



Quick answers to common problems

R Data Analysis Cookbook

Over 80 recipes to help you breeze through your data analysis projects using R

Viswa Viswanathan Shanthi Viswanathan

[PACKT] open source^{*}
PUBLISHING community experience distilled

www.allitebooks.com

R Data Analysis Cookbook

Over 80 recipes to help you breeze through your
data analysis projects using R

Viswa Viswanathan
Shanthi Viswanathan



open source community experience distilled

BIRMINGHAM - MUMBAI

R Data Analysis Cookbook

Copyright © 2015 Packt Publishing

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

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

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

First published: May 2015

Production reference: 1220515

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

ISBN 978-1-78398-906-5

www.packtpub.com

Credits

Authors

Viswa Viswanathan
Shanthy Viswanathan

Project Coordinator

Kinjal Bari

Reviewers

Kenneth D. Graves
Jithin S L
Dipanjan Sarkar
Hang (Harvey) Yu

Proofreaders

Safis Editing
Stephen Copestake

Indexer

Rekha Nair

Acquisition Editors

Richard Brookes-Bland
Richard Harvey

Graphics

Sheetal Aute
Abhinash Sahu

Content Development Editor

Samantha Gonsalves

Production Coordinator

Manu Joseph

Technical Editor

Anushree Arun Tendulkar

Cover Work

Manu Joseph

Copy Editors

Charlotte Carneiro
Sameen Siddiqui

About the Authors

Viswa Viswanathan is an associate professor of Computing and Decision Sciences at the Stillman School of Business in Seton Hall University. After completing his PhD in artificial intelligence, Viswa spent a decade in academia and then switched to a leadership position in the software industry for a decade. During this period, he worked for Infosys, Igate, and Starbase. He embraced academia once again in 2001.

Viswa has taught extensively in fields ranging from operations research, computer science, software engineering, management information systems, and enterprise systems. In addition to teaching at the university, Viswa has conducted training programs for industry professionals. He has written several peer-reviewed research publications in journals such as Operations Research, IEEE Software, Computers and Industrial Engineering, and International Journal of Artificial Intelligence in Education. He has authored a book titled *Data Analytics with R: A hands-on approach*.

Viswa thoroughly enjoys hands-on software development, and has single-handedly conceived, architected, developed, and deployed several web-based applications.

Apart from his deep interest in technical fields such as data analytics, artificial intelligence, computer science, and software engineering, Viswa harbors a deep interest in education, with special emphasis on the roots of learning and methods to foster deeper learning. He has done research in this area and hopes to pursue the subject further.

Viswa would like to express deep gratitude to professors Amitava Bagchi and Anup Sen, who were inspirational forces during his early research career. He is also grateful to several extremely intelligent colleagues, notable among them being Rajesh Venkatesh, Dan Richner, and Sriram Bala, who significantly shaped his thinking. His aunt, Analdavalli; his sister, Sankari; and his wife, Shanthi, taught him much about hard work, and even the little he has absorbed has helped him immensely. His sons, Nitin and Siddarth, have helped with numerous insightful comments on various topics.

Shanthi Viswanathan is an experienced technologist who has delivered technology management and enterprise architecture consulting to many enterprise customers. She has worked for Infosys Technologies, Oracle Corporation, and Accenture. As a consultant, Shanthi has helped several large organizations, such as Canon, Cisco, Celgene, Amway, Time Warner Cable, and GE among others, in areas such as data architecture and analytics, master data management, service-oriented architecture, business process management, and modeling. When she is not in front of her Mac, Shanthi spends time hiking in the suburbs of NY/NJ, working in the garden, and teaching yoga.

Shanthi would like to thank her husband, Viswa, for all the great discussions on numerous topics during their hikes together and for exposing her to R and Java. She would also like to thank her sons, Nitin and Siddarth, for getting her into the data analytics world.

About the Reviewers

Kenneth D. Graves believes that data science will give us superpowers. Or, at the very least, allow us to make better decisions. Toward this end, he has over 15 years of experience in data science and technology, specializing in machine learning, big data, signal processing and marketing, and social media analytics. He has worked for Fortune 500 companies such as CBS and RCA-Technicolor, as well as finance and technology companies, designing state-of-art technologies and data solutions to improve business and organizational decision-making processes and outcomes. His projects have included facial and brand recognition, natural language processing, and predictive analytics. He works and mentors others in many technologies, including R, Python, C++, Hadoop, and SQL.

Kenneth holds degrees and/or certifications in data science, business, film, and classical languages. When he is not trying to discover superpowers, he is a data scientist and acting CTO at Soshag, LLC., a social media analytics firm. He is available for consulting and data science projects throughout the Greater Boston Area. He currently lives in Wellesley, MA.

I wish to thank my wife, Jill, for being the inspiration for all that I do.

Jithin S L completed his BTech in information technology from Loyola Institute of Technology and Science. He started his career in the field of analytics and then moved to various verticals of big data technology. He has worked with reputed organizations, such as Thomson Reuters, IBM, and Flytxt, in different roles. He has worked in the banking, energy, healthcare, and telecom domains, and has handled global projects on big data technology.

He has submitted many research papers on technology and business at national and international conferences. Currently, Jithin is associated with IBM Corporation as a systems analyst—big data big insight in business analytics and optimization unit.

“Change is something which brings us to THINK beyond our limits, worries it also provides an opportunity to learn new things in a new way, experiment, explore, and advise towards success.”

-Jithin

I surrender myself to God almighty who helped me review this book in an effective way. I dedicate my work on this book to my dad, Mr. N. Subbian Asari; my lovable mom, Mrs. M. Lekshmi; and my sweet sister, Ms. S.L Jishma, for coordinating and encouraging me to review this book. Last but not least, I would like to thank all my friends.

Dipanjan Sarkar is an IT engineer at Intel, the world's largest silicon company, where he works on analytics and enterprise application development. As part of his experience in the industry so far, he has previously worked as a data engineer at DataWeave, one of India's emerging big data analytics start-ups and also as a graduate technical intern in Intel.

Dipanjan received his master's degree in information technology from the International Institute of Information Technology, Bengaluru. His interests include learning about new technology, disruptive start-ups, and data science. He has also reviewed *Learning R for Geospatial Analysis*, Packt Publishing.

Hang (Harvey) Yu graduated from the University of Illinois at Urbana-Champaign with a PhD in computational biophysics and a master's degree in statistics. He has extensive experience on data mining, machine learning, and statistics. In the past, Harvey has worked on areas such as stochastic simulations and time series (in C and Python) as part of his academic work. He was intrigued by algorithms and mathematical modeling. He has been involved in data analytics since then.

He is currently working as a data scientist in Silicon Valley. He is passionate about data sciences. He has developed statistical/mathematical models based on techniques such as optimization and predictive modeling in R. Previously, Harvey worked as a computational sciences intern for ExxonMobil.

When Harvey is not programming, he is playing soccer, reading fiction books, or listening to classical music. You can get in touch with him at hangyu1@illinois.edu or on LinkedIn at www.linkedin.com/in/hangyu1.

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	v
Chapter 1: Acquire and Prepare the Ingredients – Your Data	1
Introduction	2
Reading data from CSV files	2
Reading XML data	5
Reading JSON data	7
Reading data from fixed-width formatted files	8
Reading data from R files and R libraries	9
Removing cases with missing values	11
Replacing missing values with the mean	13
Removing duplicate cases	15
Rescaling a variable to [0,1]	16
Normalizing or standardizing data in a data frame	18
Binning numerical data	20
Creating dummies for categorical variables	22
Chapter 2: What's in There? – Exploratory Data Analysis	25
Introduction	26
Creating standard data summaries	26
Extracting a subset of a dataset	28
Splitting a dataset	31
Creating random data partitions	32
Generating standard plots such as histograms, boxplots, and scatterplots	35
Generating multiple plots on a grid	43
Selecting a graphics device	45
Creating plots with the lattice package	46
Creating plots with the ggplot2 package	49
Creating charts that facilitate comparisons	55
Creating charts that help to visualize possible causality	60

Table of Contents

Creating multivariate plots	62
Chapter 3: Where Does It Belong? – Classification	65
Introduction	65
Generating error/classification-confusion matrices	66
Generating ROC charts	69
Building, plotting, and evaluating – classification trees	72
Using random forest models for classification	78
Classifying using the support vector machine approach	81
Classifying using the Naïve Bayes approach	85
Classifying using the KNN approach	88
Using neural networks for classification	90
Classifying using linear discriminant function analysis	93
Classifying using logistic regression	95
Using AdaBoost to combine classification tree models	98
Chapter 4: Give Me a Number – Regression	101
Introduction	101
Computing the root mean squared error	102
Building KNN models for regression	104
Performing linear regression	110
Performing variable selection in linear regression	117
Building regression trees	120
Building random forest models for regression	127
Using neural networks for regression	132
Performing k-fold cross-validation	135
Performing leave-one-out-cross-validation to limit overfitting	137
Chapter 5: Can You Simplify That? – Data Reduction Techniques	139
Introduction	139
Performing cluster analysis using K-means clustering	140
Performing cluster analysis using hierarchical clustering	146
Reducing dimensionality with principal component analysis	150
Chapter 6: Lessons from History – Time Series Analysis	159
Introduction	159
Creating and examining date objects	159
Operating on date objects	164
Performing preliminary analyses on time series data	166
Using time series objects	170
Decomposing time series	177
Filtering time series data	180
Smoothing and forecasting using the Holt-Winters method	182
Building an automated ARIMA model	185

Table of Contents

Chapter 7: It's All About Your Connections – Social Network Analysis	187
Introduction	187
Downloading social network data using public APIs	188
Creating adjacency matrices and edge lists	192
Plotting social network data	196
Computing important network metrics	209
Chapter 8: Put Your Best Foot Forward – Document and Present Your Analysis	217
Introduction	217
Generating reports of your data analysis with R Markdown and knitr	218
Creating interactive web applications with shiny	228
Creating PDF presentations of your analysis with R Presentation	234
Chapter 9: Work Smarter, Not Harder – Efficient and Elegant R Code	239
Introduction	239
Exploiting vectorized operations	240
Processing entire rows or columns using the apply function	242
Applying a function to all elements of a collection with lapply and sapply	245
Applying functions to subsets of a vector	248
Using the split-apply-combine strategy with plyr	250
Slicing, dicing, and combining data with data tables	253
Chapter 10: Where in the World? – Geospatial Analysis	261
Introduction	262
Downloading and plotting a Google map of an area	262
Overlaying data on the downloaded Google map	265
Importing ESRI shape files into R	267
Using the sp package to plot geographic data	270
Getting maps from the maps package	274
Creating spatial data frames from regular data frames containing spatial and other data	275
Creating spatial data frames by combining regular data frames with spatial objects	277
Adding variables to an existing spatial data frame	282
Chapter 11: Playing Nice – Connecting to Other Systems	285
Introduction	285
Using Java objects in R	286
Using JRI to call R functions from Java	292
Using Rserve to call R functions from Java	295
Executing R scripts from Java	298
Using the xlsx package to connect to Excel	299

Table of Contents —————

Reading data from relational databases – MySQL	302
Reading data from NoSQL databases – MongoDB	307
<u>Index</u>	<u>311</u>

Preface

Since the release of version 1.0 in 2000, R's popularity as an environment for statistical computing, data analytics, and graphing has grown exponentially. People who have been using spreadsheets and need to perform things that spreadsheet packages cannot readily do, or need to handle larger data volumes than what a spreadsheet program can comfortably handle, are looking to R. Analogously, people using powerful commercial analytics packages are also intrigued by this free and powerful option. As a result, a large number of people are now looking to quickly get things done in R.

Being an extensible system, R's functionality is divided across numerous packages with each one exposing large numbers of functions. Even experienced users cannot expect to remember all the details off the top of their head. This cookbook, aimed at users who are already exposed to the fundamentals of R, provides ready recipes to perform many important data analytics tasks. Instead of having to search the Web or delve into numerous books when faced with a specific task, people can find the appropriate recipe and get going in a matter of minutes.

What this book covers

Chapter 1, Acquire and Prepare the Ingredients – Your Data, covers the activities that precede the actual data analysis task. It provides recipes to read data from different input file formats. Furthermore, prior to actually analyzing the data, we perform several preparatory and data cleansing steps and the chapter also provides recipes for these: handling missing values and duplicates, scaling or standardizing values, converting between numerical and categorical variables, and creating dummy variables.

Chapter 2, What's in There? – Exploratory Data Analysis, talks about several activities that analysts typically use to understand their data before zeroing in on specific techniques to apply. The chapter presents recipes to summarize data, split data, extract subsets, and create random data partitions, as well as several recipes to plot data to reveal underlying patterns using standard plots as well as the lattice and ggplot2 packages.

Chapter 3, Where Does It Belong? – Classification, covers recipes for applying classification techniques. It includes classification trees, random forests, support vector machines, Naïve Bayes, K-nearest neighbors, neural networks, linear and quadratic discriminant analysis, and logistic regression.

Chapter 4, Give Me a Number – Regression, is about recipes for regression techniques. It includes K-nearest neighbors, linear regression, regression trees, random forests, and neural networks.

Chapter 5, Can You Simplify That? – Data Reduction Techniques, covers recipes for data reduction. It presents cluster analysis through K-means and hierarchical clustering. It also covers principal component analysis.

Chapter 6, Lessons from History – Time Series Analysis, covers recipes to work with date and date/time objects, create and plot time-series objects, decompose, filter and smooth time series, and perform ARIMA analysis.

Chapter 7, It's All About Your Connections – Social Network Analysis, is about social networks. It includes recipes to acquire social network data using public APIs, create and plot social networks, and compute important network metrics.

Chapter 8, Put Your Best Foot Forward – Document and Present Your Analysis, considers techniques to disseminate your analysis. It includes recipes to use R markdown and KnitR to generate reports, to use shiny to create interactive applications that enable your audience to directly interact with the data, and to create presentations with RPres.

Chapter 9, Work Smarter, Not Harder – Efficient and Elegant R Code, addresses the issue of writing efficient and elegant R code in the context of handling large data. It covers recipes to use the `apply` family of functions, to use the `plyr` package, and to use data tables to slice and dice data.

Chapter 10, Where in the World? – Geospatial Analysis, covers the topic of exploiting R's powerful features to handle spatial data. It covers recipes to use `RGoogleMaps` to get GoogleMaps and to superimpose our own data on them, to import ESRI shape files into R and plot them, to import maps from the `maps` package, and to use the `sp` package to create and plot spatial data frame objects.

Chapter 11, Playing Nice – Connecting to Other Systems, covers the topic of interconnecting R to other systems. It includes recipes for interconnecting R with Java, Excel and with relational and NoSQL databases (MySQL and MongoDB respectively).

What you need for this book

We have tested all the code in this book for R versions 3.0.2 (Frisbee Sailing) and 3.1.0 (Spring Dance). When you install or load some of the packages, you may get a warning message to the effect that the code was compiled for a different version, but this will not impact any of the code in this book.

Who this book is for

This book is ideal for those who are already exposed to R, but have not yet used it extensively for data analytics and are seeking to get up and running quickly for analytics tasks. This book will help people who aspire to enhance their skills in any of the following ways:

- ▶ perform advanced analyses and create informative and professional charts
- ▶ become proficient in acquiring data from many sources
- ▶ apply supervised and unsupervised data mining techniques
- ▶ use R's features to present analyses professionally

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:
"The `read.csv()` function creates a data frame from the data in the `.csv` file."

A block of code is set as follows:

```
> names(auto)

[1] "No"           "mpg"          "cylinders"
[4] "displacement" "horsepower"   "weight"
[7] "acceleration" "model_year"  "car_name"
```

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

```
export LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/server
export MAKEFLAGS="LDFLAGS=-Wl,-rpath $JAVA_HOME/lib/server"
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."



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 and data

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.

About the data files used in this book

We have generated many of the data files used in this book. We have also used some publicly available data sets. The table below lists the sources of these public data sets. We downloaded most of the public data sets from the University of California at Irvine (UCI) Machine Learning Repository at <http://archive.ics.uci.edu/ml/>. In the table below we have indicated this as "Downloaded from UCI-MLR."

Data file name	Source
auto-mpg.csv	<i>Quinlan, R. Combining Instance-Based and Model-Based Learning, Machine Learning Proceedings on the Tenth International Conference 1993, 236-243, held at University of Massachusetts, Amherst published by Morgan Kaufmann.</i> (Downloaded from UCI-MLR).
BostonHousing.csv	<i>D. Harrison and D.L. Rubinfeld, Hedonic prices and the demand for clean air, Journal for Environmental Economics and Management, pages 81–102, 1978.</i> (Downloaded from UCI-MLR)

daily-bike-rentals.csv	Fanaee-T, Hadi, and Gama, Joao, Event labeling combining ensemble detectors and background knowledge, Progress in Artificial Intelligence (2013): pp. 1-15, Springer Berlin Heidelberg. (Downloaded from UCI-MLR)
banknote-authentication.csv	<ul style="list-style-type: none">▶ Owner of database: Volker Lohweg, University of Applied Sciences, Ostwestfalen-Lippe▶ Donor of database: Helene Darksen, University of Applied Sciences, Ostwestfalen-Lippe (Downloaded from UCI-MLR)
education.csv	<i>Robust Regression and Outlier Detection, P. J. Rousseeuw and A. M. Leroy, Wiley, 1987.</i> (Downloaded from UCI-MLR)
walmart.csv walmart-monthly.csv	Downloaded from Yahoo! Finance
prices.csv	Downloaded from the US Bureau of Labor Statistics.
infy.csv, infy-monthly.csv	Downloaded from Yahoo! Finance.
nj-wages.csv	NJ Department of Education's website and http://federalgovernmentzipcodes.us .
nj-county-data.csv	Adapted from Wikipedia: http://en.wikipedia.org/wiki/List_of_counties_in_New_Jersey

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/9065OS_ColorImages.pdf.



Errata

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

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

Piracy

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

Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

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

1

Acquire and Prepare the Ingredients – Your Data

In this chapter, we will cover:

- ▶ Reading data from CSV files
- ▶ Reading XML data
- ▶ Reading JSON data
- ▶ Reading data from fixed-width formatted files
- ▶ Reading data from R data files and R libraries
- ▶ Removing cases with missing values
- ▶ Replacing missing values with the mean
- ▶ Removing duplicate cases
- ▶ Rescaling a variable to [0,1]
- ▶ Normalizing or standardizing data in a data frame
- ▶ Binning numerical data
- ▶ Creating dummies for categorical variables

Introduction

Data analysts need to load data from many different input formats into R. Although R has its own native data format, data usually exists in text formats, such as **CSV (Comma Separated Values)**, **JSON (JavaScript Object Notation)**, and **XML (Extensible Markup Language)**. This chapter provides recipes to load such data into your R system for processing.

Very rarely can we start analyzing data immediately after loading it. Often, we will need to preprocess the data to clean and transform it before embarking on analysis. This chapter provides recipes for some common cleaning and preprocessing steps.

Reading data from CSV files

CSV formats are best used to represent sets or sequences of records in which each record has an identical list of fields. This corresponds to a single relation in a relational database, or to data (though not calculations) in a typical spreadsheet.

Getting ready

If you have not already downloaded the files for this chapter, do it now and ensure that the `auto-mpg.csv` file is in your R working directory.

How to do it...

Reading data from .csv files can be done using the following commands:

1. Read the data from `auto-mpg.csv`, which includes a header row:

```
> auto <- read.csv("auto-mpg.csv", header=TRUE, sep = ",")
```

2. Verify the results:

```
> names(auto)
```

How it works...

The `read.csv()` function creates a data frame from the data in the .csv file. If we pass `header=TRUE`, then the function uses the very first row to name the variables in the resulting data frame:

```
> names(auto)

[1] "No"          "mpg"         "cylinders"
```

```
[4] "displacement" "horsepower"    "weight"  
[7] "acceleration" "model_year"    "car_name"
```

The `header` and `sep` parameters allow us to specify whether the `.csv` file has headers and the character used in the file to separate fields. The `header=TRUE` and `sep=", "` parameters are the defaults for the `read.csv()` function—we can omit these in the code example.

There's more...

The `read.csv()` function is a specialized form of `read.table()`. The latter uses whitespace as the default field separator. We discuss a few important optional arguments to these functions.

Handling different column delimiters

In regions where a comma is used as the decimal separator, `.csv` files use `"; "` as the field delimiter. While dealing with such data files, use `read.csv2()` to load data into R.

Alternatively, you can use the `read.csv("<file name>", sep="; ", dec=", ")` command.

Use `sep="\t"` for tab-delimited files.

Handling column headers/variable names

If your data file does not have column headers, set `header=FALSE`.

The `auto-mpg-noheader.csv` file does not include a header row. The first command in the following snippet reads this file. In this case, R assigns default variable names `V1`, `V2`, and so on:

```
> auto <- read.csv("auto-mpg-noheader.csv", header=FALSE)  
> head(auto,2)  
  
V1 V2 V3 V4 V5 V6 V7 V8 V9  
1 1 28 4 140 90 2264 15.5 71 chevrolet vega 2300  
2 2 19 3 70 97 2330 13.5 72 mazda rx2 coupe
```

If your file does not have a header row, and you omit the `header=FALSE` optional argument, the `read.csv()` function uses the first row for variable names and ends up constructing variable names by adding `X` to the actual data values in the first row. Note the meaningless variable names in the following fragment:

```
> auto <- read.csv("auto-mpg-noheader.csv")  
> head(auto,2)  
  
X1 X28 X4 X140 X90 X2264 X15.5 X71 chevrolet.vega.2300  
1 2 19 3 70 97 2330 13.5 72 mazda rx2 coupe  
2 3 36 4 107 75 2205 14.5 82 honda accord
```

Acquire and Prepare the Ingredients – Your Data

We can use the optional `col.names` argument to specify the column names. If `col.names` is given explicitly, the names in the header row are ignored even if `header=TRUE` is specified:

```
> auto <- read.csv("auto-mpg-noheader.csv",
+ header=FALSE, col.names =
+ c("No", "mpg", "cyl", "dis", "hp",
+ "wt", "acc", "year", "car_name"))

> head(auto, 2)

  No mpg cyl dis hp   wt   acc year      car_name
1  1 28    4 140 90 2264 15.5    71 chevrolet vega 2300
2  2 19    3  70 97 2330 13.5    72      mazda rx2 coupe
```

Handling missing values

When reading data from text files, R treats blanks in numerical variables as NA (signifying missing data). By default, it reads blanks in categorical attributes just as blanks and not as NA. To treat blanks as NA for categorical and character variables, set `na.strings=""`:

```
> auto <- read.csv("auto-mpg.csv", na.strings = "")
```

If the data file uses a specified string (such as "N/A" or "NA" for example) to indicate the missing values, you can specify that string as the `na.strings` argument, as in `na.strings= "N/A"` or `na.strings = "NA"`.

Reading strings as characters and not as factors

By default, R treats strings as *factors* (categorical variables). In some situations, you may want to leave them as character strings. Use `stringsAsFactors=FALSE` to achieve this:

```
> auto <- read.csv("auto-mpg.csv", stringsAsFactors=FALSE)
```

However, to selectively treat variables as characters, you can load the file with the defaults (that is, read all strings as factors) and then use `as.character()` to convert the requisite factor variables to characters.

Reading data directly from a website

If the data file is available on the Web, you can load it into R directly instead of downloading and saving it locally before loading it into R:

```
> dat <- read.csv("http://www.exploredatadata.net/ftp/WHO.csv")
```

Reading XML data

You may sometimes need to extract data from websites. Many providers also supply data in XML and JSON formats. In this recipe, we learn about reading XML data.

Getting ready

If the XML package is not already installed in your R environment, install the package now as follows:

```
> install.packages("XML")
```

How to do it...

XML data can be read by following these steps:

1. Load the library and initialize:

```
> library(XML)
> url <- "http://www.w3schools.com/xml/cd_catalog.xml"
```

2. Parse the XML file and get the root node:

```
> xmldoc <- xmlParse(url)
> rootNode <- xmlRoot(xmldoc)
> rootNode[1]
```

3. Extract XML data:

```
> data <- xmlSApply(rootNode,function(x) xmlSApply(x, xmlValue))
```

4. Convert the extracted data into a data frame:

```
> cd.catalog <- data.frame(t(data),row.names=NULL)
```

5. Verify the results:

```
> cd.catalog[1:2,]
```

How it works...

The `xmlParse` function returns an object of the `XMLInternalDocument` class, which is a C-level internal data structure.

The `xmlRoot()` function gets access to the root node and its elements. We check the first element of the root node:

```
> rootNode[1]
$CD
```

```
<CD>
<TITLE>Empire Burlesque</TITLE>
<ARTIST>Bob Dylan</ARTIST>
<COUNTRY>USA</COUNTRY>
<COMPANY>Columbia</COMPANY>
<PRICE>10.90</PRICE>
<YEAR>1985</YEAR>
</CD>
attr(, "class")
[1] "XMLInternalNodeList" "XMLNodeList"
```

To extract data from the root node, we use the `xmlSApply()` function iteratively over all the children of the root node. The `xmlSApply` function returns a matrix.

To convert the preceding matrix into a data frame, we transpose the matrix using the `t()` function. We then extract the first two rows from the `cd.catalog` data frame:

```
> cd.catalog[1:2,]
      TITLE      ARTIST COUNTRY      COMPANY PRICE YEAR
1 Empire Burlesque    Bob Dylan     USA    Columbia 10.90 1985
2 Hide your heart  Bonnie Tyler     UK CBS Records  9.90 1988
```

There's more...

XML data can be deeply nested and hence can become complex to extract. Knowledge of XPath will be helpful to access specific XML tags. R provides several functions such as `xpathSApply` and `getNodeSet` to locate specific elements.

Extracting HTML table data from a web page

Though it is possible to treat HTML data as a specialized form of XML, R provides specific functions to extract data from HTML tables as follows:

```
> url <- "http://en.wikipedia.org/wiki/World_population"
> tables <- readHTMLTable(url)
> world.pop <- tables[[5]]
```

The `readHTMLTable()` function parses the web page and returns a list of all tables that are found on the page. For tables that have an `id` attribute, the function uses the `id` attribute as the name of that list element.

We are interested in extracting the "10 most populous countries," which is the fifth table; hence we use `tables[[5]]`.

Extracting a single HTML table from a web page

A single table can be extracted using the following command:

```
> table <- readHTMLTable(url, which=5)
```

Specify which to get data from a specific table. R returns a data frame.

Reading JSON data

Several RESTful web services return data in JSON format—in some ways simpler and more efficient than XML. This recipe shows you how to read JSON data.

Getting ready

R provides several packages to read JSON data, but we use the `jsonlite` package. Install the package in your R environment as follows:

```
> install.packages("jsonlite")
```

If you have not already downloaded the files for this chapter, do it now and ensure that the `students.json` files and `student-courses.json` files are in your R working directory.

How to do it...

Once the files are ready and load the `jsonlite` package and read the files as follows:

1. Load the library:

```
> library(jsonlite)
```

2. Load the JSON data from files:

```
> dat.1 <- fromJSON("students.json")
> dat.2 <- fromJSON("student-courses.json")
```

3. Load the JSON document from the Web:

```
> url <- "http://finance.yahoo.com/webservice/v1/symbols/
allcurrencies/quote?format=json"
> jsonDoc <- fromJSON(url)
```

4. Extract data into data frames:

```
> dat <- jsonDoc$list$resources$resource$fields
```

5. Verify the results:

```
> dat[1:2,]
> dat.1[1:3,]
> dat.2[,c(1,2,4:5)]
```

How it works...

The `jsonlite` package provides two key functions: `fromJSON` and `toJSON`.

The `fromJSON` function can load data either directly from a file or from a web page as the preceding steps 2 and 3 show. If you get errors in downloading content directly from the Web, install and load the `httr` package.

Depending on the structure of the JSON document, loading the data can vary in complexity.

If given a URL, the `fromJSON` function returns a list object. In the preceding list, in step 4, we see how to extract the enclosed data frame.

Reading data from fixed-width formatted files

In fixed-width formatted files, columns have fixed widths; if a data element does not use up the entire allotted column width, then the element is padded with spaces to make up the specified width. To read fixed-width text files, specify columns by column widths or by starting positions.

Getting ready

Download the files for this chapter and store the `student-fwf.txt` file in your R working directory.

How to do it...

Read the fixed-width formatted file as follows:

```
> student <- read.fwf("student-fwf.txt",
  widths=c(4,15,20,15,4),
  col.names=c("id","name","email","major","year"))
```

How it works...

In the `student-fwf.txt` file, the first column occupies 4 character positions, the second 15, and so on. The `c(4,15,20,15,4)` expression specifies the widths of the five columns in the data file.

We can use the optional `col.names` argument to supply our own variable names.

There's more...

The `read.fwf()` function has several optional arguments that come in handy. We discuss a few of these as follows:

Files with headers

Files with headers use the following command:

```
> student <- read.fwf("student-fwf-header.txt",
  widths=c(4,15,20,15,4), header=TRUE, sep="\t", skip=2)
```

If `header=TRUE`, the first row of the file is interpreted as having the column headers. Column headers, if present, need to be separated by the specified `sep` argument. The `sep` argument only applies to the header row.

The `skip` argument denotes the number of lines to skip; in this recipe, the first two lines are skipped.

Excluding columns from data

To exclude a column, make the column width negative. Thus, to exclude the e-mail column, we will specify its width as `-20` and also remove the column name from the `col.names` vector as follows:

```
> student <- read.fwf("student-fwf.txt", widths=c(4,15,-20,15,4),
  col.names=c("id", "name", "major", "year"))
```

Reading data from R files and R libraries

During data analysis, you will create several R objects. You can save these in the native R data format and retrieve them later as needed.

Getting ready

First, create and save R objects interactively as shown in the following code. Make sure you have write access to the R working directory:

```
> customer <- c("John", "Peter", "Jane")
> orderdate <- as.Date(c('2014-10-1', '2014-1-2', '2014-7-6'))
> orderamount <- c(280, 100.50, 40.25)
> order <- data.frame(customer, orderdate, orderamount)
> names <- c("John", "Joan")
> save(order, names, file="test.Rdata")
> saveRDS(order, file="order.rds")
> remove(order)
```

After saving the preceding code, the `remove()` function deletes the object from the current session.

How to do it...

To be able to read data from R files and libraries, follow these steps:

1. Load data from R data files into memory:

```
> load("test.Rdata")
> ord <- readRDS("order.rds")
```

2. The `datasets` package is loaded in the R environment by default and contains the `iris` and `cars` datasets. To load these datasets' data into memory, use the following code:

```
> data(iris)
> data(c(cars,iris))
```

The first command loads only the `iris` dataset, and the second loads the `cars` and `iris` datasets.

How it works...

The `save()` function saves the serialized version of the objects supplied as arguments along with the object name. The subsequent `load()` function restores the saved objects with the same object names they were saved with, to the global environment by default. If there are existing objects with the same names in that environment, they will be replaced without any warnings.

The `saveRDS()` function saves only one object. It saves the serialized version of the object and not the object name. Hence, with the `readRDS()` function the saved object can be restored into a variable with a different name from when it was saved.

There's more...

The preceding recipe has shown you how to read saved R objects. We see more options in this section.

To save all objects in a session

The following command can be used to save all objects:

```
> save.image(file = "all.RData")
```

To selectively save objects in a session

To save objects selectively use the following commands:

```
> odd <- c(1,3,5,7)
> even <- c(2,4,6,8)
> save(list=c("odd", "even"), file="OddEven.Rdata")
```

The `list` argument specifies a character vector containing the names of the objects to be saved. Subsequently, loading data from the `OddEven.Rdata` file creates both `odd` and `even` objects. The `saveRDS()` function can save only one object at a time.

Attaching/detaching R data files to an environment

While loading `Rdata` files, if we want to be notified whether objects with the same name already exist in the environment, we can use:

```
> attach("order.Rdata")
```

The `order.Rdata` file contains an object named `order`. If an object named `order` already exists in the environment, we will get the following error:

```
The following object is masked _by_ .GlobalEnv:
```

```
order
```

Listing all datasets in loaded packages

All the loaded packages can be listed using the following command:

```
> data()
```

Removing cases with missing values

Datasets come with varying amounts of missing data. When we have abundant data, we sometimes (*not always*) want to eliminate the cases that have missing values for one or more variables. This recipe applies when we want to eliminate cases that have any missing values, as well as when we want to selectively eliminate cases that have missing values for a specific variable alone.

Getting ready

Download the `missing-data.csv` file from the code files for this chapter to your R working directory. Read the data from the `missing-data.csv` file while taking care to identify the string used in the input file for missing values. In our file, missing values are shown with empty strings:

```
> dat <- read.csv("missing-data.csv", na.strings="")
```

How to do it...

To get a data frame that has only the cases with no missing values for any variable, use the `na.omit()` function:

```
> dat.cleaned <- na.omit(dat)
```

Now, `dat.cleaned` contains only those cases from `dat`, which have no missing values in any of the variables.

How it works...

The `na.omit()` function internally uses the `is.na()` function that allows us to find whether its argument is `NA`. When applied to a single value, it returns a boolean value. When applied to a collection, it returns a vector:

```
> is.na(dat[4,2])
[1] TRUE

> is.na(dat$Income)
[1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
[10] FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
[19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

There's more...

You will sometimes need to do more than just eliminate cases with any missing values. We discuss some options in this section.

Eliminating cases with NA for selected variables

We might sometimes want to selectively eliminate cases that have `NA` only for a specific variable. The example data frame has two missing values for `Income`. To get a data frame with only these two cases removed, use:

```
> dat.income.cleaned <- dat[!is.na(dat$Income),]
> nrow(dat.income.cleaned)
[1] 25
```

Finding cases that have no missing values

The `complete.cases()` function takes a data frame or table as its argument and returns a boolean vector with `TRUE` for rows that have no missing values and `FALSE` otherwise:

```
> complete.cases(dat)

[1] TRUE  TRUE  TRUE FALSE  TRUE  FALSE  TRUE  TRUE  TRUE
[10] TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE
[19] TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Rows 4, 6, 13, and 17 have at least one missing value. Instead of using the `na.omit()` function, we could have done the following as well:

```
> dat.cleaned <- dat[complete.cases(dat), ]
> nrow(dat.cleaned)
[1] 23
```

Converting specific values to NA

Sometimes, we might know that a specific value in a data frame actually means that data was not available. For example, in the `dat` data frame a value of 0 for `income` may mean that the data is missing. We can convert these to `NA` by a simple assignment:

```
> dat$Income[dat$Income==0] <- NA
```

Excluding NA values from computations

Many R functions return `NA` when some parts of the data they work on are `NA`. For example, computing the `mean` or `sd` on a vector with at least one `NA` value returns `NA` as the result. To remove `NA` from consideration, use the `na.rm` parameter:

```
> mean(dat$Income)
[1] NA

> mean(dat$Income, na.rm = TRUE)
[1] 65763.64
```

Replacing missing values with the mean

When you disregard cases with any missing variables, you lose useful information that the nonmissing values in that case convey. You may sometimes want to impute reasonable values (those that will not skew the results of analyses very much) for the missing values.

Getting ready

Download the `missing-data.csv` file and store it in your R environment's working directory.

How to do it...

Read data and replace missing values:

```
> dat <- read.csv("missing-data.csv", na.strings = "")  
> dat$Income.imp.mean <- ifelse(is.na(dat$Income),  
    mean(dat$Income, na.rm=TRUE), dat$Income)
```

After this, all the `NA` values for `Income` will now be the mean value prior to imputation.

How it works...

The preceding `ifelse()` function returns the imputed mean value if its first argument is `NA`. Otherwise, it returns the first argument.

There's more...

You cannot impute the mean when a categorical variable has missing values, so you need a different approach. Even for numeric variables, we might sometimes not want to impute the mean for missing values. We discuss an often used approach here.

Imputing random values sampled from nonmissing values

If you want to impute random values sampled from the nonmissing values of the variable, you can use the following two functions:

```
rand.impute <- function(a) {  
  missing <- is.na(a)  
  n.missing <- sum(missing)  
  a.obs <- a[!missing]  
  imputed <- a  
  imputed[missing] <- sample (a.obs, n.missing, replace=TRUE)  
  return (imputed)  
}  
  
random.impute.data.frame <- function(dat, cols) {  
  nms <- names(dat)  
  for(col in cols) {  
    name <- paste(nms[col], ".imputed", sep = "")  
    dat[name] <- rand.impute(dat[,col])
```

```
    }  
  dat  
}
```

With these two functions in place, you can use the following to impute random values for both `Income` and `Phone_type`:

```
> dat <- read.csv("missing-data.csv", na.strings="")  
> random.impute.data.frame(dat, c(1,2))
```

Removing duplicate cases

We sometimes end up with duplicate cases in our datasets and want to retain only one among the duplicates.

Getting ready

Create a sample data frame:

```
> salary <- c(20000, 30000, 25000, 40000, 30000, 34000, 30000)  
> family.size <- c(4,3,2,2,3,4,3)  
> car <- c("Luxury", "Compact", "Midsize", "Luxury",  
  "Compact", "Compact", "Compact")  
> prospect <- data.frame(salary, family.size, car)
```

How to do it...

The `unique()` function can do the job. It takes a vector or data frame as an argument and returns an object of the same type as its argument but with duplicates removed.

Get unique values:

```
> prospect.cleaned <- unique(prospect)  
> nrow(prospect)  
[1] 7  
> nrow(prospect.cleaned)  
[1] 5
```

How it works...

The `unique()` function takes a vector or data frame as an argument and returns a like object with the duplicate eliminated. It returns the nonduplicated cases as is. For repeated cases, the `unique()` function includes one copy in the returned result.

There's more...

Sometimes we just want to identify duplicated values without necessarily removing them.

Identifying duplicates (without deleting them)

For this, use the `duplicated()` function:

```
> duplicated(prospect)
[1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
```

From the data, we know that cases 2, 5, and 7 are duplicates. Note that only cases 5 and 7 are shown as duplicates. In the first occurrence, case 2 is not flagged as a duplicate.

To list the duplicate cases, use the following code:

```
> prospect[duplicated(prospect), ]
      salary family.size     car
5   30000          3 Compact
7   30000          3 Compact
```

Rescaling a variable to [0,1]

Distance computations play a big role in many data analytics techniques. We know that variables with higher values tend to dominate distance computations and you may want to rescale the values to be in the range 0 - 1.

Getting ready

Install the `scales` package and read the `data-conversion.csv` file from the book's data for this chapter into your R environment's working directory:

```
> install.packages("scales")
> library(scales)
> students <- read.csv("data-conversion.csv")
```

How to do it...

To rescale the `Income` variable to the range [0,1]:

```
> students$Income.rescaled <- rescale(students$Income)
```

How it works...

By default, the `rescale()` function makes the lowest value(s) zero and the highest value(s) one. It rescales all other values proportionately. The following two expressions provide identical results:

```
> rescale(students$Income)
> (students$Income - min(students$Income)) /
  (max(students$Income) - min(students$Income))
```

To rescale a different range than [0,1], use the `to` argument. The following rescales `students$Income` to the range (0,100):

```
> rescale(students$Income, to = c(1, 100))
```

There's more...

When using distance-based techniques, you may need to rescale several variables. You may find it tedious to scale one variable at a time.

Rescaling many variables at once

Use the following function:

```
rescale.many <- function(dat, column.nos) {
  nms <- names(dat)
  for(col in column.nos) {
    name <- paste(nms[col], ".rescaled", sep = "")
    dat[name] <- rescale(dat[, col])
  }
  cat(paste("Rescaled ", length(column.nos),
            " variable(s)\n"))
  dat
}
```

With the preceding function defined, we can do the following to rescale the first and fourth variables in the data frame:

```
> rescale.many(students, c(1,4))
```

See also...

- ▶ Recipe: *Normalizing or standardizing data in a data frame* in this chapter

Normalizing or standardizing data in a data frame

Distance computations play a big role in many data analytics techniques. We know that variables with higher values tend to dominate distance computations and you may want to use the standardized (or Z) values.

Getting ready

Download the `BostonHousing.csv` data file and store it in your R environment's working directory. Then read the data:

```
> housing <- read.csv("BostonHousing.csv")
```

How to do it...

To standardize all the variables in a data frame containing only numeric variables, use:

```
> housing.z <- scale(housing)
```

You can only use the `scale()` function on data frames containing all numeric variables. Otherwise, you will get an error.

How it works...

When invoked as above, the `scale()` function computes the standard Z score for each value (ignoring NAs) of each variable. That is, from each value it subtracts the mean and divides the result by the standard deviation of the associated variable.

The `scale()` function takes two optional arguments, `center` and `scale`, whose default values are `TRUE`. The following table shows the effect of these arguments:

Argument	Effect
<code>center = TRUE, scale = TRUE</code>	Default behavior described earlier
<code>center = TRUE, scale = FALSE</code>	From each value, subtract the mean of the concerned variable
<code>center = FALSE, scale = TRUE</code>	Divide each value by the root mean square of the associated variable, where root mean square is <code>sqrt(sum(x^2) / (n-1))</code>
<code>center = FALSE, scale = FALSE</code>	Return the original values unchanged

There's more...

When using distance-based techniques, you may need to rescale several variables. You may find it tedious to standardize one variable at a time.

Standardizing several variables simultaneously

If you have a data frame with some numeric and some non-numeric variables, or want to standardize only some of the variables in a fully numeric data frame, then you can either handle each variable separately—which would be cumbersome—or use a function such as the following to handle a subset of variables:

```
scale.many <- function(dat, column.nos) {  
  nms <- names(dat)  
  for(col in column.nos) {  
    name <- paste(nms[col], ".z", sep = "")  
    dat[name] <- scale(dat[, col])  
  }  
  cat(paste("Scaled ", length(column.nos), " variable(s)\n"))  
  dat  
}
```

With this function, you can now do things like:

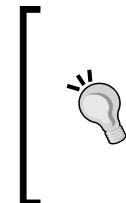
```
> housing <- read.csv("BostonHousing.csv")  
> housing <- scale.many(housing, c(1,3,5:7))
```

This will add the z values for variables 1, 3, 5, 6, and 7 with .z appended to the original column names:

```
> names(housing)  
[1] "CRIM"      "ZN"        "INDUS"     "CHAS"      "NOX"       "RM"  
[7] "AGE"        "DIS"       "RAD"        "TAX"       "PTRATIO"   "B"  
[13] "LSTAT"     "MEDV"     "CRIM.z"    "INDUS.z"  "NOX.z"    "RM.z"  
[19] "AGE.z"
```

See also...

- ▶ Recipe: Rescaling a variable to [0,1] in this chapter



Downloading the example code and data

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.

Binning numerical data

Sometimes, we need to convert numerical data to categorical data or a factor. For example, Naïve Bayes classification requires all variables (independent and dependent) to be categorical. In other situations, we may want to apply a classification method to a problem where the dependent variable is numeric but needs to be categorical.

Getting ready

From the code files for this chapter, store the `data-conversion.csv` file in the working directory of your R environment. Then read the data:

```
> students <- read.csv("data-conversion.csv")
```

How to do it...

Income is a numeric variable, and you may want to create a categorical variable from it by creating bins. Suppose you want to label incomes of \$10,000 or below as `Low`, incomes between \$10,000 and \$31,000 as `Medium`, and the rest as `High`. We can do the following:

1. Create a vector of break points:

```
> b <- c(-Inf, 10000, 31000, Inf)
```

2. Create a vector of names for break points:

```
> names <- c("Low", "Medium", "High")
```

3. Cut the vector using the break points:

```
> students$Income.cat <- cut(students$Income, breaks = b, labels =  
names)  
> students
```

	Age	State	Gender	Height	Income	Income.cat
1	23	NJ	F	61	5000	Low
2	13	NY	M	55	1000	Low
3	36	NJ	M	66	3000	Low
4	31	VA	F	64	4000	Low
5	58	NY	F	70	30000	Medium
6	29	TX	F	63	10000	Low
7	39	NJ	M	67	50000	High
8	50	VA	M	70	55000	High
9	23	TX	F	61	2000	Low
10	36	VA	M	66	20000	Medium

How it works...

The `cut()` function uses the ranges implied by the `breaks` argument to infer the bins, and names them according to the strings provided in the `labels` argument. In our example, the function places incomes less than or equal to 10,000 in the first bin, incomes greater than 10,000 and less than or equal to 31,000 in the second bin, and incomes greater than 31,000 in the third bin. In other words, the first number in the interval is not included and the second one is. The number of bins will be one less than the number of elements in `breaks`. The strings in `names` become the `factor` levels of the bins.

If we leave out `names`, `cut()` uses the numbers in the second argument to construct interval names as you can see here:

```
> b <- c(-Inf, 10000, 31000, Inf)
> students$Income.cat1 <- cut(students$Income, breaks = b)
> students
```

	Age	State	Gender	Height	Income	Income.cat	Income.cat1
1	23	NJ	F	61	5000	Low	(-Inf,1e+04]
2	13	NY	M	55	1000	Low	(-Inf,1e+04]
3	36	NJ	M	66	3000	Low	(-Inf,1e+04]
4	31	VA	F	64	4000	Low	(-Inf,1e+04]
5	58	NY	F	70	30000	Medium	(1e+04,3.1e+04]
6	29	TX	F	63	10000	Low	(-Inf,1e+04]
7	39	NJ	M	67	50000	High	(3.1e+04, Inf]
8	50	VA	M	70	55000	High	(3.1e+04, Inf]
9	23	TX	F	61	2000	Low	(-Inf,1e+04]
10	36	VA	M	66	20000	Medium	(1e+04,3.1e+04]

There's more...

You might not always be in a position to identify the `breaks` manually and may instead want to rely on R to do this automatically.

Creating a specified number of intervals automatically

Rather than determining the `breaks` and hence the intervals manually as above, we can specify the number of bins we want, say `n`, and let the `cut()` function handle the rest automatically. In this case, `cut()` creates `n` intervals of *approximately* equal width as follows:

```
> students$Income.cat2 <- cut(students$Income,
  breaks = 4, labels = c("Level1", "Level2",
  "Level3", "Level4"))
```

Creating dummies for categorical variables

In situations where we have categorical variables (factors) but need to use them in analytical methods that require numbers (for example, **K nearest neighbors (KNN)**, **Linear Regression**), we need to create dummy variables.

Getting ready

Read the `data-conversion.csv` file and store it in the working directory of your R environment. Install the `dummies` package. Then read the data:

```
> install.packages("dummies")
> library(dummies)
> students <- read.csv("data-conversion.csv")
```

How to do it...

Create dummies for all factors in the data frame:

```
> students.new <- dummy.data.frame(students, sep = ".")
> names(students.new)

[1] "Age"          "State.NJ"       "State.NY"       "State.TX"       "State.VA"
[6] "Gender.F"     "Gender.M"      "Height"        "Income"
```

The `students.new` data frame now contains all the original variables and the newly added dummy variables. The `dummy.data.frame()` function has created dummy variables for all four levels of the State and two levels of Gender factors. However, we will generally omit one of the dummy variables for State and one for Gender when we use machine-learning techniques.

We can use the optional argument `all = FALSE` to specify that the resulting data frame should contain only the generated dummy variables and none of the original variables.

How it works...

The `dummy.data.frame()` function creates dummies for all the factors in the data frame supplied. Internally, it uses another `dummy()` function which creates dummy variables for a single factor. The `dummy()` function creates one new variable for every level of the factor for which we are creating dummies. It appends the variable name with the factor level name to generate names for the dummy variables. We can use the `sep` argument to specify the character that separates them—an empty string is the default:

```
> dummy(students$State, sep = ".")
```

	State.NJ	State.NY	State.TX	State.VA
[1,]	1	0	0	0
[2,]	0	1	0	0
[3,]	1	0	0	0
[4,]	0	0	0	1
[5,]	0	1	0	0
[6,]	0	0	1	0
[7,]	1	0	0	0
[8,]	0	0	0	1
[9,]	0	0	1	0
[10,]	0	0	0	1

There's more...

In situations where a data frame has several factors, and you plan on using only a subset of these, you will create dummies only for the chosen subset.

Choosing which variables to create dummies for

To create dummies only for one variable or a subset of variables, we can use the `names` argument to specify the column names of the variables we want dummies for:

```
> students.new1 <- dummy.data.frame(students,
  names = c("State", "Gender") , sep = ".")
```


2

What's in There? – Exploratory Data Analysis

In this chapter, you will cover:

- ▶ Creating standard data summaries
- ▶ Extracting a subset of a dataset
- ▶ Splitting a dataset
- ▶ Creating random data partitions
- ▶ Generating standard plots such as histograms, boxplots, and scatterplots
- ▶ Generating multiple plots on a grid
- ▶ Selecting a graphics device
- ▶ Creating plots with the lattice package
- ▶ Creating plots with the ggplot2 package
- ▶ Creating charts that facilitate comparisons
- ▶ Creating charts that help visualize a possible causality
- ▶ Creating multivariate plots

Introduction

Before getting around to applying some of the more advanced analytics and machine learning techniques, analysts face the challenge of becoming familiar with the large datasets that they often deal with. Increasingly, analysts rely on visualization techniques to tease apart hidden patterns. This chapter equips you with the necessary recipes to incisively explore large datasets.

Creating standard data summaries

In this recipe we summarize the data using the `summary` function.

Getting ready

If you have not already downloaded the files for this chapter, do it now and ensure that the `auto-mpg.csv` file is in your R working directory.

How to do it...

Read the data from `auto-mpg.csv`, which includes a header row and columns separated by the default ", " symbol.

1. Read the data from `auto-mpg.csv` and convert `cylinders` to factor:

```
> auto <- read.csv("auto-mpg.csv", header = TRUE,  
+ stringsAsFactors = FALSE)  
> # Convert cylinders to factor  
> auto$cylinders <- factor(auto$cylinders,  
+ levels = c(3,4,5,6,8),  
+ labels = c("3cyl", "4cyl", "5cyl", "6cyl", "8cyl"))
```

2. Get the summary statistics:

```
summary(auto)
```

No	mpg	cylinders	displacement
Min. : 1.0	Min. : 9.00	3cyl: 4	Min. : 68.0
1st Qu.:100.2	1st Qu.:17.50	4cyl:204	1st Qu.:104.2
Median :199.5	Median :23.00	5cyl: 3	Median :148.5
Mean :199.5	Mean :23.51	6cyl: 84	Mean :193.4
3rd Qu.:298.8	3rd Qu.:29.00	8cyl:103	3rd Qu.:262.0
Max. :398.0	Max. :46.60		Max. :455.0
horsepower	weight	acceleration	model_year
Min. : 46.0	Min. :1613	Min. : 8.00	Min. :70.00
1st Qu.: 76.0	1st Qu.:2224	1st Qu.:13.82	1st Qu.:73.00

```
Median : 92.0    Median :2804    Median :15.50    Median :76.00
Mean   :104.1    Mean   :2970    Mean   :15.57    Mean   :76.01
3rd Qu.:125.0    3rd Qu.:3608    3rd Qu.:17.18    3rd Qu.:79.00
Max.   :230.0    Max.   :5140    Max.   :24.80    Max.   :82.00
car_name
Length:398
Class  :character
Mode   :character
```

How it works...

The `summary()` function gives a "six number" summary for numerical variables—minimum, first quartile, median, mean, third quartile, and maximum. For factors (or categorical variables), the function shows the counts for each level; for character variables, it just shows the total number of available values.

There's more...

R offers several functions to take a quick peek at data, and we discuss a few of those in this section.

Using the `str()` function for an overview of a data frame

The `str()` function gives a concise view into a data frame. In fact, we can use it to see the underlying structure of any arbitrary R object. The following commands and results show that the `str()` function tells us the type of object whose structure we seek. It also tells us about the type of each of its component objects along with an extract of some values. It can be very useful for getting an overview of a data frame:

```
> str(auto)

'data.frame': 398 obs. of 9 variables:
 $ No          : int  1 2 3 4 5 6 7 8 9 10 ...
 $ mpg         : num  28 19 36 28 21 23 15.5 32.9 16 13 ...
 $ cylinders   : Factor w/ 5 levels "3cyl","4cyl",...: 2 1 2 2 4
   2 5 2 4 5 ...
 $ displacement: num  140 70 107 97 199 115 304 119 250 318 ...
 $ horsepower  : int  90 97 75 92 90 95 120 100 105 150 ...
 $ weight      : int  2264 2330 2205 2288 2648 2694 3962 2615
   3897 3755 ...
 $ acceleration: num  15.5 13.5 14.5 17 15 15 13.9 14.8 18.5 14
   ...
 $ model_year  : int  71 72 82 72 70 75 76 81 75 76 ...
 $ car_name    : chr  "chevrolet vega 2300" "mazda rx2 coupe"
   "honda accord" "datsun 510 (sw)" ..
```

Computing the summary for a single variable

When factor summaries are combined with those for numerical variables (as in the earlier example), `summary()` gives counts for a maximum of six levels and lumps the other counts under the `Other`.

You can invoke the `summary()` function for a single variable as well. In this case, the summary you get for numerical variables remains as before, but for `factors`, you get counts for many more levels:

```
> summary(auto$cylinders)
> summary(auto$mpg)
```

Finding the mean and standard deviation

Use the functions `mean()` and `sd()` as follows:

```
> mean(auto$mpg)
> sd(auto$mpg)
```

Extracting a subset of a dataset

In this recipe, we discuss two ways to subset data. The first approach uses the row and column indices/names, and the other uses the `subset()` function.

Getting ready

Download the files for this chapter and store the `auto-mpg.csv` file in your R working directory. Read the data using the following command:

```
> auto <- read.csv("auto-mpg.csv", stringsAsFactors=FALSE)
```

The same subsetting principles apply for vectors, lists, arrays, matrices, and data frames. We illustrate with data frames.

How to do it...

The following steps extract a subset of a dataset:

1. Index by position. Get `model_year` and `car_name` for the first three cars:

```
> auto[1:3, 8:9]
> auto[1:3, c(8,9)]
```

2. Index by name. Get `model_year` and `car_name` for the first three cars:

```
> auto[1:3,c("model_year", "car_name")]
```

3. Retrieve all details for cars with the highest or lowest mpg, using the following code:

```
> auto[auto$mpg == max(auto$mpg) | auto$mpg ==  
min(auto$mpg),]
```

4. Get mpg and car_name for all cars with mpg > 30 and cylinders == 6:

```
> auto[auto$mpg>30 & auto$cylinders==6, c("car_name", "mpg")]
```

5. Get mpg and car_name for all cars with mpg > 30 and cylinders == 6 using partial name match for cylinders:

```
> auto[auto$mpg >30 & auto$cyl==6, c("car_name", "mpg")]
```

6. Using the subset() function, get mpg and car_name for all cars with mpg > 30 and cylinders == 6:

```
> subset(auto, mpg > 30 & cylinders == 6,  
select=c("car_name", "mpg"))
```

How it works...

The first index in `auto[1:3, 8:9]` denotes the rows, and the second denotes the columns or variables. Instead of the column positions, we can also use the variable names. If using the variable names, enclose them in "...".

If the required rows and columns are not contiguous, use a vector to indicate the needed rows and columns as in `auto[c(1,3), c(3,5,7)]`.



Use column names instead of column positions, as column positions may change in the data file.

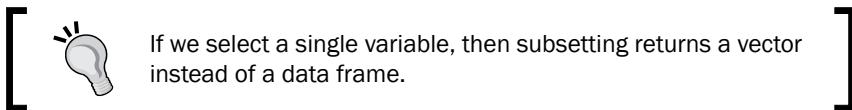
R uses the logical operators & (and), | (or), ! (negative unary), and == (equality check).

The `subset` function returns all variables (columns) if you omit the `select` argument. Thus, `subset(auto, mpg > 30 & cylinders == 6)` retrieves all the cases that match the conditions `mpg > 30` and `cylinders = 6`.

However, while using the indices in a logical expression to select rows of a data frame, you always need to specify the variables needed or indicate all variables with a comma following the logical expression:

```
> # incorrect  
> auto[auto$mpg > 30]  
Error in `^.data.frame`(.data, auto$mpg > 30) :  
  undefined columns selected  
>
```

```
> # correct  
> auto[auto$mpg > 30, ]
```



There's more...

We mostly use the indices by name and position to subset data. Hence, we provide some additional details around using indices to subset data. The `subset()` function is used predominantly in cases when we need to repeatedly apply the subset operation for a set of array, list, or vector elements.

Excluding columns

Use the minus sign for variable positions that you want to exclude from the subset. Also, you cannot mix both positive and negative indexes in the list. Both of the following approaches are correct:

```
> auto[,c(-1,-9)]  
> auto[,-c(1,9)]
```

However, this subsetting approach does not work while specifying variables using names. For example, we cannot use `-c("No", "car_name")`. Instead, use `%in%` with `!` (negation) to exclude variables:

```
> auto[, !names(auto) %in% c("No", "car_name")]
```

Selecting based on multiple values

Select all cars with `mpg = 15` or `mpg = 20`:

```
> auto[auto$mpg %in% c(15,20),c("car_name","mpg")]
```

Selecting using logical vector

You can specify the cases (rows) and variables you want to retrieve using boolean vectors.

In the following example, R returns the first and second cases, and for each, we get the third variable alone. R returns the elements corresponding to `TRUE`:

```
> auto[1:2,c(FALSE,FALSE,TRUE)]
```

You can use the same approach for rows also.

If the lengths do not match, R recycles through the boolean vector. However, it is always a good practice to match the size.

Splitting a dataset

When we have categorical variables, we often want to create groups corresponding to each level and to analyze each group separately to reveal some significant similarities and differences between groups.

The `split` function divides data into groups based on a factor or vector. The `unsplit()` function reverses the effect of `split`.

Getting ready

Download the files for this chapter and store the `auto-mpg.csv` file in your R working directory. Read the file using the `read.csv` command and save in the `auto` variable:

```
> auto <- read.csv("auto-mpg.csv", stringsAsFactors=FALSE)
```

How to do it...

Split cylinders using the following command:

```
> carslist <- split(auto, auto$cylinders)
```

How it works...

The `split(auto, auto$cylinders)` function returns a list of data frames with each data frame corresponding to the cases for a particular level of `cylinders`. To reference a data frame from the list, use the `[` notation. Here, `carslist[1]` is a list of length 1 consisting of the first data frame that corresponds to three cylinder cars, and `carslist[[1]]` is the associated data frame for three cylinder cars.

```
> str(carslist[1])
List of 1
$ 3:'data.frame': 4 obs. of 9 variables:
 ..$ No          : int [1:4] 2 199 251 365
 ..$ mpg         : num [1:4] 19 18 23.7 21.5
 ..$ cylinders   : int [1:4] 3 3 3 3
 ..$ displacement: num [1:4] 70 70 70 80
 ..$ horsepower  : int [1:4] 97 90 100 110
 ..$ weight      : int [1:4] 2330 2124 2420 2720
 ..$ acceleration: num [1:4] 13.5 13.5 12.5 13.5
 ..$ model_year  : int [1:4] 72 73 80 77
 ..$ car_name    : chr [1:4] "mazda rx2 coupe" "maxda rx3"
 "mazda rx-7 gs" "mazda rx-4"
```

```
> names(carslist[[1]])  
  
[1] "No"           "mpg"          "cylinders"    "displacement"  
[5] "horsepower"   "weight"        "acceleration" "model_year"  
[9] "car_name"
```

Creating random data partitions

Analysts need an unbiased evaluation of the quality of their machine learning models. To get this, they partition the available data into two parts. They use one part to build the machine learning model and retain the remaining data as "hold out" data. After building the model, they evaluate the model's performance on the hold out data. This recipe shows you how to partition data. It separately addresses the situation when the target variable is numeric and when it is categorical. It also covers the process of creating two partitions or three.

Getting ready

If you have not already done so, make sure that the `BostonHousing.csv` and `boston-housing-classification.csv` files from the code files of this chapter are in your R working directory. You should also install the `caret` package using the following command:

```
> install.packages("caret")  
> library(caret)  
> bh <- read.csv("BostonHousing.csv")
```

How to do it...

You may want to develop a model using some machine learning technique (like linear regression or KNN) to predict the value of the median of a home in Boston neighborhoods using the data in the `BostonHousing.csv` file. The `MEDV` variable will serve as the target variable.

Case 1 – numerical target variable and two partitions

To create a training partition with 80 percent of the cases and a validation partition with the rest, use the following code:

```
> trg.idx <- createDataPartition(bh$MEDV, p = 0.8, list = FALSE)  
> trg.part <- bh[trg.idx, ]  
> val.part <- bh[-trg.idx, ]
```

After this, the `trg.part` and `val.part` variables contain the training and validation partitions, respectively.

Case 2 – numerical target variable and three partitions

Some machine learning techniques require three partitions because they use two partitions just for building the model. Therefore, a third (test) partition contains the "hold-out" data for model evaluation.

Suppose we want a training partition with 70 percent of the cases, and the rest divided equally among validation and test partitions, use the following commands:

```
> trg.idx <- createDataPartition(bh$MEDV, p = 0.7, list = FALSE)
> trg.part <- bh[trg.idx, ]
> temp <- bh[-trg.idx, ]
> val.idx <- createDataPartition(temp$MEDV, p = 0.5, list = FALSE)
> val.part <- temp[val.idx, ]
> test.part <- temp[-val.idx, ]
```

Case 3 – categorical target variable and two partitions

Instead of a model to predict a numerical value like MEDV, you may need to create partitions for a classification application. The `boston-housing-classification.csv` file has a `MEDV_CAT` variable that categorizes the median values into *HIGH* or *LOW* and is suitable for a classification algorithm.

For a 70–30 split use the following commands:

```
> bh2 <- read.csv("boston-housing-classification.csv")
> trg.idx <- createDataPartition(bh2$MEDV_CAT, p=0.7, list =
  FALSE)
> trg.part <- bh2[trg.idx, ]
> val.part <- bh2[-trg.idx, ]
```

Case 4 – categorical target variable and three partitions

For a 70–15–15 split (training, validation, test) use the following commands:

```
> bh3 <- read.csv("boston-housing-classification.csv")
> trg.idx <- createDataPartition(bh3$MEDV_CAT, p=0.7, list =
  FALSE)
> trg.part <- bh3[trg.idx, ]
> temp <- bh3[-trg.idx, ]
> val.idx <- createDataPartition(temp$MEDV_CAT, p=0.5, list =
  FALSE)
> val.part <- temp[val.idx, ]
> test.part <- temp[-val.idx, ]
```

How it works...

The `createDataPartition()` function randomly selects row indices from the array supplied as its first argument. Rather than selecting randomly from the entire data frame, it does a more intelligent sampling as we now describe.

If supplied with a *numeric* vector as the first argument, then `createDataPartition()` applies the random selection process by percentile groups so as to get a good sampling of rows from the entire range of the target variable. This avoids the situation that can result from a completely random sampling whereby the training partition does not have a good representation from some segments of the target variable's whole range. By default, it considers five groups, but we can control this through the optional `groups` argument.

If supplied with a vector of *factors*, the function randomly samples for each value of the factor from the cases, thereby ensuring a good representation of all factor values in the training partition.

The `list` argument controls whether we want the output as a list or as a vector.

To avoid keeping duplicate data in both the original data frame as well as in the two data partitions, you can work with just the indices generated and refer to `bh[trg.idx,]` for the training partition and `bh[-trg.idx,]` for the validation partition.

When you have large data files, repeated subsetting may be inefficient and you may want to copy the data into the partitions up front.

There's more...

We discuss some additional information on data partitioning in this section.

Using a convenience function for partitioning

Rather than typing out the detailed steps each time, you can simplify the process by creating the following functions:

```
rda.cb.partition2 <- function(ds, target.index, prob) {  
  library(caret)  
  train.idx <- createDataPartition(y=ds[,target.index],  
    p = prob, list = FALSE)  
  list(train = ds[train.idx, ], val = ds[-train.idx, ])  
}
```

```
rda.cb.partition3 <- function(ds,
    target.index, prob.train, prob.val) {
  library(caret)
  train.idx <- createDataPartition(y=ds[,target.index],
    p = prob.train, list = FALSE)
  train <- ds[train.idx, ]
  temp <- ds[-train.idx, ]
  val.idx <- createDataPartition(y=temp[,target.index],
    p = prob.val/(1-prob.train), list = FALSE)
  list(train = ds[train.idx, ],
    val = temp[val.idx, ], test = temp[-val.idx, ])
}
```

With the two preceding functions in place, you can write the following single line to create two partitions (80 percent, 20 percent) of a data frame:

```
dat1 <- rda.cb.partition2(bh, 14, 0.8)
```

You can do the following to get three partitions (70 percent, 15 percent, 15 percent):

```
dat2 <- rda.cb.partition3(bh, 14, 0.7, 0.15)
```

The `rda.cb.partition2()` and `rda.cb.partition3()` functions return a list with two and three components, respectively. To use the training and validation partitions from `dat1`, you can refer to `dat1$train` and `dat1$val`. The same applies to `dat2`; to get the test partition from `dat2`, use `dat2$test`.

Sampling from a set of values

To select a random sample of size 50 cases from a `bh` data frame without replacement, use the following command:

```
sam.idx <- sample(1:nrow(bh), 50, replace = FALSE)
```

Generating standard plots such as histograms, boxplots, and scatterplots

Before even embarking on any numerical analyses, you may want to get a good idea about the data through a few quick plots. Although the base R system supports powerful graphics, we will generally turn to other plotting options like `lattice` and `ggplot` for more advanced plots. Therefore, we cover only the simplest forms of basic graphs.

Getting ready

If you have not already done so, download the data files for this chapter and ensure that they are available in your R environment's working directory and run the following commands:

```
> auto <- read.csv("auto-mpg.csv")
>
> auto$cylinders <- factor(auto$cylinders, levels = c(3,4,5,6,8),
  labels = c("3cyl", "4cyl", "5cyl", "6cyl", "8cyl"))
> attach(auto)
```

How to do it...

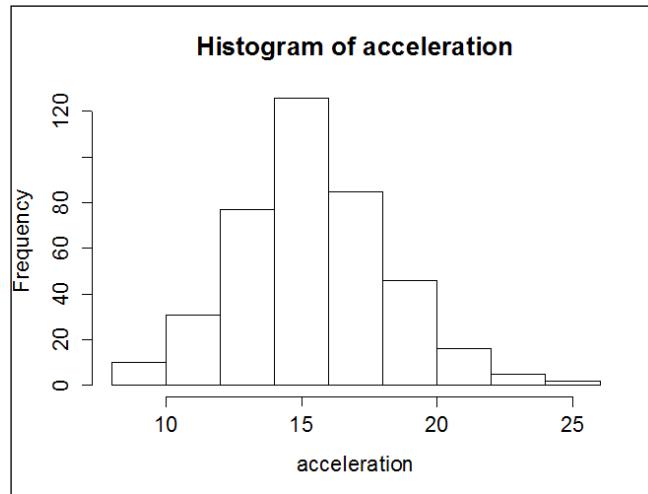
In this recipe, we cover histograms, boxplots, scatterplots and scatterplot matrices.

Histograms

Generate a histogram for acceleration:

```
> hist(acceleration)
```

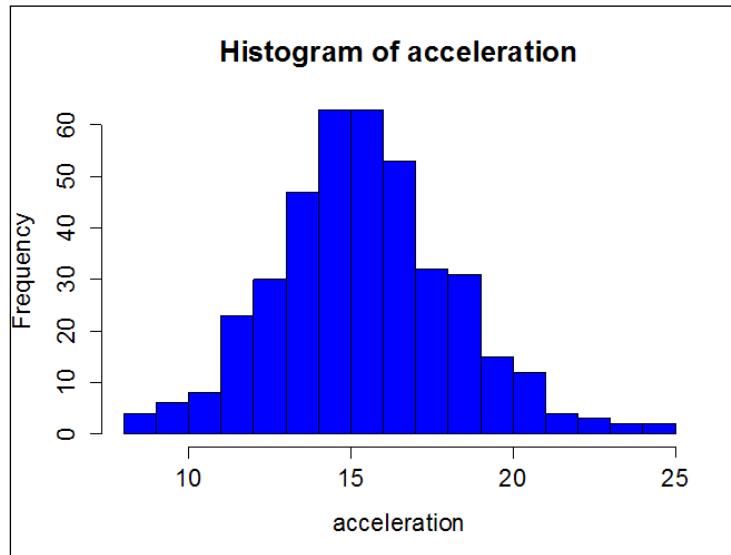
R determines the various properties of the generated graph (like bin sizes, axes scales, axes titles, chart title, bar colors,...) automatically. The following diagram shows the output of the preceding command:



You can customize everything. The following code shows some options:

```
> hist(acceleration, col="blue", xlab = "acceleration",
  main = "Histogram of acceleration", breaks = 15)
```

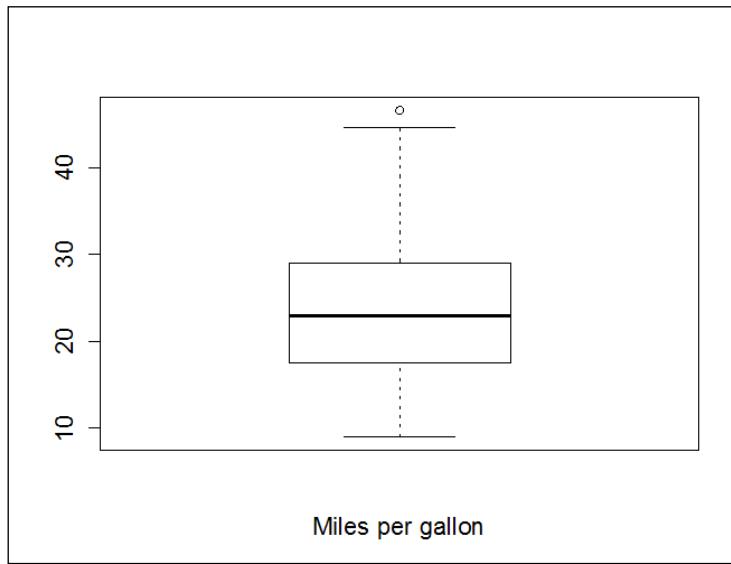
The histogram of acceleration can be seen in the following diagram:



Boxplots

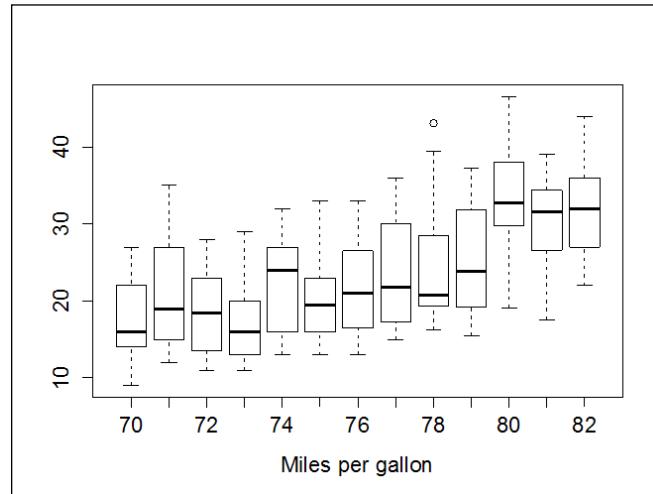
Create a boxplot for mpg using the following command:

```
> boxplot(mpg, xlab = "Miles per gallon")
```



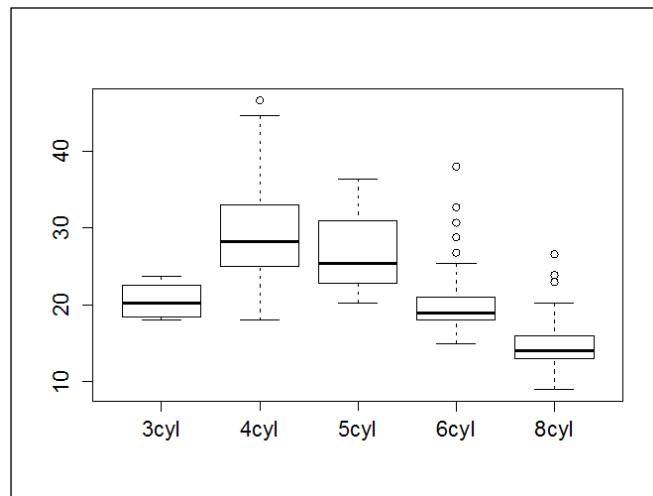
To generate boxplots for subsets within the whole dataset, you can use:

