Community Experience Distilled

Learning Predictive Analytics with R

Get to grips with key data visualization and predictive analytic skills using R

Eric Mayor

[PACKT] open source*
community experience distilled
PUBLISHING

Learning Predictive Analytics with R

Get to grips with key data visualization and predictive analytic skills using R

Eric Mayor

[PACKT] open source
community experience distilled
PUBLISHING
BIRMINGHAM - MUMBAI

Learning Predictive Analytics with R

First published: September 2015

Production reference: 1180915

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-935-2

www.packtpub.com

Credits

Author
Eric Mayor

Reviewers
Ajay Dhamija
Khaled Tannir
Matt Wiley

Commissioning Editor
Kunal Parikh

Acquisition Editor
Kevin Colaco

Content Development Editor
Siddhesh Salvi

Technical Editor
Deepti Tuscano

Copy Editors
Puja Lalwani
Merilyn Pereira

Project Coordinator
Kranti Berde

Proofreader
Safis Editing

Indexer
Rekha Nair

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Eric Mayor is a senior researcher and lecturer at the University of Neuchatel, Switzerland. He is an enthusiastic user of open source and proprietary predictive analytics software packages, such as R, Rapidminer, and Weka. He analyzes data on a daily basis and is keen to share his knowledge in a simple way.

About the Reviewers

Ajay Dhamija is a senior scientist in the Defence Research and Development Organization, Delhi. He has more than 24 years of experience as a researcher and an instructor. He holds an MTech (in computer science and engineering) from IIT Delhi and an MBA (in finance and strategy) from FMS, Delhi. He has to his credit more than 14 research works of international reputation in various fields, including data mining, reverse engineering, analytics, neural network simulation, TRIZ, and so on. He was instrumental in developing a state-of-the-art Computerized Pilot Selection System (CPSS) containing various cognitive and psycho-motor tests. It is used to comprehensively assess the flying aptitude of aspiring pilots of the Indian Air Force. Ajay was honored with an Agni Award for excellence in self reliance in 2005 by the Government of India. He specializes in predictive analytics, information security, big data analytics, machine learning, Bayesian social networks, financial modeling, neuro-fuzzy simulation, and data analysis and data mining using R. He is presently involved in his doctoral work on *Financial Modeling of Carbon Finance Data* from IIT, Delhi. He has written an international bestseller, *Forecasting Exchange Rate: Use of Neural Networks in Quantitative Finance* (`http://www.amazon.com/Forecasting-Exchange-rate-Networks-Quantitative/dp/3639161807`), and is currently authoring another book in R named *Multivariate Analysis using R*.

Apart from Analytics, Ajay Dhamija is actively involved in information security research. He has been associated with various international and national researchers in the government as well as the corporate sector to pursue his research on ways to amalgamate two important and contemporary fields of data handling, that is, predictive analytics and information security. While he was associated with researchers from the Predictive Analytics and Information Security Institute of India (PRAISIA: `www.praisia.com`) during his research endeavors, he worked on refining methods of big data analytics for security data analysis (log assessment, incident analysis, threat prediction, and so on) and vulnerability management automation.

You can connect with Ajay at:

- LinkedIn: `ajaykumardhamija`
- ResearchGate: `Ajay_Dhamija2`
- Academia: `ajaydhamija`
- Facebook: `akdhamija`
- Twitter:`@akdhamija`
- Quora: `Ajay-Dhamija`

I would like to thank my fellow scientists from the Defense Research and Development Organization and researchers from the corporate sector, including Predictive Analytics and Information Security Institute of India (PRAISIA). It is a unique institute of repute and of due to its pioneering work in marrying the two giant and contemporary fields of data handling in modern times, that is, predictive analytics and information security, by adopting bespoke and refined methods of big data analytics. They all contributed in presenting a fruitful review for this book. I'm also thankful to my wife, Seema Dhamija, the managing director at PRAISIA, who has been kind enough to share her research team's time with me in order to have a technical discussion. I'm also thankful to my son, Hemant Dhamija. Many a time, he gave invaluable feedback that I inadvertently neglected during the course of this review. I'm also thankful to a budding security researcher, Shubham Mittal from Makemytrip Inc., for his constant and constructive critiques of my work.

Matt Wiley is a tenured associate professor of mathematics who currently resides in Victoria, Texas. He holds degrees in mathematics (with a computer science minor) from the University of California and a master's degree in business administration from Texas A&M University. He directs the Quality Enhancement Plan at Victoria College and is the managing partner at Elkhart Group Limited, a statistical consultancy. With programming experience in R, C++, Ruby, Fortran, and JavaScript, he has always found ways to meld his passion for writing with his love of logical problem solving and data science. From the boardroom to the classroom, Matt enjoys finding dynamic ways to partner with interdisciplinary and diverse teams to make complex ideas and projects understandable and solvable.

Matt can be found online at `www.MattWiley.org`.

Khaled Tannir is a visionary solution architect with more than 20 years of technical experience focusing on big data technologies and data mining since 2010.

He is widely recognized as an expert in these fields and has a bachelor's degree in electronics and a master's degree in system information architectures. He completed his education with a master of research degree.

Khaled is a Microsoft Certified Solution Developer (MCSD) and an avid technologist. He has worked for many companies in France (and recently in Canada), leading the development and implementation of software solutions and giving technical presentations.

He is the author of the books *RavenDB 2.x Beginner's Guide* and *Optimizing Hadoop MapReduce, Packt Publishing*, and a technical reviewer on the books *Pentaho Analytics for MongoDB* and *MongoDB High Availability*, *Packt Publishing*.

He enjoys taking landscape and night photos, traveling, playing video games, creating funny electronics gadgets using Arduino, Raspberry Pi, and .Net Gadgeteer, and of course spending time with his wife and family.

You can reach him at `contact@khaledtannir.net`.

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

The amount of data in the world is increasing exponentially as time passes. It is estimated that the total amount of data produced in 2020 will be 20 zettabytes (Kotov, 2014), that is, 20 billion terabytes. Organizations spend a lot of effort and money on collecting and storing data, and still, most of it is not analyzed at all, or not analyzed properly. One reason to analyze data is to predict the future, that is, to produce actionable knowledge. The main purpose of this book is to show you how to do that with reasonably simple algorithms. The book is composed of chapters describing the algorithms and their use and of an appendices with exercises and solutions to the exercises and references.

Prediction

What is meant by prediction? The answer, of course, depends on the field and the algorithms used, but this explanation is true most of the time—given the attested reliable relationships between indicators (predictors) and an outcome, the presence (or level) of the indicators for similar cases is a reliable clue to the presence (or level) of the outcome in the future. Here are some examples of relationships, starting with the most obvious:

- Taller people weigh more
- Richer individuals spend more
- More intelligent individuals earn more
- Customers in segment X buy more of product Y
- Customers who bought product P will also buy product Q
- Products P and Q are bought together
- Some credit card transactions predict fraud (Chan et al., 1999)
- Google search queries predict influenza infections (Ginsberg et al., 2009)
- Tweet content predicts election poll outcomes (O'Connor and Balasubramanyan, 2010)

In the following section, we provide minimal definitions of the distinctions between supervised and unsupervised learning and classification and regression problems.

Supervised and unsupervised learning

Two broad families of algorithms will be discussed in this book:

- Unsupervised learning algorithms
- Supervised learning algorithms

Unsupervised learning

In unsupervised learning, the algorithm will seek to find the structure that organizes unlabelled data. For instance, based on similarities or distances between observations, an unsupervised cluster analysis will determine groups and which observations fit best into each of the groups. An application of this is, for instance, document classification.

Supervised learning

In supervised learning, we know the class or the level of some observations of a given target attribute. When performing a prediction, we use known relationships in labeled data (data for which we know what the class or level of the target attribute is) to predict the class or the level of the attribute in new cases (of which we do not know the value).

Classification and regression problems

There are basically two types of problems that predictive modeling deals with:

- Classification problems
- Regression problems

Classification

In some cases, we want to predict which group an observation is part of. Here, we are dealing with a quality of the observation. This is a classification problem. Examples include:

- The prediction of the species of plants based on morphological measurements

- The prediction of whether individuals will develop a disease or not, based on their health habits
- The prediction of whether an e-mail is spam or not

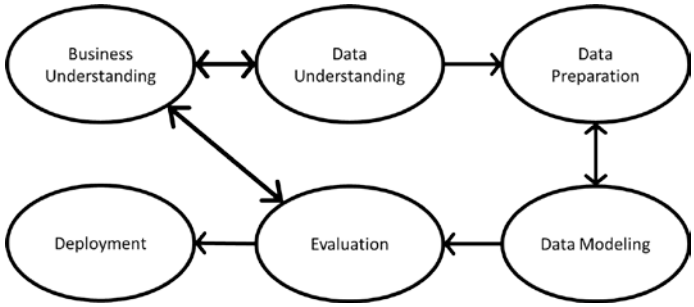
Regression

In other cases, we want to predict an observation's level on an attribute. Here, we are dealing with a quantity, and this is a regression problem. Examples include:

- The prediction of how much individuals will cost to health care based on their health habits
- The prediction of the weight of animals based on their diets
- The prediction of the number of defective devices based on manufacturing specifications

The role of field knowledge in data modeling

Of course, analyzing data without knowledge of the field is not a serious way to proceed. This is okay to show how some algorithms work, how to make use of them, and to exercise. However, for real-life applications, be sure that you know the topic well, or else consult experts for help. The Cross Industry Standard Process for Data Mining (CRISP-DM, Shearer, 2000) underlines the importance of field knowledge. The steps of the process are depicted as follows:

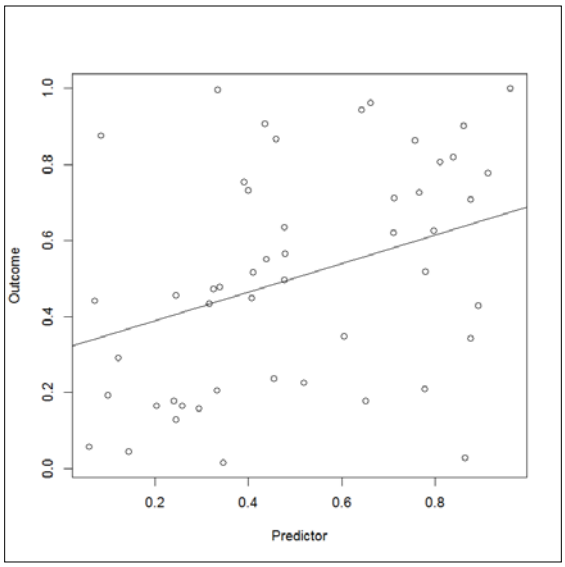


The Cross Industry Standard Process for Data Mining

As stressed upon in the preceding diagram, field knowledge (here called Business Understanding) informs and is informed by data understanding. The understanding of the data then informs how the data has to be prepared. The next step is data modeling, which can also lead to further data preparation. Data models have to be evaluated, and this evaluation can be informed by field knowledge (this is also stressed upon in the diagram), which is also updated through the data mining process. Finally, if the evaluation is satisfactory, the models are deployed for prediction. This book will focus on the data modeling and evaluation stages.

Caveats

Of course, predictions are not always accurate, and some have written about the caveats of data science. What do you think about the relationship between the attributes titled Predictor and Outcome on the following plot? It seems like there is a relationship between the two. For the statistically inclined, I tested its significance: $r = 0.4195$, $p = .0024$. The value p is the probability of obtaining a relationship of this strength or stronger if there is actually no relationship between the attributes. We could conclude that the relationship between these variables in the population they come from is quite reliable, right?



The relationship between the attributes titled Predictor and Outcome

Believe it or not, the population these observations come from is that of randomly generated numbers. We generated a data frame of 50 columns of 50 randomly generated numbers. We then examined all the correlations (manually) and generated a scatterplot of the two attributes with the largest correlation we found. The code is provided here, in case you want to check it yourself—line 1 sets the seed so that you find the same results as we did, line 2 generates to the data frame, line 3 fills it with random numbers, column by column, line 4 generates the scatterplot, line 5 fits the regression line, and line 6 tests the significance of the correlation:

```
1  set.seed(1)
2  DF = data.frame(matrix(nrow=50,ncol=50))
3  for (i in 1:50) DF[,i] = runif(50)
4  plot(DF[[2]],DF[[16]], xlab = "Predictor", ylab = "Outcome")
5  abline(lm(DF[[2]]~DF[[16]]))
6  cor.test(DF[[2]], DF[[16]])
```

How could this relationship happen given that the odds were 2.4 in 1000 ? Well, think of it; we correlated all 50 attributes 2 x 2, which resulted in 2,450 tests (not considering the correlation of each attribute with itself). Such spurious correlation was quite expectable. The usual threshold below which we consider a relationship significant is $p = 0.05$, as we will discuss in *Chapter 8*, *Probability Distributions, Covariance, and Correlation*. This means that we expect to be wrong once in 20 times. You would be right to suspect that there are other significant correlations in the generated data frame (there should be approximately 125 of them in total). This is the reason why we should always correct the number of tests. In our example, as we performed 2,450 tests, our threshold for significance should be *0.0000204 (0.05 / 2450)*. This is called the Bonferroni correction.

Spurious correlations are always a possibility in data analysis and this should be kept in mind at all times. A related concept is that of overfitting. Overfitting happens, for instance, when a weak classifier bases its prediction on the noise in data. We will discuss overfitting in the book, particularly when discussing cross-validation in *Chapter 14*, *Cross-validation and Bootstrapping Using Caret and Exporting Predictive Models Using PMML*. All the chapters are listed in the following section.

We hope you enjoy reading the book and hope you learn a lot from us!

What this book covers

Chapter 1, *Setting GNU R for Predictive Analytics*, deals with the setting of R, how to load and install packages, and other basic operations. Only beginners should read this. If you are not a beginner, you will be bored! (Beginners should find the chapter entertaining).

Chapter 2, *Visualizing and Manipulating Data Using R*, deals with basic visualization functions in R and data manipulation. This chapter also aims to bring beginners up to speed for the rest of the book.

Chapter 3, *Data Visualization with Lattice*, deals with more advanced visualization functions. The concept of multipanel conditioning plots is presented. These allow you to examine the relationship between attributes as a function of group membership (for example, women versus men). A good working knowledge of R programming is necessary from this point.

Chapter 4, *Cluster Analysis*, presents the concept of clustering and the different types of clustering algorithms. It shows how to program and use a basic clustering algorithm (k-means) in R. Special attention is given to the description of distance measures and how to select the number of clusters for the analyses.

Chapter 5, *Agglomerative Clustering Using hclust()*, deals with hierarchical clustering. It shows how to use agglomerative clustering in R and the options to configure the analysis.

Chapter 6, *Dimensionality Reduction with Principal Component Analysis*, discusses the uses of PCA, notably dimension reduction. How to build a simple PCA algorithm, how to use PCA, and example applications are explored in the chapter.

Chapter 7, *Exploring Association Rules with Apriori*, focuses on the functioning of the apriori algorithm, how to perform the analyses, and how to interpret the outputs. Among other applications, association rules can be used to discover which products are frequently bought together (marked basket analysis).

Chapter 8, *Probability Distributions, Covariance, and Correlation*, discusses basic statistics and how they can be useful for prediction. The concepts given in the title are discussed without too much technicality, but formulas are proposed for the mathematically inclined.

Chapter 9, *Linear Regression*, builds upon the knowledge acquired in the previous chapter to show how to build a regression algorithm, including how to compute the coefficients and p values. The assumptions of linear regression (ordinary least squares) are rapidly discussed. The chapter then focuses on the use (and misuse) of regression.

Chapter 10, *Classification with k-Nearest Neighbors and Naïve Bayes*, deals with the classification problems of using two of the most popular algorithms. We build our own k-NN algorithm, with which we analyze the famous iris dataset. We also demonstrate how Naïve Bayes works. The chapter also deals with the use of both algorithms.

Chapter 11, *Classification Trees*, explores classification using no less than five classification tree algorithms: C4.5, C5, CART (classification part), random forests, and conditional inference trees. Entropy, information gain, pruning, bagging, and other important concepts are discussed.

Chapter 12, *Multilevel Analyses*, deals with the use of nested data. We will briefly discuss the functioning of multilevel regression (with mixed models), and will then focus on the important aspects in the analysis, notably, how to create and compare the models, and how to understand the outputs.

Chapter 13, *Text Analytics with R*, focuses on the use of some algorithms that we discussed in other chapters, as well as new ones, with the aim of analyzing text. We will start by showing you how to perform text preprocessing, we will explain important concepts, and then jump right into the analysis. We will highlight the importance of testing different algorithms on the same corpus.

Chapter 14, *Cross-validation and Bootstrapping Using Caret and Exporting Predictive Models Using PMML*, deals with two important aspects, the first is ascertaining the validity of the models and the second is exporting the models for production. Training and testing datasets are used in most chapters. These are minimal requirements, and cross-validation as well as bootstrapping are significant improvements.

Appendix A, *Exercises and Solutions*, provides the exercises and the solutions for the chapters in the book.

Appendix B, *Further Reading and References*, it provides the references for the chapters in the book.

What you need for this book

All you need for this book is a working installation of R > 3.0 (on any operating system) and an active internet connection.

Who this book is for

If you are a statistician, chief information officer, data scientist, ML engineer, ML practitioner, quantitative analyst, or student of machine learning, this is the book for you. You should have basic knowledge of the use of R. Readers without previous experience of programming in R will also be able to use the tools in this book.

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: " Now, open the R script file called `helloworld.R`."

A block of code is set as follows:

```
print("Hello world")
```

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: " The **File** menu contains functions related to file handling."

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 `http://www.packtpub.com/sites/default/files/downloads/93520S_ColorImages.pdf`.

Errata

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

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

Piracy

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

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

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

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
Setting GNU R for Predictive Analytics

R is a relatively recent multi-purpose statistical language that originates from the older language S. R contains a core set of packages that includes some of the most common statistical tests and some data mining algorithms. One of the most important strengths of R is the degree to which its functionalities can be extended by installing packages made by users from the community. These packages can be installed directly from R, thereby making the process very comfortable. The **Comprehensive R Archive Network** (**CRAN**), which is available at `http://cran.r-project.org`, is a repository of packages, R sources, and R binaries (installers). It also contains the manuals for the packages. There are currently more than 4,500 available packages for R, and more are coming up regularly. Further, what is also great is that everything is free.

The topics covered in this chapter are:

- Installation of R
- R graphic user interface, including a description of the different menus
- Definition of packages and how to install and load them
- Along the way we will also discover parts of the syntax of R

Among almost 50 competitors, R is the most widely used tool for predictive modeling, together with RapidMiner, according to yearly software polls from KDnuggets (most recently available at `http://www.kdnuggets.com/2015/05/poll-r-rapidminer-python-big-data-spark.html`). Its broad use and the extent to which it is extendable make it an essential software package for data scientists. Competitors notably include Python, Weka, and Knime.

This book is intended for people who are familiar with R. This doesn't mean that people who do not have such a background cannot learn predictive analytics by using this book. It just means that they will require more time to use this book effectively, and might need to consult the basic R documentation along the way. With this extended readership in mind, we will just cover a few of the basics in this chapter while we set up R for predictive analytics. The writing style will be as accessible as possible. If you have trouble following through the first chapter, we suggest you first read a book on R basics before pursuing the following chapters, because the effort you will need to invest to understand and practice the content of this book will keep increasing from *Chapter 2*, *Visualizing and Manipulating Data Using R*. Unlike other chapters, this chapter explains basic information. Users who are more familiar with R are invited to skip to *Chapter 2*, *Visualizing and Manipulating Data Using R* or *Chapter 3*, *Data Visualization with Lattice*.

Installing GNU R

If this is not yet done, download the installer for your operating system on CRAN. Launch the installer and follow the specific instructions for your operating system. We will not examine these here as they are straightforward; just follow the instructions on screen. The following pages offer a quick reminder or a basic introduction to the interface in R. Here are the addresses where you can find the installers for each OS:

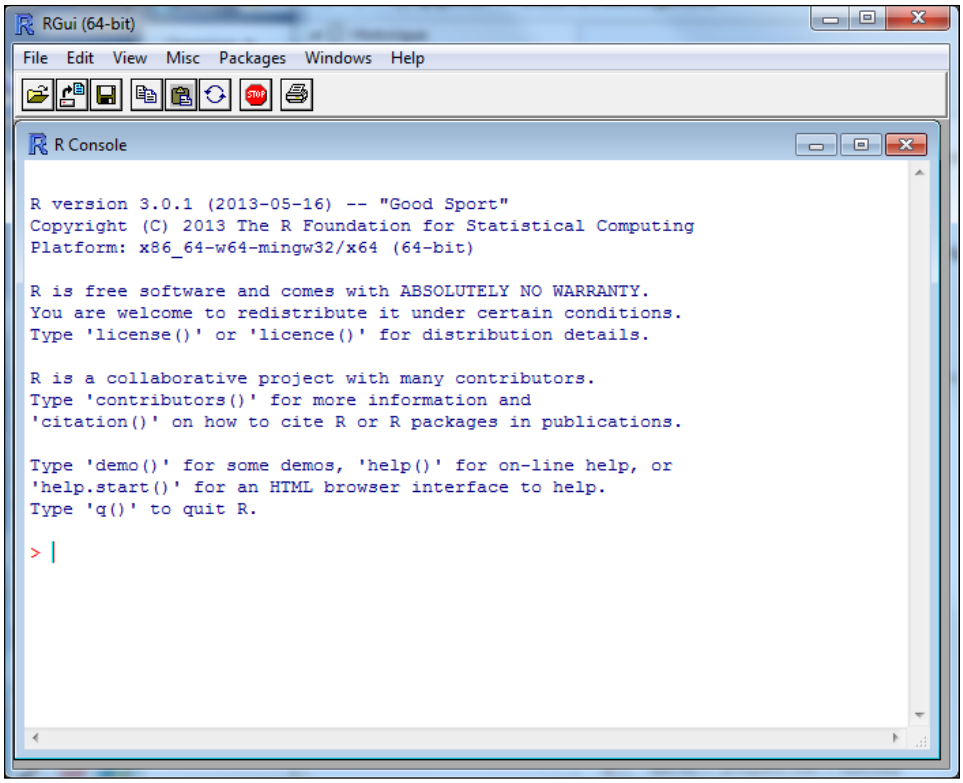
- For Windows: `http://cran.r-project.org/bin/windows/`
- For Mac OS X: `http://cran.r-project.org/bin/macosx/`
- For Linux: `http://cran.r-project.org/bin/linux/`

These links also serve as pointers to R under MacOS X and Linux, which are not fully described here.

The R graphic user interface

The following snapshot represents the default window when starting R. The default window is highly similar across platforms, which is why it is not necessary to display all screenshots here. More importantly, most of what is covered will apply to any recent build of R. Advanced readers might be interested in using a more sophisticated development tool such as RStudio available at `http://www.rstudio.com/`. Because of space limitations, we will not describe it here.

The encompassing window displayed in the picture below, **R graphic user interface** (**RGui)**, contains a basic graphic user interface. You can see the menu bar on the top of the window. We will look at some of its elements more closely in the following screenshot:



A snapshot of the RGUI window

The menu bar of the R console

When the R console window is active, there are seven accessible menus: **File**, **Edit**, **View**, **Misc**, **Packages**, **Windows**, and **Help**. If you use a platform other than Windows 7, you might notice some differences, but none are important.

Some functions of the **File** and **Misc** menus are worth commenting upon briefly. Functions from the **Packages** menu will be commented upon in the next section. Function is a term that can loosely relate to something the program does, or more specifically, a succession of steps programmatically defined, oftentimes involving an algorithm, and explicitly called by some piece of code. When discussing functions accessed through a menu, we will indicate the name of the menu item. When discussing functions as they appear in code, we will indicate the function name followed by brackets (). Sometimes, a function selectable from the menu corresponds to a single function in code; other times, several lines of code are necessary to accomplish the same thing as the menu function through code.

A quick look at the File menu

The **File** menu contains functions related to file handling. Some useful functions of the **File** menu are as follows:

- **Source R code**: Opens a dialogue box from which an R script can be selected. This script will be run in the console.
- **New script**: Opens a new window of the R editor, in which R code can be typed or pasted. When this window is active, the menu bar changes.
- **Open script**: Opens a dialogue box from which an R script can be selected. This script will be loaded in a new window of the R editor.
- **Change dir**: Opens a dialogue window where a folder can be selected. This folder will become the working folder for the current session (until changed).

Here are some quick exercises that will help you get acquainted with the **File** menu. Before this, make sure that you have downloaded and extracted the code for this book from its webpage.

Let's start by changing the working folder to the folder where you extracted this book's code. This can be done using the **Change dir** function. Simply click on it in the **File** menu and select the folder you wish to use.

Now, open the R script file called `helloworld.R`; this can be done using the **Source R code** function. The file should be listed in the dialogue box. If this is not the case, start by selecting the folder containing the R code again. The file contains the following code:

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.

```
print("Hello world")
```

This line of code calls the `print()` function with the argument `"Hello world"`.

Experiment running the first line of R code: select the content of the file, right click on it, and click on **Run** line or selection.

Alternatively you can simply hit *Ctrl* + *R* after having selected the line of code. As you might have guessed, the function returns as an output in the **Console** window:

```
[1] "Hello world"
```

Let's imagine you want to create a new script file that prints `Hi again, world` when run. This can be done by clicking on **New script** in the **File** menu and typing the following:

```
print("Hi again, world")
```

Now save this file as `hiagainworld.R` in the working folder. Use the **Save** function from the **File** menu of the R editor (not the console).

This book will not cover all functions in detail. If you want to know more about a function, simply precede its name by a question mark, for instance, `?print()`.

A quick look at the Misc menu

The **Misc** menu contains functions that are related to various aspects not otherwise classified as a menu in the RGui. Some useful functions of the **Misc** menu are as follows:

- **Stop current computation** and **Stop all computations**: When handling big datasets and computationally exigent algorithms, R may take longer than expected to complete the tasks. If for any reason, the console is needed during this time, the computations can be stopped by using this function.

- **List objects**: Pastes and runs the `ls()` function in the console. This outputs the list of objects in the current workspace.
- **List search path**: Pastes and runs the `search()` function in the console. This outputs the list of accessible packages. We will discuss this feature in the next section.

Try exercising these functions of the **Misc** menu:

Enter the following code in console:

```
repeat(a = 1)
```

This code will cause R to enter an infinite loop because the repeat statement continually runs the assignment `a = 1` in the code block, that is, what is contained between the parentheses `()`. This means that R will become unavailable for further computation. In order to give R some rest, we will now exit this loop by stopping the computation. In order to do this, select **Stop current computation** from the **Misc** menu. You can alternatively just press the *Esc* key to obtain the same result.

After doing the exercise above, get to know which objects are in the current workspace. In order to do this, simply click on **List objects**. The output should be as follows:

```
[1] "a"
```

Each time we create a variable, vector, list, matrix, data frame, or any other object, it will be accessible for the current session and visible using the `ls()` function.

Let's seize the opportunity to discuss some types of R objects and how to access their components:

- We call **variable** an object containing a single piece of information (such as the `a` object above).
- A **vector** is a group of indexed components of the same type (for instance, numbers, factors, and Booleans). Elements of vectors can be accessed using their index number between square brackets, `[ ]`. The following will create a vector `b` of three components, by using the `c()` function (for concatenate):

  ```
  b = c(1,2,3)
  ```

  The second element of vector `b` is accessed as follows:

  ```
  b[2]
  ```

- We call **attribute** a vector that is related to a measurement across observations in a dataset (for example, the heights of different individuals stored in a vector is an attribute).

- A list is a special type of vector that contains other vectors, or even matrices. Not all components of a list need to be of the same type. The following code will create a list called `c` containing a copy of variable `a` and vector `b`:

```
c = list(a,b)
```

  We use double brackets [[ ]], to access the components of a list. The copy of the `a` object stored in the list `c` that we just created can be accessed as follows:

```
c[[1]]
```

  Accessing the first element of the copy of vector `b` stored in list `c` can be done as follows:

```
c[[2]][1]
```

- A matrix can only contain elements of the same type. These are arranged in rows and columns. The following will create a 3 × 2 matrix of numbers (numbers 1 to 6), with odd numbers in the first column.

```
M = matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)
```

  The first row of the matrix can be accessed as follows:

```
M[1,]
```

  The second column of the matrix can be accessed as follows:

```
M[,2]
```

  The second element of the first column of the matrix can be accessed as follows:

```
M[2,1]
```

- A dataframe is a list of vectors that have the same dimensions, analogous to a spreadsheet. The following will create a data frame containing two vectors. The first contains the letters `a`, `b`, and `c`. The second contains the numbers `1`, `2`, and `3`.

```
f = data.frame(c("a","b","c"),c(1,2,3))
```

  The first vector of data frame `f` can be accessed as follows:

```
f[,1]
```

  This actually subsets the entire row of the first vector of the data frame. (Notice we did not have to use the double brackets notation here, but sometimes, this is necessary, depending on how the data frame has been generated.) When dealing with data frames (but not matrices), the comma can be omitted, meaning that the following is equivalent:

```
f[1]
```

The first element of the second vector of the data frame `f` (the element corresponding to the intersection of the first row and the second column of the data frame) can be accessed as follows:

```
f[1,2]
```

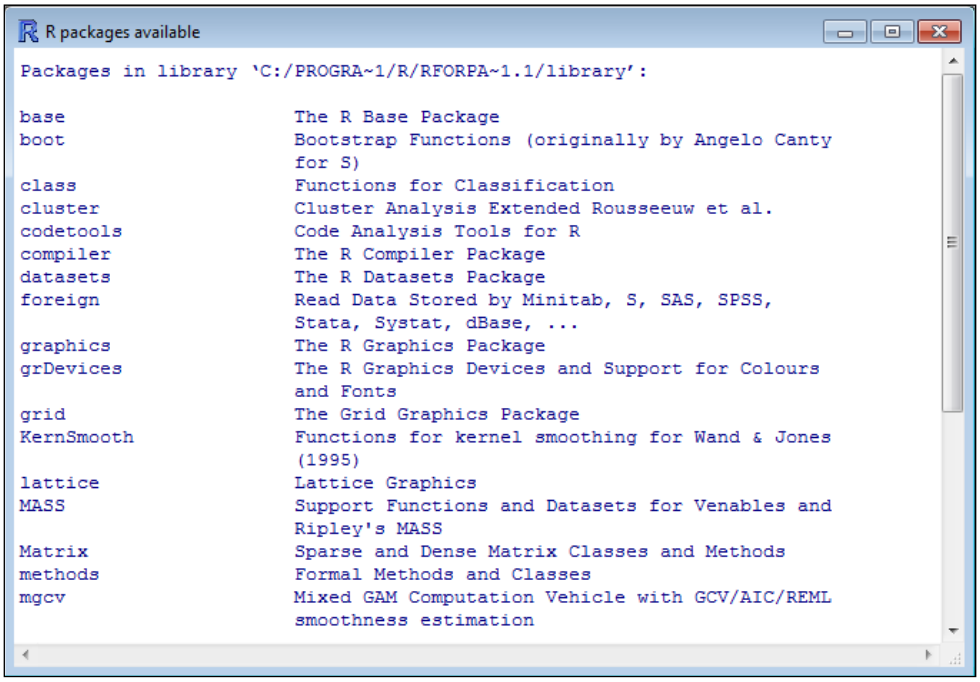
Subsetting can be more complex. For instance, the following code returns the second and the third rows of the first column of the data frame (note that matrices are subset in a similar manner):

```
f[2:3,1]
```

Packages

As mentioned earlier, GNU R is a statistical programming language that can be extended by means of packages. Packages contain functions and datasets that allow specific types of analyses to be performed in R. We have seen at the end of the last section that some packages are loaded by default in R. Others are already a part of R. The image below provides a list of the packages that come out of the box with R. This list can very easily be obtained with the following code:

```
library(lib = .Library)
```



Available packages in base R

Now, let's have a look at which packages are directly accessible, by selecting **List search path** from the **Misc** menu. This is what our output looks like:

```
[1].GlobalEnv package:stats package:graphics
[4] package:grDevices package:utils package:datasets
[7] package:methods Autoloads package:base
```

Accessible packages start with the prefix package:

Typing `search()` in the console would produce the same output.

Now, let's go a little further and list the content of one of these packages. In order to do this, type the following in the console:

```
objects(package:stats)
```

This will list the content of the **stats** package. The first two lines should look like this:

```
[1] acf acf2AR add.scope
[4] add1 addmargins aggregate
```

Installing packages in R

The content of this book is partly relying on packages that are not part of the basic installation of R. We will therefore need to install packages that we will download from CRAN. The **Packages** menu contains functions that allow installing and loading packages, as well as the configuration of local and distant repositories. Useful functions of the **Packages** menu include the following:

- **Load package**: Provides a frontend for the `library()` function, which loads a package provided as an argument.
- **Install packages**: Allows selecting a package to install. This requires configuring a mirror for CRAN first.
- **Install package(s) from local zip files**: Opens a dialogue box in which a ZIP file containing a package can be selected for installation in R.

Mirrors are basically different copies of CRAN. In case one mirror is down, the others provide redundancy. You can use any, but the closest to you will generally be faster. We use **0-Cloud** here.

We will discuss plotting in the next chapters. Most graphics in this book will be created using functions already available in R. These tools allow producing very accurate and informative graphics, but these are static. Sometimes, you might want to display your results on the web. Also, it sometimes comes in handy to be able to switch rapidly between two plots, for instance, to notice subtle differences. For these reasons, we will also introduce some basics of animation for displaying R plots on web pages. We will not discuss this in detail in this book, but we think it is something you might want a little introduction to.

In order to exercise the use of the menu and install the package required for animating graphics, let's start by installing the `animation` package. Select the **Install package(s)** function of the **Packages** menu, and then, select the `animation` package from the list. You will have to scroll down a little bit. If R asks you for a mirror, select **0-Cloud** or a location next to you, and confirm by clicking **OK**.

Alternatively, the next line of code will install the required package:

```
install.packages("animation")
```

Type this line of code in **R Console**; if you are using the e-book version of this book, copy and paste it in the console.

Alternatively, it is also possible to install packages in R from local files. This is useful in case the machine you are using R on does not have Internet access. To do so, use the **Install package(s) from local zip** function from the **Packages** menu and select the ZIP file containing the package you want to install. One easy way to do this is to copy the ZIP file in the working folder prior to attempting to install it. You can also use the following code, provided the package is called `package_0.1` and is in the working folder:

```
install.packages(paste0(getwd(),"/package_0.1.zip")), repos = NULL)
```

What we have done here deserves a little explanation. We are calling three functions here. By calling `install.packages()`, we tell R that we want to install a package. The `repos` attribute is set to `NULL`, which tells R that we do not want to download the package from a repository but prefer to install the package from a local file instead. The first argument passed to the function is therefore a filename (not a package name on CRAN as in the previous example). As we do not want to type in the whole path to the ZIP file as the first argument (we could have done so), we instead use the `paste0()` function to concatenate the output of `getwd()`, which shows the current working folder, and the filename of the ZIP file containing the package (between parentheses). The previous line of code allowed us to introduce the use of string concatenation in R while installing a package.

As R will automatically look in the working folder, we could have typed the following:

```
install.packages("package_0.1.zip")), repos = NULL)
```

Loading packages in R

Now that the animation package is installed, let's load it; select **Load package** from the **Package** menu. A dialogue box appears and prompts you to select the package that you want to load. If the installation was successful (which is most certainly the case if you didn't notice an error message), the package should be in the displayed list. Select it and confirm by clicking on **OK**.

Alternatively, you can simply type the following, which will also load the package:

```
library(animation)
```

A good thing to do when you load a package is to check that the functions you want to use are functional. For instance, it might be the case that some dependencies need to be installed first, although this should be done automatically when installing the package. In this book, we will use the `saveHTML()` function to animate some content and generate web pages from the plots. Let's test it with the following code:

```
1 df=data.frame(c(-3,3),c(3,-3))
2 saveHTML({
3    for (i in 1:20)  {
4       plot(df)
5       df = rbind(df,c(rnorm(1),rnorm(1)))
6    }
7 },
8 img.name = "plot",
9 imgdir = "unif_dir",
10 htmlfile = "test.html",
11 autobrowse = FALSE,
12 title = "Animation test",
13 description = "Testing the animation package for the first time.")
```

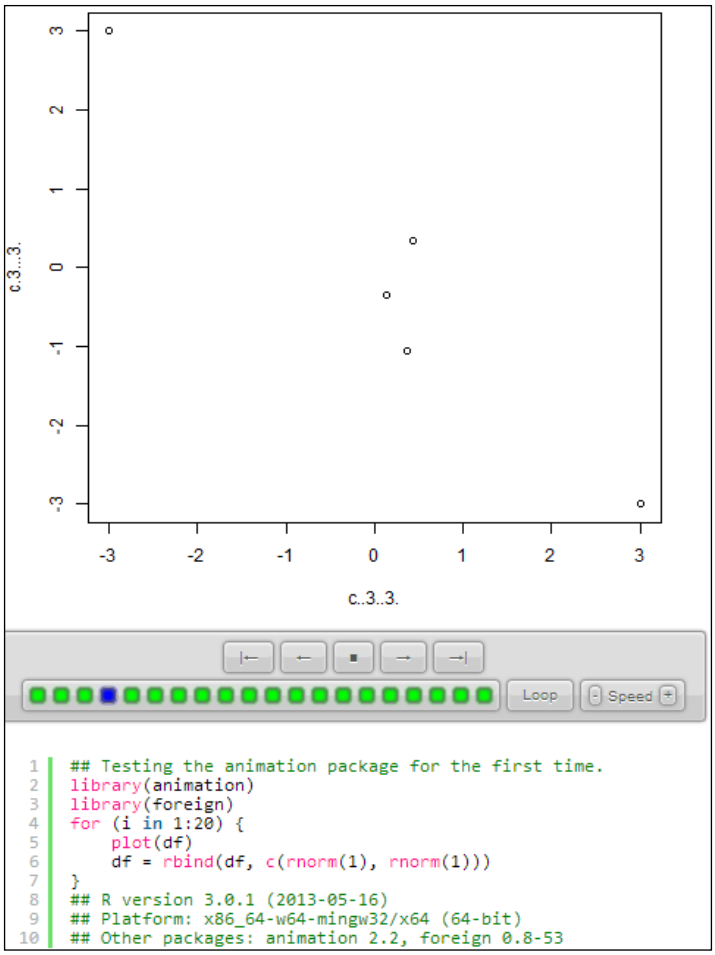
Line 1 creates a data frame of two columns. These are populated with -3 and 3 in the first row and with 3 and -3 in the second row. Lines 2 and 7 to 13 create and configure the animation. Lines 3 to 6 are where the content of the animation is generated. This is the part you might wish to modify to generate your own animations. Here, we plotted the values in the data frame and then added a new row containing random numbers. This code block will be iterated 20 times, as it is part of a `for` loop (see line 3). The reader is invited to consult an introduction to R if any of this is unclear.

For now, copy and paste the code in the console or type it in. The output should look like this:

```
animation option 'nmax' changed: 50 --> 20
animation option 'nmax' changed: 20 --> 50
HTML file created at: test.html
```

If you do not get the message above, first check whether the code that you typed in corresponds exactly to the code provided above. If the code corresponds, repeat steps 1 to 4 of the current section, as something might have gone wrong.

If you got the message above, open the HTML file in your browser. The file is in your working directory. The result should look like the image below. This is a scatter plot, which we will discuss further in the next chapter. The plot starts with the display of two data points, and then, new data points are randomly added. This plot (see below) is only provided as a test. Feel free to adapt the graphical content of the book by using the package (for example, you can simply paste the loops containing graphics in the code above, that is, instead of the `for` loop here), and of course, use your own data.



An animation produced using the Animation package

As an exercise in installing and loading packages, please install and load the `prob` package. When this is done, simply list the contents of the package.

We are sure that you have managed to do this pretty well. Here is how we would have done it. To install a package, we would have used the **Install package(s)** function in the **Package** menu. We could also have typed the following code:

```
install.packages("prob")
```

Alternatively, we would have downloaded the `.zip` file (currently, `prob_0.9-2.zip`) from CRAN: `http://cran.r-project.org/web/packages/prob/`.

Then, we would have used **Install package(s) from local zip** from the **Packages** menu and selected the ZIP file containing the `prob` package in the dialogue box.

Finally, we would have used the following code instead:

```
path = "c:\\user\\username\\downloads\\prob_0.9-2.zip"
install.packages(path, repos = NULL)
```

In order to load the package, we would have selected **Load package** from the **Package** menu, and chosen the file containing the package in the dialogue box.

This might be counterintuitive, but using code is way easier and more efficient than using the GUI. In order to load the `prob` package, we could have also simply used the following code:

```
library(prob)
```

We would have listed the contents of the package by using the `objects()` function:

```
objects(package:prob)
```

The output lists 43 functions.

We have presented the exercises in the chapter together with their solutions here. The exercises for the next chapters will be part of the *Appendix A*, *Exercises and Solutions*, together with their solutions.

Summary

In this chapter, we explained where to find the installer for R for all platforms, described the graphic user interface, and examined its menus, particularly how to create and run scripts. We also described how to load packages and discovered some of the basics of the syntax of R. In the next chapter, we will start visualizing data. We will explore this by looking at an example of a roulette game.

2
Visualizing and Manipulating Data Using R

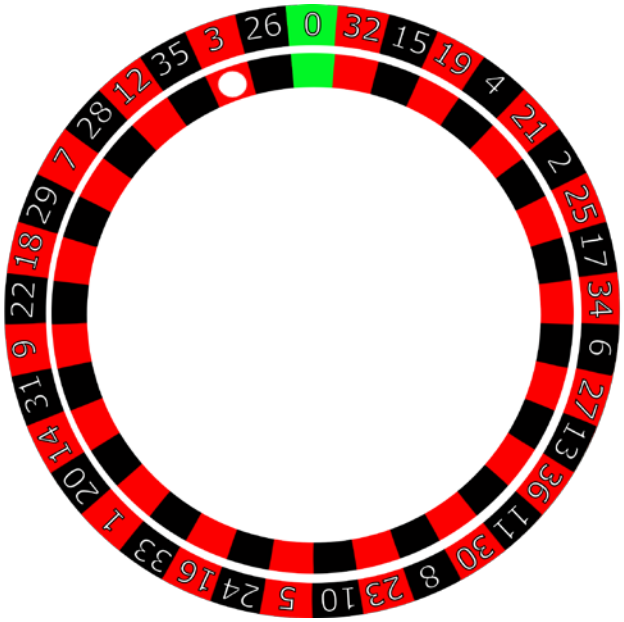
Data visualization is one of the most important processes in data science. Relationships between variables can sometimes more easily be understood visually than relying only on predictive modeling, or statistics, and this most often requires data manipulation. Visualization is the art of examining distributions and relationships between variables using visual representations (graphics), with the aim of discovering patterns in data. As a matter of fact, a number of software companies provide data visualization tools as their sole or primary product (for example, Tableau, Visual.ly). R has built-in capabilities for data visualization. These capabilities can of course (as with almost everything in R) be extended by recourse to external packages. Furthermore, graphics made for a particular dataset can be reused and adapted for another with relatively little effort. Another great advantage of R is of course that it is a fully functional statistical software, unlike most of the alternatives.

In this chapter, we will do the following:

- Examine the basic capabilities of R with regards to data visualization, by using some of the most important tools for visualization: histograms, bar plots, line plots, boxplots, and scatterplots.
- Generate data sets based on virtual European roulette spins and develop basic data manipulation (for example, subsetting) and programming skills (use of conditions and loops).
- This will give us the opportunity to have a look at visualization tools with data for which the theoretical distributions and relationships are known in advance; whereas, usually, the theoretical distribution is unknown and the aim of visualization is to get an understanding of the data structures and patterns. Working with known theoretical distribution allows for observing deviations from what is expected.

The roulette case

Roulette is a betting game which rewards the player's correct prediction of its outcome. The game consists of a ball spinning around a wheel which rotates in the opposite direction. The wheel features 37 numbered pockets. Each of the number has a color (18 are red, 18 are black and one, the zero, is green). The aim of the game is to bet on one or several outcomes regarding the pocket on which the ball lands. Numbers can range from 0 to 36, and several types of bets are available such as the color of the number, it being even or odd, and several other characteristics related to the number or the position on the wheel (as marked on the betting grid). The image below is a representation of an European roulette wheel. The ball is represented by the tiny white circle. In this example it landed on the pocket corresponding to the number 3.



A representation of a roulette wheel

Numbers are ordered on the wheel in such a way that the position of a number on the wheel is as unrelated as possible to the possible bets, (except of course bets on the position itself, which we will not describe here). The order of the numbers, starting from 0 is visible on the image above. As you can notice, the color of the numbers alternates when moving forward on the wheel (red, black, red, and so on). Also, the order seems to be unrelated to the betting grid. We will see if this is the case at the end of the chapter.

The table below is a schematic representation of the betting grid at European roulette. Red numbers are italicized. Betting on each number returns 35 times the amount if the number is drawn (plus the initial bet). Betting on color (red or black), odd vs even, 1-18 vs 19-36 return each the betted amount (plus the initial bet), if the drawn number corresponds to that attribute. The probability of occurrence of any of these is 18/37 or 0.487. Betting on the 1st dozen, 2nd dozen, 3rd dozen and each of the 2:1 column returns for each 2 times the amount if the drawn number falls in that category (plus the initial bet). The probability of occurrence of any of these is 12/37 or 0.32. Bets are lost if the drawn number fails to be within the betting category. In the example above, the ball stopped on number 3. Examples of winning bets in the depicted example are Red, first dozen, 1-18, the 3rd column, and of course betting on number 3.

The following table depicts a betting grid at roulette.

		0		
1-18	1st 12	*1*	2	*3*
		4	*5*	6
EVEN		*7*	8	*9*
		10	11	*12*
RED	2nd 12	13	*14*	15
		16	17	*18*
BLACK		*19*	20	*21*
		22	*23*	24
ODD	3rd 12	*25*	26	*27*
		28	29	*30*
19-36		31	*32*	33
		34	35	*36*
		2-1	2-1	2-1

A representation of a betting grid at roulette

Histograms and bar plots

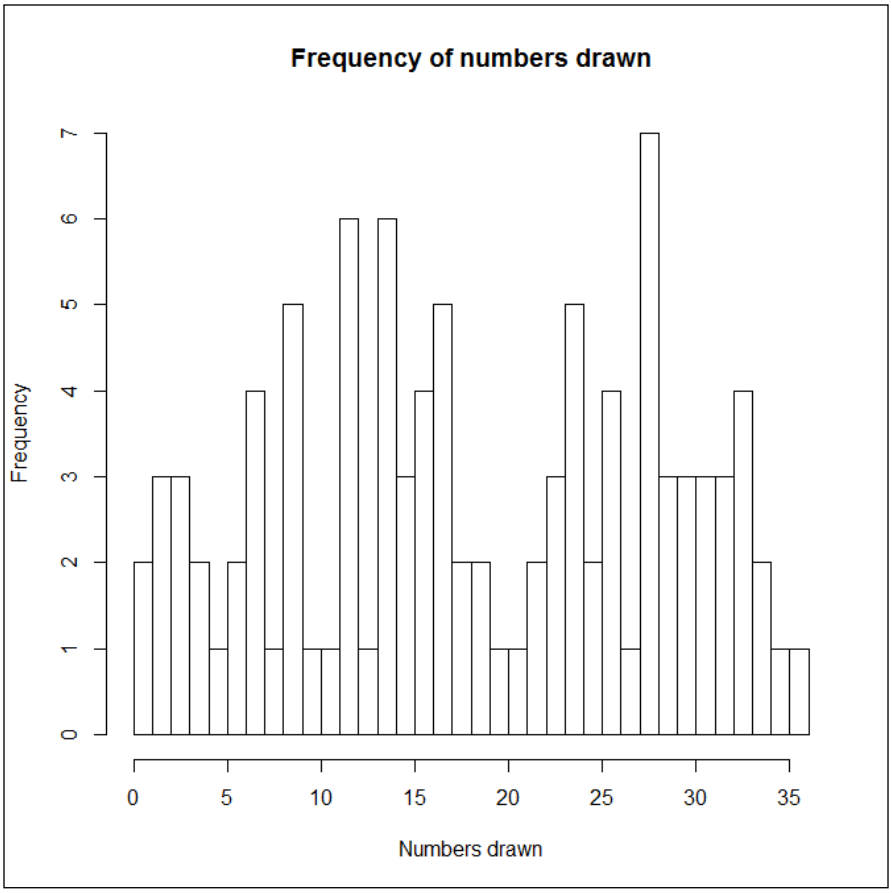
Roulette is a fascinating example of a betting game using random outcomes. In order to explore some properties of roulette spins, let's visualize some randomly drawn numbers in the range of those in an European roulette game (0 to 36). Histograms allow the graphic representation of the distribution of variables. Let's have a look at it! Type in the following code:

```
1  set.seed(1)
2  drawn = sample(0:36, 100, replace = T)
3  hist(drawn, main = "Frequency of numbers drawn",
4     xlab = "Numbers drawn", breaks=37)
```

Here we first set the seed number to `1` (see line 1). For reproducibility reasons, computer generated random numbers are generally not really random (they are in fact called pseudo-random). Exceptions exist, such as numbers generated on the website `http://www.random.org` (which bases the numbers on atmospheric variations). Setting the seed number to `1` (or any number really) makes sure the numbers we generate here will be the same as you will have on your screen, when using the same code and the same seed number. Basically, setting it allows the reproduction of random drawings. On line 2, we use the `sample()` function to generate 100 random numbers in a range of 0 to 36 (`0:36`). The `replace` argument is set to true (`T`), which means that the same number can be drawn several times.

The `hist()` function (lines 3 and 4) will plot the frequency of these numbers. The `hist()` function takes a multitude of arguments, of which we use 4 here; `main`, which sets the title of the graphic, `xlab`, which sets the title of the horizontal axis (similarity, `ylab` would set the title of the vertical axis), and `breaks`, which forces the display of 37 breaks (corresponding to the number of possible outcomes of the random drawings). For more information about the `hist()` function, you can simply type `?hist()` in your R console.

As you can notice on the graph below, the frequencies are quite different between numbers, even though each number has an equal theoretical probability to be drawn on each roll. The output is provided in the figure below:



A histogram of the frequency of numbers drawn

Let's dwell a little upon the representation of mean values using bar plots. This will allow us to have a look at other properties of the roulette drawings. The mean values will represent the proportions of presence of characteristics of the roulette outcomes (for example, proportion of red number drawn). We will therefore build some new functions.

The `buildDf()` function will return a data frame with a number of rows that correspond to how many numbers we want to be drawn, and a number of columns that correspond to the total number of attributes we are interested in (the number drawn, its position on the wheel and several possible bets), totaling 14 columns. The matrix is first filled with zeroes, and will be populated at a later stage:

```
1    buildDf = function(howmany) {
2       Matrix=matrix(rep(0, howmany * 14), nrow=howmany,ncol=14)
3       DF=data.frame(Matrix)
4       names(DF)=c("number","position","isRed","isBlack",
5         "isOdd","isEven","is1to18","is19to36","is1to12",
6         "is13to24","is25to36","isCol1","isCol2","isCol3")
7       return(DF)
8    }
```

Let's examine the code in detail: on line one, we declare the function, which we call `buildDF`. We tell R that it will have an argument called `howmany`. On line 2, we assign a matrix of `howmany` rows and `14` columns to an object called `Matrix`. The matrix is at this stage filled with zeroes. On line 3, we make a data frame called `DF` of the matrix, which will make some operations easier later. On lines 4 to 6, we name the columns of the data frame using `names()` functions. The first column will be the number drawn, the second the position on the wheel (the position for `0` will be `1`, the position for `32` will be `2`, and so on). The other names correspond to possible bets on the betting grid. We will describe these later when declaring the function that will fill in the matrix. On line 7, we specify that we want the function to return the data frame. On line 8, we close the function code block (using a closing bracket), which we opened on line 1 (using an opening bracket).

Our next function, `attributes()`, will fill the data frame with numbers drawn from the roulette, their position on the roulette, their color, and other attributes (more about this below):

```
1    attributes = function(howmany,Seed=9999) {
2       if (Seed != 9999) set.seed(Seed)
3       DF = buildDf(howmany)
4       drawn = sample(0:36, howmany, replace = T)
5       DF$number=drawn
6       numbers = c(0, 32, 15, 19, 4, 21, 2, 25, 17, 34, 6, 27,
7          13, 36, 11, 30, 8, 23, 10, 5, 24, 16, 33, 1, 20, 14,
8          31, 9, 22, 18, 29, 7, 28, 12, 35, 3, 26)
```

The function is not fully declared at this stage. We will break it down in several parts in order to explain what we are doing here. On line 1, we assign the function to object attributes, specifying that we have 2 arguments; `howmany` for the number of rows corresponding to how many numbers we want to be drawn, and `Seed` for the seed number we will use (with default value `9999`). On line 2, we set the seed to the provided seed number if it is not `9999` (as we need the function to be able not to set the seed for analyses we will do later). On line 3, we create the data frame by calling the function `buildDf()` we created before. On line 4, we sample the specified amount of numbers. On line 5, we assign these numbers to the column of the data frame called `drawn`. On line 6, we create a vector called `numbers`, which contains the numbers `0` to `36`, in the order featured on the roulette wheel (starts with 0, then 32, 15 …).

In the remaining of the function (presented below), we populate the rest of the attributes:

```
9       for (i in 1:nrow(DF)){
10         DF$position[i]= match(DF$number[i],numbers)
11         if (DF$number[i] != 0) { if (DF$position[i]%%2) {
12            DF$isBlack[i] = 1} else {DF$isRed[i] = 1}
13         if (DF$number[i]%%2) { DF$isOdd[i]=1}
14         else {DF$isEven[i]=1}
15         if (DF$number[i] <= 18){ DF$is1to18[i]=1}
16         else { DF$is19to36[i]=1}
17         if(DF$number[i] <= 12){ DF$is1to12[i]=1}
18         else if (DF$number[i]<25) { DF$is13to24[i] = 1}
19            else { DF$is25to36[i] = 1}
20         if(!(DF$number[i]%%3)){ DF$isCol3[i] = 1}
21         else if ((DF$number[i] %% 3 ) == 2) {
22           DF$isCol2[i] = 1}
23           else { DF$isCol1[i] = 1}
24           }
25         }
26       return(DF)
27    }
```

On line 9, we create a loop, meaning that the code block will iterate from i = 1, to i = the number of numbers we have drawn (the number of rows of the data frame). We open the code block using an opening bracket. On line 10, we assign to the attribute `position` of the drawn number on the wheel, using function `match()`. On lines 11 to 12, we create a nested condition, stating that if the number is not 0, we assign 1 to attribute `isBlack` is the position of the number is even, or 1 to `isRed` if the position is odd (remember the color of the numbers alternate – red, black, red ...). On line 13 and 14, we assign `1` to attribute `isOdd` if the number if odd, or `1` to attribute `isEven` if the number is even. On lines 15 and 16, we assign `1` to attribute `is1to18` if the number is smaller or equal to `18`, or `1` to attribute is`19to36` if the number is higher than `18`. On lines 17 to 18, we assign `1` to either `is1to12`, `is13to24` or `is25to36` depending on the value of the number (that's self-explanatory). Finally, on lines 20 to 26, we assign the column number on the betting grid, by setting the value of either `isCol1`, `isCol2`, or `isCol3` (on the table representing the betting grid, `isCol1` is the left 2:1 column, `isCol2` the middle 2:1 column and `isCol3` the right one). As we have used nested conditions here, we close the code block on lines 24 and 25. On line 26, we tell R that we want the function to return the resulting data frame. On line 27, we close the code block of the function (that we opened on line 1).

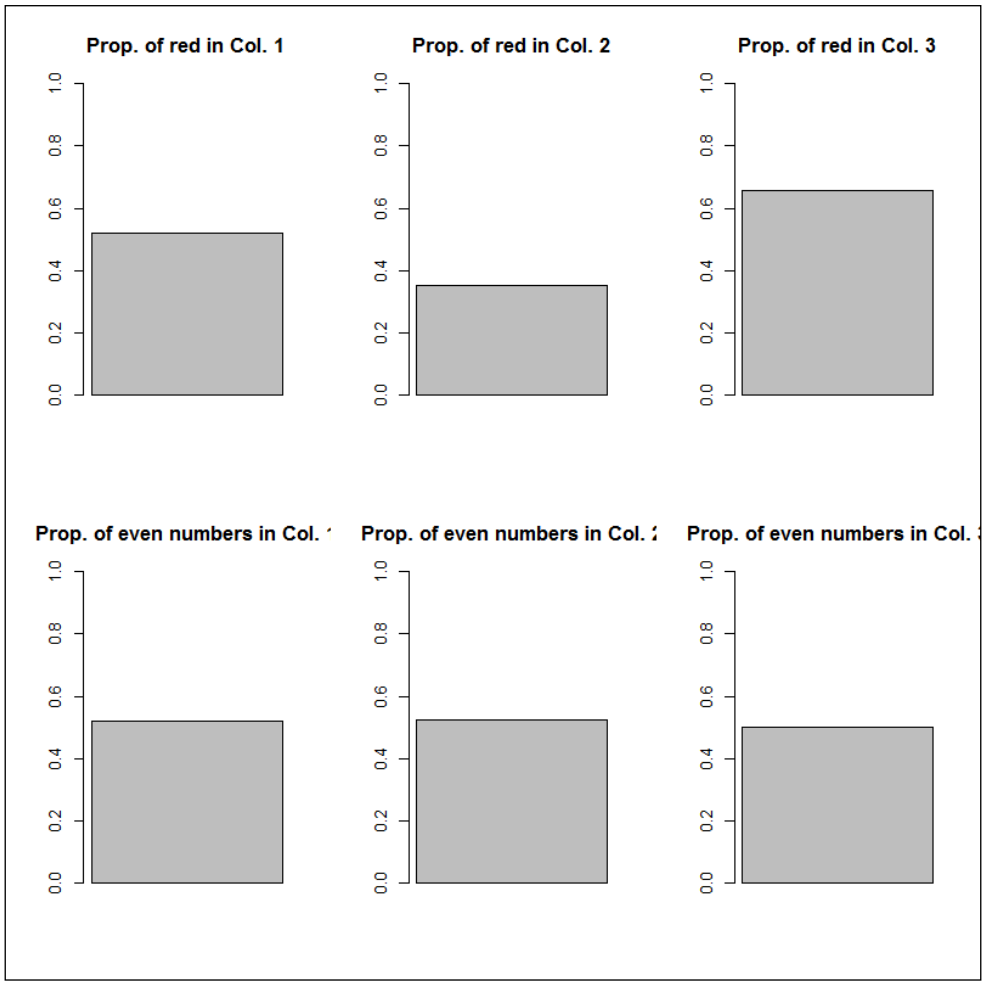
Now that we have our functions ready, we can now focus on visualizing some data. The following code will generate 1,000 roulette spins (let's use a seed number of `2` so that the calculation of the random number is the same on your machine as in mine):

```
Data=attributes(1000,2)
```

It is now time to explore the relationship between our variables in the following graph. We will first ask R to plot several graphs on the plotting area. To do so, we will rely on the `mfrow` argument of the `par()` function (line 1). We then tell R to plot `2` rows of `3` graphs corresponding to the proportion of red numbers (the mean of the values, as these are represented by `1` for presence and `0` for absence) in the 2:1 columns 1, 2 and 3 on the first row, and the proportion of even number in columns 1, 2 and 3 in the second row. Notice that for all 6 plots we use subsetting (using `subset()` function here) to select the portion of the data we are interested in. We use attribute `ylim` to define the range of the plotting area (from `0` to `1`), and attribute `main` to print the title of the plots.

```
1    par(mfrow = c(2,3))
2    barplot(mean(subset(Data, isCol1 == 1)$isRed), ylim=(c(0,1)),
3       main = "Prop. of red in Col. 1")
4    barplot(mean(subset(Data, isCol2 == 1)$isRed), ylim=(c(0,1)),
5       main = "Prop. of red in Col. 2")
6    barplot(mean(subset(Data, isCol3 == 1)$isRed), ylim=(c(0,1)),
7       main = "Prop. of red in Col. 3")
```

```
8    barplot(mean(subset(Data, isCol1 == 1)$isEven), ylim=(c(0,1)),
9       main = "Prop. of even numbers in Col. 1")
10    barplot(mean(subset(Data, isCol2 == 1)$isEven), ylim=(c(0,1)),
11       main = "Prop. of even numbers in Col. 2")
12    barplot(mean(subset(Data, isCol3 == 1)$isEven), ylim=(c(0,1)),
13       main = "Prop. of even numbers in Col. 3")
```



Bar plots of the proportion of red, and even numbers drawn from Columns 1, 2 and 3

As can be seen on the graphs, the proportion of red numbers drawn from columns 1, 2 and 3 is different, whereas the proportion of even numbers is relatively similar between all the columns. This can be expected from the betting grid.

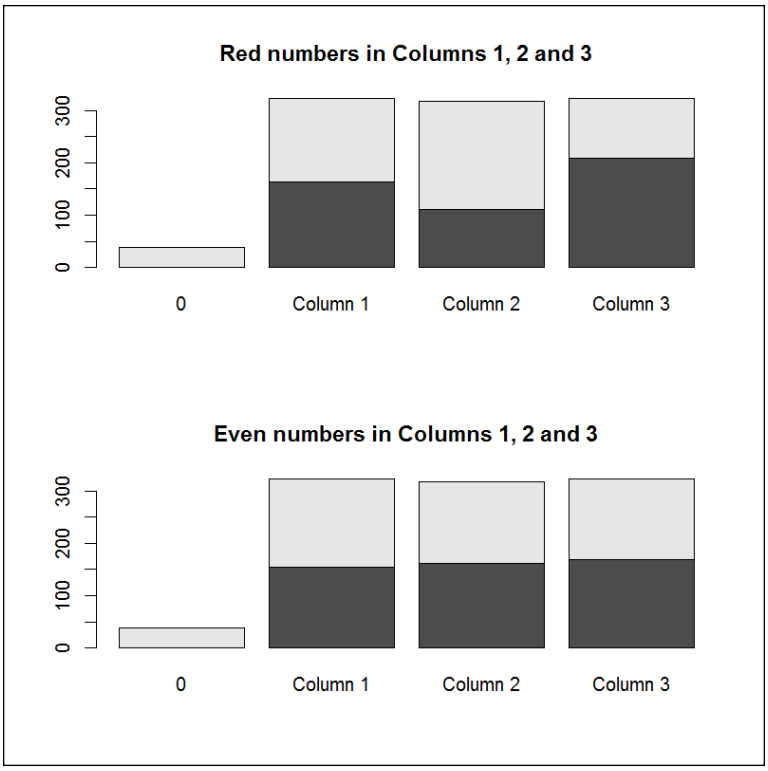
You might have noticed that we have lost important information in the process; the total of numbers drawn from each column, and the number of zeros; and we needed to produce one bar plot per column, which is a bit tricky. Let's solve these problems by first adding a single attribute which indicates the membership of the drawn numbers to `Column 1`, `Column 2` and `Column 3`.

```
1    for (i in 1:nrow(Data)){
2       if(Data$isCol1[i]== 1){ Data$Column[i]=1 }
3          else if (Data$isCol2[i] == 1 ) { Data$Column[i] = 2 }
4          else if (Data$isCol3[i] == 1 ) { Data$Column[i] = 3 }
5          else {Data$Column[i] = 0 }
6    }
```

On line 1, we start a `for` loop that will iterate from i = 1 to i = the number of rows in data frame `Data`. We use nested condition in lines 2 to 5 to determine the column number (1 if attribute `isCol1` equals to `1`, `2` if attribute `isCol2` equals to `1`, `3` if attribute `isCol3` equals to `1`, or `0` if neither of these conditions is satisfied. We close the code block on line 6.

We now can plot the column in relation to the proportion of red, and even numbers. For now, our attributes `isRed` and `isEven` are ordered with `0` coming first and `1` second. We want just the opposite, as we want the number of numbers coded `1` to appear at the bottom of the graph. We therefore reorder the values of our attributes using the levels attribute of the `factor()` function (lines 1 and 2). We will use `par()` again to get both graphs on the same plotting area. We then generate the stacked bar plots using the `barplot()` function again. Notice we do not plot mean values this time, but the content of the table in which the cells correspond to the intersections of the attributes `Column` and `isRed` or `isEven`. We rely on the argument `name.arg` to name the sections of the plots:

```
1    Data$isRed = factor(Data$isRed, levels = c(1,0))
2    Data$isEven = factor(Data$isEven, levels = c(1,0))
3    par(mfrow = c(2,1))
4    barplot(table(Data$isRed,Data$Column),
5       main = "Red numbers in Columns 1, 2 and 3",
6       names.arg = (c("0","Column 1", "Column 2", "Column 3")) )
7    barplot(table(Data$isEven,Data$Column),
8       main = "Even numbers in Columns 1, 2 and 3",
9       names.arg = (c("0","Column 1", "Column 2", "Column 3")) )
```



A bar plot of the number of Red and Even numbers drawn

As can be seen on this stacked bar plot, approximately the same amount of numbers have been drawn from each of the columns. The number 0 has been drawn around 50 times, which is about twices often as expected given its theoretical probability equal to those of the other numbers (*1000 * (1/37) = 27*).

Scatterplots

Until now we have observed frequencies of the relationship between categorical membership (nominal attributes) and frequencies or means. It is also useful to have a look at relationships between numerical attributes. We will rely on scatterplots for this purpose. This will require a little scripting again, as we will examine the relationships between proportions. Let me first introduce the function `proportions()` which will generate the proportions for us, for all of our nominal attributes. This function takes one argument, `DF`, and call our `attributes()` function by default. We could instead give as an argument the data frame with the numbers we have previously drawn and the attributes.

The body of the function computes and returns the transpose of the means of each nominal attributes:

```
1    proportions = function(n = 100) {
2       DF=attributes(n)
3       return(data.frame(t(colMeans(DF[3:ncol(DF)]))))
4    }
```

The body of this function calls our `attributes()` function and passes the number of roulette draws to it (line 2). It then returns a data frame which contains the transpose means of all the columns, except columns 1 and 2 (which are not of interest here).

Our next function `multisample()` will return a data frame containing the proportions of each attribute of each of `k` samples (one sample per row) of `n` numbers drawn. It will by default draw `100` samples of `100` numbers. After starting the function declaration on line 1, we set the seed to the provided value, or the default value on line 2. We then create a vector containing the values returned by a first call to the `proportions()` function. In the following loop, we append iteratively values returned by function `proportions()` (lines 4 to 7). Finally, we return the resulting data frame (line 8), and close the function code block (line 9).

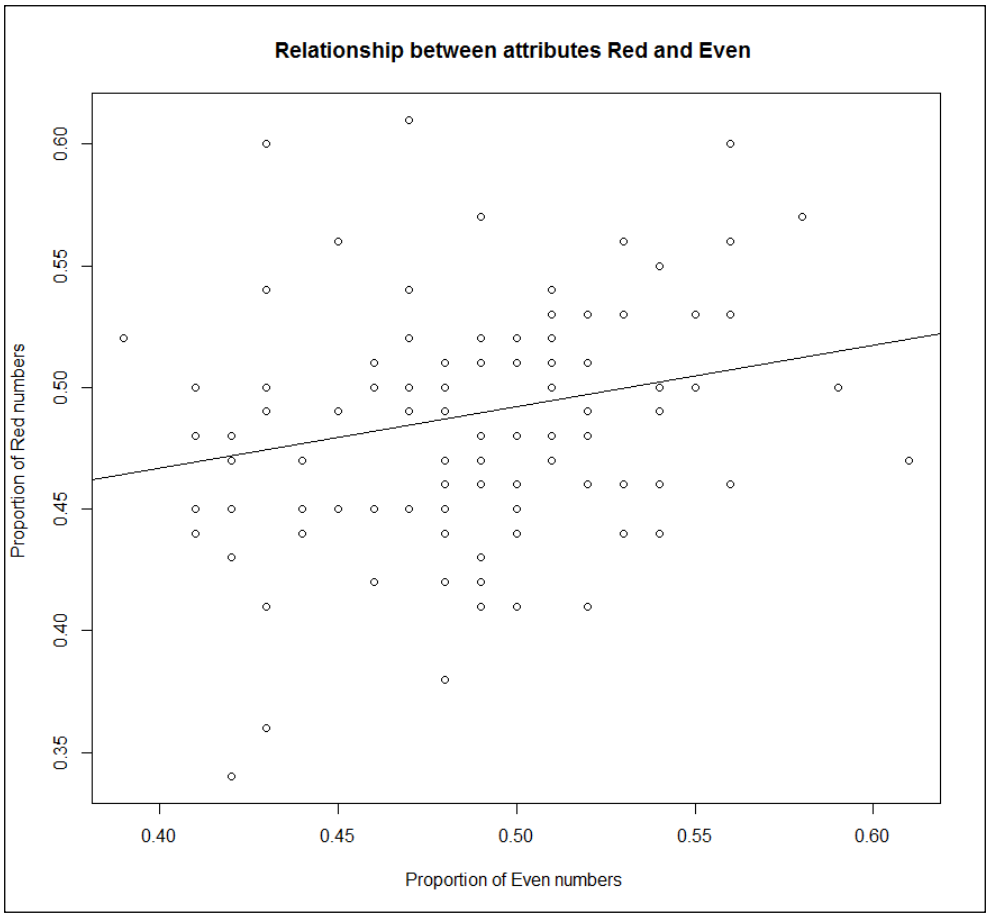
```
1    multisample = function(n=100,k=100, Seed=3){
2       set.seed(Seed)
3       ColMeans.df=proportions(n)
4       for (i in 1:k-1){
5          ColMeans.df=rbind(ColMeans.df,
6             proportions(n))
7       }
8       return(ColMeans.df)
9    }
```

We are now able to examine the relationship between proportions of numbers using scatterplots. Scatterplots display each observation on a plane by plotting the values of two attributes. On line 1, we first create a data frame of proportions using the default arguments for `multisample()` function. This will not take too long to compute. Having a look at the roulette grid, one can see that 10 out of the 18 red numbers are odd. Will we be able to spot this relationship from the random drawings? We will investigate this visually. We plot the proportions of red and the proportions of even numbers using a scatterplot (lines 3 to 6). The `main` argument set the title of the graph (line 4). The `xlab` argument sets the title of the horizontal axis (line 5). The `ylab` argument sets the title of the vertical axis (line 6). We also add and a line (called slope) showing the direction of the relationship using `abline()` function on line 7.

