



Community Experience Distilled

Mastering Machine Learning with R

Master machine learning techniques with R to deliver insights for complex projects

Cory Lesmeister

[PACKT] open source*
PUBLISHING

community experience distilled

Mastering Machine Learning with R

Master machine learning techniques with R to deliver insights for complex projects

Cory Lesmeister



BIRMINGHAM - MUMBAI

Mastering Machine Learning with R

Copyright © 2015 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

Production reference: 1231015

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78398-452-7

www.packtpub.com

Credits

Author

Cory Lesmeister

Project Coordinator

Nidhi Joshi

Reviewers

Vikram Dhillon

Miro Kopecky

Pavan Narayanan

Doug Ortiz

Shivani Rao, PhD

Proofreader

Safis Editing

Indexer

Mariammal Chettiar

Graphics

Disha Haria

Commissioning Editor

Kartikey Pandey

Production Coordinator

Nilesh Mohite

Acquisition Editor

Nadeem N. Bagban

Cover Work

Nilesh Mohite

Content Development Editor

Siddhesh Salvi

Technical Editor

Suwarna Rajput

Copy Editor

Tasneem Fatehi

About the Author

Cory Lesmeister currently works as an advanced analytics consultant for Clarity Solution Group, where he applies the methods in this book to solve complex problems and provide actionable insights. 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 Reservist, Cory was in Baghdad, Iraq, in 2009 as a 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. Cory lives in Carmel, IN, with his wife and their two teenage daughters.

About the Reviewers

Vikram Dhillon is a software developer, bioinformatics researcher, and software coach at the Blackstone LaunchPad in the University of Central Florida. He has been working on his own start-up involving healthcare data security. He lives in Orlando and regularly attends developer meetups and hackathons. He enjoys spending his spare time reading about new technologies such as the blockchain and developing tutorials for machine learning in game design. He has been involved in open source projects for over 5 years and writes about technology and start-ups at opsbug.com.

Miro Kopecky is a passionate JVM enthusiast from the first moment he joined Sun Microsystems in 2002. Miro truly believes in a distributed system design, concurrency, and parallel computing, which means pushing the system's performance to its limits without losing reliability and stability. He has been working on research of new data mining techniques in neurological signal analysis during his PhD studies. Miro's hobbies include autonomic system development and robotics.

I would like to thank my family and my girlfriend, Tanja, for their support during the reviewing of this book.

Pavan Narayanan is an applied mathematician and is experienced in mathematical programming, analytics, and web development. He has published and presented papers in algorithmic research to the Transportation Research Board, Washington DC and SUNY Research Conference, Albany, NY. An avid blogger at <https://datasciencehacks.wordpress.com>, his interests are exploring problem solving techniques—from industrial mathematics to machine learning. Pavan can be contacted at pavan.narayanan@gmail.com.

He has worked on books such as *Apache mahout essentials*, *Learning apache mahout*, and *Real-time applications development with Storm and Petrel*.

I would like to thank my family and God Almighty for giving me strength and endurance and the folks at Packt Publishing for the opportunity to work on this book.

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.

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

His hobbies include yoga and scuba diving. He is the founder of Illustris, LLC, and can be contacted at dougortiz@illustris.org.

Shivani Rao, PhD, is a machine learning engineer based in San Francisco and Bay Area working in areas of search, analytics, and machine learning. Her background and areas of interest are in the field of computer vision, image processing, applied machine learning, data mining, and information retrieval. She has also accrued industry experience in companies such as Nvidia , Google, and Box. Shivani holds a PhD from the Computer Engineering Department of Purdue University spanning areas of machine learning, information retrieval, and software engineering. Prior to that, she obtained a masters from the Computer Science and Engineering Department of the Indian Institute of Technology (IIT), Madras, majoring in Computer Vision and Image Processing.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

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

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	vii
Chapter 1: A Process for Success	1
The process	2
Business understanding	3
Identify the business objective	4
Assess the situation	5
Determine the analytical goals	5
Produce a project plan	5
Data understanding	6
Data preparation	6
Modeling	7
Evaluation	8
Deployment	8
Algorithm flowchart	9
Summary	14
Chapter 2: Linear Regression – The Blocking and Tackling of Machine Learning	15
Univariate linear regression	16
Business understanding	18
Multivariate linear regression	25
Business understanding	25
Data understanding and preparation	25
Modeling and evaluation	28
Other linear model considerations	40
Qualitative feature	41
Interaction term	43
Summary	44

Table of Contents

Chapter 3: Logistic Regression and Discriminant Analysis	45
Classification methods and linear regression	46
Logistic regression	46
Business understanding	47
Data understanding and preparation	48
Modeling and evaluation	54
The logistic regression model	54
Logistic regression with cross-validation	58
Discriminant analysis overview	62
Discriminant analysis application	64
Model selection	69
Summary	74
Chapter 4: Advanced Feature Selection in Linear Models	75
Regularization in a nutshell	76
Ridge regression	77
LASSO	77
Elastic net	78
Business case	78
Business understanding	78
Data understanding and preparation	79
Modeling and evaluation	85
Best subsets	85
Ridge regression	90
LASSO	95
Elastic net	98
Cross-validation with glmnet	101
Model selection	103
Summary	104
Chapter 5: More Classification Techniques – K-Nearest Neighbors and Support Vector Machines	105
K-Nearest Neighbors	106
Support Vector Machines	107
Business case	111
Business understanding	111
Data understanding and preparation	112
Modeling and evaluation	118
KNN modeling	118
SVM modeling	124
Model selection	128
Feature selection for SVMs	131
Summary	133

Table of Contents

Chapter 6: Classification and Regression Trees	135
Introduction	135
An overview of the techniques	136
Regression trees	136
Classification trees	137
Random forest	138
Gradient boosting	139
Business case	140
Modeling and evaluation	140
Regression tree	140
Classification tree	144
Random forest regression	147
Random forest classification	151
Gradient boosting regression	156
Gradient boosting classification	159
Model selection	163
Summary	164
Chapter 7: Neural Networks	165
Neural network	166
Deep learning, a not-so-deep overview	170
Business understanding	172
Data understanding and preparation	173
Modeling and evaluation	179
An example of deep learning	186
H2O background	187
Data preparation and uploading it to H2O	187
Create train and test datasets	191
Modeling	191
Summary	194
Chapter 8: Cluster Analysis	195
Hierarchical clustering	196
Distance calculations	197
K-means clustering	198
Gower and partitioning around medoids	199
Gower	199
PAM	200
Business understanding	200
Data understanding and preparation	201
Modeling and evaluation	203
Hierarchical clustering	203
K-means clustering	214

Table of Contents

Clustering with mixed data	217
Summary	220
Chapter 9: Principal Components Analysis	221
An overview of the principal components	222
Rotation	225
Business understanding	226
Data understanding and preparation	227
Modeling and evaluation	233
Component extraction	233
Orthogonal rotation and interpretation	236
Creating factor scores from the components	237
Regression analysis	239
Summary	244
Chapter 10: Market Basket Analysis and Recommendation Engines	245
An overview of a market basket analysis	246
Business understanding	247
Data understanding and preparation	248
Modeling and evaluation	250
An overview of a recommendation engine	255
User-based collaborative filtering	256
Item-based collaborative filtering	257
Singular value decomposition and principal components analysis	257
Business understanding and recommendations	262
Data understanding, preparation, and recommendations	262
Modeling, evaluation, and recommendations	265
Summary	276
Chapter 11: Time Series and Causality	277
Univariate time series analysis	278
Bivariate regression	283
Granger causality	284
Business understanding	286
Data understanding and preparation	289
Modeling and evaluation	293
Univariate time series forecasting	294
Time series regression	302
Examining the causality	310
Summary	317

Table of Contents

Chapter 12: Text Mining	319
Text mining framework and methods	320
Topic models	322
Other quantitative analyses	323
Business understanding	325
Data understanding and preparation	325
Modeling and evaluation	330
Word frequency and topic models	330
Additional quantitative analysis	337
Summary	344
Appendix: R Fundamentals	345
Introduction	345
Getting R up and running	345
Using R	354
Data frames and matrices	358
Summary stats	360
Installing and loading the R packages	364
Summary	365
Index	367

Preface

"He who defends everything, defends nothing."

— Frederick the Great

Machine learning is a very broad topic. The following quote sums it up nicely:
The first problem facing you is the bewildering variety of learning algorithms available. Which one to use? There are literally thousands available, and hundreds more are published each year. (Domingo, P., 2012.) It would therefore be irresponsible to try and cover everything in the chapters that follow because, to paraphrase Frederick the Great, we would achieve nothing.

With this constraint in mind, I hope to provide a solid foundation of algorithms and business considerations that will allow the reader to walk away and, first of all, take on any machine learning tasks with complete confidence, and secondly, be able to help themselves in figuring out other algorithms and topics. Essentially, if this book significantly helps you to help yourself, then I would consider this a victory. Don't think of this book as a destination but rather, as a path to self-discovery.

The world of R can be as bewildering as the world of machine learning! There is seemingly an endless number of R packages with a plethora of blogs, websites, discussions, and papers of various quality and complexity from the community that supports R. This is a great reservoir of information and probably R's greatest strength, but I've always believed that an entity's greatest strength can also be its greatest weakness. R's vast community of knowledge can quickly overwhelm and/or sidetrack you and your efforts. Show me a problem and give me ten different R programmers and I'll show you ten different ways the code is written to solve the problem. As I've written each chapter, I've endeavored to capture the critical elements that can assist you in using R to understand, prepare, and model the data. I am no R programming expert by any stretch of the imagination, but again, I like to think that I can provide a solid foundation herein.

Another thing that lit a fire under me to write this book was an incident that happened in the hallways of a former employer a couple of years ago. My team had an IT contractor to support the management of our databases. As we were walking and chatting about big data and the like, he mentioned that he had bought a book about machine learning with R and another about machine learning with Python. He stated that he could do all the programming, but all of the statistics made absolutely no sense to him. I have always kept this conversation at the back of my mind throughout the writing process. It has been a very challenging task to balance the technical and theoretical with the practical. One could, and probably someone has, turned the theory of each chapter to its own book. I used a heuristic of sorts to aid me in deciding whether a formula or technical aspect was in the scope, which was would this help me or the readers in the discussions with team members and business leaders? If I felt it might help, I would strive to provide the necessary details.

I also made a conscious effort to keep the datasets used in the practical exercises large enough to be interesting but small enough to allow you to gain insight without becoming overwhelmed. This book is not about big data, but make no mistake about it, the methods and concepts that we will discuss can be scaled to big data.

In short, this book will appeal to a broad group of individuals, from IT experts seeking to understand and interpret machine learning algorithms to statistical gurus desiring to incorporate the power of R into their analysis. However, even those that are well-versed in both IT and statistics—experts if you will—should be able to pick up quite a few tips and tricks to assist them in their efforts.

Machine learning defined

Machine learning is everywhere! It is used in web search, spam filters, recommendation engines, medical diagnostics, ad placement, fraud detection, credit scoring, and I fear in these autonomous cars that I hear so much about. The roads are dangerous enough now; the idea of cars with artificial intelligence, requiring *CTRL + ALT + DEL* every 100 miles, aimlessly roaming the highways and byways is just too terrifying to contemplate. But, I digress.

It is always important to properly define what one is talking about and machine learning is no different. The website, machinelearningmastery.com, has a full page dedicated to this question, which provides some excellent background material. It also offers a succinct one-liner that is worth adopting as an operational definition: **machine learning** is the training of a model from data that generalizes a decision against a performance measure.

With this definition in mind, we will require a few things in order to perform machine learning. The first is that we have the data. The second is that a pattern actually exists, which is to say that with known input values from our training data, we can make a prediction or decision based on data that we did not use to train the model. This is the **generalization** in machine learning. Third, we need some sort of performance measure to see how well we are learning/generalizing, for example, the mean squared error, accuracy, and others. We will look at a number of performance measures throughout the book.

One of the things that I find interesting in the world of machine learning are the changes in the language to describe the data and process. As such, I can't help but include this snippet from the philosopher, George Carlin:

"I wasn't notified of this. No one asked me if I agreed with it. It just happened. Toilet paper became bathroom tissue. Sneakers became running shoes. False teeth became dental appliances. Medicine became medication. Information became directory assistance. The dump became the landfill. Car crashes became automobile accidents. Partly cloudy became partly sunny. Motels became motor lodges. House trailers became mobile homes. Used cars became previously owned transportation. Room service became guest-room dining, and constipation became occasional irregularity.

— *Philosopher and Comedian, George Carlin*

I cut my teeth on datasets that had dependent and independent variables. I would build a model with the goal of trying to find the best fit. Now, I have labeled the instances and input features that require engineering, which will become the feature space that I use to learn a model. When all was said and done, I used to look at my model parameters; now, I look at weights.

The bottom line is that I still use these terms interchangeably and probably always will. Machine learning purists may curse me, but I don't believe I have caused any harm to life or limb.

Machine learning caveats

Before we pop the cork on the champagne bottle and rest easy that machine learning will cure all of our societal ills, we need to look at a few important considerations—caveats if you will—about machine learning. As you practice your craft, always keep these at the back of your mind. It will help you steer clear of some painful traps.

Failure to engineer features

Just throwing data at the problem is not enough; no matter how much of it exists. This may seem obvious, but I have personally experienced, and I know of others who have run into this problem, where business leaders assumed that providing vast amounts of raw data combined with the supposed magic of machine learning would solve all the problems. This is one of the reasons the first chapter is focused on a process that properly frames the business problem and leader's expectations.

Unless you have data from a designed experiment or it has been already preprocessed, raw, observational data will probably never be in a form that you can begin modeling. In any project, very little time is actually spent on building models. The most time-consuming activities will be on the engineering features: gathering, integrating, cleaning, and understanding the data. In the practical exercises in this book, I would estimate that 90 percent of my time was spent on coding these activities versus modeling. This, in an environment where most of the datasets are small and easily accessed. In my current role, 99 percent of the time in SAS is spent using PROC SQL and only 1 percent with things such as PROC GENMOD, PROC LOGISTIC, or Enterprise Miner.

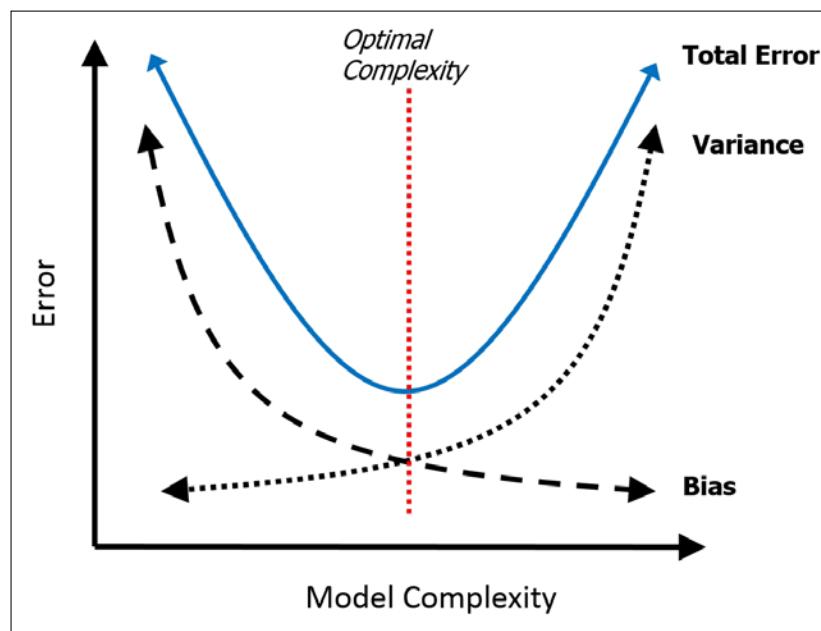
When it comes to feature engineering, I fall in the camp of those that say there is no substitute for domain expertise. There seems to be another camp that believes machine learning algorithms can indeed automate most of the feature selection/engineering tasks and several start-ups are out to prove this very thing. (I have had discussions with a couple of individuals that purport their methodology does exactly that but they were closely guarded secrets.) Let's say that you have several hundred candidate features (independent variables). A way to perform automated feature selection is to compute the univariate information value. However, a feature that appears totally irrelevant in isolation can become important in combination with another feature. So, to get around this, you create numerous combinations of the features. This has potential problems of its own as you may have a dramatically increased computational time and cost and/or overfit your model. Speaking of overfitting, let's pursue it as the next caveat.

Overfitting and underfitting

Overfitting manifests itself when you have a model that does not generalize well. Say that you achieve a classification accuracy rate on your training data of 95 percent, but when you test its accuracy on another set of data, the accuracy falls to 50 percent. This would be considered a high variance. If we had a case of 60 percent accuracy on the train data and 59 percent accuracy on the test data, we now have a low variance but a high bias. This bias-variance trade-off is fundamental to machine learning and model complexity.

Let's nail down the definitions. A bias error is the difference between the value or class that we predict and the actual value or class in our training data. A variance error is the amount by which the predicted value or class in our training set differs from the predicted value or class versus the other datasets. Of course, our goal is to minimize the total error (bias + variance), but how does that relate to model complexity?

For the sake of argument, let's say that we are trying to predict a value and we build a simple linear model with our train data. As this is a simple model, we could expect a high bias, while on the other hand, it would have a low variance between the train and test data. Now, let's try including polynomial terms in the linear model or build decision trees. The models are more complex and should reduce the bias. However, as the bias decreases, the variance, at some point, begins to expand and generalizability is diminished. You can see this phenomena in the following illustration. Any machine learning effort should strive to achieve the optimal trade-off between the bias and variance, which is easier said than done.



We will look at methods to combat this problem and optimize the model complexity, including cross-validation (*Chapter 2, Linear Regression - The Blocking and Tackling of Machine Learning*, through *Chapter 7, Neural Networks*) and regularization (*Chapter 4, Advanced Feature Selection in Linear Models*).

Causality

It seems a safe assumption that the proverbial correlation does not equal causation—a dead horse has been sufficiently beaten. Or has it? It is quite apparent that correlation-to-causation leaps of faith are still an issue in the real world. As a result, we must remember and convey with conviction that these algorithms are based on observational and not experimental data. Regardless of what correlations we find via machine learning, nothing can trump a proper experimental design. As Professor Domingos states:

If we find that beer and diapers are often bought together at the supermarket, then perhaps putting beer next to the diaper section will increase sales. But short of actually doing the experiment it's difficult to tell."

— Domingos, P., 2012)

In *Chapter 11, Time Series and Causality*, we will touch on a technique borrowed from econometrics to explore causality in time series, tackling an emotionally and politically sensitive issue.

Enough of my waxing philosophically; let's get started with using R to master machine learning! If you are a complete novice to the R programming language, then I would recommend that you skip ahead and read the appendix on using R. Regardless of where you start reading, remember that this book is about the journey to master machine learning and not a destination in and of itself. As long as we are working in this field, there will always be something new and exciting to explore. As such, I look forward to receiving your comments, thoughts, suggestions, complaints, and grievances. As per the words of the Sioux warriors: Hoka-hey! (Loosely translated it means forward together)

What this book covers

Chapter 1, A Process for Success - shows that machine learning is more than just writing code. In order for your efforts to achieve a lasting change in the industry, a proven process will be presented that will set you up for success.

Chapter 2, Linear Regression - The Blocking and Tackling of Machine Learning, provides you with a solid foundation before learning advanced methods such as Support Vector Machines and Gradient Boosting. No more solid foundation exists than the least squares linear regression.

Chapter 3, Logistic Regression and Discriminant Analysis, presents a discussion on how logistic regression and discriminant analysis is used in order to predict a categorical outcome.

Chapter 4, Advanced Feature Selection in Linear Models, shows regularization techniques to help improve the predictive ability and interpretability as feature selection is a critical and often extremely challenging component of machine learning.

Chapter 5, More Classification Techniques – K-Nearest Neighbors and Support Vector Machines, begins the exploration of the more advanced and nonlinear techniques. The real power of machine learning will be unveiled.

Chapter 6, Classification and Regression Trees, offers some of the most powerful predictive abilities of all the machine learning techniques, especially for classification problems. Single decision trees will be discussed along with the more advanced random forests and boosted trees.

Chapter 7, Neural Networks, shows some of the most exciting machine learning methods currently used. Inspired by how the brain works, neural networks and their more recent and advanced offshoot, Deep Learning, will be put to the test.

Chapter 8, Cluster Analysis, covers unsupervised learning. Instead of trying to make a prediction, the goal will focus on uncovering the latent structure of observations. Three clustering methods will be discussed: hierarchical, k-means, and partitioning around medoids.

Chapter 9, Principal Components Analysis, continues the examination of unsupervised learning with principal components analysis, which is used to uncover the latent structure of the features. Once this is done, the new features will be used in a supervised learning exercise.

Chapter 10, Market Basket Analysis and Recommendation Engines, presents the techniques that are used to increase sales, detect fraud, and improve health. You will learn about market basket analysis of purchasing habits at a grocery store and then dig into building a recommendation engine on website reviews.

Chapter 11, Time Series and Causality, discusses univariate forecast models, bivariate regression, and Granger causality models, including an analysis of carbon emissions and climate change.

Chapter 12, Text Mining, demonstrates a framework for quantitative text mining and the building of topic models. Along with time series, the world of data contains vast volumes of data in a textual format. With so much data as text, it is critically important to understand how to manipulate, code, and analyze the data in order to provide meaningful insights.

R Fundamentals, shows the syntax functions and capabilities of R. R can have a steep learning curve, but once you learn it, you will realize just how powerful it is for data preparation and machine learning.

What you need for this book

As R is a 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

If you want to learn how to use R's machine learning capabilities in order to solve complex business problems, then this book is for you. An experience with R and a working knowledge of basic statistical or machine learning will prove helpful.

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

```
cor(x1, y1) #correlation of x1 and y1  
[1] 0.8164205  
  
> cor(x2, y1) #correlation of x2 and y2  
  
[1] 0.8164205
```

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: Clicking the **Next** button moves you to the next screen.



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 from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. 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.

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/4527OS_ColouredImages.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.

eBooks, discount offers, and more

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 customercare@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.

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 Carroll

"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 and specifically, its implementation and making the changes happen. The smartest people, best software, and best algorithm do not guarantee success, no matter how 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 soft, 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 skillsets. 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, 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 any 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, and without question, I'm guilty of every sin by 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 10 and a half years.

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

The process

The CRISP-DM process was designed specifically for the data mining. However, it is flexible and thorough enough that it can be applied to any analytical project, whether it is predictive analytics, data science, or machine learning. Don't be intimidated by the numerous list 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, which facilitate its flexibility:

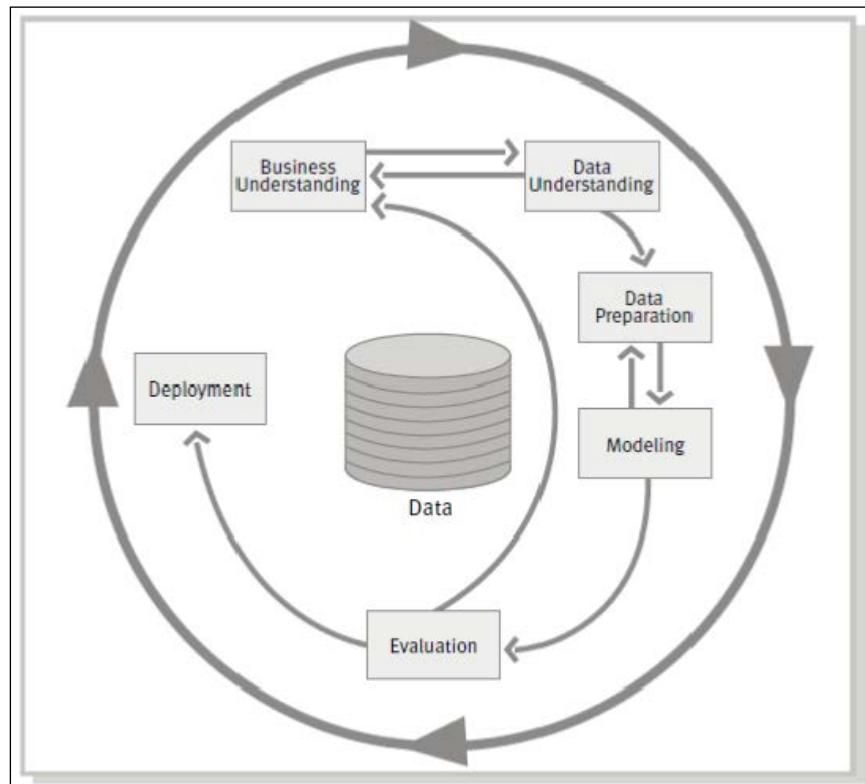


Figure from CRISP-DM 1.0, Step-by-step data mining guide

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 the detailed level of 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 the process steps will be used in the later chapters as a framework in the actual application of the machine learning methods in general and the R code specifically.

Business understanding

One cannot underestimate how important this first step of 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. Identify the business objective
2. Assess the situation
3. Determine the analytical goals
4. Produce a project plan

Identify 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 ponder 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 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, in ill-defined problems, one should go forward with an understanding of what will happen next in terms of resource commitment based on the various outcomes of hypothesis exploration.

Another thing to consider in this task is to manage expectations. There is no such thing as a perfect data, no matter what its depth and breadth is. 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 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, conduct machine learning
- **When:** This is the deadline
- **Where:** This could be geographical; by function, department, initiative, and so on
- **Why:** This is the purpose of doing 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.

Assess 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 decisions to be made.

A couple of points here. When examining the resources that are available, do not neglect to scour the records of the past and current projects. Odds are someone in the organization has 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 the 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.

Determine 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.

Produce 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 your hands on the data. The tasks in this process consist of the following:

1. Collect the data
2. Describe the data
3. Explore the data
4. Verify the data quality

This step is the classic case of ETL is **Extract, Transform, Load**. 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 if the variables are sparse and identify the extent to which the data may be missing. This may drive the learning method that you use and/or 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 an incomplete data collection, cases where unintended IT issues led to errors in the data, or there were planned changes in the business rules. This is critical in the time series where often business rules change over time on how the data is classified. 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 the heartache later on and make one.

Data preparation

Almost there! This step has the following five tasks:

1. Select the data
2. Clean the data
3. Construct the data
4. Integrate the data
5. Format 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 unforgivable 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 money step.

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. Select a modeling technique
2. Generate a test design
3. Build a model
4. Assess a model

Oddly, this process step includes the considerations that you have already thought of and prepared for. In the first step, one will need at least a modicum of an idea about how they 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. A 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 work that has been done and the model selected at this point meets the business objective. Ask yourself and others, have we achieved the 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. Evaluate the results
2. Review the process
3. Determine 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 are 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 of this step:

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

After the deployment and monitoring/maintenance is underway, it is crucial for yourself and those that 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.

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 did 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 happens

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 the possible modeling techniques but also to 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:

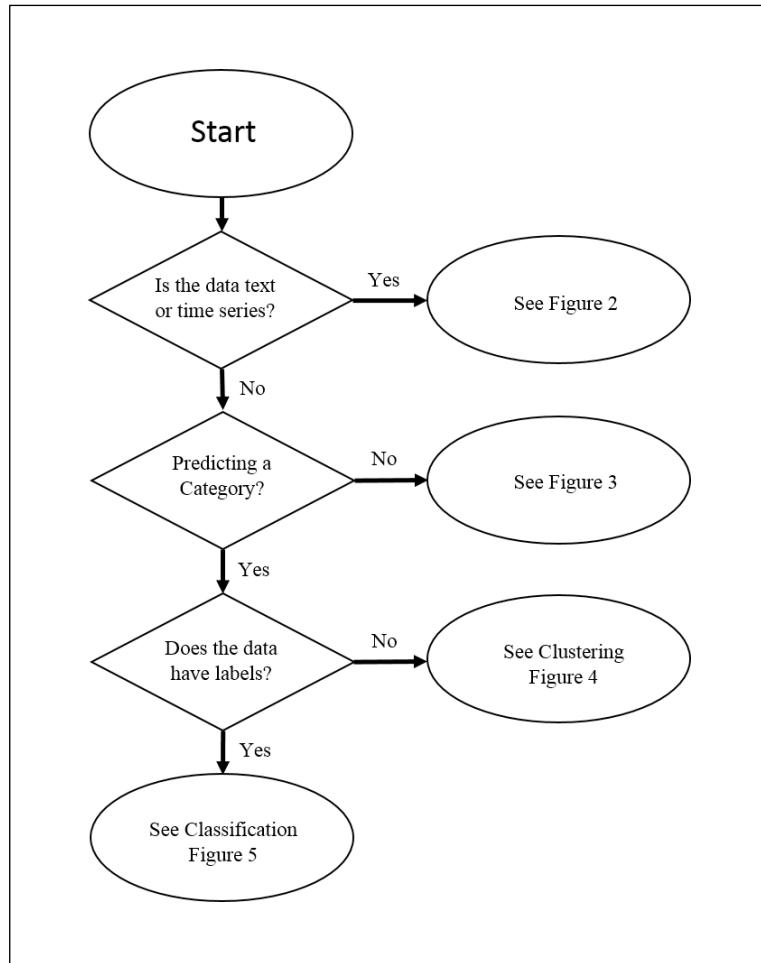


Figure 1

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

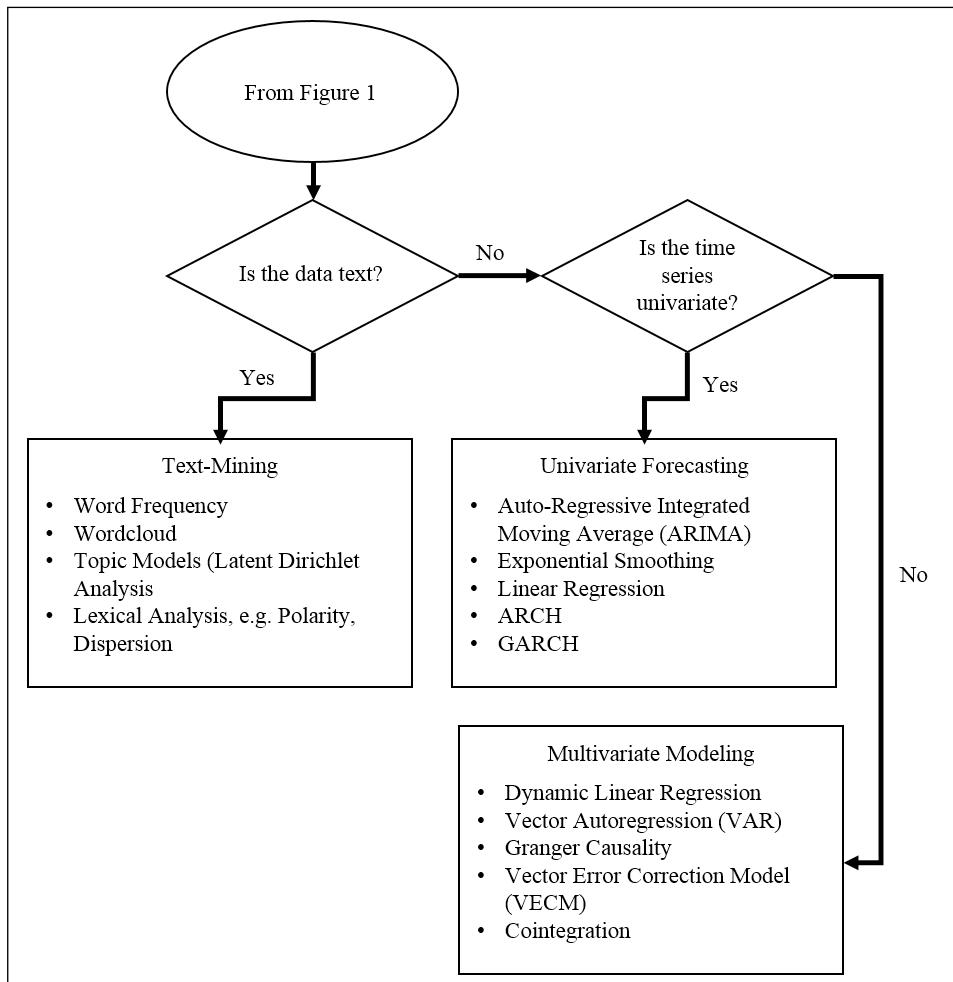


Figure 2

A Process for Success

In this branch of the algorithm, you do not have a text or the time series data. Additionally, you are not trying to predict what category the observations belong to.

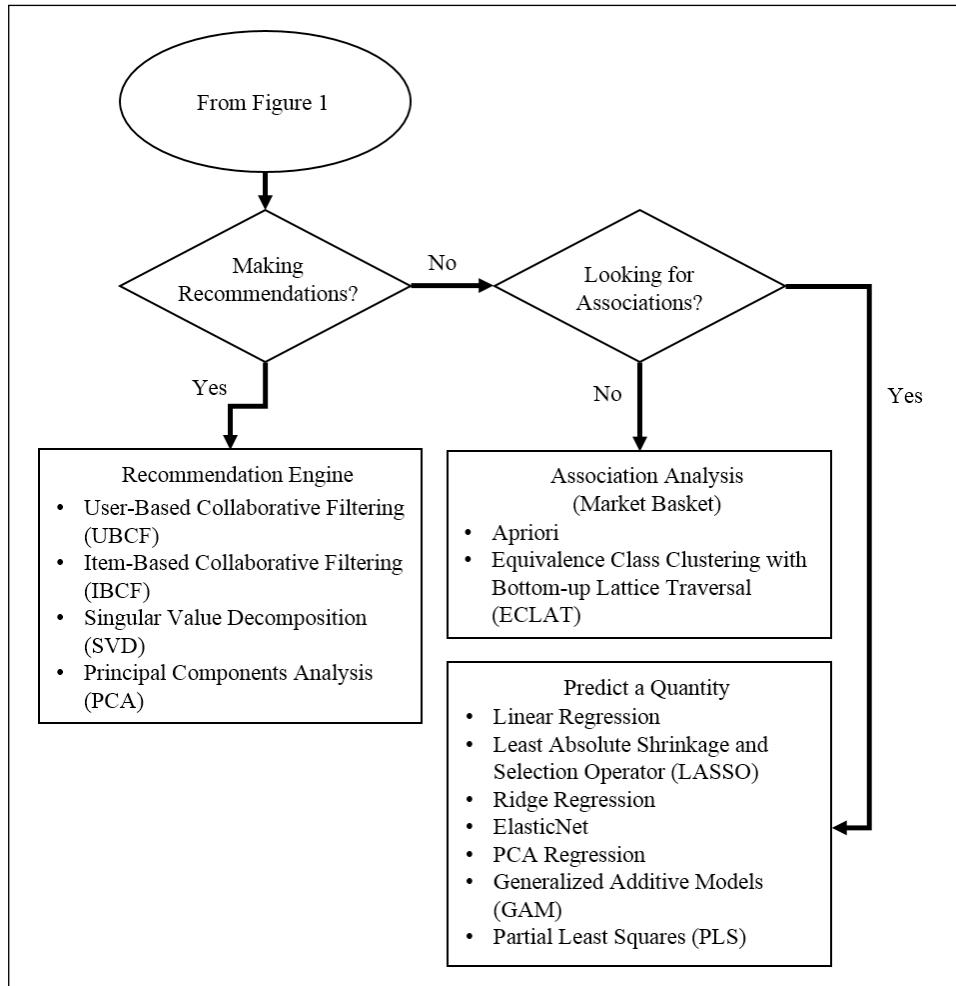


Figure 3

To get to this section, you would 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:

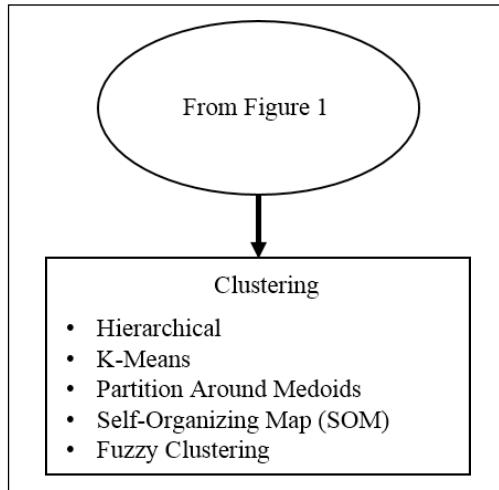


Figure 4

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

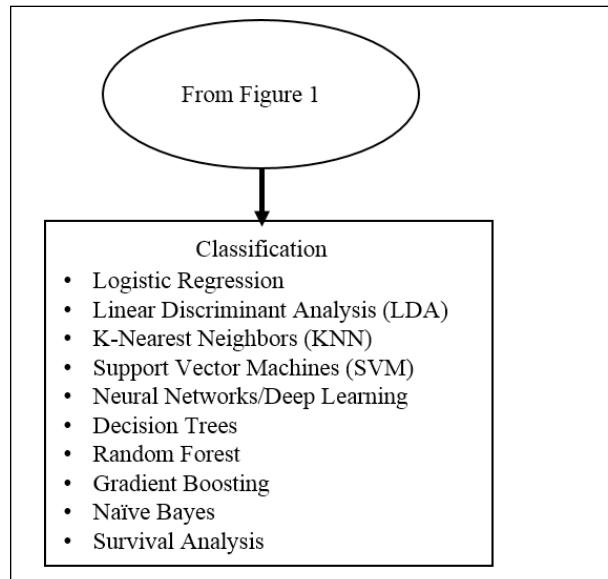


Figure 5

Summary

This chapter was about how to set yourself and your team up 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 process step and the tasks in each step were enumerated. More than that, the commentary provides some techniques and considerations to help in 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 in identifying 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 the tried and tested linear regression, which uses the **least squares approach** to predict a quantitative outcome. In fact, one could 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 point for starting start 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 on 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, $Y = B_0 + B_1x + e$. We can state it as the expected value of Y being a function of the parameters B_0 (the intercept) plus B_1 (the slope) times x , plus an error term. 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 Y_1 and Y_2 equal to 10 and 20 respectively, along with the predictions of y_1 and y_2 as 12 and 18. To calculate RSS, we add the squared differences $RSS = (Y_1 - y_1)^2 + (Y_2 - y_2)^2$, 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, do so with a jaded 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, a look at the famous Anscombe dataset available in R is in order. 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 with the following R code should you have a similar need. It is a brief tangent 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 of 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

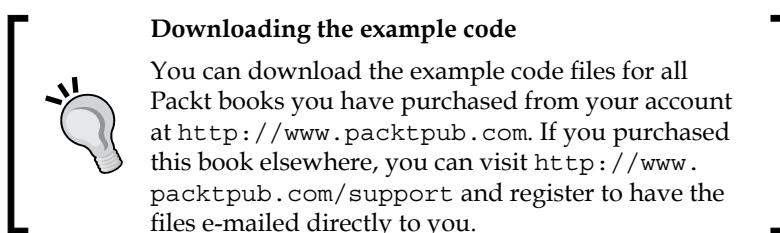
> plot(x1, y1, main="Plot 1")

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

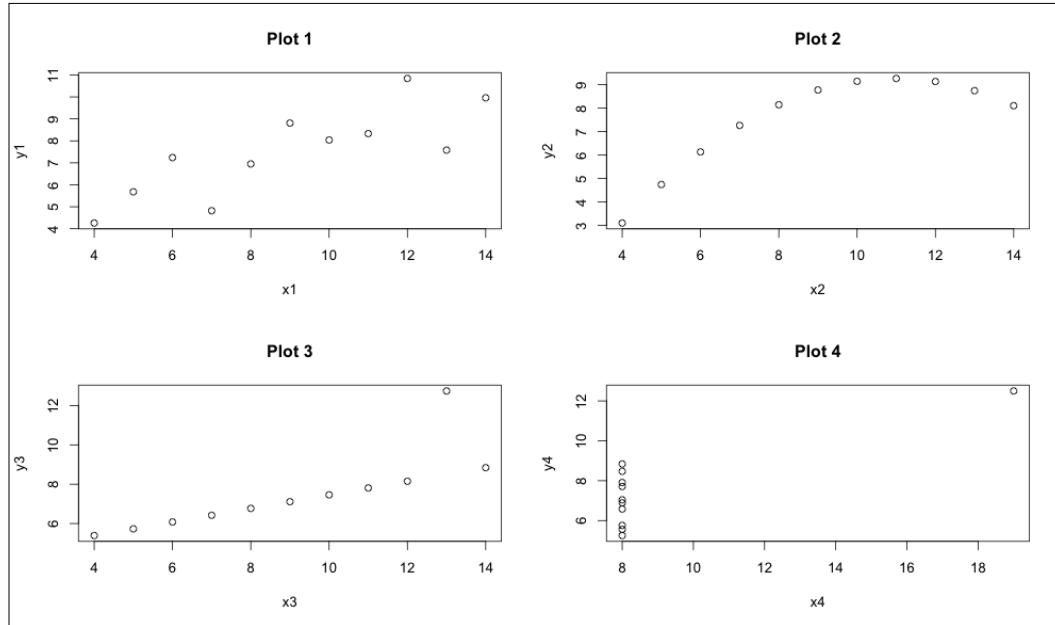
> plot(x3, y3, main="Plot 3")

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

```



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 the one outlier. There you have it, a cautionary tale of sorts.

Business understanding

The data collected measures two variables. The goal is to model the water yield (in inches) of the Snake River Watershed in Wyoming 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 the much needed irrigation water for the farms and ranches of several western states. The dataset `snake` is available in the `alr3` package (note that `alr` stands for applied linear regression):

```
> install.packages("alr3")
> library(alr3)
> data(snake)
> attach(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
```

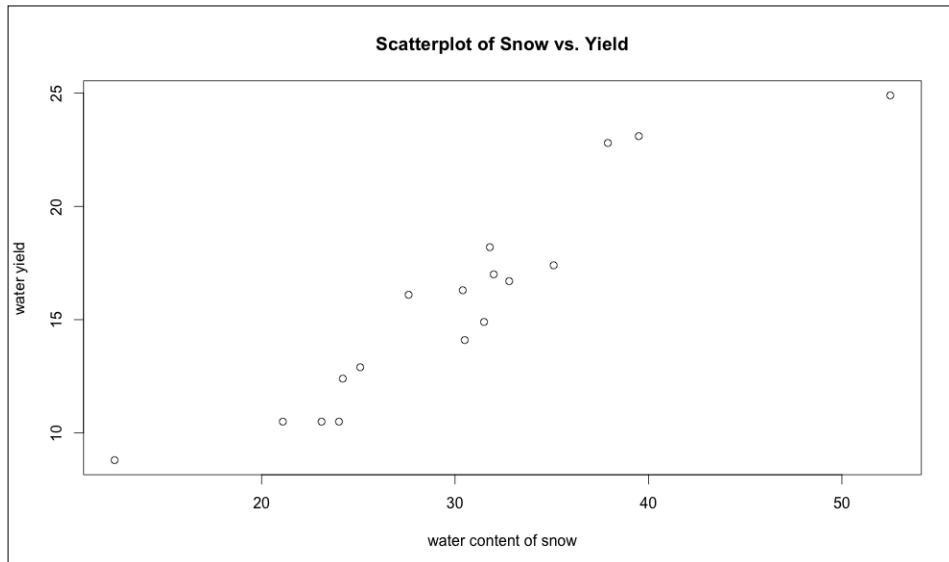
Now that we have 17 observations, data exploration can begin. But first, let's change X and Y into meaningful variable names, as follows:

```
> names(snake) = c("content", "yield")
> attach(snake) #reattach 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, a transformation of the data or deletion of 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 one unit change in the `content`, the `yield` will increase by `0.49808` units. `F-statistic` is used to test the null hypothesis that the model coefficients are all 0.

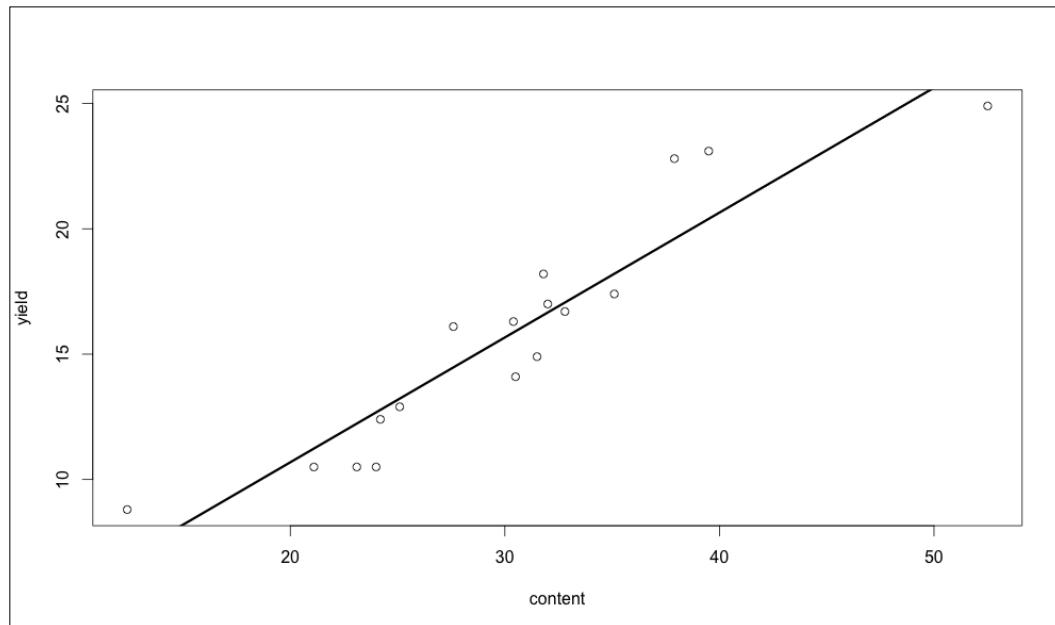
Since the p-value is highly significant, we can reject the null and move on to the t-test for content, which tests the null hypothesis that it is 0. Again, we can reject the null. Additionally, we can see Multiple R-squared and Adjusted R-squared values. The Adjusted R-squared will be covered under the 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:



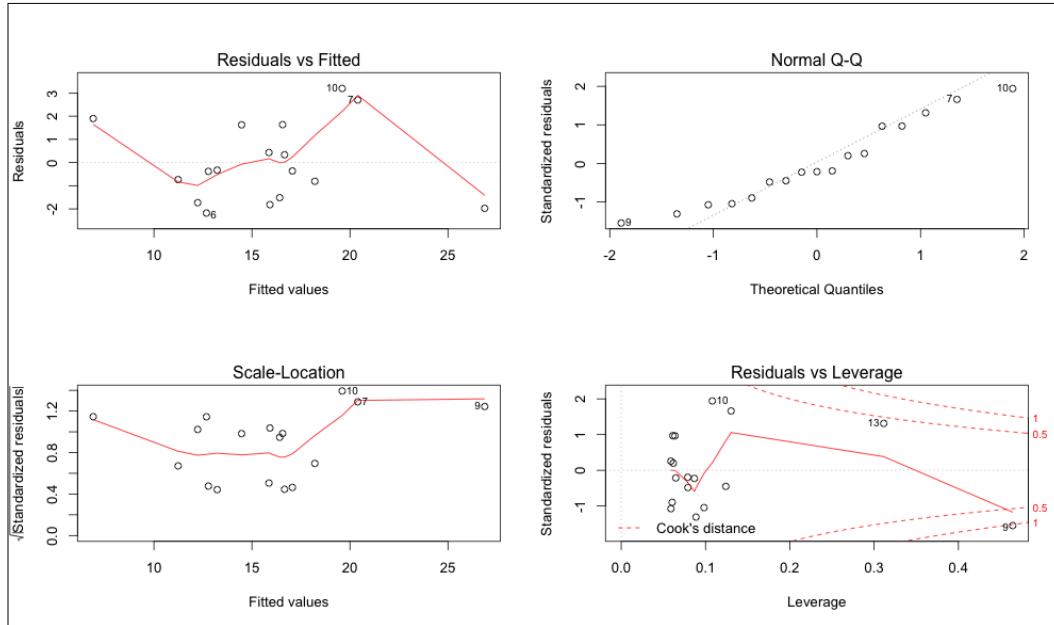
A linear regression model is only as good as the validity of its assumptions, which can be summarized as follows:

- **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 the X or Y may solve the problem.
- **Non-correlation of errors:** A common problem in the time series and panel data where $e_n = \text{beta}_{n-1} e_{n-1}$; 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 the errors is constant across the 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 the 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, must be removed prior to fitting a model using linear regression; this again can lead to a biased estimate.

As we are building a univariate model not dependent on 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 nonlinearity. 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 will cluster close together on the left side of the plot and 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.