```
> boxplot(mpg ~ model_year, xlab = "Miles per gallon")
```



Create a boxplot of mpg by cylinders:

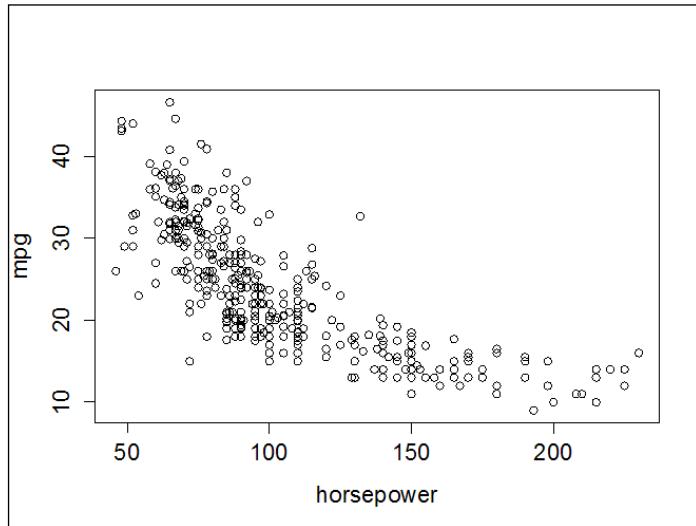
```
> boxplot(mpg ~ cylinders)
```



Scatterplots

Create a scatterplot for mpg by horsepower:

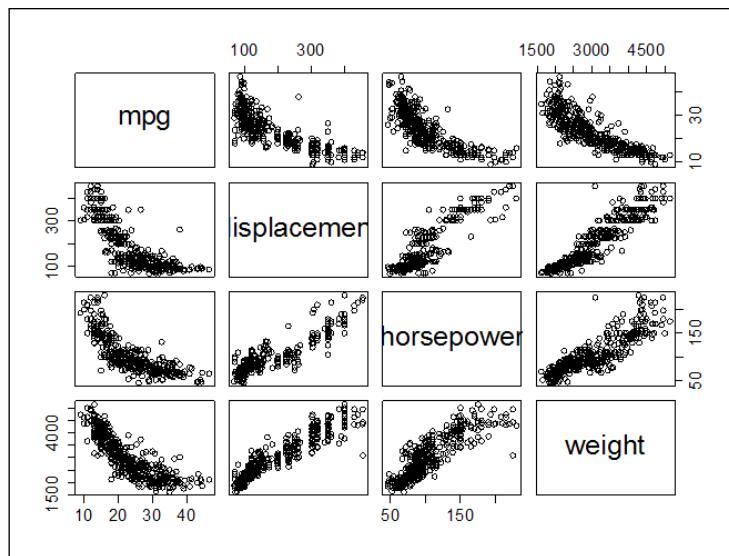
```
> plot(mpg ~ horsepower)
```



Scatterplot matrices

Create pair wise scatterplots for a set of variables:

```
> pairs(~mpg+displacement+horsepower+weight)
```



How it works...

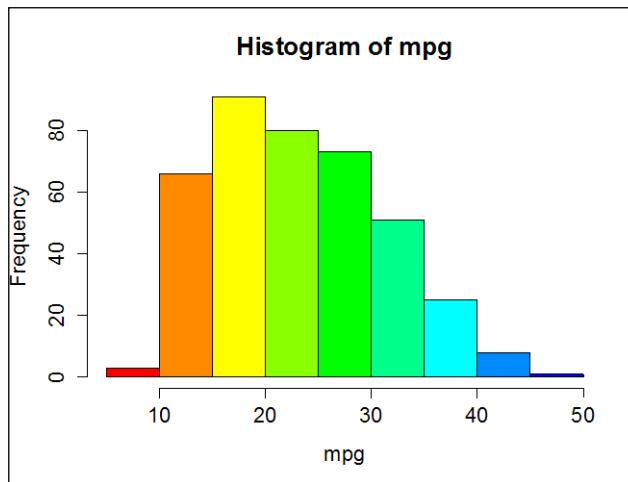
Here, we describe how the preceding lines of code work.

Histograms

By default, the `hist()` function automatically determines the number of bars to display based on the data. The `breaks` argument controls this.

You can also use a palette of colors instead of a single color using the following command:

```
> hist(mpg, col = rainbow(12))
```



The `rainbow()` function returns a color palette with the color spectrum broken up into the number of distinct colors specified, and the `hist()` function uses one color for each bar.

Boxplots

You can either give a simple vector or a formula (like `auto$mpg ~ auto$cylinders` in the preceding example) as the first argument to the `boxplot()` function. In the latter case, it creates separate boxplots for every distinct level of the right-hand side variable.

There's more...

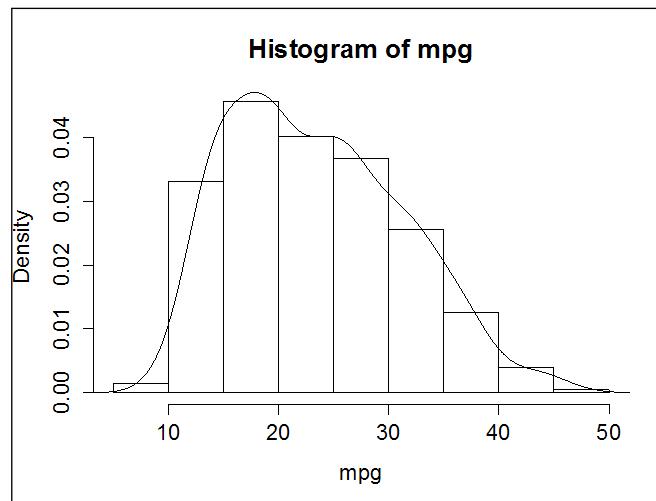
You can overlay plots and color specific points differently. We show some useful options in this section.

Overlay a density plot on a histogram

Histograms are very sensitive to the number of bins used. Kernel density plots give a smoother and more accurate picture of the distribution. Usually, we overlay a density plot on a histogram using the `density()` function which creates the density plot.

If invoked by itself, the `density()` function only produces the density plot. To overlay it on the histogram, we use the `lines()` function which does not erase the current chart and instead overlays the existing plot. Since the density plot plots relative frequencies (approximating a probability density function), we need to ensure that the histogram also shows relative frequencies. The `prob=TRUE` argument achieves this:

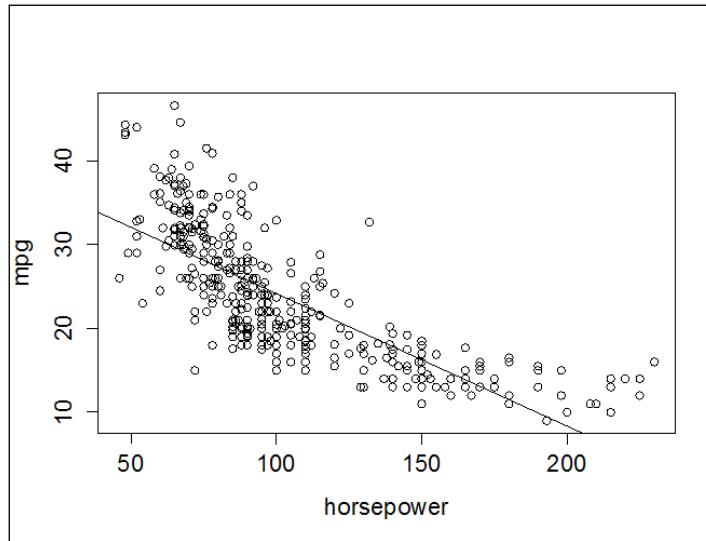
```
hist(mpg, prob=TRUE)
lines(density(mpg))
```



Overlay a regression line on a scatterplot

The following code first generates the scatterplot. It then builds the regression model using `lm` and uses the `abline()` function to overlay the regression line on the existing scatterplot:

```
> plot(mpg ~ horsepower)
> reg <- lm(mpg ~ horsepower)
> abline(reg)
```



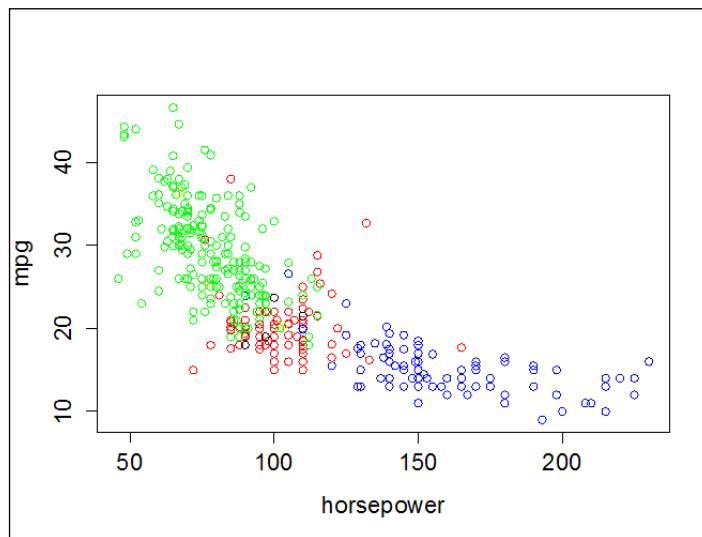
Color specific points on a scatterplot

Using the following code, you can first generate the scatterplot and then color the points corresponding to different values of cylinders with different colors. Note that `mpg` and `weight` are in different orders in the `plot` and `points` function invocations. This is because in `plot`, we ask the system to plot `mpg` as a function of `weight`, whereas in the `points` function, we just supply a set of (x,y) coordinates to `plot`:

```
> # first generate the empty plot
> # to get the axes for the whole dat
> plot(mpg ~ horsepower, type = "n")
> # Then plot the points in different colors
> with(subset(auto, cylinders == "8cyl"),
  points(horsepower, mpg, col = "blue"))
> with(subset(auto, cylinders == "6cyl"),
  points(horsepower, mpg, col = "red"))
> with(subset(auto, cylinders == "5cyl"),
  points(horsepower, mpg, col = "yellow"))
```

```
> with(subset(auto, cylinders == "4cyl"),
      points(horsepower, mpg, col = "green"))
> with(subset(auto, cylinders == "3cyl"),
      points(horsepower, mpg))
```

The preceding commands produce the following output:



Generating multiple plots on a grid

We often want to see plots side by side for comparisons. This recipe shows how we can achieve this.

Getting ready

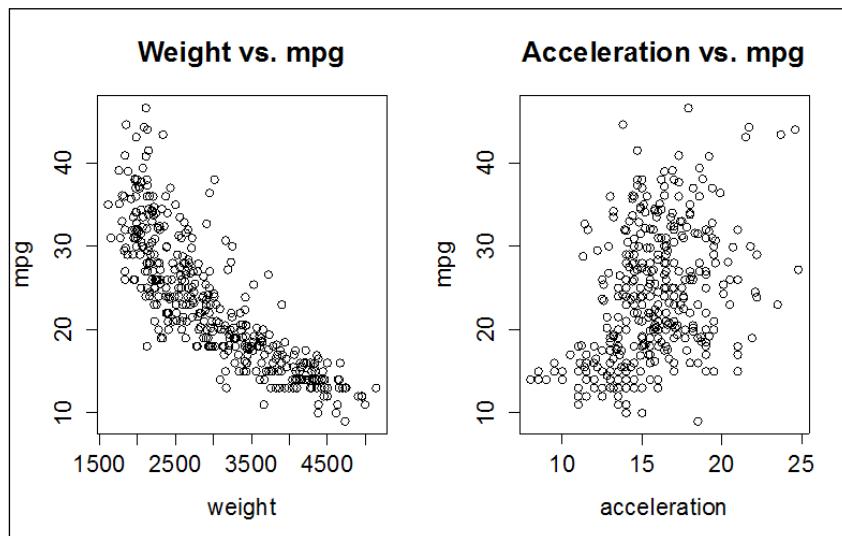
If you have not already done so, download the data files for this chapter and ensure that they are available in your R environment's working directory. Once this is done, run the following commands:

```
> auto <- read.csv("auto-mpg.csv")
> cylinders <- factor(cylinders,
  levels = c(3,4,5,6,8),
  labels = c("3cyl", "4cyl", "5cyl", "6cyl", "8cyl"))
> attach(auto)
```

How to do it...

You may want to generate two side-by-side scatterplots from the data in `auto-mpg.csv`. Run the following commands:

```
> # first get old graphical parameter settings  
> old.par = par()  
> # create a grid of one row and two columns  
> par(mfrow = c(1,2))  
> with(auto, {  
    plot(mpg ~ weight, main = "Weight vs. mpg")  
    plot(mpg ~ acceleration, main = "Acceleration vs. mpg")  
})  
>  
> # reset par back to old value so that subsequent  
> # graphic operations are unaffected by our settings  
> par(old.par)
```



How it works...

The `par(mfrow = c(1, 2))` function call creates a grid with one row and two columns. The subsequent invocations of the `plot()` function fills the charts into these grid locations row by row. Alternately, you can specify `par(mfcol = ...)` to specify the grid. In this case, the grid is created as in the case of `mfrow`, but the grid cells get filled in column by column.

Graphics parameters

In addition to creating a grid for graphics, you can use the `par()` function to specify numerous graphics parameters to control all aspects. Check the documentation if you need something specific.

See also...

- ▶ Generating standard plots such as histograms, boxplots and scatterplots

Selecting a graphics device

R can send its output to several different graphic devices to display graphics in different formats. By default, R prints to the screen. However, we can save graphs in the following file formats as well: PostScript, PDF, PNG, JPEG, Windows metafile, Windows BMP, and so on.

Getting ready

If you have not already done so, download the data files for this chapter and ensure that the `auto-mpg.csv` file is available in your R environment's working directory and run the following commands:

```
> auto <- read.csv("auto-mpg.csv")
>
> cylinders <- factor(cylinders, levels = c(3,4,5,6,8),
  labels = c("3cyl", "4cyl", "5cyl", "6cyl", "8cyl"))
> attach(auto)
```

How to do it...

To send the graphic output to the computer screen, you have to do nothing special. For other devices, you first open the device, send your graphical output to it, and then close the device to close the corresponding file.

To create a PostScript file use:

```
> postscript(file = "auto-scatter.ps")
> boxplot(mpg)
> dev.off()

> pdf(file = "auto-scatter.pdf")
> boxplot(mpg)
> dev.off()
```

How it works...

Invoking the function appropriate for the graphics device (like `postscript()` and `pdf()`) opens the file for output. The actual plotting operation writes to the device (file), and the `dev.off()` function closes the device (file).

See also...

- ▶ Generating standard plots such as histograms, boxplots and scatterplots

Creating plots with the lattice package

The `lattice` package produces Trellis plots to capture multivariate relationships in the data. Lattice plots are useful for looking at complex relationships between variables in a dataset. For example, we may want to see how `y` changes with `x` across various levels of `z`. Using the `lattice` package, we can draw histograms, boxplots, scatterplots, dot plots and so on. Both plotting and annotation are done in one single call.

Getting ready

Download the files for this chapter and store the `auto-mpg.csv` file in your R working directory. Read the file using the `read.csv` function and save in the `auto` variable. Convert cylinders into a factor variable:

```
> auto <- read.csv("auto-mpg.csv", stringsAsFactors=FALSE)
> cyl.factor <- factor(auto$cylinders, labels=c("3cyl", "4cyl",
  "5cyl", "6cyl", "8cyl"))
```

How to do it...

To create plots with lattice package, follow these steps:

1. Load the lattice package:

```
> library(lattice)
```

2. Draw a boxplot:

```
> bwplot(~auto$mpg|cyl.factor, main="MPG by Number of
  Cylinders", xlab="Miles per Gallon")
```

3. Draw a scatterplot:

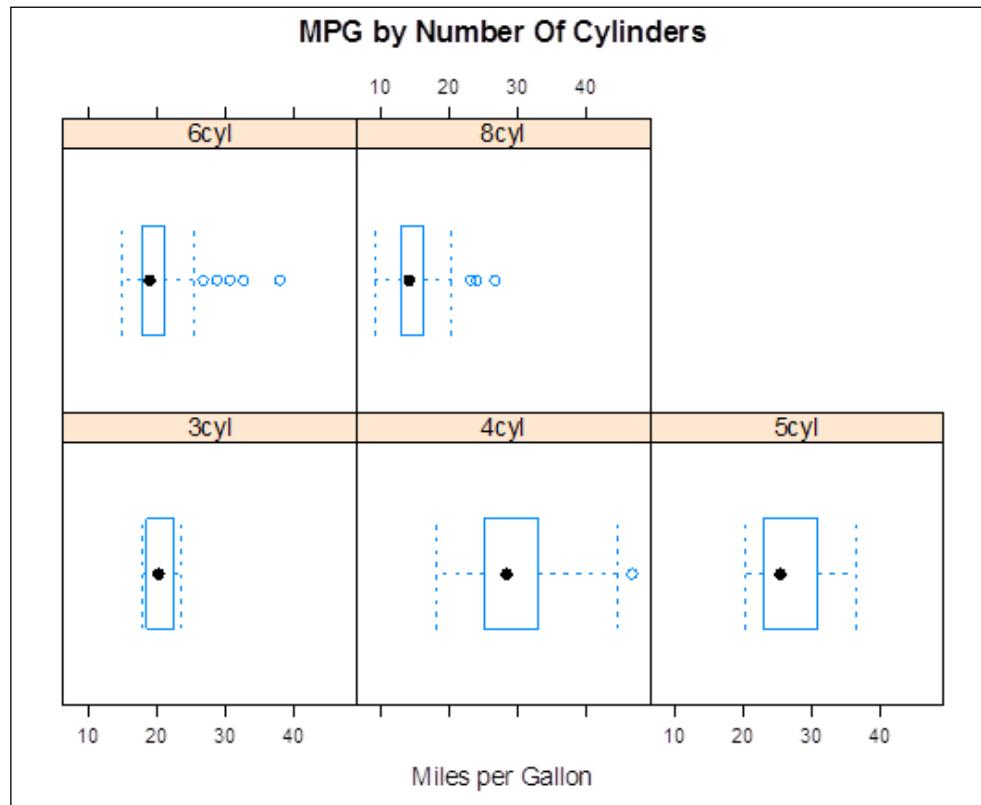
```
> xyplot(mpg~weight|cyl.factor, data=auto,
  main="Weight Vs MPG by Number of Cylinders",
  ylab="Miles per Gallon", xlab="Car Weight")
```

How it works...

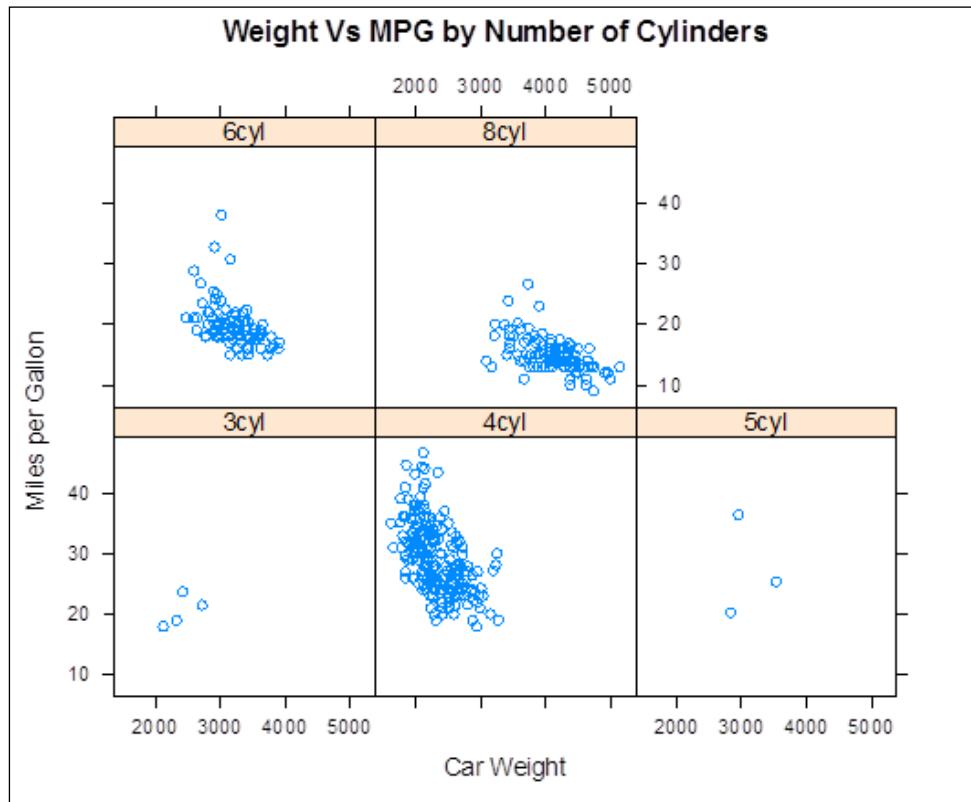
Lattice plot commands comprise the following four parts:

- ▶ **Graph type:** This can be a `bwplot`, `xyplot`, `densityplot`, `splom`, and so on
- ▶ **Formula:** Variables and factor variables separated by `|`
- ▶ **Data:** A data frame containing values
- ▶ **Annotations:** These include caption, x axis label, and y axis label

In the boxplot step 2, the `~auto$mpg | cyl`. formula instructs `lattice` to make the plot with `mpg` on the x axis grouped by factors representing cylinders. Here, we have not specified any variable for the y axis. For boxplots and density plots, we need not specify the y axis. The output for boxplot resembles the following diagram:



In the scatterplot, the `xyplot` function and the `mpg~weight | cyl`.`factor` formula instructs `lattice` to make the plot with `weight` on the x axis and `mpg` on the y axis grouped by factors representing cylinders. For `xyplot`, we need to provide two variables; otherwise, R will produce an error. The scatterplot output is seen as follows:



There's more...

Lattice plots provide default options to bring out the relationship between multiple variables. More options can be added to these plots to enhance the graphs.

Adding flair to your graphs

By default, `lattice` assigns the panel height and width based on the screen device. The plots use a default color scheme. However, these can be customized to your needs.

You should change the color scheme of all `lattice` plots before executing the plot command. The color scheme affects all Trellis plots made with the `lattice` package:

```
> trellis.par.set(theme = col.whitebg())
```

Panels occupy the entire output window. This can be controlled with `aspect`. The layout determines the number of panels on the x axis and how they are stacked. Add these to the `plot` function call:

```
> bwplot(~mpg|cyl.factor, data=auto,main="MPG by Number Of Cylinders",
  xlab="Miles per Gallon",layout=c(2,3),aspect=1)
```

See also...

- ▶ Generating standard plots such as histograms, boxplots, and scatterplots

Creating plots with the `ggplot2` package

`ggplot2` graphs are built iteratively, starting with the most basic plot. Additional layers are chained with the `+` sign to generate the final plot.

To construct a plot we need at least data, aesthetics (color, shape, and size), and **geoms** (points, lines, and smooth). The geoms determine which type of graph is drawn. Facets can be added for conditional plots.

Getting ready

Download the files for this chapter and copy the `auto-mpg.csv` file to your R working directory. Read the file using the `read.csv` command and save in the `auto` variable. Convert `cylinders` into a `factor` variable. If you have not done so already, install the `ggplot2` package as follows:

```
> install.packages("ggplot2")
> library(ggplot2)
> auto <- read.csv("auto-mpg.csv", stringsAsFactors=FALSE)
> auto$cylinders <- factor(auto$cylinders, labels=c("3cyl", "4cyl",
  "5cyl", "6cyl", "8cyl"))
```

How to do it...

To create plots with the `ggplot2` package, follow these steps:

1. Draw the initial plot:

```
> plot <- ggplot(auto, aes(weight, mpg))
```

2. Add layers:

```
> plot + geom_point()  
> plot + geom_point(alpha=1/2, size=5,  
aes(color=factor(cylinders))) +  
geom_smooth(method="lm", se=FALSE, col="green") +  
facet_grid(cylinders~.) +  
theme_bw(base_family = "Calibri", base_size = 10) +  
labs(x = "Weight") +  
labs(y = "Miles Per Gallon") +  
labs(title = "MPG Vs Weight")
```

How it works...

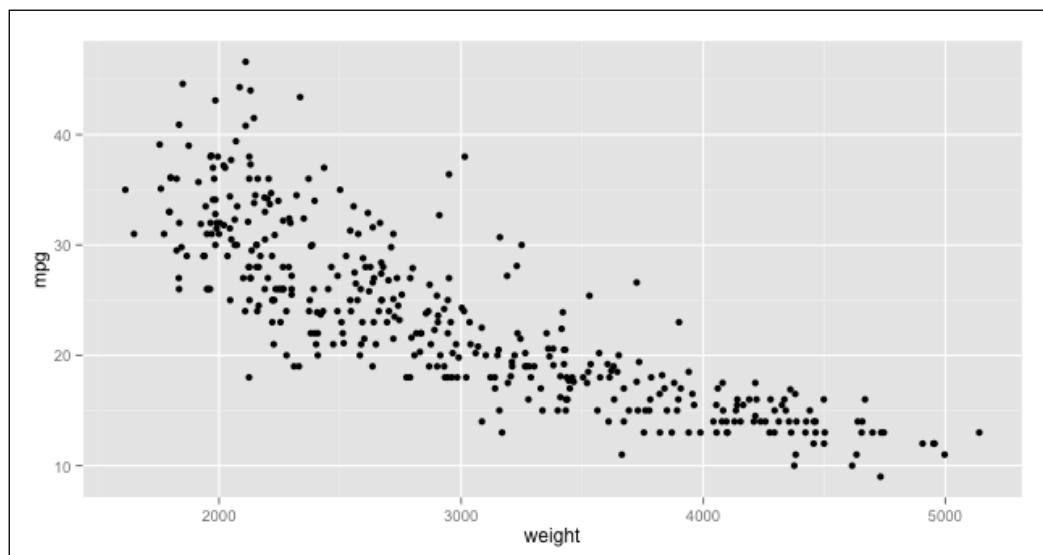
Let's start from the top and discuss some variations:

```
> plot <- ggplot(auto, aes(weight, mpg))
```

First, we draw the plot. At this point the graph is not printed, since we have not added layers to it. ggplot needs at least one layer to display the graph:

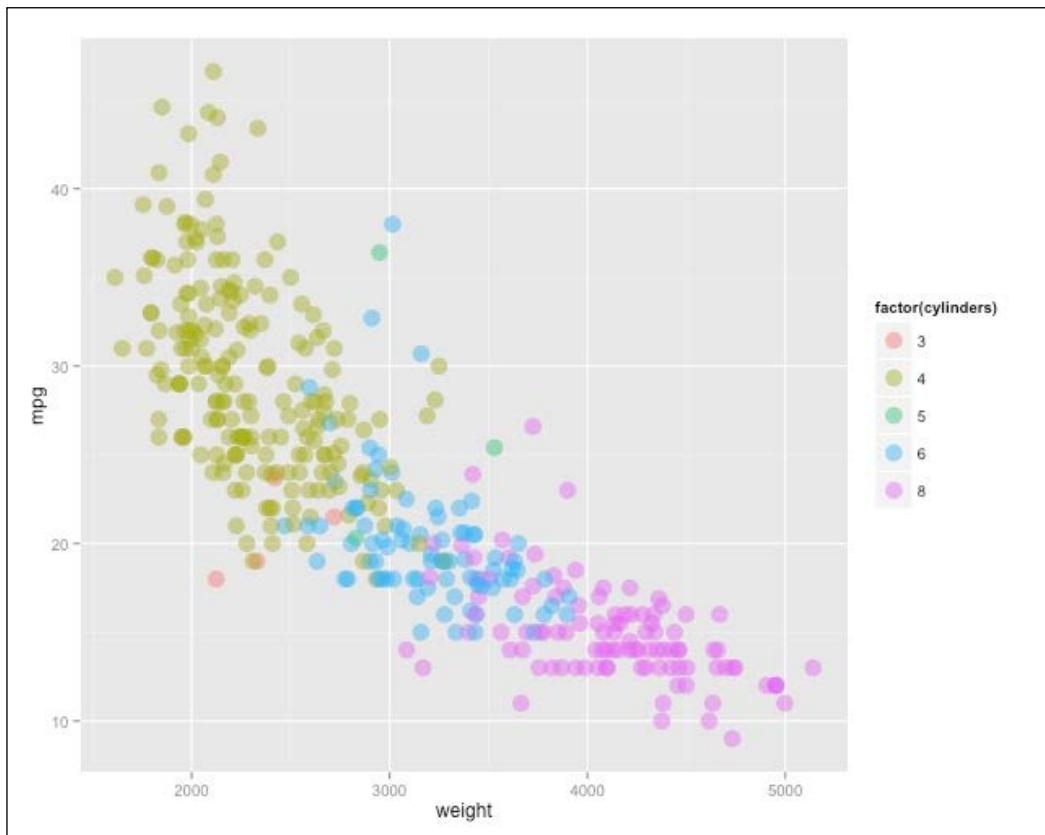
```
> plot + geom_point()
```

This plots the points to produce the scatterplot as follows:



We can use various arguments to control how the points appear—alpha for the intensity of the dots, color of the dots, the size and shape of the dots. We can also use the aes argument to add aesthetics to this layer and this produces the plot as follows:

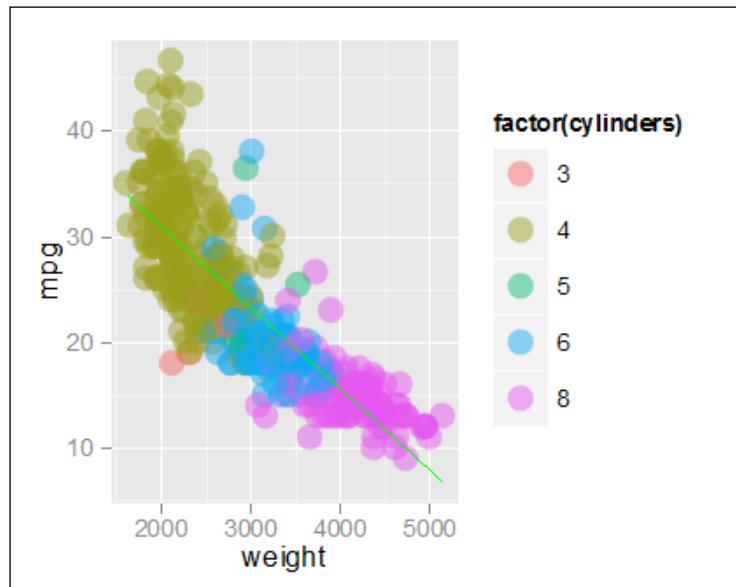
```
> plot + geom_point(alpha=1/2, size=5,  
aes(color=factor(cylinders)))
```



Append the following code to the preceding command:

```
+ geom_smooth(method="lm", se=FALSE, col="green")
```

Adding `geom_smooth` helps to see a pattern. The `method=lm` argument uses a linear model as the smoothing method. The `se` argument is set to `TRUE` by default and hence displays the confidence interval around the smoothed line. This supports aesthetics similar to `geom_point`. In addition, we can also set the `linetype`. The output obtained resembles the following diagram:

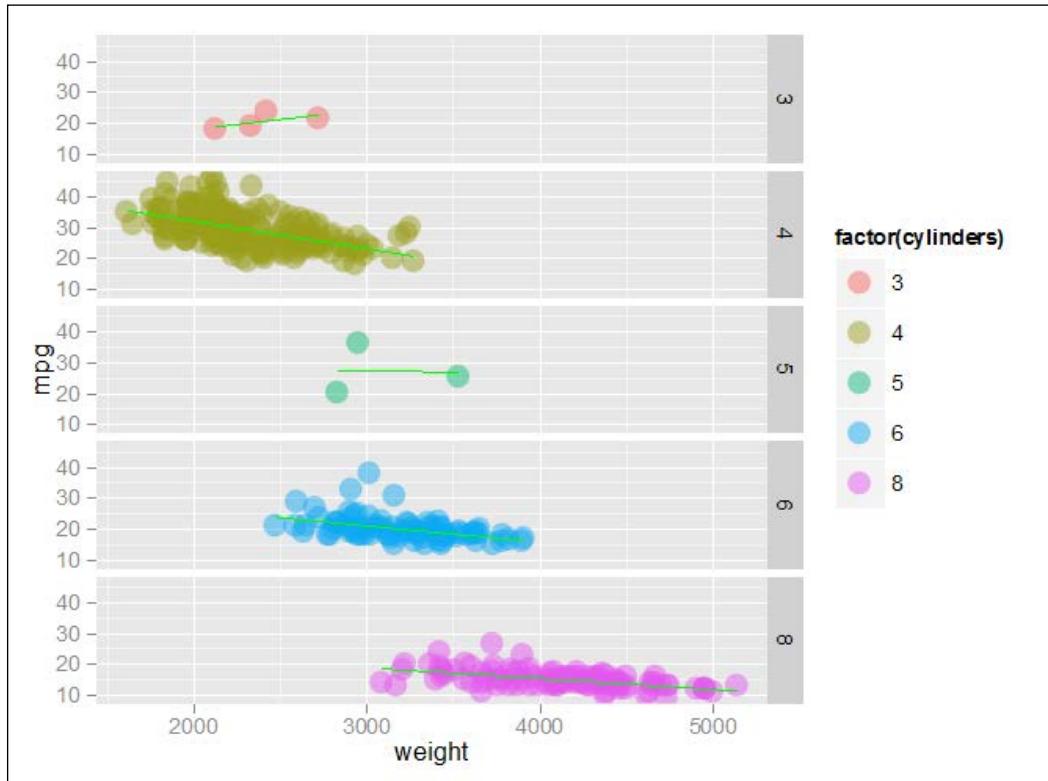


By default, the `geom_smooth` function uses two different smoothing approaches based on the number of observations. If the number of observations exceeds 1000, it uses `gam` smoothing, `loess` otherwise. Given the familiarity with linear models, people mostly use the `lm` smoothing.

Append the following code to the preceding command:

```
+ facet_grid(cylinders~.)
```

This adds additional dimensions to the graph using facets. We can add cylinders as a new dimension to the graph. Here, we use the simple `facet_grid` function. If we want to add more dimensions, we can use `facet_wrap` and specify how to wrap the rows and columns shown as follows:

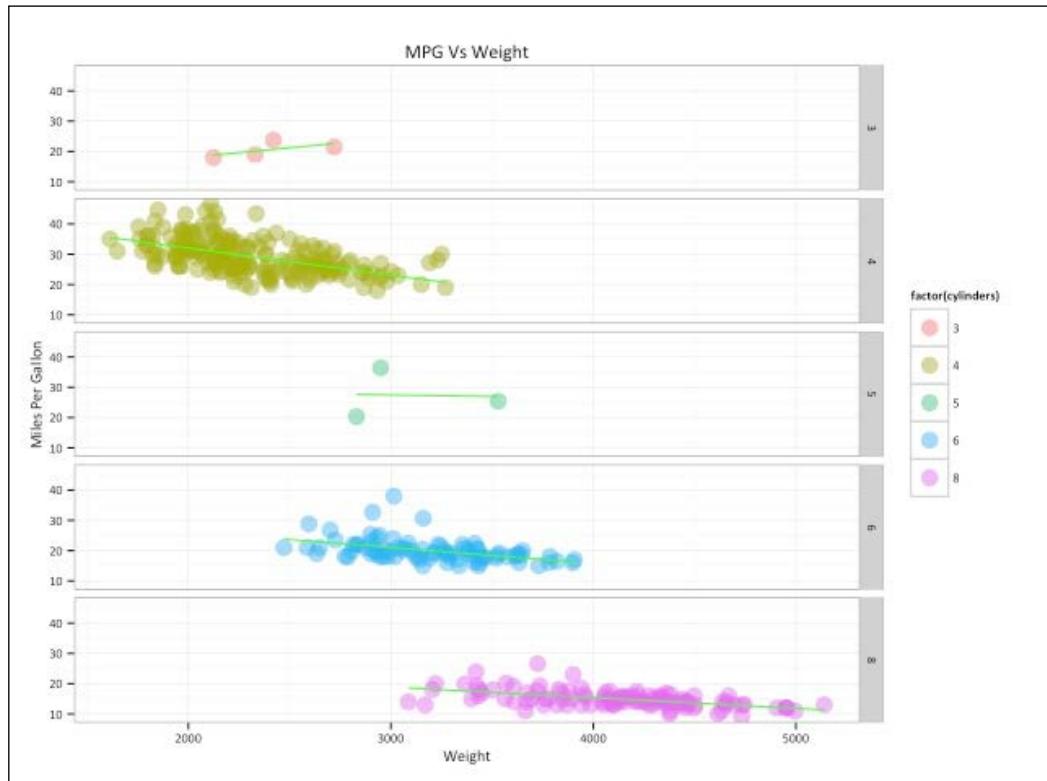


If we change to `facet_grid(~cylinders)`, the plots for each level of the cylinder are arranged horizontally.

Appending the following code adds annotations to get the final plot:

```
+ theme_bw(base_family = "Calibri", base_size = 10) + labs(x =  
"Weight") + labs(y = "Miles Per Gallon") + labs(title = "MPG Vs  
Weight")
```

The annotations added to get the final plot can be seen in the following diagram:



There's more...

The best way to learn ggplot is to try out different options to see how they impact the graph. Here, we describe a few additional variations to ggplot.

Graph using qplot

A simplistic version of ggplot is qplot and it uses the same ggplot2 package. qplot can also be chained with + to add additional layers in the plot. The generic form of qplot is as follows:

```
qplot(x, y, data=, color=, shape=, size=, alpha=, geom=, method=,
      formula=, facets=, xlim=, ylim=, xlab=, ylab=, main=, sub=)
```

For certain types of graphs such as histograms and bar charts, we need to supply only `x` (and can therefore omit `y`):

```
> # Boxplots of mpg by number of cylinders
> qplot(cylinders, mpg, data=auto, geom=c("jitter"),
  color=cylinders, fill=cylinders,
  main="Mileage by Number of Cylinders",
  xlab="", ylab="Miles per Gallon")
> # Regression of mpg by weight for each type of cylinders
> qplot(weight, mpg, data=auto, geom=c("point", "smooth"),
  method="lm", formula=y~x, color=cylinders,
  main="Regression of MPG on Weight")
```

Condition plots on continuous numeric variables

Normally, we condition plots on categorical variables. However, to add additional dimensions to an existing plot, you may want to incorporate a numeric variable. Although `qplot` will do this for us, the numerous values of the condition make the plot useless. You can make it categorical using the `cut` function as follows:

```
> # Cut with desired range
> breakpoints <- c(8,13,18,23)
> # Cut using Quantile function (another approach)
> breakpoints <- quantile(auto$acceleration, seq(0, 1, length
= 4), na.rm = TRUE)

> ## create a new factor variable using the breakpoints
> auto$accelerate.factor <- cut(auto$acceleration, breakpoints)
```

Now, we can use `auto$accelerate.factor` in the `qplot` function.

See also...

- ▶ Generating standard plots such as histograms, boxplots and scatterplots
- ▶ Creating plots with lattice package

Creating charts that facilitate comparisons

In large datasets, we often gain good insights by examining how different segments behave. The similarities and differences can reveal interesting patterns. This recipe shows how to create graphs that enable such comparisons.

Getting ready

If you have not already done so, download the book's files for this chapter and save the `daily-bike-rentals.csv` file in your R working directory. Read the data into R using the following command:

```
> bike <- read.csv("daily-bike-rentals.csv")
> bike$season <- factor(bike$season, levels = c(1,2,3,4),
  labels = c("Spring", "Summer", "Fall", "Winter"))
> attach(bike)
```

How to do it...

We base this recipe on the task of generating histograms to facilitate the comparison of bike rentals by season.

Using base plotting system

We first look at how to generate histograms of the count of daily bike rentals by season using R's base plotting system:

1. Set up a 2 X 2 grid for plotting histograms for the four seasons:

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

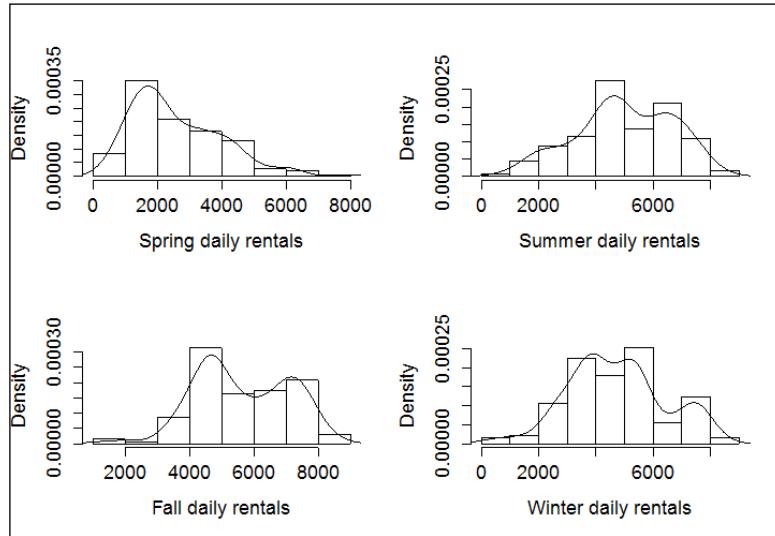
2. Extract data for the seasons:

```
> spring <- subset(bike, season == "Spring")$cnt
> summer <- subset(bike, season == "Summer")$cnt
> fall <- subset(bike, season == "Fall")$cnt
> winter <- subset(bike, season == "Winter")$cnt
```

3. Plot the histogram and density for each season:

```
> hist(spring, prob=TRUE,
  xlab = "Spring daily rentals", main = "")
> lines(density(spring))
>
> hist(summer, prob=TRUE,
  xlab = "Summer daily rentals", main = "")
> lines(density(summer))
>
> hist(fall, prob=TRUE,
  xlab = "Fall daily rentals", main = "")
> lines(density(fall))
>
> hist(winter, prob=TRUE,
  xlab = "Winter daily rentals", main = "")
> lines(density(winter))
```

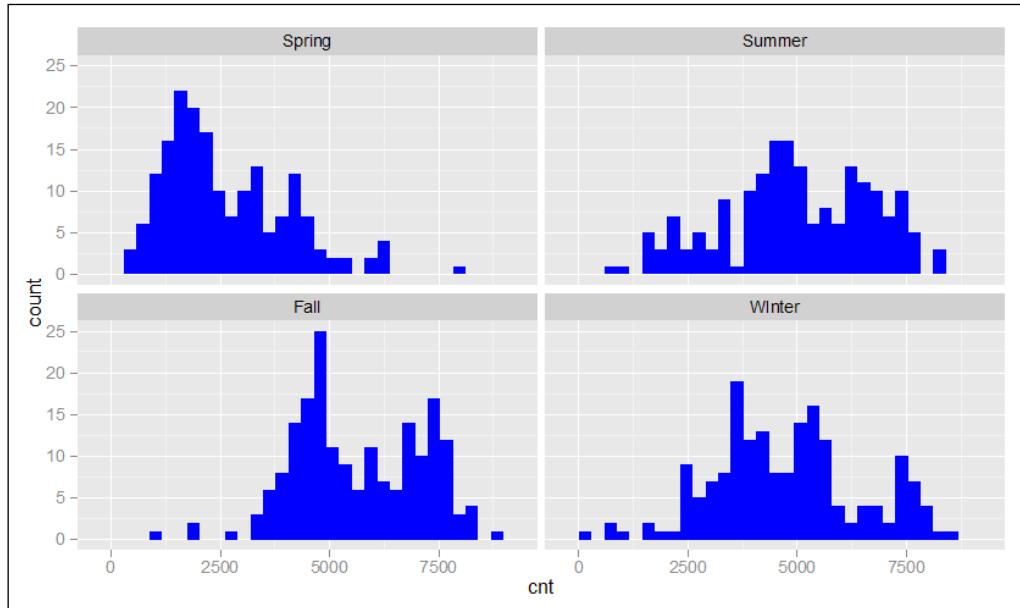
You get the following output that facilitates comparisons across the seasons:



Using ggplot2

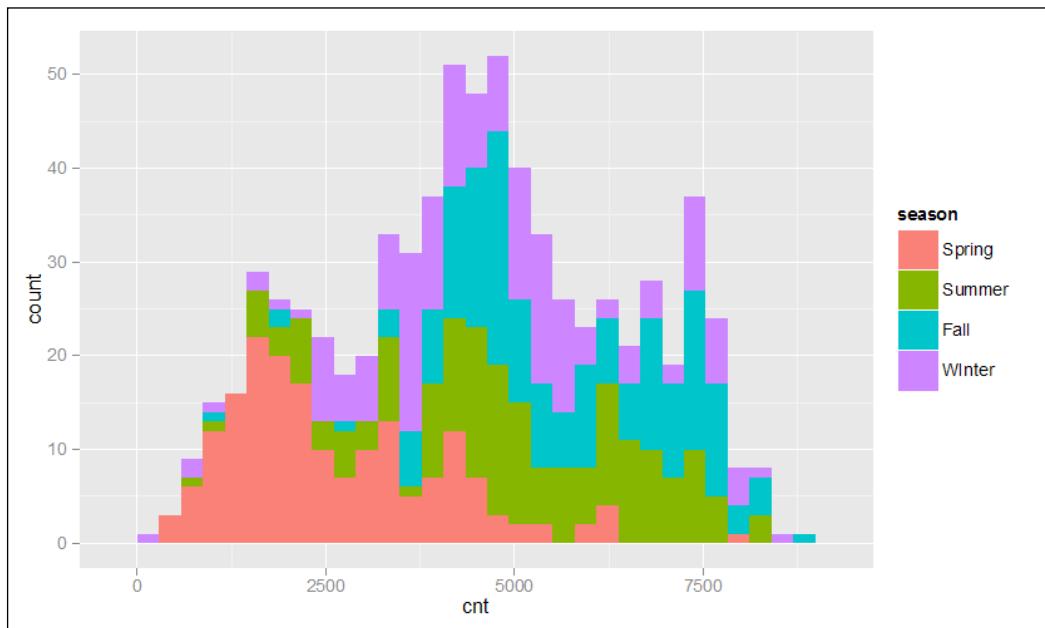
We can achieve much of the preceding results in a single command:

```
> qplot(cnt, data = bike) + facet_wrap(~ season, nrow=2) +
  geom_histogram(fill = "blue")
```



You can also combine all four into a single histogram and show the seasonal differences through coloring:

```
> qplot(cnt, data = bike, fill = season)
```



How it works...

When you plot a single variable with `qplot`, you get a histogram by default. Adding `facet` enables you to generate one histogram per level of the chosen facet. By default, the four histograms will be arranged in a single row. Use `facet_wrap` to change this.

There's more...

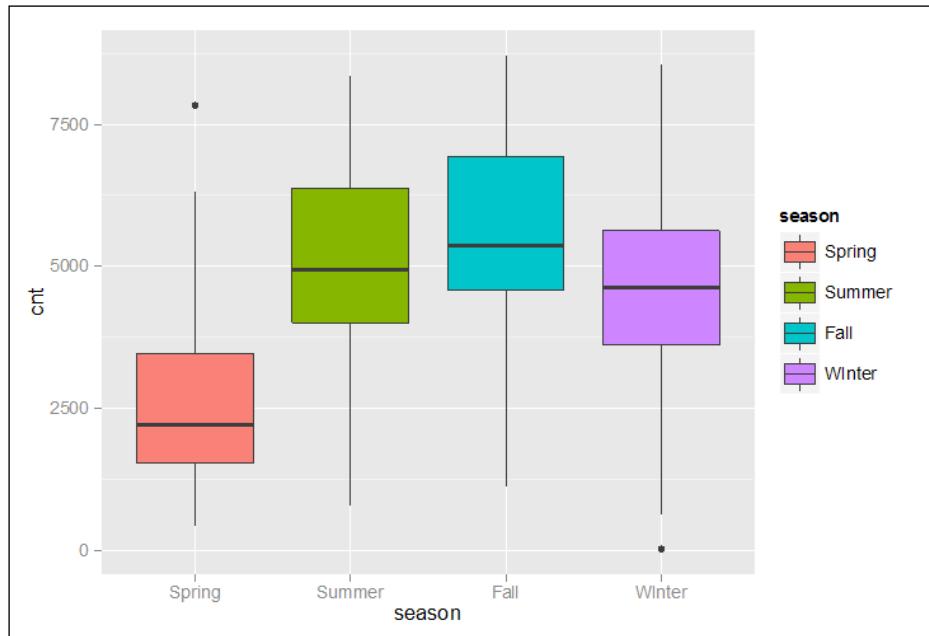
You can use `ggplot2` to generate comparative boxplots as well.

Creating boxplots with `ggplot2`

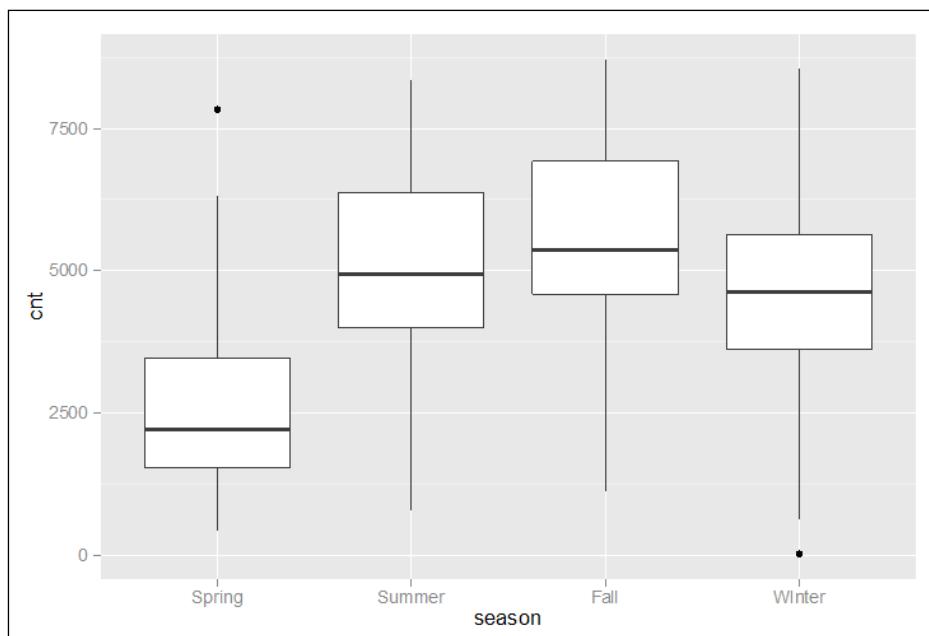
Instead of the default histogram, you can get a boxplot with either of the following two approaches:

```
> qplot(season, cnt, data = bike, geom = c("boxplot"), fill = season)
>
> ggplot(bike, aes(x = season, y = cnt)) + geom_boxplot()
```

The preceding code produces the following output:



The second line of the preceding code produces the following plot:



See also...

- ▶ Generating standard plots such as histograms, boxplots and scatterplots

Creating charts that help visualize a possible causality

When presenting data, rather than merely present information, we usually want to present an explanation of some phenomenon. Visualizing hypothesized causality helps to communicate our ideas clearly.

Getting ready

If you have not already done so, download the book's files for this chapter and save the `daily-bike-rentals.csv` file in your R working directory. Read the data into R as follows:

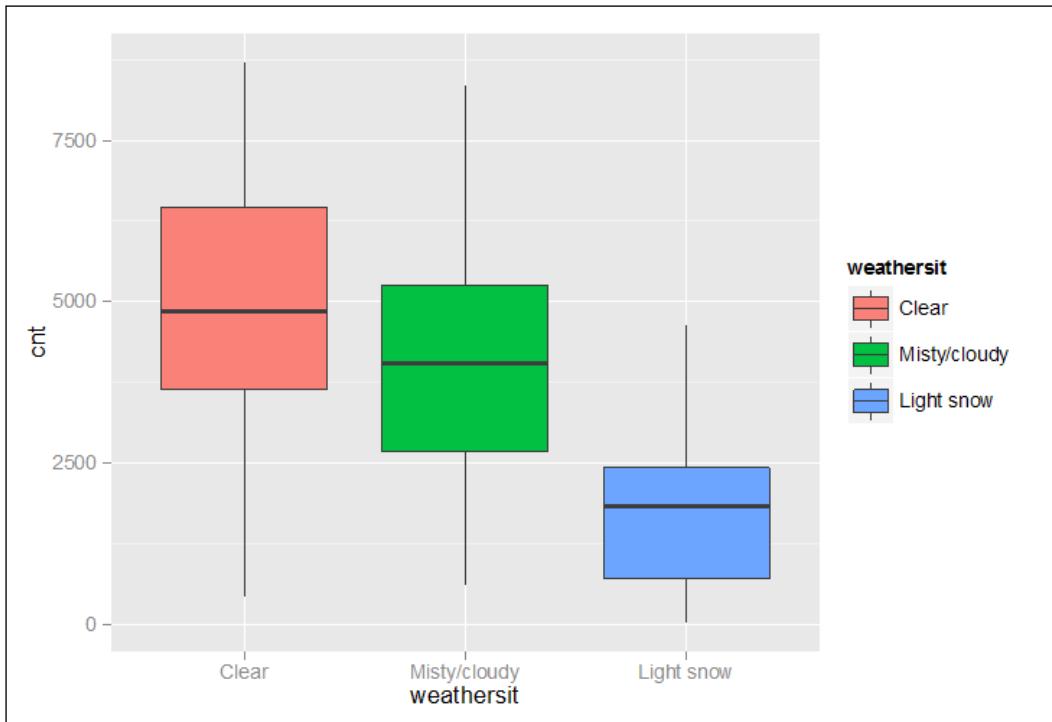
```
> bike <- read.csv("daily-bike-rentals.csv")
> bike$season <- factor(bike$season, levels = c(1,2,3,4),
  labels = c("Spring", "Summer", "Fall", "Winter"))
> bike$weathersit <- factor(bike$weathersit, levels = c(1,2,3),
  labels = c("Clear", "Misty/cloudy", "Light snow"))
> attach(bike)
```

How to do it...

With the bike rentals data, you can show a hypothesized causality between the weather situation and the number of rentals by drawing boxplots of rentals under different weather conditions:

```
> qplot(weather, cnt, data = bike, geom = c("boxplot"), fill =
  weather)
```

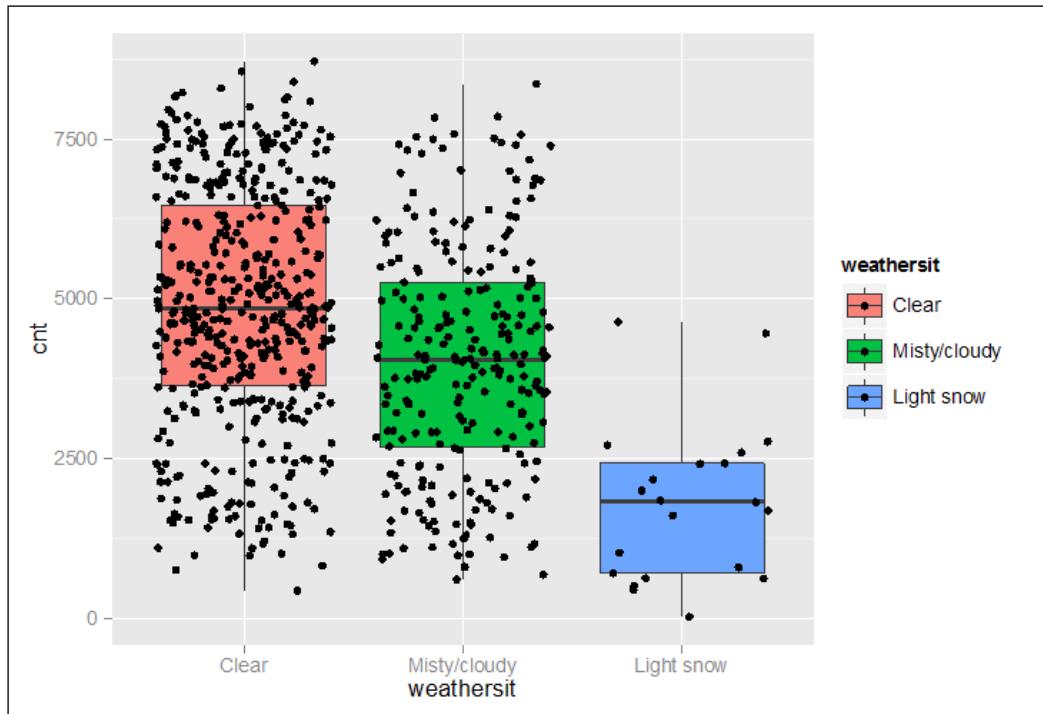
The preceding command produces the following output:



If you choose to, you can overlay the actual points as well; add "jitter" to the `geom` argument:

```
> qplot(weathersit, cnt, data = bike, geom = c("boxplot",
  "jitter"), fill = weathersit)
```

The preceding command produces the following output:



See also...

- ▶ Generating standard plots such as histograms, boxplots and scatterplots

Creating multivariate plots

When exploring data, we want to get a feel for the interaction of as many variables as possible. Although our display and print media can display only two dimensions, by creatively using R's plotting features, we can bring many more dimensions into play. In this recipe, we show you how you can bring up to five variables into play.

Getting ready

Read the data from file and create factors. We also attach the data to save on keystrokes as follows:

```
> library(ggplot2)
> bike <- read.csv("daily-bike-rentals.csv")
```

```

> bike$season <- factor(bike$season, levels = c(1,2,3,4),
+   labels = c("Spring", "Summer", "Fall", "Winter"))
> bike$weathersit <- factor(bike$weathersit, levels = c(1,2,3),
+   labels = c("Clear", "Misty/cloudy", "Light snow"))
> bike$windspeed.fac <- cut(bike$windspeed, breaks=3,
+   labels=c("Low", "Medium", "High"))
> bike$weekday <- factor(bike$weekday, levels = c(0:6),
+   labels = c("Sun", "Mon", "Tue", "Wed", "Thur", "Fri", "Sat"))

> attach(bike)

```

How to do it...

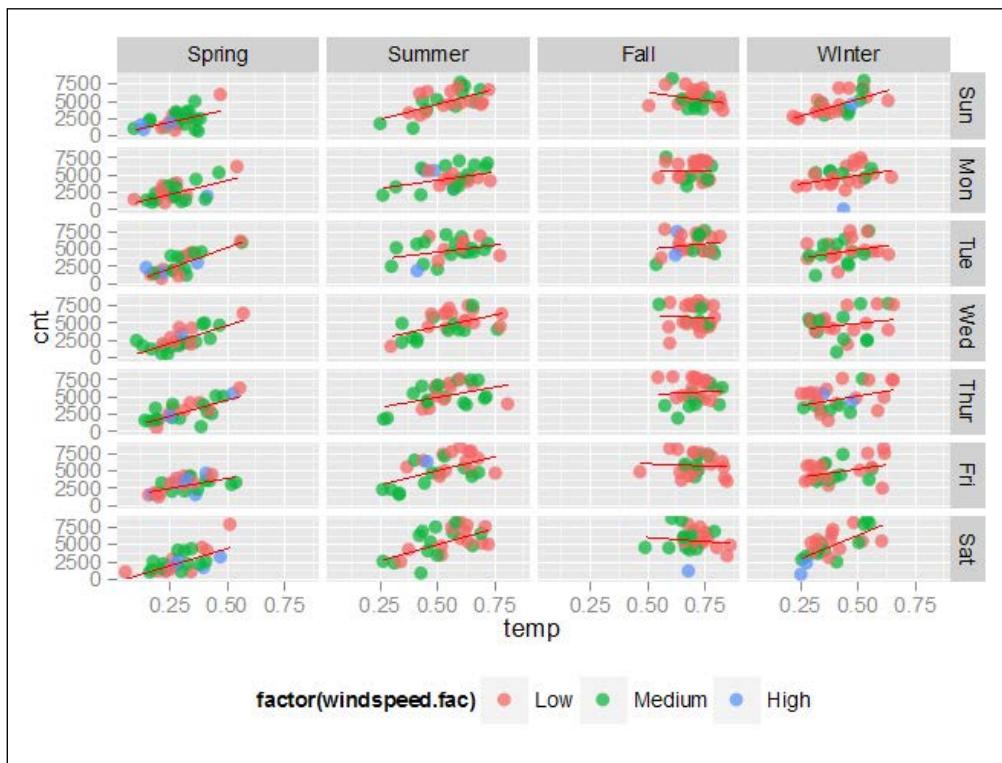
Create a multivariate plot using the following commands:

```

> plot <- ggplot(bike,aes(temp,cnt))
> plot + geom_point(size=3, aes(color=factor(windspeed.fac))) +
  geom_smooth(method="lm", se=FALSE, col="red") +
  facet_grid(weekday ~ season) + theme(legend.position="bottom")

```

The preceding commands produce the following output:



How it works...

Refer to the recipe *Create plots using ggplot2* earlier in this chapter.

See also...

- ▶ Generating standard plots such as histograms, boxplots and scatterplots
- ▶ Creating plots with ggplot2 package

3

Where Does It Belong? – Classification

In this chapter, we will cover the following recipes:

- ▶ Generating error/classification-confusion matrices
- ▶ Generating ROC charts
- ▶ Building, plotting, and evaluating – classification trees
- ▶ Using random forest models for classification
- ▶ Classifying using Support Vector Machine
- ▶ Classifying using the Naïve-Bayes approach
- ▶ Classifying using the KNN approach
- ▶ Using neural networks for classification
- ▶ Classifying using linear discriminant function analysis
- ▶ Classifying using logistic regression
- ▶ Using AdaBoost to combine classification tree models

Introduction

Analysts often seek to classify or categorize items, for example, to predict whether a given person is a potential buyer or not. Other examples include classifying—a product as defective or not, a tax return as fraudulent or not, a customer as likely to default on a payment or not, and a credit card transaction as genuine or fraudulent. This chapter covers recipes to use R to apply several classification techniques.

Generating error/classification-confusion matrices

You might build a classification model and want to evaluate the model by comparing the model's predictions with the actual outcomes. You will typically do this on the holdout data. Getting an idea of how the model does in training data itself is also useful, but you should never use that as an objective measure.

Getting ready

If you have not already downloaded the files for this chapter, do so now and ensure that the `college-perf.csv` file is in your R working directory. The file has data about a set of college students. The `Perf` variable has their college performance classified as `High`, `Medium`, or `Low`. The `Pred` variable contains a classification model's predictions of the performance level. The following code reads the data and converts the factor levels to a meaningful order—by default R orders factors alphabetically:

```
> cp <- read.csv("college-perf.csv")
> cp$Perf <- ordered(cp$Perf, levels =
+                      c("Low", "Medium", "High"))

> cp$Pred <- ordered(cp$Pred, levels =
+                      c("Low", "Medium", "High"))
```

How to do it...

To generate error/classification-confusion matrices, follow these steps:

1. First create and display a two-way table based on the actual and predicted values:

```
> tab <- table(cp$Perf, cp$Pred,
+                 dnn = c("Actual", "Predicted"))
> tab
```

This yields:

		Predicted		
Actual		Low	Medium	High
		Low	Medium	High
Low		1150	84	98
Medium		166	1801	170
High		35	38	458

2. Display the raw numbers as proportions or percentages. To get overall table-level proportions use:

```
> prop.table(tab)
```

		Predicted		
Actual				High
	Low	Medium	High	
Low	0.28750	0.02100	0.02450	
Medium	0.04150	0.45025	0.04250	
High	0.00875	0.00950	0.11450	

3. We often find it more convenient to interpret row-wise or column-wise percentages. To get row-wise percentages rounded to one decimal place, you can pass a second argument as 1:

```
> round(prop.table(tab, 1)*100, 1)
```

		Predicted		
Actual				High
	Low	Medium	High	
Low	86.3	6.3	7.4	
Medium	7.8	84.3	8.0	
High	6.6	7.2	86.3	



Passing 2 as the second argument yields column-wise proportions.



How it works...

The `table()` function performs a simple two-way cross-tabulation of values. For each unique value of the first variable, it counts the occurrences of different values of the second variable. It works for numeric, factor, and character variables.

There's more...

When dealing with more than two or three categories, seeing the error matrix as a chart could be useful to quickly assess the model's performance within various categories.

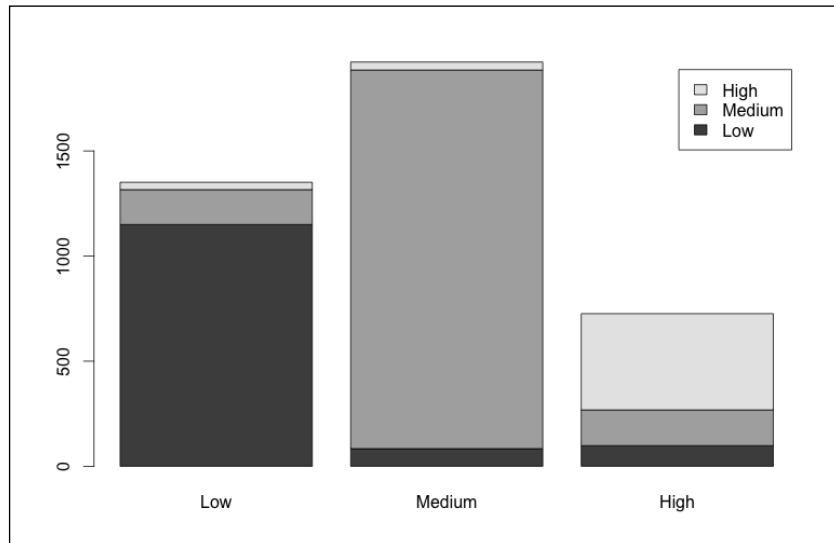
Visualizing the error/classification confusion matrix

You can create a barplot using the following command :

```
> barplot(tab, legend = TRUE)
```

Where Does It Belong? – Classification

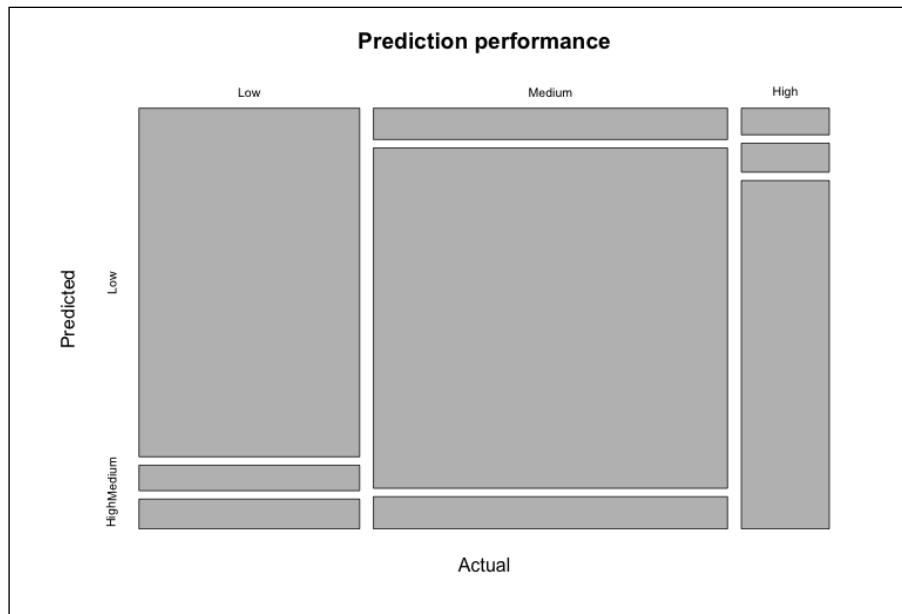
The following output is the result of the preceding command:



For a `mosaicplot` the following command is used:

```
> mosaicplot(tab, main = "Prediction performance")
```

The following output is obtained on running the command:



Comparing the model's performance for different classes

You can check whether the model's performance on different classes differs significantly by using the `summary` function:

```
> summary(tab)

Number of cases in table: 4000
Number of factors: 2
Test for independence of all factors:
  Chisq = 4449, df = 4, p-value = 0
```

The low p-value tells us that the proportions for the different classes are significantly different.

Generating ROC charts

When using classification techniques, we can rely on the technique to classify cases automatically. Alternately, we can rely on the technique to only generate the probabilities of cases belonging to various classes and then determine the cutoff probabilities ourselves. **receiver operating characteristic (ROC)** charts help with the latter approach by giving a visual representation of the true and false positives at various cutoff levels. We will use the `ROCR` package to generate ROC charts.

Getting ready

If you have not already installed the `ROCR` package, install it now. Load the data files for this chapter from the book's website and ensure that the `roc-example-1.csv` and `roc-example-2.csv` files are on your R working directory.

How to do it...

To generate ROC charts, follow these steps:

1. Load the package `ROCR`:

```
> library(ROCR)
```

2. Read the data file and take a look:

```
> dat <- read.csv("roc-example-1.csv")
> head(dat)
```

	prob	class
1	0.9917340	1
2	0.9768288	1

```
3 0.9763148      1  
4 0.9601505      1  
5 0.9351574      1  
6 0.9335989      1
```

3. Create the prediction object:

```
> pred <- prediction(dat$prob, dat$class)
```

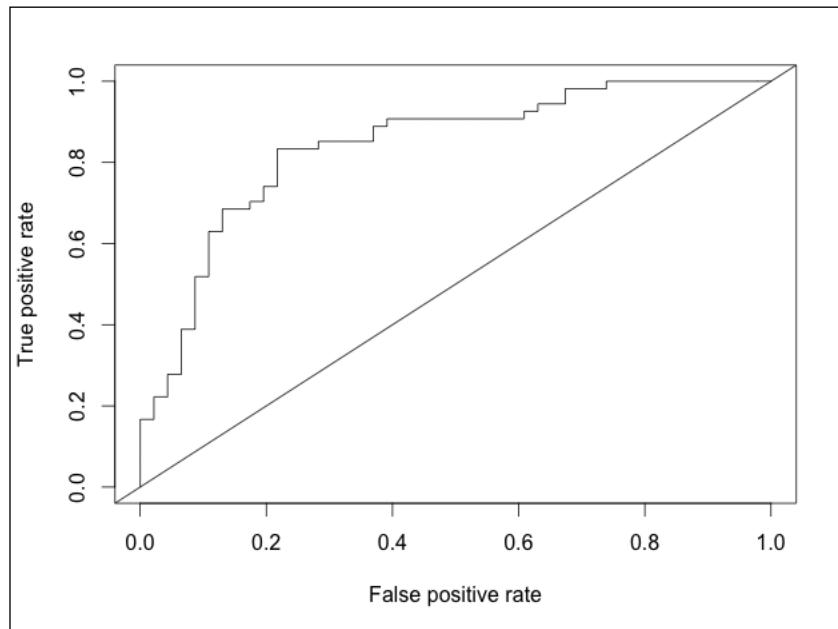
4. Create the performance object:

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

5. Plot the chart:

```
> plot(perf)  
> lines( par()$usr[1:2], par()$usr[3:4] )
```

The following output is obtained:



6. Find the cutoff values for various true positive rates. Extract the relevant data from the `perf` object into a data frame `prob.cuts`:

```
> prob.cuts <- data.frame(cut=perf@alpha.values[[1]], fpr=perf@x.  
values[[1]], tpr=perf@y.values[[1]])  
> head(prob.cuts)  
    cut   fpr       tpr  
1     Inf 0.00000000
```

```
2 0.9917340 0 0.01851852
3 0.9768288 0 0.03703704
4 0.9763148 0 0.05555556
5 0.9601505 0 0.07407407
6 0.9351574 0 0.09259259
```

