaluation
 - about 15, 211
 - Gower dissimilarity coefficient 225, 227
 - hierarchical clustering algorithm 211, 212, 218, 220
 - k-means clustering 222, 223, 224, 225
 - PAM 225, 227
 - Random Forests 227

exponential smoothing 307
Extract, Transform, Load (ETL) 13
eXtreme Gradient Boosting (Xgboost)
 about 149
 reference link 149

F

F-Measure 339
factor 50
False Positive Rate (FPR) 82
feature selection
 for support vector machines (SVM) 142
 with random forests 169, 170
feed forward 173
Final Prediction Error (FPE) 314, 328
Fine Needle Aspiration (FNA) 57
Fisher Discriminant Analysis (FDA) 70
forward stepwise selection 36
frequency 166

G

gain 166
gap statistic 212
Generalized Cross Validation (GCV) 77
Gini index 147
glmnet package
 cross-validation 111, 112, 113
global warming dataset
 URL 315
Gower dissimilarity coefficient 202, 205, 206, 225, 227
gradient boosted trees 146
gradient boosting
 about 149, 150
 reference link 149
Granger causality
 about 313, 314
 reference link 330
Graphical User Interface (GUI) 372
greedy 146

H

H2O
 about 193
 data, uploading 194, 195

 modeling 196, 197
 test dataset, creating 195
 train dataset, creating 195
 URL 193
 URL, for documentation 196
 URL, for installing 193
Hannan-Quinn Criterion (HQ) 328
Hat Matrix 50
heatmaps 34
heteroscedasticity 29
hierarchical clustering algorithm
 about 202, 203, 211, 212, 218, 220
 Euclidean distance 203, 204
Holt-Winters Method 307
Hubert Index 214
hyperbolic tangent (tanh) 175

I

indicator 50
Information Values (IV) 77
instance 359
instance-based learning 119
Integrated Development Environment (IDE) 369
interaction terms 52
interquartile range 221
item-based collaborative filtering (IBCF) 262

K

K-fold cross-validation 67
k-means clustering 202, 204, 205, 222, 223, 224, 225
k-nearest neighbors (KNN) 118, 119
K-sets 67

L

L1-norm 88
Latent Dirichlet Allocation (LDA) 337
lazy learning 119
Least Absolute Shrinkage and Selection Operator (LASSO) 87, 88, 105, 106, 107
least squares approach 22
Leave-One-Out-Cross-Validation (LOOCV) 49, 67
linear combination 231
Linear Discriminant Analysis (LDA) 70, 73, 74, 75, 76

- linear model
 - considerations 50
 - elastic net 89
 - interaction terms 52
 - LASSO 88
 - qualitative features 50
 - regularization 87
 - ridge regression 88
- linear regression
 - about 22, 56
 - multivariate linear regression 32
 - univariate linear regression 23, 25
- logistic regression model 64, 65
- logistic regression
 - about 56, 57, 63
 - business understanding 57, 58
 - data preparing 58, 60, 61, 62
 - data understanding 58, 60, 61, 62
 - evaluation 63
 - example 114, 116, 117
 - with cross-validation 67, 68, 70
- Long Short-Term Memory (LSTM) 180
- loser point 236
- loss function 149

M

- Mallow's Cp (Cp) 39
- margin 121
- market basket analysis, terminology
 - confidence 251
 - itemset 251
 - lift 252
 - support 251
- market basket analysis
 - business understanding 252, 253
 - data, preparing 253, 255
 - data, understanding 253, 255
 - evaluation 255, 257
 - modeling 255, 257
 - overview 251, 252
- matrices 379, 380
- Mean Squared Error (MSE) 99, 307
- meteorological dataset
 - URL 315
- MLR's ensemble 303

- model selection 82, 84, 113, 114
- modeling
 - about 14, 96, 211
 - cross-validation, with glmnet package 111, 112, 113
 - elastic net 108
 - evaluation 96
 - Gower dissimilarity coefficient 225, 227
 - hierarchical clustering algorithm 211, 212, 218, 220
 - k-means clustering 222, 223, 224, 225
 - LASSO 105, 106, 107
 - PAM 225, 227
 - Random Forest 227
 - ridge regression 100, 102, 104
 - subsets 96, 97, 98, 99
- multiclass classification
 - about 294, 295
 - business, understanding 295, 296, 297, 299
 - data, understanding 295, 296, 297, 299
 - model evaluation 299
 - model selection 299
 - models, URL 300
 - random forest, implementing 300, 301, 302
 - ridge regression, implementing 302, 303
- Multivariate Adaptive Regression Splines (MARS)
 - 63, 76, 77, 78, 80, 81
- multivariate linear regression
 - about 32
 - business understanding 33
 - data preparation 33, 34, 36
 - data understanding 33, 34, 36
 - evaluation 36, 37, 39, 40, 41, 42, 43, 44, 46, 48, 49, 50
 - modeling 36, 37, 39, 40, 41, 42, 43, 44, 46, 48, 49, 50
- multivariate normality 231
- MXNet
 - about 180
 - URL 180