The function here uses the coefficients of a linear model as argument. I will discuss the `lm()` function which provides such coefficients in the chapter about regression:

```
samples = multisample()
par(mfrow=c(1,1))
plot(samples$isOdd,samples$isRed,
   main = "Relationship between attributes Red and Even ",
   xlab = "Proportion of Even numbers",
   ylab = "Proportion of Red numbers")
abline(lm(samples$isOdd~samples$isRed))
```

The output is provided below:



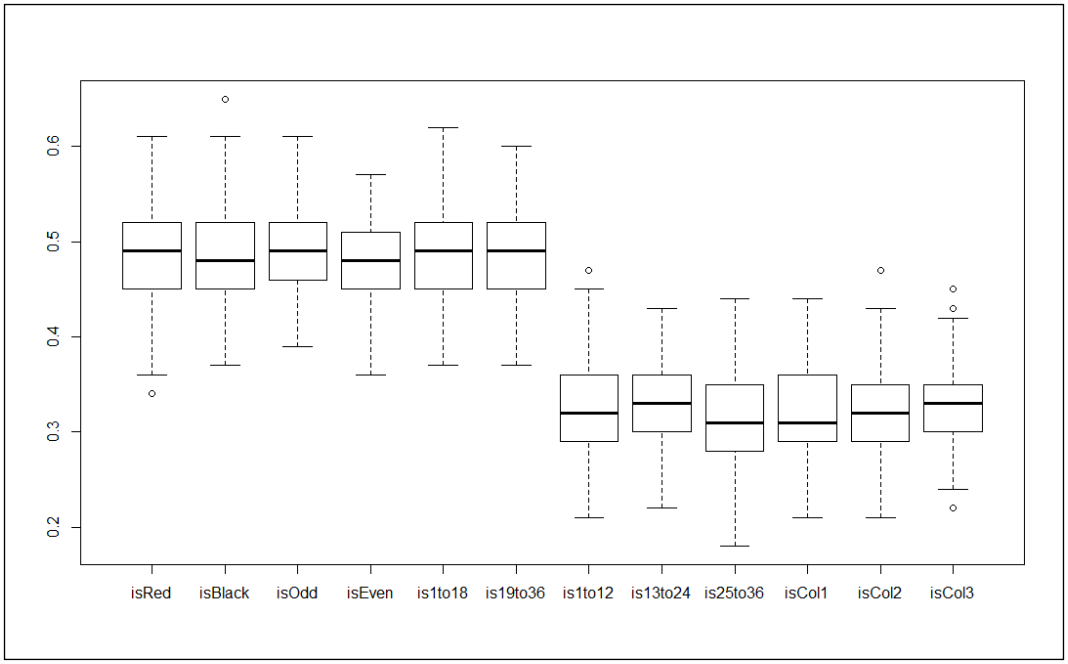
A scatterplot showing the relationship between the proportion of Red and Even numbers

The graph depicts the values on our two attributes (the proportion of even numbers on the *x* axis and the proportion of red numbers on the *y* axis) for each of our samples. The line represents the linear relationship between these two proportions; the higher the proportion of even numbers, the higher the proportion of red numbers. Again, this can be expected from the betting grid.

Boxplots

As we can also notice from the scatterplot, whereas most of the samples have a relatively balanced proportion of red or even numbers, these proportions are very small or large in some cases. We could examine the dispersion of those values using a histogram again, but the boxplot is much more interesting, so we will use it instead. Boxplots are representations of the distribution of an attribute. We could have a look at only one attribute by specifying its name as an argument from the `boxplot()` function. We will instead look at all the arguments at once by giving the data frame as an argument:

```
boxplot(samples)
```



Boxplots of all the attributes

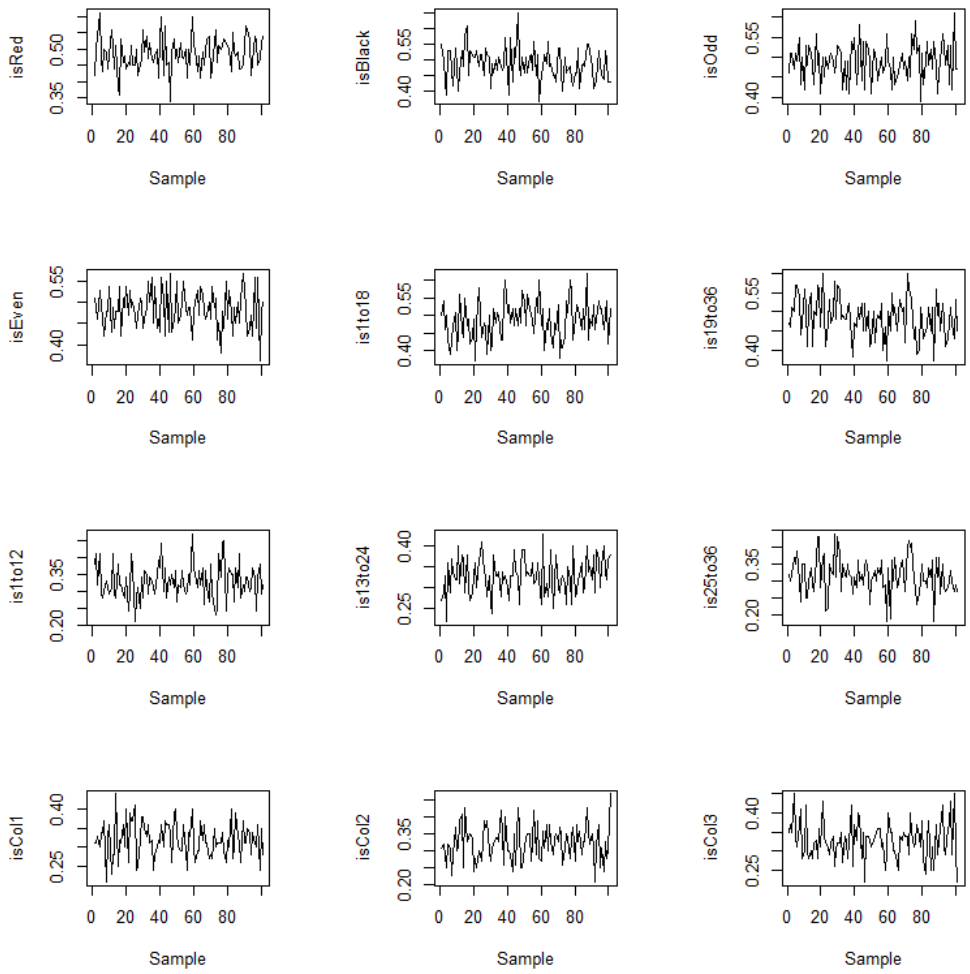
As can be seen from the boxplots, the proportions of red, black, odd, even, numbers below 18, and numbers higher than 18 are a little below 50% on average, which is what is expected as 18 of 37 numbers are in each of these categories. We can also notice that the average proportion of numbers between 1 and 12, 13 and 24, 25 and 36, as well as numbers on columns 1, 2 and 3 are a bit below 33%, which is expected as well. What might surprise us is that there is a huge variation around these average values. On each boxplot, the bottom box represents the data points that are in the second quartile. The top box represents data points that are in the third quartile. Thus, 50% of our data points fit in the two boxes. The space between the whiskers represents 150% of the interquartile range (the distance between the third and first quartile, or Q3-Q1). Finally, outliers are displayed as separate points on the boxplots. We can visually notice that the space between the whiskers it is about as large for the attributes on the right large of the graph as for attributes on the left side, even though the median proportion is much lower.

Line plots

Line plots provide the same information as bar plots. They might allow to understand relationships between attributes better because the values are linked by lines which give a better feeling of the difference between the values. We will investigate the variability of the proportions of each attribute by plotting its proportion from each sample. On line 1, we will first configure the plotting area contain 12 plots (as we have 12 attributes). Notice we use the `oma` attribute to set the outer margin, and the `mar` attribute to set the inner margin. On line 2, we set the names to be used in the titling of the axis (using the `ylab` attribute, see line 4). We then iteratively create, for each attribute, a graph plotting each value (lines 3 to 5). The type attribute is set to `l` (line 5) in order to plot lines instead of dots as in a scatterplot.

```
1    par(mfrow=c(4,3), oma = rep(0.1,4), mar = rep(4,4))
2    names=colnames(samples)
3    for (i in 1:ncol(samples)){
4       plot(samples[,i], xlab="Sample", ylab=names[i],
5          type = "l")
6    }
```

The output is provided below:



Variability of the proportion of each attribute

Application – Outlier detection

You might remember that at the beginning of the chapter, we noticed in the stacked bar plot that in our sample of 1,000 roulette spins, the zero was drawn about twice as often as we would expect. We just mentioned it but didn't really have a point of comparison. We now have proportions from 100 samples and thus can examine this a little further. The proportion of zeros can be obtained from the data we have as we simply have to subtract from `1`, the sum of proportions of red and black numbers for each of the samples. So let's do this, and add the attribute to the data frame, and get the mean value of this proportion:

```
samples$isZero = 1-(samples$isRed+samples$isBlack)
Mean = mean(samples$isZero)
Mean
```

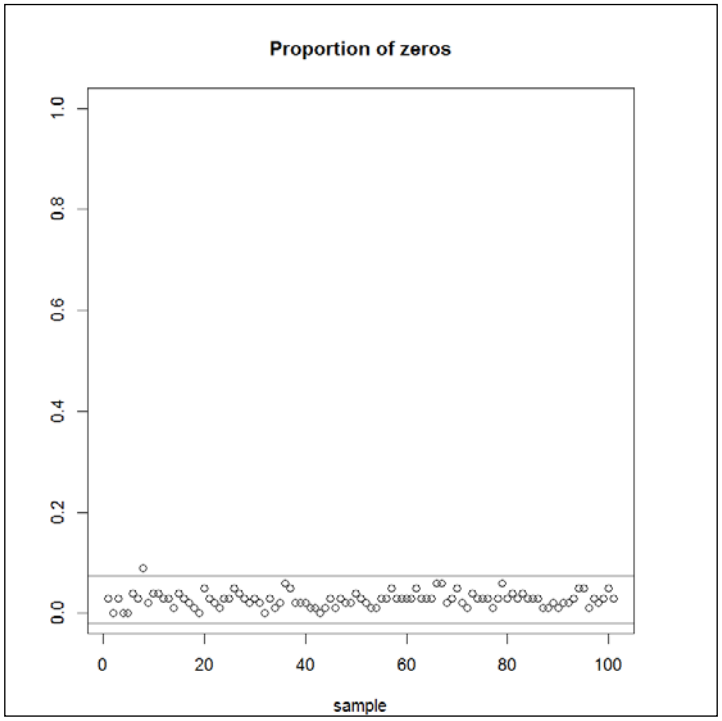
The mean value is `0.0277`. We can compute the value we would expect is 1/37, which is `0.0270`. The average value of the proportion of zeros in all our 100 samples is therefore almost identical to the expected value. This in no way means that there are no outliers.

There are several ways to detect outliers. When seeking detection of outliers in multivariate data, the Mahalanobis distance or leverage points can be used. As these do not rely on visualization, we will not discuss them here. The interested reader can refer to the paper *Unmasking outliers and leverage points* by Rousseeuw and van Zomeren.

Our problem here is univariate, and a simple visualization technique is enough for the current purpose. A simple and classic approach is to see how many of the values (here proportions of 0) fall outside of the mean + or – 3 standard deviations. So let's start by computing the thresholds (lines 1 and 2). We then plot the proportions of zeroes from all our samples (lines 3 and 4). Notice we use the `ylim` attribute (line 3) to specify that we want the vertical boundaries of our graph to include all possible values (a proportion can range from 0 to 1). We will then add two lines showing the limits of the interval between the mean and 3 standard deviations above and below it (lines 6 and 7):

```
1    upper = Mean+(3*sd(samples$isZero))
2    lower = Mean-(3*sd(samples$isZero))
3    par(mfrow=c(1,1))
4    plot(samples$isZero, main = "Proportion of zeros",
5       xlab = "sample", ylab= "", ylim = c(0,1))
```

```
6     abline(h=upper)
7     abline(h=lower)0
```



Finding extreme proportions of zeros visually

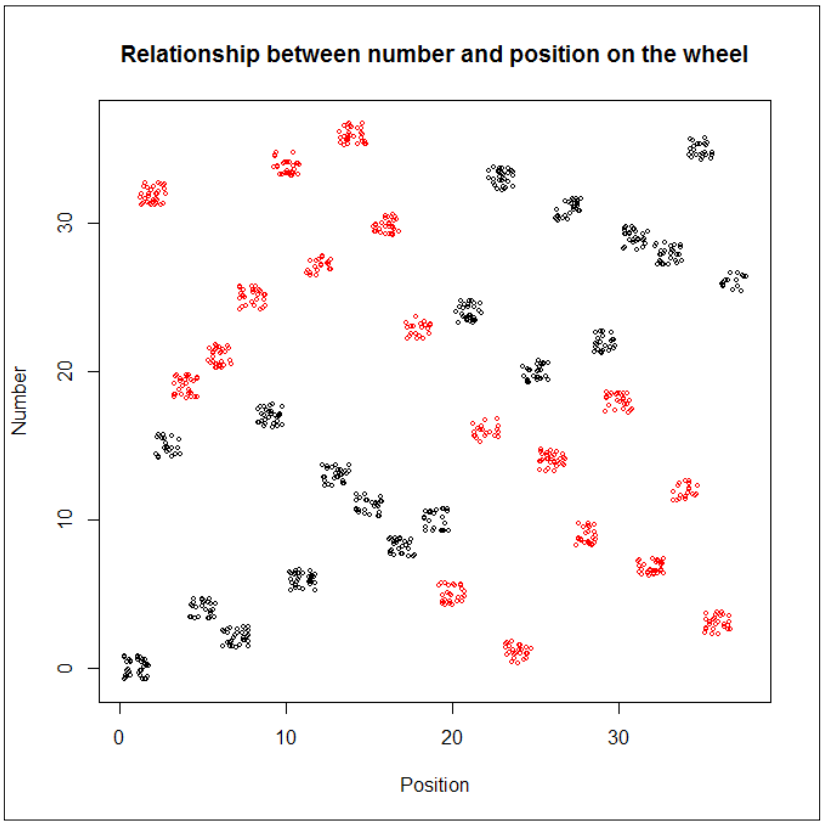
We can notice that there is only one value above the upper threshold. All other values are thus not considered as outliers, as the fit in the range of the mean plus or minus 3 standard deviations. We can also notice that the lower threshold is below 0. This is not possible for proportions.

Formatting plots

Plots in R can be formatted in many ways. We have already seen some of them in this chapter. In this section, we briefly explore some of these options. Let's go back to the data frame containing the 1,000 roulette spins and examine the relationship between the position on the roulette and the number by color. On line 1, we call the plot function. On line 2, we specify the attributes to be plotted, and add a little jitter to the data, using the `jitter()` function, otherwise, many points will be stacked over each other. The factor argument of this function controls the amount of jittering. We also reduce the size of the dots, using the `cex` attribute (line 3). We then title our graph and axes (lines 4 to 6).

Finally, we want to color the dots according to whether the number drawn is red or not (line 7). For this purpose, we use the `col` attribute:

```
1    plot(
2      jitter(Data$position, factor=4),jitter(Data$number, factor=4),
3      cex = 0.5,
4      main = "Relationship between number and position on the
5      wheel", xlab = "Position",
6      ylab="Number",
7      col=as.factor(Data$isRed))
```



Relationship between number drawn and position on the wheel

As we can see from the graph, there is a relationship between the position on the wheel and the number, when considering color. Yet, red is associated with numbers higher than 18 in the first half of the wheel, counting from 0, and with small numbers on the second half of the wheel (and reversely for Black).

Summary

In this chapter, we have examined a number of possible ways to explore data visually. We have used the flexibility of R to produce samples programmatically to generate data on the fly, which we have used to illustrate how to use basic plots. We have examined some of the associations in the game of roulette and developed functions according to our analytical needs. We have also examined how to recode data, and use only a subset of data, and introduced the concept of multiple sampling.

The next chapter will deal with more advanced graphs using the `lattice` package. This comes in handy when dealing with several group memberships (for example, ethnicity and gender at once).

3
Data Visualization with Lattice

In the previous chapter, we discovered how to easily visualize our data using standard functions of R. You might remember that at some point, when discussing bar plots, we visualized the frequency of an attribute based on the case's membership a group. This required that we generated several plots, each displaying the data in one of the groups. Dealing with this kind of issue more easily is mostly what *trellis* graphics are about.

Trellis graphics allow the visualization of data based on group membership effortlessly. With features such as *multipanel conditioning* (Becker & Cleveland, 1996, p. 6), understanding the structure of your data becomes a seamless visualizing experience.

Multipanel conditioning means that data is displayed on multiple panels which are defined as a function of group membership. It is particularly interesting when membership to several characteristics are involved (for instance age group and gender). In these cases we are confronted with multivariate data. An interesting property of trellis graphics is that they are objects, which can be assigned a name, copied, and most importantly modified on the fly. We will discover these aspects in this chapter, as well as several types of useful plots.

Trellis graphics were introduced in the S language in the 1990s (Cleveland, 1993). The `lattice` package is the implementation of trellis graphics in R. It is now part of the list of packages that comes out-of-the box with R.

Loading and discovering the lattice package

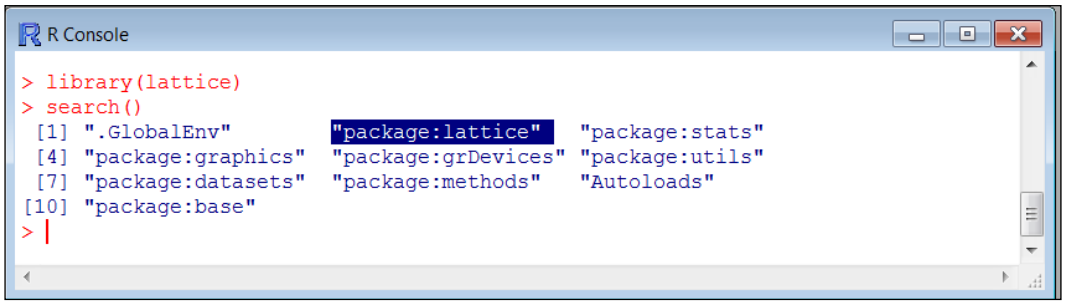
The `lattice` package is included in R version 3. We will first load the `lattice` package with the command line:

```
library(lattice)
```

We can now have a look at the objects that are included in the package. This requires listing the loaded packages, which can be done by typing:

```
search()
```

The output is displayed on the image below:



Packages in the R search path

The `Lattice` package is number 2 in this list. This position can vary, depending on whether you have loaded other packages as well.

The `ls()` function will allow us to inspect the content of `lattice` (or any other loaded package), as we have seen in *Chapter 1, Setting GNU R for Predictive Modeling*. Because the `lattice` package is in second position as just discussed, we type:

```
ls(2)
```

The output, composed of approximately 150 elements, is too long to be printed here. This illustrates the many graphing possibilities offered by lattice. We suggest you go through it on your screen. The idea is to get the feel of the content of the package by exploring it. If an object name seems intriguing, simply type its name, preceded by a question mark. This will launch the HTTP help server and provide you with information for that particular function. We will just comment on some elements of the package. There are some functions that produce graphic objects, such as `barchart()`, `bwplot()`, `cloud()`, `histogram()`, `parallel()`.

There are also functions that add elements, such as points, lines, other shapes or text, to an object (during or after its creation). These include `llines()`, `lpoints()`, `ltext()`, and `lrect()`. Another type of important object of the `lattice` package are functions that configure panels, that is, the area that contains the visualizations, or add elements to panels on the fly. These functions start with the prefix `panel`. While some work with most `lattice` plots, others are specialized. We will only have a look at some of the features of `lattice` in this chapter.

We will not discuss all functions and arguments here. The interested reader can consult the package documentation available at:

`http://cran.r-project.org/web/packages/lattice/index.html`

Discovering multipanel conditioning with xyplot()

The first thing we will do next is to check that `lattice` works properly. For this example, we will use the `iris` dataset. The `iris` dataset is one of the best known in data science. It is composed of four numeric attributes `Sepal.Length`, `Sepal.Width`, `Petal.Length` and `Petal.Width` which are measures of iris plants, as well as a factor, or nominal attribute `Species` which describes the membership of the plants to 3 different iris species: Virginica, Setosa and Versicolor. The data set is composed of 150 observations.

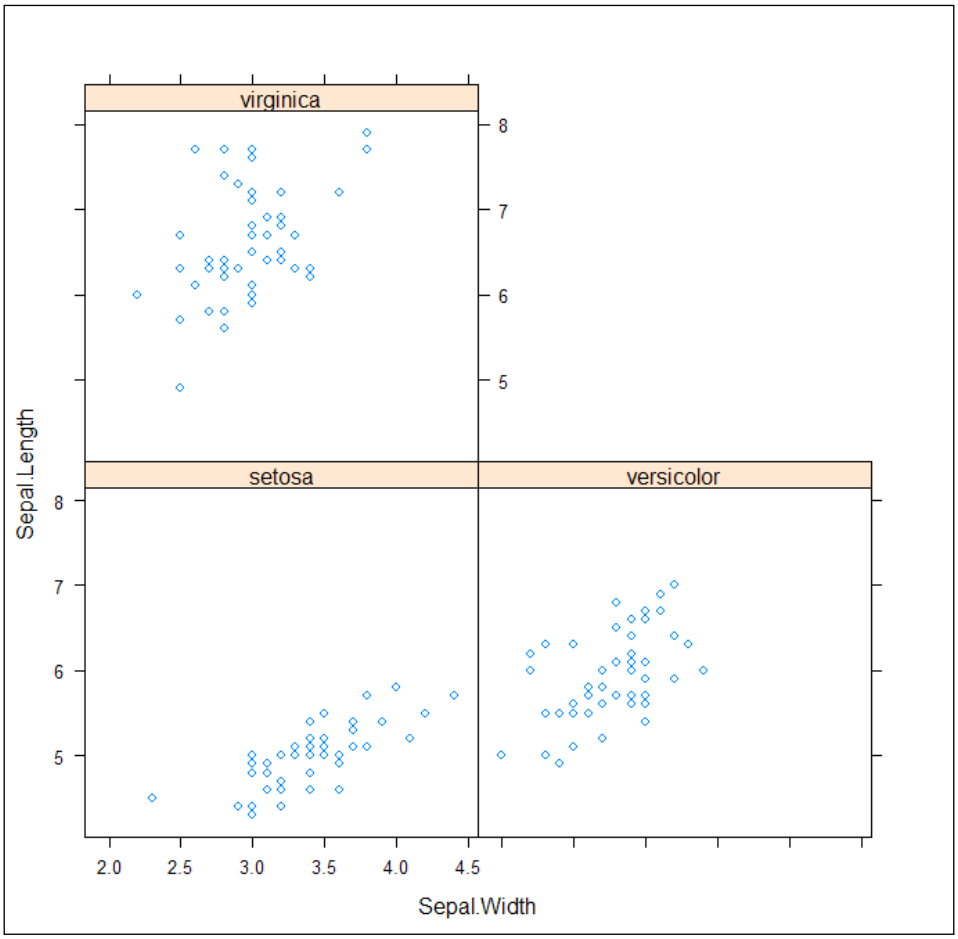
Doing this first plot will also allow us to discover the formula syntax used in most plots with `lattice`:

```
xyplot(Sepal.Length ~ Sepal.Width | Species, data = iris)
```

In this line of code, we have used the `xyplot()` function to visualize the relationship between the sepal length and the sepal width of iris flowers conditioning on `Species`. This means that one scatterplot is produced for each of the groups. The function `xyplot()`, as well as most functions in the `lattice` package uses a formula syntax, similar to what we will use when discussing regressions. So here is a quick word about the formula syntax. The formula part of the line of code above is `Sepal.Length ~ Sepal.Width`. The `~` operator (tilde or wavy dash) is used to parse the left-hand side of the formula `Sepal.Length` here, with what we want to visualize in the *y* axis (or what we want to model in the case of regression for instance), from the right-hand side of the formula, where we tell R the attributes (`Sepal.Width` here) what we want to use on the *x* axis (or the predictors in a regression model).

Values on the left-hand side are usually vectors, and those on the right-hand side are usually vectors or matrices. We specify the dataset to be used after a comma, using the argument `data`. We will use more complex formulae later, including the case of formula with no left-hand side.

The vertical bar symbol `|` is a requirement of `xyplot()`. It means that the display is conditioned on the grouping attribute that follows (here the attribute is `Species`). The plot is reproduced in the figure below:



Visualizing the features of the iris dataset conditioning on a group

Discovering other lattice plots

We have just discovered one type of plot in lattice as well as multipanel conditioning. Lattice is a rich package which features diverse plots. We have already encountered the multi-paneled scatterplot obtained using `xyplot()`. We will have a look at some more lattice multi-paneled graphs in this section: histograms, stacked bars, dotplots, as well as a customization of the scatterplot, where points are replaced by text.

Histograms

In the previous chapter, we examined the overall distribution of an attribute using the `hist()` function. The distribution of some measures can vary between groups, that is, it can be more or less skewed in some groups compared to others. The `histogram()` function in the `lattice` package allows for a visual inspection of this. We will examine variability in temperatures by month using the `airquality` dataset. This dataset has six attributes (`Ozone`, `Solar.R`, `Wind`, `Temp`, `Month`, and `Day`), of which you will find a description by typing:

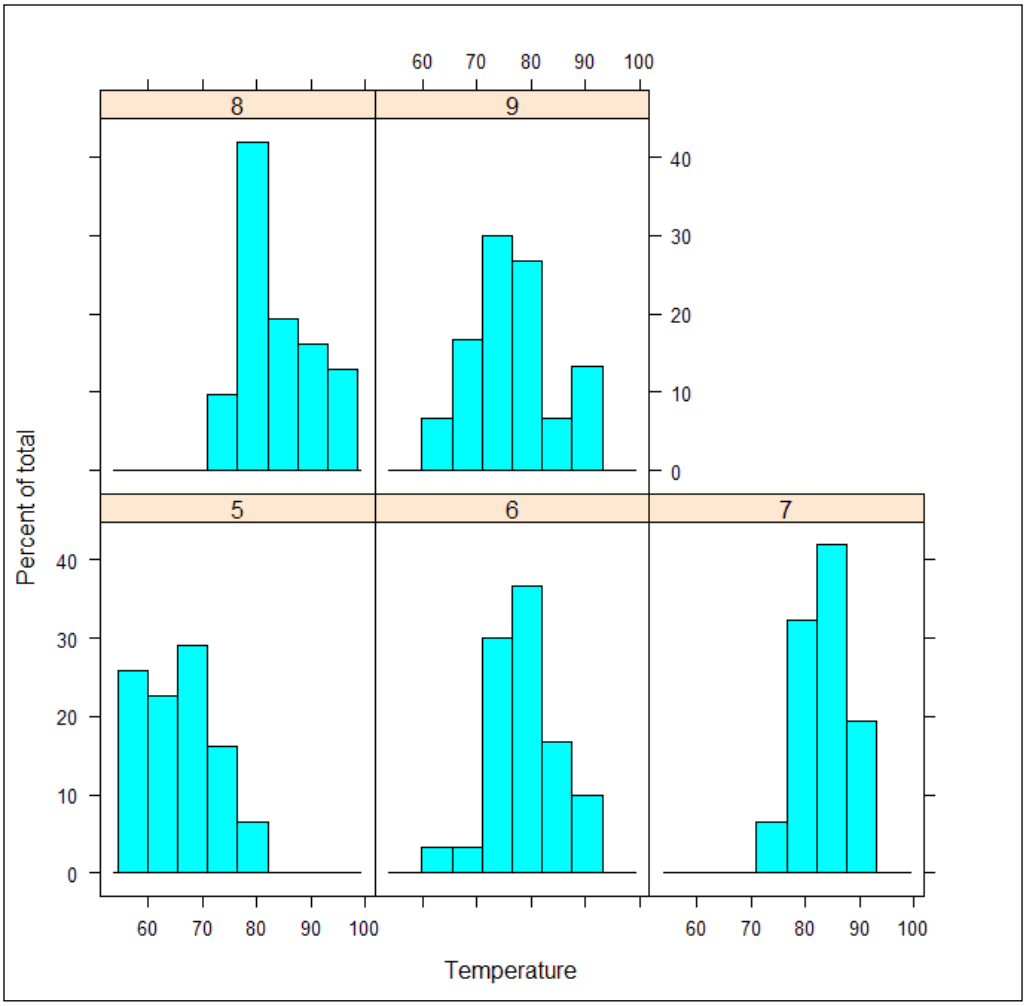
```
?airquality
```

To generate a `lattice` graphic with a histogram of `Temp` for each month, type:

```
histogram(~Temp | factor(Month), data = airquality,
   xlab = "Temperature", ylab = "Percent of total")
```

Not using a left-hand side in the formula above has the effect of plotting the values of the right-hand side on the *y* axis, instead of those of another variable, as in the previous example. Notice we use the `factor()` function here. This function allows us to tell R that we want to consider the values in the variable `Month` as categories, instead of as quantities as it would have by default.

The effect is that the plot is produced for each month (from May to September). The output is provided below. We can notice that the temperature increases from May to July and then decreases:



Histograms of temperature by month

Stacked bars

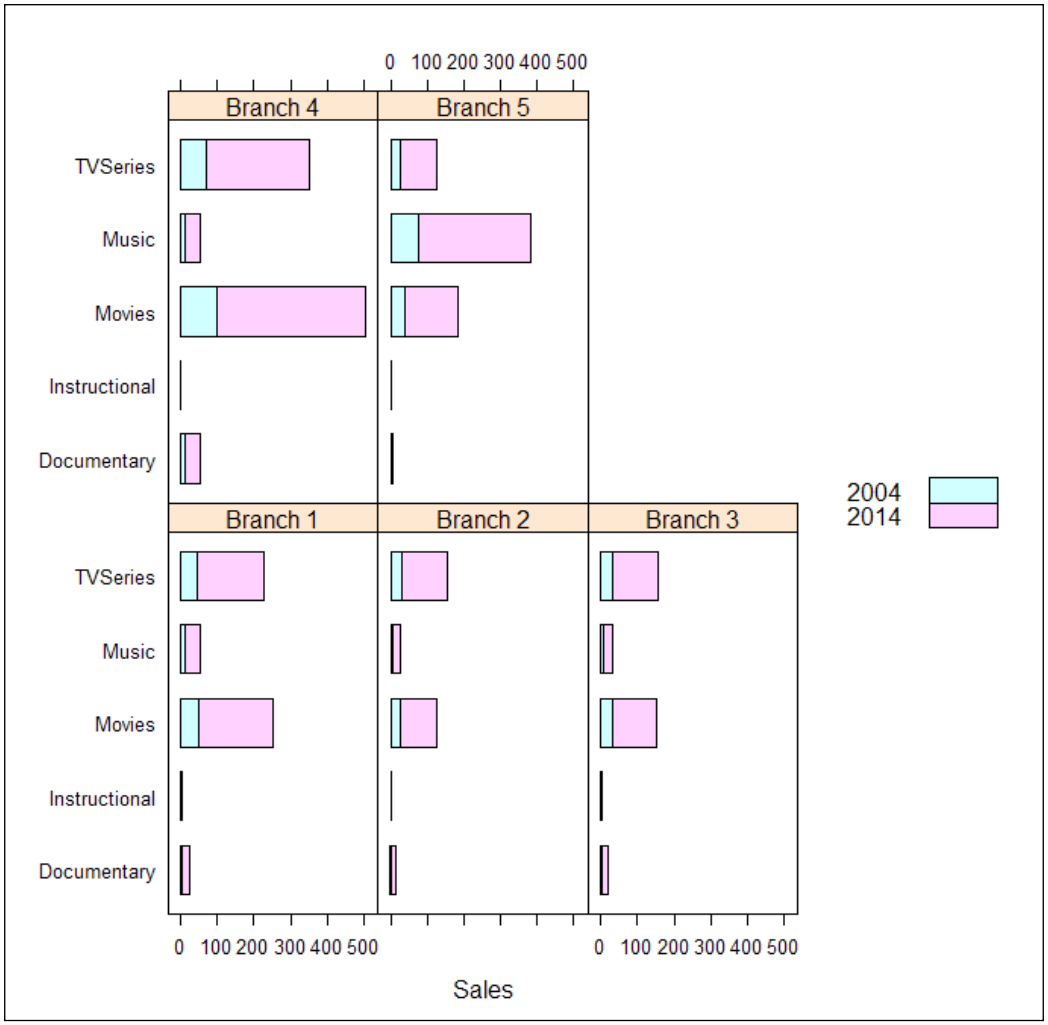
Stacked bar graphs are very useful representations of multiway data. We call multiway data in which 3 or more factors are plotted or analyzed together. In this example, we will use fictitious sales data from a company specializing in selling DVDs, and Blu-ray discs in 2006. The company has 5 branches and has 5 departments in each branch: Movies, TV series, documentary, music and instructional. The following code will create the `salesdata` data frame containing sales record for years 2004 and 2014 (in hundreds of thousands). We will then examine the sales as a function of branches, departments and year.

```
5    "Branch 3", "Branch 4", "Branch 5"), 5)
6 salesdata$Dept = c(rep("Movies",5),rep("TVSeries",5),
7    rep("Documentary",5),rep("Music", 5), rep("Instructional", 5))
8 salesdata$Sales = c(50.795, 25.469, 30.241, 100.658, 36.412,
9    45.632, 30.541, 31.421, 70.212, 25.412, 5.124, 3.124, 4.065,
10  10.258, 0.82, 10.658, 5.474, 6.541, 10.698, 76.584, 1.021,
11   0.504, 0.76, 0.15, 0.3, 203.18, 101.876, 120.964, 402.632,
12   145.648, 182.528, 122.164,125.684, 280.848, 101.648, 20.496,
13   12.496, 16.26, 41.032, 3.28, 42.632, 21.896, 26.164, 42.792,
14   306.336, 4.084, 2.016, 3.04, 0, 0)
```

On line, we build a matrix of 50 rows and 4 columns filled with zeroes, and coherce it to a data frame before populating it. As a reminder, a data frame is a list of vectors that can be of different types but all of the same length. A matrix can only contain elements of the same type. On line 2, we name the columns of the data frame. We will have attributes `Year`, `Branch`, `Dept` (for department) and `Sales` (for sales volume). From line 3 to 12, we populate the data frame.

The following code will produce a stacked bar chart of the data:

```
barchart(Dept ~ Sales | Branch, groups = Year, data = salesdata,
   auto.key = list(space = 'right'), stack = TRUE)
```



A stacked bar chart of yearly sales by department and branch

We used the `barchart()` function to generate the graph. In the formula argument, we included the department (attribute `Dept`) on the left as we want it to be displayed on the *y* axis, we want the sales (attribute `Sales`) on the *x* axis, so we put it second (after the tilde `~` symbol). We continued our formula by stating that we want the graph to be conditioned on `Branch` (after the vertical line which means conditioned on), we want each panel to discriminate between the years, so we assigned `Year` to the `groups` argument. We asked for stacked graphs using the `stack` argument, and finally we asked for the key of the graph to be placed on the right using the `auto-key` argument. The `stacked` argument allows specifying that we want a stacked bar graph.

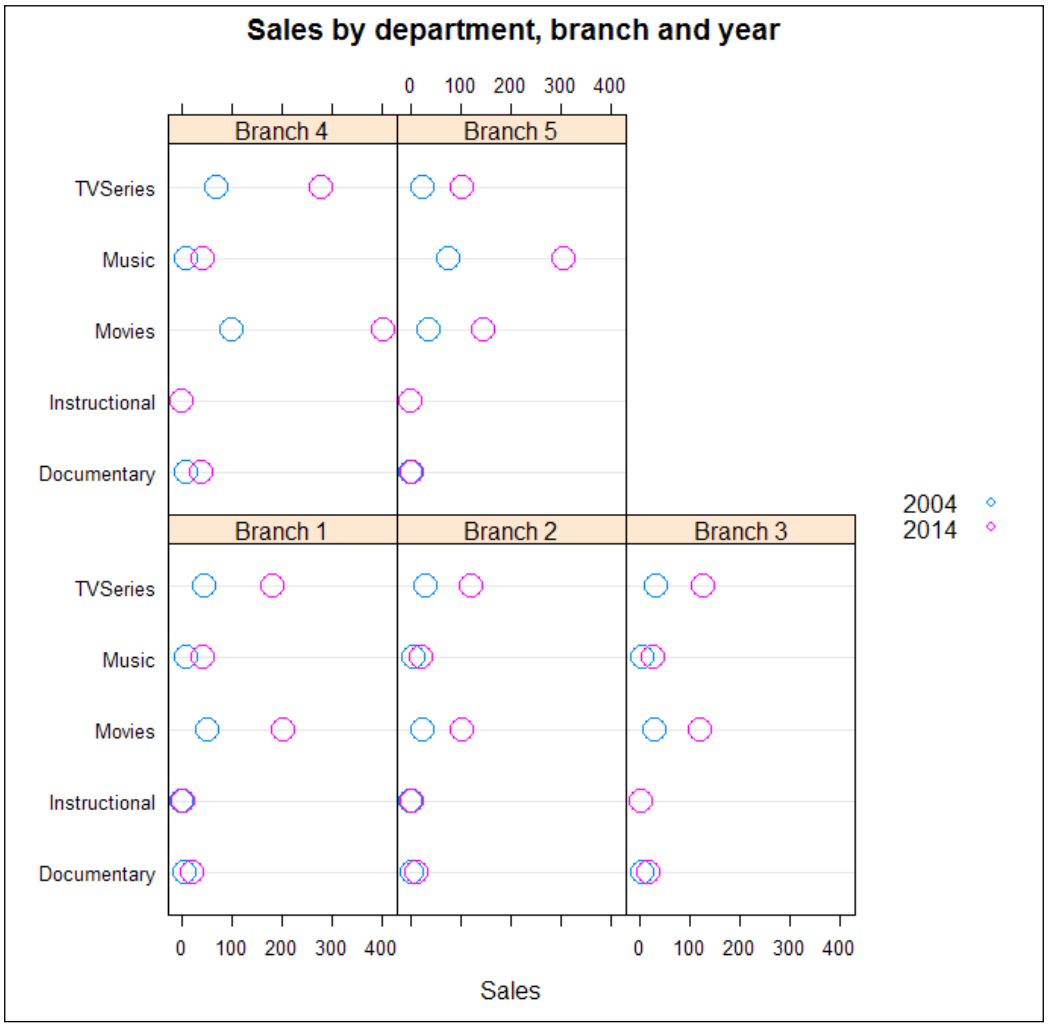
On this graph, we can see that the sales were much higher in 2014 than in 2004. Whereas branch 4 increased mostly in movies and TV series sales, branch 5 became apparently specialized in musical DVDs and Blu-rays and generated an increased income in this department. Branches 1, 2 and 3 were not as lucky and didn't increase their sales as much, but still managed to survive the crisis.

Dotplots

The `dotplot()` is a useful graphing function in R. It allows for the representation of the relationship between a numeric attribute and one or more factor attributes. We will use the `salesdata` dataset again to illustrate the use of the `dotplot()` function. We will reuse the formula we used for producing the stacked bars chart, as well as most of the argument assignations. This is possible because the same options can be used for generating most `lattice` objects. We will also add a title using the argument `main`, and increase the size of the dots a bit using the argument `cex`:

```
dotplot(Dept ~ Sales | Branch, groups = Year, data = salesdata,
   cex = 2, auto.key = list(space = "right"),
   main = "Sales by department, branch and year")
```

The following is the output:



A dot plot of yearly sales by department and branch

We can interpret this graph the same way we did for the stacked bar graph, as it represents the same data. It is sometimes useful to use multiple visualizations for a better understanding of the data.

Displaying data points as text

An interesting feature of `xyplot()` is the possibility to display data point as text in multi-paneled scatterplots conditioned on one attribute or the combination of attributes. In what follows, we will examine the relationship between `fertility` (*y* axis) and `education` (*x* axis) in Swiss districts. We will use multi-paneled scatterplots conditioned on high versus low infant mortality and whether the district is rural versus non rural. We will display the observation as the name of the district.

The following code starts by creating a new data frame from the `swiss` dataset (line 1). We then add three additional attributes. The first is `Mortality`, which is computed as whether `Infant.Mortality` is higher than the mean value across the dataset (lines 2 to 5). Remember that in the *Chapter 2*, *Visualizing and Manipulating Data Using R*, we used the `subset()` function combined with `if` statements to subset data based on a condition. Here we rely on an alternative solution by including the condition into brackets. For instance, lines 2 and 3 mean `fertility$Mortality` (the attribute `Mortality` of data frame fertility) takes the value `High infant mortality` in observations where `fertility$Infant.Mortality` is higher than the mean of `swiss$Infant.Mortality`.

The second additional attribute is `Rural`, which is computed as whether `Agriculture` is higher than the mean value across the dataset (lines 6 and 9). Finally, the attribute `District` (that is, the district where the data was collected). This is simply the row names of the dataset (line 10). After this initial part, attributes `Mortality` and `Rural` are recoded in order to make the values understandable in the graph. Here is the code for this data preparation:

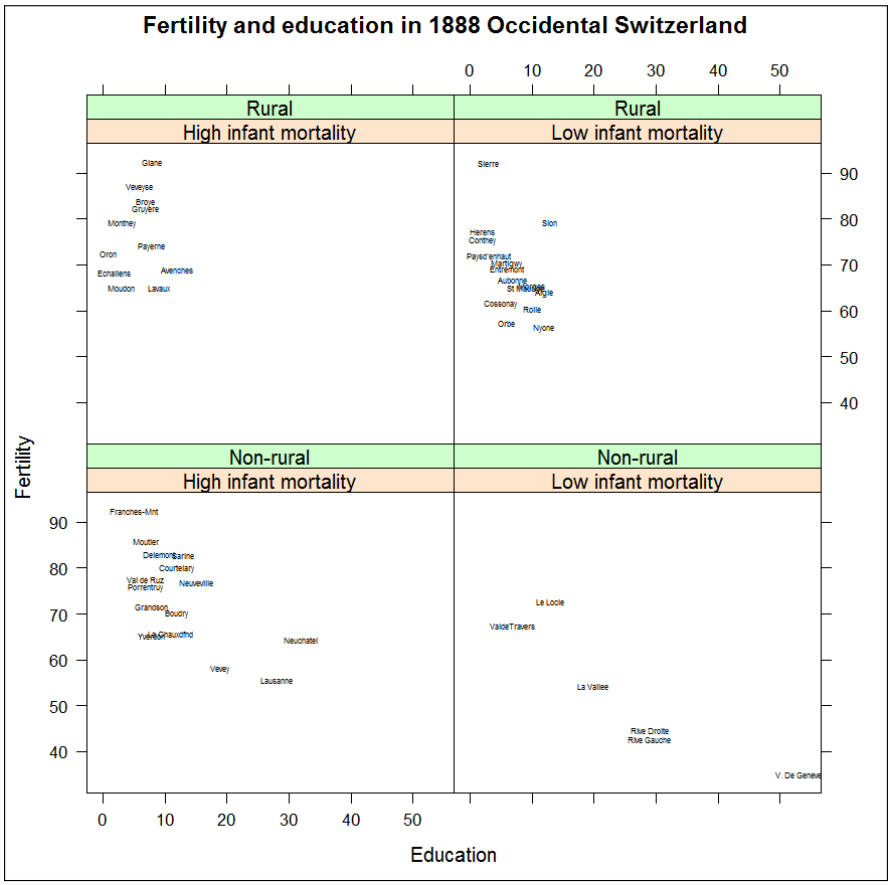
```
1  fertility=swiss
2  fertility$Mortality[(fertility$Infant.Mortality >
3     mean(swiss$Infant.Mortality))==TRUE]="High infant mortality"
4  fertility$Mortality[(fertility$Infant.Mortality >
5     mean(swiss$Infant.Mortality))==FALSE]="Low infant mortality"
6  fertility$Rural[(fertility$Agriculture >
7     mean(swiss$Agriculture)) == TRUE] = "Rural"
8  fertility$Rural[(fertility$Agriculture>
9     mean(swiss$Agriculture)) == FALSE] = "Non-rural"
10  fertility$District = rownames(fertility)
```

In the plotting section below, we first include the formula, that specifies which relationship between attributes to plot, including the conditioning on `Mortality * Rural` (the combination of which will result in 4 panes; line 1), we then configure the groups (line 2). An important part of the graph is the configuration of the panel (line 3 and 4).

We create a panel function in which we call `ltext()`, which allows printing text and lattice graphics, in this case, the district names. This replaces the default printing of data points as dots. We then configure the main title of the graph (line 5):

```
1  xyplot(Fertility ~ Education | Mortality * Rural, data =
   fertility,
2     groups = as.character(District),
3     panel = function(x, y, subscripts, groups) {
4        ltext(x, y, labels = groups[subscripts], cex=.4)},
5        main = "Fertility and education in 1888 Occidental
         Switzerland")
```

On the resulting graph, presented in the figure below, we can notice the relationship between education and fertility, especially in non-rural areas. We can also see that fertility is lowest in non-rural areas where infant mortality is low:



A multi-panel plot of the relationship between fertility and education, conditioning on infant mortality and agriculture

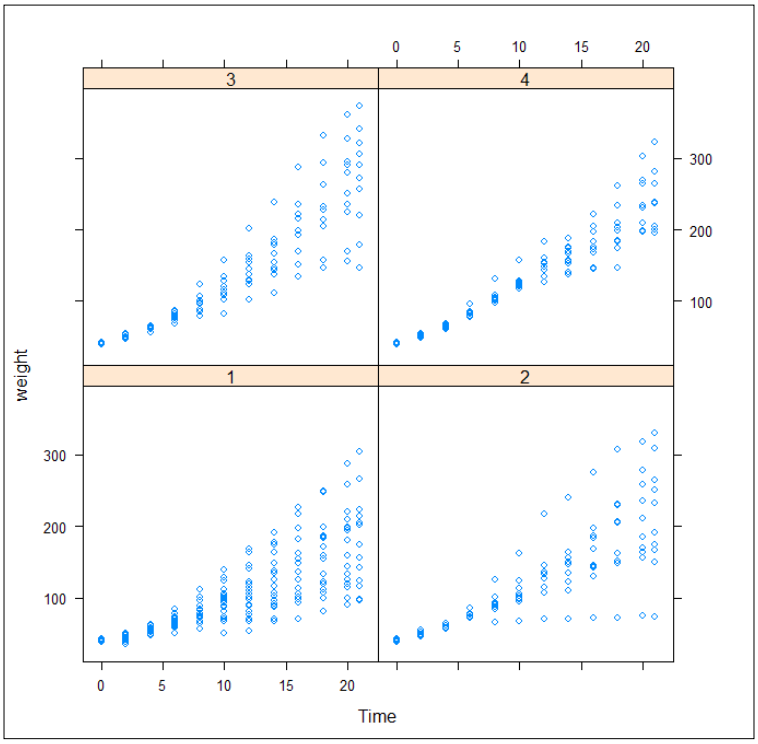
Updating graphics

We have seen how easy it is to create multi-panel plots using `xyplot()`, `barchart()` and other functions of lattice. We mentioned in the introduction that `lattice` graphics can be customized on the fly. Yet, we haven't taken that opportunity yet. In this section, we will see how `lattice` plots can be customized using the `update()` function. Note that the customizing can be done when creating the object as well. We will use another simple dataset for this purpose: the `ChickWeight` dataset. This dataset contains four attributes which describe the growth of chickens through time, with several diets; `weight`: the weight of the chicken, `Time`: the age of the chicken, `Chick`: the identifier of the chicken, and `Diet`: how the chicken was fed.

Our interest will focus on the relationship between the diet and growth (variations in weight through time).

```
xyplot(weight ~ Time | Diet, data=ChickWeight)
```

As can be seen in the figure below, chickens increase in weight through time, with all diets. It can be noticed that there is a huge variation in growth with diet 1 (bottom left of the graph), whereas diet 4 has the least variation in growth.



Chicken growth as a function of diet

We also want to see if there are individual differences in chickens, that is, how much this varies between chicks. More precisely, we will plot the data for each diet on a separate panel, and the data for each chick with a different representation (the color of the observation). We will do this using the argument groups and assign a name to the graph (the displaying of the graph will be differred until we call it by name):

```
Graph = xyplot(weight ~ Time | Diet, groups = Chick,
   data=ChickWeight)
```

Relying on the `update()` function with the name of our graph as a first argument (line 1), we add a customized title for the graphic (line 2), as well as for the *x* axis and *y* axis (lines 3 and 4):

```
1  Graph = update(Graph,
2     main = "Chicken growth by diet",
3     ylab = "Weight of the chicken",
4     xlab = "Days since birth")
```

We also want to add an index for each chicken, for instance in order to find them in our dataset more easily later on. We also want the panels to be displayed in the opposite order (diet 1 first). Note that the order of the panels can be fully customized.

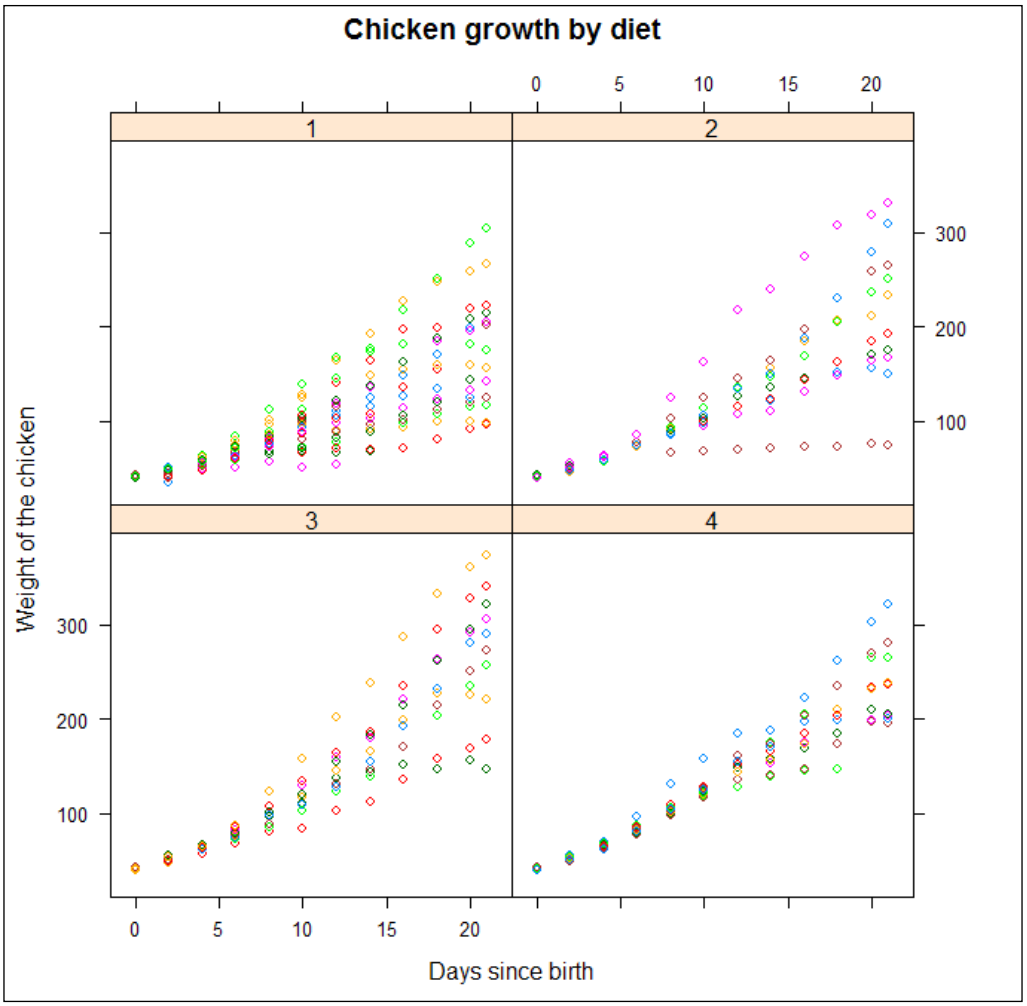
```
Graph = update(Graph, index.cond = list(c(3,4,1,2)))
```

We can now plot the graph by typing its name:

```
Graph
```

Or using the function `print()` with the name of the graph as argument:

```
print(Graph)
```



Chicken growth as a function of diet (showing values for individuals)

The resulting graph is presented on the image above. We can now notice that within each type of diet, some chickens present a lower growth than others. Other aspects of graphs can be freely customized. We will see some others more in the following pages when discussing the thorough visual inspection of a particular dataset. For now, simply type the following line in order to discover all options for `xyplot()` – most options also apply to the other types of graphs we have encountered:

```
?xyplot
```

Case study – exploring cancer-related deaths in the US

In this section, we will explore the rate of death due to cancer in the US. This will allow us to have a better feel of the importance of plotting data using a practical example. We will also discover interesting tools along the way. We will start by discovering the dataset, plotting some data, integrating data from other sources.

The dataset is part of `latticeExtra`, so we will install and load it first:

```
install.packages("latticeExtra")
library(latticeExtra)
```

Note that this may update `lattice` and require to restart R.

Discovering the dataset

Let's start by having a look at the attributes in the dataset and 3 rows of data:

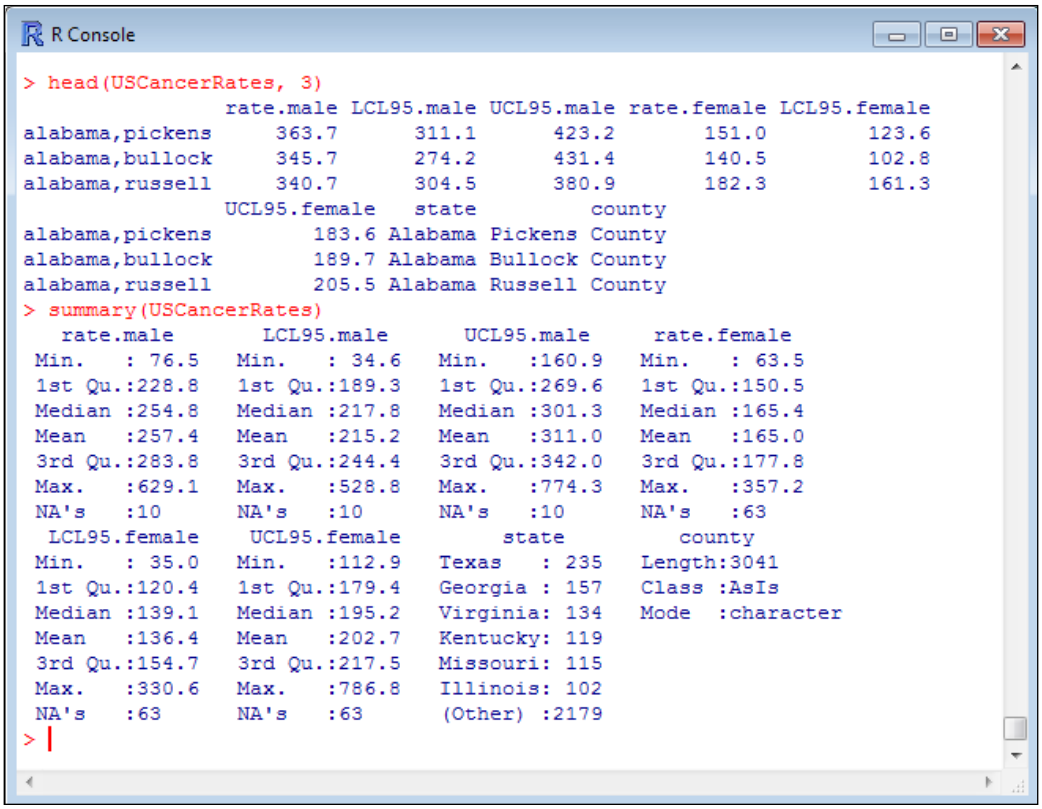
```
head(USCancerRates, 3)
```

This outputs, included in the figure below, shows that there are 8 attributes, which relate to the rate of death due to cancer for males and females as well as the 95% confidence intervals (lower and higher bounds for both). Two other attributes allow identifying the county and state were the data of each row comes from.

Let's now have a look at some global descriptive statistics:

```
summary(USCancerRates)
```

The output is included below. We can see that the rate for males (attribute `rate.males`) is apparently lower than the rate for females (attribute `rate.female`). The 6 states with the most counties are Texas, Georgia, Virginia, Kentucky, Missouri and Illinois. These represent about a third of the dataset: there are 3,041 counties altogether (attribute `county`).

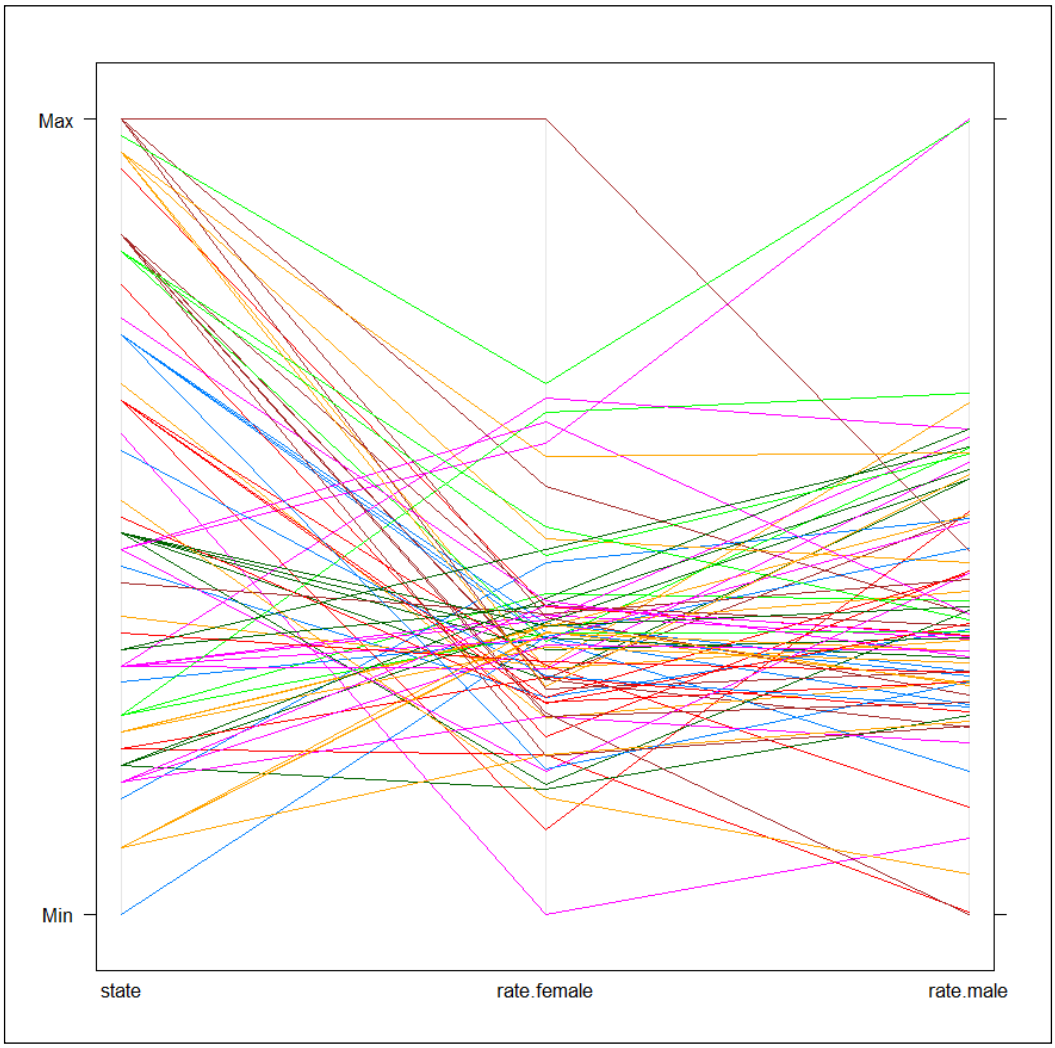


The first three observations of the data set and its summary

We will now produce a plot we haven't encountered yet in order to examine the relationship between deaths of males and females caused by cancer; the parallel coordinates plot. The plot on the whole dataset would be unreadable because there are many rows. For this reason, we will create the plot only on the basis of a subsample of 75 randomly drawn counties. We also want to discard the confidence intervals from the plot. For this reason, we will only consider columns (`male.rate`), 2 (`female.rate`) and 7 (`state`), which we will reorder (`state` first). After obtaining the subsample, we will plot it using a parallel plot. We will set the `horizontal.axis` argument to `False` in order to have the values in the *y* axis and the attribute names on the *x* axis:

```
set.seed(987)
subsample=USCancerRates[sample(1:nrow(USCancerRates),75),c(7,4,1)]
parallelplot(subsample, horizontal.axis=F, groups=subsample$state)
```

The following is the output:

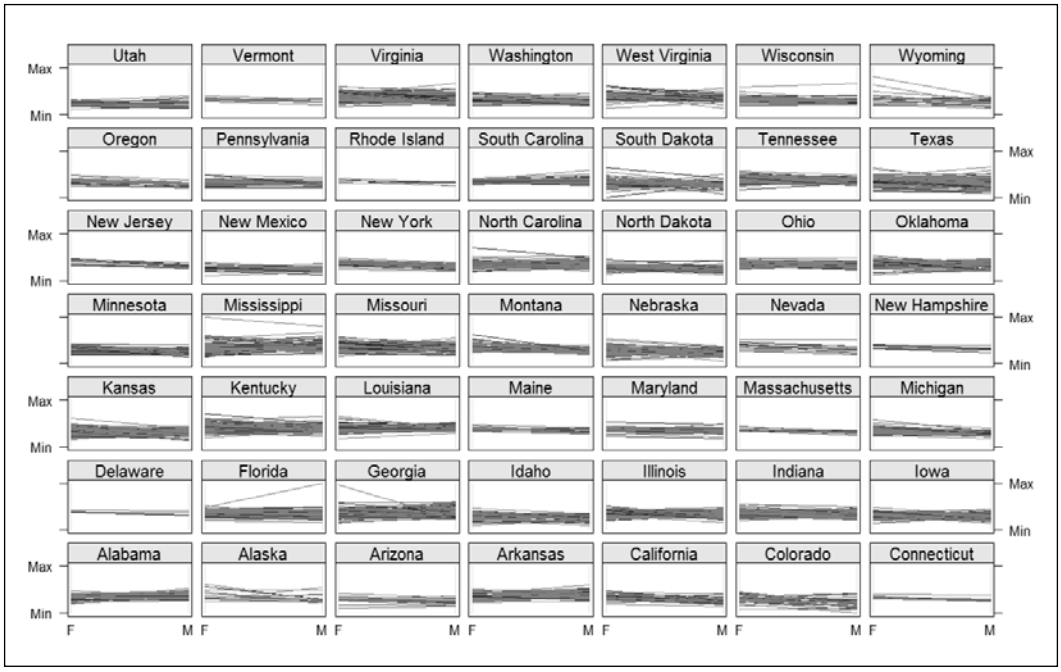


The parallel coordinates plot

The parallel coordinates plot allows us to have a look at the relationship between several attributes at the same time. You could do this for instance for 5, 10, 20 or even more attributes, plotted at the same time, and at the case level, that is, each observation is plotted in the parallel coordinates plot. We can notice that for some counties, the rates are higher for males, and for some others for females. Is this related to the state? We will now produce the same plot, but separately for each state. We will use the full dataset this time. This will require changing our call to the function a bit:

```
parallelplot(~USCancerRates[,c(4,1)]|USCancerRates$state,
   horizontal.axis=F,data=USCancerRates, varnames =c("F","M"))
```

From the graph, it can be noticed that there this is also much variation of some states: in some states, females' rates are higher than males', in other states, it is the opposite, whereas, most states feature a variety of cases. But as we have seen in the descriptive section, males' average rate seems higher:

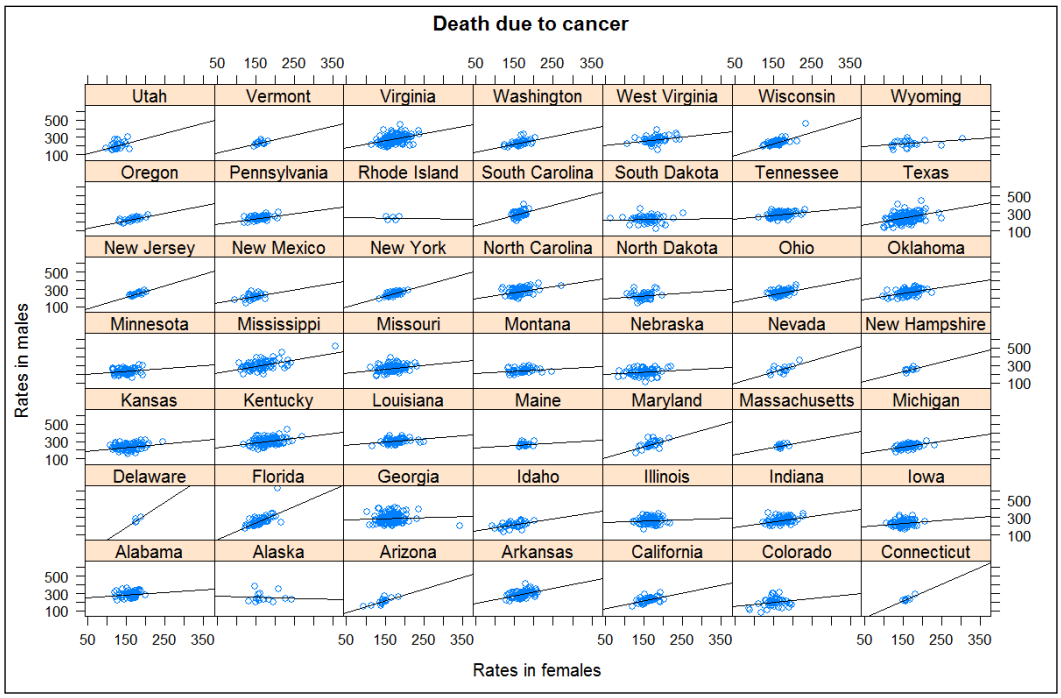


A parallel coordinates plot of rates of death due to cancer, by state

Another interesting question is whether the rate in females is related to the rate in males, that is, irrespective of which group has the highest rate, do females have higher rates in states where males have higher rates? We touched that topic before, but didn't examine the issue specifically. What we need to do now is to produce a scatter plot by state of the relationship between males and females and add a fit line in order to notice the difference. We do so by declaring a panel function (lines 5 to 7) and specifying (on line 7) that the regression line should be printed. This is very similar to what we have done in *Chapter 2*, *Visualizing and Manipulating Data Using R*. We will use `xyplot()` again. A more sophisticated version of this graph has been proposed in *Lattice: Multivariate Data Visualization with R*, *Deepayan Sarkar (2008)*, which we highly recommend reading:

```
1  xyplot(rate.male ~ rate.female | state,
2     data=USCancerRates, main = "Death due to cancer",
```

```
3      ylab= "Rates in males",
4      xlab = "Rates in females",
5      panel = function(x, y, ...) {
6         panel.xyplot(x,y)
7         panel.abline(lm(y~x))
8      }
9  )
```



Scatterplots of male and female rates of death due to cancer, by state

As we can see from the graph, there is usually a positive relationship between the rate of females and the rate of males. This means that in counties where 'male' rates are high, 'female rates are high as well.