```
> tail(prob.cuts)
      cut      fpr tpr
96 0.10426897 0.8913043 1
97 0.07292866 0.9130435 1
98 0.07154785 0.9347826 1
99 0.04703280 0.9565217 1
100 0.04652589 0.9782609 1
101 0.00112760 1.0000000 1
```

From the data frame `prob.cuts`, we can choose the cutoff corresponding to our desired true positive rate.

How it works...

Step 1 loads the package and step 2 reads in the data file.

Step 3 creates a prediction object based on the probabilities and the class labels passed in as arguments. In the current examples, our class labels are 0 and 1, and by default 0 becomes the "failure" class and 1 becomes the "success" class. We will see in the *There's more...* section below how to handle the case of arbitrary class labels.

Step 4 creates a performance object based on the data from the prediction object. We indicate that we want the "true positive rate" and the "false positive rate."

Step 5 plots the performance object. The plot function does not plot the diagonal line indicating the ROC threshold, and we added a second line of code to get that.

We generally use ROC charts to determine a good cutoff value for classification given the probabilities. Step 6 shows you how to extract from the `performance` object the cutoff value corresponding to each point on the plot. Armed with this, we can determine the cutoff that yields each of the true positive rates and, given a desired true positive rate, we can find the appropriate cutoff probability.

There's more...

We discuss in the following a few more of ROCR's important features.

Using arbitrary class labels

Unlike in the preceding example, we might have arbitrary class labels for success and failure. The `rocr-example-2.csv` file has `buyer` and `non-buyer` as the class labels, with `buyer` representing the success case.

In this case, we need to explicitly indicate the failure and success labels by passing in a vector with the failure case as the first element:

```
> dat <- read.csv("roc-example-2.csv")
> pred <- prediction(dat$prob, dat$class, label.ordering = c("non-
  buyer", "buyer"))
> perf <- performance(pred, "tpr", "fpr")
> plot(perf)
> lines( par()$usr[1:2], par()$usr[3:4] )
```

Building, plotting, and evaluating – classification trees

You can use a couple of R packages to build classification trees. Under the hood, they all do the same thing.

Getting ready

If you do not already have the `rpart`, `rpart.plot`, and `caret` packages, install them now. Download the data files for this chapter from the book's website and place the `banknote-authentication.csv` file in your R working directory.

How to do it...

This recipe shows you how you can use the `rpart` package to build classification trees and the `rpart.plot` package to generate nice-looking tree diagrams:

1. Load the `rpart`, `rpart.plot`, and `caret` packages:

```
> library(rpart)
> library(rpart.plot)
> library(caret)
```

2. Read the data:

```
> bn <- read.csv("banknote-authentication.csv")
```

3. Create data partitions. We need two partitions—training and validation. Rather than copying the data into the partitions, we will just keep the indices of the cases that represent the training cases and subset as and when needed:

```
> set.seed(1000)
> train.idx <- createDataPartition(bn$class, p = 0.7, list =
  FALSE)
```

4. Build the tree:

```
> mod <- rpart(class ~ ., data = bn[train.idx, ], method =
  "class", control = rpart.control(minsplit = 20, cp = 0.01))
```

5. View the text output (your result could differ if you did not set the random seed as in step 3):

```
> mod
n= 961

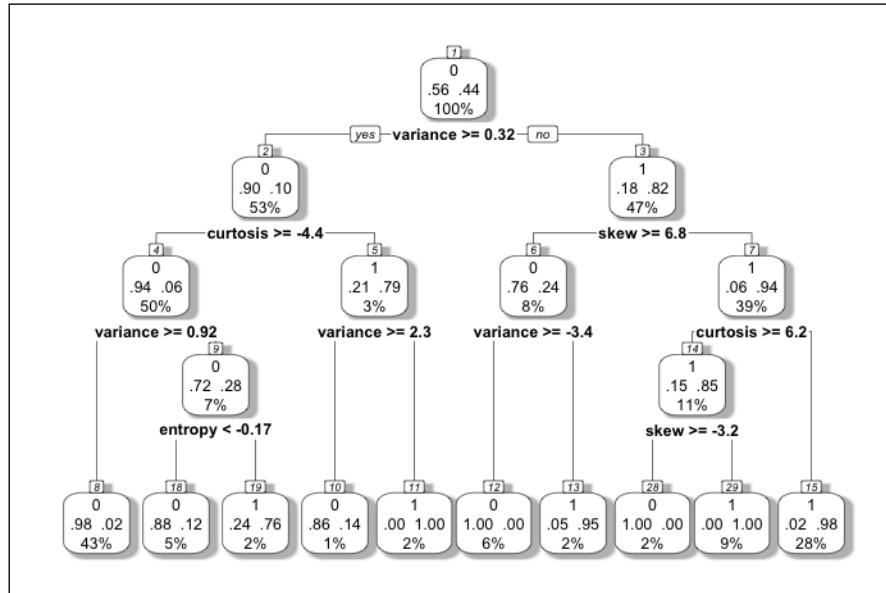
node), split, n, loss, yval, (yprob)
  * denotes terminal node

1) root 961 423 0 (0.55983351 0.44016649)
  2) variance>=0.321235 511 52 0 (0.89823875 0.10176125)
    4) curtosis>=-4.3856 482 29 0 (0.93983402 0.06016598)
      8) variance>=0.92009 413 10 0 (0.97578692 0.02421308) *
      9) variance< 0.92009 69 19 0 (0.72463768 0.27536232)
        18) entropy< -0.167685 52 6 0 (0.88461538 0.11538462) *
        19) entropy>=-0.167685 17 4 1 (0.23529412 0.76470588) *
      5) curtosis< -4.3856 29 6 1 (0.20689655 0.79310345)
      10) variance>=2.3098 7 1 0 (0.85714286 0.14285714) *
      11) variance< 2.3098 22 0 1 (0.00000000 1.00000000) *
      3) variance< 0.321235 450 79 1 (0.17555556 0.82444444)
        6) skew>=6.83375 76 18 0 (0.76315789 0.23684211)
        12) variance>=-3.4449 57 0 0 (1.00000000 0.00000000) *
        13) variance< -3.4449 19 1 1 (0.05263158 0.94736842) *
      7) skew< 6.83375 374 21 1 (0.05614973 0.94385027)
      14) curtosis>=6.21865 106 16 1 (0.15094340 0.84905660)
        28) skew>=-3.16705 16 0 0 (1.00000000 0.00000000) *
        29) skew< -3.16705 90 0 1 (0.00000000 1.00000000) *
      15) curtosis< 6.21865 268 5 1 (0.01865672 0.98134328) *
```

6. Generate a diagram of the tree (your tree might differ if you did not set the random seed as in step 3):

```
> prp(mod, type = 2, extra = 104, nn = TRUE, fallen.leaves = TRUE,
  faclen = 4, varlen = 8, shadow.col = "gray")
```

The following output is obtained as a result of the preceding command:



7. Prune the tree:

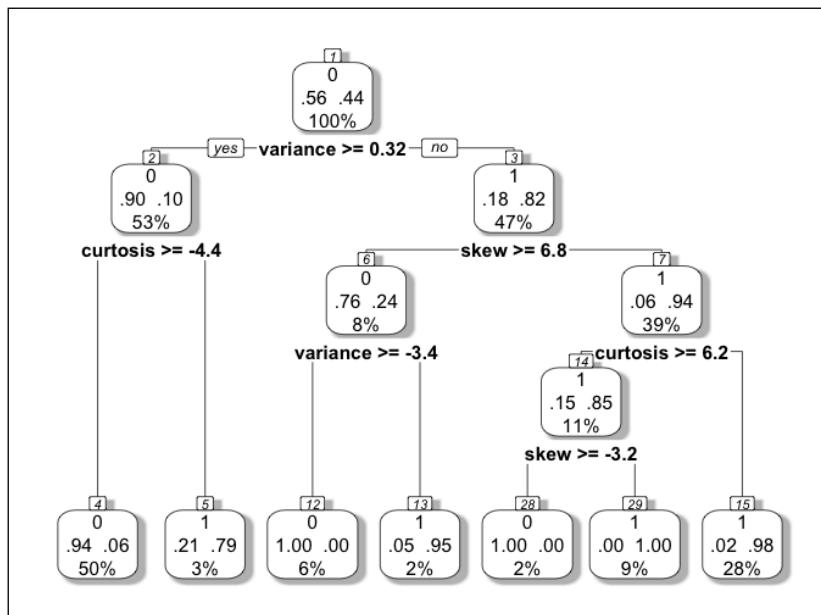
```
> # First see the cptable
> # !!Note!!: Your table can be different because of the
> # random aspect in cross-validation
> mod$cptable
```

	CP	nsplit	rel error	xerror	xstd
1	0.69030733	0	1.00000000	1.00000000	0.03637971
2	0.09456265	1	0.30969267	0.32624111	0.02570025
3	0.04018913	2	0.21513002	0.23877070	0.02247542
4	0.01891253	4	0.13475177	0.16075650	0.01879222
5	0.01182033	6	0.09692671	0.13475180	0.01731090
6	0.01063830	7	0.08510638	0.13238770	0.01716786
7	0.01000000	9	0.06382979	0.12765960	0.01687712

```
> # Choose CP value as the highest value whose
> # xerror is not greater than minimum xerror + xstd
> # With the above data that happens to be
> # the fifth one, 0.01182033
> # Your values could be different because of random
> # sampling
> mod$pruned = prune(mod, mod$cptable[5, "CP"] )
```

8. View the pruned tree (your tree will look different):

```
> prp(mod.pruned, type = 2, extra = 104, nn = TRUE, fallen.leaves
= TRUE, faclen = 4, varlen = 8, shadow.col = "gray")
```



9. Use the pruned model to predict for the validation partition (note the minus sign before `train.idx` to consider the cases in the validation partition):

```
> pred.pruned <- predict(mod, bn[-train.idx,], type = "class")
```

10. Generate the error/classification-confusion matrix:

```
> table(bn[-train.idx,]$class, pred.pruned, dnn = c("Actual",
"Predicted"))
   Predicted
Actual    0    1
  0 213   11
  1   11 176
```

How it works...

Steps 1 to 3 load the packages, read the data, and identify the cases in the training partition, respectively. See the recipe *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis* for more details on partitioning. In step 3, we set the random seed so that your results should match those that we display.

Step 4 builds the classification tree model:

```
> mod <- rpart(class ~ ., data = bn[train.idx, ], method = "class",
control = rpart.control(minsplit = 20, cp = 0.01))
```

The `rpart()` function builds the tree model based on the following:

- ▶ Formula specifying the dependent and independent variables
- ▶ Dataset to use
- ▶ A specification through `method="class"` that we want to build a classification tree (as opposed to a regression tree)
- ▶ Control parameters specified through the `control = rpart.control()` setting; here we have indicated that the tree should only consider nodes with at least 20 cases for splitting and use the complexity parameter value of 0.01—these two values represent the defaults and we have included these just for illustration

Step 5 produces a textual display of the results. Step 6 uses the `prp()` function of the `rpart.plot` package to produce a nice-looking plot of the tree:

```
> prp(mod, type = 2, extra = 104, nn = TRUE, fallen.leaves = TRUE,
faclen = 4, varlen = 8, shadow.col = "gray")
```

- ▶ use `type=2` to get a plot with every node labeled and with the split label below the node
- ▶ use `extra=4` to display the probability of each class in the node (conditioned on the node and hence summing to 1); add 100 (hence `extra=104`) to display the number of cases in the node as a percentage of the total number of cases
- ▶ use `nn = TRUE` to display the node numbers; the root node is node number 1 and node n has child nodes numbered 2n and 2n+1
- ▶ use `fallen.leaves=TRUE` to display all leaf nodes at the bottom of the graph
- ▶ use `faclen` to abbreviate class names in the nodes to a specific maximum length
- ▶ use `varlen` to abbreviate variable names
- ▶ use `shadow.col` to specify the color of the shadow that each node casts

Step 7 prunes the tree to reduce the chance that the model too closely models the training data—that is, to reduce overfitting. Within this step, we first look at the complexity table generated through cross-validation. We then use the table to determine the cutoff complexity level as the largest `xerror` (cross-validation error) value that is not greater than one standard deviation above the minimum cross-validation error.

Steps 8 through 10 display the pruned tree; use the pruned tree to predict the class for the validation partition and then generate the error matrix for the validation partition.

There's more...

We discuss in the following an important variation on predictions using classification trees.

Computing raw probabilities

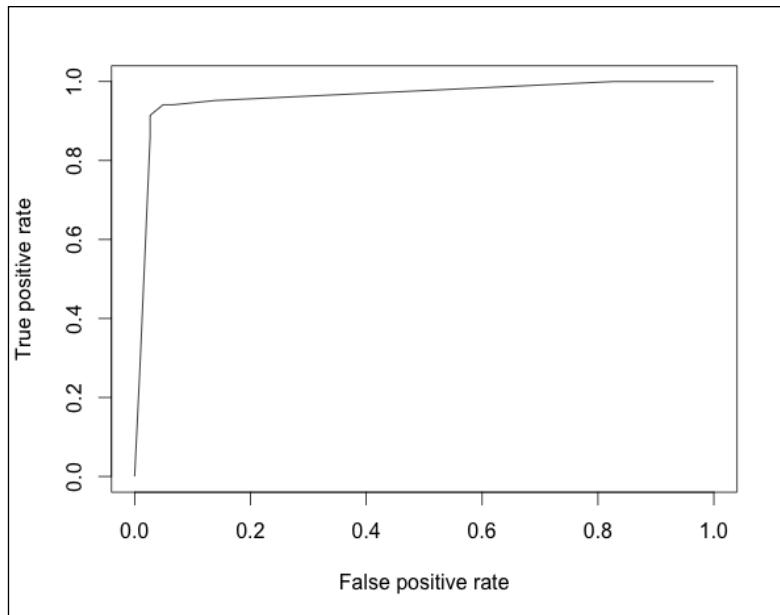
We can generate probabilities in place of classifications by specifying `type = "prob"`:

```
> pred.pruned <- predict(mod, bn[-train.idx,], type = "prob")
```

Create the ROC Chart

Using the preceding raw probabilities and the class labels, we can generate a ROC chart. See the recipe *Generating ROC charts* earlier in this chapter for more details:

```
> pred <- prediction(pred.pruned[,2], bn[-train.idx,"class"])
> perf <- performance(pred, "tpr", "fpr")
> plot(perf)
```



See also

- ▶ *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis*
- ▶ *Generating error/classification-confusion matrices* in this chapter.
- ▶ *Building regression trees* in Chapter 4, *Give Me a Number – Regression*

Using random forest models for classification

The `randomForest` package can help you to easily apply the very powerful (but computationally intensive) random forest classification technique.

Getting ready

If you have not already installed the `randomForest` and `caret` packages, install them now. Download the data files for this chapter from the book's website and place the `banknote-authentication.csv` file in your R working directory. We will build a random forest model to predict `class` based on the other variables.

How to do it...

To use Random Forest models for classification, follow these steps:

1. Load the `randomForest` and `caret` packages:

```
> library(randomForest)  
> library(caret)
```

2. Read the data and convert the response variable to a factor:

```
> bn <- read.csv("banknote-authentication.csv")  
> bn$class <- factor(bn$class)
```

3. Select a subset of the data for building the model. In Random Forests, we do not need to actually partition the data for model evaluation since the tree construction process has partitioning inherent in every step. However, we keep aside some of the data here just to illustrate the process of using the model for prediction and also to get an idea of the model's performance:

```
> set.seed(1000)  
  
> sub.idx <- createDataPartition(bn$class, p=0.7, list=FALSE)
```

4. Build the random forest model (since it builds many classification trees, the following command can take a lot of processing time on even moderately large data):

```
> mod <- randomForest(x = bn[sub.idx,1:4], y=bn[sub.  
idx,5], ntree=500, keep.forest=TRUE)
```

5. Use the model to predict for cases that we set aside in step 3:

```
> pred <- predict(mod, bn[-sub.idx,])
```

6. Build the error matrix:

```
> table(bn[-sub.idx,"class"], pred, dnn = c("Actual",
  "Predicted"))
   Predicted
Actual      0    1
  0 227    1
  1    1 182
```

How it works...

Steps 1 loads the necessary packages and step 2 reads the data and converts the response variable to a factor.

Step 3 sets aside some of the data for later use. Strictly speaking, we do not have to partition the data for random forests because, while building each tree, the method sets aside some of the cases for cross-validation. However, we set aside some of the cases just to illustrate the process of using the model for prediction. (We set the random seed to enable you to match your results with those we display.)

Step 4 uses the `randomForest` function to build the model. Since the predictor variables are in the first four variables of the data frame and since we want to use only the selected subset for model building, we specify `x= bn [sub.idx, 1:4]`. Since the target variable is in the fifth column, we specify `y= bn [sub.idx, 5]`. We specify the number of trees to build in the forest through the `ntree` argument (the default value is 500).

Step 5 illustrates how to predict using the model.

Step 6 uses the predictions and the actual values to generate an error matrix.



The model that the `randomForest` function produces does not keep information about the trees and hence we cannot use the model for predicting future cases. To force the model to keep the generated forest, specify `keep.forest=TRUE`.

There's more...

We discuss in the following a few prominent options.

Computing raw probabilities

As with simple classification tree models, we can generate probabilities in place of classifications by specifying `type="prob"` – the default value "response" generates classifications:

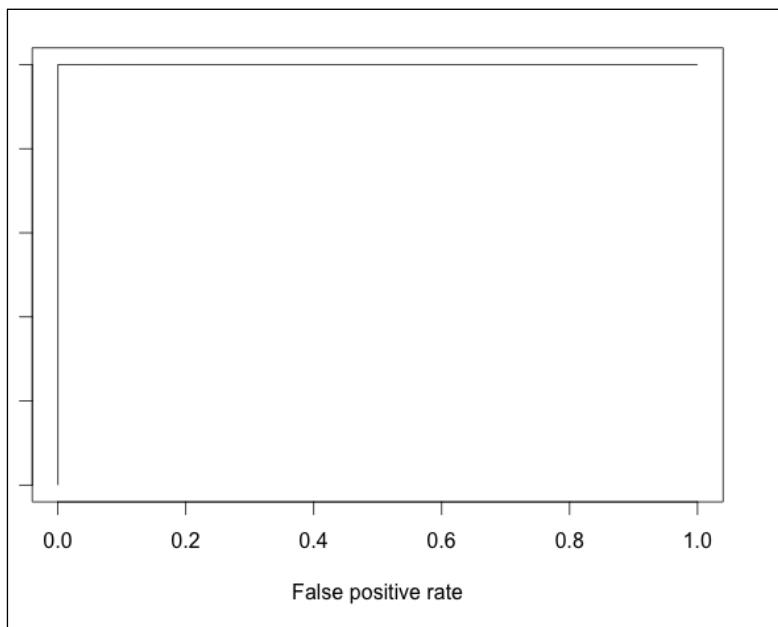
```
> probs <- predict(mod, bn[-sub.idx,], type = "prob")
```

Generating the ROC chart

Using the preceding probabilities, we can generate the ROC chart. For details, refer to *Generating ROC charts* earlier in this chapter:

```
> pred <- prediction(probs[,2], bn[-sub.idx, "class"])
> perf <- performance(pred, "tpr", "fpr")
> plot(perf)
```

The following output is the result of preceding command:



Specifying cutoffs for classification

Instead of using the default rule of simple majority for classification, we can specify cutoff probabilities as a vector of length equal to the number of classes. The proportion of the ratio of votes to the cutoff determines the winning class. We can specify this both at the time of tree construction and while using the model for predictions.

See also...

- ▶ *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis*
- ▶ *Generating error/classification-confusion matrices* in this chapter
- ▶ *Building Random Forest models for regression* in Chapter 4, *Give Me a Number – Regression*

Classifying using Support Vector Machine

The `e1071` package can help you to easily apply the very powerful **Support Vector Machine (SVM)** classification technique.

Getting ready

If you have not already installed the `e1071` and `caret` packages, install them now. Download the data files for this chapter from the book's website and place the `banknote-authentication.csv` file in your R working directory. We will build an SVM model to predict class based on the other variables.

How to do it...

To classify using SVM, follow these steps:

1. Load the `e1071` and `caret` packages:

```
> library(e1071)
> library(caret)
```

2. Read the data:

```
> bn <- read.csv("banknote-authentication.csv")
```

3. Convert the outcome variable `class` to a factor:

```
> bn$class <- factor(bn$class)
```

4. Partition the data:

```
> set.seed(1000)
> t.idx <- createDataPartition(bn$class, p=0.7, list=FALSE)
```

5. Build the model:

```
> mod <- svm(class ~ ., data = bn[t.idx,])
```

6. Check model performance on training data by generating an error/classification-confusion matrix:

```
> table(bn[t.idx,"class"], fitted(mod), dnn = c("Actual",
  "Predicted"))
  Predicted
Actual   0   1
  0 534   0
  1   0 427
```

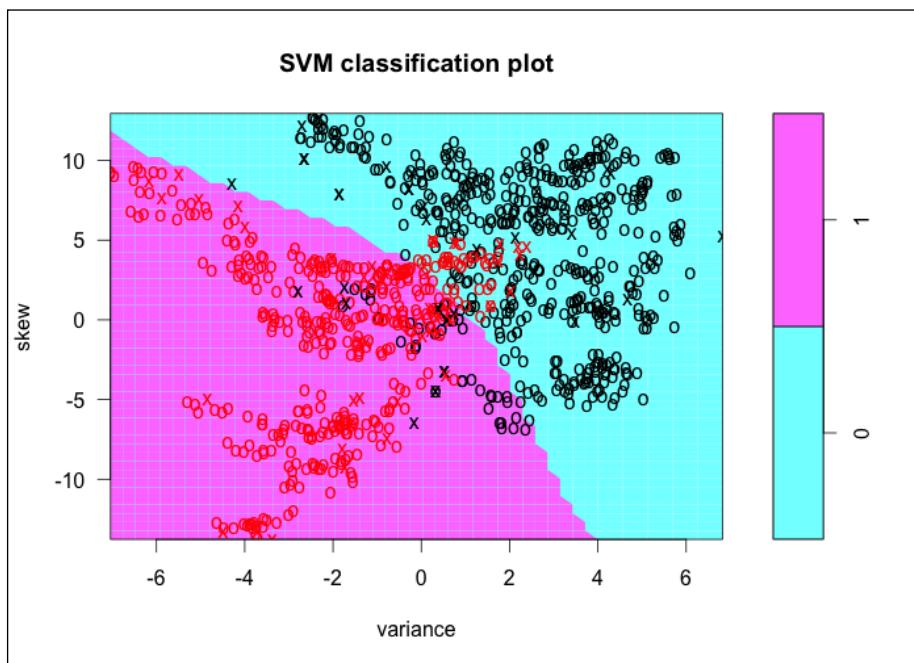
7. Check model performance on the validation partition:

```
> pred <- predict(mod, bn[-t.idx,])
> table(bn[-t.idx, "class"], pred, dnn = c("Actual", "Predicted"))
   Predicted
Actual      0     1
      0 228    0
      1     0 183
```

8. Plot the model on the training partition. Our data has more than two predictors, but we can only show two in the plot. We have selected `skew` and `variance`:

```
> plot(mod, data=bn[t.idx,], skew ~ variance)
```

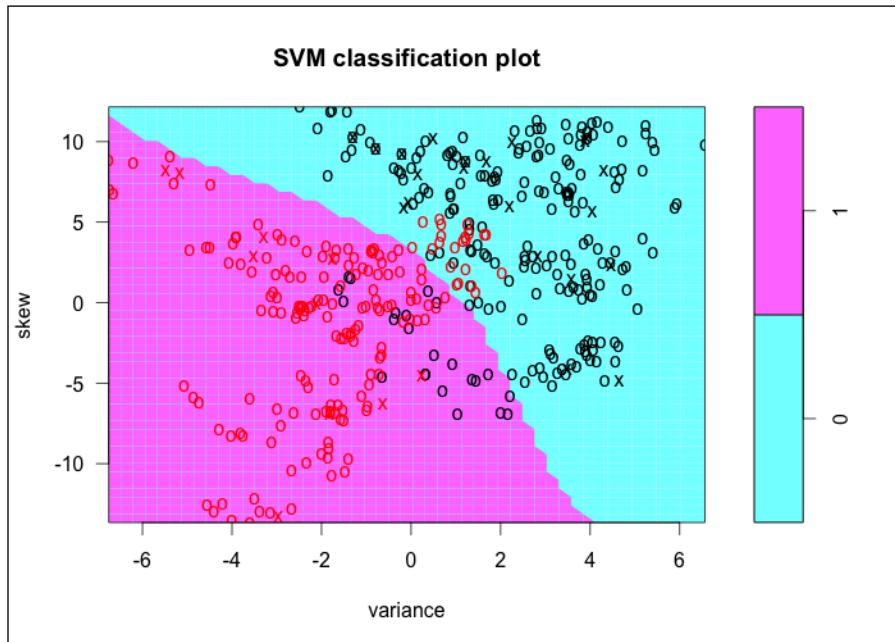
The following plot is the output of the preceding command:



9. Plot the model on the validation partition. Our data has more than two predictors, but we can only show two in the plot. We have selected `skew` and `variance`:

```
> plot(mod, data=bn[-t.idx,], skew ~ variance)
```

The follow plot is the result of the preceding command:



How it works...

Step 1 loads the necessary packages.

Step 2 reads the data.

Step 3 converts the outcome variable `class` to a factor.

Step 4 identifies the cases in the training partition (we set the random seed to enable you to match your results with ours).

Step 5 builds the SVM classification model. The `svm` function determines the type of model (classification or regression) based on the nature of the outcome variable. When the outcome variable is a factor, `svm` builds a classification model. At a minimum, we need to pass the model formula and the dataset to use as arguments. (Alternately, we can pass the outcome variable and the predictor variables separately as the `x` and `y` arguments).

Step 6 uses the resulting `svm` object `mod` containing the model to create an error/classification-confusion matrix. The `svm` model retains the fitted values on the training partition, and hence we do not need to go through the step of creating the predictions. We access the fitted values through `fitted(mod)`. Refer to the recipe *Generating error/classification-confusion matrices* in this chapter for details on the `table` function.

Step 7 generates the model's predictions for the validation partition by using the `predict` function. We pass as arguments the model and the data for which we need predictions. It then uses these predictions to generate the associated error/classification-confusion matrix.

Steps 8 and 9 use the `plot` function to plot the model's results. We pass as arguments the model and the data for which we need the plot. If the data has only two predictors, we can get the complete picture from such a plot. However, our example has four predictors and we have chosen two of them for the plot.

There's more...

The `svm` function has several additional arguments through which we can control its behavior.

Controlling scaling of variables

By default, `svm` scales all the variables (predictor and outcome) to zero mean and unit variance before building a model as this generally produces better results. We can use the `scale` argument—a logical vector—to control this. If the length of the vector is 1, then it is recycled as many times as needed.

Determining the type of SVM model

By default, when the outcome variable is a factor, `svm` performs classification. When the outcome is numeric, it performs regression. We can override the default or select other options through these values for `type`:

- ▶ `type = C-classification`
- ▶ `type = nu-classification`
- ▶ `type = one-classification`
- ▶ `type = eps-regression`
- ▶ `type = nu-regression`

Assigning weights to the classes

In cases where the sizes of the classes are highly skewed, the larger class could dominate. To balance this, we might want to weight the classes differently from the default equal weighting. We can use the `class.weights` argument for this:

```
> mod <- svm(class ~ ., data = bn[t.idx,], class.weights=c("0"=0.3,  
"1"=0.7 ))
```

See also...

- ▶ *Creating random data partitions in Chapter 2, What's in There? – Exploratory Data Analysis*
- ▶ *Generating error/classification-confusion matrices in this chapter*

Classifying using the Naïve Bayes approach

The `e1071` package contains the `naiveBayes` function for the Naïve Bayes classification.

Getting ready

If you do not already have the `e1071` and `caret` packages, install them now. Download the data files for this chapter from the book's website and place the `electronics-purchase.csv` file in your R working directory. Naïve Bayes requires all the variables to be categorical. So, if needed, you should first convert all variables accordingly—refer to the recipe *Binning numerical data in Chapter 1, Acquire and Prepare the Ingredients – Your Data*.

How to do it...

To classify using the Naïve Bayes method, follow these steps:

1. Load the `e1071` and `caret` packages:

```
> library(e1071)
> library(caret)
```

2. Read the data:

```
> ep <- read.csv("electronics-purchase.csv")
```

3. Partition the data:

```
> set.seed(1000)
> train.idx <- createDataPartition(ep$Purchase, p = 0.67, list =
FALSE)
```

4. Build the model:

```
> epmod <- naiveBayes(Purchase ~ . , data = ep[train.idx,])
```

5. Look at the model:

```
> epmod
```

6. Predict for each case of the validation partition:

```
> pred <- predict(epmod, ep[-train.idx,])
```

7. Generate and view the error matrix/classification confusion matrix for the validation partition:

```
> tab <- table(ep[-train.idx,]$Purchase, pred, dnn = c("Actual",  
"Predicted"))  
> tab  
          Predicted  
Actual  No  Yes  
  No    1    1  
  Yes   0    2
```

How it works...

Step 1 loads the required packages, step 2 reads the data, and step 3 identifies the rows in the training partition (we set the random seed to enable you to match your results with ours).

Step 4 builds the model using the `naiveBayes()` function and passing the formula and the training partition as the arguments. Step 5 displays the conditional probabilities that the `naiveBayes()` function generates for use in making predictions.

Step 6 generates the model predictions for the validation partition and step 7 builds the error matrix as follows:

Naive Bayes Classifier for Discrete Predictors							
Call: naiveBayes.default(x=X, y=Y, laplace= laplace)							
A-priori probabilities:	A-priori probabilities of each class						
Y No Yes 0.5 0.5							
Conditional probabilities:							
Education							
Y	A	B	C				
No	0.5000000	0.3333333	0.1666667				
Yes	0.1666667	0.3333333	0.5000000				
Gender							
Y	F	M	P(Education=B Purchase=Yes)				
No	0.5000000	0.5000000					
Yes	0.3333333	0.6666667					
Smart_ph							
Y	N	Y	P(Gender=M Purchase=No)				
No	0.5	0.5					
Yes	0.5	0.5					
Tablet							
Y	N	Y					
No	0.1666667	0.3333333					
Yes	0.1666667	0.8333333					

Step 6 generates the predictions for each case of the validation partition using the `predict()` function and passing the model and the validation partition as arguments. Step 8 generates the error or classification confusion matrix using the `table()` function.

See also...

- ▶ *Creating random data partitions in Chapter 2, What's in There? – Exploratory Data Analysis*

Classifying using the KNN approach

The `class` package contains the `knn` function for KNN classification.

Getting ready

If you have not already installed the `class` and `caret` packages, install them now. Download the data files for this chapter from the book's website and place the `vacation-trip-classification.csv` file in your R working directory. KNN requires all the independent/predictor variables to be numeric, and the dependent variable or target to be categorical. So, if needed, you should first convert variables accordingly—refer to the recipes *Creating dummies for categorical variables* and *Binning numerical data* in Chapter 1, *Acquire and Prepare the Ingredients – Your Data*.

How to do it...

To classify using the K-Nearest Neighbours method, follow the steps below,

1. Load the `class` and `caret` packages:

```
> library(class)
> library(caret)
```

2. Read the data:

```
> vac <- read.csv("vacation-trip-classification.csv")
```

3. Standardize the predictor variables `Income` and `Family_size`:

```
> vac$Income.z <- scale(vac$Income)
> vac$Family_size.z <- scale(vac$Family_size)
```

4. Partition the data. You need three partitions for KNN:

```
> set.seed(1000)
> train.idx <- createDataPartition(vac$result, p = 0.5, list =
  FALSE)
> train <- vac[train.idx, ]
> temp <- vac[-train.idx, ]
> val.idx <- createDataPartition(temp$result, p = 0.5, list =
  FALSE)
> val <- temp[val.idx, ]
> test <- temp[-val.idx, ]
```

5. Generate predictions for validation cases with $k=1$:

```
> pred1 <- knn(train[4:5], val[,4:5], train[,3], 1)
```

6. Generate an error matrix for $k=1$:

```
> errmat1 <- table(val$result, pred1, dnn = c("Actual", "Predicted"))
```
7. Repeat the preceding process for many values of k and choose the best value for k . Look under the following *There's more...* section for a way to automate this process.
8. Use that value of k to generate predictions and the error matrix for the cases in the test partition (in the following code, we assume that $k=1$ was preferred):

```
> pred.test <- knn(), train[4:5], test[,4:5], train[,3], 1)
> errmat.test = table(test$result, pred.test, dnn = c("Actual", "Predicted"))
```

How it works...

Steps 1 to 3 load the necessary packages and read the data file.

Step 4 creates three partitions (50 %, 25 %, and 25 %). We set the random seed to enable you to match your results with those we display. Refer to the recipe *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis* for information on data partitioning.

Step 5 uses the `knn` function to generate predictions with $k=1$. It uses only the standardized values of the predictor variables and hence specifies `train[,4:5]` and `val[,4:5]`.

Step 6 generates the error matrix for $k=1$.

There's more...

We now turn to some other ways in which you can use KNN classifications.

Automating the process of running KNN for many k values

The following convenience function helps to free you from the drudgery of repeatedly running nearly identical commands to run KNN for various values of k :

```
knn.automate <- function (trg_predictors, val_predictors, trg_target,
  val_target, start_k, end_k)
{
  for (k in start_k:end_k) {
    pred <- knn(trg_predictors, val_predictors,
                trg_target, k)
    tab <- table(val_target, pred, dnn = c("Actual", "Predicted"))
    cat(paste("Error matrix for k=", k, "\n"))
    cat("===== \n")}
```

```
    print(tab)
    cat("-----\n\n")
  }
}
```

With the preceding function in place, you can use the following to run knn for k=1 through k=7 for the example in the main recipe:

```
> knn.automate(train[,4:5], val[,4:5], train[,3], val[,3], 1,7)
```

Using KNN to compute raw probabilities instead of classifications

When we use KNN to classify cases, the underlying algorithm uses a simple majority vote to determine the class. In such a case, we implicitly consider all errors to be equally important. However, in situations with asymmetric costs—where we are prepared to make one kind of error more readily than another—we might not want to use a simple majority vote to determine the class. Instead, we might want to get the raw probabilities (proportions) for each class and choose a cutoff probability for classification. For example, it might be 10 times costlier to classify a buyer as a non-buyer than to classify a non-buyer as a buyer. In such cases, we might accept a probability far lower than 0.5 to classify a case as buyer, whereas a simple majority would require a probability slightly greater than 0.5.

To compute raw probabilities instead of classifications, use the prob=TRUE argument.

For example:

```
> pred5 <- knn(train[4:5], val[,4:5], train[,3], 5, prob=TRUE)
> pred5
[1] 1.0000000 0.8000000 1.0000000 0.6000000 0.8000000
[6] 0.6000000 0.6000000 0.8333333 0.6000000 0.8333333
Levels: Buyer Non-buyer
```

Using neural networks for classification

The nnet package contains the nnet function for classification using neural networks.

Getting ready

If you have not already installed the nnet and caret packages, install them now. Download the data files for this chapter from the book's website and place the banknote-authentication.csv file in your R working directory. We will use class as our target or outcome variable, and all the remaining variables as predictors. Using Neural Networks requires all the independent/predictor variables to be numeric and the dependent variable or outcomes to be 0-1. However, the nnet function does all the work of generating dummies (contrasts) and correctly handles categorical outcome variables.

How to do it...

To use Neural networks for classification, follow these steps:

1. Load the `nnet` and `caret` packages:

```
> library(nnet)
> library(caret)
```

2. Read the data:

```
> bn <- read.csv("banknote-authentication.csv")
```

3. Convert the outcome variable `class` to a factor:

```
> bn$class <- factor(bn$class)
```

4. Partition the data. The predictor variables are already numeric and the outcome variable `class` is already 0-1, so we do not have to do any data preparation. Refer to *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis* for details on how the following command works:

```
> train.idx <- createDataPartition(bn$class, p=0.7, list = FALSE)
```

5. Build the neural network model:

```
> mod <- nnet(class ~., data=bn[train.idx,], size=3, maxit=10000, dec
ay=.001, rang = 0.05)
```

6. Use model to predict for validation partition:

```
> pred <- predict(mod, newdata=bn[-train.idx,], type="class")
```

7. Build and display the error/classification-confusion matrix on the validation partition:

```
> table(bn[-train.idx,]$class, pred)
```

How it works...

Steps 1 loads the packages needed and step 2 reads the data.

Step 3 converts the outcome variable `class` into a factor. For `nnet` to perform classification, we need the outcome variable to be a factor. If you have predictor variables that are really categorical but have numeric values, convert them to factors so that `nnet` can treat them appropriately. Since we have only numeric predictor variables, we need not do anything for the predictor variables.

Step 4 partitions the data. See *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis* for more details on this step.

Step 5 builds the neural network model. We pass the formula and the dataset as the first two arguments:

- ▶ The `size` argument specifies the number of units in the internal layer (`nnet` works with just one hidden layer). One rule of thumb is to set the number of units in the hidden layer close to the mean of the number of units in the input and output layers. Higher values can give slightly better results at the expense of computation time.
- ▶ `maxit` specifies the maximum number of iterations to perform to try for convergence. The algorithm stops if convergence is achieved earlier. If not, it stops after `maxit` iterations.
- ▶ `decay` controls overfitting.

Step 6 uses the model to generate predictions for the validation partition. We specified `type = "class"` to generate classifications.

Step 7 generates the error/classification-confusion matrix.

There's more...

We discuss in the following some ideas for exercising greater control over the model building and prediction steps.

Exercising greater control over `nnet`

Use the following additional options:

- ▶ `na.action`: By default, any missing values cause the function to fail. You can specify `na.action = na.omit` to exclude cases with any missing values.
- ▶ Use `skip = TRUE` to add skip level direct connections from input nodes to the output nodes.
- ▶ Use the `rang` argument to specify the range for the initial random weights as `[-rang, rang]`; if the input values are large, select `rang` such that `rang * (max|variable|)` is close to 1.

Generating raw probabilities

Use the `type = "raw"` option to generate raw probabilities:

```
> pred <- predict(mod, newdata=bn[-train.idx,] type="raw")
```

Classifying using linear discriminant function analysis

The MASS package contains the `lda` function for classification using linear discriminant function analysis.

Getting ready

If you have not already installed the MASS and caret packages, install them now.

Download the data files for this chapter from the book's website and place the `banknote-authentication.csv` file in your R working directory. We will use `class` as our target or outcome variable, and all the remaining variables as predictors.

How to do it...

To classify using linear discriminant function analysis, follow these steps:

1. Load the MASS and caret packages:

```
> library(MASS)  
> library(caret)
```

2. Read the data:

```
> bn <- read.csv("banknote-authentication.csv")
```

3. Convert the outcome variable `class` to a factor:

```
> bn$class <- factor(bn$class)
```

4. Partition the data. The predictor variables are already numeric and the outcome variable `class` is already 0-1, so we do not have to do any data preparation. Refer to *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis* for details on how the following command works:

```
> set.seed(1000)  
> t.idx <- createDataPartition(bn$class, p = 0.7, list=FALSE)
```

5. Build the Linear Discriminant Function model:

```
> ldamod <- lda(bn[t.idx, 1:4], bn[t.idx, 5])
```

6. Check how the model performs on the training partition (your results could differ because of random partitioning):

```
> bn[t.idx, "Pred"] <- predict(ldamod, bn[t.idx, 1:4])$class
> table(bn[t.idx, "class"], bn[t.idx, "Pred"], dnn = c("Actual",
  "Predicted"))
   Predicted
Actual    0    1
  0 511  23
  1    0 427
```

7. Generate predictions on the validation partition and check performance (your results could differ):

```
> bn[-t.idx, "Pred"] <- predict(ldamod, bn[-t.idx, 1:4])$class
> table(bn[-t.idx, "class"], bn[-t.idx, "Pred"], dnn = c("Actual",
  "Predicted"))
   Predicted
Actual    0    1
  0 219   9
  1    0 183
```

How it works...

Step 1 loads the MASS and caret packages and step 2 reads in the data.

Step 3 converts our outcome variable into a factor.

Step 4 partitions the data. We set the random seed to enable you to match your results with those we display.

Step 5 builds the linear discriminant function model. We pass the predictors as the first argument, and the outcome values as the second argument to the lda function. We can also supply the details as a formula—see the following *There's more...* section.

Step 6 uses the predict function to generate the predictions for the training partition. We pass the model and the predictor variables. The class component of the returned object from the predict function contains the predicted class values. We then use the table function to generate a two-way cross-table.

Step 7 evaluates the model on the validation partition by repeating the preceding two steps on that partition.

There's more...

The `lda` function has several optional arguments and we have shown the most commonly used ones earlier.

Using the formula interface for lda

Instead of specifying the predictors and outcome as two separate arguments, we could have written the preceding step 5 as:

```
> ldamod <- lda(class ~ ., data = bn[t.idx,])
```

See also ...

- ▶ *Creating random data partitions in Chapter 2, What's in There? – Exploratory Data Analysis*

Classifying using logistic regression

The `stats` package contains the `glm` function for classification using logistic regression.

Getting ready

If you have not already installed the `caret` package, install it now. Download the data files for this chapter from the book's website and place the `boston-housing-logistic.csv` file in your R working directory. We will use `CLASS` as our target or outcome variable, and all the remaining variables as predictors. Our outcome variable has values of 0 or 1, with 0 representing neighborhoods with "Low" median home values and 1 representing neighborhoods with "High" median home values. Logistic regression requires all the independent/predictor variables to be numeric, and the dependent variable or outcome to be categorical and binary. However, the `glm` function does all the work of generating dummies (contrasts) for categorical variables.

How to do it...

To classify using logistic regression, follow these steps:

1. Load the `caret` package:

```
> library(caret)
```

2. Read the data:

```
> bh <- read.csv("boston-housing-logistic.csv")
```

3. Convert the outcome variable class to a factor:

```
> bh$CLASS <- factor(bh$CLASS, levels = c(0,1))
```

4. Partition the data. The predictor variables are already numeric and the outcome variable CLASS is already 0-1, so we do not have to do any data preparation. Refer to the recipe *Creating random data partitions* in Chapter 2, What's in There? – Exploratory Data Analysis, for details on how the following command works:

```
> set.seed(1000)
> train.idx <- createDataPartition(bh$CLASS, p=0.7, list = FALSE)
```

5. Build the logistic regression model:

```
> logit <- glm(CLASS~., data = bh[train.idx], family=binomial)
```

6. Examine the model (your results could differ because of random partitioning):

```
> summary(logit)
Call:
glm(formula = CLASS ~ ., family = binomial, data = bh[train.idx,
])

```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-2.2629	-0.3431	0.0603	0.3251	3.3310

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	33.452508	4.947892	6.761	1.37e-11 ***
NOX	-31.377153	6.355135	-4.937	7.92e-07 ***
DIS	-0.634391	0.196799	-3.224	0.00127 **
RAD	0.259893	0.087275	2.978	0.00290 **
TAX	-0.007966	0.004476	-1.780	0.07513 .
PTRATIO	-0.827576	0.138782	-5.963	2.47e-09 ***
B	0.006798	0.003070	2.214	0.02680 *

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

(Dispersion parameter for binomial family taken to be 1)

```
Null deviance: 353.03 on 254 degrees of freedom
Residual deviance: 135.08 on 248 degrees of freedom
AIC: 149.08
```

Number of Fisher Scoring iterations: 6

7. Compute the probabilities of "success" for cases in the validation partition and store them in a variable called PROB_SUCC:

```
> bh[-train.idx,"PROB_SUCC"] <- predict(logit, newdata = bh[-train.idx,], type="response")
```

8. Classify the cases using a cutoff probability of 0.5:

```
> bh[-train.idx,"PRED_50"] <- ifelse(bh[-train.idx, "PROB_SUCC"] >= 0.5, 1, 0)
```

9. Generate the error/classification-confusion matrix (your results could differ):

```
> table(bh[-train.idx, "CLASS"], bh[-train.idx, "PRED_50"], dnn=c("Actual", "Predicted"))
```

		Predicted	
		0	1
Actual	0	42	9
	1	10	47

How it works...

Steps 1 loads the `caret` package and step 2 reads the data file.

Step 3 converts the outcome variable to a factor. When the outcome variable is a factor, the `glm` function treats the first factor level as failure and the rest as "success." In the present case, we wanted it to treat "0" as failure and "1" as success. To force 0 to be the first level (and hence "failure") we specified `levels = c(0,1)`.

Step 4 creates the data partition, (we set the random `seed` to enable you to match your results with those we display).

Step 5 builds the logistic regression model, and stores it in the `logit` variable. Note that we have specified the data to be only the cases in the training partition.

Step 6 displays important information about the model. The `Deviance Residuals:` section gives us an idea of the spread of the deviation of the log odds and not of the probability. The `coefficients` section shows us that all the coefficients used are statistically significant.

Step 7 uses `logit`, our logistic regression model, to generate probabilities for the cases in the validation partition. There is no direct function to make actual classifications using the model. This is why we first generate the probabilities by specifying `type = "response"`.

Step 8 uses a cutoff probability of 0.5 to classify the cases. You can use a different value depending on the relative costs of misclassification for the different classes.

Step 9 generates the error/classification-confusion matrix for the preceding classification.

Using AdaBoost to combine classification tree models

R has several libraries that implement boosting where we combine many relatively inaccurate models to get a much more accurate model. The `ada` package provides boosting functionality on top of classification trees.

Getting ready

If you have not already installed the `ada` and `caret` package, install them now. Download the data files for this chapter from the book's website and place the `banknote-authentication.csv` file in your R working directory. We will use `class` as our target or outcome variable, and all the remaining variables as predictors.

How to do it...

To use AdaBoost for combining classification tree models, follow these steps:

1. Load the `caret` and `ada` packages:

```
> library(caret)
> library(ada)
```

2. Read the data:

```
> bn <- read.csv("banknote-authentication.csv")
```

3. Convert the outcome variable `class` to a factor:

```
> bn$class <- factor(bn$class)
```

4. Create partitions:

```
> set.seed(1000)
> t.idx <- createDataPartition(bn$class, p=0.7, list=FALSE)
```

5. Create an `rpart.control` object:

```
> cont <- rpart.control()
```

6. Build the model:

```
> mod <- ada(class ~ ., data = bn[t.idx,], iter=50, loss="e",
  type="discrete", control = cont)
```

7. View the model results—among other things, they show the error/classification-confusion matrix on the training partition (your results could differ because of random partitioning):

```
> mod

Call:
ada(class ~ ., data = bn[t.idx, ], iter = 50, loss = "e", type =
"discrete",
    control = cont)

Loss: exponential Method: discrete Iteration: 50

Final Confusion Matrix for Data:
          Final Prediction
True value      0     1
      0 534     0
      1     0 427

Train Error: 0

Out-Of-Bag Error: 0.002 iteration= 49

Additional Estimates of number of iterations:

train.err1 train.kap1
      33         33

8. Generate predictions on the validation partition:
> pred <- predict(mod, newdata = bn[-t.idx, ], type = "vector")

9. Build the error/classification-confusion matrix on the validation partition:
> table(bn[-t.idx, "class"], pred, dnn = c("Actual", "Predicted"))
```

How it works...

Step 1 loads the necessary `caret` and `ada` packages.

Step 2 reads in the data.

Step 3 converts the outcome variable `class` to a factor because we are applying a classification method.

Step 4 creates the partitions (we set the random seed to enable you to match your results with those we display).

The `ada` function uses the `rpart` function to generate many classification trees. To do this, it needs us to supply an `rpart.control` object. Step 5 creates a default `rpart.control()` object.

Step 6 builds the AdaBoost model. We pass the formula and the data frame for which we want to build the model and enter `type = "discrete"` to specify classification as opposed to regression. In addition, we also specify the number of boosting iterations and `loss="e"` for boosting under exponential loss.

Step 7 displays the model.

Step 8 builds the predictions on the validation partition and then step 9 builds the error/classification-confusion matrix.

4

Give Me a Number – Regression

In this chapter, you will cover:

- ▶ Computing the root mean squared error
- ▶ Building KNN models for regression
- ▶ Performing linear regression
- ▶ Performing variable selection in linear regression
- ▶ Building regression trees
- ▶ Building random forest models for regression
- ▶ Using neural networks for regression
- ▶ Performing k-fold cross-validation
- ▶ Performing leave-one-out-cross-validation

Introduction

In many situations, data analysts seek to make numerical predictions and use regression techniques to do so. Some examples can be the future sales of a product, the amount of deposits that a bank will receive during the next month, the number of copies that a particular book will sell, and the expected selling price for a used car. This chapter covers recipes to use R to apply several regression techniques.

Computing the root mean squared error

You may build a regression model and want to evaluate the model by comparing the model's predictions with the actual outcomes. You will generally evaluate a model's performance on the training data, but will rely on the model's performance on the hold out data to get an objective measure.

Getting ready

If you have not already downloaded the files for this chapter, do so now and ensure that the `rmse.csv` file is in your R working directory. The file has data about a set of actual prices and the predicted values from some regression method. We will compute the **root mean squared (RMS)** error of these predictions.

How to do it...

When using any regression technique, you will be able to generate predictions. This recipe shows you how to calculate the RMS error given the predicted and actual numerical values of the outcome variable:

1. Compute the RMS error as follows:

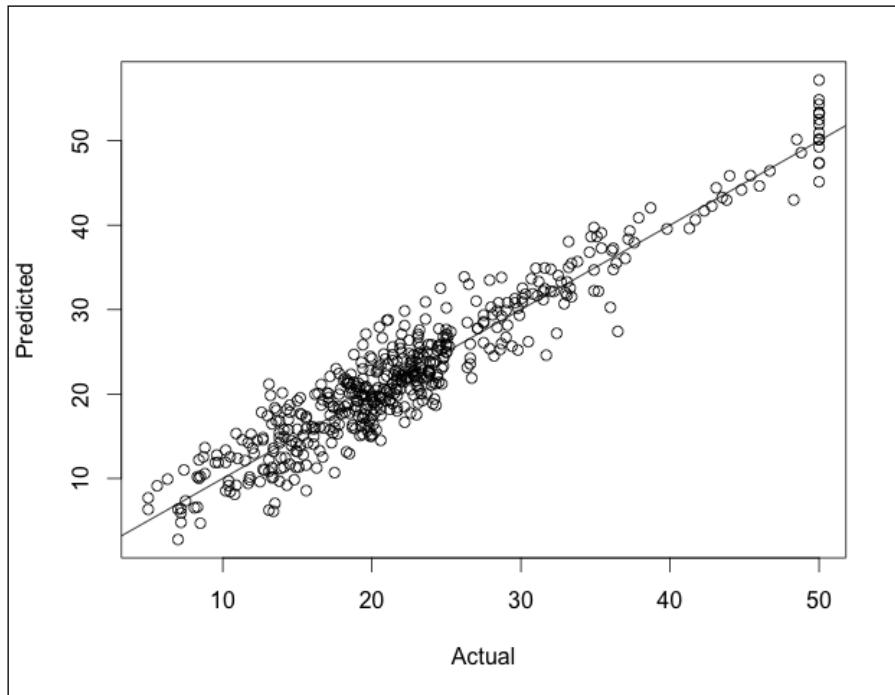
```
> dat <- read.csv("rmse-example.csv")
> rmse <- sqrt(mean((dat$price-dat$pred)^2))
> rmse

[1] 2.934995
```

2. Plot the results and show the 45 degree line:

```
> plot(dat$price, dat$pred, xlab = "Actual",
       ylab = "Predicted")
> abline(0, 1)
```

The following output is obtained as a result of the preceding command:



How it works...

Step 1 computes the RMS error as defined—the square root of the mean squared errors. The `dat$price - dat$pred` expression computes the vector of errors, and the code surrounding it computes the average of the squared errors and then finds the square root.

Step 2 generates the standard scatterplot and then adds on the 45 degree line.

There's more...

Since we compute RMS errors quite often, the following may be useful.

Using a convenience function to compute the RMS error

The following function may be handy when we need to compute the RMS error:

```
rdacb.rmse <- function(actual, predicted) {
  return (sqrt(mean((actual-predicted)^2)))
}
```

Armed with the function, we can compute the RMS error as:

```
> rmse <- rdacb.rmse(dat$price, dat$pred)
```

Building KNN models for regression

The FNN package provides the necessary functions to apply the KNN technique for regression. In this recipe, we look at the use of the `knn.reg` function to build the model and then the process of predicting with the model as well. We also show some additional convenience mechanisms to make the process easier.

Getting ready

Install the `FNN`, `dummies`, `caret`, and `scales` packages if you do not already have them installed. If you have not already downloaded the data files for this chapter, do so now and ensure that the `education.csv` file is in R working directory. The file has data about several school districts in the US. The following table describes the variables:

Variable	Meaning
state	US state code
region	Region of the country (1 = NE, ...)
urban	Number of residents per thousand residing in urban areas in 1970
income	Per-capita personal income in 1973
under18	Number of residents per thousand under 18 years of age in 1974
expense	Per capita expenditure on public education in a state, projected for 1975

We will build a `knn` model to predict `expense` based on all other predictors except `state`.

How to do it...

To build KNN models for regressions, follow these steps:

1. Load the `dummies`, `FNN`, `scales`, and `caret` packages as follows:

```
> library(dummies)
> library(FNN)
> library(scales)
```

2. Read the data:

```
> educ <- read.csv("education.csv")
```

3. Generate dummies for the categorical variable `region` and add them to `educ` as follows:

```
> dums <- dummy(educ$region, sep="_")
> educ <- cbind(educ, dums)
```

4. Because KNN performs distance computations, we should either rescale or standardize the predictors. In the present example, we have three numeric predictors and a categorical predictor in the form of three dummy variables. Standardizing dummy variables is tricky, and hence we will scale the numeric ones to [0, 1] and leave the dummies alone because they are already in the 0-1 range:

```
> educ$urban.s <- rescale(educ$urban)
> educ$income.s <- rescale(educ$income)
> educ$under18.s <- rescale(educ$under18)
```

5. Create three partitions (because we are creating random partitions, your results can differ) as follows:

```
> set.seed(1000)
> t.idx <- createDataPartition(educ$expense, p = 0.6,
  list = FALSE)
> trg <- educ[t.idx,]
> rest <- educ[-t.idx,]
> set.seed(2000)
> v.idx <- createDataPartition(rest$expense, p=0.5,
  list=FALSE)
> val <- rest[v.idx,]
> test <- rest[-v.idx,]
```

6. Build the model for several values of `k`. In the following code, we show how to compute the RMS error from scratch. You can also use the convenience `rdabch.rmse` function, which was shown in the recipe *Computing the root mean squared error* earlier in this chapter:

```
> # for k=1
> res1 <- knn.reg(trg[, 7:12], val[,7:12], trg[,6], 1,
  algorithm="brute")
> rmse1 = sqrt(mean((res1$pred-val[,6])^2))
> rmse1
```

```
[1] 59.66909
```

```
> # Alternately you could use the following to
> # compute the RMS error. See the recipe
> # "Compute the Root Mean Squared error" earlier
> # in this chapter
```

Give Me a Number – Regression

```
> rmse1 = rdacb.rmse(res1$pred, val[,6])  
  
> # for k=2  
> res2 <- knn.reg(trg[, 7:12], val[,7:12], trg[,6], 2,  
algorithm="brute")  
> rmse2 = sqrt(mean((res2$pred-val[,6])^2))  
> rmse2  
  
[1] 38.09002  
  
># for k=3  
> res3 <- knn.reg(trg[, 7:12], val[,7:12], trg[,6], 3,  
algorithm="brute")  
> rmse3 = sqrt(mean((res3$pred-val[,6])^2))  
> rmse3  
  
[1] 44.21224  
  
> # for k=4  
> res4 <- knn.reg(trg[, 7:12], val[,7:12], trg[,6], 4,  
algorithm="brute")  
> rmse4 = sqrt(mean((res4$pred-val[,6])^2))  
> rmse4  
  
[1] 51.66557
```

7. We obtained the lowest RMS error for k=2. Evaluate the model on the test partition as follows:

```
> res.test <- knn.reg(trg[, 7:12], test[,7:12], trg[,6], 2,  
algorithm="brute")  
> rmse.test = sqrt(mean((res.test$pred-test[,6])^2))  
rmse.test  
  
[1] 35.05442
```

We obtain a much lower RMS error on the test partition than on the validation partition. Of course, this cannot be trusted too much since our dataset was so small.

How it works...

Step 1 loads the required packages and step 2 reads the data.

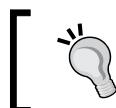
Since KNN requires all the predictors to be numeric, step 3 uses the `dummy` function from the `dummies` package to generate dummies for the categorical variable `region` and then adds the resulting dummy variables to the `educ` data frame.

Step 4 scales the numeric predictor variables to the $[0, 1]$ range using the `rescale` function from the `scales` package. Standardizing the numerical predictors will be another option, but standardizing dummy variables will be tricky. Some analysts standardize numerical predictors and leave the dummy variables as they are. However, for consistency, we choose to have all of our predictors in the $[0, 1]$ range. Since the dummies are already in that range, we rescale only the other predictors.

Step 5 creates the three partitions that KNN requires. We set the random seed to enable you to match your results with those that we display. See recipe *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis* for more details. Since we have only 50 cases in our dataset, we have chosen to partition it roughly into 60 %, 20 %, and 20 %. Instead of creating three partitions, we can manage with two and have the model building process use "leave one out" cross-validation. We discuss this under the *There's more...* section.

Step 6 builds the models for $k=1$ through $k=4$. We use only three of the four dummy variables. It invokes the `knn.reg` function and passes the following as arguments:

- ▶ Training predictors.
- ▶ Validation predictors.
- ▶ Outcome variable in the training partition.
- ▶ Value for k .
- ▶ The algorithm to use for distance computations. We specified `brute` to use the brute-force method.



If the dataset is large, one of the other options `kd_tree` or `cover_tree` may run faster.



Step 6 has highly repetitive code and we show a convenience function under the *There's more...* section to get all the results using a single command.

The model resulting from the call has several components. To compute the RMS error, we have used the `pred` component, which contains the predicted values.

Step 7 repeats the process for the test partition.

There's more...

Here we discuss some variations in running KNN.

Running KNN with cross-validation in place of validation partition

We used three partitions in the preceding code. A different approach will be to use two partitions. In this case, `knn.reg` will use "leave one out" cross-validation and predict for each case of the training partition itself. To use this mode, we pass only the training partition as argument and leave the other partition as `NULL`. After performing steps 1 through 4 from the main recipe, do the following:

```
> t.idx <- createDataPartition(educ$expense, p = 0.7,
  list = FALSE)
> trg <- educ[t.idx,]
> val <- educ[-t.idx,]
> res1 <- knn.reg(trg[,7:12], test = NULL, y = trg[,6],
  k=2, algorithm="brute")
> # When run in this mode, the result object contains
> # the residuals which we can use to compute rmse
> rmse <- sqrt(mean(res1$residuals^2))
> # and so on for other values of k
```

Using a convenience function to run KNN

We would normally run `knn` and compute the RMS error. The following convenience function can help:

```
rdacb.knn.reg <- function (trg_predictors, val_predictors,
  trg_target, val_target, k) {
  library(FNN)
  res <- knn.reg(trg_predictors, val_predictors, trg_target,
    k, algorithm = "brute")
  errors <- res$pred - val_target
  rmse <- sqrt(sum(errors * errors)/nrow(val_predictors))
  cat(paste("RMSE for k=", toString(k), ":", sep = ""), rmse,
    "\n")
  rmse
}
```

With the preceding function, we can execute the following after reading the data, creating dummies, rescaling the predictors and partitioning—that is, executing steps 1 through 4 of the main recipe:

```
> set.seed(1000)
> t.idx <- createDataPartition(educ$expense, p = 0.6, list = FALSE)
> trg <- educ[t.idx,]
> rest <- educ[-t.idx,]
> set.seed(2000)
> v.idx <- createDataPartition(rest$expense, p=0.5, list=FALSE)
> val <- rest[v.idx,]
```

```
> test <- rest[-v.idx,]
> rdacb.knn.reg(trg[,7:12], val[,7:12], trg[,6], val[,6], 1)

RMSE for k=1: 59.66909
[1] 59.66909
> rdacb.knn.reg(trg[,7:12], val[,7:12], trg[,6], val[,6], 2)

RMSE for k=2: 38.09002
[1] 38.09002

> # and so on
```

Using a convenience function to run KNN for multiple k values

Running `knn` for several values of `k` to choose the best one involves repetitively executing almost similar lines of code several times. We can automate the process with the following convenience function that runs `knn` for multiple values of `k`, reports the RMS error for each, and also produces a scree plot of the RMS errors:

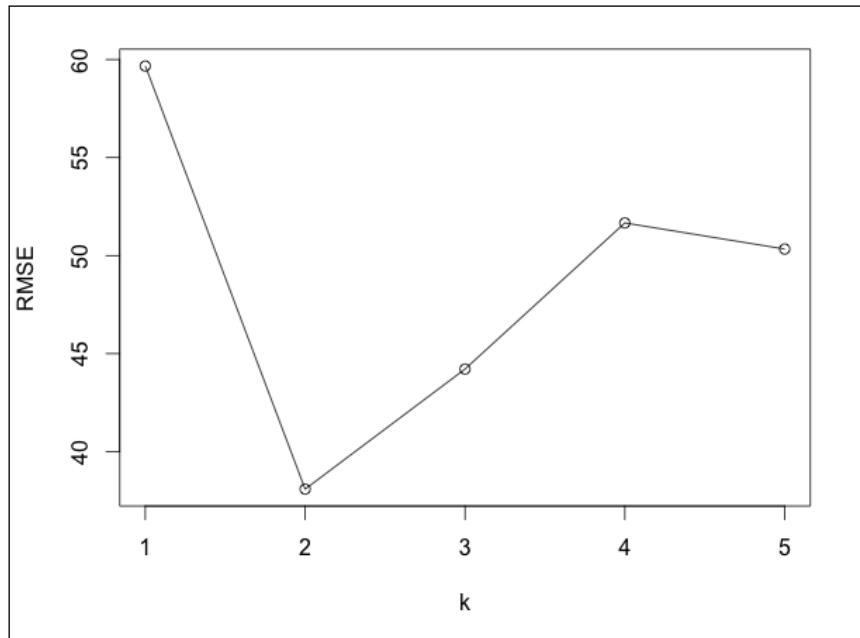
```
rdacb.knn.reg.multi <- function (trg_predictors, val_predictors, trg_target, val_target, start_k, end_k)
{
  rms_errors <- vector()
  for (k in start_k:end_k) {
    rms_error <- rdacb.knn.reg(trg_predictors, val_predictors,
                                trg_target, val_target, k)
    rms_errors <- c(rms_errors, rms_error)
  }
  plot(rms_errors, type = "o", xlab = "k", ylab = "RMSE")
}
```

With the preceding function, we can execute the following after reading the data, creating dummies, rescaling the predictors and partitioning—that is, executing steps 1 through 4 of the main recipe. The code runs `knn.reg` for values of `k` from 1 to 5:

```
> rdacb.knn.reg.multi(trg[,7:12], val[,7:12], trg[,6], val[,6], 1, 5)

RMSE for k=1: 59.66909
RMSE for k=2: 38.09002
RMSE for k=3: 44.21224
RMSE for k=4: 51.66557
RMSE for k=5: 50.33476
```

The preceding code also produces a plot of the RMS errors, as shown in the following:



See also...

- ▶ *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis*
- ▶ *Computing the root mean squared error* in this chapter
- ▶ *Classifying using the KNN approach* in Chapter 3, *Where Does It Belong? – Classification*

Performing linear regression

In this recipe, we discuss linear regression, arguably the most widely used technique. The `stats` package has the functionality for linear regression and R loads it automatically at startup.

Getting ready

If you have not already done so, download the data files for this chapter and ensure that the `auto-mpg.csv` file is in your R working directory. Install the `caret` package if you have not already done so. We want to predict `mpg` based on `cylinders`, `displacement`, `horsepower`, `weight`, and `acceleration` variables.

How to do it...

To perform linear regression, follow these steps:

1. Load the caret package:

```
> library(caret)
```

2. Read the data:

```
> auto <- read.csv("auto-mpg.csv")
```

3. Convert the categorical variable cylinders into a factor with appropriate renaming of the levels:

```
> auto$cylinders <- factor(auto$cylinders,  
levels = c(3,4,5,6,8), labels = c("3cyl", "4cyl", "5cyl",  
"6cyl", "8cyl"))
```

4. Create partitions:

```
> set.seed(1000)  
> t.idx <- createDataPartition(auto$mpg, p = 0.7,  
list = FALSE)
```

5. See the names of the variables in the data frame:

```
> names(auto)
```

```
[1] "No"          "mpg"  
[3] "cylinders"   "displacement"  
[5] "horsepower"   "weight"  
[7] "acceleration" "model_year"  
[9] "car_name"
```

6. Build the linear regression model:

```
> mod <- lm(mpg ~ ., data = auto[t.idx, -c(1,8,9)])
```

7. View the basic results (your results may differ because of random sampling differences in creating the partitions):

```
> mod
```

Call:

```
lm(formula = mpg ~ ., data = auto[t.idx, -c(1, 8, 9)])
```

Coefficients:

	(Intercept)	cylinders4cyl	cylinders5cyl	cylinders6cyl
	39.450422	6.466511	4.769794	1.967411
cylinders8cyl		displacement	horsepower	weight

Give Me a Number – Regression

```
6.291938      0.004790      -0.081642      -0.004666  
acceleration  
0.003576
```

8. View more detailed results (your results may differ because of random sampling differences in creating the partitions):

```
> summary(mod)
```

```
Call:  
lm(formula = mpg ~ ., data = auto[-t.idx, -c(1, 8, 9)])  
  
Residuals:  
    Min      1Q  Median      3Q     Max  
-9.8488 -2.4015 -0.5022  1.8422 15.3597  
  
Coefficients:  
            Estimate Std. Error t value Pr(>|t|)  
(Intercept) 39.4504219  3.3806186 11.670 < 2e-16 ***  
cylinders4cyl 6.4665111  2.1248876  3.043  0.00257 **  
cylinders5cyl 4.7697941  3.5603033  1.340  0.18146  
cylinders6cyl 1.9674114  2.4786061  0.794  0.42803  
cylinders8cyl 6.2919383  2.9612774  2.125  0.03451 *  
displacement   0.0047899  0.0109108  0.439  0.66100  
horsepower     -0.0816418  0.0200237 -4.077 5.99e-05 ***  
weight         -0.0046663  0.0009857 -4.734 3.55e-06 ***  
acceleration   0.0035761  0.1426022  0.025  0.98001  
---  
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 3.952 on 271 degrees of freedom  
Multiple R-squared:  0.756,      Adjusted R-squared:  0.7488  
F-statistic: 105 on 8 and 271 DF,  p-value: < 2.2e-16
```

9. Generate predictions for the test data:

```
> pred <- predict(mod, auto[-t.idx, -c(1,8,9)])
```

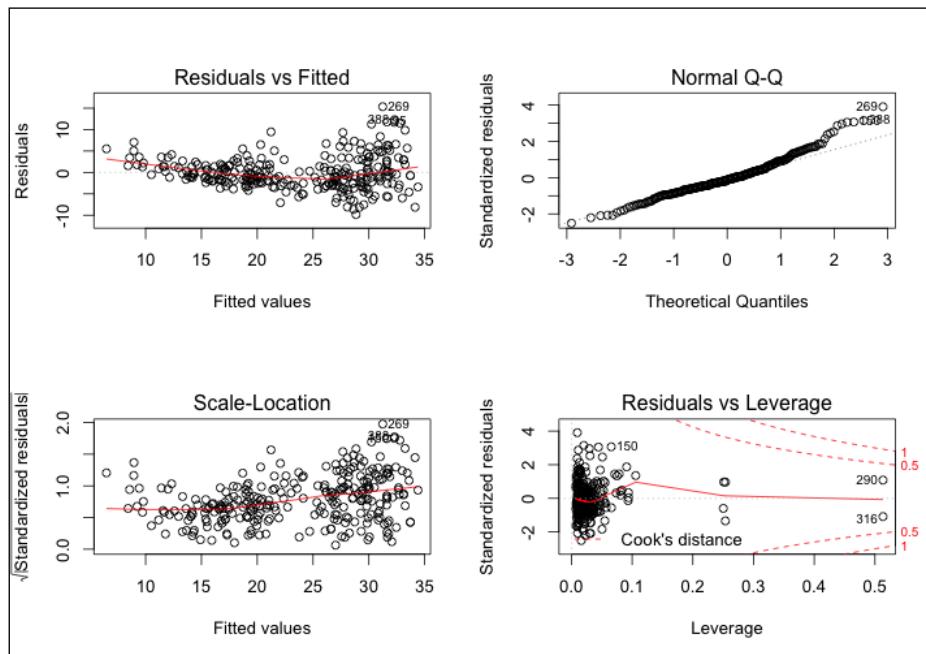
10. Compute the RMS error on the test data (your results can differ):

```
> sqrt(mean((pred - auto[-t.idx, 2])^2))  
[1] 4.333631
```

11. View diagnostic plots of the model:

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

The following diagnostic plots are obtained as an output:



How it works...

Step 1 loads the `caret` package, step 2 reads the data, and step 3 converts the categorical variable `cylinders` (which has numeric values which R treats as a number by default) into a factor.

Step 4 creates the partitions—see recipe *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis* for more details. We set the random seed to enable you to match your results with what we have displayed.

Step 5 prints the variable names in the file so that we can use the appropriate variables in the linear regression model.

Step 6 uses the `lm` function which builds the linear regression model. We specified `data = auto[t.idx, -c(1, 8, 9)]` because we want the model to use only the training data and because we do not want to use `No`, `model_year`, and `car_name`, which correspond to variables 1, 8, and 9, respectively. We could instead have included all variables, but that would have meant having to explicitly specify only the required predictors in the formula expression. We chose the shorter version.

Give Me a Number – Regression

Although one of our predictors, `cylinders`, is a factor (categorical variable), we did not generate dummies for it because the `lm` function takes care of this automatically, and the regression coefficients in the output show this clearly.

Step 7 shows how we can simply print the value of the model variable to get the values of the regression coefficients.

Step 8 uses the `summary` function to get more information about the model:

```
Residuals:
    Min      1Q  Median      3Q     Max 
-9.8488 -2.4015 -0.5022  1.8422 15.3597 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 39.4504219 3.3806186 11.670 < 2e-16 ***
cylinders4cyl 6.4465111 2.1248876  3.043 0.00257 **  
cylinders5cyl 4.7697941 3.5603033  1.340 0.18146    
cylinders6cyl 1.9674114 2.4796061  0.794 0.42803    
cylinders8cyl 6.29193   0.00478   15.03451 *   
displacement 0.00478   -0.0816418  0.0200237 -4.077 5.99e-05 *** 
horsepower    -0.0046663 0.0009857  -4.734 3.55e-06 *** 
weight        -0.0046663 0.0009857  -4.734 3.55e-06 *** 
acceleration  0.0035761 0.1426022   0.025 0.98001    
---
Signif. codes:  0 '***' 1 ' .' 0.1 ' ' 0.1