N

- neural networks 173, 175, 177
- nhlTest.csv file
 - URL 236

nhlTrain.csv file

URL 236

nonlinear techniques, business case

about 124

business, understanding 124, 125

data, preparing 125, 126, 128, 129

data, understanding 125, 126, 128, 129

evaluation 131

KNN, modeling 131, 132, 133, 134

model, selecting 139

modeling 131

SVM, modeling 136, 138, 139

Normal Q-Q plot 31

O

Ordinary Least Squares (OLS) 55

out-of-bag (oob) 148

P

Pair Key 360

PAM clustering algorithm 205

Partial Autocorrelation Function (PACF) 309

Partitioning Around Medoids (PAM) 202, 206,
207, 225, 227

pearson correlation coefficient 261

Pearson's r 34

polarity 338

porter stemming algorithm 336

Prediction Error Sum of Squares (PRESS) 49

Principal Components Analysis (PCA)

about 205, 230, 262, 266

business, understanding 236

component, extraction 239

data, preparing 237, 238

data, understanding 237, 238

evaluation 239

factor scores, creating from components 242

interpretation 240, 242

modeling 239

orthogonal rotation 240, 242

regression analysis 243, 246, 247, 248, 249

principal components

overview 231, 232, 233, 234

rotation 234, 235, 236

Prostate Specific Antigen (PSA) 89

Q

Quadratic Discriminant Analysis (QDA) 70

qualitative features 50

Quantile-Quantile (Q-Q) 31

quartimax 234

R

R packages

installing 385

loading 385

R

executing 368, 369, 371, 372, 373, 374

installing 368, 369, 371, 372, 373, 374

URL 369

using 374, 376, 378

radical 336

Random Forests

about 146, 148, 149, 202, 207, 208, 227

used, for feature selection 169, 170

Receiver Operating Characteristic (ROC)

about 82

reference link 82

recommendation engine

business, understanding 266

data, preparing 267, 269

data, understanding 267, 269

evaluation 269, 272, 275, 276, 277, 279

modeling 269, 272, 275, 276, 277, 279

overview 260

recommenderlab

URL 267

Rectified Linear Unit (ReLU) 180

recurrent neural networks (RNN) 180

regression trees 146, 147

regularization

about 114

logistic regression, example 114, 116

of linear model 87

Residual Sum of Squares (RSS) 23

Residuals vs Leverage plot 31

Restricted Boltzmann Machine 178

ridge regression 88, 100, 102, 104

Root Mean Square Error (RMSE) 109, 247

rotation 234, 235, 236

RStudio
 starting 365, 367
 URL 372

S

Schwarz-Bayes Criterion (SC) 328
Secure Shell (SSH) 359
sequential data analysis
 about 279
 applying 280, 281, 283
 dataset, URL 280
shrinkage penalty 87
singular value decomposition (SVD) 262, 266
space shuttle
 reference link 182
Sparse Coding Model 178
sum of squared error (SSE) 174, 307
summary statistics
 creating 381, 382, 383, 384
supervised learning 201
support vector 122
support vector machines (SVM)
 about 118, 120, 121, 123
 feature selection 142
suspected outliers 221
Synthetic Minority Over-Sampling Technique (SMOTE) 296

T

TensorFlow
 reference link 180
 URL 179
Term-Document Matrix (TDM) 336
text mining
 business understanding 340
 data, preparing 340, 341, 342, 343
 data, understanding 340, 341, 342, 343
 evaluation 343
 framework 335, 336
 methods 335, 336
 modeling 343

 with additional quantitative analysis 348, 351, 355, 356
 with other quantitative analyses 338, 339
 with topic models 343, 344, 345, 346
 word frequency, obtaining 343, 344, 345, 346

tinker 179
topic models 337
tree-based learning 149
True Positive Rate (TPR) 82

U

univariate linear regression
 about 23, 25
 business understanding 26, 27, 29, 30, 32
univariate time series analysis
 business, understanding 314
 causality, examining 324
 data, preparing 316
 data, understanding 316
 evaluation 320, 321, 323, 324
 Granger causality 313, 314
 modeling 320, 321, 323, 324
 performing 307, 308, 310, 311, 313
 with linear regression 324, 326
 with vector autoregression 327, 328, 329, 330, 331
unsupervised learning 201
user-based collaborative filtering (UBCF) 261

V

valence shifters 338
Variance Inflation Factor (VIF) 43
varimax 234
Vector Autoregression (VAR) 314
virtual machine
 launching 361, 364

W

Weight-of-Evidence (WOE) 77
word frequency
 obtaining 343, 344, 345, 346