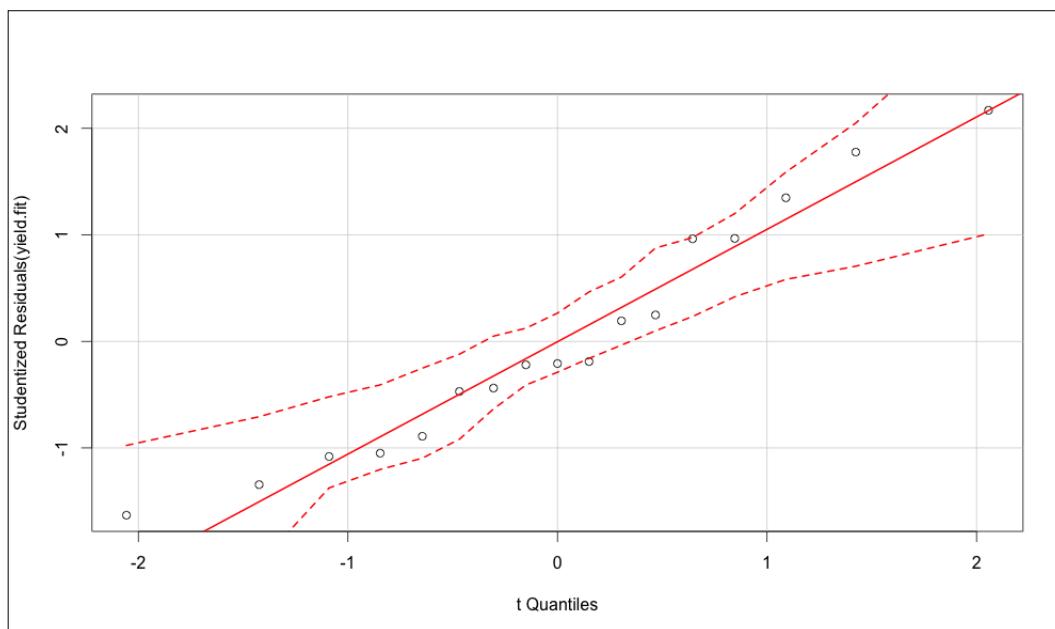
The **Normal Q-Q** plot in the upper-right corner helps us to determine if the residuals are normally distributed. The **Quantile-Quantile (Q-Q)**, represent the quantile values of one variable plotted against the quantile values of another. It appears that the outliers (observations 7, 9, and 10), may 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 one 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 of 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 the 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 would be 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 as follows:

```
> 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 some confidence to select the model with all the observations. 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 the question if in the real world you would ever have just one predictor variable; that is, indeed, fair. 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`. Lately, the severe drought in Southern California has caused much alarm. Even the Governor, Jerry Brown, has begun 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 we are not concerned with what year the observations occurred, 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 worked with the head() function as follows:

```
> 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 prefer the **ellipse** method, but feel free to experiment with the various methods. 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)

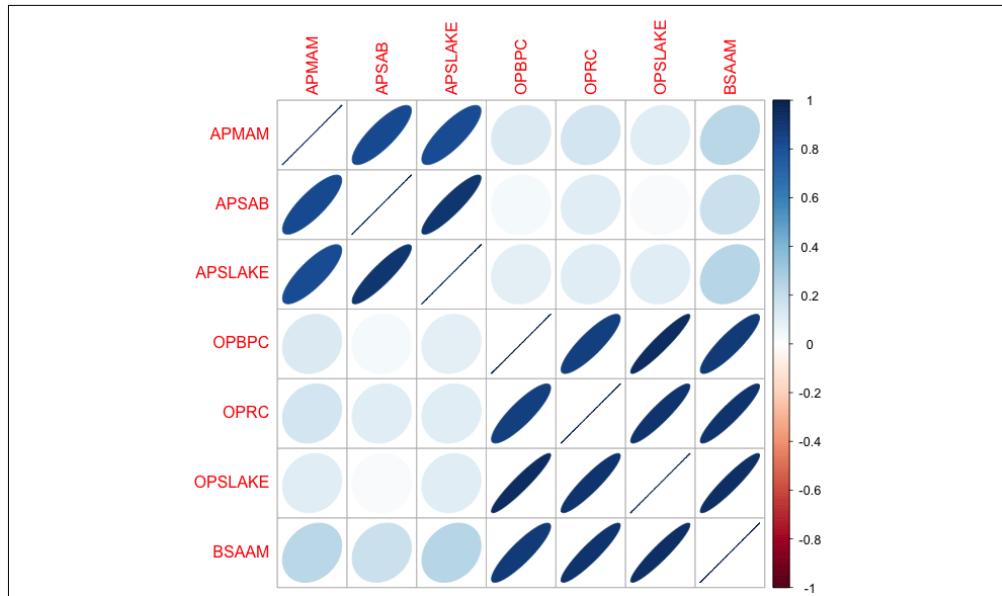
> water.cor

          APMAM      APSAB      AP SLAKE      OPBPC
APMAM  1.0000000  0.82768637  0.81607595  0.12238567
APSAB  0.8276864  1.00000000  0.90030474  0.03954211
AP SLAKE 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
OP SLAKE 0.1075421  0.02961175  0.10058669  0.94334741
BSAAM  0.2385695  0.18329499  0.24934094  0.88574778
          OPRC      OP SLAKE      BSAAM
APMAM  0.1544155  0.10754212  0.2385695
APSAB  0.1056396  0.02961175  0.1832950
AP SLAKE 0.1063836  0.10058669  0.2493409
OPBPC  0.8647073  0.94334741  0.8857478
OPRC   1.0000000  0.91914467  0.9196270
OP SLAKE 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 OP SLAKE 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 multicollinearity. The correlation plot matrix provides a nice visual of the correlations as follows:

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

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. The 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 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 bottom line 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, 23 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: $fit = lm(y \sim x_1 + x_2 + x_3...x_n)$. 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 with univariate regression, we examine the p-value on the F-statistic to see if at least one of the coefficients is not zero and, 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=socal.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, as follows:

```
> 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 like the number of wins for the Los Angeles Lakers and RSS would decrease and R-squared would increase. The amount would likely be minuscule, 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, 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 that 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

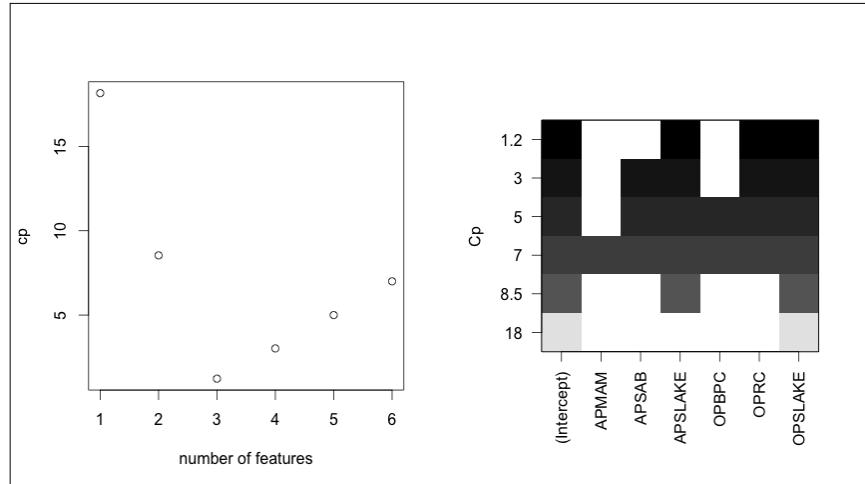
In a linear model, AIC and Cp are proportional to each other, so we will only concern ourselves with Cp, which follows the output available in the `leaps` package. BIC tends to select the models with fewer variables than Cp, so we will compare both. To do so, we can create and analyze two plots side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

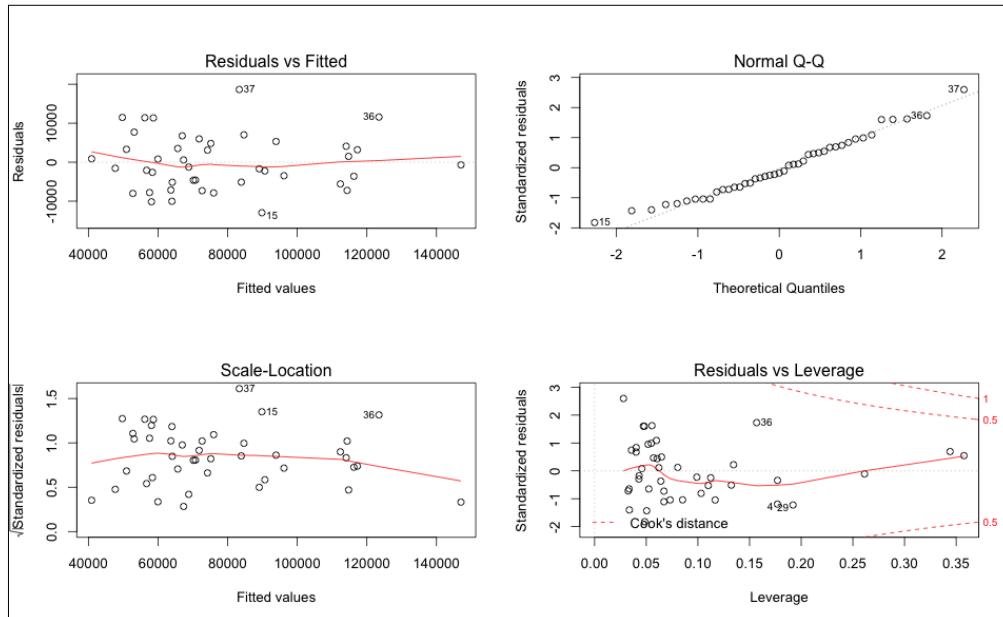
```

> 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 fit 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 one, which means no collinearity at all. There are no hard and fast rules, but in general, a VIF value that exceeds 5 or 10 indicates a problematic amount of collinearity (James, 2013). A precise value is difficult to select, because there is no hard statistical cut-off point for when multicollinearity 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:

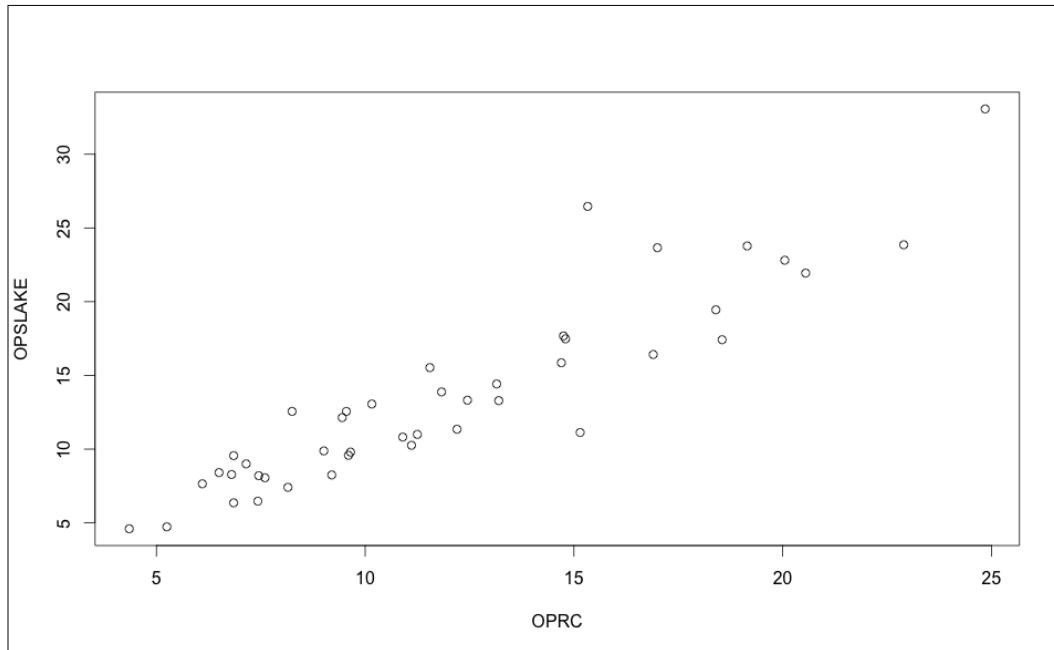
```
> vif(best.fit)

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 five) based upon the correlation analysis. A plot of the two variables drives the point home, as seen in the following image:

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

The output of 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 AP SLAKE and OPSLAKE produced a value of 0.90, while adding OPRC 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, as follows:

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

```
> summary(fit.2)
```

Call:

```
lm(formula = BSAAM ~ AP SLAKE + 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

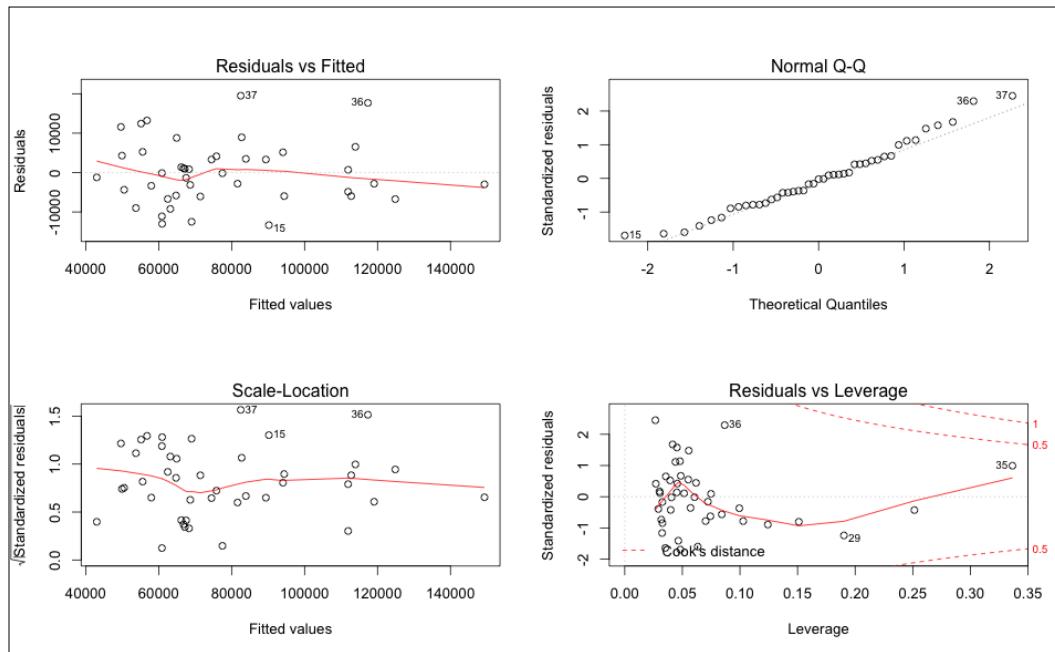
Multiple R-squared: 0.9049, Adjusted R-squared: 0.9002

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 as follows:

```
> 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 questions, 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 hypotheses that the error variances are zero versus the alternative of not zero.

```
> library(lmtest)

> bptest(fit.2)

studentized Breusch-Pagan test
```

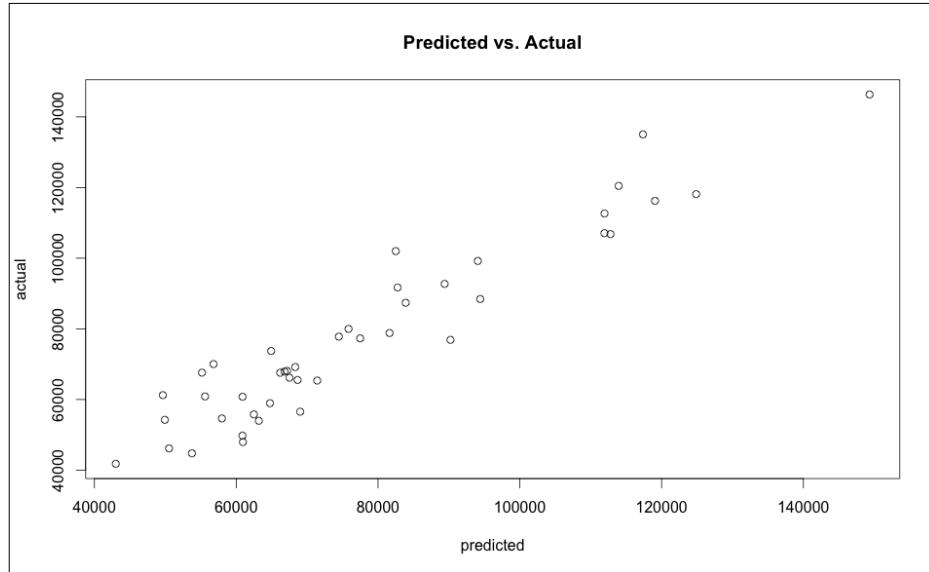
```
data: fit.2
BP = 0.0046, df = 2, p-value = 0.9977
```

We do not have evidence to reject the null that implies the error variances are zero because `p-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 19145 (the intercept) plus 1769 times the measurement at `APSLAKE` plus 3690 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, and 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 in the following code snippet:

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

> socal.water["Forecast"] = NA #create a vector for the predictions named Forecast, first using NA to create empty observations

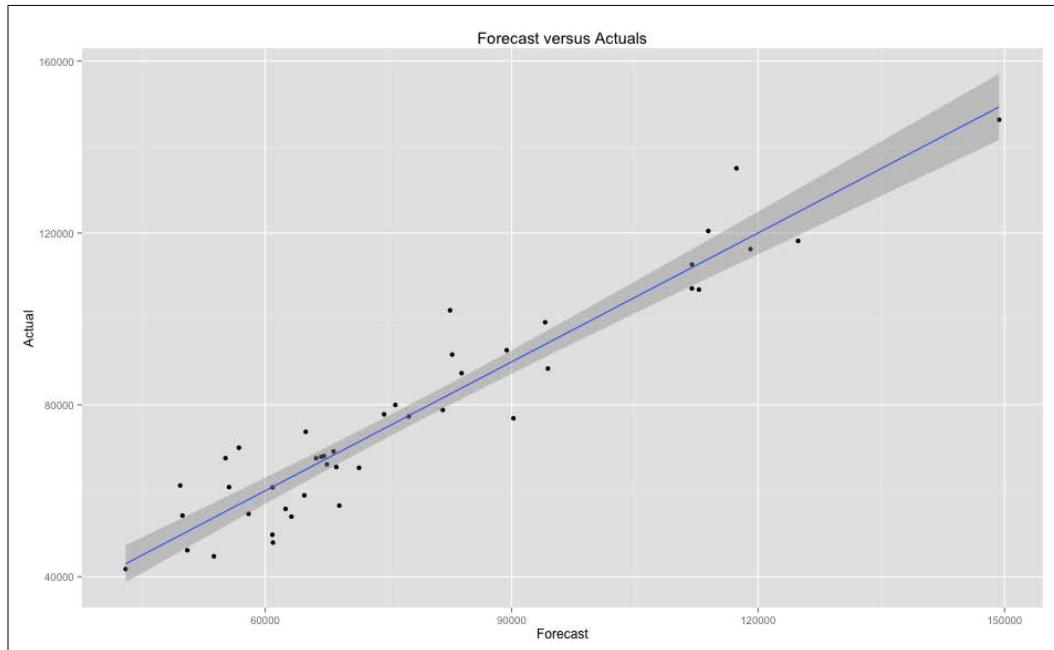
> socal.water$Forecast = predict(fit.2) #populate Forecast with the predicted values
```

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

```
> library(ggplot2)

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

The output of the preceding code snippet 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 would this be necessary at all? It comes down to the bias-variance tradeoff. 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 say? Well, if you 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, the 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 one 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 feature

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 is equal to zero and female is equal to one, 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 the levels, you would fall into the dummy variable trap, which results in perfect multicollinearity.

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 by 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 ...
 $ ShelveLoc  : 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 than when the location is bad, given the intercept of 4.89662. To see how R codes the indicator features, you can use the `contrasts()` function as follows:

```
> contrasts(Carseats$ShelveLoc)
```

	Good	Medium
Bad	0	0
Good	1	0
Medium	0	1

Interaction term

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 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 as lstat, and the age of the home in years, which is termed as 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 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 ...
 $ chas    : int  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 ...
 $ 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 have had an in-depth look at the simple yet extremely effective method of linear regression to predict a quantitative response. The later chapters will cover more advanced techniques, but many of them are mere extensions of what we have learned in this chapter. We've 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, you can also include quantitative variables and interaction terms among the features. Indeed, it 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, 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, purchases versus does not purchase, tumor is benign versus malignant) or multinomial categories (education level or eye color). Regardless of whether or not the outcome of interest is binary or multinomial, the task of the analyst is to predict the probability that an observation would belong to which category of the outcome variable. In other words, we develop an algorithm in order to classify the observations.

To begin exploring the 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 about 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 the logistic regression models. We will also begin examining methods so as to select features and the most appropriate model. Next, we will discuss about both linear and quadratic discriminant analyses and comparing and contrasting these with logistic regression. Then, building predictive models on the breast cancer data will follow. Finally, we will wrap it all up by looking at 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 the subsequent chapters.

Classification methods and linear regression

So, why can't we just use the least squares 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 zero is equal to mild, one is equal to moderate, and two 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 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 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:

$$\text{Probability of } Y = \frac{e^{B_0 + B_1 x}}{1 + e^{B_0 + B_1 x}}$$

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 the 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 finding the estimates for B_0 and B_1 that 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 of 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 the 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 most likely will require surgery, 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 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: This is the sample code number
- v1: This is the thickness
- v2: This is the uniformity of the cell size
- v3: This is the uniformity of the cell shape
- v4: This is the marginal adhesion
- v5: This is the single epithelial cell size
- v6: This is the bare nucleus (16 observations are missing)
- v7: This is the bland chromatin
- v8: This is the normal nucleolus
- v9: This is the mitosis
- class: This is the tumor diagnosis 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 will first load the library and then the dataset, and using the `str()` function, will examine the underlying structure of the data, as follows:

```
> 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. 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. 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 two percent of all the observations. A thorough discussion of how to handle the missing data is outside the scope of this chapter. 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)
```

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)
```

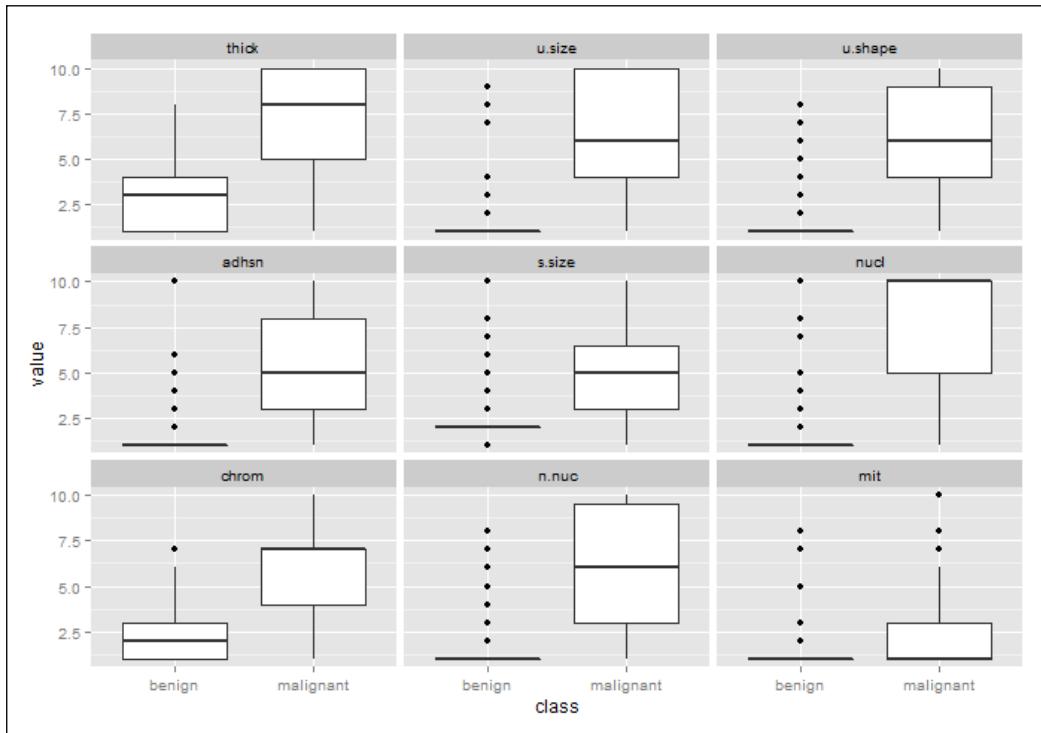
The following code melts the data by their values into one overall feature and groups them by class:

```
> 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 image, 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!

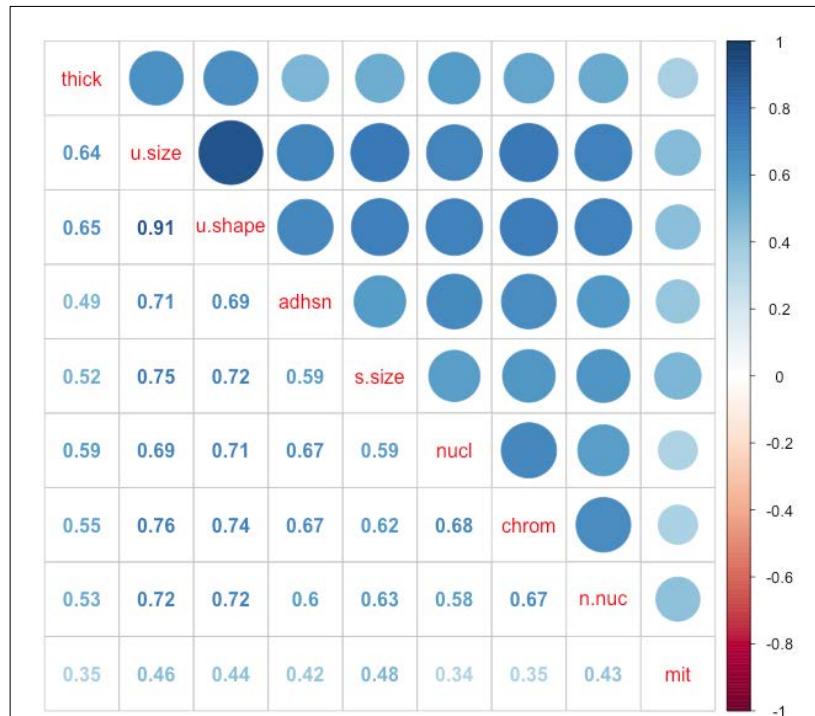
With all of our features quantitative, we can also do a correlation analysis as we did with linear regression. Collinearity with logistic regression can bias our estimates just as we discussed with linear regression. Let's load the `corrplot` package and examine the correlations as we did in the previous 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 a 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, but 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 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, and 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 both linear and quadratic discriminant analyses.

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 that fits 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. The 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 the 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 one, it indicates that as the feature increases, the odds of the outcome increase. Conversely, a value less than one 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 the data exploration was the potential issue of multicollinearity. 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
```

The `contrasts()` function allows us to confirm that the model was created with `benign` as 0 and `malignant` as 1:

```
> contrasts(train$class)
  malignant
benign      0
malignant    1
```

Next, in order to create a meaningful table of the `fit` model that is referred to as a confusion matrix, we will need to produce a vector that codes the predicted probabilities as either `benign` or `malignant`. We will see that in the other packages this is not necessary, but for the `glm()` function it is necessary as it defaults to a predicted probability and not a `class` prediction. There are a number of ways to do this. Using the `rep()` function, a vector is created with all the values called `benign` and a total of 474 observations, which match the number in the training set. Then, we will code all the values as `malignant` where the predicted probability was greater than 50 percent, as follows:

```
> train$predict = rep("benign", 474)

> train$predict[train$probs>0.5] = "malignant"
```

The `table()` function produces our confusion matrix:

```
> table(train$predict, train$class)
  benign malignant
benign      294       7
malignant     8      165
```

The rows signify the predictions and 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. The `mean()` function shows us what percentage of the observations were predicted correctly, as follows:

```
> mean(train$predict==train$class)
[1] 0.9683544
```

It seems we have done a fairly good job with our almost 97 percent prediction rate on the training set. As we previously discussed, we must be able to accurately predict the 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$prob = predict(full.fit, newdata=test, type="response")
```

In the preceding code, we just specified that we want to predict the `test` set with `newdata=test`.

As we did with the training data, we need to create our predictions for the `test` data:

```
> test$predict = rep("benign", 209)
```

```
> test$predict[test$prob>0.5]="malignant"
```

```
> table(test$predict, test$class)
```

	benign	malignant
benign	139	2
malignant	3	65

```
> mean(test$predict==test$class)
```

```
[1] 0.9760766
```

It appears that 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 see if there is room for improvement. Imagine that you or your loved one is a patient that 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?

Logistic regression with cross-validation

The purpose of cross-validation is to improve our prediction of the `test` set and minimize the chance of over fitting. 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** and 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 selected. You can also do the **Leave-One-Out-Cross-Validation (LOOCV)**, where K is equal to one. 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 requires 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. All you have to do is add a vector to the `train` set, code it all with zeroes, and then code it to one where the `class` vector is equal to `malignant`, as follows:

```
> train$y=rep(0,474)

> train$y[train$class=="malignant"]=1
```

A quick double check is required in order to confirm that it worked:

```
> head(train[,13])
[1] 0 0 1 0 0 0
```

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 by simply deleting any unwanted columns. The outcome is column 10, and if in the process of doing other analyses we added columns 11 and 12, they must be removed as well:

```
> biopsy.cv = train[,-10:-12]

> head(biopsy.cv)
  thick u.size u.shape adhsn s.size nucl chrom n.nuc mit y
1      5       1       1       1       2       1       3       1       1   0
3      3       1       1       1       2       2       3       1       1   0
6      8      10      10       8       7      10       9       7       1   1
7      1       1       1       1       2      10       3       1       1   0
9      2       1       1       1       2       1       1       1       5   0
10     4       2       1       1       2       1       2       1       1   0
```

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 = biopsy.cv`, points to our properly formatted data frame. `IC="CV"` tells the package that the information criterion to use is cross-validation. `CVArcs` are the CV arguments that we want to use. The `HTF` method is K-fold, which is followed by the number of folds of `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 train and test sets. The `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)
```

Using the same style of code as we did in the last section, we will save the probabilities and create the confusion matrices, as follows:

```
> train$cv.probs = predict(reduce.fit, type="response")  
  
> train$cv.predict = rep("benign", 474)  
  
> train$cv.predict[train$cv.probs>0.5] = "malignant"  
  
> table(train$cv.predict, train$class)  
  
            benign malignant  
benign      294        9  
malignant      8      163
```

Interestingly, the reduced feature model had two more false negatives than the full model.

As before, the following code allows us to compare the predicted labels versus the actual ones:

```
> test$cv.probs = predict(reduce.fit, newdata=test, type="response")

> test$predict = rep("benign", 209)

> test$predict[test$cv.probs>0.5]="malignant"

> table(test$predict, test$class)

      benign malignant
benign       139        5
malignant      3       62
```

The reduced feature model again produced more false negatives than when all the features were included. This is quite disappointing, 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= biopsy.cv, 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")
```

```
> test$bic.predict = rep("benign", 209)

> test$bic.predict[test$bic.probs>0.5]="malignant"

> table(test$bic.predict, test$class)

      benign malignant
benign       138        1
malignant      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 most interpretable model given the 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 would likely vary wildly from one sample to another (James, 2013). DA does not suffer from this problem, and as a result, may outperform and be more generalizable 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 to logistic regression, we will explore DA, both **Linear Discriminant Analysis (LDA)** and **Quadratic Discriminant Analysis (QDA)**.

DA utilizes Bayes' 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:

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

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 of which class to assign an observation is by choosing the class with the larger probability (the posterior probability).

The process of attaining the 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 matrix is 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:

- $\pi_k = \# \text{ of samples in class } k / \text{total sample size}$ is the prior probability of a randomly chosen observation in the k th class.
- $f_k(X) = P(X = x | Y = k)$ is the density function of an observation that comes from the k th class. We will assume that this comes from a normal (Gaussian) distribution; with multiple features, the assumption is that it comes from a multivariate Gaussian distribution.
- Using $p_k(X) = \text{probability of } Y \text{ given } x$, we can adjust Bayes' theorem accordingly.

- $p_x(X) = \pi_k f_k(X) / \sum_{j=1}^k \pi_j f_j(X)$ is the posterior probability that an observation comes from the k class when the feature values for this observation are given.
- Assuming that $k=2$ and the prior probabilities are the same, $\pi_1 = \pi_2$, then an observation is assigned to the one class if $2x(\mu_1 - \mu_2) > \mu_1^2 - \mu_2^2$, otherwise it is assigned to the two class. This is known as the decision boundary. DA creates the $k-1$ decision boundaries, that is, with three classes ($k=3$), there will be two decision boundaries.

Even though LDA is elegantly simple, it has the limitation of 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 the observations come from a normal distribution, but 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 is 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. To facilitate the simplicity of this code, we will create new data frames for the LDA by deleting the columns that we had added to the `train` and `test` sets, as follows:

```
> lda.train = train[ , -11:-15]

> lda.train[1:3,]
  thick u.size u.shape adhsn s.size nucl chrom n.nuc mit  class
1      5      1      1      1      2      1      3      1      1  benign
3      3      1      1      1      2      2      3      1      1  benign
6      8     10     10      8      7     10      9      7      1 malignant

> lda.test = test[ , -11:-15]
> lda.test[1:3,]
```

```

thick u.size u.shape adhsn s.size nucl chrom n.nuc mit class
2      5       4       4       5       7     10      3      2     1 benign
4      6       8       8       1       3      4      3      7     1 benign
5      4       1       1       3       2      1      3      1     1 benign

```

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

```

> lda.fit = lda(class~, data=lda.train)

> lda.fit
Call:
lda(class ~ ., data = lda.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

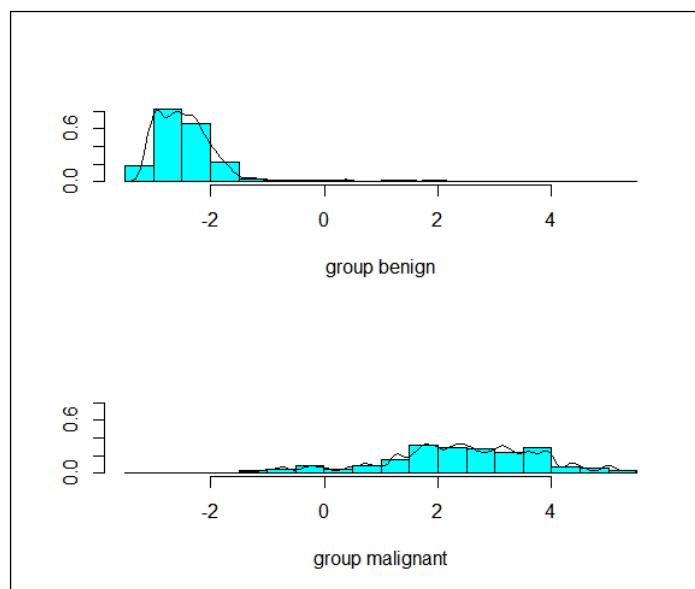
```

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. It is easier to produce the confusion matrix with the help of the following function than with logistic regression:

```
> lda.predict = predict(lda.fit)  
  
> train$lda = lda.predict$class
```

```
> table(train$lda, train$class)

      benign malignant
benign       296      13
malignant      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:

```
> lda.test = predict(lda.fit, newdata = test)

> test$lda = lda.test$class

> table(test$lda, test$class)

      benign malignant
benign       140      6
malignant      2     61
```

That's actually not as bad as I thought, given the poor 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):

```
> mean(test$lda==test$class)
[1] 0.9617225
```

We will now move on to fit a QDA model to data.

In R, QDA is also part of the MASS package and the function is `qda()`. We will use the `train` and `test` sets that we used for LDA. Building the model is rather straightforward and we will store it in an object called `qda.fit`, as follows:

```
> qda.fit = qda(class~, data=lda.train)

> qda.fit
Call:
qda(class ~ ., data = lda.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 as 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:

```
> qda.predict = predict(qda.fit)

> train$qda = qda.predict$class

> table(train$qda, train$class)

      benign malignant
benign        287       5
malignant      15      167
```

We can quickly tell that QDA has performed the worst on the training data with the confusion matrix.

We will see how it works on a test set:

```
> qda.test = predict(qda.fit, newdata=test)

> test$qda = qda.test$class

> table(test$qda, test$class)

      benign malignant
benign        132       1
malignant      10      66
```

QDA classified the `test` set poorly with 11 incorrect predictions. In particular, it has a high rate of false positives.

Model selection

What are we to make of all this? We have the confusion matrices 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 the 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:

- $TPR = \text{Positives correctly classified} / \text{total positives}$
- $FPR = \text{Negatives incorrectly classified} / \text{total negatives}$

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 it is a great package that 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 <http://rocr.bioinf.mpi-sb.mpg.de/>.

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

```
> 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$prob, test$class)
```

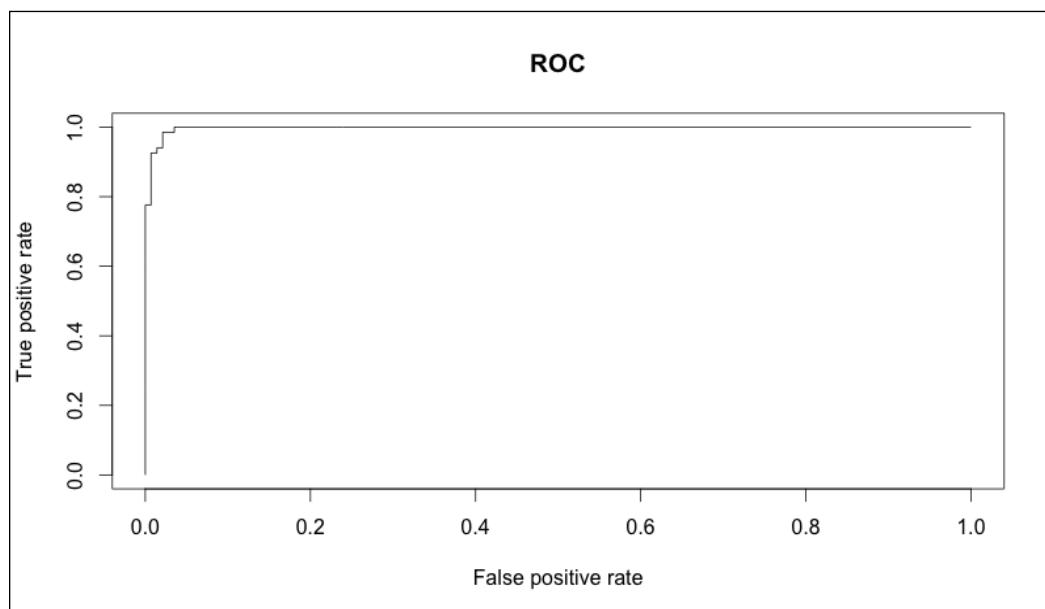
The following is the performance object with the TPR and FPR:

```
> 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 vertical 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 poor performing model and include a legend chart, as follows:

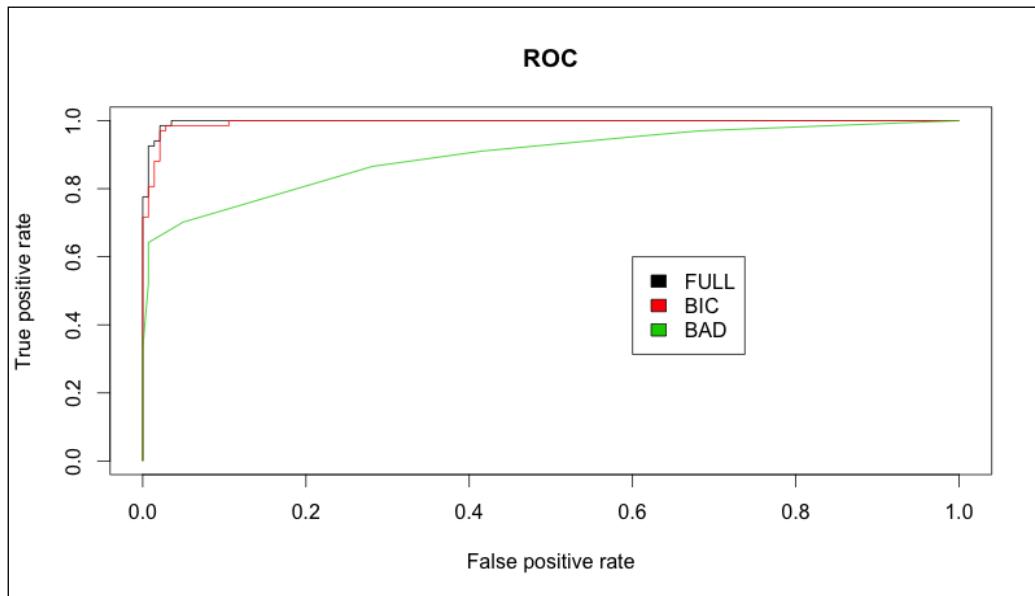
```
> pred.bad = prediction(test$bad, test$class)

> perf.bad = performance(pred.bad, "tpr", "fpr")

> plot(perf.bad, col=3, add=TRUE)

> legend(0.6, 0.6, c("FULL", "BIC", "BAD"), 1:3)
```

We can see that the **FULL** model and **BIC** model are nearly superimposed. As you may recall, previously the only difference in the confusion matrices was the fact that the **BIC** model had one false positive more and one false negative less. It is also quite clear that the **BAD** model performed as poorly as was expected, which can be seen in the following image:



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 is as follows:

```
> auc.full = performance(pred.full, "auc")

> auc.full
An object of class "performance"
Slot "x.name":
[1] "None"

Slot "y.name":
[1] "Area under the ROC curve"
```

```
Slot "alpha.name":  
[1] "none"
```

```
Slot "x.values":  
list()
```

```
Slot "y.values":  
[[1]]  
[1] 0.9972672
```

```
Slot "alpha.values":  
list()
```

The values that we are looking for are under the `Slot "y.values"` section of the output. The AUC for the full model is 0.997. I've abbreviated the output for the other two models of interest, as follows:

```
> auc.bic = performance(pred.bic, "auc")
```

```
> auc.bic
```

```
Slot "y.values":  
[[1]]  
[1] 0.9944293
```

```
> auc.bad = performance(pred.bad, "auc")
```

```
> auc.bad
```

```
Slot "y.values":  
[[1]]  
[1] 0.8962056
```

The AUCs were 99.7 percent for the full model, 99.4 percent for the BIC model, and 89.6 percent for the bad model. So, for all intents and purposes, the full model and the BIC model have no difference in predictive powers between them. What are we to do? A simple solution would be to rerandomize the `train` and `test` sets and try this analysis again, perhaps using a 60/40 split and different randomization seed. However, 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. Maybe there is another option? Let me propose that we can tackle this problem in the upcoming chapters with more complex techniques and 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 the two most common methods: logistic regression and discriminant analysis. Additionally, we began the process of using the ROC charts in order to explore the model selection visually and statistically. We also briefly discussed the model selection and trade-offs that you need to consider. In the future chapters, we will revisit the breast cancer dataset to see if we can improve our predictive ability with more complex techniques.

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

So far, we 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 the machine learning problems. However, newer techniques that have been developed and refined in the last couple of decades or so can improve the predictive ability and interpretability above and beyond the linear models that we've 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 ever have to work 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; but 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 the techniques to 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 a **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 $(RSS + \lambda(\text{normalized coefficients}))$. We will select the λ , which is known as the tuning parameter in our model building process. Please note that if lambda is equal to zero, 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 **2p models** and in large datasets, it may just not be feasible to attempt. In R, we are only fitting one model to each value of lambda and this is therefore far and away more efficient. Another reason goes back to our bias-variance trade-off that is 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 the coefficients works to solve the multicollinearity 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 do not ever 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, alpha, in conjunction with lambda. Alpha will be between 0 and 1 and as before, lambda will regulate the size of the penalty. Please note that an alpha of zero is equal to ridge regression and an alpha of one is equivalent to LASSO. Essentially, we are blending the L1 and L2 penalties by including a second tuning parameter to 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 the 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 with 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 the traditional techniques. We will start by performing best subsets regression to identify the features and use this as a baseline for the comparison.

Business understanding

The Stanford University Medical Center has provided the 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 noncancerous 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; this 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 then 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, as follows:

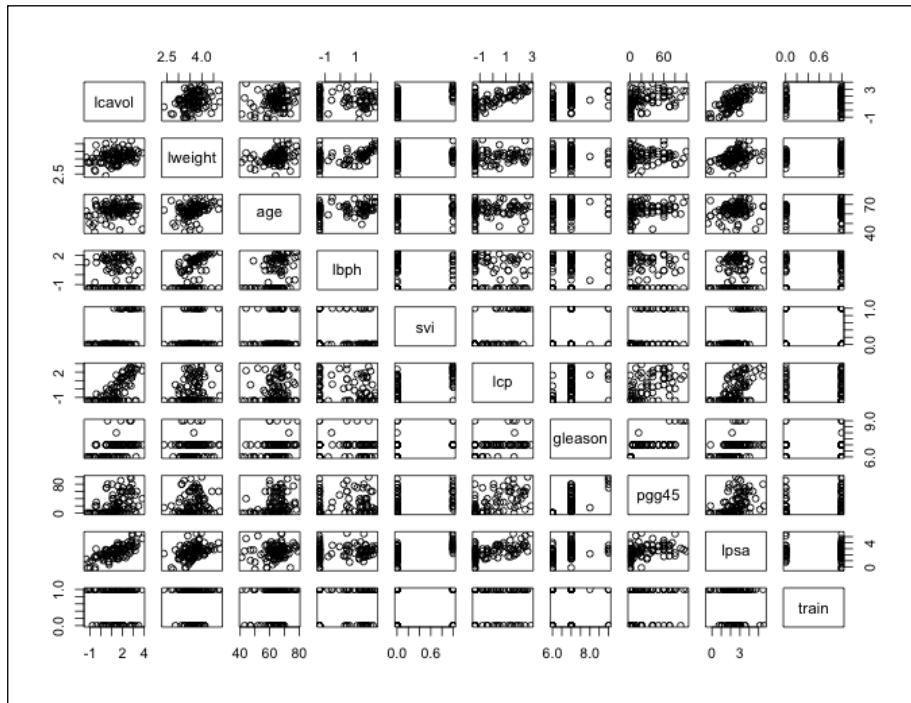
```
> data(prostate)

> 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 ...
 $ lcp     : num -1.39 -1.39 -1.39 -1.39 -1.39 ...
 $ gleason: int 6 6 7 6 6 6 6 6 6 ...
 $ pgg45   : int 0 0 20 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 ten 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)
```

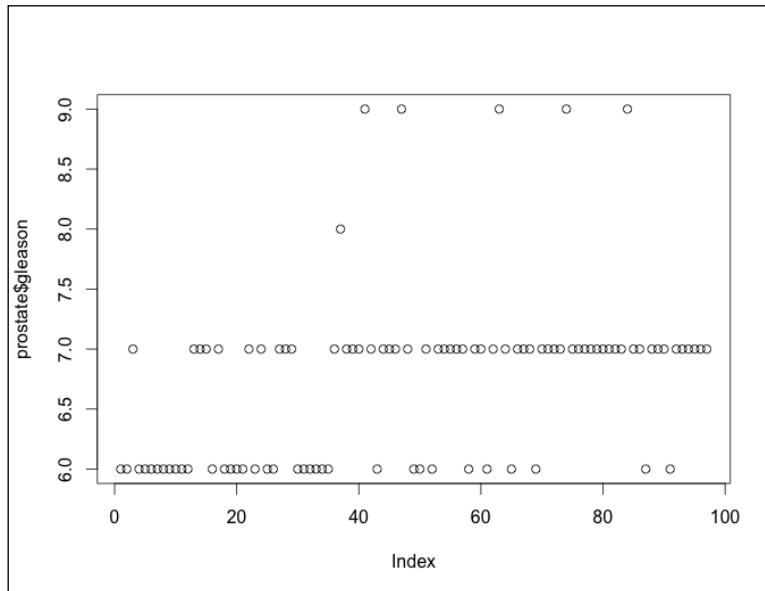
The output of the preceding command is as follows:



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, as follows:

```
> table(prostate$gleason)
```



```
6  7  8  9  
35 56  1  5
```

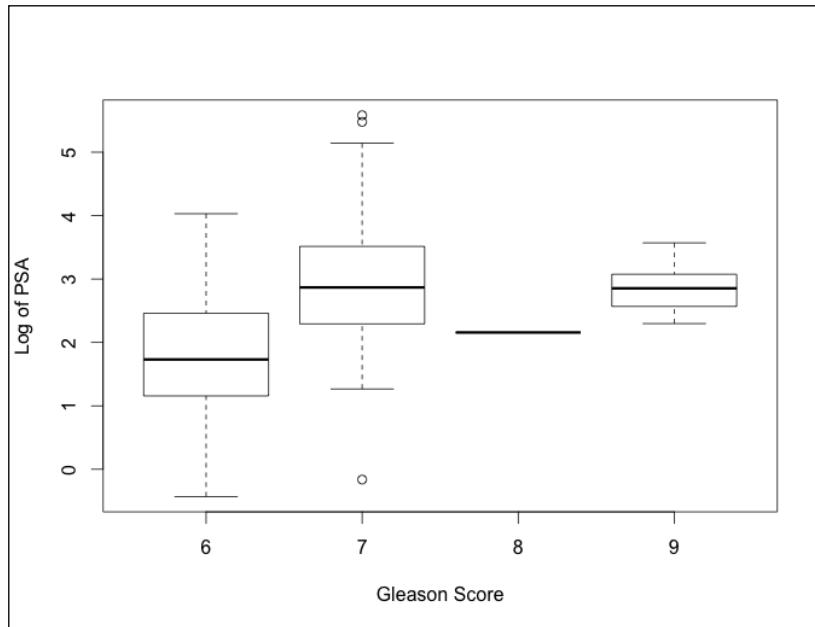
What are our options? We could do any of the following:

- 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 following 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:

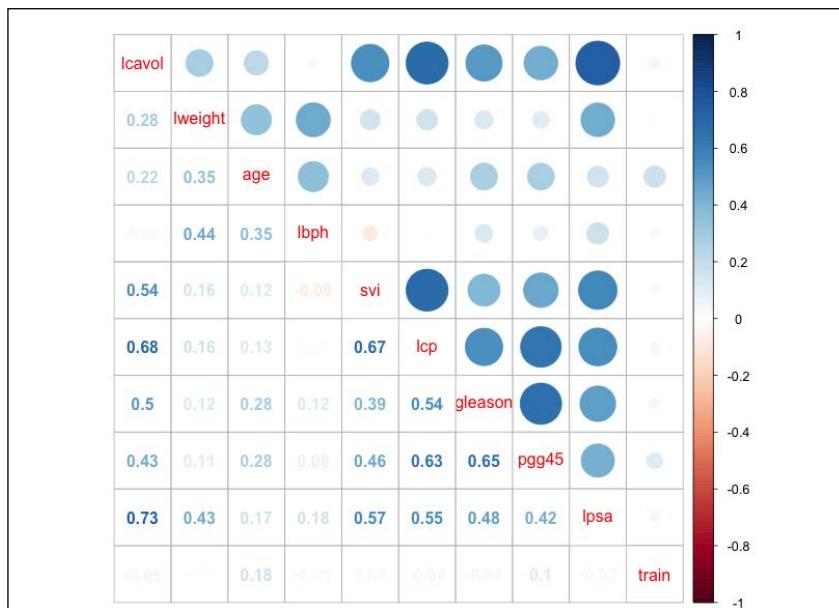
```
> table(prostate$gleason)
0  1
35 62
```

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 as follows:



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, as follows:

```
> 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 ...
 $ 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 ...
 $ 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 tilda and period stating that we want to use all the remaining variables in our data frame with the exception of the response, as follows:

```
> 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)
```

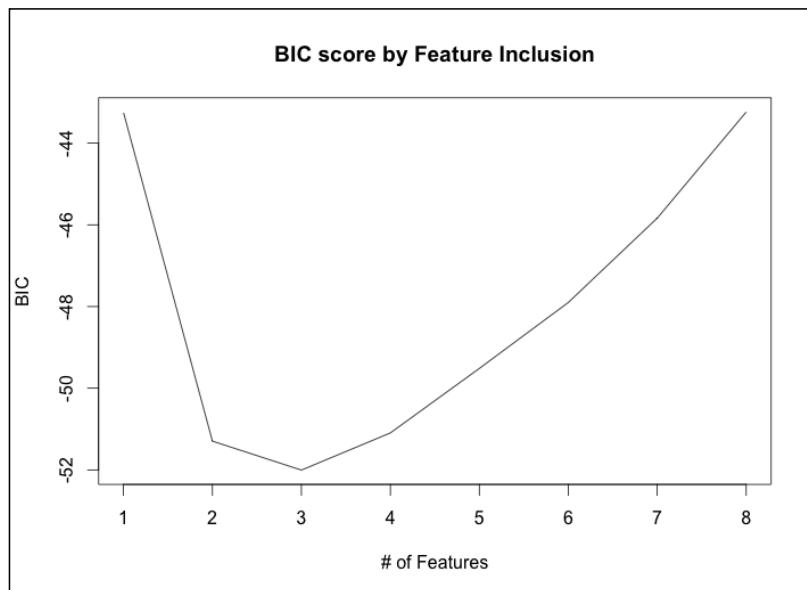
```
> which.min(b.sum$bic)
```

```
[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")
```

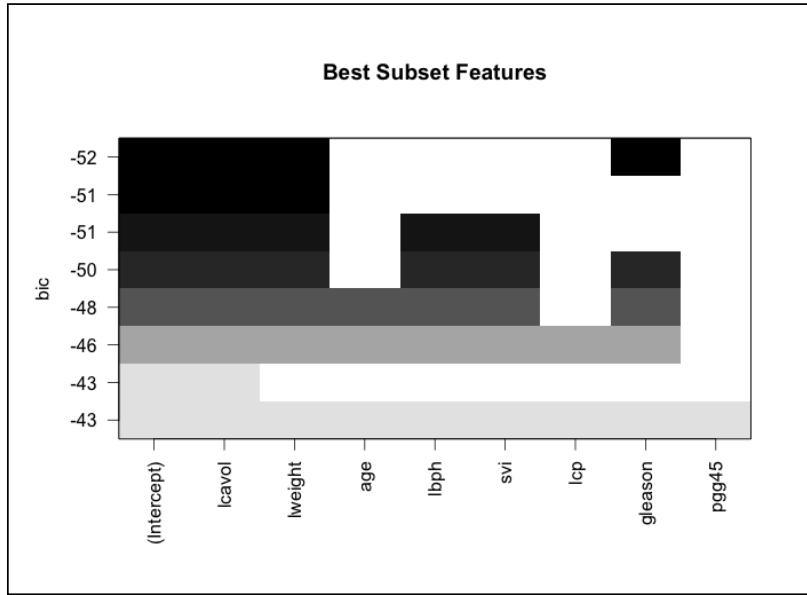
The following is the output of the preceding command:



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 as follows:

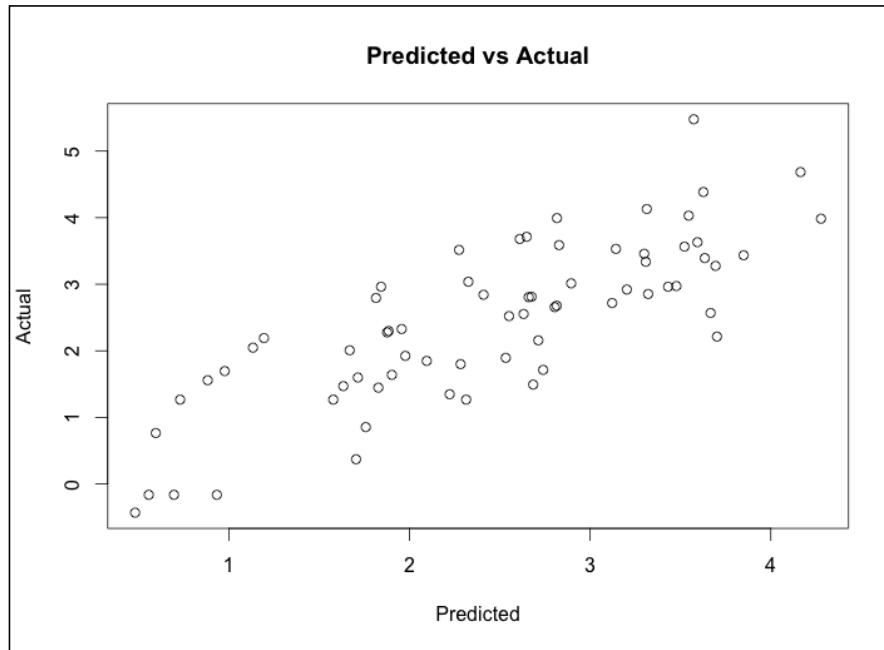


So, the previous plot shows us that the three features included in the lowest BIC are `lcavol`, `lweight`, and `gleason`. It is noteworthy that `lcavol` is included in every combination of the models. This is consistent with our earlier exploration of the data. We are now ready to try this model on the `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 actuals 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")
```

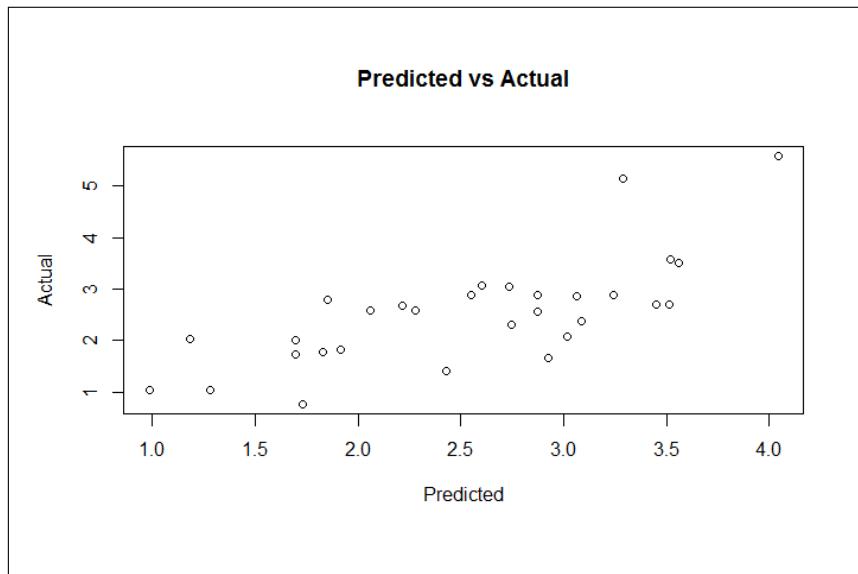
The following is the output of the preceding command:



An inspection of the plot shows that a linear fit should perform well on this data and that the nonconstant 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 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)
```