Residual standard error: 7.057 on 271 degrees of freedom
Multiple R-squared:  0.7488
F-statistic: 105 on 8 and 271 DF, p-value: < 2.2e-16
```

In the detailed output, the **Residuals** section shows the distribution of the residuals on the training data through the quartiles. The **Coefficients** section gives details about the coefficients. The first column, **Estimate**, gives the estimates of the regression coefficients. The second column, **Std. Error**, gives the standard error of that estimate. The third column converts this standard error into a **t** value by dividing the coefficient by the standard error. The **Pr (> |t|)** column converts the **t** value into a probability of the coefficient being 0. The annotation after the last column symbolically shows the level of significance of the coefficient estimate by a dot, blank, or a few stars. The legends below the table explain what the annotation means. It is customary to consider a coefficient to be significant at a 95 % level of significance (that is, probability being less than 0.05), which is represented by a "*".

The next section gives information about how the regression performed as a whole. The **Residual standard error** is just the RMS adjusted for the degrees of freedom and is an excellent indicator of the average deviation of the predicted value from the actual value. The **Adjusted R-squared** value tells us what percentage of the variation in the outcome variable the regression model explains. The last line shows the **F-statistic** and the corresponding p-value for the whole regression.

Step 9 generates the predictions on the test data using the `predict` function.

Step 10 computes the RMS error.

Step 11 generates the diagnostic plots. Since the standard `plot` function for `lm` produces four plots, we set up a matrix of four plots up front before calling the function and reset it after we finish plotting:

- ▶ `Residuals vs Fitted`: As its name suggests, this plots the residuals against the values fitted by the model to enable us to examine if the residuals exhibit a trend. Ideally, we would like them to be trendless and almost a horizontal straight line at 0. In our example, we see a very slight trend.
- ▶ `Normal Q-Q`: This plot helps us to check the extent to which the residuals are normally distributed by plotting the standardized residuals against the theoretical quartiles for the standard normal distribution. If the points all lie close to the 45 degree line, then we will know that the normality condition is met. In our example, we note that at the right extreme or at high values of the residuals, the standardized residuals are higher than expected and do not meet the normality requirement.
- ▶ `Scale Location`: This plot is very similar to the first plot, except that the square root of the standardized residuals is plotted against the fitted values. Once again this is used to detect if the residuals exhibit any trend.
- ▶ `Residuals vs Leverage`: You can use this plot to identify outlier cases that are exerting undue influence on the model. In our example, the labeled cases 150, 290, and 316 can be candidates for removal.

There's more...

We discuss in the following sections a few options for using the `lm` function.

Forcing `lm` to use a specific factor level as the reference

By default, `lm` uses the lowest factor as the reference level. To use a different one as reference, we can use the `relevel` function. Our original model uses `3cyl` as the reference level. To force `lm` to instead use `4cyl` as the reference:

```
> auto <- within(auto, cylinders <- relevel(cylinders,  
ref = "4cyl") )  
> mod <- lm(mpg ~., data = auto[t.idx, -c(1, 8, 9)])
```

The resulting model will not have a coefficient for `4cyl`.

Using other options in the formula expression for linear models

Our example only showed the most common form of the formula for `lm`. The following table shows options to create models with interaction effects or to create models that apply arbitrary functions to predictor variables:

Formula expression	Corresponding regression model	Explanation
$Y \sim P$	$\hat{Y} \sim \beta_0 + \beta_1 P$	Straight line with Y intercept
$Y \sim P + Q$	$\hat{Y} \sim \beta_0 + \beta_1 P + \beta_2 Q$	Linear model with P and Q with no interaction terms
$Y \sim -1 + P$	$\hat{Y} \sim \beta_1 P$	Linear model with no intercept term
$Y \sim P : Q$	$\hat{Y} \sim \beta_0 + \beta_1 PQ$	Model with only the first order interaction terms for P and Q
$Y \sim P * Q$ or $Y \sim P + Q + P : Q$	$\hat{Y} \sim \beta_0 + \beta_1 P + \beta_2 Q + \beta_3 PQ$	Complete first order model with all interaction terms
$Y \sim P + I(\log(Q))$	$\hat{Y} \sim \beta_0 + \beta_1 P + \beta_2 \log(Q)$	Model with arbitrary function applied on predictor variable. The <code>I</code> or Identity function is used for this.
$Y \sim (P + Q + R)^2$ or $Y \sim P * Q * R - P : Q : R$	$\hat{Y} \sim \beta_0 + \beta_1 P + \beta_2 Q + \beta_3 R + \beta_4 PQ + \beta_5 QR + \beta_6 PR$	Complete first order model and interaction terms for all orders up to the nth order where n is the exponent

See also...

- ▶ *Creating random data partitions in Chapter 2, What's in There? – Exploratory Data Analysis*
- ▶ *Performing variable selection in linear regression* in this chapter

Performing variable selection in linear regression

The MASS package has the functionality for variable selection and this recipe illustrates its use.

Getting ready

If you have not already done so, download the data files for this chapter and ensure that the `auto-mpg.csv` file is in your R working directory. We want to predict `mpg` based on `cylinders`, `displacement`, `horsepower`, `weight`, and `acceleration`.

How to do it...

To perform variable selection in linear regression, follow the steps below:

1. Load the `caret` and `MASS` packages:

```
> library(caret)
> library(MASS)
```

2. Read the data:

```
> auto <- read.csv("auto-mpg.csv")
```

3. Convert the categorical variable `cylinders` into a factor with appropriate renaming of the levels:

```
> auto$cylinders <- factor(auto$cylinders,
  levels = c(3,4,5,6,8), labels = c("3cyl", "4cyl", "5cyl", "6cyl",
  "8cyl"))
```

4. Create partitions:

```
> set.seed(1000)
> t.idx <- createDataPartition(auto$mpg, p = 0.7, list = FALSE)
```

5. See the names of the variables in the data frame:

```
> names(auto)
[1] "No"          "mpg"
[3] "cylinders"   "displacement"
[5] "horsepower"  "weight"
[7] "acceleration" "model_year"
[9] "car_name"
```

6. Build the linear regression model:

```
> fit <- lm(mpg ~ ., data = auto[t.idx, -c(1,8,9)])
```

Give Me a Number – Regression

7. Run the variable selection procedure. This will produce quite a lot of output, which we will display and discuss later. Because of random partitioning, your actual numbers will vary:

```
> step.model <- stepAIC(fit, direction = "backward")
```

8. See the final model (your results may differ because of variations in your training sample):

```
> summary(step.model)
Call:
lm(formula = mpg ~ cylinders + horsepower + weight, data = auto[t.
idx,
-c(1, 8, 9)])
```

Residuals:

Min	1Q	Median	3Q	Max
-9.7987	-2.3676	-0.6214	1.8625	15.3231

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)		
(Intercept)	39.1290155	2.5434458	15.384	< 2e-16 ***		
cylinders4cyl	6.7241124	2.0140804	3.339	0.000959 ***		
cylinders5cyl	5.0579997	3.4762178	1.455	0.146810		
cylinders6cyl	2.5090718	2.1315214	1.177	0.240170		
cylinders8cyl	7.0991790	2.3133286	3.069	0.002365 **		
horsepower	-0.0792425	0.0148396	-5.340	1.96e-07 ***		
weight	-0.0044670	0.0007512	-5.947	8.34e-09 ***		

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

Residual standard error: 3.939 on 273 degrees of freedom
Multiple R-squared: 0.7558, Adjusted R-squared: 0.7505
F-statistic: 140.9 on 6 and 273 DF, p-value: < 2.2e-16

How it works...

For a description of steps 1 through 6, refer to the *How it works...* section of the *Performing linear regression* recipe in this chapter (the previous recipe).

Step 7 runs the variable selection procedure. We have chosen to illustrate *backward elimination* in which the system first builds the model with all the predictors and eliminates predictors based on AIC scores. We show the sample output in the following:

```
Start: AIC=778.38
mpg ~ cylinders + displacement + horsepower + weight + acceleration
```

```
Df Sum of Sq    RSS    AIC
- acceleration 1     0.01 4232.1 776.38
- displacement 1     3.01 4235.1 776.58
<none>                   4232.1 778.38
- horsepower    1     259.61 4491.7 793.05
- weight        1     349.99 4582.1 798.63
- cylinders     4     859.84 5091.9 822.17

Step: AIC=776.38
mpg ~ cylinders + displacement + horsepower + weight

Df Sum of Sq    RSS    AIC
- displacement 1     3.02 4235.1 774.58
<none>                   4232.1 776.38
- horsepower    1     404.33 4636.4 799.93
- weight        1     451.22 4683.3 802.75
- cylinders     4     862.88 5094.9 820.34

Step: AIC=774.58
mpg ~ cylinders + horsepower + weight

Df Sum of Sq    RSS    AIC
<none>                   4235.1 774.58
- horsepower    1     442.36 4677.4 800.40
- weight        1     548.60 4783.7 806.69
- cylinders     4     862.50 5097.6 818.49
```

From the preceding output, you can see that the system first built the complete model. In that model, acceleration had the lowest AIC score of 776.38 and was therefore not included in the next one. In this process, the system eliminated displacement as well.

The complete model had five predictors and the final one has three. In the process, the multiple R² has remained almost unchanged, but we got a less complex model.



In the forward selection model, the system takes the reverse approach and adds predictors.

See also...

- ▶ *Creating random data partitions in Chapter 2, What's in There? – Exploratory Data Analysis*
- ▶ *Performing linear regression in this chapter*

Building regression trees

This recipe covers the use of tree models for regression. The `rpart` package provides the necessary functions to build regression trees.

Getting ready

Install the `rpart`, `caret`, and `rpart.plot` packages if you do not already have them installed. If you have not already downloaded the data files for this chapter, do so now and ensure that the `BostonHousing.csv` and `education.csv` files are in the R working directory.

How to do it...

To build regression trees, follow the steps below:

1. Load the `rpart`, `rpart.plot`, and `caret` packages:

```
> library(rpart)
> library(rpart.plot)
> library(caret)
```

2. Read the data:

```
> bh <- read.csv("BostonHousing.csv")
```

3. Partition the data:

```
> set.seed(1000)
> t.idx <- createDataPartition(bh$MEDV, p=0.7, list = FALSE)
```

4. Build and view the regression tree model:

```
> bfit <- rpart(MEDV ~ ., data = bh[t.idx,])
> bfit
n= 356
```

```
node), split, n, deviance, yval
      * denotes terminal node
```

```
1) root 356 32071.8400 22.61461
   2) LSTAT>=7.865 242 8547.6860 18.22603
      4) LSTAT>=14.915 114 2451.4590 14.50351
         8) CRIM>=5.76921 56 796.5136 11.63929 *
         9) CRIM< 5.76921 58 751.9641 17.26897 *
      5) LSTAT< 14.915 128 3109.5710 21.54141
     10) DIS>=1.80105 121 1419.7510 21.12562 *
     11) DIS< 1.80105 7 1307.3140 28.72857 *
```

```

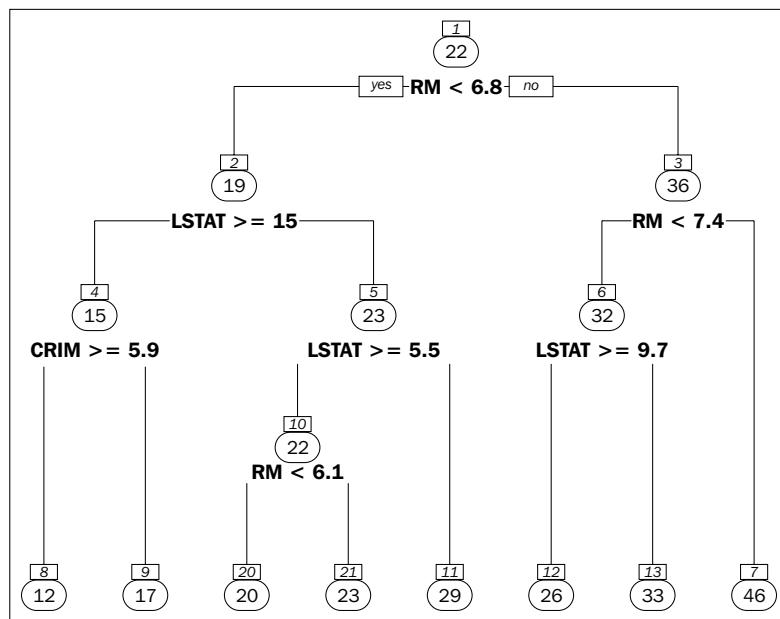
3) LSTAT< 7.865 114 8969.3230 31.93070
6) RM< 7.4525 93 3280.1050 28.70753
12) RM< 6.659 46 1022.5320 25.24130 *
13) RM>=6.659 47 1163.9800 32.10000
26) LSTAT>=5.495 17 329.2494 28.59412 *
27) LSTAT< 5.495 30 507.3747 34.08667 *
7) RM>=7.4525 21 444.3295 46.20476 *

```

5. Plot the tree. Use the `prp` function from the `rpart.plot` package and select the options shown in the following to get a good-looking plot. For convenience, the plot rounds off the y values.

```
> prp(bfit, type=2, nn=TRUE, fallen.leaves=TRUE, faclen=4,
varlen=8, shadow.col="gray")
```

The plot obtained appears as follows:



6. Look at the `cptable`. Your `cptable` may differ from that shown in the following output because of the random numbers used in the cross-validation process:

```
> bfit$cptable
```

	CP	nsplit	rel error	xerror	xstd
1	0.45381973	0	1.0000000	1.0068493	0.09724445
2	0.16353560	1	0.5461803	0.6403963	0.06737452
3	0.09312395	2	0.3826447	0.4402408	0.05838413

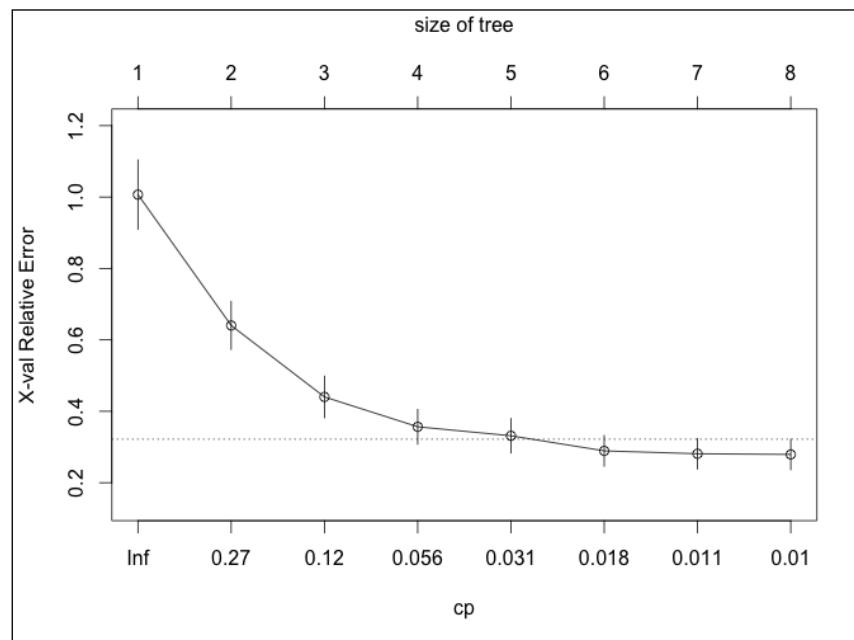
Give Me a Number – Regression

4 0.03409823	3 0.2895207 0.3566122 0.04889254
5 0.02815494	4 0.2554225 0.3314437 0.04828523
6 0.01192653	5 0.2272675 0.2891804 0.04306039
7 0.01020696	6 0.2153410 0.2810795 0.04286100
8 0.01000000	7 0.2051341 0.2791785 0.04281285

7. You can either choose the tree with the lowest cross-validation error (`xerror`) or use the 1 SD rule and choose the tree that comes to within 1 SD (`xstd`) of the minimum `xerror` and has fewer nodes. The former approach will cause us to select the tree with seven splits (on the last row). That tree will have eight nodes. To apply the latter approach, $\min \text{xerror} + 1 \text{ SE} = 0.2791785 + 0.04281285 = 0.3219914$ and hence leads us to select the tree with five splits (on row 6).
8. You can simplify the process by just plotting `cptree` and using the resulting plot to select the cutoff value to use for pruning. The plot shows the size of the tree—which is one more than the number of splits. The table and the plot differ in another important way—the complexity or `cp` values in these differ. The table shows the minimum `cp` value for which corresponding split occurs. The plot shows the geometric means of the successive splits. As with the table, your plot may differ because of the random numbers used during cross-validation:

```
> plotcp(bfit)
```

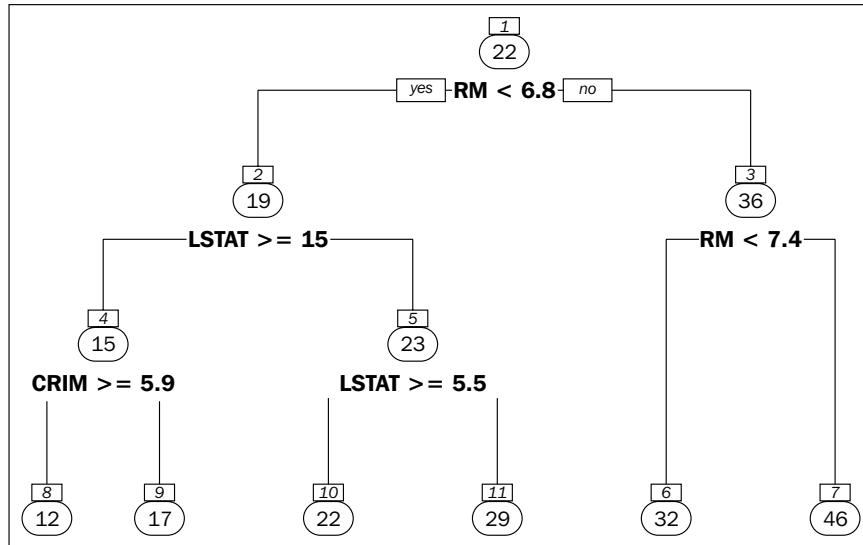
To select the best `cp` value from the plot using the 1 SD rule, pick the leftmost `cp` value for which the cross-validation relative error (y axis) lies below the dashed line. Using this value, we will pick a `cp` value of 0.018:



9. Prune the tree with the chosen `cp` value and plot it as follows:

```
> # In the command below, replace the cp value
> # based on your results
> bfitpruned <- prune(bfit, cp= 0.01192653)
> prp(bfitpruned, type=2, nn=TRUE, fallen.leaves=TRUE, faclen=4,
varlen=8, shadow.col="gray")
```

The following output is obtained:



10. Use the chosen tree to compute the RMS error for the training partition:

```
> preds.t <- predict(bfitpruned, bh[t.idx,])
> sqrt(mean((preds.t-bh[t.idx, "MEDV"])^2))
[1] 4.524866
```

11. Generate predictions and the RMS error for the validation partition:

```
preds.v <- predict(bfitpruned, bh[-t.idx,])
> sqrt(mean((preds.v - bh[-t.idx, "MEDV"])^2))
[1] 4.535723
```

How it works...

Steps 1 and 2 load the required packages and read the data.

Step 3 partitions the data. See recipe *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis* for more details. We set the random seed to enable you to match your results with those that we have displayed.

Step 4 uses the `rpart` function to build the tree model. It passes the formula as `MEDV ~ .` to indicate that `MEDV` is the outcome and that all the remaining variables will be predictors. It specifies `data = bh[t.idx,]` to indicate that only the rows in the training partition should be used to build the model. It then prints the model details in textual form. The output shows information for the root node and subsequently for each split. Each row has the following information:

- ```
node), split, n, deviance, yval
```
- ▶ node number
  - ▶ splitting condition that generated the node (for the root it just says "root")
  - ▶ number of cases at the node
  - ▶ sum of squared errors at the node based on average value of outcome variable of the cases at the node
  - ▶ average value of outcome variable of the cases at the node

We have many options to control how the `rpart` function works. It uses the following important defaults among others:

- ▶ 0.01 for the complexity factor, `cp`
- ▶ Minimum node size of 20 to split a node, `minsplit`
- ▶ The function does not split if a split will create a node with less than `round(minsplit/3)` cases, `minbucket`

You can control these by passing an `rpart.control` object while invoking `rpart`. The following code shows an example. Here we use values of 0.001, 10, and 5 for `cp`, `minsplit` and `minbucket`, respectively:

```
> fit <- rpart(MEDV ~ ., data = bh[t.idx,], control = rpart.
control(minsplit = 10, cp = 0.001, minbucket = 5)
```

Step 5 plots the tree model using the `prp` function from the `rpart.plot` package. The function provides several parameters through which we can control the plot's appearance. We describe a few options as follows; check the documentation for the other numerous options:

- ▶ `Type`: This refers to the amount of information and its placement
- ▶ `nn`: This refers to whether to display node numbers or not
- ▶ `fallen.leaves`: This refers to whether to display all the leaf nodes at the same level (bottom most)—this results in a plot with only horizontal and vertical lines and make it easier on the eye; otherwise, the plot has diagonal lines
- ▶ `faclen`: This refers to the length of factor level names in the splits – abbreviates if needed

- ▶ `varlen`: This refers to the length of variable names on the plot – truncates if needed
- ▶ `shadow.col`: This refers to the color of the shadow that each node casts

Step 6 prints the `cptable`, which is a component of the fitted tree model. The `cptable` shows comprehensive results of trees with differing number of nodes as well as the mean and standard deviation of the error on cross-validation for each tree size. This information helps us to select our optimal tree. We explain the columns of the table as follows:

- ▶ `cp`: This refers to the complexity factor.
- ▶ `nsplit`: This refers to the number of splits in the best tree that the corresponding `cp` yields.
- ▶ `rel error`: For the best tree with the specified number of splits, the overall squared classification error (on the data used to build the tree) as a proportion of the total squared error at the root node. The error at the root node is based on predicting every case as the average of the value of outcome variable across all cases.
- ▶ `xerror`: This refers to mean cross-validation error using the best tree with the specified number of splits.
- ▶ `xstd`: This refers to standard deviation of the cross-validation error using the best tree with the specified number of splits.

Step 7 explains how we can use the information in `cptable` to prune the tree and prevent overfitting. We can choose either the tree with the lowest cross-validation error or the smallest one that comes within one SD of the cross-validation error. We can select the `cp` value corresponding to the selected tree and use that value to prune the tree.

Step 8 shows an easier way to select the best `cp` value by plotting the `cptable` with the `plotcp` function.

Step 9 prunes the tree with the chosen `cp` value.

Step 10 uses the `predict` function to generate predictions for the training partition and then computes the RMS error.

Step 11 does the same for the validation partition.

### There's more...

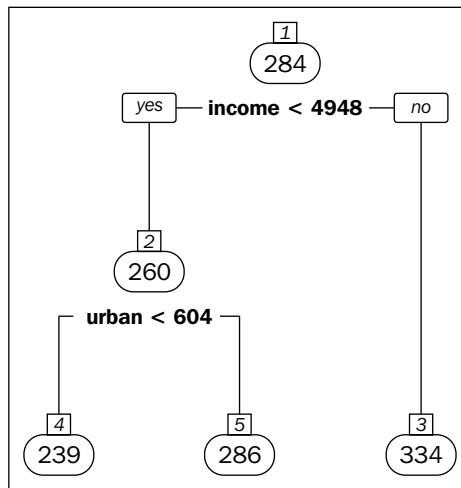
Regression trees can also be built for categorical predictors as explained in this section.

## Generating regression trees for data with categorical predictors

The `rpart` function works even when a dataset has categorical predictor variables. You just have to ensure that the variable is tagged as a factor. See the following example:

```
> ed <- read.csv("education.csv")
> ed$region <- factor(ed$region)
> set.seed(1000)
> t.idx <- createDataPartition(ed$expense, p = 0.7, list = FALSE)
> fit <- rpart(expense ~ region+urban+income+under18, data = ed[t.
idx,])
> prp(fit, type=2, nn=TRUE, fallen.leaves=TRUE, faclen=4, varlen=8,
shadow.col="gray")
```

The following output is obtained:



### See also...

- ▶ *Creating random data partitions in Chapter 2, What's in There? – Exploratory Data Analysis*
- ▶ *Building, plotting, and evaluating classification trees in Chapter 3, Where Does It Belong? – Classification*

## Building random forest models for regression

This recipe looks at random forests—one of the most successful machine learning techniques.

### Getting ready

If you have not already installed the `randomForest` and `caret` packages, install them now. Download the data files for this chapter from the book's website and place the `BostonHousing.csv` file in your R working directory. We will build a random forest model to predict `MEDV` based on the other variables.

### How to do it...

To build random forest models for regression, follow the steps below:

1. Load the `randomForest` and `caret` packages:

```
> library(randomForest)
> library(caret)
```

2. Read the data:

```
> bn <- read.csv("BostonHousing.csv")
```

3. Partition the data:

```
> set.seed(1000)
> t.idx <- createDataPartition(bh$MEDV, p=0.7, list=FALSE)
```

4. Build the random forest model. Since this command builds many regression trees, it can take significant processing time on even moderate datasets:

```
> mod <- randomForest(x = bh[t.idx,1:13],
y=bh[t.idx,14],ntree=1000, xtest = bh[-t.idx,1:13],
ytest = bh[-t.idx,14], importance=TRUE, keep.forest=TRUE)
```

5. Examine the results (your results may differ slightly because of the random factor):

```
> mod
Call:
randomForest(x = bh[t.idx, 1:13], y = bh[t.idx, 14], xtest =
bh[-t.idx, 1:13], ytest = bh[-t.idx, 14], ntree = 1000,
importance = TRUE, keep.forest = TRUE)
 Type of random forest: regression
 Number of trees: 1000
No. of variables tried at each split: 4
```

Mean of squared residuals: 12.61296

## Give Me a Number – Regression

---

```
% Var explained: 86
Test set MSE: 6.94
% Var explained: 90.25
```

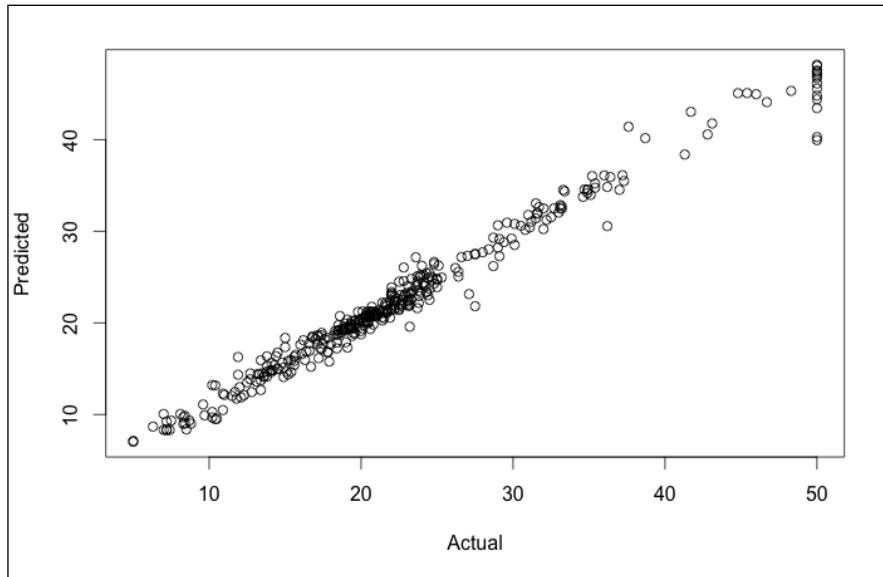
6. Examine variable importance:

```
> mod$importance
 %IncMSE IncNodePurity
CRIM 9.5803434 2271.5448
ZN 0.3410126 142.1191
INDUS 6.6838954 1840.7041
CHAS 0.6363144 193.7132
NOX 9.3106894 1922.5483
RM 36.2790912 8540.4644
AGE 3.7186444 820.7750
DIS 7.4519827 2012.8193
RAD 1.7799796 287.6282
TAX 4.5373887 1049.3716
PTRATIO 6.8372845 2030.2044
B 1.2240072 530.1201
LSTAT 67.0867117 9532.3054
```

7. Compare predicted and actual values for the training partition:

```
> plot(bh[t.idx,14], predict(mod, newdata=bh[t.idx,]), xlab =
"Actual", ylab = "Predicted")
```

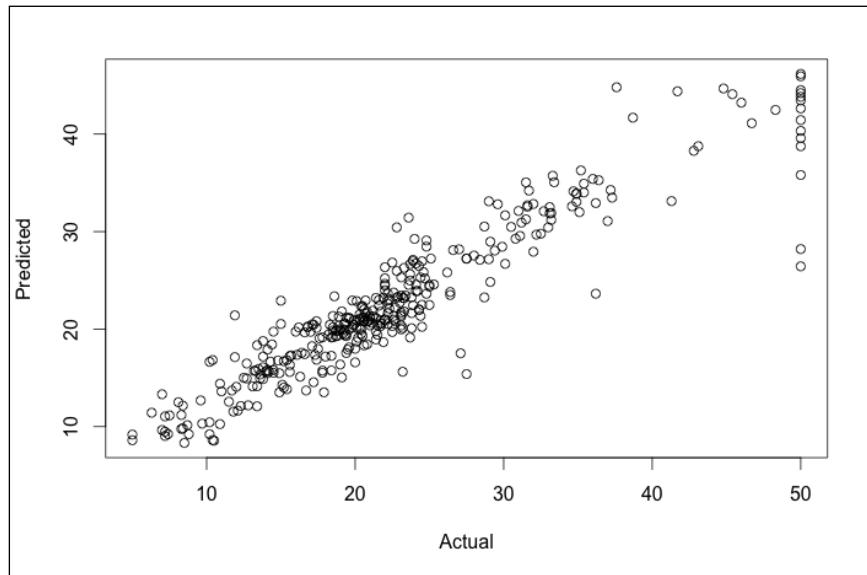
The following output is obtained on executing the preceding command:



8. Compare the **out of bag (OOB)** predictions with actuals in the training partition:

```
> > plot(bh[t.idx,14], mod$predicted, xlab = "Actual", ylab =
"Predicted")
```

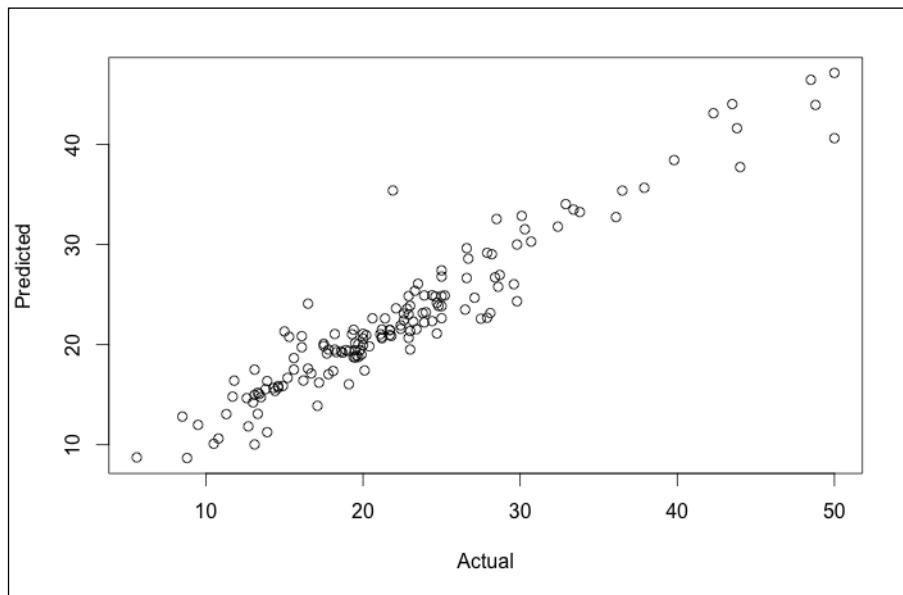
The preceding command produces the following output:



9. Compare predicted and actual values for the test partition:

```
> plot(bh[-t.idx,14], mod$test$predicted, xlab = "Actual", ylab =
"Predicted")
```

The following plot is obtained as a result of the preceding command:



## How it works...

Steps 1 and 2 load the necessary packages and read the data.

Step 3 partitions the data. See recipe *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis* for more details. We set the random seed to enable you to match your results with those that we have displayed. Technically speaking, we do not really need to partition the data for random forests because it builds many trees and uses only a subset of the data each time. Thus, each case is OOB for about a third of the trees built and can be used for validation. However, the method also provides for us to provide a validation dataset separately and we illustrate that process here.

Step 4 builds the random forest model. We show the command and describe the arguments as follows:

```
> mod <- randomForest(x = bh[t.idx,1:13],
y=bh[t.idx,14],ntree=1000, xtest = bh[-t.idx,1:13],
ytest = bh[-t.idx,14], importance=TRUE, keep.forest=TRUE)
```

- ▶ `x`: the predictors
- ▶ `y`: This is the outcome variables
- ▶ `ntree`: This is the number of trees to build

- ▶ `xtest` : These are predictors in the validation partition
- ▶ `ytest` : These are outcome variables in the validation partition
- ▶ `importance` : This refers to whether or not to compute the importance scores of the predictor variables
- ▶ `keep.forest` : This refers to whether or not to keep the trees built in the resulting model; only if we keep the trees can we generate predictions based on the model

Step 5 prints the model. This shows the mean squared error on the training and the validation partitions, as well as the percentage of variability in the outcome variable that the model explains.

Step 6 uses the `importance` component of the model to print the computed importance level of each variable. For each tree generated, the method first generates the prediction. Then, for every variable (one at a time), it randomly permutes the values across the OOB cases and generates the predictions. The degradation in prediction with the variable permuted indicates how important the variable is. For each predictor variable, the importance table reports the average value of importance across all trees. Higher values indicate higher importance.

Step 7 plots the predictions for the training partition against the actual values.

Step 8 plots the OOB predictions against the actual values using the `predicted` component of the model—`mod$predicted`.

Step 9 uses `mod$test$predicted` to plot the performance of the model on the test cases against actuals.

### There's more...

We discuss a few prominent options in this section.

#### Controlling forest generation

You can use the following additional options to control how the algorithm builds the forest:

- ▶ `mtry` : This is the number of predictors to randomly sample at each split; the default is  $m/3$  where  $m$  is the number of predictors
- ▶ `nodesize` : This is the minimum size of terminal nodes; the default is 5, setting it higher causes smaller trees
- ▶ `maxnodes` : This is the maximum number of terminal nodes that a tree can have; if unspecified, the trees are grown to the maximum size possible, subject to `nodesize`

## See also...

- ▶ *Creating random data partitions in Chapter 2, What's in There? – Exploratory Data Analysis*
- ▶ *Using random forest models for classification in Chapter 3, Where Does It Belong? – Classification*

# Using neural networks for regression

The `nnet` package contains functionality to build neural network models for classification as well as prediction. In this recipe, we cover the steps to build a neural network regression model using `nnet`.

## Getting ready

If you do not already have the `nnet`, `caret`, and `devtools` packages installed, install them now. If you have not already downloaded the data files for this chapter, download them now and ensure that the `BostonHousing.csv` file is in your R working directory. We will build a model to predict MEDV based on all of the remaining variables.

## How to do it...

To use neural networks for regression, follow these steps:

1. Load the `nnet` and `caret` packages:

```
> library(nnet)
> library(caret)

> library(devtools)
```

2. Read the data:

```
> bh <- read.csv("BostonHousing.csv")
```

3. Partition the data:

```
> set.seed(1000)
> t.idx <- createDataPartition(bh$MEDV, p=0.7, list=FALSE)
```

4. Find the range of the response variable to be able to scale it to [0,1]:

```
> summary(bh$MEDV)
Min. 1st Qu. Median Mean 3rd Qu. Max.
5.00 17.02 21.20 22.53 25.00 50.00
```

## 5. Build the model:

```
> fit <- nnet(MEDV/50 ~ ., data=bh[t.idx,], size=6, decay = 0.1,
maxit = 1000, linout = TRUE)
```

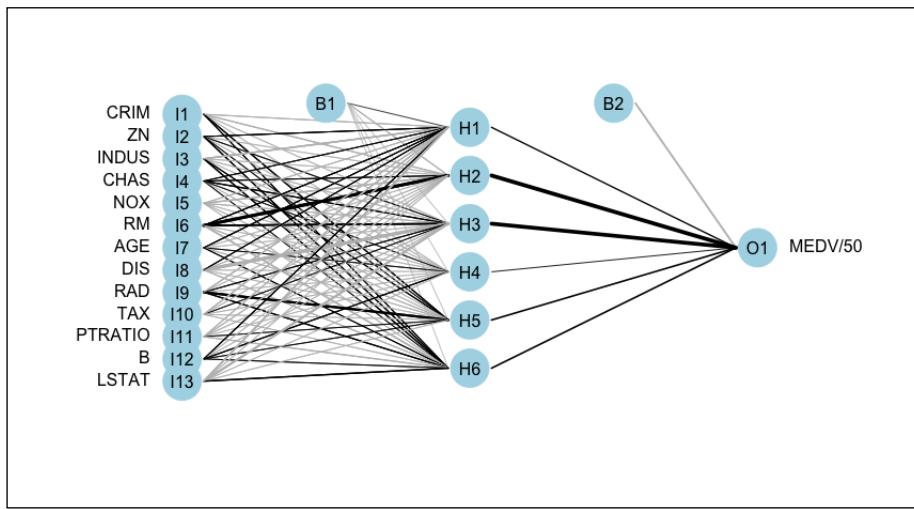
## 6. In preparation for plotting the network, get the code for the plotting function plot.nnet from fawda123's GitHub page. The following loads the function into R:

```
> source_url('https://gist.githubusercontent.com/fawda123/7471137/
raw/466c1474d0a505ff044412703516c34f1a4684a5/nnet_plot_update.r')
```

## 7. Plot the network:

```
> plot(fit, max.sp = TRUE)
```

The following plot is obtained as a result of the preceding commands:



## 8. Compute the RMS error on the training data (your results can differ):

```
> t.rmse = sqrt(mean((fit$fitted.values * 50 - bh[t.idx,
"MEDV"])^2))
> t.rmse
[1] 2.797945
```

## 9. Generate predictions on the validation partition and generate the RMS error (your results may differ):

```
> v.rmse <- sqrt(mean((predict(fit,bh[-t.idx,]*50 - bh[-t.idx,
"MEDV"])^2)))
> v.rmse
[1] 0.42959
```

## How it works...

Step 1 loads the necessary packages—`nnet` for neural network modeling and `caret` for data partitioning. We also load `devtools` because we will be sourcing code using a web URL for printing the network.

Step 2 reads the file.

Step 3 partitions the data. See recipe *Creating random data partitions* from Chapter 2, for more details. We have set the random seed to enable you to match your results with those that we have displayed.

Step 4 builds the neural net model using the `nnet` function of the `nnet` package:

```
> fit <- nnet(MEDV/50 ~ ., data=bh[t.idx,], size=6, decay = 0.1, maxit
= 1000, linout = TRUE)
```

We divide our response variable by 50 to scale it to the range [0,1]. We pass the following arguments:

- ▶ `size = 6`: This indicates the number of nodes in the hidden layer.
- ▶ `decay = 0.1`: This indicates the decay.
- ▶ `maxit = 1000`: Stops if the process does not converge in the `maxit` iterations. The default value for `maxit` is 100. Provide a value based on trial and error.
- ▶ `linout = TRUE`: This specifies that we want a linear output unit and not logistic.

Step 6 loads code for the printing function from an eternal `url` using the `source_url` function of the `devtools` package.

Step 7 then plots the network. The thickness of the lines indicates the strength of the corresponding weights. We use `max.sp = TRUE` to cause the plot to have maximum possible spacing between the nodes.

Step 8 uses the `fitted` component of the model to compute the RMS error on the training partition.

Step 9 uses the `predict` function on the validation partition to generate predictions to compute the RMS error on the validation partition.

## See also...

- ▶ *Creating random data partitions* in Chapter 2, *What's in There? – Exploratory Data Analysis*
- ▶ *Using neural networks for classification* in Chapter 3, *Where Does It Belong? – Classification*

## Performing k-fold cross-validation

The R implementation of some techniques, such as classification and regression trees, performs cross-validation out of the box to aid in model selection and to avoid overfitting. However, some others do not. When faced with several choices of machine learning methods for a particular problem, we can use the standard approach of partitioning the data into training and test sets and select based on the results. However, cross-validation gives a more thorough evaluation of a model's performance on hold-out data. Comparing the performance of methods using cross-validation can paint a truer picture of their relative performance.

### Getting ready

We illustrate the approach with the Boston Housing data, and thus you should download the code for this chapter and ensure that the `BostonHousing.csv` file is in your R working directory.

### How to do it...

In this recipe, we show you basic code to perform k-fold cross-validation for linear regression. You can adapt the same code structure for all other regression methods. Although some packages like `caret`, `DAAG` and `boot` provide cross-validation functionality out of the box, they cover only a few machine-learning techniques. You might find a generic framework to be useful and be able to adapt it to whatever machine-learning technique you might want to apply it to. To do this, follow these steps:

1. Read the data:

```
> bh <- read.csv("BostonHousing.csv")
```

2. Create the two functions shown as follows; we show line numbers for discussion:

```
1 rdacb.kfold.crossval.reg <- function(df, nfolds) {
2 fold <- sample(1:nfolds, nrow(df), replace = TRUE)
3 mean.sqr errs <- sapply(1:nfolds,
4 rdacb.kfold.cval.reg.ite
5 df, fold)
6 list("mean_sqr_errs"= mean.sqr errs,
7 "overall_mean_sqr_err" = mean(mean.sqr errs),
8 "std_dev_mean_sqr_err" = sd(mean.sqr errs))
9 }
```

```
6 rdacb.kfold.cval.reg.ite <- function(k, df, fold) {
7 trg.idx <- !fold %in% c(k)
8 test.idx <- fold %in% c(k)
```

```
9 mod <- lm(MEDV ~ ., data = df[trg.idx,])
10 pred <- predict(mod, df[test.idx,])
11 sqr errs <- (pred - df[test.idx, "MEDV"])^2
12 mean(sqr errs)
13 }
```

3. With the preceding two functions in place, you can run k-fold cross-validation with k=5 as follows:

```
> res <- rdacb.kfold.crossval.reg(bh, 5)
> # get the mean squared errors from each fold
> res$mean_sqr_errs
> # get the overall mean squared errors
> res$overall_mean_sqr_err
> # get the standard deviation of the mean squared errors
> res$std_dev_mean_sqr_err
```

## How it works...

Step 1 reads the data file.

In step 2, we define two functions to perform k-fold cross-validation. Rows 1-5 define the first function and rows 6-13 define the second function.

The first function `rdacb.kfold.crossval.reg` sets up the k-folds and uses the second one to build the model and compute the errors for each fold.

Line 2 creates the folds by randomly sampling from 1 to k. Thus, if a data frame has 1000 elements, this line will generate 1000 random integers from 1 to k. The idea is that if the  $i^{\text{th}}$  random number is, say, 3, then the  $i^{\text{th}}$  case of the data frame belongs to the third fold.

Line 3 invokes the second function to compute the errors for each fold.

Line 4 creates a list with the raw values of the mean squared errors for each partition, the overall mean across all the folds, and the standard deviation of the mean squared errors.

The second function computes the error for a particular partition.

Lines 7 and 8 set up the training and test data. The fold number is passed in as the argument `k` and line 7 treats all data rows belonging to folds other than `k` as the training data. Line 8 sets up the data rows belonging to the `k`th fold as the test data.

Line 9 builds the linear regression model with the training data alone.

Line 10 generates the predictions on the test data.

Line 11 computes the squared errors.

Line 12 returns the mean of the squared errors.

## See also...

- ▶ *Performing leave-one-out-cross-validation to limit overfitting* in this chapter.

# Performing leave-one-out-cross-validation to limit overfitting

We provide the framework of the code to perform leave-one-out-cross-validation for linear regression. You should be able to easily adapt this code for any other regression technique. The rationale and explanation presented under the previous recipe *Performing k-fold cross-validation* apply to this one as well.

## How to do it...

To perform **leave-one-out-cross-validation (LOOCV)** to limit overfitting, follow the steps below:

1. Read the data:

```
> bh <- read.csv("BostonHousing.csv")
```

2. Create the two functions shown as follows; we show line numbers for discussion:

```
1 rdacb.loocv.reg <- function(df) {
2 mean.sqr errs <- sapply(1:nrow(df),
3 rdacb.loocv.reg.iter, df)
3 list("mean_sqr_errs"= mean.sqr errs,
4 "overall_mean_sqr_err" = mean(mean.sqr errs),
5 "std_dev_mean_sqr_err" = sd(mean.sqr errs))
4 }

5 rdacb.loocv.reg.iter <- function(k, df) {
6 mod <- lm(MEDV ~ ., data = df[-k,])
7 pred <- predict(mod, df[k,])
8 sqr.err <- (pred - df[k, "MEDV"])^2
9 }
```

3. With the preceding two functions in place, you can run leave-one-out-cross-validation as follows (this runs 506 linear regression models and will take some time):

```
> res <- rdacb.loocv.reg(bh)
> # get the raw mean squared errors for each case
> res$mean_sqr_errs
> # get the overall mean squared error
> res$overall_mean_sqr_err
> # get the standard deviation of the mean squared errors
> res$std_dev_mean_sqr_err
```

## How it works...

Step 1 reads the data.

Step 2 creates two functions for performing leave-one-out-cross-validation. Lines 1 to 4 define the first function `rdacb.loocv.reg` and lines 5 to 9 define the second one, `rdacb.loocv.reg.iter`:

- ▶ Line 2 of the first function `rdacb.loocv.reg` repeatedly calls the second function `rdacb.loocv.reg.iter` to build the regression model leaving one case out and compute the squared error
- ▶ Line 3 creates a list with the output elements
- ▶ Line 6 in the second function `rdacb.loocv.reg.iter` builds the regression model on the data frame leaving out one case
- ▶ Line 7 generates the prediction for the case that was left out
- ▶ Line 8 computes the squared error

Step 3 uses the preceding functions to perform leave-one-out-cross-validation and displays the results.

## See also...

- ▶ *Performing k-fold cross-validation* in this chapter.

# 5

## Can You Simplify That? – Data Reduction Techniques

In this chapter, we will cover:

- ▶ Performing cluster analysis using K-means clustering
- ▶ Performing cluster analysis using hierarchical clustering
- ▶ Reducing dimensionality with principal component analysis

### Introduction

When confronted with large datasets, either in terms of the number of cases or the number of variables, or both, analysts often seek to reduce the complexity. They can use **cluster analysis** to condense the number of cases to a manageable number of representative points, or they may use **principal component analysis (PCA)** to identify a smaller set of variables or dimensions that capture the information content of most of the larger set of original variables. This chapter will cover R recipes for cluster analysis and PCA.

## Performing cluster analysis using K-means clustering

The standard R package `stats` provides the function for K-means clustering. We also use the `cluster` package to plot the results of our cluster analysis.

### Getting ready

If you have not already downloaded the files for this chapter, do it now and ensure that the `auto-mpg.csv` file is in your R working directory. Also, ensure that you have installed the `cluster` package.

### How to do it...

To perform cluster analysis using K-means clustering, follow these steps:

1. Read the data:

```
> auto <- read.csv("auto-mpg.csv")
```

2. Define a convenience function to standardize the relevant variables and append the resulting variables to the original data:

```
rdacb.scale.many <- function (dat, column_nos) {
 nms <- names(dat)
 for (col in column_nos) {
 name <- paste0(nms[col], "_z")
 dat[name] <- scale(dat[, col])
 }
 cat(paste("Scaled", length(column_nos), "variable(s)\n"))
 dat
}
```

3. Use the preceding convenience function to standardize the variables of interest. We will ignore the variables `No`, `model_year`, and `car_name`:

```
> auto <- rdacb.scale.many(auto, 2:7)
> # See the variables now in auto
> names(auto)
[1] "No" "mpg"
[3] "cylinders" "displacement"
[5] "horsepower" "weight"
[7] "acceleration" "model_year"
[9] "car_name" "mpg_z"
[11] "cylinders_z" "displacement_z"
```

```
[13] "horsepower_z" "weight_z"
[15] "acceleration_z"
```

4. Perform K-means clustering for a given value of K. Let's use  $K=5$ . We show how you can settle on a good value for K in the *There's more...* section of this recipe. Due to the random choice of K starting points, your results may differ:

```
> set.seed(1020)
> fit <- kmeans(auto[, 10:15], 5)
> # Examine the fit object - produces a lot of output
> # Your results could differ slightly
> fit
K-means clustering with 5 clusters of sizes 36, 96, 62, 117, 87

Cluster means:
 mpg_z cylinders_z displacement_z
1 -0.4141251 0.2388808 0.2772370
2 -1.1538840 1.4963079 1.4943315
3 -0.4317115 0.3679422 0.1875709
4 0.3259756 -0.8753429 -0.7189046
5 1.3138889 -0.8349721 -0.9305048

 horsepower_z weight_z acceleration_z
1 -0.28320032 0.5386915 1.29988821
2 1.50450532 1.3943873 -1.06420891
3 0.03201748 0.1614095 -0.12178037
4 -0.43500729 -0.6304741 -0.06498252
5 -0.98076471 -1.0286895 0.81059100

Clustering vector:
[1] 4 4 5 4 3 4 2 4 1 2 2 1 2 4 2 4 4 2 2 4 1 4
[23] 4 4 4 2 5 2 3 5 4 2 4 5 4 2 1 5 4 5 3 2 3 2
[45] 4 4 2 1 2 3 4 3 4 2 3 4 4 4 2 2 5 4 2 2 2 5
[67] 3 2 1 2 5 1 3 5 3 2 1 4 2 2 5 5 5 4 3 2 5 2
[89] 5 4 2 2 4 5 5 5 4 2 4 2 5 5 4 2 5 4 5 2 4 3
[111] 5 2 2 4 3 5 3 5 4 4 4 2 1 3 2 2 5 4 2 1 4 4
[133] 2 2 2 5 3 4 2 2 2 5 2 5 3 5 3 5 5 2 5 4
[155] 5 4 3 2 4 4 3 5 2 2 4 2 2 5 4 2 5 2 5 3 2 5
[177] 1 3 4 4 4 3 3 3 1 2 2 4 4 2 4 4 2 4 4 4 4 2
[199] 4 5 3 3 1 5 5 2 3 4 3 4 4 4 2 5 3 5 4 4 3 4
[221] 4 2 3 2 3 2 4 4 5 4 4 4 2 5 2 4 4 1 1 2 4 4
[243] 4 4 4 5 3 2 5 4 4 2 2 2 4 4 5 3 5 1 5 5 5 4
[265] 5 4 2 5 5 3 4 5 1 3 4 3 1 4 5 4 2 3 2 1 2 4
[287] 1 4 4 1 2 1 4 3 5 4 4 5 4 5 4 5 4 5 3 2 4 3
[309] 4 3 2 3 5 4 1 3 5 5 5 2 2 4 1 3 3 4 5 1 4 4
```

## Can You Simplify That? – Data Reduction Techniques

---

```
[331] 5 5 3 1 5 3 3 4 1 2 1 5 3 2 2 2 4 2 4 5 1 2 5
[353] 3 1 5 1 2 2 4 5 5 2 3 3 4 3 1 2 5 3 4 4 1 3
[375] 5 4 3 2 4 1 3 1 5 4 1 5 2 5 2 2 5 4 2 3 5 5
[397] 5 3
```

Within cluster sum of squares by cluster:

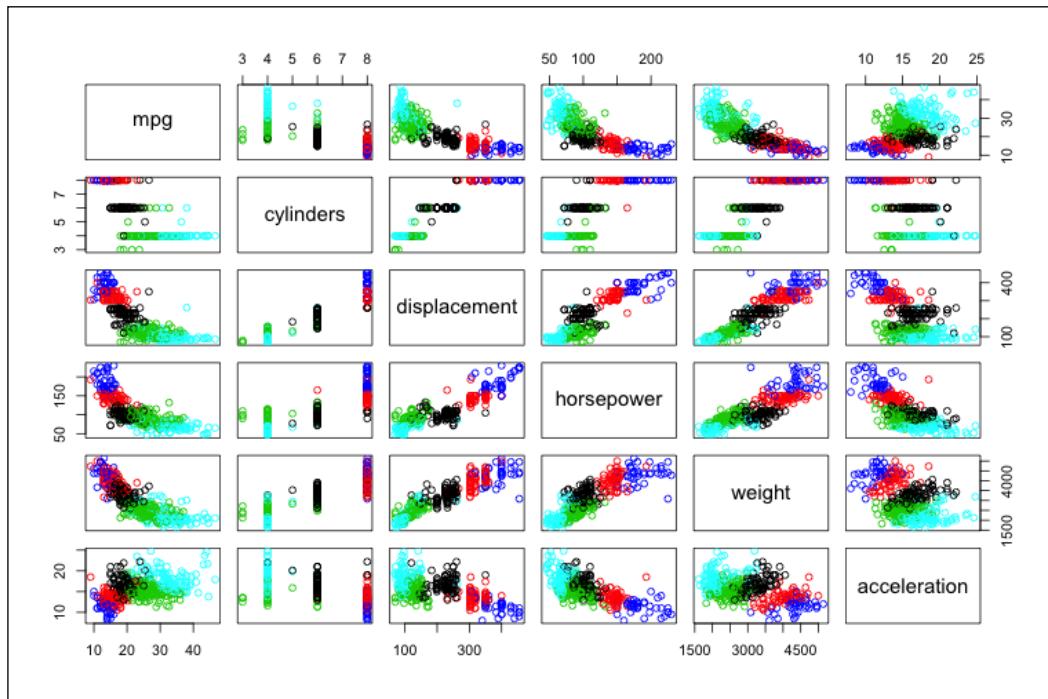
```
[1] 53.49325 134.03814 51.86729 96.53647
[5] 115.59778
(between_SS / total_SS = 81.0 %)
```

Available components:

```
[1] "cluster" "centers" "totss"
[4] "withinss" "tot.withinss" "betweenss"
[7] "size" "iter" "ifault"
```

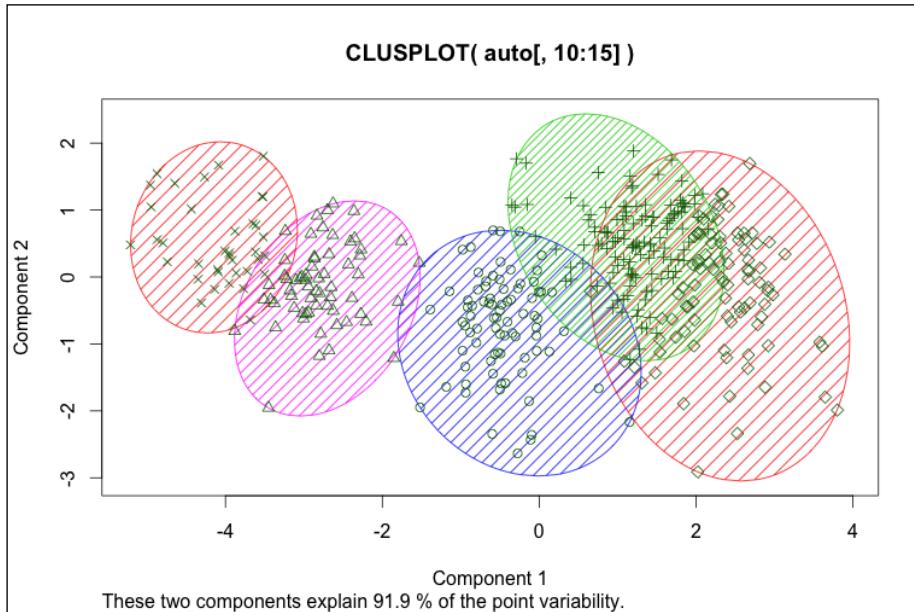
5. We performed cluster analysis on six dimensions and thus cannot visualize the complete analysis. However, we can creatively use a pairwise plot to get an idea of the clustering and visualize the results:

```
> pairs(auto[,2:7], col=c(1:5)[fit$cluster])
```



6. The `clusplot` function from the `cluster` package can help us visualize the clustering based on the first two principal components by generating a bivariate cluster plot using the following command:

```
> library(cluster)
> clusplot(auto[,10:15], fit$cluster, color = TRUE,
 shade = TRUE, labels=0, lines=0)
```



## How it works...

In step 1 the data is loaded.

In step 2 a convenience function is defined to standardize several variables at once. Although the `scale` function does this, the names it assigns to the standardized variables mimic the original ones and hence can cause confusion. This convenience function creates more meaningful names by appending `_z` to the original variable names.

In step 3 the convenience function is used to scale only the variables of interest—we leave out `No`, `model_year`, and `car_name`.

In step 4 the K-means clustering algorithm is run by invoking the `kmeans` function and then printing the resulting object. We used `K=5` as an illustration. The next section, *There's more...*, shows how we can settle on a suitable value for  $K$ . While invoking the `kmeans` function, we have the option of selecting the specific K-means clustering algorithm by passing a value for the `algorithm` argument. If left out, the function uses the Hartigan Wong algorithm by default. Other options are `Lloyd`, `Forgy`, and `MacQueen`.

From the output, we can see that the resulting model contains information about the centers of the clusters, information about which cluster each case of the data falls under, the sum of squares within each cluster, and finally, the proportion of total variability that the K clusters retain. We see that the 5-cluster solution retains 81 percent of the variability.

The output also shows the components of the fitted model, from which we can extract relevant information. The following table summarizes the information:

| Command                        | Function                                                                                                                                                        |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>fit\$cluster</code>      | The clustering vector, specifying the cluster to which each case belongs.                                                                                       |
| <code>fit\$centers</code>      | The center of each cluster.                                                                                                                                     |
| <code>fit\$totss</code>        | Total sum of squares of the variables used. We used standardized values and hence this number represents the total sum of squares of these standardized values. |
| <code>fit\$withinss</code>     | Within sum of squares for each cluster.                                                                                                                         |
| <code>fit\$tot.withinss</code> | Total of the withinss values.                                                                                                                                   |
| <code>fit\$betweenss</code>    | Total sum of squares if we represent each case by just the center of its cluster.                                                                               |
| <code>fit\$size</code>         | Number of cases in each cluster.                                                                                                                                |
| <code>fit\$iter</code>         | Number of iterations used.                                                                                                                                      |
| <code>fit\$ifault</code>       | An indicator of a possible algorithm problem—for experts.                                                                                                       |

In step 5 a scatterplot matrix is generated and the points are colored based on the clustering vector from the results of K-means clustering.

In step 6 use the `clusplot` function from the `cluster` package to generate a bivariate plot of the first two principal components (see *Reducing dimensionality with principal components analysis* in this chapter). From the bottom of the plot, we see that the first two principal components explain nearly 92 percent of the overall variability and hence the plot can be treated as a good representation of the overall clustering along the six original dimensions.

### There's more...

Once the value of K is known, K-means clustering can be performed. We now provide you with a recipe to ease the process of choosing a suitable value for K.

## Use a convenience function to choose a value for K

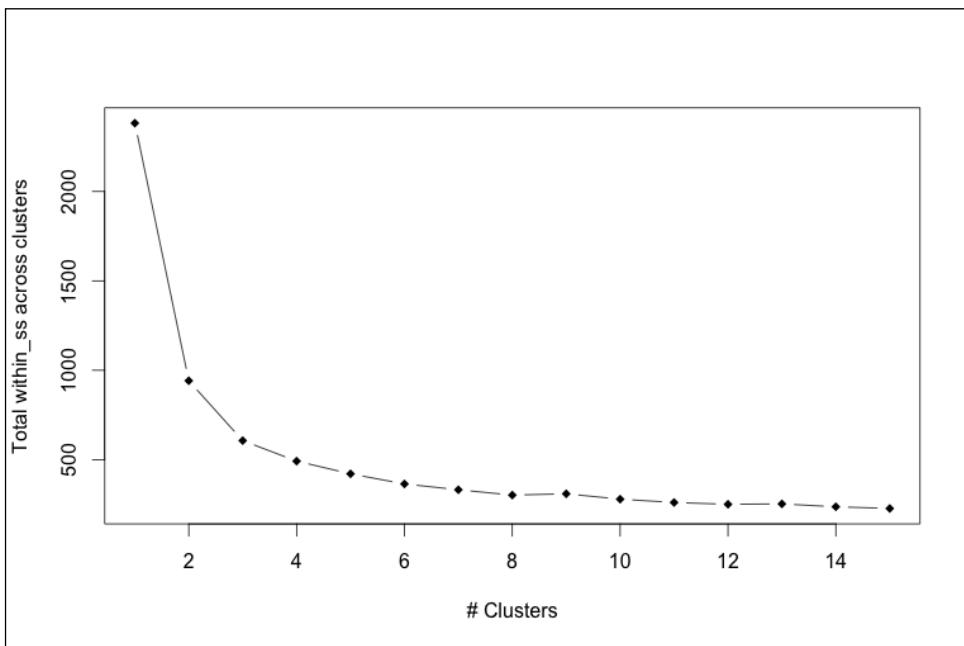
Using what we covered, you can try out various values of K and choose a suitable one. To avoid the repetitive tasks involved in doing this, we suggest the following convenience function:

```
rdacb.kmeans.plot <- function (data, num_clust = 15, seed = 9876) {
 set.seed(seed)
 ss <- numeric(num_clust)
 ss[1] <- (nrow(data) - 1) * sum(apply(data, 2, var))
 for (i in 2:num_clust) {
 ss[i] <- sum(kmeans(data, centers = i)$withinss)
 }
 plot(1:num_clust, ss, type = "b", pch = 18, xlab = "# Clusters",
 ylab = "Total within_ss across clusters")
}
```

Since the method generates a plot and helps you identify the place where the gain in performance tapers off—the elbow in the graph—it is sometimes referred to as the "elbow" technique.

The preceding function generates and plots the total `withinss` across all clusters for values of K between 1 and 15. We can also supply an upper limit for K through the `num_clust` argument. Armed with the function, we can standardize the variables of interest as in the main recipe and then run:

```
> rdacb.kmeans.plot(auto[,10:15])
```



From the resulting plot, we can identify the value of K where the drop in the total withinss tapers off—the elbow in the graph. We may choose  $K=4$  or  $K=5$  in this situation. If we really wanted very few clusters,  $K=3$  may also be a good choice. We can always get a very low value for withinss by increasing the number of clusters. However, we want to strike a balance between the value of withinss and the number of clusters.

## See also...

- ▶ *Performing cluster analysis using hierarchical clustering* in this chapter

# Performing cluster analysis using hierarchical clustering

The `hclust` function in the package `stats` helps us perform hierarchical clustering.

## Getting ready

If you have not already downloaded the data files for this chapter, do it now and ensure that the `auto-mpg.csv` file is in R's working directory.

We will hierarchically cluster the data based on the variables `mpg`, `cylinders`, `displacement`, `horsepower`, `weight`, and `acceleration`.

## How to do it...

To perform cluster analysis using hierarchical clustering, follow these steps:

1. Read the data:

```
> auto <- read.csv("auto-mpg.csv")
```

2. Define a convenience function to standardize the relevant variables and append the resulting variables to the original data:

```
rdacb.scale.many <- function (dat, column_nos) {
 nms <- names(dat)
 for (col in column_nos) {
 name <- paste0(nms[col], "_z")
 dat[name] <- scale(dat[, col])
 }
 cat(paste("Scaled", length(column_nos), "variable(s)\n"))
 dat
}
```

3. Use the preceding convenience function to standardize the variables of interest.

We will ignore the variables `No`, `model_year`, and `car_name`:

```
> auto <- rdacb.scale.many(auto, 2:7)
> # See the variables now in auto
> names(auto)
[1] "No" "mpg"
[3] "cylinders" "displacement"
[5] "horsepower" "weight"
[7] "acceleration" "model_year"
[9] "car_name" "mpg_z"
[11] "cylinders_z" "displacement_z"
[13] "horsepower_z" "weight_z"
[15] "acceleration_z"
```

4. Compute the distance matrix to provide as input to the `hclust` function in the next step. We use Euclidean distances here:

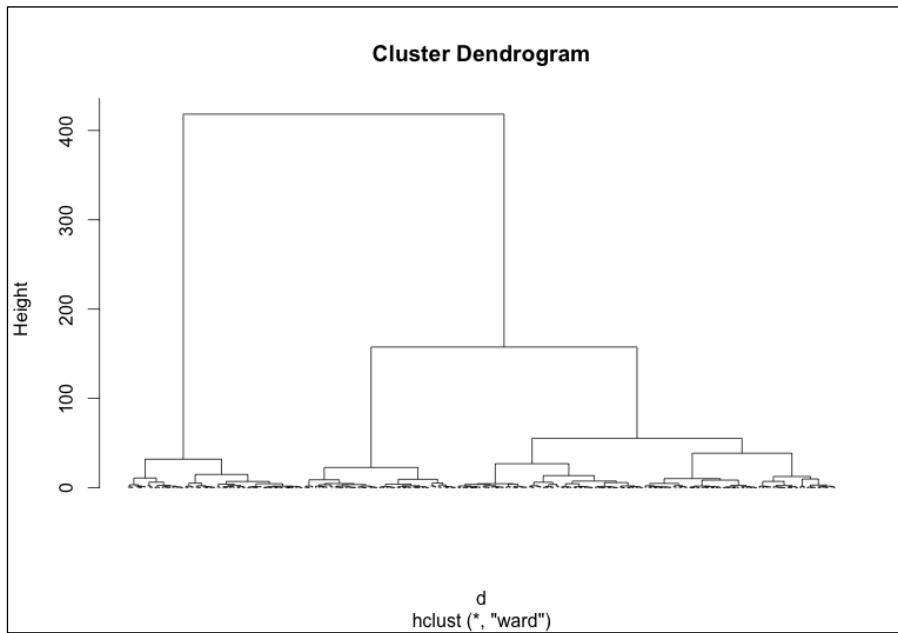
```
> dis <- dist(auto[, 10:15], method = "euclidean")
```

5. Use the `hclust` function to perform hierarchical clustering:

```
> fit <- hclust(dis, method = "ward")
```

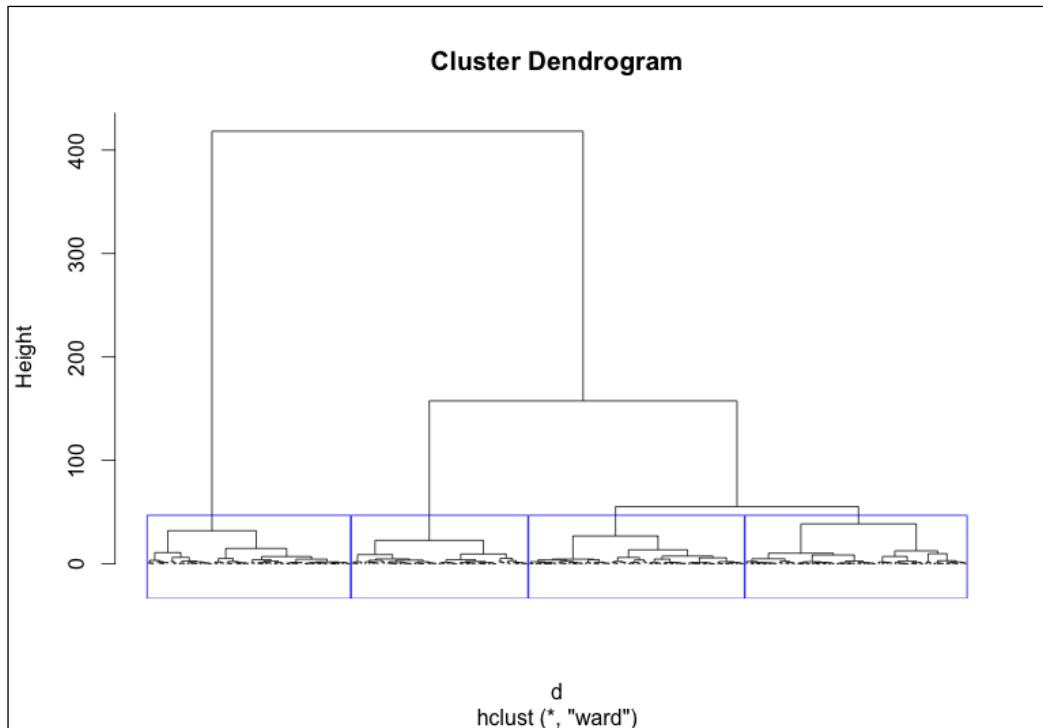
6. Plot the dendrogram representing the result of clustering. The result looks very crowded at the bottom because the function plots every single case in the data:

```
> plot(fit, labels = FALSE, hang = 0)
```



7. Select a value for K and place a rectangle around each of the K clusters. We use k=4 in the following code:

```
> rect.hclust(fit, k=4, border="blue")
```



8. Get the cluster to which each case belongs:

```
> cluster <- cutree(fit, k=4)
> cluster
[1] 1 1 2 1 3 1 4 2 3 4 4 4 1 4 1 4 1 2 4 4 1 3 1 1 1 1
[26] 4 2 4 3 2 1 4 1 2 1 4 3 2 2 2 3 4 3 4 2 1 4 3 4 3
[51] 1 3 1 4 4 2 2 1 4 4 2 1 4 4 4 2 4 4 3 4 2 3 3 1 3
[76] 4 3 1 4 4 2 2 2 1 3 4 2 4 2 1 4 4 2 2 2 1 2 4 1 4
[101] 2 1 1 4 2 1 2 4 1 3 2 4 4 1 3 2 3 2 1 1 1 4 3 3 4
[126] 4 1 2 4 1 1 2 4 4 4 4 1 3 1 4 4 4 2 4 2 4 2 3 2 3 2
[151] 2 4 2 1 2 1 3 4 1 1 3 1 4 4 4 2 4 4 2 1 4 1 4 2 3 4
[176] 2 3 3 2 1 1 3 3 3 1 4 4 1 1 4 1 1 4 1 1 1 4 1 2
[201] 3 3 3 1 2 4 3 1 3 2 1 1 4 2 3 2 1 2 3 1 1 4 3 4 3
[226] 4 2 1 2 2 1 1 4 2 4 1 2 3 3 4 1 1 1 2 1 2 3 4 2 1
[251] 1 4 4 4 1 2 2 3 2 1 2 1 2 2 2 2 4 2 2 3 2 2 3 3 1
[276] 3 3 2 1 1 4 3 4 3 4 1 3 1 1 4 3 1 3 2 2 1 2 1 1
```

```
[301] 2 2 2 2 3 4 1 3 1 3 4 3 1 2 3 1 2 2 2 4 4 1 1 3 3
[326] 2 2 3 2 1 2 2 3 3 2 3 3 2 1 4 3 2 3 4 4 1 4 2 2 3
[351] 4 2 3 3 2 3 4 4 1 2 2 4 3 3 1 3 1 4 2 3 1 2 3 3 2
[376] 2 3 4 1 3 3 3 2 1 3 2 4 2 4 4 2 1 4 3 2 2 2 3
```

## How it works...

In step 1 the data is read and in step 2 we define the convenience function for scaling a set of variables in a data frame.

In step 3 the convenience function is used to scale only the variables of interest. We leave out the `No`, `model_year`, and `car_name` variables.

In step 4 the distance matrix is created based on the standardized values of the relevant variables. We have computed Euclidean distances; other possibilities are: `maximum`, `manhattan`, `canberra`, `binary`, and `minkowski`.

In step 5 the distance matrix is passed to the `hclust` function to create the clustering model. We specified `method = "ward"` to use **Ward's** method, which tries to get compact spherical clusters. The `hclust` function also supports `single`, `complete`, `average`, `mcquitty`, `median`, and `centroid`.

In step 6 the resulting dendrogram is plotted. We specified `labels=FALSE` because we have too many cases and printing them will only add clutter. With a smaller dataset, using `labels = TRUE` will make sense. The `hang` argument controls the distance from the bottom of the dendrogram to the labels. Since we are not using labels, we specified `hang = 0` to prevent numerous vertical lines below the dendrogram.

The dendrogram shows all the cases at the bottom (too numerous to distinguish in our plot) and shows the step-by-step agglomeration of the clusters. The dendrogram is organized in such a way that we can obtain a desired set of clusters, say  $k$ , by drawing a horizontal line in such a way that it cuts across exactly  $k$  vertical lines on the dendrogram.

Step 7 show how to use the `rect.hclust` function to demarcate the cases comprising the various clusters for a selected value of  $k$ .

Step 8 shows how we can use the `cutree` function to identify, for a specific  $k$ , which cluster each case of our data belongs to.

## See also...

- ▶ *Performing cluster analysis using K-means clustering* in this chapter

## Reducing dimensionality with principal component analysis

The `stats` package offers the `prcomp` function to perform PCA. This recipe shows you how to perform PCA using these capabilities.

### Getting ready

If you have not already done so, download the data files for this chapter and ensure that the `BostonHousing.csv` file is in your R working directory. We want to predict `MEDV` based on the remaining 13 predictor variables. We will use PCA to reduce the dimensionality.

### How to do it...

To reduce dimensionality with PCA, follow the steps:

1. Read the data:

```
> bh <- read.csv("BostonHousing.csv")
```

2. View the correlation matrix to check whether some variables are highly correlated and whether PCA has the potential to yield some dimensionality reduction. Since we are interested in reducing the dimensionality of the predictor variables, we leave out the outcome variable `MEDV`:

```
> round(cor(bh[,-14]),2)
```

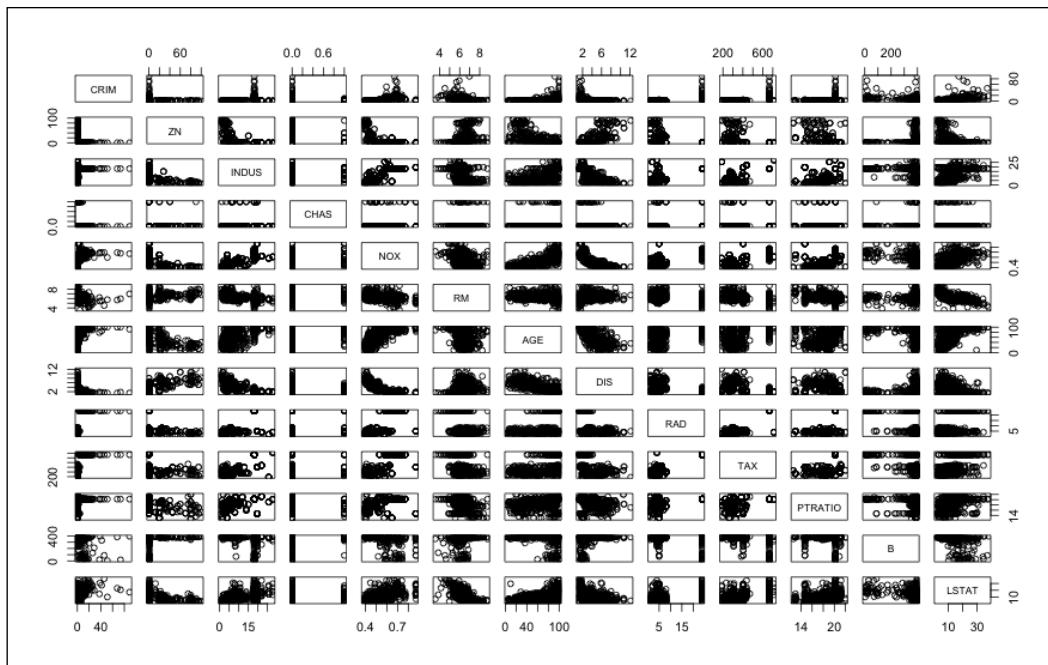
|         | CRIM  | ZN    | INDUS | CHAS    | NOX   | RM    | AGE   |
|---------|-------|-------|-------|---------|-------|-------|-------|
| CRIM    | 1.00  | -0.20 | 0.41  | -0.06   | 0.42  | -0.22 | 0.35  |
| ZN      | -0.20 | 1.00  | -0.53 | -0.04   | -0.52 | 0.31  | -0.57 |
| INDUS   | 0.41  | -0.53 | 1.00  | 0.06    | 0.76  | -0.39 | 0.64  |
| CHAS    | -0.06 | -0.04 | 0.06  | 1.00    | 0.09  | 0.09  | 0.09  |
| NOX     | 0.42  | -0.52 | 0.76  | 0.09    | 1.00  | -0.30 | 0.73  |
| RM      | -0.22 | 0.31  | -0.39 | 0.09    | -0.30 | 1.00  | -0.24 |
| AGE     | 0.35  | -0.57 | 0.64  | 0.09    | 0.73  | -0.24 | 1.00  |
| DIS     | -0.38 | 0.66  | -0.71 | -0.10   | -0.77 | 0.21  | -0.75 |
| RAD     | 0.63  | -0.31 | 0.60  | -0.01   | 0.61  | -0.21 | 0.46  |
| TAX     | 0.58  | -0.31 | 0.72  | -0.04   | 0.67  | -0.29 | 0.51  |
| PTRATIO | 0.29  | -0.39 | 0.38  | -0.12   | 0.19  | -0.36 | 0.26  |
| B       | -0.39 | 0.18  | -0.36 | 0.05    | -0.38 | 0.13  | -0.27 |
| LSTAT   | 0.46  | -0.41 | 0.60  | -0.05   | 0.59  | -0.61 | 0.60  |
|         | DIS   | RAD   | TAX   | PTRATIO | B     | LSTAT |       |
| CRIM    | -0.38 | 0.63  | 0.58  | 0.29    | -0.39 | 0.46  |       |
| ZN      | 0.66  | -0.31 | -0.31 | -0.39   | 0.18  | -0.41 |       |
| INDUS   | -0.71 | 0.60  | 0.72  | 0.38    | -0.36 | 0.60  |       |

|         |       |       |       |       |       |       |
|---------|-------|-------|-------|-------|-------|-------|
| CHAS    | -0.10 | -0.01 | -0.04 | -0.12 | 0.05  | -0.05 |
| NOX     | -0.77 | 0.61  | 0.67  | 0.19  | -0.38 | 0.59  |
| RM      | 0.21  | -0.21 | -0.29 | -0.36 | 0.13  | -0.61 |
| AGE     | -0.75 | 0.46  | 0.51  | 0.26  | -0.27 | 0.60  |
| DIS     | 1.00  | -0.49 | -0.53 | -0.23 | 0.29  | -0.50 |
| RAD     | -0.49 | 1.00  | 0.91  | 0.46  | -0.44 | 0.49  |
| TAX     | -0.53 | 0.91  | 1.00  | 0.46  | -0.44 | 0.54  |
| PTRATIO | -0.23 | 0.46  | 0.46  | 1.00  | -0.18 | 0.37  |
| B       | 0.29  | -0.44 | -0.44 | -0.18 | 1.00  | -0.37 |
| LSTAT   | -0.50 | 0.49  | 0.54  | 0.37  | -0.37 | 1.00  |

Ignoring the main diagonal, we see several correlations above 0.5, and a PCA can help to reduce the dimensionality.

3. We can perform the preceding step visually as well by plotting the scatterplot matrix:

```
> plot(bh[, -14])
```



4. Build the PCA model:

```
> bh.pca <- prcomp(bh[, -14], scale = TRUE)
```

5. Examine the rotations for the principal components generated:

```
> print(bh.pca)
```

Standard deviations:

```
[1] 2.4752472 1.1971947 1.1147272 0.9260535 0.9136826
[6] 0.8108065 0.7316803 0.6293626 0.5262541 0.4692950
[11] 0.4312938 0.4114644 0.2520104
```

Rotation:

|         | PC1          | PC2          | PC3          | PC4          |
|---------|--------------|--------------|--------------|--------------|
| CRIM    | 0.250951397  | -0.31525237  | 0.24656649   | -0.06177071  |
| ZN      | -0.256314541 | -0.32331290  | 0.29585782   | -0.12871159  |
| INDUS   | 0.346672065  | 0.11249291   | -0.01594592  | -0.01714571  |
| CHAS    | 0.005042434  | 0.45482914   | 0.28978082   | -0.81594136  |
| NOX     | 0.342852313  | 0.21911553   | 0.12096411   | 0.12822614   |
| RM      | -0.189242570 | 0.14933154   | 0.59396117   | 0.28059184   |
| AGE     | 0.313670596  | 0.31197778   | -0.01767481  | 0.17520603   |
| DIS     | -0.321543866 | -0.34907000  | -0.04973627  | -0.21543585  |
| RAD     | 0.319792768  | -0.27152094  | 0.28725483   | -0.13234996  |
| TAX     | 0.338469147  | -0.23945365  | 0.22074447   | -0.10333509  |
| PTRATIO | 0.204942258  | -0.30589695  | -0.32344627  | -0.28262198  |
| B       | -0.202972612 | 0.23855944   | -0.30014590  | -0.16849850  |
| LSTAT   | 0.309759840  | -0.07432203  | -0.26700025  | -0.06941441  |
|         | PC5          | PC6          | PC7          | PC8          |
| CRIM    | 0.082156919  | -0.21965961  | 0.777607207  | -0.153350477 |
| ZN      | 0.320616987  | -0.32338810  | -0.274996280 | 0.402680309  |
| INDUS   | -0.007811194 | -0.07613790  | -0.339576454 | -0.173931716 |
| CHAS    | 0.086530945  | 0.16749014   | 0.074136208  | 0.024662148  |
| NOX     | 0.136853557  | -0.15298267  | -0.199634840 | -0.080120560 |
| RM      | -0.423447195 | 0.05926707   | 0.063939924  | 0.326752259  |
| AGE     | 0.016690847  | -0.07170914  | 0.116010713  | 0.600822917  |
| DIS     | 0.098592247  | 0.02343872   | -0.103900440 | 0.121811982  |
| RAD     | -0.204131621 | -0.14319401  | -0.137942546 | -0.080358311 |
| TAX     | -0.130460565 | -0.19293428  | -0.314886835 | -0.082774347 |
| PTRATIO | -0.584002232 | 0.27315330   | 0.002323869  | 0.317884202  |
| B       | -0.345606947 | -0.80345454  | 0.070294759  | 0.004922915  |
| LSTAT   | 0.394561129  | -0.05321583  | 0.087011169  | 0.424352926  |
|         | PC9          | PC10         | PC11         | PC12         |
| CRIM    | 0.26039028   | -0.019369130 | -0.10964435  | -0.086761070 |
| ZN      | 0.35813749   | -0.267527234 | 0.26275629   | 0.071425278  |
| INDUS   | 0.64441615   | 0.363532262  | -0.30316943  | 0.113199629  |
| CHAS    | -0.01372777  | 0.006181836  | 0.01392667   | 0.003982683  |
| NOX     | -0.01852201  | -0.231056455 | 0.11131888   | -0.804322567 |

|         |              |              |             |              |
|---------|--------------|--------------|-------------|--------------|
| RM      | 0.04789804   | 0.431420193  | 0.05316154  | -0.152872864 |
| AGE     | -0.06756218  | -0.362778957 | -0.45915939 | 0.211936074  |
| DIS     | -0.15329124  | 0.171213138  | -0.69569257 | -0.390941129 |
| RAD     | -0.47089067  | -0.021909452 | 0.03654388  | 0.107025890  |
| TAX     | -0.17656339  | 0.035168348  | -0.10483575 | 0.215191126  |
| PTRATIO | 0.25442836   | -0.153430488 | 0.17450534  | -0.209598826 |
| B       | -0.04489802  | 0.096515117  | 0.01927490  | -0.041723158 |
| LSTAT   | -0.19522139  | 0.600711409  | 0.27138243  | -0.055225960 |
|         | PC13         |              |             |              |
| CRIM    | 0.045952304  |              |             |              |
| ZN      | -0.080918973 |              |             |              |
| INDUS   | -0.251076540 |              |             |              |
| CHAS    | 0.035921715  |              |             |              |
| NOX     | 0.043630446  |              |             |              |
| RM      | 0.045567096  |              |             |              |
| AGE     | -0.038550683 |              |             |              |
| DIS     | -0.018298538 |              |             |              |
| RAD     | -0.633489720 |              |             |              |
| TAX     | 0.720233448  |              |             |              |
| PTRATIO | 0.023398052  |              |             |              |
| B       | -0.004463073 |              |             |              |
| LSTAT   | 0.024431677  |              |             |              |

6. Examine the importance of the principal components:

```
> summary(bh.pca)
```

Importance of components:

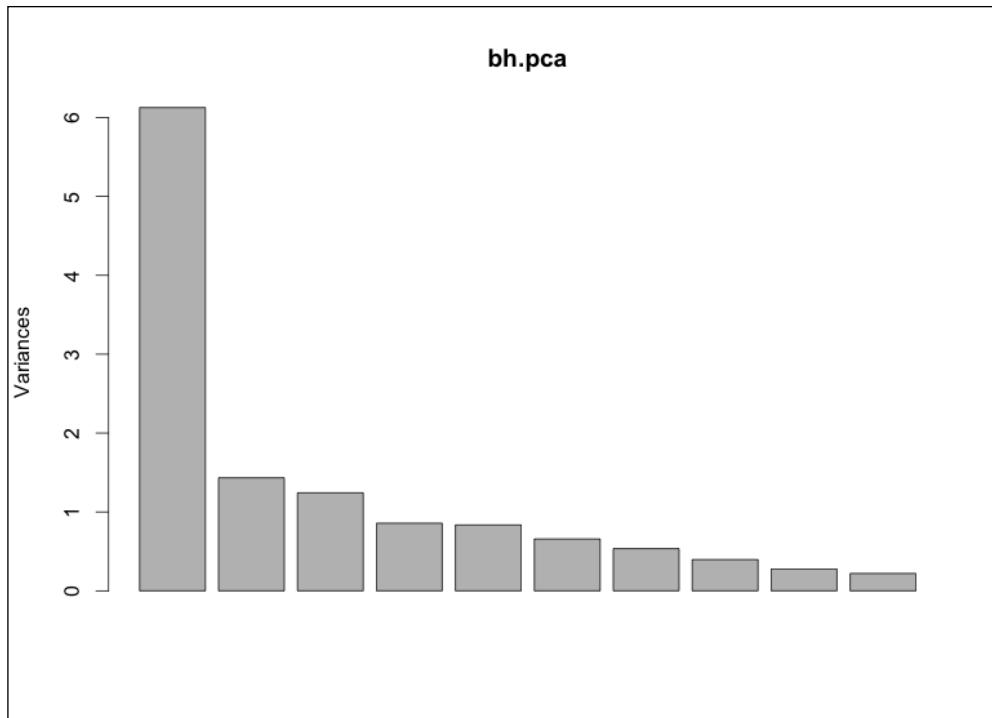
|                        | PC1     | PC2     | PC3     | PC4     |
|------------------------|---------|---------|---------|---------|
| Standard deviation     | 2.4752  | 1.1972  | 1.11473 | 0.92605 |
| Proportion of Variance | 0.4713  | 0.1103  | 0.09559 | 0.06597 |
| Cumulative Proportion  | 0.4713  | 0.5816  | 0.67713 | 0.74310 |
|                        | PC5     | PC6     | PC7     | PC8     |
| Standard deviation     | 0.91368 | 0.81081 | 0.73168 | 0.62936 |
| Proportion of Variance | 0.06422 | 0.05057 | 0.04118 | 0.03047 |
| Cumulative Proportion  | 0.80732 | 0.85789 | 0.89907 | 0.92954 |
|                        | PC9     | PC10    | PC11    | PC12    |
| Standard deviation     | 0.5263  | 0.46930 | 0.43129 | 0.41146 |
| Proportion of Variance | 0.0213  | 0.01694 | 0.01431 | 0.01302 |
| Cumulative Proportion  | 0.9508  | 0.96778 | 0.98209 | 0.99511 |
|                        | PC13    |         |         |         |
| Standard deviation     | 0.25201 |         |         |         |
| Proportion of Variance | 0.00489 |         |         |         |
| Cumulative Proportion  | 1.00000 |         |         |         |

Note that from the reported cumulative proportions, the first seven principal components account for almost 90% of the variance.

7. Visualize the importance of the components through a scree plot or a barplot:

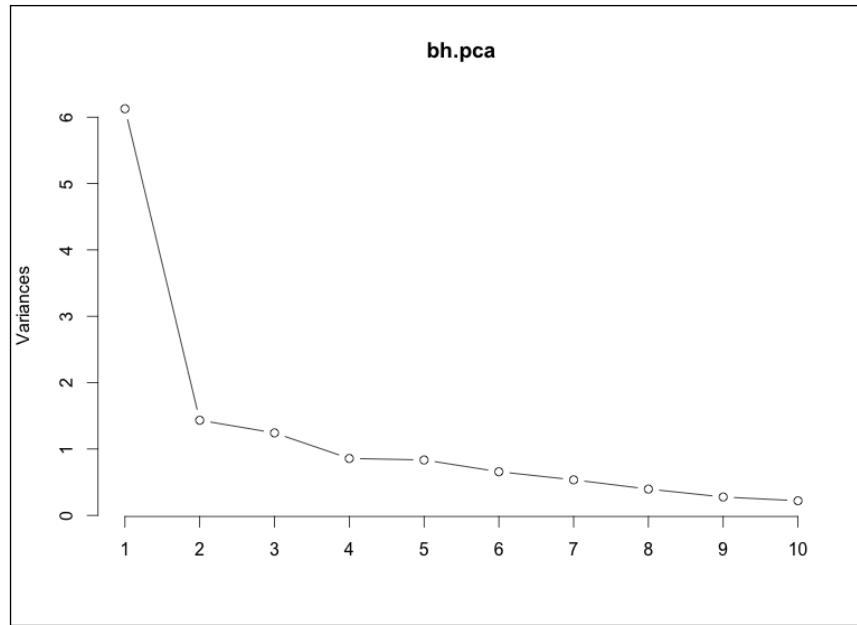
For a barplot use the following command:

```
> # barplot
> plot(bh.pca)
```



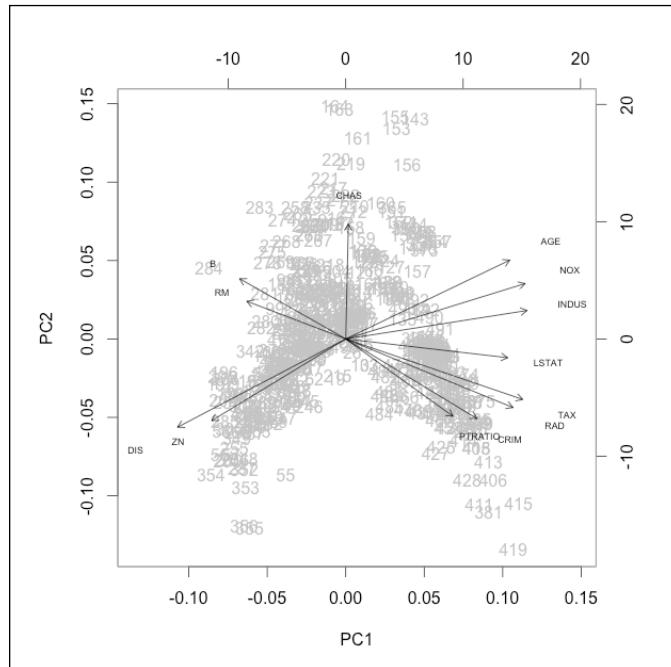
For a scree plot the following command can be used:

```
> # scree plot
> plot(bh.pca, type = "lines")
```



8. Create a biplot of the PCA results:

```
> biplot(bh.pca, col = c("gray", "black"))
```



9. Use the `x` component of `bh.pca` to see the computed principal component values for each of these cases:

```
> head(bh.pca$x, 3)
 PC1 PC2 PC3 PC4
[1,] -2.096223 0.7723484 0.3426037 0.8908924
[2,] -1.455811 0.5914000 -0.6945120 0.4869766
[3,] -2.072547 0.5990466 0.1669564 0.7384734
 PC5 PC6 PC7 PC8
[1,] 0.4226521 -0.3150264 0.3183257 -0.2955393
[2,] -0.1956820 0.2639620 0.5533137 0.2234488
[3,] -0.9336102 0.4476516 0.4840809 -0.1050622
 PC9 PC10 PC11
[1,] -0.42451671 -0.63957348 -0.03296774
[2,] -0.16679701 -0.08415319 -0.64017631
[3,] 0.06970615 0.18020170 -0.48707471
 PC12 PC13
[1,] -0.01942101 0.36561351
[2,] 0.12567304 -0.07064958
[3,] -0.13319472 -0.01400794
```

10. See the rotations and standard deviations using the following commands:

```
> bh.pca$rotation
> bh.pca$sdev
```

## How it works...

In step 1 the data is read.

In step 2 the correlation matrix of the relevant dimensions is generated to examine if there is scope for PCA to yield some dimensionality reduction. If most of the correlations are low, then PCA might not yield any reductions.

In step 3 the same is done graphically by showing a scatterplot matrix of the relevant variables.

In step 4 the PCA model is generated using the `prcomp` function. We used `scale=TRUE` to generate the model based on the correlation matrix and not the covariance matrix.

In step 5 the resulting model is printed. It shows the standard deviations of the variables used and the rotations for all the principal components in decreasing order of importance.

In step 6 the `summary` function is used to get different information on the model. This display is also ordered in decreasing order of importance of the components. For each principal component, this shows its standard deviation, proportion of variance, and the cumulative proportion of variance. We can use this to identify the components that capture most of the variability in the dataset. For example, the output tells us that the top 7 of the 13 principal components account for almost 90 percent of the variance.

In step 7 a barplot as well as a scree plot of the variances by PCA is generated using the `plot` function.

Step 8 shows how to generate the biplot. It uses the first two principal components as the main axes and shows how each variable loads on these two components. The top and right axes correspond to the scores of the data points on the two principal components.

In step 9 the `predict` function is used to view the principal component scores for each of our data points. You can also use this function to compute the principal component scores for new data:

```
> predict(bh.pca, newdata = ...)
```



# 6

## Lessons from History – Time Series Analysis

In this chapter, we will cover:

- ▶ Creating and examining date objects
- ▶ Operating on date objects
- ▶ Performing preliminary analyses on time series data
- ▶ Using time series objects
- ▶ Decomposing time series
- ▶ Filtering time series data
- ▶ Smoothing and forecasting using the Holt-Winters method
- ▶ Building an automated ARIMA Model

### Introduction

R has exceptional features for time series analysis and this chapter covers the topic through a chosen set of recipes. The `stats` package provides a basic set of features and several other packages go beyond.

### Creating and examining date objects

The base R package provides date functionality. This grab-bag recipe shows you several date-related operations in R. R internally represents dates as the number of days from 1 January, 1970.

## Getting ready

In this recipe, we will only be using features from the base package and not from any external data. Therefore, you do not need to perform any preparatory steps.

## How to do it...

Internally, R represents dates as the number of days from 1 January, 1970:

1. Get today's date:

```
> Sys.Date()
```

2. Create a date object from a string:

```
> # Supply year as two digits
> # Note correspondence between separators in the date string and
the format string
> as.Date("1/1/80", format = "%m/%d/%y")
```

```
[1] "1980-01-01"
```

```
> # Supply year as 4 digits
> # Note uppercase Y below instead of lowercase y as above
> as.Date("1/1/1980", format = "%m/%d/%Y")
```

```
[1] "1980-01-01"
```

```
> # If you omit format string, you must give date as "yyyy/mm/dd"
or as "YYYY-mm-dd"
> as.Date("1970/1/1")
```

```
[1] "1970-01-01"
```

```
> as.Date("70/1/1")
```

```
[1] "0070-01-01"
```

3. Use other options for separators (this example uses hyphens) in the format string, and also see the underlying numeric value:

```
> dt <- as.Date("1-1-70", format = "%m-%d-%y")
> as.numeric(dt)
```

```
[1] 0
```

## 4. Explore other format string options:

```
> as.Date("Jan 15, 2015", format = "%b %d, %Y")

[1] "2015-01-15"

> as.Date("January 15, 15", format = "%B %d, %y")

[1] "2015-01-15"
```

## 5. Create dates from numbers by typecasting:

```
> dt <- 1000
> class(dt) <- "Date"
> dt # 1000 days from 1/1/70

[1] "1972-09-27"

> dt <- -1000
> class(dt) <- "Date"
> dt # 1000 days before 1/1/70

[1] "1967-04-07"
```

## 6. Create dates directly from numbers by setting the origin date:

```
> as.Date(1000, origin = as.Date("1980-03-31"))

[1] "1982-12-26"

> as.Date(-1000, origin = as.Date("1980-03-31"))

[1] "1977-07-05"
```

## 7. Examine date components:

```
> dt <- as.Date(1000, origin = as.Date("1980-03-31"))
> dt

[1] "1982-12-26"

> # Get year as four digits
> format(dt, "%Y")

[1] "1982"

> # Get the year as a number rather than as character string
```

```
> as.numeric(format(dt, "%Y"))

[1] 1982

> # Get year as two digits
> format(dt, "%y")

[1] "82"

> # Get month
> format(dt, "%m")

[1] "12"

> as.numeric(format(dt, "%m"))

[1] 12

> # Get month as string
> format(dt, "%b")

[1] "Dec"

> format(dt, "%B")

[1] "December"

> months(dt)

[1] "December"

> weekdays(dt)

[1] "Sunday"

> quarters(dt)
[1] "Q4"

> julian(dt)

[1] 4742
attr(,"origin")
[1] "1970-01-01"
```

```
> julian(dt, origin = as.Date("1980-03-31"))

[1] 1000
attr(,"origin")
[1] "1980-03-31"
```

## How it works...

Step 1 shows how to get the system date.

Steps 2 through 4 show how to create dates from strings. You can see that, by specifying the format string appropriately, we can read dates from almost any string representation. We can use any separators, as long as we mimic them in the format string. The following table summarizes the formatting options for the components of the date:

| Format specifier | Description                                                    |
|------------------|----------------------------------------------------------------|
| %d               | Day of month as a number—for example, 15                       |
| %m               | Month as a number—for example, 10                              |
| %b               | Abbreviated string representation of month—for example, "Jan"  |
| %B               | Complete string representation of month—for example, "January" |
| %y               | Year as two digits—for example, 87                             |
| %Y               | Year as four digits—for example, 2001                          |

Step 5 shows how an integer can be typecast as a date. Internally, R represents dates as the number of days from 1 January, 1970 and hence zero corresponds to 1 January, 1970. We can convert positive and negative numbers to dates. Negative numbers give dates before 1/1/1970.

Step 6 shows how to find the date with a specific offset from a given date (origin).

Step 7 shows how to examine the individual components of a date object using the `format` function along with the appropriate format specification (see the preceding table) for the desired component. Step 7 also shows the use of the `months`, `weekdays`, and `julian` functions for getting the month, day of the week, and the Julian date corresponding to a date. If we omit the origin in the `julian` function, R assumes 1/1/1970 as the origin.

## See also...

- ▶ The *Operating on date objects* recipe in this chapter

## Operating on date objects

R supports many useful manipulations with date objects such as date addition and subtraction, and the creation of date sequences. This recipe shows many of these operations in action. For details on creating and examining date objects, see the previous recipe *Creating and examining date objects*, in this chapter.

### Getting ready

The base R package provides date functionality, and you do not need any preparatory steps.

### How to do it...

1. Perform the addition and subtraction of days from date objects:

```
> dt <- as.Date("1/1/2001", format = "%m/%d/%Y")
> dt

[1] "2001-01-01"

> dt + 100 # Date 100 days from dt

[1] "2001-04-11"

> dt + 31

[1] "2001-02-01"
```

2. Subtract date objects to find the number of days between two dates:

```
> dt1 <- as.Date("1/1/2001", format = "%m/%d/%Y")
> dt2 <- as.Date("2/1/2001", format = "%m/%d/%Y")
> dt1-dt1
```

Time difference of 0 days

```
> dt2-dt1
```

Time difference of 31 days

```
> dt1-dt2
```

Time difference of -31 days

```
> as.numeric(dt2-dt1)
```

```
[1] 31
```

3. Compare date objects:

```
> dt2 > dt1
```

```
[1] TRUE
```

```
> dt2 == dt1
```

```
[1] FALSE
```

4. Create date sequences:

```
> d1 <- as.Date("1980/1/1")
```

```
> d2 <- as.Date("1982/1/1")
```

```
> # Specify start date, end date and interval
> seq(d1, d2, "month")
```

```
[1] "1980-01-01" "1980-02-01" "1980-03-01" "1980-04-01"
[5] "1980-05-01" "1980-06-01" "1980-07-01" "1980-08-01"
[9] "1980-09-01" "1980-10-01" "1980-11-01" "1980-12-01"
[13] "1981-01-01" "1981-02-01" "1981-03-01" "1981-04-01"
[17] "1981-05-01" "1981-06-01" "1981-07-01" "1981-08-01"
[21] "1981-09-01" "1981-10-01" "1981-11-01" "1981-12-01"
[25] "1982-01-01"
```

```
> d3 <- as.Date("1980/1/5")
```

```
> seq(d1, d3, "day")
```

```
[1] "1980-01-01" "1980-01-02" "1980-01-03" "1980-01-04"
[5] "1980-01-05"
```

```
> # more interval options
```

```
> seq(d1, d2, "2 months")
```

```
[1] "1980-01-01" "1980-03-01" "1980-05-01" "1980-07-01"
[5] "1980-09-01" "1980-11-01" "1981-01-01" "1981-03-01"
[9] "1981-05-01" "1981-07-01" "1981-09-01" "1981-11-01"
[13] "1982-01-01"
```

```
> # Specify start date, interval and sequence length
> seq(from = d1, by = "4 months", length.out = 4)
```

```
[1] "1980-01-01" "1980-05-01" "1980-09-01" "1981-01-01"
```

5. Find a future or past date from a given date, based on an interval:

```
> seq(from = d1, by = "3 weeks", length.out = 2) [2]
```

```
[1] "1980-01-22"
```

## How it works...

Step 1 shows how you can add and subtract days from a date to get the resulting date.

Step 2 shows how you can find the number of days between two dates through subtraction. The result is a `difftime` object that you can convert into a number if needed.

Step 3 shows the logical comparison of dates.

Step 4 shows two different ways to create sequences of dates. In one, you specify the `from` date, the `to` date, and the fixed interval `by` between the sequence elements as a string. In the other, you specify the `from` date, the interval, and the number of sequence elements you want. If using the latter approach, you have to name the arguments.

Step 5 shows how you can create sequences by specifying the intervals more flexibly.

## See also...

- ▶ The *Creating and examining date objects* recipe in this chapter

## Performing preliminary analyses on time series data

Before creating proper time series objects, we may want to do some preliminary analyses. This recipe shows you how.

## Getting ready

The base R package provides all the necessary functionality. If you have not already downloaded the data files for this chapter, please do it now and ensure that they are located in your R working directory.

## How to do it...

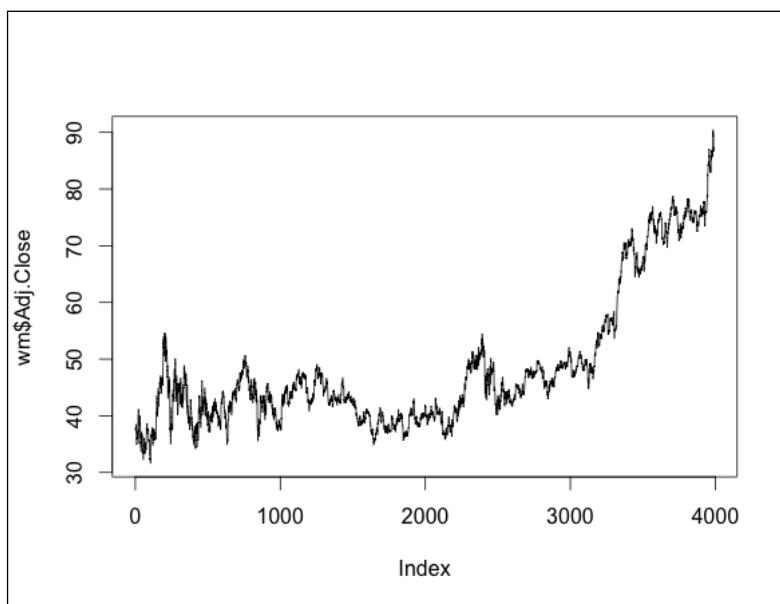
1. Read the file. We will use a data file that has the share prices of Walmart (downloaded from Yahoo Finance) between March 11, 1999 and January 15, 2015:

```
> wm <- read.csv("walmart.csv")
```

2. View the data as a line chart:

```
> plot(wm$Adj.Close, type = "l")
```

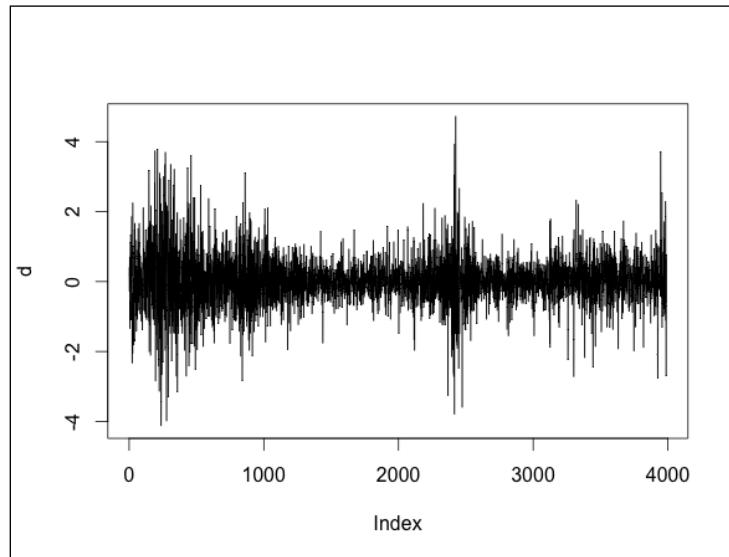
The data can be viewed as a line chart as follows:



3. Compute and plot daily price movements:

```
> d <- diff(wm$Adj.Close)
> plot(d, type = "l")
```

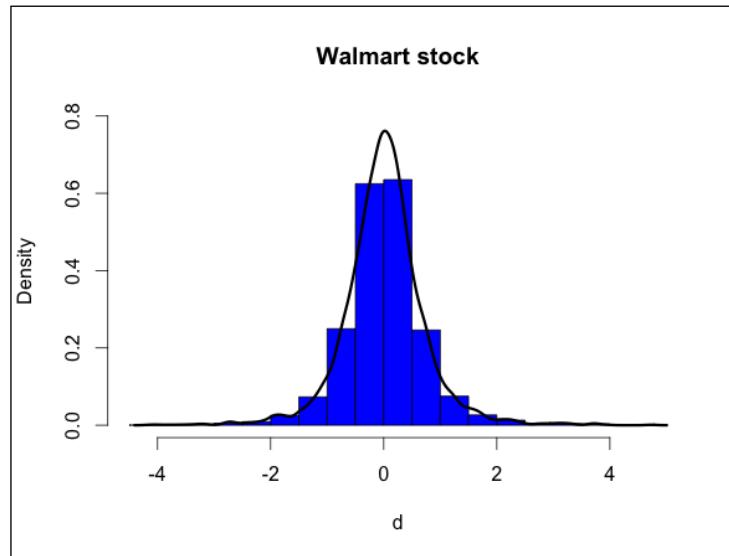
The plotted daily price movements appear as follows:



4. Generate a histogram of the daily price changes, along with a density plot:

```
> hist(d, prob = TRUE, ylim = c(0,0.8), main = "Walmart stock",
 col = "blue")
> lines(density(d), lwd = 3)
```

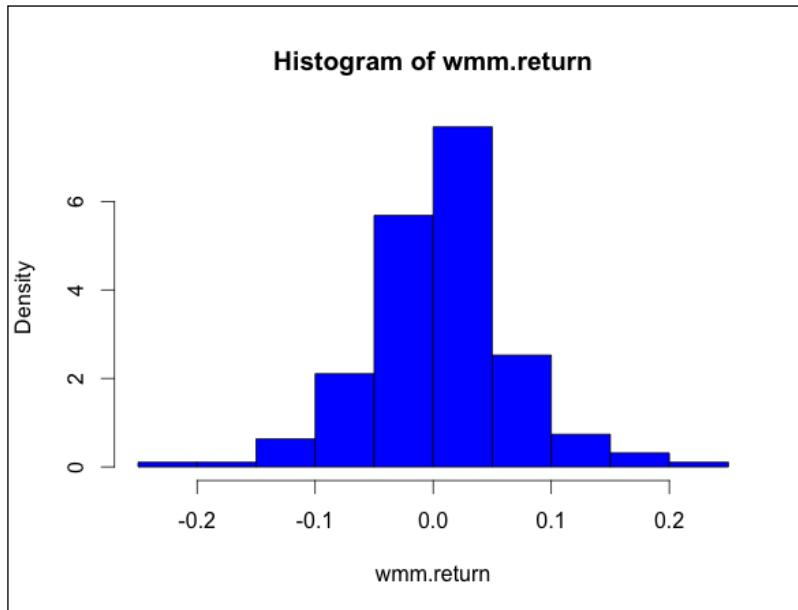
The following histogram shows daily price change:



5. Compute one-period returns:

```
> wmm <- read.csv("walmart-monthly.csv")
> wmm.ts <- ts(wmm$Adj.Close)
> d <- diff(wmm.ts)
> wmm.return <- d/lag(wmm.ts, k=-1)
> hist(wmm.return, prob = TRUE, col = "blue")
```

The following histogram shows the output of the preceding command:



## How it works...

Step 1 reads the data and step 2 plots it as a line chart.

Step 3 uses the `diff` function to generate single-period differences. It then uses the `plot` function to plot the differences. By default, the `diff` function computes single-period differences. You can use the `lag` argument to compute differences for greater lags. For example, the following calculates two-period lagged differences:

```
> diff(wmm$Adj.Close, lag = 2)
```

Step 4 generates a histogram of one-period price changes. It uses `prob=TRUE` to generate a histogram based on proportions, and then adds on a density plot as well to give a higher-granularity view of the shape of the distribution.

Step 5 computes one-period returns for the stock. It does this by dividing the one-period differences by the stock value at the first of the two periods that the difference is based on. It then generates a histogram of the returns.

### See also...

- ▶ The *Using time series objects* recipe in this chapter

## Using time series objects

In this recipe, we look at various features to create and plot time-series objects. We will consider data with single and multiple time series.

### Getting ready

If you have not already downloaded the data files for this chapter, do it now and ensure that the files are in your R working directory.

### How to do it...

1. Read the data. The file has 100 rows and a single column named `sales`:

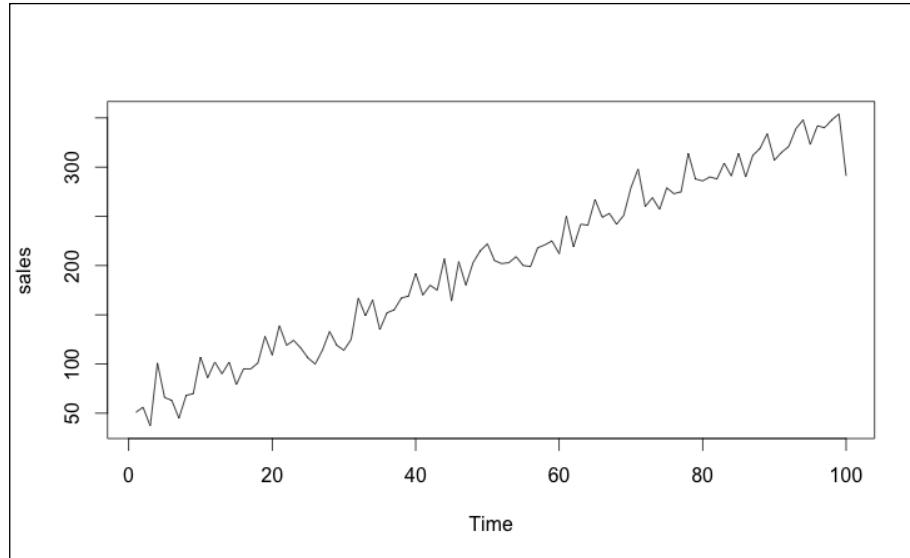
```
> s <- read.csv("ts-example.csv")
```

2. Convert the data to a simplistic time series object without any explicit notion of time:

```
> s.ts <- ts(s)
> class(s.ts)
[1] "ts"
```

3. Plot the time series:

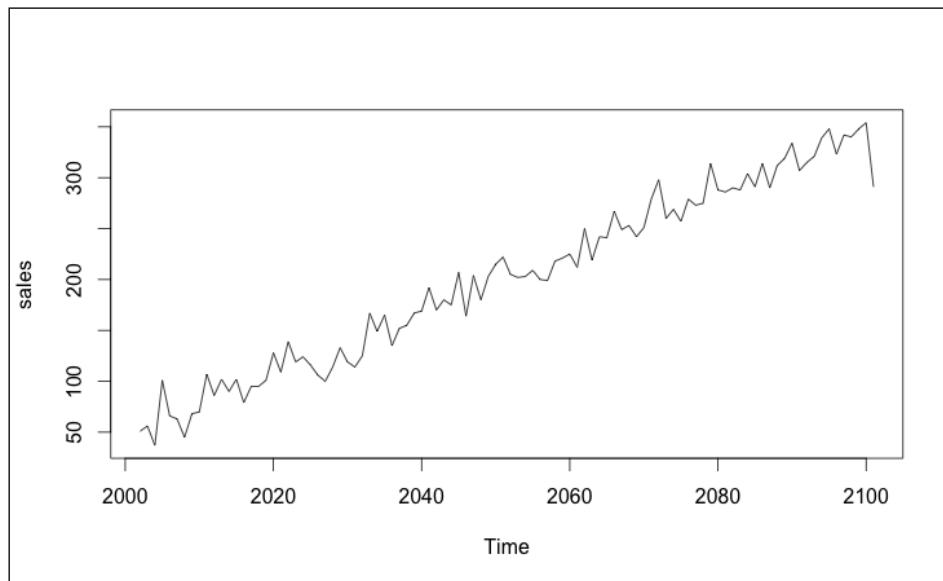
```
> plot(s.ts)
```



4. Create a proper time series object with time points:

```
> s.ts.a <- ts(s, start = 2002)
> s.ts.a
Time Series:
Start = 2002
End = 2101
Frequency = 1
 sales
[1,] 51
[2,] 56
[3,] 37
[4,] 101
[5,] 66
(output truncated)
> plot(s.ts.a)
> # results show that R treated this as an annual
> # time series with 2002 as the starting year
```

The result of the preceding commands is seen in the following graph:



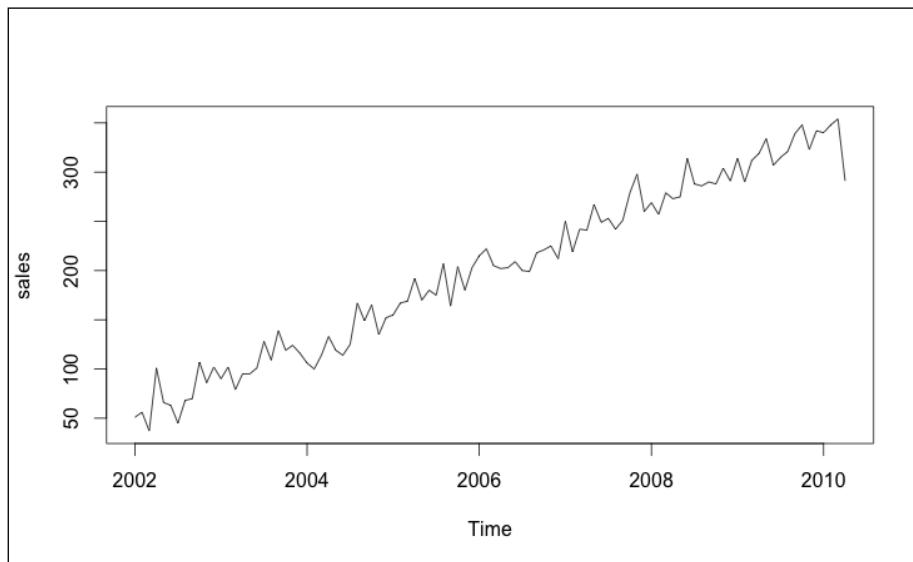
To create a monthly time series run the following command:

```
> # Create a monthly time series
> s.ts.m <- ts(s, start = c(2002,1), frequency = 12)
> s.ts.m
```

|      | Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 2002 | 51  | 56  | 37  | 101 | 66  | 63  | 45  | 68  | 70  | 107 | 86  | 102 |
| 2003 | 90  | 102 | 79  | 95  | 95  | 101 | 128 | 109 | 139 | 119 | 124 | 116 |
| 2004 | 106 | 100 | 114 | 133 | 119 | 114 | 125 | 167 | 149 | 165 | 135 | 152 |
| 2005 | 155 | 167 | 169 | 192 | 170 | 180 | 175 | 207 | 164 | 204 | 180 | 203 |
| 2006 | 215 | 222 | 205 | 202 | 203 | 209 | 200 | 199 | 218 | 221 | 225 | 212 |
| 2007 | 250 | 219 | 242 | 241 | 267 | 249 | 253 | 242 | 251 | 279 | 298 | 260 |
| 2008 | 269 | 257 | 279 | 273 | 275 | 314 | 288 | 286 | 290 | 288 | 304 | 291 |
| 2009 | 314 | 290 | 312 | 319 | 334 | 307 | 315 | 321 | 339 | 348 | 323 | 342 |
| 2010 | 340 | 348 | 354 | 291 |     |     |     |     |     |     |     |     |

```
> plot(s.ts.m) # note x axis on plot
```

The following plot can be seen as a result of the preceding commands:



```
> # Specify frequency = 4 for quarterly data
> s.ts.q <- ts(s, start = 2002, frequency = 4)
> s.ts.q
```

|      | Qtr1 | Qtr2 | Qtr3 | Qtr4 |
|------|------|------|------|------|
| 2002 | 51   | 56   | 37   | 101  |
| 2003 | 66   | 63   | 45   | 68   |
| 2004 | 70   | 107  | 86   | 102  |
| 2005 | 90   | 102  | 79   | 95   |
| 2006 | 95   | 101  | 128  | 109  |

(output truncated)

```
> plot(s.ts.q)
```

5. Query time series objects (we use the `s.ts.m` object we created in the previous step):

```
> # When does the series start?
> start(s.ts.m)
[1] 2002 1
> # When does it end?
> end(s.ts.m)
[1] 2010 4
> # What is the frequency?
> frequency(s.ts.m)
[1] 12
```

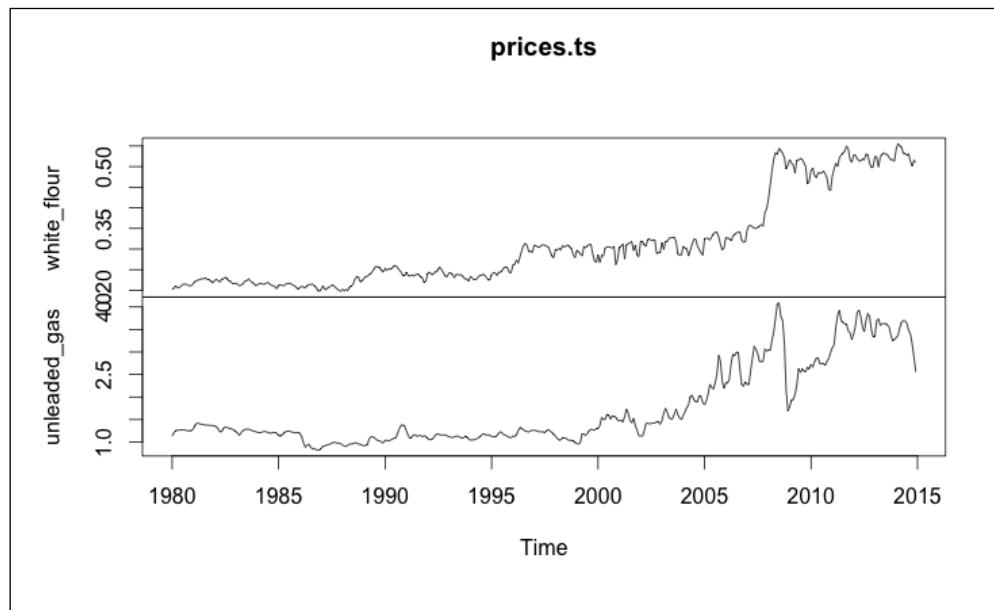
6. Create a time series object with multiple time series. This data file contains US monthly consumer prices for white flour and unleaded gas for the years 1980 through 2014 (downloaded from the website of the US Bureau of Labor Statistics):

```
> prices <- read.csv("prices.csv")
> prices.ts <- ts(prices, start=c(1980,1), frequency = 12)
```

7. Plot a time series object with multiple time series:

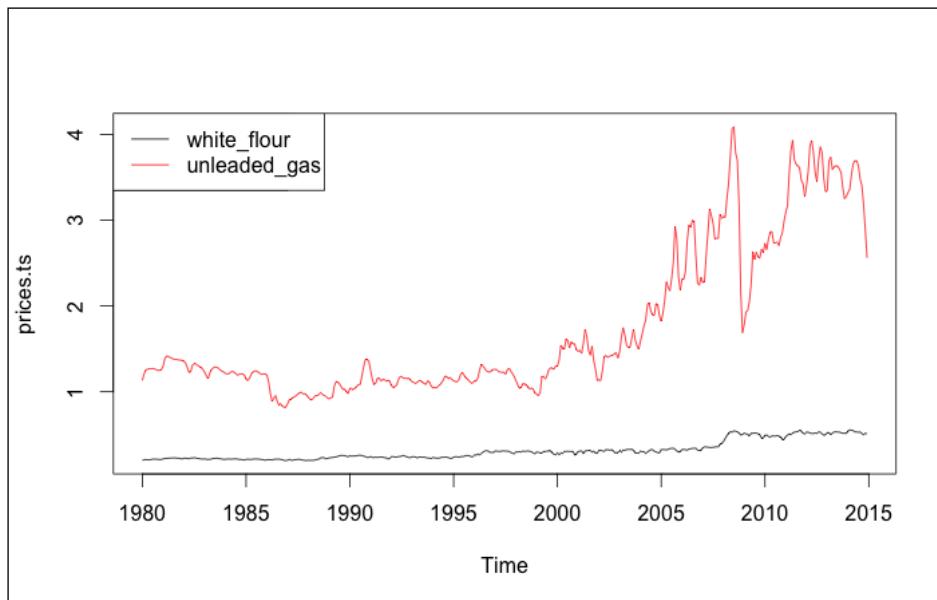
```
> plot(prices.ts)
```

The plot in two separate panels appears as follows:



```
> # Plot both series in one panel with suitable legend
> plot(prices.ts, plot.type = "single", col = 1:2)
> legend("topleft", colnames(prices.ts), col = 1:2, lty = 1)
```

Two series plotted in one panel appear as shown here:



## How it works...

Step 1 reads the data.

Step 2 uses the `ts` function to generate a time series object based on the raw data.

Step 3 uses the `plot` function to generate a line plot of the time series. We see that the time axis does not provide much information. Time series objects can represent time in more friendly terms.

Step 4 shows how to create time series objects with a better notion of time. It shows how we can treat a data series as an annual, monthly, or quarterly time series. The `start` and `frequency` parameters help us to control these data series.

Although the time series we provide is just a list of sequential values, in reality our data can have an implicit notion of time attached to it. For example, the data can be annual numbers, monthly numbers, or quarterly ones (or something else, such as 10-second observations of something). Given just the raw numbers (as in our data file, `ts-example.csv`), the `ts` function cannot figure out the time aspect and by default assumes no secondary time interval at all.

We can use the `frequency` parameter to tell `ts` how to interpret the time aspect of the data. The `frequency` parameter controls how many secondary time intervals there are in one major time interval. If we do not explicitly specify it, by default `frequency` takes on a value of 1. Thus, the following code treats the data as an annual sequence, starting in 2002:

```
> s.ts.a <- ts(s, start = 2002)
```

The following code, on the other hand, treats the data as a monthly time series, starting in January 2002. If we specify the `start` parameter as a number, then R treats it as starting at the first subperiod, if any, of the specified `start` period. When we specify `frequency` as different from 1, then the `start` parameter can be a vector such as `c(2002, 1)` to specify the series, the major period, and the subperiod where the series starts. `c(2002, 1)` represents January 2002:

```
> s.ts.m <- ts(s, start = c(2002, 1), frequency = 12)
```

Similarly, the following code treats the data as a quarterly sequence, starting in the first quarter of 2002:

```
> s.ts.q <- ts(s, start = 2002, frequency = 4)
```

The `frequency` values of 12 and 4 have a special meaning—they represent monthly and quarterly time sequences.

We can supply `start` and `end`, just one of them, or none. If we do not specify either, then R treats the `start` as 1 and figures out `end` based on the number of data points. If we supply one, then R figures out the other based on the number of data points.

While `start` and `end` do not play a role in computations, `frequency` plays a big role in determining seasonality, which captures periodic fluctuations.

If we have some other specialized time series, we can specify the `frequency` parameter appropriately. Here are two examples:

- ▶ With measurements taken every 10 minutes and seasonality pegged to the hour, we should specify `frequency` as 6
- ▶ With measurements taken every 10 minutes and seasonality pegged to the day, use `frequency = 24*6` (6 measurements per hour times 24 hours per day)

Step 5 shows the use of the functions `start`, `end`, and `frequency` to query time series objects.

Steps 6 and 7 show that R can handle data files that contain multiple time series.

## See also...

- ▶ The *Performing preliminary analyses on time series objects* recipe in this chapter

## Decomposing time series

The `stats` package provides many functions to process time series. This recipe covers the use of the `decompose` and `stl` functions to extract the seasonal, trend, and random components of time series.

### Getting ready

If you have not already downloaded the data files for this chapter, do it now and ensure that the files are in your R working directory.

### How to do it...

The following steps decompose time series:

1. Read the data. The file has the Bureau of Labor Statistics monthly price data for unleaded gas and white flour for 1980 through 2014:

```
> prices <- read.csv("prices.csv")
```

2. Create and plot the time series of gas prices:

```
> prices.ts = ts(prices, start = c(1980,1), frequency = 12)
> plot(prices.ts[,2])
```

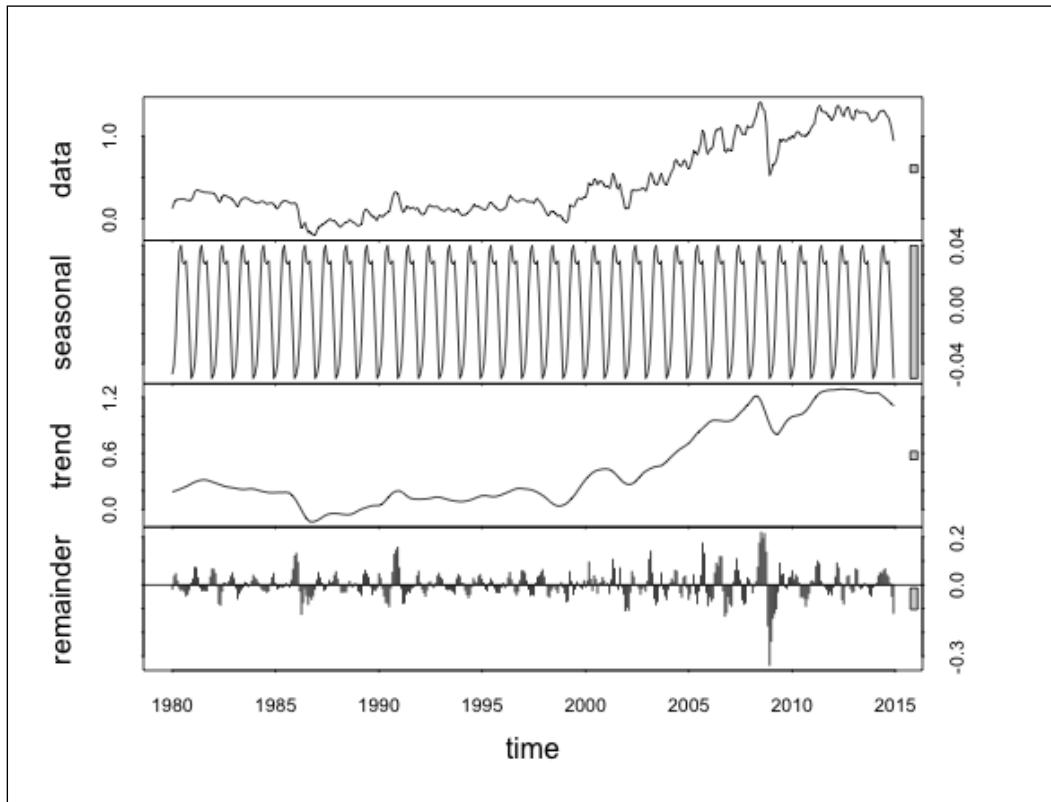
3. The plot shows seasonality in gas prices. The amplitude of fluctuations seems to increase with time and hence this looks like a multiplicative time series. Thus, we will use the log of the prices to make it additive. Use the `stl` function to perform a Loess decomposition of the gas prices:

```
> prices.stl <- stl(log(prices.ts[,1]), s.window =
 "period")
```

4. Plot the results of `stl`:

```
> plot(prices.stl)
```

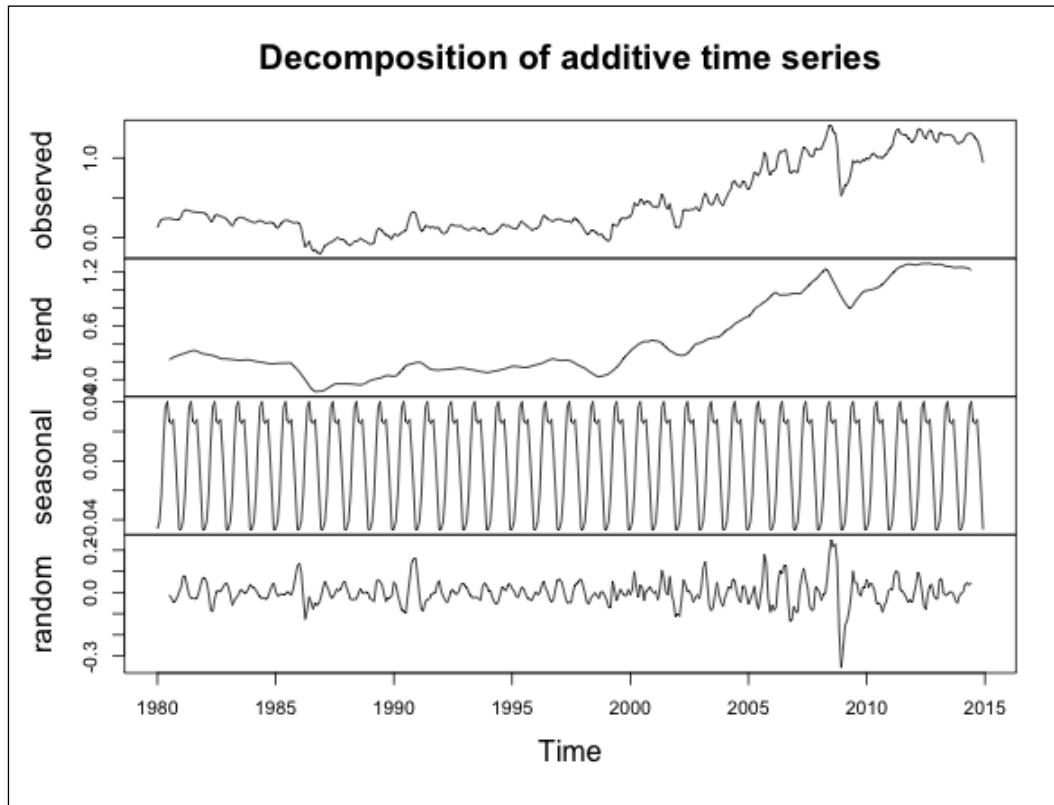
The following plot is the result of the preceding command:



5. Alternately, you can use the `decompose` function to perform a decomposition by moving averages:

```
> prices.dec <- decompose(log(prices.ts[,2]))
> plot(prices.dec)
```

The following graph shows the output of the preceding command:



6. Adjust the gas prices for seasonality and plot it:

```
> gas.seasonally.adjusted <- prices.ts[,2] -
 prices.dec$seasonal
> plot(gas.seasonally.adjusted)
```

### How it works...

Step 1 reads the data and Step 2 creates and plots the time series. For more details, see the recipe, *Using time series objects*, earlier in this chapter.

Step 3 shows the use of the `stl` function to decompose an additive time series. Since our earlier plot indicated that the amplitude of the fluctuations increased with time, thereby suggesting a multiplicative time series, we applied the `log` function to convert it into an additive time series and then decomposed it.

Step 4 uses the `plot` function to plot the results.

Step 5 uses the `decompose` function to perform a decomposition through moving averages and then plots it.

Step 6 adjusts gas prices for seasonality by subtracting the seasonal component from the original time series of the gas prices, and then plots the resulting time series.

### See also...

- ▶ The *Using time series objects* recipe in this chapter
- ▶ The *Filtering time series data* recipe in this chapter
- ▶ The *Smoothing and forecasting using the Holt-Winters method* recipe in this chapter

## Filtering time series data

This recipe shows how we can use the `filter` function from the `stats` package to compute moving averages.

### Getting ready

If you have not already done so, download the data files for this chapter and ensure that they are available in your R working directory.

### How to do it...

To filter time series data, follow these steps:

1. Read the data. The file has fictitious weekly sales data for some product:

```
> s <- read.csv("ts-example.csv")
```

2. Create the filtering vector. We assume a seven-period filter:

```
> n <- 7
> wts <- rep(1/n, n)
```

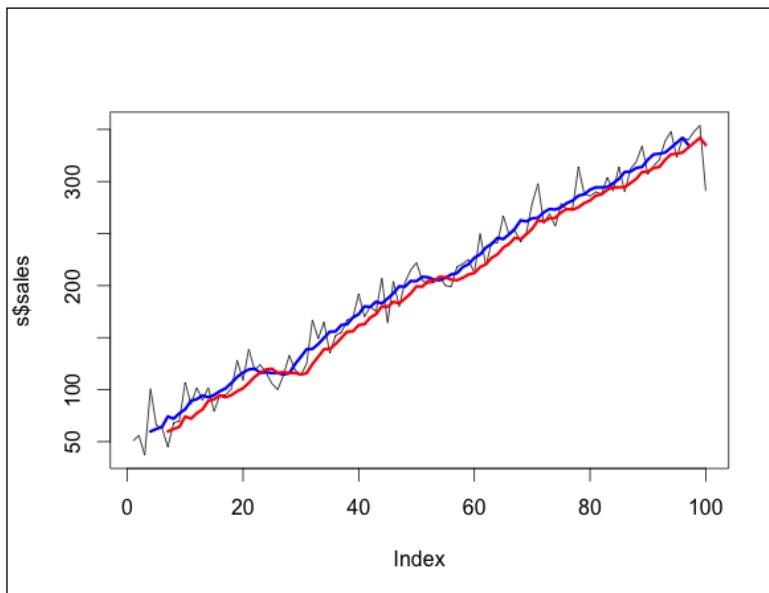
3. Compute the symmetrically filtered values (three past values, one current value, and three future values) and one-sided values (one current and six past values):

```
> s.filter1 <- filter(s$sales, filter = wts, sides = 2)
> s.filter2 <- filter(s$sales, filter = wts, sides = 1)
```

4. Plot the filtered values:

```
> plot(s$sales, type = "l")
> lines(s.filter1, col = "blue", lwd = 3)
> lines(s.filter2, col = "red", lwd = 3)
```

The plotted filtered values appear as follows:



## How it works...

Step 1 reads the data.

Step 2 creates the filtering weights. We used a window of seven periods. This means that the weighted average of the current value and six others will comprise the filtered value at the current position.

Step 3 computes the two-sided filter (the weighted average of the current value and three prior and three succeeding values) and a one-sided filter based on the current value and six prior ones.

Step 4 plots the original data and the symmetric and one-sided filters. We can see that the two-sided filter tracks the changes earlier.

## See also...

- ▶ The *Using time series objects* recipe in this chapter
- ▶ The *Decomposing time series* recipe in this chapter
- ▶ The *Smoothing and forecasting using the Holt-Winters method* recipe in this chapter

## Smoothing and forecasting using the Holt-Winters method

The `stats` package contains functionality for applying the `HoltWinters` method for exponential smoothing in the presence of trends and seasonality, and the `forecast` package extends this to forecasting. This recipe addresses these topics.

### Getting ready

If you have not already downloaded the files for this chapter, do it now and place them in your R working directory. Install and load the `forecast` package.

### How to do it...

To apply the `HoltWinters` method for exponential smoothing and forecasting, follow these steps:

1. Read the data. The file has monthly stock prices from Yahoo! Finance for Infosys between March 1999 and January 2015:

```
> infy <- read.csv("infy-monthly.csv")
```

2. Create the time series object:

```
> infy.ts <- ts(infy$Adj.Close, start = c(1999,3),
frequency = 12)
```

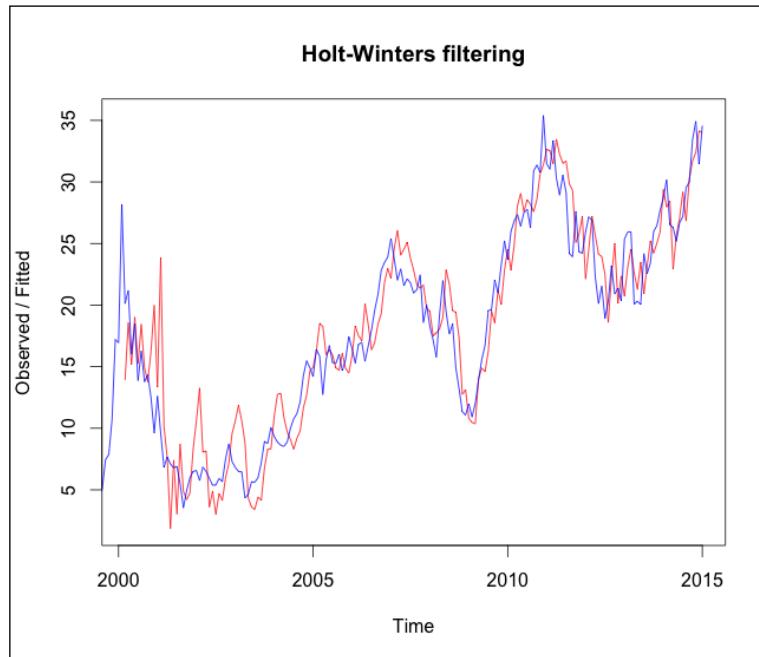
3. Perform Holt-Winters exponential smoothing:

```
> infy.hw <- HoltWinters(infy.ts)
```

4. Plot the results:

```
> plot(infy.hw, col = "blue", col.predicted = "red")
```

The plotted result can be seen as follows:



Examine the results:

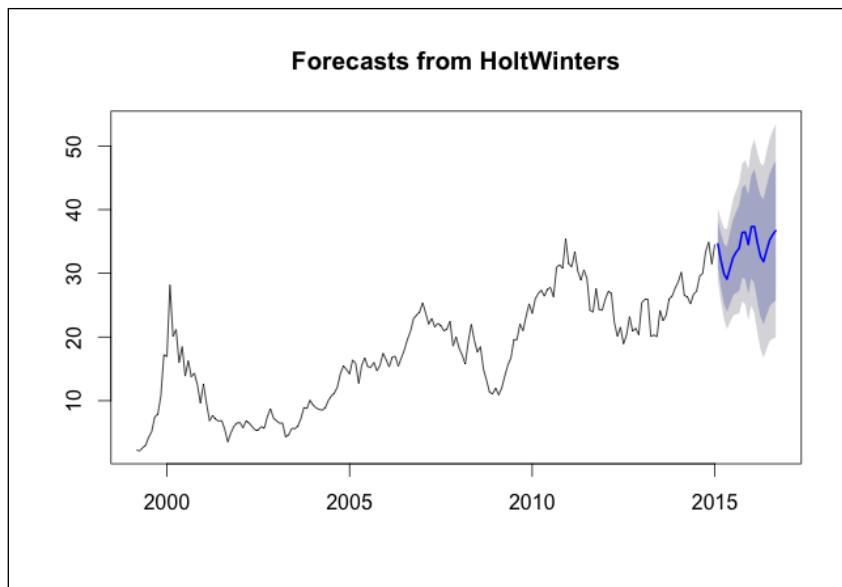
```
> # See the squared errors
> infy.hw$SSE
[1] 1446.232
> # The alpha beta and gamma used for filtering
> infy.hw$alpha
 alpha
0.5658932
> infy.hw$beta
 beta
0.009999868
> infy.hw$gamma
 gamma
1
> # the fitted values
> head(infy.hw$fitted)
 xhat level trend season
[1,] 13.91267 11.00710 0.5904618 2.31510417
[2,] 18.56803 15.11025 0.6255882 2.83218750
```

```
[3,] 15.17744 17.20828 0.6403124 -2.67114583
[4,] 19.01611 18.31973 0.6450237 0.05135417
[5,] 15.23710 18.66703 0.6420466 -4.07197917
[6,] 18.45236 18.53545 0.6343104 -0.71739583
```

Generate and plot forecasts with the Holt-Winters model:

```
> library(forecast)
> infy.forecast <- forecast(infy.hw, h=20)
> plot(infy.forecast)
```

The following is the resulting plot:



## How it works...

Step 1 reads the data and in step 2 the time series object, `ts`, is created. For more details, see the recipe, *Using time series objects*, earlier in this chapter.

In step 3 the `HoltWinters` function is used to smooth the data.

In step 4 the resulting `HoltWinters` object is plotted. It shows the original time series as well as the smoothed values.

Step 5 shows the functions available to extract information from the Holt-Winters model object.

In step 6 the `predict.HoltWinters` function is used to predict future values. The colored bands show the 85 % and 95 % confidence intervals.

## See also...

- ▶ The *Using time series objects* recipe in this chapter
- ▶ The *Decomposing time series* recipe in this chapter
- ▶ The *Filtering time series data* recipe in this chapter

# Building an automated ARIMA model

The `forecast` package provides the `auto.arima` function to fit the best ARIMA models for a univariate time series.

## Getting ready

If you have not already downloaded the files for this chapter, do it now and place them in your R working directory. Install and load the `forecast` package.

## How to do it...

To build an automated ARIMA model, follow these steps:

1. Read the data. The file has monthly stock prices from Yahoo! Finance for Infosys between March 1999 and January 2015:

```
> infy <- read.csv("infy-monthly.csv")
```

2. Create the time series object:

```
> infy.ts <- ts(infy$Adj.Close, start = c(1999, 3),
frequency = 12)
```

3. Run the ARIMA model:

```
> infy.arima <- auto.arima(infy.ts)
```

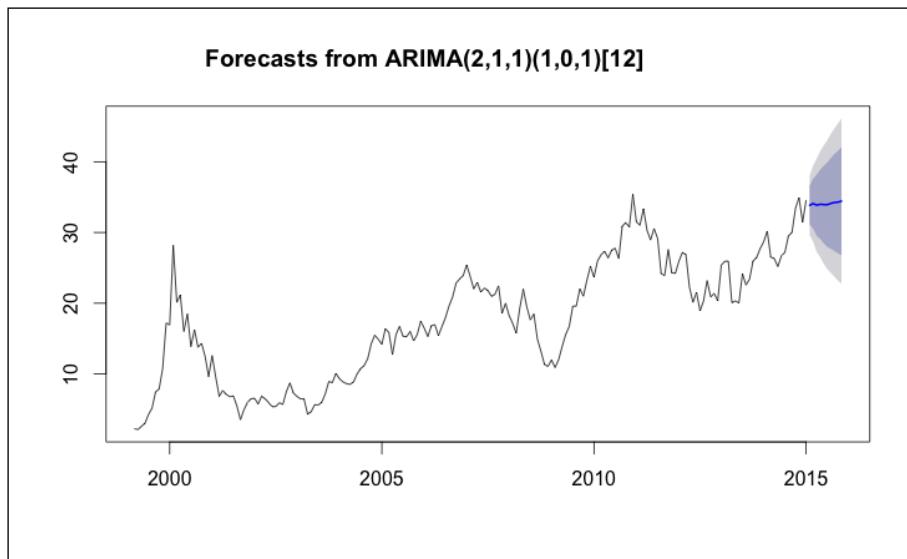
4. Generate the forecast using the ARIMA model:

```
> infy.forecast <- forecast(infy.arima, h=10)
```

5. Plot the results:

```
> plot(infy.forecast)
```

The plotted result can be seen as follows:



## How it works...

Step 1 reads the data.

In step 2, the time series object, `ts`, is created. For more details, see the recipe *Using time series objects*, earlier in this chapter.

In step 3, the `auto.arima` function in the `forecast` package is used to generate the ARIMA model. This function conducts an orderly search to generate the best ARIMA model according to the AIC, AICc, or the BIC value. We control the criterion used through the `ic` parameter (for example, `ic = "aicc"`). If we provide no value, the function uses AICc.

In step 4, the forecast for the specified time horizon (the `h` parameter) is generated.

Step 5 plots the results. The two bands show the 85 percent and the 95 percent confidence intervals. You can control the color of the data line through the `col` parameter and the color of the forecast line through `fcol`.

## See also...

- ▶ The *Using time series objects* recipe in this chapter

# 7

## **It's All About Your Connections – Social Network Analysis**

In this chapter, you will cover:

- ▶ Downloading social network data using public APIs
- ▶ Creating adjacency matrices and edge lists
- ▶ Plotting social network data
- ▶ Computing important network metrics

### **Introduction**

When we think of the term "social network," sites such as Twitter, Facebook, Google+, LinkedIn, and Meetup immediately come to mind. However, data analysts have applied the concepts of social network analysis in domains as varied as co-authorship networks, human networks, the spread of disease, migratory birds, and interlocking corporate board memberships, just to name a few. In this chapter, we will cover recipes to make sense of social network data. R provides multiple packages to manipulate social network data; we address the most prominent of these, `igraph`.

## Downloading social network data using public APIs

Social networking websites provide public APIs to enable us to download their data. In this recipe, we cover the process of downloading data from Meetup.com using their public API. You can adapt this basic approach to download data from other websites. In this recipe, we get the data in JSON format and then import it into an R data frame.

### Getting ready

In this recipe, you will download data from Meetup.com. To gain access to the data, you need to be a member:

- ▶ If you do not have an account in <http://www.meetup.com>, sign up and become a member of a couple of groups.
- ▶ You need your own API key to download data through scripts. Get your own by clicking on the **API Key** link on [http://www.meetup.com/meetup\\_api/](http://www.meetup.com/meetup_api/). Save the key.

You can replicate the steps in this recipe without additional information. However, if you would like more details, you can read about the API at [http://www.meetup.com/meetup\\_api/](http://www.meetup.com/meetup_api/).

Download the `groups.json` file and place it in your R working directory.

### How to do it...

In this recipe, we see how to get data from the console and how R scripts are used:

1. Download information about Meetup groups that satisfy certain criteria.  
`http://www.meetup.com` allows you to download data from their **Console** by filling up fields to define your criteria. Alternately, you can also construct a suitable URL to directly get the data. The former approach has some limitations when downloading large volumes of data.

2. In this recipe, we first show you how to get data from the console. We then show you how to use R scripts that use URLs for larger data volumes. We will first get a list of groups. Use your browser to visit [http://www.meetup.com/meetup\\_api/](http://www.meetup.com/meetup_api/). Click on **Console** and then click on the first link under **Groups** on the right (GET /2/groups). Enter the topic you are interested in and enter your two character ISO country code (see [http://en.wikipedia.org/wiki/ISO\\_3166-1](http://en.wikipedia.org/wiki/ISO_3166-1) for country codes) and city or zip code. You can specify "radius" to restrict the number of groups returned. We used hiking for topic, US for country, 08816 for zip code, and 25 for radius. Click on **Show Response** to see the results. If you get an error, try with different values. If you used different selection criteria, you may see many groups. From the earlier results, you will notice information on many attributes for each group. You can use the "only" field to get some of these attributes, for example, to get only the group ID, name, and number of members, enter id, name, members in the "only" textbox (note that there is no space after the commas).
3. You can also get information using a URL directly, but you will need to use your API key for this. For example, the following URL gets the ID, name, and number of members for all hiking-related groups in Amsterdam, NL (replace <<your api key>> with the API key you downloaded earlier in the *Getting started* part of this recipe; no angle braces needed):

```
http://api.meetup.com/2/groups?topic=hiking&country=NL&city=Amsterdam&only=id,name,members&key=<<your api key>>
```

You will note that the results look different in the console when using the URL directly, but this is just a formatting issue. The console pretty prints the JSON, whereas we see unformatted JSON when we use a URL.

4. We will now save the downloaded data; the process depends on whether you used the console approach in step 2 or the URL approach in step 3. If you used the console approach, you should select the block of text starting with the { before results and ending with the very last }. Paste the copied text into a text editor and save the file as `groups.json`. On the other hand, if you used the URL approach of step 3, right-click and select "Save As" to save the displayed results as a file named `groups.json` in your R working directory. You can also use the `groups.json` file that you downloaded earlier.
5. We will now load the data from the saved JSON file into an R data frame. For details, refer to the recipe *Reading JSON data* in Chapter 1, *Acquire and Prepare the Ingredients – Your Data*:

```
> library(jsonlite)
> g <- fromJSON("groups.json")
> groups <- g$results
> head(groups)
```

6. For each group, we will now use the Meetup.com API to download member information into a data frame called `users`. Among the code files that you downloaded for this chapter is a file called `rdacb.getusers.R`. Source this file into your R environment now and run the following code. For each group from our list of groups, this code uses the Meetup.com API to get the group's members. It generates a data frame with a set of `group_id`, `user_id` pairs. In the following command, replace `<>apikey>>` with your actual API key from the *Getting ready* section. Be sure to enclose the key in double quotes and also be sure that you do not have any angle brackets in the command. This command can take a while to execute because of the sheer number of web requests and the volume of data involved. If you get an error message, see the *How it works...* section for this recipe:

```
> source("rdacb.getusers.R")
> # in command below, substitute your api key for
> # <>apikey>> and enclose it in double-quotes
> members <- rdacb.getusers(groups, <>apikey>>)
```

This creates a data frame with the variables (`group_id`, `user_id`).

7. The `members` data frame now has information about the social network, and normally we will use it for all further processing. However, since it is very large, many of the steps in subsequent recipes will take a lot of processing time. For convenience, we reduce the size of the social network by retaining only members who belong to more than 16 groups. This step uses data tables; see *Chapter 9, Work Smarter, Not harder – Efficient and Elegant R code* for more details. If you would like to work with the complete network, execute the `users <- members` command and skip to step 8 without executing these two code lines:

```
> library(data.table)
> users <- setDT(members) [, .SD[.N > 16], by = user_id]
```

8. Save the members data before further processing:

```
> save(users, file="meetup_users.Rdata")
```

## How it works...

Steps 2 and 3 get data from Meetup.com using their console.

By default, Meetup APIs return 20 results (JSON documents) for each call. However, by adding `page=n` in the console or in the URL (where  $n$  is a number), we can get more documents (up to a maximum of 200). The API response contains metadata information including the number of documents returned (in the `count` element), the total documents available (the `total_count` element), the URL used to get the response, and the URL to get the next set of results (the `next` element).

Step 4 saves the results displayed in the browser as `groups.json` in your R working directory.

Step 5 loads the JSON data file into an R data frame using the `fromJSON` function in the `jsonlite` package. For more details on this, refer to the recipe *Reading JSON data* in Chapter 1, *Acquire and Prepare the Ingredients – Your Data*.

The returned object `g` contains the results in the `results` element, and we assign `g$results` (a data frame) to the variable `groups`.

Step 6 uses the `rdacb.getusers` convenience function to iterate through each group and get its members. The function constructs the appropriate API URL using group `id`. Since each call returns only a fixed number of users, the `while` loop of the function iterates till it gets all the users. The next element of the result returned tells us if the group has more members.

There can be groups with no members and hence we check if `temp$results` returns a data frame. The API returns group and user IDs as they are in Meetup.com.

If the Meetup.com site is overloaded with several API requests during the time of your invocation, you may get an error message "Error in function (type, msg, asError = TRUE) : Empty reply from server". Retry the same step again. Depending on the number of groups and the number of members in each group, this step can take a long time.

At this point, we have the data for a very large social network. Subsequent recipes in this chapter use the social network that we create in this recipe. Some of the steps in subsequent recipes can take a lot of processing time if run on the complete network. Using a smaller network will suffice for illustration. Therefore, step 7 uses `data.table` to retain only members who belong to more than 16 groups.

Step 8 saves the member data in a file for possible future use. We now have a two-mode network, where the first mode is a set of groups and the second mode is the list of members. From this, we can either create a network of users based on common group memberships or a network of groups based on common members. In the rest of this chapter, we do the former.

We created a data frame in which each row represents a membership of an individual user in a group. From this information, we can create representations of social networks.

## See also...

- ▶ *Reading JSON data* in Chapter 1, *Acquire and Prepare the Ingredients – Your Data*
- ▶ *Slicing, dicing, and combining data with data tables* in Chapter 9, *Work Smarter, Not Harder – Efficient and Elegant R Code*

## Creating adjacency matrices and edge lists

We can represent social network data in different formats. We cover two common representations: sparse adjacency matrices and edge lists.

Taking data from the Meetup.com social networking site (from the previous recipe in this chapter—*Downloading social network data using public APIs*), this recipe shows how you can convert a data frame with membership information into a sparse adjacency matrix and then to an edge list.

In this application, nodes represent users of Meetup.com and an edge connects two nodes if they are members of at least one common group. The number of common groups for a pair of people will represent the weight of the connection.

### Getting ready

If you have not yet installed the `Matrix` package, you should do so now using the following code:

```
> install.packages("Matrix")
```

If you completed the prior recipe *Downloading social network data using public APIs* and have the `meetup_users.Rdata` file, you can use it. Otherwise, you can download that data file from the book's website and place it in your R working directory.

### How to do it...

To create adjacency matrices and edge lists, follow these steps:

1. Load Meetup.com user information from the `meetup_users.Rdata` file. This creates a data frame called `users` which has the variables (`user_id`, `group_id`):

```
> load("meetup_users.Rdata")
```

2. Create a sparse matrix with groups on the rows and users on the columns with `TRUE` on each (`group_id`, `user_id`) position, where the group has the user as a member. If you have a large number of users, this step will take a long time. It also needs a lot of memory to create a sparse matrix. If your R session freezes, you will have to either find a computer with more RAM or reduce the number of users and try again:

```
> library(Matrix)
> grp.membership = sparseMatrix(users$group_id, users$user_id, x =
 TRUE)
```

3. Use the sparse matrix to create an adjacency matrix with users on both rows and columns with the number of common groups between a pair of users as the matrix element:

```
> adjacency = t(grp.membership) %*% grp.membership
```

4. We can use the group membership matrix to create a network of groups instead of users with groups as the nodes and edges representing common memberships across groups. In this example, we will consider only a network of users.

5. Use the adjacency matrix to create an edge list:

```
> users.edgelist <- as.data.frame(summary(adjacency))
> names(users.edgelist)
[1] "i" "j" "x"
```

6. The relationship between any two users is reciprocal. That is, if users 25 and 326 have 32 groups in common, then the edge list currently duplicates that information as (25, 362, 32) and (362, 25, 32). We need to keep only one of these. Also, our adjacency matrix has non-zero diagonal elements and the edge list has edges corresponding to those. We can eliminate these by keeping only the edges corresponding to the upper or lower triangle of the adjacency matrix:

```
Extract upper triangle of the edgelist
> users.edgelist.upper <- users.edgelist[users.edgelist$i < users.edgelist$j,]
```

7. Save the data, just in case:

```
> save(users.edgelist.upper, file = "users_edgelist_upper.Rdata")
```

## How it works...

Step 1 loads the users' group membership data from the `meetup_users.Rdata` saved file. This creates a `users` data frame with the `(user_id, group_id)` structure. From this, we want to create a network in which the nodes are users and a pair of users has an edge if they are members of at least one common group. We want to have the number of common group memberships as the weight of an edge.

Step 2 converts the information in the group membership data frame to a sparse matrix with groups on the rows and users on the columns. To clarify, the following table shows a sample matrix with four groups and nine users. The first group has users 1, 4, and 7 as members, while the second has users 1, 3, 4, and 6 as members, and so on. We have shown the complete matrix here, but step 2 creates a much more space efficient sparse representation of this information:

|        |   | Users |   |   |   |   |   |   |   |   |
|--------|---|-------|---|---|---|---|---|---|---|---|
| Groups |   | 1     | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|        | 1 | 1     | . | . | 1 | . | . | 1 | . | . |
|        | 2 | 1     | . | 1 | 1 | . | 1 | . | . | . |
|        | 3 | 1     | 1 | . | 1 | . | 1 | 1 | . | 1 |
|        | 4 | .     | 1 | 1 | . | . | 1 | . | . | 1 |

While the sparse matrix has all the network information, several social network analysis functions work with the data represented as an "adjacency matrix" or as an "edge list." We want to create a social network of Meetup.com users. The adjacency matrix will have the shape of a square matrix with users on both rows and columns, and the number of shared groups as the matrix element. For the sample data in the preceding figure, the adjacency matrix will look like this:

|       |   | Users |   |   |   |   |   |   |   |   |
|-------|---|-------|---|---|---|---|---|---|---|---|
| Users |   | 1     | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|       | 1 | 3     | 1 | 1 | 3 | 0 | 2 | 2 | 0 | 1 |
|       | 2 | 1     | 2 | 1 | 1 | 0 | 2 | 1 | 0 | 2 |
|       | 3 | 1     | 1 | 2 | 1 | 0 | 2 | 0 | 0 | 1 |
|       | 4 | 3     | 1 | 1 | 3 | 0 | 2 | 2 | 0 | 1 |
|       | 5 | 0     | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|       | 6 | 2     | 2 | 2 | 2 | 0 | 3 | 1 | 0 | 2 |
|       | 7 | 2     | 1 | 0 | 2 | 0 | 1 | 2 | 0 | 1 |
|       | 8 | 0     | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|       | 9 | 1     | 2 | 1 | 1 | 0 | 2 | 1 | 0 | 2 |

Step 3 creates a sparse adjacency matrix from the earlier sparse group membership matrix. From the earlier figure, we see that users 1 and 4 have three groups in common (groups 1, 2, and 3) and thus the elements (1, 4) and (4, 1) of the matrix have a 3 in them. The diagonal elements simply indicate the number of groups to which the corresponding users belong. User 1 belongs to three groups and hence (1, 1) has a 3. We only need the upper or lower triangle of the matrix, and we take care of that in a later step.

Step 4 creates an edge list from the sparse adjacency matrix. An edge list will have the following structure: (`user_id1`, `users_id2`, `number of common groups`). For the sample data in the preceding figure, the edge list will look like this (only 10 of the 47 edges are shown in the following figure). Once again, we need only the upper or lower triangle of this, and we take care of that in a later step:

| i   | j   | x   |
|-----|-----|-----|
| 1   | 1   | 3   |
| 2   | 1   | 1   |
| 3   | 1   | 1   |
| 4   | 1   | 3   |
| 6   | 1   | 2   |
| 7   | 1   | 2   |
| 9   | 1   | 1   |
| 1   | 2   | 1   |
| 2   | 2   | 2   |
| 3   | 2   | 1   |
| ... | ... | ... |

We have a symmetric network. Saying that users A and B have  $n$  groups in common is the same thing as saying that users B and A have  $n$  groups in common. Thus, we do not need to represent both of these in the adjacency matrix or in the edge list. We also do not need any edges connecting users to themselves and thus do not need the diagonal elements in the sparse matrix or elements with the same value for `i` and `j` in the edge list.

Step 6 eliminates the redundant edges and the edges that connect a user to themselves.

Step 7 saves the sparse network for possible future use.

We have created both a sparse adjacency matrix and an edge list representation of the social network of users in our chosen Meetup groups. We can use these representations for social network analyses.

## See also...

- ▶ *Downloading social network data using public APIs*, from this chapter

## Plotting social network data

This recipe covers the features in the `igraph` package to create graph objects, plot them, and extract network information from graph objects.

### Getting ready

If you have not already installed the `igraph` package, do it now using the following code:

```
> install.packages("igraph")
```

Also, download the `users_edgelist_upper.Rdata` file from the book's data files to your R working directory. Alternately, if you worked through the previous recipe *Creating adjacency matrices and edge lists* from this chapter, you will have created the file and ensured that it is in your R working directory.

### How to do it...

To plot social network data using `igraph`, follow these steps:

1. Load the data. The following code will restore, from the saved file, a data frame called `users.edgelist.upper`:

```
> load("users_edgelist_upper.Rdata")
```

2. The data file that we have provided `users.edgelist.upper` should now have 1953 rows of data. Plotting such a network will take too much time—even worse, we will get too dense a plot to get any meaningful information. Just for convenience, we will create a much smaller network by filtering our edge list. We will consider as connected only users who have more than 16 common group memberships and create a far smaller network for illustration only:

```
> edgelist.filtered <-
 users.edgelist.upper[users.edgelist.upper$x > 16,]
```

```
> edgelist.filtered # Your results could differ
```

|          | i       | j       | x  |
|----------|---------|---------|----|
| 34364988 | 2073657 | 3823125 | 17 |
| 41804209 | 2073657 | 4379102 | 18 |
| 53937250 | 2073657 | 5590181 | 17 |
| 62598651 | 3823125 | 6629901 | 18 |

```
190318039 5286367 13657677 17
190321739 8417076 13657677 17
205800861 5054895 14423171 18
252063744 5054895 33434002 18
252064197 5590181 33434002 17
252064967 6629901 33434002 18
252071701 10973799 33434002 17
252076384 13657677 33434002 17
254937514 5227777 34617262 17
282621070 5590181 46801552 19
282621870 6629901 46801552 18
282639752 33434002 46801552 20
307874358 33434002 56882992 17
335204492 33434002 69087262 17
486425803 33434002 147010712 17

> nrow(edgelist.filtered)

[1] 19
```

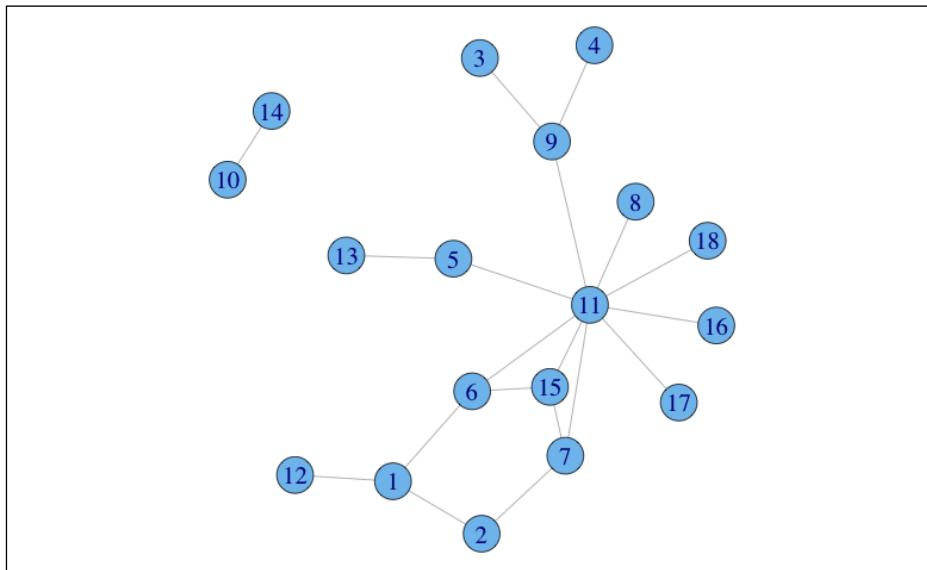
3. Renumber the users. Since we filtered the graph significantly, we have only 18 unique users left, but they retain their original user IDs. We will find it convenient to sequence them with unique numbers from 1 through 18. This step is not strictly needed, but will make the social network graph look cleaner:

```
> uids <- unique(c(edgelist.filtered$i, edgelist.filtered$j))
> i <- match(edgelist.filtered$i, uids)
> j <- match(edgelist.filtered$j, uids)
> nw.new <- data.frame(i, j, x = edgelist.filtered$x)
```

4. Create the graph object and plot the network:

```
> library(igraph)
> g <- graph.data.frame(nw.new, directed=FALSE)
> g
IGRAPH UN-- 18 19 --
+ attr: name (v/c), x (e/n)
> # Save the graph for use in later recipes:
> save(g, file = "undirected-graph.Rdata")
> plot.igraph(g, vertex.size = 20)
```

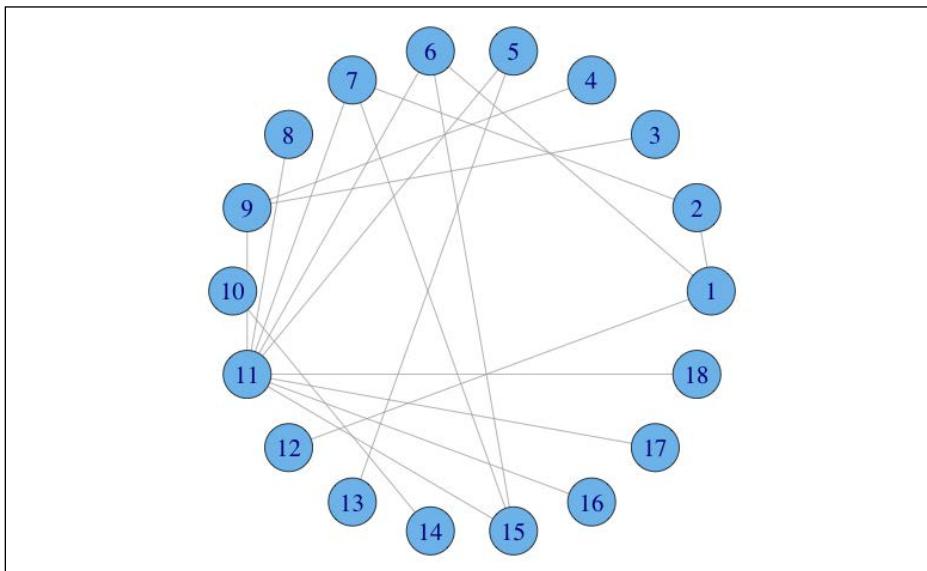
Your plot may look different in terms of layout, but if you look carefully, you will see that the nodes and edges are identical:



5. Plot the graph object with a different layout:

```
> plot.igraph(g, layout=layout.circle, vertex.size = 20)
```

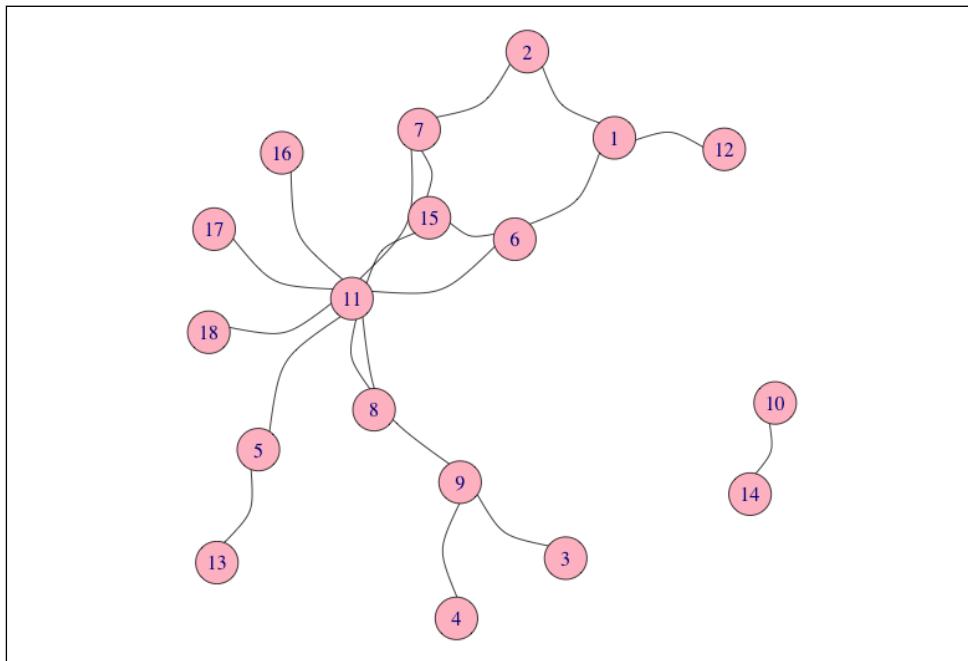
The following graph is the output of the preceding command:



6. Plot the graph object using colors for the vertices and edges:

```
> plot.igraph(g, edge.curved=TRUE, vertex.color="pink",
edge.color="black")
```

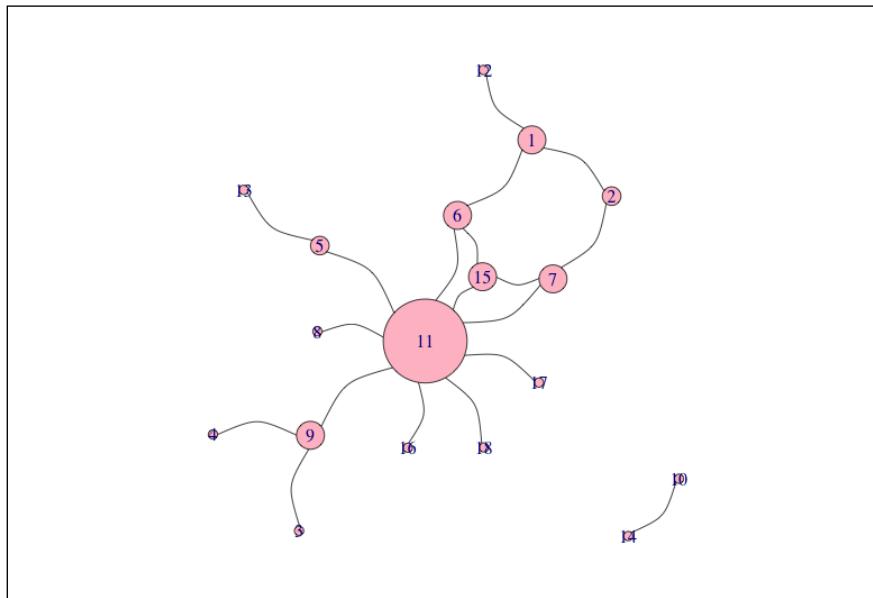
Again, your plot may be laid out differently, but should be functionally identical to the following plot:



7. Plot the graph with node size proportional to node degree:

```
> V(g)$size=degree(g) * 4
> plot.igraph(g, edge.curved=TRUE, vertex.color="pink",
edge.color="black")
```

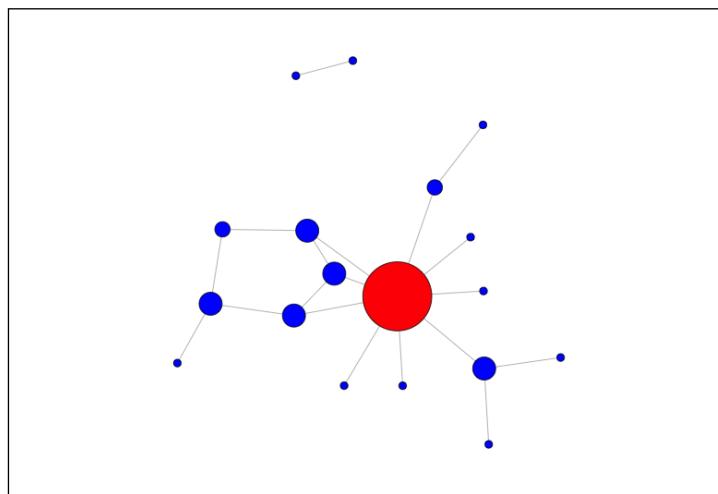
The output is similar to the following plot:



8. Plot the graph with the node size and color based on degree:

```
> color <- ifelse(degree(g) > 5, "red", "blue")
> size <- degree(g)*4
> plot.igraph(g,vertex.label=NA,layout= layout.fruchterman.
reingold,vertex.color=color,vertex.size=size)
```

A plot identical to the following will be obtained:



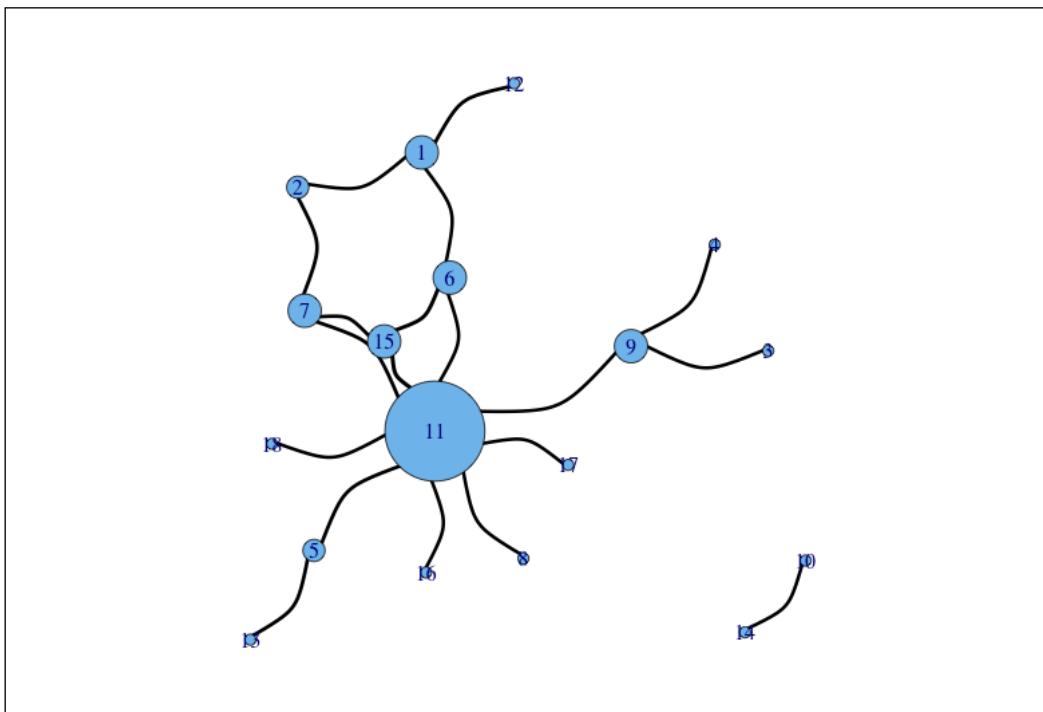
9. Plot the graph with the edge thickness proportional to edge weights:

```
> E(g)$x
```

```
[1] 17 18 17 18 17 17 18 18 17 18 17 17 17 17 19 18 20 17 17 17
```

```
> plot.igraph(g, edge.curved=TRUE, edge.color="black", edge.width=E(g)$x/5)
```

The output is similar to the following plot:



### How it works...

Step 1 loads the saved network data in the form of an edge list.

With so much data, we will not find a plot to be a useful visual aid because it will be too crowded. Thus, for illustration, we redefine two users (nodes) to be related (connected in the social network) only if they have more than 16 group memberships in common.

Step 2 filters the edge list and retains only edges that meet the preceding criterion.

Although we have filtered the data, users still retain their original IDs, which are big numbers. Having user numbers in sequence starting with 1 might be nice.

Step 3 does this conversion.

Step 4 uses the `graph.data.frame` function from the `igraph` package to create a graph object. The function treats the first two columns of the `nw.new` data frame argument as the edge list, and treats the rest of the columns as edge attributes.

We created an undirected graph by specifying `directed = FALSE`. Specifying `directed = TRUE` (or omitting this argument altogether, since `TRUE` is the default) will create a directed graph.

Here `g` is an *Undirected Named* graph represented by `UN`. A graph is treated as *named*, if the vertex has a `name` attribute. The third letter indicates if the graph is weighted (`w`), and the fourth letter indicates if it is a bipartite (`B`) graph. The two numbers indicate the number of vertices and the number of edges. The second line `+ attr: name (v/c) , x (e/n)` gives details about the attributes. The `name` attribute represents the vertex, and the attribute `x` represents the edge.

The step then uses the `plot.igraph` function from the `igraph` package to plot it.

We specified the `vertex.size` argument to ensure that the node circles were large enough to fit the node numbers.

Step 5 plots the very same graph but with the nodes laid out on the circumference of a circle.

Step 6 shows other options which should be self-explanatory.

In step 7, we set the node (or vertex) size. We use `V(g)` to access each vertex in the graph object and use the `$` operator to extract attributes, for example, `V(g)$size`. Similarly, we can use `E(g)` to access the edges.

Step 8 goes further by assigning node color and size based on a node's degree. It also shows the use of the `layout` option.

## There's more...

We can do much more with the graph object that the `graph.data.frame` function of the `igraph` package creates.

## Specifying plotting preferences

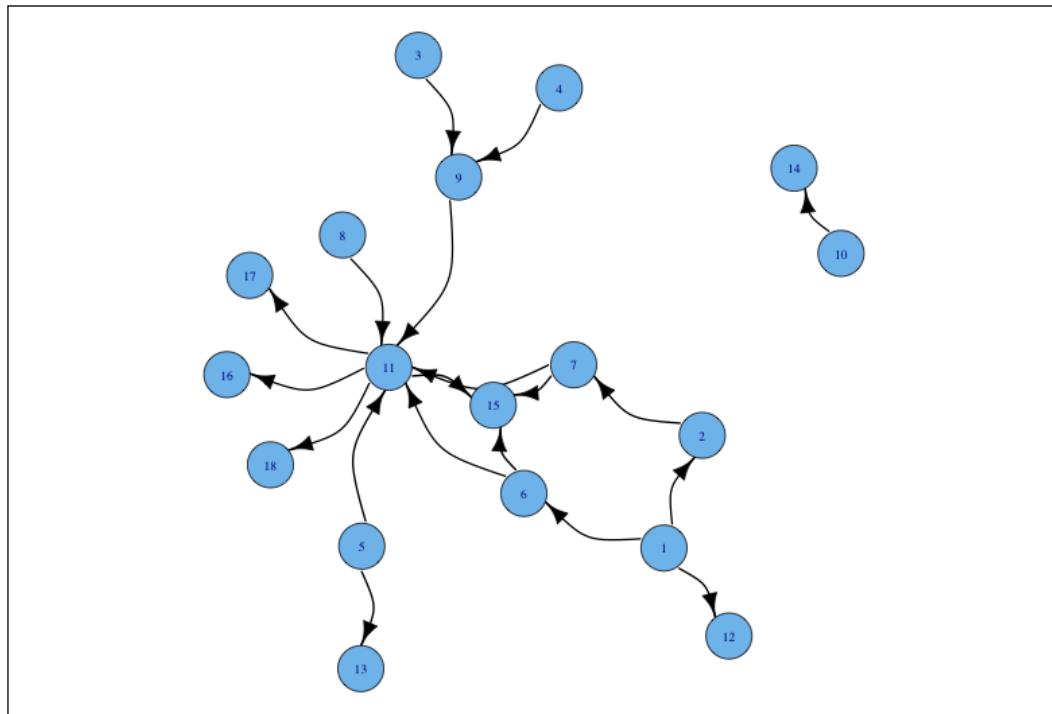
We showed only a few options to control the look of the plot. You have many more options for controlling the look of the edges, nodes, and other aspects. The documentation of `plot.igraph` mentions these options. When using them, remember to prefix the node options with `(vertex.)` and edge options with `(edge.)`.

## Plotting directed graphs

In step 4 of the main recipe, we created an undirected graph. Here, we create a directed graph object `dg`:

```
> dg <- graph.data.frame(nw.new)
> # save for later use
> save(dg, file = "directed-graph.Rdata")
> plot.igraph(dg, edge.curved=TRUE, edge.color="black", edge.
width=E(dg)$x/10, vertex.label.cex=.6)
```

On plotting the preceding graph, we get the following output:



### Creating a graph object with weights

If the name of the third column in the edge list passed to `graph.data.frame` is called `weight`, it creates a weighted graph. Hence, we can rename the third column from `x` to `weight` and redraw the graph:

```
> nw.weights <- nw.new
> names(nw.weights) <- c("i","j","weight")
> g.weights <- graph.data.frame(nw.weights, directed=FALSE)
> g.weights
IGRAPH UNW- 18 19 --
+ attr: name (v/c), weight (e/n)
```

When we check the graph properties, we see `w` in the third position of `UNW-`.

### Extracting the network as an adjacency matrix from the graph object

Earlier, we created an edge list from a sparse adjacency matrix. Here, we show how to get the sparse adjacency matrix from the graph object that we created in step 4 of the main recipe. In the following, we used `type="upper"` to get the upper triangular matrix. Other options are `lower` and `both`.

```
> get.adjacency(g,type="upper")

18 x 18 sparse Matrix of class "dgCMatrix"
[[suppressing 18 column names '1', '2', '3' ...]]

 1 . 1 . . . 1 1
 2 1
 3 1
 4 1
 5 1 . 1
 6 1 . . . 1 . .
 7 1 . . . 1 . .
 8 1
 9 1
10 1 . .
11 1 1 1 1
12
13
14
15
16
17
18
```

In the preceding code, we did not get back the weights and got back a 0-1 sparse matrix instead.

If we want the weights, we can implement the following techniques.

### Extracting an adjacency matrix with weights

The `graph.data.frame` function from the `igraph` package treats the first two columns of the data frame supplied as making up the edge list and the rest of the columns as edge attributes. By default, the `get.adjacency` function does not return any edge attributes and instead returns a simple 0-1 sparse matrix of connections.

However, you can pass the `attr` argument to tell the function which of the remaining attributes you want as the elements of the sparse matrix (and hence the edge weight). In our situation, this attribute will be `x`, representing the number of common group memberships between two users. In the following, we have specified `type = "lower"` to get the lower triangular matrix. Other options are `upper` and `both`.

```
> get.adjacency(g, type = "lower", attr = "x")

18 x 18 sparse Matrix of class "dgCMatrix"
[[suppressing 18 column names '1', '2', '3' ...]]

 1
 2 17
 3
 4
 5
 6 17
 7 . 18
 8
 9 . . 17 17
10
11 18 17 18 17 17
12 18
13 18
14 17
15 19 18 . . . 20
16 17
17 17
18 17
```

## **Extracting edge list from graph object**

You can use the `get.data.frame` on an `igraph` object, to get the edge list:

```
> y <- get.data.frame(g)
```

Use this to get only the vertices:

```
> y <- get.data.frame(g, "vertices")
```

## **Creating bipartite network graph**

Say we have a set of groups and a set of users with each user belonging to several groups and each group potentially has several members.

We can represent this information as a bipartite graph with the groups forming one set, the users forming the other, and edges linking members of one set to the other. You can use the `graph.incidence` function from the `igraph` package to create and visualize this network:

```
> set.seed(2015)
> g1 <- rbinom(10,1,.5)
> g2 <- rbinom(10,1,.5)
> g3 <- rbinom(10,1,.5)
> g4 <- rbinom(10,1,.5)
> membership <- data.frame(g1, g2, g3, g4)
> names(membership)

[1] "g1" "g2" "g3" "g4"

> rownames(membership) = c("u1", "u2", "u3", "u4", "u5", "u6", "u7",
"u8", "u9", "u10")

> rownames(membership)

[1] "u1" "u2" "u3" "u4" "u5" "u6" "u7" "u8"
[9] "u9" "u10"

> # Create the bipartite graph through the
> # graph.incidence function
> bg <- graph.incidence(membership)
> bg

IGRAPH UN-B 14 17 --
+ attr: type (v/1), name (v/c)

> # The B above tells us that this is a bipartite graph
> # Explore bg
```

```
> V(bg)$type

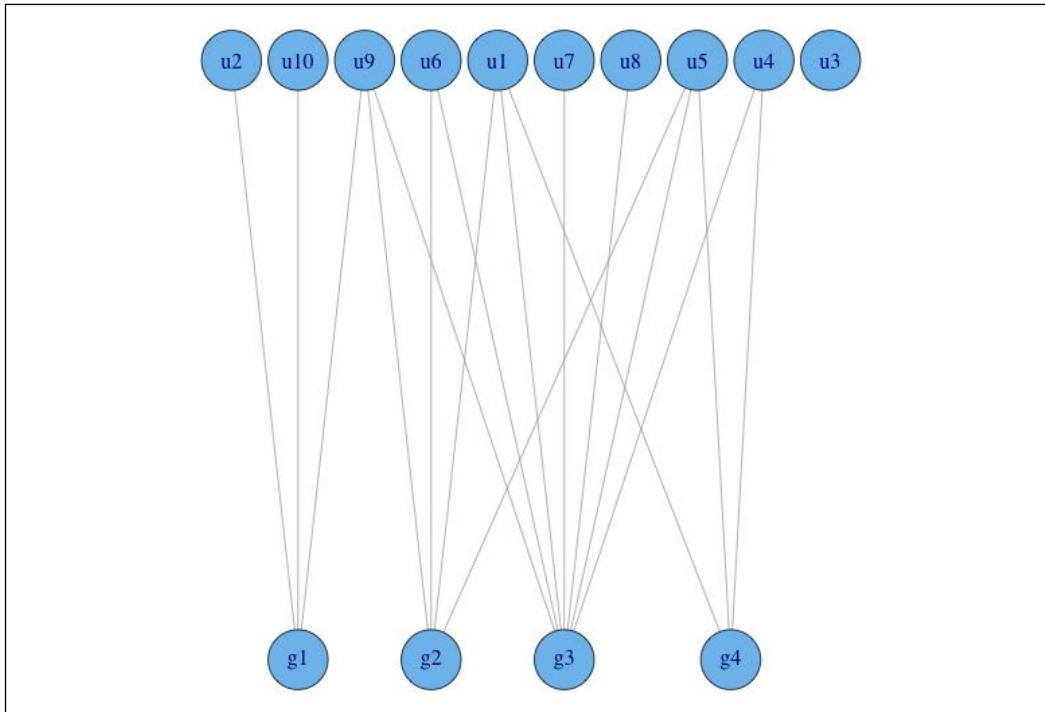
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[9] FALSE FALSE TRUE TRUE TRUE TRUE

> # FALSE represents the users and TRUE represents the groups
> # See node names
> V(bg)$name

[1] "u1" "u2" "u3" "u4" "u5" "u6" "u7" "u8"
[9] "u9" "u10" "g1" "g2" "g3" "g4"

> # create a layout
> lay <- layout.bipartite(bg)
> # plot it
> plot(bg, layout=lay, vertex.size = 20)
> # save for later use
> save(bg, file = "bipartite-graph.Rdata")
```

We created a random network of four groups and ten users which when plotted appears as follows:



## Generating projections of a bipartite network

Very often, we need to extract adjacency information about one or both types of nodes in a bipartite network. In the preceding example with users and groups, we may want to consider two users as related or connected if they have a common group membership and create a graph only of the users. Analogously, we may consider two groups as connected if they have at least one user in common. Use the `bipartite.projection` function to achieve this:

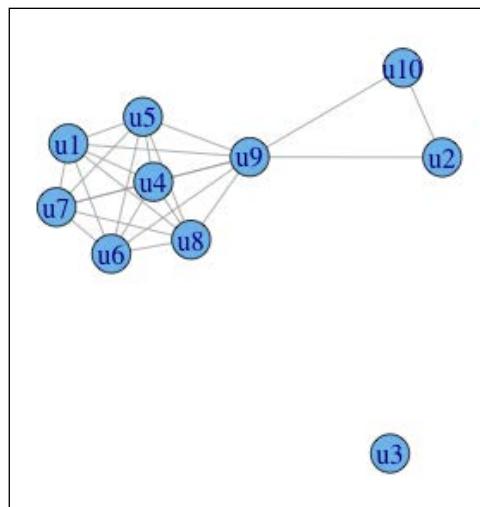
```
> # Generate the two projections
> p <- bipartite.projection(bg)
> p

$proj1
IGRAPH UNW- 10 24 --
+ attr: name (v/c), weight (e/n)

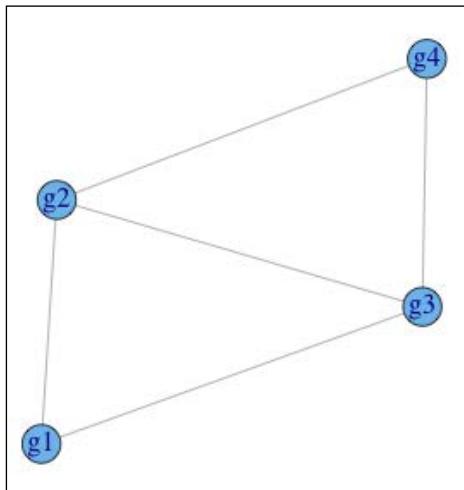
$proj2
IGRAPH UNW- 4 5 --
+ attr: name (v/c), weight (e/n)

> plot(p$proj1, vertex.size = 20)
> plot(p$proj2, vertex.size = 20)
```

The first projection is plotted as follows:



The next projection is generated as follows:



### See also...

- ▶ *Creating adjacency matrices and edge lists* from this chapter

## Computing important network metrics

This recipe covers the methods used to compute some of the common metrics used on social networks.

### Getting ready

If you have not yet installed the `igraph` package, do it now. If you worked through the earlier recipes in this chapter, you should have the data files `directed-graph.Rdata`, `undirected-graph.Rdata`, and `bipartite-graph.Rdata` and should ensure that they are in your R working directory. If not, you should download these data files and place them in your R working directory.

## How to do it...

To compute important network metrics, follow these steps:

1. Load the data files:

```
> load("undirected-graph.Rdata")
> load("directed-graph.Rdata")
> load("bipartite-graph.Rdata")
```

2. The degree centrality can be measured as follows:

```
> degree(dg)
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
3 2 1 1 2 3 3 1 3 1 9 1 1 1 3 1 1 1
```

```
> degree(g)
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
3 2 1 1 2 3 3 1 3 1 9 1 1 1 3 1 1 1
```

```
> degree(dg, "7")
```

```
7
3
```

```
> degree(dg, 9, mode = "in")
```

```
9
2
```

```
> degree(dg, 9, mode = "out")
```

```
9
1
```

```
> # Proportion of vertices with degree 0, 1, 2, etc.
```

```
> options(digits=3)
```

```
> degree.distribution(bg)
```

```
[1] 0.0714 0.2857 0.1429 0.3571
```

```
[5] 0.0714 0.0000 0.0000 0.0714
```

3. The betweenness centrality can be measured as follows:

```
> betweenness(dg)

 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
 0 1 0 0 0 5 5 0 10 0 32 0 0 0 0 0 0 0 0

> betweenness(g)

 1 2 3 4 5 6 7 8 9 10 11 12
15.0 2.0 0.0 0.0 14.0 22.0 11.0 0.0 27.0 0.0 86.5 0.0
 13 14 15 16 17 18
 0.0 0.0 0.5 0.0 0.0 0.0

> betweenness(dg, 5)

5
0

> edge.betweenness(dg)

[1] 2 1 6 7 6 6 1 5 8 8 5 15 1 2 2 6 10 10
10

> edge.betweenness(dg, 10)

[1] 8
```

4. The closeness centrality can be measured as follows:

```
> options(digits=3)
> closeness(dg, mode="in")

 1 2 3 4 5 6
0.00327 0.00346 0.00327 0.00327 0.00327 0.00346
 7 8 9 10 11 12
0.00366 0.00327 0.00368 0.00327 0.00637 0.00346
 13 14 15 16 17 18
0.00346 0.00346 0.00690 0.00671 0.00671 0.00671

> closeness(dg, mode="out")

 1 2 3 4 5 6
0.00617 0.00472 0.00469 0.00469 0.00481 0.00446
 7 8 9 10 11 12
```

```
0.00446 0.00444 0.00444 0.00346 0.00420 0.00327
13 14 15 16 17 18
0.00327 0.00327 0.00327 0.00327 0.00327 0.00327

> closeness(dg, mode="all")

1 2 3 4 5 6
0.01333 0.01316 0.01220 0.01220 0.01429 0.01515
7 8 9 10 11 12
0.01493 0.01389 0.01471 0.00346 0.01724 0.01124
13 14 15 16 17 18
0.01190 0.00346 0.01493 0.01389 0.01389 0.01389

> closeness(dg)

1 2 3 4 5 6
0.00617 0.00472 0.00469 0.00469 0.00481 0.00446
7 8 9 10 11 12
0.00446 0.00444 0.00444 0.00346 0.00420 0.00327
13 14 15 16 17 18
0.00327 0.00327 0.00327 0.00327 0.00327 0.00327
```

## How it works...

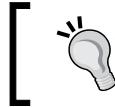
Step 1 computes various metrics dealing with the degree of the vertices in a graph. **Degree** measures the number of edges connected to a vertex or node. Degree distribution provides the frequency of all degree measures up to the maximum degree. For an undirected graph, the degree is always the total adjacent edges. However, for a directed graph, the degree depends on the mode argument passed. Mode can be out, in, all, or total. Both all and total return the total degree of that node or vertex. You should be able to verify some of the numbers from the plots provided earlier.

**Centrality** determines how individual nodes fit within a network. High centrality nodes are influencers: positive and negative. There are different types of centrality, and we discuss a few important ones here.

Step 2, computes **betweenness**, which quantifies the number of times a node falls in the shortest path between two other nodes. Nodes with high betweenness sit between two different clusters. To travel from any node in one cluster to any other node in the second cluster, one will likely need to travel through this particular node.

Edge betweenness computes the number of shortest paths through a particular edge. If a graph includes the weight attribute, then it is used by default. In our example, we have an attribute x. You will get different results if you rename the column to weight.

Step 3 computes **closeness**, which quantifies the extent to which a node is close to other nodes. It is a measure of the total distance from a node to all others. A node with high closeness has easy access to many other nodes. In a directed graph, if mode is not specified, then out is the default. In an undirected graph, mode does not play any role and is ignored.



Closeness measures the extent of access a node has to others, and betweenness measures the extent to which a node acts as an intermediary.



## There's more...

We show a few additional options to work on the graph objects.

### Getting edge sequences

You can look at the edges—the connections—in the figure showing the directed graph in *Plotting directed graphs*. You can identify edges as `E(1) ... E(19)`:

```
> E(dg)
Edge sequence:
```

```
[1] 1 -> 2
[2] 1 -> 12
[3] 1 -> 6
[4] 2 -> 7
[5] 3 -> 9
[6] 4 -> 9
[7] 5 -> 13
[8] 5 -> 11
[9] 6 -> 11
[10] 7 -> 11
[11] 8 -> 11
[12] 9 -> 11
[13] 10 -> 14
[14] 6 -> 15
[15] 7 -> 15
[16] 11 -> 15
[17] 11 -> 16
[18] 11 -> 17
[19] 11 -> 18
```

## Getting immediate and distant neighbors

The `neighbors` function lists the neighbors of a given node (excluding itself):

```
> neighbors(g, 1)
[1] 2 6 12
> neighbors(bg, "u1")
[1] 12 13 14
> # for a bipartite graph, refer to nodes by node name and
> # get results also as node names
> V(bg)$name[neighbors(bg, "g1")]
[1] "u2" "u9" "u10"
```

The `neighborhood` function gets the list of neighbors lying at most a specified distance from a given node or a set of nodes. The node in question is always included in the list since it is of distance 0 from itself:

```
> #immediate neighbors of node 1
> neighborhood(dg, 1, 1)
[[1]]
[1] 1 2 6 12

> neighborhood(dg, 2, 1)
[[1]]
[1] 1 2 6 12 7 11 15
```

## Adding vertices or nodes

We can add nodes to an existing graph object:

```
> #Add a new vertex
> g.new <- g + vertex(19)
> # Add 2 new vertices
> g.new <- g + vertices(19, 20)
```

## Adding edges

If we need to add a new relationship between nodes 15 and 20, we can do the following:

```
> g.new <- g.new + edge(15, 20)
```

## Deleting isolates from a graph

Isolated nodes have no connections or edges and therefore have degree 0. We can use this to select vertices that have 0 degree and delete them using `delete.vertices` as follows:

```
> g.new <- delete.vertices(g.new, V(g.new) [degree(g.new) == 0])
```

We can also use `delete.vertices` to delete a specific vertex. This function creates a new graph. If the vertex does not exist in the graph, you will see an error message `Invalid vertex names`. Plot the new graph to check if the isolated vertex has been removed:

```
> g.new <- delete.vertices(g,new,12)
```

Deletion reassigns the vertex IDs even in some cases when edges are deleted. Hence, if you are using IDs instead of vertex names, exercise caution as the IDs may change.

## Creating subgraphs

You can create new graphs by selecting the vertices you are interested in using the following code:

```
> g.sub <- induced.subgraph(g, c(5, 10, 13, 14, 17, 11, 7))
```

You can also create new graphs by selecting the edges you are interested in using the following code:

```
> E(dg)
```

Edge sequence:

```
[1] 1 -> 2
[2] 1 -> 12
[3] 1 -> 6
[4] 2 -> 7
[5] 3 -> 9
[6] 4 -> 9
[7] 5 -> 13
[8] 5 -> 11
[9] 6 -> 11
[10] 7 -> 11
[11] 8 -> 11
[12] 9 -> 11
[13] 10 -> 14
[14] 6 -> 15
[15] 7 -> 15
[16] 11 -> 15
[17] 11 -> 16
[18] 11 -> 17
[19] 11 -> 18
```

```
> eids <- c(1:2, 9:15)
> dg.sub <- subgraph.edges(dg, eids)
```



# 8

## **Put Your Best Foot Forward – Document and Present Your Analysis**

In this chapter, you will cover:

- ▶ Generating reports of your data analysis with R Markdown and knitr
- ▶ Creating interactive web applications with shiny
- ▶ Creating PDF presentations of your analysis with R Presentation

### **Introduction**

Other than helping us analyze data, R has libraries that help you with professional presentations as well. You can perform the following tasks:

- ▶ Create professional web pages that showcase your analysis and allow others to actively experiment with the underlying data
- ▶ Generate PDF reports of your analysis; your report can include embedded R commands for the system to execute and fill live data and charts so that, when the data changes, you can regenerate the report with a single button click
- ▶ Generate PDF presentations of your analysis

This chapter provides recipes for you to exploit all of these capabilities.

## **Generating reports of your data analysis with R Markdown and knitr**

R Markdown provides a simple syntax to define analysis reports. Based on such a report definition, `knitr` can generate reports in HTML, PDF, Microsoft Word format, and several presentation formats. R Markdown documents contain regular text, embedded R code chunks, and inline R code. `knitr` parses the markdown document and inserts the results of executing the R code at specified locations within regular text to produce a well-formatted report.

R Markdown extends the regular markdown format to enable us to embed R code.

We can create R Markdown documents either in RStudio or directly in R using the `markdown` package. In this recipe, we describe the RStudio approach.

### **Getting ready**

If you have not already downloaded the files for this chapter, do it now and place the `auto-mpg.csv` and `knitr.Rmd` files in a known location (this need not necessarily be the working directory of your R installation).

Install the latest version of the `knitr` and `rmarkdown` packages:

```
> install.packages("knitr")
> install.packages("rmarkdown")
```

### **How to do it...**

To generate reports using `rmarkdown` and `knitr`, follow these steps:

1. Open RStudio.
2. Create a new R Markdown document as follows:
  1. Select the menu option by navigating to **File | New File | R Markdown**.
  2. Enter the title as "Introduction", leave the other defaults as is, and click on **OK**.

This generates a sample R Markdown document that we can edit to suit our needs. The sample document resembles the following screenshot:

```
1 ---
2 title: "Introduction"
3 author: "Shanthi Viswanathan"
4 date: "March 21, 2015"
5 output: html_document
6 ---
7
8 This is an R Markdown document. Markdown is a simple formatting
9 syntax for authoring HTML, PDF, and MS Word documents. For more
10 details on using R Markdown see <http://rmarkdown.rstudio.com>.
11
12 When you click the **Knit** button a document will be generated that
13 includes both content as well as the output of any embedded R code
14 chunks within the document. You can embed an R code chunk like this:
15
16 You can also embed plots, for example:
17
18 Note that the `echo = FALSE` parameter was added to the code chunk
19 to prevent printing of the R code that generated the plot.
20
21
22
23
```

3. Take a quick look at the document. You do not need to understand everything in it. In this step, we are just trying to get an overview.
4. Generate an HTML document based on the markdown file. Depending on the width of your editing pane, you may either see just the knitr icon (a blue bale of wool and a knitting needle) with a downward-facing arrow or the icon and the text **Knit HTML** beside it. If you only see the icon, click on the downward arrow beside the icon and select **Knit HTML**. If you see the text in addition to the icon, just click on **Knit HTML** to generate the HTML document. RStudio may render the report in a separate window or in the top pane on the right side. The menu that you used to generate HTML has options to control where RStudio will render the report—choose either **View in pane** or **View in window**.
5. With the same file, you can generate a PDF or Word document by invoking the appropriate menu option. To generate a Word document, you need to have Microsoft Word installed on your system and, to generate a PDF, you need to have the Latex PDF generator *pdflatex* installed. Note that the output item in the metadata changes according to the output format you choose from the menu.

6. Now that you have an idea of the process, use the menu option by navigating to **File** | **Open file** to open the knitr.Rmd file. Before proceeding further, edit line 40 of the file and change the `root.dir` location to wherever you downloaded the files for this chapter. For ease of discussion, we show the output incrementally.
7. The metadata section is between two lines, each with just three hyphens, shown as follows:

```

```

```
title: "Markdown Document"
author: "Shanthi Viswanathan"
date: "December 8, 2014"
output:
 html_document:
 theme: cosmo
 toc: yes

```

# Markdown Document

*Shanthi Viswanathan*

*December 8, 2014*

- [Introduction](#)
- [HTML Content](#)
- [Embed Code](#)
  - [Set Directory](#)
  - [Load data](#)
    - [Plot Data](#)
    - [Plot with format options](#)

8. The Introduction section of the R Markdown document appears as follows:

```
* * *
Introduction
This is an *R Markdown document*. Markdown is a simple formatting
syntax for authoring HTML, PDF, and MS Word documents. For more
details on using R Markdown see <http://rmarkdown.rstudio.com>.
```

When you click the **\*\*Knit\*\*** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document.

This is also seen in the following screenshot:

## Introduction

This is an *R Markdown document*. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document.

9. The HTML content of the document is as follows:

```
#HTML Content
<p> This is a new paragraph written with the HTML tag
<table border=1>
<th> Pros </th>
<th> Cons </td>
<tr>
<td>Easy to use</td>
<td>Need to Plan ahead </td>
<tr>
</table>
<hr/>
```

In the document it appears as follows:

## HTML Content

This is a new paragraph written with the HTML tag

Pros	Cons
Easy to use	Need to Plan ahead

10. Embed the R Code. Change the following `root.dir` path to the folder where you stored the `auto-mpg.csv` and `knitr.Rmd` files:

```
Embed Code
Set Directory
```

You can embed any R code chunk within 3 ticks. If you add `echo=FALSE` the code chunk is not displayed in the document. We can set knitr options either globally or within a code segment. The options set globally are used throughout the document.

We set the `root.dir` before loading any files. By enabling `cache=TRUE`, a code chunk is executed only when there is a change from the prior execution. This enhances knitr performance.

```
```{r setup, echo=FALSE, message=FALSE, warning=FALSE}  
knitr::opts_chunk$set(cache=TRUE)  
knitr::opts_knit$set(root.dir = "/Users/shanthiviswanathan/  
projects/RCookbook/chapter8/")  
```
```

This can be seen in the following screenshot:

## Embed Code

### Set Directory

You can embed any R code chunk within 3 ticks. If you add `echo=FALSE` the code chunk is not displayed in the document. We can set knitr options either globally or within a code segment. The options set globally are used throughout the document.

We set the `root.dir` before loading any files. By enabling `cache=TRUE`, a code chunk is executed only when there is a change from the prior execution. This enhances knitr performance.

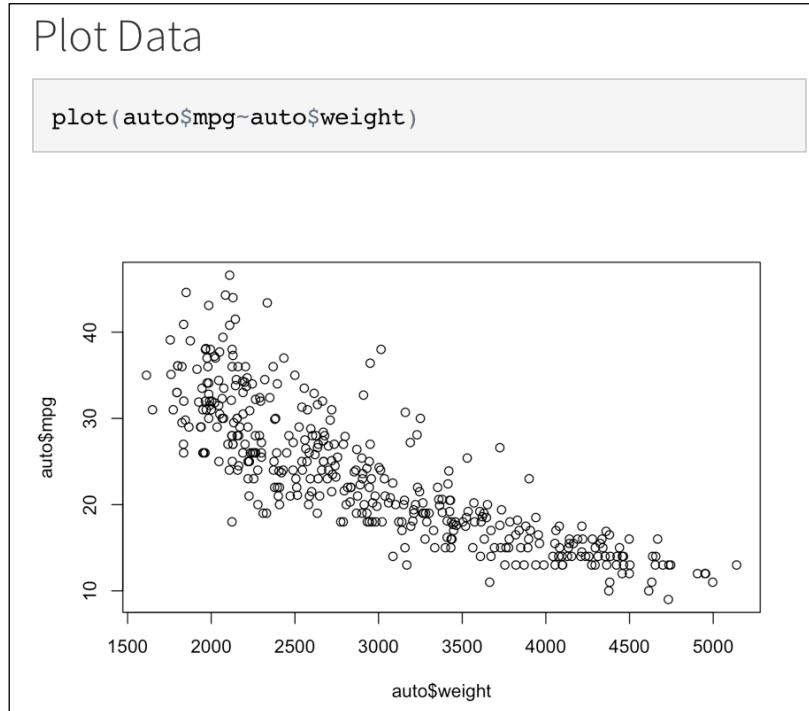
11. Load the data:

```
##Load Data
```{r loadData, echo=FALSE}  
auto <- read.csv("auto-mpg.csv")  
```
```

12. Plot the data:

```
```{r plotData }
plot(auto$mpg~auto$weight)
```
```

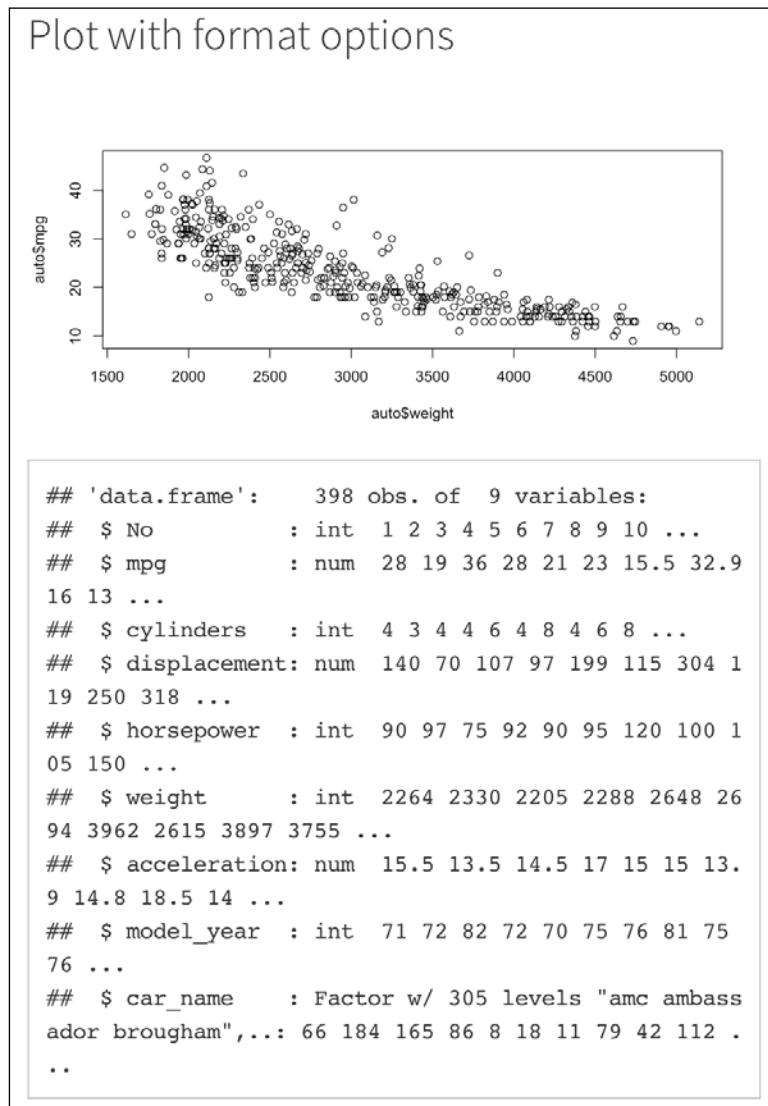
The output of the preceding command can be seen here:



13. Plot with the format options:

```
```{r plotFormatData, echo=FALSE, fig.height=4, fig.width=8}
plot(auto$mpg~auto$weight)
str(auto)
```
```

The following screenshot shows the plotted output:



14. Embed the code within a sentence:

There are `r nrow(auto)` cars in the auto data set.

Here's the output of the preceding command:

There are 398 cars in the auto data set.

## How it works...

Step 1 opens RStudio.

Step 2 creates a new R Markdown document. A new document includes a default metadata section between lines containing three dashes. This metadata section includes the title and output sections and can optionally also specify the author and date.

Step 4 shows you how to generate the HTML document. If running in RStudio, you can indicate the desired output format by selecting the appropriate menu option. However, it is possible to run `knitr` within a standard R environment; in this case, the output specified in the markdown document determines the format of the output document.

Step 5 shows you how to generate a PDF or Word document based on the markdown document.

Step 6 opens a precreated document that illustrates many of the important features of `knitr`. We explain the code in parts here.

Step 7 contains the metadata of the document:

- ▶ There are three output types: Word, PDF, and HTML.
- ▶ The initial three hyphens indicate the start of the metadata section.
- ▶ `toc: TRUE` causes the table of contents to be generated based on the headings in the document. This is explained in the next section.
- ▶ The final three hyphens end the metadata section.

Step 8 contains the Introduction section of our sample document:

- ▶ The three asterisks on a line by itself cause a horizontal line to be output.
- ▶ Lines starting with a single # signify a first-level heading and, if `toc: TRUE` is set in the metadata, it is added in the table of contents. Lines starting with ## signify second-level headings.
- ▶ Text surrounded by a single asterisk displays as italic, and text surrounded by double asterisks displays as bold.
- ▶ Text starting with `<http` and ending with `>` is displayed as a URL.
- ▶ Checkout *There's more...* for the most commonly used syntax.

Step 9 includes the HTML content of our sample document:

- ▶ Regular HTML coding can be embedded in a R Markdown document. `knitr` will only properly display the HTML if the output format is set in `HTML`; you must leave an empty line before starting the HTML code.
- ▶ In this segment, we used the HTML table syntax to produce a table.

Step 10 shows how to embed R code in a markdown document:

- ▶ R code fragments or chunks begin on new lines with three back quotes (` `` `) at the start. These chunks end with a line containing just three back quotes. The text of the R code segment starts with `x`, followed by an optional name for the chunk (we can choose any unique name).
- ▶ We recommend that you do not mix the code for `knitr` settings with regular R code in a single chunk. Keep them in separate chunks. In this step, we set `cache=TRUE` and also set the home directory for `knitr`. The `knitr` options set here apply to the whole document. Thus, `cache` is enabled for each R code chunk that follows this setting.
- ▶ We set up display options for the current code chunk. If `echo=FALSE`, then the code chunk is not displayed in the document. Similarly, `message=FALSE` and `warning=FALSE` suppress any R messages and warnings in the document.

Step 11 loads data from a file:

- ▶ We show a code chunk named `loadData` to read a `.csv` file into a variable. This code chunk does not appear in the report because we have chosen `echo=FALSE`.
- ▶ The location of the file is taken from the directory that we set in the earlier step. Also, since we have enabled `cache`, the file is not read each time the document is generated.

Steps 12 and 13 plot data:

- ▶ We create a code chunk called `plotData`. The R code appears in the report because the default value for `echo` is `TRUE`. If an R code chunk produces any output, `knitr` automatically includes that output in the generated report.

Step 14 illustrates how to embed R code in-line:

- ▶ We enclose the R code between a set of single back quotes. `knitr` substitutes the output of the R command in place of the in line R code.
- ▶ Thus, `nrow(auto)` returns the number of autos and is included in the generated document.

### There's more...

The following is a list of the most commonly used markdown syntax elements.

See <http://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf> for a complete list of markdown syntax elements:

| Option                   | Syntax                                   | Remarks                                                                                    |
|--------------------------|------------------------------------------|--------------------------------------------------------------------------------------------|
| Italics                  | *text*                                   |                                                                                            |
| Bold                     | **text**                                 |                                                                                            |
| New Paragraph            | Leave 2 spaces after the end of the line |                                                                                            |
| Header1-6                | # text, ## text, ### text, and so on     | Headers display the word in an appropriate font                                            |
| Horizontal Rule <hr>     | ***                                      | Draws a horizontal line                                                                    |
| Unordered List (*, +, -) | * List Item 1                            | Note that the space between * and the list item + and – can also be used                   |
| Unordered SubItem        | + sub item 1                             | Use an indented + to create a sublist or an indented – or an indented * to create sublists |
| Ordered list             | 1. List item<br>1. Another List item     | Even for the ordered list, the indented + is used to create subitems                       |

The following table shows the various display options in a code chunk:

| Option  | Description                         | Possible Values                           |
|---------|-------------------------------------|-------------------------------------------|
| eval    | Evaluate the code in the code chunk | TRUE, FALSE; default: TRUE                |
| echo    | Display code along with the output  | TRUE, FALSE; default: TRUE                |
| warning | Display warning messages            | TRUE, FALSE; default: TRUE                |
| error   | Display errors                      | TRUE, FALSE; default: FALSE               |
| message | Display R messages                  | TRUE, FALSE; default: TRUE                |
| results | Display results                     | markup, asis, hold, hide; default: markup |
| cache   | Cache the results                   | TRUE, FALSE; default: FALSE               |

## Using the render function

In RStudio, document output can be generated using `knitr` by clicking on the knit button. You can also directly enter a command in the R command line. If you leave out the second argument, then the output specification in the markdown document determines the output format:

```
markdown::render("introduction.Rmd", "pdf_document") .
```

To create the output in all formats mentioned in the markdown document, use the following command:

```
rmarkdown::render("introduction.Rmd", "all")
```

## **Adding output options**

The following output options can be added:

- ▶ Type of output document to build:
  - output:html\_document, output:pdf\_document, output:beamer\_presentation, output:ioslides\_presentation, output:word\_document
- ▶ Number the section headings. If the sections are not named, then they are incrementally numbered:
  - number\_sections = TRUE
- ▶ The fig\_width, fig\_height options are the default width and height in inches
  - figures:fig\_width=7, fig\_height=5
- ▶ Theme: Visual theme; pass null to use custom CSS
- ▶ CSS: Include filename

## **Creating interactive web applications with shiny**

The shiny package helps in building interactive web applications using R. This recipe illustrates the main components of a shiny application through examples.

### **Getting ready**

Download the files for this chapter and store them in your R working directory. The code for this chapter contains files in various subfolders (named DummyApp, SimpleApp, TabApp, ConditionalApp, and SingleFileApp). Copy these folders into your R working directory.

Install and load the shiny package as follows:

```
> install.packages("shiny")
```

Restart RStudio after installing shiny.

## How to do it...

To create interactive web applications with shiny, follow the steps below:

1. Get a feel for shiny by examining a dummy application with no functionality. The folder called DummyApp in your R working directory contains the ui.R and server.R files with the following code:

```
ui.R
library(shiny)
shinyUI(pageWithSidebar(
 headerPanel("Dummy Application"),
 sidebarPanel(h3('Sidebar text')),
 mainPanel(h3('Main Panel text'))))
```