Integrating supplementary external data

Something we might be interested in knowing is whether the percentage of individuals without health insurance is predictive of the rates of deaths due to cancer, and if this varies geographically.

We might hypothesize, for instance, that people without insurance seek less treatment and for this reason could be more likely to die from cancer. Testing this requires adding this attribute to the dataset. Information about health insurance by state is readily available from Cambell (1999), and represents the average for years 1996 to 1998.

```
NoInsur = c(15.1, 16.3, 24.3, 21.6, 21.2, 15.6, 11.8, 13.7,
   18.7, 17.6, 17.3, 12.9, 12.1, 11, 11.1, 14.8, 19.8,
   13.2, 13.8, 11.8, 11.2, 9.6, 19.6, 12.1, 17.6, 10.4,
   18.1, 10.9, 16.5, 22, 17.2, 15.5, 13.1, 11.1, 17.7,
   14.3, 10, 10, 16.4, 11.9, 13.9, 24.4, 13.1, 10.1,
   13.1, 12.4, 16.5, 9.4, 15.3)
```

There could be geographical differences in rates of death due to cancer. For instance, counties from southern states are exposed to more sun rays, which are causes from skin cancers. We will consider the states' central longitude and latitude. These have been computed from the table *United States State* available at `http://www.ala.org/magirt/publicationsab/usa`. In order to import this in R, we start by creating a vector with the list of states. We then create two vectors, for the longitude (negative is West, positive is East) and the latitude (negative is South, positive is North):

```
state = c("Alabama", "Alaska", "Arizona", "Arkansas",
   "California", "Colorado", "Connecticut", "Delaware",
   "Florida", "Georgia", "Idaho", "Illinois", "Indiana",
   "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine",
   "Maryland", "Massachusetts", "Michigan", "Minnesota",
   "Mississippi", "Missouri", "Montana", "Nebraska",
   "Nevada", "New Hampshire", "New Jersey", "New Mexico",
   "New York", "North Carolina", "North Dakota", "Ohio",
   "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island",
   "South Carolina", "South Dakota", "Tennessee", "Texas",
   "Utah", "Vermont", "Virginia", "Washington",
   "West Virginia", "Wisconsin", "Wyoming")
```

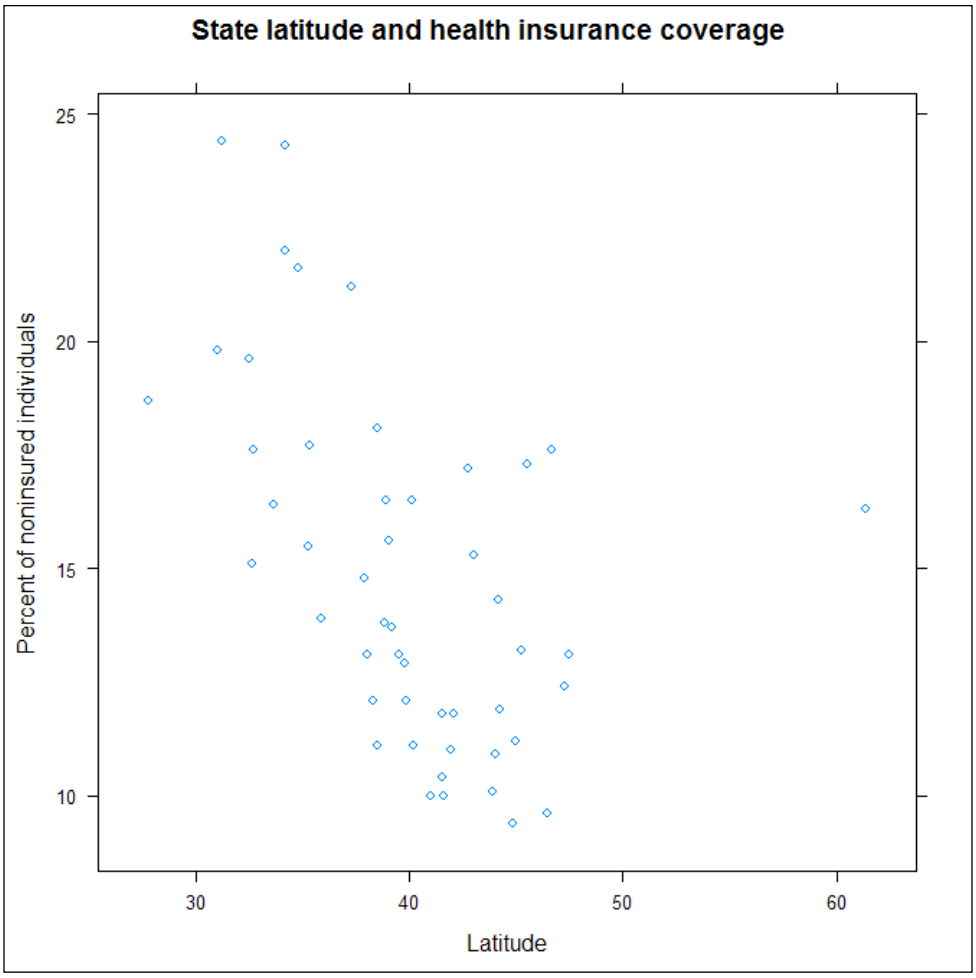
```
CentLong = c(-86.68, 21.5, -111.93, -92.12, -119.27, -105.56,
   -72.75, -75.40, -83.81, -83.18, -114.13, -89.50, -86.43,
   -93.37, -98.54, -85.77, -91.43, -69.01, -77.25, -71.72,
   -86.43, -93.38, -89.88, -92.44, -110.04, -99.68, -117.03,
   -71.58, -74.71, -106.03, -75.82, -79.88, -100.30, -82.67,
   -98.72, -120.52, -77.60, -71.52, -80.94, -100.24, -85.98,
   -99.58, -111.53, -72.53, -79.47, -120.84, -80.19, -89.83,
   -107.55)
CentLat = c(32.63, 61.38, 34.17, 34.75, 37.27, 39.00,
   41.53, 39.15, 27.75, 32.68, 45.50, 39.75, 39.81, 41.93,
   38.50, 37.88, 30.97, 45.22, 38.81, 42.04, 44.99, 46.44,
   32.50, 38.31, 46.68, 41.50, 38.50, 44.03, 40.14, 34.17,
   42.76, 35.23, 47.47, 40.20, 35.31, 44.13, 40.99, 41.58,
   33.61, 44.21, 35.83, 31.17, 39.50, 43.86, 38.00, 47.27,
   38.92, 44.81, 43.00)
```

Now let's group the four vectors in a data frame:

```
DF = data.frame(state, CentLong, CentLat, NoInsur)
```

We are now finally going to know if there is a relationship between the percentage of individuals without insurance and the central latitude of the state:

```
1  xyplot(DF$NoInsur~DF$CentLat,
2     main = "State latitude and health insurance coverage",
3     xlab = "Latitude",
4     ylab = "Percent of noninsured individuals")
```



The relationship between state latitude and percent age of non-insured individuals

We can notice a strong negative relationship is more in the North, the less is the percentage of noninsured people.

We now need a function to match the states' names in the `USCancerRates` dataset and the states' names in our dataset of supplementary attributes. On lines 1 to 3, we start by creating three vectors (filled with zeros for now). We then create a copy of `USCancerRates`, with these three additional attributes included (line 4):

```
1  CLong = rep(0,nrow(USCancerRates))
2  CLat = rep(0,nrow(USCancerRates))
3  NInsur = rep(0,nrow(USCancerRates))
4  cancer = data.frame(USCancerRates[c(7,1,4)],CLong, CLat, NInsur)
```

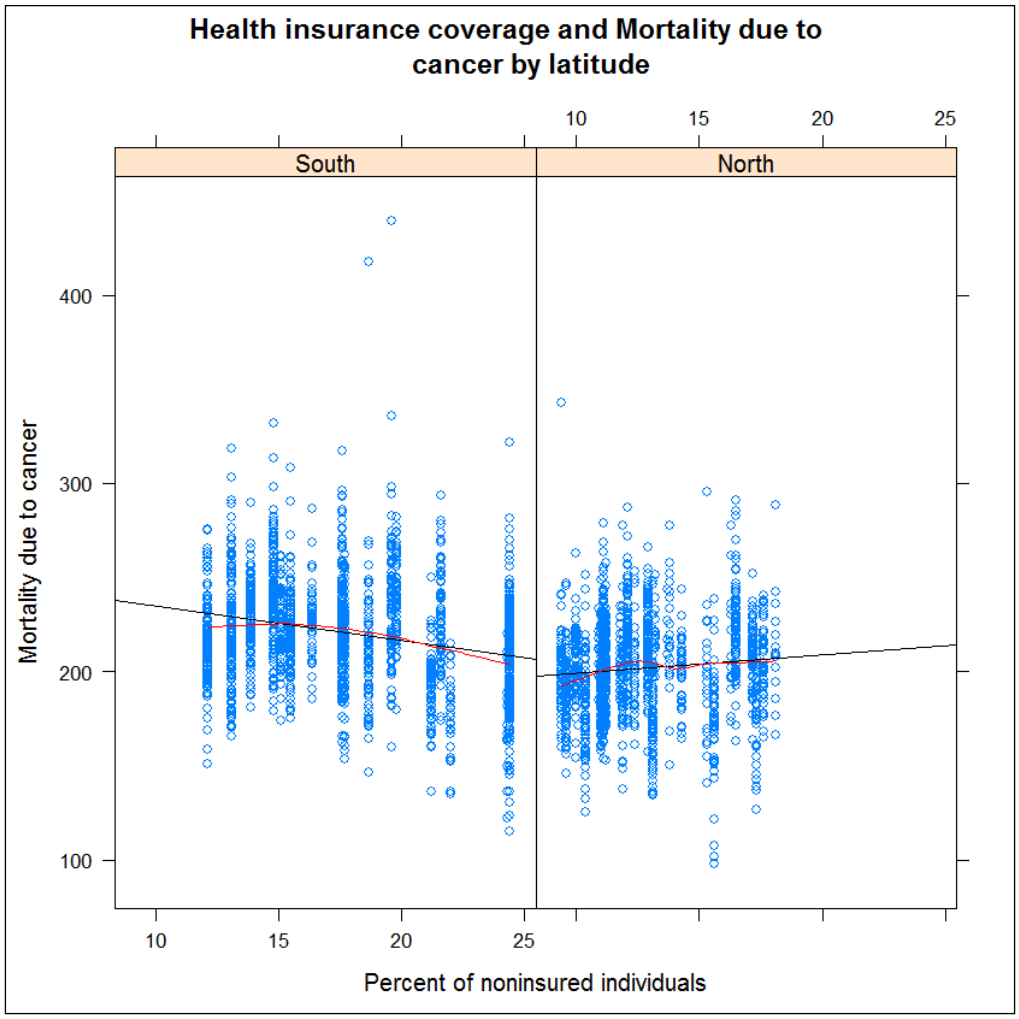
The following code will fill the three attributes with the correct values for us:

```
1  for (i in 1:nrow(cancer)) {
2     for (j in 1:nrow(DF)) {
3        if (cancer[i,1] == DF[j,1]) {
4              cancer[i,4:6] = DF[j,2:4]
5              next
6        }
7     }
8  }
```

Here we will loop through each row of the `cancer` data frame (line 1), and of the `DF` data frame (line 2). This is called *nested looping*. If the value of state on the current row (`i`) of the cancer data set corresponds to the current value of state (`j`) in the `DF` dataset (line 2), the values for row `j` of attributes `CentLong`, `CentLat` and `NoInsur` ([2:4]) from data frame `DF` will be copied to in attributes `CLong`, `CLat` and `NInsur` ([4:6]) of data frame `cancer` (line 4).

Now that we have our dataset ready, let's examine the relationship between healthcare coverage and the rate of male and female death due to cancer:

```
1  xyplot((rate.female + rate.male)/2 ~ NInsur |
2        ifelse(cancer$CLat>median(cancer$CLat),"North","South"),
3     data = cancer,
4     index.cond=list(c(2,1)),
5     panel = function(x, y, ...) {
6        panel.xyplot(x,y)
7        panel.abline(lm(y~x))
8        panel.loess(x,y, col = "Red") },
9     xlab = "Percent of noninsured individuals",
10     ylab = "Mortality due to cancer",
11     main = "Health insurance coverage and Mortality due to
12        cancer by latitude"
13  )
```



The relationship between health insurance coverage and mortality due to cancer

The attribute on the *y* axis is the average of female and male values for each state (see line 1). We group on latitude in order to examine if the pattern is different North and South. As `CLat` (the central latitude) is a numeric attribute, we split it in two with the `ifelse()` function, using the median value as a criterion (see line 2 below). This is a nice way to assign different values depending on a condition (we have discussed other ways above and before in this book) ! On line 4, we use the `index.cond` argument to determine the order of the panels (we want South first).

We added more information on the graph: the black line is the regression slope, configured with the call to the `panel.abline()` function (see line 7). The red curve is the `loess` function (see line 8), which fits to a polynomial by performing local regressions (we thank a reviewer for this added information). The result is that the line is locally smoothed, highlighting deviations from linearity. It is configured with the `panel.loess()` function. Note that these functions are embedded in the definition of the panel which takes place by assigning a new function to the panel argument (starting at line 5). This new function starts with a call to the default setting for `xyplot()`.

The graph shows that the relationship between the rate of death due to cancer and percent age of noninsured individuals is somewhat positive (although close to flat) in counties from northern states, whereas, unexpectedly, it is negative (and more pronounced) in southern states. This might be due to confound non-measured factors which might mediate or moderate the relationship between the number of noninsured individuals and cancer-related mortality. Globally, counties in northern states have a lower rate of death due to cancer compared to counties in southern states.

Summary

In this chapter, we have explored some of the functions of the `lattice` package. We have seen how easy it is to plot data as a function of groups. We have described some of the customization possibilities of `lattice` graphs. Through the example of the `USCancerRates` dataset, we have discussed the importance of data integration, which is merging data from several sources in order to build knowledge. In the process of exploring the functions of `lattice`, we have made some fascinating discoveries. In the next chapter, we will discover unsupervised clustering with `kmeans()`.

4
Cluster Analysis

Unsupervised cluster analysis refers to algorithms that aim at producing homogeneous groups of cases from unlabeled data. The algorithm doesn't know beforehand what the membership to the groups is, and its goal is to find the structure of the data from similarities (or differences) between the cases; a cluster is a group of cases, observations, individuals, or other units, that are similar to each other on the considered characteristics. These characteristics can be anything measurable or observable. The choice of characteristics, or attributes, is important as different attributes will lead to different clusters.

In this chapter, we will discuss the following topics:

- Distance measures
- Partition clustering with k-means, including the steps in the computations of clusters, and the selection of the best number of clusters
- Applications of k-means clustering

Clustering algorithms use distance measures between the cases in order to create these homogeneous groups of cases (we will discuss this next). It is therefore important to transform the data on all dimensions to a similar scale before performing partition clustering with tools such as `kmeans()`. This is important because the distances are computed from all the dimensions we consider. If one dimension has a range of values higher than the others (for example, values in centimeters compared to meters), differences in this dimension will be given much more importance compared to the others. This is also true for agglomerative clustering, with tools such as `hclust()`, or any algorithm using distance measures, such as nearest neighbor classification, which we will discover in an upcoming chapter.

There are several ways of scaling the dimensions for this purpose. Examples include:

- In case of comparable units (for example, centimeter and meter), the transformation of one metric to the other (for example, dividing the attribute measured in centimeters by 100).
- Normalizing, which is subtracting from each observation the lowest value and performing the division of the result by the difference between the maximum and the minimum. The mathematical equation is provided next:

$$\frac{x - \min(x)}{\max(x) - \min(x)}$$

- The use of `z` scores, which can be computed by subtracting the mean from the value of each observation, and dividing the result by the standard deviation. The mathematical equation is provided next:

$$\frac{x - \bar{x}}{\sigma(x)}$$

  This should be preferred for most cases, and can be done easily in R using the `scale()` function.

- When using scores based on textual data, one can compute the ratio of the frequency of appearance of each term (relative to the total number of words) in each document multiplied by the ratio of the total number of documents over the inverse of the number of documents featuring the considered term. This is known as the *tf-idf* ratio. In the equations, *t* is the term under consideration, *d* is a specific document, *D* is the corpus containing all documents, and *N* is the number of documents in *D*:

$$tf(t,d) = 1 + \log\big(f(t,d)\big)$$
$$idf(t,D) = \log\frac{N}{\{d \in D : t \in d\}}$$
$$tfidf(t,d,D) = tf(t,d)\, idf(t,D)$$

Distance measures

Partitioning clustering algorithms iteratively define *k* cluster centers and assign cluster membership (or the probability of group membership) to cases based on distances between the case and the cluster. Agglomerative clustering algorithms also create clusters based on distances, starting with each individual belonging to a separate cluster and the grouping clusters two by two. The k-nearest neighbors algorithm also uses distance measures.

Consider only one attribute, for instance the height of individuals. The distance of someone measuring 180 cm and someone measuring 170 cm will be 10 on this sole dimension considering the algebraic difference between the two measures as our distance metric. Things get a little more complicated when we add more attributes, such as `weight` (we will not consider variable scaling here). Let's say the first individual is clearly overweight (90 kg), and the second has a normal weight (80 kg). Considering only the sum of the difference between the measures as our distance metric, the difference between the individuals would be: *(180-170) + (90-100) = 0*. This clearly doesn't reflect the huge differences between these individuals; one is bigger and slimmer than the other. Several distance metrics are available. Here are some examples:

The metrics closest to the *sum of differences* measure we just examined are the Euclidean and the Manhattan distances.

The Manhattan distance sums the absolute value of the differences on all considered dimensions. Its mathematical equation is provided next:

$$\sum_{i=1}^{n} |p_i - q_i|$$

Take the case of our previous example, *abs(180-170)+abs(90-100) = 20*. For one dimension, it is equivalent to the difference between two observations: *abs(180-170) = 180-170 = 10*. The Manhattan distance can be selected in `hclust()`, which we will discover later, but not in `kmeans()`.

The Euclidean distance sums the squares of the differences and then performs a square root on the result. In case of only one dimension, the result is equal to the difference between observations. Its mathematical equation is provided below:

$$\sqrt{(\sum_{i=1}^{n} p_i - q_i)^2}$$

Considering our previous example, *sqrt((170-180)^2) = 10*, just as *180-170*. The Euclidean distance is the only distance available in `kmeans()` from the `stats` package (provided by default in R). We will only work with this function here (and one of our own in order to better understand how k-means works). But, in case you need to run k-means using other distances, you might be happy to know that the `Kmeans()` function from the `amap` package allows to select other distances.

The cosine similarity is another important distance (similarity measure). It is used in information retrieval and text mining. It is computed by performing the ratio of the dot product of the considered dimensions and the product of the square root of the sum of the squared values on the dimensions. The mathematical equation is provided below:

$$\frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

The `skmeans()` function from the `skmeans` package allows one to run k-means with cosine similarity as the distance metric, but not `kmeans()`, the standard implementation of k-means in R.

The correlation coefficient, an association measure, can also be used as a distance measure. Its mathematical formula is presented below.

$$\frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

The correlation coefficient can only be used with two dimensions.

The Jaccard index measures the similarity between two sets. It is used for categorical attributes. It is computed by dividing the number of elements that are common between two sets over the sum of common elements, elements only in the first set and elements only in the second set. For instance if A, B, C, and D are in set 1, C, D, E, and F are in set 2, and X, Y, Z, and A are in set 3, the similarity of set 1 and set 2 will be 0.33: *2/(2+2+2)*. On this metric, the similarity between set 1 and set 3 will be 0.14: *1/(1+3+3)*. Its mathematical formula for two sets (A and B) is presented below:

$$\frac{|A \cap B|}{|A \cup B|}$$

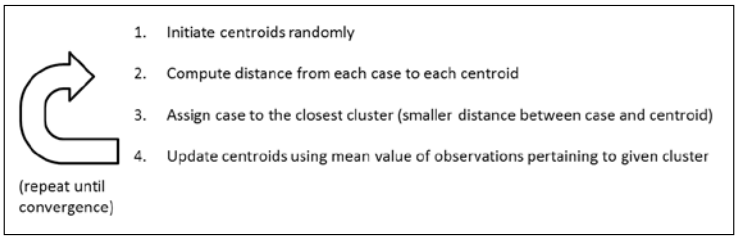
The Jaccard index is not readily accessible either in `Kmeans()`, but `hclust()` uses it as a custom distance matrix, which can be constructed in this case with the `vegdist()` function from the vegan package (along with other distance measures). Distance matrices for common distance metrics can be computed using the `dist()` function from the stats package (provided with R).

Readers interested in distance measures might find having a look at the *Encyclopedia of Distances*, by Michel Marie Deza and Elena Deza (Second edition, 2013) worthwhile.

Learning by doing – partition clustering with kmeans()

Perhaps the most widely used clustering family of algorithms is k-means. In this section, we will examine how it works and ways to assess the quality of a clustering solution.

K-means is a partitioning algorithm that produces `k` (user-defined number) clusters of cases that are more similar to each other than to cases outside the cluster. K-means starts by randomly initiating the centroid (the value of the considered dimensions) of each cluster. From now, the process, aiming at creating homogenous clusters, is iterative until a final solution is found. For each case, the distance from the centroid of each cluster is computed, and cases are assigned to the closest cluster. After this step, k-means computes the new values of the centroid of each cluster, as the means of all the cases belonging to the cluster. The process stops when the distance between the cases and the centroid is not decreasing anymore. It is noteworthy that the final result at convergence depends upon the initial random values of the centroids. For this reason, it is advocated to configure `kmeans()` to repeat the process several times and select the best solution (using the `nstart` argument). The figure below illustrates this process. We will only use the default algorithm with `kmeans()`, but the function provides several to choose from—Hartigan-Wong, Lloyd, Forgy, and MacQueen—using the algorithm argument. We will discuss this later. The interested reader can inquire about the differences using the references provided in the ad-hoc R manual page (type `?kmeans` in the console).



Steps in the k-means algorithm family

In what follows, we will build our own k-means implementation in order to better understand its functioning. There is a slight difference in our implementation in that we do not initiate the centroids randomly. Instead we will attribute membership to the clusters randomly, and then compute the centroids. This is equivalent, but easier programmatically. This implementation is intended only for pedagogical purposes. It lacks several important features, and sometimes converges non-optimally. For proper analyses, please use `kmeans()` or an implementation from another package available on CRAN.

Setting the centroids

First, we need a function that attributes membership to the clusters randomly. We do this for all the observations at the same time, and return the result as vector `clusters` (see line 2). Our function takes 2 arguments: the number of observations (`numrows`) and the number of clusters (`k`).

```
1  set.random.clusters = function (numrows, k) {
2     clusters = sample(1:k, numrows, replace=T)
3  }
```

We then create a function that computes the `centroids`—that is, the means for each case on each dimension and each cluster. Our function takes 2 arguments: the data frame on which to cluster the data (`df`), and the current cluster assignments (`clusters`).

```
1  compute.centroids = function (df, clusters) {
2      means = tapply(df[,1], clusters, mean)
3      for (i in 2:ncol(df)) {
4          mean.case = tapply(df[,i], clusters, mean)
5          means=rbind(means, mean.case)
6      }
7      centroids = data.frame(t(means))
8      names(centroids) = names(df)
9      centroids
10 }
```

On line 2, we assign to vector `means` the mean values of attribute in column 1 for each of the clusters. In a loop, we assign to vector `mean.case` the values of attribute in column `i` for each cluster, and append it to means (lines 4 and 5). We then make a data frame from the transpose of object means and assign it to object `centroids` (line 8) We name the columns of this object as the columns in the original data frame (line 9). Finally we return `centroids` (line 9).

Computing distances to centroids

We then need a function that computes the distance between the actual values and the centroid of the clusters. This function takes the data and the centroids as arguments. It first creates a blank matrix that will contain the distances (line 2). It then loops over the cases and the clusters (see the code blocks started on lines 3 and 4) and computes the squares of the differences for the current case and current cluster, computes the square root of those, and assigns the value to the correct spot in the matrix (line 5). It finally returns the matrix of Euclidean distances (line 8).

```
1  euclid.sqrd = function (df, centroids) {
2      distances = matrix(nrow=nrow(df), ncol=nrow(centroids))
3      for (i in 1:nrow(df)) {
4          for (j in 1:nrow(centroids)) {
5              distances[i,j] = sum((df[i,]-centroids[j,])^2)
6          }
7      }
8  distances
9  }
```

Computing the closest cluster for each case

We now need a function that will compare, for each case, its distance to each of the clusters and select the cluster where this value is minimum—that is, assign the case to a cluster based on this comparison. For this purpose, on line 2, we simply use the `which.min()` function, which indicates in which column the minimal value is, as an argument of the `apply()` function, which applies a function to each row of its input. We wrap this in the `cbind()` function, which allows it to return column-shaped vectors. If the number of returned clusters is less than expected (tested on line 3), we restart the process by assigning cases to random clusters, using the `set.random.clusters()` function (line 5). In other words, we take the precaution of restarting the process of setting the centroids randomly if we find an empty cluster.

```
1  assign= function (distances) {
2     clusters=data.frame(cbind(c(apply(distances, 1, which.min))))
3     if(nrow(unique(clusters))<ncol(distances)){
4     #precaution in case of empty cluster
5     clusters=set.random.clusters(nrow(distances),ncol(distances))
6  }
7  clusters
8  }
```

Tasks performed by the main function

We now almost have everything we need for our basic k-means implementation. We finally need to wrap this together in a main function, which we call `kay.means()`. Here we set initial cluster value (line 2) and then iterate over the computation of centroids (line 7), the calculation of distances (line 8), and the re-assignment of clusters (line 11). Notice the code block is contained in a `while` loop, which stops when the sum of squares of the distances (that is the total sum of squares within clusters) is the same twice in a row (when `ss.old` equal to `ss` – this value is set on line 10), and output the clustering solution (line 14).

```
1  kay.means = function (df, k) {
2      clusters = set.random.clusters(nrow(df),k)
3      ss.old = 1e100
4      ss = 1e99
5      while(ss!=ss.old) {
6
7          centroids = compute.centroids(df, clusters)
8          distances = euclid.sqrd(df, centroids)
9          ss.old=ss
10          ss = sum(distances)
11          clusters = assign(distances)
12      }
13      names(clusters) = "Clusters"
14      clusters
15  }
```

Let's try this using a very popular dataset, which we have already encountered in the previous chapter: the `iris` dataset, where observations are 150 iris flowers. Attributes are the species of the flowers and their petal and sepal length and width. So, in this case, we know the groups beforehand and are interested in knowing whether k-means can predict it from the other attributes. Let's start by having a look at the correct classification, which is in the 5th column of the data set.

```
iris[5]
```

We will not display the full output here, but you will see in your console that cases 1 to 50 are of species `Setosa`, cases 51 to 100 are of species `Versicolor`, and cases 101 to 150 of species `Virginica`.

We will use our knowledge of the dataset to determine the number of clusters. We select three clusters as we know there are three species. We will talk later about determining the number of clusters when this information is not available.

Internal validation

Our goal now is to assess the quality of our clustering solution. We are lucky, as we have the right answer already (which is not always the case). In order to check how well our clustering solution did, we first append a column to the `iris` dataset with the clustering solution of our implementation of k-means, and observe the convergence between the clustering and the species by creating a cross-tabulation of the data.

```
set.seed(1)
irisClust = cbind(iris, kay.means(iris[1:4], 3))
tableClust=table(unlist(irisClust[5]), unlist(irisClust[6]))
tableClust
```

The output is provided here:

	1	**2**	**3**
setosa	**0**	**0**	**50**
versicolor	**3**	**47**	**0**
virginica	**36**	**14**	**0**

Our implementation of k-means did a good job on this dataset: flowers of the `Setosa` species are classified in cluster 3, flowers of the `Versicolor` species are almost all classified in cluster 2, while flowers of the `Virginica` species are mostly classified in cluster 1, with a larger degree of misclassifications in cluster 2 (about a third). As we have information about the correct solution, we can compute indices of the correctness of our cluster solution. For instance, we can compute Cohen's kappa, which assesses the agreement of our clustering solution with the correct one. See the paper *A coefficient of agreement for nominal scales*, by Cohen (1960) for a description of the measure. The closer its value is to `1`, the better the agreement. This index is in the `psych` package, which needs installing and loading before we proceed with the analysis.

```
install.packages("psych"); library(psych)
```

The index requires that categories are given the same name in both the classification and the original solution. We therefore need to recode our data to proceed:

```
irisClust = cbind(irisClust, rep(0,nrow(irisClust)))
names(irisClust[7]) = "Species.recode"
irisClust[7][irisClust[5]=="setosa"] = 3
irisClust[7][irisClust[5]=="versicolor"] = 2
irisClust[7][irisClust[5]=="virginica"] = 1
```

We now have the data of both our clustering solution (in column 6) and the correct solution (in column 7) in the same format. We can now apply Cohen's kappa to our data:

```
kappa=cohen.kappa(cbind(irisClust[6],irisClust[7]))
```

The value of Cohen's kappa, `0.83` for the unweighted kappa (which we rely upon), shows good agreement, whereas the weighted value shows even better agreement:

```
Call: cohen.kappa1(x = x, w = w, n.obs = n.obs, alpha = alpha)
```

Cohen Kappa and Weighted Kappa correlation coefficient and confidence boundary output is as follows:

	lower	**estimate**	**upper**
unweighted kappa	**0.75**	**0.83**	**0.91**
weighted kappa	**0.87**	**0.91**	**0.95**
Number of subjects = 150			

We can also compute the rand index, which is another measure of agreement. We can find it in the `flexclust` package. Please install and load it before you proceed:

```
install.packages("flexclust"); library(flexclust)
```

Here we can simply reuse the cross-tabulation we produced earlier:

```
randIndex(tableClust)
```

The value of the index is `.716`, indicating also a good agreement. Try using the `kay.means()` function with a different seed, such as 3 for instance, and notice the differences.

While clustering solutions using k-means are always dependent on initial centroid values, our implementation is even more vulnerable. It is always a good idea to perform a cluster analysis with different seeds, which we will do later using the `nstart` argument with `kmeans()`. Nevertheless, our custom function fulfilled its purpose, which is to give the reader a sense of how k-means works.

Using k-means with public datasets

In what follows, we are going to learn more about partition clustering with k-means while exploring a dataset from the `cluster.datasets` package. This package contains datasets that were published in the book, *Clustering algorithms*, by Hartigan (1975), with examples of analyses. So let's start by installing this dataset on your machine, and loading it.

```
install.packages("cluster.datasets")
library(cluster.datasets)
```

Understanding the data with the all.us.city.crime.1970 dataset

We will first focus on getting to know the data, scaling the data to a common metric, and cluster interpretability. Our first exploration will concern the crime rates among different US cities in 1970. The dataset `all.us.city.crime.1970` affords such investigation:

```
data(all.us.city.crime.1970)
crime = all.us.city.crime.1970
```

Let's investigate the attributes in the dataset:

```
ncol(crime)
names(crime)
summary(crime)
```

There are 10 attributes. A look at the R manual page (type `?all.us.city.crime.1970`) allows us to understand what these variables are about. Most of them are pretty obvious considering their name, and we will not comment further here. Looking at the descriptive statistics, one can notice that there was a quite important number of crimes in the 24 cities for which data is available in this dataset: summing over `murder`, `rape`, `robbery`, `assault`, `burglary`, and `car.theft`, around 2,500 crimes took place per 100,000 residents, which means that about 2.5 percent of the population was the victim of a crime that year (considering that one person could only be a victim of one crime). It might be interesting to know if cities differ in relation to the crimes that are committed. We will manually explore several clustering solutions. We will only consider here dimensions related to crime, which is attributes 5 to 10. Before we run `kmeans()`, let's have a look at the relationship between the attributes.

```
plot(crime[5:10])
```

The resulting image is not displayed here because it will be updated some lines below. As you can see on your screen, there is visibly a strong positive association between the rate of some crimes (such as `burglary` and `rape`), and a weaker for others (such as `murder` and `burglary`). Overall it seems that the more of one `crime` type is committed, the more the others are as well. We can confirm this intuition looking at the correlation matrix (rounded to three decimals).

```
round(cor(crime[5:10]),3)
```

Here is the output:

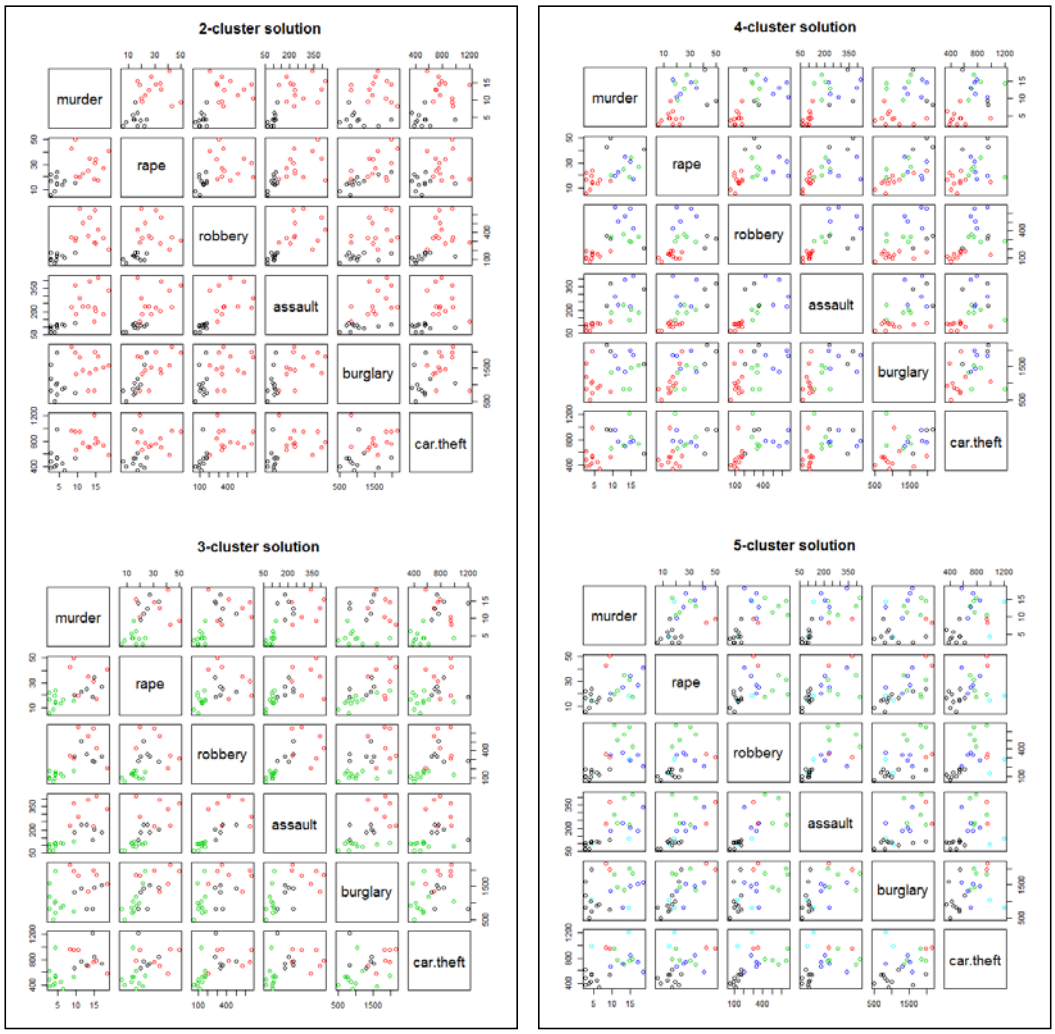
	murder	rape	robbery	assault	burglary	car.theft
murder	**1**	**0.526**	**0.638**	**0.709**	**0.353**	**0.495**
rape	**0.526**	**1**	**0.414**	**0.667**	**0.694**	**0.410**
robbery	**0.638**	**0.414**	**1**	**0.699**	**0.551**	**0.559**
assault	**0.709**	**0.667**	**0.699**	**1**	**0.596**	**0.428**
burglary	**0.353**	**0.694**	**0.551**	**0.596**	**1**	**0.382**
car.theft	**0.495**	**0.410**	**0.559**	**0.428**	**0.382**	**1**

Yet, the relatively modest values of some correlations permits to imagine a specialization of crime in some cities.

We will run `kmeans()` on this dataset with an increasing number of clusters (from 2 to 5), and will examine to solutions visually and concurrently. We will have a detailed look at the output of only the first and last clustering models, at the end. We will let the reader modify the code with regards to the number of clusters. We could have implemented a loop to do this, but we think it is more interesting if you have a look at each solution individually at your pace. In all our models, we will ask k-means to repeat the procedures 25 times (using argument `nstart`) in order to be sure to have a good clustering solution. We will of course start by standardizing our data, in order to avoid one attribute that is more important than the others in computing the distances.

```
1  crime.scale = data.frame(scale(crime[5:10]))
2  set.seed(234)
3  TwoClusters = kmeans(crime.scale, 2, nstart = 25)
4  plot(crime[5:10],col=as.factor(TwoClusters$cluster),
5  main = "2-cluster solution")
6  ThreeClusters = kmeans(crime.scale, 3, nstart = 25)
7  plot(crime[5:10],col=as.factor(ThreeClusters$cluster),
8    main = "3-cluster solution")
9  FourClusters= kmeans(crime.scale, 4, nstart = 25)
```

```
10  plot(crime[5:10],col=as.factor(FourClusters$cluster),
11     main = "4-cluster solution")
12  FiveClusters = kmeans(crime.scale, 5, nstart = 25)
13  plot(crime[5:10],col=as.factor(FiveClusters$cluster),
14  main = "5-cluster solution")
```



The relationship between several types of crimes and cluster membership for k=2 to k=5

An important aspect of cluster analysis is the interpretation of the clusters. As can be seen in the preceding screenshot, the interpretation of the clusters in the 2-cluster solution is quite straightforward. Cities with a low criminality make up the black cluster, whereas the red cluster is composed of cities with higher criminality.

The pattern is more complex in the model with three clusters. At first sight, it seems that this cluster is about a low average and high criminality. But this is denied by a closer inspection: `burglary` and `car.theft` can be high in the green cluster, `rape` and `murder` can be low to average, while `assault` and `robbery` are low. The black cluster seems to be concerned with cities with average crime. But looking more closely, `murder` can be higher in this cluster than in the red one; this is true to a lesser extent for `rape` and `car.theft`. We could consider this cluster as representing cities with a high murder rate and an average rate of other crimes. The red cluster is the most dispersed of the three, yet it is the easiest to interpret. Cities in this cluster have average to high values for all the study's dimensions of crime. The solutions with four and five clusters are even more difficult to interpret. It is usually advised to consider a number of clusters manageable for interpretation (not hundreds of clusters) and that are meaningful, even if a larger number of clusters explains the data better.

Let's now examine the textual output of R for our first (`TwoClusters`) solution.

```
TwoClusters
```

Here is the output:

```
K-means clustering with 2 clusters of sizes 11, 13

Cluster means:
      murder       rape    robbery    assault   burglary  car.theft
1 -0.9128346 -0.6991864 -0.8438639 -0.8328348 -0.5708682 -0.7166146
2  0.7723985  0.5916192  0.7140387  0.7047064  0.4830424  0.6063662
Clustering vector:
 [1] 1 2 1 1 2 1 2 2 2 2 2 1 1 2 2 1 1 1 2 2 1 1 2
Within cluster sum of squares by cluster:
[1] 18.39421 47.16265
 (between_SS / total_SS =  52.5 %)
Available components:
[1] "cluster"      "centers"    "totss"      "withinss"    "tot.withinss"
[6] "betweenss"    "size"       "iter"       "ifault"
```

The *Cluster means* reports the centroids for the final iteration of the algorithm, usually when convergence is achieved. This information confirms our visual interpretation of the clustering solution—one factor has high means on all crime dimensions, whereas the other has low means. This section is directly accessible as data by typing: `TwoClusters$centers`.

The *Clustering vector* reports on the membership of the observations to each of the clusters—for instance, the first observation is part of cluster 2 (low criminality), whereas the last is part of cluster 1 (average to high criminality). This section is directly accessible as data by typing: `TwoClusters$cluster`.

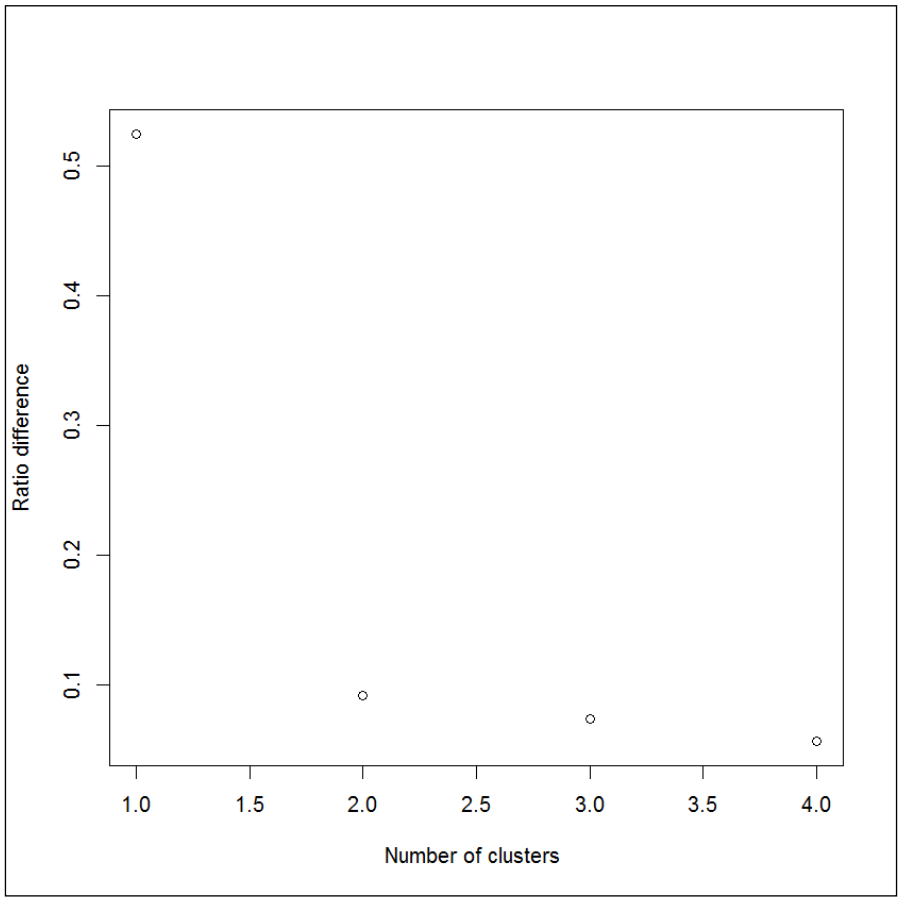
The section *Within cluster sum of squares by cluster* reports on the overall squared distance between the data points and their centroid, within each of the clusters. We can also see a division between the **between sum of squares** (**BSS**) and the **total sum of square** (**TSS**). The BSS refers to the overall squared difference, for each data point, between the mean of its centroid and the overall mean. The TSS refers to the overall squared distance of the data points to the mean of all the means.

We can also see (under *Available components*) that we can examine other values we have not yet seen—*totss* is the total sum of squares, *tot.withinss* is the total of the sum of squares within clusters, `between ss` is the total sum of squares within clusters, *size* is the number of cases classified in each of the clusters, *iter* is the number of iterations required for convergence, and *ifault* signals warnings and problems (with a value of 0 if there is no issue).

We are now going to plot the differences between the value BSS/TSS for each of the clusters. Basically, this value shows how much of the data is explained by the clustering solution, as it divides the BSS by the TSS. It involves computing the ratio differences to a vector and using the `plot()` function. The first value is the ratio for the `TwoClusters` model.

```
1  v=rep(0,4)
2  v[1] = TwoClusters[[6]]/TwoClusters[[3]]
3  v[2] = (ThreeClusters[[6]]/ThreeClusters[[3]]) - v[1]
4  v[3] = (FourClusters[[6]]/FourClusters[[3]]) - sum(v[1:2])
5  v[4] = (FiveClusters[[6]]/FiveClusters[[3]]) - sum(v[1:3])
6  plot(v, xlab = "Number of clusters ",
7     ylab = "Ratio difference")
```

Following is the output:



Differences between the between and total sum of squares among our models

We can see in the preceding graph that the ratio is around `.5` in the `TwoClusters` solution, and that it doesn't increase much with more clusters. The `TwoClusters` solution should therefore be preferred. Moreover, we have seen that solutions with more than two clusters are difficult to interpret. A BSS/TSS = 1 is the best possible value, yet it will seldom be reached.

Finding the best number of clusters in the life.expectancy.1971 dataset

We will now use the `life.expectancy.1971` dataset about life expectancy in several countries in 1971. It includes 10 attributes: the country were the data has been collected, the year of data collection, and the life expectancy (remaining) for male and female individuals aged 0 years old, 25, 50, and 75. As with the previous dataset, this one also does not specify the membership of our cases to categories. So again, we will have to decide on the number of clusters by ourselves. We will examine how to do so more precisely. We will create a function for this purpose. Before we do that, let's discover the dataset we will use.

Let's start by loading and examining to dataset.

```
data(life.expectancy.1971)
life.expectancy.1971
```

A partial view of the output is provided below:

	country	year	m0	m25	m50	m75	f0	f25	f50	f75
1	Algeria	1965	63	51	30	13	67	54	34	15
2	Cameroon	1964	34	29	13	5	38	32	17	6
3	Madagascar	1966	38	30	17	7	38	34	20	7
4	Mauritius	1966	59	42	20	6	64	46	25	8
...										
22	Trinidad	1962	64	43	21	7	68	47	25	9
23	Trinidad	1967	64	43	21	6	68	47	24	8
24	US	1966	67	45	23	8	74	51	28	10
25	US (Nonwhite)	1966	61	40	21	10	67	46	25	11
26	US (White)	1966	68	46	23	8	75	52	29	10
27	US	1967	67	45	23	8	74	51	28	10

Even without computing the mean and standard deviations for the variables, we can notice that there is quite some variation regarding life expectancy (please refer to the complete output on your screen as well). A first observation, which is broadly documented, is that women have a longer remaining life expectancy than men, at all ages. A country strikes in this list—in Madagascar, at the time of data collection, women apparently did not have longer life expectancy than men in their young and old years. Further, the mean life expectancy at birth was only 38 for both women and men. This is also the life expectancy of females in Cameroon at that time, whereas males were expected to live even a little less (34 years). Looking at the table, we can notice that Trinidad and the US are entered several times, as data collection was carried out more than once. We will therefore discard case 23 (the second entry for Trinidad), as well as both cases 24 and 27 (US, data collected in 1966 and 1967) because cases 25 and 26 are more specific, as they provide estimations for White and Nonwhite individuals. Let's create a new dataset without these cases before we proceed with cluster analysis.

```
life = life.expectancy.1971[-c(23,24,27),]
```

Here we will scale the data. The importance of scaling data has been discussed in the first section of this chapter. We also add some attributes to the dataset, corresponding to the ratio of male life expectancy to female life expectancy at all ages, as the difference between male and females would be lost in data scaling (all means will be 0).

```
life.temp = cbind(life, life$m0/life$f0, life$m25/life$f25,
   life$m50/life$f50, life$m75/life$f75)
```

If you run this, you will notice an error. It happens that attribute `f50` is composed of strings instead of numeric values (type `mode(life$f50)` to check this). This is a type of problem you might encounter when dealing with data you have not prepared yourself (and sometimes even with your data). The solution is obviously to convert the attribute to numeric values before being able to compute the ratios.

```
life$f50 = unlist(lapply(life$f50, as.numeric))
```

We can now repeat our assignment to `life.temp` with a successful result, and scale the data frame (omitting rows 1 and 2: name of country and year of data collection). We first convert to a data frame to get rid of information about mean and standard deviation that is contained in the returned object; we then convert to a matrix again.

```
life.scaled = as.matrix(data.frame( scale(life.temp[-(c(1,2))]) ))
```

We continue discussing this data next.

External validation

When examining the `iris` dataset, we had the correct solution regarding the number of clusters and the classification of cases. This is not the case here—we can not tell before running the analyses the number of groups in our data. We will therefore rely on computational trickery to discover them; cluster analysis will be performed iteratively and the clustering solutions will be compared using several indexes for determining the ideal number of clusters. More information about such indexes can be found in the paper *Experiments for the number of clusters in k-means*, by Chiang and Mirkin (2007). Here we rely on `NbClust()` function from the `NbClust` package, which we install and load:

```
install.packages("NbClust"); library(NbClust)
```

We simply call the function specifying the data and clustering algorithm to be used. By default, the function will perform clustering using the Euclidean distance and compute all available indexes. The reader is advised to consult the documentation for more information about customization.

```
NbClust(life.scaled, method = "kmeans")
```

Part of the output is provided below. This shows that three clusters is the most appropriate solution.

```
*******************************************************************
* Among all indices:
* 3 proposed 2 as the best number of clusters
* 10 proposed 3 as the best number of clusters
* 1 proposed 8 as the best number of clusters
* 1 proposed 12 as the best number of clusters
* 1 proposed 13 as the best number of clusters
* 7 proposed 15 as the best number of clusters
                   ***** Conclusion *****
* According to the majority rule, the best number of clusters is 3
```

Summary

In this chapter, we discovered clustering with the k-means algorithm family. We explored several distance measurements and learned how to scale data on a similar metric. We explored in detail the mechanism through which k-means creates clusters. We also examined how to select the best number of clusters and assessed the quality of clustering solutions. In the next chapter, we will explore hierarchical clustering using `hclust()`.

5
Agglomerative Clustering Using hclust()

Unlike partition clustering, which requires the user to specify the number of *k* clusters and create homogeneous *k* groups, hierarchical clustering defines clusters without user intervention from distances in the data and defines a tree of clusters from this. Hierarchical clustering is particularly useful when the data is suspected to be hierarchical (leaves nested in nodes nested in higher level nodes). It can also be used to determine the number of clusters to be used in k-means clustering. Hierarchical clustering can be agglomerative or divisive. Agglomerative clustering usually yields a higher number of clusters, with less leaf nodes by cluster.

Agglomerative clustering refers to the use of algorithms, which start with a number of clusters that is equal to the number of cases (each case being a cluster) and merges clusters iteratively one by one, until there is only one cluster that corresponds to the entire dataset. Divisive cluster is the opposite, it starts with one cluster, which is then divided in two as a function of the similarities or distances in the data. These new clusters are then divided, and so on, until each case is a cluster. In this chapter, we will discuss agglomerative clustering. The reader might be interested in consulting the paper *Hierarchical clustering schemes* by Johnson (1967).

In this chapter, we will cover the following:

- The inner working of agglomerative clustering
- The use of `hclust()` for agglomerative clustering with numerical attributes
- The use of `hclust()` with binary attributes

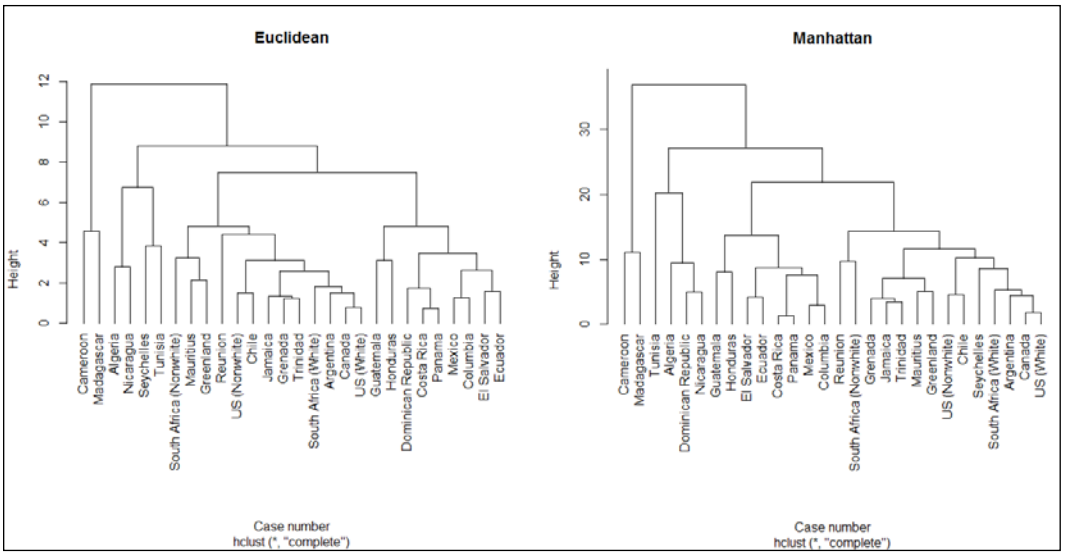
The inner working of agglomerative clustering

As briefly mentioned, agglomerative clustering refers to algorithms. Let's start with the example of the data we used last in the previous chapter:

```
1  rownames(life.scaled) = life$country
2  a=hclust(dist(life.scaled))
3  par(mfrow=c(1,2))
4  plot(a, hang=-1, xlab="Case number", main = "Euclidean")
```

We started by adding the name of each country as the row name of the related case (line 1), in order to display it on the graph. The function `hclust()` was then used to generate a hierarchical agglomerative clustering solution from the data (line 2). The algorithm uses a distance matrix, provided as an argument (here the default is the Euclidean distance) to determine how to create a hierarchy of clusters. We have discussed measures of distance in the previous chapter. Please refer to this explanation if in doubt. Finally, the `hclust` object `a` at line 2 was plotted in a dendrogram (line 4 in the following diagram). At line 3, we set the plotting area to include two plots.

The following figure (left panel), displays the clustering tree. At the beginning of the creation of the cluster hierarchy (or tree), each case is a cluster. Clusters are joined one by one on the basis of the distance between them (represented by the vertical lines); clusters with smaller distances are selected for merging. As can be seen from the distances plotted in the figure, `hclust()` first joined **Panama** and **Costa Rica** in a new cluster. It then aggregated **Canada** and **US (White)** in a cluster, as they were the next cases with the smallest Euclidean distance, and it also did so for **Grenada** and **Trinidad**, **Mexico** and **Columbia**, and **El Salvador** and **Ecuador**. Next, it joined the cluster formed by **Grenada** and **Trinidad** with **Jamaica**. Clusters with the next smaller distance were then **US (Nonwhite)** and **Chile**, which were merged together. Then the cluster **Canada** and **US (White)** was merged with **Argentina**. The algorithm continued merging clusters until only one remained. It is quite interesting that `hclust()` produces life expectancy clusters that are closely related to the distance between countries; countries that are close geographically generally have similar life expectancies.



Dendrograms of remaining life expectancy in several countries (Euclidean distance)

As we have mentioned, we used `hclust()` with the default (Euclidean distance) matrix in the previous example. Results can be different, although usually not dramatically, using other distance metrics. We have no reason to infer that the shortest distance between the points is not the best measurement here, but to illustrate the potentially different results using different measure distances, we will be using the Manhattan distance instead of Euclidean distance. On line 1 of the following code, we assign the `hclust` object returned by `hclust()` with argument configuration `manhattan`, which sets the distance metric. We then simply plot the resulting tree. The `hang` is set to a negative value in order to get the distance that is to be plotted from 0:

```
1  a = hclust(dist(life.scaled, method= "manhattan"))
2  plot(a, hang=-1, xlab="Case number", main = "Manhattan")
```

The preceding figure (right panel) displays the result of our aggregative cluster analysis with `hclust()`. We can notice that, although there are many similarities, the clustering solution is not the same, as compared to an Euclidean distance. We can see that **Madagascar** and **Cameroon** form a cluster that cannot be joined to the other clusters until reaching the last step in both solutions. Also, in both solutions, the **US (White)** and **Canada** are very similar to each other (form a cluster quite fast), as well as **Costa Rica** and **Panama**. But let's have a look at **Algeria**. In the Euclidean distance solution, it forms a cluster with **Nicaragua**, but in the Manhattan distance solution, it clusters with **Nicaragua** and **Dominican Republic**. **Seychelles** forms a cluster with **Tunisia** in the Euclidean distance solution, but they cluster with **South Africa**, **Argentina**, **Canada**, and **US** in the Manhattan distance solution.

Indeed, the choice of the distance matrix is not without incidence on the results. It has to be chosen as a function of the data (what they represent). Choosing the method to determine cluster proximity also plays an important role in agglomerative clustering. `hclust()` provides several methods including *single linkage*, *complete linkage*, *average linkage*, and *Ward's minimum variance*. The *single linkage* method computes the proximity of clusters as the smallest distance between the points of the clusters. The *complete linkage* method computes it as the maximum distance between points of the cluster; average linkage uses the average distance of the points from one cluster to the points of the other. Finally, *Ward minimum variance* agglomerates clusters by minimizing the variance around centroids in the resulting clusters. Note that *Ward's minimum variance* method requires a squared Euclidean distance matrix.

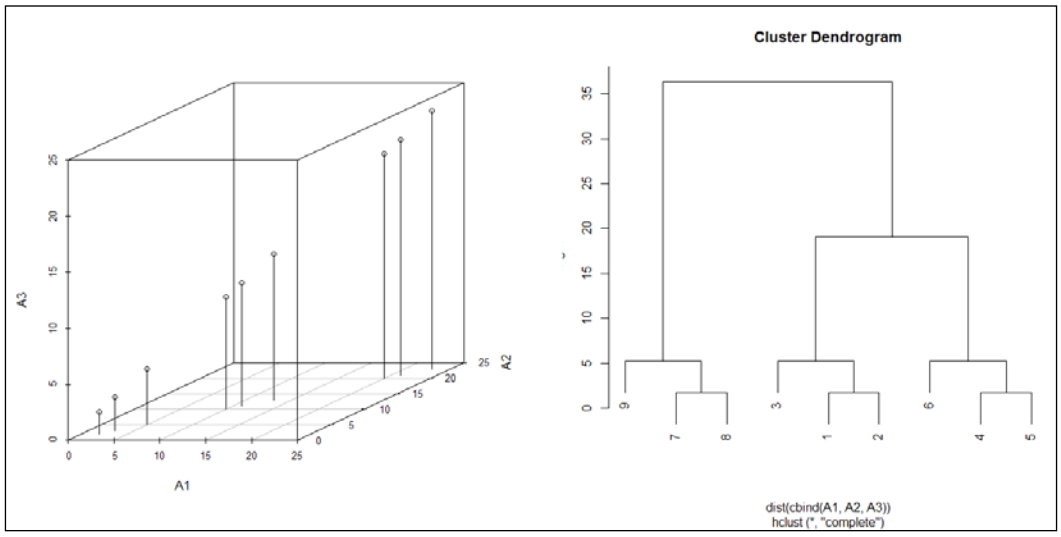
These methods have different properties (The R Core Team, 2013, p.1301):

> *"Ward's minimum variance method aims at finding compact, spherical clusters. The complete linkage method finds similar clusters. The single linkage method (which is closely related to the minimal spanning tree) adopts a 'friends of friends' clustering strategy. The other methods can be regarded as aiming for clusters with characteristics somewhere between the single and complete link methods."*

We will examine differences in results later in the chapter.

Before we proceed, let's examine, this time visually, how the algorithm proceeds using a simple fictitious dataset. As visual representations are limited to three dimensions, we will only use three attributes, but the computation is similar with more attributes. We will display these using the `scatterplot3d()` function of the `plot3D` package, which we will install and load after creating the attributes. We then examine the clustering solution provided by `hclust()` in order to assess whether it confirms the impressions we get from visual inspection:

```
A1 = c(2,3,5,7,8,10,20,21,23)
A2 = A1
A3 = A1

install.packages("scatterplot3d")
library(scatterplot3d)
scatterplot3d(A1,A2,A3, angle = 25, type = "h")

demo = hclust(dist(cbind(A1,A2,A3)))
plot(demo)
```



A 3D scatterplot and dendrogram exemplifying agglomerative clustering

As can be noticed on the left panel of the previous screenshot, there are three groups of two points that are very close to each other. Another point is quite close to each of these groups of two. Consider that the groups of two constitute a group of three with the points that lie closest to them. Finally, the two groups on the left are closer to each other than they are to the group of three on the right. If we have a look at the dendrogram, we can see that the very same pattern is visible.

Agglomerative clustering with hclust()

In what follows, we are going to explore the use of agglomerative clustering with `hclust()` using numerical and binary data in two datasets.

Exploring the results of votes in Switzerland

In this section, we will examine the case of another dataset. This dataset represents the percentage of acceptance of the themes of federal (national) voting objects in Switzerland in 2001. The first rows of data are in the following table. The rows represent the cantons (the Swiss name for states). The columns (except the first) represent the topic of the voting. The values are the percentage of acceptance of the topic of voting. The data has been retrieved from the Swiss Statistics Office (`www.bfs.admin.ch`) and are provided in the folder for this chapter (file `swiss_votes.dat`).

Canton	Europe	Medicine	Speed	Military1	Military2	Bishopric	Taxes1	Military3	Protection	Taxes2
AG	17	34	17	50.9	51.1	64.5	18.8	17.1	17.2	28.4
AI	6.8	24.9	10.6	37.3	37.8	67.2	14.3	11.5	10.4	22.7
AR	13.5	28.4	18.5	45.6	46.1	65.9	20.9	17.6	17.2	33.7
BE	23.4	31.4	22.1	57.7	57.4	60.2	24.2	19.6	20.9	41.5
BL	22.6	30.6	23.1	54.5	53.2	67.3	23.6	22.9	23.4	31.3

The first five rows of the dataset

To load the data, save the file in your working directory or change the working directory (use `setwd()` to the path of the file) and type the following line of code. Here, the `sep` argument is set to indicate that tabulations are used as separators, and the header argument is set to `T` (true), meaning that the provided data has column headers (attribute names):

```
swiss_votes = read.table("swiss_votes.dat", sep = "\t",
   header = T)
```

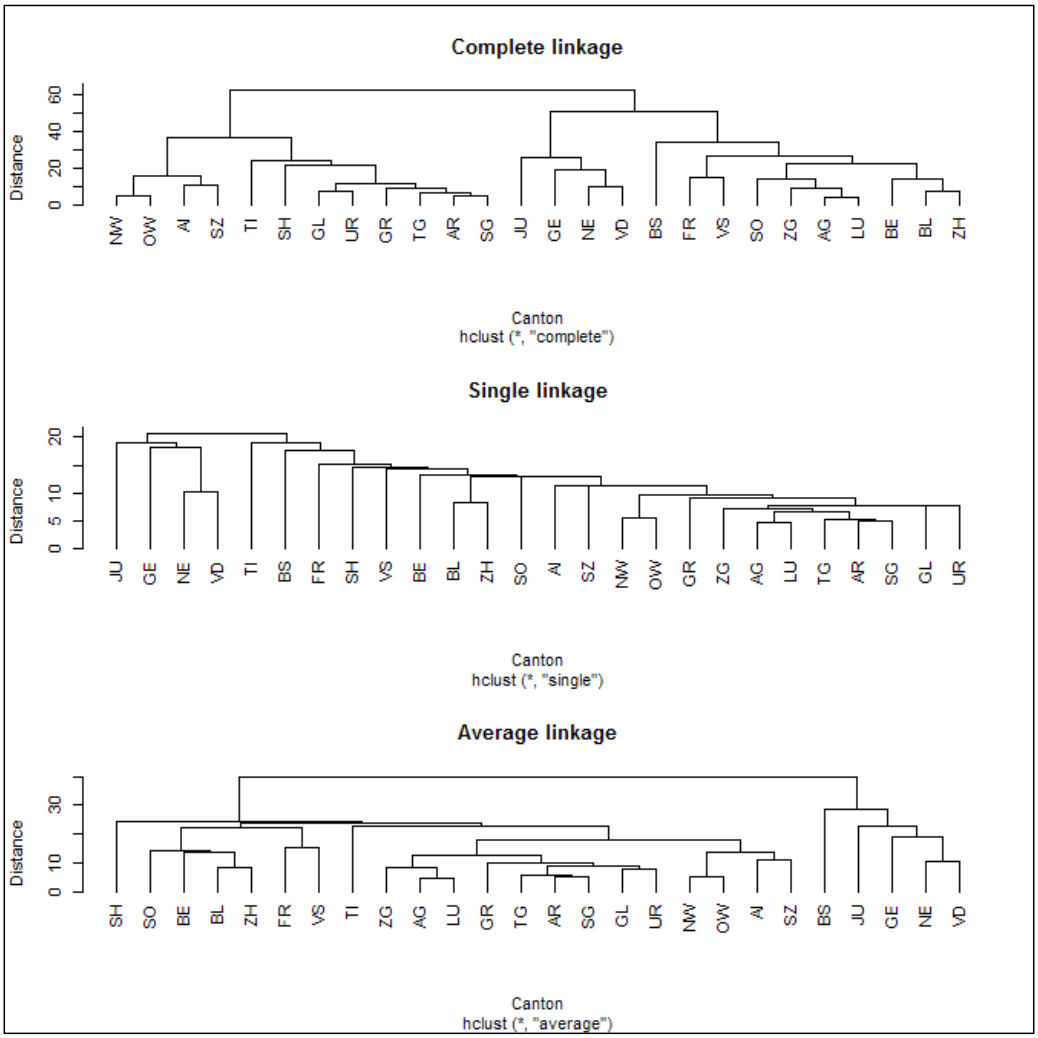
Here we are interested in knowing whether, in year 2001, we can find clusters in the voting behavior of the populations of the cantons. We will perform the analysis repetitively using the three methods discussed previously and examine the potential differences.

After computing the distance matrix (line 1), we start with the default method (complete linkage as on line 2), and then the single (line 3) and average (line 4) methods. On line 5, we set the plotting area to contain three vertical graphs on line 4, and proceed to plot the graphs on lines 6 to 14:

```
1  dist_matrix = dist(swiss_votes[2:11])
2  clust_compl  = hclust(dist_matrix)
3  clust_single  = hclust(dist_matrix, method = "single")
4  clust_ave  = hclust(dist_matrix, method = "average")
5  par(mfrow = c(3,1))
6  plot(clust_compl, labels=swiss_votes$Canton, hang = -1,
7     main = "Complete linkage", xlab = "Canton",
8     ylab = "Distance")
9  plot(clust_single, labels=swiss_votes$Canton, hang = -1,
10     main = "Single linkage", xlab = "Canton",
11     ylab = "Distance")
12  plot(clust_ave, labels=swiss_votes$Canton, hang = -1,
13     main = "Average linkage", xlab = "Canton",
14     ylab = "Distance")
```

It can be seen that, as mentioned by the R Core Team (2013), the *Complete linkage* method produces compact clusters, whereas the *Single linkage* function is more inclusive. The *Average linkage* function produces a clustering solution that lies somewhere in between. What is also interesting to notice is that points (here cantons) that are grouped together using one method are not necessarily grouped together using another method. The choice of the clustering solution is therefore important in analyzing the data.

One way to proceed to such a choice is based on the interpretability of the results and domain knowledge. One thing that explains shared opinions is geographical proximity, as people who are close communicate more frequently and thereby share ideas (Latané, 1996). The clustering solution which relies on *Complete linkage* better exemplifies the impact of geographical proximity (take a look at a map of Switzerland), which allows interpreting the results using this rationale. We will therefore investigate the differences between clusters more precisely in this solution.



Dendrograms of Swiss votes

We can see, on the corresponding cluster tree for *Complete linkage*, that that there are four subgroups. The first is composed of **NW**, **OW**, **AI**, **SZ**; the second of **TI**, **SH**, **GL**, **UR**, **TG**, **AR**, and **SG**; and so on. We know that most of the cantons composing a cluster are close, geographically, but we don't know exactly what this means in relation to the data. We now want to examine the differences between the clusters in terms of voting behavior. The first thing we want to do now is assign the cantons to a cluster on the basis of the clustering solution with the *Complete linkage* method. We determined that four clusters are good for this data, after examining the clustering tree. We now want to display the cluster assignment:

```
clusters=cutree(clust_compl, k = 4)
cbind(clusters,swiss_votes[1])
```

The output is provided below:

	clusters	Canton
1	**1**	**AG**
2	**2**	**AI**
3	**3**	**AR**
4	**1**	**BE**
5	**1**	**BL**
6	**1**	**BS**
7	**1**	**FR**
8	**4**	**GE**
9	**3**	**GL**
10	**3**	**GR**
11	**4**	**JU**
12	**1**	**LU**
13	**4**	**NE**
14	**2**	**NW**
15	**2**	**OW**
16	**3**	**SG**
17	**3**	**SH**
18	**1**	**SO**
19	**2**	**SZ**

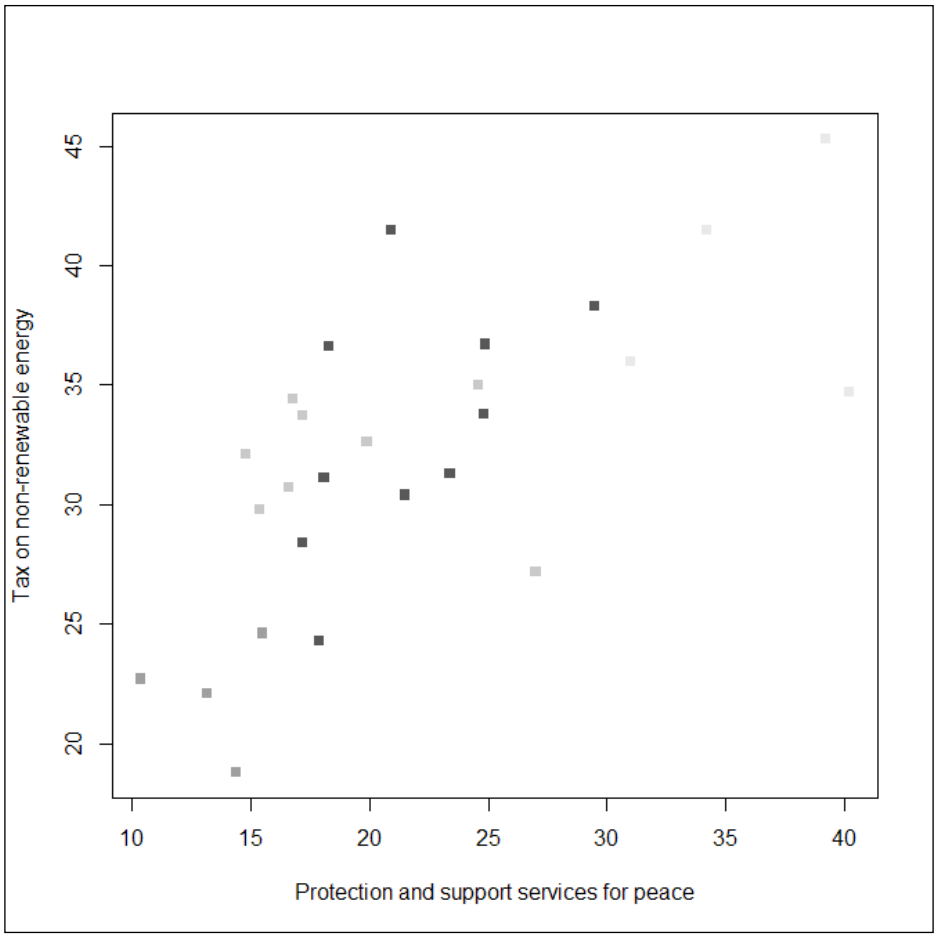
	clusters	Canton
20	**3**	**TG**
21	**3**	**TI**
22	**3**	**UR**
23	**4**	**VD**
24	**1**	**VS**
25	**1**	**ZG**
26	**1**	**ZH**

We can see that cluster 1 corresponds to the last cluster, the subcluster on the dendrogram, cluster 2 to the first, cluster 3 to the second, and cluster 4 to the third (on the diagram).

We will create a scatterplot with the acceptance of two votes in order to examine these patterns. The first vote, of which the acceptance rates are depicted on the *x* axis (called `Protection` in the data frame), was related to the implementation of a protection and support service that would aim at guaranteeing peace within the country. The second vote, of which the acceptance rates are depicted on the *y* axis (called `Taxes2` in the data frame), was related to the implementation of an environmental tax on nonrenewable energy. Both themes were rejected by the population in all cantons. Yet, there are important differences in the extent of the rejection.

We plot the graphic using the following code. Notice the order of the colors in order to understand which cluster they represent:

```
plot(swiss_votes$Protection,swiss_votes$Taxes2,
   pch=15, col=gray.colors(4)[clusters],
   xlab="Protection and support services for peace",
   ylab = "Tax on non-renewable energy")
```

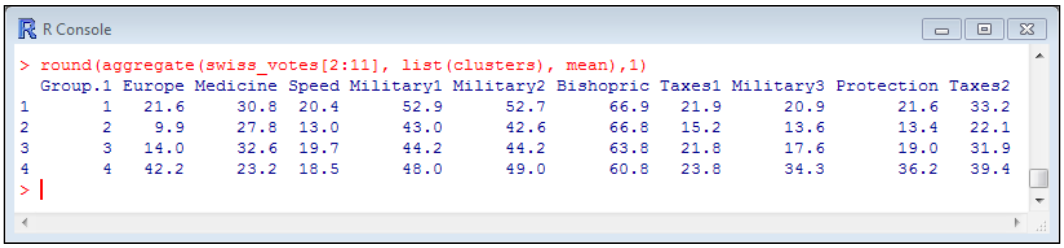


A scatterplot of 2 votes, by cluster

Generally, the agreement to both votes are related; the more the population of a canton agrees with one of the votes, the more it will agree with the other. We can notice that the cluster on the left (cluster 2 in the preceding output) has the lowest agreement in the case of both votes, and the cluster on the right (cluster 4 in the output), has the higher agreement in the case of both votes as well. The other two clusters are in between in the case of these two votes. We think this is an interesting visual representation, but wouldn't it be nice to have a table with the mean agreement for each vote and cluster? This is very easy to obtain:

```
round(aggregate(swiss_votes[2:11], list(clusters), mean),1)
```

The output is provided in the following figure:



One can notice that the second cluster (corresponding to the first cluster on the left on the scatterplot) has a mean value that is generally smaller than the other clusters. We can see that the agreements in the other clusters vary as a function of the topic of the votes, with cluster 4 being generally less conservative. The interested reader can find more about Swiss politics on the following page: `http://www.admin.ch/org/polit/00054/index.html?lang=en`.

The use of hierarchical clustering on binary attributes

The previous datasets that we have used are composed of numerical attributes. Data is sometimes composed of binary attributes. By binary, we mean that there are only two possible modalities for the attribute. Examples of such attributes include characteristics such as gender (woman/man), currently married (yes/no), and organization type (private/public).

The distance metric to be provided to `hclust()` has to take the nature of the attributes into account; that is, it must be computed accordingly. The `binary` distance is the required type for such data. A distance matrix containing the `binary` distance can be obtained using the following line of code, where `df` is the data frame on which to compute the distance:

```
dist(df, method="binary")
```

Here we will use the `Trucks` dataset from the `vcd` package. Install and load it (as well as the data) using the following code:

```
install.packages("vcd")
library(vcd)
data(Trucks)
head(Trucks)
```

The few lines of the data displayed here (requested on the last line of the preceding code) allow us to notice that it requires frequency weighting:

	Freq	period	collision	parked	light
1	712	before	back	yes	daylight
2	613	after	back	yes	daylight
3	192	before	forward	yes	daylight
4	179	after	forward	yes	daylight
5	2557	before	back	no	daylight
6	2373	after	back	no	daylight

For instance, the first row represents `712` cases of truck accidents that occurred during the day before a new safety policy was implemented, with a collision at the back of the vehicle, when it was parked. We want `712` cases instead of one row. This can be done using the following code:

```
Trucks.wd<- Trucks[rep(1:nrow(Trucks),Trucks$Freq),]
```

For further analyses, we will remove attributes `Freq` (number 1) and `light` (number 5):

```
Trucks.rm = Trucks.wd[, -(c(1,5))]
```

As the number of cases makes the process very slow, and also for visibility reasons, we will randomly sample `100` cases only:

```
set.seed(456)
Trucks.sample = Trucks.rm[sample(nrow(Trucks.rm), 100), ]
```

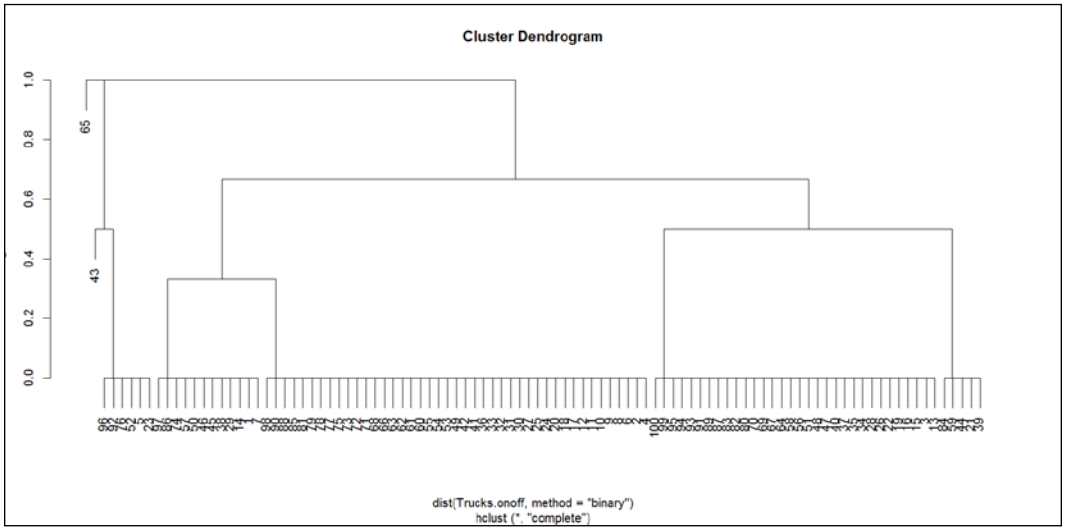
We also need to set one value to `off` (0), for each attribute. All other values will be considered `on` (1):

```
1  Trucks.onoff = data.frame(matrix(nrow =
2     nrow(Trucks.sample), ncol = ncol (Trucks.sample)))
3  for (i in 1:nrow(Trucks.sample)) {
4     for (j in 1:ncol(Trucks.sample)) {
5        if (Trucks.sample[i,j] != Trucks.sample[1,j])
6           Trucks.onoff[i,j] = 0
7        else Trucks.onoff[i,j] = 1
8     }
9  }
10  names(Trucks.onoff)=names(Trucks.sample)
```

We can now proceed with the analysis and plot the dendrogram:

```
b = hclust(dist(Trucks.onoff, method= "binary"))
plot(b)
```

As can be seen in the following figure (because of random sampling, your figure might look different, but what follows should be valid anyway), there are lots of cases that have a distance of 0 in the figure, as we didn't configure the plot to be aligned on 0; this means that it is the actual distance of the cases. As the distance is often 0, `hclust()` didn't proceed to the usual grouping by pair that we have seen in the previous examples. One can further see that most cases are part of clusters that are relatively distant from one another, which makes sense from observing the data.



A dendrogram of truck accidents (binary data)

Even though we have randomly selected 100 cases only, the dendrogram is still quite dense. For this reason, we will only comment on the left part. So let's have a look at cases `96`, `92`, `76`, `52`, `5`, and `23`. In the output, we can see that all cases share the same values on these attributes:

```
Trucks.onoff[c(96,92,76,52,5,23),]
```

The output is provided here:

	period	collision	parked
96	**1**	**1**	**0**
92	**1**	**1**	**0**
76	**1**	**1**	**0**
82	**1**	**1**	**0**
5	**1**	**1**	**0**
23	**1**	**1**	**0**

Summary

In this chapter, we discovered hierarchical (or nested) clustering, particularly in its agglomerative form. We used several distance metrics (Euclidean, Manhattan, and binary) as well as several linkage functions. We discussed how to interpret the result of clustering and how cluster analysis can be used for further inquiry of the data, and discussed real-life examples. Another popular application we did not discuss here is text classification. We have also seen that datasets sometimes require some effort (preprocessing) to be made compliant with analytic requirements. In the next chapter, we will see how to use principal component analysis, notably to perform dimensionality reduction.

6

Dimensionality Reduction with Principal Component Analysis

Nowadays, accessing data is easier and cheaper than ever before. This has led to the proliferation of data in organizations' data warehouses and on the Internet. Analyzing this data is not a trivial task, as its quantity often makes analysis difficult or unpractical. For instance, the data is often more abundant than available memory on the machines. The available computational power is also often not enough to analyze the data in a reasonable time frame. One solution is to have recourse to technologies that deal with high dimensionality in data (Big Data). These solutions typically use the memory and computing power of several machines for analysis (computer clusters). But most organizations do not have such an infrastructure. Therefore, a more practical solution is to reduce the dimensionality of the data while keeping the essential information intact.

Another reason to reduce dimensionality is that, in some cases, there are more attributes than observations. If scientists were to store the genome of all inhabitants of Europe and the United States, the number of cases (approximately 1 billion) would be much less than the 3 billion base pairs in the human DNA.

Most analyses do not work well or at all when observations are fewer than attributes. Confronted with this problem, data analysts might select groups of attributes that go together (for instance, height and weight) according to their domain knowledge, and reduce the dimensionality of the dataset.

The data is often structured across several relatively independent dimensions, with each dimension measured using several attributes. This is where **Principal Component Analysis** (**PCA**) is an essential tool, as it permits each observation to receive a score on each dimension (determined by PCA itself) while allowing us to discard the attributes from which the dimensions are computed. What is meant here is that, for each of the obtained dimensions, values (scores) are produced that combine several attributes. These can be used for further analyses. In the next section, we will use PCA to combine several attributes in questionnaire data. We will see that participants' self-reports on items such as lively, excited, enthusiastic, (and many more) can be combined in a single dimension we call positive arousal. In this sense, PCA performs both dimensionality reduction (discards the attributes) and feature extraction (computes the dimensions).

Another use of PCA is to check that the underlying structure of the data corresponds to a theoretical model. For instance, in a questionnaire, a group of questions might assess construct A, another group construct B, and so on. PCA will find two different factors, if indeed there is more similarity in the answers of participants within each group of questions compared to the overall questionnaire. Researchers in fields such as psychology use PCA mainly to test their theoretical model in this fashion.

In what follows, we will:

- Examine how PCA works
- Continue with a tutorial on using PCA in R, in which we will notably discover how to interpret the results, select the appropriate number of dimensions, and perform diagnostics

The inner working of Principal Component Analysis

Principal Component Analysis aims at finding the dimensions (principal component) that explain most of the variance in a dataset. Once these components are found, a principal component score is computed for each row and each principal component. Remember the example of the questionnaire data we discussed in the preceding section. These scores can be understood as summaries (combinations) of the attributes that compose the data frame.

PCA produces the principal components by computing the eigenvalues of the covariance matrix of a dataset. There is one eigenvalue for each row in the covariance matrix. The computation of eigenvectors is also required to compute the principal component scores. The eigenvalues and eigenvectors are computed using the following equation, where A is the covariance matrix of interest, I is the identity matrix, k is a positive integer, λ is the eigenvalue and v is the eigenvector:

$$(A - \lambda I)^k \, \mathrm{v} = 0$$

What is important to understand for current purposes is that the principal components are sorted by descending order as a function of their eigenvalues (each row is a principal component): the higher the eigenvalues, the higher the variance explained. The more the variance is explained, the more useful the principal component is in summarizing the data. The part of variance explained by a principal component is computed as a function of the eigenvalues by dividing the eigenvalue of the principal component of interest by the sum of the eigenvalues. The equation is as follows, where *partVar* is the part of variance explained and *eigen* is the eigenvalue:

$$partVar_i = \frac{eigen_i}{\sum_{i=i}^{n} eigen_i}$$

Although this equation is more complex than this short explanation, the scores can be thought of as the matrix multiplication (operator `%*%` in R) of the factor loadings and the mean centered data matrix. In other words, the scores are made from the original data weighted by the factor loadings. The factor loadings are equal to the eigenvectors in an unrotated solution (we will see later what unrotated means). By definition, factorial scores also allow us to examine the relationship between attributes and dimensions: the higher the loading, the higher the strength of the relationship.

In what follows, we create our own PCA function. The reader is advised to use it for didactic purposes only and not for actual analyses, as some adjustments made in PCA are not implemented here. Once we have our function ready, we will examine the principal components in the iris dataset using our own solution and then compare it with the results of the `princomp()` function provided in the `stats` package. We will see that the mentioned adjustments usually leave the components that explain most variance (those of most interest) are largely unaffected, but less important components are quite affected:

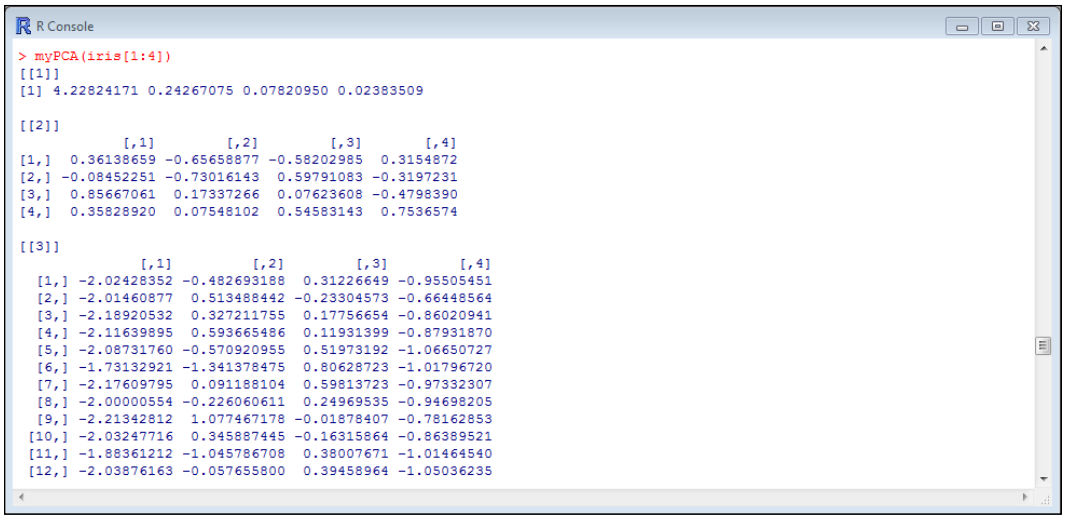
```
1  myPCA = function (df) {
2     eig = eigen(cov(df))
3     means = unlist(lapply(df,mean))
4     scores = scale(df, center = means) %*% eig$vectors
5     list(values = eig$values,vectors = eig$vectors, scores = scores)
6  }
```

In this function, we first compute in line 2 the eigenvalues and eigenvectors using function `eigen()`. We then create on line 3 a vector called `means` containing the means for each of the attributes of the original dataset. In line 4, we then use this vector to center the values around the means and multiply (matrix multiplication) the resulting data frame by the eigenvectors data frame to produce the principal component scores. In line 5, we create an object (then returned by the function) of a class list containing the eigenvalues, the eigenvectors, and the principal component scores.

In what follows, we examine the principal components in the `iris` dataset (omitting the `Species` attribute):

```
my_pca = myPCA(iris[1:4])
my_pca
```

The following screenshot provides part of the output:



PCA results using the custom myPCA function

Now let's compare the eigenvalues in our function and the corresponding values when using `princomp()`. We first run `princomp()` on the data and assign the result to the `pca` object. The `sdev` component of the `princomp` objects is the square root of the eigenvalues. In order to obtain a comparable metric, we apply this transformation to the eigenvalues in the `my_pca` object for comparison using the `sqrt()` function:

```
pca = princomp(iris[1:4], scores = T)
cbind(unlist(lapply(my_pca[1], sqrt)), pca$sdev)
```

The following is the output:

	[,1]	[,2]
values1	**2.0562689**	**2.0494032**
values2	**0.4926162**	**0.4909714**
values3	**0.2796596**	**0.2787259**
values4	**0.1543862**	**0.1538707**

We can see that the standard deviation of the principal components (the eigenvalues squared) is quite similar between our function in the first column and `princomp()` in the second column.

The summary of the object generated by `princomp()` provides the amount of variance explained by each of the components:

```
summary(pca)
```

The output is as follows:

Importance of components	**Comp. 1**	**Comp. 2**	**Comp. 3**	**Comp. 4**
Standard deviation	**2.0494032**	**0.49097143**	**0.27872586**	**0.153870700**
Proportion of variance	**0.9246187**	**0.05306648**	**0.01710261**	**0.005212184**
Cumulative proportion	**0.9246187**	**0.97768521**	**0.99478782**	**1.000000000**

We can compare these values to the result using our function:

```
my_pca[[1]] / sum(my_pca[[1]])
```

The output appears as follows:

```
[1] 0.924618723 0.053066483 0.017102610 0.005212184
```

We can see that the values are identical and, clearly, the first component is the most important, as it explains more than 92 percent of the variance in the dataset.

The eigenvectors are not directly accessible from the `princomp()` function, but we computed the scores, which are much more useful. So, let's compare the scores generated by our function and `princomp()`. We will do so by measuring the correlation between them. We start by creating a dataset with the score of both analyses, then run the correlation analysis but display only the part of the output we are interested in, that is, the correlation between our principal component scores, and those of `princomp()`:

```
scores = cbind(matrix(unlist(my_pca[3]),ncol = 4), pca$scores)
round(cor(scores)[1:4,5:8],3)
```

The output is provided here:

Comp. 1	Comp. 2	Comp. 3	Comp. 4
0.999	**-0.008**	**0.014**	**0.032**
0.059	**-0.985**	**-0.151**	**-0.055**
-0.293	**-0.44**	**-0.848**	**-0.041**
0.936	**0.219**	**0.056**	**0.269**

On the diagonal lines, we can see that the correlation between our scores and the scores of `princomp()` is almost perfect for the first two components (the sign is not important in this case as we will explain later). The third component is already less correlated, and the correlation with the last is not that good. In what follows, we will discover how to use PCA with existing datasets.

Learning PCA in R

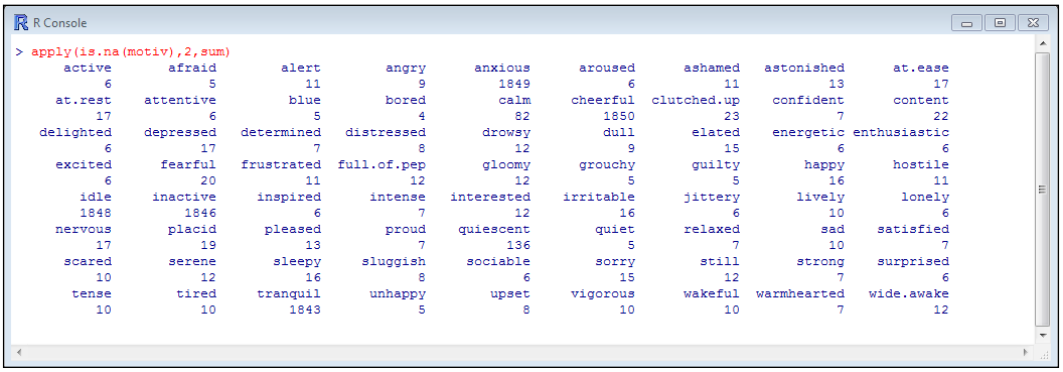
In this section, we will learn more about how to use PCA in order to obtain knowledge from data, and ultimately reduce the number of attributes (also called features). The first dataset we will use is the `msg` dataset from the `psych` package. The motivational state questionnaire (`msq`) dataset is composed of `92` attributes, of which `72` are ratings of adjectives by `3,896` participants describing their mood. We will only use these `72` attributes for current purpose, which is the exploration of the structure of the questionnaire. We will therefore start by installing and loading the package and the data, and assign the data we are interested in (the mentioned `72` attributes) to an object called `motiv`:

```
install.packages("psych")
library(psych)
data(msq)
motiv = msq[,1:72]
```

Dealing with missing values

Missing values are a common problem in real-life datasets, such as the one we use here. There are several ways to deal with them, but here we will only mention omitting the cases where missing values are encountered. Let's see how many missing values (we will call them NAs) there are for each attribute in this dataset:

```
apply(is.na(motiv),2,sum)
```



A view of the missing data in the dataset

We can see that, for many attributes, this is unproblematic (few missing values). But, in the case of several attributes, the number of NAs is quite high (`anxious`: `1849`; `cheerful`: `1850`, `idle`: `1848`, `inactive`: `1846`, `tranquil`: `1843`). The most probable explanation is that these items have been dropped from some of the samples in which the data has been collected.

Removing all cases with NAs from the dataset would dramatically reduce the number of rows of the dataset in this particular case. Try the following in your console to verify this claim:

```
na.omit(motiv)
```

For this reason, we will simply deal with the problem by removing these attributes from the analysis. Remember, there are other ways, such as data imputation, to solve this issue while keeping the attributes. In order to do so, we first need to know which column number corresponds to the attributes we want to suppress. An easy way to do this is simply by printing the names of each column vertically (using `cbind()` for something it was not exactly made for); we will omit cases with missing values on other attributes from the analysis later:

```
head(cbind(names(motiv)),5)
```

Here, we only print the result for the first five columns of the data frame:

	[,1]
[1,]	active
[2,]	afraid
[3,]	alert
[4,]	angry
[5,]	anxious

We invite the reader to verify on his screen that we suppress the correct columns from the analysis using the following vector to match the attributes:

```
ToSuppress = c(5, 15, 37, 38, 66)
```

Let's check whether it is correct:

```
names(motiv[ToSuppress])
```

Here is the output:

```
[1] "anxious"  "cheerful" "idle"     "inactive" "tranquil"
```

Selecting how many components are relevant

In the following line of code, we run PCA on a data frame where we suppress the columns with several NAs (using `-ToSuppress`), and omit cases with NAs on other attributes:

```
Pca = princomp(na.omit(motiv[,-ToSuppress]))
```

There are as many components as there are attributes, as can be verified using the following code, which displays the dimensions of an object provided as an argument:

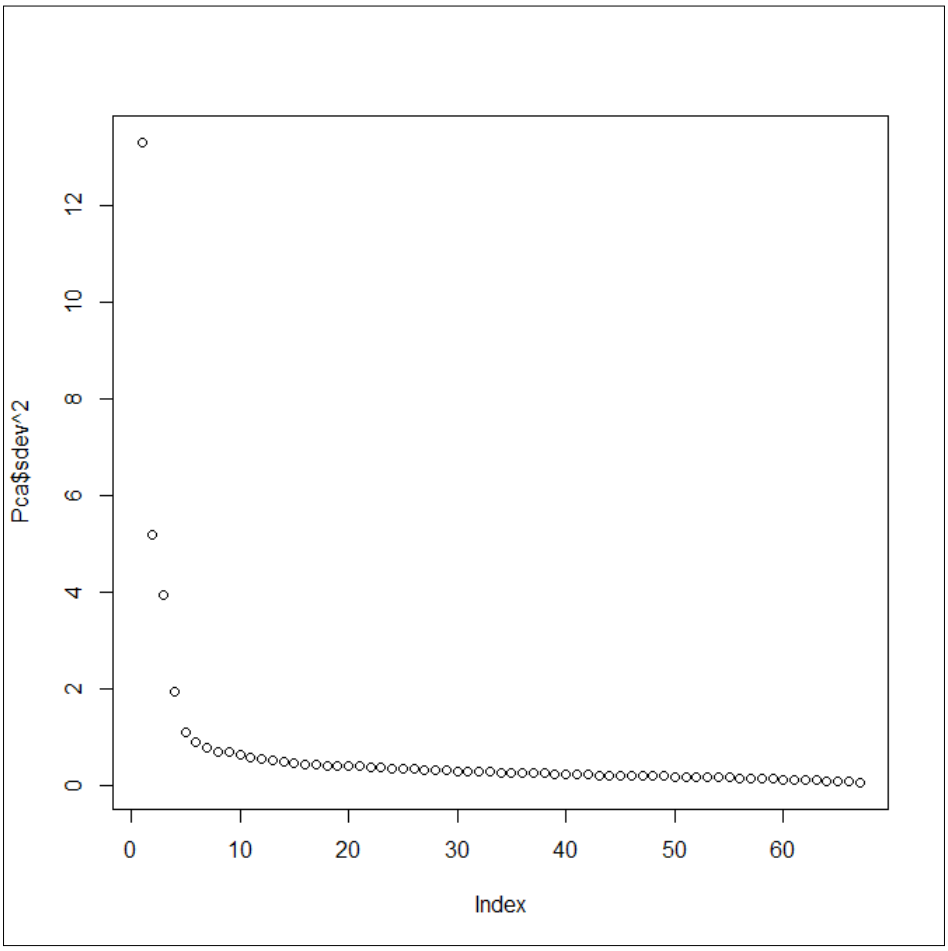
```
dim(Pca$loadings)
```

Here is the output:

```
[1] 67 67
```

The element loadings of `princomp` object `Pca` contain the loadings of each attribute on each principal component. These were defined when discussing the inner working of PCA. We will inspect those later, once we determine the number of meaningful principal components. This can be done using a scree test, a plot of the eigenvalues. The square root of the eigenvalues is given in the `stdev` element of `princomp` objects (called `Pca` in this case), as we have seen earlier. We therefore need to square these values in order to obtain the eigenvalues back, as follows:

```
plot(Pca$sdev^2)
```



Cattel's scree test

The Cattel scree test is represented in the preceding figure. An elbow in the plot is observed after the fifth value. We will therefore consider that five factors represent our data sufficiently well. Another way to determine the number of relevant principal components, the Kaiser criterion, is to keep those with an eigenvalue higher than `1`. As can be seen on the graph (the eigenvalues are plotted on the *y* axis), the two criteria converge. We will therefore inspect the loadings on five factors. These are automatically ordered by decreasing eigenvalue (the square root of the eigenvalue, to be more specific).

Naming the components using the loadings

Now that we know the number of factors we need, we will rerun the analysis using the `principal()` function from the `psych` package. This will allow us to examine the loadings in a sorted fashion and make them independent from the others using a rotation. The `psych` package has already been loaded, as the data we are analyzing come from a dataset that is included in `psych`.

We can now run the analysis. This time, we want to apply an orthogonal rotation (`varimax`), in order to obtain independent factorial scores. Precisions are provided in the paper *Principal component analysis*, by Abdi and Williams (2010). We also want the analysis to use imputed data for missing values and estimate the scores for each observation on each retained principal component. We use the `print.psych()` function to print the sorted loadings, which will make interpretation of the principal components easier:

```
Pca2 = principal(motiv[,-ToSuppress],nfactors = 5,
   rotate = "varimax", missing = T, scores = T)
print.psych(Pca2, sort =T)
```

The annotated output displayed in the following screenshot has been slightly altered in order to allow it to fit on one page.

```
> print.psych(Pca2, sort =T)
Principal Components Analysis
Call: principal(r = motiv[, -ToSuppress], nfactors = 5, rotate = "varimax",
    scores = T, missing = T)
Standardized loadings (pattern matrix) based upon correlation matrix
              item   RC1   RC2   RC3   RC4   RC5   h2   u2
lively          40  0.82 -0.06 -0.05 -0.27 -0.03 0.75 0.25
excited         26  0.80 -0.08 -0.11 -0.13  0.10 0.69 0.31
enthusiastic    25  0.80 -0.16  0.03 -0.12  0.07 0.68 0.32
full.of.pep     29  0.79 -0.07 -0.10 -0.31 -0.05 0.73 0.27
energetic       24  0.79 -0.04 -0.06 -0.36 -0.04 0.75 0.25
active           1  0.78 -0.02 -0.07 -0.27 -0.06 0.70 0.30
elated          23  0.77 -0.07 -0.02 -0.05  0.01 0.60 0.40
vigorous        64  0.75  0.05 -0.11 -0.25 -0.06 0.63 0.37
happy           33  0.72 -0.30  0.24 -0.09  0.00 0.68 0.32
pleased         44  0.71 -0.22  0.24 -0.05  0.04 0.62 0.38
aroused          5  0.71  0.03 -0.10 -0.20  0.03 0.56 0.44
inspired        35  0.71  0.02 -0.02 -0.09  0.13 0.52 0.48
proud           45  0.70 -0.05  0.14  0.04 -0.03 0.51 0.49
determined      19  0.69  0.09  0.03 -0.05  0.17 0.51 0.49
strong          58  0.69  0.07  0.07 -0.07 -0.05 0.49 0.51
delighted       17  0.68 -0.21  0.05 -0.02  0.06 0.52 0.48
sociable        55  0.66 -0.23  0.10 -0.11  0.01 0.51 0.49
confident       15  0.63 -0.11  0.29 -0.08 -0.15 0.53 0.47
alert            3  0.63  0.02  0.05 -0.54 -0.06 0.69 0.31
warmhearted     66  0.61 -0.25  0.35  0.01  0.09 0.57 0.43
satisfied       50  0.61 -0.28  0.33 -0.05 -0.03 0.57 0.43
interested      37  0.61 -0.14  0.20 -0.17  0.14 0.48 0.52
attentive       10  0.61 -0.02  0.15 -0.46  0.00 0.60 0.40
wakeful         65  0.56 -0.02  0.07 -0.56 -0.05 0.64 0.36
content         16  0.55 -0.31  0.46 -0.09 -0.07 0.62 0.38
intense         36  0.53  0.34 -0.22 -0.05  0.11 0.46 0.54
surprised       59  0.40  0.12 -0.12  0.01  0.21 0.24 0.76
astonished       7  0.34  0.17 -0.09  0.04  0.30 0.25 0.75
unhappy         62 -0.16  0.74  0.00  0.03  0.25 0.64 0.36
irritable       38 -0.09  0.70 -0.22  0.22 -0.03 0.60 0.40
grouchy         31 -0.13  0.70 -0.11  0.28 -0.03 0.60 0.40
upset           63 -0.08  0.70 -0.06  0.02  0.36 0.62 0.38
gloomy          30 -0.18  0.69  0.04  0.16  0.22 0.59 0.41
blue            11 -0.14  0.69  0.08  0.04  0.28 0.58 0.42
depressed       18 -0.18  0.68  0.03  0.05  0.31 0.60 0.40
frustrated      28 -0.01  0.68 -0.11  0.06  0.32 0.58 0.42
sad             49 -0.10  0.68  0.07  0.01  0.34 0.59 0.41
angry            4  0.00  0.66 -0.13  0.00  0.21 0.50 0.50
hostile         34  0.00  0.64 -0.21  0.11 -0.01 0.47 0.53
distressed      20  0.02  0.59 -0.07  0.04  0.47 0.57 0.43
lonely          41 -0.10  0.57  0.11  0.01  0.26 0.42 0.58
tense           60  0.18  0.52 -0.34  0.03  0.31 0.51 0.49
clutched.up     14  0.17  0.50 -0.27  0.07  0.24 0.41 0.59
bored           12 -0.20  0.34  0.05  0.31 -0.21 0.29 0.71
calm            13  0.11 -0.18  0.73  0.00 -0.10 0.59 0.41
serene          52  0.14 -0.15  0.71  0.05 -0.03 0.55 0.45
at.ease          8  0.30 -0.26  0.67 -0.11 -0.15 0.63 0.37
relaxed         48  0.22 -0.27  0.65 -0.05 -0.11 0.56 0.44
still           57 -0.16  0.06  0.64  0.19 -0.03 0.47 0.53
at.rest          9  0.18 -0.12  0.64 -0.17 -0.08 0.49 0.51
placid          43 -0.04  0.05  0.59  0.18 -0.01 0.38 0.62
quiet           47 -0.22  0.24  0.52  0.17  0.05 0.40 0.60
quiescent       46  0.12  0.14  0.41  0.14  0.05 0.22 0.78
jittery         39  0.34  0.23 -0.39 -0.03  0.19 0.36 0.64
sleepy          53 -0.23  0.15  0.11  0.84  0.06 0.80 0.20
drowsy          21 -0.22  0.16  0.13  0.83  0.04 0.79 0.21
tired           61 -0.27  0.18  0.11  0.80  0.05 0.76 0.24
sluggish        54 -0.32  0.22  0.14  0.68  0.05 0.63 0.37
wide.awake      67  0.59  0.02  0.03 -0.59 -0.08 0.71 0.29
dull            22 -0.34  0.37  0.19  0.39 -0.01 0.44 0.56
afraid           2  0.07  0.26 -0.09  0.05  0.76 0.66 0.34
fearful         27  0.06  0.26 -0.08  0.04  0.74 0.63 0.37
scared          51  0.08  0.27 -0.12  0.05  0.72 0.62 0.38
ashamed          6 -0.03  0.33  0.01 -0.01  0.61 0.49 0.51
guilty          32  0.02  0.31  0.02  0.01  0.60 0.45 0.55
sorry           56 -0.02  0.45  0.06  0.01  0.57 0.53 0.47
nervous         42  0.18  0.30 -0.26  0.03  0.55 0.50 0.50

                       RC1  RC2  RC3  RC4  RC5
SS loadings          14.25 8.56 5.09 4.86 4.57
Proportion Var        0.21 0.13 0.08 0.07 0.07
Cumulative Var        0.21 0.34 0.42 0.49 0.56
Proportion Explained  0.38 0.23 0.14 0.13 0.12
Cumulative Proportion 0.38 0.61 0.75 0.88 1.00

Test of the hypothesis that 5 components are sufficient.

The degrees of freedom for the null model are  2211  and the objective function was  45.86
The degrees of freedom for the model are 1886  and the objective function was  6.28
The total number of observations was  3896  with MLE Chi Square =  24294.41  with prob <  0

Fit based upon off diagonal values = 0.99>
```

The results of the PCA with the principal() function using the varimax rotation

The names of the items are displayed in the first column of the first matrix of results. The column `item` is their order. The component loadings for the five retained components are displayed next (`RC1` to `RC5`). We will comment on `h2` and `u2` later. The loadings of the attributes can be used to name the principal components. As can be seen in the preceding screenshot, the first component could be called Positive arousal, the second component Negative arousal, the third Serenity, the fourth Exhaustion, and the last Fear. It is worth noting that the MSQ has theoretically four components obtained from two dimensions: energy and tension. Therefore, the fifth component we found is not accounted for by the model.

The `h2` value indicates the proportion of variance of the variable that is accounted for by the selected components. The `u2` value indicates the part of variance not explained by the components. The sum of `h2` and `u2` is `1`.

Below the first result matrix, a second matrix indicates the proportion of variance explained by the components on the whole dataset. For instance, the first component explains 21 percent of the variance in the dataset (`Proportion Var`) and 38 percent of the variance is explained by the five components. Overall, the five components explain 56 percent of the variance in the dataset (`Cumulative Var`). As a remainder, the purpose of PCA is to replace all the original attributes by the scores on these components in further analyses. This is what we discuss next.

PCA scores

At this point, you might wonder where data reduction comes into play. Well, the computation of the scores for each factor allows reducing the number of attributes for a dataset. This has been done in the previous section.

Accessing the PCA scores

We will now examine these scores thoroughly. Let's start by checking that the scores are indeed uncorrelated:

```
round(cor(Pca2$scores),3)
```

This is the output:

	RC1	RC2	RC3	RC4	RC5
RC1	**1.000**	**-0.001**	**0.000**	**0.000**	**0.001**
RC2	**-0.001**	**1.000**	**0.000**	**0.000**	**0.000**
RC3	**0.000**	**0.000**	**1.000**	**-0.001**	**0.001**
RC4	**0.000**	**0.000**	**-0.001**	**1.000**	**0.000**
RC5	**0.001**	**0.000**	**0.001**	**0.000**	**1.000**

As can be seen in the previous output, the correlations between the values are equal to or lower than `0.001` for every pair of components. The components are basically uncorrelated, as we requested. As the `principal()` function didn't remove any cases but imputed the data instead, we can now append the factor scores to the original dataset (a copy of it). Let's start by checking that indeed it contains the same number of rows:

```
nrow(Pca2$scores) == nrow(msq)
```

Here's the output:

```
[1] TRUE
```

Indeed, both datasets contain the same number of observations. As `principal()` retains the order of the original attributes, we can merge the PCA scores with the data:

```
bound = cbind(msq,Pca2$score)
```

PCA scores for analysis

As mentioned at the beginning of this section, the dataset also contains other attributes. As a preliminary to the next chapter, we will now examine the relationship between Extraversion, Neuroticism, Lie, Sociability, and Impulsivity (columns 80 to 84) and the PCA scores (columns 93 to 97).

```
Correl = cor(na.omit(cbind(bound[80:84],bound[93:97])))
```

In some disciplines, the significance of the estimates is essential. We will discuss significance in the next chapter. But, if you want to know the significance of the correlations in this example, use the following code instead:

```
install.packages("Hmisc")
library(Hmisc)
Correl_and_sig =
   rcorr(as.matrix(na.omit(cbind(bound[80:84],bound[93:97])))),
      type = "pearson")
Correl = Correl_and_sig[[1]]
Sig = Correl_and_sig[[3]]
```

The significance of the correlations is in the `Sig` matrix. The correlations are in the `Correl` object.

In a correlation table, the information below and above the diagonal line is repeated. Further, we are not interested in knowing the correlations between the original attributes and the correlations between the PCA scores (which we know to be approximately `0`). It therefore makes sense to select only a portion of the correlation matrix. We further want the results to be rounded to the second decimal in order to make the reading easier. The following line of code does all of this for us:

```
round(Correl, digits = 2)[6:10,1:5]
```

	Extraversion	Neuroticism	Lie	Sociability	Impulsivity
RC1	**0.16**	**-0.16**	**0.07**	**0.18**	**0.07**
RC2	**-0.06**	**0.27**	**-0.11**	**-0.1**	**0.01**
RC3	**-0.04**	**-0.16**	**0.06**	**-0.01**	**-0.06**
RC4	**0.08**	**0.07**	**-0.06**	**0.08**	**0.05**
RC5	**-0.11**	**0.18**	**0.02**	**-0.1**	**-0.07**

We can also display the significance of these correlations:

```
round(Sig, 2) [6:10,1:5]
```

The output is as follows (*p* values are displayed):

	Extraversion	Neuroticism	Lie	Sociability	Impulsivity
RC1	**0.000**	**0**	**0.000**	**0.000**	**0.000**
RC2	**0.000**	**0**	**0.000**	**0.000**	**0.430**
RC3	**0.013**	**0**	**0.000**	**0.648**	**0.000**
RC4	**0.000**	**0**	**0.000**	**0.000**	**0.004**
RC5	**0.000**	**0**	**0.318**	**0.000**	**0.000**

We will just comment that most correlations (see the preceding table) reach significance at the usual (but still arbitrary) threshold of $p < 0.05$. Exceptions are r(`RC5`, Lie) and r(`RC3`, Sociability).

The *p* value refers to the probability of obtaining the value or a higher value (a lower value if a negative value is obtained) if the correlation is actually 0 in the population.

We will only comment on the correlations of which the absolute value is `0.16` or more. This corresponds to at least (another arbitrary threshold) `2.5` percent of shared variance or more: The part of shared variance is the correlation squared. `RC1` (which we called Positive arousal) is positively related to `Extraversion` (for example, being outgoing) and negatively to `Neuroticism` (for example, being anxious) and `Sociability`. `RC2` (which we called Negative arousal) is positively correlated with `Neuroticism`. `RC3` (which we called Serenity) is negatively related to `Neuroticism`. `RC5` (which we called Fear) is positively related to `Neuroticism`.

It is worth noting that, in disciplines such as psychology, the mean of the items that load more on a component than on others are computed and used instead of the PCA scores. While this practice has its justifications, it does not maximize the explained variance as using PCA scores does.

PCA diagnostics

Here we will briefly discuss two diagnostics that can be performed on the dataset that will be subjected to analysis. These diagnostics should be performed by analyzing the data with PCA in order to ascertain that PCA is an optimal analysis for the considered data.

The first diagnostic is Bartlett's test of sphericity. This test examines the relationship between the variables together, instead of two by two as in a correlation. To be more specific, it tests whether the correlation matrix is different from an identity matrix (which has ones on the diagonals and zeroes elsewhere). The null hypothesis is that the variables are independent (no underlying structure). The paper *When is a correlation matrix appropriate for factor analysis? Some decision rules* by Dzuiban and Shirkey (1974) provides more information on the topic.

This test can be performed by the `cortest.normal()` function in the `psych` package. In this case, the first argument is the correlation matrix of the data we want to subject to PCA, and the `n1` argument is the number of cases. We will use the same dataset as in the PCA (we first assign the name `M` to it):

```
M = na.omit(motiv[-ToSuppress])
cortest.normal(cor(M), n1 = nrow(M))
```

The output is as follows:

```
Tests of correlation matrices
Call:cortest.normal(R1 = cor(M), n1 = nrow(M))
Chi Square value 829268  with df =  2211   with probability < 0
```

The last line of the output shows that the data subjected to the analysis is clearly different from an identity matrix. The probability to obtain these results if it were an identity matrix is close to `0` (do not pay attention to the > 0; it is simply extremely close to `0`). You might be curious about what the output of an identity matrix is. It turns out we have something close to it: the correlations of the PCA scores with `varimax` rotation that we examined before. Let's subject the scores to analysis:

```
cortest.normal(cor(Pca2$scores), n1 = nrow(Pca2$scores))
```

Tests of correlation matrices:

```
Call:cortest.normal(R1 = cor(Pca2$scores), n1 = nrow(Pca2$scores))
Chi Square value 0.01  with df =  10   with probability < 1
```

In this case, the results show that the correlation matrix is not significantly different from an identity matrix.

The other diagnostic is the **Kaiser Meyer Olkin** (**KMO**) index, which indicates the part of the data that can be explained by elements present in the data set. The higher this score, the more the proportion of the data is explainable, for instance, by PCA. The KMO (also called **Measure of Sample Adequacy**(**MSA**)) ranges from `0` (nothing is explainable) to `1` (everything is explainable). It can be returned for each item separately, or for the overall dataset. We will examine this value. It is the first component of the object returned by the `KMO()` function from the `psych` package. The second component is the list of the values for the individual items (not examined here). It simply takes a matrix, a data frame, or a correlation matrix as an argument. Let's run in on our data:

```
KMO(motiv)[1]
```

This returns a value of `0.9715381`, meaning that most of our data is explainable by the analysis.

Summary

In this chapter, we examined how PCA works. We briefly discussed how to deal with a dataset in cases where most values are missing on some attributes. We examined how to determine the adequate number of components and the proportion of variance they explain. We also saw how to give a meaningful name to the component. Finally, we began examining linear relationships between attributes using correlations. In the next chapter, we will discuss association rules with `apriori`.

7
Exploring Association Rules with Apriori

Association rules allow us to explore the relationship between items and sets of items. Such items can be as diverse as the contents of a market basket, the words used in sentences, the components of food products, and so on. Let's go back to the first example: transactions in a shop. Each transaction is composed of one or more items. We are interested in transactions of at least two items because, of course, there cannot be relationships between several items in the purchase of a single item. Imagine customers are purchasing the following sets of items, for which each row represents a transaction. We will use this example more thoroughly in this section:

- Cherry coke, chips, lemon
- Cherry coke, chicken wings, lemon
- Cherry coke, chips, chicken wings, lemon
- Chips, chicken wings, lemon
- Cherry coke, lemon, chips, chocolate cake

At first sight, you will notice that there seems to be an association between purchases of cherry coke and lemon, as four out of five (80 percent) transactions have both elements. Other possible associations are featured in this short list of transactions. Can you discover them?

Now, imagine doing this task for lists of thousands of transactions, comprising dozens of items. I bet you'd be bored before finishing this task, and you might miss important associations. The point of mining association rules is to do exactly that job in an automated way and derive indicators of the reliability of these associations.

In this chapter, we will:

- Examine the important concepts in associations rules
- Examine how `apriori`, an algorithm frequently used for such analysis, works
- Discover the use of market basket analysis with apriori in R

Apriori – basic concepts

There are some concepts about apriori that need to be understood before going further in this chapter: association rules, itemsets, support, confidence, and lift.

Association rules

An association rule is the explicit mention of a relationship in the data, in the form `X => Y`, where `X` (the antecedent) can be composed of one or several items. `X` is called an itemset. In what we will see, `Y` (the consequent) is always one single item. We might, for instance, be interested in what the antecedents of lemon are if we are interested in promoting the purchase of lemons.

Itemsets

Frequent itemsets are items or collections of items that occur frequently in transactions. Lemon is the most frequent itemset in the previous example, followed by cherry coke and chips. Itemsets are considered frequent if they occur more frequently than a specified threshold. This threshold is called **minimal support**. The omission of itemsets with support less than the minimal support is called **support pruning**. Itemsets are often described by their items between brackets: `{items}`.

Support

The support for an itemset is the proportion among all cases where the itemset of interest is present. As such, it allows estimation of how interesting an itemset or a rule is: when support is low, the interest is limited. The support for `{Lemon}` in our example is 1, because all transactions contain the purchase of Lemon. The support for `{Cherry Coke}` is 0.8 because Cherry Coke is purchased in four of five transactions (4/5 = 0.8). The support for `{Cherry Coke, Chips}` is 0.6 as three transactions contain both Cherry Coke and Chips. It is now your turn to do some math. Can you find the support for `{Chips, Chicken wings}`?

Confidence

Confidence is the proportion of cases of `X` where `X => Y`. This can be computed as the number of cases featuring `X` and `Y` divided by the number of cases featuring `X`. Let's consider the example of the association rule `{Cherry Coke, Chips} => Chicken wings`. As we have previously mentioned, the `{Cherry Coke, Chips}` itemset is present in three out of five transactions. Of these three transactions, chicken wings are only purchased in one transaction. So the confidence for the `{Cherry Coke, Chips} => Chicken wings` rule is 1/3 = 0.33.

Lift

Imagine both the antecedent and the consequent are frequent. For instance, consider the association rule, `{Lemon} => Cherry Coke`, in which lemon has a support of 1 and cherry coke a support of 0.8. Even without true relationship between the items, they could co-occur quite often. The proportion of cases where this can occur is computed as `support(X) * support(Y)`. In our case, 1 * 0.8 = 0.8. Lift is a measure of the improvement of the rule support over what can be expected by chance—that is, in comparison to the value we just computed. It is computed as `Support(X=>Y) / Support(X) * Support(Y)`.

In the current case:

Lift = support({Lemon, Cherry Coke}) / Support(Lemon) Support(Cherry Coke) =*

*(4/5) / ((5/5) * (4/5)) = 1*

As the lift value is not higher than 1, the rule does not explain the relationship between lemon and cherry coke better than could be expected by chance.

Now that we have discussed some basic terminology, we can continue with describing how the frequently used algorithm, `apriori`, works.

The inner working of apriori

The goal of `apriori` is to compute the frequent itemsets and the association rules in an efficient way, as well as to compute support and confidence for these. Going into the details of these computations is beyond the scope of this chapter. In what follows, we briefly examine how itemset generation and rule generation are accomplished.

Generating itemsets with support-based pruning

The most straightforward way to compute frequent itemsets would be to consider all the possible itemsets and discard those with support lower than minimal support. This is particularly inefficient, as generating itemsets and then discarding them is a waste of computation power. The goal is, of course, to generate only the itemsets that are useful for the analysis: those with support higher than minimal support. Let's continue with our previous example. The following table presents the same data using a binary representation:

Transaction	Cherry Coke	Chicken wings	Chips	Chocolate cake	Lemon
1	1	0	1	0	1
2	1	1	0	0	1
3	1	1	1	0	1
4	0	1	1	0	1
5	1	0	1	1	1

With a minimal support higher than 0.2, we can intuitively see that any itemset containing chocolate cake would be a waste of resources, as its support could not be higher than that of chocolate cake (which is lower than minimal support). The `{Cherry Coke, Chips, Lemon}` itemset is frequent, considering a minimal support of 0.6 (three out of five). We can intuitively see that all itemsets that feature items in `{Cherry Coke, Chips, Lemon}` are necessarily also frequent: `Support({Cherry Coke}) = 0.8`, `Support({Chips}) = 0.8`; `Support({Lemon}) = 1`, `Support({Cherry Coke, Chips}) = 0.6`; `Support({Cherry Coke, Lemon}) = 0.8`; and `Support({Chips, Lemon}) = 0.8`.

`Apriori` uses such strategies to generate itemsets. In short, it discards supersets of infrequent itemsets without having to compute their support. Subsets of frequent itemsets, which are necessarily frequent as well, are included as frequent itemsets. These are called **support-based pruning**.

Generating rules by using confidence-based pruning

`Apriori` generates rules by computing the possible association rules that have one item as consequent. By merging such rules with high confidence, it builds rules with two items as consequents and so on. If it is known that both association rules `{Lemon, Cherry Coke, Chicken Wings} => {Chips}` and `{Lemon, Chicken wings, Chips} => {Cherry Coke}` have high confidence, we know that `{Lemon, Chicken wings} => {Chips, Cherry Coke}` will have high confidence as well. As an exercise, examine the preceding table to discover whether or not these rules have high confidence.

We now know more about how `apriori` works. Let's start analyzing some data in R!

Analyzing data with apriori in R

In this section, we will continue with another supermarket example and analyze associations in the `Groceries` dataset. In order to use this dataset and to explore association rules in R, we need to install and load the `arules` package:

```
install.packages("arules")
library(arules)
data(Groceries)
```

Using apriori for basic analysis

We can now explore relationships between purchased products in this dataset. This dataset is already in a form exploitable by `apriori` (transactions). We will first use the default parameters as follows:

```
rules = apriori(Groceries)
```

The output is provided in the following screenshot:

```
> rules = apriori(Groceries)

parameter specification:
 confidence minval smax arem  aval originalSupport support minlen maxlen target   ext
        0.8    0.1    1 none FALSE            TRUE     0.1      1     10  rules FALSE

algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

apriori - find association rules with the apriori algorithm
version 4.21 (2004.05.09)        (c) 1996-2004   Christian Borgelt
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[169 item(s), 9835 transaction(s)] done [0.02s].
sorting and recoding items ... [8 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 done [0.00s].
writing ... [0 rule(s)] done [0.00s].
creating S4 object  ... done [0.00s].
```

Running apriori on the Groceries dataset with default parameters

We can see on the first line the parameters used in the analysis—in this case, the default. Around the middle of the output (where the arrow is), we see that there are `169` items in `9835` transactions in this dataset, and that `0` rules have been found (see second to last line). If you try this with your own data, you might find rules with the default parameters if you have very solid associations in your data. Here, the confidence and support thresholds are clearly too strict. Therefore, we will try again with a more relaxed minimal support and confidence, as follows:

```
rules = apriori(Groceries, parameter =
   list(support = 0.05, confidence = .1))
```

The output is provided on the top part of the following screenshot. We notice that five rules that satisfy both minimal support and confidence have been generated. We can examine these rules using the `inspect()` function (see the bottom part of the screenshot) as follows:

```
inspect(rules)
```

Let's examine the output:

```
> rules = apriori(Groceries, parameter = list(support = 0.05, confidence = .1))

parameter specification:
 confidence minval smax arem  aval originalSupport support minlen maxlen target   ext
        0.1    0.1    1 none FALSE            TRUE    0.05      1     10  rules FALSE

algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

apriori - find association rules with the apriori algorithm
version 4.21 (2004.05.09)        (c) 1996-2004   Christian Borgelt
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
sorting and recoding items ... [28 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 done [0.00s].
writing ... [14 rule(s)] done [0.00s].
creating S4 object  ... done [0.00s].
> inspect(rules)
   lhs                    rhs                    support     confidence     lift
1  {}                  => {bottled water}     0.11052364  0.1105236 1.000000
2  {}                  => {tropical fruit}    0.10493137  0.1049314 1.000000
3  {}                  => {root vegetables}   0.10899847  0.1089985 1.000000
4  {}                  => {soda}              0.17437722  0.1743772 1.000000
5  {}                  => {yogurt}            0.13950178  0.1395018 1.000000
6  {}                  => {rolls/buns}        0.18393493  0.1839349 1.000000
7  {}                  => {other vegetables}  0.19349263  0.1934926 1.000000
8  {}                  => {whole milk}        0.25551601  0.2555160 1.000000
9  {yogurt}            => {whole milk}        0.05602440  0.4016035 1.571735
10 {whole milk}        => {yogurt}            0.05602440  0.2192598 1.571735
11 {rolls/buns}        => {whole milk}        0.05663447  0.3079049 1.205032
12 {whole milk}        => {rolls/buns}        0.05663447  0.2216474 1.205032
13 {other vegetables}  => {whole milk}        0.07483477  0.3867578 1.513634
14 {whole milk}        => {other vegetables}  0.07483477  0.2928770 1.513634
```

Running apriori on the Groceries dataset with different support and confidence thresholds

The first column (`lhs`) displays the antecedents of each rule (X itemsets in our description in the first section). The second column (`rhs`) displays the consequents of the rules. We can see that whole milk is bought relatively frequently when yogurt, rolls/buns, and other vegetables are bought, and that other vegetables are bought frequently when milk is bought.

Detailed analysis with apriori

In this section, we will examine more complex relationships using the `ICU` dataset: a dataset about the outcomes of hospitalizations in the ICU. This dataset has `200` observations and `22` attributes. In order to access this dataset, we first need to install the package that contains it (`vcdExtra`). We will then have a look at the attributes:

```
install.packages("vcdExtra")
library(vcdExtra)
data(ICU)
summary(ICU)
```

The output is provided in the screenshot that follows. The attribute `died` refers to whether the patient died or not. The attributes `age`, `sex`, and `race` refer to the age (in years), sex (Female, Male) and race (black, white, or other) of the patient. The attribute `service` is the type of ICU the patient has been admitted into (medical or surgical). Attributes `cancer`, `renal`, `infect`, and `fracture` refer to the conditions the patient suffered during their stay in the ICU. The `cpr` attribute refers to whether or not the patient underwent cardiopulmonary resuscitation. The `systolic` and `heartrate` attributes refer to measures of cardiac activity. The `previcu` and `admit` attributes refer to whether the patient has been in the ICU previously, and whether the admission was elective or an emergency. Attributes `po2`, `ph`, `pco`, `bic`, and `creatin` refer to blood measures. Attributes `coma` and `uncons` refer to whether the patient has been in a coma or unconscious at any moment during the stay in the ICU.

```
> summary(ICU)
  died         age            sex         race         service    cancer    renal     infect
No :160   Min.   :16.00   Female: 76   Black: 15   Medical : 93   No :180   No :181   No :116
Yes: 40   1st Qu.:46.75   Male  :124   Other: 10   Surgical:107   Yes: 20   Yes: 19   Yes: 84
          Median :63.00                White:175
          Mean   :57.55
          3rd Qu.:72.00
          Max.   :92.00
 cpr         systolic        hrtrate       previcu         admit     fracture    po2
No :187   Min.   : 36.0   Min.   : 39.00   No :170   Elective : 53   No :185   >60 :184
Yes: 13   1st Qu.:110.0   1st Qu.: 80.00   Yes: 30   Emergency:147   Yes: 15   <=60: 16
          Median :130.0   Median : 96.00
          Mean   :132.3   Mean   : 98.92
          3rd Qu.:150.0   3rd Qu.:118.25
          Max.   :256.0   Max.   :192.00
     ph         pco         bic       creatin       coma           white     uncons
>=7.25:187   <=45:180   >=18:185   <=2:190   None  :185   White    : 25   No :185
<7.25 : 13   >45 : 20   <18 : 15   >2 : 10   Stupor:  5   Non-white:175   Yes: 15
                                             Coma  : 10
```

The summary of the ICU dataset

Preparing the data

As can be seen in the preceding screenshot, the `race` and `white` attributes are a bit redundant. We will therefore remove the `race` attribute (the fourth column). One can also see that there are both numerical (`age`, `systolic`, `hrtrate`) and categorical (`died`, `sex`, `race`) attributes in the dataset. We need only categorical attributes. Therefore, we will recode the numeric attributes into categorical attributes. We will do this with the `cut()` function on a copy of our dataset. This function simply creates bins by dividing the distance between the minimal and maximal values by the number of bins. As always, domain knowledge would be useful to create more meaningful bins. The reader is advised to take some time to become familiar with the dataset by typing `?ICU`:

```
ICU2 = ICU[-4]
ICU2$age = cut(ICU2$age, breaks = 4)
ICU2$systolic = cut(ICU2$systolic, breaks = 4)
ICU2$hrtrate = cut(ICU2$hrtrate, breaks = 4)
```

The dataset isn't in a format readily usable with `apriori` (transactions) yet. We first need to convert the coercions to transaction format before we can use it:

```
ICU_tr = as(ICU2, "transactions")
```

Using the `discretize()` function from the `arules` package allows the use of different types of binning. For instance, the following code line creates a new attribute named `agerec` with four bins of approximately equal frequency:

```
agerec = discretize(ICU$age, method="frequency",
categories=4)
```

Analyzing the data

We will first perform an analysis of all associations with thresholds of `.85` for support and `.95` for confidence, as follows:

```
rules = apriori (ICU_tr,
   parameter = list(support = .85, confidence = .95))
```

This leads to 43 rules. Let's have a closer look at these. Only the first 10 will be displayed in the following screenshot:

```
> inspect(rules)
   lhs              rhs             support confidence     lift
1  {}            => {creatin=<=2}   0.950  0.9500000 1.000000
2  {cancer=No}   => {creatin=<=2}   0.855  0.9500000 1.000000
3  {pco=<=45}    => {po2=>60}       0.860  0.9555556 1.038647
4  {pco=<=45}    => {ph=>=7.25}     0.875  0.9722222 1.039810
5  {renal=No}    => {cpr=No}        0.860  0.9502762 1.016338
6  {renal=No}    => {ph=>=7.25}     0.860  0.9502762 1.016338
7  {renal=No}    => {creatin=<=2}   0.880  0.9723757 1.023553
8  {po2=>60}     => {ph=>=7.25}     0.880  0.9565217 1.023018
9  {po2=>60}     => {creatin=<=2}   0.875  0.9510870 1.001144
10 {uncons=No}   => {coma=None}     0.925  1.0000000 1.081081
```

A view of association rules in the modified ICU dataset

With high confidence and support, the absence of a cancer is associated with creatin levels lower than or equal to 2. Low arterial concentration of carbonic oxide (<= 45) is associated with high blood concentration of oxygen (>= 60). Patients who did not have a history of renal failure did not need CPR, had a blood pH higher or equal to 7.25, and had a creatin level lower or equal to 2. Patients with a blood concentration of oxygen higher than or equal to 60 had a blood pH higher than or equal to 7.25 and a creatin level equal to or lower than 2. We let the reader interpret the rest of the relationships.

Interestingly, even though the confidence and support of the rules are high, the lift value is not higher than 1—that is, not better than could be expected by chance given the support of the antecedent and the consequent. Further testing might allow us to know more about this. The fisher exact test permits the testing of statistical interdependence in 2x2 tables. Each of our rules can be represented in a 2x2 table—for instance, the antecedent itemset in the rows (yes versus no) and the consequent in the columns (yes versus no). This test is available in the `interestMeasure()` function, as well as other tests and measures. I am not giving too much detail about this measure; instead, I am focusing on interpreting the results. Only the significance of the test is returned here. If you need the test value, please refer to the next subsection about how to export rules to a data frame, and then use the `fisher.test()` function from the `stats` package.

Regarding the significance value returned here (also known as the p value), when it is lower than 0.05 this means that the antecedent and the consequent are related for a particular rule. If it is higher than 0.05, it is considered non-significant, which means we cannot trust the rule. We will discover more about statistical distributions in the next chapter, but you might want to have a look now! Let's use this test to investigate the rules we generated before, rounding the results to two digits after the decimal point:

```
IM = interestMeasure(rules, "fishersExactTest", ICU_tr)
round(IM, digits=2)
```

The results are provided in the following table, in order of the rules:

[1]	**1**	**0.66**	**0**	**0**	**0.02**	**0.02**	**0**	**0**	**0.57**	**0**	**0**
[12]	**0**	**0.17**	**0**	**0.17**	**0**	**0**	**0.55**	**0.13**	**0.13**	**0**	**0**
[23]	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
[34]	**0**	**0**	**0**	**0.17**	**0**	**0**	**0.02**	**0.01**	**0**	**0.02**	

We can see that the first and second rules are non-significant, but the following are significant, in most instances. For instance, the `{cancer = No} => {creatin = <=2}` rule as well as `{pco = <=45} => {po2=>60}` are significant, which means that, when the antecedent is present, the consequent is present relatively more often than absent (and conversely).

We might have higher lift values when looking at what the antecedents of death in the ICU are. For this analysis, we will set the `rhs` parameter of the appearance argument to `died=Yes`. We will use lower confidence and support thresholds for this analysis as follows:

```
rulesDeath = apriori(ICU_tr,
   parameter = list(confidence = 0.3,support=.1),
   appearance = list(rhs = c("died=Yes"), default="lhs"))
```

The analysis returned 63 association rules with these confidence and support thresholds. Let's have a look. Again, we only display the first 10 association rules in the following screenshot:

```
> inspect(rulesDeath)
   lhs                          rhs          support confidence     lift
1  {infect=Yes,
    admit=Emergency}         => {died=Yes}    0.120  0.3478261 1.739130
2  {service=Medical,
    white=Non-white}         => {died=Yes}    0.120  0.3037975 1.518987
3  {infect=Yes,
    admit=Emergency,
    white=Non-white}         => {died=Yes}    0.115  0.3650794 1.825397
4  {infect=Yes,
    admit=Emergency,
    pco=<=45}                => {died=Yes}    0.100  0.3448276 1.724138
5  {cancer=No,
    infect=Yes,
    admit=Emergency}         => {died=Yes}    0.115  0.3432836 1.716418
6  {infect=Yes,
    admit=Emergency,
    po2=>60}                 => {died=Yes}    0.105  0.3620690 1.810345
7  {infect=Yes,
    admit=Emergency,
    fracture=No}             => {died=Yes}    0.110  0.3437500 1.718750
8  {infect=Yes,
    admit=Emergency,
    ph=>=7.25}               => {died=Yes}    0.100  0.3333333 1.666667
9  {cancer=No,
    infect=Yes,
    white=Non-white}         => {died=Yes}    0.110  0.3098592 1.549296
10 {infect=Yes,
    po2=>60,
    white=Non-white}         => {died=Yes}    0.100  0.3076923 1.538462
```

View of association rules in the modified ICU dataset, with patient death as a consequence

Instead of running `apriori` again, it is also possible to use the `subset()` function to select rules. The following line of code will create an object called `rulesComa` containing only rules where the consequent is `coma=None` using the existing rules. In the previous and following code, using `rhs` instead of `lhs` would have included the rules containing the selected antecedents instead of the selected consequents:

```
rulesComa = subset(rules, subset = rhs %in%
   "coma=None")
```

We can see that 34 percent of patients who were admitted in emergency and had an infection died in the ICU, as did 30 percent of patients who were non-white and whose ICU admission service was medical. Skipping through the results (rule 9 and 10), 31 percent of patients who didn't have a cancer, but were infected and non-white, died in the ICU, as did 31 percent of non-white patients who were infected with a blood oxygen concentration of 60 or more. We will let the reader examine the rest of the results. Looking at the lift value for all the rules here, we can see that these are a bit higher than 1, suggesting that the rules are more reliable than could be randomly expected.

Coercing association rules to a data frame

We have seen how to generate association rules using `apriori` with constraints, such as minimal support and confidence, or a given consequent. Suppose we want to perform some action on the rules—for example, sort them by decreasing lift values. The easiest way to do this is to coerce the association rules to a data frame, and then perform the operations as we would usually do. In what follows, we will use the rules we generated last and coerce them to a data frame using the `as()` function. We will then sort the data frame by decreasing lift and display the first five lines of the sorted data frame:

```
rulesDeath.df = as(rulesDeath,"data.frame")
rulesDeath.df.sorted =
   rulesDeath.df[order(rulesDeath.df$lift,decreasing = T),]
head(rulesDeath.df.sorted)
```

The following screenshot shows the output:

```
> head(rulesDeath.df.sorted)
                                                                                rules support confidence     lift
45 {cancer=No,infect=Yes,admit=Emergency,po2=>60,white=Non-white} => {died=Yes}   0.100  0.3921569 1.960784
19          {infect=Yes,admit=Emergency,po2=>60,white=Non-white} => {died=Yes}   0.100  0.3846154 1.923077
47     {cancer=No,infect=Yes,admit=Emergency,fracture=No,po2=>60} => {died=Yes}   0.100  0.3773585 1.886792
23              {infect=Yes,admit=Emergency,fracture=No,po2=>60} => {died=Yes}   0.100  0.3703704 1.851852
21                {cancer=No,infect=Yes,admit=Emergency,po2=>60} => {died=Yes}   0.105  0.3684211 1.842105
3                   {infect=Yes,admit=Emergency,white=Non-white} => {died=Yes}   0.115  0.3650794 1.825397
```

Data frame displaying the five highest lift values

The output shows that the following association rules have the highest lift values, and therefore have the highest performance compared to a random model:

- `{cancer=No,infect=Yes,admit=Emergency,po2=>60,white=Non-white} => {died=Yes}`
- `{infect=Yes,admit=Emergency,po2=>60,white=Non-white} => {died=Yes}`
- `{cancer=No,infect=Yes,admit=Emergency,fracture=No,po2=>60} => {died=Yes}`

Also note that the equivalent could have been obtained without coercing the association rules to a data frame with the following code line:

```
rulesDeath.sorted = sort(rulesDeath, by ="lift")
inspect(head(rulesDeath.sorted,5))
```

Visualizing association rules

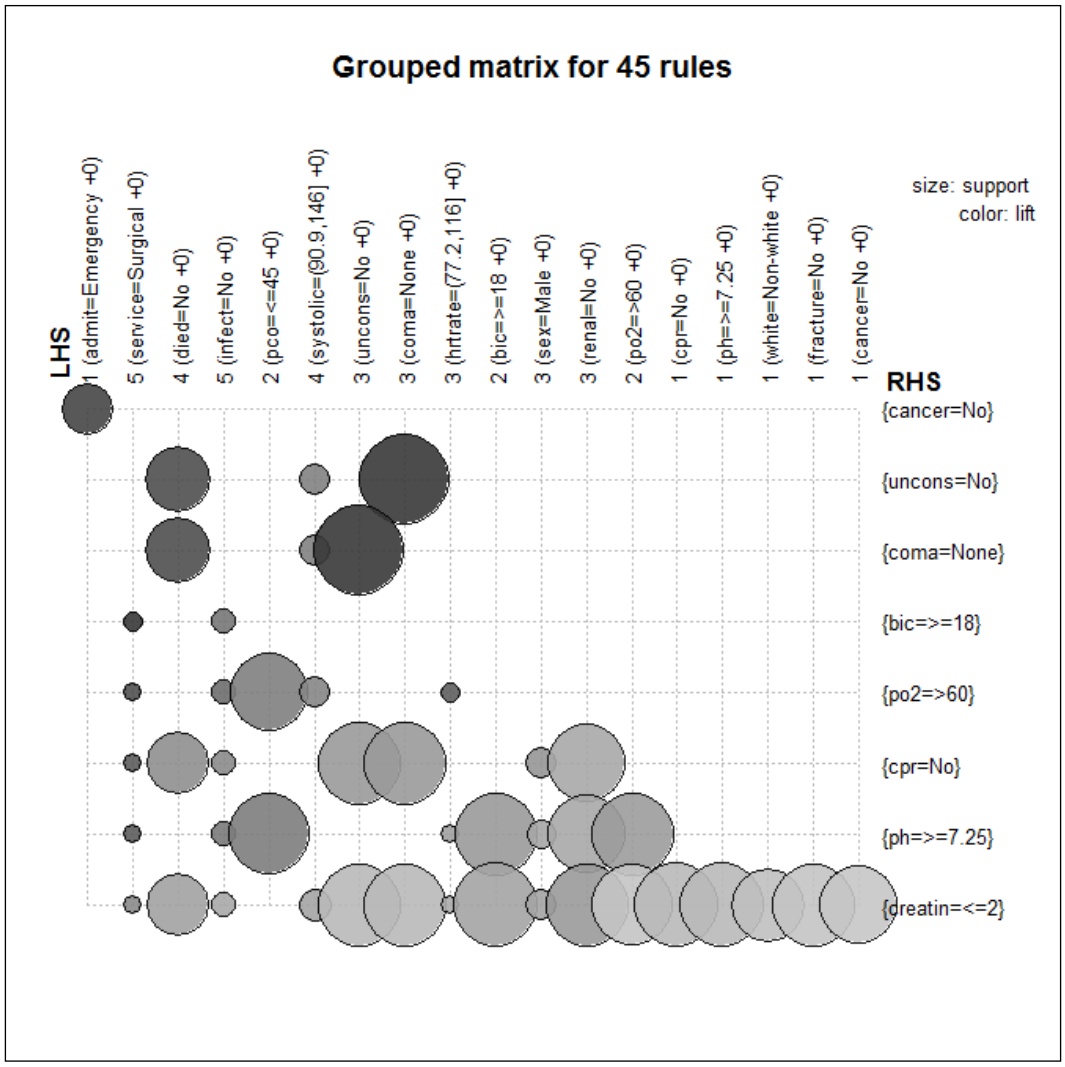
As for other analyses, visualization is an important tool when examining association rules. The `arulesViz` package provides visualization tools for association rules. There are plenty of examples in the *Visualizing Association Rules: Introduction to the R-extension Package arulesViz* article by Hasler and Chelluboina (2011). Here, we will only discover a plotting method that provides great added value to existing plotting tools in R, because of the informative graphics it provides, and also because of its simplicity. This method is called **Grouped matrix-based visualization**. It uses clustering to group association rules. The reader is advised to read the mentioned paper to learn more about it. Here are a few examples using our data. Let's start by installing and loading the package:

```
install.packages("arulesViz"); library(arulesViz).
```

For this purpose, we will first create a new set of association rules for the `ICU_tr` object, using a minimal support of 0.5. As confidence will not be displayed on the graph, we will set the threshold to 0.95—that is, all included rules will have high support. We will also use the `minlen = 2` and `maxlen = 2` parameters in order to obtain only rules with exactly one populated antecedent as follows:

```
morerules = apriori(ICU_tr, parameter=list(confidence=.95,
   support=.5, minlen=2,maxlen=2))
plot(morerules, method = "grouped")
```

The resulting graphic is shown in the following screenshot. The graphic displays antecedents in the columns and consequents in the rows. The size of the circles displays support (bigger circles mean higher support) and their color displays the lift value (a darker color means a higher lift). Looking at the graph, we can see that the `{uncons=No} => {coma = None}` rules have high support and high lift value. The `{coma = None} => {unconsc = No}` rule, of course, displays the same pattern. We can see that rules with a consequent creatin level <=2 have a generally low lift value. Even if the support is high, interpreting these rules must be done with caution, as they do not show an improvement compared to a random model. The reader is free to interpret the other rules displayed in the following screenshot:



Grouped matrix-based visualization of association rules in the ICU dataset

A final word regarding visualization of association rules: as you now know, it is easy to coerce association rules to a data frame. Once this is done, you can use the tools we discussed in *Chapter 2*, *Visualizing and Manipulating Data Using R*, and *Chapter 3*, *Data Visualization with Lattice*, and others to visualize the support, confidence, or lift of the rules, or perform analyses using these values.

Summary

In this chapter, we discovered some important concepts regarding association rules. In particular, we examined how important support, confidence, and lift measures are in the assessment of association rules, and that high support and confidence do not necessarily mean that an association rule is useful. We uncovered the efficient working of the apriori algorithm for mining association rules and discovered the use of `apriori` in R in mining several datasets. We have also seen that it is often necessary to recode some variables before being able to analyze the data. Finally, we have discovered Grouped matrix-based visualization.

In next chapter, we will examine statistical distributions and correlations.

8
Probability Distributions, Covariance, and Correlation

In *Chapter 6*, *Dimensionality Reduction with Principal Component Analysis*, we discussed principal component analysis. In the previous chapter, we discovered association rules using `apriori` in R. In this chapter, we will examine the following:

- Probability distributions
- A short introduction to descriptive statistics (mean and standard deviation)
- Covariance and correlation, notably what they mean and how they are computed
- How to perform correlation analysis in R

Probability distributions

In this section, we very briefly examine important distributions for common statistical problems with data consisting of quantities: the normal distribution and Student's t-distributions. We first introduce the idea of distributions with a discrete uniform distribution. We conclude with binomial distribution. We will try to be as non-technical as possible in this introduction to allow readers without statistical knowledge to follow easily; however, don't worry, we will be highly technical when explaining how to build functions that estimate correlations and regression coefficients.

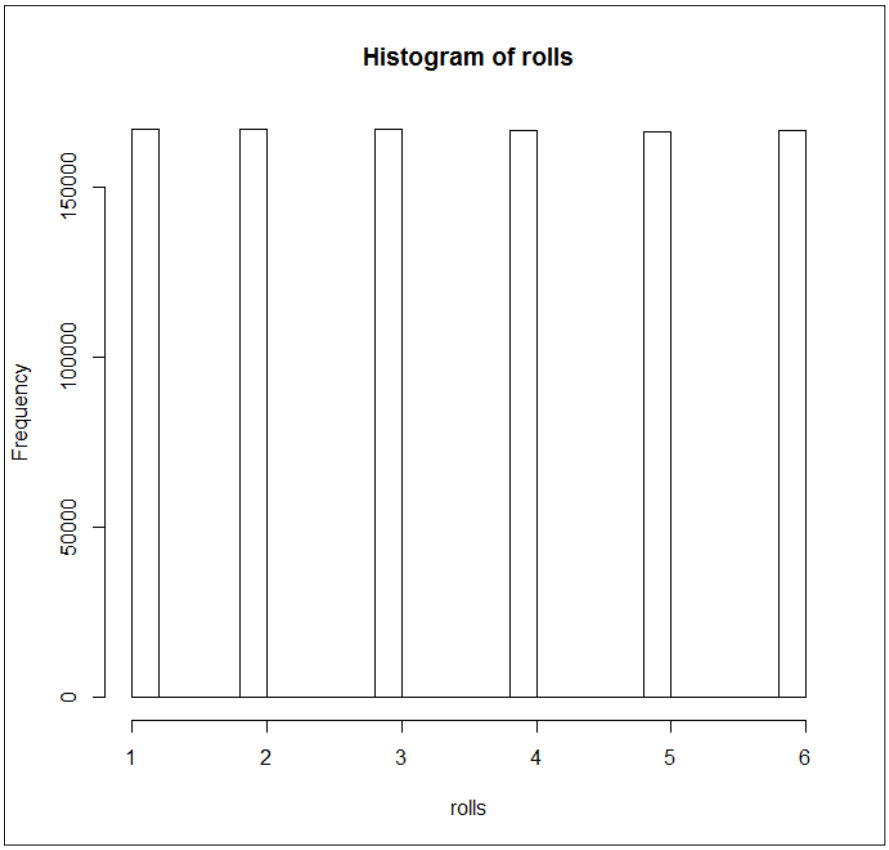
Introducing probability distributions

Here, we introduce the idea of distributions using discrete uniform and binomial distributions.

Discrete uniform distribution

You might remember that, in *Chapter 2*, *Visualizing and manipulating data using R*, we examined outcomes of the roulette game. We showed that each of the 37 numbers (0 to 36) in European roulette has an equal probability of occurring, 1/37, that is approximately 0.02702. This is called a Bernoulli trial. The outcome of infinite draws would form a uniform distribution. Another example we have examined is rolling a die. We have shown that the probability of each number occurring is 1/6, that is, approximately 0.16667. If we rolled a die infinite times (or a large number of times), the histogram of the outcomes will show that each number has occurred an equal number of times. Let's examine this with the following code:

```
rolls = sample(6, size = 1000000, replace = TRUE)
hist(rolls)
```



A histogram of a million die rolls

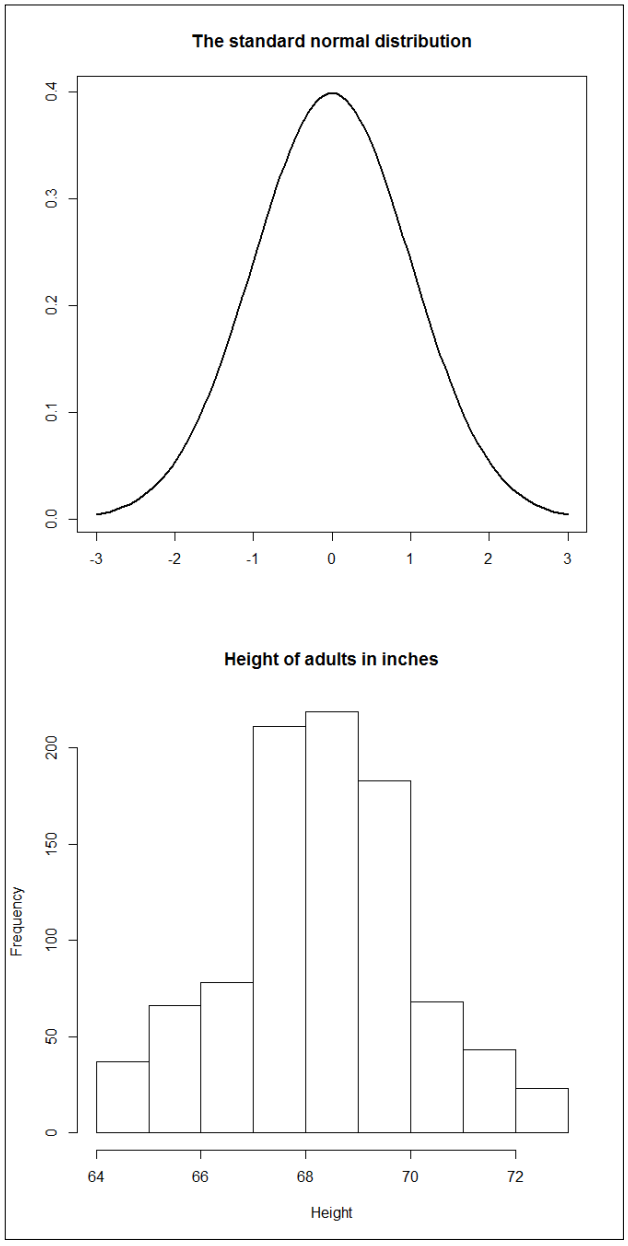
The normal distribution

Of course, not all attributes will follow such a distribution (in fact, most do not). Imagine the height of adults; you do not see as many people measuring, say, 140 cm, 180 cm, or 200 cm. Some heights are much more common than others, right? The normal distribution is usually applied to attributes such as height. The normal distribution acknowledges that some values of an attribute are much more likely to occur than other; values close to the arithmetic mean. The more a value is distant from the mean, the less likely it is to occur under the normal distribution. In fact, around 68 percent of the observations should have values between the mean minus one standard deviation and the mean plus one standard deviation, and 95 percent of observations should have values between the mean minus two standard deviations and the mean plus two standard deviations. It is important to know that the normal distribution assumes that the entire population is known, but it is widely used to analyze samples with a large number of observations.

The following code plots the shape (the probability density function) of the standard normal distribution (also called the z distribution), which has a mean of `0` and a standard deviation of `1`:

```
curve(dnorm(x, 0 ,1), lwd = 2, xlim=c(-3,3), xlab="", ylab="",
   main = "The standard normal distribution")
```

The following diagram (at the top of the frame) presents this plot. You will notice that this distribution is symmetrical.



The standard normal distribution (at the top of the frame) and a histogram of the heights of adults

Let's compare this shape to that of self-reported height (in inches). The data is from the `Galton` dataset of the `HistData` package:

```
install.packages("HistData")
library(HistData)
hist(Galton$parent, xlab="Height",
   main="Height of adults in inches")
```

The preceding diagram (bottom frame) presents this plot. We can see that the histogram of the height of adults is quite close to the shape of the normal distribution.

Inspecting data visually is not always enough. The Shapiro test (`shapiro.test()`) might be more informative. Let's create a vector that is not randomly distributed (`x1`) and another that is randomly distributed. We then test whether both attributes are normally distributed or not:

```
x1 = runif(1000)
x2 = rnorm(1000)
shapiro.test(x1)
```

The output for `x1` is provided here:

```
        Shapiro-Wilk normality test

data:  x1
W = 0.957, p-value < 2.2e-16
```

Given the extremely low `p-value`, it is almost impossible to obtain this result if the data in the population the sample was drawn from was normally distributed. As `p-value` is lower than 0.05 (the usual threshold), we can conclude that the data is not normally distributed—just as we designed it to be. Let's examine `x2`:

```
shapiro.test(x2)
```

The output for `x2` is provided here:

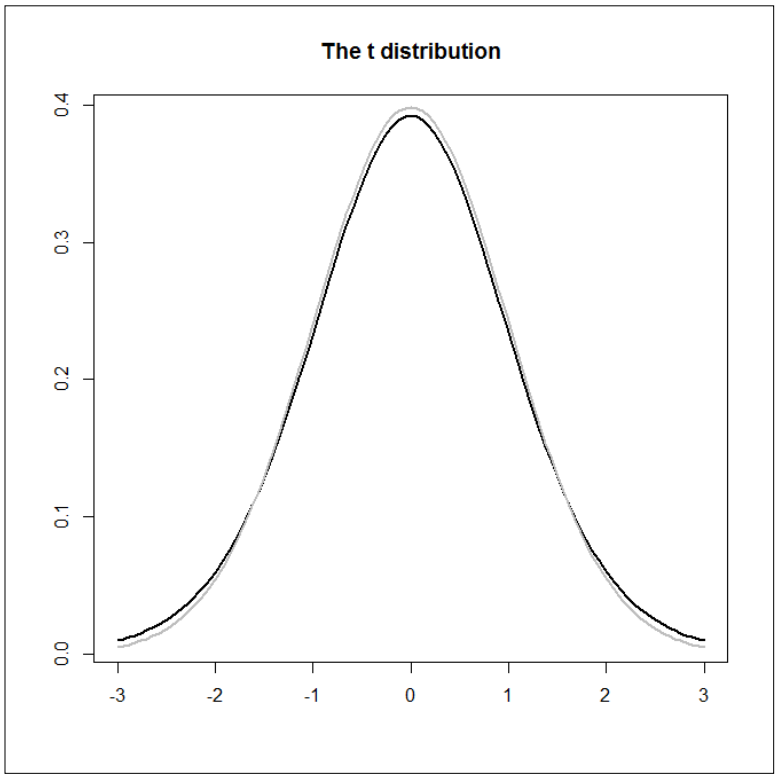
```
        Shapiro-Wilk normality test

data:  x2
W = 0.9988, p-value = 0.7777
```

The `p-value` is very different in this case. The probability (to obtain this result if the data, in the population that the sample was drawn from, was normally distributed) is about 77.7 percent. As the `p-value` is higher than 0.05, we conclude that the data is not different from a normal distribution.

The Student's t-distribution

The Student's t distribution resembles the normal distribution but is used when there are not many observations. What is important to know is that the sample size affects the shape of the t distribution. The degrees of freedom are the number of independent parameters that remain to be known before the data is fully known. Let's take the example of a sample of 10 observations on an attribute `a`. If we know the mean of the `a` attribute, we only need nine observations to know the value of the 10th, as we can rely on the mean and the nine known observations to infer the value of the remaining observation. The following plot shows the t distribution for 14 (in black) and 199 degrees of freedom (in gray) corresponding to sample sizes of 14 and 200:

```
curve(dt(x, 14), col = "black", lwd = 2, xlim=c(-3,3), xlab="",
   ylab="", main = "The t distribution")
curve(dt(x, 199), col = "grey", lwd = 2, add=T)
```



The t-distribution for 14 (in black) and 199 (in grey) degrees of freedom

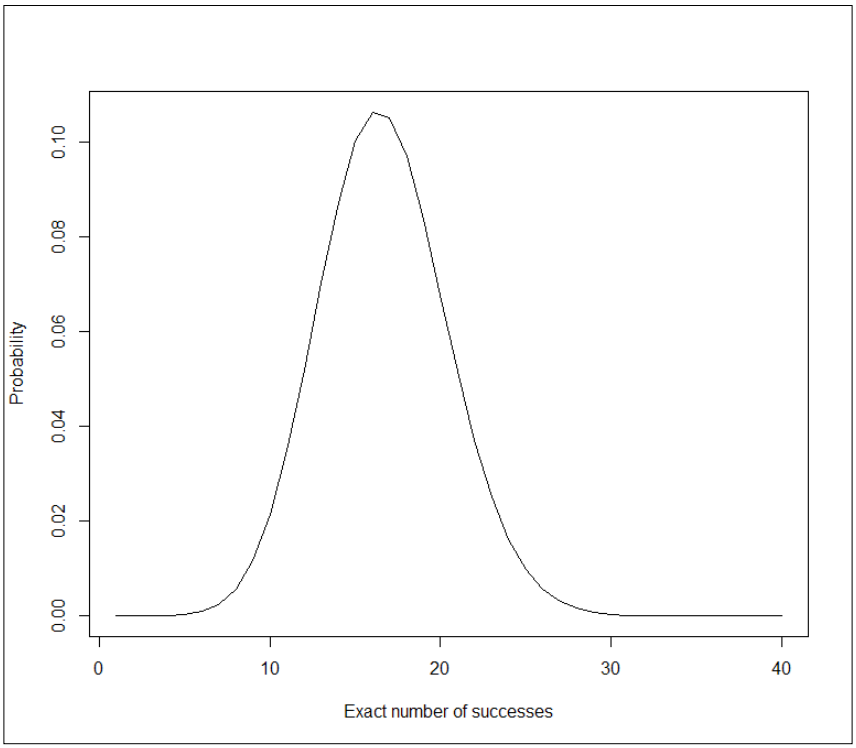
The binomial distribution

Now imagine that we want to know the chances that a specific outcome (say, 6 for instance) will have; we will call these successes for a precise number times when throwing a die repeatedly (for example, 100 times). This kind of questioning requires us to draw upon the binomial distribution.

We can compute this as follows: we first obtain the binomial coefficient, which is computed as the factorial of the number of throws of dice, divided by the number of the expected number of successes, minus the difference between the number of throws of dice and the expected number of successes. We multiply this result by the probability of a single success to the power of the expected number of successes multiplied by 1 (the probability of a single success), to the power of the difference between the number of throws of dice and the expected number of outcomes. Sorry, this was a little complicated. In order to show this more practically, let's use R code to examine the probability that a six exactly appears different number of times (from 0 to 20) in 100 draws. We will rely on the `choose()` function to compute the binomial coefficient. Here is the code for computing and plotting the probabilities:

```
p = 1/6
N = 100
n = 1
v = rep(1,40)
for (n in 0:40) {
   v[n] = choose(N, n) * (p^n) * (1 - p)^(N-n)
}
plot(v, type="l", xlab = "Exact number of successes", ylab =
  "Probability")
```

The resulting binomial distribution is displayed here:



Binomial distribution for 100 throws of dice (p = 1/6)

We can notice that we reach the highest probability of getting an exact number of successes when that number is around 16. It is almost impossible to get more than 30 successes in 100 throws of a fair dice.

The importance of distributions

Probability distributions are important because the significance of the tests performed are based upon them. Significance is tested by looking where the `t` or `z` values corresponding to the estimates (we will see how they are obtained later) lie on the distribution, with the corresponding degrees of freedom (for the t distribution). Further, most statistical tests, such as regression and correlation, assume the data is distributed normally. In most cases, a value must be at least in the extreme 5 percent of the distribution to be considered significant. This is the approach we will rely on in this chapter.

Covariance and correlation

Before going in depth into the topic of this section, let me remind the reader of three mathematical notions that will be used in this chapter: arithmetic mean, variance, and standard deviation. Some have been already discussed in other chapters, but a more formal definition is interesting for the purposes of the chapter.

The **arithmetic mean** is a measure of central tendency. Considering a sample of observations of an attribute—for instance, the height of individuals—the arithmetic mean is simply the sum of the values of the observations divided by the number of observations. We are interested in computing the mean height of three individuals measuring 160 cm, 170 cm, and 180 cm.

The formula for the mean is:

$$\bar{x} = \frac{\sum_{i=1}^{n} x}{n}$$

Type the following in the R console to compute the arithmetic mean of this sample:

```
(160 + 170 + 180) / 3
```

R outputs the following:

```
[1] 170
```

Check the solution by typing this:

```
mean(c(160,170,180))
```

Our computation of the mean was correct—R outputs:

```
[1] 170
```

Variance is a measure of dispersion of the data—that is, how different the values are in a sample or population. Considering a sample of observations of an attribute, the variance is computed as the sum of the squared mean subtracted observations (the sum of squares) divided by the number of observations minus 1 (the degrees of freedom). The formula for the variance is:

$$s^2 = \frac{\sum_{i=1}^{n}(x - \bar{x})^2}{n - 1}$$

Type the following to obtain the variance of heights of the three individuals:

```
Variance = ( (160-170)^2 + (170-170)^2 + (180-170)^2 ) / (3-1)
Variance
```

The output is as follows:

```
[1] 100
```

Now type the following to check our solution:

```
var(c(160,170,180))
```

The output is `100` again.

Standard deviation is another measure of dispersion. Unlike variance, it is expressed in the same unit as the data. The formula for standard deviation is as follows:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x - \bar{x})^2}{n-1}}$$

In other words, considering a sample of observations of an attribute, the standard deviation is the square root of the variance. Type the following to obtain the standard deviation of height in the three individuals presented previously:

```
sqrt(Variance)
```

The output is `10`. Now type the following:

```
sd(c(160,170,180))
```

Our computation of the standard deviation is correct; the output is `10` as well.

Let's now proceed to the main topics of this section.

Covariance and correlation are measures of how much two attributes are related—that is, how much they change together. For instance, one can easily figure out that the weight of individuals is related to their height (positive relation) more than to the length of their hair. The weight of individuals is most probably not at all related to the length of the last movie they have seen.

Covariance

The covariance of two normally distributed numeric attributes (data consisting of quantities or that can be treated as quantities) is computed as the sum of the mean subtracted observations of both attributes multiplied together, divided by the number of observations in the sample minus 1. The formula for the covariance is as follows:

$$cov(x,y) = \frac{\sum_{i=1}^{n}(x - \bar{x})^2 (y - \bar{y})^2}{n-1}$$

Imagine our three individuals (ordered by increasing height) weigh 55 kgs, 70 kgs, and 85 kgs. The arithmetic mean for the weight is therefore 70. The covariance of the two measures (height and weight) can be computed as follows:

```
Covariance = ((160-170) * (55-70) + (170-170) * (70-70) + (180-170) *
(85-70)) / (3-1)
Covariance
```

The output is `150`. Let's check it using the ad hoc function:

```
heights=c(160,170,180)
weights=c(55,70,85)
cov(heights,weights)
```

The output is 150 again! Our solution is correct.

The problem with covariance is that it is not a standardized measure of association—that is, the value of the measure depends upon the unit in which the attributes are measured. In our example, the heights were previously measured in centimeters. Let's try it with the same values converted to meters. For this purpose, we will divide the height measures in centimeters by `100`:

```
cov(heights/100, weights)
```

The result is `1.5`.

Measuring the height in inches and the weight in pounds would have led to a totally different covariance value. The covariance allows knowing about the direction of the relationship between two attributes, but not the magnitude of the relationship. This is because, as mentioned previously, the covariance is not a standardized measure of association between two attributes. The correlation does not present such a drawback.

Correlation

The correlation is a standardized measure of association between attributes. We have already mentioned the correlation a few times in the previous chapters, but let's examine it in more details.

Pearson's correlation

Pearson's correlation indicates the strength of the association between two normally distributed numeric attributes.

Before we continue on the topic, let's examine how the measure is computed. There are multiple ways to compute the correlation. The easiest to remember (considering we already know how to compute the covariance) is to simply divide the covariance by the product of the standard deviations of both attributes. Let's try again, using our previous example (measures of height in centimeter). To obtain the correlation of height and weight, we simply type:

```
Covariance / (sd(heights) * sd(weights))
```

The output is `1`. Let's check that we computed the correlation correctly, with the following line of code:

```
cor(heights,weights)
```

We were right! The output is `1` again. So what does this value mean? A correlation can have any value comprised between `-1` and `1`. A value of `-1` means a complete and negative correspondence of the changes in the values of the two attributes. A correlation of `1` means a complete and positive correspondence. A value of `0` means that the two attributes are independent of each other. The correlation allowed us to examine the strength of the correspondence of changes in height and weight in our example, that is, a perfect association between the two attributes.

It is worth mentioning that the Pearson's correlation only assesses linear relationships between the attributes. Let's have a look at what is meant here using the classic example of Anscombe's quartet. The dataset is part of the datasets package and, therefore, directly available to us:

```
data(anscombe)
```

The dataset is composed of eight attributes `x1`, `x2`, `x3`, `x4`, `y1`, `y2`, `y3`, and `y4`. We want to know the correlation between each of the x and y attributes that share their numbers (`x1` and `y1`; `x2`, and `y2`, …). This is achieved using the following code:

```
c1=cor(anscombe$x1, anscombe$y1)
c2=cor(anscombe$x2, anscombe$y2)
c3=cor(anscombe$x3, anscombe$y3)
```

```
c4=cor(anscombe$x4, anscombe$y4)
c1; c2; c3; c4
```

These four correlations have a value of 0.816. Does this mean that the relationship between each of the x and y attributes is the same? You might already suspect that this is not the case at all. In *Chapter 2*, *Visualizing and Manipulating Data Using R* and *Chapter 3*, *Data Visualization with Lattice*, we have looked at scatterplots already, and discovered that they allow visualizing the relationship between two attributes. Let's examine the relationships that we are interested in here. This demonstration is inspired by the example in the documentation (type `?anscombe`):

```
par(mfcol=c(1,4))
plot(anscombe$x1, anscombe$y1)
plot(anscombe$x2, anscombe$y2)
plot(anscombe$x3, anscombe$y3)
plot(anscombe$x4, anscombe$y4)
```

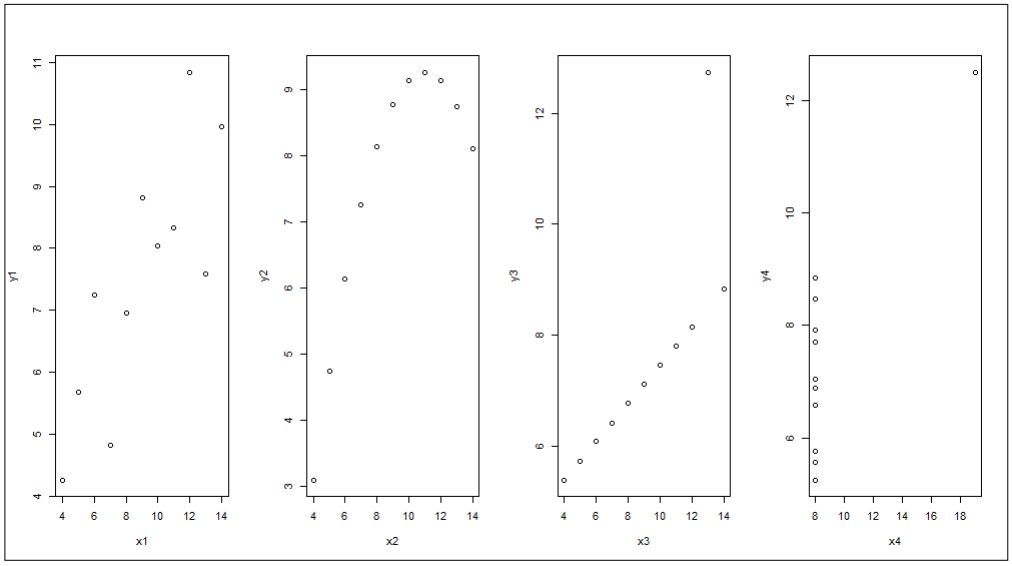


Figure 8.4: Scatterplots of four relationships yielding the same correlation (Anscombe's quartet)

From the preceding diagram, we can see that the relationship between `x1` and `y1` is positive and linear (`y1` increases as `x1` increases), but a bit noisy. The relationship between `x2` and `y2` is curvilinear—`y2` increases as `x2` increases up to an `x1` value of `11`, and decreases from the `x1` value of `12`. The relationship between `x3` and `y3` is linear but not as strong as for the first example. The correlation is so high because of a bivariate outlier at `x1 = 13`.

Finally, we can see that `x4` is constant (with values equal to `8`) and that, here again, a bivariate outlier is responsible for the strong association between `x4` and `y4`. Note that, without this observation, the correlation could not be computed as there would be no variance in `x4`. In this extreme case, looking at the plot would already have discouraged any further analysis of the relationship between `x4` and `y4`!

It is quite tempting to simply take note of the correlation between attributes, especially when they show interesting linear patterns in the data or confirm our hypotheses. We should refrain from drawing conclusions from a mere look at the correlations, and always visualize the data first. It is also necessary to always examine the significance of the correlations before drawing any conclusion from them. Therefore, I suggest using the `cor.test()` function instead of the `cor()` function, as it performs a significance test and informs about the 95 percent confidence intervals. In the case of the relationship between `x1` and `y1`, check the following code:

```
cor.test(x1, y1)
```

The output follows and shows that the correlation is significantly different than `0` (`p-value = 0.00217`). Relying on 95 percent confidence intervals, the true value of the correlation lies between lower and upper bounds of the confidence intervals. These are respectively of `0.42` and `0.95`. Note that we can know whether the correlation is significant by looking at the confidence interval; if it doesn't include `0`, the correlation is significant at the given threshold (here 95 percent). The estimate of the correlation lies exactly in between these values and is `0.814205`, which we can round to `0.816`.

Pearson's product-moment correlation is as follows:

```
data:  x1 and y1
t = 4.2415, df = 9, p-value = 0.00217
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.4243912 0.9506933
sample estimates:
      cor
0.8164205
```

We obtain the part of variance shared between two attributes by squaring the correlation. In the present case, the two attributes share *0.8164205^2*100 = 66.65424* % of their variance.

For the interested reader, assessing the significance of the correlation requires the computation of the corresponding `t` value, which is obtained by dividing the correlation by the square root of 1 minus the correlation, divided by the degrees of freedom (the sample size minus 2). The significance is then obtained from the Student's t distribution.

Spearman's correlation

When the attributes are not normally distributed, Spearman's correlation should be used. This correlation coefficient first ranks the observations of both attributes included in the analysis. It then computes the differences between the ranks of each observation on these two attributes. Finally, it computes the correlation coefficient. In the computation, 6 times the sum of the observation-wise differences, divided by the number of observations multiplied by the number of observations squared minus `1`, is subtracted from `1`.

Let's examine this in an example using the following attributes named `A` and `B`:

```
A = c(3,4,2,6,7)
B = c(4,3,1,6,5)
```

We first compute the ranks of observations of `A` and `B`:

```
RankA = rank(A);  RankB = rank(B)
```

The spearman correlation can be computed like this:

```
1 - ( (6 * sum((RankA-RankB)^2)) / (5* (5^2 -1)) )
```

The output is `0.8`. Let's check if our answer is right:

```
cor(A,B,method = "spearman")
```

We did it correctly; the output using the function in R is `0.8` as well:

As mentioned earlier, it is necessary to know the significance of the correlation, which we can obtain using the following code:

```
cor.test(A,B, method = "spearman")
```

The output follows and can be interpreted as the output of the test for a Pearson's correlation (but note, no confidence intervals are provided here):

```
        Spearman's rank correlation rho
data:  A and B
S = 4, p-value = 0.1333
alternative hypothesis: true rho is not equal to 0
sample estimates:
rho
0.8
```

In this case, the correlation, although high, is not significantly different from `0`, as the p value fails to reach the threshold of 0.05 (`p-value = 0.1333`).

Summary

In this chapter, we have briefly examined what statistical distributions are and why they are important when doing statistical inference. After a short introduction to descriptive statistics (mean and standard deviation), we have discovered how to obtain covariance and correlation coefficients programmatically. In the next chapter, we will discuss regression analysis in R.

9
Linear Regression

In *Chapter 7*, *Exploring Association Rules with Apriori*, we examined association rules with `apriori`. In the previous chapter, we have notably examined statistical distribution and the relationships between two attributes using several measures of association. These didn't infer any causation between the attributes, only dependence. If we have normally distributed attributes and want to examine how one attribute affects another attribute, we can rely on simple linear regression instead. If we want to examine how several attributes affect an attribute, we can rely on multiple linear regressions.

In this chapter, we will notably:

- Build and use our own simple linear regression algorithm
- Create multiple linear regression models in R
- Perform diagnostic tests of such models
- Score new data using a linear regression model
- Examine how well the model predicts the new data
- Have a quick look at robust regression and bootstrapping

Understanding simple regression

In simple regression, we analyze the relationship between a predictor (the attribute we think to be the cause) and the criterion (the attribute we think is the consequence). There are two very important parameters (among others) that result from a regression analysis:

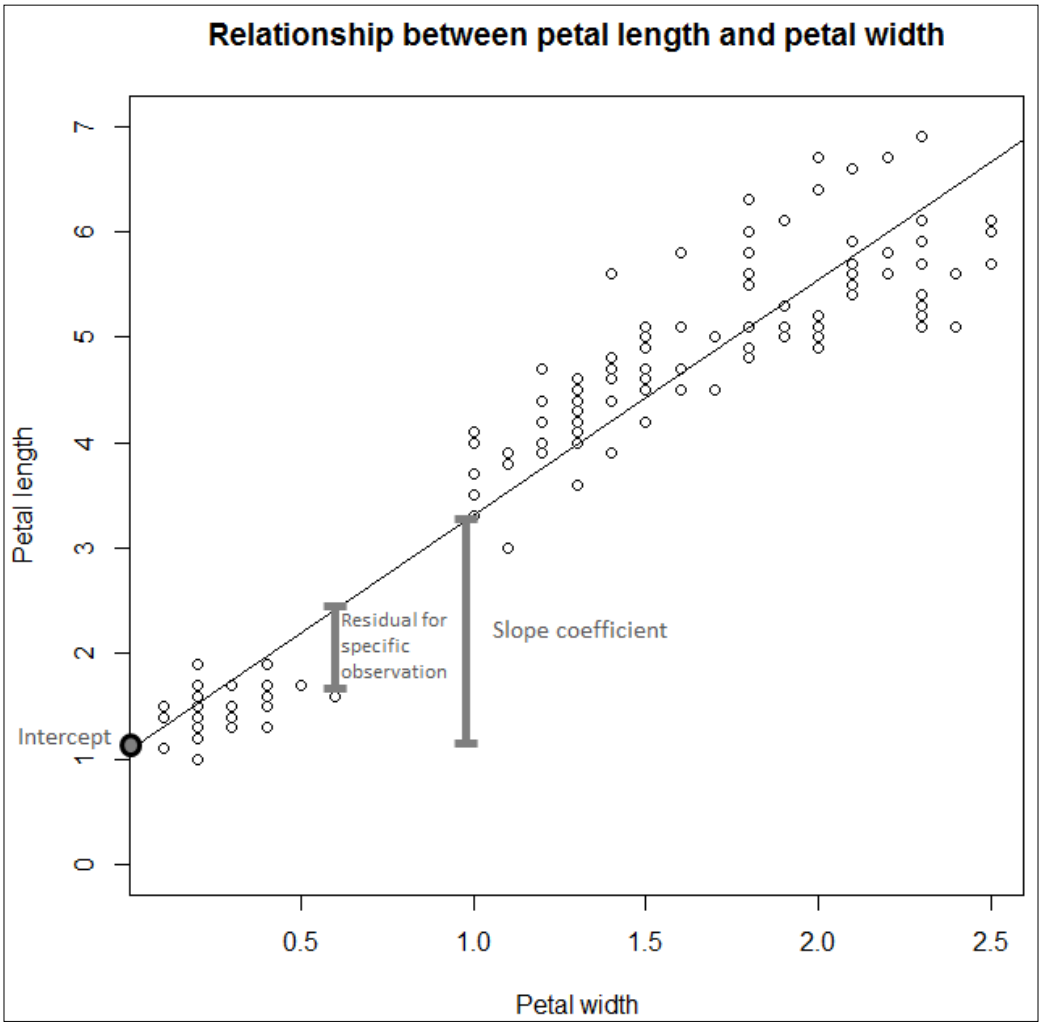
- The intercept: This is the average value of the criterion when the predictor is 0, which is when the effect of the predictor is partialed out
- The slope coefficient: This indicates by how many units, on average, the criterion changes (with reference to the intercept) when the predictor increases by one unit

Regression seeks to obtain the values that explain the relationship the best, but such a model only seldom reflects the relationship entirely. Indeed, measurement error, but also attributes that are not included in the analysis affect also the data. The residuals express the deviation of the observed data points to the model. Its value is the vertical distance from a point to the regression line. Let's examine this with an example of the `iris` dataset. We have already seen that the dataset contains data about iris flowers. For the purpose of this example, we will consider the petal length as the criterion and the petal width as the predictor.

We will now create a scatterplot, with the petal width on the *x* axis and the petal length on the *y* axis, in order to display the data points on these dimensions. We will then compute the regression model and use it to add the regression line to the plot. This should look familiar, as we have already done this in *Chapter 2*, *Visualizing and Manipulating Data Using R*, and *Chapter 3*, *Data Visualization with Lattice*, when discussing plots in R. This redundancy is not accidental—plotting data and their relationship is one of the most important aspects of analyzing data:

```
1  plot(iris$Petal.Length ~ iris$Petal.Width,
2     main = "Relationship between petal length and petal width",
3     xlab = "Petal width", ylab = "Petal length")
4  iris.lm = lm(iris$Petal.Length ~ iris$Petal.Width)
5  abline(iris.lm)
```

The following plot has been manually annotated in gray in order to make the discussion more intelligible:



Annotated scatterplot of petal length and petal width (iris dataset) with regression line

In this example, the intercept is around 1.1 and the slope coefficient is around 2.2 (about 3.3 minus the intercept). As mentioned before, the vertical distance from the line to a point is the residual for that specific point.

Now that this is understood, we will examine how these values can be computed, before going into the results in greater depth.

Computing the intercept and slope coefficient

In simple regression, data can be modeled as the intercept, plus the slope multiplied by the value of the predictor, plus the residual. We are now going to explain how to compute these.

The slope coefficient can be computed in several ways. One is to multiply the correlation coefficient by the standard deviation of the criterion divided by the standard deviation of the predictor. Another is to first compute the value corresponding to the number of observations multiplied by: the sum of the observation-wise products of the criterion and the predictor minus the sum of the values of the predictor multiplied by the sum of the values of the criterion multiplied. The result is then divided by the number of observations multiplied by the sum of the squared values of the predictor minus the squared sum of the predictors. Another way is to rely on matrix computations, which we will not examine here.

The intercept can simply be computed as the mean of the criterion minus the slope coefficient multiplied by the mean of the predictor.

Let's take the same example as before to compute the regression coefficient (using the two computations we have seen), and the intercept.

To compute the slope coefficient using the first way presented, we start by computing the correlation coefficient of the petal length and petal width, and the standard deviation of the predictor and criterion. We then perform the described computation:

```
SlopeCoef = cor(iris$Petal.Length,iris$Petal.Width) *
   (sd(iris$Petal.Length) / sd(iris$Petal.Width))
SlopeCoef
```

The outputted value is `2.22994`. Let's program a function that implements the other way to compute the slope we've seen. The criterion will be called `y` and the predictor `x`:

```
1  coeffs = function (y,x) {
2      ( (length(y) * sum( y*x)) -
3         (sum( y) * sum(x)) )  /
4         (length(y) * sum(x^2) - sum(x)^2)
5  }
6  coeffs(iris$Petal.Length, iris$Petal.Width)
```

The output is `2.22994` again. Let's compare it to the model we built using the `lm()` function previously:

```
iris.lm
```

The output first reminds us of the function call we used, which is pretty handy as working with many different models can sometimes be confusing. The intercept and slope coefficients are provided. As can be seen, with a difference of 0.07, we are very close with our own computations:

```
Call:
lm(formula = iris$Petal.Length ~ iris$Petal.Width)
Coefficients:
     (Intercept)  iris$Petal.Width
           1.084             2.230
```

Let's now build a function that computes the intercept and returns both intercept and coefficient:

```
1  regress =  function (y,x) {
2     slope = coeffs(y,x)
3     intercept = mean(y) - (slope * mean(x))
4     model = c(intercept, slope)
5     names(model) = c("intercept", "slope")
6     model
7  }
8  model = regress(iris$Petal.Length, iris$Petal.Width)
9  model
```

The value of the intercept is `1.08358`, which is the same (but unrounded) as that in the output of the `lm()` function.

Now that we have seen how to compute the intercept and slope coefficients, let's turn to how residuals can be obtained.

Obtaining the residuals

Let's say it once more; the criterion value of any observation can be obtained by summing the intercept, the slope coefficient multiplied by its predictor value, and the residuals. As we now know the intercept and slope coefficient and have the data, we can compute the residuals as follows:

```
resids = function (y,x, model) {
  y - model[1] - (model[2] * x)
}
```

Let's compute the residuals for our model:

```
Residuals = resids(iris$Petal.Length, iris$Petal.Width, model)
```

Let's display the first six residuals:

```
head(round(Residuals,2))
```

The output is as follows:

```
[1] -0.13  -0.13  -0.23  -0.03  -0.13  -0.28
```

Let's also display the residuals computed by the `lm()` function:

```
head(round(residuals(iris.lm),2))
```

Comparing the preceding and following outputs, we can see that the values are the same:

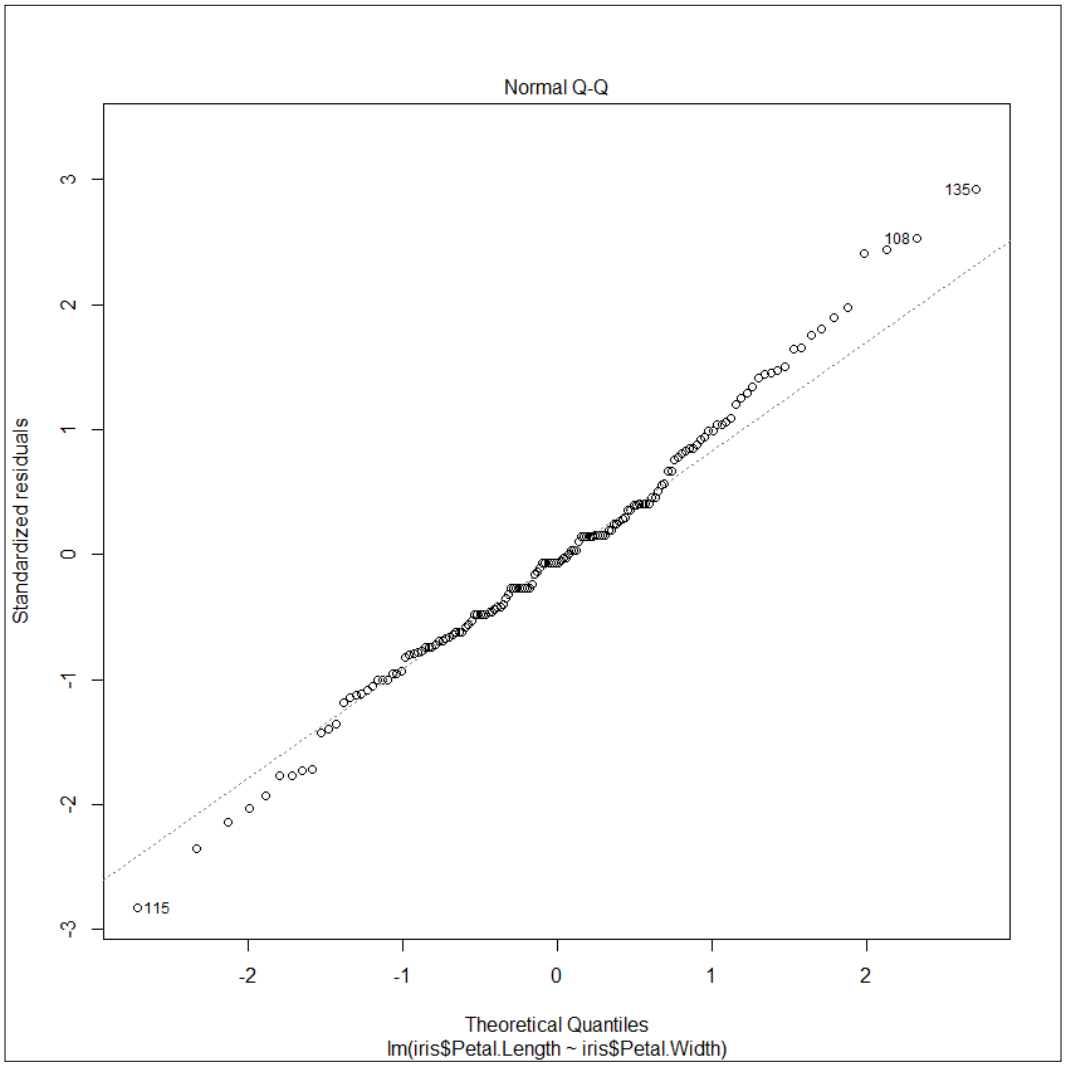
1	2	3	4	5	6
-0.13	**-0.13**	**-0.23**	**-0.03**	**-0.13**	**-0.28**

Residuals are very important for several reasons. There is not enough space to explain all of them. Let's just say that an assumption of regression is that residuals are normally distributed. If residuals are not normally distributed, this can be caused by the non-normal distribution of our data, and/or because of nonlinear relationships between the predictors and criterion.

The **Quantile-Quantile plot** (**Q-Q plot**) allows visually comparing the actual distribution of the residuals in terms of quantiles, to the theoretical distribution (quantiles as well). This plot can be easily obtained in R. Type the following:

```
plot(iris.lm)
```

Then simply click on the R Graphics window (or, if you are using RStudio, hit **Return** in the console) until the R Graphics window displays the following:



Q-Q plot of the residuals in the iris.lm model

As the residuals fit on the dotted line reasonably well, we can conclude they are distributed normally. But notice that the values in both extremities do not fit the line so well, so these observations might threaten the reliability of our results. We should be fine but still, we will examine robust regressions and bootstrapping at the end of the chapter.

We will examine the linearity of residuals further in our practical example in the next section.

Computing the significance of the coefficient

As we have seen in the first section of the chapter, determining the significance of the estimates is essential for interpretation; even a big coefficient cannot be interpreted if it is not significantly different from 0. Here, you will learn a little more about the computation of the significance for simple regression:

1. The first thing we need to do is to compute the standard error of the slope coefficient (a value that assesses its precision).
2. We obtain the standard error by first taking the square root of: the sum of the squared residuals (SSR) divided by the degrees of freedom (DF—that is, the number of observations minus two).
3. We then divide this value (called S in the following code) by the square root of the squared mean subtracted values of x.
4. After we obtain the standard error, we can compute a t-score by dividing the slope coefficient by the standard error.
5. The score is then compared to 0 on a t-distribution.

There is also a significance test for the intercept. In order to compute the standard error, we first:

1. Compute 1 divided by the number of observations, plus the square mean of the predictor, divided by the sum of the squared mean subtracted values of the predictor.
2. We take the square root of this value and multiply it by the value S that we saw previously.

After we obtain the standard error for the intercept, its t-score can be computed as seen previously. The following code implements this and returns the standard error, t score, and significance for both the slope coefficient and the intercept of a simple linear regression:

```
1  Significance = function (y, x, model) {
2     SSE = sum (resids(y,x,model)^2)
3     DF = length(y)-2
4     S = sqrt ( SSE / DF)
5     SEslope = S / sqrt(sum( (x-mean(x))^2 ))
6     tslope = model[2] / SEslope
7     sigslope = 2*(1-pt(abs(tslope),DF))
8     SEintercept = S * sqrt((1/length(y) +
9     mean(x)^2 / sum( (x- mean(x))^2)))
10     tintercept = model[1] / SEintercept
11     sigintercept = 2*(1-pt(abs(tintercept),DF))
```

```
    RES = c(SEslope, tslope, sigslope, SEintercept,
       tintercept, sigintercept)
    names(RES) = c("SE slope", "T slope", "sig slope",
       "SE intercept", "t intercept", "sig intercept")
    RES
}
```

Let's see this in practice using the example of the iris dataset.

```
round(Significance(iris$Petal.Length,iris$Petal.Width, model), 3)
```

The output is as follows:

SE slope	T slope	sig slope	SE intercept	T intercept	sig intercept
0.051	43.387	0	0.073	14.85	0

Let's now compare our results with the results obtained with `lm()`:

```
summary(iris.lm)
```

The following output shows that we obtain the exact same results using our function:

Call:
lm(formula = iris$Petal.Length ~ iris$Petal.Width)

Residuals:

Min	1Q	Median	3Q	Max
-1.33542	-0.30347	-0.02955	0.25776	1.39453

Coefficients:

	Estimate	Std. Error	t value	Pr(>|t|)	
(Intercept)	1.08356	0.07297	14.85	<2e-16	***
iris$Petal.Width	2.22994	0.0514	43.39	<2e-16	***

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

Residual standard error: 0.4782 on 148 degrees of freedom
Multiple R-squared: 0.9271, Adjusted R-squared: 0.9266
F-statistic: 1882 on 1 and 148 DF, p-value: < 2.2e-16

If you have been following all along, you now know how to write functions that compute the intercept, slope coefficient and residuals, standard errors, t values, and significance for simple regression. Congratulations! We wish to mention that, as always, the code is presented for pedagogical purposes only, and that the tools provided by default in R or the packages available on CRAN should always be used for real applications.

In the next section, we will briefly examine how multiple regression works, then switch to a more practical section using multiple regression. We will discover important new concepts in the process.

Working with multiple regression

In multiple regression, we are interested in testing the impact of several predictors on a criterion, instead of just one in simple regression. Here, the value of the observations can be computed as the intercept plus the slope coefficient multiplied by the predictor value (for each predictor) plus the residuals.

The analysis estimates the unique contribution of the predictors to the criterion—that is, each obtained slope coefficient value (there is one for each predictor) and the intercepts that are controlled for the influence of the other predictors on the criterion. We are not going to detail the calculation of the slope and intercept for multiple regression as this involves more complex explanations than for simple regression and will not add much to your understanding; most of what we have seen (except the calculation of the coefficients and degrees of freedom) remains valid for multiple regression. We will now directly skip to a more practical section.

Analyzing data in R: correlation and regression

In the previous section, we saw how to perform simple regression analysis in R. We also saw that multiple regression is more complex to compute but have discussed that most of what we have already seen applies to multiple regression as well.

First steps in the data analysis

In what follows, we will use a dataset of 40 cases generated from a covariance matrix obtained from a subsample of real data we collected, which is about burnout components, work satisfaction, work-family conflict, and organizational commitment in hospitals. There are six attributes in the dataset that we will analyze here; all are self-assessments made by nurses:

- `Commit`: Commitment to their hospital (criterion here)
- `Exhaust`: Emotional exhaustion (one of the three components of burnout)
- `Depers`: Depersonalization (one of the three components of burnout)
- `Accompl`: Accomplishment (one of the three components of burnout)
- `WorkSat`: Work satisfaction
- `WFC`: Work-family conflict

Our goal here is to understand how burnout dimensions and work satisfaction affect commitment of nurses to their hospital.

We start by generating the data and examining the correlation table and significance. Make sure the `matcov.txt` file is in your working directory before running this code:

```
1  library(psych)
2  install.packages("MASS"); library(MASS)
3  matcov = unlist(read.csv("matcov.txt", header=F))
4  covs = matrix(matcov, 6, 6)
5  means = c(4.47,14.95,4.87,36.08,5,1.88)
6  set.seed(987)
7  nurses = data.frame(mvrnorm(n=40, means, covs))
8  colnames(nurses)= c("Commit","Exhaus","Depers","Accompl",
9     "WorkSat","WFC")
10  corr.test(nurses)
```

The output is provided here:

Call:corr.test(x = nurses)

Correlation matrix

	Commit	Exhaus	Depers	Accompl	WorkSat	WFC
Commit	**1.00**	**-0.64**	**-0.27**	**0.27**	**0.76**	**-0.52**
Exhaus	**-0.64**	**1.00**	**0.20**	**0.12**	**-0.50**	**0.68**
Depers	**-0.27**	**0.2**	**1.00**	**0.04**	**-0.51**	**-0.02**
Accompl	**0.27**	**0.12**	**0.04**	**1.00**	**0.23**	**0.15**
WorkSat	**0.76**	**-0.50**	**-0.51**	**0.23**	**1.00**	**-0.39**
WFC	**-0.52**	**0.68**	**-0.02**	**0.15**	**-0.39**	**1.00**

Sample Size

[1] 40

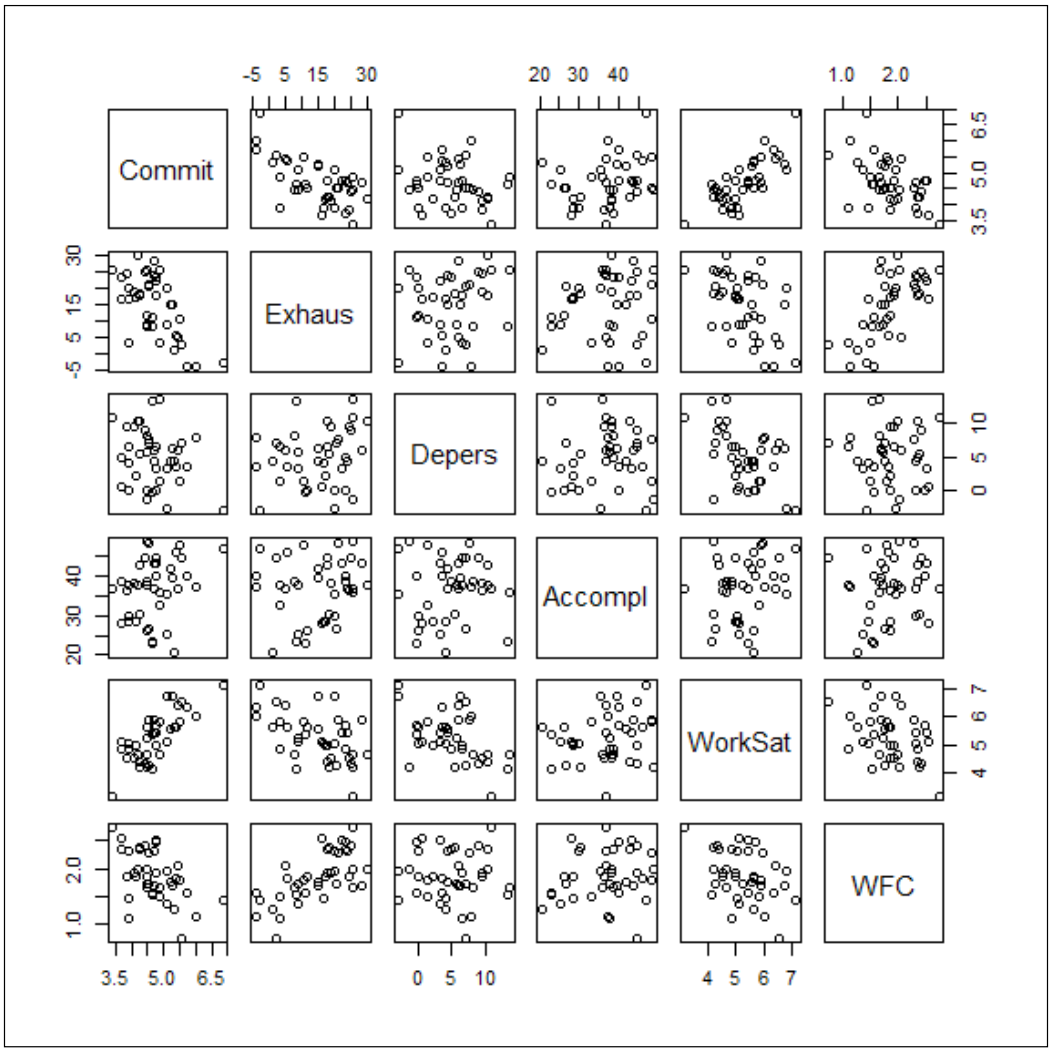
Probability values (entries above the diagonal are adjusted for multiple tests)

	Commit	Exhaus	Depers	Accompl	WorkSat	WFC
Commit	**0.00**	**0.00**	**0.73**	**0.73**	**0.00**	**0.01**
Exhaus	**0.00**	**0.00**	**1.00**	**1.00**	**0.01**	**0.00**
Depers	**0.10**	**0.23**	**0.00**	**1.00**	**0.01**	**1.00**
Accompl	**0.09**	**0.45**	**0.79**	**0.00**	**0.96**	**1.00**
WorkSat	**0.00**	**0.00**	**0.00**	**0.16**	**0.00**	**0.13**
WFC	**0.00**	**0.00**	**0.92**	**0.35**	**0.01**	**0.00**

The values with a probability value lower than `0.05` are significant by common standards. We can see, for instance, that, in this subsample, commitment is significantly correlated with exhaustion, work satisfaction, and work-family conflict, but not with depersonalization and accomplishment. We can also see that the predictors are intercorrelated—that is, they share part of their variance. We will examine whether this constitutes a problem for a regression analysis later.

Let's plot the relationship to see if the relationships indeed seem linear:

```
plot(nurses)
```



Scatterplots of attributes in the provided nurse's dataset

Here, we will only comment on the scatterplots in which commitment is included. We can see that there is visibly a negative linear association between commitment and exhaustion and work-family conflict. There is visibly a positive linear relationship between commitment and work satisfaction. Notice that there are also other relations visible on the plots, such as the visible relation between work-family conflict and exhaustion. From these scatterplots, nothing in the data seems problematic for the relationships we are exploring.

We will include some more regression diagnostics, but you are also encouraged to read about the assumptions of linear regression (for instance, at `http://www.basic.northwestern.edu/statguidefiles/mulreg_ass_viol.html`), and check whether the data fulfils these assumptions. If the assumptions of regression are violated, it is possible (actually probable) that the results are unreliable. Readers can also read about the detection of multivariate outliers here: `https://stat.ethz.ch/education/semesters/ss2012/ams/slides/v2.2.pdf`.

Sadly, there is not enough space to check all these here.

Performing the regression

We want to know if there is a relationship between our predictors and the criterion. We first want to know whether the three burnout dimensions predict commitment to the hospital.

We create the model by using the formula syntax as an argument in the `lm()` function. What is on the left of the tilde (`~`) sign is the criterion, on the right are the predictors, separated by a plus (`+`) sign:

```
model1 = lm(Commit ~ Exhaus + Depers + Accompl, data = nurses)
```

Let's examine the coefficients and their significance in the summary of the model:

```
summary(model1)
```

The following output shows that exhaustion and accomplishment are predictors of commitment to the hospital (look at p-value under Pr(>|t|) or refer to `*`)—exhaustion negatively (more emotionally exhausted people are less committed) and accomplishment positively (more accomplished people are more committed):

Call:

lm(formula = Commit ~ Exhaus + Depers + Accompl, data = nurses)

Residuals:

	Min	**1Q**	**Median**	**3Q**	**Max**
	-1.35915	**-0.3259**	**0.02808**	**0.35635**	**0.97905**

Coefficients:

	Estimate	**Std. Error**	**t value**	**Pr(>|r|)**	
(Intercept)	**4.331261**	**0.398985**	**10.856**	**6.62E-13**	*******
Exhaus	**-0.048725**	**0.008625**	**-5.649**	**2.05E-06**	*******
Depers	**-0.027053**	**0.019795**	**-1.367**	**0.18021**	

Accompl 0.032923 0.010392 3.168 0.00313 **

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

Residual standard error: 0.4892 on 36 degrees of freedom

Multiple R-squared: 0.55, Adjusted R-squared: 0.5125

F-statistic: 14.67 on 3 and 36 DF, p-value: 2.116e-06

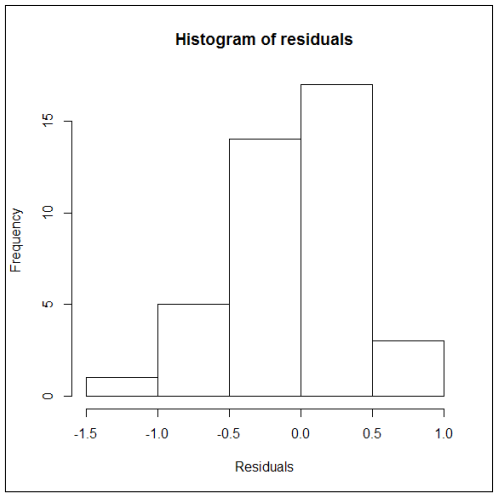
We can also see that p-value for F-statistic is significant (bottom of the output), and that 55 percent of variance (see Multiple R-squared) is predicted. The adjusted R-squared considers the number of predictors in the calculation of its value. It is recommended that you specify which value you use when reporting the results, or you can also report both values. Here, we can see that Adjusted R-squared is just a bit lower than Multiple R-squared, meaning that the results are not much affected by the number of predictors.

Checking for the normality of residuals

We have seen that it is important that residuals are normally distributed. We can do this visually by plotting, as in the following line of code:

```
hist(resid(model1), main="Histogram of residuals",
   xlab="Residuals")
```

From the preceding output, we might suspect a slight deviation from normality.



Histogram of the residuals

We can test this suspicion with the Shapiro-Wilk test, using the following code:

```
shapiro.test(resid(model1))
```

The output follows and shows that the residuals do not significantly depart from normality:

```
        Shapiro-Wilk normality test

data:  resid(model1)
W = 0.9776, p-value = 0.6001
```

Let's examine the Q-Q plot (type `plot(model1)` and click on the **RGraphics** window until the appropriate plot appears). This leads to a similar conclusion but, as in the `iris` preceding dataset example, some observations might prove problematic. We briefly discuss robust regression at the end of the chapter. Robust regression does not assume a normal distribution of residuals. For now, we proceed with the following regression diagnostic.

Checking for variance inflation

We also want to check whether there is a problem of variance inflation in our analysis—that is, whether the predictors are correlated a lot (multicollinear). For this purpose, we will rely on the `vif()` function of the `HH` package. The function takes the `lm` formula as an argument:

```
install.packages("HH"); library(HH)
vif(Commit ~ Exhaus + Depers + Accompl, data = nurses)
```

The output follows:

Exhaus	Depers	Accompl
1.054369	1.040318	1.015791

There are several rules-of-thumb to assess this. One is to consider `vif` values higher than 10 to be problematic, another is to consider a predictor as problematic if the square root of the `vif` value is higher than 2. This is not the case here, therefore, we consider our data to be non-multicollinear here.

Examining potential mediations and comparing models

Let's now examine whether including work-family conflict and work satisfaction permits to predict an additional part of variance. We first will ask R to fit a second model, and then will compare `model1` and `model2` using the `anova()` function:

```
model2 = lm(Commit ~ Exhaus + Depers + Accompl + WorkSat,
   data = nurses)
anova(model1,model2)
```

The following output shows that indeed the second model predicts additional variance in comparison to `model1` (see the significance of the F statistic for the comparison (under `Pr(>F)`):

Here is an analysis of the variance table:

Model 1: Commit~Exhaus+Depers+Accompl

Model 2: Commit~Exhaus+Depers+Accompl+WorkSat

	Res.Df	RSS	Df	Sum of Sq	F	Pr(>F)	
1	36	8.6161					
2	35	5.7181	1	2.898	17.738	0.0001685	***

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

We will now examine the second model, as the additional variance predicted is significantly different from 0:

```
summary(model2)
```

The output is provided here:

Call:
lm(formula = Commit ~ Exhaus + Depers + Accompl + WorkSat, data = nurses)

Residuals:

Min	1Q	Median	3Q	Max
-0.98119	-0.22736	-0.01279	0.26613	0.73625

Coefficients:

	Estimate	Std. Error	t value	Pr(>|t|)	
(Intercept)	1.969672	0.65044	3.028	0.004598	**
Exhaus	-0.029524	0.00846	-3.49	0.001326	**
Depers	0.014686	0.019123	0.768	0.447656	
Accompl	0.017392	0.009345	1.861	0.071142	.
WorkSat	0.46372	0.110103	4.212	0.000168	***

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

Residual standard error: 0.4042 on 35 degrees of freedom
Multiple R-squared: 0.7014, Adjusted R-squared: 0.6673
F-statistic: 20.55 on 4 and 35 DF, p-value: 8.662e-09

This model predicts 70 percent of variance in commitment, which is pretty good. We can see that work satisfaction is a significant predictor of commitment to the hospital, that the unique contribution of accomplishment is no longer significant (there is therefore a potential mediation), and that the contribution of exhaustion has been reduced when including work satisfaction in the model (there is therefore a potential partial mediation). This might be because of a mediation of the relationship between the two burnout components and commitment by job satisfaction. Let's test this relationship:

```
model3 = lm(WorkSat ~ Exhaus + Depers + Accompl, data = nurses)
summary(model3)
```

Let's examine the output:

Call:
lm(formula = WorkSat ~ Exhaus + Depers + Accompl, data = nurses)

Residuals:

Min	1Q	Median	3Q	Max
-1.57359		**-0.26967**	**-0.06299**	**0.24855**	**1.47504**

Coefficients:

	Estimate	Std.	Error	t	value
(Intercept)	**5.0927**	**0.49899**	**10.206**	**3.59E-12**	***
Exhaus	**-0.04141**	**0.01079**	**-3.839**	**0.000482**	***
Depers	**-0.09001**	**0.02476**	**-3.636**	**0.00086**	***
Accompl	**0.03349**	**0.013**	**2.577**	**0.014217**	*

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

Residual standard error: 0.6118 on 36 degrees of freedom
Multiple R-squared: 0.5162, Adjusted R-squared: 0.4758
F-statistic: 12.8 on 3 and 36 DF, p-value: 7.59e-06

We can notice that 51 percent of the variance of job satisfaction is predicted by the burnout components. All three burnout components are significantly related to work satisfaction ($p < .05$), negatively for emotional exhaustion and depersonalization and positively for personal accomplishment.

In order to ascertain mediation, we need to proceed to Sobel tests. The `bda` package contains the necessary function, called `mediation.test()`. Let's try to see whether the effect of exhaustion on commitment is mediated by work satisfaction:

```
install.packages("bda"); library(bda)
mediation.test(nurses$WorkSat,nurses$Exhaus,nurses$Commit)
```

In the following output, under Sobel, we can see that p.value is significant, as the presence of work satisfaction in the model decreases the effect of exhaustion, that work satisfaction is significant even though exhaustion is present in the model, and that, because the Sobel test is significant, we can confirm that there is indeed a partial mediation of the effect of exhaustion on commitment by work satisfaction. In other words, exhaustion decreases work satisfaction, and in turn, work satisfaction increases commitment.

	Sobel	**Aroian**	**Goodman**
z.value	**-2.9722704**	**-2.93647119**	**-3.00941168**
p.value	**0.00295606**	**0.0033197**	**0.00261754**

The value resulting from the Sobel test follows a `z` distribution. In order to obtain this value, the slope coefficients of the predictor regressed on the mediator (*a*) are multiplied by the slope coefficient of the mediator regressed on the criterion (*b*). This value is then divided by the square root of: *b* squared multiplied by the squared standard error of *a* plus *a* squared multiplied by the squared standard error of *b*. The formula is as follows:

$$z = \frac{a*b}{\sqrt{(b^2*s_a^2 + a^2*s_b^2)}}$$

Showing this is important, as very often, analysts include dozens or hundreds of predictors in their models without taking into consideration that the included predictors could themselves be related to each other. Readers are therefore advised to check for meaningful relationships between the attributes they intend to include as predictors in regression analyses before drawing conclusions on the final model!

Predicting new data

A particularly interesting use of regression is to examine how well a model predicts new data. This is easily achieved in R. We will first build the dataset named `nurses2` in the same way we did for the first dataset:

```
1  matcov2 = unlist(read.csv("matcov2.txt", header = F))
2  covs2 = matrix(matcov2, 6, 6)
3  means2 = c(4.279, 13.152, 5.156, 39.28, 5.153, 1.875)
```

```
4  set.seed(987)
5  nurses2 = data.frame(mvrnorm(n=40, means2, covs2))
6  colnames(nurses2)= c("Commit","Exhaus","Depers","Accompl",
7     "WorkSat","WFC")
```

To fit new data, we rely on the `predict.lm()` function:

```
predicted = predict.lm(model1, nurses2)
```

This results in the vector of predicted values being assigned to the criterion, which we call `predicted`.

As we have the real values for the commitment of individuals to the hospital in the second dataset as well, we can examine the correlations between those:

```
cor.test(predicted, nurses2$Commit)
```

The following output shows that the correlation between the predicted values and the real values is `0.5766194`. This value is significant and might seem pretty good at first sight:

```
        Pearson's product-moment correlation

data:  predicted and nurses2$Commit
t = 4.3506, df = 38, p-value = 9.848e-05
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.3231561 0.7528925
sample estimates:
      cor
0.5766194
```

Let's square this value to know how much of the variance in the commitment of the individuals of the second sample is predicted by the model:

```
0.5766194 ^2 *100
```

The output is `33.24899`. This means only 33 percent of the variance in commitment is predicted by the model, compared to 55 percent in the training data!

Now, we can also compute the residuals:

```
residuals_test = nurses2$Commit - predicted
```

We are now able to compute the `F` value for our model. We have seen that the `F` value is used to assess the overall significance of the model. In our case, the `F` value is obtained as follows:

1. First, we need to know the number of degrees of freedom for the model; this is equal to the number of predictors we have, which is 3. We also need the degrees of freedom for the error; this is the number of observations minus the degrees of freedom of the model, minus 1.
2. We then compute the sum of squares for the model as the sum of squared differences between the predicted values and the mean of the criterion. The sum of squares for the error is obtained as the sum of the squared differences between the observed and the predicted values.
3. We then compute the mean squares for the model as the sum of squares for the model divided by the degrees of freedom for the model. We compute the mean squares for the error as the sum of squares for the error divided by the degrees of freedom for the error.
4. Finally, we obtain the F-statistic by dividing the means squares for the model by the mean squares for the error.

The following function does just that:

```
1  ComputeF = function(predicted, observed, npred) {
2     DFModel = npred  # the number of predictors
3     DFError = length(observed) - DFModel -1
4     SSModel = sum((predicted - mean(observed))^2)
5     SSError = sum((observed - predicted)^2)
6     MSModel = SSModel / DFModel
7     MSError = SSError / DFError
8     F = MSModel / MSError
9     F
10  }
```

Let's try, first, with the original model to check whether the function works fine. The output is `14.67`, which is the same as the one outputted when we requested the summary of `model1`:

```
ComputeF(unlist(model1[5]), nurses$Commit, 3)
```

Now let's try to see if the new data is predicted well enough by the model:

```
ComputeF(predicted, nurses2$Commit, 3)
```

The outputted `F` value is `10.4842`.

We can test this value using the following line of code. The output shows that the threshold F value at a ceiling of 0.05 on the F distribution for our model is `2.866266`:

```
qf(.95, df1=3, df2=36)
```

We can therefore, trust that our model significantly predicts new data.

Robust regression

In the example datasets that we used in this chapter, we have seen that some observations might threaten the reliability of our results, because of the deviations of their residuals from a normal distribution. The Shapiro test performed on the residuals of `model1` (`nurses` dataset) has shown that the distribution of the residuals was not significantly different from a normal distribution. However, let's be particularly cautious and analyze the same data using robust regression.

As we mentioned earlier, robust regression does not require the residuals to be normally distributed, and therefore, fits our purpose. We will not explore the algorithm. For details about this, the reader can consult *Robust Regression in R* by Fox and Weisberg (2012). Here, we simply perform robust regression using the `rlm()` function of the `MASS` package. Let's first install and load it:

```
install.packages("MASS"); library(MASS)
model1.rr = rlm(Commit ~ Exhaus + Depers + Accompl, data = nurses)
summary(model1.rr)
```

The output is as follows:

Call: rlm(formula = Commit ~ Exhaus + Depers + Accompl, data = nurses)

Residuals:

Min	1Q	Median	3Q	Max
-1.4052046	-0.3233886	-0.0003426	0.3734567	1.0108386

Coefficients:

	Value	Std. Error	t value
(Intercept)	4.3602	0.3849	11.3271
Exhaus	-0.0518	0.0083	-6.2306
Depers	-0.0279	0.0191	-1.4602

Accompl	**0.0338**	**0.01**	**3.3676**

Residual standard error: 0.5536 on 36 degrees of freedom

You might notice that the output of `rlm()` is laconical in comparison to the output of `lm()`. There are no p-values provided, no R-squared values, no F test. This makes the use of `rlm()` quite unpractical, as the user will have to compute them by hand. There is so much controversy on how to do it that the computations in other software packages are currently questioned! The reader interested in computing the robust R-squared can read the paper *A robust coefficient of determination for regression* by Renaud and Victoria-Feser (2010).

For our example, it seems that the results using `lm()` and `rlm()` are pretty similar (see the output of the preceding summary of `model1`). Therefore, relying on `lm()` is advised here. However, if you want to be really sure, why not try bootstrapping.

Bootstrapping

The principle of (nonparametric) bootstrapping is to create a number of sample `K` of size `N` drawn with replacement from the original sample, where `N` is the original sample size. The parameters are estimated for each sample separately. This allows computing their confidence intervals, a measure of the variability of the parameters. Apart from making deviations from normal distributions less problematic, using bootstrapping is useful for samples that have a small number of observations (less than 100), as with ours.

We will discuss bootstrapping in *Chapter 14*, *Cross-validation and Bootstrapping Using Caret and Exporting Predictive Models Using PMML*, but let's have a sneak-peek now! Bootstrapping is easily performed using several functions in R—for instance, the `boot()` function in the `boot` package. But let's have a little fun and perform bootstrapping ourselves, 2,000 times. We will first generate the samples and obtain the estimates. We then display the estimates for the first six samples (rounded to the third decimal):

```
1  ret = data.frame(matrix(nrow=0, ncol=6))
2  set.seed(567)
3  for (i in 1:2000) {
4    data = nurses[sample(nrow(nurses), 40, replace = T),]
5    model_i <- lm(Commit ~ Exhaus + Depers + Accompl,
```

```
6      data = data)
7     ret = rbind(ret,c(coef(model_i),summary(model_i)$r.square,
8        summary(model_i)$fstatistic[1]))
9   }
10   names(ret) = c("Intercept","Exhaus","Depers",
11      "Accomp","R2","F")
12   round(head(ret), 3)
```

The following is the output. As you can see, the values of the parameters are all different. This is because, as we mentioned, they are based on different samples. Yet, they are not far apart from the others:

	Intercept	Exhaus	Depers	Accomp	R2	F
1	**4.080**	**-0.037**	**-0.055**	**0.041**	**0.585**	**16.928**
2	**4.196**	**-0.052**	**-0.048**	**0.040**	**0.694**	**27.273**
3	**5.054**	**-0.052**	**-0.047**	**0.022**	**0.736**	**33.416**
4	**4.103**	**-0.041**	**-0.042**	**0.037**	**0.545**	**14.373**
5	**4.663**	**-0.041**	**-0.022**	**0.022**	**0.454**	**9.980**
6	**4.525**	**-0.049**	**-0.035**	**0.029**	**0.497**	**11.874**

Using the same seed number as before, we can see which observations of the original data were selected for, say, the first sample:

```
set.seed(567)
sample(nrow(nurses), 40, replace = T)
```

In the following output, we can see that the first and 21st observations of the original dataset appear three times, the 3rd, 11th, and 12th, while the others appear twice. The 6th, 9th, 10th, and others appear once. And, finally, observations 4, 5, 7, and others do not appear in this sample:

```
[1] 30 36 26 20 11 10  3 21 24 22 14 11 15 24  1  3 21 30  2 12
[21]  6 21  9 16  1 34 37 34 16 39 22 29  2 38 29 38 37 28 12  1
```

As we mentioned, generating several samples in this fashion allows you to get a sense of the variability of the parameters, and confidence intervals are a good way to determine this variability. So let's compute the 95 percent confidence intervals based on the data we just generated. The formula to compute confidence intervals for the mean is:

$$\bar{x} \pm z * \frac{s}{\sqrt{n}}$$

Here `z` is the threshold value of the 97.5th percentile `(1 - (0.05/2))` in the `z` distribution. It is obtained with the following line of code:

```
qnorm(0.975)
```

So here we go:

```
CIs = data.frame(matrix(nrow = ncol(ret), ncol = 2))
for (j in 1:ncol(ret)) {
   M = mean(ret[,j])
   SD = sd(ret[,j])
   lowerb = M - (1.96* (SD / sqrt(2000)))
   upperb = M + (1.96* (SD / sqrt(2000)))
   CIs[j,1] = round(lowerb,3)
CIs[j,2] = round(upperb,3)
}
names(CIs) = c("95% C.I.lower bound", "95% C.I.upper bound")
 rownames(CIs) = colnames(ret)
 CIs
```

The resulting confidence intervals are provided here:

	95% C.I. lower bound	95% C.I. upper bound
Intercept	**4.297**	**4.325**
Exhaus	**-0.048**	**-0.048**
Depers	**-0.029**	**-0.027**
Accomp	**0.033**	**0.033**
R2	**0.558**	**0.570**
F	**18.179**	**19.139**

The confidence intervals encompass all the values between the lower and upper bounds. We can see that no confidence interval contains 0, meaning that, with a 95 percent threshold, values reported are statistically different from 0 (more correctly put, there is only a 5 percent chance of observing values inside these bounds if the true value of the parameters in the population is 0). So we conclude that bootstrapped coefficients are different from 0, as is the multiple R-squared value.

As you might have noticed, the value to which to compare the confidence intervals for `F` is not 0, but a value that depends upon the degrees of freedom. We computed this value earlier and it was `2.866266`. As the confidence interval for `F` does not include this value, we can be assured that the bootstrapped model predicts a significant part of variance.

Summary

In this chapter, we examined how to develop functions that perform simple regression analyses, and how to multiply regression in R using a real life example. We have examined the importance of significance tests for regression, and have briefly discussed robust regression and bootstrapping. Note that, when data about the predictors and the criterion are collected simultaneously, causation cannot be established. In order to ascertain causation, data must be collected longitudinally—that is, the predictors before the criterion.

In the next chapter, we will examine the classification of observations using the Naïve Bayes and k-Nearest Neighbor algorithms.

10
Classification with k-Nearest Neighbors and Naïve Bayes

In *Chapter 8*, *Probability Distributions, Covariance, and Correlation*, we examined statistical distributions, covariance, and correlation. In the previous chapter, you learned about regression. Here, we will focus on classification using Naïve Bayes and **k-Nearest Neighbors** (**k-NN**). The problem we want to solve, when using both algorithms, is as follows:

- We have data in which class (the attribute we want to predict) values are known. We call this training data.
- We have data in which class values are not known (or we pretend we don't know to test that our classifier works, in which case we call this testing data).
- We want to predict unknown class values using information from data where the class is known.

For instance, imagine we have collected data about the health habits of individuals. For half of these individuals, we know whether or not they have developed a disease, say, in the following year. For the other half of our sample, we don't know if they have developed this disease. We will seek to gain knowledge about the unknown values by using the observed relationships in the data where we know the disease outcome (the class).

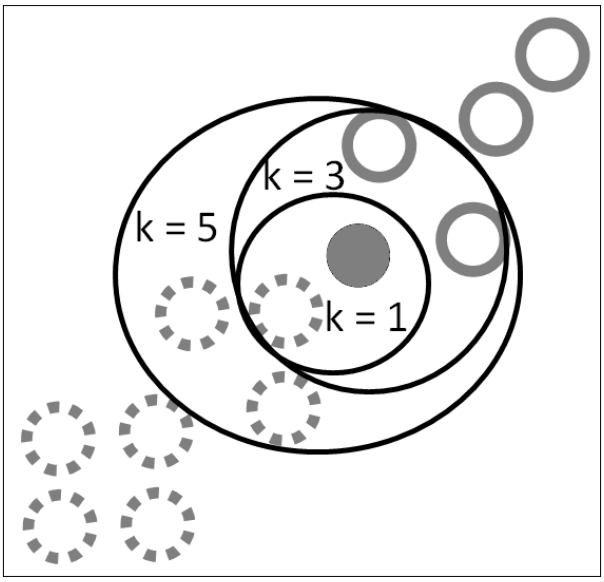
In this chapter, we will discover two new algorithms for data classification:

- Naïve Bayes
- k-Nearest Neighbors

Understanding k-NN

Remember that, in *Chapter 4*, *Cluster Analysis*, we discovered that distance matrices are used by k-means to cluster data into a user-specified number of groups of homogenous cases. k-NN uses distances to select the user-defined number of observations that are closest (neighbors) to each of the observations to classify. In k-NN, any attribute can be categorical or numeric, including the target. As we discuss categorization in this chapter, I will limit the description to categorical target (called class attributes).

The classification of a given observation will be made as a majority vote in the neighbors—that is, the most frequent class among the k closest observations. This means that the classification of observations will depend on the chosen number of neighbors. Let's have a look at this. The following figure represents the membership of gray-outlined circles to two class values: the plain grey-lined and the dotted grey-lined. Notice there is one filled grey circle as well, of which we don't know the class membership. The black circles or ellipses contain the closest neighbors on two imaginary attributes, with k = 1, k = 3, and k = 5 being the closest neighbors:



Nearest neighbors with different k values

One can see that, with k = 1, the membership of the filled circle would be dotted grey-lined, as the neighbor has this membership, but, with k = 3, the filled circles would be classified as filled grey-lined as the majority of the neighbors (two out of three) have this membership. With k = 5, the filled circle would be classified as dotted grey-lined, because three out of five of the neighbors have this membership.

The neighbors are determined according to a distance measure. We discussed distances in *Chapter 4*, *Cluster Analysis*, so I'll just provide a quick reminder here.

Distances can be understood as a measure of how much observations differ from one another, overall, on all the considered attributes. There are several types of distances. For instance, the Euclidean distance is the square root of the sum of the squared differences on each attribute between two observations. Its formula is provided here:

$$\sqrt{(\sum_{i=1}^{n} p_i - q_i)^2}$$

The Manhattan (or taxi cab distance) is the sum of the absolute differences on each attribute between two observations. Its formula is:

$$\sum_{i=1}^{n} |p_i - q_i|$$

We will use the most common distance—Euclidean distance—in this chapter.

Before going in depth into the analyses of data with R, let's classify some data by hand and examine whether we did it right by relying on the `knn()` function.

For this exploration, we will rely on the `iris` dataset one more time.

We will start by computing the Euclidean distances between each observation. We want the full distance matrix so that we can determine the k-NN more easily (but not optimally):

```
1  distances = dist(iris[1:4], upper = T, diag = T)
```

We now need to create a data frame from the distances. This is only done in our manual implementation of k-Nearest Neighbors. For this purpose, we will install and load the `reshape2` package. Using the `melt()` function, we will obtain an attribute containing all the distances, instead of the matrix of distances we created previously:

```
2  install.packages("reshape2"); library(reshape2)
3  distances.df  <- melt(as.matrix(distances))
```

Now that we have our data frame, we will select the k closest observation to each observation and assign a value to each unclassified observation based on the values of the majority of its neighbors:

```
k = 5
N = length(levels(as.factor(distances.df$Var2)))
Nearest = matrix(nrow= N, ncol = k)
level_count = length(levels(as.factor(iris[[5]])))
classif = rep(0,N)
for (i in 1:N) {
    temp = subset(distances.df, Var2 == i)
    nearest =
    unlist(head(temp[order(temp$Var2,temp$value),],k)[1])
    votes = iris[0,5]
    for (j in 1:length(nearest)) {
         votes[j] = iris[5][nearest[j],]
         classif[i]= which.max(table(votes))
     }
    }
iris$Species_class[classif == 1] = "setosa"
iris$Species_class[classif == 2] = "versicolor"
iris$Species_class[classif == 3] = "virginica"
table(iris$Species,iris$Species_class)
```

The following output shows some differences in our classification and the correct classes of the data, but overall, our version of k-NN worked pretty fine:

	setosa	**versicolor**	**virginica**
setosa	**50**	**0**	**0**
versicolor	**0**	**47**	**3**
virginica	**0**	**2**	**48**

The following line of code allows us to identify which observations were classified incorrectly:

```
rownames(iris)[iris$Species != iris$Species_class]
```

The output shows that this was the case for observations in rows 71, 73, 84, 107, and 120.

Does our version of k-NN work as well as the version implemented in R in the `knn()` function? Let's find out! We first install and load the `class` package, which contains the `knn()` function:

```
install.packages("class"); library(class)
```

As we did with our version of `knn`, we will use the same data as training and testing data (this is easier but not optimal as we will see later):

```
iris$knn_class = knn(iris[1:4],iris[1:4],iris[[5]], 5)
```

There are 150 observations in the `iris` dataset. Let's see how many are classified in the same class by our implementation of k-NN and the `knn()` function:

```
sum(iris$knn_class == iris$Species_class)
```

The output is `150`, which means that both implementations performed as well with this data. Nevertheless, as always, the code presented here is meant for understand the algorithm only, and will only work if the training and testing data are the same. It therefore has only pedagogical applications. Be sure to always use functions available on CRAN for real data analysis.

Working with k-NN in R

When explaining the way k-NN works, we have used the same data as training and testing data. The risk here is overfitting: there is noise in the data almost always (for instance due to measurement errors) and testing on the same dataset does not let us examine the impact of noise on our classification. In other words, we want to ensure that our classification reflects real associations in the data.

There are several ways to solve this issue. The most simple is to use a different set of data for training and testing. We have already seen this when discussing Naïve Bayes. Another, better, solution is to use cross-validation. In cross-validation, the data is split in any number of parts lower than the number of observations. One part is then left out for testing and the rest is used for training. Training is then performed again, leaving another part of the data out for testing, but including the part that was previously used for testing. We will discuss cross-validation in more detail in *Chapter 14*, *Cross-validation and Bootstrapping Using Caret and Exporting Predictive Models Using PMML*.

Here, we will use a special case of cross-validation: leave-one-out cross-validation, as this is readily implemented in the function `knn.cv()`, which is also included in the `class` package. In leave-one-out cross-validation, each observation is iteratively left out for testing, and all other observations are used for training.

We will perform leave-one-out cross-validation using the `Ozone` dataset, which contains air quality data. We will now install and load the `mlbench` package, as it contains the data. Missing data will be omitted here:

```
install.packages("mlbench"); library(mlbench)
data(Ozone)
Oz = na.omit(Ozone)
```

The dataset originally contains 366 observations (203 after omitting observations containing a missing gate) and 13 attributes. Type `?Ozone` after installing the package for a description of the attributes. The first attribute is the month where the observation was collected. For the purpose of the analysis, we will recode the month in a new attribute—data collected between April and September will be coded 1, and the others 0:

```
Oz$AprilToSeptember = rep(0,length(Oz[,1]))
Oz$AprilToSeptember[as.numeric(Oz[[1]])>=4 &
   as.numeric(Oz[[1]])<=9] = 1
```

There are 92 observations that were collected between April and September. The task will be to classify the observations on this attribute. In order to illustrate the importance of using cross-validation, we will do this first using the whole dataset for training and testing. Later, we will rely on cross validation. Notice that the first argument of `knn()` is the training dataset, the second is the testing dataset, the third is the class attribute, and the fourth is the number of neighbors. It is recommended to set `k` to odd values in order to avoid ties. We will use three neighbors here. We will see how to select `k` later:

```
Oz$classif = knn(Oz[2:13],Oz[2:13],Oz[[14]], 3)
```

We then examine the confusion matrix:

```
table(Oz$classif,Oz[[14]])
```

The following confusion matrix shows that the majority of the observations were classified correctly, but about 15 percent were misclassified, (12+19) / 203 = 0.1527094:

	0	**1**
0	**92**	**12**
1	**19**	**80**

Remember that, in this case, all the observations are part of the training dataset. What happens if we classify observations that are not used for training ? Let's find out:

```
Oz$classif2 = knn.cv(Oz[2:13],Oz[[14]], 3)
```

Note that `knn.cv()` takes the dataset as the first argument, the class as the second argument, and k as the third argument.

Let's examine the confusion matrix:

```
table(Oz$classif2,Oz[[14]])
```

The following confusion matrix shows that, again, most observations were correctly classified, but the proportion of incorrectly classified observations is higher (27+28)/203 = 0.270936:

	0	**1**
0	**83**	**27**
1	**28**	**65**

The proportion of incorrectly classified observations is almost twice as high as before. This matters because, when attempting to predict data that is really unknown, we could have unrealistic expectations about the performance of k-NN, if relying on the data used for training for testing. We will discuss performance measures at the end of the chapter.

How to select k

Duda, Hart, and Stork (in their book, *Pattern Classification*, 2000) propose to select `k` as the square root of the number of observations. While such a rule-of-thumb has the merit of simplicity, it does not always lead to better classifications. In the case of our example, the ratio of misclassified observations using leave-one-out cross-validation rises to 35 percent when setting `k` to the square root of the number of observations.

Another way to select `k` is to choose the number of neighbors that maximizes some performance measure. This implies running the analysis several times with different `k` values until such maxima is found. We can also set a reasonable limit of neighbors, an arbitrary 10 percent of the dataset. So let's do this with the data in our example. We will be maximizing accuracy (the opposite of the misclassification ratio), but another measure (for instance precision or recall—see the end of the chapter) can be used:

```
1  Accur = rep(0,20)
2  for (i in 1:20) {
3     classification = knn.cv(Oz[2:13],Oz[[14]], i)
4     Accur[i] = sum(classification == Oz[[14]])/203
5     }
```

So let's examine the best number of neighbors for our data:

```
6  which.max(Accur)
```

The output is `3`, so we selected the best number initially!

Now that we have a working knowledge of classification using k-NN, let's see how to do it with Naïve Bayes.

Understanding Naïve Bayes

Naïve Bayes uses conditional probabilities in order to classify the observations. In this section, you will learn how it works. We will invent a simple dataset, and a disease, for this purpose. Let's have a look at the table. The table shows health behaviors of 11 individuals and whether or not 10 of them have developed `DiseaseZ` (the name of our made up disease) one year after these behaviors have been assessed. What we want to know is whether the individual is at risk of developing the disease. We will solve this using existing data about the individual and associations previously found in other individuals:

Smoking	Drinking	PhysicalActivity	Movies	Music	Sunbathing	DiseaseZ
YES	YES	NO	NO	NO	YES	YES
YES	NO	YES	NO	YES	YES	NO
NO	YES	NO	NO	YES	NO	YES
NO	NO	YES	NO	NO	YES	YES
YES	YES	NO	NO	NO	NO	YES
NO	NO	YES	YES	NO	NO	NO
NO	YES	YES	YES	NO	NO	NO
YES	YES	YES	YES	YES	YES	YES
NO	NO	NO	NO	NO	NO	NO
YES	YES	YES	YES	YES	YES	YES
NO	YES	NO	YES	NO	YES	

What we can tell from the table is that the probability of developing `DiseaseZ` is 6/10 = 0.6. We will call it the prior probability—if we don't know anything about an individual, we can tell he/she has a 60 percent chance of developing `DiseaseZ`. The posterior probability is what we want to know—that is, the probability that an individual will develop the disease knowing their health behavior. This requires computing the conditional probabilities for each of the health behaviors—that is, what is the probability that a behavior is performed by someone who has developed the disease, and someone who hasn't.

The reader can load the dataset as follows (make sure the file `diseaseZ.txt` is in your working directory):

```
1  DiseaseZ = read.table("DiseaseZ.txt", header = T, sep="\t")
```

We first create two datasets, one with individuals with `DiseaseZ` and the other with individuals without `DiseaseZ`, and compute the number of cases in each:

```
2  Sick = subset(DiseaseZ, DiseaseZ=="YES")
3  NotSick = subset(DiseaseZ, DiseaseZ=="NO")
4  dim(Sick)[1]
5  dim(NotSick)[1]
```

The output indicates that there are six individuals in the `Sick` data frame, and four in the `NotSick` data frame, which is what we computed before. We can now obtain the conditional probabilities using the following code:

```
6  prob.Sick = colSums(Sick[,1:6]== "YES")/6
7  prob.NotSick  = colSums(NotSick[,1:6]== "NO")/4
```

The probabilities of having performed the behaviors for individuals who were and were not sick (rounded to the second decimal) are displayed in the following table. As a reminder, behaviors performed by the individual to classify are indicated by an `X` in the last row of the table:

	Smoking	Drinking	PhysicalActivity	Movies	Music	Sunbathing
DiseaseZ == 1	**0.67**	**0.83**	**0.5**	**0.33**	**0.5**	**0.67**
DiseaseZ == 0	**0.25**	**0.25**	**0.75**	**0.50**	**0.25**	**0.25**
		X		**X**		**X**

As can be noticed from the conditional probabilities alone, there are differences in the performance of the behaviors between people who have and have not developed the disease. Proportionally, more people who have been smoking, drinking, listening to music, and have been sunbathing have developed the disease, compared to those who haven't. Performing physical activities, going to see movies and listening to music are related to not having the disease in this fictitious example.

But is there an association between the behaviors? I mean, do people who smoke also drink? This is a possibility, but Naïve Bayes assumes that all the attributes that are used to predict the classes are independent of each other. This is clearly not the case in the real world. But, it turns out, Naïve Bayes classifies the observations quite reliably even though it is based on this unrealistic assumption. Naïve Bayes uses the conditional joint probabilities to determine the class of the observations for which the class is unknown. So let's try computing those ourselves. In order to do this, we also need the probabilities of not having performed the behaviors given the class. They are equal to 1 minus the probability of performing the behaviors.

The probabilities of not having performed the behaviors are displayed here:

	Smoking	**Drinking**	**PhysicalActivity**	**Movies**	**Music**	**Sunbathing**
DiseaseZ == 1	**0.33**	**0.17**	**0.5**	**0.67**	**0.5**	**0.33**
DiseaseZ == 0	**0.75**	**0.75**	**0.25**	**0.5**	**0.75**	**0.75**

We now can compute the probability that our unclassified individual has or has not developed `DiseaseZ` by computing the joint probability of each behavior given both outcomes, multiplied by the prior probability. Let's recall that the individual has not been smoking, has been drinking, didn't take physical activities, has been to the movies, has not listened to music, and has been sunbathing.

For the outcome `DiseaseZ == 1`, the joint probability is:

```
(.33 * .83 * .5 * .33 * .5 * .67) * .6 = 0.009083894
```

For the outcome `DiseaseZ == 0`, the joint probability is:

```
(.75 * .25 * .25 * .5 * .75 * .25) * .4 =  0.001757813
```

As this value is higher for `DiseaseZ == 1` as compared to `DiseaseZ ==0`, we will conclude that the individual to classify is at risk of developing the disease.

Now let's have a look at what an R implementation of Naïve Bayes finds out! We start by installing and loading the `e1071` package that contains the `naiveBayes()` function:

```
1  install.packages("e1071")
2  library(e1071)
```

We then train a classifier (we will use observations 1 to 10 for training), and inspect its content:

```
3  Classify = naiveBayes(DiseaseZ[1:10,1:6],
4     DiseaseZ[1:10,7])
5  Classify
```

Let's examine the values as shown in the following output. We can see that the prior probabilities are the same as we computed before: 0.4 for not having `DiseaseZ`, and 0.6 for having `DiseaseZ`, so we computed this right. Good! Now let's examine the conditional probabilities. We are not going to comment on all. We'll just have a look at those under smoking. We reported a conditional probability of smoking of 0.67 among individuals who developed `DiseaseZ`. The classifier reports the same thing (but rounds after seven decimals, whereas we did this after two). We found a conditional probability of not smoking of 0.33 (that is 1 minus 0.67, as probabilities must sum to 1) among those individuals. That's what the classifier found as well. For individuals who didn't develop the disease, we computed a probability of 0.25 of being a smoker, and a probability of 0.75 of not being a smoker. I will let you examine the output to see what values are reported by the classifier:

```
Naive Bayes Classifier for Discrete Predictors

Call:
naiveBayes.default(x = as.matrix(DiseaseZ[1:10, 1:6]), y = as.matrix(DiseaseZ[1:10,
    7]))

A-priori probabilities:
as.matrix(DiseaseZ[1:10, 7])
 NO YES
0.4 0.6

Conditional probabilities:
                           Smoking
as.matrix(DiseaseZ[1:10, 7])        NO       YES
                         NO  0.7500000 0.2500000
                         YES 0.3333333 0.6666667

                           Drinking
as.matrix(DiseaseZ[1:10, 7])        NO       YES
                         NO  0.7500000 0.2500000
                         YES 0.1666667 0.8333333

                           PhysicalActivity
as.matrix(DiseaseZ[1:10, 7])   NO  YES
                         NO  0.25 0.75
                         YES 0.50 0.50

                           Movies
as.matrix(DiseaseZ[1:10, 7])        NO       YES
                         NO  0.5000000 0.5000000
                         YES 0.6666667 0.3333333

                           Music
as.matrix(DiseaseZ[1:10, 7])   NO  YES
                         NO  0.75 0.25
                         YES 0.50 0.50

                           Sunbathing
as.matrix(DiseaseZ[1:10, 7])        NO       YES
                         NO  0.7500000 0.2500000
                         YES 0.3333333 0.6666667
```

The classifier for our first example

We determined that the individual to classify was at risk of developing `DiseaseZ`, based on her or his behaviors and the associated probabilities. Let's now see what Naïve Bayes estimates:

```
predict(Classify, DiseaseZ[11,1:6])
```

As indicated in the output, the `predict()` function, given the `naiveBayes()` classifier and the behavior of the individual to classify as arguments, gives the same answer to this classification problem as we did:

```
[1] YES
Levels:  NO YES
```

It is worth mentioning that Naïve Bayes is not limited to categorical predictors, and works in a similar way with continuous ones, using density estimations instead of conditional probabilities.

Now that we have demonstrated how Naïve Bayes works, we are going to examine its use in detail.

Working with Naïve Bayes in R

For this example of working with Naïve Bayes in R, we are going to use the `Titanic` dataset. The classification problem we have is to know whether or not individuals died in the Titanic accident. We will create a training dataset and a testing dataset (in order to test how well the classifier performs).

The first thing we need to know is how to convert the Titanic dataset (of class `table`) to a data frame:

```
1  Titanic.df_weighted = data.frame(Titanic)
```

Let's have a look at the dataset:

	Class	Sex	Age	Survived	Freq
1	**1st**	**Male**	**Child**	**No**	**0**
2	**2nd**	**Male**	**Child**	**No**	**0**
3	**3rd**	**Male**	**Child**	**No**	**35**
4	**Crew**	**Male**	**Child**	**No**	**0**
5	**1st**	**Female**	**Child**	**No**	**0**
6	**2nd**	**Female**	**Child**	**No**	**0**
7	**3rd**	**Female**	**Child**	**No**	**17**

	Class	Sex	Age	Survived	Freq
8	Crew	Female	Child	No	0
9	1st	Male	Adult	No	118
10	2nd	Male	Adult	No	154
11	3rd	Male	Adult	No	387
12	Crew	Male	Adult	No	670
13	1st	Female	Adult	No	4
14	2nd	Female	Adult	No	13
15	3rd	Female	Adult	No	89
16	Crew	Female	Adult	No	3
17	1st	Male	Child	Yes	5
18	2nd	Male	Child	Yes	11
19	3rd	Male	Child	Yes	13
20	Crew	Male	Child	Yes	0
21	1st	Female	Child	Yes	1
22	2nd	Female	Child	Yes	13
23	3rd	Female	Child	Yes	14
24	Crew	Female	Child	Yes	0
25	1st	Male	Adult	Yes	57
26	2nd	Male	Adult	Yes	14
27	3rd	Male	Adult	Yes	75
28	Crew	Male	Adult	Yes	192
29	1st	Female	Adult	Yes	140
30	2nd	Female	Adult	Yes	80
31	3rd	Female	Adult	Yes	76
32	Crew	Female	Adult	Yes	20

As can be seen, there are five attributes: the class of the individuals (first class, second class, third class, or crew), their sex and age, whether they survived, and the frequency of cases in each cell. In order to create our training and testing datasets, we first need to reconstruct the full dataset—that is, without the frequency weightings. We'll write some ad hoc code to do just this:

```
Titanic.df_weighted = data.frame(Titanic)
# creating empty data frame to be populated
Titanic.df = Titanic.df_weighted[0,1:4]

# populating the data frame
k=0
for (i in 1:nrow(Titanic.df_weighted)){
   if (Titanic.df_weighted[i,5]>0) {
      n = Titanic.df_weighted[i,5]
       for (j in 1:n) {
          k = k + 1
          Titanic.df [k,] =
             unlist(Titanic.df_weighted[i,1:4])
       }
    }
}
```

Let's check whether we obtained the right output. We'll create a table again, and compare it to the `Titanic` dataset:

```
table(Titanic.df) == Titanic
```

We omit the output here. Suffice to say that the entries in both tables (the one we just built, and the `Titanic` dataset) are identical. We now know we computed the data correctly.

What we want to do now is create the two datasets that we need—the training dataset and the test dataset. We will go for a simple way of doing this instead of using stratified sampling. Random sampling is enough for our demonstration purposes here. We will set the seed to 1 so that the samples are identical on your screen and in this book:

```
set.seed(1)
Titanic.df$Filter= sample(c("TRAIN","TEST"),
   nrow(Titanic.df), replace = T)
TRAIN = subset (Titanic.df, Filter == "TRAIN")
TEST = subset (Titanic.df, Filter == "TEST")
```

Now let's build a classifier based on the data in the `TRAIN` dataset, and have a look at the prior and conditional probabilities:

```
6  Classify = naiveBayes(TRAIN[1:3],TRAIN[[4]])
7  Classify
```

The output, in the following screenshot shows that the conditional probability (hereafter, simply probability) of dying, for individuals in the training dataset, is `0.687395`, whereas the probability of surviving is `0.3126095`. Looking at the conditional probabilities, we can see that people in first class have a higher probability of surviving; in second class, the relative probability of surviving is reduced; and third class passengers and crew members had high probabilities of dying rather than that of surviving. Looking at the attribute `Sex`, we can see that males had a higher probability of dying, which is the opposite of what is observed for females. Children had a higher probability of surviving, whereas for adults, these probabilities were quite similar.

```
Naive Bayes Classifier for Discrete Predictors

Call:
naiveBayes.default(x = TRAIN[1:3], y = TRAIN[[4]])

A-priori probabilities:
TRAIN[[4]]
       No       Yes
0.6873905 0.3126095

Conditional probabilities:
          Class
TRAIN[[4]]        1st        2nd        3rd       Crew
       No  0.08535032 0.11719745 0.33503185 0.46242038
       Yes 0.30812325 0.15406162 0.23529412 0.30252101

          Sex
TRAIN[[4]]       Male     Female
       No  0.91592357 0.08407643
       Yes 0.50140056 0.49859944

          Age
TRAIN[[4]]      Child      Adult
       No  0.03057325 0.96942675
       Yes 0.07002801 0.92997199
```

The classifier for our second example

We will now examine how well the classifier will be able to use this information to classify the other individuals—that is, those of the testing dataset:

```
TEST$Classified = predict(Classify,TEST[1:3])
table(TEST$Survived,TEST$Classified)
```

As shown in the diagonals of the outputted confusion table presented here, the classifier correctly predicted nonsurvival (true negatives) in `645` cases, and survival (true positive) in `168` cases. In addition, the classifier incorrectly predicted nonsurvival in `186` cases (false negative), and survival (false positive) in `60` cases:

	No	**Yes**
No	**645**	**60**
Yes	**186**	**168**

Computing the performance of classification

There are several measures of performance that we can compute from the preceding table:

- The true positive rate (or sensitivity) is computed as the number of true positives divided by the number of true positives plus the number of false negatives. In our example, sensitivity is the probability that a survivor is classified as such. Sensitivity in this case is:

  168 / (168 + 186) = 0.4745763

- The true negative rate (or specificity) is computed as the number of true negatives divided by the number of true negatives plus the number of false positives. In our example, specificity is the probability that a nonsurvivor is classified as such. Specificity in this case is:

  645 / (645 + 60) = 0.9148936

- The positive predictive value (or precision) is computed as the number of true positives divided by the number of true positives plus the number of false positives. In our example, precision is the probability that individuals classified as survivors are actually survivors. Precision in this case is:

  168 / (168 + 60) = 0.7368421

- The negative predictive value is computed as the number of true negatives divided by the number of true negative plus the number of false negatives. In our example, the negative predictive value is the probability that individuals classified as nonsurvivors are actually nonsurvivors. In this case, its value is:

  645 / (645 + 186) = 0.7761733

- The accuracy is computed as the number of correctly classified instances (true positives plus true negatives) divided by the number of cases correctly classified plus the number of incorrectly classified instances. In our example, accuracy is the number of survivors and nonsurvivors correctly classified as such. In this case, accuracy is:

  (645 + 168) / ((645 + 168) + (60 + 186)) = 0.7677054

- Cohen's kappa is a measure that can be used to assess the performance of classification. Its advantage is that it adjusts for correct classification happening by chance. Its drawback is that it is slightly more complex to compute. Considering two possible class values, let's call them No and Yes. The kappa coefficient is computed as the accuracy minus the probability of correct classification by chance, divided by 1 minus the probability of correct classification by chance. The probability of correct classification by chance is the probability of correct classification of "Yes" by chance plus the probability of correct classification of "No" by chance. The probability of the correct classification of "Yes" by chance can be computed as the probability of observed "Yes" multiplied by the probability of classified "Yes". Likewise, the probability of correct classification of "No" by chance can be computed as the probability of observed "No" multiplied by the probability of classified "No". Let's compute the kappa for the preceding example.
- In the preceding example, the probability of the observed "No" is (645 + 60) / (647+60+186+168) = 0.6644675. The probability of the classified "No" is: (645 + 186) / (645 + 186 + 60 + 168) = 0.7847025. The probability of correct classification of "No" by chance is therefore: 0.6644675 * 0.7847025 = 0.5214093. The probability of observed "Yes" is: (186 + 168) / (186+168+645+60) = 0.3342776. The probability of classified "Yes" is: (60+186) / (60+186+645+186) = 0.2284123. The probability of correct classification of "Yes" by chance is therefore: 0.3342776 * 0.2284123 = 0.07635312. We can now compute the probability of correct classification by chance as: 0.5214093 + 0.07635312 = 0.5977624. We have seen that accuracy is 0.7677054. We can compute the kappa as follows:

  (0.7677054 – 0.5977624) / (1 - 0.5977624) = 0.4224941.

Kappa values can range from `-1` and `1`. Values below zero are meaningless. A kappa of zero means that the classification is not better than what can be obtained by chance. A kappa of one means a perfect classification. Usually values below `.60` are considered bad, and values above `.80` are preferred. In the case of our example (kappa below `.60`), we would not trust the classification, and refrain from using the classifier any further.

Summary

In this chapter, we have seen how k-NN and Naïve Bayes work by programming our own implementation of the algorithms. You have discovered how to perform these analyses in R. We have shown you that it is not optimal to test our classifier with the data it has been trained with. We have seen that the number of neighbors selected in k-NN impacts the performance of the classification and examined different performance measures. In the next chapter, you will learn about decision trees.

11
Classification Trees

In *Chapter 9, Linear Regression* we discussed regression. In the previous chapter, we were interested in classification using k-NN and Naïve Bayes. In this chapter, we will continue the topic of classification and discuss it in the context of decision trees. Decision trees notably allow class predictions (group membership) of previously unseen observations (testing datasets or prediction datasets) using statistical criteria applied on the seen data (training set).

Here, we will briefly examine the statistical criteria of six algorithms:

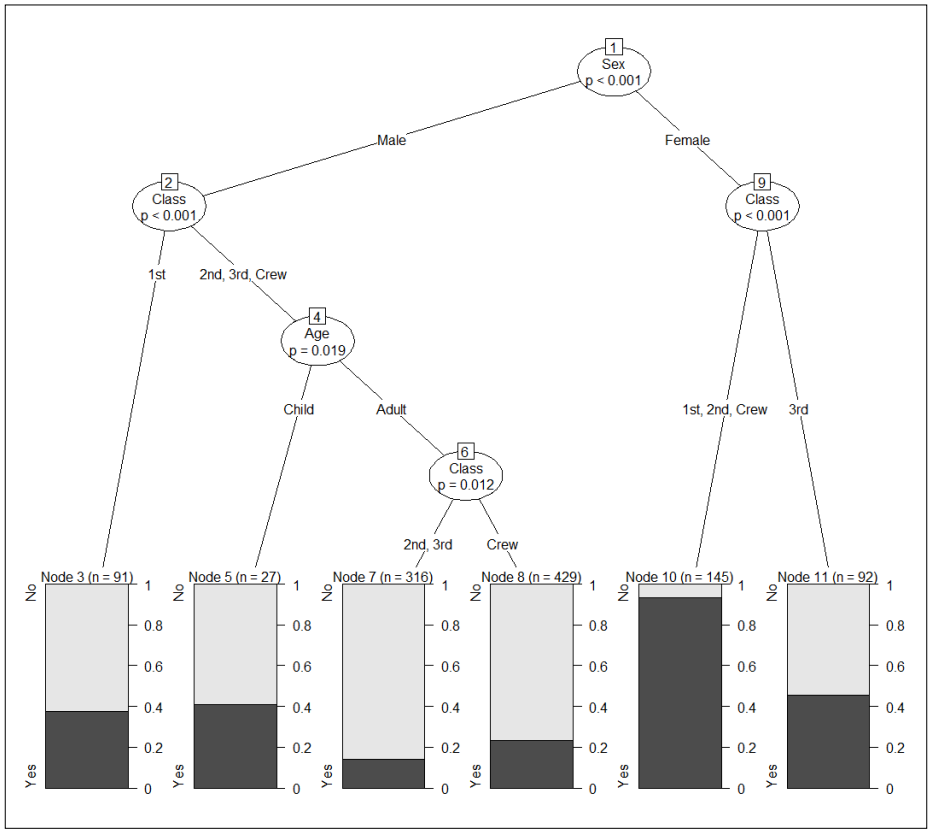
- ID3
- C4.5
- C5.0
- Classification and regression trees (CART)
- Random forest
- Conditional inference trees

We will also examine how to use decision trees in R, notably, how to measure the reliability of the classifications using training and test sets.

Understanding decision trees

Before we go in depth into how decision tree algorithms work, let's examine their outcome in more detail. The goal of decision trees is to extract from the training data the succession of decisions about the attributes that explain the best class, that is, group membership.

In the following example of the conditional inference tree, we try to predict survival (there are two classes: `Yes` and `No`) in the `Titanic` dataset we used in the previous chapter. Now to simplify things, there is an attribute called `Class` in the dataset. When discussing the outcome we want to predict (the survival of the passenger), we will use a lowercase `c` (class), and when discussing the `Class` attribute (with 1st, 2nd, 3rd, and Crew), we will use a capital `C`. The code to generate the following plot is provided at the end of the chapter, when we describe conditional inference trees:



Example of decision tree (conditional inference tree)

Decision trees have a root (here: **Sex**), which is the best attribute to split the data upon, in reference to the outcome (here: whether the individuals survived or not). The dataset is partitioned into branches on the basis of this attribute. The branches lead to other nodes that correspond to the next best partition for the considered branch. We can see that the attribute called **Class** is the best for both **Male** and **Female** branches. The process continues until we reach the terminal nodes, where no more partitioning is required. The proportion of individuals that survived and didn't is indicated at the bottom of the plot.

Here, we have used only categorical attributes, but numeric attributes can also be used in the prediction of a categorical outcome in C45, CART, and conditional inference trees.

We will now examine different algorithms for the generation of the decision about the attributes upon which to partition the data. We will start with an easy algorithm and continue with more complex ones.

ID3

ID3 is one of the simplest algorithms to produce decision trees with categorical classes and attributes. We chose to explain it because of its simplicity (but will not examine its use here). We will then build upon this understanding when discussing the other algorithms.

ID3 relies on a measure called information gain to build the trees. The goal is to maximize the predictive power of the tree by reducing the uncertainty in the data.

Entropy

Entropy is a measure of uncertainty in a source of information. We discuss it before we talk about information gain, as information gain relies on the computation of entropy.

Entropy is easily understood using an example. Let's consider three opaque boxes containing 100 M&Ms each. In box 1, there are 99 red M&Ms and 1 yellow. In box 2, there are as many red and yellow M&Ms. In box 3, there are 25 red M&Ms and 75 yellow. Knowing this, we want to guess the color of the next M&M we pick from each of the boxes.

As you have guessed, it is easier to guess the color of the picked M&M in the first box, because there is only one yellow M&M in the box. In this case, there is almost no uncertainty. Contrarily, there is maximal uncertainty in the case of the second box, because there is an equal probability that the M&M we pick from the box is red or yellow. Finally, in the third box there is a relative uncertainty, because picking a yellow M&M is three times as likely as picking a red one.

Entropy is a measure of what we could intuitively grasp from previous knowledge of the content of the boxes. In the following formula for the computation of entropy (where *c* is the number of classes), *p* refers to the probability of each of the outcomes:

$$-\sum_{i=1}^{c} p_i(log_2\ p_i)$$

Let's examine the entropy for the M&Ms problem.

Here, entropy is measured as minus the probability of `red` multiplied by the base two logarithm of the probability of `red`, minus the probability of `yellow` multiplied by the base two logarithms of the probability of `yellow`:

```
- p_red * log2(p_red) -  p_yellow * log2(p_yellow)
```

In box 1, the entropy is:

```
- (99/100) * log2(99/100) - (1/100) * log2(1/100) #= 0.08079314
```

In box 2, the entropy is:

```
- (50/100) * log2(50/100) - (50/100) * log2(50/100) # = 1
```

In box 3, the entropy is:

```
- (25/100) * log2(25/100) - (75/100) * log2(75/100) #= 0.8112781
```

Computing the entropy of attributes containing more categories can be done in a similar way: the sum of all `- p_category * log2(p_category)`, where `p_category` is the probability of each of the different categories iteratively considered. For instance, let's imagine a yellow M&M is replaced by a blue M&M in the third box. The entropy in that box is now:

```
- (25/100) * log2(25/100) - (74/100) * log2(74/100) - (1/100) *
    log2(1/100)  #= 0.8735663
```

Now that entropy is understood, let's discuss information gain.

Information gain

In the context of decision trees, information gain measures the difference in uncertainty that is obtained by a partition of the data. It consists of the entropy before the partition is performed from the entropy after the partition is done. To illustrate this, let's take the example of the `Titanic` data we have discussed. We need to preprocess the data first:

```
1  #preparing the dataset
2  Titanic.Weights = as.data.frame(Titanic)
3  Titanic.df = Titanic.Weights[rep(1:nrow(Titanic.Weights),
4     Titanic.Weights$Freq),]
5  Titanic.df = Titanic.df[,1:4]
```

The following code shows how many people did and didn't survive on the `Titanic`:

```
table(Titanic.df$Survived)
```

The output is provided here:

```
No    Yes
1490  711
```

The entropy among all cases is computed as:

```
EntropAll = -(1490/2201) * log2(1490/2201) -(711/2201) *
   log2(711/2201) #= 0.9076514
```

The following code displays how many of the male and female individuals did and didn't survive:

```
table(Titanic.df$Sex,Titanic.df$Survived)
```

Here is the output:

```
         No  Yes
Male   1364  367
Female  126  344
```

The entropy among males is:

```
EntropM = -(1364/1731) * log2(1364/1731) -(367/1731) *
   log2(367/1731) # = 0.745319
```

The entropy among females is:

```
EntropF = -(126/470) * log2(126/470) -(344/470) *
   log2(344/470)  # = 0.8387034
```

The entropy after the partition is performed is computed as the sum of the entropies for each category weighted by the proportion of the observations they contain:

```
EntropSex = ( (1731/ 2201) * EntropM ) + ((470/ 2201) *
   EntropF ) # = 0.7652602
```

The information gain is:

```
InformationGain = EntropAll - EntropSex  # = 0.1423912
```

While we provided an example only for the sex, ID3 computes the information gain for all attributes that have not been selected at higher nodes. It iteratively creates the next node selecting the attribute that provides the highest information gain. It then continues until there are no more unused attributes, or if the node consists only of attributes of the same class. In both cases, a leaf node is created. Generally speaking, ID3 favors splits on attributes with a high number of modalities, and therefore, can produce inefficient trees.

C4.5

C4.5 works in ways similar to ID3, but uses the gain ratio as a partitioning criterion, which in part resolves the issue mentioned previously. Another advantage is that it accepts partition on numeric attributes, which it splits into categories. The value of the split is selected in order to decrease the entropy for the considered attribute. Other differences from ID3 are that C4.5 allows for post-pruning, which is basically the bottom up simplification of the tree to avoid overfitting to the training data.

The gain ratio

Using the gain ratio as partitioning criterion overcoming a shortcomes of ID3, which is to prefer attributes with many modalities as nodes because they have a higher information gain. The gain ratio divides the information gain by a value called split information. This value is computed as minus the sum of: the ratio of the number of cases in each modality of the attribute divided by the number of cases to partition upon, multiplied by the base 2 logarithm of the number of cases in each modality of the attribute (iteratively) divided by the number of cases to partition upon.

The formula for split information is provided here:

$$-\sum_{i=1}^{c}\frac{|Ti|}{|T|}\ log_2\frac{|Ti|}{|T|}$$

Post-pruning

A problem frequently observed using predictive algorithms is overfitting. When overfitting occurs, the classifier is very good at classifying observations in the training dataset, but classifies unseen observations (for example, those from a testing dataset) poorly. Pruning reduces this problem by decreasing the size of the tree bottom up (replacing a node by a leaf on the basis of the most frequently observed class), thereby making the classifier less sensitive to noise in the training data.

There are several ways to perform pruning. C4.5 implements pessimistic pruning. In pessimistic pruning, a measure of incorrectly classified observations is used to determine which nodes are to be replaced by leaves on the basis of the training data only. In order to do this, a first measure is computed as: the multiplication of the number of leaves multiplied by 0.5, plus the sum of the errors. If the number of correctly classified observations in the most accurate leaf of the node plus 0.5 is within one standard deviation around the previously computed value, the node is replaced by the classification of the leaf.

C5.0

C5.0 is an improvement of C4.5. New features include boosting and winnowing. The aim of boosting is to increase the reliability of the predictions by performing the analysis iteratively and adjusting observation weights after each iteration. Higher weight is given to misclassified observations giving them a higher importance in the classification, which usually makes the predictions better. Winnowing refers to the suppression of useless attributes for the main analysis.

Classification and regression trees and random forest

We will now introduce classification and regression trees (CART) and random forest. CART uses different statistical criteria to decide on tree splits. Random forest uses ensemble learning (a combination of CART trees) to improve classification using a voting principle.

CART

There are a number of differences between CART used for classification and the family of algorithms we just discovered. Here, we only superficially discuss the partitioning criterion and pruning.

In CART, the attribute to be partition is selected with the Gini index as a decision criterion. In classification trees, the Gini index is simply computed as: 1—the sum of the squared probabilities for each possible partition on the attribute. The formula notation is:

$$1 - \sum_{j=1}^{c} p_j^2$$

This is more efficient compared to information gain and information ratio. Note that CART does not only do necessary partitioning on the modalities of the attribute, but also merges modalities together for the partition (for example, modality A versus modalities B and C).

CART can predict a numeric outcome, which is not the case for the attributes we have seen previously. In the case of regression trees, CART performs regression and builds the tree in a way that minimizes the squared residuals. As this chapter is about classification, we will focus on this aspect here.

Regarding pruning, CART uses another dataset and generated pruned trees from iterative simplification of the previous tree (replacement of nodes by leaves). It then selects the tree that minimizes a measure called cost complexity. Cost complexity is computed as the error rate of the considered tree, plus the number of leaves multiplied by a complexity parameter. The complexity parameter can be set by the user (or determined by CART). It can range from 0 to 1. The higher the value, the smaller the obtained tree.

Random forest

In random forest, an algorithm close to CART is used to produce trees. Differences include the use of bagging and the selection of random predictors at each partition of the tree. Each tree votes on the classification and the majority vote wins—how democratic!

Bagging

The algorithm generates T trees (the default in the function we will use is `T=500`). For each T, a different random sample (sampling with replacement) of observation is obtained from the observation pool. The size of each of the samples is always the same as the number of observations in the original data. The aim of bagging is to reduce the impact of measurement error (noise) in the data and therefore avoid overfitting.

Random selection of attributes: for each partition, a number of attributes is selected. The analysis is used on these predictors. As for bagging, this procedure avoids overfitting to the training set, and therefore, a potentially better performance in predicting a test set.

Conditional inference trees and forests

Unlike previous algorithms, conditional inference trees rely on statistical significance in the selection of attributes on which to perform partitions. In conditional inference trees, the class attribute is defined as a function of the other attributes (iteratively). In short, the algorithm first searches for the attributes that significantly predict the class, in a null hypothesis test that can be selected in the call of the function. The strongest predictor (if any) is then selected for the first partition. Nodes are created after splitting the partition attribute, if numeric, in a way that maximizes the goodness of the split (we do not detail the required computations here). The algorithm then repeats the operation for each of the nodes and continues until no attribute remains, or none is significantly related to the class. More information is available in the documentation of the `partykit` package.

Installing the packages containing the required functions

Let's start by installing the different packages.

Installing C4.5

J48 is the implementation of C4.5 in Weka. It is readily available to R users through the `RWeka` package. Let's start by downloading and installing `RWeka`:

```
install.packages("RWeka"); library(RWeka)
```

This should work fine if the Java version (32 or 64 bit) matches the version of R that you are running. If the JAVA_HOME cannot be determined from the Registry error occurs, please install the correct Java version as explained at `http://www.r-statistics.com/2012/08/how-to-load-the-rjava-package-after-the-error-java_home-cannot-be-determined-from-the-registry/`. The package discussed is the `rJava` package, but the advice given should work just fine for issues with `RWeka` too.

Installing C5.0

C5.0 is included in the `C50` package, which we will install and load:

```
install.packages("C50"); library(C50)
```

Installing CART

The `rpart()` function of the `rpart` package allows us to run CART in R. Let's start by installing and loading the package:

```
install.packages("rpart"); library(rpart)
```

We'll also install the `rpart.plot` package, which will allow us to plot the trees:

```
install.packages("rpart.plot"); library(rpart.plot)
```

Installing random forest

Let's install and load the `randomForest` package:

```
install.packages("randomForest"); library(randomForest)
```

Installing conditional inference trees

Let's install and load the `partykit` package that contains the `ctree()` function, which we will use. We also install the `Formula` package, which is required by `partykit`:

```
install.packages(c("Formula","partykit"))
library(Formula); library(partykit)
```

Loading and preparing the data

Let's start by loading `arules`; the package contains the `AdultUCI` dataset, which we will use first:

```
install.packages("arules")
library(arules)
```

In this dataset, the class (the attribute named `income`) is the annual salary of individuals—whether it is above (modality large) or below (modality small) $50,000. The attributes and their type can be seen on the following summary of the dataset:

```
data(AdultUCI)
summary(AdultUCI)
```

Here is the output:

```
      age                workclass        fnlwgt             education     education-num                  marital-status          occupation
Min.   :17.00   Private         :33906   Min.   :  12285   HS-grad     :15784   Min.   : 1.00   Divorced              : 6633   Prof-specialty : 6172
1st Qu.:28.00   Self-emp-not-inc: 3862   1st Qu.: 117551   Some-college:10878   1st Qu.: 9.00   Married-AF-spouse     :   37   Craft-repair   : 6112
Median :37.00   Local-gov       : 3136   Median : 178145   Bachelors   : 8025   Median :10.00   Married-civ-spouse    :22379   Exec-managerial: 6086
Mean   :38.64   State-gov       : 1981   Mean   : 189664   Masters     : 2657   Mean   :10.08   Married-spouse-absent:  628   Adm-clerical   : 5611
3rd Qu.:48.00   Self-emp-inc    : 1695   3rd Qu.: 237642   Assoc-voc   : 2061   3rd Qu.:12.00   Never-married         :16117   Sales          : 5504
Max.   :90.00   (Other)         : 1463   Max.   :1490400   11th        : 1812   Max.   :16.00   Separated             : 1530   (Other)        :16548
                NA's            : 2799                     (Other)     : 7625                   Widowed               : 1518   NA's           : 2809
       relationship                   race           sex         capital-gain     capital-loss     hours-per-week       native-country     income
Husband       :19716   Amer-Indian-Eskimo:  470   Female:16192   Min.   :    0   Min.   :   0.0   Min.   : 1.00   United-States:43832   small:24720
Not-in-family :12583   Asian-Pac-Islander: 1519   Male  :32650   1st Qu.:    0   1st Qu.:   0.0   1st Qu.:40.00   Mexico       :  951   large: 7841
Other-relative: 1506   Black             : 4685                 Median :    0   Median :   0.0   Median :40.00   Philippines  :  295   NA's :16281
Own-child     : 7581   Other             :  406                 Mean   : 1079   Mean   :  87.5   Mean   :40.42   Germany      :  206
Unmarried     : 5125   White             :41762                 3rd Qu.:    0   3rd Qu.:   0.0   3rd Qu.:45.00   Puerto-Rico  :  184
Wife          : 2331                                           Max.   :99999   Max.   :4356.0   Max.   :99.00   (Other)      : 2517
                                                                                                                  NA's         :  857
```

Summary of the AdultUCI dataset

We can see that there are a lot of missing values. We will therefore now remove the observations containing missing values from the dataset. There are also two attributes we will not use: `fnlwgt` and `education-num`. The following are attributes three and five, which we will therefore also remove from the dataset:

```
ADULT = na.omit(AdultUCI)[,-c(3,5)]
```

We are now ready to generate our training and testing datasets. For this purpose, we will use the `createDataPartition()` function of the `caret` package, which we install with its dependencies:

```
install.packages("caret", dependencies =
   (c("Suggests","Depends")));
library(caret)
set.seed(123)
TrainCases = createDataPartition(ADULT$income, p = .5,
  list=F)
```

Notice the income attribute is unbalanced; there are far more small income than large income individuals. Hence, we will rebalance this skewed dataset by oversampling the minority class in the training dataset:

```
1  TrainTemp = ADULT[TrainCases,]
2  AdultTrainSmallIncome = TrainTemp[TrainTemp$income == "small",]
3  AdultTrainLargeIncome = TrainTemp[TrainTemp$income == "large",]
4  Oversample = sample(nrow(AdultTrainLargeIncome),
5     nrow(AdultTrainSmallIncome), replace = TRUE)
6  AdultTrain = rbind(AdultTrainSmallIncome,
7  AdultTrainLargeIncome[Oversample,])
8  AdultTest = ADULT[-TrainCases,]
```

Performing the analyses in R

Now that we have our data ready, we will focus on performing the analyses in R.

Classification with C4.5

We will first predict the income of the participants using C4.5.

The unpruned tree

We will start by examining the unpruned tree. This is configured using the `Weka_Control(U= TRUE).J48()` argument in `RWeka`, which uses the formula notation we have seen previously. The dot (`.`) after the tilde indicates that all attributes except the `class` attribute have to be used. We used the `control` argument to tell R that we want an unpruned tree (we will discuss pruning later):

```
C45tree = J48(income ~ . , data= AdultTrain,
   control= Weka_control(U=TRUE))
```

You can examine the tree by typing:

```
C45tree
```

We will not display it here as it is very big: the size of the tree is 5,715, with 4,683 leaves; but we can examine how well the tree classified the cases:

```
summary(C45tree)
```

```
Correctly Classified Instances       20206               89.194  %
Incorrectly Classified Instances      2448               10.806  %
Kappa statistic                          0.7839
Mean absolute error                      0.1448
Root mean squared error                  0.2836
Relative absolute error                 28.9603 %
Root relative squared error             56.7242 %
Coverage of cases (0.95 level)          98.2829 %
Mean rel. region size (0.95 level)      69.6654 %
Total Number of Instances            22654

=== Confusion Matrix ===

     a     b   <-- classified as
  9699  1628 |     a = small
   820 10507 |     b = large
```

The performance of the classifier on the training dataset

We can see that even though about 89 percent of cases are correctly classified, the kappa statistic (which we discussed in the previous chapter) is .78, which is not bad. In practice, a value of 0.60 or higher is highly recommended.

The following will try to classify the test set and assign the predictions to a new attribute in a data frame called `Predictions`:

```
Predictions = data.frame(matrix(nrow = nrow(AdultTest), ncol=0))
Predictions$C45 = predict(C45tree, AdultTest)
```

The pruned tree

Let's examine what happens with a pruned tree, before we see the result on unseen data (the testing dataset):

```
C45pruned = J48(income ~ . , data= AdultTrain,
   control= Weka_control(U=FALSE))
```

The resulting tree is smaller, but still quite big; it has a size of 2,278 and 1,767 leaves. Typing the following line, you will see that around the same number of instances were correctly classified and that kappa is now 0.76. Pruning the tree decreased the classification performance on the training data:

```
summary(C45pruned)
```

The following will try to classify the test set and assign those to a new attribute called `PredictedC45pr`, which we will examine later:

```
Predictions$C45pr = predict(C45pruned, AdultTest)
```

C50

As a reminder, C50 performs boosting, which is the reiteration of the classification with a higher weight given to misclassified observations. Let's run a boosted C5.0 with 10 trials (boosted 10 times) and examine the output. Only the accuracy and confusion matrix are displayed here, but you can examine the tree on your screen:

```
C50tree = C5.0(y = AdultTrain$income, x = AdultTrain[,-13],
   Trials = 10)
summary(C50tree)
```

Here is part of the output (at the bottom of your screen):

```
Evaluation on training data (22654 cases):
Decision Tree
----------------
Size      Errors
708  1980 (8.7%)   <<
(a)   (b)    <-classified as
----  ----
10061  1266  (a): class small
714  10613  (b): class large
```

The size of the tree is 714, which is much smaller than the previously unpruned version of C4.5. We can see that the accuracy is a bit better using boosted C5.0 (8.4 percent misclassified observations). What about the kappa value? Let's try another way to get it this time! It can be obtained by first creating a table from the confusion matrix and then, calling the `cohen.kappa()` function on it:

```
1  TabC5.0= as.table(matrix(c(10061,1980,714,10613),
2     byrow=T, nrow = 2))
```

```
3  library(psych)
4  cohen.kappa(TabC5.0)
```

The output shows that the kappa value is `.77`. In this case, C5.0 is similar to C4.5, on the training data. Let's now create the prediction on unseen data, which we will examine later:

```
Predictions$C5.0 = predict(C50tree, AdultTest)
```

CART

We will try to classify the same data again and see how well the CART performs on our training and testing datasets:

```
CARTtree = rpart(income ~. , data= AdultTrain)
```

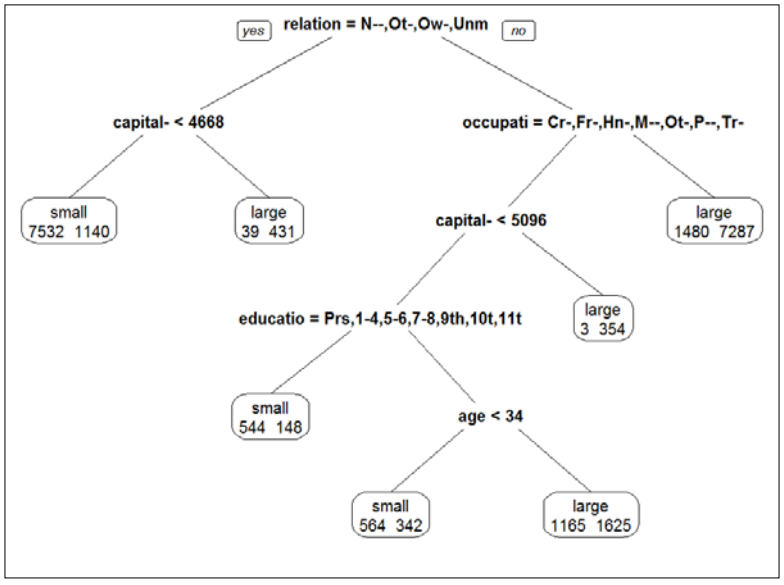
The following line of code displays a big output about the performed splits. We will not comment on this here as the following plot is more informative:

```
summary(CARTtree)
```

Here, we will simply include the plot of the tree that is more readable:

```
rpart.plot(CARTtree, extra = 1)
```

As can be noted on the graph, the tree is far simpler than with C4.5 or C5.0:



A graph of the tree using CART

Obtaining the confusion matrix for the training data is a bit more difficult with CART. It requires predicting the class on the training data. The predictions are made in terms of probabilities (as you can see when looking at the tree in its textual form). We, therefore, recode values higher than `.5` as a large income and those lower, as a small income. We will also create a temporary data frame of the data without missing values and will create the confusion matrix on this data frame. Finally, we display the confusion matrix and the kappa value:

```
1  ProbsCART = predict(CARTtree, AdultTrain)
2  PredictCART = rep(0, nrow(ProbsCART))
3  PredictCART[ProbsCART[,1] <=.5] = "small"
4  PredictCART[ProbsCART[,1] >.5] = "large"
5  TabCART = table(AdultTrain$income, PredictCART)
6  TabCART
7  cohen.kappa(TabCART)
```

The output is shown here:

```
       large  small
small   8640   2687
large   1630   9697
```

The accuracy for this classification is not very different from the other algorithms:

```
(8640+9696)/sum(TabCART) = 81%.
```

However, the kappa value is lower (0.62). Classification with CART is not good for this dataset, but there are ways to improve it.

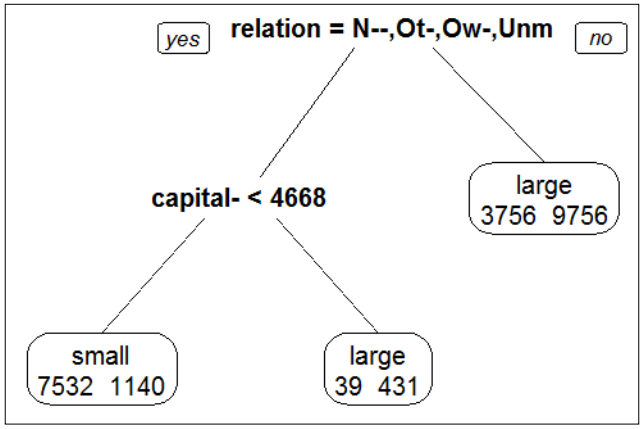
Pruning

As mentioned previously, pruning in CART is based on the generation simplified trees from another dataset. So let's try to obtain an even simpler tree than before. The `cp` argument on the code corresponds to the complexity parameter that we discussed at the beginning of the section:

```
CARTreePruned = prune(CARTtree, cp=0.03)
```

The tree is even smaller now:

```
rpart.plot(CARTtreePruned, extra = 1)
```



A graph of a pruned tree using CART

I will now comment on this tree a bit. The root node is on the relationship attribute. Let's examine the modalities of this attribute:

```
levels(AdultTrain$relationship)
```

The output is as follows:

```
[1] "Husband"        "Not-in-family"  "Other-relative" "Own-child"
[5] "Unmarried"      "Wife"
```

From the plot, we can see that people who are `Not-in-family`, live with `Other-relative`, with their `Own-child`, and are `Unmarried` have mostly a small income. Among people who are not in these categories, the tree splits on the attribute `capital-gain`. People with a capital gain lower than 4668 mostly have a low income. The people in the other categories mostly have a large income.

Using the same approach as before, we obtain an accuracy of 78 percent on the training dataset, and 81 percent on the second set. The kappa values here are only 0.56. This is not encouraging. However, remember that things can go much better with other datasets, including your data (give it a try).

We now classify the test set for later:

```
ProbsCARTtest = predict(CARTtreePruned, AdultTest)
Predictions$CART[ProbsCARTtest[,1] <=.5] = "small"
Predictions$CART[ProbsCARTtest[,1] >.5] = "large"
```

Random forests in R

```
RF = randomForest(y = AdultTrain$income, x = AdultTrain[,-13])
```

The confusion matrix can be obtained by typing:

```
RF
```

The output is as follows:

```
Call:
 randomForest(x = AdultTrain[, -13], y = AdultTrain$income)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3
        OOB estimate of  error rate: 35.79%
Confusion matrix:
small  large  class.error
small  11301   26  0.0022954
large   8081  3246  0.7134281
```

This is pretty disappointing. The classification is even worse than with CART. What about prediction accuracy on the testing dataset? Let's add this to our `Predictions` data frame:

```
Predictions$RF = predict(RF, AdultTest)
```

Here, we used the default parameters. The classification should depend notably on two arguments (type `?randomForest()` in the console for more arguments):

- `mtry`: This determines the number of predictors to be included for each split. By default, `mtry = sqrt(p)`, where `p` is the total number of predictors in the analysis.
- `cutoff`: This determines the cutoff for the probability of membership to be used in the classification. By default, for dichotomous classification, `cutoff = c(0.5, 0.5)`, which means that a cutoff value of 0.5 is used as a threshold.

Examining the predictions on the testing set

In the previous section, we predicted the income attribute on the testing set using several algorithms. We can now examine which algorithm was better in these predictions. We could do this manually, but it is faster and more fun to design some code that gives immediate access to the accuracy of the different algorithms on this data. So here we go:

```
values = data.frame(matrix(ncol = ncol(Predictions), nrow = 6))
rownames(values) = c("True +", "True -", "False +", "False -",
   "Accuracy", "Kappa")
names(values) = names(Predictions)
for (i in 1:ncol(Predictions)) {
   tab = table(AdultTest$income,Predictions[,i])
   values[1,i] = tab[1,1]
   values[2,i] = tab[2,2]
   values[3,i] = tab[1,2]
    values[4,i] = tab[2,1]
    values[5,i] = sum(diag(tab))/sum(tab)
    values[6,i] = cohen.kappa(tab)[1]
}
round(values,2)
```

In the output, we can see that although the algorithms all reached more than 70 percent of classification accuracy, the kappa value was never above 0.56. This is not sufficient and we would not trust such a classification in practice. Contrary to what could be expected, random forest performed the worst. It is, therefore, crucial to always try several algorithms, and pick the one that works the best with your data:

	C45	C45pr	C5.0	CART	RF
True +	**9400**	**9352**	**9438**	**7543**	**11300**
True -	**2737**	**2920**	**2949**	**3338**	**793**
False +	**1927**	**1975**	**1889**	**3784**	**27**
False -	**1017**	**834**	**805**	**416**	**2961**
Accuracy	**0.8**	**0.81**	**0.82**	**0.72**	**0.80**
Kappa	**0.52**	**0.55**	**0.56**	**0.43**	**0.28**

Conditional inference trees in R

The following code was used to produce the figure we saw at the beginning of the chapter:

```
1  set.seed(999)
2  TitanicRandom = Titanic.df[sample(nrow(Titanic.df)),]
3  TitanicTrain = TitanicRandom[1:1100,]
4  TitanicTest = TitanicRandom[1101:2201,]
```

Let's now generate and plot the tree:

```
CItree <- ctree(Survived ~ Class + Sex + Age, data=TitanicTrain)
plot(CItree)
```

We can examine the classification of the confusion matrix of the training and testing datasets as we did using the previously presented algorithms:

```
1  CIpredictTrain = predict(CItree, TitanicTrain)
2  CIpredictTest = predict(CItree, TitanicTest)
3  TabCI_Train = table(TitanicTrain$Survived,CIpredictTrain)
4  TabCI_Test = table(TitanicTest$Survived,CIpredictTest)
5  TabCI_Train
```

The training dataset has the following confusion matrix:

```
CIpredictTrain
      No  Yes
No   724   10
Yes  231  135
```

Here, we display the confusion matrix for the testing dataset:

```
CIpredictTest
      No  Yes
No   746   10
Yes  226  119
```

We can see that in both datasets, only about a third of the individuals who survived were correctly classified. The kappa values are very low—0.42 for the training set and 0.4 for the testing set, despite statistical significance at all splits in the training set!

Yet, having a look at the figure at the beginning of the chapter, we can see that almost all women were correctly classified. Sometimes, some subgroups are easier to classify than others. This means that depending on the aim of your analysis, even the preceding results might be very informative!

Caret – a unified framework for classification

As we have seen, there are a number of differences between algorithms. For instance, some use the formula notation and some the matrix notation. The `caret` package uses a similar notation for all the algorithms it supports. Further, it contains tools that perform sampling operations, such as generating training and testing data with the same characteristics (stratified sampling), the use of boosting of bagging with several algorithms, and cross-validation samples. Examples of cross-validation include, for instance, the use of 10 subsamples, of which one is iteratively used as testing data and the rest as training data (or the leave-one-out cross-validation, where one observation is iteratively used as testing data and the rest as training data). Other features are included as well, such as examining the performance of the classification, as we have done previously. The `caret` package will be further discussed in *Chapter 14, Cross-validation and Bootstrapping Using Caret and Exporting Predictive Models Using PMML*.

Summary

In this chapter, we performed classifications using decision trees. We explored the criteria used by several algorithms to perform the splits and described the features of these algorithms. We described how to examine the performance of the algorithms and discussed the importance of doing this on the training and testing datasets. In the next chapter, we will discuss multilevel regression in R.

12
Multilevel Analyses

In *Chapter 10, Classification with k-Nearest Neighbors and Naïve Bayes*, we discussed association with k-Nearest Neighbors and Naïve Bayes. In the previous chapter, we examined classification trees using notably C4.5, C50, CART, random forests, and conditional inference trees.

In this chapter, we will discuss:

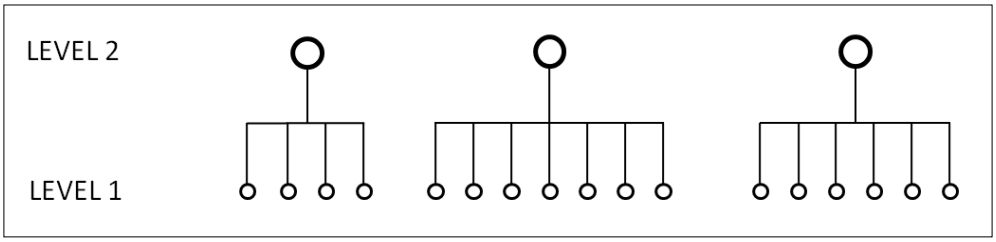
- Nested data and the importance of dealing with them appropriately
- Multilevel regression including random intercepts and random slopes
- The comparison of multilevel models
- Prediction using multilevel modeling

Nested data

If you have nested data, this chapter is essential for you! What is meant by *nested data* is that observations share a common context. The examples include:

- Consumers nested within shops
- Employees nested within managers
- Teachers and/or students nested within schools
- Nurses, patients, and/or physicians nested within hospitals
- Inhabitants nested in neighborhoods

We could imagine way more cases of data nesting. What they all have in common is a data structure similar to the one depicted in the following figure:



A depiction of nested data

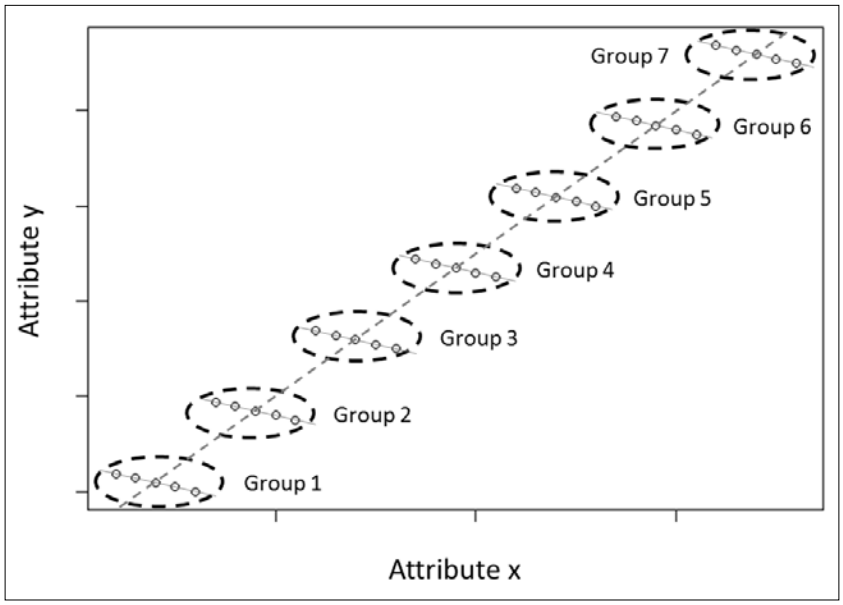
We will only discuss two levels of data with unique membership in this chapter, but of course, more complex situations can arise. For instance, in all the preceding examples, shops, managers, schools, hospitals, and neighborhoods can be nested within higher level units (for example, companies, cities) which could be a third level in the analyses). Also, crossed memberships could be imagined, for example, patients sharing a hospital but not a neighborhood. This type of data is more complex to analyze and as always, space is scarce in this chapter. Note also that it usually happens that data is collected at both levels, for instance: the job satisfaction of employees (level 1), and the type of leadership of the managers (level 2).

If your data has a hierarchical structure, traditional regression analysis will most likely produce unreliable results. This is because the observations are not independent, but the analysis assumes they are. One of the consequences is that standard errors can be underestimated, which could lead to spuriously significant results.

Another problem is known as the Robinson effect, which refers to the increase in the statistical relationship between attributes when data is aggregated as compared to when they are not. While the phenomenon has not been been named after the castaway, but the researcher who discovered the phenomenon, the result of blindly aggregating data on the higher level and examining relationships between attributes at that level might only lead to the shipwreck of the analysis.

Spurious results can also be due to characteristics of the context that are shared by observations within the groups. Drawing conclusions at one level with data collected at another level is likely to be erroneous because relationships might be different at different levels. The atomistic fallacy is drawing conclusions at the lower level from data at a higher level. The ecological fallacy is just the opposite—drawing conclusions at a higher level from data at a lower level.

Let's examine an example of ecological fallacy visually in the following figure:



A depiction of opposing findings at different levels

This plot represents fictional data from seven groups on two hypothetical attributes: **Attribute x** and **Attribute y**. Imagine we compute the average values for each of the groups (the mean of the dots in the dashed ovals); these aggregated values would show a strong positive relationship (thick dashed line) between **Attribute x** and **Attribute y**. However, if we examine the real relationship within each group, we can see that **Attribute x** is actually slightly negatively related to **Attribute y** within each group (thin plain lines).

Another related problem is the Simpson's paradox (named after the statistician, not the cartoon character). In analyzing this dataset, we would also find a positive relationship if we simply considered all the groups together in a regression analysis at level 1; the positive relationship is due to the fact that the groups differ in the values of **Attribute y** because of an unmeasured attribute, and concurrently, they also differ in the values of **Attribute x**. In other words, the level of the unmeasured attribute (here related to an increase in both x and y) is shared by the observations within each group, but not between groups. Not making the distinction between groups in the analysis would also lead to inaccurate results.

Multilevel regression

To solve all these issues, we can rely on a kind of analysis that can partial out (take away) the variance due to the context. This can be done using multilevel regression analysis (also known as mixed-effect regression). We will not go into the detail of the computations of such highly complex analyses but will simply provide the amount of information necessary to understand and perform the analysis at a basic level. The necessary diagnostic checks are not fully presented here. Simply note that diagnostics for linear regression apply, and that additional diagnostics should be performed, such as checking the normality of residuals at level 2. We will not discuss this further here. The *Handbook of multilevel analysis* book, edited by De Leeuw and Meijer, provides the necessary information for diagnostics of multilevel models.

When we discussed regression in *Chapter 9*, *Linear Regression*, we showed that the value of a criterion attribute for an observation is computed as the sum of:

- The intercept (the average value when the value of all included predictors equal 0)
- The slope coefficient multiplied by the value of the predictor (for each predictor)
- The residual (the difference between the predicted value and the observed value)

We explained how the regression algorithm finds the parameters that minimize the residuals on the whole sample.

Random intercepts and fixed slopes

In multilevel modeling, when considering predictors only at level 1 and considering a common slope for all groups, the value of a criterion attribute on an observation is schematically computed as the sum of:

- A common intercept
- A group-specific residual (which is the difference between the group's intercept and the common intercept)
- The slope coefficient multiplied by the value of the predictor (for each predictor)
- An observation specific residual

In this type of model, the effect of the attributes at level 1 is considered the same across all groups. Only the intercept varies.

When considering predictors at levels 1 and 2 and considering common slopes for all groups, the computation is schematically the sum of:

- A common intercept
- A group-specific residual corresponding to the difference between the group's intercept and the common intercept
- An overall slope coefficient multiplied by the value of the predictor (for each predictor at level 1)
- An observation-specific residual

The computations are actually more complex, but this goes beyond the material covered in this chapter. Simply note that it is the job of multilevel regression to find the parameters that minimize the residuals on the whole sample.

Random intercepts and random slopes

Until now, we have considered that the slope of level 1 predictors is the same across groups. This is not always the case. Let's examine this with an example. We will use simulated data generated from real data, with attributes about burnout (personal accomplishment, depersonalization, and emotional exhaustion) and work satisfaction. You might remember that we have used similar data in the chapter about regression.

The following code loads the covariance's data and generates the dataset from it (100 observations for each of 17 hospitals):

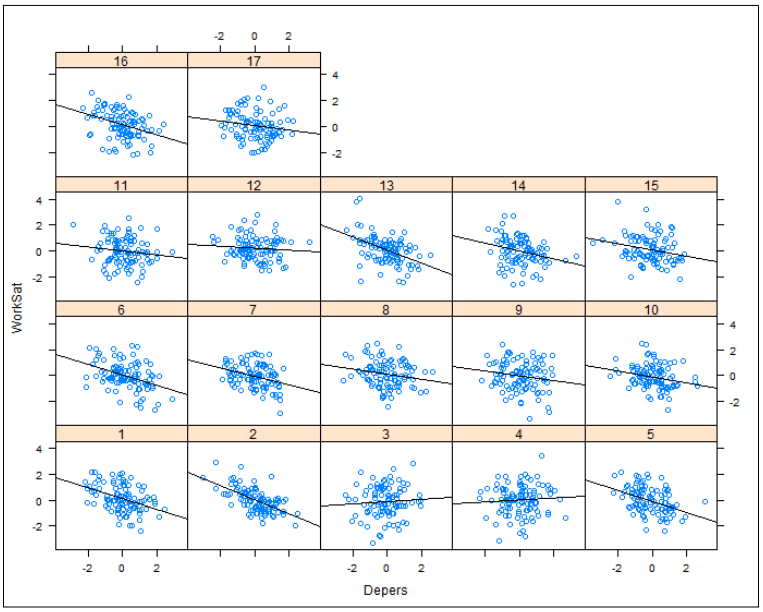
```
1  library(MASS)
2  set.seed(999)
3  Covariances = read.table("Covariances.dat", sep = "\t", header=T)
4  df = data.frame(matrix(nrow=0,ncol=4))
5  colnames(df) = c("Hospital","Accomp","Depers","Exhaus","WorkSat")
6  for (i in 1:17){
7    if(i == 1) {start_ln = 1}
8    else start_ln = 1+((i-1)*4)
9    end_ln = start_ln + 3
10    covs = Covariances[start_ln:end_ln, 3:6]
11    rownames(covs)=Covariances[start_ln:end_ln,2]
12    dat=mvrnorm(n=100, c(rep(0,4)), covs)
13    df = rbind(df,dat)
14  }
```

```
15  df$hosp = as.factor(c(rep(1,100), rep(2,100), rep(3,100),
16     rep(4,100), rep(5,100), rep(6,100),
17     rep(7,100),rep(8,100),rep(9,100),
18     rep(10,100),rep(11,100),rep(12,100),
19     rep(13,100),rep(14,100),rep(15,100),
20     rep(16,100),rep(17,100)))
```

The following code will plot the relationship (using an `lm()` model) between depersonalization and work satisfaction with each of the hospitals:

```
1  library(lattice)
2  attach(df)
3  xyplot(WorkSat~Depers | hosp, panel = function(x,y) {
4    panel.xyplot(x,y)
5    panel.lmline(x,y)
6  })
```

As can be seen on the following plot, there is usually a negative relationship between depersonalization and work satisfaction, but groups do not show this pattern to the same extent, and in some cases the relationship is not even present.



The relationship between depersonalization and work satisfaction by the hospital

Whether to take into account these variations or not in the analysis is the decision of the analyst. We will discuss this further in the next section. A random slopes model refers to a model in which the slopes are allowed to vary between groups.

For now, let's examine the schematic computation of the individual values when dealing with random slopes models. This value is obtained as the sum of:

- A common intercept.
- A group-specific residual (the difference between the group's intercept and the common intercept).
- A group-specific coefficient multiplied by the value of the predictor, for each predictor for which random slopes are included. This group's specific coefficients are composed of a fixed part, the common slope; and a random part, the residual that corresponds for each slope to the variation of the group around that slope.
- For each predictor for which the slope is not allowed to vary between groups (if any), a slope coefficient multiplied by the value for the predictor.
- An observation-specific residual.

Again, the job of multilevel regression is to find the parameters that minimize the residuals. Note that the effect of some predictors can be defined as varying between groups, and the others as not varying.

Multilevel modeling in R

Now that we have examined (laconically) the basics of multilevel modeling equations, we can turn to how to build multilevel models in R and predict unseen data.

For this purpose, we will first load our dataset produced using the same procedure as mentioned previously (except that the attributes are not scaled). Here again, there are 100 generated observations for each of the 17 hospitals:

```
NursesML = read.table("NursesML.dat", header = T, sep = " ")
```

The null model

We will examine the variation in our attributes considering hospitals and observations as a unit of analysis, that is, we will compare whether there is more variation at the hospital and observation levels. What we could do is compute this by hand.

The following will compute the mean for the attribute we want to predict (`WorkSat`) for each of the hospitals:

```
means = aggregate(NursesML[,4], by=list(NursesML[,5]),
   FUN=mean)[2]
```

We can display the variance of work satisfaction in hospitals and observations as follows:

```
var(unlist(means)) #at the hospital level
var(NursesML[,4]) #at the observation level
```

The output is `0.0771365` for hospitals and `0.7914461` for observations. Far more variance lies at the observation level than at the hospital level. Yet, the variance at the hospital level is present at the observation level and vice versa. The results are therefore not trustworthy.

Using multilevel modeling to examine such differences is the correct way to perform the comparison. In order to do it, we need to fit a multilevel model that only includes a constant and the clustering variable. This is known as the `null` model. We will start by installing and loading the `lme4` package. The `lmer()` function in this package allows fitting multilevel models. The version download at the time of writing this text is 1.1-8. Your output could vary if you download a different version:

```
install.packages("lme4"); library(lme4)
```

We first tell R that the `hosp` attribute is a factor:

```
NursesML$hosp = factor(NursesML$hosp)
```

We fit the `null` model as follows:

```
null = lmer(WorkSat ~ 1 + (1|hosp), data=NursesML)
```

Let's examine the summary of the `null` model:

```
summary(null)
```

The output is provided here:

```
Linear mixed model fit by REML ['lmerMod
Formula: WorkSat ~ 1 + (1 | hosp)
   Data: NursesML

REML criterion at convergence: 4321.9

Scaled residuals:
    Min      1Q  Median      3Q     Max
-3.8527 -0.6556 -0.0038  0.6823  4.0239

Random effects:
 Groups   Name        Variance Std.Dev.
 hosp     (Intercept) 0.06988  0.2643
 Residual             0.72564  0.8518
Number of obs: 1700, groups:  hosp, 17

Fixed effects:
            Estimate Std. Error t value
(Intercept)  5.10679    0.06736   75.81
```

The output of the null model

Under `Random effects`, we can see that `Variance` for `Intercept` is `0.06988`. This is the variance at the hospital level. The residual variance, that is, the variance at the observation level is `0.72564`. The total variance in work satisfaction is therefore *0.06988 + 0.72564 = 0.79552*. We can compute the proportion of variance at the hospital level (known as the intraclass correlation) as *0.06988 / 0.79552 = 0.08784191*. Approximately, 9 percent of variance lies at the hospital level. A rule of thumb is to consider that datasets with less than 5 percent of variance at level 2 (the hospital here) could be analyzed using traditional regression without being much concerned about the nesting of data. Note that this applies only if there are no predictors at level 2. Under `Fixed effects`, we only have `Intercept` here. Its value of `5.10679` in the null model means that the average value among observations is about `5.10`. We can compare this value with the value that is returned by the simple `mean()` function:

```
mean(NursesML[,4])
```

The result, `5.106792`, is identical.

It is possible to obtain the intercept in each of the hospitals as follows:

```
coef(null)
```

We have already computed the mean at the hospital level (which is stored in the means object). We can therefore display those easily:

```
means
```

Both outputs are presented here (intercepts on the left). Note that we can notice minor differences due to the computations using `lmer()`, because these are based on the distribution of the values in the hospitals:

	(Intercept)		x
1	**5.313038**	**1**	**5.334455**
2	**4.945022**	**2**	**4.928223**
3	**5.810795**	**3**	**5.883899**
4	**5.113663**	**4**	**5.114376**
5	**5.184669**	**5**	**5.192756**
6	**5.055792**	**6**	**5.050496**
7	**5.359504**	**7**	**5.385746**
8	**4.975973**	**8**	**4.962389**
9	**4.759738**	**9**	**4.7237**
10	**5.16671**	**10**	**5.172932**
11	**5.330523**	**11**	**5.353755**
12	**4.829187**	**12**	**4.80036**
13	**5.042653**	**13**	**5.035993**
14	**5.061342**	**14**	**5.056623**
15	**4.999852**	**15**	**4.988747**
16	**5.056085**	**16**	**5.05082**
17	**4.810918**	**17**	**4.780195**

The intercepts we display here are composed of a fixed part (the overall intercept) and a random part (one value per hospital), which correspond to the deviation of each hospital. The random part can be obtained using the `ranef()` function:

```
ranef(null)
```

Note that the `coef()` function only returns the preceding intercepts because we have not yet included predictors in the analysis. We will do this in a moment.

We will examine how to test the normality of residuals after we introduce random slopes.

Random intercepts and fixed slopes

We will now perform our first analysis for `sep`. In this case, we want to examine the relative impact of personal accomplishment, depersonalization, and emotional exhaustion on work satisfaction. We will not yet include potential variation of the effect of the predictors between hospitals. It is common to center the predictors around the grand mean before running the analyses. We therefore prepared the following training and testing datasets, which contain 50 observations per hospital. You can now load the datasets:

```
NursesMLtrain = read.table("NursesMLtrain.dat",
   header = T, sep = " ")
NursesMLtest = read.table("NursesMLtest.dat",
   header = T, sep = " ")
```

Let's make sure the `hosp` attribute is considered a factor in both datasets:

```
NursesMLtrain$hosp = factor(NursesMLtrain$hosp)
NursesMLtest$hosp = factor(NursesMLtest$hosp)
```

Let's now fit the model in the training data:

```
model = lmer(WorkSat ~ Accomp + Depers + Exhaust + (1|hosp),
   data=NursesMLtrain, REML = F)
```

The first thing we want to know is whether the model we just computed fits the data better than a null model. This requires comparing a value in both models: the `-2loglikelihood`. As we now have a different dataset than when we computed the null model, we have to fit this model again. Another reason is that the comparison of `-2loglikelihood` values is unreliable with restricted maximum likelihood (`REML`, the estimator used by default in `lmer()`). We will not explain this further and simply use **maximum likelihood** (**ML**) instead by stating `REMPL = F`:

```
null = lmer(WorkSat ~ 1 +  (1|hosp), data=NursesMLtrain, REML = F)
```

We can compare the `-2loglikelihood` values using the `anova()` function (even though a chi-square test is actually performed), as for traditional regression models:

```
anova(null, model)
```

The output is as follows:

```
Data: NursesMLtrain
Models:
null: WorkSat ~ 1 + (1 | hosp)
model: WorkSat ~ Accomp + Depers + Exhaust + (1 | hosp)
      Df    AIC    BIC   logLik deviance  Chisq Chi Df Pr(>Chisq)
null   3 2156.3 2170.5 -1075.15   2150.3
model  6 1984.2 2012.6  -986.08   1972.2 178.15      3  < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Model comparison (null versus model)

The `AIC` and `BIC` columns refer to the Akaike Information Criterion and Bayes Information Criterion. As for the -2loglikelihood value, these are measures of how well the data fits the model, but they take the model complexity (the number of included parameters) into account. The deviance column is also a measure of model fit. Smaller values are preferred for AIC, BIC and deviance, whereas the -2loglikelihood values should increase with better fit. The Chisq column refers to the difference in -2loglikelihood between models (which closely follow a chi-square distribution). The degrees of freedom (next column) for the chi-square test are computed as the difference in degrees between the two models. Finally, the last column is p-value for the test. The three asterisks show that the model is significant at a value close to `0`, as displayed on the significance codes below the table. From this, we conclude that the model with the three predictors included is better than the null model.

We can compute the additional part of variance explained by our model using the `r.squaredLR()` function from the `MuMIn` package, which we first install and load. The `r.squaredLR()` function takes our model with the predictors and the null model as arguments:

```
install.packages("MuMIn"); library(MuMIn)
r.squaredLR(model,null)
```

The output shows that the (pseudo) `R squared` value is `0.189`, meaning that our model predicted about 19 percent of the variance in the null model.

We can now examine the summary of the model as follows:

```
summary(model)
```

The output is as follows:

```
Linear mixed model fit by maximum likelihood  ['lmerMod']
Formula: WorkSat ~ Accomp + Depers + Exhaust + (1 | hosp)
   Data: NursesMLtrain

     AIC      BIC   logLik deviance df.resid
  1984.2   2012.6   -986.1   1972.2      844

Scaled residuals:
    Min      1Q  Median      3Q     Max
-3.0252 -0.6756  0.0225  0.6616  3.4649

Random effects:
 Groups   Name        Variance Std.Dev.
 hosp     (Intercept) 0.06674  0.2583
 Residual             0.57345  0.7573
Number of obs: 850, groups:  hosp, 17

Fixed effects:
            Estimate Std. Error t value
(Intercept)  5.11854    0.06783   75.46
Accomp       0.17611    0.03749    4.70
Depers      -0.07335    0.03135   -2.34
Exhaust     -0.29215    0.02935   -9.95

Correlation of Fixed Effects:
        (Intr) Accomp Depers
Accomp   0.000
Depers   0.000  0.041
Exhaust  0.000  0.044 -0.490
```

A summary of the random intercept model

Unfortunately, `lmer()` does not provide p values for the coefficients. Obtaining p values is often considered essential in hypothesis testing (although examining the confidence intervals is sometimes preferred). To obtain the p values, we need to perform their computation ourselves. Note that there is some debate as to the reliability of the computation of p values in multilevel modeling, which is why this is not included by default in the output. We start by extracting the t-values. We then compared these to a normal distribution and output them:

```
tvals = coef(summary(model))[,3]
tvals.p <- 2 * (1 - pnorm(abs(tvals)))
round(tvals.p,3)
```

The following output shows that the intercept, personal accomplishment, depersonalization, and emotional exhaustion are significantly different than `0` at `p < .001`.

(Intercept)	Accomp	Depers	Exhaust
0.000	0.000	0.019	0.000

We can now examine `Fixed effects`. We can see that at the mean level of each of the predictors (we are using centered attributes), the average work satisfaction when all predictors are at their average level is `5.11854`. An increase of one unit in personal accomplishment is related to an increase of `0.17611` in work satisfaction, whereas an increase of one unit in depersonalization and emotional exhaustion are related to a decrease of `0.07335` and `0.29215`. Of course, these are just estimates. The confidence intervals can be obtained using the `confint.merMod()` function:

```
confint.merMod(model)
```

The following output shows the true values with a 95 percent confidence. Examining whether the confidence intervals include 0 is another way of determining whether a predictor is significant. Notice that all predictors have confidence intervals that do not include 0 (meaning that they are significant). We will plot relationships between predictors and the criterion attribute in the next section. Note that the `.sig01` and `sigma` values refer to the standard deviations at level 2 (`.sig01`) and level 1 (`sigma`). Both are different from 0 (as 0 is not included in the confidence intervals):

	2.50%	97.50%
.sig01	0.1740378	0.39715380
sigma	0.7222870	0.79515650
(Intercept)	4.9773312	5.25975309
Accomp	0.1013788	0.25092248
Depers	-0.1350151	-0.01166813
Exhaust	-0.3498438	-0.23442925

Random intercepts and random slopes

In the previous model, we did consider common slopes for all hospitals. Now we want to draw conclusions on hospitals in general (the population), rather than on the hospitals in which we collected data. We therefore need to allow the slopes to vary between hospitals. Also, a visual inspection of the slopes in each hospital might warrant the inclusion of random slopes in the model in case of simple variations. This is the case in our data— we have presented the second figure here.

We, therefore, fit a new model with random slopes:

```
modelRS = lmer(WorkSat ~ Accomp + Depers + Exhaust +
   (1+Accomp+Depers+Exhaust|hosp), data=NursesMLtrain, REML = F)
```

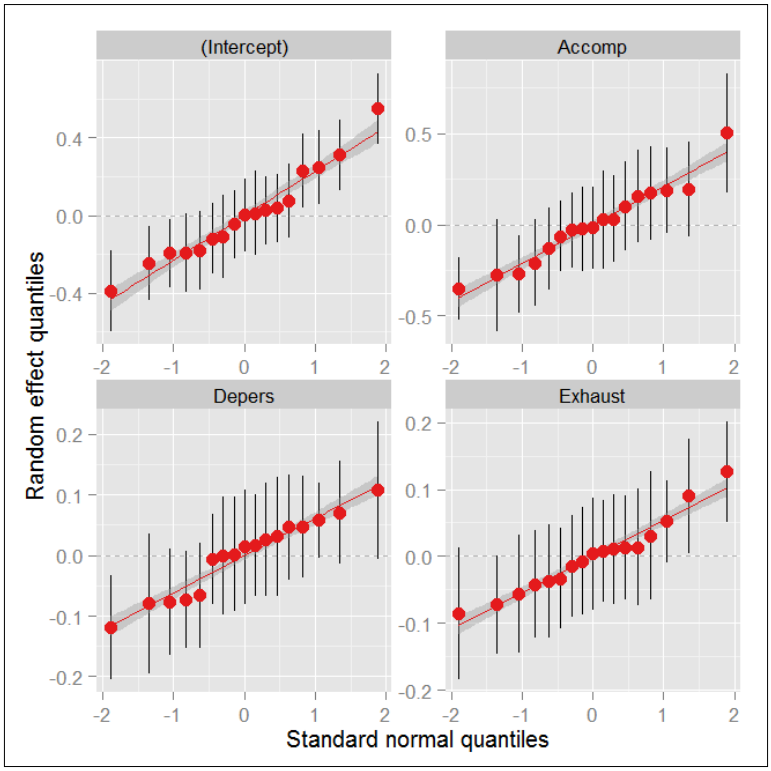
We compare this model to the null model:

```
anova(null, modelRS)
```

The output (not provided here) shows that our last model fits the data better than the null model. We now examine the level 2 residuals for normality using the `sjp.lmer()` function of the `sjPlot` package:

```
install.packages("sjPlot"); library(sjPlot)
sjp.lmer(modelRS, type = "re.qq")
```

As displayed in the following figure, the residuals for the intercept and each of the predictors are fairly normal, as almost points aligned to the normal distribution. Yet some deviation is observed, particularly for emotional exhaustion.

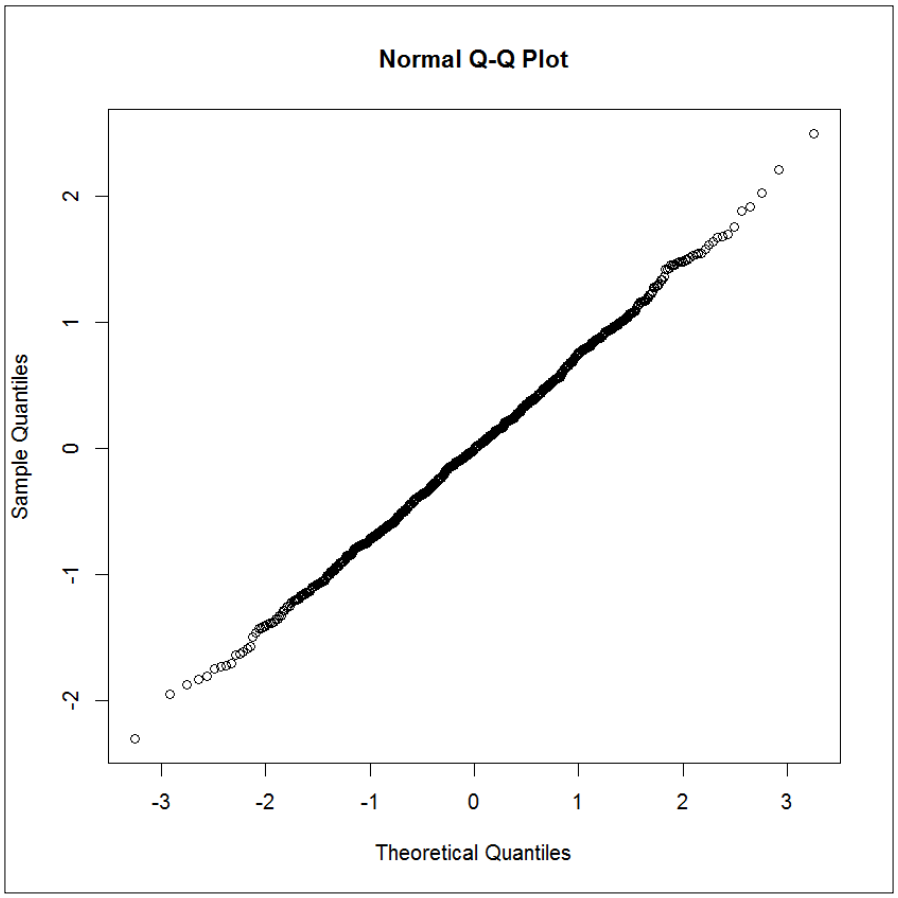


A QQ plot of level 2 residuals

We perform the same operation for the level 1 residuals using the `qqnorm()` function:

```
qqnorm(resid(modelRS))
```

The following screenshot shows that the level 1 residuals are fairly normal as well:



A QQ plot of level 1 residuals

Using the following code we observe that the variance explained by our model is around 19 percent:

```
r.squaredLR(model,null)
```

We can now examine our model in more detail:

```
summary(modelRS)
```

The output is as follows:

```
Linear mixed model fit by maximum likelihood  ['lmerMod']
Formula: WorkSat ~ Accomp + Depers + Exhaust + (1 + Accomp + Depers +
    Exhaust | hosp)
   Data: NursesMLtrain

     AIC      BIC   logLik deviance df.resid
  1970.5   2041.7   -970.2   1940.5      835

Scaled residuals:
    Min      1Q  Median      3Q     Max
-3.1422 -0.6826  0.0028  0.6510  3.4045

Random effects:
 Groups   Name        Variance Std.Dev. Corr
 hosp     (Intercept) 0.060492 0.24595
          Accomp      0.058726 0.24233  -0.56
          Depers      0.005805 0.07619  -0.17  0.83
          Exhaust     0.004594 0.06778   0.74 -0.24  0.34
 Residual             0.536849 0.73270
Number of obs: 850, groups:  hosp, 17

Fixed effects:
            Estimate Std. Error t value
(Intercept)  5.06994    0.06531   77.63
Accomp       0.21722    0.07044    3.08
Depers      -0.09408    0.03628   -2.59
Exhaust     -0.28444    0.03371   -8.44

Correlation of Fixed Effects:
        (Intr) Accomp Depers
Accomp  -0.448
Depers  -0.074  0.384
Exhaust  0.336 -0.073 -0.251
```

A summary of the random intercept and the slopes model

We notice that new values have appeared, notably the variance and standard deviations for the slopes of our three predictors at level 2, and the correlations between those (under `Random effects`). We notice that the coefficients (under `Fixed effects`) are also different.

Again, we test for the impact of the predictors on work satisfaction as follows:

```
tvals = coef(summary(modelRS))[,3]
tvals.p <- 2 * (1 - pnorm(abs(tvals)))
round(tvals.p,3)
```

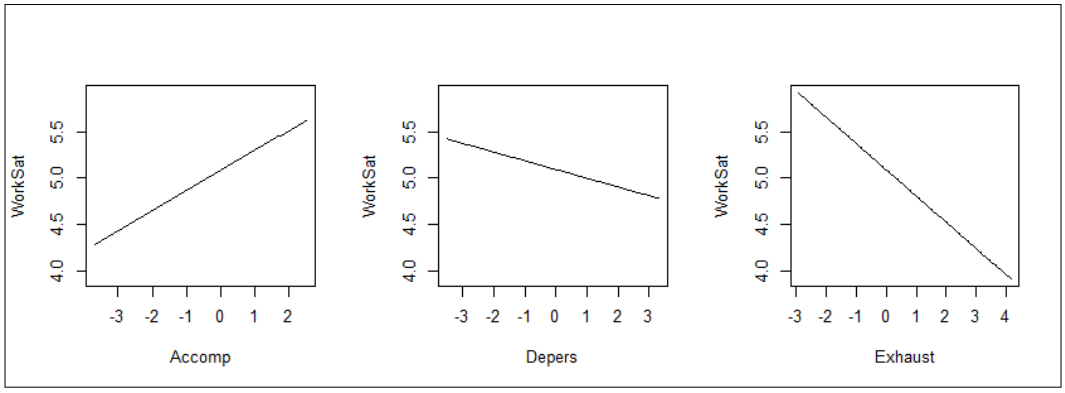
The following output shows that all three predictors are significant at `p < .05`:

(Intercept)	Accomp	Depers	Exhaust
0	0.002	0.010	0

We can also plot the slopes using the `plotLMER.fnc()` function from the `language` package:

```
install.packages("languageR"); library(languageR)
par(mfrow=c(1,3))
plotLMER.fnc(modelRS)
```

The three plots are presented as follows:



The relationship between the predictors and work satisfaction in the previous model

Remember that we centered our predictors. The 0 values on the *x* axis, therefore, refer to the mean of the predictors.

If you are curious about the impact of sampling on the estimates, you can run the following code and see what changes in the models we computed. You should find small differences each time you run the code! Just a heads up: use other model names as compared to those we used here. Otherwise, you will not find the same results in the following section:

```
#loading the initial dataset
NursesML = read.table("NursesML.dat", header = T,
   sep = " ")
NursesML$hosp = factor(NursesML$hosp)
#creating the training and testing sets (50% in each)
library(caret)
trainObs = createDataPartition(NursesML[,5], p = .5,
  list=F)
NursesMLtrain = NursesML[trainObs,]
NursesMLtest = NursesML[-trainObs,]

# grand mean centering the predictors
for (i in 1:3){
   NursesMLtrain[i] = NursesMLtrain[i]-
      colMeans(NursesMLtrain[i])
   NursesMLtest[i] = NursesMLtest[i]-
      colMeans(NursesMLtest[i])
}
```

Predictions using multilevel models

Now that we have our model ready, we can predict work satisfaction in the testing dataset.

Using the predict() function

One way to do so is simply to use the `predict()` function. The `allow.new.levels` argument specifies that we allow new hospitals in the analysis. As we have the same hospitals in the training and testing sets, we set its value to `F` (false) (which is actually the default value):

```
NursesMLtest$predicted = predict(modelRS, NursesMLtest,
   allow.new.levels = F)
```

Assessing prediction quality

There is no perfect way to measure the quality of the predictions for nested data. A simple estimate of the quality of our prediction is the correlation test. Because of the nested structure of our dataset, we will perform the test for each hospital separately:

```
1  correls = matrix(nrow=17,ncol=3)
2  colnames(correls) = c("Correlation", "p value", "r squared")
3  for (i in 1:17){
4     dat = subset(NursesMLtest, hosp == i)
5     correls[i,1] = cor.test(dat$predicted, dat$WorkSat)[[4]]
6     correls[i,2] = cor.test(dat$predicted, dat$WorkSat)[[3]]
7     correls[i,3] = correls[i,1]^2
8  }
9  round(correls, 3)
```

The output provided here shows some variation in the correlations, which are all significant at $p < .05$, except for hospital number 10. The third column displays the part of variance that is shared by the predictions and the observed values:

	Correlation	p value	r squared
[1,]	0.488	0.000	0.238
[2,]	0.282	0.047	0.080
[3,]	0.511	0.000	0.262
[4,]	0.481	0.000	0.232
[5,]	0.471	0.001	0.222
[6,]	0.342	0.015	0.117
[7,]	0.347	0.014	0.120
[8,]	0.385	0.006	0.148
[9,]	0.498	0.000	0.248
[10,]	0.203	0.157	0.041
[11,]	0.482	0.000	0.232
[12,]	0.459	0.001	0.211
[13,]	0.594	0.000	0.353
[14,]	0.358	0.011	0.128
[15,]	0.565	0.000	0.320
[16,]	0.442	0.001	0.195
[17,]	0.823	0.000	0.677

We can rely on multilevel analyses to test how well the predicted values are related to the observed values (the following model named `modelPred`), and compare it to a null model (model called `nullPred`). We will include random slopes, as the preceding correlations show some variation between hospitals in the data. Before we do that, we start by centering the predicted values:

```
NursesMLtest$predicted = NursesMLtest$predicted –
mean(NursesMLtest$predicted)
nullPred = lmer(WorkSat ~ 1 + (1|hosp), data=NursesMLtest,
REML = F)
modelPred = lmer(WorkSat ~ predicted + (1+predicted|hosp),
   data=NursesMLtest, REML = F)
```

The output of the following line of code shows that `modelPred` fits the data better than `nullPred`:

```
anova(nullPred,modelPred)
```

The output of the following line shows that 27.99 percent of the variance of work satisfaction is accounted for by the prediction, which is in line with the model performance in the testing set. Whether this value is good or bad depends on the context:

```
r.squaredLR(modelPred,nullPred)
```

Summary

In this chapter, we saw why it is necessary to use analyses that account for the structure of the data when dealing with nested data. We have examined how to fit several types of multilevel models and saw how to predict new data. In the next chapter, we will deal with text mining, including document classification.

13
Text Analytics with R

In the previous chapter, we examined how to deal with nested data using multilevel analyses. In *Chapter 11*, *Classifiation Trees* we discovered how to classify data using decision trees. Here, we will deal with textual data. This chapter will cover the following topics:

- A brief introduction to text analytics
- How to load and preprocess text
- How to perform document classification
- How to perform basic topic modeling to extract meaning
- How to download news articles using R

An introduction to text analytics

It might come as a surprise, or not, textual data represents the greatest part of the overall data accessible to companies and data analysts. Textual data is often available only in unstructured form. Imagine, for instance, an e-mail, a company memo, or a post on a blog. What they have in common is that text is mostly presented in the form of words arranged in sentences arranged in paragraphs. More complex documents are also composed of sub-sections, sections, and chapters. Humans derive meaning from this basic structure and the relationships between these elements. But for machines to classify documents and extract meaning, text preprocessing is required.

There are several usual steps in the preprocessing of textual documents for classification. These include:

1. Importing the corpus.
2. Converting text to lowercase, so that, in the analyses, words that include capital letters are not distinguished from words that do not. For instance, the following words are the same after converting to lowercase:
    - Documents
    - DOCUMENTS
    - documents
3. Removing punctuation so that words followed/preceded by punctuation signs are not treated differently compared to words that are not. For instance, the following words are the same after removing punctuation:
    - documents.
    - documents:
    - documents
4. Removing numbers contained in the text. Text often includes numbers. These can interfere with the analyses (as they are considered as words). It is, therefore, useful to discard numbers.
5. Stop word filtering, which is the suppression of uninformative words (contained in most documents), such as:
    - it
    - me
    - where
    - some
6. Removing extra whitespaces occurring in the original text or resulting from the previous operations.
7. Performing stemming. Textual analysis often requires replacing words with their stem. The following words are the same after stemming:
    - documentation
    - documented
    - documents
    - document

8. Performing other necessary transformations, which we will examine in the next section. It is important to note that the task of the analyst is to determine which of the preceding steps are necessary for a particular analysis. Additional steps include:
    - **Tokenizing** and building the **term-document matrix**. In this step, each unit (usually the unit is a single word, but can also be an n-gram, that is, two or more contiguous words) is assigned to a column. Documents are presented in rows. The cells (the intersection of columns and words) represent either the presence/absence of each word in each document, the count of each word in each document, or the term frequency–inverse document frequency (the tf–idf measure). This measure takes into account the rarity of the words in the whole corpus (scarcer words are more informative than common words). We discussed how to compute the measure in *Chapter 4*, *Cluster Analysis*.
    - Pruning rare tokens, which is the suppression of words that occur infrequently in the corpus.

Once these steps are performed, it is possible to analyze term associations and perform document classification on the basis of the term-document matrix in an efficient way, using algorithms we have already discovered and others we will discover here.

In the following sections, we will examine how to perform text analytics with R, focusing on classification and the extraction of meaning.

Loading the corpus

Before we start, let's perform some preliminary steps by running the following code:

```
1  URL = "http://www.cs.cornell.edu/people/pabo/
2  movie-review-data/review_polarity.tar.gz"
3  download.file(URL,destfile = "reviews.tar.gz")
4  untar("reviews.tar.gz")
```

This downloads the data you will use in a compressed file. Line 1 and 2 here should be typed on the same line in your console or script window with nospace between the quotation marks. Next, the file is uncompressed in a folder called `txt_sentoken` in your working directory. Change your working directory to point to this folder by using the following code line:

```
setwd("txt_sentoken")
```

The folder contains the subfolders `pos` and `neg`. The `pos` folder contains 1,000 positive film reviews, whereas the `neg` folder contains 1,000 negative film reviews. The reviews were collected by researchers at Cornell University. We will analyze these texts here. The first thing we will do is load both corpora into R.

For this purpose, and to accomplish most of the tasks we will deal with here, we will download and load the `tm` package:

```
install.packages("tm"); library(tm)
```

We will now load the two corpora separately into R—the first corpus containing the positive reviews followed by the corpus containing the negative reviews. The `pattern="cv"` argument allows us to specify that we only want to load the files that contain `cv` in their name:

```
1  SourcePos = DirSource(file.path(".", "pos"), pattern="cv")
2  SourceNeg = DirSource(file.path(".", "neg"), pattern="cv")
3  pos = Corpus(SourcePos)
4  neg = Corpus(SourceNeg)
```

There are other ways to load a corpus. We will not list them all, but we will provide the most common ones here so that you're all set if you wish to use other formats.

If we were using a data frame source or a vector source for the positive examples, instead of using `DirSource`, we would have written (do not run this code now):

```
Pos = DataframeSource(theDataframeName) # for a data frame corpus
Pos = VectorSource(theVectorName) #for a vector corpus
```

Both types of corpora can simply be created from CSV files.

If we were using PDF files (which would require installing the application *xpdf*), or word documents (which would require the application *antiword*), we would have written (do not run this code now either):

```
SourcePos = DirSource(file.path(".", "pos"),
  readerControl=list(reader=readPDF) # for pdf files
SourcePos = DirSource(file.path(".", "pos"),
  readerControl=list(reader=readDOC) # for word files
```

There are other sources available. For a full list, type:

```
getSources(); getReaders()
```

Going back to our analysis, we can check if our corpora have been loaded correctly:

```
pos
```

The following output shows that the positive reviews have been loaded correctly:

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 1000
```

Let's examine the corpus of negative reviews:

```
neg
```

The following output shows the negative reviews have been loaded correctly:

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 1000
```

We can now append the corpora and check whether this operation worked as well (it did, as displayed in the following code). Remember, the first 1,000 reviews are the positive ones and the other 1,000 reviews are the negative ones:

```
reviews = c(pos, neg)
reviews
```

The output shows we do have 2,000 cases here:

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 2000
```

We can see that the joint corpus contains 2,000 documents as we requested

Data preparation

In this section, we will start by preprocessing the corpus for analysis and then inspecting it. We will then build the training and testing data frames.

Preprocessing and inspecting the corpus

We can see that the joint corpus contains 2,000 documents as we requested. We can now perform the steps we discussed in the preceding section. We will build a function that performs them all at once for this purpose (we will use this function again later in the chapter):

```
1  install.packages("SnowballC")
2  preprocess = function(corpus, stopwrds =
3     stopwords("english")){
```

```
    library(SnowballC)
    corpus = tm_map(corpus, content_transformer(tolower))
    corpus = tm_map(corpus, removePunctuation)
    corpus = tm_map(corpus,
    content_transformer(removeNumbers))
    corpus = tm_map(corpus, removeWords, stopwrds)
     corpus = tm_map(corpus, stripWhitespace)
     corpus = tm_map(corpus, stemDocument)
     corpus
}
```

Let's run the function on our corpus:

```
processed = preprocess(reviews)
```

Now that our corpus is preprocessed, we can create the term-document matrix. We can use the following code to generate the term-document matrix with frequencies in the cells:

```
term_documentFreq = TermDocumentMatrix(processed)
```

Using this term-document matrix, we can examine, for instance, the five most frequent terms in the corpus:

```
asMatrix = t(as.matrix(term_documentFreq))
Frequencies = colSums(asMatrix)
head(Frequencies[order(Frequencies, decreasing=T)], 5)
```

This is the output:

```
film  movi  one  like  charact
11109  6857  5759  3998  3855
```

The following code will show which terms occur more than 3,000 times:

```
Frequencies[Frequencies > 3000]
```

This will generate the following output:

```
charact  film  get  like  make  movi  one
3855  11109  3189  3998  3152  6857  5759
```

By making a little change to the matrix, we can also examine words that occur in most documents. We start by creating a new matrix in which terms are either present (1) or absent (0)–this can take some time:

```
Present = data.frame(asMatrix)
Present [Present>0] = 1
```

The following code will display the frequency of the five terms with higher document frequency:

```
DocFrequencies = colSums(Present)
head(DocFrequencies[order(DocFrequencies, decreasing=T)], 5)
```

This will provide the following output:

```
film  one  movi  like  charact
1797  1763  1642  1538  1431
```

The following code will show which terms occur in more than 1,400 documents:

```
DocFrequencies[DocFrequencies > 1400]
```

This will result in the following output:

```
charact  film  like  make  movi  one
1431  1797  1538  1430  1642  1763
```

We can simply compute the number of terms that occur more than once in the corpus and their proportion relative to all the terms by:

```
1  total = ncol(asMatrix)
2  moreThanOnce = sum(DocFrequencies != Frequencies)
3  prop = moreThanOnce / total
4  moreThanOnce
5  total
6  prop
```

The output shows that the total number of terms is 30,585. Within these, 9,748 terms occur more than once, which is 31.9 percent of the terms.

Our interest is in performing prediction on unseen cases. The tf–idf measure is more meaningful for such tasks, as it increases the weights of terms that occur in many documents, thereby making the classification more reliable. Therefore, we will use it for what comes next instead of raw frequencies in a new term-document matrix:

```
term_documentTfIdf= TermDocumentMatrix(processed,
control = list(weighting = function(x) weightTfIdf(x,
normalize = TRUE)))
```

We have just seen that there are plenty of infrequent terms (about two-thirds of these terms occur only once). Infrequent terms might degrade performance in classification. We will remove sparse terms from the matrix by using the `removeSparseTerms()` function. This function has an argument called sparse, which allows a limit to be set for the degree of sparsity of the terms. A sparsity of `0` means that all documents must contain the term, whereas a sparsity of `1` means that none contain the term. We use a value higher than 0.8 to filter out most terms but still have enough terms to perform the analysis using the following code:

```
SparseRemoved = as.matrix(t(removeSparseTerms(
term_documentTfIdf, sparse = 0.8)))
```

Let's examine how many terms are now included in the term-document matrix:

```
ncol(SparseRemoved)
```

The output shows that there are now only `202` terms remaining. We will now check if all the documents contain the remaining terms by examining if the documents have a value of `0` as the total of all terms (that is, they contain no term):

```
sum(rowSums(as.matrix(SparseRemoved)) == 0)
```

The output is `0`, which means that all of the documents contain at least 1 remaining term.

Before we continue with the classification, you can, if you wish, examine the list of terms by using the following (not reproduced here):

```
colnames(SparseRemoved)
```

Computing new attributes

Now, we will use these 202 terms to classify our documents based on whether the reviews are positive or negative. Remember that the rows 1 to 1,000 represent positive reviews, and rows 1,001 to 2,000 negative ones. We will now create a vector that reflects this:

```
quality = c(rep(1,1000),rep(0,1000))
```

The length of the reviews may be related to their positivity or negativity. We will, therefore, also include an attribute that reflects review length in the processed corpus (before the removal of sparse terms):

```
lengths = colSums(as.matrix(TermDocumentMatrix(processed)))
```

Creating the training and testing data frames

We now need to create a data frame that includes the criterion attribute (`quality`), the length of the reviews, and the term-document matrix:

```
DF = as.data.frame(cbind(quality, lengths, SparseRemoved))
```

Let's now create our training and a testing dataset:

```
1  set.seed(123)
2  train = sample(1:2000,1000)
3  TrainDF = DF[train,]
4  TestDF = DF[-train,]
```

Classification of the reviews

At the beginning of this section, we will try to classify the corpus using algorithms we have already discussed (Naïve Bayes and k-NN). We will then briefly discuss two new algorithms: logistic regression and support vector machines.

Document classification with k-NN

We know k-Nearest Neighbors, so we'll just jump into the classification. We will try with three neighbors and five neighbors:

```
1  library(class) # knn() is in the class packages
2  library(caret) # confusionMatrix is in the caret package
```

```
3  set.seed(975)
4  Class3n = knn(TrainDF[,-1], TrainDF[,-1], TrainDF[,1], k = 3)
5  Class5n = knn(TrainDF[,-1], TrainDF[,-1], TrainDF[,1], k = 5)
6  confusionMatrix(Class3n,as.factor(TrainDF$quality))
```

The confusion matrix and the following statistics (the output has been partially reproduced) show that classification with three neighbors doesn't seem too bad: the accuracy is 0.74; yet, the kappa value is not good (it should be at least 0.60):

```
Confusion Matrix and Statistics


Reference
Prediction    0    1
         0  358  126
         1  134  382


Accuracy : 0.74
95% CI : (0.7116, 0.7669)
No Information Rate : 0.508
P-Value [Acc > NIR] : <2e-16


Kappa : 0.4876
```

Let's examine the solution with five neighbors:

```
confusionMatrix(Class5n,as.factor(TrainDF$quality))
```

The output shows that there is not much difference between the three neighbor and five neighbor solutions, but five neighbors is worse:

```
Confusion Matrix and Statistics
Reference
Prediction    0    1
         0  358  126
         1  134  382
Accuracy : 0.682
95% CI : (0.6521, 0.7108)
```

```
No Information Rate : 0.508
P-Value [Acc > NIR] : <2e-16
Kappa : 0.364
```

Further, we have only looked at the training dataset. How well would things go with the testing data set? We'll only take a look at the three neighbors solution:

```
set.seed(975)
Class3nTest = knn(TrainDF[,-1], TestDF[,-1], TrainDF[,1], k = 3)
confusionMatrix(Class3nTest,as.factor(TestDF$quality))
```

The following output shows that the classification is pretty bad on the testing dataset; the accuracy is just about what we would expect by attributing all attributes to one class and the kappa value is about 0, showing that there is no improvement over classification by chance:

```
Confusion Matrix and Statistics

Reference
Prediction    0    1
         0  235  226
         1  273  266

Accuracy : 0.501
95% CI : (0.4695, 0.5324)
No Information Rate : 0.508
P-Value [Acc > NIR] : 0.68243

Kappa : 0.032
```

Maybe we will have more luck using Naïve Bayes. Let's see!

Document classification with Naïve Bayes

Let's start by computing the model, following which we will try to classify the training dataset:

```
1  library(e1071)
2  set.seed(345)
3  model <- naiveBayes(TrainDF[-1], as.factor(TrainDF[[1]]))
4  classifNB = predict(model, TrainDF[,-1])
5  confusionMatrix(as.factor(TrainDF$quality),classifNB)
```

The partial output here presents the confusion matrix for the training dataset and some performance information. We can see that the classification is not too bad with regard to accuracy, yet, the kappa value is too low (it should be at least 0.60):

```
Confusion Matrix and Statistics


Reference
Prediction    0    1
         0  353  139
         1   74  434


Accuracy : 0.787
95% CI : (0.7603, 0.812)
No Information Rate : 0.573
P-Value [Acc > NIR] : < 2e-16


Kappa : 0.573
```

Let's examine how well we can classify the test dataset using the model we just computed:

```
classifNB = predict(model, TestDF[,-1])
confusionMatrix(as.factor(TestDF$quality),classifNB)
```

The following output shows that the results on the testing data are still disappointing; the accuracy has dropped to 71 percent and the kappa value is quite bad:

```
Confusion Matrix and Statistics


Reference
Prediction    0    1
         0  335  173
         1  120  372


Accuracy : 0.707
95% CI : (0.6777, 0.7351)
No Information Rate : 0.545
P-Value [Acc > NIR] : < 2e-16


Kappa : 0.4148
```

Classification using logistic regression

We can check the association between review length and quality using logistic regression as a quick tutorial. There is sadly no space to explain logistic regression in detail. Let's simply say that logistic regression predicts the probability of an outcome rather than a value, as in linear regression. We will only give an interpretation of the results, but first let's compute the model:

```
model = glm(quality~ lengths, family = binomial)
summary(model)
```

The following is the output of this model:

Call:
glm(formula = quality ~ lengths, family = binomial)

Deviance Residuals:

Min	1Q	Median	3Q	Max
-1.7059	-1.1471	-0.1909	1.1784	1.398

Coefficients:

	Estimate	Std. Error	z value	Pr(>|z|)	
(Intercept)	-0.6383373	0.1171536	-5.449	5.07E-08	***
lengths	0.0018276	0.0003113	5.871	4.32E-09	***

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

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 2772.6 on 1999 degrees of freedom
Residual deviance: 2736.4 on 1998 degrees of freedom
AIC: 2740.4

Number of Fisher Scoring iterations: 4

The output shows that the length of the review is significantly associated with the perceived quality of the movie. The estimates are given in log odds, so we must perform an exponentiation to know what the slope means:

```
exp(0.0018276)
```

The output is `1.001828`. A value of exactly 1 would mean there is no relationship between the length of the review and the perceived quality of the movie. Here, the value is slightly higher than 1, but remember that the unit is the term in the processed review. The intercept represents the odds that a movie with a review containing 0 terms is considered good. It has therefore no interpretable meaning here. The value of the exponentiated slope means the odds of a movie being considered good increases by 0.01829 percent for each additional term. Let's compute the fitted values in terms of probability and examine the relationship by plotting them.

We obtain the probabilities of movies being classified as good for each review using either of the following lines of code:

```
Prob1 = exp(-0.6383373 + lengths * 0.0018276) /
   (1 + exp(-0.6383373 + lengths * 0.0018276))
Prob2 = model$fitted
```

The attribute `Prob1` is the value computed manually, whereas the attribute `Prob2` contains the values computed by the `glm()` function. Both attributes have the same value up to the fourth decimal.

We can classify documents with a probability higher than 0.5 as positive reviews and display the confusion matrix:

```
1  classif = Prob1
2  classif[classif>0.5] = 1
3  classif[classif<=0.5] = 0
4  table(classif, quality)
```

The following output shows that more instances are correctly classified than incorrectly classified, but that many instances are not correctly classified:

	quality	
classif	**0**	**1**
0	**614**	**507**
1	**386**	**493**

The output of the following line of code shows that the kappa value is only 0.11, which is pretty bad:

```
cohen.kappa(table(classif, quality))
```

Now that the basics of logistic regression are understood, let's get to business. Can we obtain a reliable classification using logistic regression by including the linguistic content of the review in the analysis (the 100 terms)? We will attempt this here:

```
model2 = glm(quality ~ ., family = binomial, data = TrainDF)
```

In *Linear Regression*, we warned you about multicollinearity: when predictors are strongly correlated, the reliability of the estimates (the slopes coefficients) is threatened because of the assumption of the independence of the predictors. But be assured, this is not a problem for the predictive power of a model; the adjusted coefficient of determination (the value-adjusted R squared we discussed in that chapter) is unaffected by multicollinearity. In other words, in case of multicollinearity, we cannot disentangle the contribution of the separate predictors, but we can accurately know their contribution together. Including so many predictors is therefore not much of a problem because we are not interested in the slope coefficients but only the predictions.

Let's now examine the fitted values in the training set and how well we can predict perceived movie quality:

```
TrainDF$classif = fitted.values(model2, type= "response")
TrainDF$classif[TrainDF$classif>0.5] = 1
TrainDF$classif[TrainDF$classif<=0.5] = 0
```

Let's now examine the performance of our classification in detail. We will omit the output and briefly show the results:

```
confusionMatrix(TrainDF$quality, TrainDF$classif)
```

We can see that we have a 0.857 percent accuracy (which is meaningful given we have an almost equal number of positive and negative reviews) and a kappa value of 0.71. We can therefore be satisfied by our classification.

We can now use the model we just created to predict the values in the testing set:

```
1  TestDF$classif = predict(model2, TestDF, type = "response")
2  TestDF$classif[TestDF$classif>0.5] = 1
3  TestDF$classif[TestDF$classif<=0.5] = 0
4  confusionMatrix(TestDF$quality, TestDF$classif)
```

As you can see on your screen, the predictions in the testing dataset are poorer than what we observed in the training dataset. We still have an accuracy of 0.72, which is not too bad, but the kappa value has dropped to 0.45. We might obtain better results using another algorithm we have yet to discover: support vector machines.

Document classification with support vector machines

Support vector machines (**SVM**) attempt to find a separation between the two classes that is as broad as possible. Cases are then classified depending on their position in the separations. Unlike logistic regression, SVMs are not limited to linear relationships. Actually, any kind of relationship can be discovered with SVM by using the kernel trick. We will not go into detailed explanations of SVM here, as these are quite complex. The interested reader can refer to the book, *Learning with kernels: Support Vector Machines, Regularization, Optimization, and Beyond*, by Scholkopf and Smola (2001).

Let's directly fit the model using SVM and examine the reliability of the predictions:

```
1    library(e1071)
2    modelSVM = svm (quality ~ ., data = TrainDF)
3    probSVMtrain = predict(modelSVM, TrainDF[,-1])
4    classifSVMtrain = probSVMtrain
5    classifSVMtrain[classifSVMtrain>0.5] = 1
6    classifSVMtrain[classifSVMtrain<=0.5] = 0
7    confusionMatrix(TrainDF$quality, classifSVMtrain)
```

We have an excellent classification using SVM on the training dataset. It is even better than with logistic regression: the accuracy is 0.93 and the kappa value is 0.85.

What performance will we get for the classification using the testing set? Let's find out:

```
1    probSVMtest = predict(modelSVM, TestDF[,-1])
2    classifSVMtest = probSVMtest
```

```
3    classifSVMtest[classifSVMtest>0.5] = 1
4    classifSVMtest[classifSVMtest<=0.5] = 0
5    confusionMatrix(TestDF$quality, classifSVMtest)
```

Sadly, our classification of the testing set was not better than with logistic regression (it was even a little worse). Depending on the context, the performance of the algorithm (here, 71 percent of data was correctly classified, kappa = 0.42) might be sufficient. In other contexts, much better performance might be required.

We have shown you different alternatives so that you can try them on your data. We didn't have much luck with our classification of the reviews on the testing dataset here. This really depends a lot on the dataset. Before you get too disappointed by document classification, let's examine a successful case.

Mining the news with R

In this section, we discuss news mining in R. We start with a successful document classification and then discuss how to collect news articles directly from R.

A successful document classification

In this section, we examine a particular dataset which features a term-document matrix of 2,071 press articles containing the word *flu* in their title. The articles were found on LexisNexis using this search term in two newspapers, *The New York Times* and *The Guardian*, between January 1980 and May 2013. For copyright reasons, we cannot include the original articles here. These have been preprocessed in a similar way to what we have seen before with another software, Rapidminer 5. In addition to the term-document matrix, the type of seasonal flu versus other (avian and swine flu)–is included in the first column of the data frame (the `SEASONAL.FLU` attribute). When articles discussed seasonal flu and other strands, they were coded as other (value 0). Terms were coded as present (1) or absent (0) in each article. Let's load the dataset (make sure it is in your working directory):

```
Strands = read.csv("StrandsPackt.csv", header = T)
```

Let's examine how many articles about seasonal flu we have here:

```
colSums(Strands[1])
```

The output is `777`.

Our task here will be to predict whether the flu discussed in the article is seasonal flu or another type. Before we do this, we can examine the most frequent terms associated with each class. Remember, the first column is the class, which we exclude from the document frequencies:

```
Seasonal = subset(Strands, SEASONAL.FLU == 1)
FreqSeasonal = colSums(Seasonal)[-1]
head(FreqSeasonal[order(FreqSeasonal, decreasing=T)], 20)
```

The following output shows the top 20 terms in document frequency in the articles that discuss seasonal flu:

```
    year     peopl    vaccin influenza
     513       460       431       380
  diseas       com      http       www
     362       359       357       357
  nytim      week       get      time
     356       354       340       324
  state       url    season    report
     314       311       302       295
   viru   control      risk    center
     295       284       281       272
```

Let's examine other strands:

```
Other = subset(Strands, SEASONAL.FLU == 0)
FreqOther = colSums(Strands[Strands[,1] == 0])[-1]
head(FreqOther[order(FreqOther, decreasing=T)], 20)
```

The output of these strands follows. One can easily see that terms with most document frequency are different between the classes. Yet, there are some terms featured in both such as `influenza`, `vaccine`, and `viru`. The low overlapping of the most frequent terms can give us hope that we will be able to classify the instances:

```
    peopl      viru       com       www
     1449      1281      1108      1108
     case     nytim       url    vaccin
     1104      1091      1044      1043
   offici    infect    report influenza
     1008      1000       975       964
     time  outbreak     world    strain
```

```
    918       915       867       864
govern   countri       dai       sai
    845       836       835       828
```

What we didn't mention before is that there are thousands of terms included in the term-document matrix here. We are faced with an example of the curse of dimensionality as logistic regression cannot handle more features than cases. We could use more complex analyses, such as ridge logistic regression. But let's be creative! We are going to use logistic regression by repeating the analysis say 100 times, with each time a different random set of predictors, say 300. We will aggregate the resulting probabilities, which will help us determine the class of our articles. This is known as ensemble learning, which we have already discovered when discussing random forests. The usefulness of ensemble learning for strong learners, such as logistic regression, is under debate, yet we will see that this works pretty well here.

We will first determine training instances and build matrices to store the predictions in the training and testing sets:

```
1    set.seed(1234)
2    TrainCases = sample(1:nrow(Strands),1086)
3    TrainPredictions = matrix(ncol=1086,nrow=100)
4    TestPredictions = matrix(ncol=1085,nrow=100)
```

We will now run the 100 logistic regression analyses. We will first determine which terms to exclude (we only want 300 terms), and then assign the columns we wish to keep (the class and the 300 terms). From these, we will create a training set and a testing set. We will then fit the model and assign the predictions for the testing and training sets for the current iteration in the respective matrices. Be aware that running the loop will take some time:

```
1    for (i in 1:100) {
2       UNUSED = sample(2:ncol(Strands), ncol(Strands)-300)
3       Strands2 = Strands[,-UNUSED]
4       StrandsTrain = Strands2[TrainCases,]
5       StrandsTest = Strands2[-TrainCases,]
6       model = glm(StrandsTrain$SEASONAL.FLU~.,
7          data = StrandsTrain, family="binomial")
8       TrainPredictions[i,] = t(predict(model,StrandsTrain,
9          type="response"))
10       TestPredictions[i,] = t(predict(model,StrandsTest,
11          type="response"))
12    }
```

Latest versions of R display warning messages: `fitted probabilities numerically 0 or 1 occurred`. The inspection of the estimates (not displayed) show this is not a problem here. But this might be a concern sometimes. It is the analyst's job to check that everything works fine. In the present case, this happened because some terms were never used in each of the classes, leading to infinitely small or large odds ratios.

Now that we have this ready, we will compute the mean of the probability predictions for each article in the training and testing datasets and assign a classification:

```
1    PredsTrain = colMeans(TrainPredictions)
2    PredsTrain[PredsTrain< .5] = 0
3    PredsTrain[PredsTrain>= .5] = 1
4    PredsTest = colMeans(TestPredictions)
5    PredsTest[PredsTest< .5] = 0
6    PredsTest[PredsTest>= .5] = 1
```

Let's first examine the reliability of the classification on the training set:

```
confusionMatrix(PredsTrain,StrandsTrain$SEASONAL.FLU)
```

The following partial output shows that the classification was almost perfect; the accuracy is above 99 percent and the kappa value is about 0.98:

```
Confusion Matrix and Statistics
Reference
Prediction     0    1
         0   689    5
         1     1  391

Accuracy : 0.9945
95% CI : (0.988, 0.998)
No Information Rate : 0.6354
P-Value [Acc > NIR] : <2e-16

Kappa : 0.9881
```

We can be very happy about the classification of the training set. But what about the testing set? Let's have a look:

```
confusionMatrix(PredsTest,StrandsTest$SEASONAL.FLU)
```

The output shows that things also went quite well in the training set; more than 86% of cases are correctly classified, and the kappa value is 0.69, which is above the 0.65 threshold we fixed. Of course, as in most cases, the classification of the training set had better reliability.

```
Confusion Matrix and Statistics


Reference
Prediction    0    1
         0  655   98
         1   49  283

Accuracy : 0.8645
95% CI : (0.8427, 0.8843)
No Information Rate : 0.6488
P-Value [Acc > NIR] : < 2.2e-16

Kappa : 0.6936
```

Classifying documents can be tricky and depends as much on your data as on how imaginative you are in circumventing the limitations of the algorithms you use.

Extracting the topics of the articles

We have seen that we can reliably classify whether the articles discuss seasonal flu or not. It is therefore likely that the topics discussed in the articles are different. Wouldn't it be nice to be able to get a better understanding of this in a straightforward manner? It turns out we can!

We first separate the two types of articles (discussing `Seasonal` versus `Non.Seasonal` flu) and remove the attribute that makes this distinction:

```
Seasonal = subset(Strands,SEASONAL.FLU ==1)[,-1]
Non.Seasonal = subset(Strands,SEASONAL.FLU ==0)[,-1]
```

We will need to convert these data frames to a `tm` document-term matrix. The `as.wfm()` and `as.dtm()` functions of the `qdap` package will help us do that:

```
install.packages("qdap"); library(qdap)
seasonal.tm <- as.dtm(as.wfm(t(Seasonal)))
non.seasonal.tm <- as.dtm(as.wfm(t(Non.Seasonal)))
```

We also need to install the `topicmodels` package and load it:

```
install.packages("topicmodels"); library(topicmodels)
```

We will skip the explanations of the inner working of the algorithms for lack of space and simply perform a basic analysis with default parameters.

For each document-term matrix, we will arbitrarily generate two topics. For more information about how many topics to include, the interested reader can refer to the paper *How many topics? Stability analysis for topic models*, by Greene and colleagues (2014).

```
Topics.seasonal = LDA(seasonal.tm, 2)
Topics.non.seasonal = LDA(non.seasonal.tm, 2)
```

We can now examine the top 10 terms associated with each of the topics in both document-term matrices:

```
1    Terms.seasonal = terms(Topics.seasonal, 20)
2    Terms.non.seasonal = terms(Topics.non.seasonal, 20)
3    Terms.seasonal
4    Terms.non.seasonal
```

The output of both document-term matrices is displayed side by side in the following screenshot:

```
> Terms.seasonal
      Topic 1     Topic 2
 [1,] "vaccin"    "year"
 [2,] "state"     "peopl"
 [3,] "get"       "influenza"
 [4,] "com"       "center"
 [5,] "dai"       "time"
 [6,] "month"     "diseas"
 [7,] "million"   "strain"
 [8,] "viru"      "doctor"
 [9,] "yesterdai" "case"
[10,] "report"    "nytim"
[11,] "week"      "winter"
[12,] "shot"      "week"
[13,] "feder"     "protect"
[14,] "drug"      "url"
[15,] "nytim"     "control"
[16,] "death"     "sai"
[17,] "problem"   "nation"
[18,] "countri"   "recommend"
[19,] "season"    "ag"
[20,] "expect"    "risk"
```

```
> Terms.non.seasonal
      Topic 1   Topic 2
 [1,] "nytim"   "infect"
 [2,] "offici"  "diseas"
 [3,] "viru"    "year"
 [4,] "world"   "week"
 [5,] "million" "peopl"
 [6,] "prevent" "outbreak"
 [7,] "nation"  "pandem"
 [8,] "work"    "influenza"
 [9,] "come"    "sai"
[10,] "confirm" "countri"
[11,] "unit"    "case"
[12,] "peopl"   "com"
[13,] "found"   "human"
[14,] "strain"  "url"
[15,] "month"   "di"
[16,] "case"    "test"
[17,] "start"   "spread"
[18,] "get"     "govern"
[19,] "report"  "includ"
[20,] "effect"  "viru"
```

Terms associated with the topics in the seasonal and non-seasonal flu articles

For seasonal flu, Topic 1 is associated with terms related to seasonality, prevention in a concrete sense (for example, vaccin or shot), and counts of cases of the flu. Topic 2 is also associated with terms regarding seasonality, as well as with terms regarding the action that can be taken in a more abstract sense (for example, protect or recommend) with regard to the flu. In the non-seasonal articles, both topics are related to the spread of the virus, but the first topic discusses reports on the virus, including its strain, the number of cases, and prevention. The second topic focuses on the spread of the virus with terms such as outbreak, pandem, and spread. There are some similarities and differences in the articles on both seasonal and non-seasonal flu strands.

Collecting news articles in R from the New York Times article search API

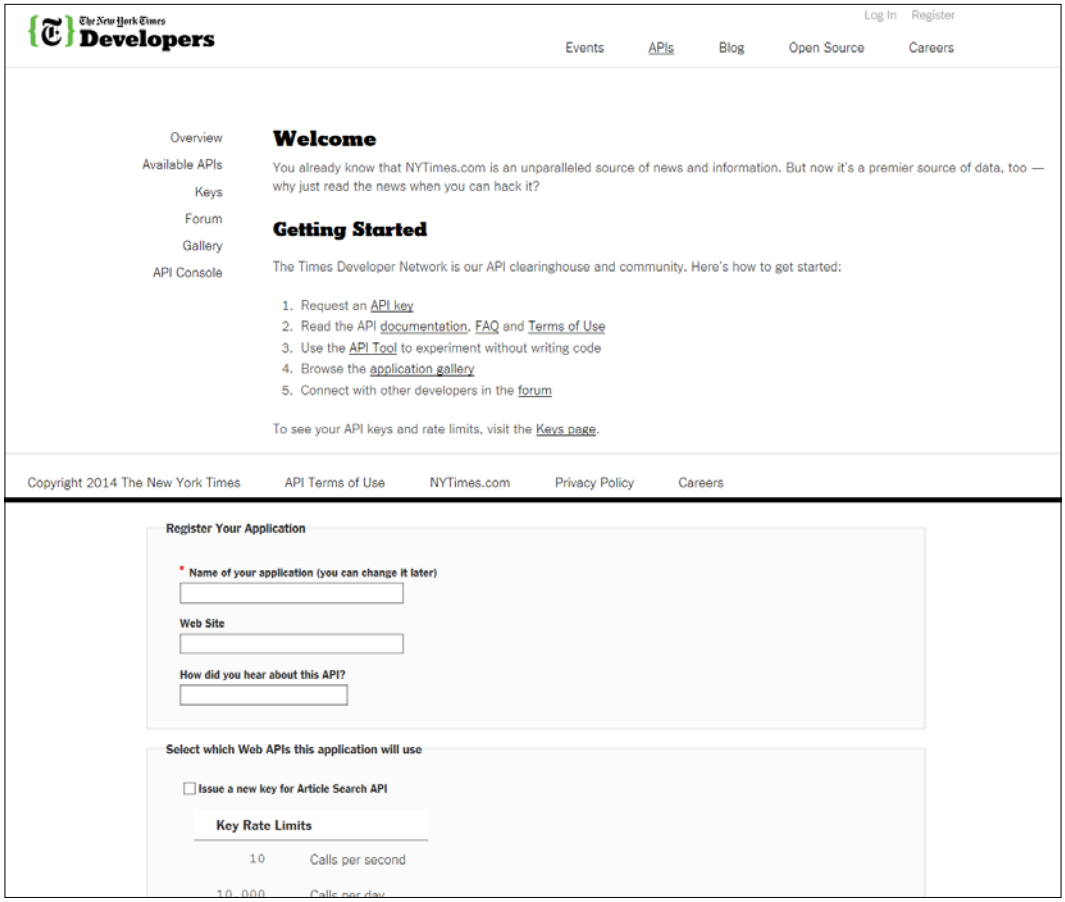
Searching the news and comparing information can take a huge amount of time. In this section, we will show how to simplify the process using R. First, of course, we need to download the recent news related to our interest. The fall of the Euro is currently much debated. What is the current news on the topic? How can we use text mining to learn more about the topic? Let's find out!

The first step is to install and load the `tm.plugin.webmining` package that will help us in this task:

```
install.packages("tm.plugin.webmining")
library(tm.plugin.webmining)
```

The package can download news from several generic sources such as *Yahoo News*, *Google News*, *The New York Times News*, and *Reuters News*. Due to space restrictions, we will here examine the news from *The New York Times* only.

First, we need to obtain a developer key from `http://developer.nytimes.com/`. Click on **1 Request an API key** (see the top panel of the following screenshot). You will first have to register on the website and sign in. After this, you will reach the API key registration page (see the bottom panel of the screenshot). Simply enter your details and select **Issue a new key for Article search API**. Your key will be displayed on the next screen. Be sure to keep it safe. You are all set.



Registering a key on the NY Times article search API

We start by downloading the news related to the Euro. The following code will download 100 articles:

```
1    nytimes_appid = "YOUR_KEY_HERE"
2    NYtimesNews <- WebCorpus(NYTimesSource("Euro",
3       appid = nytimes_appid))
4    NYtimesNews
```

The output shows that 100 articles were downloaded, as set by default:

```
<<WebCorpus>>
Metadata:  corpus specific: 3, document level (indexed): 0
Content:  documents: 100
```

You might want to save the content of the articles. This can be done by simply using the `writeCorpus()` function. The following code will save individual text files to the working directory:

```
writeCorpus(NYtimesNews, path = "M:/")
```

There are cases in which you might want to add the newly published articles to your corpus. This can be done using the `corpus.update()` function. Here, the following line of code could be used to do this once:

```
updated = corpus.update(NYtimesNews)
```

We can now preprocess the text as we did before:

```
preprocessedNews = preprocess(NYtimesNews)
```

We also build the term-document matrix:

```
tdmNews = TermDocumentMatrix(preprocessedNews)
```

I saved the term-document matrix so that you can do the following with me and get the same results (results with your corpus will differ as you have not retrieved the same articles as I did). In order to load your file, type the following:

```
loaded_tdm = dget("tdmNews")
```

We can inspect the most frequent terms (those that occur over 100 times) like this:

```
findFreqTerms(loaded_tdm, low = 100)
```

The following output shows the list of terms that satisfy the query. We can notably see that Europe, Germany, and Greece are mentioned as top terms. Other mentions related to finance are also present such as `bank`, `debt`, and `currency`:

bank	**countri**	**currenc**	**debt**	**econom**	**euro**	**europ**
european	**germani**	**govern**	**greec**	**greek**	**minist**	**new**
percent	**said**	**union**	**will**	**zone**		

What are the word associations we can observe in the data? Let's focus on the terms `bank` and `greec`. We will ask for correlations higher than `0.5` for `bank` and `0.45` for `greek`:

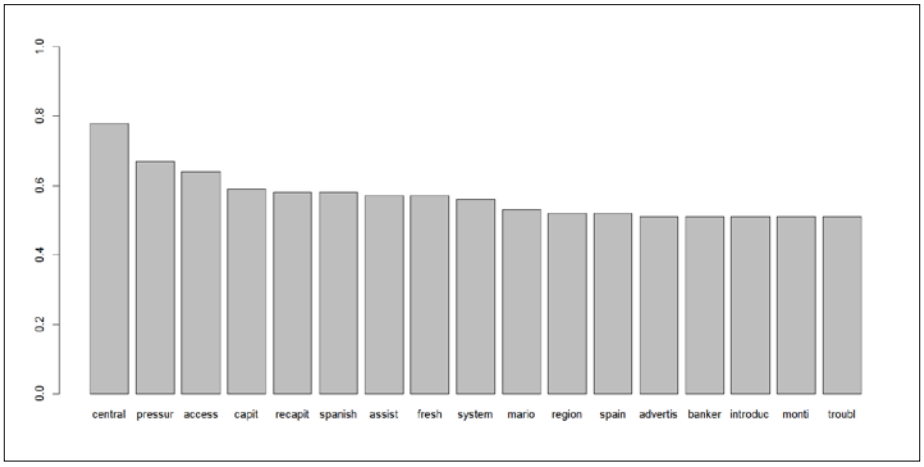
```
Assocs = findAssocs(loaded_tdm, terms = c("bank", "greek"), corlim =
 c(0.5, 0.45))
```

We can display the correlations in textual form by typing (the output is not displayed here):

```
Assocs
```

We can also examine relationships visually here for bank:

```
barplot(Assocs[[1]], ylim = c(0,1))
```



A visualization of high term correlations related to "bank"

Summary

In this chapter, we discussed how to deal with text in R in order to perform classification. We examined how to load documents from several sources, preprocess them, and how to compute term frequencies. We compared the reliability of various algorithms in the classification such as Naïve Bayes, k-Nearest Neighbors, logistic regression, and support vector machines. Additionally, we examined how to perform basic topic modeling in order to extract meaning. We then studied how to automatically download news articles from sources such as *The New York Times Article Search API* and extract and visualize associations between terms.

In the next chapter, we will discuss cross-validation and how to export models using the PMML.

14
Cross-validation and Bootstrapping Using Caret and Exporting Predictive Models Using PMML

In *Chapter 12*, *Multilevel Analyses*, we examined how to fit and predict nested data using multilevel analyses. In the previous chapter, we discussed text mining in R. In this chapter, we will discover how to perform cross-validation and bootstrapping using the `caret` package and how to export models using **Predictive Model Markup Language** (**PMML**).

Cross-validation and bootstrapping of predictive models using the caret package

In this section, we will discuss how to examine the reliability of a model with cross-validation. We start by discussing what cross-validation is.

Cross-validation

You might remember that, in several chapters, we used half of the data to train the model and half of it to test it. The aim of this process was to ensure that the high reliability of a classification, for instance, was not due to the fitting of noise in the data rather than true relationships. We have seen, for instance, in the previous chapter, that the reliability of a classification on the training set is usually higher than in the test set (unseen data).

The process of using half of the data for training and half for testing is actually a special case of cross-validation, that is, two-fold cross-validation. We can perform cross-validation using more folds. Two very common approaches are ten-fold cross-validation and leave-one-out cross-validation. In ten-fold cross-validation, the data is randomly split into 10 groups that contain the same number of cases (or approximately). For the following explanation, we will call these groups 1 to 10. The analysis is performed 10 times. The first time the analysis is performed, groups 1 to 9 are used to train the algorithm and group 10 is used to test the model. The second time, groups 1 to 8 and 10 are used for training and group 9 for testing, and so on (see the following figure). So each group is included at some point for training, with 8 other groups, and each group is used for testing individually. Doing things this way allows for more accurate estimates of the reliability of the algorithm on the data as the analysis is performed several times with different sets, which permits the obtaining of distribution of measures of reliability. Another advantage is that 9/10th of the data is used in the training set at each iteration, and we can use all cases at both testing and training stages.

Fold	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Iteration 7	Iteration 8	Iteration 9	Iteration 10
Fold 1	Train	Train	Train	Train	Train	Train	Train	Train	Train	Test
Fold 2	Train	Train	Train	Train	Train	Train	Train	Train	Test	Train
Fold 3	Train	Train	Train	Train	Train	Train	Train	Test	Train	Train
Fold 4	Train	Train	Train	Train	Train	Train	Test	Train	Train	Train
Fold 5	Train	Train	Train	Train	Train	Test	Train	Train	Train	Train
Fold 6	Train	Train	Train	Train	Test	Train	Train	Train	Train	Train
Fold 7	Train	Train	Train	Test	Train	Train	Train	Train	Train	Train
Fold 8	Train	Train	Test	Train	Train	Train	Train	Train	Train	Train
Fold 9	Train	Test	Train	Train	Train	Train	Train	Train	Train	Train
Fold 10	Test	Train	Train	Train	Train	Train	Train	Train	Train	Train

Representation of training and testing sets in 10-fold cross validation

Leave-one-out cross-validation is quite similar, except that the data is not split into groups. Using this approach, the analysis is performed as many times as there are cases, and each of the cases is used once for testing and the rest of the time for training.

Performing cross-validation in R with caret

There are several ways of performing cross-validation in R. Using the `caret` package is one of the most effective ways, as the package provides a unified framework to use with different algorithms. This is exactly what we are going to do in the rest of this section, keep it simple.

First, we start by installing (if not already done) and loading the package:

```
install.packages("caret"); library (caret)
```

We can determine the number of folds with the `trainControl()` function.

Here for ten-fold cross-validation the number of folds will be as follows:

```
CtrlCV = trainControl(method = "cv", number = 10)
```

Here, for leave-one-out cross-validation the number of folds will be as follows:

```
CtrlLOO = trainControl(method = "LOOCV")
```

Now that we have set this, we can perform the analyses! We will do so using ten-fold cross-validation (replace `CtrlCV` by `CtrlLOO` for leave-one-out cross validation). To simplify, we will use the `iris` dataset for some examples. If any of the following required packages are not installed on your system, please use the `install.packages()` function to install them:

- Naive Bayes:

  ```
  install.packages("klaR")
  modelNB = train(Species ~ ., data = iris, trControl = CtrlCV,
  method = "nb")
  ```

  Packages `klaR` and `MASS` are required and loaded automatically

- C4.5:

  This requires loading `RWeka`

  ```
  library(RWeka)
  modelC45 = train(Species ~ ., data = iris,
     trControl = CtrlCV, method = "J48")
  ```

- C5.0:

  ```
  modelC50 = train(Species ~ ., data = iris,
     trControl =CtrlCV, method = "C5.0")
  ```

- The `C50` and `plyr` packages are loaded automatically:

  ```
  modelCART = train(Species ~ ., data = iris,
     trControl =CtrlCV, method = "rpart")
  ```

- The `rpart` package is loaded automatically:

```
modelRF = train(Species ~ ., data = iris, trControl = CtrlCV,
   method = "rf")
```

- The `randomForest` package is loaded automatically

We have covered the other algorithms in this book, and many more are available. To get the list, simply type:

```
names(getModelInfo())
```

We can inspect the accuracy of the models as follows (we take the example of the Naïve Bayes classification):

```
modelNB
```

The output reproduced here displays information about the sample, classes, and predictors; a summary of the method used; and more importantly the average accuracy and kappa values, and their standard deviations for different tuning parameters, which differ depending of the algorithm used (see the end of the output for a description). We can see that the average accuracy and kappa values are excellent, with small standard deviations:

```
Naive Bayes

150 samples
  4 predictor
  3 classes: 'setosa', 'versicolor', 'virginica'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 135, 135, 135, 135, 135, 135, ...
Resampling results across tuning parameters:

  usekernel  Accuracy   Kappa  Accuracy SD  Kappa SD
  FALSE      0.9533333  0.93   0.05488484   0.08232726
   TRUE      0.9533333  0.93   0.05488484   0.08232726
Tuning parameter fL was held constant at a value of 0.
Accuracy was used to select the optimal model using  the largest value.
 The final values used for the model were fL = 0 and usekernel = FALSE.
```

Bootstrapping

The aim of bootstrapping is also to obtain a more precise image of the reliability of the model on the data. This is done in a different fashion. Instead of partitioning the data for training and testing, a random sample of *n* cases is selected *N* times from the original set with replacement (meaning that the same case can occur several times at each iteration), where *N* is the number of iterations and *n* is the number of cases. The analysis is performed on each of the samples independently, which gives mean and standard deviation for the estimates.

Performing bootstrapping in R with caret

Bootstrapping is done in a way similar to cross-validation, simply by specifying it using the `trainControl` function:

```
CtrlBoot = trainControl(method="boot", number=1000)
```

Let's take our examples again:

- Naive Bayes:

```
modelNBboot = train(Species ~ ., data = iris,
   trControl = CtrlBoot, method = "nb")
```

- C4.5:

```
modelC45boot = train(Species ~ ., data = iris,
   trControl = CtrlBoot, method = "J48")
```

- C5.0:

```
modelC50boot = train(Species ~ ., data = iris,
   trControl = CtrlBoot, method = "C5.0")
```

- CART:

```
modelCARTboot = train(Species ~ ., data = iris,
   trControl = CtrlBoot, method = "rpart")
```

- Random forests:

```
modelRFboot = train(Species ~ ., data = iris,
   trControl = CtrlBoot, method = "rf")
```

The output is quite similar to what we have seen previously. We will, therefore, not comment on it.

Predicting new data

Predictive models are built with the intent of predicting unseen data.
This can be done very easily. In what follows, we first partition the data in two sets using stratified sampling, one of 75 percent, which we will use to train and test using cross-validation, and another of 25 percent, which contains unseen data (data that we have not yet used):

```
forCV = createDataPartition(iris$Species, p=0.75, list=FALSE)
CVset = iris[forCV,]
NEWset = iris[-forCV,]
```

We now create the cross-validated model (with Naïve Bayes):

```
model = train(Species ~ ., data = CVset, trControl = CtrlCV,
   method = "nb")
```

We can now predict our unseen data using this model:

```
Predictions = predict(model, NEWset)
```

Exporting models using PMML

Let's get started with PMML and the way models are exported using it.

What is PMML?

PMML is a standard for sharing predictive models across software. The standard has been developed and improved by the Data Mining Group since 1997. Using PMML, the user can notably build a model using one software package and use another software package for prediction. The export and/or import of models using PMML is currently supported by a wide range of solutions including (but not restricted to) R, Rapidminer, SAS Enterprise Miner, SPSS Modeler, and Weka.

Numerous algorithms are supported by PMML. The following table presents the list of algorithms we have explored for which models can be exported using the `PMML` package in R (actually, most of them). The function to generate the models, the package containing the function, and the chapter of this book where we have discussed it are also indicated:

ALGORITHM	FUNCTION	PACKAGE	CHAPTER
K-means clustering	`kmeans`	`(stats)`	4
Hierarchical clustering	`hclust`	`(stats)`	5
Association rule mining	`apriori`	`arules`	7
Linear regression	`lm`	`(stats)`	9
Naïve Bayes classification and regression	`naiveBayes`	`e1071`	10
Classification and regression trees	`rpart`	`rpart`	11
Random forest for classification and regression	`randomForest`	`randomForest`	11
Logistic regression	`glm`	`(stats)`	13
Classification with Support Vector Machines	`svm`	`e1071`	13

The `PMML` package also supports other models that we have not discussed here. The list can be found in the package documentation at `http://cran.r-project.org/web/packages/pmml/pmml.pdf`.

A brief description of the structure of PMML objects

PMML objects are generated using XML. The PMML translates a simple linear regression model (preceded by the R output). As always, we start by installing and loading the required package:

```
install.packages("pmml"); library(pmml)
model = lm (Sepal.Length ~ Sepal.Width, data = iris)
model
```

The model output is as follows:

```
Call:
lm(formula = Sepal.Length ~ Sepal.Width, data = iris)


Coefficients:
(Intercept)  Sepal.Width
     6.5262      -0.2234
```

We now generate the PMML code for the model:

```
pmml(model)
```

The following output is displayed, in a commented form:

Information about the format of the document is first included, which is as follows:

```
<PMML version="4.2" xmlns="http://www.dmg.org/PMML-4_2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.dmg.org/PMML-4_2
http://www.dmg.org/v4-2/pmml-4-2.xsd">
```

The header, featuring details about the user, the software package, the date, and the algorithm that generated the model is then included:

```
<Header copyright="Copyright (c) 2015 mayore" description="Linear
Regression Model">
<Extension name="user" value="mayore" extender="Rattle/PMML"/>
<Application name="Rattle/PMML" version="1.4"/>
<Timestamp>2015-02-18 13:36:46</Timestamp>
</Header>
```

Next is the data dictionary that describes the attributes included in the analysis:

```
<DataDictionary numberOfFields="2">
<DataField name="Sepal.Length" optype="continuous" dataType="double"/>
<DataField name="Sepal.Width" optype="continuous" dataType="double"/>
</DataDictionary>
```

Then comes information about the model, including the algorithm used, the role of the variables included in the analysis, and the generated output:

```
<RegressionModel modelName="Linear_Regression_Model"
functionName="regression" algorithmName="least squares">
<MiningSchema>
<MiningField name="Sepal.Length" usageType="predicted"/><MiningField
name="Sepal.Width" usageType="active"/>
</MiningSchema>
<Output>
<OutputField name="Predicted_Sepal.Length" feature="predictedValue"/>
</Output>
```

```
<RegressionTable intercept="6.52622255089448">
<NumericPredictor name="Sepal.Width" exponent="1"
coefficient="-0.2233610611299"/>
</RegressionTable>
</RegressionModel>
</PMML>
```

The PMML code for different algorithms can include more or less information, but the structure is always quite similar to the one we presented previously.

Examples of predictive model exportation

In this section, we present some examples of model exportation using PMML. As we have already discovered using the linear regression example previously, the process is quite simple: it usually consists of presenting the model as an argument to the `pmml()` function. Here, we simply propose some very basic examples of exporting to PMML. If you want to know more about PMML, I suggest reading the book *PMML in Action: Unleashing the Power of Open Standards for Data Mining and Predictive Analytics*, *Guazzelli, Wen-Ching, Tridivesh, CreateSpace Independent Publishing*.

Exporting k-means objects

Here we start by creating a k - means model:

```
iris.kmeans = kmeans(iris[1:4],3)
```

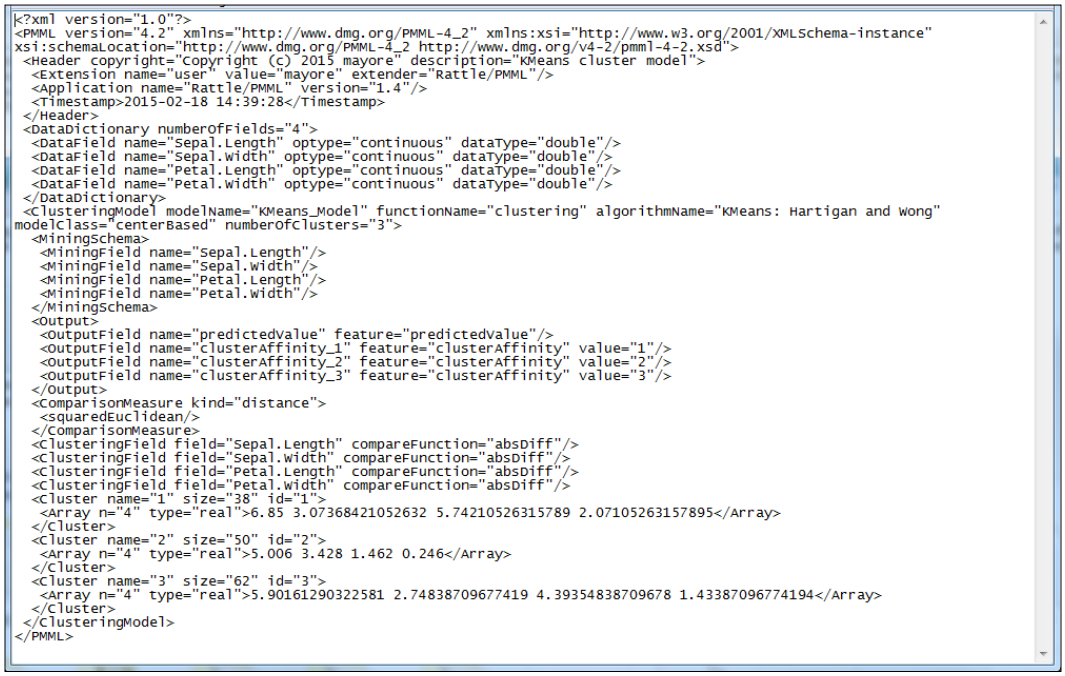
We then export the model to PMML:

```
pmml_kmeans = pmml(iris.kmeans)
```

Next, save it as an XML file:

```
saveXML(pmml_kmeans, data=iris, "iris_kmeans.PMML")
```

Opening the document (see figure) allows us to check whether the model has been appropriately exported.



```
<?xml version="1.0"?>
<PMML version="4.2" xmlns="http://www.dmg.org/PMML-4_2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.dmg.org/PMML-4_2 http://www.dmg.org/v4-2/pmml-4-2.xsd">
 <Header copyright="Copyright (c) 2015 mayore" description="KMeans cluster model">
  <Extension name="user" value="mayore" extender="Rattle/PMML"/>
  <Application name="Rattle/PMML" version="1.4"/>
  <Timestamp>2015-02-18 14:39:28</Timestamp>
 </Header>
 <DataDictionary numberOfFields="4">
  <DataField name="Sepal.Length" optype="continuous" dataType="double"/>
  <DataField name="Sepal.Width" optype="continuous" dataType="double"/>
  <DataField name="Petal.Length" optype="continuous" dataType="double"/>
  <DataField name="Petal.Width" optype="continuous" dataType="double"/>
 </DataDictionary>
 <ClusteringModel modelName="KMeans_Model" functionName="clustering" algorithmName="KMeans: Hartigan and Wong"
modelClass="centerBased" numberOfClusters="3">
  <MiningSchema>
   <MiningField name="Sepal.Length"/>
   <MiningField name="Sepal.Width"/>
   <MiningField name="Petal.Length"/>
   <MiningField name="Petal.Width"/>
  </MiningSchema>
  <Output>
   <OutputField name="predictedValue" feature="predictedValue"/>
   <OutputField name="clusterAffinity_1" feature="clusterAffinity" value="1"/>
   <OutputField name="clusterAffinity_2" feature="clusterAffinity" value="2"/>
   <OutputField name="clusterAffinity_3" feature="clusterAffinity" value="3"/>
  </Output>
  <ComparisonMeasure kind="distance">
   <squaredEuclidean/>
  </ComparisonMeasure>
  <ClusteringField field="Sepal.Length" compareFunction="absDiff"/>
  <ClusteringField field="Sepal.Width" compareFunction="absDiff"/>
  <ClusteringField field="Petal.Length" compareFunction="absDiff"/>
  <ClusteringField field="Petal.Width" compareFunction="absDiff"/>
  <Cluster name="1" size="38" id="1">
   <Array n="4" type="real">6.85 3.07368421052632 5.74210526315789 2.07105263157895</Array>
  </Cluster>
  <Cluster name="2" size="50" id="2">
   <Array n="4" type="real">5.006 3.428 1.462 0.246</Array>
  </Cluster>
  <Cluster name="3" size="62" id="3">
   <Array n="4" type="real">5.90161290322581 2.74838709677419 4.39354838709678 1.43387096774194</Array>
  </Cluster>
 </ClusteringModel>
</PMML>
```

A snapshot of the content of the file with the PMML code

Hierarchical clustering

Exporting a hierarchical clustering model using PMML is a bit more complex. As an example, we first generate a data frame for hierarchical clustering:

```
DF = cbind(c(rep(1,4),rep(2,4),rep(3,4),rep(4,4)), rep(c(1,2,3,4),4),
 rep(c(rep(1,2),rep(2,2),rep(3,2),rep(4,2)),2))
```

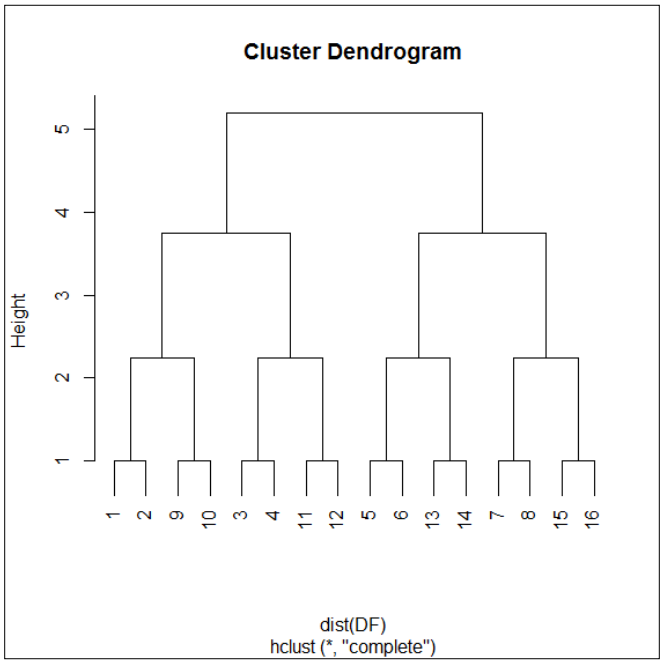
We then create our `hclust` object using the default parameters:

```
DF.hclust = hclust(dist(DF))
```

We now want to export it to a PMML object. Note that this will convert the model to a `kmeans` representation, which is required to include the cluster `centroids`. So we need to determine the number of clusters. We discussed ways to do this in *Chapter 5*, *Agglomerative Clustering Using hclust()*.

Here, we will simply plot the dendrogram and decide on this basis:

```
plot(DF.hclust)
```



Dendrogram for our dataset

In the figure, we can see that a two or four cluster model would describe the data well. The four cluster cut would have few data points, so we prefer the two cluster solution.

We now cut the tree using two clusters:

```
Cut = cutree(DF.hclust, k = 2)
```

Then, we extract `centroids`:

```
centroids = aggregate(DF, list(Cut), mean)
```

We can now export to PMML and save:

```
pmml_hclust = pmml(DF.hclust, centers = centroids)
saveXML(pmml_hclust, data=DF, "DF_hclust.PMML")
```

Exporting association rules (apriori objects)

It is easier to export association rules using PMML. We will use the `Adult` dataset that already contains the transactions for the `AdultUCI` data as follows:

```
library(arules)
data(Adult,AdultUCI)
names(AdultUCI)
```

As a reminder, the attributes are as follows:

```
[1]  age  workclass  fnlwgt  education  education-num
[6]  marital-status  occupation  relationship  race  sex
[11]  capita-gain  capital-loss  hours-per-week  native-country
income
```

We generate the rules as follows:

```
Adult.Apriori = apriori(Adult)
```

Save the PMML code:

```
saveXML(pmml(Adult.Apriori), "Adult_Apriori.PMML")
```

Exporting Naïve Bayes objects

Here, first load the `e1071` package containing the `naiveBayes()` function:

```
library(e1071)
```

We then build the classifier:

```
iris.NaiveBayes = naiveBayes(Species ~ Sepal.Length + Sepal.Width +
   Petal.Length + Petal.Width, data = iris)
```

Finally, we save to a file containing the PMML code:

```
saveXML(pmml(iris.NaiveBayes,dataset = iris, predictedField =
   "Species"), "iris_NaiveBayes.pmml")
```

Exporting decision trees (rpart objects)

We start by creating the classifier:

```
iris.rpart = rpart(Species ~ Sepal.Length + Sepal.Width +
   Petal.Length + Petal.Width, data = iris)
```

The tree (not displayed here) is obtained by typing:

```
iris.rpart
```

We can now export it to PMML and save it to an XML file:

```
saveXML(pmml(iris.rpart), data = iris, "iris_rpart.pmml")
# typing pmml(iris.rpart) would display the pmml object
```

Exporting random forest objects

We start by loading the `randomForest` package:

```
library(randomForest)
```

We then grow the forest (we use default parameters here):

```
iris.RandomForest = randomForest(Species ~ Sepal.Length + Sepal.Width
 + Petal.Length + Petal.Width, data = iris)
```

Finally, we export to PMML and save the file:

```
saveXML(pmml(iris.RandomForest), data = iris,
   "iris_randomForest.pmml")
```

Exporting logistic regression objects

We first generate some data:

```
set.seed(1234)
y = c(rep(0,50),rep(1,50))
x = rnorm(100)
x[51:100] = x[51:100] + 0.2
```

Then, we build the model (attribute `y` is the class):

```
glm.model = glm(y ~ x, family = "binomial")
```

Next, save it to a file containing the PMML code:

```
saveXML(pmml(glm.model), "glm_model.PMML")
```

Exporting support vector machine objects

We use the `iris` dataset again (`Species` is the class):

```
iris.svm = svm(Species ~ Sepal.Length + Sepal.Width + Petal.Length +
   Petal.Width, data = iris)
```

We now export to PMML and save:

```
saveXML(pmml(iris.svm), "iris_svm.PMML")
```

Summary

In this chapter, we discussed the basics of cross-validation and bootstrapping as well as exporting predictive models using PMML. We saw some examples of how to perform cross-validation and bootstrapping using the `caret` package. We showed how to predict data usage from the models we created. We explored the PMML structure, examined how to export to PMML, and saw how to save the resulting object to an XML file for use in other software packages.

A
Exercises and Solutions

Exercises

Here, we provide the exercises for most chapters and recommend that you practice your newly acquired skills after reading each chapter.

Chapter 1 – Setting GNU R for Predictive Modeling

Chapter 1 already contains the exercises and solutions.

Chapter 2 – Visualizing and Manipulating Data Using R

Have a look at the following exercises and try to perform the required tasks.

Let's have a little fun now! For this exercise, imagine a player betting on red for 1,000 consecutive trials. You'll have to plot the variations in money throughout the game. Use the `isRed` attribute of the data frame `Data` that we built at the beginning of the chapter. The player here starts with $1,000 and bets 1 in every game. The worst possible outcome is leaving with nothing, but also without debts. The line graph you will use has to be wide rather than tall; that is, 10 x 4 inches (use the documentation of the `par()` function to know how to configure this). Does the player end up winning or losing money?

Chapter 3 – Data Visualization with Lattice

Here, simply plot the relationship between `Petal.Length` and `Petal.Width` in the `iris` dataset and include the regression line.

Chapter 4 – Cluster Analysis

Here, simply determine the best number of clusters in the `iris` dataset (omit the `Species` attribute), using several distances measures (use `distance = "euclidean"`, `distance = "maximum"`, and `distance = "manhattan"`). Always use `method = "kmeans"`. What is the best number of clusters for each distance (use a majority rule). Do the results surprise you?

Chapter 5 – Agglomerative Clustering Using hclust()

Use `hclust()` to perform clustering on the `iris` dataset (omit the `Species` attribute). Use different methods for distance calculation (configurable using the `method` argument of the `dist()` function); and different linkage options (configurable using the `method` argument of the `hclust()` function).

Chapter 6 – Dimensionality Reduction with Principal Component Analysis

The `bfi` dataset (in the `psych` package) contains the responses of 2,800 participants to the Big Five Inventory (`http://www.ocf.berkeley.edu/~johnlab/bfi.htm`), which measures the five dimensions of personality. This contains 25 items of the inventory, five per dimension of personality: Neuroticism (N1-N5), Extraversion (E1-E5), Conscience (C1-C5), Agreeability (A1-A5), and Openness (O1-O5); as well as the variable's gender, education, and age at the end of the data frame.

Perform the following using the 25 items:

- Examine the missing values.
- Perform the diagnostics (omit cases with missing values). What do you find?
- Run PCA using `princomp()`.
- Plot the eigenvalues to determine the number of components to be retained.
- Rerun the analysis with that number of components using `principal()` with the varimax rotation and save the PCA scores.
- What is the proportion of cumulative variance explained by all the components?

- Name the components by looking at the loadings.
- What is the relationship (correlation) between each component and attribute age?

Chapter 7 – Exploring Association Rules with Apriori

Using the `ICU` dataset, without the attribute `race`, obtain the association rules with `support = 0.1`, `confidence = 0.8, minlen = 2` containing `pco=<=45` as an antecedent. You should obtain 13 rules. Convert the rules object to a data frame. Create an object containing the significance values of fisher's exact test for these rules (rounded to two decimal places), and append it as a column to the data frame you just created. Visualize the relationship between lift and significance of fisher's exact test (the *p* value) using the `plot()` function.

Chapter 8 – Probability Distributions, Covariance, and Correlation

Try performing the following exercises:

1. Adapt the code we used when discussing the binomial distribution to compute the probability of getting a red number in European roulette spins between:
    - 40 and 49 times
    - 51 and 60 times

    Are these numbers different ? If so, or if not, why?

2. Compute the correlation between petal length and petal width in the iris dataset using the `cor.test()` function. Is the correlation positive or negative? Is it significant?

Chapter 9 – Linear Regression

Try performing the following exercises:

1. Using the `nurses` dataset, examine the effect of a work-family conflict (attribute `WFC`) on work satisfaction (`WorkSat`) in the first model called `model01`.
2. Create a second model called `model02`, in which you include `WFC` and exhaustion (`Exhaus`) as predictors of `WorkSat`.

What happens to the relationship between `WFC` and `WorkSat`?

1. Test the relationship between `WFC` (predictor) and `Exhaus` (criterion).
2. If it is significant, perform a sobel test for the mediation of the relationship between `WFC` and `WorkSat` by `Exhaus`.

Chapter 10 – Classification with k-Nearest Neighbors and Naïve Bayes

In this exercise, you will try to classify the observations in the `Ozone` dataset using `knn()`. The class is the season and is computed (approximately) as follows:

```
1 library(mlbench)
2 data(Ozone)
3 Oz = na.omit(Ozone)
4 Oz$season = rep("winter",length(Oz[,1]))
5 Oz$season[as.numeric(Oz[[1]])>=3 & as.numeric(Oz[[1]])<=5]
6   = "spring"
7 Oz$season[as.numeric(Oz[[1]])>=6 & as.numeric(Oz[[1]])<=8]
8   = "summer"
9 Oz$season[as.numeric(Oz[[1]])>=9 & as.numeric(Oz[[1]])<=11]
10   = "autumn"
```

You will determine the best number of neighbors on the basis of the kappa value in the training set (higher is better). Finally, based on the kappa value in the testing set with the best number of neighbors, would you trust the classification?

The training and testing datasets are obtained as follows:

```
1  set.seed(5)
2  Oz$samples = sample(0:1, nrow(Oz), replace =T)
3  TRAIN = subset(Oz, samples == 0)
4  TEST = subset(Oz, samples == 1)
```

The class (`season`, the `target` attribute) is in column 14. Do not include columns 1 and 15 in the analyses. Take care of unlisting the class, for instance, with the `unlist()` function, if you use subsetting, otherwise, use the `df$attribute` notation for the class.

Chapter 11 – Classification Trees

Classify the observations in the `iris` dataset (class is `Species`) using C4.5 (pruned tree) and CART (using the default arguments). Which produces the best classification in terms of accuracy in the testing set? Create a function that assesses accuracy.

The training and testing sets are generated as follows:

```
IRIStrain = iris[as.numeric(row.names(iris)) %% 2 == T,]
IRIStest = iris[as.numeric(row.names(iris)) %% 2 == F,]
```

Chapter 12 – Multilevel Analyses

Try performing the following exercises:

- Using the `NursesML` dataset, visualize whether the relationship between exhaustion (attribute `Exhaust`) and work satisfaction (`WorkSat`) varies between hospitals. Include the regression line. Perform the same step for the relationship of depersonalization (`Depers`) and work satisfaction.
- Using the `modelPred` model, determine which difference in the observed work satisfaction is obtained from an increase of 1 in the predicted values.
- What is the intercept of the model (that is, the average value of work satisfaction for the average predicted value)?

Chapter 13 – Text Analytics with R

The `tm` package contains a corpus of 50 news articles that we access as follows:

```
data(acq)
acq
```

What are the terms that occur more than 100 times is this corpus before and after preprocessing with the `preprocess()` function?

Using the preprocessed data, plot the sorted term frequencies above 10 with terms (row names) on the *x* axis. Use the `barplot()` function.

Solutions

You will find the solutions to the exercises below.

Chapter 1 – Setting GNU R for Predictive Modeling

Chapter 1 already contains the exercises and solutions.

Chapter 2 – Visualizing and Manipulating Data Using R

One way to solve this is to first create a vector with a size of `1001` (line 1). The first value is the money that the player comes with (assigned on line 2). On line 3, we assign a copy of the `isRed` attribute to the `wins` vector. On lines 5 to 7, we compute the amount of money at each trial by adding 1 to the money on the next turn if the drawn number is red, and remove 1 if the number is not red. On line 8, we set the graphic parameters to contain a single plot (using argument `mfrow`) and to be of the required dimensions (10 by 4 inches) using the `pin` argument. Finally, we generate the plot on line 9:

```
1  money = rep(0,1001)
2  money[1] = 1000
3  wins = Data$isRed
4  for (i in 1:nrow(Data)) {
5     if (wins[i] == 1) money[i+1] = money[i] + 1
6     else money[i+1] = money[i] - 1
7  }
8  par (mfrow = c(1,1), pin=c(5,2))
9  plot(money, type = "l", xlab = "Trial number")
```

Run the code and have a look at your screen to know more about the outcomes of the player's game!

Chapter 3 – Data Visualization with Lattice

We plot these elements as follows:

```
library(lattice)
xyplot(Petal.Length ~ Petal.Width | Species, data = iris,
   panel = function(x, y, ...) {
```

```
        panel.xyplot(x,y)
        panel.abline(lm(y~x))
}
)
```

Chapter 4 – Cluster Analysis

The following code performs all the operations:

```
library(NbClust)
NbClust(iris[1:4],distance="euclidean", method= "kmeans")
NbClust(iris[1:4],distance="manhattan", method= "kmeans")
NbClust(iris[1:4],distance="maximum", method= "kmeans")
```

The outputs all show that a two-cluster solution is the best according to a majority rule. This is surprising because we know there are three classes in the dataset, and `kmeans()` is very accurate at classifying the cases. But apparently, this is not the best way to classify the data. Without prior knowledge, we could have concluded that there are observations for only two species of flowers in the dataset.

Chapter 5 – Agglomerative Clustering Using hclust()

We first create a vector with all the possible distances allowed by the `dist()` function; we then create a list, called `d`, of one element—the first distance matrix (with the Euclidean distance). We then add the other distance matrices to the list. Please note that the binary distance was included here for demonstration purposes, but the binary distance is of no practical use with quantitative data:

```
1  dist.types = c("euclidean", "maximum", "manhattan", "canberra",
2     "binary", "minkowski")
3  d = list(dist(iris[1:4], method = dist.types[1]))
4  for (i in 2:length(dist.types)) d = c(d,list(dist(iris[1:4],
5     method = dist.types[i])))
```

Next, we create a vector with all the possible agglomerating methods allowed by `hclust()`. We then perform the analysis six times for each of these methods (once per distance matrix), saving the result of each iteration in a matrix of lists:

```
1  agglo.types = c("ward.D", "ward.D2", "single", "complete",
2     "average", "mcquitty", "median", "centroid")
```

```
3  M = array(list(), 8)
4  for (i in 1:length(agglo.types)) {
5    OneDList = list(hclust(d[[1]], method = agglo.types[i]))
6    for (j in 2:length(dist.types)) OneDList =
7       c(OneDList, list(hclust(d[[j]], method = agglo.types[i])))
8    M[[i]] = OneDList
9  }
```

The 48 `hclust` models stored in the `M` matrix could be used for selecting the best models, for instance, by examining the plotted models. We can plot results with complete linkage and Euclidean distance (graph not included) like this:

```
plot(M[[4]][[1]])
```

The reader can notice that the results using binary distance (for example, with complete linkage as follows) are meaningless:

```
plot(M[[4]][[5]])
```

Chapter 6 – Dimensionality Reduction with Principal Component Analysis

Here are the solutions. Before anything, let's make sure the `psych` package is loaded:

```
library(psych)
```

- The display of the number of missing values can be done for all items in one line of code:

  ```
  summary(bfi[,1:25])
  ```

  We can see that there are between 9 and 36 missing values for each of the items.

  We assign the cases without missing values to a new object called `Dat`, retaining only the `bfi` measures:

  ```
  Dat = na.omit(bfi[,1:25])
  ```

  We can know how many cases of the original dataset contain missing values using the following line of code. The answer is `364`:

  ```
  dim(bfi)[1] - dim(Dat)[1]
  ```

- We examine the results of Bartlett's test of sphericity as follows:

  ```
  cortest.normal(Dat)
  ```

  The results show that the data set is significantly different from an identity matrix. We can examine the KMO index using the following code:

  ```
  KMO(Dat)
  ```

  The results show that the overall MSA value is large (0.85), as are most individual item values (lowest at 0.76). We deduce from these tests that performing PCA on this data is warranted.

- We run the PCA using the following line of code:

  ```
  pcas = princomp(Dat)
  ```

- We plot the eigenvalues (the squared standard deviations) like this:

  ```
  plot(pcas$sd^2)
  ```

  We find that a five-factor solution seems optimal, which is great as we have five theoretical factors of personality (the questionnaire is designed that way)

- We rerun the analysis with five factors and include the scores as follows:

  ```
  pcas2 = principal(Dat,nfactors=5, rotate="varimax", scores=T)
  ```

  We add the factorial scores to the dataset as follows:

  ```
  Dat = cbind(Dat, pcas2$scores)
  ```

- The output of the following line of code shows that the cumulative proportion of variance is 54 percent (under `Cumulative var`, last column).

  ```
  pcas2
  ```

- The component loadings also appear on the output (under Standardized loadings). We can see that the component called RC1 is mostly related to items with letter E (extraversion). RC2 loads higher on items with letter N (neuroticism), RC3 with items with letter C (conscience), RC4 with items with letter O (openness). Finally, RC5 has a higher loading with items with letter A (agreeability).

Chapter 7 – Exploring Association Rules with Apriori

You will find the solution to this exercise below.

- Here we will start from the beginning, by first obtaining a dataset without an attribute in column 4 (race):

```
library(vcdExtra)
ICU_norace = ICU[-4]
```

- We then load the arules package and generate the transaction list:

```
library(arules)
```

- We then prepare the dataset:

```
ICU_norace$age = cut(ICU_norace$age, breaks = 4)
ICU_norace$systolic = cut(ICU_norace$systolic, breaks = 4)
ICU_norace$hrtrate = cut(ICU_norace$hrtrate, breaks = 4)
ICU_norace_tr = as(ICU_norace, "transactions")
```

- Then, we generate the rules:

```
rulesLowpco = apriori(ICU_norace_tr, parameter =
   list(confidence = 0.8, support=.1, minlen = 2),
   appearance = list(lhs = c("pco=<=45"), default="rhs"))
```

- We create a data frame from the rules as follows:

```
rulesLowpco.df = as(rulesLowpco,"data.frame")
```

- We create the vector of the p value for Fisher's exact test as follows:

```
IM = interestMeasure(rulesLowpco, "fishersExactTest",
   ICU_norace_tr)
```

- We then append it to the data frame:

```
rulesLowpco.df$IM= round(IM, digits = 2)
```

- Finally, we plot the relationship between lift and the p value. We can see that the higher the lift, the lower the p value:

```
plot (rulesLowpco.df$lift, rulesLowpco.df$IM, xlab = "lift",
   ylab = "Significance")
```

Chapter 8 – Probability Distributions, Covariance, and Correlation

The following code will compute the probabilities of each outcome of interest:

```
1  p = 18/37
2  N = 100
3  n = 1
4  v40_49 = rep(1,10)
5  v51_60 = v40_49
6  i=1
7  for (n in 40:49) {
8    v40_49[i] = choose(N, n) * (p^n) * (1 - p)^(N-n)
9    i = i + 1
10 }
11 i=1
12 for (n in 51:60) {
13  v51_60[i] = choose(N, n) * (p^n) * (1 - p)^(N-n)
14  i = i + 1
15 }
```

- We can display the probability of obtaining between 40 and 49 red numbers drawn by summing the individual probabilities. The result is approximately 53.5 percent:

  ```
  sum(v40_49)
  ```

- We can do the same for the probability of obtaining between 51 and 60 red numbers. The result is approximately 34.7 percent:

  ```
  sum(v51_60)
  ```

  So it would be better to bet on black, right ? Well, this wouldn't make a difference. Because of the presence of the 0, which isn't black nor red, the probabilities of winning a several bets is strongly lower than the probabilities of losing, as the risk of losing each trial is higher than the chances of winning (19/37 versus 18/37).

- The correlation between petal width and petal length can be computed as follows:

  ```
  cor.test(iris$Petal.Width,iris$Petal.Length)
  ```

  The correlation between these attributes is around 0.963, and is significant at `p < .001`.

Chapter 9 – Linear Regression

- We examine the effect of work-family conflict (attribute `WFC`) on work satisfaction (`WorkSat`) in the first model called `model01`:

```
model01 = lm(WorkSat ~ WFC, data = nurses)
```

- We create a second model called `model02`, in which you include `WFC` and exhaustion (`Exhaus`) as predictors of `WorkSat`:

```
model02 = lm(WorkSat ~ WFC + Exhaus, data = nurses)
```

- We can check the relationship between `WFC` and `Worksat` in both models using the following code. We notice that the effect of `WFC` on work satisfaction is significant in `model01`, but not anymore in `model02`, that is when exhaustion is included in the model.

```
summary(model01)
summary(model02)
```

- Here is the code for computing `model03` with `WFC` included as a predictor of exhaustion:

```
model03 = lm(Exhaus ~ WFC, data = nurses)
```

  The summary of `model03` shows that `WFC` is a significant predictor of `Exhaus`:

```
summary(model03)
```

- We therefore perform a Sobel test for the mediation of the relationship between `WFC` and `WorkSat` by `Exhaus` as follows:

```
library(bda)
mediation.test(nurses$Exhaus, nurses$WFC, nurses$WorkSat)
```

  We can see that the value for the Sobel test is significant, which attests to the presence of the mediation

Chapter 10 – Classification with k-Nearest Neighbors and Naïve Bayes

The following code will generate the `kappas` values for numbers of neighbors ranging from 2 to 10 (the first value is for two neighbors, the second for three):

```
1  library(psych); library(class)
2  kappas = rep(0,9)
3  for (i in 1:9) {
```

```
4     set.seed(2222)
5     pred.train = knn(train = TRAIN[,2:13], test = TRAIN[,2:13],
6        cl = TRAIN$season, k = i+1)
7     tab = table(pred.train,TRAIN[,14])
8     kappas[i] = cohen.kappa(tab)[1]
9  }
```

Using the following code, we notice that the two clusters' solution has the highest kappa:

```
which.max(kappas)
```

We therefore use this solution to predict our data in the testing set and compute the kappa value, which is only 0.19 (that's pretty bad!):

```
pred.test = knn(train = TRAIN[,2:13], test = TEST[,2:13],
   cl = TRAIN$season, k = 2)
cohen.kappa(table(pred.test,TEST[,14]))[1]
```

Chapter 11 – Classification Trees

The C4.5 model is generated as follows:

```
library(RWeka)
IRIS.C45 = J48(Species~ . , data= IRIStrain,
   control= Weka_control(U=FALSE))
```

We compute the predictions as follows:

```
C45.preds = predict(IRIS.C45, IRIStest)
```

The CART model is generated as follows:

```
library(rpart)
IRIS.CART = rpart(Species ~. , data= IRIStrain)
```

The following line of code computes a data frame with the predicted probabilities:

```
CART.probs = as.data.frame(predict(IRIS.CART, IRIStest))
```

We now obtain the column number in which the predicted probabilities are the highest for each case:

```
CART.preds = apply(CART.probs, 1, which.max)
```

Finally, we assign the name of that column to each observation:

```
for (i in 1:length(CART.preds)) {
 col= as.numeric(CART.preds[i])
 CART.preds[i] = names(CART.probs[col])
}
```

Let's obtain the confusion matrix for both classifications:

```
CONF.C45 = table(IRIStest$Species, C45.preds)
CONF.CART = table(IRIStest$Species, CART.preds)
```

We create a function that computes the accuracy of a confusion matrix for us:

```
acc = function(table) {
   sum(diag(dim(table)[1])*table) / sum(CONF.C45)
}
```

We can see that, for this dataset, CART has a little advantage in accuracy (0.95 versus 0.92):

```
acc(CONF.C45)
acc(CONF.CART)
```

Chapter 12 – Multilevel Analyses

We obtain the graph for the relationship between `Exhaust` and `WorkSat` as follows:

```
library(lattice)
attach(NursesML)
xyplot(WorkSat~Exhaust | as.factor(hosp), panel = function(x,y) {
    panel.xyplot(x,y)
    panel.lmline(x,y)
})
```

We generate the second graph as follows:

```
xyplot(WorkSat~Depers | as.factor(hosp), panel = function(x,y) {
     panel.xyplot(x,y)
     panel.lmline(x,y)
})
```

We obtain the answer to questions 2 and 3 using the following line of code. The first value is the intercept, the second value is the coefficient for the predicted value (the increase on the actual work satisfaction for an increase of 1 in the predicted value):

```
coeffs(modelPred)
```

Chapter 13 – Text Analytics with R

This is easily done as follows. For the original corpus type the following code:

```
DTM = as.matrix(DocumentTermMatrix(acq))
FR = colSums(DTM)
FR[FR>100]
```

For the preprocessed corpus type the following code:

```
acq2 = preprocess(acq)
DTM2 = as.matrix(DocumentTermMatrix(acq2))
FR2 = colSums(DTM2)
FR2[FR2>100]
```

We obtain the graph as follows:

```
barplot(sort(FR2[FR2>30]), names.arg = rownames(sort(FR2[FR2>30])))
```

B
Further Reading and References

Preface

Chan, P. K., Fan, W., Prodromidis, A. L., and Stolfo, S. J. Distributed data mining in credit card fraud detection. *Intelligent Systems and their Applications, IEEE*, 14. 67-74. 1999.

Culotta, A. Towards detecting influenza epidemics by analyzing Twitter messages in *Proceedings of the first workshop on social media analytics*. ACM. 115-122. July 2010.

Ginsberg, J., Mohebbi, M. H., Patel, R. S., Brammer, L., Smolinski, M. S., and Brilliant, L. Detecting influenza epidemics using search engine query data. *Nature*, 457. 1012-1014. 2009.

Kotov, V. E. Big data, big systems, big challenges in M. Broy & A. V. Zamulin (Eds.). *Perspectives of System Informatics* . 41-44. Springer, Berlin. 2014.

O'Connor, B., Balasubramanyan, R., Routledge, B. R., and Smith, N. A. From tweets to polls: Linking text sentiment to public opinion time series. *ICWSM*, 11. 122-129. 2010.

Shearer C. The CRISP-DM model: the new blueprint for data mining. *Journal of Data Warehousing*, 5. 13-22. 2000.

Chapter 1 – Setting GNU R for Predictive Modeling

Becker, R. A., Cleveland, W. S., and Shyu, M. J. The visual design and control of trellis display. *Journal of Computational and Graphical Statistics, 5*. 123-155. 1996.

Leisch. F. *Creating R packages: A tutorial. Technical report*. Ludwig Maximilians Universität München. 2009.

Chapter 2 – Visualizing and Manipulating Data Using R

Lillis, D. A. *R Graph Essentials*. Packt Publishing. 2014.

Rousseeuw, P. J. and Van Zomeren, B. C. Unmasking multivariate outliers and leverage points. *Journal of the American Statistical Association, 85*. 633-639. 1990.

Chapter 3 – Data Visualization with Lattice

Becker, R. A., Cleveland, W. S., and Shyu, M. J. The visual design and control of trellis display. *Journal of Computational and Graphical Statistics, 5*. 123-155. 1996.

Campbell, J. A. Health insurance coverage in Department of Commerce (Ed.) *Current population reports*. 15-44. Department of Commerce, Washington. 1999.

Cleveland, W. S. *Visualizing data*. Hobart Press, Summit. 1993.

Sarkar, D. *Lattice: multivariate data visualization with R*. Springer, New York. 2008.

Chapter 4 – Cluster Analysis

Chiang, M. M. T., and Mirkin, B. Experiments for the number of clusters in k-means. In L. P. R. Correia and J. Cascalho (Eds.). *Progress in Artificial Intelligence*. 395-405. Springer, Berlin. 2007.

Cohen, J. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement, 20*. 37-46. 1960.

Deza, M. M. and Deza, E. *Encyclopedia of distances*. Springer, Berlin. 2013.

Hartigan, J. A. *Clustering algorithms*. Wiley Publishing, New York. 1975.

Chapter 5 – Agglomerative Clustering Using hclust()

Day, W. H. and Edelsbrunner, H. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification, 1*. 7-24. 1984.

Johnson, S. C. Hierarchical clustering schemes. *Psychometrika, 32*. 241-254. 1967.

Latané, B. Dynamic social impact: The creation of culture by communication. *Journal of Communication, 46*. 13-25. 1996.

The R Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. 2013.

Chapter 6 – Dimensionality Reduction with Principal Component Analysis

Abdi, H., and Williams, L. J. Principal Component Analysis. *Wiley Interdisciplinary Reviews: Computational Statistics, 2*. 433-459. 2010.

Dziuban, C. D., and Shirkey, E. C. When is a correlation matrix appropriate for factor analysis? Some decision rules. *Psychological Bulletin, 81*. 358-361. 1974.

Chapter 7 – Exploring Association Rules with Apriori

Adamo, J. M. *Data mining for association rules and sequential patterns: sequential and parallel algorithms*. Springer, Berlin. 2012.

Wu, X., Kumar, V., Quinlan, J. R., Ghosh, J., Yang, Q., Motoda, H., ... and Steinberg, D. Top 10 algorithms in data mining. *Knowledge and Information Systems, 14*. 1-37. 2008.

Chapter 8 – Probability Distributions, Covariance, and Correlation

Forbes, C., Evans, M., Hastings, N., and Peacock, B. *Statistical distributions.* Wiley Publishing, Hokoben. 2011.

Chapter 9 – Linear Regression

Montgomery, D. C., Peck, E. A., and Vining, G. G. *Introduction to linear regression analysis.* Wiley Publishing, Hokoben. 2012.

Renaud, O. and Victoria-Feser, M. P. A robust coefficient of determination for regression. *Journal of Statistical Planning and Inference, 140*. 1852-1862. 2010.

Chapter 10 – Classification with k-Nearest Neighbors and Naïve Bayes

Duda, R. O., Hart, P. E., and Stork, D. G. *Pattern classification*. Wiley Publishing, New York. 2001.

Chapter 11 – Classification Trees

Mingers, J. An empirical comparison of selection measures for decision-tree induction. *Machine Learning, 3*. 319-342. 1989.

Mitchel, T. Decision tree learning. *Machine learning*. 52-80. McGraw Hill, Burr Ridge. 1997.

Chapter 12 – Multilevel Analyses

Hox, J. *Multilevel analysis. Techniques and applications*. Routledge, New York. 2010.

Chapter 13 – Text Analytics with R

Delgado, M., Martín-Bautista, M. J., Sánchez, D., and Vila, M. A. Mining text data: special features and patterns in Hand, David J. (Eds.). *Pattern Detection and Discovery*. 140-153. Springer, Berlin. 2002.

Greene, D., O'Callaghan, D., and Cunningham, P. How many topics? Stability analysis for topic models in P. A. Flach, T. De Bie, and N. Cristianini (Eds.). *Machine learning and knowledge discovery in databases*. 498-513. Springer, Berlin. 2014.

Scholkopf, B., and Smola, A. J. Learning with kernels: support vector machines, regularization, optimization, and beyond. Cambridge: MIT press. 2001.

Chapter 14 – Cross-validation and Bootstrapping Using Caret and Exporting Predictive Models Using PMML

Guazzelli, A., Lin, W. C., and Jena, T. *PMML in action: unleashing the power of open standards for data mining and predictive analytics*. CreateSpace. 2012.

Kohavi, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. Proceedings of the 14th *International Joint Conference on Artificial Intelligence*. 1137-1145. Morgan Kaufmann, Montreal. 1995.

Index

D

E

F

G

N

O

P

Q

R

S

T

U

V

W

X

Thank you for buying

Learning Predictive Analytics with R

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

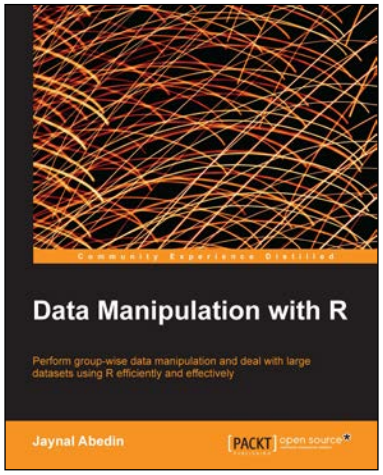
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

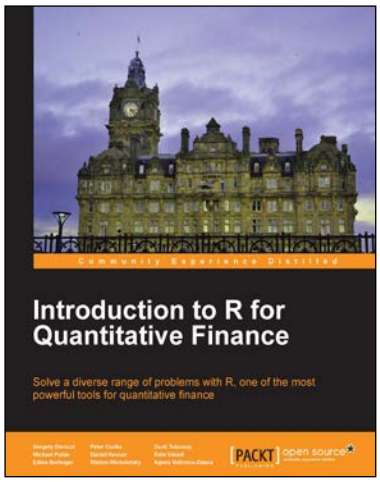


Data Manipulation with R

ISBN: 978-1-78328-109-1 Paperback: 102 pages

Perform group-wise data manipulation and deal with large datasets using R efficiently and effectively

1. Perform factor manipulation and string processing.
2. Learn group-wise data manipulation using plyr.
3. Handle large datasets, interact with database software, and manipulate data using sqldf.



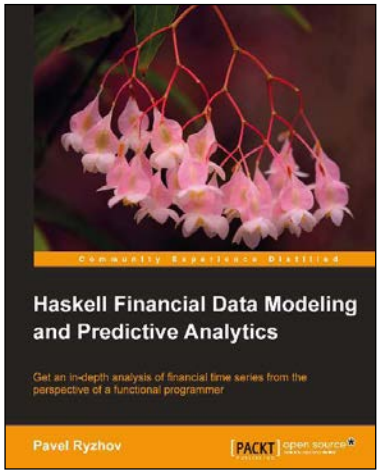
Introduction to R for Quantitative Finance

ISBN: 978-1-78328-093-3 Paperback: 164 pages

Solve a diverse range of problems with R, one of the most powerful tools for quantitative finance

1. Use time series analysis to model and forecast house prices.
2. Estimate the term structure of interest rates using prices of government bonds.
3. Detect systemically important financial institutions by employing financial network analysis.

[PACKT] open source
community experience distilled
PUBLISHING

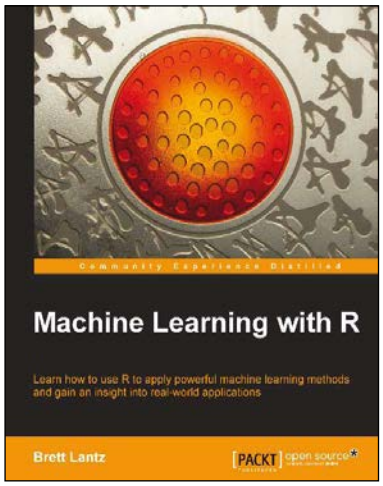


Haskell Financial Data Modeling and Predictive Analytics

ISBN: 978-1-78216-943-7 Paperback: 112 pages

Get an in-depth analysis of financial time series from the perspective of a functional programmer

1. Understand the foundations of financial stochastic processes.
2. Build robust models quickly and efficiently.
3. Tackle the complexity of parallel programming.



Machine Learning with R

ISBN: 978-1-78216-214-8 Paperback: 396 pages

Learn how to use R to apply powerful machine learning methods and gain an insight into real-world applications

1. Harness the power of R for statistical computing and data science.
2. Use R to apply common machine learning algorithms with real-world applications.
3. Prepare, examine, and visualize data for analysis.
4. Understand how to choose between machine learning models.