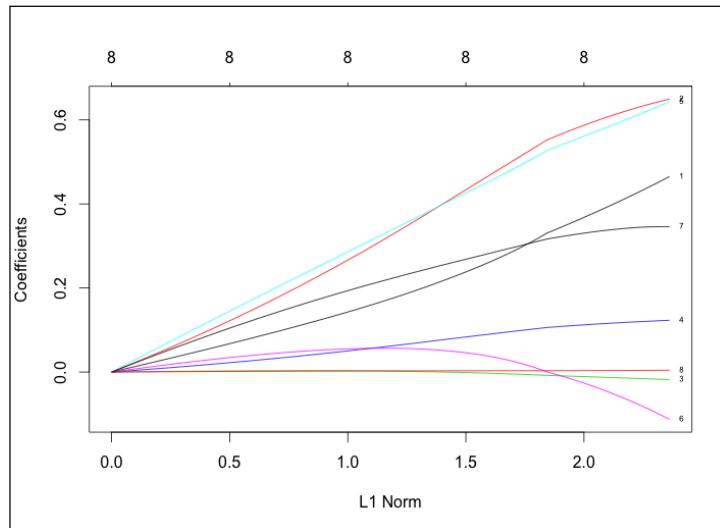
	Df	%Dev	Lambda
[1,]	8	3.801e-36	878.90000
[2,]	8	5.591e-03	800.80000
[3,]	8	6.132e-03	729.70000
[4,]	8	6.725e-03	664.80000

```
[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 an 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)
```

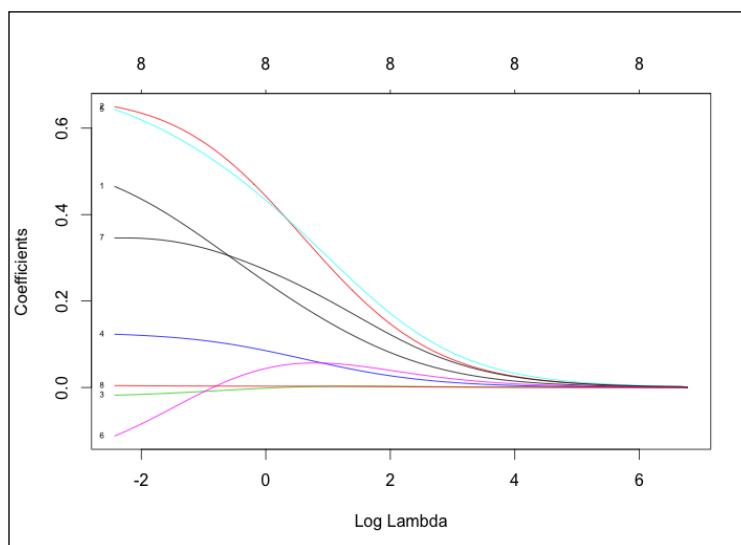
The following is the output of the preceding command:



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
```

```

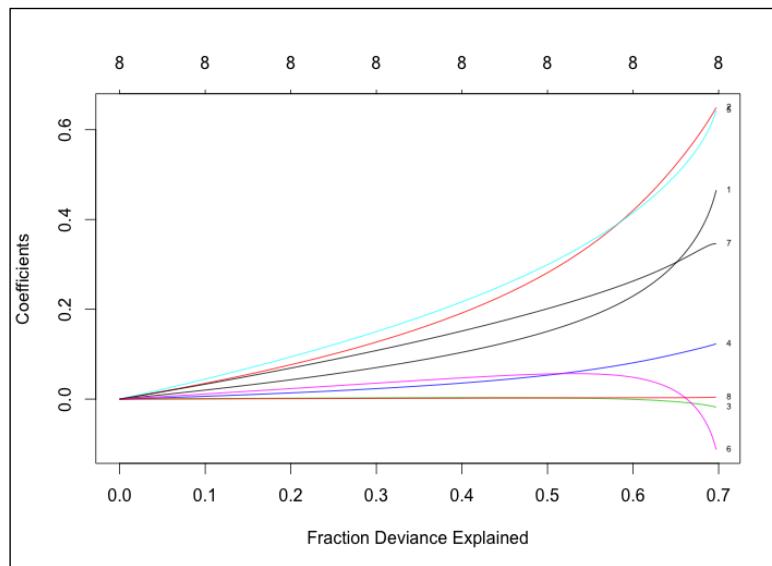
lcavol      0.45721270
lweight     0.64579061
age        -0.01735672
lbph       0.12249920
svi        0.63664815
lcp        -0.10463486
gleason    0.34612690
pgg45      0.00428580

```

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 would 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, you will have to transform the features as we did for the training data:

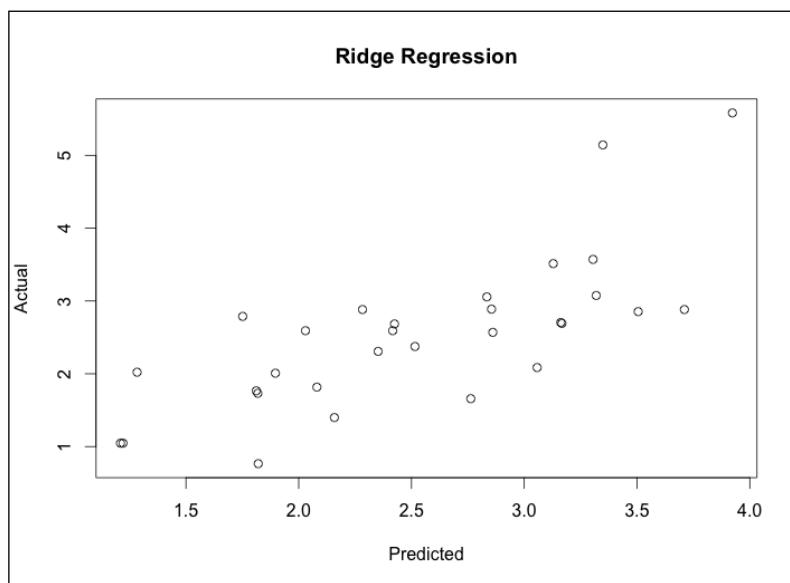
```
> newx = as.matrix(test[,1:8])
```

Then, 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 if they are truly unusual or if 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 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 ten 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
```

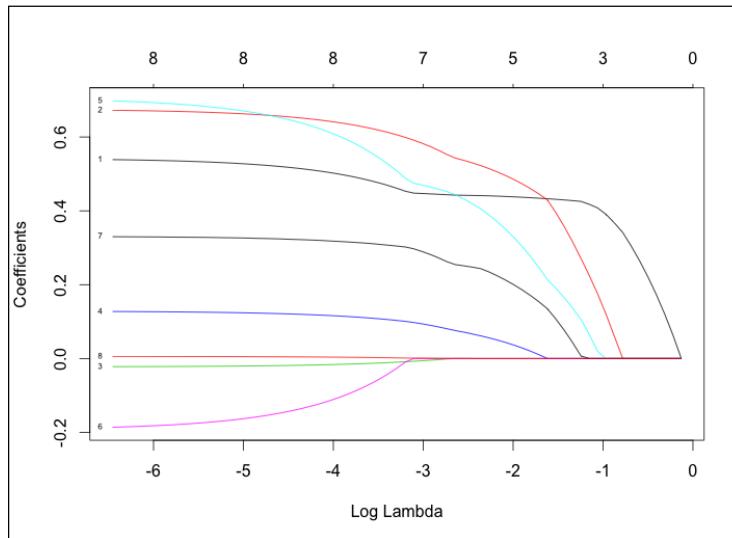
Note that the model building process stopped at step 69 as the deviance explained no longer improved as lambda decreased. Also, note that the `Df` column now changes along with lambda. At first glance, here it seems that all the eight features should be in the model with a lambda of 0.001572. However, 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 as follows:

```
> 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)

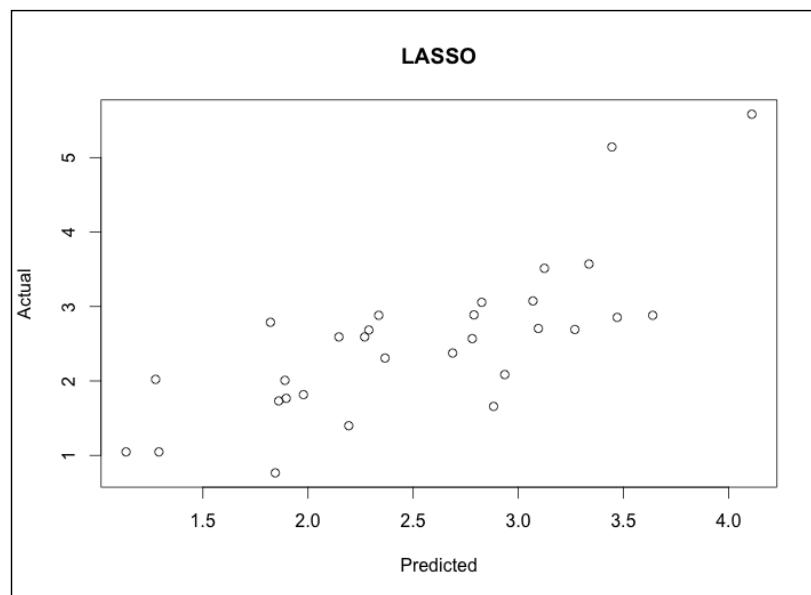
> lasso.coef
9 x 1 sparse Matrix of class "dgCMatrix"
   1
(Intercept) -0.1305852115
lcavol       0.4479676523
lweight      0.5910362316
age          -0.0073156274
lbph         0.0974129976
svi          0.4746795823
lcp          .
gleason     0.2968395802
pgg45        0.0009790322
```

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 $0 \leq \text{alpha} \leq 1$. 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. I think we can do the following:

- 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 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	1
0 . 8	1	1	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 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)

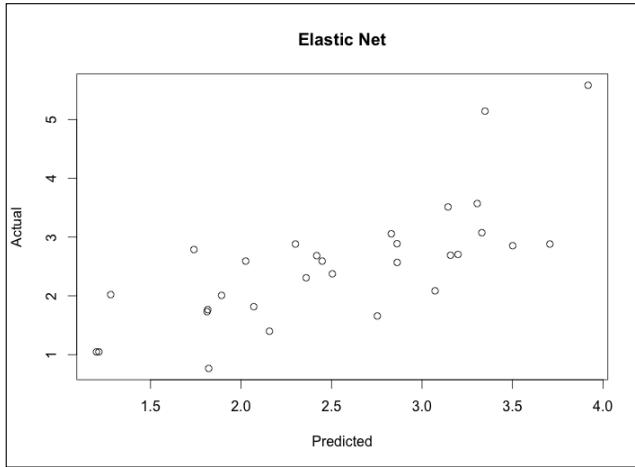
> enet.coef = coef(enet, s=.08, exact=TRUE)

> 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 a 10-fold cross-validation in the `glmnet` package to possibly identify a better solution.

Cross-validation with `glmnet`

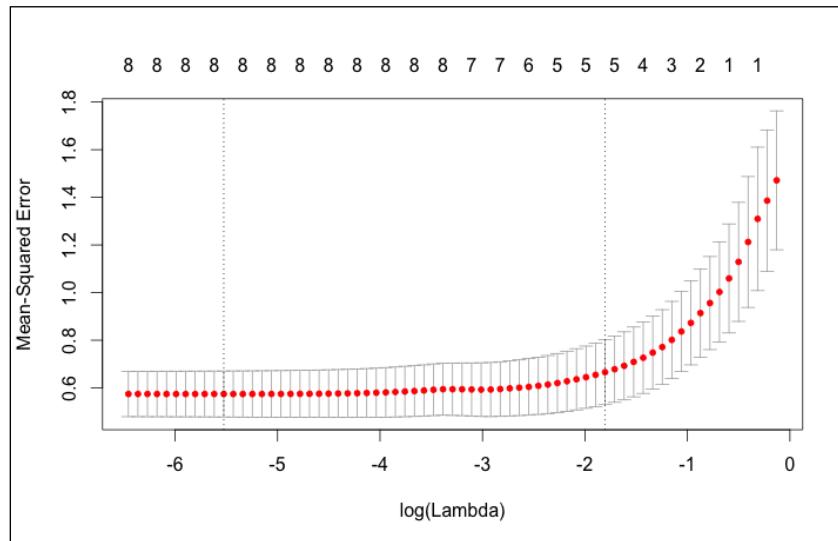
We have used LOOCV with the `caret` package; now we will try k-fold cross-validation. The `glmnet` package defaults to ten folds when estimating lambda in `cv.glmnet()`. In k-fold CV, the data is partitioned into an equal number of subsets (folds) and a separate model is built on each k-1 set and then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

```
> set.seed(317)
```

```
> lasso.cv = cv.glmnet(x, y)

> 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] 0.003985616

> lasso.cv$lambda.1se #one standard error away
[1] 0.1646861
```

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

```
> coef(lasso.cv, s = "lambda.1se")
9 x 1 sparse Matrix of class "dgCMatrix"
 1
```

```
(Intercept) 0.04370343
lcavol      0.43556907
lweight     0.45966476
age         .
lbph       0.01967627
svi        0.27563832
lcp         .
gleason    0.17007740
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] 0.4559446
```

This model achieves an error of 0.46 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 0.46

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 ~0.165 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. However, 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, Business Understanding – The Road to Actionable Insights*.

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 that we used will also support 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 versus 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.

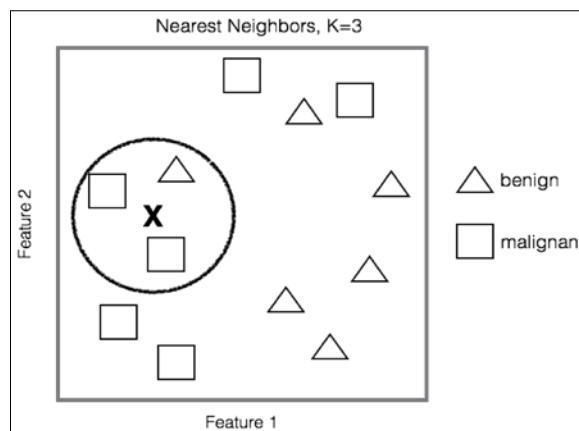
In this chapter, we will delve into two nonlinear techniques: **K-Nearest Neighbors (KNN)** and **Support Vector Machines (SVM)**. These techniques are more sophisticated than what we've discussed earlier because the assumptions on linearity can be relaxed, which means a linear combination of the features in order to define the decision boundary is not needed. Be forewarned though, 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 on 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, then 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 **Euclidian 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 Euclidian 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:

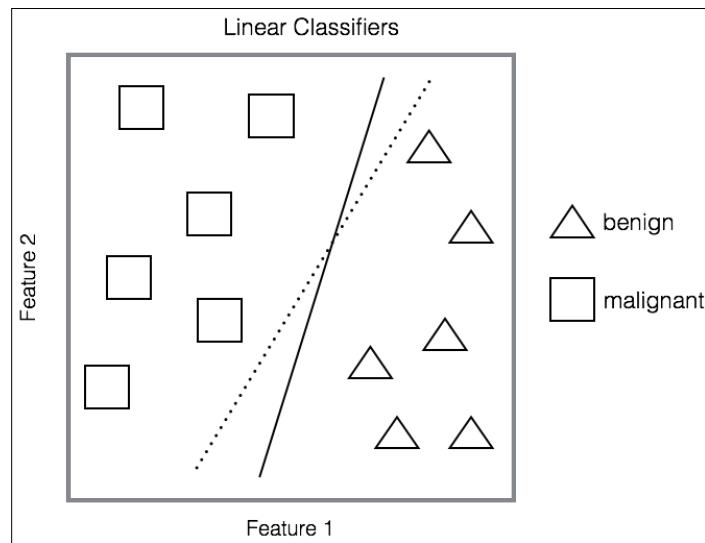
$$\text{Euclidian Distance}(A, B) = \sqrt{\sum_{i=1}^n (p_i - q_i)^2}$$

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

Support Vector Machines

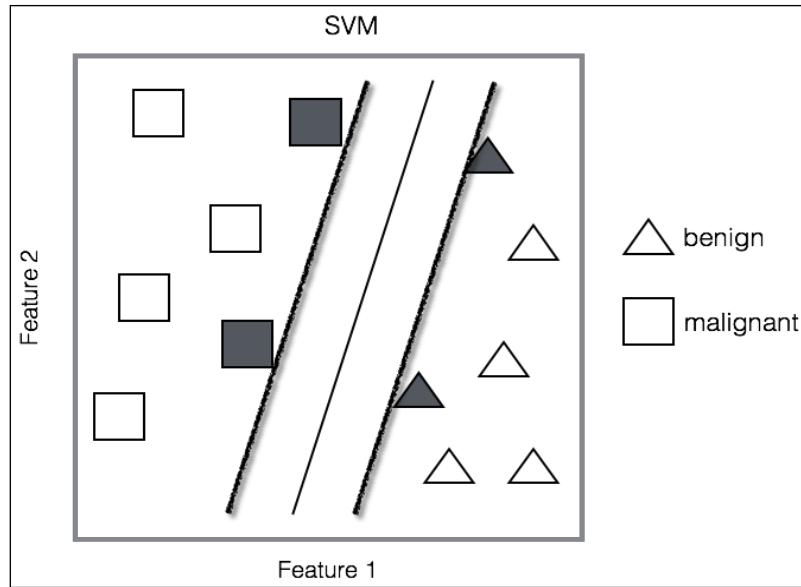
The first time I heard of support vector machines, I have to admit that I was scratching my head, thinking that this was some form of an academic obfuscation or inside joke. However, my open-minded review of SVM has replaced this natural skepticism with a healthy respect for the technique.

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 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, then the opposite might occur. According to James et al. who refers 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, 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 observations violate 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 $(3x1) + (4x2) + (2x3)$ 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:

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

Where (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. 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 with no transformation: $K(x_i, x_j) = x_i \cdot x_j$
- polynomial where d is equal to the degree of the 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 since the past 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 percent 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 disease, and kidney disease. 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  
> library(pROC)
```

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 to 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 as follows:

```
> pima = rbind(Pima.tr, Pima.te)
```

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

```
> 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 ...
```

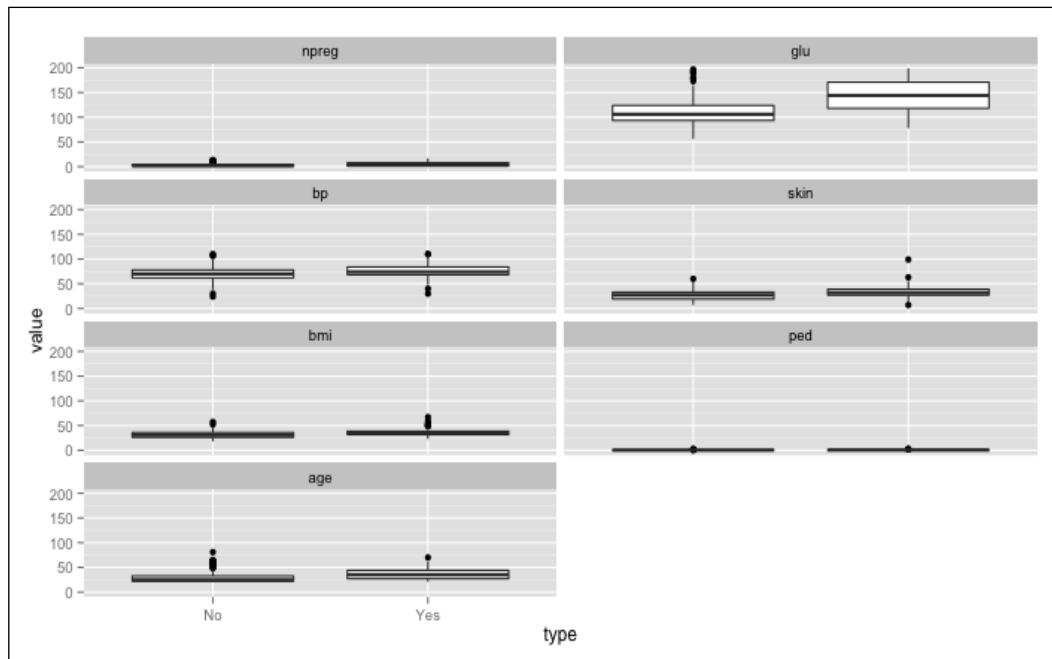
Let's do some exploratory analysis by putting this in boxplots. For this, we want to use the outcome variable, "type", as our ID variable. As we did with logistic regression, the `melt()` function will do this and prepare a data frame that we can use for the boxplots. We will call the new data frame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

```
> pima.scale = as.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:

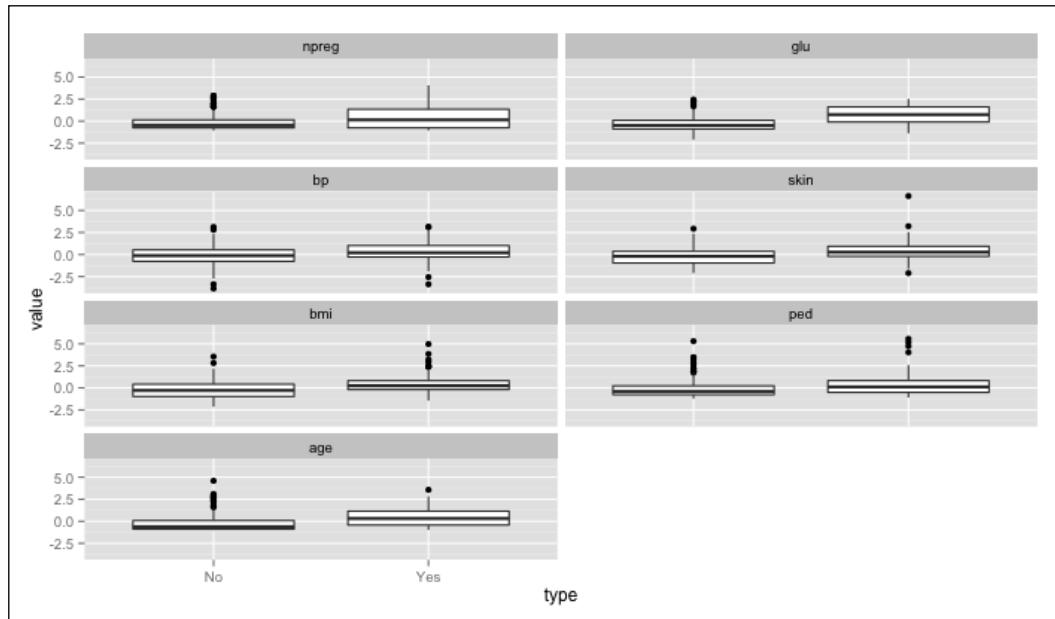
```
> pima.scale$type = pima$type
```

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 1.000000000 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 1.00000000
```

There are a couple of correlations to point out, npreg/age and skin/bmi. Multi-collinearity 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 are no hard and fast rules 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 the building of our predictive models and evaluate them, starting with KNN.

Modeling and evaluation

Now, we will see 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~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 can two evaluators 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 $Kappa = (Percent\ of\ agreement - Percent\ of\ chance\ agreement) / (1 - Percent\ of\ chance\ agreement)$.

The *Percent of agreement* is the rate that the evaluators agreed on 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
```

This is slightly less than our accuracy of 71 percent that we achieved on the `train` data alone of almost eight percent. We can now discuss the code of finding the kappa statistic. We have our accuracy and the chance calculation is simply the first row counts divided by the total rows multiplied by the first column counts divided by the total rows, 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 at 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 K	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 percent on the test set, we should see if 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 further away. The further away the observation is from a point in space, the more its influence is penalized. 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:

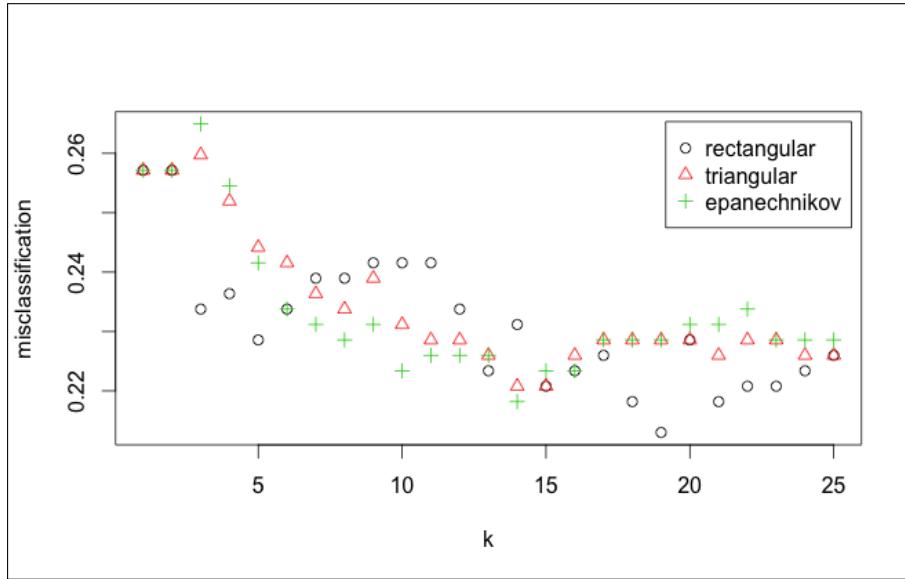
```
> set.seed(123)

> 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 surprise, the unweighted (**rectangular**) version at **k**: 19 performs the best. You can also call the object to see what is the classification error and the best parameter in the following way:

```
> kknn.train
```

Call:

```
train.kknn(formula = type ~ ., data = train, kmax = 25, distance = 2,
kernel = c("rectangular", "triangular", "epanechnikov", "gaussian"))
```

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. 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 percent. 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   80   22
  Yes  13   32

> (80+32)/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 non-linear 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 that are `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 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 percent. Conversely, with SVM, we could obtain an accuracy close to 80 percent. Before just simply selecting the most accurate model—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. This 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 stats that it produces:

- No Information Rate is the proportion of the largest class—63 percent 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 and 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 in 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)
> rfeCRTL = rfeControl(functions=lrFuncs, method="cv", number=10)

> svm.features = rfe(train[,1:7], train[,8], sizes = c(7, 6, 5, 4),
rfeControl = rfeCRTL, 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 in the full model was 76.2 percent:

```
> 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 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 is 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)

Introduction

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

In the previous chapters, we examined the techniques to predict either a quantity or a label classification. Here we will apply them on 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 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 or better than any technique in this book. These methods are known as **random forests** and **gradient boosted trees**.

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.

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 a regard as to how well it will perform on the later 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, high variance can become an issue. As with the other methods, you can use cross-validation to select the appropriate tree depth size.

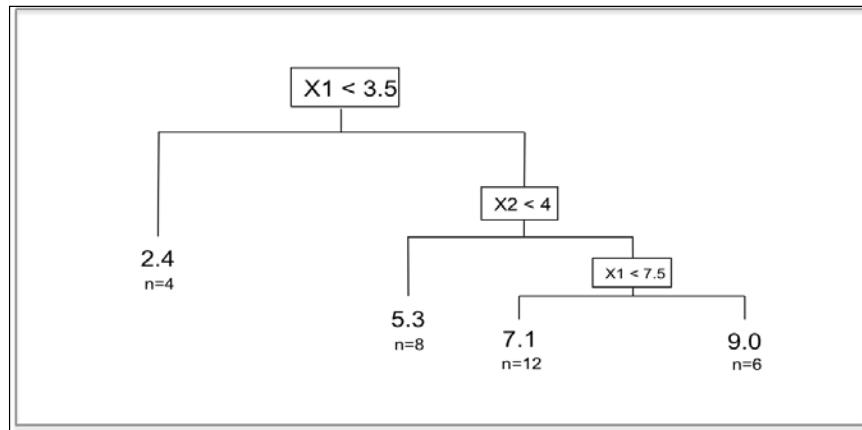


Figure 6.1: Regression Tree with 3 splits and 4 terminal nodes and the corresponding node average and number of observations

Classification trees

Classification trees operate under the same principal 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 N_0 where you have seven observations labeled No 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:

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

Then, for N_0 , the Gini is $1 - (.7)^2 - (.3)^2$, which is equal to 0.42, versus the misclassification rate of 30 percent.

Taking this example further, we will now create node N_1 with 3 observations from Class 1 and none from Class 2, along with N_2 , which has 4 observations from Class 1 and three from Class 2. Now, the overall misclassification rate for this branch of the tree is still 30 percent, but look at how the overall *Gini index* has improved:

- $Gini(N_1) = 1 - (3/3)^2 - (0/3)^2 = 0$
- $Gini(N_2) = 1 - (4/7)^2 - (3/7)^2 = 0.49$
- *New Gini index* = (*proportion of N_1 x $Gini(N_1)$*) + (*proportion of N_2 x $Gini(N_2)$*),
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 are 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 is another tuning parameter that we will need to identify and this is shrinkage. You can think of shrinkage as the rate at which your model is learning generally and as the contribution of each tree or stump to the model specifically. This learning rate acts as a regularization parameter, similar to what we discussed in *Chapter 4, Advanced Feature Selection in Linear Models*.

The other thing about our boosting algorithm is that it is stochastic, meaning that it adds randomness by taking a random sample of data at each iteration of the algorithm used in each iteration of the tree. Introducing some randomness to a boosted model usually improves the accuracy and speed and reduces the overfitting (Friedman, 2002).

As you may have guessed, tuning these parameters can be quite a challenge. These parameters can interact with each other, and if you just tinker with one without considering the other, your model may actually perform worse. The `caret` package will help us in this endeavor.

Business case

The overall business objective in this situation is to see if 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 percent predictive accuracy. For the diabetes data, we are seeking to improve on the 79.6 percent 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 in to 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(gbm) #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)
```

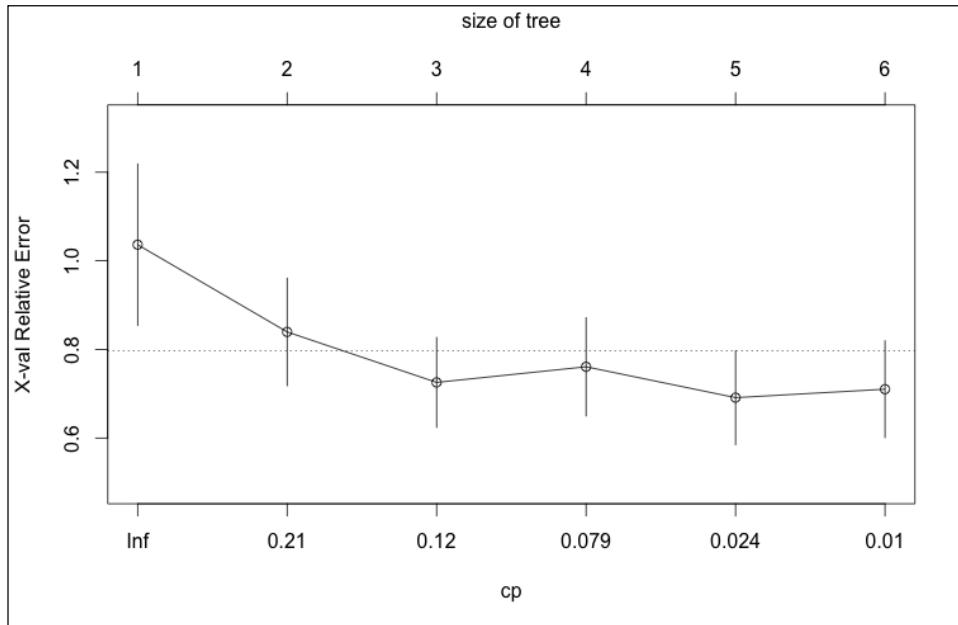
We can call this object using the `print()` function and `cptable` and then examine the error per split in order to determine the optimal number of splits in the tree:

```
> 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:



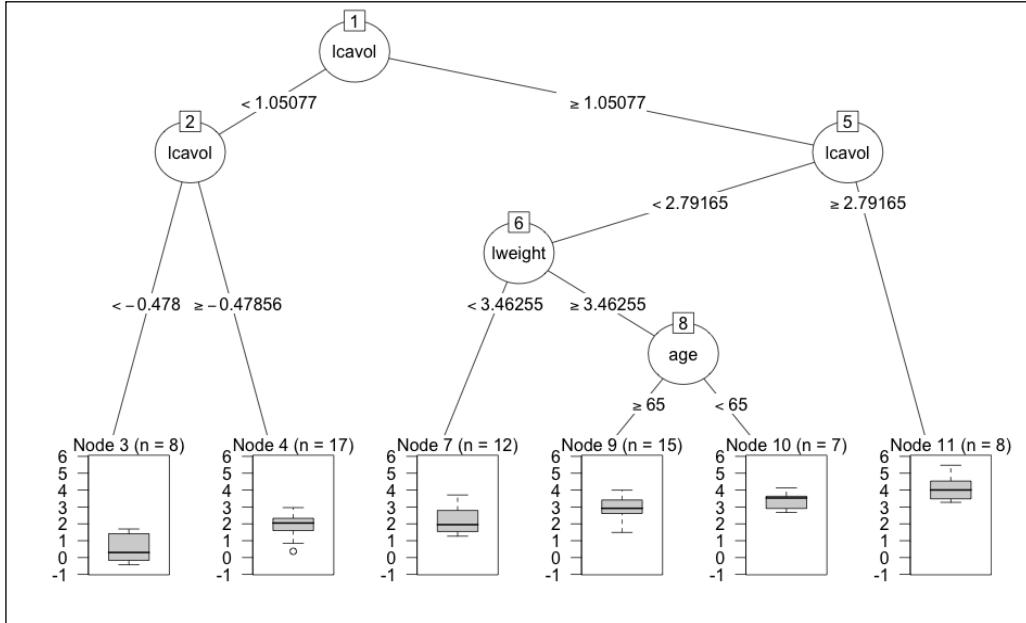
The plot shows us the relative error by the tree size with the corresponding error bars. The horizontal line on the plot is the upper limit of the lowest standard error. Selecting a tree size, 5, which is four splits, we can build a new tree object where `xerror` is minimized by pruning our tree accordingly 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$cptable[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))
```

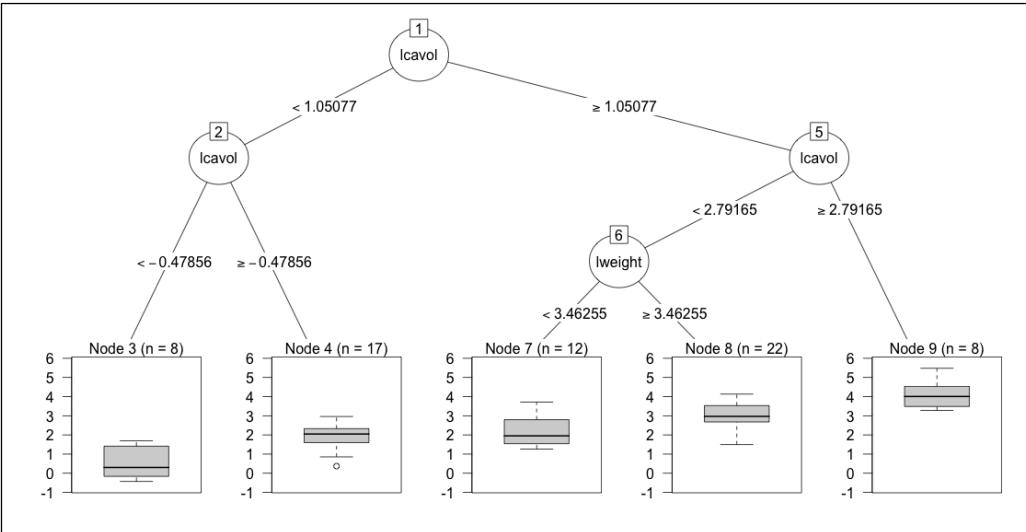
The output of the preceding command is as follows:



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 #calculate residuals

> mean(rpart.resid^2) #caluclate 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 dataset 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)

> print(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
```

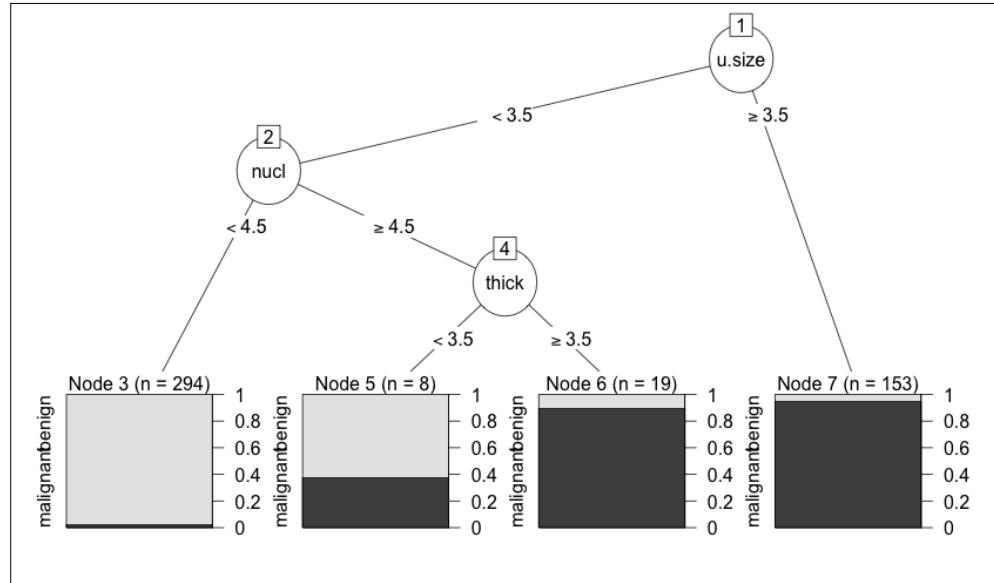
The cross-validation error is at a minimum with only two splits (row 3). We can now prune the tree, plot the full and pruned trees, and see how it performs on the test set:

```
> cp = min(tree.biop$cptable[3,])

> prune.tree.biop = prune(tree.biop, cp = cp)

> plot(as.party(tree.biop))
> plot(as.party(prune.tree.biop))
```

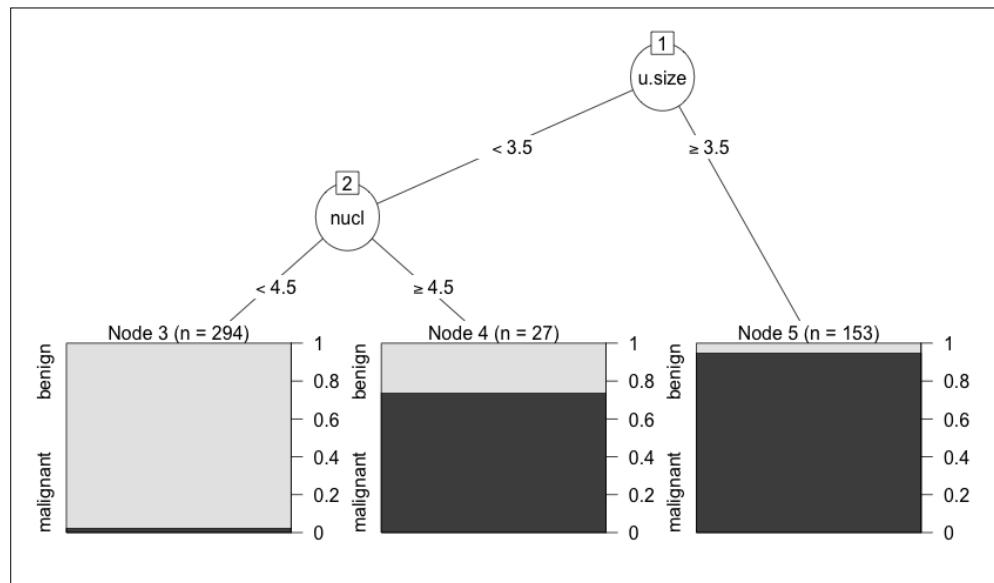
The output of the preceding command is as follows:



Now we will plot the pruned trees using the following command:

```
> plot(as.party(prune.tree.biop))
```

The output of the preceding command is as follows:



An examination of the tree plots 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 again on the prostate data 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)

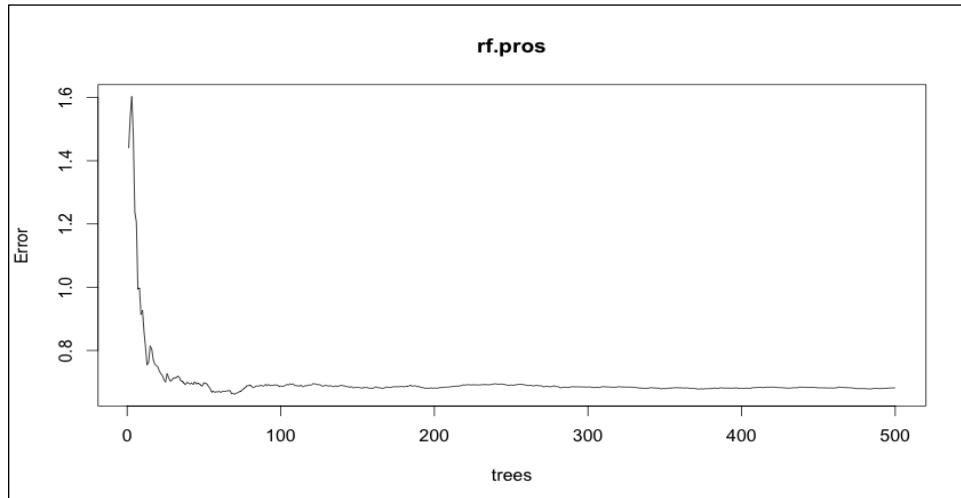
> print(rf.pros)
```

```
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: 0.6813944  
% Var explained: 52.58
```

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 much 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)
```

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] 70
```

We can try 70 trees in the random forest by just specifying `ntree=70` in the model syntax:

```
> set.seed(123)

> rf.pros.2 = randomForest(lpsa~., data=pros.train, ntree=70)

> print(rf.pros.2)

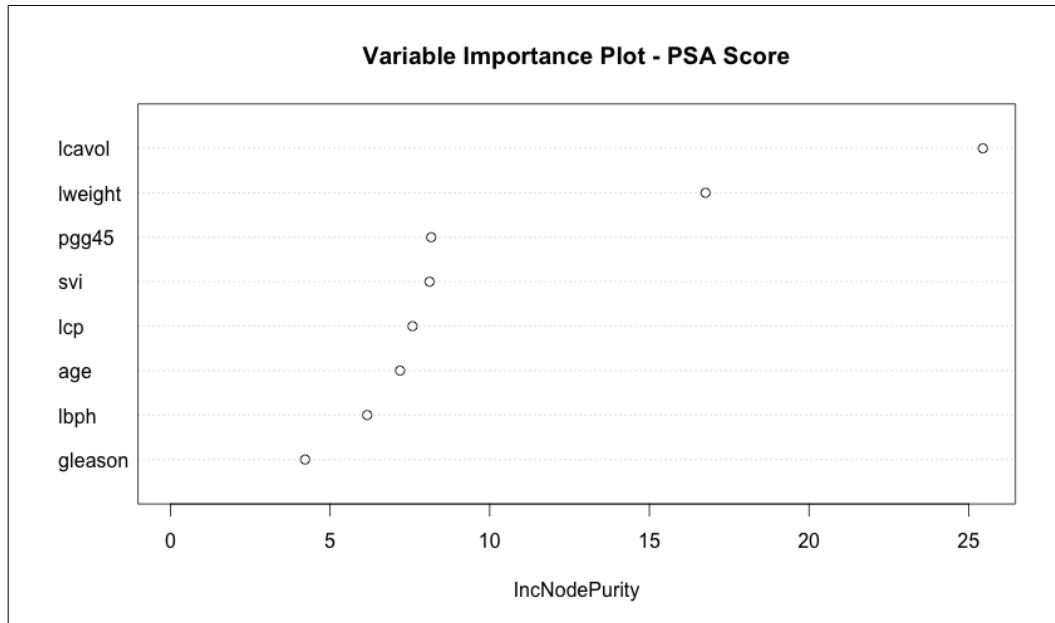
Call:
randomForest(formula = lpsa ~ ., data = pros.train, ntree = 70)
                 Type of random forest: regression
                         Number of trees: 70
No. of variables tried at each split: 2

               Mean of squared residuals: 0.6617529
                           % Var explained: 53.95
```

You can see that the MSE and variance explained have both improved slightly. Let's see one other plot before testing the model. If we are combining the results of 70 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. Only 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, 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     25.446395
lweight    16.758646
age        7.191313
lbph       6.161000
svi        8.114879
lcp        7.580892
gleason    4.218471
pgg45      8.166068
```

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] 0.5420387
```

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)

> print(rf.biop)

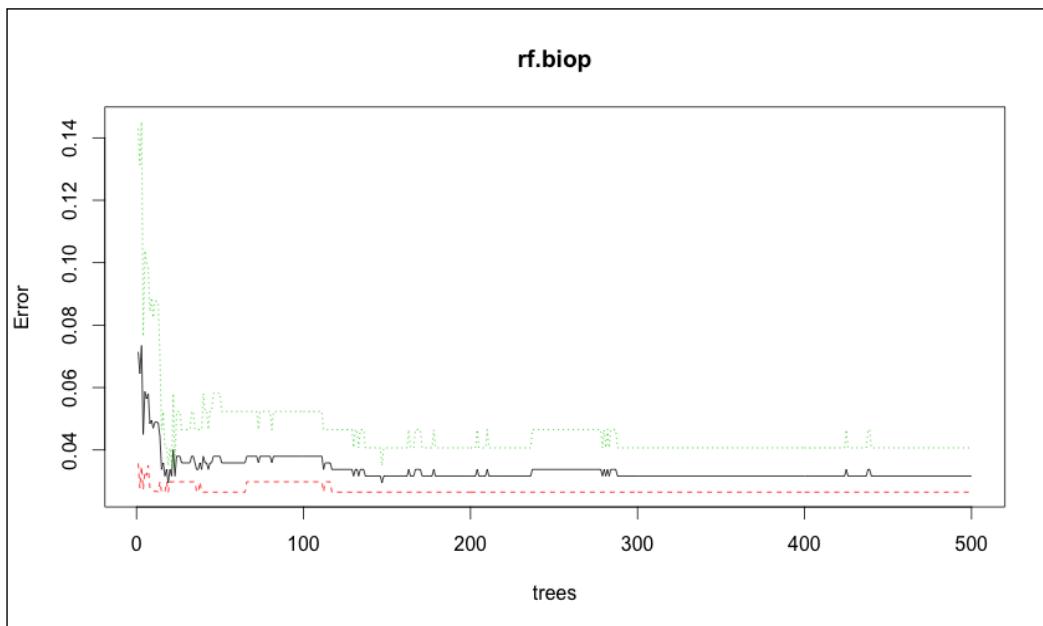
Call:
randomForest(formula = class ~ ., data = biop.train)
Type of random forest: classification
Number of trees: 500
No. of variables tried at each split: 3

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:

```
> 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)

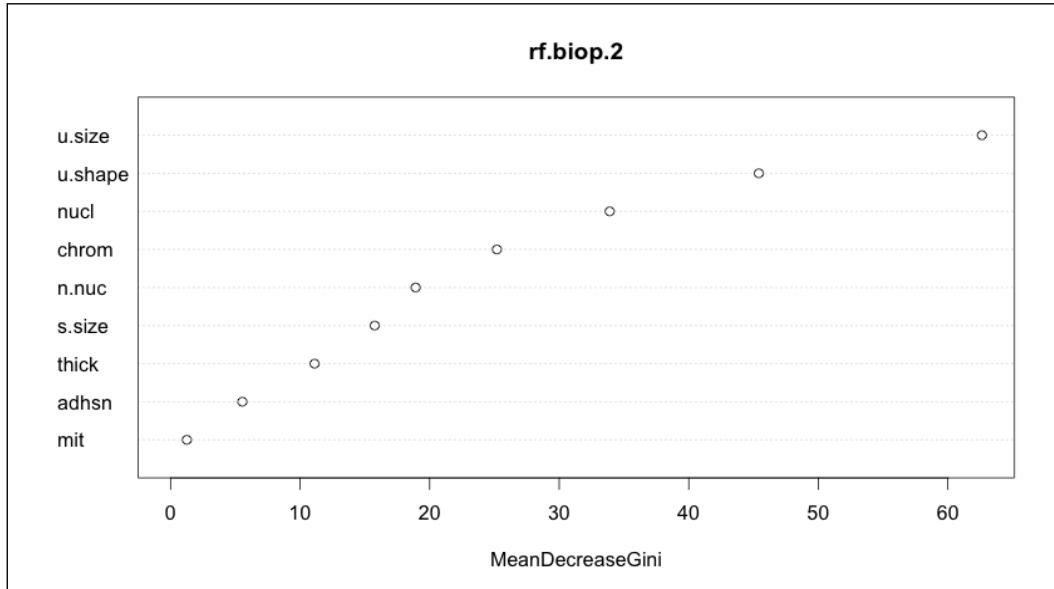
rf.biop.test benign malignant
benign        139       0
malignant       3      67

> (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 each variable's contribution to the mean decrease in the Gini index. This is rather different from the splits of the single tree. Recall 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)

> print(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 percent 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$err.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.

Gradient boosting regression

The R package that we will use in this section is called `gbm`, which we have already loaded. As stated in the boosting overview, we will be tuning three boosting parameters: the number of trees, interaction depth, and shrinkage. As we did in the prior chapters, it will be helpful to call on the `caret` package to help in the tuning process. The default parameter in the package is 100 trees, interaction depth is 1, and shrinkage is 0.001.

Using the `expand.grid()` function, we will build our experimental grid to run through the training process of the `caret` package. Let's do the trees from 100 to 500 `by = 200`, interaction depth 1 to 4 `by = 1`, and shrinkage as 0.001, 0.01, and 0.1:

```
> grid = expand.grid(.n.trees=seq(100,500, by=200), .interaction.
  depth=seq(1,4, by=1), .shrinkage=c(.001,.01,.1), .n.minobsinnode=10)
```

This creates a grid of 36 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 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. For the `prostate` data, we will use `LOOCV`, which is `LOOCV`, as follows:

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

To utilize the `train()` function, just specify the formula as we did with the other models: the `train` dataset, method, train control, and experimental grid. Remember to set the random seed!

```
> gbm.pros.train = train(lpsa~, data=pros.train, method="gbm",
  trControl=control, tuneGrid=grid)
```

Calling the object gives us the optimal parameters and the results of each of the parameter settings, as follows;

```
> gbm.pros.train
Stochastic Gradient Boosting

67 samples
 8 predictor

No pre-processing
Resampling:

Summary of sample sizes: 66, 66, 66, 66, 66, 66, ..

Resampling results across tuning parameters:


```

n.trees	interaction.depth	shrinkage	RMSE	Rsquared
100	1	0.001	1.184	0.132
100	1	0.010	0.989	0.441
100	1	0.100	0.914	0.431
.....				
500	4	0.010	0.857	0.490
500	4	0.100	1.022	0.357

```
RMSE was used to select the optimal model using the smallest value.  
The final values used for the model were n.trees = 300, interaction.depth  
= 3 and shrinkage = 0.01.
```

Having identified the tuning parameters, just place this in the `gbm()` function to build the model in the `train` set. Additionally, we will specify the `distribution = "gaussian"` for a squared error as this is a continuous outcome:

```
> gbm.pros = gbm(lpsa~, data=pros.train, n.trees=300, interaction.  
depth=3, shrinkage=0.01, distribution="gaussian")
```

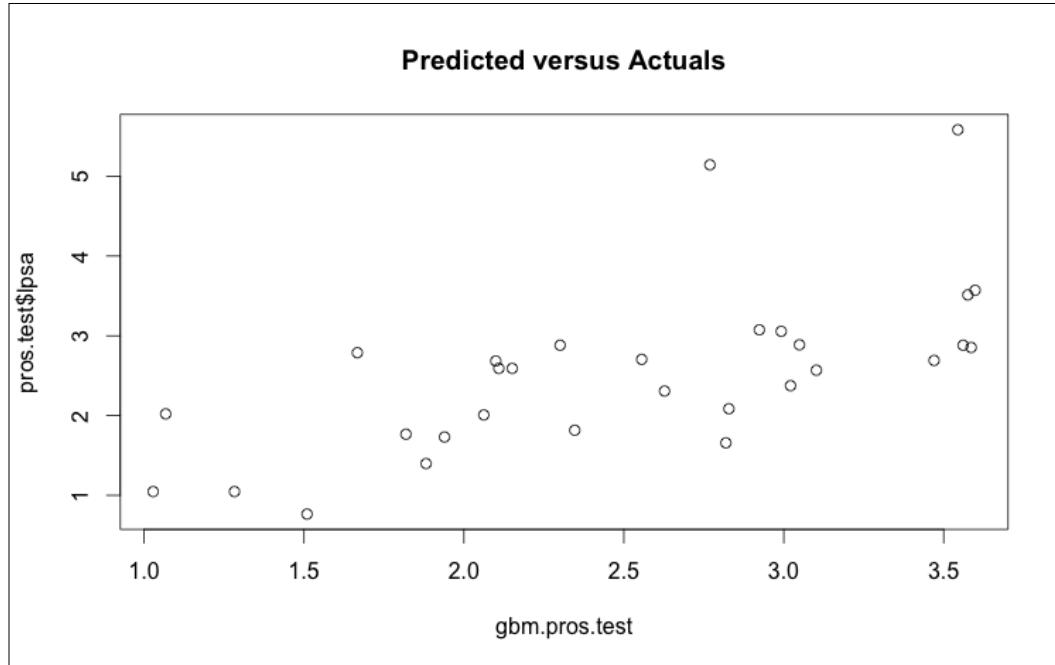
With an object created from the `train` data, we can see how it performs on the `test` data using the `predict()` function as we did with the other methods. Then calculate the residuals and MSE:

```
> gbm.pros.test = predict(gbm.pros, newdata=pros.test, n.trees=300)  
  
> gbm.resid = gbm.pros.test - pros.test$lpsa  
  
> mean(gbm.resid^2)  
[1] 0.6208321
```

An MSE of 0.62 on the `test` set does not improve our predictive ability at all. Let's take a look at the plot of the **Predicted versus Actuals** values. As it turns out, gradient boosting can be very susceptible to a high variance in the presence of outliers. The plot shows that the model performed very well except for two observations. The key takeaway for this dataset is that it is probably too small to provide any benefit of using the boosting methods in terms of the number of observations. Boosting may become beneficial in a dataset of this n-size in a situation of high dimensionality:

```
> plot(gbm.pros.test, pros.test$lpsa, main="Predicted versus Actuals")
```

The output of the preceding command is as follows:



Gradient boosting classification

For the breast cancer problem, we will again use the `train()` function from the `caret` package. The only change to the syntax is that we will use 10-fold cross-validation as `trainControl`:

```
> control = trainControl(method="CV", number=10)

> set.seed(123)

> gbm.biop.train = train(class~, data=biop.train, method="gbm",
+ trControl=control, tuneGrid=grid)

> gbm.biop.train
Stochastic Gradient Boosting

474 samples
  9 predictor
```

```
2 classes: 'benign', 'malignant'

No pre-processing
Resampling: Cross-Validated (10 fold)
..... Accuracy was used to select the optimal model using the largest value.
The final values used for the model were n.trees = 100, interaction.depth
= 1 and shrinkage = 0.1.
```

We will again put these tuning parameters into the `gbm()` function. However, we will have to modify our dataset slightly. The function will not accept a factor for a 0/1 classification problem. This is a quick fix as we will use the `ifelse()` function to code benign as 0 and malignant as 1:

```
> biop.train$class = ifelse(biop.train$class=="benign",0,1)
```

This gives us the 0/1 class labels that we will need in our response variable for `gbm()`. Remember that for regression, we used the gaussian distribution. For a 0/1 label problem that we have here, let's use `bernoulli`, which is a logistic loss function. The Bernoulli distribution where the random variable takes a value of one for success with a probability p and a failure value of zero with a probability $q = 1 - p$. Other distributions are available, but I'll refer you to the package's documentation for other alternatives.

```
> gbm.biop = gbm(class~, distribution="bernoulli", data=biop.train,
n.trees=100, interaction.depth=1, shrinkage=0.1)
```

For the prediction of the test set values, you will again need to change how you analyze the results versus regression. The `predict()` function in the package will provide the probabilities of the class membership, just as logistic regression did in *Chapter 3, Logistic Regression and Discriminant Analysis*. In other words, anything less than 50 percent predicted probability is benign, otherwise it is malignant. We will again use the `ifelse()` function to handle this properly:

```
> gbm.biop.test = predict(gbm.biop, newdata=biop.test, type="response",
n.trees=100)
```

```
> gbm.class = ifelse(gbm.biop.test <0.5,"benign", "malignant")
```

```
> table(gbm.class, biop.test$class)
```

gbm.class	benign	malignant
benign	140	2

```
malignant      2       65
```

```
> (140+65)/209  
[1] 0.9808612
```

This model performed quite well, only misclassifying four of the 209 observations, only one more than the random forest model.

We will now move on to the the final challenge of the chapter, the diabetes data, which has proved such a challenge to the improvement of predictive power. The process for this problem is no different than what we just produced with the breast cancer diagnosis data, as follows:

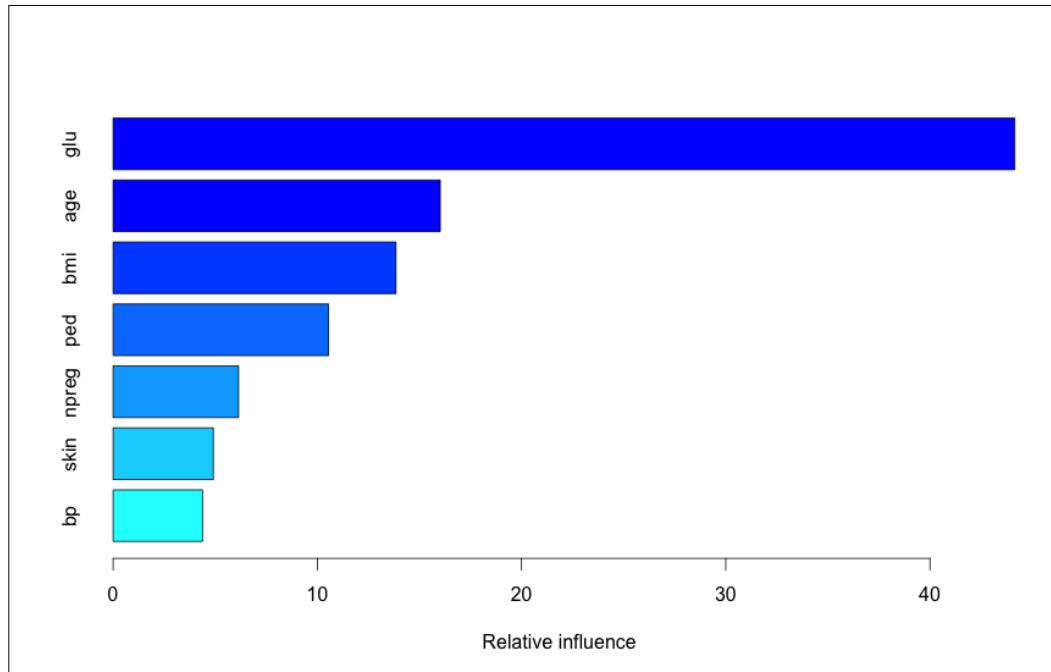
```
> set.seed(123)  
  
> gbm.pima.train = train(type~., data=pima.train, method="gbm",  
trControl=control, tuneGrid=grid)  
  
> gbm.pima.train  
Stochastic Gradient Boosting  
  
385 samples  
7 predictor  
2 classes: 'No', 'Yes'  
  
No pre-processing  
Resampling: Cross-Validated (10 fold)  
.....  
Accuracy was used to select the optimal model using the largest value.  
The final values used for the model were n.trees = 500, interaction.depth  
= 3 and shrinkage = 0.01.  
  
> pima.train$type = ifelse(pima.train$type=="No", 0, 1)  
  
> gbm.pima = gbm(type~., distribution="bernoulli", data=pima.train,  
n.trees=500,interaction.depth=3,shrinkage=0.01)  
  
> gbm.pima.test = predict(gbm.pima, newdata=pima.test, type="response",  
n.trees=500)
```

```
> gbm.type = ifelse(gbm.pima.test <0.5, "No", "Yes")  
  
> table(gbm.type, pima.test$type)  
  
gbm.type  No  Yes  
  No    77   22  
  Yes   16   32  
  
> (77+32)/147  
[1] 0.7414966
```

The accuracy at 74 percent does not beat the benchmark that we attained with the SVM, meaning that none of the tree-based methods improved the predictive ability on the Pima Indian data. Before closing this section, I would like to point out that you can also produce the variable importance with boosted trees, similar to what we did with random forests. The `summary()` function in the `gbm` package produces a table of the relative influence values and a barplot as well:

```
> summary(gbm.pima)  
      var  rel.inf  
glu      glu 44.152295  
age      age 16.019096  
bmi      bmi 13.849614  
ped      ped 10.545554  
npreg  npreg  6.144122  
skin     skin  4.908916  
bp       bp   4.380403
```

The output of the preceding command is as follows:



As the influence is relative, we can conclude that the `glu` variable (glucose levels) account for 44 percent of the variance in the predicted outcome. These influence/importance plots provide an effective tool in feature selection and in presenting the results to the business partners.

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 and boosted trees both 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.

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* by introducing the *Deep learning* section.

Summary

In this chapter, you learned both the power and limitations of tree-based learning methods for both the 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, dozens or hundreds 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. 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.

While these methods are indeed 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.

7

Neural Networks

"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 and deep-seated anxiety for 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, machine, and automation. If self-driving cars ever stop running off the road and into each other, it will likely 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.

Neural network

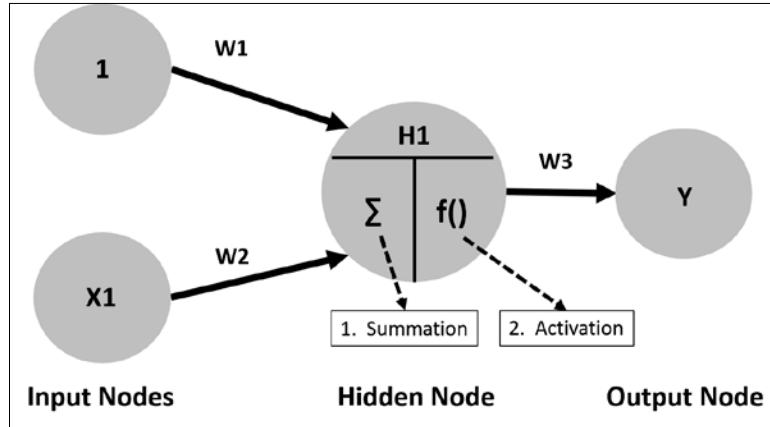
Neural network is a fairly broad term that covers a number of related methods, but in our case, we will focus on a **Feed Forward** network that trains with **Back Propagation**. I'm not going to waste our time discussing how the machine learning methodology is similar or dissimilar to how a biological 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.

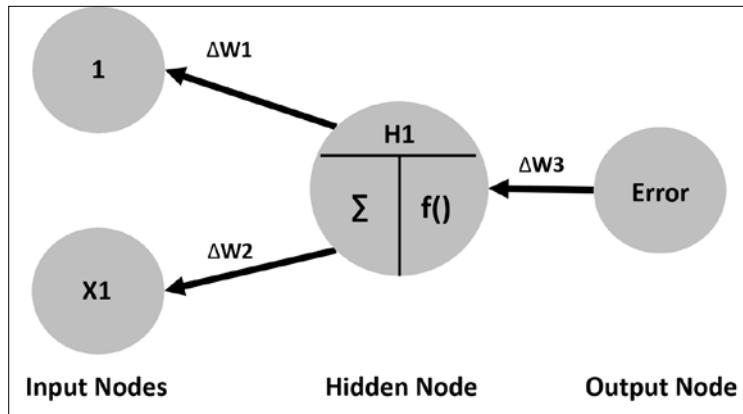
The motivation or benefit of ANNs is that they allow the modeling of highly complex relationships between inputs/features and response variable(s), especially if the relationships are highly nonlinear. No underlying assumptions are required to create and evaluate the model and it can be used with qualitative 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 **Ws** 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, **W_1** and **W_2** , 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 $Y = W_3(W_1(1) + W_2(X_1))$.

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 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 logistic function. (*Chapter 3, Logistic Regression and Discriminant Analysis*) A threshold function can be used where the output is binary (0 or 1) based on some threshold value. Other common activation functions are sigmoid and hyperbolic tangent (\tanh).

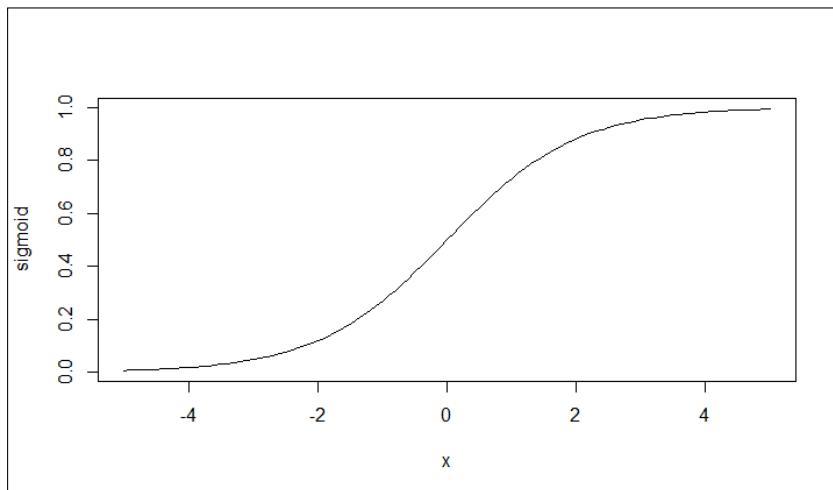
The sigmoid function is similar to the logistic function but is not bound between zero and one. (Note that the logistic function is the inverse of the sigmoid.) We can plot a sigmoid function in R. We will first create an R function in order to calculate the sigmoid function values:

```
> 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:

```
> plot(sigmoid, -5, 5)
```

The output of the preceding command is as follows:



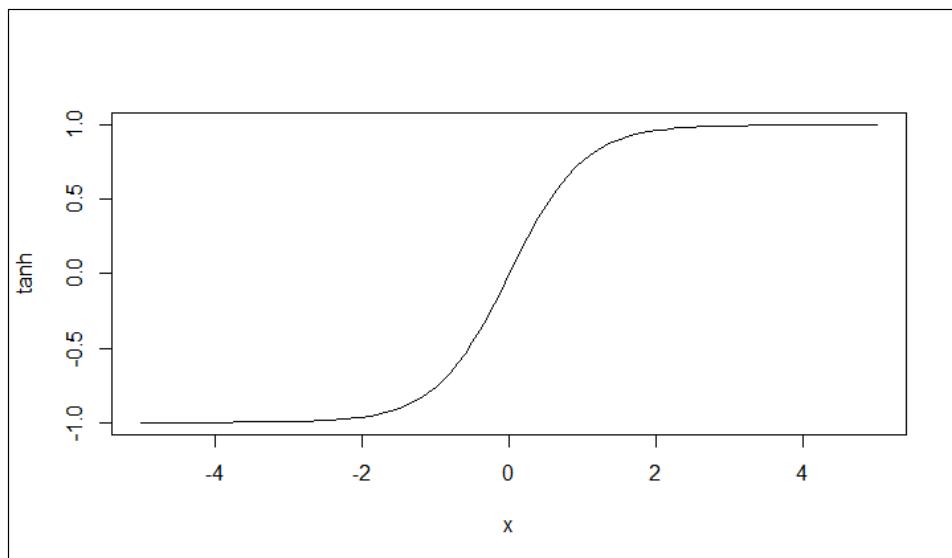
You can also plot code font using base R. Again, let's examine it between -5 and 5:

```
> x = seq(-5, 5, by=0.1)

> t = tanh(x)

> plot(x,t, type="l", ylab="tanh")
```

The output of the preceding command is as follows:



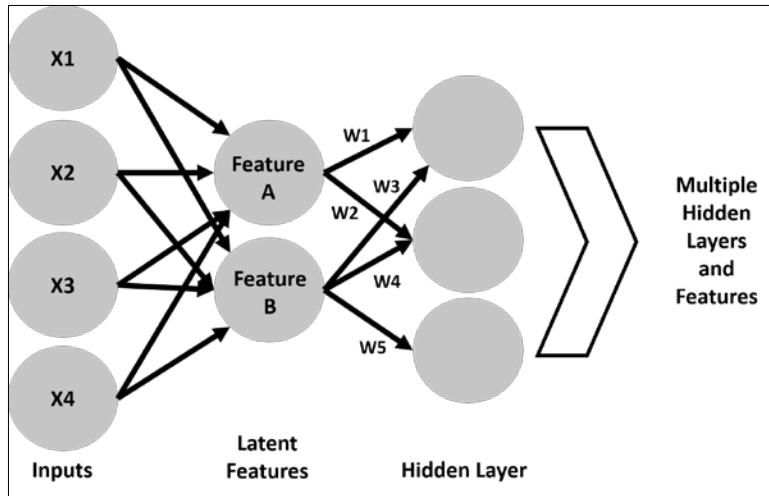
This all sounds fascinating, but the ANN almost went the way of disco as it just did not perform well, especially when trying to use deep networks with many hidden layers and neurons. It seems that a slow, yet gradual revival came about with the seminal paper by Hinton and Salakhutdinov (2006) in the 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 non-linear 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 the 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 if 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 vast amounts 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 take 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 happening in this case is 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. However, it is appropriate for problems where an explanation of How is not a problem and the important question is What. 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, this gap has closed, if not been eliminated altogether.

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 of the hidden layers

- Dimensionality reduction, that is, Restricted Boltzmann versus Autoencoder versus ...
- The number of epochs
- The gradient descent learning rate
- The loss function
- Regularization

You can imagine that this can be no small feat, but enough of the overview; let's move on to some practical applications.

Business understanding

It was a calm, clear night of 20th 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 DME (Distance Measuring Equipment) 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 windscreens 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, destroying the windscreens. 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 autolanding 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 three packages. The data is in the `MASS` package, but this is a required dependent package loaded with `neuralnet`:

```
> library(caret)

> library(neuralnet)
Loading required package: grid
Loading required package: MASS

> 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 2 2  
...  
$ error     : Factor w/ 4 levels "LX","MM","SS",...: 1 1 1 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 1 ...  
$ use       : Factor w/ 2 levels "auto","noauto": 1 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 percent 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 versus 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:

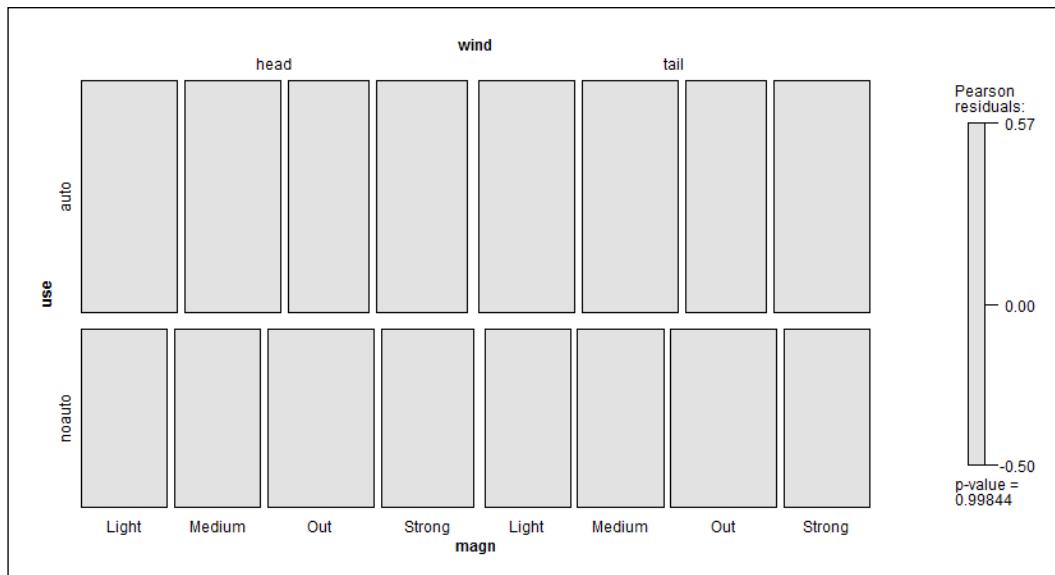
```
> table1=structable(wind+magn~use, shuttle)  
  
> table1
```

	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, gp=shading_hcl)
```

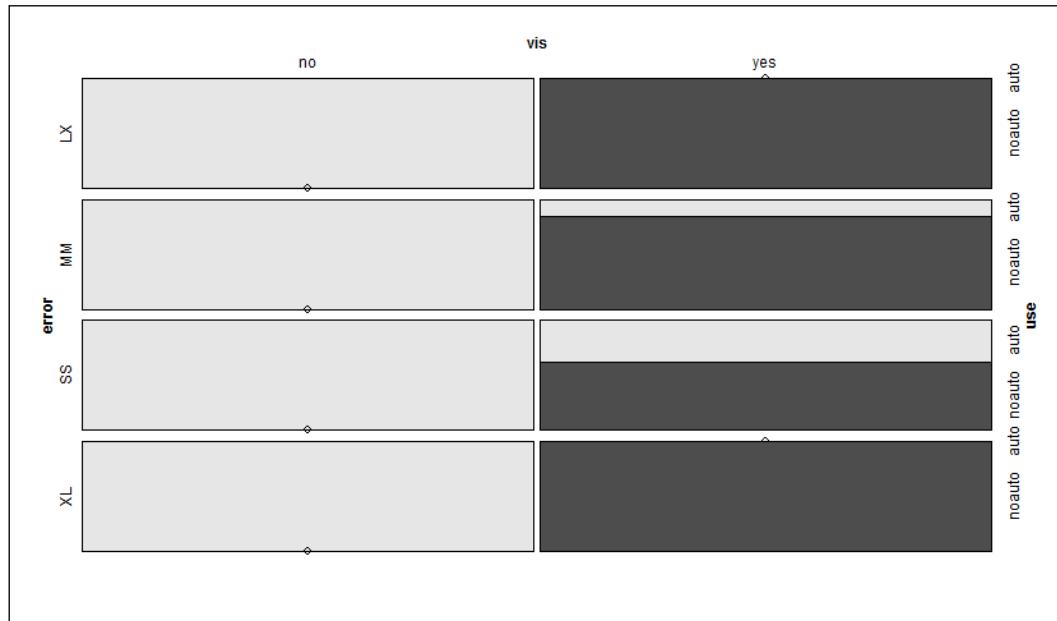
The output of the preceding command is as follows:



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. We will also drop the shading syntax, which drops the colored shading and residuals bar on the right. This can make for a cleaner plot:

```
> 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 is 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, we have all of them categorical. The `caret` package allows us to quickly create dummy variables as our input features:

```
> dummies = dummyVars(use~,shuttle, fullRank=TRUE)

> 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. 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 a 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,
times = 1)

> head(trainIndex)
  Resample1
[1,]      1
[2,]      2
[3,]
```

```
[4,]      9  
[5,]     10  
[6,]     11
```

The values in `trainIndex` provide us with a row number; in our case, 70 percent of the total row numbers in `shuttle.2`. It is now a simple case of creating the `train`/`test` datasets and to make sure that the response variable is balanced between them.

```
> shuttleTrain = shuttle.2[trainIndex,]  
  
> shuttleTest  = shuttle.2[-trainIndex,]  
  
> table(shuttleTrain$use)  
  
0 1  
81 99  
> table(shuttleTest$use)  
  
0 1  
30 46
```

Nicely done! We are now ready to begin building the neural networks.

Modeling and evaluation

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 this. 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)  
  
> n  
[1] "stability.xstab" "error.MM"          "error.SS"  
[4] "error.XL"         "sign.pp"           "wind.tail"  
[7] "magn.Medium"     "magn.Out"          "magn.Strong"  
[10] "vis.yes"         "use"
```

```
> 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          0.008356573154
reached.threshold 0.008074195519
steps          689.000000000000
Intercept.to.1layhid1 -4.285590085839
stability.xstab.to.1layhid1 1.929309791976
error.MM.to.1layhid1 -1.724487179503
error.SS.to.1layhid1 -2.634918418999
error.XL.to.1layhid1 -0.438426863298
sign.pp.to.1layhid1 -0.857597732824
wind.tail.to.1layhid1 0.095087222283
```

```

magn.Medium.to.1layhid1      -0.016988896781
magn.Out.to.1layhid1         1.861116820668
magn.Strong.to.1layhid1      0.121644784735
vis.yes.to.1layhid1          6.272628328980
Intercept.to.use             32.409085601639
1layhid.1.to.use             -67.336820525475

```

We can see that the error is extremely low at 0.0081. 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 threshold (default = 0.1). The highest weight of the first neuron is `vis.yes.to.1layhid1` at 6.27.

You can also look at what are known as generalized weights. According to the `neuralnet` package authors, the generalized weight is defined as the contribution of the i th covariate to the log-odds:

$$w_i = \frac{\partial \log\left(\frac{\theta(x)}{1-\theta(x)}\right)}{\partial x_i}$$

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:

```

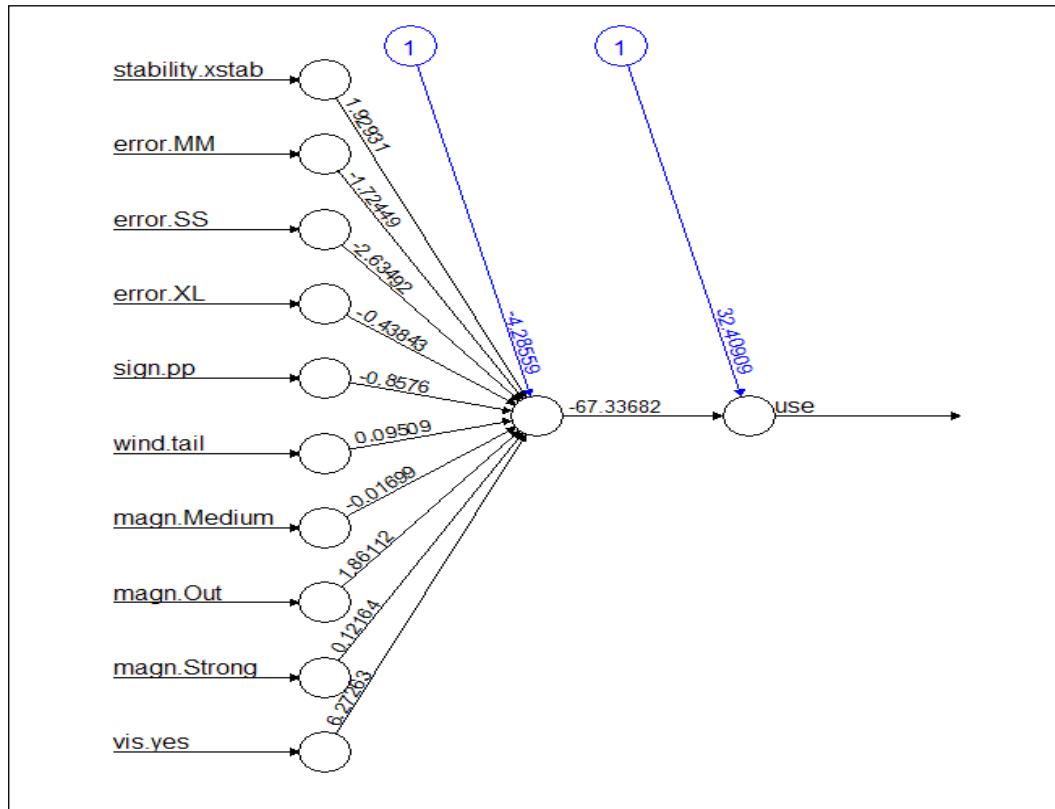
> head(fit$generalized.weights[[1]])
      [,1]      [,2]      [,3]      [,4]
1  -4.826708143 4.314287082 6.591985508 1.096847443
2  -4.751585064 4.247139344 6.489387581 1.079776065
6  -5.884240759 5.259548151 8.036290710 1.337166913
9  -11.346058891 10.141519613 15.495665693 2.578340208
10 -11.104906734 9.925969054 15.166316688 2.523539479
11 -10.952642060 9.789869358 14.958364085 2.488938025

```

To visualize the neural network, simply use the `plot()` function:

```
> plot(fit)
```

The following is the output of the preceding command:

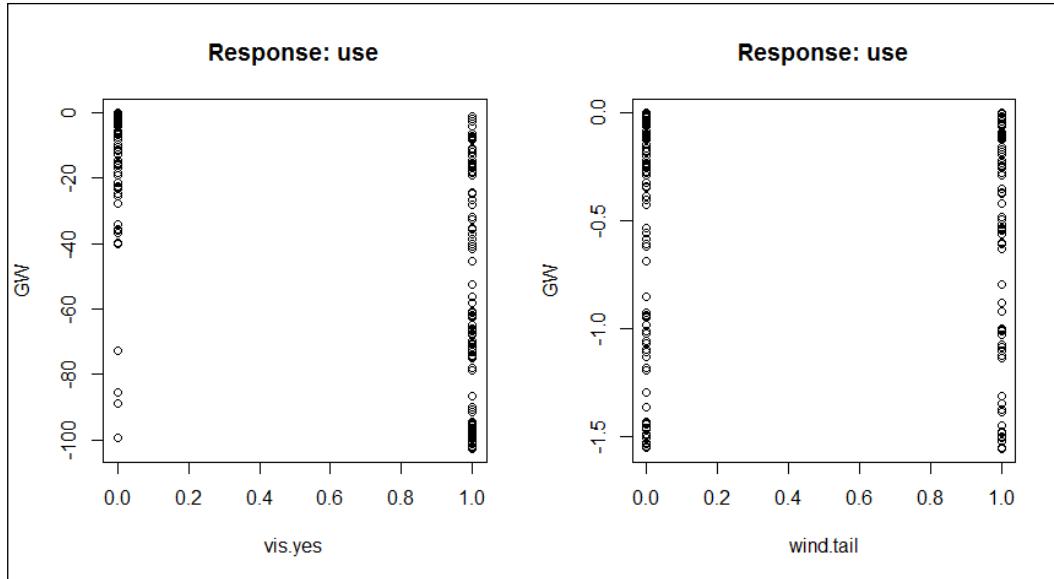


This plot shows the weights of the variables and intercepts. You can also examine the generalized weights in a plot. Let's look at vis.yes versus wind.tail, which has a low overall synaptic weight. Notice how vis.yes is skewed and wind.tail has an even distribution of weights, implying little predictive power:

```
> par(mfrow=c(1,2))

> gwplot(fit, selected.covariate = "vis.yes")

> gwplot(fit, selected.covariate = "wind.tail")
```



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`:

```
> res = compute(fit, shuttleTrain[,1:10])
```

```
> predTrain = res$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 percent accuracy. We will now hold our breath and see how it does on the test set:

```
> res2 = compute(fit, shuttleTest[,1:10])

> predTest = res2$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:

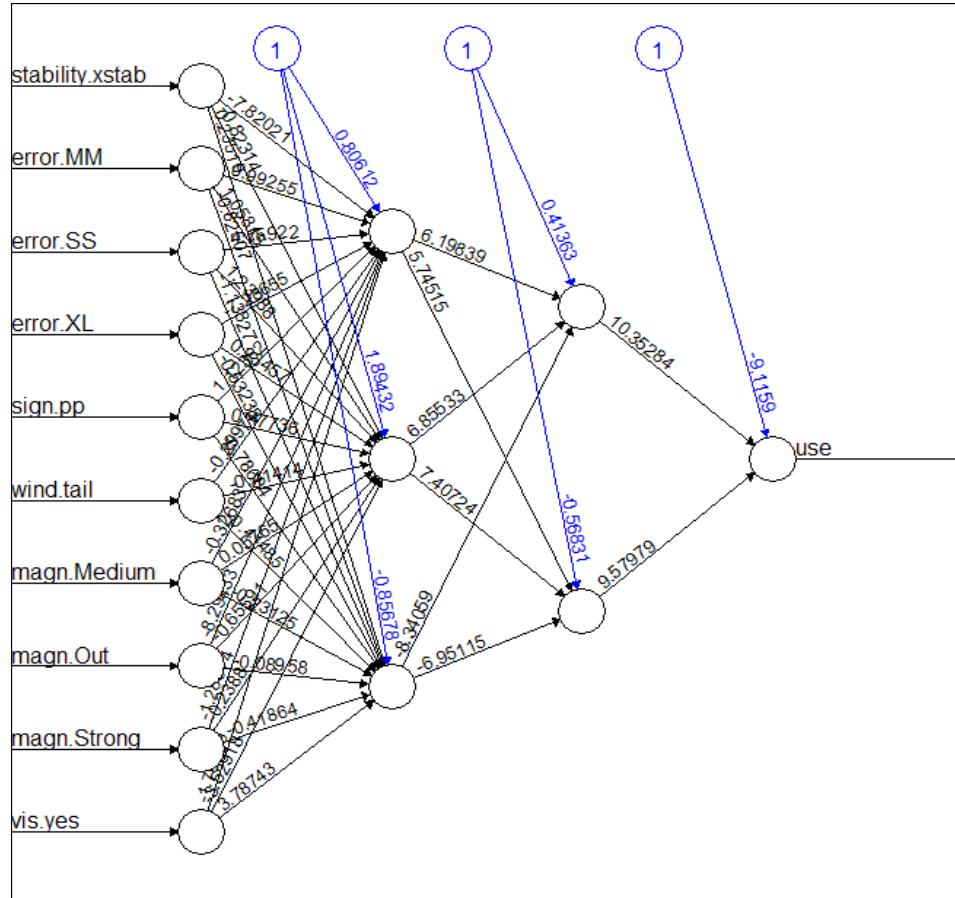
```
> which(predTest==1 & shuttleTest$use==0)
[1] 62
> shuttleTest[62,]
  stability.xstab error.MM error.SS error.XL sign.pp
203          0         1         0         0         1
  wind.tail magn.Medium magn.Out magn.Strong vis.yes
203          0         0         0         1         1
  use
203    0
```

It is row 62 in the test set and observation 203 in the full dataset. Can we improve on this result and achieve 100 percent accuracy in the test set? To do this, we will increase the complexity by specifying three neurons in the first layer and two neurons in the second layer:

```
> fit2 = neuralnet(form, data=shuttleTrain, hidden=c(3,2), err.fct="ce",
linear.output=FALSE)
```

The plot results now get quite busy and extremely hard to interpret:

```
> plot(fit2)
```



It's now time to see how this performs on both the datasets:

```
> res = compute(fit2, shuttleTrain[,1:10])

> predTrain = res$net.result

> predTrain = ifelse(predTrain >= 0.5, 1, 0)

> table(predTrain, shuttleTrain$use)

predTrain  0  1
      0 81  0
      1  0 99
```

Perform this on the `train` data as well:

```
> res2 = compute(fit2, shuttleTest[,1:10])

> predTest = res2$net.result

> predTest = ifelse(predTest>=0.5,1,0)

> table(predTest, shuttleTest$use)

predTest  0   1
      0 29   1
      1   1 45
```

However, we've now added a false negative to the mix. We added complexity but increased the out-of-sample variance, as follows:

```
> which(predTest==1 & shuttleTest$use==0)
[1] 62

> which(predTest==0 & shuttleTest$use==1)
[1] 63
```

The false positive observation has not changed from the prior fit and row 63 marks the false negative. In this case, adding complexity did not improve the `test` set's performance; in fact, it is less generalized than the simpler model.

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 the data that we used for some of the chapters: the Pima Indian diabetes data. In *Chapter 5, More Classification Techniques – K-Nearest Neighbors and Support Vector Machines*, the best classifier was the sigmoid kernel, Support Vector Machine. We've already gone through the business and data understanding work in that chapter, so in this section, we will focus on how to load the data in the H2O platform and run the deep learning code.

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/>.

Data preparation and uploading it to H2O

What we will do here is prepare the data, save it to the drive, and load it in H2O. The data is in two different datasets and we will first combine them. We will also need to scale the inputs. For labeled outcomes, the deep learning algorithm in H2O does not require numbered responses but factors, which means that we will not need to transform it. This code gets us to where we need to be. The `rbind()` function concatenates the datasets, as follows:

```
> data(Pima.tr)

> data(Pima.te)

> pima = rbind(Pima.tr, Pima.te)

> pima.scale = as.data.frame(scale(pima[,-8]))

> pima.scale$type = pima$type

> str(pima.scale)
'data.frame': 532 obs. of  8 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 ...
 $ type  : Factor w/ 2 levels "No","Yes": 1 2 1 1 1 2 1 1 1 2 ...
```

I will save this to the hard drive, but you can save it anywhere that is accepted by H2O. Let's make a note of the working directory as well:

```
> getwd()
[1] "C:/Users/clesmeister/chap7 NN"

> write.csv(pima.scale, file="pimaScale.csv", row.names=FALSE)
```

We can now connect to H2O and start an instance. Please note that your output will differ than what is displayed below:

```
> library(h2o)

> localH2O = h2o.init()
Successfully connected to http://127.0.0.1:54321/

> h2o.getConnection()
IP Address: 127.0.0.1
Port      : 54321
Session ID: _sid_8102ec2ab4585cc63b8186735b594e00
Key Count : 39
```

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`

There are a number of arguments that you can use to upload data, but the two that we need are the path of the file and a key specification. I like to specify the path outside of the function, as follows:

```
> path = "C:/Users/clesmeister/chap7 NN/pimaScale.csv"
```

We will specify the key with `destination_frame=" "` in the function. It is quite simple to upload the file and a percent indicator tracks the status:

```
> pima.hex = h2o.uploadFile(path=path, destination_frame="pima.hex")
|=====| 100%
```

The data is now in `H2OFrame`, which you can verify with `class()`, as follows:

```
> class(pima.hex)
[1] "H2OFrame"
attr(,"package")
[1] "h2o"
```

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:

```
> str(pima.hex)
Formal class 'H2OFrame' [package "h2o"] with 4 slots
  ..@ conn      :Formal class 'H2OConnection' [package "h2o"] with 3
  slots
    ... . . . @ ip      : chr "127.0.0.1"
    ... . . . @ port    : num 54321
    ... . . . @ mutable:Reference class 'H2OConnectionMutableState' [package
  "h2o"] with 2 fields
      ... . . . . $ session_id: chr "_sid_8102ec2ab4585cc63b8186735b594e00"
      ... . . . . $ key_count : int 43
      ... . . . .and 13 methods, of which 1 is possibly relevant:
      ... . . . .   initialize
    ..@ frame_id  : chr "pima.hex"
    ..@ finalizers: list()
    ..@ mutable    :Reference class 'H2OFrameMutableState' [package "h2o"]
with 4 fields
      ... . $ ast      : NULL
      ... . $ nrows    : num 532
      ... . $ ncols    : int 8
      ... . $ col_names: chr "npreg" "glu" "bp" "skin" ...
      ... .and 13 methods, of which 1 is possibly relevant:
      ... .   initialize
```

The `head()` and `summary()` functions work exactly the same. Here are the first six rows of our data in H2O along with a summary:

```
> head(pima.hex)
      npreg      glu      bp      skin      bmi
1  0.4477858 -1.1300306 -0.2847739 -0.1123474 -0.3909581
2  1.0516440  2.3861862 -0.1223077  0.3627626 -1.1321178
```

Neural Networks

```
3  0.4477858 -1.4203605  0.8524894  1.1229387  0.4228642
4 -1.0618597  1.4184201  0.3650908  1.3129827  2.1813017
5 -1.0618597 -0.4525944 -0.9346387 -0.3974135 -0.9431947
6  0.4477858 -0.7751831  0.3650908 -0.2073695  0.3937991

      ped          age   type
1 -0.4033309 -0.7075782   No
2 -0.9867069  2.1730387  Yes
3 -1.0070235  0.3145762   No
4 -0.7080796 -0.5217319   No
5 -1.0737779 -0.8005013   No
6 -0.3626978  1.8942693  Yes

> summary(pima.hex)

npreg          glu          bp
Min. : -1.062e+00 Min. : -2.098e+00 Min. : -3.859e+00
1st Qu.: -7.642e-01 1st Qu.: -7.220e-01 1st Qu.: -6.105e-01
Median : -4.613e-01 Median : -1.972e-01 Median : 3.918e-02
Mean   : -1.252e-15 Mean   : -1.557e-16 Mean   : -1.004e-17
3rd Qu.:  4.472e-01 3rd Qu.:  6.504e-01 3rd Qu.:  6.889e-01
Max.   :  4.071e+00 Max.   :  2.515e+00 Max.   :  3.127e+00

skin          bmi          ped
Min. : -2.108e+00 Min. : -2.135e+00 Min. : -1.213e+00
1st Qu.: -6.829e-01 1st Qu.: -7.313e-01 1st Qu.: -7.116e-01
Median : -1.847e-02 Median : -1.715e-02 Median : -2.541e-01
Mean   :  5.828e-17 Mean   :  3.085e-16 Mean   :  6.115e-17
3rd Qu.:  6.459e-01 3rd Qu.:  5.798e-01 3rd Qu.:  4.490e-01
Max.   :  6.634e+00 Max.   :  4.972e+00 Max.   :  5.564e+00

age          type
Min. : -9.863e-01 No  : 355
1st Qu.: -8.024e-01 Yes:177
Median : -3.396e-01
Mean   : -1.876e-16
3rd Qu.:  5.915e-01
Max.   :  4.589e+00
```

Create train and test datasets

You can upload your own `train` and `test` partitioned datasets to H2O or you can use the built-in functionality. I will demonstrate the latter with a 70/30 split. The first thing to do is create a vector of random and uniform numbers for our data:

```
> rand = h2o.runif(pima.hex, seed = 123)
```

You can then build your partitioned data and assign it with your desired `key` name, as follows:

```
> train = pima.hex[rand <= 0.7, ]  
  
> train = h2o.assign(train, key = "train")  
  
> test = pima.hex[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 8:

```
> h2o.table(train[,8])  
H2OFrame with 2 rows and 2 columns  
  type Count  
1  No    253  
2  Yes   124  
  
> h2o.table(test[,8])  
H2OFrame with 2 rows and 2 columns  
  type Count  
1  No    102  
2  Yes    53
```

This appears well and good and with that, we can 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/>.

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)`.

The critical items to specify in this example will be as follows:

- The input variable
- The response variable
- The training data with `training_frame = train`
- The testing data with `validation_frame = test`
- An initial seed for the sampling
- The variable importance with `variable_importances = TRUE`

When combined, this code will create the deep learning object, as follows:

```
> dlmodel <- h2o.deeplearning(x=1:7, y=8, training_frame = train,
validation_frame = test, seed = 123, variable_importances = TRUE)
| ====== | 100%
```

An indicator bar tracks the progress and with this relatively small dataset, it only takes a couple of seconds.

Calling the `dlmodel` object produces quite a bit of information. The two things that I will show here are the confusion matrices for the `train` and `test` sets:

```
> dlmodel
Model Details:
=====
      No Yes    Error     Rate
No      204  49  0.193676  =49/253
Yes      29   95  0.233871  =29/124
Totals  233 144  0.206897  =78/377

      No Yes    Error     Rate
No      86   16  0.156863  =16/102
Yes      18   35  0.339623  =18/53
Totals 104   51  0.219355  =34/155
```

The first matrix shows the performance on the `test` set with the columns as the predictions and rows as the actuals and the model achieved 79.3 percent accuracy. The false positive and false negative rates are roughly equivalent. The accuracy on the `test` data was 78 percent, but with a higher false negative rate.

A further exploration of the model parameters can also be called, which produces a lengthy output:

```
> dlmodel@allparameters
```

Let's have a look at the variable importance:

```
> dlmodel$model$variable_importances
Variable Importances:
  variable relative_importance scaled_importance percentage
1      glu        1.000000        1.000000    0.156574
2      bp         0.942461        0.942461    0.147565
3     npreg        0.910888        0.910888    0.142622
4      age         0.902482        0.902482    0.141305
5      skin        0.894196        0.894196    0.140008
6      ped         0.882988        0.882988    0.138253
7      bmi         0.853734        0.853734    0.133673
```

The variable importance 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 they are all relatively similar in their contribution and probably the table is not of much value, which is a common criticism of the neural network techniques. The importance is also subject to the sampling variation, and if you change the seed value, the order of the variable importance can change quite a bit.

You are also able to see the predicted values and put them in a data frame, if you want. We will first create the predicted values:

```
> dlPredict = h2o.predict(dlmodel,newdata=test)

> dlPredict
H2OFrame with 155 rows and 3 columns

First 10 rows:
  predict      No      Yes
1      No 0.9973673 0.002632645
2      Yes 0.2167641 0.783235848
```

```
3      Yes 0.1707465 0.829253495
4      Yes 0.1609832 0.839016795
5      No 0.9740857 0.025914235
6      No 0.9957688 0.004231210
7      No 0.9989172 0.001082811
8      No 0.9342141 0.065785915
9      No 0.7045852 0.295414835
10     No 0.9003637 0.099636205
```

As it defaults to the first ten observations, putting it in a data frame will give us all of the predicted values, as follows:

```
> dlPred = as.data.frame(dlPredict)
> head(dlPred)
  predict      No      Yes
1      No 0.9973673 0.002632645
2      Yes 0.2167641 0.783235848
3      Yes 0.1707465 0.829253495
4      Yes 0.1609832 0.839016795
5      No 0.9740857 0.025914235
6      No 0.9957688 0.004231210
```

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 in order to optimize the model fit. 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 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 hyperparameter tuning, are the quintessential blackboxes, and mostly not interpretable. 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 prior 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 s 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.

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". 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, but other distance measures are 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 datasets 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	Distance between two clusters is the maximum distance between an observation in one cluster and an observation in the other cluster
Single	Distance between two clusters is the minimum distance between an observation in one cluster and an observation in the other cluster
Average	Distance between two clusters is the mean distance between an observation in one cluster and an observation in the other cluster
Centroid	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. 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:

$$d(A, B) = \text{sqrt}((5-3)^2 + (3-5)^2) = \text{sqrt}(8) = 2.83$$

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 detail in 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. 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 plays out for a situation where we will specify three clusters:

1. Each observation is randomly assigned by the algorithm to one of the three clusters.
2. Each cluster has a centroid calculated by the algorithm, which is a vector of the variable means for the observations. For example, if you have five input variables, your centroid would be a vector of five values.
3. Reshuffle the observations to the cluster with the centroid; this minimizes the Euclidean distance.
4. Iterate through steps 2 and 3 until the within-cluster variation improves.

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 are 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, I prefer to use the Gower dissimilarity coefficient to turn mixed data to the proper feature space. In R, you can even include factors as input variables to cluster. 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. 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, they make an effective method together to handle the 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:

$$S_{ij} = \frac{\text{sum}(W_{ijk} * S_{ijk})}{\text{sum}(W_{ijk})}$$

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 $x_{ij} = x_{ik}$, 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.

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 ...
 $ 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 and conduct PAM.

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? 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 dozens of them.

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. 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 this stage set, let's walk through an example 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 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.

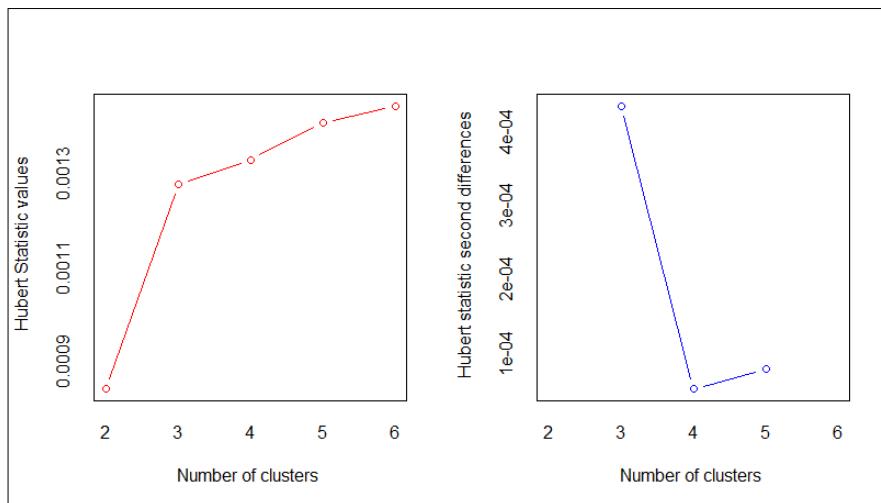
* Among all indices:

- * 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 *****

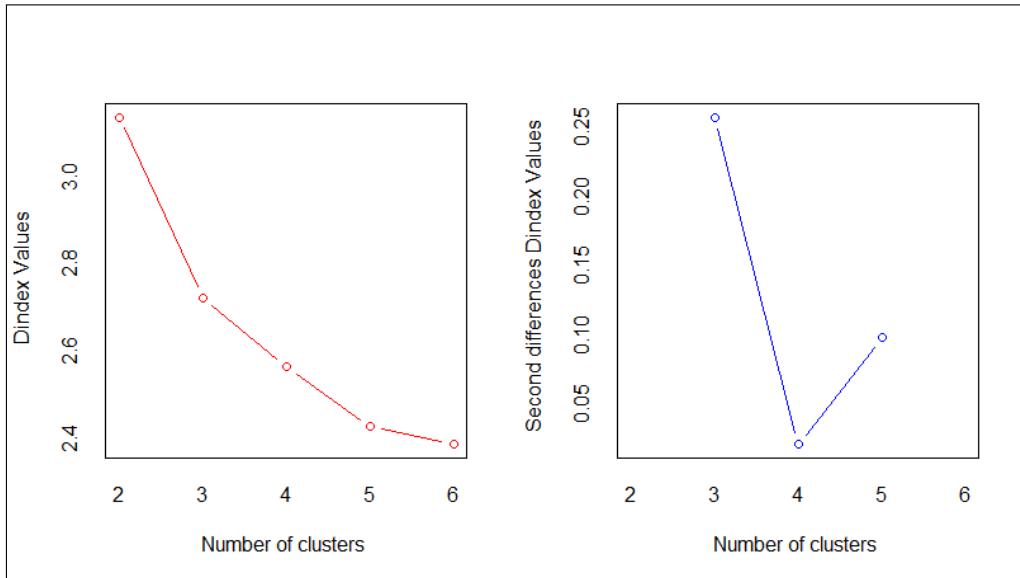
* According to the majority rule, the best number of clusters is 3

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 that 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:



Cluster Analysis

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 at 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:

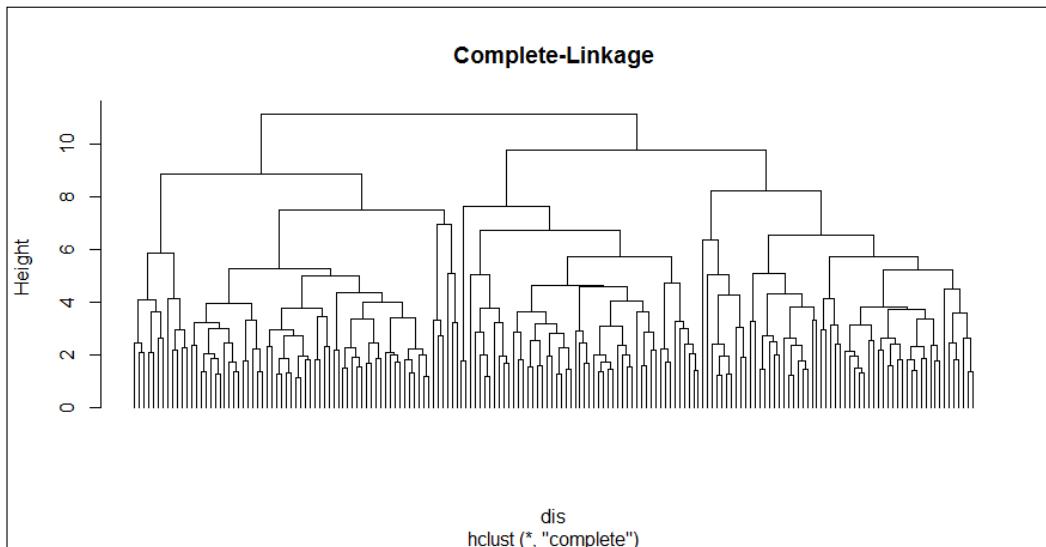
```
> dis = dist(df, method="euclidean")
```

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) tell 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.

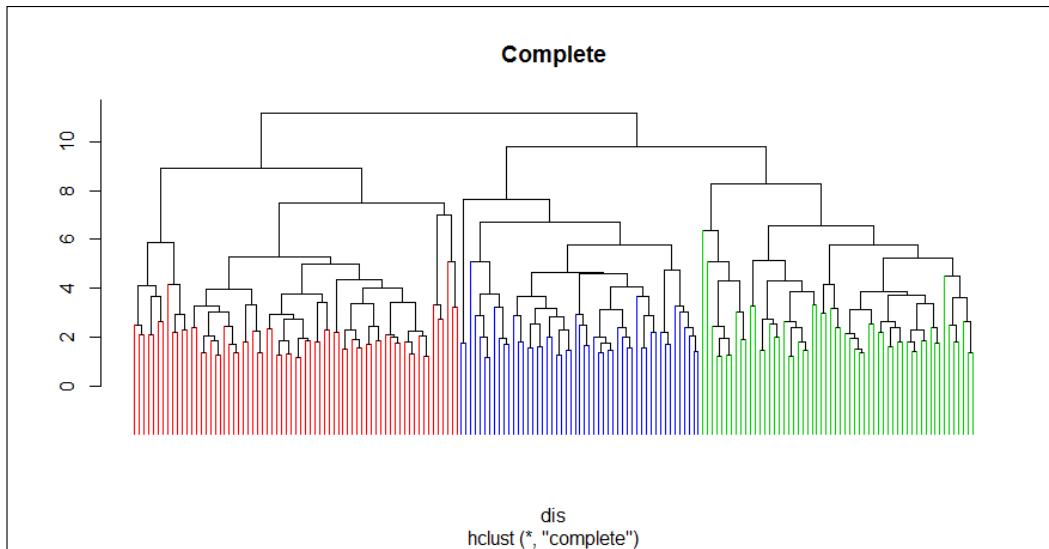
To aid in visualizing the clusters, you can produce a colored dendrogram using the `spacel` package. To color the appropriate number of clusters, you need to cut the dendrogram tree to the proper number of clusters using the `cutree()` function. This will also create the cluster label for each of the observations:

```
> comp3 = cutree(hc, 3)
```

Cluster Analysis

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 compared 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 apriori 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:

```
> 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.

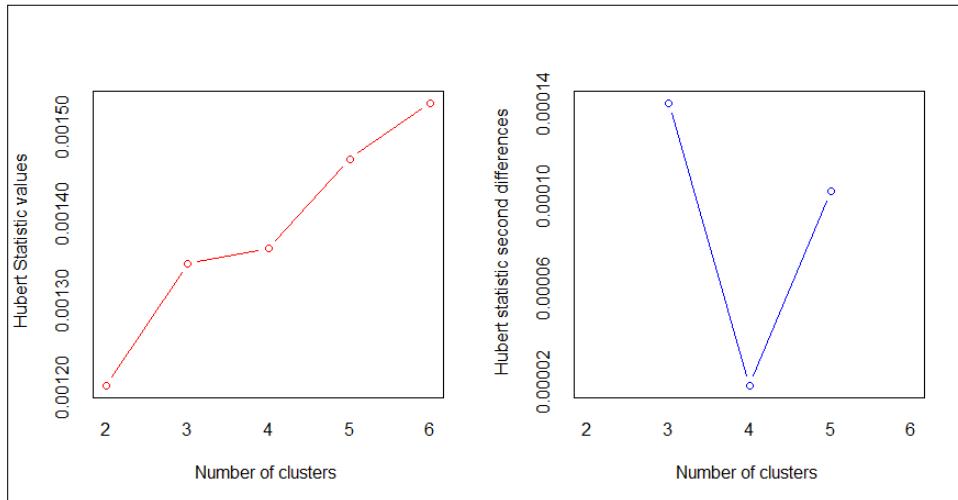
- * Among all indices:
- * 2 proposed 2 as the best number of clusters
- * 18 proposed 3 as the best number of clusters
- * 2 proposed 6 as the best number of clusters

***** Conclusion *****

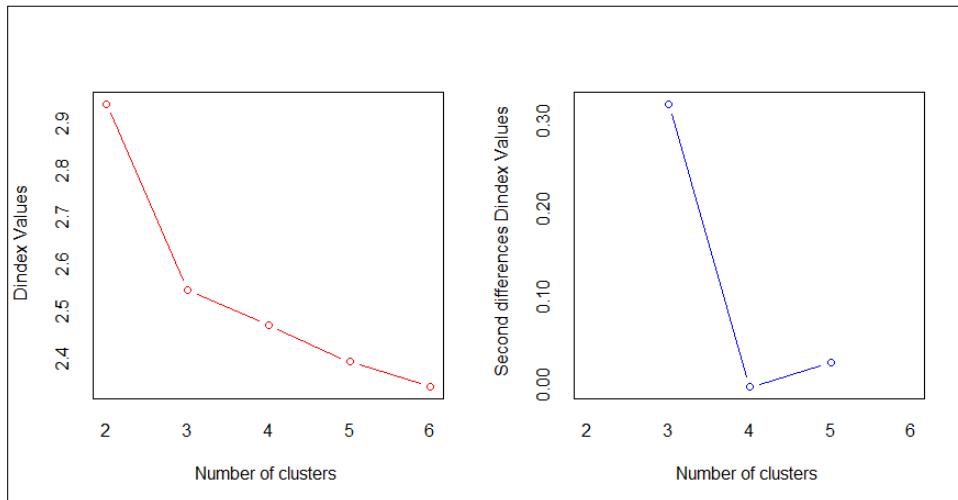
* According to the majority rule, the best number of clusters is 3

Cluster Analysis

This time around also, the majority rules was for a three cluster solution. Looking at the Hubert Index, the best solution is a three cluster as well:

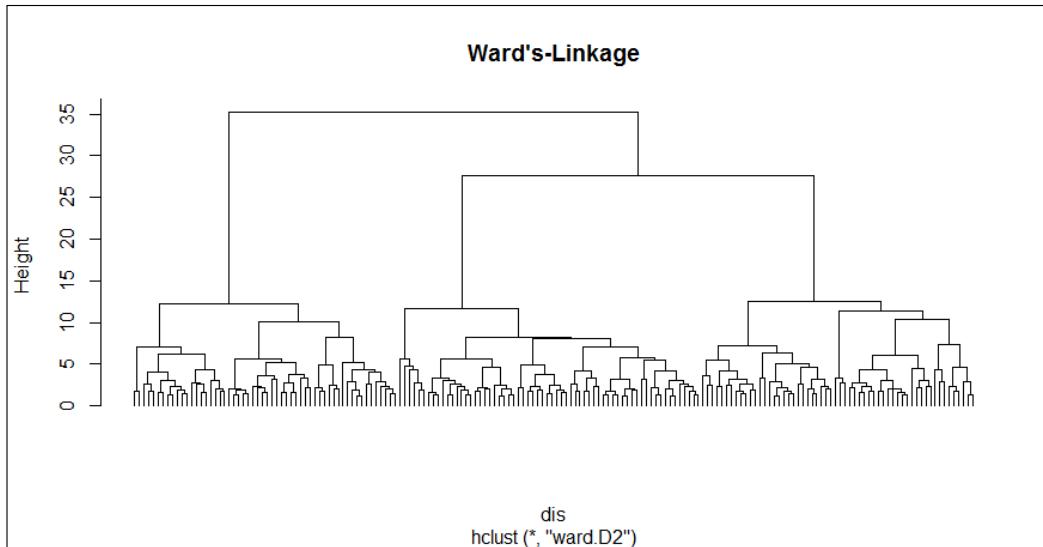


The Dindex adds further support to the three cluster solution:



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 with 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 gave 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 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)`.

With the `boxplot()` function in R, your variables for the *x* and *y* axis need to be in the same dataset and so we will need to turn the clusters from both the methods to variables in the wine dataset as follows:

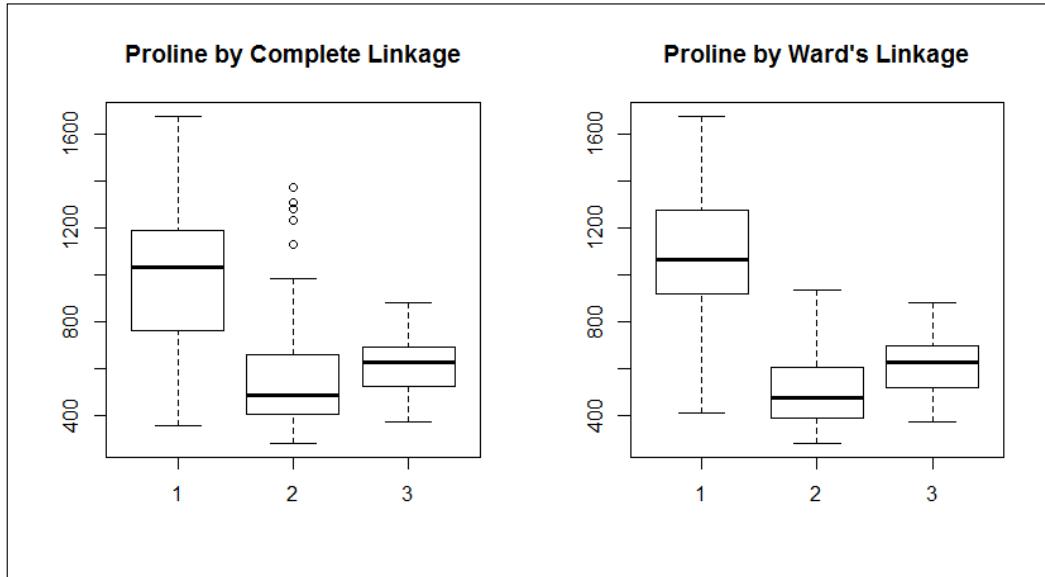
```
> wine$comp_cluster = comp3
```

```
> wine$ward_cluster = ward3
```

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(Proline~comp_cluster, data=wine, main="Proline by Complete Linkage")
```

```
> boxplot(Proline~ward_cluster, 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 would be 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

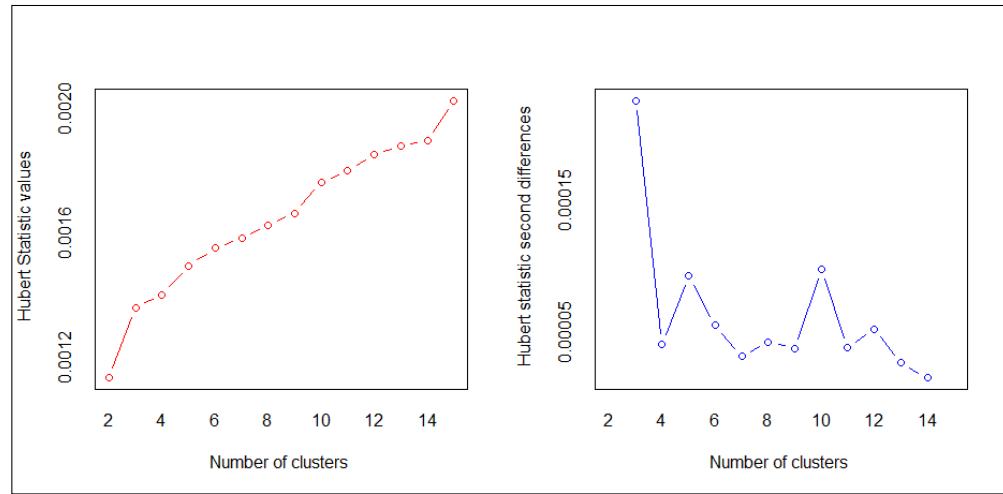
As we did with hierarchical clustering, we can also use `NbClust()` to determine the optimum number of clusters for k-means. All you need to do is specify `kmeans` as the method in the function. Let's also loosen up the maximum number of clusters to 15. I've abbreviated the following output to just the majority rules portion:

```
> 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 in 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
```

Cluster Analysis

```
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

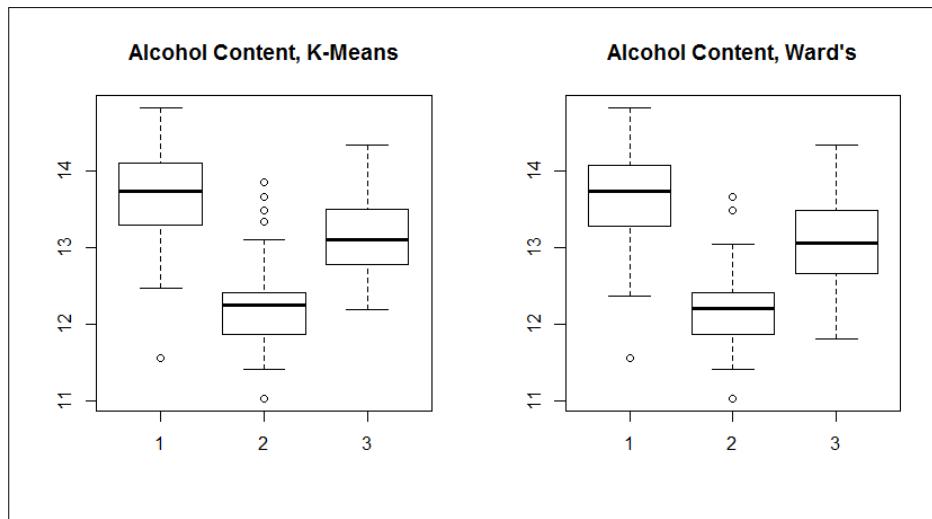
  Proline
1  1.1220202
2 -0.7517257
3 -0.4059428
```

Note that cluster one has, on an 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:

```
> wine$km_cluster = km$cluster

> boxplot(Alcohol~km_cluster, data=wine, main="Alcohol Content, K-Means")

> boxplot(Alcohol~ward_cluster, 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 a final example.

Clustering with mixed data

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 as well as incorporate the cultivar class as a factor. The easiest step is to incorporate the cultivars:

```
> df$class = as.factor(wine$Class)
```

To change alcohol, it also takes only one line of code but we will need to utilize the `ifelse()` function and 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`:

```
> df$Alcohol = as.factor(ifelse(df$Alcohol>0,"High","Low"))
```

Check the structure to verify that it all worked:

```
> str(df)
'data.frame': 178 obs. of 17 variables:
 $ Alcohol      : Factor w/ 2 levels "High","Low": 1 1 1 1 1 1 1 1 1 1 ...
 $ 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 ...
 $ comp_cluster: num -1.1 -1.1 -1.1 -1.1 0.124 ...
 $ ward_cluster: num -1.16 -1.16 -1.16 -1.16 -1.16 ...
```

Cluster Analysis

```
$ km_cluster : num -1.18 -1.18 -1.18 -1.18 -1.18 ...
$ class       : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 ...
```

We are now ready to create the dissimilarity matrix using the `daisy()` function from the `cluster` package and specifying the method as `gower`:

```
> disMat = daisy(df, 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(disMat, k=3)
```

```
> table(pamFit$clustering)
```

```
1 2 3
60 69 49
```

Now, let's see how it does compared to the cultivar labels:

```
> table(pamFit$clustering, wine$Class)
```

```
1 2 3
1 59 1 0
2 0 69 0
3 0 1 48
```

Well, no surprise that by actually including the cultivar class, the clusters almost achieved a perfect match. So, 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, so we will do this as a .csv. If you want, you can also export the table as a .pdf, HTML, or the LaTeX format:

```
> df$cluster = pamFit$clustering
```

```
> group = compareGroups(cluster~, data=df)
```

```
> clustab = createTable(group)

> clustab

-----Summary descriptives table by 'cluster'-----


```

	1 N=60	2 N=69	3 N=49	p.overall
Alcohol:				
High	58 (96.7%)	6 (8.70%)	28 (57.1%)	<0.001
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:				
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%)	

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")
```

Cluster Analysis

If you open this file, you will get this table, which is conducive to further analysis and can be easily manipulated for the presentation purposes:

	1 N=60	2 N=69	3 N=49	p.overall
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%)	

Summary

In this chapter, we started to explore unsupervised learning techniques. We focused on cluster analysis to both provide data reduction and data understanding of the observations. Three methods were introduced: the traditional hierarchical and k-means clustering algorithms along with the Gower metric and PAM for mixed data. We applied these three 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 the unsupervised learning techniques. In the prior 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. It may additionally suffer from high dimensionality or, as it is 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.

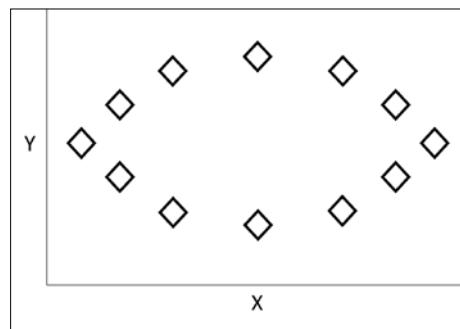
From over a decade of either doing or supporting analytics using PCA, it has been my experience that it is widely used but poorly understood, especially among people who don't do the analysis but consume the results. It is intuitive to understand that you are creating a new variable from the other correlated 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.

An overview of the principal components

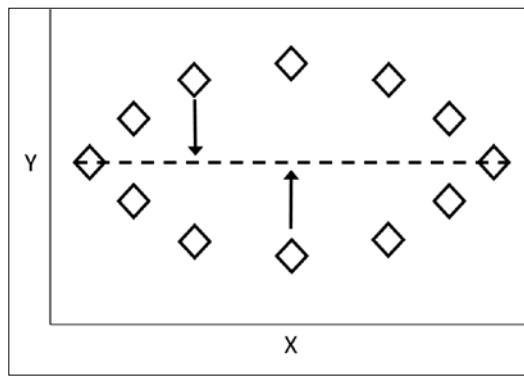
PCA is the process of finding the principal components. What exactly are these? We can consider that a component is a *normalized linear combination of the features*. (James, 2012) The first principal component in a dataset is the linear combination that captures the maximum variance in the data. A second 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.

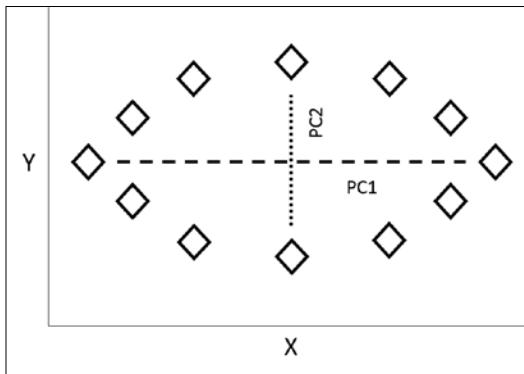
Now, what is this direction described here and how is the linear combination determined? The best way to grasp this subject is with a visualization. Let's take a small dataset with two variables and plot it. PCA is sensitive to scale, so the data has been scaled with a mean of zero and standard deviation 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 $PC1 = a_{11}X_1 + a_{12}X_2$, 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: $Z_{11} = a_{11} * (X_{11} - \text{average of } X_1) + a_{12} * (X_{12} - \text{average of } X_2)$. For **PC2** and the first observation, the equation would be $Z_{12} = a_{21} * (X_{11} - \text{average of } X_1) + a_{22} * (X_{12} - \text{average of } X_2)$. 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.

Recall that the equation for the first principal component is $PC1 = a_{11}X_1 + a_{12}X_2$. 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. 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.

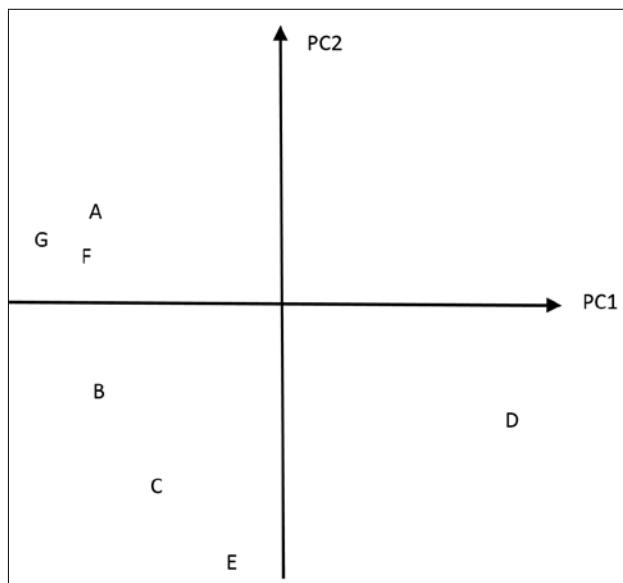
The eigenvalue criterion is certainly not a hard-and-fast rule and must be balanced with your knowledge of the data and business problem at hand. Once you have selected the number of principal components, you can rotate them in order to simplify their interpretation.

Rotation

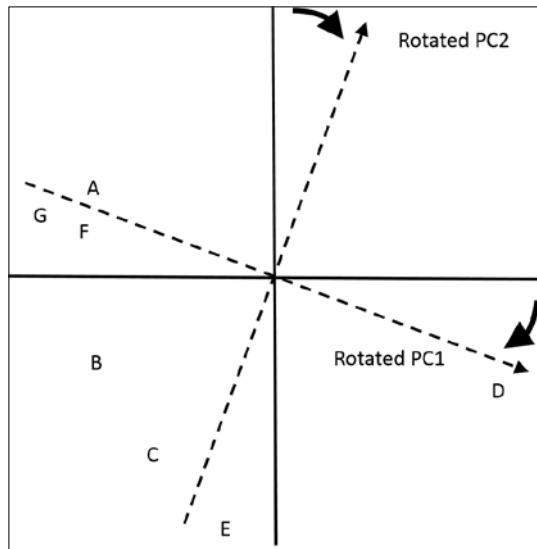
Should you rotate or not? As stated previously, rotation helps in the interpretation of the principal components by modifying the loadings of each variable. The overall variation explained by the rotated number of components will not change, but the contributions to the total variance explained by each 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 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 and download the data from a table at www.nhl.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. There will be 14 variables included in the PCA and we can reasonably expect to end up with less than half a dozen principal components.

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(corrplot) #correlation plot

> library(FactoMineR) #additional PCA analysis

> library(ggplot2) #support scatterplot

> library(GPArotation) #supports rotation

> library(psych) #PCA package
```

The data is available online as a comma delimited file. My original intention was to show how to use R to scrape a table, in this case, on the website nhl.com. However, with the number of periodic changes to websites, the code to do this may become useless. Therefore, I've compiled the data and stored it on textloader.com. The first thing to do is create an object containing the URL as follows:

```
> url="http://textuploader.com/ae6t4/raw"
```

Now, just create the dataset, calling it `nhl` using the `read.csv()` function with `as.data.frame()` as a wrapper. Also, as no headings exist in the data, specify `header = FALSE`:

```
> nhl = as.data.frame(read.csv(url, header=FALSE))  
> nhl
```

To verify that we have the correct data, let's use the data structure function, `str()`. For brevity, I've included only the first few lines of the output of the command:

```
> str(nhl)  
'data.frame': 30 obs. of 25 variables:  
 $ V1 : int 1 2 3 4 5 6 7 8 9 10 ...  
 $ V2 : Factor w/ 30 levels "ANAHEIM","ARIZONA",... : 20 25 1 16 26 7 28 17  
 19 15 ...  
 $ V3 : int 82 82 82 82 82 82 82 82 82 82 ...  
 $ V4 : int 53 51 51 50 50 48 48 47 47 46 ...
```

The next thing that we will need to do is look at the variable names. When downloading an HTML table, this can produce the following nonsensical names and we will need to specify our own:

```
> names(nhl)  
[1] "V1"   "V2"   "V3"   "V4"   "V5"   "V6"   "V7"   "V8"   "V9"   "V10"  "V11"  
"V12"  
[13] "V13"  "V14"  "V15"  "V16"  "V17"  "V18"  "V19"  "V20"  "V21"  "V22"  "V23"  
"V24"  
[25] "V25"
```

To change all the variable names, we will use the following `names()` function, putting the new names in the `c()` function, which stands for combine:

```
> names(nhl) = c("rank", "team", "played", "wins", "losses", "OTL", "pts", "ROW",  
,"HROW", "RROW", "ppc", "gg", "gag", "five", "PPP", "PKP", "shots", "sag", "sc1", "tr1",  
"lead1", "lead2", "wop", "wosp", "face")
```

With the variables renamed, let's go over what they mean:

- `rank`: This is the rank of a team's total points
- `team`: This is the team's city
- `played`: This is the games that are played
- `wins`: This is the number of total wins
- `losses`: This is the number of total losses in regulation
- `OTL`: This is the number of total losses in overtime
- `pts`: This is the number of total points
- `RW`: This is the number of regulation plus overtime wins
- `HROW`: This is the number of home regulation plus overtime wins
- `RROW`: This is the number of road regulation plus overtime wins
- `ppc`: This is the percentage of the points per games played
- `gg`: This is the number of goals per game
- `gag`: This is the number of goals allowed per game
- `five`: This is the 5-on-5 goals against/for ratio that is both the teams' full strength
- `PPP`: This is the percentage of the time goals scored on the power play
- `PKP`: This is the percentage of the time goals allowed on the power play
- `shots`: This is the number of shots on the goal per game
- `sag`: This is the number of shots on the goal allowed per game
- `sc1`: This is the winning percentage when scoring first
- `tr1`: This is the winning percentage when trailing first
- `lead1`: This is the winning percentage when leading after the first period
- `lead2`: This is the winning percentage when leading after the second period
- `wop`: This is the winning percentage when outshooting an opponent
- `wosp`: This is the winning percentage when outshot by an opponent
- `face`: This is the percentage of the faceoffs won

One of the things that you can do with the data in R is sort it by one or more variables. Here is an example to identify the team with the fewest goals per game and also the team with the most goals per game. If the data was numeric, you could use the `max()` and `min()` functions. As the variables are characters, we will use the `order()` function to perform a sorting on our variable of interest and then call the appropriate row and column in order to identify the teams. This function defaults to an ascending order, but if your data is numeric, you can also sort the data in a descending order using the minus sign in front of your variable. Here is the code to sort and identify our teams of interest:

```
> nhl=nhl[order(nhl$gg),]  
  
> nhl[1,2]  
[1] BUFFALO  
  
> nhl[30,2]  
[1] TAMPA BAY
```

Note that once we sorted it, the process to identify a team was as simple as specifying the appropriate row (1 for the worst and 30 for the best) and the column two for the team name. As you can see, `BUFFALO` is the worst and `TAMPA BAY` is the best for average goals per game.

As we are concerned about creating the principal components in order to predict a team's points, we do not need to include a few of the variables in the data. In fact, let's only focus on variables 12 through 25 and create a new data frame call `pca.df` and drop columns 1 through 11, as follows:

```
> pca.df = nhl[,c(-1:-11)]
```

The next order of business is to convert the data frame's values to numeric; recall that all the variables are currently characters. For this task, we will break out the `lapply()` function, which applies another function over a list or vectors. So, what we want to apply over the data frame in order to convert the values to numeric is `as.numeric()`. Additionally, when we are done applying this function, we want `pca.df` to still be a data frame. Therefore, we will put `lapply()` and `as.numeric()` inside `as.data.frame()`.

This is all easy for me to say, but it is actually rather elegant and powerful to transform the values in a data frame:

```
> pca.df = as.data.frame(lapply(pca.df , as.numeric))
```

Anytime that you create a subset and/or make a major transformation, it is probably a good idea to check the structure after this transformation. In our case, we want to make sure that the values are indeed numeric, as follows:

```
> str(pca.df)
'data.frame': 30 obs. of 14 variables:
 $ gg    : num  1.87 2.01 2.15 2.23 2.35 2.42 2.51 2.55 2.55
2.58 ...
 $ gag   : num  3.28 3.26 2.55 2.67 3.37 2.6 3.13 2.45 2.72
2.72 ...
 $ five  : num  0.61 0.62 0.93 0.76 0.68 1.03 0.8 1.04 0.99
1.01 ...
 $ PPP   : num  13.4 20 19.3 18.8 17.7 16.3 15.9 17.8 15
23.4 ...
 $ PKP   : num  75.1 76.7 80.6 84.7 76.7 80 80.4 82 84.6
77.1 ...
 $ shots: num  24.2 29.2 24.5 30.8 28.4 30.7 29.2 31 27.9
29.4 ...
 $ sag   : num  35.6 33.2 30.7 27.3 30 29.6 33.5 29.9 33.2
30.3 ...
 $ sc1   : num  0.528 0.424 0.533 0.611 0.471 0.697 0.677
0.674 0.676 0.688 ...
 $ tr1   : num  0.087 0.204 0.216 0.174 0.167 0.306 0.176
0.308 0.311 0.22 ...
 $ lead1: num  0.563 0.522 0.593 0.708 0.435 0.789 0.762
0.742 0.696 0.731 ...
 $ lead2: num  0.647 0.824 0.781 0.759 0.63 0.75 0.826
0.871 0.852 0.864 ...
 $ wop   : num  0.5 0.385 0.235 0.423 0.333 0.386 0.375
0.512 0.6 0.475 ...
 $ wosp  : num  0.246 0.259 0.438 0.25 0.273 0.541
0.382 0.471 0.418 0.341 ...
 $ face  : num  44.9 51.8 47.3 53 48.2 48.6 49 53.6 50.8
51.1 ...
```

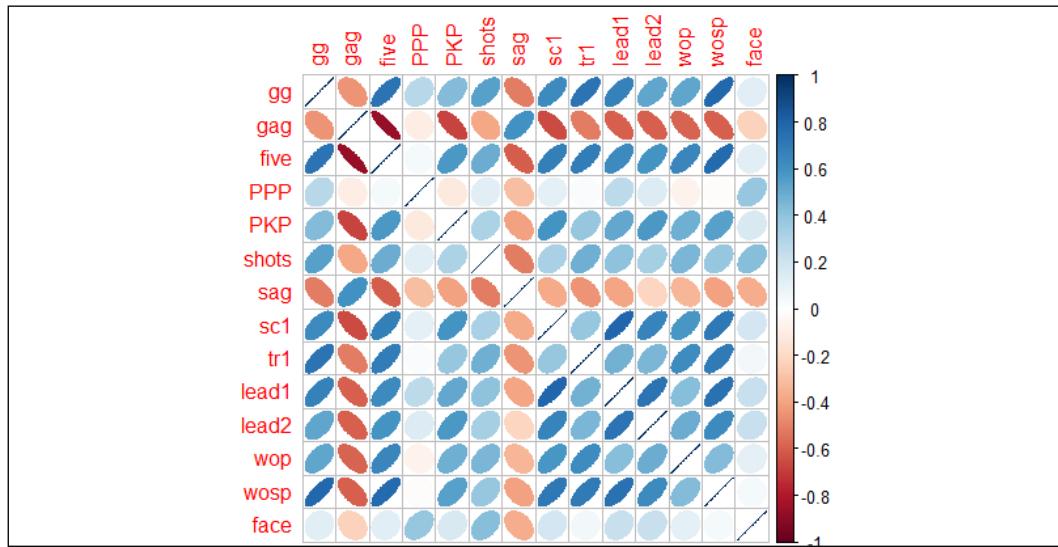
There you have it! Now, let's do what we did in the prior chapters and build a correlation matrix and plot it using a function from the `corrplot` package. First, create an object using the `cor()` function, which builds the correlation matrix, as follows:

```
> nhl.cor = cor(pca.df)
```

You can call `nhl.cor` if you want, but I am very partial to plotting this data. We will do a matrix plot of the correlation values. In the syntax of the `corrplot()` function we will specify the method as `ellipse`:

```
> corrplot(nhl.cor, method="ellipse")
```

The following is the output of the preceding command:



The first thing that jumps out and actually takes me by surprise is that the faceoff percentage and power play percentage don't show a strong correlation with anything. They do show a small correlation with each other. Another thing of interest is how strongly and negatively correlated are the 5-on-5 ratio and goals allowed per game, but it is not quite as strong as the correlation between playing full strength and the goals scored per game. If you follow hockey, you may find other interesting correlations here, but one thing is clear and that is the fact that many of the variables are highly correlated. As such, this should be a good dataset to extract several principal components. If we had a few, if any, correlations in this or any other dataset, then mostly any effort to extract the components would be futile.

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

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. For rotation with this package, you will also need to load `GPArotation`.

Component extraction

To extract the components with the `psych` package, you will use the `principal()` function. The syntax will include the data (`pca.df`) and number of the components to extract. We will try 5, and we will state that we do not want to rotate the components at this time. You can choose not to specify `nfactors`, but the output would be rather lengthy as it would produce $k-1$ components:

```
> pca = principal(pca.df, nfactors=5, rotate="none")
```

We will examine the components by calling the `pca` object that we created, as follows:

```
> pca
```

```
Principal Components Analysis
Call: principal(r = pca.df, nfactors = 5, rotate = "none")
Standardized loadings (pattern matrix) based upon
correlation matrix

      PC1    PC2    PC3    PC4    PC5     h2     u2
gg      0.83   0.05 -0.10   0.45   0.04  0.90  0.10
gag     -0.82   0.04 -0.01   0.35   0.31  0.89  0.11
five    0.90  -0.11 -0.15  -0.02  -0.20  0.89  0.11
PPP     0.16   0.76   0.30   0.34  -0.20  0.86  0.14
PKP     0.70  -0.20   0.12  -0.44  -0.08  0.75  0.25
```

Principal Components Analysis

shots	0.61	0.35	-0.43	-0.04	0.38	0.81	0.19
sag	-0.63	-0.42	0.33	0.12	0.45	0.90	0.10
scl	0.82	-0.11	0.36	-0.06	0.02	0.81	0.19
tr1	0.74	-0.14	-0.40	0.31	0.08	0.83	0.17
lead1	0.82	0.04	0.40	0.15	0.06	0.86	0.14
lead2	0.75	-0.10	0.42	-0.04	0.25	0.81	0.19
wop	0.71	-0.20	-0.24	-0.16	0.24	0.68	0.32
wosp	0.84	-0.24	0.07	0.27	-0.03	0.84	0.16
face	0.28	0.74	0.06	-0.36	0.31	0.86	0.14

	PC1	PC2	PC3	PC4	PC5
SS loadings	7.19	1.63	1.13	1.00	0.75
Proportion Var	0.51	0.12	0.08	0.07	0.05
Cumulative Var	0.51	0.63	0.71	0.78	0.84
Proportion Explained	0.61	0.14	0.10	0.09	0.06
Cumulative Proportion	0.61	0.75	0.85	0.94	1.00

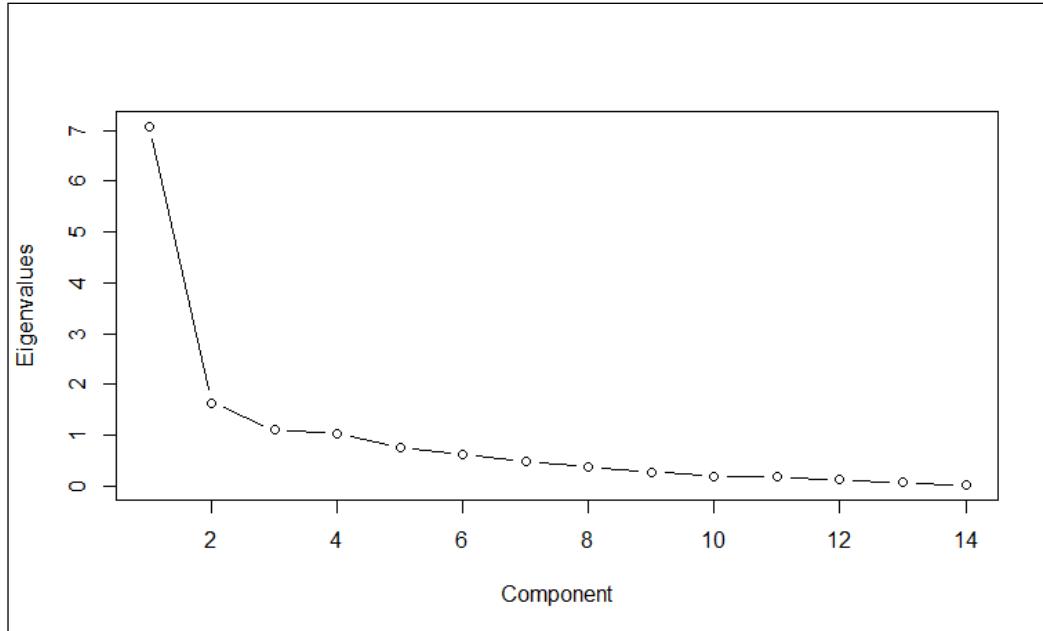
At first glance, this looks quite a bit to digest. First of all, let's disregard the h_2 and u_2 columns. We will come back to h_2 after the rotation. There are two important things to digest here in the output. The first is the variable loadings for each of the five components that are labeled PC1 through PC5. We see with component one that almost all the variables have high loadings except for the power play percentage, PPP, and the faceoff win percentage, face. These two variables have high loading values on component two. Component three is quite a hodgepodge of the mid-value loadings. The highest two variables are tr1 and PKP. Component four seems rather odd and difficult for us to pull out a meaningful construct. So, we will move on to the second part to examine: 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 just over half (61 percentage) of all the variance explained by the five components.

There are a number of ways in which we can select our final set of components. We could simply select all the components with an eigenvalue greater than one. In this case, we would keep the first four. We could use the proportion of total variance accounted for, which in the preceding table is the `cumulative var` row. A good rule of thumb is to select the components that account for at least 70 percent of the total variance, which means that the variance explained by each of the selected components accounts for 70 percent of the variance explained by all the components. In this case, three or four components would work. Perhaps, the best thing to do is apply judgment in conjunction with the other criteria. With this in mind, I would recommend going with just three components. It captures 71 percent of the total variance in the data and from my hockey fanatic point of view, is easier to interpret.

Another visual technique is to do a scree plot. 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. Therefore, I will present to you the much used scree plot as follows. We want to plot the associated eigenvalues from the `pca` object.

```
> plot(pca$values, type="b", ylab="Eigenvalues", xlab="Component")
```

The following is the output of the preceding command:



What you are looking in a scree plot is similar to what I described previously about the eigenvalues that are greater than one and the point where the 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, three components look pretty compelling.

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 just 3 components and orthogonal rotation, as follows:

```
> pca.rotate = principal(pca.df, nfactors=3, rotate = "varimax")

> pca.rotate
Principal Components Analysis
Call: principal(r = pca.df, nfactors = 3, rotate = "varimax")
Standardized loadings (pattern matrix) based upon correlation matrix
      PC1    PC3    PC2    h2    u2 com
gg     0.57   0.59   0.15  0.69  0.31 2.1
gag    -0.65  -0.49  -0.09  0.67  0.33 1.9
five   0.64   0.66  -0.01  0.85  0.15 2.0
PPP    0.09  -0.05   0.83  0.70  0.30 1.0
PKP    0.66   0.32  -0.06  0.55  0.45 1.5
shots  0.13   0.75   0.31  0.67  0.33 1.4
sag    -0.18  -0.70  -0.41  0.68  0.32 1.8
scl    0.87   0.21   0.11  0.81  0.19 1.2
trl    0.38   0.75  -0.12  0.73  0.27 1.5
lead1  0.85   0.19   0.26  0.83  0.17 1.3
lead2  0.84   0.13   0.13  0.75  0.25 1.1
wop    0.46   0.60  -0.15  0.60  0.40 2.0
wosp   0.76   0.44  -0.08  0.77  0.23 1.6
face   0.05   0.21   0.77  0.63  0.37 1.2

PC1    PC3    PC2
```

```
SS loadings      4.78 3.44 1.72
Proportion Var  0.34 0.25 0.12
Cumulative Var 0.34 0.59 0.71
Proportion Explained 0.48 0.35 0.17
Cumulative Proportion 0.48 0.83 1.00

Mean item complexity = 1.5
Test of the hypothesis that 3 components are sufficient.
```

```
The root mean square of the residuals (RMSR) is 0.08
with the empirical chi square 34.83 with prob < 0.97
```

```
Fit based upon off diagonal values = 0.97
```

You can see that the rotation has changed the loadings, but the cumulative variance has not changed. We can now really construct some meaning around the components.

If you examine PC1, you will see the high negative correlation with the goals allowed per game in conjunction with a high rate of killing penalties and a high win rate if scoring first. Additionally, this component captures the high loadings with winning after having a lead at the end of the first or second period. The 5-on-5 ratio and winning when outshot are the frosting on the cake here. What I think we have in this component is the essence of defensive hockey including the killing penalties. The teams with high factor scores on this component would appear to be able to get a lead on an opponent and then shut them down to gain a win.

PC3 appears to simply be a good offense. We will see a high loading for the goals scored per game, 5-on-5 scoring ratio, and shots per game. Additionally, look at how high the loading is for the winning after trailing at the end of the first period variable.

I think that PC2 is the easiest to identify. It is the high loading value for the power play success and winning faceoffs. When we did the correlation analysis during the data exploration, we saw that these two variables were moderately correlated with each other. Any hockey fan can tell you how important it is for a team on the power play to win the faceoff, control the puck, and run their set plays. This analysis seems to show that these two variables do indeed go hand in hand.

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 = pca.rotate$scores
```

Principal Components Analysis

```
> pca.scores = as.data.frame(pca.scores)

> pca.scores
   PC1        PC3        PC2
1 -1.33961165 -2.07275625 -2.38781073
2 -1.83244527 -1.12058364  0.77749711
3 -0.31094630 -1.62938873 -0.44133187
4 -1.02855311 -0.15704421  1.37199872
5 -2.68517373 -0.13231693 -0.27385509
6 -0.20853619  0.01524553 -0.46863716
7  0.10587046 -1.62940123 -0.42949901
8 -0.01324802  0.14959828  0.68875225
9  0.38541531 -0.59307204 -1.08937869
10 -0.15009774 -0.86256283  1.53513596
11  0.84099339  0.25043743 -0.52846404
12  0.24423775  0.36681320 -0.08995115
13 -0.68188634  1.09817128  0.53636881
14  1.18243869  0.34770042  0.35777454
15 -0.34720957  0.73064512 -0.53796910
16 -0.27227305 -0.09055833  1.34309546
17 -0.46350916  1.76830702 -0.94260357
18  1.06460458  0.24810967 -0.42634091
19  0.68720947 -1.39593941  0.35981827
20  0.72661065  0.31415750 -0.32705332
21  0.05533420  0.33660090  1.34849134
22 -0.23245895  0.77728066 -1.22841259
23  0.80098114  0.20634248 -0.93200321
24  1.22510788 -1.05511121  1.96984298
25  1.07648218 -0.54999695 -0.47064277
26  0.40911681  0.97327087  1.29704632
27 -1.53540552  2.44287010 -0.29119483
28  1.65378121  0.40998320 -1.20495883
29 -0.47171243  0.26403013  0.67952937
30  1.11488330  0.58916795 -0.19524429
```

We now have the scores for each factor for each team. These are simply the variables for each observation multiplied by the loadings and summed. Remember that we ordered the observations by total points from the worst to the best. As such, you can see that a team such as BUFFALO in the first row has negative factor scores across the board.

Before we begin the regression analysis, let's prepare the NHL data frame. The two major tasks are to convert the total points to numeric and attach the factor scores as new variables. We can accomplish this with the following code:

```
nhl$pts = as.numeric(nhl$pts)
nhl$Def = pca.scores$PC1
nhl$Off = pca.scores$PC3
nhl$PPlay = pca.scores$PC2
```

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 3 factors as inputs and then summarize the results:

```
> nhl.lm = lm(pts~Def+Off+PPlay, data=nhl)

> summary(nhl.lm)

Call:
lm(formula = pts ~ Def + Off + PPlay, data = nhl)

Residuals:
    Min      1Q  Median      3Q     Max 
-8.7891 -1.3948  0.5619  2.0406  6.6021 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  92.2000    0.7699 119.754 < 2e-16 ***
Def          11.2957    0.7831  14.425 6.41e-14 ***
Off          10.4439    0.7831  13.337 3.90e-13 ***
PPlay        0.6177    0.7831    0.789    0.437    
---

```

Principal Components Analysis

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

```
Residual standard error: 4.217 on 26 degrees of freedom
```

```
Multiple R-squared: 0.937, Adjusted R-squared: 0.9297
```

```
F-statistic: 128.9 on 3 and 26 DF, p-value: 1.004e-15
```

The good news is that our overall model is highly significant statistically, with p-value of 1.004e-15 and Adjusted R-squared is over 92 percent. The bad news is the power play factor is not with p-value of 0.437. We could simply choose to keep it in our model, but let's see what happens if we exclude it:

```
> nhl.lm2 = lm(pts~Def+Off, data=nhl)
> summary(nhl.lm2)
```

Call:

```
lm(formula = pts ~ Def + Off, data = nhl)
```

Residuals:

Min	1Q	Median	3Q	Max
-8.3786	-2.0258	0.7259	2.4748	6.3295

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	92.2000	0.7645	120.60	< 2e-16 ***
Def	11.2957	0.7776	14.53	2.79e-14 ***
Off	10.4439	0.7776	13.43	1.81e-13 ***

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

```
Residual standard error: 4.187 on 27 degrees of freedom
```

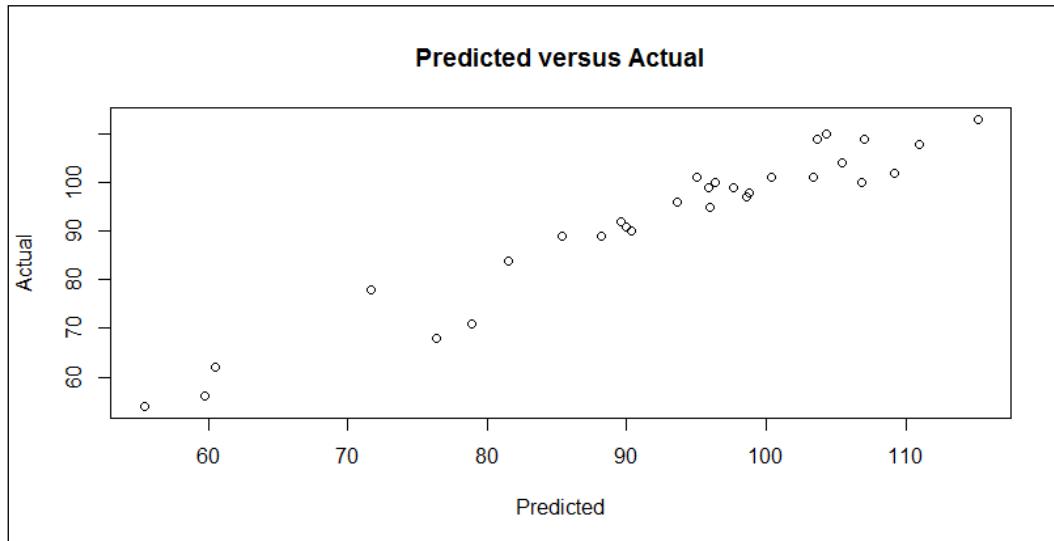
```
Multiple R-squared: 0.9355, Adjusted R-squared: 0.9307
```

```
F-statistic: 195.7 on 2 and 27 DF, p-value: < 2.2e-16
```

This model still achieves a high 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 (the total team points) with the base R graphics, as follows:

```
> plot(nhl.lm2$fitted.values, nhl$pts, 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 factors to predict the team's success and also highlights the strong linear relationship between the principal components and team points. Let's kick it up a notch by doing a scatterplot using the `ggplot2` package and include the team names in it. We will also focus only on the upper half of the team's point performance. We will first create a data frame of the 15 best teams in terms of points and call it `nhl.best`. Let's also include the predictions in the `nhl` data frame and sort it with the points in a descending order:

```
> nhl$pred = nhl.lm2$fitted.values  
  
> nhl=nhl[order(-nhl$pts),]  
  
> nhl.best = nhl[1:15,]
```

The next endeavor is to build our scatterplot using the `ggplot()` function. 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`, as follows:

```
> p = ggplot(nhl.best, aes(x=pred, y=pts, label=team))
```

Principal Components Analysis

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 now 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() +
```

Then, we will specify how we want our `team` labels to appear. This takes quite a bit of trial and error to get the font size and position in order:

```
geom_text(size=3.5, hjust=.2, vjust=-0.5, angle=15) +
```

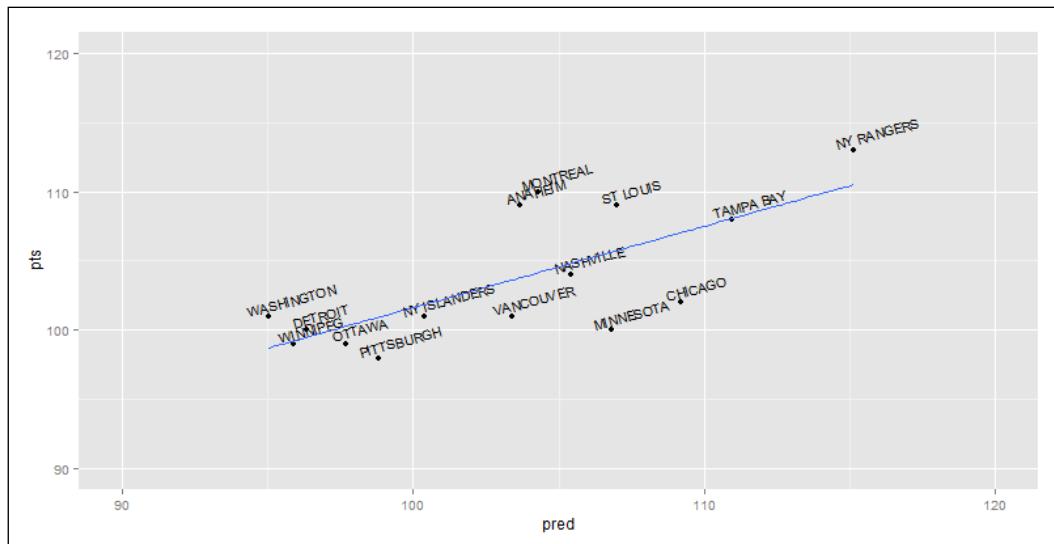
Now, specify the `x` and `y` axis limits, otherwise the plot will cut out any observations that fall outside them, as follows:

```
xlim(90,120) + ylim(90, 120) +
```

Finally, add a best fit line with no standard error shading and you get the following plot:

```
stat_smooth(method="lm", se=FALSE)
```

The following is the output of the preceding command:

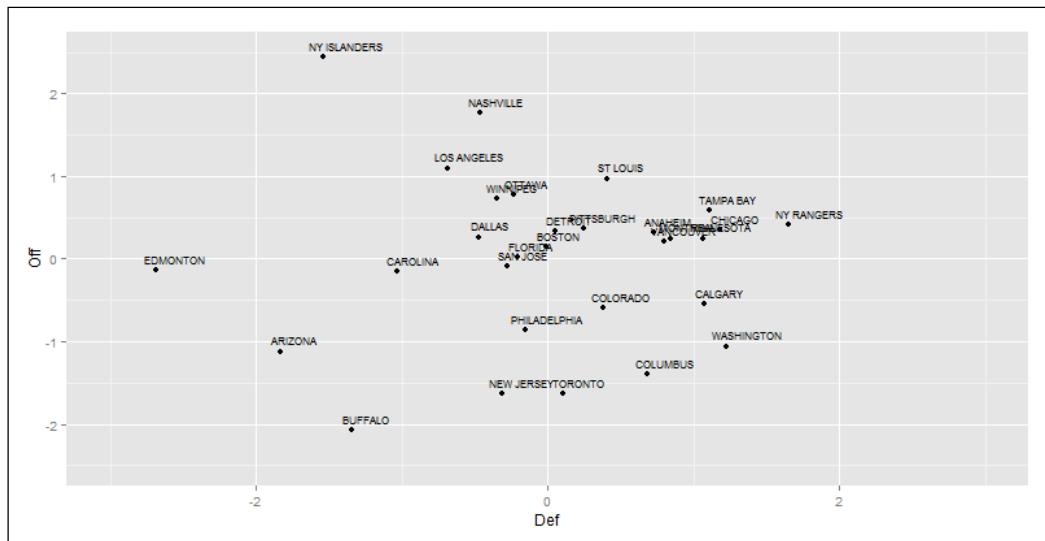


I guess one way to think about this plot is that the teams below the line underachieved, while those above it, overachieved. For instance, the **MINNESOTA** Wild had a predicted point total of 102 but only achieved 95. At any rate, this is overly simplistic but is fun to ponder nonetheless.

The final bit of analysis will be to plot the teams in relationship to their factor scores. Once again, `ggplot()` facilitates this analysis. Using the preceding code as our guide, let's update it and see what the result is:

```
> p2 = ggplot(nhl, aes(x=Def, y=Off, label=team)) +  
  
> p2 + geom_point() +  
  
geom_text(size=3, hjust=.2, vjust=-0.5, angle=0) +  
  
xlim(-3,3) + ylim(-3,3)
```

The output of the preceding command is as follows:



As you can see, the *x* axis is the defensive factor scores and the *y* axis is the offensive factor scores. Look at the **NY ISLANDERS** with the highest offensive factor scores but one of the worst defensive scores. I predict an early exit in the upcoming playoffs for them, considering the old adage that offense puts people in the seats but defense wins championships. Indeed, here is a link to an excellent article discussing the Islander's struggles and their poor ratio of winning when leading after the second period at <http://www.cbssports.com/nhl/eye-on-hockey/25134251/how-concerned-should-the-islanders-be-heading-into-the-playoffs>. Do the most balanced teams such as the **NY RANGERS**, **TAMPA BAY**, and others have the best chance to win a title? We shall see!

When I first wrote this chapter, the NHL playoffs were just starting. It was probably the best race for Lord Stanley's Cup that I have seen. Many of my fellow fans agree with this assertion. As predicted, the Islanders were out after the first round after a brutal series with the Washington Capitals as their defense lived up to its reputation, or lack thereof. The big surprise was how the Blackhawks refused to lose and ended up winning it all. Like a boxer with a puncher's chance, they got off the canvas and threw some well-timed haymakers. Note that I am a longtime Blackhawks fan going back to the days of Troy Murry and Eddie Belfour (all former Fighting Sioux). It was good for Jonathan Toews to win another title, especially after being robbed by the refs in the 2007 NCAA Frozen Four semifinal with T.J. Oshie as his linemate. However, I digress.

Summary

In this chapter, we took a second stab at the 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, especially when confronted with numerous highly correlated variables. Then, we applied it to a real and current dataset from the National Hockey League, using the resulting principal components in a regression analysis and exploring ways to visualize the data. As an unsupervised learning technique, it requires some judgment along with trial and error to arrive at an optimal solution. 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 and Recommendation Engines

"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.in, 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 your volume and the amount of 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.

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 and then dig into building a recommendation engine on website reviews.

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 the product placement, or develop marketing programs that take advantage of cross-selling. In short, the idea is to identify what items go well together, and profit from this.

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. (With all the travelling in my business, this is a never-ending annoyance for me.)

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 of Left-Hand Side (LHS) and y is the consequence of 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 is 10 percent and the probability of x is 20 percent and y 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 we will focus on it.

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 $k + (1\dots n)$, 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 it will be 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 the potential customers will purchase along with beer.

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)
> str(Groceries)
> Groceries
transactions in sparse format with
9835 transactions (rows) and
169 items (columns)
```

This dataset is structured as a sparse matrix object known as the class of transaction.

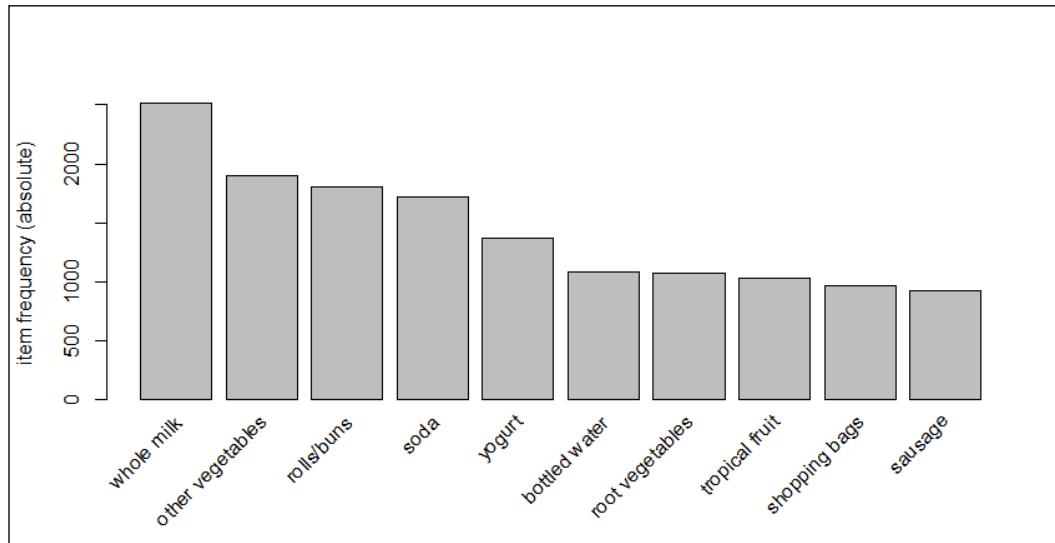
So, once the structure is that of the class transaction, our standard exploration techniques will not work, but the `arules` package offers us other techniques to explore the data. On a side note, if you have a data frame or matrix and want to convert it to the transaction class, you can do this with 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")
```

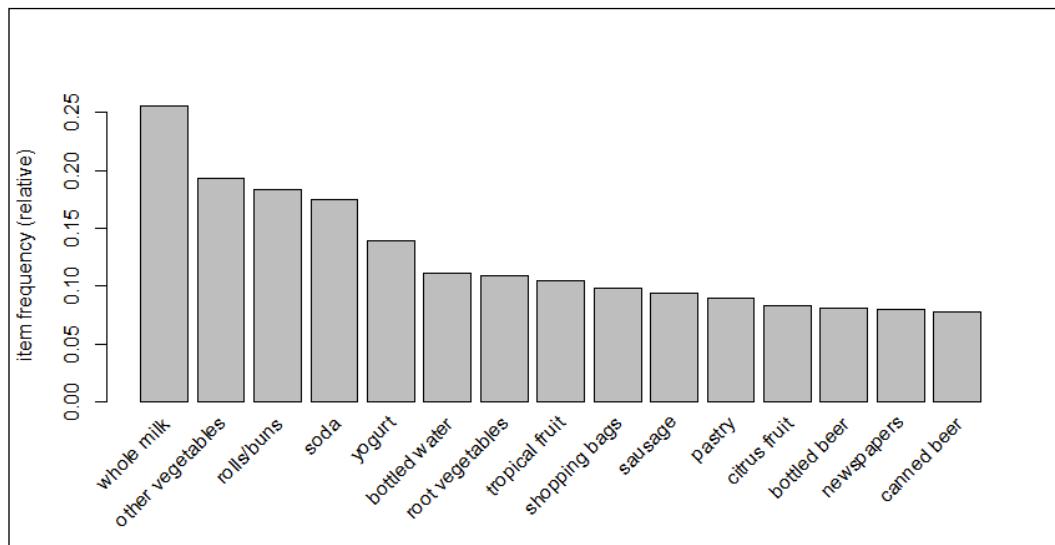
The output of the preceding command is as follows:



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 purposes of this exercise, this is all we really need to do, and 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 algorithm, which is the appropriately named `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 at specifying the minimum support and 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 up 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. 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 table with our dataset:

```
> table = crossTable(Groceries)
```

With table created, we can now examine the joint occurrences between the items. Here, we will look at just the first three rows and columns:

```
> table[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 and 10 times, they felt compelled to grab liver loaf. (Desperate times call for desperate measures!) If we 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 is between bottled beer and canned beer:

```
> 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 know I do.

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
```

So, we find ourselves with only 4 association rules. We saw one of them already; now let's have a look at 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
```

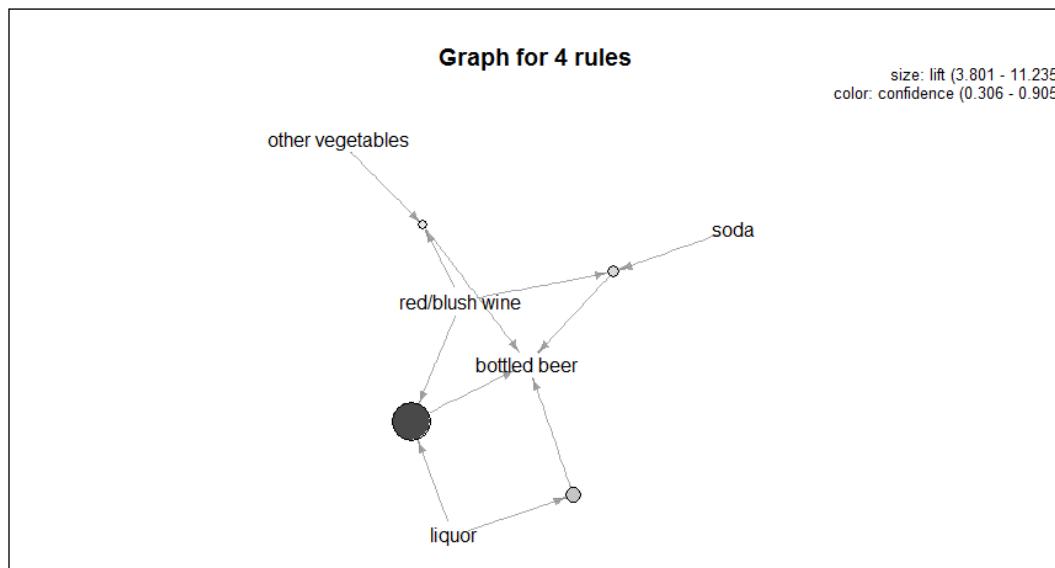
In all of the instances, the purchase of bottled beer is associated with booze, either liquor and/or red wine probably, which is no surprise to anyone. What is interesting is that white wine is not in the mix here. Let's take a closer look at this and compare the joint occurrences of bottled beer and types of wine:

```
> table["bottled beer", "red/blush wine"]  
[1] 48  
  
> table["red/blush wine", "red/blush wine"]  
[1] 189  
  
> 48/189  
[1] 0.25  
  
> table["white wine", "white wine"]  
[1] 187  
  
> table["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 the why in this analysis, but this could help us to potentially determine how we should position our product in this grocery store. One other 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 graph, showing lift 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 analysis is shown 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, incorporate customer segmentation, longitudinal purchase history, and so on as well as how to use it in ad displays, copromotions, and on and on. Now, let's move on to a situation where the customers are rating the items.

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, Essegaeier, 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, and 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: 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 can be predicted by first finding a neighborhood of similar users and then aggregate 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 some 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.

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** is 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 could 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 the 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 with 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. You should also be aware of the R package, `bcv`, Cross-Validation for the SVD. This package has the `impute.svd()` function, which will impute the missing values in a matrix.

So what is SVD? It is simply a method for matrix factorization. Say that you have a matrix called A. This matrix will factor into three matrices: U, D, and VT. U is an orthogonal matrix, D is a non-negative, diagonal matrix, and VT is a transpose of an orthogonal matrix. Furthermore, U is the eigenvector of AAT and V is the eigenvector of ATA. 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, 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", "Flanders", "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
Flanders     1             1            4            1
Me           1             5            2            4
```

Now, we will use the `svd()` function in base R to create the three matrices, 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:

```
> 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 can see the similarities with PCA. In fact, the two are closely related and often used interchangeably as they both utilize the matrix factorization. However, 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 on the centered data, diagonalize it, and create the components. Now, one can derive the principal components using SVD. 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 and 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.

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:

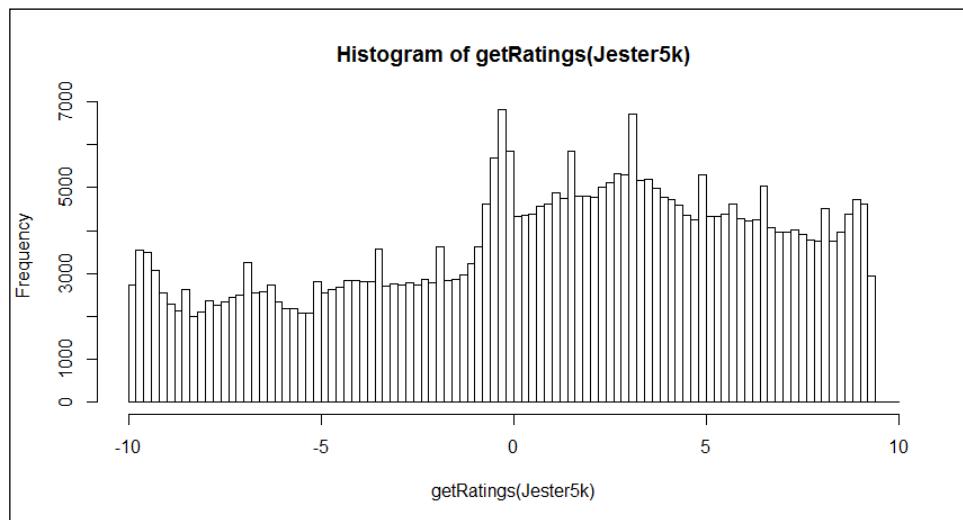
```
> rowMeans(Jester5k[10,])
u12843
-1.6

> 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)
```

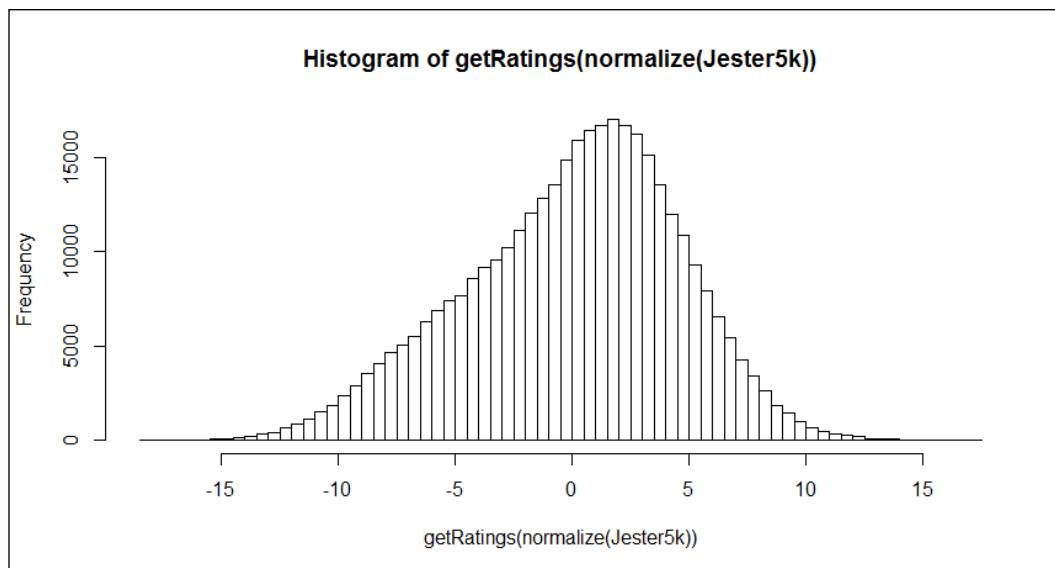
The output of the preceding command is as follows:



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 want. 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 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")

$IBCF_realRatingMatrix
Recommender method: IBCF
Description: Recommender based on item-based collaborative filtering
(real data).
Parameters:
k method normalize normalize_sim_matrix alpha na_as_zero minRating
1 30 Cosine      center          FALSE    0.5      FALSE      NA

$PCA_realRatingMatrix
Recommender method: PCA
Description: Recommender based on PCA approximation (real
data).
Parameters:
categories method normalize normalize_sim_matrix alpha na_as_zero
1           20 Cosine      center          FALSE    0.5      FALSE
```

```
minRating
1      NA

$POPULAR_realRatingMatrix
Recommender method: POPULAR
Description: Recommender based on item popularity (real
data).
Parameters: None

$RANDOM_realRatingMatrix
Recommender method: RANDOM
Description: Produce random recommendations (real ratings).
Parameters: None

$SVD_realRatingMatrix
Recommender method: SVD
Description: Recommender based on SVD approximation (real
data).
Parameters:
  categories method normalize normalize_sim_matrix alpha
  treat_na
  1      50 Cosine    center      FALSE   0.5   median
  minRating
  1      NA

$UBCF_realRatingMatrix
Recommender method: UBCF
Description: Recommender based on user-based collaborative filtering
(real data).
Parameters:
  method nn sample normalize minRating
  1 cosine 25 FALSE    center      NA
```

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. For instance, SVD treats the missing values as the column median. If you wanted to have the missing values coded as zero, you would need to include `param=list(treat_na="0")`:

```
> 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")

> pca_pred = predict(pca, 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 six 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(pca_pred, getData(e, "unknown"))

> P6 = calcPredictionAccuracy(rand_pred, getData(e, "unknown"))

> error = rbind(P1,P2,P3,P4,P5,P6)

> rownames(error) = c("UBCF", "IBCF", "SVD", "Popular", "PCA", "Random")

> error
      RMSE      MSE      MAE
UBCF 4.467276 19.95655 3.496973
IBCF 4.651552 21.63693 3.517007
SVD  5.275496 27.83086 4.454406
Popular 5.064004 25.64414 4.233115
PCA   4.711496 22.19819 3.725162
Random 7.830454 61.31601 6.403661
```

We can see in the output that the user-based algorithm slightly outperforms IBCF and PCA. It is also noteworthy that a simple algorithm such as the popular-based recommendation does fairly well.

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 IBCF algorithms performed the best, we will look at them along with the popular-based one.

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"
```

You can adjust the parameters with `param=...` in the `list()` function just as the preceding example. In the next step, you can create the results using `evaluate()` and also set up a comparison on a specified number of recommendations. 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 [0.05sec/1.02sec]

UBCF run
1 [0.03sec/68.26sec]

IBCF run
1 [2.03sec/0.86sec] 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:

```
> avg(evlist)

$POPULAR
    TP      FP      FN      TN precision      recall      TPR
5  2.092  2.908 14.193 70.807      0.4184 0.1686951 0.1686951
10 3.985  6.015 12.300 67.700      0.3985 0.2996328 0.2996328
15 5.637  9.363 10.648 64.352      0.3758 0.4111718 0.4111718

      FPR
5  0.03759113
10 0.07769088
15 0.12116708

$UBCF
    TP      FP      FN      TN precision      recall      TPR
5  2.074  2.926 14.211 70.789      0.4148 0.1604751 0.1604751
10 3.901  6.099 12.384 67.616      0.3901 0.2945067 0.2945067
15 5.472  9.528 10.813 64.187      0.3648 0.3961279 0.3961279

      FPR
5  0.03762910
10 0.07891524
15 0.12362834

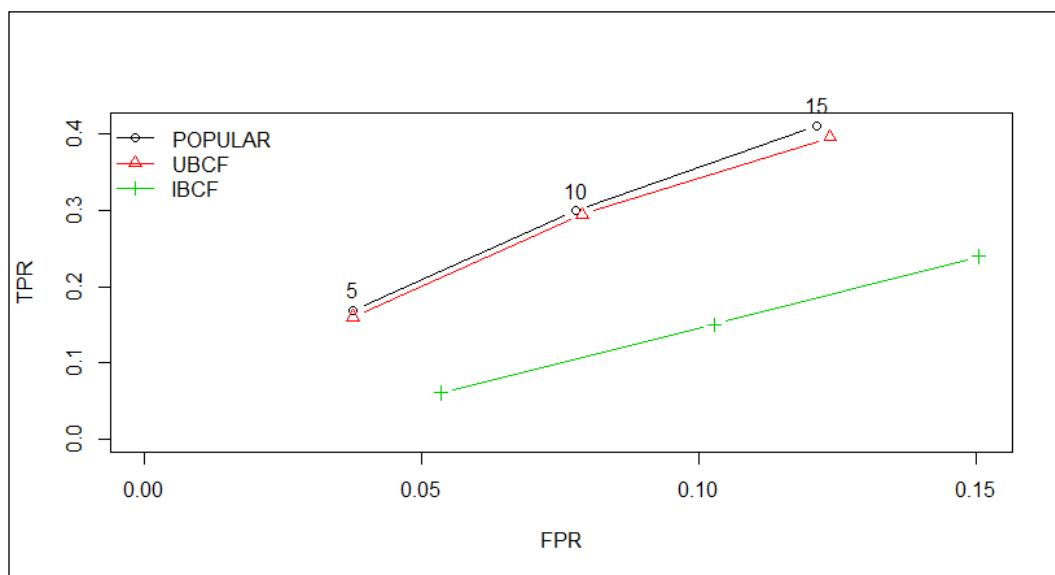
$IBCF
    TP      FP      FN      TN precision      recall
5  1.010  3.990 15.275 69.725      0.2020 0.06047142
10 2.287  7.713 13.998 66.002      0.2287 0.15021068
15 3.666 11.334 12.619 62.381      0.2444 0.23966150

      TPR      FPR
5  0.06047142 0.0534247
10 0.15021068 0.1027532
15 0.23966150 0.1504704
```

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. Indeed, what is disappointing about this whole exercise is the anemic TPR, for example, for `UBCF` of 15 recommendations, only an average of 5.5 were truly accurate. As mentioned, we can plot and compare the results as **Receiver Operating Characteristic Curves (ROC)**, where you can compare TPR and FPR or as precision/recall curves, as follows:

```
> plot(evlist, legend="topleft", annotate=TRUE)
```

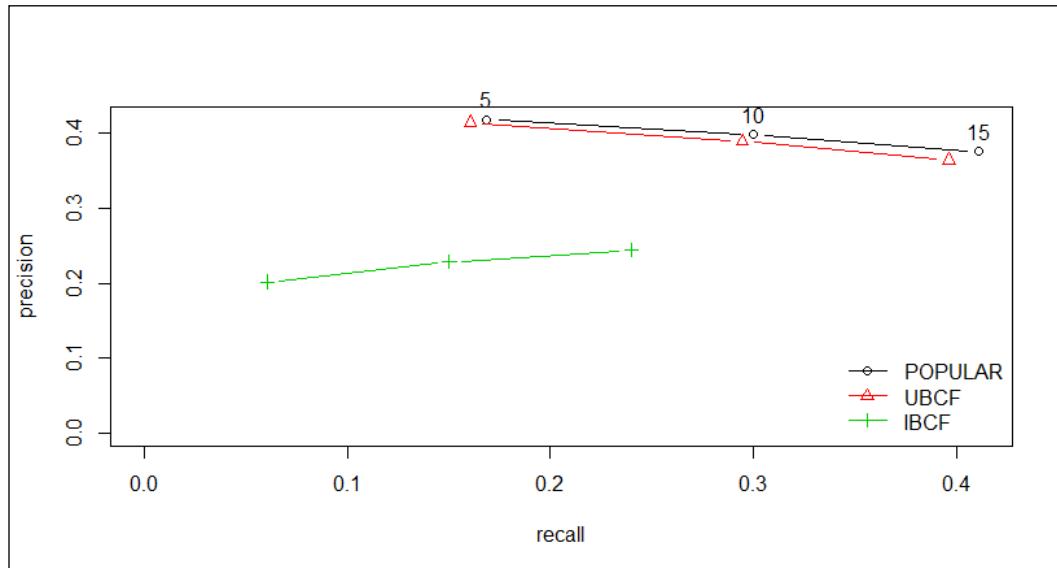
The following is the output of the preceding command:



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 user-based 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="UBCF")  
  
> R1  
Recommender of type 'UBCF' for 'realRatingMatrix'  
learned using 5000 users.
```

Now, we just need to get the top five recommendations—in order—for the first two raters and produce them as a list:

```
> recommend = predict(R1, Jester5k[1:2], n=5)  
  
> as(recommend, "list")
```

```
[[1]]
[1] "j81" "j78" "j83" "j80" "j73"
```

```
[[2]]
[1] "j96" "j87" "j89" "j76" "j93"
```

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
[1,] -0.8055227 -0.05159179 -0.3244485
[2,]       NA          NA          NA
[3,] -1.2472200          NA -1.5193913
[4,]  4.0659217  4.45316186  4.0651614
[5,]       NA          NA          NA
[6,]  1.1233854  1.37527380          NA
[7,]  0.4938482  0.18357168 -0.1378054
[8,]  0.2004399  0.58525761  0.2910901
[9,] -0.5184774  0.03067017  0.2209107
[10,]  0.1480202  0.35858842         NA
```

The numbers in the matrix indicate the predicted ratings 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 `given=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]

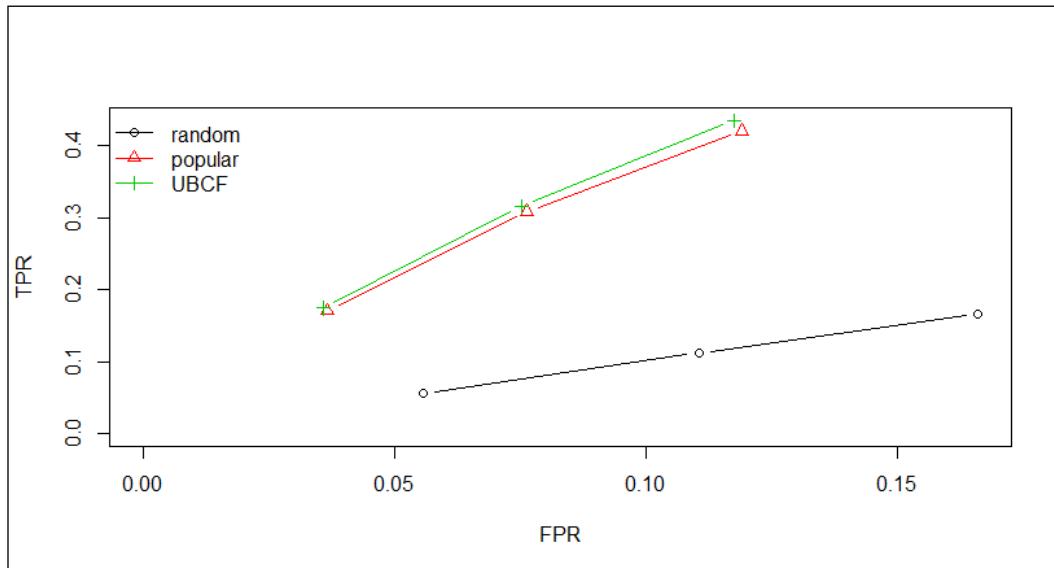
POPULAR run
1 [0.01sec/3.79sec]
2 [0sec/3.81sec]
3 [0sec/3.82sec]
4 [0sec/3.92sec]
5 [0.02sec/3.78sec]
```

```
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:

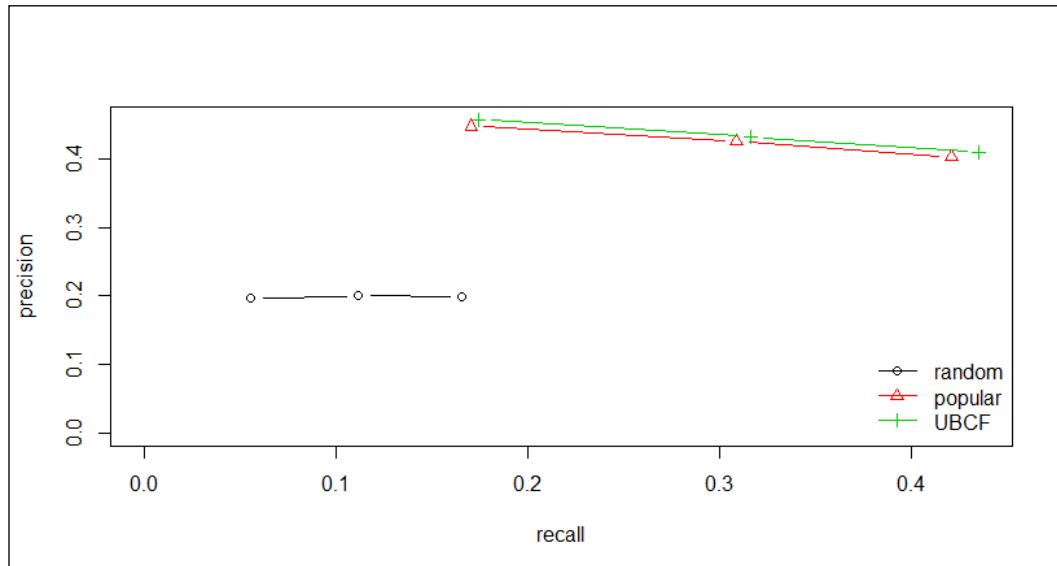
```
> plot(results.bin, legend="topleft")
```

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.

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 rather to develop and test algorithms.

11

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 collected 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 the 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 indeed exists.

In this chapter, we will apply time series/econometric techniques to identify univariate forecast models, bivariate regression 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.

Exponential smoothing models use weights for past observations, such as a moving average model, 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:

$$Y_t + 1 = \alpha(Y_t) + (1 - \alpha)Y_{t-1} + (1 - \alpha)^2 Y_{t-2} + \dots, \text{ where } 0 < \alpha \leq 1$$

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, $Y_t + h = A + hB_t + St$
- The trend, where $B_t = B_t = \beta(A_t - A_{t-1}) + (1 - \beta)B_{t-1}$
- The seasonality, where m is the number of seasonal periods,
$$St = \Omega(Y_t - A_t - B_t - 1) + (1 - \Omega)St - m$$

This equation is referred to as the **Holt-Winter's 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. It has been my experience that 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 the 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 $Y_t = \text{constant} + \Phi Y_{t-1} + E_t$. The critical assumptions for the model are as follows:

- E_t is the errors that are identically and independently distributed with a mean zero and constant variance
- The errors are independent of Y_t
- $Y_t, Y_{t-1}, Y_{t-n} \dots$ is stationary, which means that the absolute value of Φ is less than one

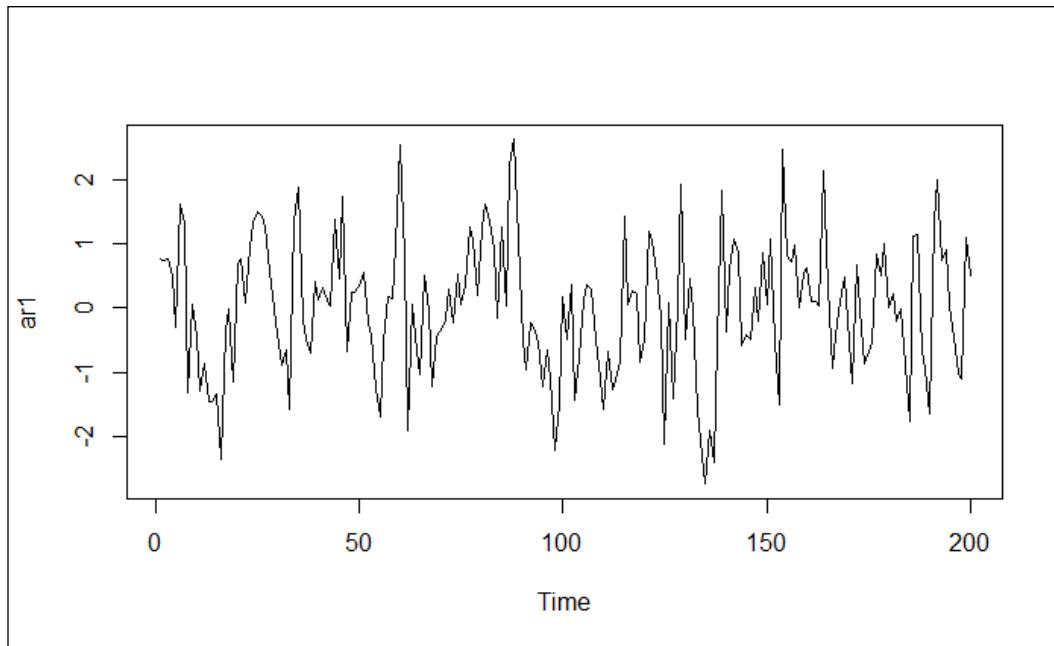
With a stationary time series, you can examine **Autocorrelation Function (ACF)**. The ACF of a stationary series gives correlations between Y_t and Y_{t-h} for $h = 1, 2 \dots n$. Let's use R to create an AR(1) series and plot it:

```
> set.seed(123)

> ar1 = arima.sim(list(order=c(1,0,0), ar=0.5), n=200)

> plot(ar1)
```

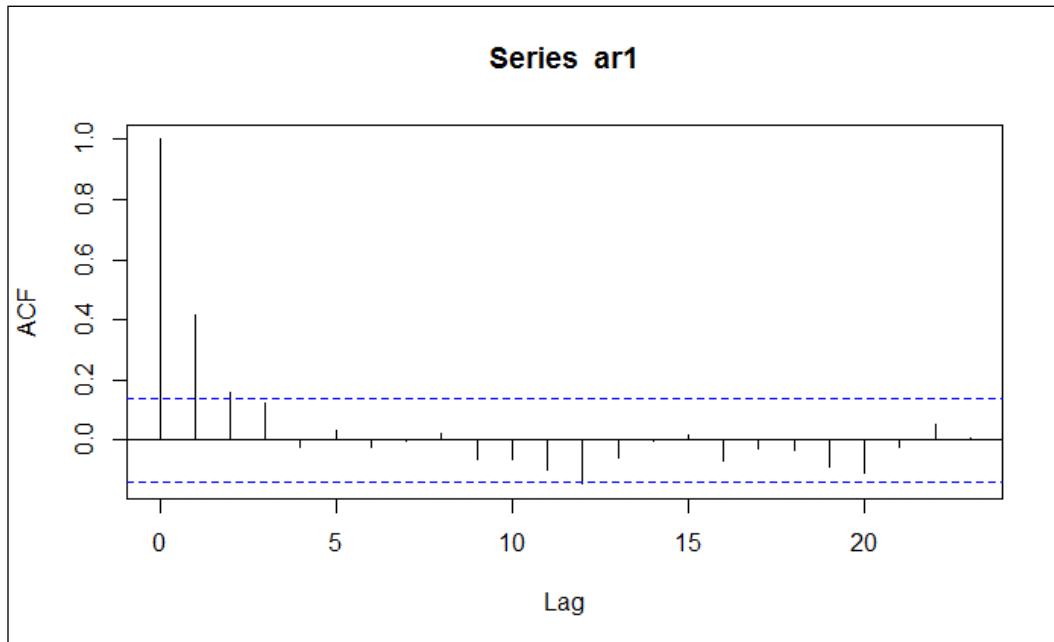
The following is the output of the preceding command:



Now, let's examine ACF:

```
> acf(ar1)
```

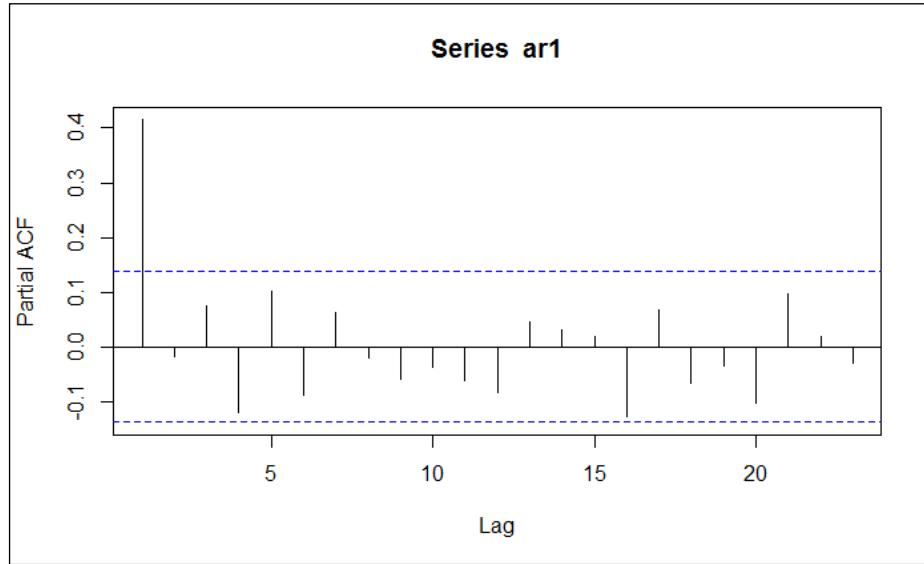
The output of the preceding command is as follows:



The ACF plot shows the correlations exponentially decreasing as the **Lag** increases. The dotted blue lines indicate a significant correlation. 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.

```
> pacf(ar1)
```

The following is the output of the preceding command:



We will assume that the series is stationary and the preceding time series plot confirms this. We'll look at a couple of statistical tests in the practical exercise to ensure that the data is stationary, but most of the times, the eyeball test is sufficient. If the data is not stationary, then it is possible to de-trend 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).

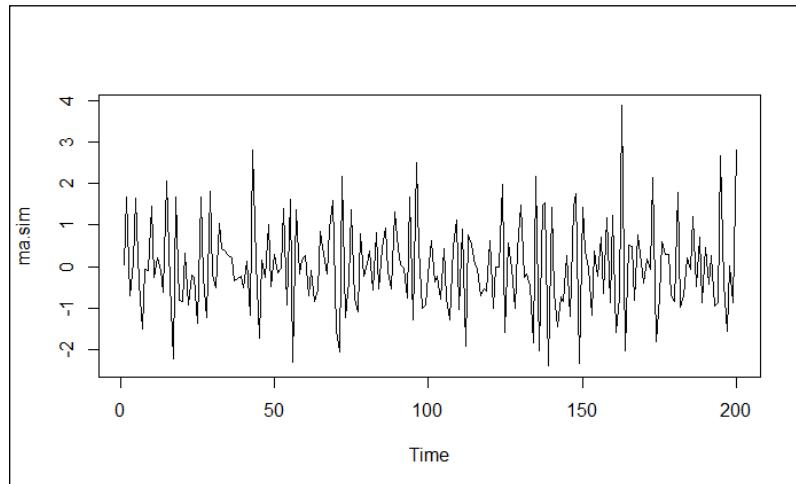
The MA stands for moving average. This is not the simple moving average as the 50-day moving average of a stock price but rather, a coefficient that is applied to the errors. The errors are, of course, identically and independently distributed with a mean zero and constant variance. The formula for an MA(1) model is $Y_t = \text{constant} + E_t + \Theta E_{t-1}$. As we did with the AR(1) model, we can build an MA(1) in R, as follows:

```
> set.seed(123)

> ma.sim = arima.sim(list(order=c(0,0,1), ma=-0.5), n=200)

> plot(ma.sim)
```

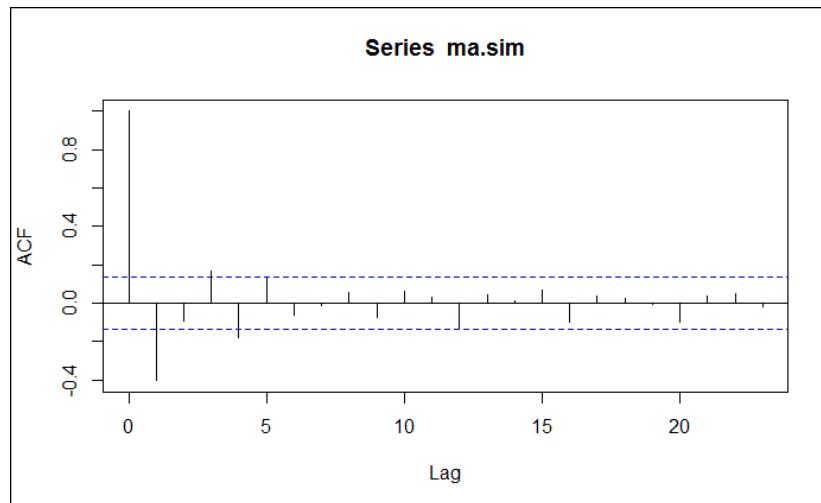
The following is the output of the preceding command:



The ACF and PACF plots are a bit different from the AR(1) model. Note that there are some rules of thumb in looking at the plots in order to determine if 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:

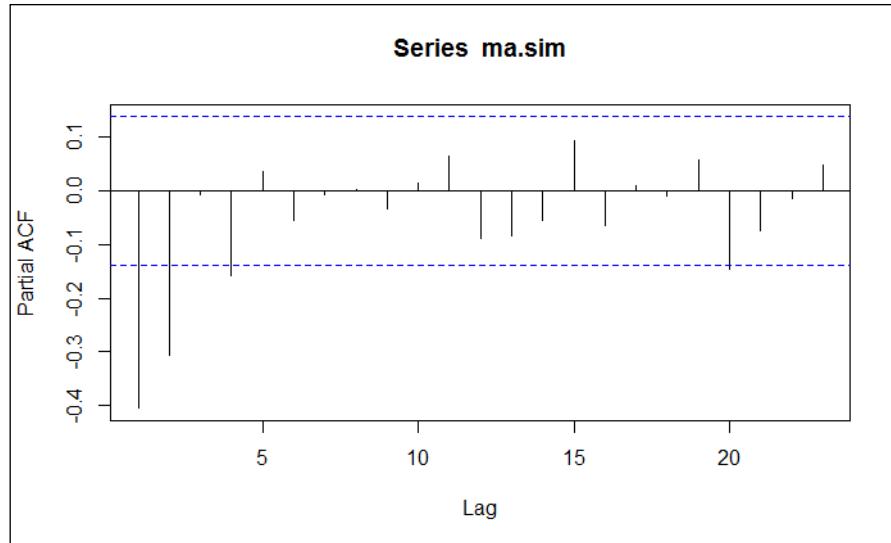
```
> acf(ma.sim)
```

The output of the preceding command is as follows:



The preceding figure is the ACF plot, and now, we will see the PACF plot:

```
> pacf(ma.sim)
```



With 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 $(p,d,q) \times (P,D,Q)_{12}$. In the packages that we will use, R will automatically identify if the seasonality should be included and if so, then the optimal terms will be included as well.

Bivariate regression

Having covered regression way back in *Chapter 2, Linear Regression – The Blocking and Tackling of Machine Learning*, we can skip many of the basics. However, when doing regression with time series, it is important to understand how the regression may be spurious and/or missing vital information. With time series regression, it is necessary to understand how the lagged variables can contribute to the model. Let's take advertising expenditure and product sales with the data available on a weekly basis. In many cases, you will need to model the lagged values of an advertising campaign in the prediction of sales, as it may take time for the campaign to be effective and the impact can last beyond its termination.

In R, you can manually code the lagged values, trends, and seasonality; however, the `dynlm` package for dynamic linear regression offers tremendous flexibility and ease in doing this type of analysis. In the practical exercise, we will put the package through its paces. To examine the problem of spurious regression, we will need to test the assumption of no serial correlation, which is the autocorrelation of the residuals. Examining the ACF plot and statistical tests will address the question. If autocorrelation exists, then you might run into the problem of spurious regression. Here, the beta coefficients are not the best estimates as important information is being ignored and the statistical tests on these coefficients are no longer valid, which means that you may overfit your model as some predictor variables will appear important when actually they are not important.

One method to deal with serial correlation is to include the ARIMA errors with regression models. A simplified notation for this type of model is $Y_t = \beta_0 + \beta_1 t + N_t + E_t$, where N_t is an ARIMA model for the errors and E_t is the remaining errors of the ARIMA model that are not correlated and are referred to as white noise.

This type of regression can be implemented in R using functions in the `forecast` or `orcutt` packages. The interpretation of these methods can get quite complicated and challenging to explain to business partners. Not to fear; because in general, if you find the right lag structure, you can forgo incorporating the ARIMA errors in your regression model. We will explore this in detail in the practical exercise.

Granger causality

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:

- $\Omega = y_t = \beta_0 + b_1 y_{t-1} + \dots + \beta_k y_{t-k} + e$
- $\pi = y_t = \beta_0 + \beta_1 y_{t-1} + \dots + \beta_k y_{t-k} + \alpha_1 y_{t-1} + \dots + \alpha_k y_{t-k} + e$

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 if 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 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 for non-stationary series and also nonlinear models, but this is outside of the scope for this book. 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 are appropriate.

- $$Y = \text{constant}_1 + \beta_{11}Y_{t-1} + \beta_{12}X_{t-1} + e_1$$
- $$Y = \text{constant}_1 + \beta_{21}Y_{t-1} + \beta_{22}X_{t-1} + e_2$$

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 always will, but the big question—at least from a political and economic standpoint—is that is the climate change man-made? Even Pope Francis and the Vatican have weighed in on the controversy, casting aspersions on man-made climate change deniers. Not one to shy away from a political donnybrook, I will use this chapter to put econometric time series modeling to the test to try and determine if man-made carbon emissions cause—statistically speaking—climate change, 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, we should choose the **HadCRUT4** annual time series. 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 year 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. Recently, 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 utilize this data as it is.

To read this data in R, which is in a fixed-width format, we will use the `read.fwf()` function. The first thing to do that helps in the web-scraping process is to specify an object with the appropriate URL as follows:

```
> url1 = "http://www.metoffice.gov.uk/hadobs/hadcrut4/data/current/time_series/HadCRUT.4.4.0.0.annual_ns_avg.txt"
```

From this URL, we only want the year and first column, which is the annual anomaly. To do this, we will specify in the function the width of the items that we want. The simplest way to know these widths is to take a few rows of the data, paste it in a text editor, and do some counting. Having done this, we will put in `widths` as 4 for the year, 3 for the space, and 6 for the anomaly:

```
> temp = read.fwf(url1, widths=c(4,3,6), sep="")
```

```
> head(temp)
      V1      V2
1 1850 -0.376
2 1851 -0.222
3 1852 -0.225
4 1853 -0.270
5 1854 -0.247
6 1855 -0.270
```

```
> tail(temp)
      V1      V2
161 2010 0.555
162 2011 0.421
163 2012 0.467
164 2013 0.492
165 2014 0.564
166 2015 0.670
```

With this data frame created, we can name the columns properly:

```
> names(temp)=c("Year", "Temperature")
```

Furthermore, this data needs to be converted to a time series object with the `ts()` function. We will also need to specify the frequency of the data, for example, 1 is annual, 4 is quarterly, 12 is monthly, and the start and end year:

```
> T = ts(temp$Temperature, frequency=1, start=c(1850), end=c(2015))
```

Global CO emission estimates can be found at the **Carbon Dioxide Information Analysis Center (CDIAC)** of the US Department of Energy at <http://cdiac.ornl.gov/>. We will download the data of total emissions of fossil fuel combustion and cement manufacture. We could use the `read.fwf()` function for this data as well, but let's look at a different method, where we will put the URL in a `read.csv()` function:

```
> url2 = "http://cdiac.ornl.gov/ftp/ndp030/CSV-FILES/global.1751_2011.csv"  
  
> co2 = read.csv(file=url2,skip=2,header=FALSE, col.names=c("Year","Total","3","4","5","6","7","8"))
```

What we did here was specify that we wanted to skip the first two rows of the commentary. By stating `header=FALSE`, we will prevent the first row of data from becoming our column names, which we will create with `col.names()`.

Looking at the structure, we want to keep only the first two columns (the Year and co2 emissions). Putting this together, we can then look at the first six and last six observations as a double check:

```
> str(co2)  
  
'data.frame': 261 obs. of  8 variables:  
 $ Year : int  1751 1752 1753 1754 1755 1756 1757 1758 1759 1760 ...  
 $ Total: int  3 3 3 3 3 3 3 3 3 3 ...  
 $ X3   : int  0 0 0 0 0 0 0 0 0 0 ...  
 $ X4   : int  0 0 0 0 0 0 0 0 0 0 ...  
 $ X5   : int  3 3 3 3 3 3 3 3 3 3 ...  
 $ X6   : int  0 0 0 0 0 0 0 0 0 0 ...  
 $ X7   : int  0 0 0 0 0 0 0 0 0 0 ...  
 $ X8   : num  NA ...  
  
> co2 = co2[,1:2]  
  
> head(co2)  
    Year Total  
1 1751     3  
2 1752     3  
3 1753     3  
4 1754     3  
5 1755     3
```

```
6 1756      3
> tail(co2)
  Year Total
256 2006  8363
257 2007  8532
258 2008  8740
259 2009  8700
260 2010  9140
261 2011  9449
```

Finally, we will put this in a time series:

```
> E = ts(co2$Total, frequency=1, start=c(1751), end=c(2011))
```

With our data downloaded and put in time series structures, we can now begin to understand and further prepare it, prior to analysis.

Data understanding and preparation

Only three packages are required for this effort, so ensure that they are installed on your system. Now, let's get them loaded:

```
> library(dynlm)

> library(forecast)

> library(tseries)

> library(vars)
```

As always, we will want to produce plots of the data and will do so in a reasonable timeframe. Let's look at the data starting shortly after the end of World War I. The Co2 emissions data ends in 2011, so we will truncate the temperature data to match:

```
> SurfaceTemp = window(T, start=c(1920), end=c(2011))

> Emissions = window(E, start=c(1920), end=c(2011))
```

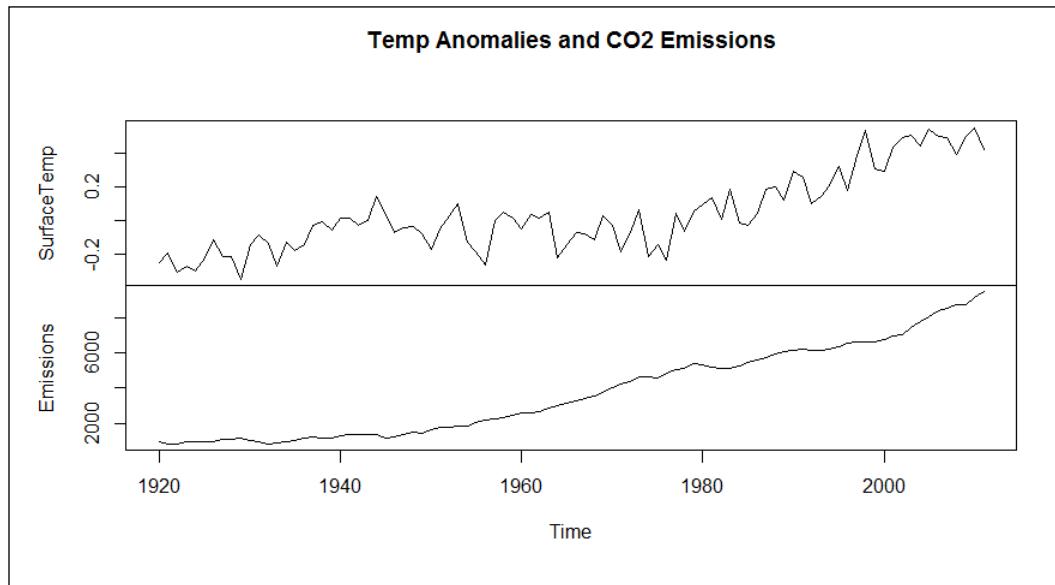
Time Series and Causality

Now, combine both the time series to one object and plot it:

```
> climate = cbind(SurfaceTemp, Emissions)

> plot(climate, main="Temp Anomalies and CO2 Emissions")
```

The output of the preceding command is as follows:



This plot shows that the temperature anomalies started to increase from the norm roughly around 1970. **Emissions** seem to begin a slow uptick in the mid-40s and a possible trend increase in 2000. There does not appear to be any cyclical patterns or obvious outliers. Variation over time appears constant. Using the standard procedure, we can see that the two series are highly correlated, as follows:

```
> cor(climate)
      SurfaceTemp  Emissions
SurfaceTemp  1.0000000  0.8264994
Emissions    0.8264994  1.0000000
```

As discussed earlier, this is nothing to jump for joy as it proves absolutely nothing. We will look for the structure by plotting ACF and PACF for both the series on the same plot. You can partition your plot using the `par()` function and specifying the rows and columns; in our case, a 2×2 , as follows:

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

Calling ACF and PACF for each series will automatically populate the plot area:

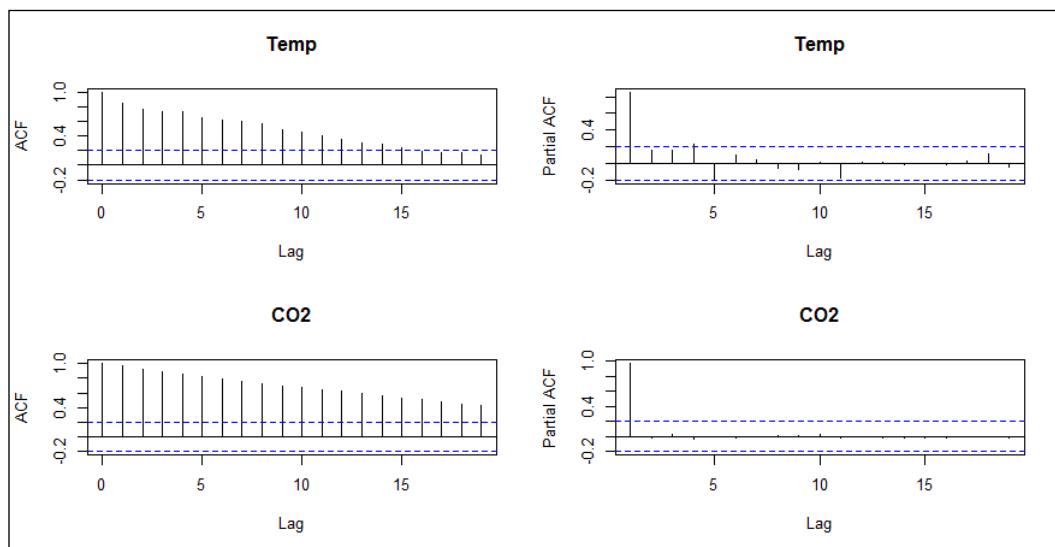
```
> acf(climate[,1], main="Temp")
```

```
> pacf(climate[,1], main="Temp")
```

```
> acf(climate[,2], main="CO2")
```

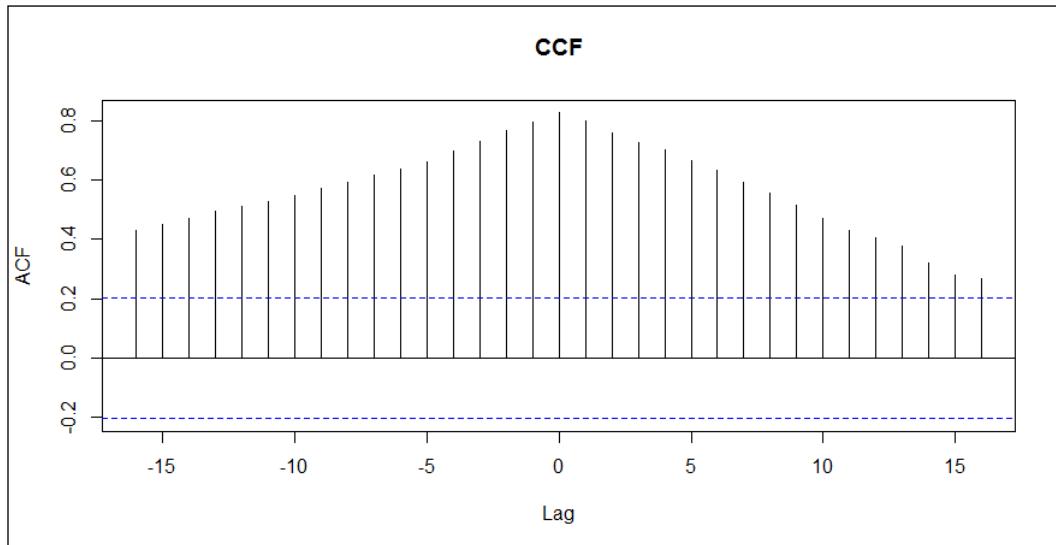
```
> pacf(climate[,2], main="CO2")
```

The output of the preceding code snippet is as follows:



With the decaying ACF patterns and rapidly decaying PACF patterns, we can assume that these series are both autoregressive. For temperature, there is a slight spike at lag-4 and lag-5, so there may be an MA term as well. 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. For instance, the correlation between the temperature and CO₂ at lag-5 is just over **0.6**. 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 for the latter. We will see during the VAR and Granger causality analysis if this will matter or not.

Additionally, to a calibrated eye, the data is not 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 = -1.8429, Lag order = 4, p-value = 0.641
alternative hypothesis: stationary
```

```
> adf.test(climate[,2])
```

```
Augmented Dickey-Fuller Test
```

```
data: climate[, 2]
Dickey-Fuller = -1.0424, Lag order = 4, p-value = 0.9269
alternative hypothesis: stationary
```

You can see that, in both the cases, the p-values are not significant, so we fail to reject the null hypothesis of the test that the data is not stationary.

Having explored the data, let's begin the modeling process by applying univariate techniques to the temperature anomalies.

Modeling and evaluation

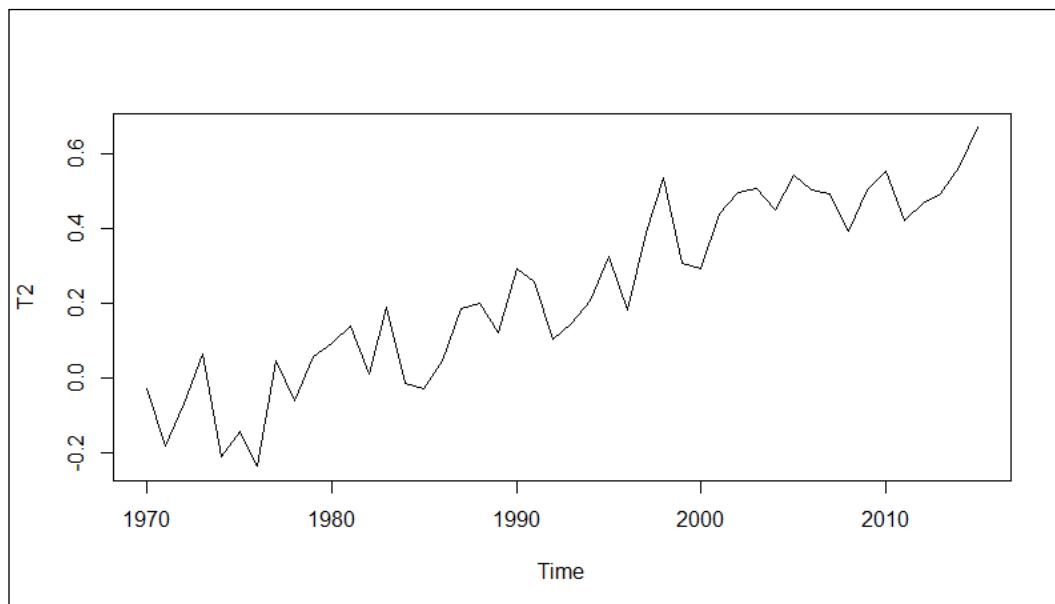
For the modeling and evaluation step, we will focus on three tasks. The first is to produce a univariate forecast model applied to just the surface temperature. The second is developing a regression model of the surface temperature based on itself and carbon emissions. Finally, we will try and discover if emissions 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. As discussed previously, the temperature anomalies start to increase around 1970. Therefore, I recommend looking at it from this point to the present. The following code creates the subset and plots the series:

```
> T2 = window(T, start=1970)
```

```
> plot(T2)
```



Our train and test sets will be through 2007, giving us eight years of data to evaluate for the selection. Once again, the `window()` function allows us to do this in a simple fashion, as follows:

```
> train = window(T2, end=2007)
```

```
> test = window(T2, start=2008)
```

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=...`, and method to select the initial state values, either "optimal" or "simple", and if we want a damped trend, then `damped=TRUE`. Optimal finds the values along with the smoothing parameters, while simple uses 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. Now, moving on to the `holt` model without a damped trend, as follows:

```
> fit.holt=holt(train, h=8, initial="optimal")

> summary(fit.holt)

Forecast method: Holt's method

Model Information:
ETS (A,A,N)

Call:
holt(x = train, h = 8, initial = "optimal")

Smoothing parameters:
alpha = 0.0271
beta = 0.0271

Initial states:
l = -0.1464
b = 0.0109

sigma: 0.0958

      AIC      AICc      BIC
-32.03529 -30.82317 -25.48495

Error measures:
MAPE
```

87.56256

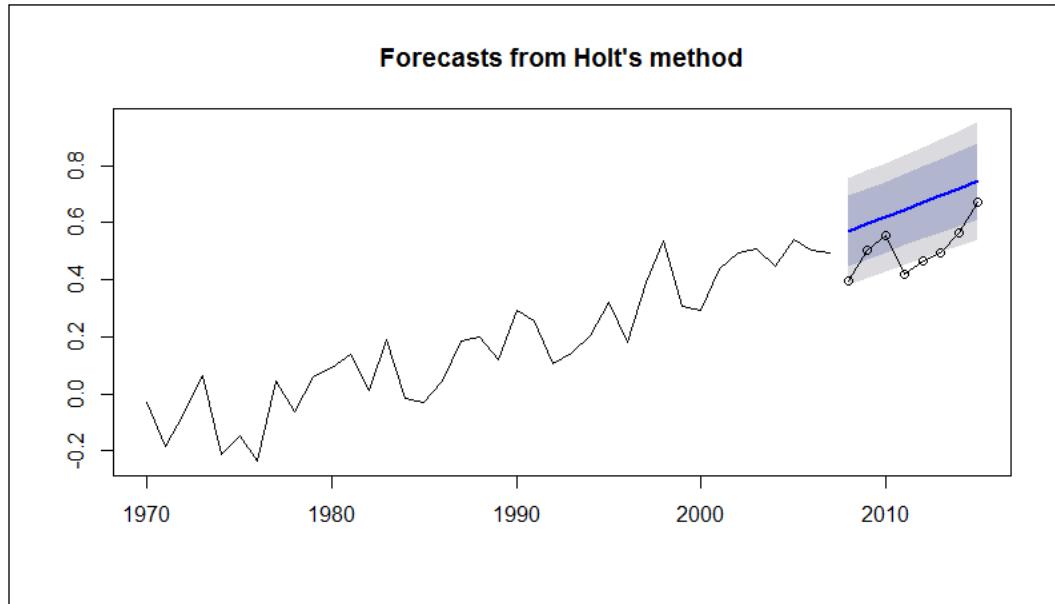
Forecasts:

	Point Forecast
2008	0.5701693
2009	0.5951016
2010	0.6200340
2011	0.6449664
2012	0.6698988
2013	0.6948311
2014	0.7197635
2015	0.7446959

This is quite a bit of output, and for brevity, I've even eliminated all the error measures other than MAPE and deleted the 80 and 95 percent confidence intervals. You can see these along with the parameters and Initial states. We can also plot forecast and see how well it did, out-of-sample:

```
> plot(forecast(fit.holt))
```

```
> lines(test, type="o")
```



Looking at the plot, it seems that this forecast overshot the mark a little bit. Let's have a go by including the damped trend, as follows:

```
> fit.holtd=holt(train, h=8, initial="optimal", damped=TRUE)

> summary(fit.holtd)

Forecast method: Damped Holt's method

Model Information:
ETS (A,Ad,N)

Call:
holt(x = train, h = 8, damped = TRUE, initial = "optimal")

Smoothing parameters:
alpha = 1e-04
beta  = 1e-04
phi   = 0.98

Initial states:
l = -0.2277
b = 0.0266

sigma: 0.0986

      AIC      AICc      BIC
-27.86479 -25.98979 -19.67686

      MAPE
Training set 120.6198

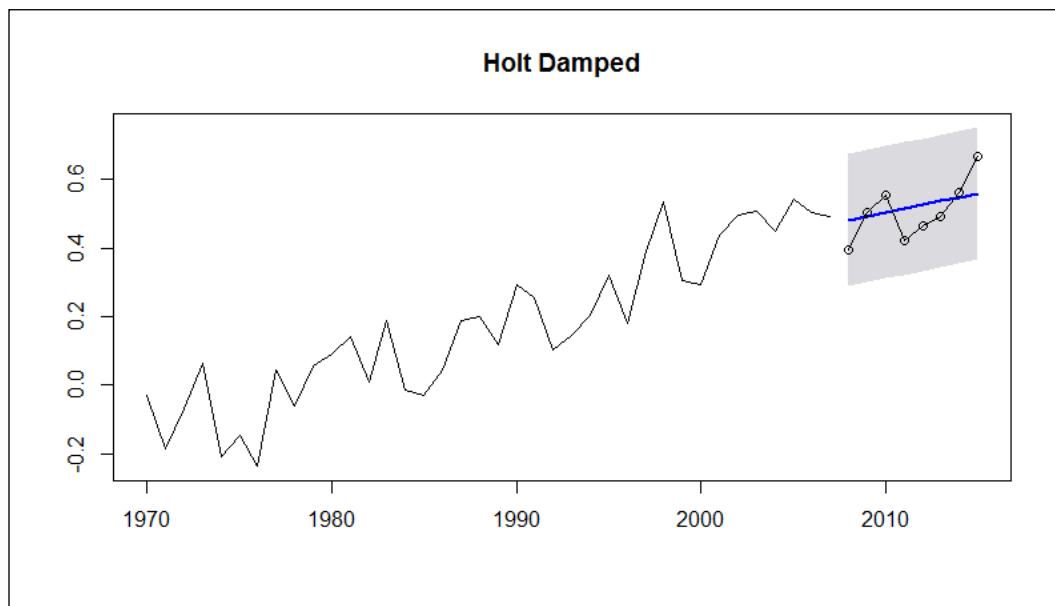
Forecasts:
      Point Forecast
2008    0.4812987
2009    0.4931311
2010    0.5047266
```

2011	0.5160901
2012	0.5272261
2013	0.5381393
2014	0.5488340
2015	0.5593147

Notice in the output that it now includes the phi parameter for the trend dampening. Additionally, you can see that the point forecasts are lower in the damped method but MAPE is higher. Let's examine again how it performs out-of-sample, as follows:

```
> plot(forecast(fit.holtd), "Holt Damped")  
  
> lines(test, type="o")
```

The following is the output of the preceding command:



Looking at the plot, you can see that the damped method performed better on the test set. Finally, for the ARIMA models, you can use `auto.arima()` again 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(2,1,0)

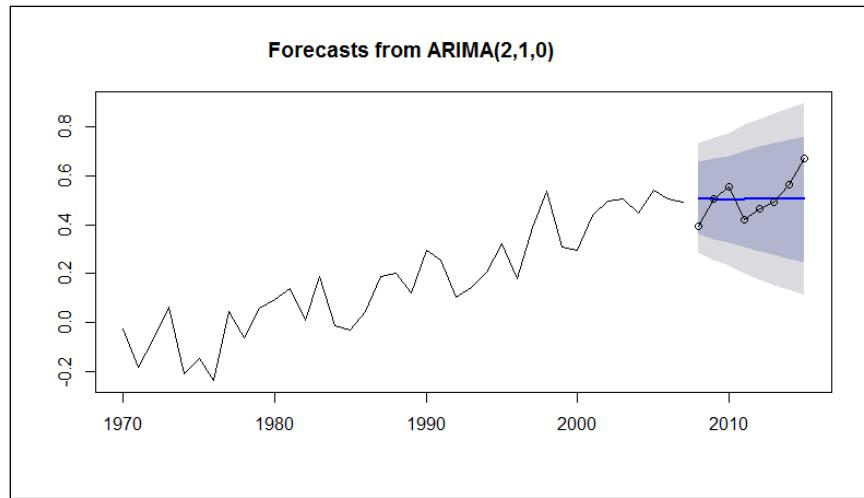
Coefficients:
          ar1      ar2
-0.5004 -0.2947
s.e.    0.1570  0.1556

sigma^2 estimated as 0.01301:  log likelihood=27.65
AIC=-49.3    AICc=-48.58    BIC=-44.47

Training set error measures:
      MAPE
Training set 115.9148
```

The output shows that the model selected is an AR-2, I-1, or $\text{ARIMA}(2, 1, 0)$. The AR coefficients are produced, and again, I've abbreviated the output including only the error measure of MAPE, which is slightly better than the damped trend in the Holt model. We can examine the test data in the same fashion; just remember to include the number of the forecast periods, as follows:

```
> plot(forecast(fit.arima, h=8))
> lines(test, type="o")
```



Interestingly, the forecast shows a relatively flat trend. To examine MAPE on the test set, run the following code:

```
> mape1 = sum(abs((test-fit.holtd$mean)/test))/8

> mape1
[1] 0.1218026

> mape2 = sum(abs((test-forecast(fit.arima)$mean)/test))/8

> mape2
[1] 0.1312118
```

The forecast error is indeed slightly less for the **Holt Damped** trend model versus $\text{ARIMA}(2, 1, 0)$. Notice that the code to pull in the forecast values is slightly different for the ARIMA models produced with `auto.arima()`.

With the statistical and visual evidence, it seems that the best choice for a univariate forecast model is the Holt's method with a damped trend. The final thing that we can do is examine a plot with all the three forecasts side-by-side. To help with the visualization, we will start the actual data from 1990 onwards. The actual data will form the basis of the plots and the forecast will be added as lines with different line types (`lty`) and different plot symbols (`pch`). On the base plot, notice that the `y` axis limits (`ylim`) have to be set, otherwise the Holt forecast will be off the chart:

```
> T3>window(T2, start=1990)

> plot(T3, ylim=c(0.1,0.8))

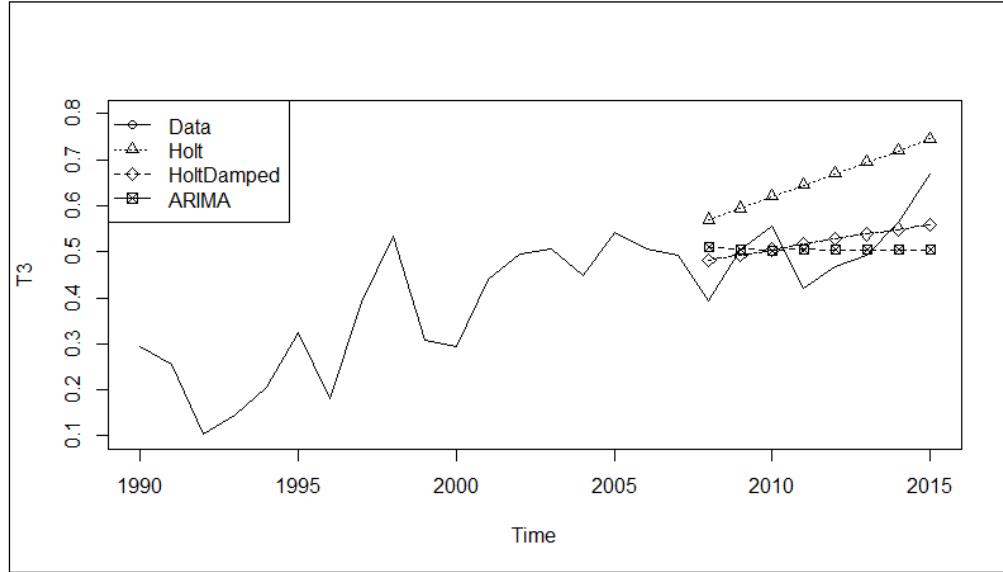
> lines(forecast(fit.holt)$mean, type="o",pch=2,lty="dotted")

> lines(forecast(fit.holtd)$mean, type="o",pch=5,lty=6)

> lines(forecast(fit.arima,h=8)$mean, type="o",pch=7,lty="dashed")

> legend("topleft", lty=c("solid","dotted","dashed"), pch=c(1,2,5,7), c("Data","Holt","HoltDamped","ARIMA"))
```

The output of the preceding code snippet is as follows:



With this, we completed the building of a univariate forecast model for the surface temperature anomalies and now we will move on to the next task.

Time series regression

In this second task of the modeling effort, we will apply the techniques to the climate change data. We will seek to predict the surface temperature anomalies using lags of itself and lags of emissions.

For starters, we will just build a linear model without using the lags in order to examine the serial correlation of the residuals with the `lm()` function. The other thing to do is to create an appropriate timeframe to examine. Recall that we saw the CO₂ emissions gradually increase around the end of World War II. Therefore, let's start the data in 1945, once again using the `window()` function and applying it to the climate data:

```
> y = window(climate[,1], start=1945)
```

```
> x = window(climate[,2], start=1945)
```

With this done, we can build the linear model and examine it:

```
> fit.lm = lm(y~x)
```

```
> summary(fit.lm)
```

Call:

```
lm(formula = y ~ x)
```

Residuals:

Min	1Q	Median	3Q	Max
-0.35257	-0.08782	0.00224	0.09732	0.27931

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-2.786e-01	3.890e-02	-7.161	9.02e-10 ***
x	8.082e-05	7.374e-06	10.960	< 2e-16 ***

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

```
Residual standard error: 0.1357 on 65 degrees of freedom
```

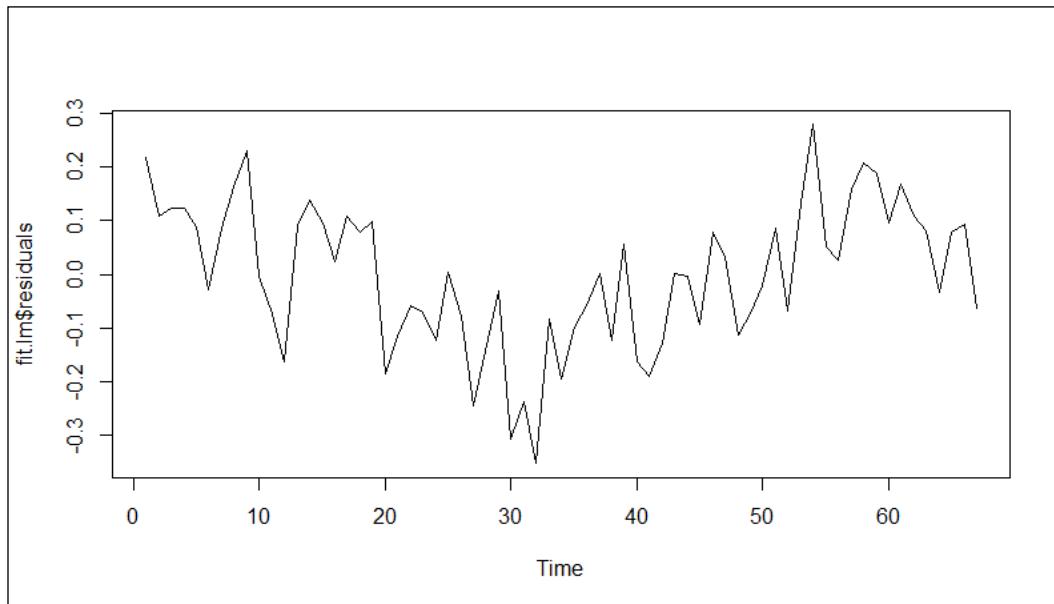
```
Multiple R-squared: 0.6489 Adjusted R-squared: 0.6435
```

```
F-statistic: 120.1 on 1 and 65 DF, p-value: < 2.2e-16
```

You can see that F-statistic for the overall model is highly statistically significant ($< 2.2e-16$) and that the x variable (CO2) is also highly significant. Adjusted R-squared is 0.6435. Our work here is not done as we still need to check the assumption of no correlation in the residuals. Two plots can provide the necessary insight; the first being `plot.ts()`, which will provide a time series plot of the residuals and also the autocorrelation plot that we used previously:

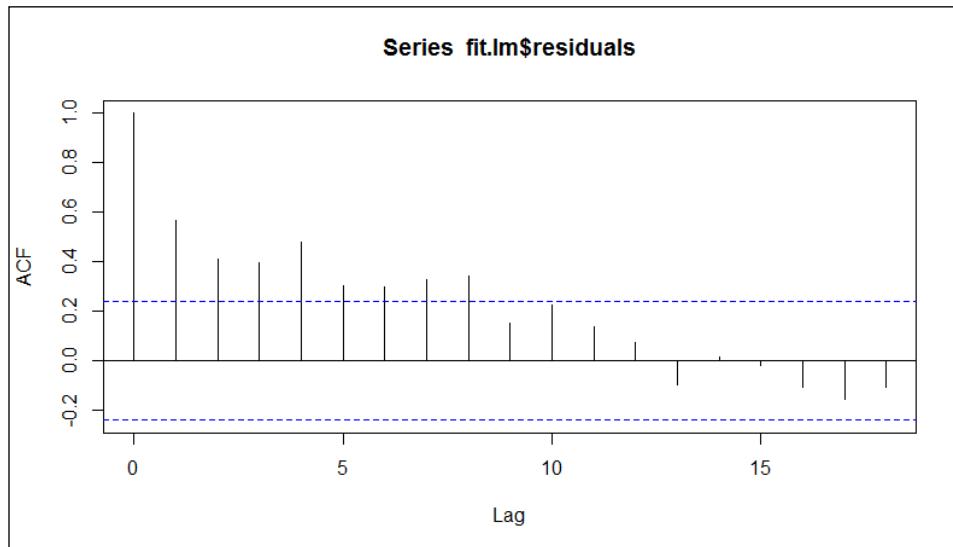
```
> plot.ts(fit.lm$residuals)
```

The output of the preceding command is as follows:



Note that there is a possible cyclical pattern in the residuals over time. To show conclusively that the model violates the assumption of no serial correlation, let's have a look at the following autocorrelation plot:

```
> acf(fit.lm$residuals)
```



You can clearly see that the first eight lags have significant spikes (correlation) and we can dismiss the assumption that the residuals do not have serial correlation. This is a classic example of the problem of looking solely at linear relationships of two time series without considering the lagged values.

Another thing available is Durbin-Watson test. This tests the null hypothesis that the residuals have zero autocorrelation. The test is available in the `lmtest` package, which automatically gets loaded with the `forecast` package. You can either specify your own linear model in the function or the object that contains a model; in our case, `fit.lm`, which you can see leads us to reject the null hypothesis and conclude that true autocorrelation is greater than 0, as follows:

```
> dwtest(fit.lm)
```

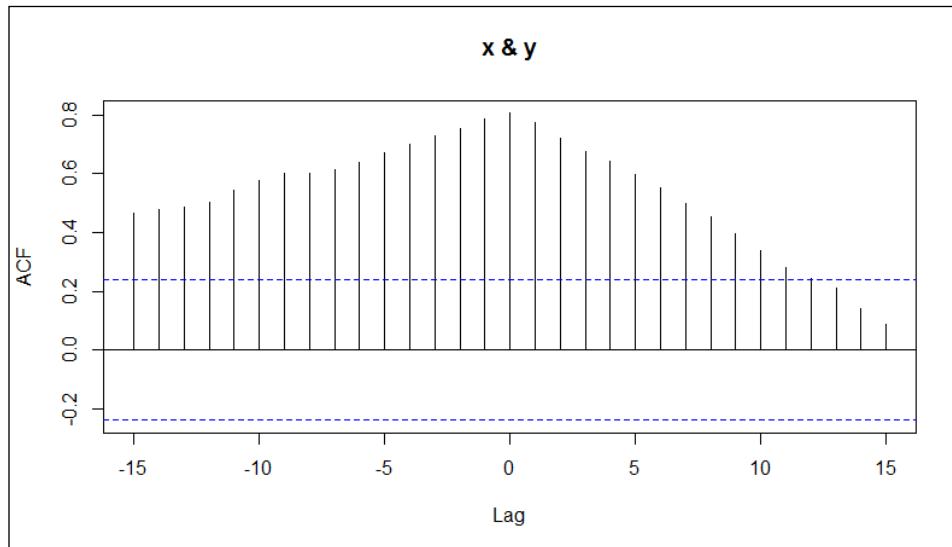
```
Durbin-Watson test

data: fit.lm
DW = 0.8198, p-value = 1.73e-08
alternative hypothesis: true autocorrelation is greater than 0
```

Having done this, where do we start in constructing a meaningful lag structure for the building of the model? Probably our best bet is to look at the cross correlation structure again, which is as follows:

```
> ccf(x, y)
```

The following is the output of the preceding command:



We have significant correlations of the lags of x through lag 15. Applying some judgment here (along with much trial and error on my part), let's start by looking at six lags of x and lag-1 and lag-4 of y in the regression model. A convenient way to do this is to use the dynamic linear regression package called `dynlm`. The only function available in the package is `dynlm()`; however, it offers quite a bit of flexibility in building models. The syntax of `dynlm()` follows the same procedure of `lm()`, but allows the inclusion of lag terms, seasonal terms, trends, and even harmonic patterns. To incorporate the lag terms as we want in our model, it will be necessary to specify the lags using `L()` in the function. For instance, if we wanted to regress the temperature by emissions and lag-1 emissions, the syntax would be `y~x+L(x, 1:6)`. Note that in `L()`, the variable and its lag is all that you need to specify. Here is how to build and examine the model with the first six lags of x and the first and fourth lag of y :

```
> fit.dyn = dynlm(y~x+L(x, 1:6)+L(y, c(1, 4)))
```

```
> summary(fit.dyn)
```

```
Time series regression with "ts" data:
```

Start = 1951, End = 2011

Call:

dynlm(formula = y ~ x + L(x, 1:6) + L(y, c(1, 4)))

Residuals:

Min	1Q	Median	3Q	Max
-0.241333	-0.049877	-0.000018	0.065519	0.155488

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)		
(Intercept)	-5.604e-02	5.417e-02	-1.034	0.305785		
x	-4.944e-05	1.235e-04	-0.400	0.690621		
L(x, 1:6)1	-2.556e-05	2.056e-04	-0.124	0.901545		
L(x, 1:6)2	3.110e-04	2.112e-04	1.472	0.147074		
L(x, 1:6)3	-2.798e-04	2.296e-04	-1.218	0.228667		
L(x, 1:6)4	2.086e-04	2.447e-04	0.852	0.398061		
L(x, 1:6)5	-5.687e-04	2.393e-04	-2.377	0.021262 *		
L(x, 1:6)6	4.319e-04	1.382e-04	3.125	0.002928 **		
L(y, c(1, 4))1	3.859e-01	1.079e-01	3.578	0.000769 ***		
L(y, c(1, 4))4	3.957e-01	1.085e-01	3.649	0.000619 ***		

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

Residual standard error: 0.1009 on 51 degrees of freedom

Multiple R-squared: 0.8377, Adjusted R-squared: 0.8091

F-statistic: 29.26 on 9 and 51 DF, p-value: < 2.2e-16

So, we have a highly significant p-value for the overall model and Adjusted R-squared of 0.8091. Looking at the p-values for the coefficients, we have p-values less than 0.05 with both the lags of y (temperature) and lag five and six of x (emissions). We can adjust the model by dropping the insignificant x lags and then test the assumption of no serial correlation, as follows:

```
> fit.dyn2 = dynlm(y~L(x,c(5,6))+L(y,c(1,4)))

> summary(fit.dyn2)

Time series regression with "ts" data:
Start = 1951, End = 2011

Call:
dynlm(formula = y ~ L(x, c(5, 6)) + L(y, c(1, 4)))

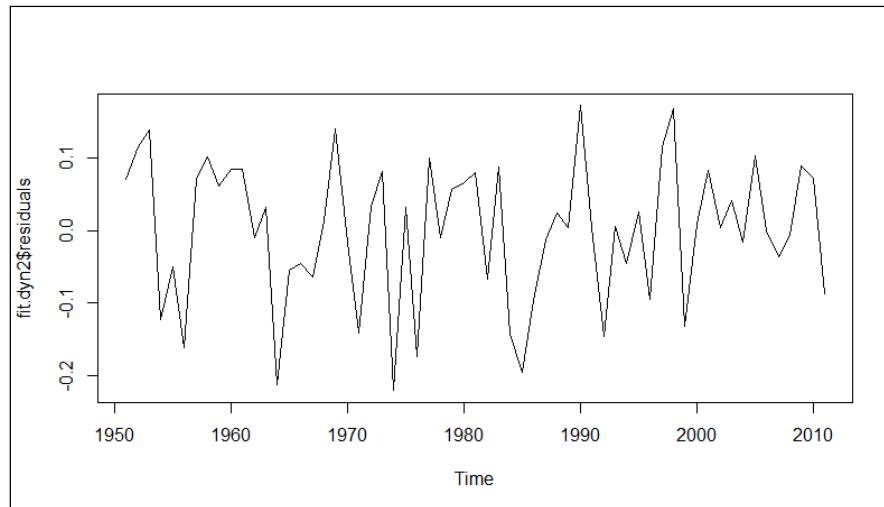
Residuals:
    Min      1Q  Median      3Q     Max 
-0.220798 -0.054835  0.005527  0.079318  0.172035 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -0.0367531  0.0480225 -0.765 0.447288  
L(x, c(5, 6))5 -0.0002963  0.0001217 -2.434 0.018157 *  
L(x, c(5, 6))6  0.0003224  0.0001222  2.638 0.010765 *  
L(y, c(1, 4))1  0.4238987  0.1043383  4.063 0.000153 *** 
L(y, c(1, 4))4  0.3735944  0.1047414  3.567 0.000749 *** 
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1006 on 56 degrees of freedom
Multiple R-squared:  0.823, Adjusted R-squared:  0.8104 
F-statistic: 65.1 on 4 and 56 DF,  p-value: < 2.2e-16
```

Our overall model is again significant and Adjusted R-squared has improved slightly as we dropped the irrelevant terms. The lagged coefficients are all positive with the exception of lag-5 of x. Examining the residual plots, we end up with the following plots:

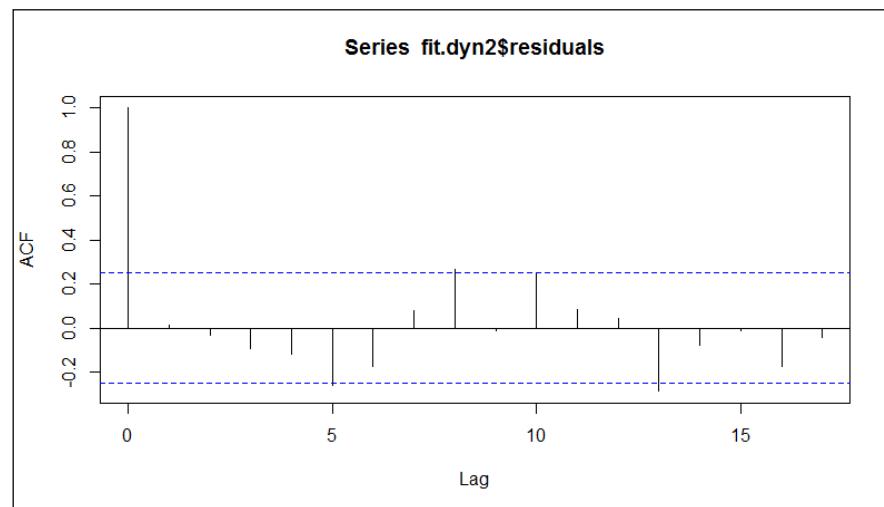
```
> plot(fit.dyn2$residuals)
```



And, then the ACF plot of the residuals:

```
> acf(fit.dyn2$residuals)
```

The output of the preceding command is as follows:



The first plot does not show any obvious pattern in the residuals. The `acf` plot does show the slightest significant correlation at a couple of lags, but they are so minor that we can choose to ignore them. Now, if you exclude lag-4 of `y`, the `acf` plot will show a rather significant spike at lag-4. I'll let you try this out for yourself. Let's wrap this up with Durbin-Watson test, which does not lead to a rejection of the null hypothesis:

```
> dwtest(fit.dyn2)

Durbin-Watson test

data: fit.dyn2
DW = 1.9525, p-value = 0.3018
alternative hypothesis: true autocorrelation is greater than 0
```

With this model, we can say that the linear model to predict the surface temperature anomalies is the following:

```
y = -0.37 - 0.000296*lag5(x) + 0.000322*lag6(x) + 0.424*lag1(y) +
0.374*lag4(y)
```

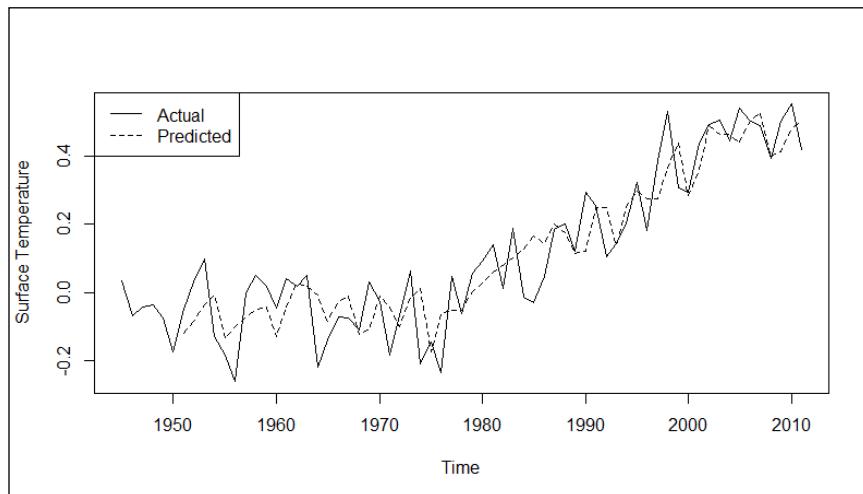
Interestingly, `lag6` of `x` has a slight positive effect on `y` but is nearly cancelled out by `lag5` of `x`. Let's plot the **Actual** versus **Predicted** values to get a visual sense of how well the model performs:

```
> plot(y, ylab="Surface Temperature")

> lines(fitted(fit.dyn2), pch=2, lty="dashed")

> legend("topleft", lty=c("solid","dashed"), c("Actual","Predicted"))
```

The output of the preceding command is as follows:



I would like to point out one thing here. If we had started in **1970** as we did with the univariate forecast and incorporated a linear trend in the model, the emissions' lags would not have been significant. So, by starting earlier, we did capture significant lags of the emissions. But what does it all mean? For our purpose in trying to find a link between human CO₂ emissions and global warming, it doesn't seem to amount to much of anything. Now, let's turn our attention to trying to prove the statistical causality between the two.

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 Granger causality.

The plan here is to go with the data starting in 1945 as we did with the bivariate regression. To explore the issues that Kodra (2010) had, we will need to see if and how we can make the data stationary. To do this, the `forecast` package provides the `ndiffs()` function, which provides you with 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), ADF, or Philips-Peron (PP). I will use KPSS in the following code, which has a null hypothesis that the data is stationary. If the null hypothesis is rejected, the function will return the number of differences in order to achieve stationarity. Note that `adf` and `pp` have the null hypothesis that the data is not stationary.

```
> ndiffs(x, test="kpss")
[1] 1

> ndiffs(y, test="kpss")
[1] 1
```

In both the cases, the first-order differencing will achieve stationarity, allowing us to perform Granger causality with a high degree of confidence. To get started, we will put both the time series into one dataset and then create the first-order differenced series, as follows:

```
> Granger = cbind(y,x)

> dGranger = diff(Granger)
```

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 10 lags:

```
> lag=VARselect(dGranger, lag.max=10)
```

The information criteria can be called using `lag$selection`. Four different criteria are provided including AIC, **Hannan-Quinn Criterion (HQ)**, **Schwarz-Bayes Criterion (SC)**, and FPE. Note that AIC and SC are covered in *Chapter 2, Linear Regression – The Blocking and Tackling of Machine Learning*, so I will not go over the criterion formulas or differences here. If you want to see the actual results, you can use `lag$criteria`:

```
> lag$selection
AIC(n)   HQ(n)   SC(n)   FPE(n)
      5        1        1        5
```

We can see that AIC and FPE have selected lag 5 as the optimal structure to a VAR model. We can forgo lag-1 as it doesn't seem to make sense in the world of climate change while a lag of 5 years does. Therefore, we will examine a lag of 5 using the var() function and the results:

```
> lag5 = VAR(dGranger, p=5)

> summary(lag5)

VAR Estimation Results:
=====
Endogenous variables: y, x
Deterministic variables: const
Sample size: 61
Log Likelihood: -310.683
Roots of the characteristic polynomial:
0.8497 0.8183 0.8183 0.8108 0.8108 0.7677 0.7499 0.7499 0.7076 0.7076
Call:
VAR(y = dGranger, p = 5)

Estimation results for equation y:
=====
y = y.11 + x.11 + y.12 + x.12 + y.13 + x.13 + y.14 + x.14 + y.15 + x.15 +
const

      Estimate Std. Error t value Pr(>|t|) 
y.11   -4.992e-01 1.272e-01 -3.925 0.000266 ***
x.11   -1.268e-04 1.245e-04 -1.019 0.313027  
y.12   -5.057e-01 1.409e-01 -3.589 0.000754 ***
x.12    2.570e-04 1.367e-04  1.879 0.066018  
y.13   -4.174e-01 1.455e-01 -2.868 0.006030 ** 
x.13   -7.257e-05 1.448e-04 -0.501 0.618358  
y.14    3.467e-02 1.417e-01  0.245 0.807735  
x.14    1.511e-04 1.489e-04  1.014 0.315328  
y.15   -2.015e-01 1.285e-01 -1.568 0.123245  
x.15   -4.041e-04 1.383e-04 -2.922 0.005208 **
```

```
const  4.762e-02  2.768e-02   1.720  0.091542 .

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

Residual standard error: 0.1022 on 50 degrees of freedom
Multiple R-Squared: 0.4481, Adjusted R-squared: 0.3378
F-statistic: 4.06 on 10 and 50 DF,  p-value: 0.00041

Estimation results for equation x:
=====
x = y.l1 + x.l1 + y.l2 + x.l2 + y.l3 + x.l3 + y.l4 + x.l4 + y.l5 + x.l5 +
const

      Estimate Std. Error t value Pr(>|t| )
y.l1    -73.67538 141.97873 -0.519  0.60611
x.l1     0.35225   0.13896  2.535  0.01442 *
y.l2   -221.78216 157.29843 -1.410  0.16475
x.l2    -0.06238   0.15267 -0.409  0.68457
y.l3   -121.46591 162.47887 -0.748  0.45822
x.l3     0.24408   0.16161  1.510  0.13725
y.l4   -251.22176 158.22613 -1.588  0.11865
x.l4    -0.21250   0.16627 -1.278  0.20714
y.l5   -170.93505 143.49020 -1.191  0.23917
x.l5     0.05856   0.15438  0.379  0.70604
const   87.31476  30.89869  2.826  0.00676 **

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

Residual standard error: 114.1 on 50 degrees of freedom
Multiple R-Squared: 0.2281, Adjusted R-squared: 0.07366
F-statistic: 1.477 on 10 and 50 DF,  p-value: 0.1759
```

The results shown are for both the models. You can see that the overall model to predict y is significant with the lags 2 and 5 of x having p-values less than 0.1. The model to predict x is not significant. As we did in the previous section, we should check for the 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, which is the default. The null hypothesis is that autocorrelations are zero and the alternate is that they are not zero:

```
> serial.test(lag5,type="PT.asymptotic")

Portmanteau Test (asymptotic)

data: Residuals of VAR object lag5
Chi-squared = 36.4377, df = 44, p-value = 0.7839
```

With p-value at 0.7839, we do not have evidence to reject the null and can say that the residuals are not autocorrelated.

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 `lag5` object previously created:

```
> x2y = causality(lag5,cause="x")

> y2x = causality(lag5,cause="y")

It is now just a simple matter to call the Granger test results:
```

```
> x2y$Granger

Granger causality H0: x do not Granger-cause y

data: VAR object lag5
F-Test = 2.0883, df1 = 5, df2 = 100, p-value = 0.07304

> y2x$Granger

Granger causality H0: y do not Granger-cause x

data: VAR object lag5
F-Test = 0.731, df1 = 5, df2 = 100, p-value = 0.6019
```

The p-value value for x Granger-causing y is 0.07304 and for y causing x is 0.6019. So what does all this mean? The first thing that we can say is 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 this the relevant conclusion here? Remember that 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 were a controlled experiment, then likely we wouldn't hesitate to say that we had insufficient evidence to reject the null, for example, a phase-3 clinical trial. As this study is based on observational data, I believe we can say that it is highly probable that CO₂ emissions Granger-cause the surface temperature anomalies. However, there is a lot of room for criticism in this 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 from. I chose 1945 because it looked about right; you could say that 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. The other thing that I want to point out is the lag of five years and the coefficient, which is negative.

Now, the Granger causality test is not designed to use the coefficients from a vector autoregression in a forecast, so it is not safe to say that an increase in the CO₂ emissions would lead to lower temperatures five years later. We are merely looking for a causal relationship, which seems to be based on a five-year lag. Assume that the real-world relationship was 20 or 30 years, then this technique would be irrelevant to the problem given the timeframe in question. It would also be interesting to include a third variable such as a measure of annual solar radiation, but this was beyond the scope of this chapter.

The last thing to show here is how to use vector autoregression in order to produce a forecast. With the following predict() function, we can produce a point estimate and confidence intervals for a timeframe that we specify:

```
> predict(lag5, n.ahead=10, ci=0.95)
$y
    fcst      lower      upper       CI
[1,] 0.081703785 -0.1186856 0.28209313 0.2003893
[2,] 0.017119634 -0.2076806 0.24191986 0.2248002
[3,] 0.096799604 -0.1424826 0.33608184 0.2392822
[4,] -0.149113923 -0.3888954 0.09066751 0.2397814
[5,] -0.011618877 -0.2611685 0.23793073 0.2495496
[6,] 0.054878791 -0.2137249 0.32348244 0.2686036
[7,] 0.021115281 -0.2479521 0.29018265 0.2690674
[8,] -0.035593173 -0.3056111 0.23442472 0.2700179
```

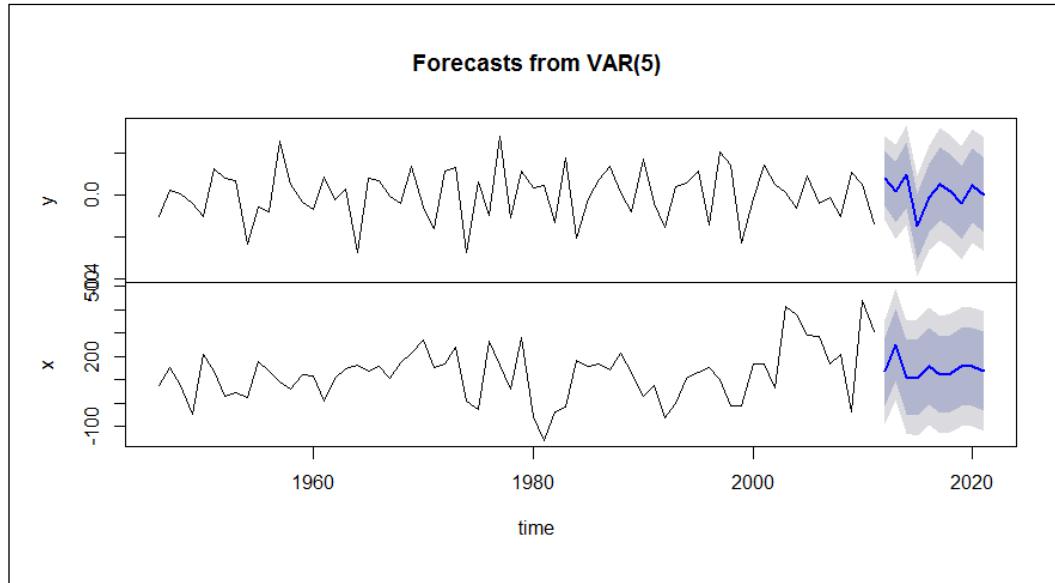
```
[9,] 0.045634059 -0.2251345 0.31640259 0.2707685  
[10,] 0.003465538 -0.2687530 0.27568406 0.2722185
```

\$x

	fcst	lower	upper	CI
[1,]	136.7695	-86.94633	360.4853	223.7158
[2,]	251.0853	13.06491	489.1057	238.0204
[3,]	112.0863	-130.39147	354.5642	242.4778
[4,]	108.7158	-140.90916	358.3408	249.6250
[5,]	158.7534	-93.41522	410.9221	252.1686
[6,]	122.3686	-131.04145	375.7787	253.4101
[7,]	127.3391	-126.76240	381.4405	254.1015
[8,]	155.3462	-99.09757	409.7899	254.4437
[9,]	156.2730	-98.52536	411.0713	254.7983
[10,]	137.2128	-118.36867	392.7943	255.5815

Finally, a plot is available to view the forecasts:

```
> plot(forecast(lag5))
```



When all is said and done, it clearly seems that more work needs to be done, but I believe that Granger causality has pointed us in the right direction. 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 examine the climate change data in more detail. There should be a high bar when it comes to demonstrating the causality, and Granger causality is a great tool for assisting in this 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, 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 if we can say, statistically speaking, that human CO₂ emissions cause surface temperature anomalies. While the results—that the temperature change is caused by CO₂ emissions—were compelling, they do not seem definitive. However, 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 toolbox.

12

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 in textual data. If you Google, Bing, or Yahoo how much of the 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 the 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 distil the data into meaningful quantitative results, support the qualitative understanding, and maybe even sway the swooper.

Over the last several years, I've applied the techniques discussed here to mine physician-patient interactions, understand FDA fears on prescription drug advertising, and capture patient concerns in a rare cancer, to name just a few. Using R and the methods in this chapter, you too can extract the powerful information in the 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 a bit complicated. Gathering and compiling the text data is a topic that could take up several chapters. Therefore, let's begin with the assumption that the 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 comprises 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 the 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 the 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 the words can get a bit tricky and might add to your confusion because it deletes the word suffixes, creating the base word or what is known as the **radical**. We will use the stemming algorithm included in the R package, `tm`, where the function calls the **Porter stemming algorithm**. 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. We will see in the business case that you can use replacement to delete unnecessary text and characters.

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 do 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. The text mining can be performed on either matrix.

With a matrix, you can begin to analyze the text by examining the 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 the 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 <https://www.cs.princeton.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 do 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 where 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 is the text. 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 numbers 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 the 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 has 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 with the most recent speech in January, 2015 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(SnowballC)

> library(wordcloud)

> library(RColorBrewer)
```

To bring the data in R, we could scrape the `www.whitehouse.gov` website. I dismissed this idea as this chapter would turn into a web scraping exposition and not one on text mining. So, I've pasted and stored the necessary data in the free website, `www.textuploader.com`. Each year's speech has a separate URL and we will only need to reference them to acquire the data. Two functions will accomplish this for us; the first being `scan()`, which reads the data and `paste()` to concatenate it properly:

```
> sou2010 = paste(scan(url("http://textuploader.com/a5vq4/raw") ,
what="character"), collapse=" ")

Read 7415 items
```

This is the 2010 speech. Now, one issue that you need to deal with when using text data in R is that it should be in the ASCII format. If not (the 2010 speech is not), then you must convert it to ASCII. The text that we pulled in previously is filled with numerous non-ASCII characters that would take many lines of code to try and delete/replace with the `gsub()` function. However, let's deal with this problem in one line of code, putting the `iconv()` function to good use. Remember that if you pull in text to R and see a number of funky characters, check if you need to convert it:

```
> sou2010=iconv(sou2010, "latin1", "ASCII", "")
```

We can pull up the entire speech by making a call to `sou2010`, but I'll just present the first few and last few sentences:

```
> sou2010  
[1] "THE PRESIDENT: Madam Speaker, Vice President Biden, members  
of Congress, distinguished guests, and fellow Americans: Our  
Constitution declares that from time to time, the President shall  
give to Congress information about the state of our union. For 220  
years, our leaders have fulfilled this duty. They've done so during  
periods of prosperity and tranquility. And they've done so in the  
midst of war and depression; at moments of great strife and great  
struggle.....
```

Let's seize this moment -- to start anew, to carry the dream forward, and
to strengthen our union once more. (Applause.) Thank you. God bless you.
And God bless the United States of America. (Applause.)

Let's bring in the other five speeches:

```
> sou2011 = paste(scan(url("http://textuploader.com/a5vm0/raw"),  
what="character"),collapse=" ")
```

Read 7017 items

```
> sou2011=iconv(sou2011, "latin1", "ASCII", "")
```

```
> sou2012 = paste(scan(url("http://textuploader.com/a5vmp/raw"),  
what="character"),collapse=" ")
```

Read 7132 items

```
> sou2012=iconv(sou2012, "latin1", "ASCII", "")
```

```
> sou2013 = paste(scan(url("http://textuploader.com/a5vh0/raw"),  
what="character"),collapse=" ")
```

```
Read 6908 items

> sou2013=iconv(sou2013, "latin1", "ASCII", "")

> sou2014 = paste(scan(url("http://textuploader.com/a5vhp/raw"),
what="character"),collapse=" ")
Read 6829 items

> sou2014=iconv(sou2014, "latin1", "ASCII", "")

> sou2015 = paste(scan(url("http://textuploader.com/a5vhb/raw"),
what="character"),collapse=" ")
Read 6849 items

> sou2015=iconv(sou2015, "latin1", "ASCII", "")
```

We should put this in a file to hold the documents that will form the corpus. If you don't know the current working directory, you can pull it up with `getwd()` and change it with `setwd()`. Do not put your text files with any other file; create a new file folder for the speeches, otherwise your corpus will contain the R code, some other file, or blow up when you try to create it:

```
> getwd()
[1] "C:/Users/clesmeister/chap12/textmine"

> write.table(sou2010, "c:/Users/clesmeister/chap12/text/sou2010.txt")

> write.table(sou2011, "c:/Users/clesmeister/chap12/text/sou2011.txt")

> write.table(sou2012, "c:/Users/clesmeister/chap12/text/sou2012.txt")

> write.table(sou2013, "c:/Users/clesmeister/chap12/text/sou2013.txt")

> write.table(sou2014, "c:/Users/clesmeister/chap12/text/sou2014.txt")

> write.table(sou2015, "c:/Users/clesmeister/chap12/text/sou2015.txt")
```

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("C:/Users/clesmeister/chap12/text")

> length(dir(name))
[1] 6

> dir(name)
[1] "sou2010.txt" "sou2011.txt" "sou2012.txt" "sou2013.txt"
[5] "sou2014.txt" "sou2015.txt"
```

We will call the corpus `docs` and it is created with the `Corpus()` function, wrapped around the `DirSource()` function, which is also part of the `tm` package:

```
> docs = Corpus(DirSource(name))

> docs
<<VCorpus>>

Metadata: corpus specific: 0, document level (indexed): 0
Content: documents: 6
```

Note that there is no `corpus` or `document` level `Metadata` in this data. There are functions in the `tm` package to apply things such as authors' 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)

> docs = tm_map(docs, stemDocument)
```

At this point, it is a good idea to eliminate the unnecessary words. For example, during the speeches, when Congress applauds a statement, you will find (Applause) in the text. This must go away. Keep in mind that we stemmed the documents and so we need to get rid of applaus:

```
> docs = tm_map(docs, removeWords,c("""applaus""", ""can"", ""cant"" , ""will"""
,""that"" , ""weve"" , ""dont"" , ""wont""))
```

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] 6 3080
```

The six speeches contain 3080 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. So, with six documents, by specifying 0.51 as the sparsity number, the resulting matrix would have words that occurred in at least three documents, as follows:

```
> dtm = removeSparseTerms(dtm, 0.51)

> dim(dtm)
[1] 6 1132
```

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")
```

Using the `inspect()` function, you can examine the matrix. Here, we will look at all the six rows and the first five columns:

```
> inspect(dtm[1:6, 1:5])  
  Terms  
  Docs abl abroad absolut abus accept  
  2010 1 2 2 1 1  
  2011 4 3 0 0 0  
  2012 3 1 1 1 0  
  2013 3 2 1 0 1  
  2014 1 4 0 0 0  
  2015 1 1 0 2 1
```

It appears that our data is ready for analysis, starting with looking at the word frequency counts.

Modeling and evaluation

Modeling will be broken in 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 `head` and `tail` of the object with the following code:

```
> freq[head(ord)]  
american      year       job      work   america      new
```

```
243      241      212      195      187      177

> freq[tail(ord)]
    voic      welcom worldclass      yearold      yemen
    3          3          3          3          3
youll
    3
```

The most frequent word is `american`—as you might expect from the President—but notice how important its employment is with `job` and `work`. You can see how stemming changed `voice` to `voic` and `welcome`/`welcoming` to `welcom`.

To look at the frequency of the word frequency, you can create tables, as follows:

```
> head(table(freq))
freq
 3   4   5   6   7   8
127 118 112  75  65  50

> tail(table(freq))
freq
177 187 195 212 241 243
 1   1   1   1   1   1
```

What these tables show is the number of words with that specific frequency, so 127 words occurred three times and one word, `american` in our case, occurred 243 times.

Using `findFreqTerms()`, we can see what words occurred at least 100 times. Looks like he talked quite a bit about business and it is clear that the government, including the IRS, is here to "help", perhaps even help "now". That is a relief!

```
> findFreqTerms(dtm, 100)
[1] "america"  "american" "busi"       "countri"   "everi"
[6] "get"       "help"      "job"        "let"       "like"
[11] "make"     "need"     "new"        "now"       "one"
[16] "peopl"    "right"    "time"      "work"     "year"
```

You can find associations with words by correlation with the `findAssocs()` function. Let's look at business and also job as two examples using 0.9 as the correlation cutoff:

```
> findAssocs(dtm, "busi", corlimit=0.9)
$busi
drop eager hear fund add main track
0.98 0.98 0.92 0.91 0.90 0.90 0.90

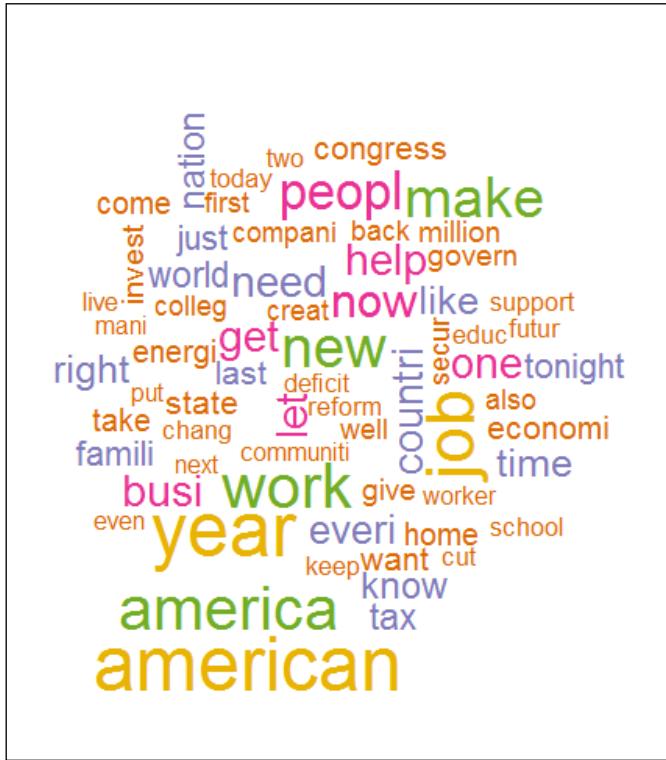
> findAssocs(dtm, "job", corlimit=0.9)
$job
hightech      lay    announc natur
0.94        0.94        0.93     0.93
aid  alloftheabov burma cleaner
0.92        0.92        0.92     0.92
ford   gather involv poor
0.92        0.92        0.92     0.92
redesign skill yemen sourc
0.92        0.92        0.92     0.91
```

Business needs further exploration, but jobs is interesting in the focus on high-tech jobs. It is curious that burma and yemen show up; I guess we still have a job to do on these countries, certainly in yemen.

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 50:

```
> wordcloud(names(freq), freq, min.freq=50, scale=c(3, .5),
colors=brewer.pal(6, "Dark2"))
```

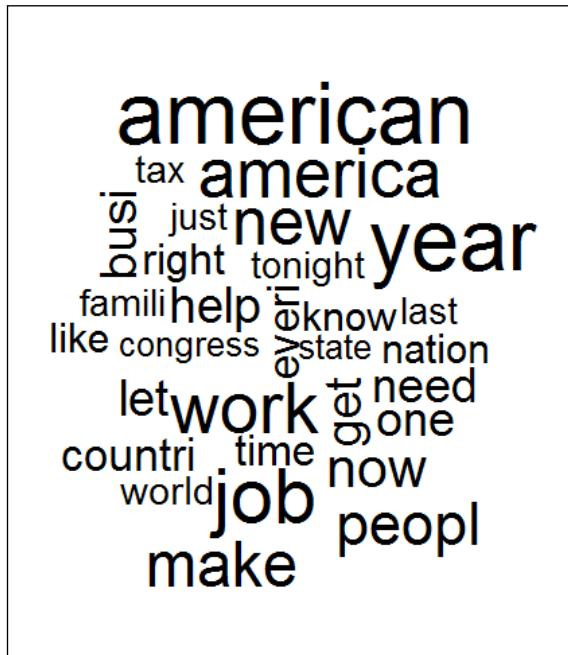
The output of the preceding command is as follows:



One can forgo all the fancy graphics as we will in the following image, capturing 30 most frequent words:

```
> wordcloud(names(freq), freq, max.words=30)
```

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:

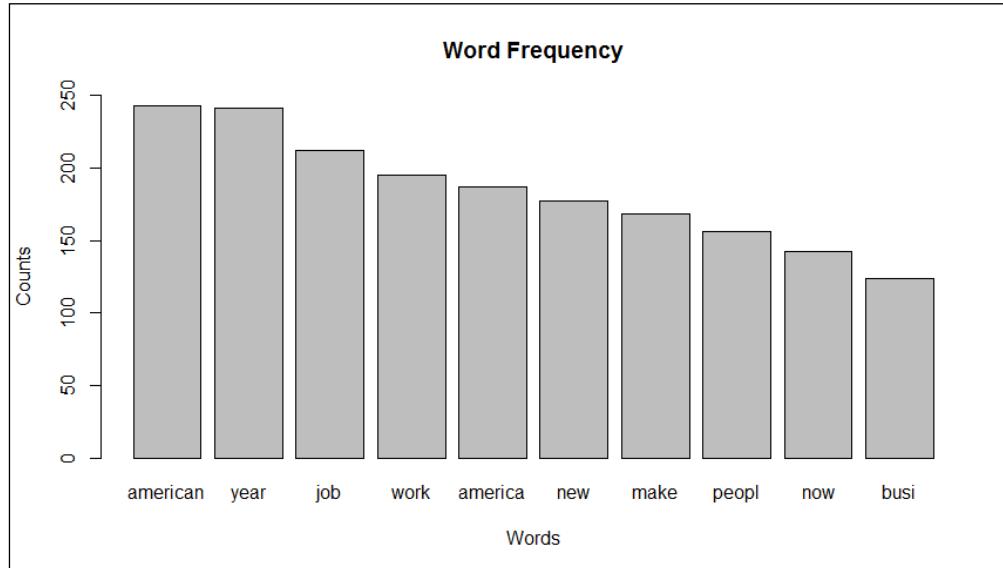
```
> freq = sort(colSums(as.matrix(dtm)), decreasing=TRUE)

> 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 the building of 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 or four, so we will try both, starting with three topics ($k=3$):

```
> library(topicmodels)

> set.seed(123)

> lda3 = LDA(dtm, k=3, method="Gibbs")

> topics(lda3)
2010 2011 2012 2013 2014 2015
      3      3      1      1      2      2
```

We can see that `topics` are grouped every two years.

Now we will try for topics ($k=4$):

```
> set.seed(456)

> lda4 = LDA(dtm, k=4, method="Gibbs")

> topics(lda4)
2010 2011 2012 2013 2014 2015
4     4     3     2     1     1
```

Here, the topic groupings are similar to the preceding ones, except that the 2012 and 2013 speeches have their own topics. For simplicity, let's have a look at three topics for the speeches. 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, 20)
  Topic 1    Topic 2    Topic 3
[1,] "american" "new"      "year"
[2,] "job"       "america"   "peopl"
[3,] "now"       "work"     "know"
[4,] "right"     "help"     "nation"
[5,] "get"       "one"      "last"
[6,] "tax"       "everi"    "take"
[7,] "busi"      "need"     "invest"
[8,] "energi"    "make"     "govern"
[9,] "home"      "world"    "school"
[10,] "time"     "countri"   "also"
[11,] "like"      "let"      "cut"
[12,] "million"   "congress"  "two"
[13,] "give"      "state"    "next"
[14,] "well"      "want"     "come"
[15,] "compani"   "tonight"   "deficit"
[16,] "reform"    "first"    "chang"
[17,] "back"      "futur"    "famili"
[18,] "educ"      "keep"     "care"
[19,] "put"       "today"    "econom"
[20,] "unit"      "worker"   "work"
```

Topic 3 covers the first two speeches. Some key words stand out, such as "invest", "school", "economy", and "deficit". During this time, Congress passed and implemented the \$787 billion **American Recovery and Reinvestment Act** with the goal of stimulating the economy.

Topic 1 covers the next two speeches. Here, the message transitions to "job", "tax", "busi", and what appears to be some comments on the "energi" policy. A supposed comprehensive policy put forward under the rhetorical All of the above in the 2012 speech. Note the association with the rhetorical comment and jobs when we examined it with `findAssocs()`.

Topic 2 brings us to the last two speeches. There doesn't appear to be a clear topic that rises to the surface like the others. It appears that these speeches were less about specific calls to action and more about what was done and the future vision of the country and the world. In the next section, we can dig into the exact speech content further, along with comparing and contrasting his first State of the Union speech with the most recent one.

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 2015 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. You can include multiple variables in your groups. We will not need to do any of the other transformations such as stemming or lowering the case.

Before creating a data frame, we will need to get rid of that pesky (`Applause.`) text with the `gsub` function. We will also need to load the library:

```
> library(qdap)

> state15 = gsub("(Applause.)", "", sou2015)

Now, put this in df and split it into sentences, which will put one sentence per row. As proper punctuation is in the text, you can use the sentSplit function. If punctuation was not there, other functions are available to detect the sentences:

> speech15 = data.frame(speech=state15)

> sent15 = sentSplit(speech15, "speech")
```

The last thing is to create the year variable:

```
> sent15$year = "2015"
```

Repeat the steps for the 2010 speech:

```
> state10 = gsub("(Applause.)", "", sou2010)
```

```
> speech10 = data.frame(speech=state10)
```

```
> sent10 = sentSplit(speech10, "speech")
```

```
> sent10$year = "2010"
```

Now, concatenate the two datasets:

```
> sentences = rbind(sent10, sent15)
```

To compare the polarity (sentiment scores), use the `polarity()` function, specifying the text and grouping variables:

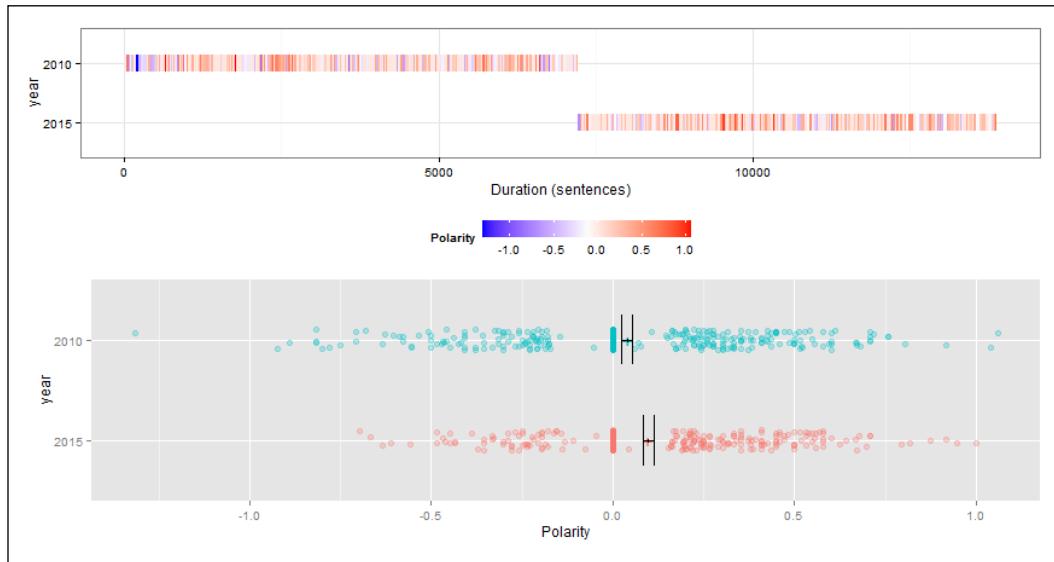
```
> pol = polarity(sentences$speech, sentences$year)
```

```
> pol
  year total.sentences total.words ave.polarity sd.polarity stan.mean.
polarity
  1 2010          443        7233      0.040      0.319
  0.124
  2 2015          378        6712      0.098      0.274
  0.356
```

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 (0.356) than 2010 (0.124). This is in line with what we expect. 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 more negative than 2015. We can identify this sentence by creating a data frame of the `pol` object, find the sentence number, and call this sentence:

```
> 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! 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      7207          443      33623
2 2015      6671          378      30469
Automated_Readability_Index
1                      8.677994
2                      8.906440
```

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)

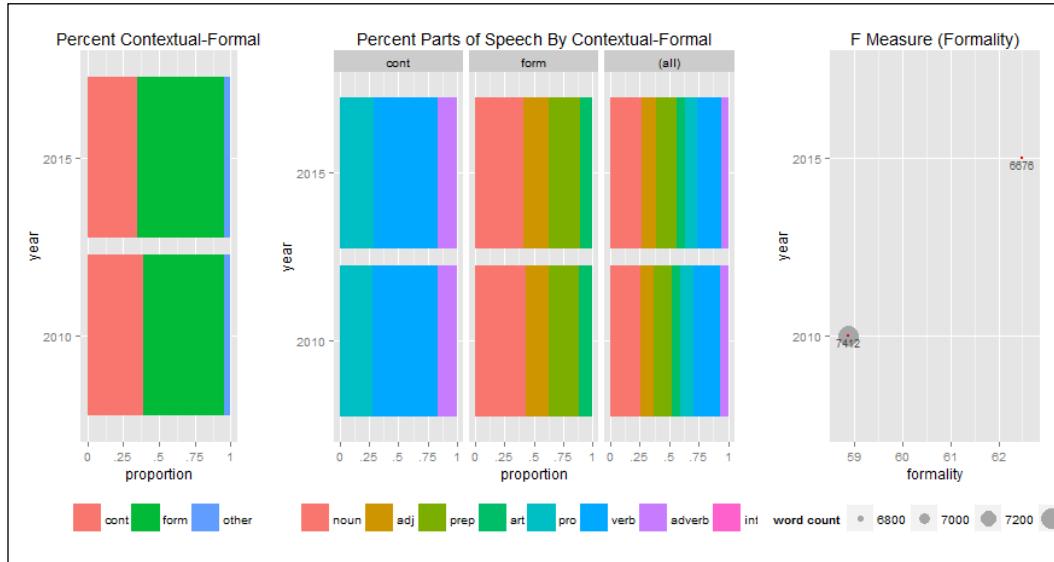
> form
  year word.count formality
1 2015      6676      62.49
2 2010      7412      58.88
```

This looks to be very similar. We can examine the proportion of the parts of the speech and also produce a plot that confirms this, as follows:

```
> form$form.prop.by
  year word.count   noun    adj    prep articles pronoun
1 2010      7412 24.22 11.39 14.64      6.46   10.75
2 2015      6676 24.94 12.46 16.37      6.34   10.23
  verb adverb interj other
1 21.57   6.58   0.03  4.36
2 19.19   5.69   0.01  4.76

> plot(form)
```

The following is the output of the preceding command:



Now, the diversity measures have been produced. Again, they are nearly identical. A plot is also available, (`plot(div)`), but being so similar, it adds no value. It is important to note that Obama's speech writer for 2010 was Jon Favreau, and in 2015, it was Cody Keenan:

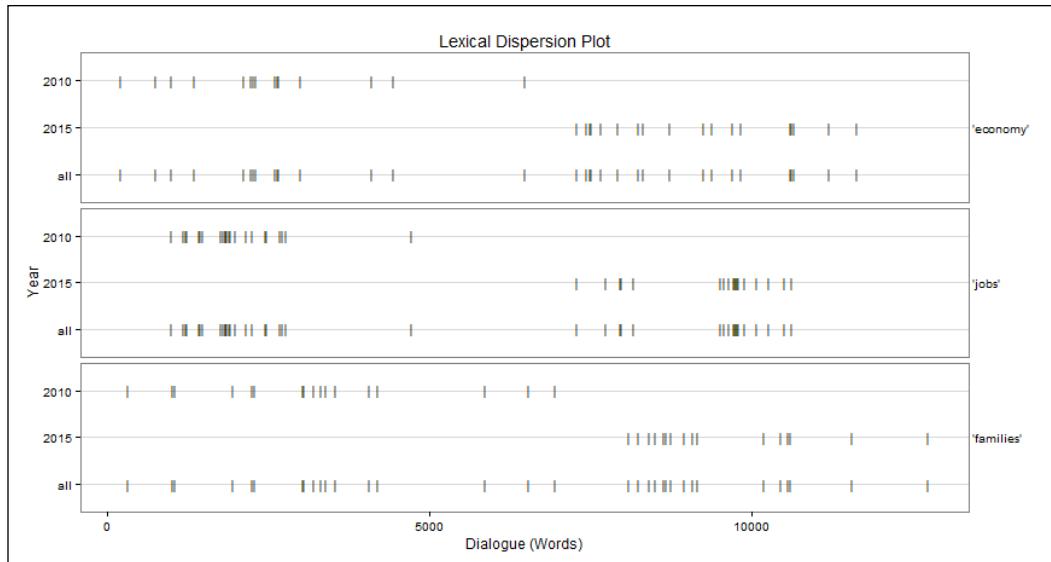
```
> div = diversity(sentences$speech, sentences$year)

> div
  year   wc simpson shannon collision berger_parker brillouin
1 2010 7207    0.992     6.163      4.799        0.047     5.860
2 2015 6671    0.992     6.159      4.791        0.039     5.841
```

Text Mining

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$speech, grouping.var=sentences$year,
  c("economy", "jobs", "families"), color="black", bg.color="white")
```



This is quite interesting as these topics were discussed early on in the 2010 speech but at the end in the 2015 speech.

Many of the tasks that we performed earlier with the `tm` package can also be done in `qdap`. So, the last thing that I want to do is show you how to execute the word frequency with `qdap` and count the top ten words for each speech. This is easy with the `freq_terms()` function. In addition to specifying the top ten words, we will also specify one of the `stopwords` defaults available in `qdap`. In this case, 200 versus the other option of 100:

```
> freq2010 = freq_terms(sent10$speech, top=10, stopwords=Top200Words)

> freq2010
    WORD      FREQ
1 americans    28
2 that's       26
3 jobs         23
4 it's         20
```

```
5 years      19
6 american   18
7 businesses 18
8 those      18
9 families   17
10 last      16
```

```
> freq2015 = freq_terms(sent15$speech, top=10, stopwords=Top200Words)
```

```
> freq2015
    WORD      FREQ
1 that's     28
2 years      25
3 every      24
4 american   19
5 country    19
6 economy    18
7 jobs       18
8 americans  17
9 lets       17
10 families  16
```

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 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 six 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.

R Fundamentals

"One of my most productive days was throwing away 1000 lines of code."

— Ken Thompson

Introduction

This chapter covers the basic programming syntax functions and capabilities of R. It has been put forward to introduce you to R and accelerate your learning. The objectives are as follows:

- Install R and RStudio
- Create and explore vectors
- Create data frames and matrices
- Explore mathematical and statistical functions
- Build simple plots
- Install and load 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:

The screenshot shows the main page of the R Project. On the left is a sidebar with the R logo and links to [Home], Download (CRAN), R Project (About R, Contributors, What's New?, Mailing Lists, Bug Tracking, Conferences, Search), and R Foundation (Foundation, Board, Members, Donors, Donate). The main content area has a large title "The R Project for Statistical Computing". Below it is a section titled "Getting Started" with a paragraph about R being a free software environment for statistical computing and graphics. It includes a link to download R from CRAN mirrors. Another section titled "News" lists recent releases: R version 3.2.2 (Fire Safety) released on 2015-08-14, The R Journal Volume 7/1 available, R version 3.1.3 (Smooth Sidewalk) released on 2015-03-09, useR! 2015 at the University of Aalborg, Denmark, June 30 - July 3, 2015, and useR! 2014 at the University of California, Los Angeles, USA June 30 - July 3, 2014.

You can see that there is a link, **download R**, and in the **News** section, the latest **R version is 3.2.2 (Fire Safety)**, which was released on **2015-08-14**. 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**:

The screenshot shows the "CRAN Mirrors" page. It starts with a note: "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](#)". Below this is a table of mirrors categorized by location:

Location	Mirror URL	Description
O-Cloud	https://cran.rstudio.com/ http://cran.rstudio.com/	RStudio, automatic redirection to servers worldwide
Algeria	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	http://cran.csiro.au/ http://cran.mss.unimelb.edu.au/	CSIRO University of Melbourne

Appendix

These are the links by country and sorted alphabetically that will take you to the download page. Being in Indiana, USA, I will scroll down and find that **Indiana University** has a link:

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
http://cran.mirorcatalogs.com/	Qarea Inc.
http://mirror.las.iastate.edu/CRAN/	Iowa State University, Ames, IA
http://ftp.usgs.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
http://watson.nci.nih.gov/cran_mirror/	National Cancer Institute, Bethesda, MD
https://cran.mtu.edu/	Michigan Technological University, Houghton, MI

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:

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.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2015-08-14, Fire Safety) [R-3.2.2.tar.gz](#), read [what's new](#) in the latest version.
- Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are [available here](#). Please read about [new features and bug fixes](#) before filing corresponding feature requests or bug reports.
- Source code of older versions of R is [available here](#).
- Contributed extension [packages](#)

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

Now, click on your appropriate operating system:

R for Windows

Subdirectories:

[base](#) Binaries for base distribution (managed by Duncan Murdoch). This is what you want to [install R for the first time](#).
[contrib](#) Binaries of contributed packages (managed by Uwe Ligges). There is also information on [third party software](#) available for CRAN Windows services and corresponding environment and make variables.
[Rtools](#) Tools to build R and R packages (managed by Duncan Murdoch). This is what you want to build your own packages on Windows, or to build R itself.

Please do not submit binaries to CRAN. Package developers might want to contact Duncan Murdoch or Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the [R FAQ](#) and [R for Windows FAQ](#).

Note: CRAN does some checks on these binaries for viruses, but cannot give guarantees. Use the normal precautions with downloaded executables.

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 that will initiate the download:

R-3.2.2 for Windows (32/64 bit)

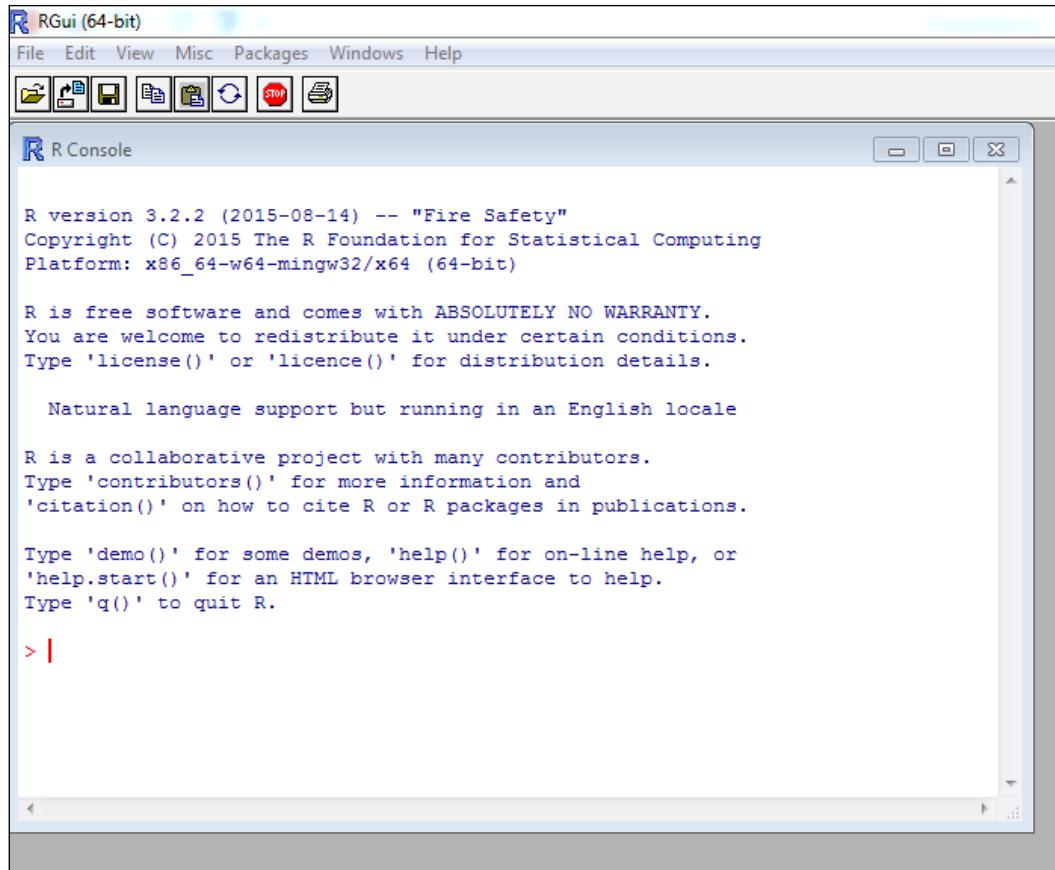
[Download R 3.2.2 for Windows](#) (62 megabytes, 32/64 bit)
[Installation and other instructions](#)
[New features in this version](#)

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?](#)

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:

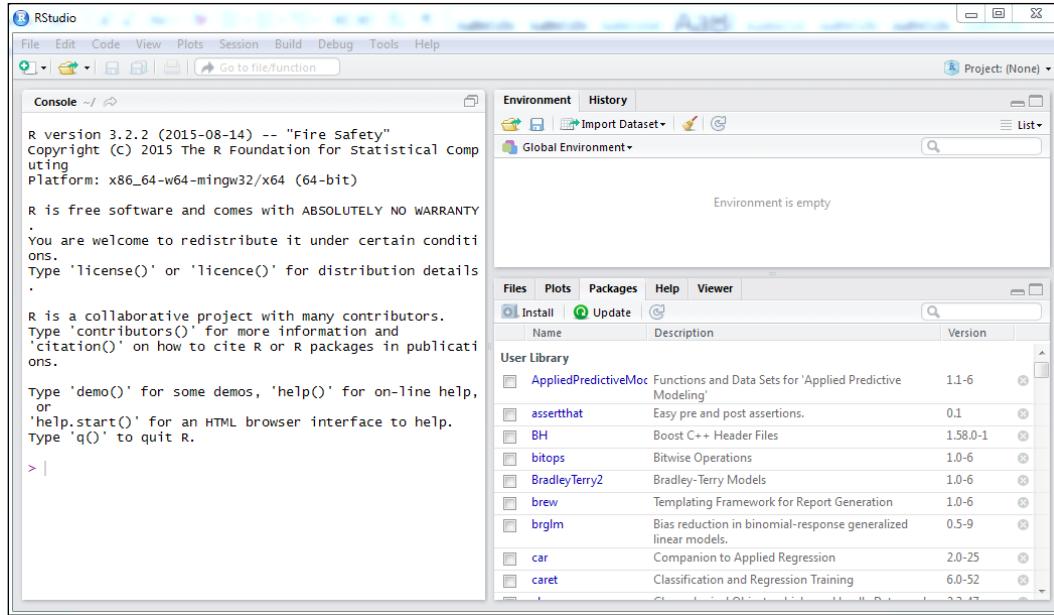
<https://www.rstudio.com/products/RStudio/>.

On this 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:

RStudio Desktop

	Open Source Edition	Commercial License
Overview	<ul style="list-style-type: none">• Access RStudio locally• Syntax highlighting, code completion, and smart indentation• Execute R code directly from the source editor• Quickly jump to function definitions• Easily manage multiple working directories using projects• Integrated R help and documentation• Interactive debugger to diagnose and fix errors quickly• Extensive package development tools	All of the features of open source; plus: <ul style="list-style-type: none">• A commercial license for organizations not able to use AGPL software• Access to priority support
Support	Community forums only	<ul style="list-style-type: none">• Priority Email Support• 8 hour response during business hours (ET)
License	AGPL v3	RStudio License Agreement
Pricing	Free	\$995/year
	DOWNLOAD RSTUDIO DESKTOP	BUY NOW

After this is installed and opened for the first time, you will see something as the following. Keep in mind that it will be different from what you see here based on the packages that I have loaded and the operating system:

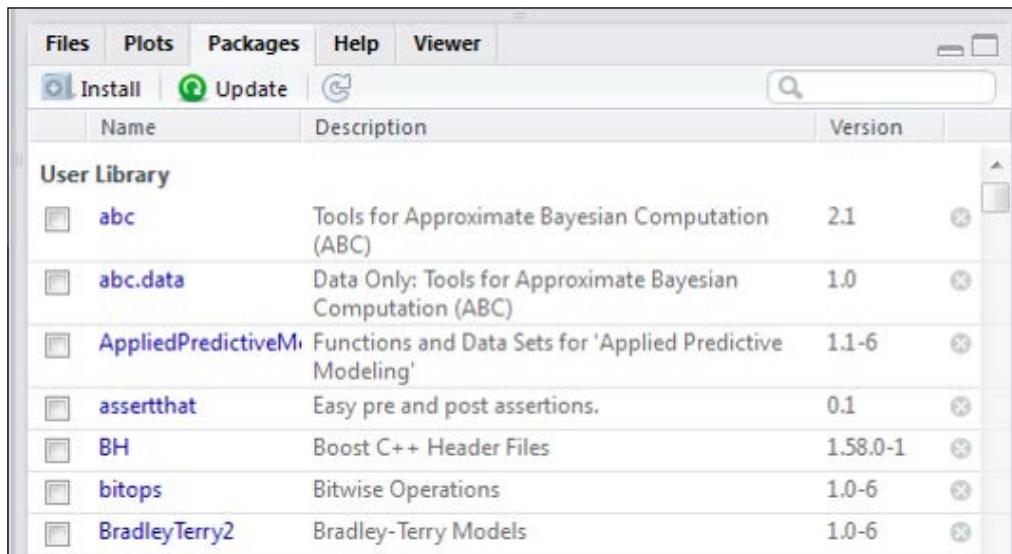


Note that on the left is the same console with the command prompt that you can see in the preceding figure. The IDE improves the experience such that you can manage **Environment** and **History** (to the upper right) and **Files**, **Plots**, **Packages**, and **Help** (to the lower right).

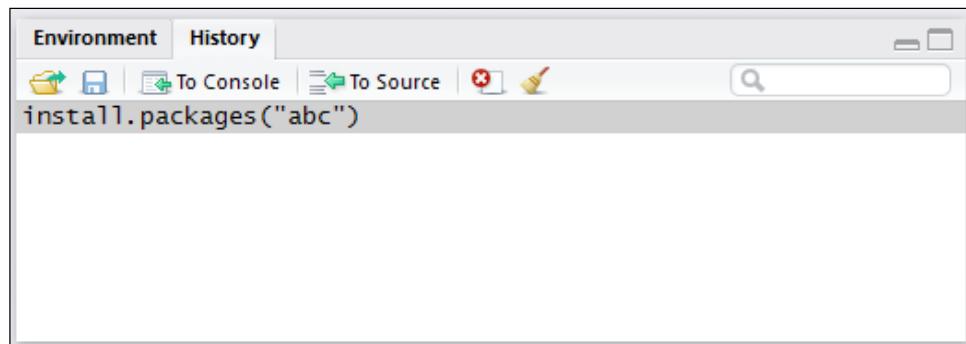
Let's not get distracted here with a full tutorial on what RStudio can do but focus on a couple of important items. One of the great benefits of R is the vast number of high quality packages to various analyses. Let's look at how the IDE ties it all together by loading a package called `abc`, which stands for approximate Bayesian computation. Go to the command prompt and type the following:

```
> install.packages("abc")
```

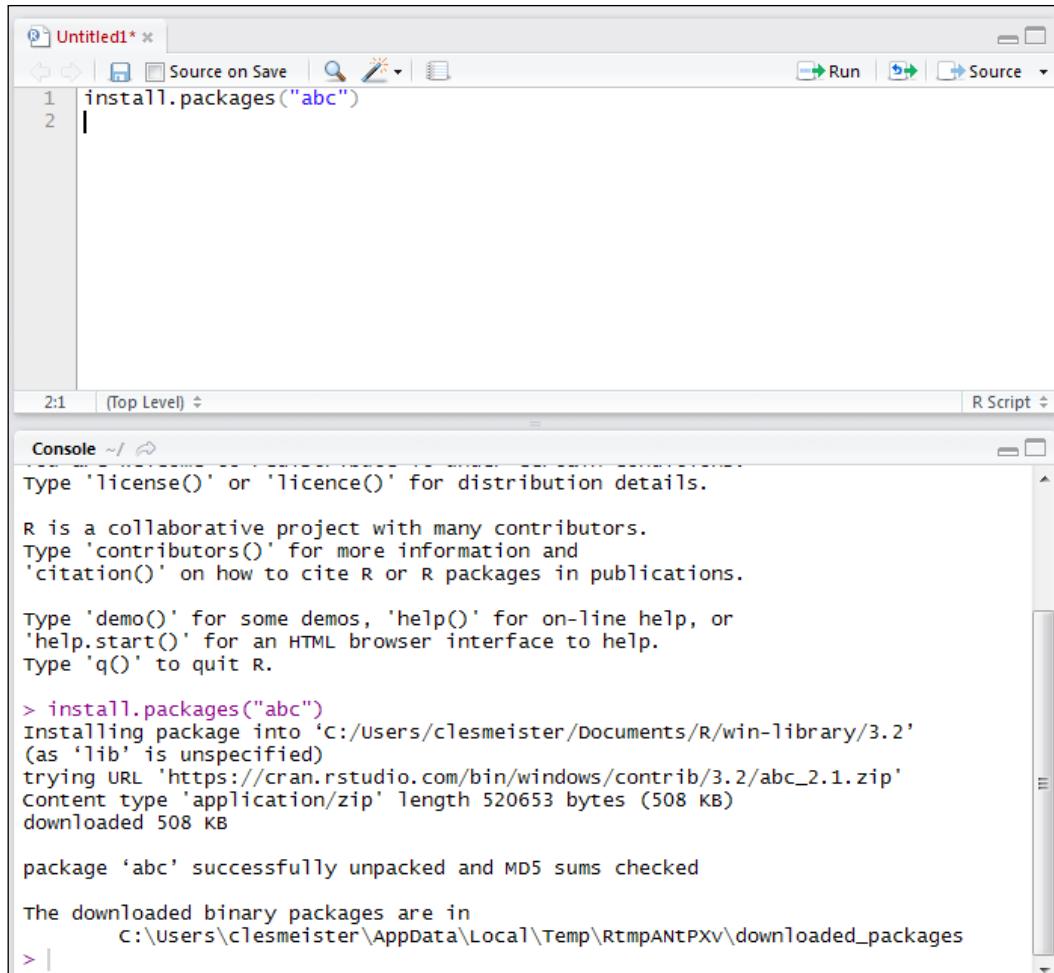
After this runs, notice that in the lower right panel (ensure that the **Packages** tab is clicked on) that the abc package is now installed as well as the dependent abc.data package:



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, it will be placed in front of the command prompt. If you click **To Source**, you will see a new area open that will allow you to put your project script together. If you click both the buttons, you will end up with something similar to this output:



The screenshot shows the RStudio interface. The top panel is a code editor titled "Untitled1" containing the R command `install.packages("abc")`. The bottom panel is a "Console" window displaying the output of running this command. The output includes standard R startup messages about distribution details and citation information, followed by the progress of the package download from CRAN, the extraction of the package, and the confirmation that the package was successfully unpacked and MD5 sums checked.

```
R install.packages("abc")
Installing package into 'C:/Users/clesmeister/Documents/R/win-library/3.2'
(as 'lib' is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.2/abc_2.1.zip'
Content type 'application/zip' length 520653 bytes (508 KB)
downloaded 508 KB

package 'abc' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
  C:\Users\clesmeister\AppData\Local\Temp\RtmpANTPXV\downloaded_packages
> |
```

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 are saved in a source file.

Using R

With all the systems ready to launch, let's start our first commands. R will take both the strings in the 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 ten 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 of the R code, you will see the assign symbol as `<-`, which is read as gets. Instead, I will use the `=` (equals) symbol. This may be computer science heresy, but I have not heard a convincing argument to dissuade me from my use of `=`. 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
```

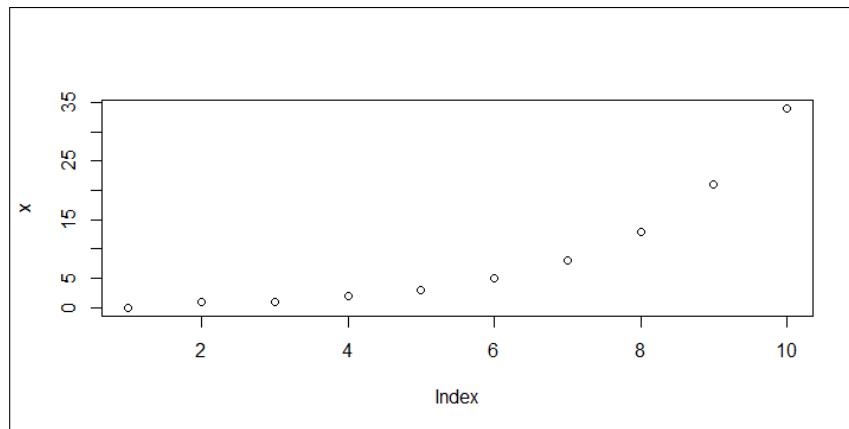
One can use a negative sign in the bracketed numbers in order to exclude the observations:

```
> x[-5:-6]  
[1] 0 1 1 2 8 13 21 34
```

To visualize this sequence, we will utilize the `plot()` function:

```
> plot(x)
```

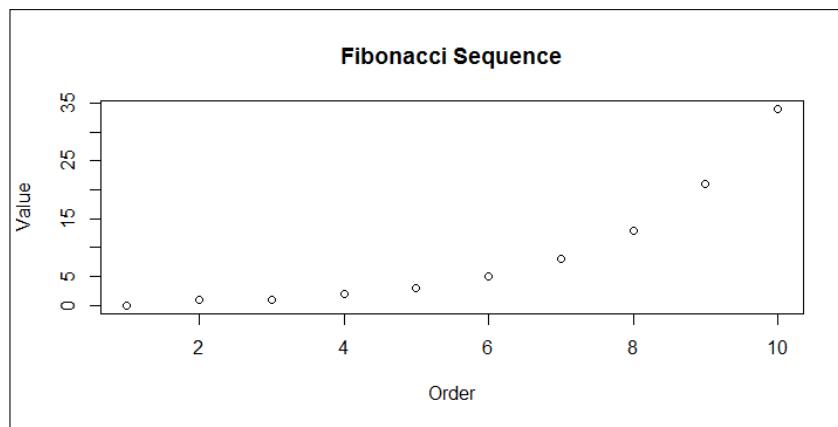
The output of the preceding command is as follows:



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
```

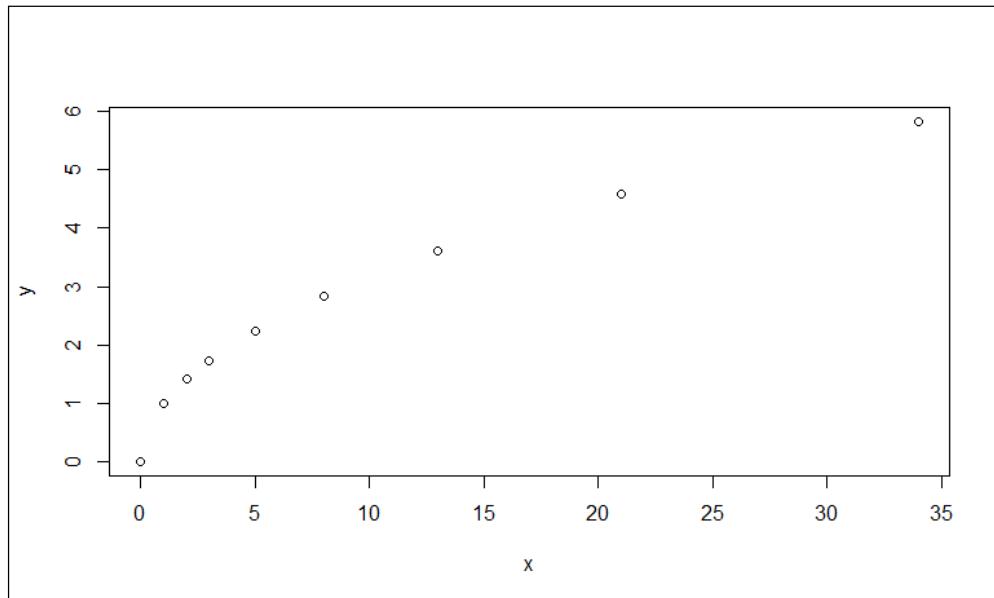
It is important here to point out that if you are unsure of what syntax can be used in a function, then using `?` in front of it will pull up help on the topic. Try this!

```
> ?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 if one value is less than another:

```
> 5 < 6  
[1] TRUE  
  
> 6 < 5  
[1] FALSE
```

In the first instance, R returned TRUE and in the latter, FALSE. If you want to find out if a value is equal to another value, then you would use two equal symbols. Remember, the equal symbol assigns a value. 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, \leq , $<$, == , $>$, \geq , and $!$, stand for less than or equal, less than, equal, greater than, greater than or equal, and not equal.

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", times=3)  
[1] "North Dakota Hockey" "North Dakota Hockey"  
[3] "North Dakota Hockey"
```

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, `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 `as.data.frame()` function. After that, examine the structure:

```
> s = as.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 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.

Summary stats

We will now cover some basic measures of central tendency, dispersion, and simple plots. The first question that we will address is how R handles the 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 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 the measures of 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
```

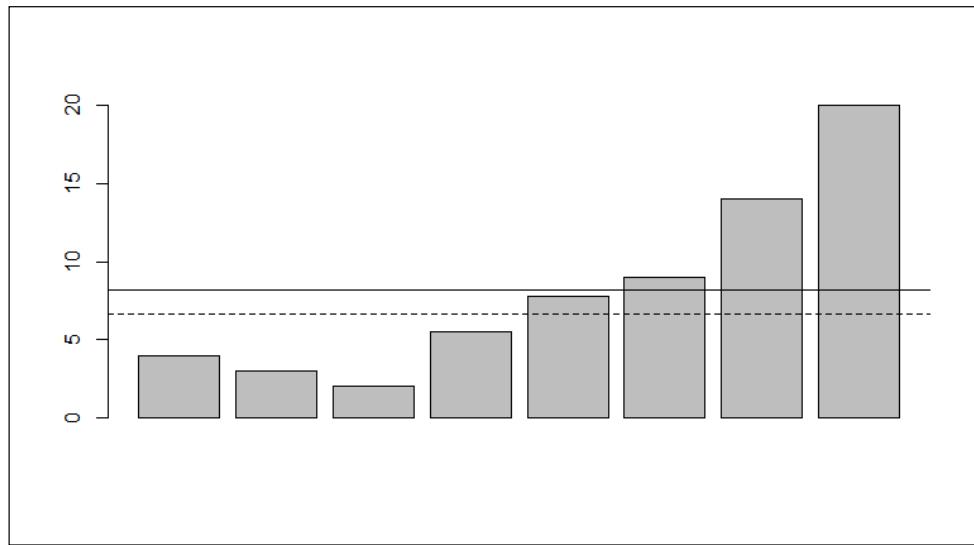
We can use plots to visualize the data. The base plot here will be `barplot`, then we will use `abline()` to include the mean and median. As the default line is solid, we will create a dotted line for `median` with `lty=2` to distinguish it from `mean`:

```
> 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 standard deviation of one 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()`:

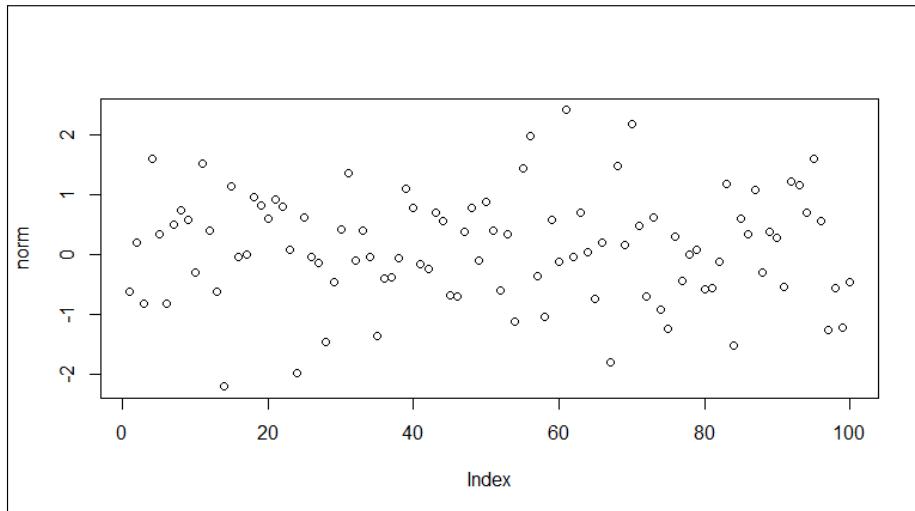
```
> set.seed(1)

> norm = rnorm(100)
```

This is the plot of the 100 data points:

```
> plot(norm)
```

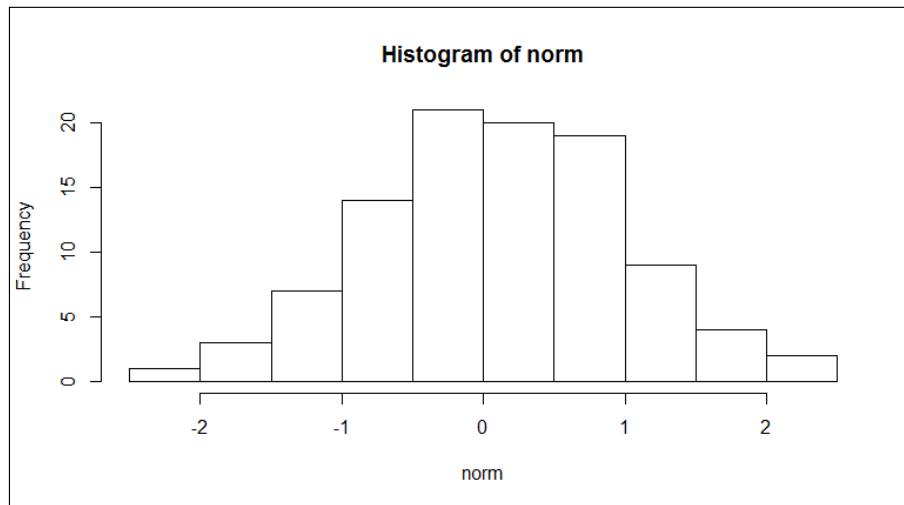
The output of the preceding command is as follows:



Finally, produce a histogram with `hist(norm)`:

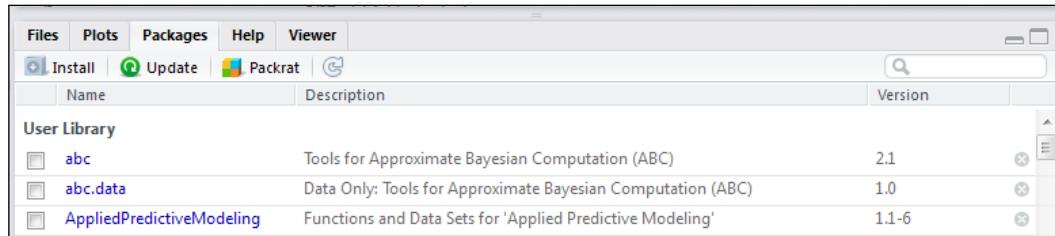
```
> hist(norm)
```

The following is the output of the preceding command:

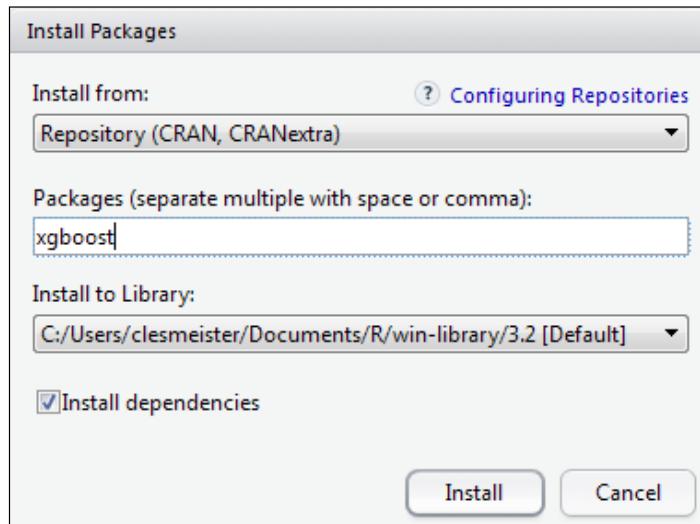


Installing and loading the R packages

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 in the package.

Summary

The purpose of this appendix was to allow the R novice 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. Finally, we covered how to install and load a package in R using RStudio. 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.

Index

A

- Aikake's Information Criterion (AIC)** 31, 285
- algorithm flowchart** 9-13
- a priori algorithm** 247
- Area Under the Curve (AUC)** 69
- Artificial Neural Networks (ANNs)**
 - about 166
 - reference link 166
- arules: Mining Association Rules and Frequent Itemsets** 247
- Augmented Dickey-Fuller (ADF) test** 293
- Autocorrelation Function (ACF)** 279
- Autoregressive Integrated Moving Average (ARIMA) models** 278

B

- Back Propagation** 166
- backward stepwise regression** 28
- bagging** 138
- Bayesian Information Criterion (BIC)** 31
- bias-variance trade-off** 64
- bivariate regression**
 - for univariate time series 283, 284
- bootstrap aggregation** 138
- Breusch-Pagan (BP)** 37
- business case, regularization**
 - about 78
 - business understanding 78
 - data, preparing 79-84
 - data, understanding 79-84
- business understanding, CRISP-DM process**
 - about 3
 - analytical goals, determining 5

business objective, identifying 4, 5
project plan, producing 5
situation, assessing 5

C

- caret package**
 - about 98
 - URL 98
- Change Agent** 1
- classification methods** 46
- classification models**
 - selecting 69-74
- classification trees**
 - business case 140
 - evaluation 144-147
 - modeling 144-147
 - overview 137, 138
- cluster analysis**
 - about 195
 - business understanding 200
 - data, preparing 201, 202
 - data, understanding 201, 202
 - with mixed data 217-219
- Cohen's Kappa statistic** 120
- collaborative filtering**
 - about 255
 - item-based collaborative filtering (IBCF) 257
 - principal components analysis (PCA) 257-262
 - singular value decomposition (SVD) 257-262
- user-based collaborative filtering**
 - (UBCF) 256

collinearity 15
Cook's distance 23
Corpus 320
Cosine Similarity 256
Cross Correlation Function (CCF) 292
Cross-Entropy 167
Cross-Industry Standard Process for Data Mining (CRISP-DM)
 about 1
 algorithm flowchart 9-13
 business understanding 3
 data preparation 6, 7
 data, understanding 6
 deployment 8, 9
 evaluation 8
 modeling 7
 process 2, 3
 URL 3
cross-validation
 for logistic regression 58-62
curse of dimensionality 221

D

data frame
 creating 358-360
data preparation process 6, 7
data understanding process 6
deep learning
 example 186
 H2O 187
 overview 170, 171
 reference link 171
deployment process 8, 9
dirichlet distribution 322
Discriminant Analysis (DA)
 application 64-68
 Linear Discriminant Analysis (LDA) 62
 overview 62-64
 Quadratic Discriminant Analysis (QDA) 62
Document-Term Matrix (DTM) 321
dynamic topic modelling 323

E

ECLAT algorithms 247
eigenvalues 224

eigenvectors 224
elastic net
 about 78
 using 98-101
equimax 225
Euclidian Distance 107
evaluation process 8
exponential smoothing models 278
Extract, Transport, and Load (ETL) 6

F

False Positive Rate (FPR) 69
Feed Forward network 166
Final Prediction Error (FPE) 285
Fine Needle Aspiration (FNA) 47
first principal component 223
Fisher Discriminant Analysis (FDA). *See Discriminant Analysis (DA)*
F-Measure 324
forward stepwise selection 28

G

Gedeon Method 193
glmnet package
 used, for performing cross-validation
 for regularization 101-103
Gower 199
Gower-based metric dissimilarity matrix 196
gradient boosted trees 136
gradient boosting
 business case 140
 model selection 163
 overview 139
 reference link 139
gradient boosting classification
 evaluation 159-163
 modeling 159-163
gradient boosting regression
 evaluation 156-158
 modeling 156-158
Granger causality 284, 285
Graphical User Interface (GUI) 348

H

H2O

about 187
data, preparing 187-189
data, uploading 187-189
modeling 191-194
test dataset, creating 191
train dataset, creating 191
URL 187

Hannan-Quinn Criterion (HQ) 311

Hat Matrix 40

heatmaps 26

heteroscedasticity 22

hierarchical clustering

about 196, 197
distance calculations 197
evaluation 203-214
modeling 203-214

Holt-Winter's Method 278

I

Integrated Development Environment (IDE) 345

interquartile range 214
item-based collaborative filtering (IBCF) 257

K

kernel trick 109

K-fold cross-validation 58

k-means clustering

about 196-198
evaluation 214-216
modeling 214-216

K-Nearest Neighbors (KNN)

about 105-107
business understanding 111
case study 111
data, preparing 112-118
data, understanding 112-118
modeling 118-123

KNN modeling

versus SVM modeling 128-130

K-sets 58

L

L1-norm 77

L2-norm 77

LASSO

about 77
executing 95-97

Latent Dirichlet Allocation (LDA) 322

lazy learning 106

Leave-One-Out-Cross-Validation (LOOCV) 39, 58

Linear Discriminant Analysis (QDA) 62

linear model

considerations 40
interaction term 43
qualitative feature 41, 42

linear regression 15, 46

linear regression model

homoscedasticity 22
linearity 22
no collinearity 22
non-correlation of errors 22
presence of outliers 22

logistic regression

about 46, 47
business understanding 47, 48
data, preparing 48-54
data, understanding 48-54
Discriminant Analysis (DA) 62-64
evaluation 54
modeling 54
with cross-validation 58-62

logistic regression model 54-58

loss function 139

M

Mallow's Cp (Cp) 31

margin 108

market basket analysis

about 246, 247
business understanding 247
data, preparing 248, 249
data, understanding 248, 249
evaluation 250-254
modeling 250-254

matrices

creating 358-360

mean squared error (MSE) 89
medoid 200
modeling process 7
multivariate linear regression
 about 25
 business understanding 25
 data, preparing 25-27
 data, understanding 25-27
 evaluation 28-40
 modeling 28-40

N

neural network
 about 166-169
 business understanding 172
 data, preparing 173-179
 data, understanding 173-179
 evaluation 179-186
 modeling 179-186
 reference link 173
Normal Q-Q plot 24

O

OPRC 34
OPSLAKE 34
Ordinary Least Squares (OLS) 45
out-of-bag (oob) 138

P

Partial Autocorrelation Function (PACF) 280
Partitioning Around Medoids (PAM) 196-200
Pearson Correlation Coefficient 256
Polarity 323
Porter stemming algorithm 321
Prediction Error Sum of Squares (PRESS) 39
principal components
 overview 222-224
 rotation 225, 226
Principal Components Analysis (PCA)
 about 199-221, 257-262
 business understanding 226
 component, extraction 233-235

data, preparing 227-232
data, understanding 227-232
evaluation 233
factor scores, creating from
 components 237-239
interpretation 236, 237
modeling 233
orthogonal rotation 236, 237
regression analysis 239-243

Q

Quadratic Discriminant Analysis (QDA) 62
Quantile-Quantile (Q-Q) 23
quartimax 225

R

R
 installing 345-353
 running 345-353
 URL 346
 using 354-357

radical 321

random forest
 about 136
 business case 140
 model selection 163
 overview 138

random forest classification
 evaluation 151-156
 modeling 151-156

random forest regression
 evaluation 147-151
 modeling 147-151

Receiver Operating Characteristic (ROC)
 about 69
 reference link 69

recommendation engine
 business understanding 262
 collaborative filtering 255
 data, preparing 262-265
 data, understanding 262-265
 evaluation 265-276
 modeling 265-276
 overview 255
 recommendations 265-276

recommenderlab library

URL 262

regression trees

business case 140
evaluation 140-144
modeling 140-144
overview 136

regularization

about 76
business case 78
cross-validation, performing with
 glmnet package 101-103
elastic net 78
evaluation 85
LASSO 77
modeling 85
model selection 103, 104
ridge regression 77

regularization, modeling

best subsets, creating 85-89
elastic net, using 98-101
LASSO, running 95-97
ridge regression 90-94

Residual Sum of Squares (RSS) 16**Restricted Boltzmann Machine 171****ridge regression**

about 77
executing 90-94

Root Mean Square Error (RMSE) 99**R packages**

installing 364
loading 364

RStudio

URL 349

S**Schwarz-Bayes Criterion (SC) 311****second principal component 223****shrinkage penalty 76****singular value decomposition
(SVD) 257-262****slack variables 109****Sparse Coding Model 171****summary stats**

displaying 360-363

sum of squared error (SSE) 278**supervised learning 195****Support Vector Machines (SVM)**

about 105-111
business understanding 111
case study 111
data, preparing 112-118
data, understanding 112-118
feature selection 132, 133
modeling 124-127

suspected outliers 214**SVM modeling**

versus KNN modeling 128-130

T**Term-Document Matrix (TDM) 321****text mining**

business understanding 325
data, preparing 325-330
data, understanding 325-330
evaluation 330
methods 320, 321
modeling 330
other quantitative analyses 323, 324
quantitative analysis, with
 qdap package 337-343
topic models 322, 323
topic models, building 330-337
word frequency, exploring 330-337

tree-based learning 139**True Positive Rate (TPR) 69****U****univariate linear regression**

about 16-18
business understanding 18-25

univariate time series

about 277
analyzing 278-283
analyzing, with Granger causality 284, 285
business understanding 286-289
data, preparing 289-293
data, understanding 289-293
evaluation 293
examining, with regression 302-310
forecasting 294-302

Granger causality, examining 310-317
modeling 293
with bivariate regression 283, 284
unsupervised learning 195
user-based collaborative filtering (UBCF) 256

V

valence shifters 323
Variance Inflation Factor (VIF) 34
varimax 225
Vector Autoregression (VAR) 285

W

whiskers 214