```
#server.R
library(shiny)
shinyServer(function(input,output) { })
```

Run the application using runApp ("DummyApp"). If you have the files loaded in the code pane in RStudio, click on **Run App**.

2. The SimpleApp directory in your R working directory contains the files ui.R and server.R with the following code:

```
ui.R
library(shiny)
shinyUI(fluidPage(
 titlePanel("Simple Shiny Application"),
 sidebarLayout(
 sidebarPanel(
 p("Create plots using the auto data"),
 selectInput("x", "Select X axis",
 choices = c("weight","cylinders","acceleration"))
),
 mainPanel(
 h4(textOutput("outputString")),
 plotOutput("autoplot"))
)))
))
```

```
server.R
auto <- read.csv("auto-mpg.csv")
shinyServer(function(input, output) {
 output$outputString <- renderText(paste("mpg ~",
```

```
 input$x))
 output$autoplot <- renderPlot(
 plot(as.formula(paste("mpg ~", input$x)), data=auto))
)}
```

3. Enter the `runApp ("SimpleApp")` command or, if the preceding files are loaded in the code pane, click on **Run App** to run the shiny application in the RStudio environment. The application opens in a separate window. When the application is running, RStudio cannot execute any command. Either close the application window, or press the `Esc` key in RStudio to exit the application.

## How it works...

A shiny application typically includes a folder with the `ui.R` and `server.R` files. The code in `ui.R` controls the user interface, and `server.R` controls the data that the application renders on the user interface as well as how the application responds to user actions on the interface.

In step 1, the dummy application presents a simple static user interface with no scope for user interaction.

The `shinyUI` function of `ui.R` constructs the user interface. In `DummyApp`, the function constructs the user interface with three static elements. It uses the `pageWithSidebar` function to create a page with static text in `headerPanel`, `sidebarPanel`, and `mainPanel`.

The `shinyServer` function in the `server.R` file controls how the application responds to user actions. This function represents the listeners on the server side. For every user action, the `shinyServer` function gets the relevant values from the user interface. The relevant parts of the server's listener get executed and send the output back to the user interface, which then updates the screen with the new values. The reactivity of the shiny package is explained as follows. Since `DummyApp` has no elements with which a user can actually interact and also has an empty `shinyServer` function, it cannot respond to user actions.

Step 2 builds a simple application in the `SimpleApp` folder. This application showcases the elements of a reactive application—one where the application truly reacts to user actions on the user interface. shiny has functions to generate static `html` as well as `html` code for user interface widgets, such as buttons, checkboxes, and drop-down lists, among others. The `ui.R` file in `SimpleApp` shows the use of the `p` function to add an `HTML` paragraph, the `selectInput` function to create a drop-down listbox, the `textOutput` and `h4` function to create a level 4 heading text, and the `plotOutput` function to plot the output image from the server.

shiny has several layout options to customize the look and feel of the user interface. In this recipe, we used `fluidPage` with three panels: `titlePanel`, `sidebarPanel`, and `mainPanel`.

shiny uses *reactive-programming*. A user input—such as the user entering text, selecting an item from a list, or clicking on a button—is a reactive source. A server output, such as a plot or data table, is a reactive endpoint that appears on the user's browser window. Whenever a reactive source changes, the reactive end point that uses the source is notified to re-execute. In this recipe, both `renderText` and `renderPlot` are reactive. `renderText` depends on input `x`, which means that `renderText()` is executed each time the user selects a different `x`. `renderPlot` depends on both `x` and the color and hence any change to either of these two input values causes `renderPlot` to be invoked.

When you run the application, the preceding statements `shinyServer(function(input, output)` in `server.R` are executed just once during the first application load. After this, only the relevant portions of the listener function execute for each change in the user interface.

We loaded the `auto-mpg.csv` data file here once for the application. We typically load packages, data, and dependent R source files once during the initial load of the application.

## There's more...

For a complete tutorial on shiny, refer to <http://shiny.rstudio.com/tutorial>. We provide a few key additions relating to building shiny web applications here.

### Adding images

To add images in the user interface, save the image file in the `www` directory under the application folder. Include the saved image file in `ui.R` with height and width in pixels as follows:

```
img(src = "myappimage.png", height = 72, width = 72)
```

The `css`, `javascript`, and `jquery` files are all stored in the `www` folder.

### Adding HTML

shiny provides R functions for several HTML markup tags. We have seen a sample of the R function `h3("text here")` in our first dummy application. Similarly, there are R functions such as `p()`, `h1()` to include paragraph, header 1, and so on, in a shiny application.

### Adding tab sets

Tabs are created by the `tabPanel()` function and each tab can hold its own output UI components. The `TabApp` folder has the `ui.R` and `server.R` files for a tabbed user interface. We give the main excerpts from each in the following code:

```
Excerpt from ui.R
mainPanel(
 tabsetPanel(
 tabPanel("Plot", textOutput("outputString"),
```

```
 plotOutput("plot")),
 tabPanel("Summary", verbatimTextOutput("summary")),
 tabPanel("Table", tableOutput("table")),
 tabPanel("DataTable", dataTableOutput("datatable"))
)
)
```

Add functionality in `server.R` to get the summary, table, and the `data.table` output:

```
Excerpt from server.R to generate a summary of the data
output$summary <- renderPrint({
 summary(auto)
})

Generate an HTML table view of the data
output$table <- renderTable({
 data.frame(x=auto)
})

Generate an HTML table view of the data
output$datatable <- renderDataTable({
 auto
}, options = list(aLengthMenu = c(5, 25, 50), iDisplayLength = 5))
})
```

The `options` argument in `renderDataTable` expects a list. The preceding code specifies the items in the list and the number of items to display in the drop-down box.

Run the application with `runApp ("TabApp")` to see the tabs and tab contents. If not all items are visible in the "table" tab, increase the size of the browser window.

## **Adding a dynamic UI**

You can create dynamic user interfaces in two different ways: using `conditionalPanel` or `renderUI`. We show an example of each in this section.

We use `conditionalPanel` to show or hide a UI component based on a condition. In this sample, we draw a histogram or scatterplot of `mpg` based on user selection. For the scatterplot, we fix `mpg` on the y axis and allow the user to pick a variable for the x axis. Hence, we need to show the list of possible variables for the x axis only when the user chooses the scatterplot option. In `ui.R`, we check for the condition with `input.plotType != 'hist'` and then display the list of choices to the user:

```
sidebarPanel(
 selectInput("plotType", "Plot Type",
 c("Scatter plot" = "scatter", Histogram = "hist")),
```

```
conditionalPanel(condition="input.plotType != 'hist'",
 selectInput("xaxis", "X Axis Variable",
 choices = c(Weight="wt", Cylinders="cyl", "Horse Power"="hp"))
),
mainPanel(plotOutput("plot"))
```

To check how `conditionalPanel` works, execute `runApp ("conditionalApp")` in your R environment and select the plot type. You will see "X Axis Variable" only when you choose scatter plot. In the case of `conditionalPanel`, the entire work is done in `ui.R` and is executed by the client.

In the preceding example, we had a fixed set of choices for the variables and hence we could build `selectInput` with these choices. What if this list is not known and is dependent on the user's selection of a dataset? The application in the `renderUIApp` folder illustrates this. In this application, the server builds the list of variable names dynamically based on the dataset chosen.

In the UI component, we need a placeholder, `uiOutput ("var")`, to display the list that will be populated by the server every time a different dataset is chosen in the user interface.

In the server component, we use the `renderUI` function call to populate `output$var` using the variable names of the chosen dataset. In this sample, we also use a reactive expression as given here:

```
datasetInput <- reactive({
 switch(input$dataset,
 "rock" = rock,
 "mtcars" = mtcars)
})
```

A reactive expression reads input and returns an output. It regenerates the output only when the input it depends on changes. Every time it generates the output, it caches the output and uses it until the input changes. The reactive expression can be called from another reactive expression or from a `render*` function.

## **Creating single file web application**

In R 3.0.0 Version, a shiny application can include just a single `app.R` file in a separate folder along with any needed data files and dependent R source code files. Create a new `SingleFileApp` folder and save the downloaded `app.R` here. Take a look at the downloaded `app.R` file and start the application by entering the `runApp ("SingleFileApp")` command.

In `app.R`, the `shinyApp(ui = ui, server = server)` line is executed first by R. This `shinyApp` function returns an object of class `shiny.appobj` to the console. When this object is printed, the shiny app is launched in a separate window.

It is possible to create a single-file shiny application without a specific application directory and with a filename other than `app.R`. There should be a call to the `shinyApp` function, which is what tells R that it is a shiny application. We can then run `print(source("appfilename"))` to launch the application. The caveat if you run with a different name is that, when you modify the file, the application is not automatically relaunched.

## **Creating PDF presentations of your analysis with R Presentation**

`Rpres`, built into RStudio, enables you to create PDF slide presentations of your data analysis. In this recipe, we develop a small application that showcases the important `Rpres` features.

### **Getting ready**

Download the files for this chapter and store the `sample-image.png` and `Introduction.Rpres` files in your R working directory.

### **How to do it...**

1. Open RStudio.
2. Create a new R Presentation document using the following steps:
  1. Navigate to **File | New File** and click on **R Presentation**.
  2. Enter the filename as `RPresentation` and save it in your R working directory.
  3. RStudio creates a file with the extension `Rpres`. This file includes a default title slide (the very first slide) and a few other sample slides. Creating this file also results in a preview being displayed in the upper right of the RStudio environment.
  4. Fill in author and date and click on **Preview**.
  5. By default, the preview appears in RStudio itself. However, to see all features properly, drop down the menu on the top right of the tab where the presentation appears and select **View in browser**.
  6. Click on the arrow button at the bottom right to navigate through the slides.
3. Open the R Presentation document that you downloaded earlier:
  1. Navigate to **File | Open File**.
  2. Open the `Introduction.Rpres` file.

4. To embed an image use the following code:

```
Slide with image
=====
! [Sample Image] (sample-image.png)
```

5. To create a two-column layout perform the following steps:

- The two columns are separated by \*\*\* on a separate line.
- Add the following slide:

```
Two Columns
=====
left:40%

ColumnOne
- this slide has two columns
- the first column has text
- the second column has an image

ColumnTwo
```

```
! [Sample Image] (sample-image.png) Two Columns
```

6. To add a transition to the slides:

- Global transition setting:

```
Introduction
=====
author: Shanthi Viswanathan
date: 16 Dec 2014
transition:rotate
transition-speed:slow
```

7. To add incremental displays:

- Add the following in the first slide:

```
Incremental Display
=====
transition: concave
incremental: true
```

## How it works...

In steps 1 and 2, a simple presentation is created.

In step 3, an R presentation file is opened. When a text is followed by a set of = characters (at least 3 on a line by itself) it is taken as the slide title.

In step 4, an image is added with the standard markdown syntax of the exclamation mark followed in square brackets by the alt text, followed by the image file's name in parentheses. The image occupies the entire slide if it is the only content on that slide.

In step 5, a two-column slide is created. The two columns are separated by three \* characters. This time, the image occupies the entire column into which it is added.

The double asterisks signify a column. By default, the two columns occupy 50% of the slide width. In a two-column layout, each column by default occupies 50% of the slide width. Use left or right to change this. We used left: 40%.

A transition effect can be applied to all slides within a presentation or specifically for each slide. The default transition is linear. To apply to all slides, add it to the title slide as in step 6.

By default, all elements on the slide appear when RPres shows the slide. We can change this by setting incremental = TRUE. With this setting, list items, code blocks, and paragraphs are displayed incrementally with a mouse-click. The first paragraph in a slide is immediately displayed and the increment rule is applied to the subsequent content. In step 7, we see this behavior in the display of bullet points.

## There's more...

We now describe additional options to control the display.

### Using hyperlinks

External or internal links can be added to an R presentation. External links use the same R Markdown syntax. For internal links, we first need to add an id to a slide and then create a link using it, as the following code shows:

```
Two Columns
=====
id: twocols

First Slide
=====
[Go to Slide] (#/twocols)
```

## Controlling the display

The default size of an R presentation is 960 x 700 pixels. However, adding a specific width or height to a slide can change its default size:

```
Slide with plot
=====
title: false
```{r renderplot,echo=FALSE,out.width="1920px"}
plot(cars)
```
```

If the entire plot is not displayed in the slide when viewed in a browser, you can include `fig.width` and `fig.height` as follows:

```
Slide with plot
=====
title: false
```{r renderplot,echo=FALSE, fig.width=8,fig.height=4,
out.width="1920px"}
plot(cars)
```
```

## Enhancing the look of the presentation

You can set the font in the title slide, after which the font is applied to all the slides. This global font can also be overridden in specific slides:

```
Introduction
=====
author: Shanthi Viswanathan
date: 16 Dec 2014
font-family: Arial
```

You can include a `.css` file in the title slide and use the styles defined in the `.css` file in the slides as follows:

```
Introduction
=====
author: Shanthi Viswanathan
date: 16 Dec 2014
css: custom.css

Two Columns
=====
class: highlight
left:70%
```



# 9

## Work Smarter, Not Harder – Efficient and Elegant R Code

In this chapter, we will cover recipes for doing the following without explicit iteration:

- ▶ Exploiting vectorized operations
- ▶ Processing entire rows or columns using the apply function
- ▶ Applying a function to all elements of a collection with lapply and sapply
- ▶ Applying functions to subsets of a vector
- ▶ Using the split-apply-combine strategy with plyr
- ▶ Slicing, dicing, and combining data with data tables

### Introduction

The R programming language, being procedural, provides looping control structures. Most people will therefore tend to automatically use these control structures in their own code and end up with performance issues because R handles loops very inefficiently. Serious number crunching and handling large datasets in R require us to exploit powerful, alternative ways to write succinct, elegant, and efficient code as follows:

- ▶ Vectorized operations process collections as a whole instead of operating element by element
- ▶ The `apply` family of functions processes rows, columns, or lists as a whole without the need for explicit iteration

- ▶ The `plyr` package provides a wide range of `**ply` functions with additional functionality, including parallel processing
- ▶ The `data.table` package provides helpful functions to manipulate data easily and efficiently

This chapter provides recipes using all of these features.

## Exploiting vectorized operations

Some R functions can operate on vectors as a whole. The function can either be a built-in R function or a custom function. In your own code, before you resort to a loop to process all elements of a vector, see whether you can exploit an existing vectorized function.

### Getting ready

If you have not already downloaded the files for this chapter, do it now and ensure that the `auto-mpg.csv` file is in your R working directory.

### How to do it...

To exploit vectorized operations follow these steps:

1. Operate on all elements of vector(s) without explicit iteration (vectorized operations):

```
> first.name <- c("John", "Jane", "Tom", "Zach")
> last.name <- c("Doe", "Smith", "Glock", "Green")
> # The paste function below operates on vectors
> paste(first.name, last.name)

[1] "John Doe" "Jane Smith" "Tom Glock" "Zach Green"

> # This works even with different sized vectors
> new.last.name <- c("Dalton")
> paste(first.name, new.last.name)

[1] "John Dalton" "Jane Dalton" "Tom Dalton" "Zach
Dalton"
```

2. Use vectorized operations within your own functions:

```
> username <- function(first, last) {
 tolower(paste0(last, substr(first,1,1)))
}
```

```
> username(first.name, last.name)

[1] "doej" "smithj" "glockt" "greenz"
```

3. Apply an arithmetic operation implicitly on all elements of a vector:

```
> auto <- read.csv("auto-mpg.csv")
> auto$kmpg <- auto$mpg*1.6
```

## How it works...

By operating on entire vectors at a time, vectorized operations eliminate the need for explicit loops. R processes loops inefficiently because it interprets the statements in a loop over and over again. Thus, loops with much iteration tend to perform poorly. Vectorized operations help us to get around this bottleneck, while at the same time making our code compact and more elegant.

Several built-in functions are vectorized and step 1 illustrates this with the `paste` function that concatenates strings.

The later part of step 1 shows that, if the vectors have unequal length, then the shorter vector recycles the list of vectors as needed. The `new.last.name` vector of size 1 repeats itself to match the size of the `first.name` vector. Hence, the last name Dalton is pasted to each element of `first.name`.

Vector operations work for built-in functions, custom functions, and arithmetic operations. A custom function to generate usernames using the two vectors `first.name` and `last.name` is seen in step 3.

Vector operations work even when we combine vectors and scalars in arithmetic operations. A new variable is created in step 4 in the `auto` data frame to represent fuel efficiency in kilometers per gallon (`kmpg`) using a simple formula combining a vector and a scalar.

## There's more...

R functions such as `sum`, `min`, `max`, `range`, and `prod` combine their arguments into vectors:

```
> sum(1, 2, 3, 4, 5)

[1] 15
```

On the contrary, beware of functions such as `mean` and `median` that *do not* combine arguments into vectors and yield misleading results:

```
> mean(1, 2, 3, 4, 5)

[1] 1
```

```
> mean(c(1,2,3,4,5))
[1] 3
```

## Processing entire rows or columns using the apply function

The `apply` function can apply a user-specified function to all rows or columns of a matrix and return an appropriate collection with the results.

### Getting ready

This recipe uses no external objects or resources.

### How to do it...

To process entire rows or columns using the `apply` function, follow these steps:

1. Calculate row minimums for a matrix:

```
> m <- matrix(seq(1,16), 4, 4)
> m

[,1] [,2] [,3] [,4]
[1,] 1 5 9 13
[2,] 2 6 10 14
[3,] 3 7 11 15
[4,] 4 8 12 16

> apply(m, 1, min)

[1] 1 2 3 4
```

2. Calculate column maximums for a matrix:

```
> apply(m, 2, max)

[1] 4 8 12 16
```

3. Create a new matrix by squaring every element of a given matrix:

```
> apply(m, c(1,2), function(x) x^2)

[,1] [,2] [,3] [,4]
[1,] 1 25 81 169
```

```
[2,] 4 36 100 196
[3,] 9 49 121 225
[4,] 16 64 144 256
```

4. Apply a function to every row and pass an argument to the function:

```
> apply(m, 1, quantile, probs=c(.4,.8))
 [,1] [,2] [,3] [,4]
40% 5.8 6.8 7.8 8.8
80% 10.6 11.6 12.6 13.6
```

## How it works...

Step 1 creates a matrix and generates the row minimums for it.

- ▶ The first argument for `apply` is a matrix or array.
- ▶ The second argument (called the `margin`) specifies how we want to split the matrix or array into pieces. For a two-dimensional structure, we can operate on rows as 1, columns as 2, or elements as `c(1,2)`. For matrices of more than two dimensions, `margin` can be more than two and specifies the dimension(s) of interest (see the *There's more...* section).
- ▶ The third argument is a function—built-in or custom. In fact, we can even specify an unnamed function in-line as step 3 shows.

The `apply` function invokes the specified function with each row, column, or element of the matrix depending on the second argument.

The return value from `apply` depends on `margin` and the type of return value from the user-specified function.

If we supply more than three arguments to `apply`, it passes these along to the specified function. The `probs` argument in step 4 serves as an example. In step 4, `apply` passes along the `probs` vector to the `quantile` function.



To calculate row/column means or sums for a matrix, use the highly optimized `colMeans`, `rowMeans`, `colSums`, and `rowSums` functions instead of `apply`.

## There's more...

The `apply` function can use an array of any dimension as input. Also, you can use `apply` on a data frame after converting it into a matrix using `as.matrix`.

### Using `apply` on a three-dimensional array

1. Create a three-dimensional array:

```
> array.3d <- array(seq(100,69), dim = c(4,4,2))
> array.3d

, , 1

 [,1] [,2] [,3] [,4]
[1,] 100 96 92 88
[2,] 99 95 91 87
[3,] 98 94 90 86
[4,] 97 93 89 85

, , 2

 [,1] [,2] [,3] [,4]
[1,] 84 80 76 72
[2,] 83 79 75 71
[3,] 82 78 74 70
[4,] 81 77 73 69
```

2. Calculate the sum across the first and second dimensions. We get a one-dimensional array with two elements:

```
> apply(array.3d, 3, sum)
[1] 1480 1224

> # verify
> sum(85:100)
[1] 1480
```

3. Calculate the sum across the third dimension. We get a two-dimensional array:

```
> apply(array.3d,c(1,2),sum)
[,1] [,2] [,3] [,4]
[1,] 184 176 168 160
[2,] 182 174 166 158
[3,] 180 172 164 156
[4,] 178 170 162 154
```

## Applying a function to all elements of a collection with lapply and sapply

The `lapply` function works on objects of type vector, list, or data frame. It applies a user-specified function to each element of the passed-in object and returns a list of the results.

### Getting ready

Download the files for this chapter and store the `auto-mpg.csv` file in your R working directory. Read the data:

```
> auto <- read.csv("auto-mpg.csv", stringsAsFactors=FALSE)
```

### How to do it...

To apply a function to all elements of a collection with `lapply` and `sapply`, follow these instructions:

1. Operate on a simple vector:

```
> lapply(c(1,2,3), sqrt)
```

```
[[1]]
```

```
[1] 1
```

```
[[2]]
```

```
[1] 1.414214
```

```
[[3]]
```

```
[1] 1.732051
```

2. Use `lapply` and `sapply` to calculate the means of a list of collections:

```
> x <- list(a = 1:10, b = c(1,10,100,1000),
 c=seq(5,50,by=5))
> lapply(x, mean)
```

```
$a
```

```
[1] 5.5
```

```
$b
```

```
[1] 277.75
```

```
$c
```

```
[1] 27.5
```

```
> class(lapply(x,mean))
[1] "list"

> sapply(x, mean)

 a b c
5.50 277.75 27.50

> class(sapply(x,mean))

[1] "numeric"
```

3. Calculate the minimum value for each variable in the `auto` data frame:

```
> sapply(auto[,2:8], min)
 mpg cylinders displacement horsepower
 9 3 68 46
 weight acceleration model_year
 1613 8 70
```

## How it works...

The `lapply` function accepts three arguments—the first argument is the object, the second is the user-specified function, and the optional third argument specifies the additional arguments to the user-specified function. The `lapply` function always returns a list irrespective of the type of the first argument.

In step 1, the `lapply` function is used to apply `sqrt` to each element of a vector. The `lapply` function always returns a list.

In step 2, a list with three elements is involved, each of which is a vector. It calculates the mean of these vectors. The `lapply` function returns a list, whereas `sapply` returns a vector in this case.

In step 3, `sapply` is used to apply a function to columns of a data frame. For obvious reasons, we pass only the numeric columns.

## There's more...

The `sapply` function returns a vector if every element of the result is of length 1. If every element of the result list is a vector of the same length, then `sapply` returns a matrix. However, if we specify `simplify=F`, then `sapply` always returns a list. The default is `simplify=T`. See the following:

## Dynamic output

In the next two examples, `sapply` returns a matrix. If the function that it executes has row and column names defined, then `sapply` uses these for the matrix:

```
> sapply(auto[,2:6], summary)

 mpg cylinders displacement horsepower weight
Min. 9.00 3.000 68.0 46.0 1613
1st Qu. 17.50 4.000 104.2 76.0 2224
Median 23.00 4.000 148.5 92.0 2804
Mean 23.51 5.455 193.4 104.1 2970
3rd Qu. 29.00 8.000 262.0 125.0 3608
Max. 46.60 8.000 455.0 230.0 5140

> sapply(auto[,2:6], range)

 mpg cylinders displacement horsepower weight
[1,] 9.0 3 68 46 1613
[2,] 46.6 8 455 230 5140
```

## One caution

As we mentioned earlier, the output type of `sapply` depends on the input object. However, because of how R operates with data frames, it is possible to get an "unexpected" output:

```
> sapply(auto[,2:6], min)

 mpg cylinders displacement horsepower weight
 9 3 68 46 1613
```

In the preceding example, `auto[,2:6]` returns a data frame and hence the input to `sapply` is a data frame object. Each variable (or column) of the data frame is passed as an input to the `min` function, and we get the output as a vector with column names taken from the input object. Try this:

```
> sapply(auto[,2], min)

[1] 28.0 19.0 36.0 28.0 21.0 23.0 15.5 32.9 16.0 13.0 12.0 30.7
[13] 13.0 27.9 13.0 23.8 29.0 14.0 14.0 29.0 20.5 26.6 20.0 20.0
[25] 26.4 16.0 40.8 15.0 18.0 35.0 26.5 13.0 25.8 39.1 25.0 14.0
[37] 19.4 30.0 32.0 26.0 20.6 17.5 18.0 14.0 27.0 25.1 14.0 19.1
[49] 17.0 23.5 21.5 19.0 22.0 19.4 20.0 32.0 30.9 29.0 14.0 14.0
[61]
```

This happened because R treats `auto[, 2:6]` as a data frame, but `auto[, 2]` as just a vector. Hence, in the former case `sapply` operated on each column separately and in the latter case it operated on each element of a vector.

We can fix the preceding code by coercing the `auto[, 2]` vector to a data frame and then pass this data frame object as an input to the `min` function:

```
> sapply(as.data.frame(auto[, 2]), min)
auto[, 2]
[1] 9
```

In the following example, we add `simplify=F` to force the return value to a list:

```
> sapply(as.data.frame(auto[, 2]), min, simplify=F)
$`auto[, 2]`
[1] 9
```

## Applying functions to subsets of a vector

The `tapply` function applies a function to each partition of the dataset. Hence, when we need to evaluate a function over subsets of a vector defined by a factor, `tapply` comes in handy.

### Getting ready

Download the files for this chapter and store the `auto-mpg.csv` file in your R working directory. Read the data and create factors for the `cylinders` variable:

```
> auto <- read.csv("auto-mpg.csv", stringsAsFactors=FALSE)
> auto$cylinders <- factor(auto$cylinders, levels = c(3,4,5,6,8),
 labels = c("3cyl", "4cyl", "5cyl", "6cyl", "8cyl"))
```

### How to do it...

To apply functions to subsets of a vector, follow these steps:

1. Calculate mean mpg for each cylinder type:

```
> tapply(auto$mpg, auto$cylinders, mean)
```

|          | 3cyl     | 4cyl     | 5cyl     | 6cyl     | 8cyl     |
|----------|----------|----------|----------|----------|----------|
| 20.55000 | 20.55000 | 29.28676 | 27.36667 | 19.98571 | 14.96311 |

2. We can even specify multiple factors as a list. The following example shows only one factor since the out file has only one, but it serves as a template that you can adapt:

```
> tapply(auto$mpg, list(cyl=auto$cylinders), mean)
```

```
cyl
 3cyl 4cyl 5cyl 6cyl 8cyl
20.55000 29.28676 27.36667 19.98571 14.96311
```

## How it works...

In step 1 the `mean` function is applied to the `auto$mpg` vector grouped according to the `auto$cylinders` vector. The grouping factor should be of the same length as the input vector so that each element of the first vector can be associated with a group.

The `tapply` function creates groups of the first argument based on each element's group affiliation as defined by the second argument and passes each group to the user-specified function.

Step 2 shows that we can actually group by several factors specified as a list. In this case, `tapply` applies the function to each unique combination of the specified factors.

## There's more...

The `by` function is similar to `tapply` and applies the function to a group of rows in a dataset, but by passing in the entire data frame. The following examples clarify this.

## Applying a function on groups from a data frame

In the following example, we find the correlation between `mpg` and `weight` for each cylinder type:

```
> by(auto, auto$cylinders, function(x) cor(xmpg, xweight))
auto$cylinders: 3cyl
[1] 0.6191685

auto$cylinders: 4cyl
[1] -0.5430774

auto$cylinders: 5cyl
[1] -0.04750808

auto$cylinders: 6cyl
[1] -0.4634435

auto$cylinders: 8cyl
[1] -0.5569099
```

## Using the split-apply-combine strategy with `plyr`

Many data analysis tasks involve first splitting the data into subsets, applying some operation on each subset, and then combining the results suitably. A common wrinkle in applying this happens to be the numerous possible combinations of input and output object types. The `plyr` package provides simple functions to apply this pattern while simplifying the specification of the object types through systematic naming of the functions.

A `plyr` function name has three parts:

- ▶ The first letter represents the input object type
- ▶ The second letter represents the output object type
- ▶ The third to fifth letters are always `ply`

In the `plyr` function names, `d` represents a data frame, `l` represents a list, and `a` represents an array. For example, `ddply` has its input and output as data frames, and `ldply` takes a list input and produces a data frame as output. Sometimes, we apply functions only for their side effects (such as plots) and do not want the output objects at all. In such cases, we can use `_` for the second part. Therefore, `d_ply()` takes a data frame as input and produces no output—only the side effects of the function application occur.

### Getting ready

Download the files for this chapter and store the `auto-mpg.csv` file in your R working directory. Read the data and create factors for `auto$cylinders`:

```
> auto <- read.csv("auto-mpg.csv", stringsAsFactors=FALSE)
> auto$cylinders <- factor(auto$cylinders, levels = c(3,4,5,6,8),
 labels = c("3cyl", "4cyl", "5cyl", "6cyl", "8cyl"))
```

Install the `plyr` package in your R environment if you do not have it already. This can be done using the following commands:

```
> install.packages("plyr")
> library(plyr)
```

### How to do it...

To use the split-apply-combine strategy for data analysis with `plyr` follow these steps:

1. Calculate mean `mpg` for each cylinder type (two versions):

```
> ddply(auto, "cylinders", function(df) mean(df$mpg))
```

```
> ddply(auto, ~ cylinders, function(df) mean(df$mpg))
cylinders V1
1 3cyl 20.55000
2 4cyl 29.28676
3 5cyl 27.36667
4 6cyl 19.98571
5 8cyl 14.96311
```

2. Calculate the mean, minimum, and maximum mpg for each cylinder type and model year:

```
> ddply(auto, c("cylinders", "model_year"),
 function(df) c(mean=mean(df$mpg),
 min=min(df$mpg), max=max(df$mpg)))
> ddply(auto, ~ cylinders + model_year, function(df)
 c(mean=mean(df$mpg), min=min(df$mpg), max=max(df$mpg)))
```

|    | cylinders | model_year | mean     | min  | max  |
|----|-----------|------------|----------|------|------|
| 1  | 3cyl      | 72         | 19.00000 | 19.0 | 19.0 |
| 2  | 3cyl      | 73         | 18.00000 | 18.0 | 18.0 |
| 3  | 3cyl      | 77         | 21.50000 | 21.5 | 21.5 |
| 4  | 3cyl      | 80         | 23.70000 | 23.7 | 23.7 |
| 5  | 4cyl      | 70         | 25.28571 | 24.0 | 27.0 |
| 6  | 4cyl      | 71         | 27.46154 | 22.0 | 35.0 |
| 7  | 4cyl      | 72         | 23.42857 | 18.0 | 28.0 |
| 8  | 4cyl      | 73         | 22.72727 | 19.0 | 29.0 |
| 9  | 4cyl      | 74         | 27.80000 | 24.0 | 32.0 |
| 10 | 4cyl      | 75         | 25.25000 | 22.0 | 33.0 |
| 11 | 4cyl      | 76         | 26.76667 | 19.0 | 33.0 |
| 12 | 4cyl      | 77         | 29.10714 | 21.5 | 36.0 |
| 13 | 4cyl      | 78         | 29.57647 | 21.1 | 43.1 |
| 14 | 4cyl      | 79         | 31.52500 | 22.3 | 37.3 |
| 15 | 4cyl      | 80         | 34.61200 | 23.6 | 46.6 |
| 16 | 4cyl      | 81         | 32.81429 | 25.8 | 39.1 |
| 17 | 4cyl      | 82         | 32.07143 | 23.0 | 44.0 |
| 18 | 5cyl      | 78         | 20.30000 | 20.3 | 20.3 |
| 19 | 5cyl      | 79         | 25.40000 | 25.4 | 25.4 |
| 20 | 5cyl      | 80         | 36.40000 | 36.4 | 36.4 |
| 21 | 6cyl      | 70         | 20.50000 | 18.0 | 22.0 |
| 22 | 6cyl      | 71         | 18.00000 | 16.0 | 19.0 |
| 23 | 6cyl      | 73         | 19.00000 | 16.0 | 23.0 |
| 24 | 6cyl      | 74         | 17.85714 | 15.0 | 21.0 |
| 25 | 6cyl      | 75         | 17.58333 | 15.0 | 21.0 |
| 26 | 6cyl      | 76         | 20.00000 | 16.5 | 24.0 |
| 27 | 6cyl      | 77         | 19.50000 | 17.5 | 22.0 |
| 28 | 6cyl      | 78         | 19.06667 | 16.2 | 20.8 |

```
29 6cyl 79 22.95000 19.8 28.8
30 6cyl 80 25.90000 19.1 32.7
31 6cyl 81 23.42857 17.6 30.7
32 6cyl 82 28.33333 22.0 38.0
33 8cyl 70 14.11111 9.0 18.0
34 8cyl 71 13.42857 12.0 14.0
35 8cyl 72 13.61538 11.0 17.0
36 8cyl 73 13.20000 11.0 16.0
37 8cyl 74 14.20000 13.0 16.0
38 8cyl 75 15.66667 13.0 20.0
39 8cyl 76 14.66667 13.0 17.5
40 8cyl 77 16.00000 15.0 17.5
41 8cyl 78 19.05000 17.5 20.2
42 8cyl 79 18.63000 15.5 23.9
43 8cyl 81 26.60000 26.6 26.6
```

## How it works...

In step 1 `ddply` is used. This function takes a data frame as input and produces a data frame as output. The first argument is the `auto` data frame. The second argument `cylinders` describes the way to split the data. The third argument is the function to perform on the resulting components. We can add additional arguments if the function needs arguments. We can specify the splitting factor using the formula interface, `~ cylinders`, as the second option of step 1 shows.

Step 2 shows how data splitting can occur across multiple variables as well. We use the `c("cylinders", "model_year")` vector format to split the data using two variables. We also named the variables as `mean`, `min`, and `max` instead of the default `V1`, `V2`, and so on. The second option here also shows the use of the formula interface.

## There's more...

In this section, we discuss the `transform` and `summarize` functions as well as the identity function `I`.

### Adding a new column using `transform`

Suppose you want to add a new column to reflect each auto's deviation from the mean mpg of the cylinder group to which it belongs:

```
> auto <- ddply(auto, .(cylinders), transform, mpg.deviation =
 round(mpg - mean(mpg), 2))
```

## Using summarize along with the plyr function

The following command shows the output when `summarize` is used:

```
> ddply(auto, .(cylinders), summarize, freq=length(cylinders),
 meanmpg=mean(mpg))
 cylinders freq meanmpg
 1 3cyl 4 20.55000
 2 4cyl 204 29.28676
 3 5cyl 3 27.36667
 4 6cyl 84 19.98571
 5 8cyl 103 14.96311
```

We calculate the number of rows and the mean `mpg` of the data frame grouped by `cylinders`.

## Concatenating the list of data frames into a big data frame

Run the following commands:

```
> autos <- list(auto, auto)
> big.df <- ldply(autos, I)
```

The `ldply` function takes a list input and spits out a data frame output. The identity function `I` returns the input as is. If the input is a list then there is no split by argument; each list element is passed as an argument to the function.

## Slicing, dicing, and combining data with data tables

R provides several packages to do data analysis and data manipulation. Over and above the `apply` family of functions, the most commonly used packages are `plyr`, `reshape`, `dplyr`, and `data.table`. In this recipe, we will cover `data.table`, which processes large amounts of data very efficiently without our having to write detailed procedural code.

### Getting ready

Download the files for this chapter and store the `auto-mpg.csv`, `employees.csv`, and `departments.csv` files in your R working directory. Read the data and create factors for `cylinders` in `auto-mpg.csv`:

```
> auto <- read.csv("auto-mpg.csv", stringsAsFactors=FALSE)
> auto$cylinders <- factor(auto$cylinders, levels = c(3,4,5,6,8),
 labels = c("3cyl", "4cyl", "5cyl", "6cyl", "8cyl"))
```

Install the `data.table` package in your R environment as follows:

```
> install.packages("data.table")
> library(data.table)
> autoDT <- data.table(auto)
```

## How to do it...

In this recipe, we cover `data.table` which processes large amounts of data very efficiently without our having to write detailed procedural code. To do this follow these steps:

1. Calculate the mean mpg for each cylinder type:

```
> autoDT[, mean(mpg), by=cylinders]
```

```
 cylinders V1
1: 4cyl 29.28676
2: 3cyl 20.55000
3: 6cyl 19.98571
4: 8cyl 14.96311
5: 5cyl 27.36667
```

2. Add a column for the mean mpg for each cylinder type:

```
> autoDT[, meanmpg := mean(mpg), by=cylinders]
```

```
> autoDT[1:5,c(1:3,9:10), with=FALSE]
```

```
 No mpg cylinders car_name meanmpg
1: 1 28 4cyl chevrolet vega 2300 29.28676
2: 2 19 3cyl mazda rx2 coupe 20.55000
3: 3 36 4cyl honda accord 29.28676
4: 4 28 4cyl datsun 510 (sw) 29.28676
5: 5 21 6cyl amc gremlin 19.98571
```

3. Create an index on cylinders by defining a key:

```
> setkey(autoDT,cylinders)
```

```
> tables()
```

```
 NAME NROW NCOL MB
[1,] autoDT 398 10 1
 COLS
[1,] No,mpg,cylinders,displacement,horsepower,weight,acceleration,
model_year,car_name
 KEY
[1,] cylinders
```

Total: 1MB

```
> autoDT["4cyl", c(1:3, 9:10), with=FALSE]
```

| No   | mpg | cylinders | car_name              | meanmpg       |
|------|-----|-----------|-----------------------|---------------|
| 1:   | 1   | 28.0      | chevrolet vega        | 2300 29.28676 |
| 2:   | 3   | 36.0      | honda accord          | 29.28676      |
| 3:   | 4   | 28.0      | datsun 510 (sw)       | 29.28676      |
| 4:   | 6   | 23.0      | audi 100ls            | 29.28676      |
| 5:   | 8   | 32.9      | datsun 200sx          | 29.28676      |
| ---  |     |           |                       |               |
| 200: | 391 | 32.1      | chevrolet chevette    | 29.28676      |
| 201: | 392 | 23.9      | datsun 200-sx         | 29.28676      |
| 202: | 395 | 34.5      | plymouth horizon tc3  | 29.28676      |
| 203: | 396 | 38.1      | toyota corolla tercel | 29.28676      |
| 204: | 397 | 30.5      | chevrolet chevette    | 29.28676      |

4. Calculate mean, min and max mpg grouped by cylinder type:

```
> autoDT[, list(meanmpg=mean(mpg), minmpg=min(mpg),
maxmpg=max(mpg)), by=cylinders]
```

|    | cylinders | meanmpg  | minmpg | maxmpg |
|----|-----------|----------|--------|--------|
| 1: | 3cyl      | 20.55000 | 18.0   | 23.7   |
| 2: | 4cyl      | 29.28676 | 18.0   | 46.6   |
| 3: | 5cyl      | 27.36667 | 20.3   | 36.4   |
| 4: | 6cyl      | 19.98571 | 15.0   | 38.0   |
| 5: | 8cyl      | 14.96311 | 9.0    | 26.6   |

## How it works...

Data tables in the `data.table` package outperform the `*apply` family of functions and the `**ply` functions. The simple `data.table` syntax is `DT[i, j, by]`, where the data table `DT` is subset using rows in `i` to calculate `j` grouped by `by`.

In step 1 `mean(mpg)` is calculated, grouped by `cylinders` for all rows of the data table; omitting `i` causes all rows of the data table to be included.

To create a new column for the calculated `j`, just add `:=` as in step 2. Here, we added a new column `meanmpg` to the data table to store `mean(mpg)` for each cylinder type.

By default, `with` is set to `TRUE` and `j` is evaluated for subsets of the data frame. However, if we do not need any computation and just want to retrieve data, then we can specify `with=FALSE`. In this case, data tables behave just like data frames.

Unlike data frames, data tables do not have row names. Instead, we can define keys and use these keys for row indexing. Step 3 defines `cylinders` as the key, and then uses `autoDT[["4cyl", c(1:3, 9:10), with=FALSE]]` to extract data for the key-column value `4cyl`.

We can define multiple keys using `setkeyv(DT, c("col1", "col2"))`, where `DT` is the data table and `col1` and `col2` are the two columns in the data table. In step 3, if multiple keys are defined, then the syntax to extract the data is `autoDT[.( "4cyl"), c(1:3, 9:10), with=FALSE]`.

If, in `DT[i, j, by]`, `i` is itself a `data.table`, then R joins the two data tables on keys. If keys are not defined, then an error is displayed. However, for `by`, keys are not required.

## There's more...

We see some advanced techniques using `data.table`.

### Adding multiple aggregated columns

In step 2, we added one calculated column `meanmpg`. The `:=` syntax computes the variable and merges it into the original data:

```
> # calculate median and sd of mpg grouped by cylinders
> autoDT[,c("medianmpg","sdmpg") := list(median(mpg),sd(mpg)),
 by=cylinders]
> # Display selected columns of autoDT table for the first 5 rows
> autoDT[1:5,c(3,9:12), with=FALSE]
 cylinders car_name meanmpg medianmpg sdmpg
1: 3cyl mazda rx2 coupe 20.55000 20.25 2.564501
2: 3cyl mazda rx3 20.55000 20.25 2.564501
3: 3cyl mazda rx-7 gs 20.55000 20.25 2.564501
4: 3cyl mazda rx-4 20.55000 20.25 2.564501
5: 4cyl chevrolet vega 2300 29.28676 28.25 5.710156
```

### Counting groups

We can easily count the number of rows in each group as follows:

```
> autoDT[, .N ,by=cylinders]
 cylinders N
1: 3cyl 4
2: 4cyl 204
3: 5cyl 3
4: 6cyl 84
5: 8cyl 103
```

We can also count after subsetting as follows:

```
> autoDT["4cyl", .N]
[1] 204
```

## Deleting a column

We can easily delete a column by setting it to `NULL` as follows:

```
> autoDT[, medianmpg := NULL]
```

## Joining data tables

We can define one or more keys on data tables and use them for joins. Suppose that a data table `DT` has a key defined. Then if, in `DT[i, j, by]`, `i` is also a data table, R outer joins the two data tables on the key of `DT`. It joins the first key field of `DT` with the first column of `i`, the second key field of `DT` with the second column of `i`, and so on. If no keys are defined in `DT`, then R returns an error:

```
> emp <- read.csv("employees.csv", stringsAsFactors=FALSE)
> dept <- read.csv("departments-1.csv", stringsAsFactors=FALSE)
> empDT <- data.table(emp)
> deptDT <- data.table(dept)
> setkey(empDT, "DeptId")
```

At this point, we have two data tables `empDT` and `deptDT` and a key field in `empDT`. The department ID in `deptDT` also happens to be the first column. We can now join the two tables on department ID by the following code. Note that the column name in `deptDT` does not have to match the name of the key field in `empDT`—only the column position matters:

```
> combine <- empDT[deptDT]
> combine[, .N]
[1] 100
```

To prevent creating large result sets inadvertently, the data table's `join` operation checks to see if the result set has become larger than the size of either table and stops with an error immediately. Unfortunately, this check results in an error in some perfectly valid situations.

For example, if there were two departments in the `deptDT` table that did not appear in the `empDT` table, then the outer join operation will yield 102 rows and not 100. Since the number of resultant rows is larger than the larger of the two tables, the preceding check results in an error message. The following code illustrates this:

```
> dept <- read.csv("departments-2.csv", stringsAsFactors=FALSE)
> deptDT <- data.table(dept)
> # The following line gives an error
> combine <- empDT[deptDT]
```

```
Error in vecseq(f__, len__, if (allow.cartesian) NULL else
 as.integer(max(nrow(x)), : Join results in 102 rows; more than 100 =
 max(nrow(x), nrow(i)) ... (error message truncated)
```

If we know for sure that what we are doing is correct, we can force R to perform the join by using `allow.cartesian=TRUE`:

```
combine <- empDT[deptDT, allow.cartesian=TRUE]
combine[, .N]
102
```

We get 102 rows because of the two departments that had no employees and the default outer join added two extra rows for these two departments. We can force an inner join by passing `nomatch=0` as follows:

```
> mash <- empDT[deptDT, nomatch=0]
> mash[, .N]
[1] 100
```

## Using symbols

We can use special symbols such as `.SD`, `.EACHI`, `.N`, `.I`, and `.BY` in `data.table` to enhance the functionality. We already saw some examples on `.N`, which represents the number of rows or the last row.

The `.SD` symbol holds all columns except the columns in `by` and can be used only in the `j` evaluation part of `data.table`. The `.SDcols` symbol is used along with `.SD` and has columns to be included or excluded in the `j` part of `data.table`.

The `.EACHI` symbol is used in the `by` grouping to group each subset of the groups in `i`. This needs a key to be defined. If there is no key, R throws an error.

In the following example, we calculate the maximum salary in each department. If we omit `.SDcols="Salary"` then R will try to find the max for all columns since, by default, `.SD` includes all columns. In this case, R will throw an error since there are columns with textual values in the `empDT` data table:

```
> empDT[deptDT, max(.SD), by=.EACHI, .SDcols="Salary"]
```

|    | DeptId | V1    |
|----|--------|-------|
| 1: | 1      | 99211 |
| 2: | 2      | 98291 |
| 3: | 3      | 70655 |
| 4: | 4      | NA    |
| 5: | 5      | 99397 |
| 6: | 6      | 92429 |
| 7: | 7      | NA    |

In the following example, we calculate the average salary in each department. We give the name AvgSalary to this calculated column. We can either use list or .() notation in the j evaluation part:

```
> empDT[, .(AvgSalary = lapply(.SD, mean)),
 by="DeptId", .SDcols="Salary"]
```

|    | DeptId | AvgSalary |
|----|--------|-----------|
| 1: | 1      | 63208.02  |
| 2: | 2      | 59668.06  |
| 3: | 3      | 47603.64  |
| 4: | 5      | 59448.24  |
| 5: | 6      | 51957.44  |

In the following example, we calculate the average salary in each department. We also include the department name DeptName by joining empDT with deptDT:

```
> empDT[deptDT, list(DeptName, AvgSalary = lapply(.SD, mean)),
 by=.EACHI, .SDcols="Salary"]
```

|    | DeptId | DeptName   | AvgSalary |
|----|--------|------------|-----------|
| 1: | 1      | Finance    | 63208.02  |
| 2: | 2      | HR         | 59668.06  |
| 3: | 3      | Marketing  | 47603.64  |
| 4: | 4      | Sales      | NA        |
| 5: | 5      | IT         | 59448.24  |
| 6: | 6      | Service    | 51957.44  |
| 7: | 7      | Facilities | NA        |



# 10

## **Where in the World? – Geospatial Analysis**

In this chapter, we will cover:

- ▶ Downloading and plotting a Google map of an area
- ▶ Overlaying data on the downloaded Google map
- ▶ Importing ESRI shape files into R
- ▶ Using the `sp` package to plot geographic data
- ▶ Getting maps from the `maps` package
- ▶ Creating spatial data frames from regular data frames containing spatial and other data
- ▶ Creating spatial data frames by combining regular data frames with spatial objects
- ▶ Adding variables to an existing spatial data frame

## Introduction

Maps and other forms of geographical displays surround us. With the proliferation of location-aware mobile devices, people are also finding it increasingly easy to add spatial information to other data. We can also easily add a geographic dimension to other data based on the address and related information. As expected, R has packages to process and visualize geospatial data and therefore processing geospatial data has come within easy reach of most people. The `sp`, `maptools`, `maps`, `rgdal`, and `RgoogleMaps` packages have the necessary features. In this chapter, we cover recipes to perform the most common operations that most people will need: getting geospatial data into R and visualizing such data. People who need to perform more advanced operations should consult books dedicated to the topic.

## Downloading and plotting a Google map of an area

You can use the `RgoogleMaps` package to get and plot Google maps of specific areas based on latitude and longitude. This approach offers tremendous ease of use. However, we do not gain much control over the map elements and how we plot the maps. For fine control, you can use some of the later recipes in this chapter.

### Getting ready

Install the `RgoogleMaps` package using the following command:

```
install.packages("RgoogleMaps")
```

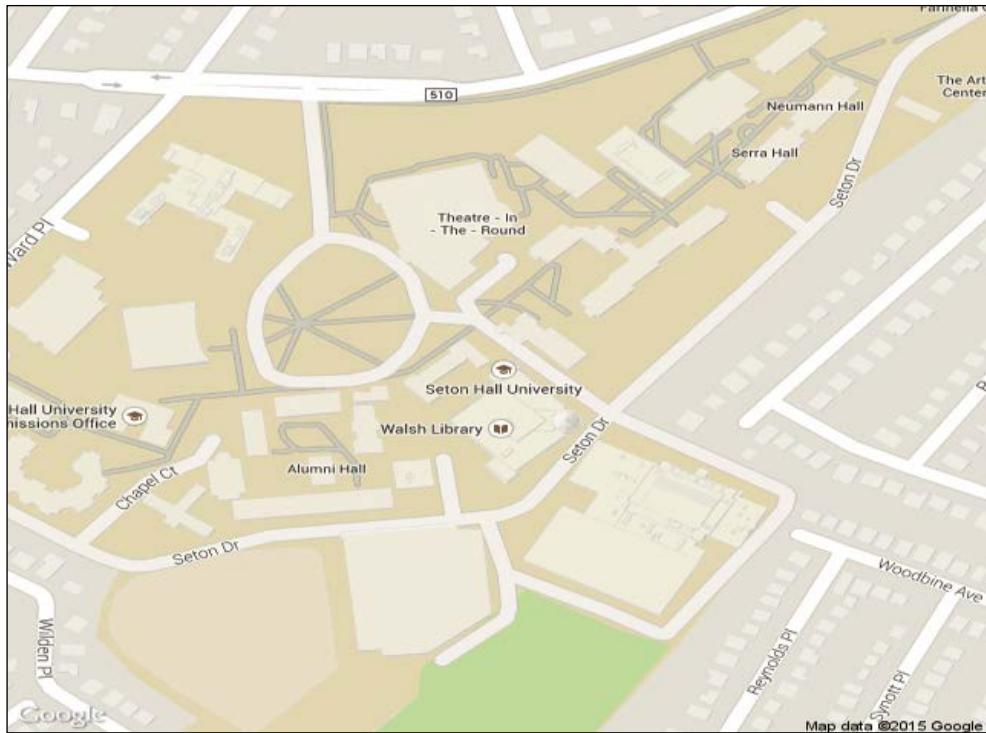
### How to do it...

To download and use Google maps of an area, follow these steps:

1. Load the `RgoogleMaps` package:  

```
> library(RgoogleMaps)
```
2. Determine the latitude and longitude of the location for which you need a map.  
In this recipe, you will get the map for the neighborhood of Seton Hall University in New Jersey, USA. The location is: (lat, long) = (40.742634, -74.246215).
3. Get the static map from Google Maps and then plot it as follows:  

```
> shu.map <- GetMap(center = c(40.742634, -74.246215),
+ zoom=17)
> PlotOnStaticMap(shu.map)
```



## How it works...

In step 1 we load the `RgoogleMaps` package.

In step 2 the latitude and longitude of the location for which we want a map are determined.

Having determined the latitude and longitude of the location, step 3 uses the `GetMap` function to acquire and store the map in an R variable called `shu.map`. The `zoom` option controls the zoom level of the returned map. The `zoom=1` option gives the whole world, and `zoom=17` covers a square area approximately a quarter of a mile on each side.

In step 3 the `PlotOnStaticMap` function is used to plot the map.

## There's more...

In the main recipe, we acquired and stored the map in an R variable. However, we can store it as an image file as well. We also have several options for the kind of map we can download.

## Saving the downloaded map as an image file

Use the `destfile` option to save the downloaded map as an image file:

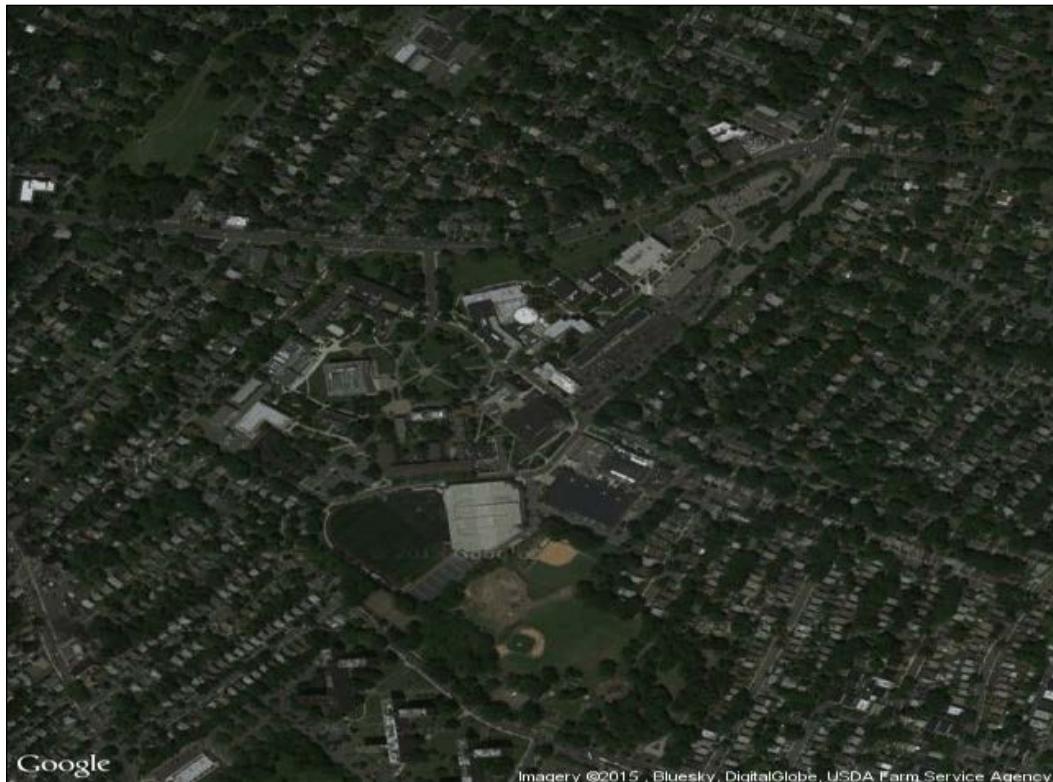
```
> shu.map = GetMap(center = c(40.742634, -74.246215),
+ zoom=16, destfile = "shu.jpeg", format = "jpeg")
```

`GetMap` also supports other formats such as `png` and `gif`. See the help file for more details.

## Getting a satellite image

By default, `GetMap` returns a road map. You can use the `maptype` argument to control what is returned as follows:

```
> shu.map = GetMap(center = c(40.742634, -74.246215), zoom=16,
+ destfile = "shu.jpeg", format = "jpeg", maptype = "satellite")
> PlotOnStaticMap(shu.map)
```



`GetMap` supports other map types such as `roadmap` and `terrain`. See the help file for details.

## Overlaying data on the downloaded Google map

In addition to plotting static Google maps, RgoogleMaps also allows you to overlay your own data points on static maps. In this recipe, we will use a data file with wage and geospatial information to plot a Google map of the general area covered by the data points; we will then overlay the wage information. The RgoogleMaps package offers tremendous ease of use but does not allow you control over the map elements and how you plot the maps. For fine control, you can use some of the later recipes in this chapter.

### Getting ready

Install the RgoogleMaps package. If you have not already downloaded the `nj-wages.csv` file, do it now and ensure that it is in your R working directory. The file contains information downloaded from the New Jersey Department of Education mashed up with latitude and longitude information downloaded from <http://federalgovernmentzipcodes.us>.

### How to do it...

To overlay data on the downloaded Google map, follow these steps:

1. Load RgoogleMaps and read the data file:

```
> library(RgoogleMaps)
> wages <- read.csv("nj-wages.csv")
```

2. Convert the wages into quantiles for ease of plotting:

```
> wages$wgclass <- cut(wages$Avgwg, quantile(wages$Avgwg,
 probs=seq(0,1,0.2)), labels=FALSE, include.lowest=TRUE)
```

3. Create a color palette:

```
> pal <- palette(rainbow(5))
```

4. Attach the data frame:

```
> attach(wages)
```

5. Get the Google map for the area covered by the data:

```
> MyMap <- MapBackground(lat=Lat, lon=Long)
```

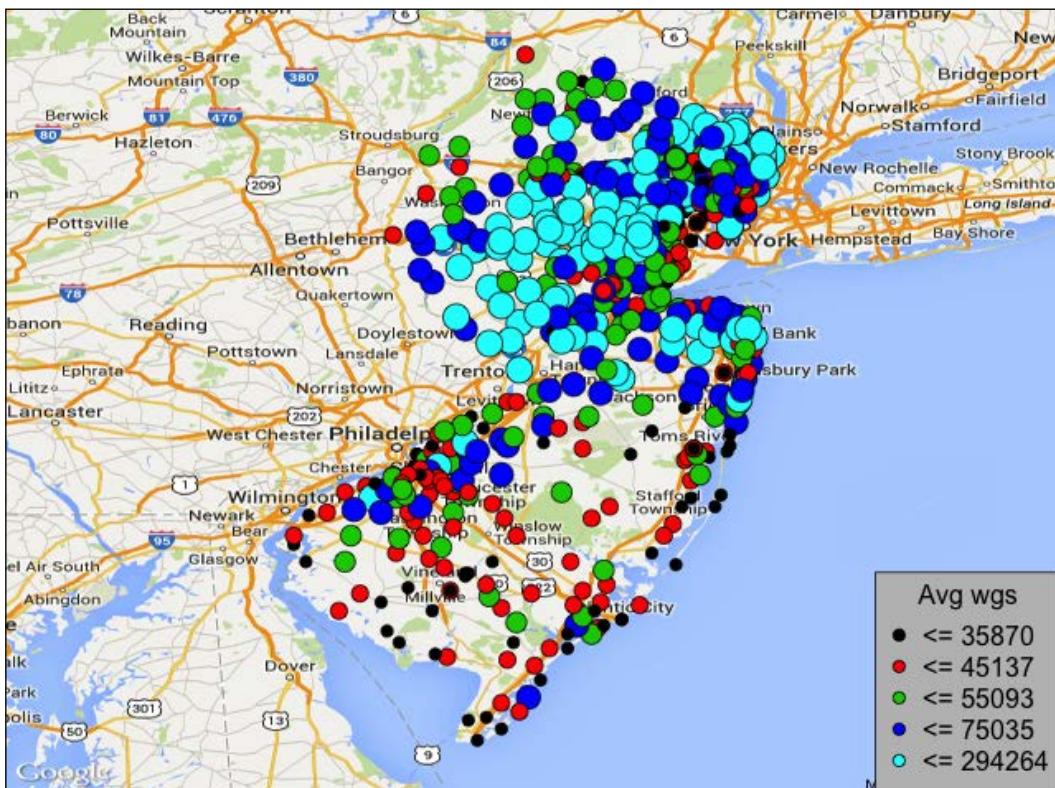
```
[1] "http://maps.google.com/maps/api/staticmap?center=40.115,-
74.715&zoom=8&size=640x640&maptype=mobile&format=png32&sensor=true"
center, zoom: 40.115 -74.715 8
```

6. Plot the map with the average wages overlaid with color and size proportional to the quintile:

```
> PlotOnStaticMap(MyMap, Lat, Long, pch=21, cex =
 sqrt(wgclass), bg=pal[wgclass])
```

7. Add a legend:

```
> legend("bottomright", legend=paste("<=",
 round(tapply(Avgwg, wgclass, max))), pch=21, pt.bg=pal,
 pt.cex=1.0, bg="gray", title="Avg wgs")
```



## How it works...

In step 1 the RgoogleMaps package is loaded and the data file is read. The file has geographic and other data for several school districts in New Jersey. We aim to show a Google map of the general area and to overlay the average wages for each school district on the map.

In step 2 the `cut` function is used on the `Avgwg` column to create a new column called `wgclass`. This column represents the quintile to which a school district belongs.

In step 3 a color palette is created with five colors—one for each quantile.

In step 4 the `wages` data frame is attached to ease variable references.

In step 5 the `MapBackground` function is used to get the static Google map for the general area. We pass all the latitudes and longitudes, and the `MapBackground` function uses these to determine the overall extent of the map.

In step 6 the `PlotOnStaticMap` function is used to plot the map from step 5. Apart from plotting the static map from step 5, this step also plots the individual points because the call passes the latitudes and longitudes as the second and third arguments to the call. The other arguments play the following roles:

- ▶ `pch` determines the character used to plot each point
- ▶ `cex` determines the size of each point based on its quantile
- ▶ `bg` determines the background color of each point based on its quantile

In step 7 we add a legend by calling the `legend` function. The arguments work as follows:

- ▶ The first argument determines the position of the legend.
- ▶ The `legend` function provides the vector of text for the legend. It creates the vector by finding the maximum `Avgwg` value for each wage class.
- ▶ As before, `pch` determines the character used to plot each point.
- ▶ The `pt.bg` argument determines the palette applied to the background color for the legend points.
- ▶ The `pt.cex` argument determines the size of the legend points.
- ▶ The `bg` argument determines the background color for the legend as a whole.
- ▶ The `title` argument specifies the title for the legend.

## Importing ESRI shape files into R

Several organizations make ESRI shape files freely available, and you can adapt them for your purposes. Using `RgoogleMaps` is easy, and we have seen that it offers very little control over map elements and plotting. Importing shape files, on the other hand, gives us total control. We should prefer this approach when we need fine control over the rendering of individual elements rather than just plotting a map image as a whole. The `rgdal` package offers the functionality to download shape files into R in a format that the `sp` package can handle.

## Getting ready

Install the `rgdal` and `sp` packages. At the time of writing, installing `rgdal` on Mac OS X is tricky. Binary packages are unavailable and different versions of the OS require us to do different things. You will need to research this on the web and get it installed.

Copy the following files to your R working directory:

- ▶ `ne_50m_admin_0_countries.shp`
- ▶ `ne_50m_admin_0_countries.prj`
- ▶ `ne_50m_admin_0_countries.shx`
- ▶ `ne_50m_admin_0_countries.VERSION.txt`
- ▶ `ne_50m_airports.shp`
- ▶ `ne_50m_airports.prj`
- ▶ `ne_50m_airports.shx`,
- ▶ `ne_50m_airports.VERSION.txt`

We obtained these files from <http://www.naturalearthdata.com/>.

## How to do it...

To import ESRI shape files into R, follow these steps:

1. Load the `sp` and `rgdal` packages:

```
> library(sp)
> library(rgdal)
```

2. Read the ESRI file of countries:

```
> countries_sp <- readOGR(".", "ne_50m_admin_0_countries")

OGR data source with driver: ESRI Shapefile
Source: ".", layer: "ne_50m_admin_0_countries"
with 241 features and 63 fields
Feature type: wkbPolygon with 2 dimensions

> class(countries_sp)

[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
```

## 3. Read the ESRI file of airports:

```
> airports_sp <- readOGR(".", "ne_50m_airports")

OGR data source with driver: ESRI Shapefile
Source: ".", layer: "ne_50m_airports"
with 281 features and 10 fields
Feature type: wkbPoint with 2 dimensions

> class(airports_sp)

[1] "SpatialPointsDataFrame"
attr(, "package")
[1] "sp"
```

**How it works...**

In step 1 the `sp` and `rgdal` packages are loaded.

In step 2 the `readOGR` function from the `rgdal` package is used to read the `ne_50m_admin_0_countries.shp` shape file layer. An ESRI shape file comes in layers with all files in a layer having the same filename and different filename extensions. Each file contains some information about the map in a layer. The first argument to the `readOGR` function specifies the **dsn (data source name)**, or a directory containing the layer, and the second argument specifies the layer to be read.

The result of step 2 shows that `readOGR` returns an object of the `SpatialPolygonsDataFrame` class. The `sp` package defines several spatial classes including `SpatialPolygonsDataFrame`. This class stores spatial information for each country as a polygon, and additionally has nonspatial attributes for each country stored in a slot called `data`. Effectively, a `SpatialPolygonsDataFrame` object is a spatial object (a collection of polygons) embellished with nonspatial attributes.

Step 3 uses the `readOGR` function to read another layer called `ne_50m_airports`. Examining the class of this object reveals it to be a `SpatialPointsDataFrame` object. Like `SpatialPolygonsDataFrame`, a `SpatialPointsDataFrame` object is also a spatial object (a collection of points) embellished with nonspatial attributes.

## Using the `sp` package to plot geographic data

The `sp` package has the necessary features to store and plot geographic data. In this recipe, we will use the `sp` package to plot imported shape files.

### Getting ready

Install the packages `rgdal` and `sp`. If you have issues installing the `rgdal` package on Mac or Linux, refer to the earlier recipe for details.

Copy the following files to your R working directory:

- ▶ `ne_50m_admin_0_countries.shp`
- ▶ `ne_50m_admin_0_countries.prj`
- ▶ `ne_50m_admin_0_countries.shx`
- ▶ `ne_50m_admin_0_countries.VERSION.txt`
- ▶ `ne_50m_airports.shp`
- ▶ `ne_50m_airports.prj`
- ▶ `ne_50m_airports.shx`
- ▶ `ne_50m_airports.VERSION.txt`

We obtained these files from <http://www.naturalearthdata.com/>.

### How to do it...

To plot geographic data using the `sp` package, follow these steps:

1. Load the `sp` and `rgdal` packages:

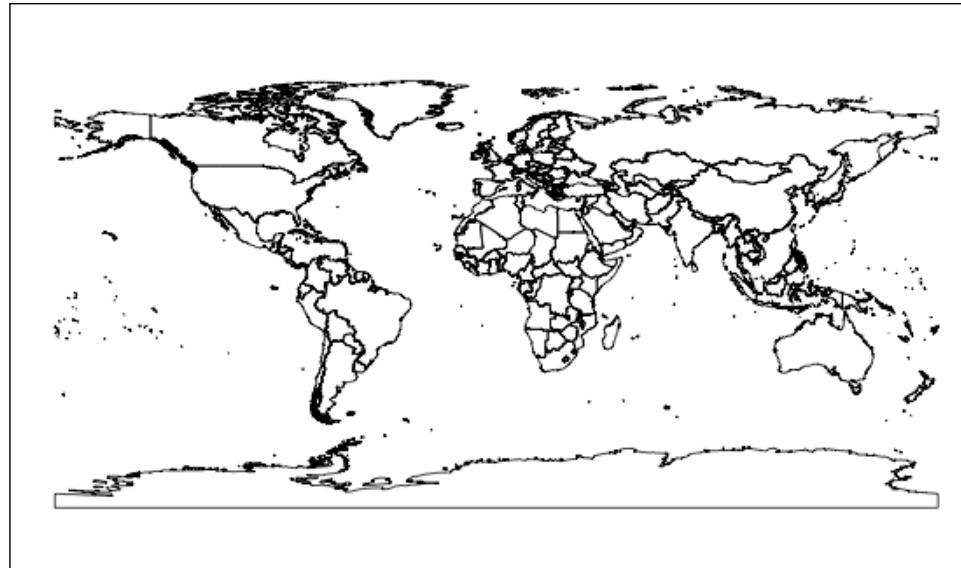
```
> library(sp)
> library(rgdal)
```

2. Read the data:

```
> countries_sp <- readOGR(".", "ne_50m_admin_0_countries")
> airports_sp <- readOGR(".", "ne_50m_airports")
```

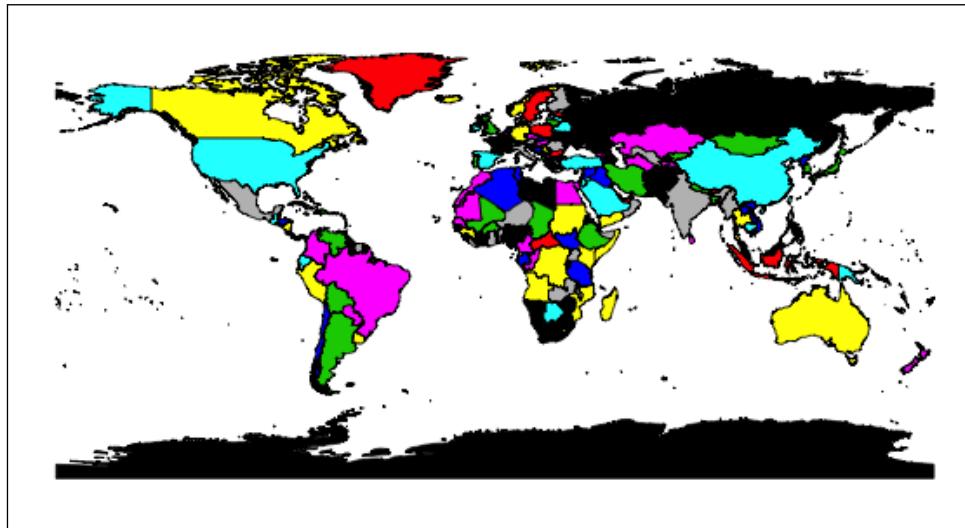
3. Plot the countries without color:

```
> # without color
> plot(countries_sp)
```



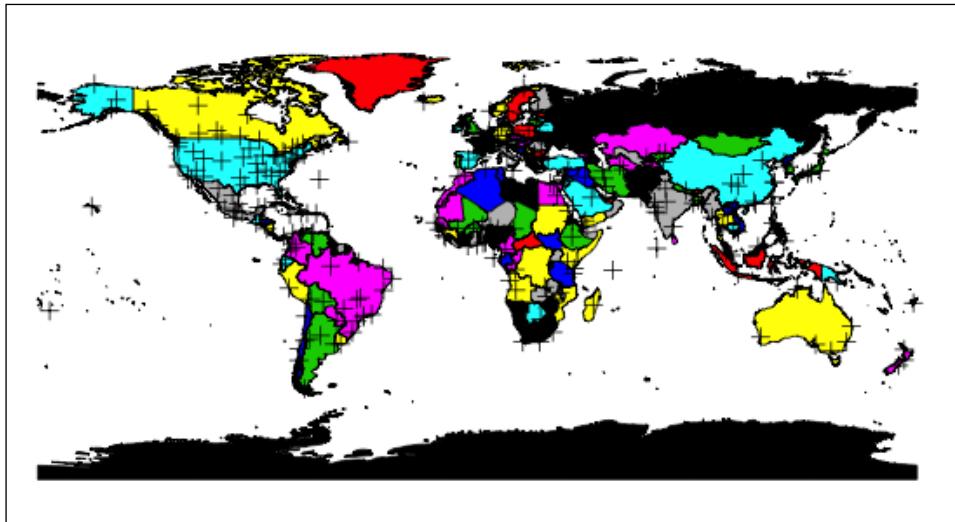
4. Plot the countries with color:

```
> # with color
> plot(countries_sp, col = countries_sp@data$admin)
```



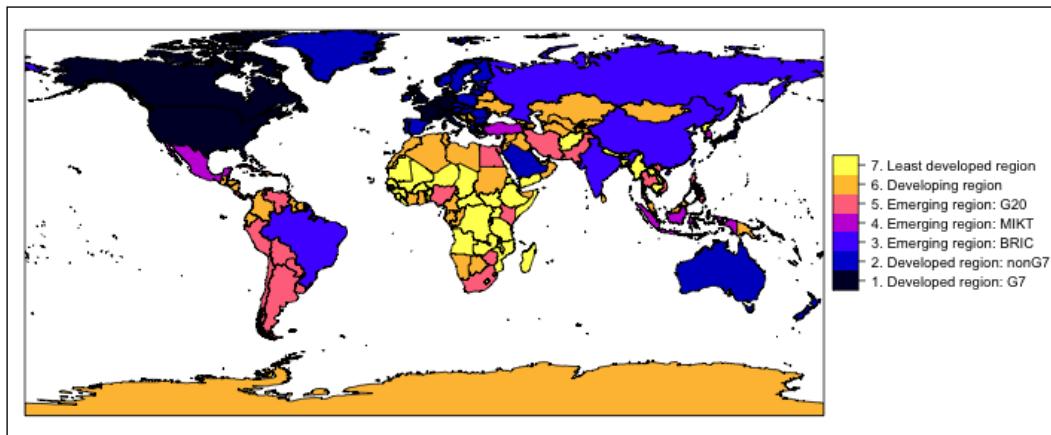
5. Add the airports. Do not close the previous plot:

```
> plot(airports_sp, add=TRUE)
```



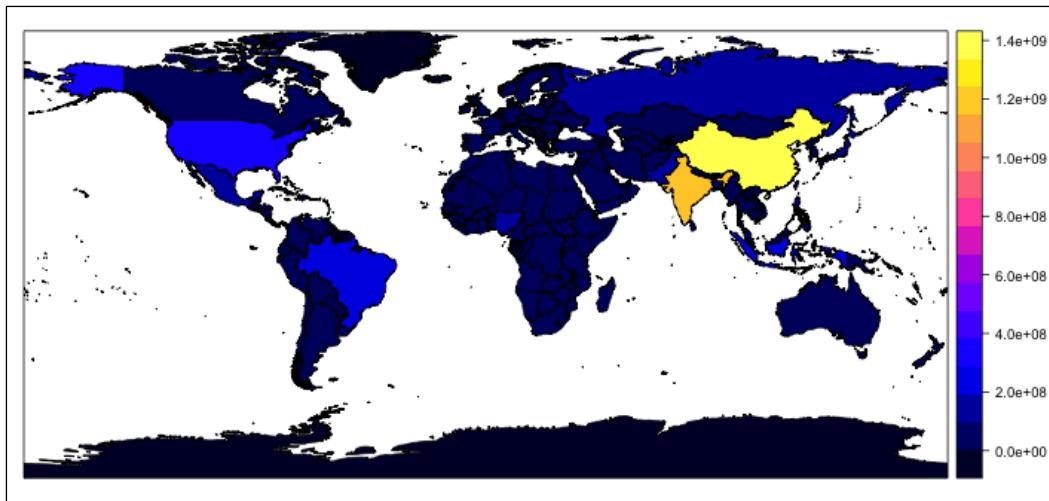
6. Plot the economic level (factor):

```
> spplot(countries_sp, c("economy"))
```



7. Plot the population (numeric):

```
> spplot(countries_sp, c("pop_est"))
```



### How it works...

If you have not already done so, you should read the recipe *Importing ESRI shape files into R* from this chapter.

In step 1 the `sp` and `rgdal` packages are loaded.

In step 2 `readOGR` is used to read the ESRI shape files of countries and airports.

Step 3 shows how to plot the countries without color using the `plot` function. The `plot` function plots several polygon objects in `countries_sp`.

Step 4, similar to step 3, plots countries but adds color to them.

Step 5 adds the airport information using the `plot` function, with the `add=TRUE` option. The `airports_sp` object contains several points, and the `plot` function plots each point with specified properties such as plot character and size.

In steps 6 and 7 the use of the `spplot` function is demonstrated, which exploits the lattice plotting features. These steps show that `spplot` can handle both factors and numeric values.

## Getting maps from the maps package

The maps package has several pre-built maps that we can download and adapt. This recipe demonstrates the capabilities of these maps.

### Getting ready

Install the `maps` package.

### How to do it...

To get maps from the `maps` package, follow these steps:

1. Load the `maps` package:

```
> library(maps)
```

2. Plot the world map:

```
> # with country boundaries
> map("world")
> # without country boundaries
> map("world", interior=FALSE)
```

3. Plot the world map with colors:

```
> map("world", fill=TRUE, col=palette(rainbow(7)))
```

4. Plot the map of a country:

```
> # for most countries, we access the map as a region on the world
map
> map("world", "tanzania")
> # some countries (Italy, France, USA) have dedicated maps that
we can directly access by name
> map("france")
> map("italy")
```

5. Plot a map of the USA:

```
> # with state boundaries
> map("state")
> # without state boundaries
> map("state", interior = FALSE)
> # with county boundaries
> map("county")
```

6. Plot a map of a state in the USA:

```
> # only state boundary
> map("state", "new jersey")
> # state with county boundaries
> map("county", "new jersey")
```

## How it works...

The fact that the `maps` package has several databases has enabled us to access several capabilities of the maps.

# Creating spatial data frames from regular data frames containing spatial and other data

When you have a regular data frame that has spatial attributes in addition to other attributes, processing them becomes easier if you convert them to full-fledged spatial objects. This recipe shows how to accomplish this.

## Getting ready

Install the `sp` package. Download the `nj-wages.csv` file and ensure that it is in your R working directory.

## How to do it...

To process a regular data frame with spatial attributes, follow these steps:

1. Load the `sp` package:

```
> library(sp)
```

2. Read the data:

```
> nj <- read.csv("nj-wages.csv")
> class(nj)
[1] "data.frame"
```

3. Convert `nj` into a spatial object:

```
> coordinates(nj) <- c("Long", "Lat")
> class(nj)

[1] "SpatialPointsDataFrame"
```

```
attr(, "package")
[1] "sp"
```

4. Plot the points:

```
> plot(nj)
```



5. Convert the points to lines and plot them:

```
> nj.lines <-
 SpatialLines(list(Lines(list(Line(coordinates(nj)))),
 "linenj")))
> plot(nj.lines)
```



## How it works...

In step 1 the `sp` package is loaded.

In step 2 the data file is read, showing that `nj` is now a regular data frame object.

In step 3 the `Lat` and `Long` variables are identified from the `nj` data frame as spatial coordinates through the `coordinates` function. We see that `nj` has now been transformed into a `SpatialPointsDataFrame` object—a full-fledged spatial object.

## Creating spatial data frames by combining regular data frames with spatial objects

Often we have data that has some geographical aspect to it (such as postal codes) but does not have sufficient geographic coordinate information for plotting. In order to display such information on a map representation, we will need to embellish the basic data with enough geographic coordinate information for plotting. The `sp` package has several `SpatialXXXDataFrame` classes to represent geographic information along with additional descriptive data. This recipe shows how we can create and plot such objects. In this recipe, we demonstrate how to get a map from the `maps` package and convert it into a `SpatialPolygons` object. We then add data from a normal data frame to create a `SpatialPolygonsDataFrame` object, which we then plot.

## Getting ready

Install the `sp`, `maps`, and `maptools` packages. Download the `nj-county-data.csv` file into your R working directory.

## How to do it...

In this recipe we demonstrate how to get a map from the `maps` package, convert it into a `SpatialPolygons` object, and then add on data from a normal data frame to create a `SpatialPolygonsDataFrame` object, which we then plot. To do this, follow these steps:

1. Load the packages needed:

```
> library(maps)
> library(maptools) # this also loads the sp package
```

2. Get the county map of New Jersey:

```
> nj.map <- map("county", "new jersey", fill=TRUE,
 plot=FALSE)
> str(nj.map)
```

```
List of 4
$ x : num [1:774] -75 -74.9 -74.9 -74.7 -74.7 ...
$ y : num [1:774] 39.5 39.6 39.6 39.7 39.7 ...
$ range: num [1:4] -75.6 -73.9 38.9 41.4
$ names: chr [1:21] "new jersey,atlantic" "new jersey,bergen"
"new jersey,burlington" "new jersey,camden" ...
- attr(*, "class")= chr "map"

3. Extract the county names:
> county_names <- sapply(strsplit(nj.map$names, ","),
 function(x) x[2])

4. Convert the map to SpatialPolygon:
> nj.sp <- map2SpatialPolygons(nj.map, IDs = county_names,
 proj4string = CRS("+proj=longlat +ellps=WGS84"))
> class(nj.sp)

[1] "SpatialPolygons"
attr(,"package")
[1] "sp"

5. Create a regular data frame from the file:
> nj.dat <- read.csv("nj-county-data.csv")

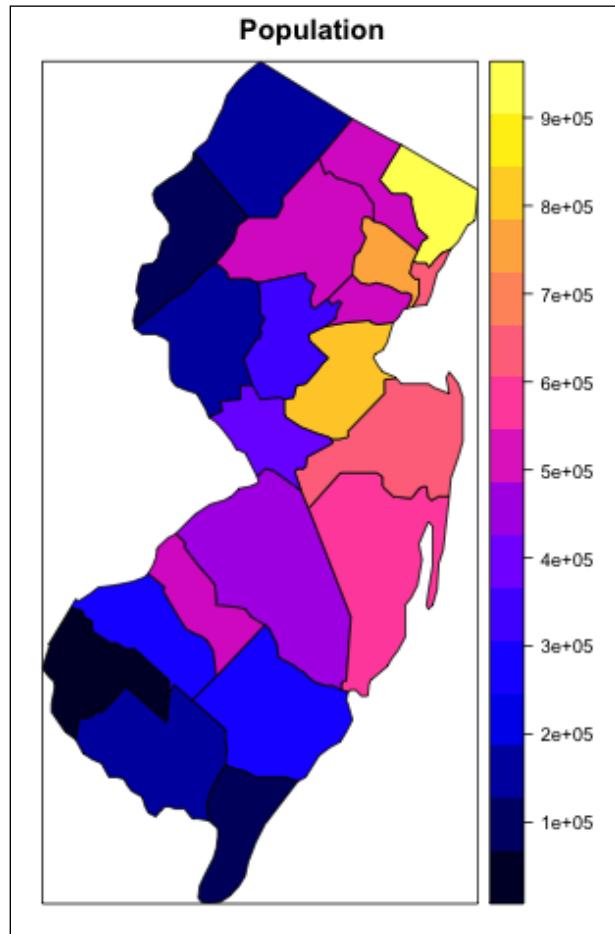
6. Create row names to match those in the map:
> rownames(nj.dat) <- nj.dat$name

7. Create SpatialPolygonsDataFrame:
> nj.spdf <- SpatialPolygonsDataFrame(nj.sp, nj.dat)
> class(nj.spdf)

[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"

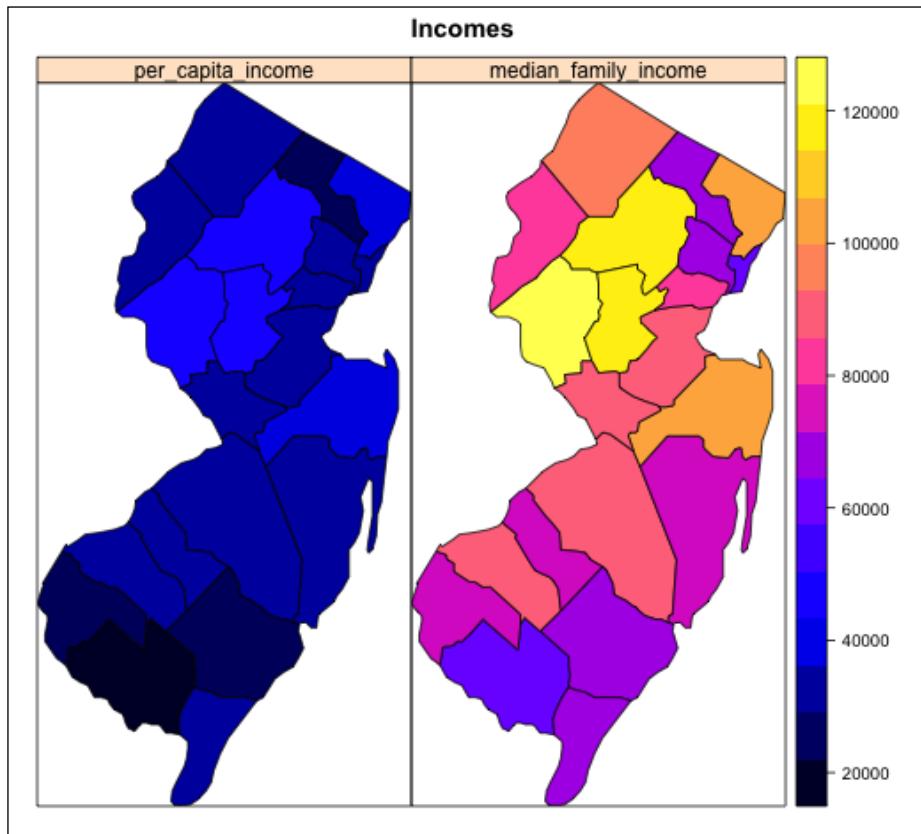
8. Plot the map:
> # plain plot of the object
> plot(nj.spdf)
```

```
> # Plot of population:
> spplot(nj.spdf, "population", main = "Population")
```



9. Based on incomes, a comparison can be obtained between per capita income and median family income:

```
> spplot(nj.spdf,
 c("per_capita_income", "median_family_income"),
 main = "Incomes")
```



### How it works...

In step 1 the `maps`, `maptools`, and `sp` packages are loaded.

In step 2 the `Map` function in the `maps` package is used to get the county map of New Jersey.

From the `maps` package, we can get maps as lines or as polygons. To color regions (such as countries on a world map or states or counties on country maps) based on their data values, we need to have the regions represented as polygons. The `fill` parameter controls whether or not we get a map as lines or as polygons. We used the `fill = TRUE` option to get the map as polygons.

We will first convert the map into a `SpatialPolygons` object and then add on nonspatial attribute values to make it a `SpatialPolygonsDataFrame` object.

Every polygon in a `SpatialPolygons` object must have a unique ID. From the output generated by step 2, we see that the individual regions (polygons, corresponding to the counties) in the map have names like `new_jersey, atlantic`.

In step 3 just the county names from the region names in the map are extracted by applying the `strsplit` function to each of the region names. We use the extracted county names as identifiers for the polygons. To combine spatial data with normal data frames, identifiers of polygons are matched with the row names of regular data frames. This is why we need to assign identifiers for the polygons.

In step 4 the `map2SpatialPolygons` function from the `maptools` package is used to generate a `SpatialPolygons` object `nj.sp` from the map `nj.map`. This function uses the `IDs` argument supplied to name the polygons in the resultant `SpatialPolygons` object. If the length of the `IDs` argument does not match the number of polygons, then the function generates an error. At this point, we have a spatial object without any nonspatial attributes. Map files have geographic coordinate information in many different formats. The `proj4string` argument indicates the kind of coordinate information by creating a **Coordinate Reference System (CRS)** object. In the current example, we indicate that the coordinates are represented as longitudes and latitudes, and that the **World Geodetic System 1984 (WGS84)** standard is used. Depending on the coordinates in the map, other CRS objects may need to be created.

In step 5, data on the counties in New Jersey is read from a file and a normal data frame `nj.dat` is created. This data frame has no spatial attributes. We want to add the attributes from this data frame to the `SpatialPolygons` `nj.sp` to create a `SpatialPolygonsDataFrame` object.

In step 6 the county names are assigned as the row names for the data frame. We will shortly see why.

In step 7 the `SpatialPolygonsDataFrame` function is used to combine spatial and nonspatial information into a single `SpatialPolygonsDataFrame` object. The function uses the `SpatialPolygons` object `nj.sp` as well as the `nj.dat` data frame. It matches both objects by matching the row names in the data frame with the polygon IDs in the `SpatialPolygons` objects. This is why we assigned the county names as the row names in step 6 and also generated the county names in step 3. At this point, we have a `SpatialPointsDataFrame` object `nj.spdf` that contains both spatial and nonspatial information.

Step 8 shows that a regular plot of the `SpatialPointsDataFrame` object with the `plot` function displays only spatial information.

In step 9 the `spplot` function is used to plot the data and color each county based on its population. The last plot from step 9 clearly shows that `spplot` is based on the `lattice` package.

## Adding variables to an existing spatial data frame

This recipe shows how you can add variables to spatial data frame objects. One approach (see the recipe *Creating spatial data frames by combining regular data frames with spatial objects*, earlier in this chapter) will be to create all the necessary variables before creating the spatial data frame object. However, this might not always be feasible. This recipe shows how you can add nonspatial variables to an existing spatial data frame object.

### Getting ready

Install the `sp`, `maps`, and `maptools` packages. Download and place the `nj-county-data.csv` file in your R working directory.

### How to do it...

To add variables to an existing spatial data frame, follow these steps:

1. Follow the following steps shown (from the recipe *Creating spatial data frames by combining regular data frames with spatial objects*, earlier in this chapter):

```
> library(maps)
> library(maptools)
> nj.map <- map("county", "new jersey", fill=T, plot=FALSE)
> county_names <- sapply(strsplit(nj.map$names, ","),
+ function(x) x[2])
> nj.sp <- map2SpatialPolygons(nj.map, IDs = county_names,
+ proj4string = CRS("+proj=longlat +ellps=WGS84"))
> nj.dat <- read.csv("nj-county-data.csv")
> rownames(nj.dat) <- nj.dat$name
> nj.spdf <- SpatialPolygonsDataFrame(nj.sp, nj.dat)
```

2. Compute the population density for each county:

```
> pop_density <-
 nj.spdf@data$population/nj.spdf@data$area_sq_mi
```

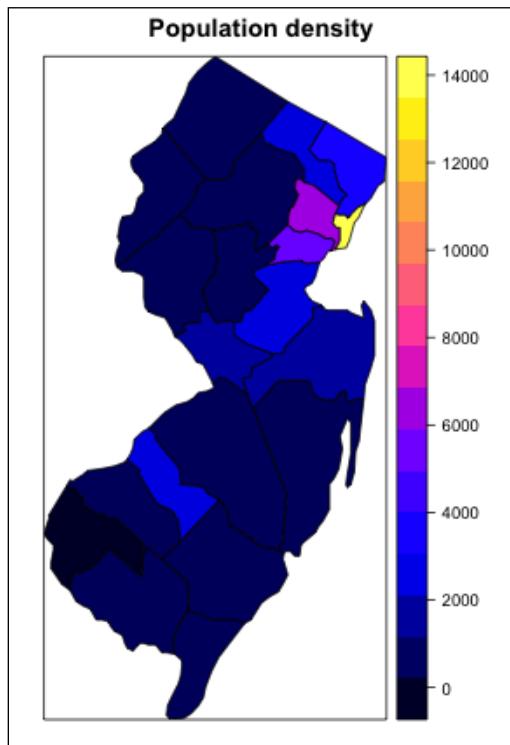
3. Add a new variable to `nj.spdf`:

```
> nj.spdf <- spCbind(nj.spdf, pop_density)
> names(nj.spdf@data)
```

```
[1] "name" "per_capita_income"
[3] "median_household_income" "median_family_income"
[5] "population" "no_households"
[7] "area_sq_mi" "pop_density"
```

4. Plot the data:

```
> spplot(nj.spdf, "pop_density")
```



### How it works...

In step 1 the code from the recipe *Creating spatial data frames by combining regular data frames with spatial objects* is repeated to create the `SpatialPointsDataFrame` object of New Jersey with county level data.

In step 2 the underlying data frame is accessed through the `nj.spdf@data` variable and computes the population density based on the `population` and `area_sq_mi` variables.

In step 3 the `maptools` package method, `spCbind`, is used to add the new variable to the underlying data frame in the `SpatialpointsDataFrame` object `nj.spdf`.

In step 4 the new variable is then plotted.



# 11

## Playing Nice – Connecting to Other Systems

In this chapter, we will cover the following recipes to connect to other systems:

- ▶ Using Java objects in R
- ▶ Using JRI to call R functions from Java
- ▶ Using Rserve to call R functions from Java
- ▶ Executing R scripts from Java
- ▶ Using the xlsx package to connect to Excel
- ▶ Reading data from relational databases – MySQL
- ▶ Reading data from NoSQL databases – MongoDB

### Introduction

R is an open source product and hence its capabilities are constantly expanding. R is specifically useful for its numerous statistical packages and powerful visualization. When applications written in other environments (such as Java, C++, Python, and Excel) need to exploit R's special capabilities, we need to smoothly integrate these environments with R. In this chapter, we will discuss working with R from Java and Excel and reading data from databases.

The `rJava` package allows us to create and access Java objects using **Java Native Interface (JNI)** from within R. The **Java-R Interface (JRI)** and **Rserve** packages allow us to do the reverse by invoking R from within Java programs. We also discuss various ways to work on Excel files directly from R.

## Using Java objects in R

Sometimes, we develop parts of an application in Java and need to access them from R. The `rJava` package allows us to access Java objects directly from within R.

### Getting ready

If you have not already downloaded the files for this chapter, do it now and ensure that these files are in your R working directory:

1. Create a folder called `javasamples` and move all the files with extension `.java` or `.class` into this folder under your working directory.
2. Install `rJava` using the `install.packages ("rJava")` command.
3. Load the package using the `library (rJava)` command.
4. For `rJava` to work in your environment, the JDK version should be identical for the following, and we explain how to get them in sync for Mac OS X:
  - ❑ The environment JDK version: Execute `java -version` in your command line to get the installed version of Java. You will be using this version to create `.jar` files or to compile Java programs.
  - ❑ The JDK version in R: After you install and load the `rJava` package, check the JVM version in the R environment. We execute the commands to check this in the previous step. This should match with the response you get in the `java -version` command.
5. If there is a mismatch in the versions and you are using Mac OS X, do the following to install the latest version of `rJava` from source:
  1. Download the latest `rJava` source `rJava_0.9-7.tar.gz` from <http://www.rforge.net/rJava/files/>. The filename can be different with each new version of `rJava`:

```
sudo R CMD javareconf
```
  2. Include the following lines in your shell profile file (`.bash_profile` in bash, `.profile` in csh, and so forth); make sure to change the following folders as per your environment:

```
export JAVA_HOME="/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/
```

```
export LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/server
export MAKEFLAGS="LDFLAGS=-Wl,-rpath $JAVA_HOME/lib/server"
```

3. Close your terminal window and reopen it so that the profile settings take effect.
  4. Install the downloaded rJava package and make sure to change the filename:  

```
sudo R CMD INSTALL rJava_0.9-7.tar.gz
```
  5. Close R or RStudio session, whichever you are using, and reopen it from the same terminal window by executing `open -a R` or `open -a RStudio`.
  6. Load the library again using `library(rJava)`.
  7. Check the JVM version using step 1.
6. Download the three JAR files `JRI.jar`, `REngine.jar`, and `JRIEngine.jar` from <http://www.rforge.net/JRI/files/>; the `RserveEngine.jar` from <http://www.rforge.net/Rserve/files/>. Copy the four downloaded JAR files to the `lib` folder under your R working directory.
  7. You can either use the class files provided or compile the Java code. These class files are created with JDK 1.8.0\_25, and if your JDK version is different, follow the next step to compile all the Java programs.
  8. To compile the downloaded Java programs, go to the `javasamples` folder and execute the `javac -cp ../../lib/* *java` command. You should see files with the `.class` extensions for each of the downloaded Java programs.

## How to do it...

To use Java objects in R, follow these steps:

1. From within R, start the JVM, check the Java version, and set `classpath`:

```
> .jinit()

> .jcall("java/lang/System", "S", "getProperty", "java.runtime.
version")
[1] "1.8.0_25-b17"

> .jaddClassPath(getwd())

> .jclassPath()
[1] "/Library/Frameworks/R.framework/Versions/3.1/Resources/
library/rJava/java"
[2] "/Users/sv/book/Chapter11" => my working directory
```

2. Perform these Java string operations in R:

```
> s <- .jnew("java/lang/String", "Hello World!")
> print(s)
[1] "Java-Object{Hello World!}"

> .jstrVal(s)
[1] "Hello World!"

> .jcall(s, "S", "toLowerCase")
[1] "hello world!"

> .jcall(s, "S", "replaceAll", "World", "SV")
[1] "Hello SV!"
```

3. Perform these Java vector operations:

```
> javaVector <- .jnew("java/util/Vector")
> months <- month.abb

> sapply(months, javaVector$add)
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
TRUE TRUE
TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

> javaVector$size()
[1] 12

> javaVector$toString()
[1] "[Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec]"
```

4. Perform these Java array operations:

```
> monthsArray <- .jarray(month.abb)
> yearsArray <- .jarray(as.numeric(2010:2015))
> calArray <- .jarray(list(monthsArray, yearsArray))

> print(monthsArray)
[1] "Java-Array-Object [Ljava/lang/String;:[Ljava.lang.String;@lff4689e"

> .jevalArray(monthsArray)
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct"
"Nov" "Dec"
```

```
> print(l <- .jevalArray(calArray))
[[1]]
[1] "Java-Object{ [Ljava.lang.String;@30f7f540}"

[[2]]
[1] "Java-Object{ [D@670655dd}"

> lapply(l, .jevalArray)
[[1]]
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct"
"Nov" "Dec"

[[2]]
[1] 2010 2011 2012 2013 2014 2015
```

5. Insert this simple Java class HelloWorld:

```
> hw <- .jnew("javasamples.HelloWorld")
> hello <- .jcall(hw, "S", "getString")
> hello
[1] "Hello World"
```

6. Insert this simple Java class Greeting with a method that accepts an argument:

```
> greet <- .jnew("javasamples.Greeting")
> print(greet)
[1] "Java-Object{Hi World!}"

> g <- .jcall(greet, "S", "getString", "Shanthi")
> print(g)
[1] "Hello Shanthi"

> .jstrVal(g)
[1] "Hello Shanthi"
```

## How it works...

The `.jinit()` initializes the **Java Virtual Machine (JVM)** and needs to be executed before invoking any of the `rJava` functions. If you encounter errors at this point, the issue is usually a lack of sufficient memory. Close unwanted processes or programs, including R, and retry.

For rJava to work, we need to sync up the Java version in the system environment with the rJava version. We used the `.jcall("java/lang/System", "S", "getProperty", "java.runtime.version")` command to get the Java version within the R environment.

After making sure that the Java versions are identical, the first thing we need to do to access any Java object is to set up `classpath`. We do this using `.jaddClassPath`. We pass the R working directory, since our Java classes reside here. However, if you have the Java class files in a different location or if you created a `.jar` file, include that location instead. Once the `classpath` is set using `.jaddClassPath`, you can verify it by executing `.jclassPath()`.

Step 2 illustrates string operations. We use `.jnew` to instantiate any Java object. The `classname` is the full classname separated by `/`. Hence, we refer to the string class as `java/lang/String` instead of `java.lang.String`.

The `jstrVal` function emits the equivalent of `toString()` for any Java object. In our example, we get the content of string `s`.

We use `.jcall` to execute any method on a Java object. In `jcall(s, "S", "toLowerCase")`, we are invoking the `toLowerCase` method on the string object `s`. The `"S"` in the call specifies the return type of the method invocation. In `.jcall(s, "S", "replaceAll", "World", "SV")`, we invoke the `replaceAll` method and get a new replaced string back.

We list the possible return types in the following table:

| Return Type | Java Type                    | Remarks                            |
|-------------|------------------------------|------------------------------------|
| I           | int                          |                                    |
| D           | double                       |                                    |
| J           | long                         |                                    |
| F           | float                        |                                    |
| V           | void                         |                                    |
| Z           | boolean                      |                                    |
| C           | char                         |                                    |
| B           | byte (raw)                   |                                    |
| L<class>    | Java object of class <class> | Eg: Ljava.awt.Component            |
| S           | java.lang.String             | S is special for Ljava/long/object |
| [<type>]    | array of objects of <type>   | [D for array of doubles            |

Step 3 illustrates vector operations in Java from R. We first create a Java vector object using `javaVector <- .jnew("java/util/Vector")`. We then use the `add` method to add elements to this vector. Earlier in step 2, we used the `.jcall` function to invoke a method on an object, but now we use a shortcut that closely resembles what we typically do in Java. In Java, to call a method, we use the `"."` operator and in R we use the `$` operator. Thus, we use `javaVector$add` to invoke the `add` method on the `javaVector` object.

Step 4 illustrates Java array operations. The two key functions are `.jarray` to create an array object and `.jevalArray` to return an array object. We create three array objects `monthsArray`, `yearsArray`, and `calArray` using the `.jarray` function. When we print the array object using `print(monthsArray)`, we get the object type of each of the array elements. However, when we execute `.jevalArray(monthsArray)`, we get the contents of the array. The `calArray` object is a list of two Java array objects, and we also see how to extract array elements in this step.

Step 5 shows how to instantiate a custom Java object and invoke methods on it. If you have not already compiled the Java code, refer to *Getting Ready* at the beginning of this recipe for the instructions. We used `.jnew` to instantiate a `HelloWorld` object called `hw`. We always pass the classname along with the package to the `.jnew` function. Once the object is created, we can invoke methods. An example of invoking the `getString` method is shown here.

Step 6 shows the instantiation of another custom object `Greeting` and the invocation of a method. The arguments to the method follow the method name as in `.jcall(greet, "S", "getString", "Shanthi")`. Here, the string `"Shanthi"` is an argument passed to the `getString` method.

### There's more...

The following are a few key additional useful commands to invoke Java objects from the R environment.

### Checking JVM properties

You may want to check the Java Virtual Machine properties if you encounter issues in executing Java commands in the R console:

```
> jvm = .jnew("java.lang.System")
> jvm.props = jvm$getProperties()$toString()
> jvm.props <- strsplit(gsub("\\{(.*)\\}", "\\\1", jvm.props), ", "
") [[1]]
> jvm.props
[1] "java.runtime.name=Java(TM) SE Runtime Environment"
[2] "sun.boot.library.path=/System/Library/Java/
JavaVirtualMachines/1.6.0.jdk/Contents/Libraries"
[3] "java.vm.version=20.65-b04-462"
```

```
[4] "awt.nativeDoubleBuffering=true"
[5] "gopherProxySet=false"
[6] "mrj.build=11M4609"
[7] "java.vm.vendor=Apple Inc."
[8] "java.vendor.url=http://www.apple.com/"
[9] "path.separator=:"
[10] "java.vm.name=Java HotSpot(TM) 64-Bit Server VM"
....
```

## Displaying available methods

The following commands are useful to get a list of available methods or to get the method signature:

```
> .jmethods(s, "trim")
[1] "public java.lang.String java.lang.String.trim()"
```

The preceding command indicates that the `trim` method can be invoked on a `String` object and it returns a `String` object:

```
> # To get the list of available methods for an object
> .jmethods(s)
[1] "public boolean java.lang.String.equals(java.lang.Object)"
[2] "public java.lang.String java.lang.String.toString()"
[3] "public int java.lang.String.hashCode()"
[4] "public int java.lang.String.compareTo(java.lang.String)"
[5] "public int java.lang.String.compareTo(java.lang.Object)"
[6] "public int java.lang.String.indexOf(int)"
....
```

## Using JRI to call R functions from Java

The JRI allows you to execute R commands inside Java applications as a single thread. JRI loads R libraries into Java and thus provides a Java API to R functions.

### Getting ready

Make sure all the steps in the earlier recipe *Using Java objects in R* are completed.

## How to do it...

To use JRI to call R functions from Java, follow these steps:

1. Set up environment variables `R_HOME` to where R has been installed and add the R bin directory to the environment variable `PATH`.
2. Open a new terminal window (on OS X and Linux systems) or open a command prompt window on Windows. Make sure to change the values according to your environment. The following commands help to set up environment variables on OS X and Linux systems. Make sure to change your directory location:

```
export R_HOME=/Library/Frameworks/R.framework/Resources
```

```
export
PATH=$PATH:/Library/Frameworks/R.framework/Resources/bin/
```

3. Execute the Java command as follows from the `javasamples` directory. Make sure to change the values according to your environment. Also, there is no space between `-D` and `java.library.path`:

```
cd javasamples
java -Djava.library.path=/Library/Frameworks/R.framework/
Resources/library/rJava/jri -cp ../../lib/* javasamples.
SimpleJRIStat
```

1520.15

## How it works...

In step 1, we set up the environment variables `R_HOME` to where R has been installed and add the R bin directory to the environment variable `PATH`. If these environment variables are not set, then you will see the following error message:

```
"R_HOME is not set. Please set all required environment variables
before running this program.
Unable to start R."
```

In step 2, we run the `SimpleJRIStat` Java program. Open the `SimpleJRIStat.java` code:

- ▶ In the main method, we first create an instance of `Rengine` to begin an R session.
- ▶ We check to make sure that the R session is active with the `waitForR` method.
- ▶ We create an array of doubles in Java and assign it to a variable called "values". The `values` variable exists in the R environment and not in our Java environment.

- ▶ The eval method of Rengine is equivalent to executing commands in the R console. The output from the eval method is an org.rosuda.JRI.REXP object. Depending on the content of REXP, methods such as asString(), asDouble() can be executed to extract the result returned by R. In our Java code, we use the R function mean to calculate the average of the array and assign it to a Java REXP variable mean.
- ▶ We then use the asDouble method to get the value from mean and print it out.
- ▶ We finally close the R session by calling the end method.

To execute the Java code, we need to add the -Djava.library.path switch (there is no space between -D and java.library.path) and point to the rJava location. To use REngine from Java, we add the appropriate JAR files to the classpath. Since we are executing the command from the javasamples folder, we add .. to the classpath to refer to the parent folder where our library files are located under the lib folder.

## There's more...

It is possible to create graphs using R from a Java program. Let's look at SimplePlot.java:

- ▶ In the main method, first we create an Rengine instance and check if the R session is created successfully.
- ▶ We set the working directory in R either from the last argument that was passed while executing the command or from the current user directory from where the Java command was executed. The args.length == 0 expression indicates that no argument is passed during the execution of the code and hence we use the user directory as the R working directory.
- ▶ We use the read.csv function to read the file in R and load it into an R variable auto.
- ▶ We use the nrow function to get the number of rows in auto and print the value.
- ▶ We set png as device and use auto.png as filename to be created.
- ▶ We then use the plot function to plot weight vs mpg.
- ▶ We turn the device off to flush the file contents.
- ▶ We finally end the R session.

To execute the Java code, use the following command. Change the argument to the call to reflect your R working directory where the auto-mpg.csv file resides:

```
java -Djava.library.path=/Library/Frameworks/R.framework/Resources/library/rJava/jri/ -cp ../../lib/* javasamples.SimplePlot /Users/sv/book/Chapter11
```

## Using Rserve to call R functions from Java

The Rserve package is a TCP/IP server that accepts requests from clients. RServe allows other technologies to access R. Every connection has a separate workspace and working directory.

### Getting ready

If you have not already downloaded the files for this chapter, do so now and ensure that the files are in your R working directory:

- ▶ Create a `javademos` folder and move all the files with extension `Java` and `class` into this folder under your working directory.
- ▶ Install Rserve using `install.packages('Rserve')`.
- ▶ Load the package using `library(Rserve)`.
- ▶ Download the three JAR files `JRI.jar`, `REngine.jar`, and `JRIEngine.jar` from <http://www.rforge.net/JRI/files/> and the `RserveEngine.jar` from <http://www.rforge.net/Rserve/files/>. Copy the four downloaded JAR files to the `lib` folder under your R working directory.
- ▶ You can either use the class files provided or compile the Java code. These class files are created with JDK 1.8.0\_25 and, if your JDK version is different, follow the next step to compile all the Java programs.
- ▶ To compile the downloaded Java programs, go to the `javademos` folder and execute the `javac -cp ../../lib/* *java` command. You should see files with the `class` extensions for each of the downloaded Java programs.

### How to do it...

To use Rserve to call R functions from Java, follow these steps:

1. Start the Rserve server to accept client connections.

```
> Rserve(args="--no-save") - On Mac and Linux
> Rserve() - on windows
Rserv started in daemon mode.
```

2. Execute the Java program to draw `ggplot` in R and display the image. Change the argument to the call to reflect your R working directory where the `auto-mpg.csv` file resides:

```
java -cp ../../lib/* javademos.SimpleGGPlot /Users/sv/book/
Chapter11
```

## How it works...

In step 1, we start the RServer server from R. If the server is already up and running, you will see the: `##> SOCK_ERROR: bind error #48 (address already in use)` message.

You can remove the current RServer process by killing it at the OS level. We can also start Rserve as a daemon process from the command line by executing `R CMD Rserve`.

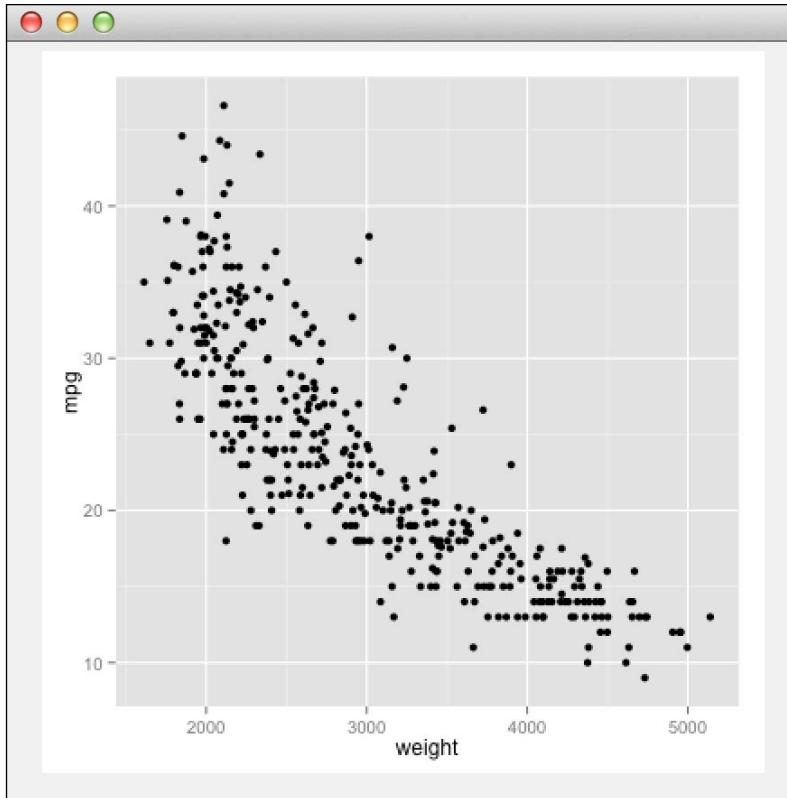
Rserve can run locally or on a remote server accessible by multiple clients. To access the remote Rserve server, provide the hostname or the IP address of the server while creating the RConnection.

In step 2, we run the SimpleGGPlot Java program. To connect to RServer from Java, we add the appropriate jars to classpath. Since we are executing the command from the javasamples folder, we add `..` to the classpath to refer to the parent folder where our library files are located under the `lib` folder. We also pass the folder name where the `auto-mpg.csv` file resides, since that folder is our R working directory.

We now explain the code in `SimpleGGPlot.java`:

- ▶ In the main method, first we create an RConnection object.
- ▶ We invoke the eval method on the RConnection object to execute commands in R.
- ▶ The RConnection object throws the REngineException and hence we add try and catch blocks to catch the exception.
- ▶ We evaluate the following functions in R from Java:
  - We first load the ggplot2 package in R.
  - We set the working directory in R using the argument passed. If no argument is passed, then the user's current directory is used to set the R working directory.
  - We read the contents of the `auto-mpg.csv` file using `read.csv`.
  - We create a device to save the graph and then generate `ggplot` for `weight vs mpg`.
  - We close the device to flush out the contents to the file.
  - We then read the binary content of the file.
  - The output of the eval or parseAndEval methods is the `org.rosuda.REngine.REXP` objects and, depending on the content of the REXP methods such as `asString()`, `asBytes()` can be executed to extract the result returned by R. In our Java code, we read the binary content of the file from the REXP object `xp` using `asBytes()`.

- We create an image object that we finally display in a JFrame as follows:



- We close the connection after the user closes the JFrame image window
- If RServe is not running, then you will see a message:

```
Exception in thread "main" org.rosuda.REngine.Rserve.
RserveException: Cannot connect: Connection refused".
```

### There's more...

- In the recipe, *Using JRI to call R functions from Java*, we showed how to execute a function in R and retrieve the value from R in Java. Here, we show how to retrieve an array from R into Java.

## Retrieving an array from R

The following steps help in retrieving an array from R:

- ▶ Open the Java program `SimpleRservStat.java`:
  - We instantiate a new `RConnection` object.
  - We assign a Java array of doubles to an R variable
  - We calculate the mean of this array in R and print it in Java
  - We then calculate the range and, since `range` is an array, we invoke a method `asDoubles()` on the `REXP` object that is returned from the `eval` method
  - We then print the array of doubles after converting it into a string
- ▶ Execute the following Java code command line from the `javasamples` directory—be sure to pass your own R working directory in place of the last part of the command:

```
java -cp ../../lib/* javasamples.SimpleRservStat /Users/sv/book/Chapter11
```

## Executing R scripts from Java

In earlier recipes, we executed R functions from within Java. In this recipe, we execute an R script from Java and read the results from R into Java for further processing.

### Getting ready

Make sure all the steps in the first recipe, *Using Java objects in R*, in this chapter are completed. Also make sure the `auto-mpg.csv` and `corr.R` files are in your R working directory.

### How to do it...

Execute the Java program from your command prompt to invoke an R script from a Java program. Be sure to change the last part to reflect your R working directory:

```
java -Djava.library.path=/Library/Frameworks/R.framework/Resources/library/rJava/jri/ -cp ../../lib/* javasamples.InvokeRScript mpg weight /Users/sv/book/Chapter11
```

## How it works...

We execute the Java program `InvokeScript` with three arguments. The first two arguments mention the columns of the `auto` table for which correlation is computed and the optional third argument is the working directory, where the `auto-mpg.csv` file and the R script reside.

Let's look at the `InvokeScript.java` code:

- ▶ In the main method, first we create an `Rengine` instance and check if the R session is created successfully.
- ▶ We check if there are at least two arguments passed to the `InvokeScript` Java program. If not, we display an error message:  
`To execute, please provide 2 variable names from auto-mpg dataset.`
- ▶ If the length of the arguments array, `args.length`, is equal to 2, we know that the user did not provide the R working directory; hence, take the user's current directory as the working directory and set it in R.
- ▶ We set two variables `var1` and `var2` from the arguments using the method `assign`. These variables are created in the R environment.
- ▶ We then invoke the `eval` method to source the R script file `corr.R`.
- ▶ We get the "result" into a `REXP` object.
- ▶ We print the value by invoking the `asDouble` method on the `REXP` object.
- ▶ Finally, we close the `Rengine` object to release the R session.

Let's look at the R script `corr.R`:

- ▶ Load the contents of the `auto-mpg.csv` file into an R object `auto`
- ▶ Execute the `cor` function to calculate the correlation between the two variables that were passed as an argument to the Java program `InvokeScript`

## Using the `xlsx` package to connect to Excel

There are multiple packages to connect Excel with R; in this recipe, we discuss the `xlsx` package. Other commonly used packages are `RExcel` and `XLConnect`.

## Getting ready

If you have not already downloaded the files for this chapter, do it now and ensure that the files are in your R working directory:

- ▶ Install `xlsx` using `install.packages("xlsx")`
- ▶ Load the library using `library(xlsx)`
- ▶ Read the data:  

```
> auto <- read.csv("auto-mpg.csv", stringsAsFactors=FALSE)
```

## How to do it...

To connect to Excel using the `xlsx` package, follow the steps:

1. Save a data frame to an Excel workbook:

```
> write.xlsx(auto, file = "auto.xlsx", sheetName =
 "autobase", row.names = FALSE)
```

2. Add two new columns to the auto data frame:

```
> auto$kmpg <- auto$mpg * 1.6
> auto$mpg_deviation <- (auto$mpg -
 mean(auto$mpg))/auto$mpg
```

3. Create Excel objects such as workbooks, worksheets, rows, and cells:

```
> auto.wb <- createWorkbook()
> sheet1 <- createSheet(auto.wb, "auto1")
> rows <- createRow(sheet1, rowIndex=1)
> cell.1 <- createCell(rows, colIndex=1) [[1,1]]
> setCellValue(cell.1, "Hello Auto Data!")
> addDataFrame(auto, sheet1, startRow=3, row.names=FALSE)
```

4. Assign styles to cells:

```
> cs <- CellStyle(auto.wb) +
 Font(auto.wb, isBold=TRUE, color="red")
> setCellStyle(cell.1, cs)
> saveWorkbook(auto.wb, "auto_wb.xlsx")
```

5. Add another sheet to an Excel workbook:

```
> wb <- loadWorkbook("auto_wb.xlsx")
> sheet2 <- createSheet(auto.wb, "auto2")
> addDataFrame(auto[,1:9], sheet2, row.names=FALSE)
> saveWorkbook(auto.wb, "auto_wb.xlsx")
```

6. Add columns to a worksheet and save the workbook:

```
> wb <- loadWorkbook("auto_wb.xlsx")
> sheets <- getSheets(wb)
> sheet <- sheets[[2]]
> addDataFrame(auto[,10:11], sheet, startColumn=10, row.
names=FALSE)
> saveWorkbook(wb, "newauto.xlsx")
```

7. Read from an Excel workbook:

```
> new.auto <- read.xlsx("newauto.xlsx", sheetIndex=2)
> head(new.auto)
> new.auto <- read.xlsx("newauto.xlsx", sheetName="auto2")
```

8. Read a specific region from an Excel workbook:

```
> sub.auto <- read.xlsx("newauto.xlsx",
sheetName="autobase", rowIndex=1:4, colIndex=1:9)
```

## How it works...

There are multiple options for reading and saving a worksheet. We see a few examples.

Step 1 saves the auto data frame to a new worksheet called "autobase" and creates the Excel file. If we do not include `row.names=FALSE`, the row numbers are displayed as the first column in the spreadsheet.

Step 2 adds two additional columns to the `auto` data frame. The first column `kmpg` is kilometers per gallon and the second new column is the mean `mpg` deviation. We used vector operations to compute the columns.

Step 3 shows the following functions to create workbooks, worksheets, rows, and cells:

- ▶ `createWorkbook`: This creates a workbook object and returns a reference to the object.
- ▶ `createSheet`: This creates a worksheet and gives it the name passed in. If a sheet name is not provided, a default sheet name `sheetx` is used.
- ▶ `createRow`: This creates a row within the sheet. `RowIndex` specifies the row number.
- ▶ `createCell`: This creates a cell in the given row at a specific column index.
- ▶ `setCellValue`: This assigns a value to the specified cell.

- ▶ `addDataFrame`: This includes a data frame to the specified sheet. By default, `row.names` are included and the starting row and column is 1. However, these can be passed as an argument to specify a different row and column index. In our example, we used `startRow=3` since we manually created the first row to hold the heading followed by an empty row. We defaulted the column to 1.

Step 4 shows how styles can be added to a cell. We can add styles while creating the row, column, or adding a data frame. Whatever you can do in Excel by way of styling can be done from within R. Here, we see an example of adding a color and font to our heading row cell.

Step 5 shows the addition of a new sheet. We use `addDataFrame` to add a data frame. Once again we use `row.names=FALSE` so as not to include the row numbers as a column. Since we did not specify the `startRow`, it is taken as 1. We save the workbook with the two new sheets as `auto_wb.xlsx`.

Step 6 uses the `addDataFrame` function to add a data frame to a worksheet. We first read the previously saved workbook file `auto.xlsx` using `loadWorkbook` and save it in a variable `wb`. We then call the `getSheets` function to get all the worksheets in this workbook. The `getSheets` function returns an array and we can get a specific sheet by mentioning its index. Hence, `sheets[[2]]` returns the second sheet in the workbook.

We add the two new columns that we created in step 2 to the sheet. We finally save the workbook.

Step 7 shows how to read directly from an Excel file using `read.xlsx`. We can refer to a specific sheet either using `sheetIndex` or `sheetName`. The `sheetIndex` attribute starts from 1.

Step 8 shows how to load a specific region from an Excel sheet. The `RowIndex` attribute is set to `1:4` and extracts the header row and the first three data rows.

## Reading data from relational databases – MySQL

You can connect to relational databases using several different approaches.

The `RODBC` package provides access to most relational databases through the **ODBC (Open Database Connectivity)** interface. The `RJDBC` package provides access to databases through the JDBC interface and hence needs a Java environment.

There are database packages such as `ROracle`, `RMySQL`, and so on to provide connectivity to the specific relational databases.

Each of the aforementioned options performs differently and has different requirements. You should benchmark and select the package that performs best for your specific needs. In general, `RJDBC` performs poorly and hence you will likely choose `RODBC` or your database-specific R package. In this recipe, we describe the steps to work with the MySQL database.

## Getting ready

First create a data frame to work with as follows:

```
> customer <- c("John", "Peter", "Jane")
> orddt <- as.Date(c('2014-10-1','2014-1-2','2014-7-6'))
> ordamt <- c(280, 100.50, 40.25)
> order <- data.frame(customer,orddt,ordamt)
```

Then install the MySQL server and create in it a database called `Customer`.

To use the `RODBC` package:

1. Download and install MySQL Connector/ODBC for your operating system.
2. Create a DSN called `order_dsn` in the ODBC Configuration Manager by selecting the correct driver for your platform.
3. In R, execute `install.packages ("RODBC")`.

To use the `RJDBC` package:

1. Download and install MySQL Connector/J for your operating system.
2. Install Java Runtime and set the `JAVA_HOME` environment variable accordingly.
3. In R, execute `install.packages ("RJDBC")`.

To use the `RMySQL` package:

1. Download and install MySQL Connector/J for your operating system.
2. Create an environment variable `MYSQL_HOME` pointing to the folder where MySQL is installed.
3. Only on Windows: Copy `libmysql.dll` from the `lib` directory of your MySQL installation to the `bin` directory.
4. In R, execute `install.packages ("RMySQL")`.

## How to do it...

We show how to use each of the preceding packages to connect to the database.

### Using RODBC

To use the RODBC package to connect to the database follow these steps:

1. Load the RODBC library and create a connection object:

```
> library(RODBC)
> con <- odbcConnect("order_dsn", uid="user", pwd="pwd")
```

2. Save the order object into a table in the database:

```
> sqlSave(con, order, "orders", append=FALSE)
```

3. Get all orders from the database table:

```
> custData <- sqlQuery(con, "select * from orders")
```

4. Close the connection object:

```
> close(con)
```

### Using RMySQL

To use the RMySQL package, follow these steps:

1. Load the RMySQL library and create a connection object:

```
> library(RMySQL)
> con <- dbConnect("MySQL", dbname="Customer",
 host="127.0.0.1", port=8889, username="root",
 password="root")
```

2. Save the order object into a table in the database:

```
> dbWriteTable(con, "orders", order)
```

3. Get all orders from the database table:

```
> dbReadTable(con, "Orders")
> dbGetQuery(con, "select * from orders")
```

4. Get all orders from the database table using a loop:

```
> rs <- dbSendQuery(con, "select * from orders")
> while(!dbHasCompleted(rs)) {
 fetch(rs, n=2)
}
```

```
> dbClearResult(rs)
> dbDisconnect(con)
> dbListConnections(dbDriver("MySQL"))
```

## Using RJDBC

To use the RJDBC package follow these steps:

1. Load the RJDBC library and create a connection object. Make sure to point to the correct location of the downloaded .jar file.

```
> library(RJDBC)
> driver <- JDBC("com.mysql.jdbc.Driver",
 classpath=
 "/etc/jdbc/mysql-connector-java-5.1.34-bin.jar", "")
> con <- dbConnect(driver, "jdbc:mysql://host:port/Customer"
 , "username", "password")
```

2. The remaining operations are identical to those in the *Using RMySQL* section.

## How it works...

The preceding code first creates a data frame called `order` with three rows. It then connects to a MySQL database using different methods.

## Using RODBC

In this method we executed the following command:

```
> con <- odbcConnect("cust_dsn", uid="user", pwd="pwd")
```

The `con` variable now has a connection to the database associated to the DSN. All subsequent database calls use this connection object. When all database operations are done, we close the connection.

Although we will typically not create tables or insert data from R, we have shown the code for this just for illustration. The `sqlSave` function saves the data in the R data object to the specified table. We used `append=FALSE` because the table does not already exist and hence we will want R to create the table first and then insert the data. If the table already exists, you can use `append=TRUE`.

The `sqlQuery` function executes the supplied query and returns the result set as a data frame.

## **Using RMySQL**

The RMySQL package uses the MYSQL\_HOME environment variable to get to the needed libraries:

```
> dbWriteTable(con, "orders", order)
```

The dbWriteTable function inserts records into the table. If the table does not exist, it creates the table. By default, row.names of the data frame is added as a column to the table; if it is not needed, remember to set it to FALSE:

```
> dbReadTable(con, "Orders")
```

The dbReadTable function reads the table and creates a data frame:

```
> dbGetQuery(con, "select * from orders")
```

The dbGetQuery function executes the query and returns all the results as a data frame. When the table is large, it is better to use dbSendQuery and fetch results as needed:

```
> rs <- dbSendQuery(con, "select * from orders")
> while(!dbHasCompleted(rs)) {
+ fetch(rs,n=2)
+ }
> dbClearResult(rs)
> dbDisconnect(con)
```

The dbSendQuery function returns rs, a result set object. When you fetch rows with this object, since n=2, two records are returned from the database. While using dbSendQuery, it is a good idea to use a loop until dbHasCompleted is TRUE. Remember to clear the pointer with dbClearResult and close the connection using dbDisconnect:

```
> dbListConnections(dbDriver("MySQL"))
```

The dbListConnections function lists all the open connections.

## **Using RJDBC**

With JDBC, we can connect to any database. Hence, we need to tell R which driver to use. Once the driver is assigned in R, we use this later to create a connection to the database using the appropriate .jar files.

If connecting to a MySQL database, all the statements after getting the connection object are identical to those in the RMySQL scenario.

## There's more...

The database-specific packages provide a lot of functionality, and pretty much most of what can be done within a SQL client can also be done from within an R environment. We show a few examples as follows.

### Fetching all rows

The following command is used to fetch all rows:

```
> fetch(rs, n=-1)
```

Use `n=-1` to fetch all rows.

### When the SQL query is long

When the SQL query is long, it becomes unwieldy to specify the whole query as one long string spanning several lines. Use the `paste()` function to break the query over multiple lines and make it more readable:

```
> dbSendQuery(con, statement=paste(
 "select ordernumber, orderdate, customername",
 "from orders o, customers c",
 "where o.customer = c.customer",
 "and c.state = 'NJ'",
 "ORDER BY ordernumber"))
```



Note the use of the single quotes to specify a string literal.



## Reading data from NoSQL databases – MongoDB

Unlike relational databases for which a somewhat standard approach works across all relational databases, the fluid state of NoSQL databases means that no such standard approach has yet evolved. We illustrate this with MongoDB using the `rmongodb` package.

## Getting ready

Prepare the environment by following these steps:

1. Download and install MongoDB.
2. Start mongod as a background process and then start mongo.
3. Create a new database `customer` and a collection called `orders`:

```
> use customer
> db.orders.save({customername:"John",
 orderdate:ISODate("2014-11-01"),orderamount:1000})
> db.orders.find()
> db.save
```

## How to do it...

To read data from MongoDB, follow these steps:

1. Install the `rmongodb` package and create a connection:

```
> install.packages("rmongodb")
> library(rmongodb)
> mongo <- mongo.create()
> mongo.create(host = "127.0.0.1", db = "customer")
> mongo.is.connected(mongo)
```

2. Get all the collections in the MongoDB database:

```
> coll<- mongo.get.database.collections(mongo,"customer")
```

3. Find all records matching search criteria:

```
> json <- "{\"orderamount\":{\"$lte\":25000},
 \"orderamount\":{\"$gte\":1000}}"
> dat <- mongo.find.all(mongo,coll,json)
```

## How it works...

The `mongo_create` function creates a `mongo` session. If no argument is passed, it connects to the `localhost` at the default port `27017`, where `mongod` is running.

Ensure that R has a valid `mongo` session using `mongo.is.connected(mongo)`.

The `mongo.get.database.collections` function lists all the collections in that database.

The `mongo.find.all` function lists all the rows in that collection. By passing in a valid JSON object, the query results are limited by the search condition specified in the JSON object. If a JSON object is not passed, all rows are returned. R creates a data frame with the returned result.

### There's more...

The fluidity of the NoSQL environment and the newness of MongoDB mean that the `rmongodb` package changes frequently. You should update the `rmongodb` package to get the latest enhancements into your R environment. You should consider validating JSON expressions before using them in code.

### Validating your JSON

Creating a JSON structure in R can get quite complex due to its special characters. Use the `validate()` function to ensure that the JSON structure is error-free:

```
> library(jsonlite)
> json <- "{\"orderamount\":{\"$lte\":25000},
 \"orderamount\":{\"$gte\":1500}}"
> validate(json)
```



# Index

## A

**abline() function 42**

**AdaBoost**

used, for combining classification tree models 98-100

**adjacency matrices**

creating 192-195

**Adjusted R-squared value 115**

**apply function**

used, for processing entire rows or columns 242, 243

using, on three-dimensional array 244

**auto.arima function 186**

**automated ARIMA model**

building 185, 186

## B

**boxplot**

creating 37, 38

working 40

## C

**cache option 227**

**cases, with missing values**

cases with NA, eliminating 12

NA values, excluding from computations 13

no missing values cases, searching 13

removing 11, 12

specific values, converting to NA 13

**categorical variables**

dummies, creating 22, 23

selecting, for dummy creation 23

**charts**

creating, for comparison 56

creating, to visualize causality 60-62

**charts, creating for comparison**

about 55, 56

base plotting system, using 56

boxplots, creating with ggplot2 58, 59

ggplot2, using 57, 58

**classification**

KNN approach, using 88, 89

linear discriminant function analysis, using 93-95

logistic regression, using 95-97

Naïve Bayes approach, using 85-87

neural networks, using 90-92

random forest models, using 78, 79

SVM, using 81-84

**classification tree models**

combining, with AdaBoost 98-100

**classification trees**

building 72-76

creating 77

evaluating 72-76

plotting 72-76

raw probabilities, computing 77

**cluster analysis**

about 139

performing, with hierarchical clustering 146-149

performing, with K-means clustering 140-144

**Comma Separated Values.** See **CSV files**

**Coordinate Reference System (CRS) 281**

**coordinates function 277**

**createDataPartition() function 34**

**CSV files**

column delimiters, handling 3  
column headers/variable names, handling 3  
data, reading directly from website 4  
data, reading from 2  
missing values, handling 4  
strings, reading as characters 4

**cut function 266****D****data, reading**

from CSV files 2  
from fixed width formatted files 8, 9  
from MongoDB 307-309  
from MySQL 302  
from NoSQL database 307-309  
from relational databases 303  
from R files 9, 10  
from R libraries 9, 10

**data frames**

data, normalizing 18  
data, standardizing 18  
several variables, standardizing 19

**dataset**

splitting 31

**data tables**

column, deleting 257  
joining 257  
multiple aggregated columns, adding 256  
symbols, using 258  
used, for combining data 253-255  
used, for dicing data 253-255  
used, for slicing data 253-255

**date objects**

creating 159-163  
examining 159-163  
format specifier 163  
operating 164-166

**decompose function 178****density() function 41****diff function 169****downloaded Google map**

data, overlaying 265-267

**dummies**

creating, for categorical variables 22, 23

**duplicate cases**

duplicates, identifying 16  
removing 15

**E****echo option 227****edge lists**

creating 192-195

**error/classification-confusion matrices**

generating 66, 67  
model's performance, comparing 69  
visualizing 67, 68

**error option 227****ESRI shape files**

importing, into R 267-269

**eval option 227****Excel**

connecting, with xlsx package 299-302

**Extensible Markup Language. See XML data****F****filter function 180****fit\$betweenss command 144****fit\$centers command 144****fit\$cluster command 144****fit\$ifault command 144****fit\$iter command 144****fit\$size command 144****fit\$totss command 144****fit\$tot.withinss command 144****fit\$withinss command 144****fixed width formatted files**

columns, excluding from data 9

data, reading 8, 9

files, with headers 9

**format function 163****fromJSON function 8****F-statistic 115****functions**

applying, on groups from data frame 249

applying, to subsets of vector 248, 249

## G

### **geographic data**

plotting, with sp package 270-273

### **geoms 49**

### **geom\_smooth function 52**

### **GetMap function 263**

### **ggplot2 package**

used, for creating plots 49-54

### **Google map**

downloaded map, saving as image file 264

of specific area, downloading 262, 263

of specific area, plotting 262, 263

satellite image, obtaining 264

### **graphics device**

selecting 45

### **grid**

graphics parameters 45

multiple plots, generating 43, 44

## H

### **hierarchical clustering**

used, for performing cluster analysis 146-149

### **histograms**

about 36, 37

density plot, overlaying 41

working 40

### **Holt-Winters method**

used, for forecasting 182-184

used, for smoothing 182-184

## I

### **interactive web applications**

creating, with shiny 228-230

dynamic UI, adding 232, 233

HTML, adding 231

images, adding 231

single file web application, creating 233

tab sets, adding 231, 232

## J

### **JAR files**

URL 287, 295

## Java

R functions, calling with Rserve 295-297

R scripts, executing from 298, 299

### **Java Native Interface (JNI) 286**

### **Java objects**

available methods, displaying 292

JVM properties, checking 291

using, in R 286-291

### **Java-R Interface. See JRI**

### **Java Virtual Machine (JVM) 289**

### **JRI**

about 286

used, for calling R functions  
from Java 292-294

### **JSON data**

reading 7, 8

## K

### **k-fold cross-validation**

performing 135, 136

### **K-means clustering**

K value, selecting with convenience  
function 145, 146

used, for performing cluster  
analysis 140-144

### **knitr**

about 218

output options, adding 228

render function, using 228

used, for generating data analysis  
reports 218-226

### **KNN**

models for regression, building 104-107

running, with convenience function 108

running, with convenience function for

multiple k values 109, 110

running, with cross-validation 108

running process, automating 89, 90

used, for classification 88, 89

used, for computing raw probabilities 90

## L

### **lapply function**

used, for applying function to collection  
elements 245, 246

**lattice package**  
 used, for creating plots 46, 48

**leave-one-out-cross-validation.** *See LOOCV*

**legend function** 267

**linear discriminant function analysis**  
 formula interface, using 95  
 used, for classification 93, 94

**linear regression**  
 options, using in formula expression 116  
 performing 110-115  
 relevel function, using 115  
 variable selection, performing 117-119

**lines() function** 41

**logistic regression**  
 used, for classification 95-97

**LOOCV**  
 performing, for overfitting limitation 137, 138

## M

**map2SpatialPolygons function** 281

**MapBackground function** 267

**maps**  
 obtaining, from maps package 274, 275

**mean() function** 28

**message option** 227

**missing values**  
 random values sampled, imputing from nonmissing values 14, 15  
 replacing, with mean 13, 14

**MongoDB**  
 JSON, validating 309

**multiple plots**  
 creating, on grid 43, 44

**multivariate plots**  
 creating 62-64

**MySQL**  
 data, reading from 302, 303

## N

**Naïve Bayes approach**  
 used, for classification 85-87

**naiveBayes() function** 86

**network metrics**  
 betweenness 212

centrality 212  
 closeness 213  
 computing 209-213  
 degree 212  
 edges, adding 214  
 edge sequences, obtaining 213  
 isolates, deleting from graph 214, 215  
 neighbors, obtaining 214  
 subgraphs, creating 215  
 vertices or nodes, adding 214

**neural networks**  
 for generation, using 132-134  
 greater control over nnet, exercising 92  
 raw probabilities, generating 92  
 ROC curve, plotting 92  
 using, for classification 90-92

**NoSQL database**  
 data, reading from 307, 308

**numerical data**  
 binning 20, 21  
 specified number of intervals, creating 21

## O

**Open Database Connectivity (ODBC)** 302

## P

**PDF presentations**  
 creating, with R Presentation 234-236

**plot function** 175, 273

**PlotOnStaticMap function** 263, 267

**plots**  
 conditioning, on continuous numeric variables 55  
 creating, with ggplot2 package 49-54  
 creating, with lattice package 46-48  
 graphs, customizing 48, 49

**plyr function**  
 list of data frames, concatenating into big data frame 253  
 new column, adding with transform 252  
 split-apply-combine strategy, using with 250-252  
 summarize, using 253

**principal component analysis (PCA)**  
about 139  
used, for reducing dimensionality 150-156

**R**

**R**  
array, retrieving 298  
ESRI shape files, importing into 267-269  
Java objects, using 286-291  
professional presentation, tasks 217

**random data partitions**  
categorical target variable and three partition 33  
categorical target variable and two partitions 33  
convenience function, using 34, 35  
creating 32  
numerical target variable and three partitions 33  
numerical target variable and two partitions 32  
sampling, from set of values 35  
working 34

**randomForest function 79**

**random forest models**  
cutoffs, specifying for classification 80  
forest generation, controlling 131, 132  
for regression, building 127-131  
raw probabilities, computing 79  
ROC chart, generating 80  
using, for classification 78, 79

**R data file**  
attaching/detaching, to environment 11

**read.csv() function 2**

**readOGR function 269**

**receiver operating characteristic charts.**  
*See ROC charts*

**regression**  
neural networks, using for 132-134  
random forest models, building for 127-131

**regression trees**  
building 120-125  
generating, for categorical predictors

data 126

**relational databases**  
data, reading from 302, 303  
RJDBC, using 305, 306  
RMySQL, using 304-306  
RODBC, using 304, 305  
rows, fetching 307  
SQL query 307

**render function 227**

**Residual standard error 115**

**R files**  
data, reading from 9, 10

**R functions**  
calling, from Java with JRI 292-294  
calling, from Java with Rserve 295-297

**R libraries**  
data, reading from 9, 10

**R Markdown**  
about 218  
output options, adding 228  
render function, using 227  
URL 226  
used, for generating data analysis reports 218-226

**R objects**  
saving, in session 10 ,11

**ROC charts**  
arbitrary class labels, using 72  
generating 69-71

**root mean squared (RMS) error**  
computing, with convenience function 103

**R Presentation**  
appearance, enhancing 237  
display, controlling 237  
hyperlinks, using 236  
used, for creating PDF analysis presentations 234-236

**R scripts**  
executing, from Java 298, 299

**Rserve**  
about 286  
used, for calling R functions from Java 295-297

## S

### **sapply function**

caution 247  
dynamic output 247  
used, for applying function to collection elements 245, 246

### **saveRDS() function 10**

### **scatterplot**

color-specific points 42, 43  
creating 39  
matrices 39  
regression line, overlaying 42

### **sd() function 28**

### **selectInput function 230**

### **shiny**

URL 231  
used, for creating interactive web applications 228-230

### **shinyServer function 230**

### **shinyUI function 230**

### **social network data**

adjacency matrix with weights, extracting 205  
bipartite network graph, creating 206, 207  
bipartite network projections, generating 208  
directed graphs, plotting 203  
downloading, with public APIs 188-191  
edge list, extracting from graph objects 206  
graph object, creating with weights 204  
network as adjacency matrix, extracting 204  
plotting 196-202  
plotting preferences, specifying 203

### **spatial data frames**

creating, by combining regular data frame with spatial objects 277-282  
creating, from regular data frames 275-277

### **SpatialPolygonsDataFrame function 281**

### **split-apply-combine strategy**

using, with plyr function 250-252

### **sp package**

used, for plotting geographic data 270-273

### **spplot function 273**

### **standard data summaries**

creating 26, 27  
mean, finding 28  
single variable summary, computing 28

standard deviation, finding 28

str() function, using 27

### **standard plots**

boxplot 37, 38  
generating 35, 36  
histograms 36, 37  
scatterplot 39  
scatterplot matrices 39, 40

### **str() function 27**

### **strsplit function 281**

### **subset function 29**

### **subset of dataset**

columns, excluding 30  
extracting 28-30  
logical vector, used for selections 30  
multiple value based selections 30

### **subsets of vector**

functions, applying to 248, 249

### **summary() function 27**

### **Support Vector Machine (SVM)**

model type, determining 84  
used, for classification 81-84  
variable scaling, controlling 84  
weights, assigning to classes 84

## T

### **table() function 67**

### **tabPanel() function 231**

### **tapply function 248**

### **time series**

decomposing 177-180

### **time series data**

filtering 180, 181  
preliminary analyses, performing 166-170

### **time series objects**

using 170-176

### **ts function 175**

## V

### **variables**

adding, to existing spatial data frame 282, 283  
multiple variables, rescaling 17  
rescaling 16, 17

**variable selection**

performing, in linear regression 117-119

**vectorized operations**

exploiting 240, 241

**W****Ward's method 149****warning option 227****X****xlsx package**

used, for connecting to Excel 299-302

**XML data**

HTML table data, extracting from web page 6

reading 5, 6

single HTML table, extracting from

web page 7





## Thank you for buying R Data Analysis Cookbook

### 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](http://www.packtpub.com).

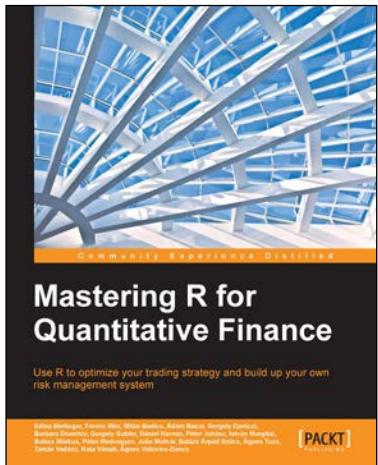
### About Packt Open Source

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](mailto: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.

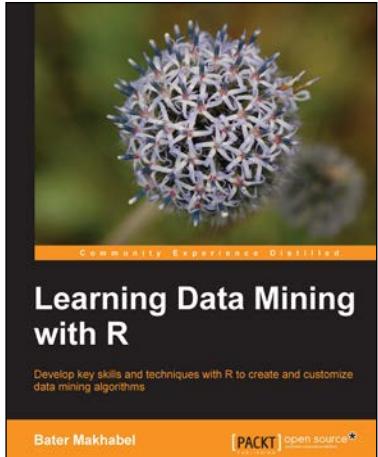


## Mastering R for Quantitative Finance

ISBN: 978-1-78355-207-8      Paperback: 362 pages

Use R to optimize your trading strategy and build up your own risk management system

1. Learn to manipulate, visualize, and analyze a wide range of financial data with the help of built-in functions and programming in R.
2. Understand the concepts of financial engineering and create trading strategies for complex financial instruments.
3. Explore R for asset and liability management and capital adequacy modeling.



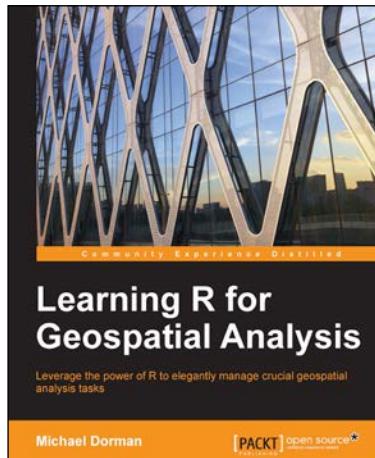
## Learning Data Mining with R

ISBN: 978-1-78398-210-3      Paperback: 314 pages

Develop key skills and techniques with R to create and customize data mining algorithms

1. Develop a sound strategy for solving predictive modeling problems using the most popular data mining algorithms.
2. Gain understanding of the major methods of predictive modeling.
3. Packed with practical advice and tips to help you get to grips with data mining.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

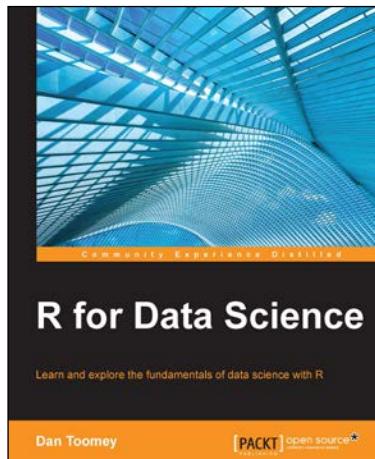


## Learning R for Geospatial Analysis

ISBN: 978-1-78398-436-7      Paperback: 364 pages

Leverage the power of R to elegantly manage crucial geospatial analysis tasks

1. Write powerful R scripts to manipulate your spatial data.
2. Gain insight from spatial patterns utilizing R's advanced computation and visualization capabilities.
3. Work within a single spatial analysis environment from start to finish.



## R for Data Science

ISBN: 978-1-78439-086-0      Paperback: 364 pages

Learn and explore the fundamentals of data science with R

1. Familiarize yourself with R programming packages and learn how to utilize them effectively.
2. Learn how to detect different types of data mining sequences.
3. A step-by-step guide to understanding R scripts and the ramifications of your changes.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles