

R: Recipes for Analysis, Visualization and Machine Learning

Get savvy with R language and actualize projects
aimed at analysis, visualization and machine learning

A course in three modules

Packt

BIRMINGHAM - MUMBAI

R: Recipes for Analysis, Visualization and Machine Learning

Copyright © 2016 Packt Publishing

Published on: November 2016

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

ISBN 978-1-78728-959-8

www.packtpub.com

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.

Our ability to generate data has improved tremendously with the advent of technology. The data generated has become more complex with the passage of time. The complexity in data forces us to develop new tools and methods to analyze it, interpret it, and communicate with the data. Data visualization empowers us with the necessary skills required to convey the meaning of underlying data. Data visualization is a remarkable intersection of data, science, and art, and this makes it hard to define visualization in a formal way; a simple Google search will prove me right. The Merriam-Webster dictionary defines visualization as "formation of mental visual images".

Big data has become a popular buzzword across many industries. An increasing number of people have been exposed to the term and are looking at how to leverage big data in their own businesses, to improve sales and profitability. However, collecting, aggregating, and visualizing data is just one part of the equation. Being able to extract useful information from data is another task, and much more challenging.

Traditionally, most researchers perform statistical analysis using historical samples of data. The main downside of this process is that conclusions drawn from statistical analysis are limited. In fact, researchers usually struggle to uncover hidden patterns and unknown correlations from target data. Aside from applying statistical analysis, machine learning has emerged as an alternative. This process yields a more accurate predictive model with the data inserted into a learning algorithm. Through machine learning, the analysis of business operations and processes is not limited to human-scale thinking. Machine-scale analysis enables businesses to discover hidden values in big data.

What this learning path covers

Module 1, R Data Analysis Cookbook, this module, 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.

Module 2, R Data Visualization Cookbook, in this module you will learn how to generate basic visualizations, understand the limitations and advantages of using certain visualizations, develop interactive visualizations and applications, understand various data exploratory functions in R, and finally learn ways of presenting the data to our audience. This module is aimed at beginners and intermediate users of R who would like to go a step further in using their complex data to convey a very convincing story to their audience.

Module 3, Machine Learning with R Cookbook, this module covers how to perform statistical analysis with machine learning analysis and assessing created models, which are covered in detail later on in the book. The module includes content on learning how to integrate R and Hadoop to create a big data analysis platform. The detailed illustrations provide all the information required to start applying machine learning to individual projects.

What you need for this learning path

Module 1:

We have tested all the code in this module 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 module.

Module 2:

You need to download R to generate the visualizations. You can download and install R using the CRAN website available at <http://cran.r-project.org/>. All the recipes were written using RStudio. RStudio is an integrated development environment (IDE) for R and can be downloaded from <http://www.rstudio.com/products/rstudio/>. Many of the visualizations are created using R packages and they are discussed in their respective recipes.

In few of the recipes, I have introduced users to some other open source platforms such as ScapeToad, ArcGIS, and Mapbox. Their installation procedures are outlined in their respective recipes.

Module 3:

To follow the course's examples, you will need a computer with access to the Internet and the ability to install the R environment. You can download R from <http://www.cran.r-project.org/>. Detailed installation instructions are available in the first chapter.

The examples provided in this book were coded and tested with R Version 3.1.2 on a computer with Microsoft Windows installed on it. These examples should also work with any recent version of R installed on either MAC OSX or a Unix-like OS.

Who this learning path is for

This Learning Path is ideal for those who are already exposed to R, but have not yet used it extensively. This Learning Path will set you up with an extensive insight into professional techniques for analysis, visualization and machine learning with R. Regardless of your level of experience, this course also covers the basics of using R and it is written keeping in mind new and intermediate R users interested in learning.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – 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 course'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 course, see our author guide at www.packtpub.com/authors.

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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

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

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/R-Recipes-for-Analysis-Visualization-and-Machine-Learning>. We also have other code bundles from our rich catalog of books, videos and courses available at <https://github.com/PacktPublishing/>. Check them out!

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 course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, 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.

Contents

Module 1

1

Chapter 1: A Simple Guide to R	3
Installing packages and getting help in R	4
Data types in R	6
Special values in R	7
Matrices in R	9
Editing a matrix in R	10
Data frames in R	11
Editing a data frame in R	11
Importing data in R	12
Exporting data in R	13
Writing a function in R	14
Writing if else statements in R	15
Basic loops in R	16
Nested loops in R	16
The apply, lapply, sapply, and tapply functions	17
Using par to beautify a plot in R	18
Saving plots	19
Chapter 2: Practical Machine Learning with R	21
Introduction	21
Downloading and installing R	23
Downloading and installing RStudio	31
Installing and loading packages	35
Reading and writing data	37
Using R to manipulate data	40
Applying basic statistics	44
Visualizing data	48
Getting a dataset for machine learning	52

Chapter 3: Acquire and Prepare the Ingredients – Your Data	57
Introduction	58
Reading data from CSV files	58
Reading XML data	61
Reading JSON data	63
Reading data from fixed-width formatted files	64
Reading data from R files and R libraries	65
Removing cases with missing values	67
Replacing missing values with the mean	69
Removing duplicate cases	71
Rescaling a variable to [0,1]	72
Normalizing or standardizing data in a data frame	74
Binning numerical data	76
Creating dummies for categorical variables	78
Chapter 4: What's in There? – Exploratory Data Analysis	81
Introduction	82
Creating standard data summaries	82
Extracting a subset of a dataset	84
Splitting a dataset	87
Creating random data partitions	88
Generating standard plots such as histograms, boxplots, and scatterplots	91
Generating multiple plots on a grid	99
Selecting a graphics device	101
Creating plots with the lattice package	102
Creating plots with the ggplot2 package	105
Creating charts that facilitate comparisons	111
Creating charts that help visualize a possible causality	116
Creating multivariate plots	118
Chapter 5: Where Does It Belong? – Classification	121
Introduction	121
Generating error/classification-confusion matrices	122
Generating ROC charts	125
Building, plotting, and evaluating – classification trees	128
Using random forest models for classification	134
Classifying using Support Vector Machine	137
Classifying using the Naïve Bayes approach	141
Classifying using the KNN approach	144
Using neural networks for classification	146
Classifying using linear discriminant function analysis	149

Classifying using logistic regression	151
Using AdaBoost to combine classification tree models	154
Chapter 6: Give Me a Number – Regression	157
Introduction	157
Computing the root mean squared error	158
Building KNN models for regression	160
Performing linear regression	166
Performing variable selection in linear regression	173
Building regression trees	176
Building random forest models for regression	183
Using neural networks for regression	188
Performing k-fold cross-validation	191
Performing leave-one-out-cross-validation to limit overfitting	193
Chapter 7: Can You Simplify That? – Data Reduction Techniques	195
Introduction	195
Performing cluster analysis using K-means clustering	196
Performing cluster analysis using hierarchical clustering	202
Reducing dimensionality with principal component analysis	206
Chapter 8: Lessons from History – Time Series Analysis	215
Introduction	215
Creating and examining date objects	215
Operating on date objects	220
Performing preliminary analyses on time series data	222
Using time series objects	226
Decomposing time series	233
Filtering time series data	236
Smoothing and forecasting using the Holt-Winters method	238
Building an automated ARIMA model	241
Chapter 9: It's All About Your Connections – Social Network Analysis	243
Introduction	243
Downloading social network data using public APIs	244
Creating adjacency matrices and edge lists	248
Plotting social network data	252
Computing important network metrics	265
Chapter: Put Your Best Foot Forward – Document and Present Your Analysis	273
Introduction	273
Generating reports of your data analysis with R Markdown and knitr	274

Creating interactive web applications with shiny	284
Creating PDF presentations of your analysis with R Presentation	290
Chapter 11: Work Smarter, Not Harder – Efficient and Elegant R Code	295
Introduction	295
Exploiting vectorized operations	296
Processing entire rows or columns using the apply function	298
Applying a function to all elements of a collection with lapply and sapply	301
Applying functions to subsets of a vector	304
Using the split-apply-combine strategy with plyr	306
Slicing, dicing, and combining data with data tables	309
Chapter 12: Where in the World? – Geospatial Analysis	317
Introduction	318
Downloading and plotting a Google map of an area	318
Overlaying data on the downloaded Google map	321
Importing ESRI shape files into R	323
Using the sp package to plot geographic data	326
Getting maps from the maps package	330
Creating spatial data frames from regular data frames containing spatial and other data	331
Creating spatial data frames by combining regular data frames with spatial objects	333
Adding variables to an existing spatial data frame	338
Chapter 13: Playing Nice – Connecting to Other Systems	341
Introduction	341
Using Java objects in R	342
Using JRI to call R functions from Java	348
Using Rserve to call R functions from Java	351
Executing R scripts from Java	354
Using the xlsx package to connect to Excel	355
Reading data from relational databases – MySQL	358
Reading data from NoSQL databases – MongoDB	363
Module 2	367
Chapter 1: Basic and Interactive Plots	369
Introduction	370
Introducing a scatter plot	370
Scatter plots with texts, labels, and lines	373

Connecting points in a scatter plot	376
Generating an interactive scatter plot	379
A simple bar plot	382
An interactive bar plot	384
A simple line plot	388
Line plot to tell an effective story	390
Generating an interactive Gantt/timeline chart in R	393
Merging histograms	394
Making an interactive bubble plot	397
Constructing a waterfall plot in R	399
Chapter 2: Heat Maps and Dendrograms	401
Introduction	401
Constructing a simple dendrogram	402
Creating dendrograms with colors and labels	406
Creating a heat map	408
Generating a heat map with customized colors	412
Generating an integrated dendrogram and a heat map	414
Creating a three-dimensional heat map and a stereo map	417
Constructing a tree map in R	419
Chapter 3: Maps	423
Introduction	423
Introducing regional maps	424
Introducing choropleth maps	426
A guide to contour maps	429
Constructing maps with bubbles	432
Integrating text with maps	436
Introducing shapefiles	438
Creating cartograms	442
Chapter 4: The Pie Chart and Its Alternatives	445
Introduction	445
Generating a simple pie chart	446
Constructing pie charts with labels	450
Creating donut plots and interactive plots	453
Generating a slope chart	456
Constructing a fan plot	459
Chapter 5: Adding the Third Dimension	461
Introduction	461
Constructing a 3D scatter plot	462
Generating a 3D scatter plot with text	465
A simple 3D pie chart	468

A simple 3D histogram	470
Generating a 3D contour plot	472
Integrating a 3D contour and a surface plot	474
Animating a 3D surface plot	477
Chapter 6: Data in Higher Dimensions	481
Introduction	481
Constructing a sunflower plot	482
Creating a hexbin plot	484
Generating interactive calendar maps	486
Creating Chernoff faces in R	489
Constructing a coxcomb plot in R	491
Constructing network plots	493
Constructing a radial plot	495
Generating a very basic pyramid plot	498
Chapter 7: Visualizing Continuous Data	501
Introduction	501
Generating a candlestick plot	502
Generating interactive candlestick plots	505
Generating a decomposed time series	507
Plotting a regression line	510
Constructing a box and whiskers plot	512
Generating a violin plot	514
Generating a quantile-quantile plot (QQ plot)	515
Generating a density plot	517
Generating a simple correlation plot	520
Chapter 8: Visualizing Text and XKCD-style Plots	525
Introduction	525
Generating a word cloud	526
Constructing a word cloud from a document	529
Generating a comparison cloud	532
Constructing a correlation plot and a phrase tree	535
Generating plots with custom fonts	538
Generating an XKCD-style plot	540
Chapter 9: Creating Applications in R	543
Introduction	543
Creating animated plots in R	544
Creating a presentation in R	546
A basic introduction to API and XML	549
Constructing a bar plot using XML in R	553
Creating a very simple shiny app in R	556

Chapter 1: Data Exploration with RMS Titanic	561
Introduction	561
Reading a Titanic dataset from a CSV file	563
Converting types on character variables	566
Detecting missing values	568
Imputing missing values	571
Exploring and visualizing data	574
Predicting passenger survival with a decision tree	582
Validating the power of prediction with a confusion matrix	587
Assessing performance with the ROC curve	589
Chapter 2: R and Statistics	591
Introduction	591
Understanding data sampling in R	592
Operating a probability distribution in R	593
Working with univariate descriptive statistics in R	598
Performing correlations and multivariate analysis	602
Operating linear regression and multivariate analysis	604
Conducting an exact binomial test	607
Performing student's t-test	609
Performing the Kolmogorov-Smirnov test	613
Understanding the Wilcoxon Rank Sum and Signed Rank test	616
Working with Pearson's Chi-squared test	617
Conducting a one-way ANOVA	621
Performing a two-way ANOVA	624
Chapter 3: Understanding Regression Analysis	629
Introduction	629
Fitting a linear regression model with lm	630
Summarizing linear model fits	632
Using linear regression to predict unknown values	635
Generating a diagnostic plot of a fitted model	636
Fitting a polynomial regression model with lm	639
Fitting a robust linear regression model with rlm	641
Studying a case of linear regression on SLID data	643
Applying the Gaussian model for generalized linear regression	650
Applying the Poisson model for generalized linear regression	653
Applying the Binomial model for generalized linear regression	654
Fitting a generalized additive model to data	656
Visualizing a generalized additive model	658
Diagnosing a generalized additive model	661

Chapter 4: Classification (I) – Tree, Lazy, and Probabilistic	665
Introduction	665
Preparing the training and testing datasets	666
Building a classification model with recursive partitioning trees	668
Visualizing a recursive partitioning tree	671
Measuring the prediction performance of a recursive partitioning tree	673
Pruning a recursive partitioning tree	675
Building a classification model with a conditional inference tree	678
Visualizing a conditional inference tree	679
Measuring the prediction performance of a conditional inference tree	682
Classifying data with the k-nearest neighbor classifier	684
Classifying data with logistic regression	687
Classifying data with the Naïve Bayes classifier	694
Chapter 5: Classification (II) – Neural Network and SVM	699
Introduction	699
Classifying data with a support vector machine	700
Choosing the cost of a support vector machine	703
Visualizing an SVM fit	707
Predicting labels based on a model trained by a support vector machine	709
Tuning a support vector machine	713
Training a neural network with neuralnet	717
Visualizing a neural network trained by neuralnet	721
Predicting labels based on a model trained by neuralnet	723
Training a neural network with nnet	726
Predicting labels based on a model trained by nnet	728
Chapter 6: Model Evaluation	731
Introduction	731
Estimating model performance with k-fold cross-validation	732
Performing cross-validation with the e1071 package	734
Performing cross-validation with the caret package	735
Ranking the variable importance with the caret package	737
Ranking the variable importance with the rminer package	739
Finding highly correlated features with the caret package	741
Selecting features using the caret package	742
Measuring the performance of the regression model	748
Measuring prediction performance with a confusion matrix	751
Measuring prediction performance using ROCR	753
Comparing an ROC curve using the caret package	755

Measuring performance differences between models with the caret package	758
Chapter 7: Ensemble Learning	763
Introduction	763
Classifying data with the bagging method	764
Performing cross-validation with the bagging method	768
Classifying data with the boosting method	769
Performing cross-validation with the boosting method	773
Classifying data with gradient boosting	774
Calculating the margins of a classifier	780
Calculating the error evolution of the ensemble method	784
Classifying data with random forest	786
Estimating the prediction errors of different classifiers	792
Chapter 8: Clustering	795
Introduction	795
Clustering data with hierarchical clustering	796
Cutting trees into clusters	802
Clustering data with the k-means method	806
Drawing a bivariate cluster plot	809
Comparing clustering methods	811
Extracting silhouette information from clustering	814
Obtaining the optimum number of clusters for k-means	815
Clustering data with the density-based method	818
Clustering data with the model-based method	821
Visualizing a dissimilarity matrix	826
Validating clusters externally	829
Chapter 9: Association Analysis and Sequence Mining	833
Introduction	833
Transforming data into transactions	834
Displaying transactions and associations	836
Mining associations with the Apriori rule	840
Pruning redundant rules	845
Visualizing association rules	847
Mining frequent itemsets with Eclat	851
Creating transactions with temporal information	854
Mining frequent sequential patterns with cSPADE	857
Chapter 10: Dimension Reduction	861
Introduction	861
Performing feature selection with FSelector	863
Performing dimension reduction with PCA	866

Determining the number of principal components using the scree test	871
Determining the number of principal components using the Kaiser method	873
Visualizing multivariate data using biplot	875
Performing dimension reduction with MDS	879
Reducing dimensions with SVD	883
Compressing images with SVD	887
Performing nonlinear dimension reduction with ISOMAP	890
Performing nonlinear dimension reduction with Local Linear Embedding	895
Chapter 11: Big Data Analysis (R and Hadoop)	899
Introduction	899
Preparing the RHadoop environment	901
Installing rmr2	904
Installing rhdfs	905
Operating HDFS with rhdfs	907
Implementing a word count problem with RHadoop	909
Comparing the performance between an R MapReduce program and a standard R program	911
Testing and debugging the rmr2 program	913
Installing plyrnr	915
Manipulating data with plyrnr	916
Conducting machine learning with RHadoop	919
Configuring RHadoop clusters on Amazon EMR	923
Appendix A: Resources for R and Machine Learning	931
Appendix B: Dataset – Survival of Passengers on the Titanic	933
Bibliography	935

Module 1

R Data Analysis Cookbook

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

1

A Simple Guide to R

In this chapter, we will cover the following recipes:

- ▶ Installing packages and getting help in R
- ▶ Data types in R
- ▶ Special values in R
- ▶ Matrices in R
- ▶ Editing a matrix in R
- ▶ Data frames in R
- ▶ Editing a data frame in R
- ▶ Importing data in R
- ▶ Exporting data in R
- ▶ Writing a function in R
- ▶ Writing if else statements in R
- ▶ Basic loops in R
- ▶ Nested loops in R
- ▶ The apply, lapply, sapply, and tapply functions
- ▶ Using par to beautify a plot in R
- ▶ Saving plots

Installing packages and getting help in R

If you are a new user and have never launched R, you must definitely start the learning process by understanding the use of `install.packages()`, `library()`, and getting help in R. R comes loaded with some basic packages, but the R community is rapidly growing and active R users are constantly developing new packages for R.

As you read through this cookbook, you will observe that we have used a lot of packages to create different visualizations. So the question now is, how do we know what packages are available in R? In order to keep myself up-to-date with all the changes that are happening in the R community, I diligently follow these blogs:

- ▶ Rblogger
- ▶ Rstudio blog

There are many blogs, websites, and posts that I will refer to as we go through the book. We can view a list of all the packages available in R by going to <http://cran.r-project.org/>, and also <http://www.inside-r.org/packages> provides a list as well as a short description of all the packages.

Getting ready

We can start by powering up our R studio, which is an **Integrated Development Environment (IDE)** for R. If you have not downloaded Rstudio, then I would highly recommend going to <http://www.rstudio.com/> and downloading it.

How to do it...

To install a package in R, we will use the `install.packages()` function. Once we install a package, we will have to load the package in our active R session; if not, we will get an error. The `library()` function allows us to load the package in R.

How it works...

The `install.packages()` function comes with some additional arguments but, for the purpose of this book, we will only use the first argument, that is, the name of the package. We can also load multiple packages by using `install.packages(c("plotrix", "RColorBrewer"))`. The name of the package is the only argument we will use in the `library()` function. Note that you can only load one package at a time with the `library()` function unlike the `install.packages()` function.

There's more...

It is hard to remember all the functions and their arguments in R, unless we use them all the time, and we are bound to get errors and warning messages. The best way to learn R is to use the active R community and the help manual available in R.

To understand any function in R or to learn about the various arguments, we can type `?<name of the function>`. For example, I can learn about all the arguments related to the `plot()` function by simply typing `?plot` or `?plot()` in the R console window. You will now view the help page on the right side of the screen. We can also learn more about the behavior of the function using some of the examples at the bottom of the help page.

If we are still unable to understand the function or its use and implementation, we could go to Google and type the question or use the Stack Overflow website. I am always able to resolve my errors by searching on the Internet. Remember, every problem has a solution, and the possibilities with R are endless.

See also

- ▶ [Flowing Data](http://flowingdata.com/) (`http://flowingdata.com/`): This is a good resource to learn visualization tools and R. The tutorials are based on an annual subscription.
- ▶ [Stack Overflow](http://stackoverflow.com/) (`http://stackoverflow.com/`): This is a great place to get help regarding R functions.
- ▶ [Inside-R](http://www.inside-r.org/) (`http://www.inside-r.org/`): This lists all the packages along with a small description.
- ▶ [Rblogger](http://www.r-bloggers.com/) (`http://www.r-bloggers.com/`): This is a great webpage to learn about new R packages, books, tutorials, data scientists, and other data-related jobs.
- ▶ [R forge](https://r-forge.r-project.org/) (`https://r-forge.r-project.org/`).
- ▶ [R journal](http://journal.r-project.org/archive/2014-1/) (`http://journal.r-project.org/archive/2014-1/`).

Data types in R

Everything in R is in the form of objects. Objects can be manipulated in R. Some of the common objects in R are numeric vectors, character vectors, complex vectors, logical vectors, and integer vectors.

```
> x = c(1:5) # Numeric Vector
> y ="I am Home" # Character Vector
> c = c(1+3i) #complex vector
>
> x = c(1:5) # Numeric Vector
> x
[1] 1 2 3 4 5
>
> y ="I am Home" # Character Vector
> y
[1] "I am Home"
>
> c = c(1+3i) #complex vector
>
> c
[1] 1+3i
> |
```

How to do it...

In order to generate a numeric vector in R, we can use the `c()` notation to specify it as follows:

```
x = c(1:5) # Numeric Vector
```

To generate a character vector, we can specify the same within quotes (" ") as follows:

```
y ="I am Home" # Character Vector
```

To generate a complex vector, we can use the `i` notation as follows:

```
c = c(1+3i) #complex vector
```

A **list** is a combination of a character and a numeric vector and can be specified using the `list()` notation:

```
z = list(c(1:5), "I am Home") # List
```

Special values in R

R comes with some special values. Some of the special values in R are NA, Inf, -Inf, and NaN.

```
Console ~/ 
> z = c( 1,2,3, NA,5,NA) # NA in R is missing Data
> z
[1] 1 2 3 NA 5 NA
> complete.cases(z) # function to detect NA
[1] TRUE TRUE TRUE FALSE TRUE FALSE
>
> is.na(z) # function to detect NA
[1] FALSE FALSE FALSE TRUE FALSE TRUE
> 0/0
[1] NaN
> m = c(2/3,3/3,0/0)
> m
[1] 0.6666667 1.0000000      NaN
>
> is.finite(m)
[1] TRUE TRUE FALSE
> is.infinite(m)
[1] FALSE FALSE FALSE
> is.nan(m)
[1] FALSE FALSE TRUE
> k = 1/0
> k
[1] Inf
> |
```

How to do it...

The missing values are represented in R by NA. When we download data, it may have missing data and this is represented in R by NA:

```
z = c( 1,2,3, NA,5,NA) # NA in R is missing Data
```

To detect missing values, we can use the ~~install.packages()~~ function or `is.na()`, as shown:

```
complete.cases(z) # function to detect NA
is.na(z) # function to detect NA
```

To remove the NA values from our data, we can type the following in our active R session console window:

```
clean <- complete.cases(z)
z[clean] # used to remove NA from data
```

Please note the use of square brackets ([]) instead of parentheses.

In R, not a number is abbreviated as NaN. The following lines will generate NaN values:

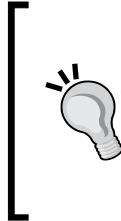
```
##NaN
0/0
m <- c(2/3,3/3,0/0)
m
```

The `is.finite`, `is.infinite`, or `is.nan` functions will generate logical values (TRUE or FALSE).

```
is.finite(m)
is.infinite(m)
is.nan(m)
```

The following line will generate `inf` as a special value in R:

```
## infinite
k = 1/0
```



Downloading the example code

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

How it works...

`complete.cases(z)` is a logical vector indicating complete cases that have no missing value (NA). On the other hand, `is.na(z)` indicates which elements are missing. In both cases, the argument is our data, a vector, or a matrix.

Console ~/ ↵

```
> z = c( 1,2,3, NA,5,NA)
> complete.cases(z)
[1] TRUE TRUE TRUE FALSE TRUE FALSE
> is.na(z)
[1] FALSE FALSE FALSE TRUE FALSE TRUE
> dk = c(1,45,67,20)
> anyNA(dk)
[1] FALSE
> dk[3] = NA
> dk
[1] 1 45 NA 20
> anyNA(dk)
[1] TRUE
> |
```

R also allows its users to check if any element in a matrix or a vector is NA by using the `anyNA()` function. We can coerce or assign NA to any element of a vector using the square brackets (`[]`). The `[3]` input instructs R to assign NA to the third element of the `dk` vector.

Matrices in R

In this recipe, we will dive into R's capability with regard to matrices.

```
Console ~/ ~
> mat = matrix(c(1,2,3,4,5,6,7,8,9,10), nrow = 2, ncol = 5)
> mat
[1,] 1 3 5 7 9
[2,] 2 4 6 8 10
> t(mat)
[1,] 1 2
[2,] 3 4
[3,] 5 6
[4,] 7 8
[5,] 9 10
t() function in R
generates a transposed
matrix. Rows are
transposed into columns.

> d = diag(3)
> d
[1,] 1 0 0
[2,] 0 1 0
[3,] 0 0 1
d() generates an
identity matrix in R.

> zro = matrix(rep(0,6), ncol = 2, nrow = 3 )
> zro
[1,] 0 0
[2,] 0 0
[3,] 0 0
> mat = matrix(c(1:10), nrow = 2, ncol = 5)
> mat
[1,] 1 3 5 7 9
[2,] 2 4 6 8 10
> mat[2,3]
[1] 6
> mat = mat[, -2]
> mat
[1,] 1 5 7 9
[2,] 2 6 8 10
```

How to do it...

A vector in R is defined using the `c()` notation as follows:

```
vec = c(1:10)
```

A vector is a one-dimensional array. A matrix is a multidimensional array. We can define a matrix in R using the `matrix()` function. Alternatively, we can also coerce a set of values to be a matrix using the `as.matrix()` function:

```
mat = matrix(c(1,2,3,4,5,6,7,8,9,10), nrow = 2, ncol = 5)
mat
```

To generate a transpose of a matrix, we can use the `t()` function:

```
t(mat) # transpose a matrix
```

In R, we can also generate an identity matrix using the `diag()` function:

```
d = diag(3) # generate an identity matrix
```

We can nest the `rep` () function within `matrix()` to generate a matrix with all zeroes as follows:

```
zro = matrix(rep(0,6),ncol = 2,nrow = 3 )# generate a  
matrix of Zeros  
zro
```

How it works...

We can define our data in the `matrix` () function by specifying our data as its first argument. The `nrow` and `ncol` arguments are used to specify the number of rows and column in a matrix. The `matrix` function in R comes with other useful arguments and can be studied by typing `?matrix` in the R command window.

The `rep` () function nested in the `matrix()` function is used to repeat a particular value or character string a certain number of times.

The `diag` () function can be used to generate an identity matrix as well as extract the diagonal elements of a matrix. More uses of the `diag` () function can be explored by typing `?diag` in the R console window.

The code file provides a lot more functions that can be used along with matrices—for example, functions related to finding a determinant or inverse of a matrix and matrix multiplication.

Editing a matrix in R

R allows us to edit (add, delete, or replace) elements of a matrix using the square bracket notation, as depicted in the following lines of code:

```
mat = matrix(c(1:10),nrow = 2, ncol = 5)  
mat  
mat [2,3]
```

How to do it...

In order to extract any element of a matrix, we can specify the position of that element in R using square brackets. For example, `mat [2, 3]` will extract the element under the second row and the third column. The first numeric value corresponds to the row and the second numeric value corresponds to a column [row, column].

Similarly, to replace an element, we can type the following lines in R:

```
mat [2,3] = 16
```

To select all the elements of the second row, we can use `mat [2,]`. If we do not specify any numeric value for a column, R will automatically assume all columns.

Data frames in R

One of the useful and widely used functions in R is the `data.frame()` function. Data frame, according to the R manual, is a matrix structure whose columns can be of differing types, such as numeric, logical, factor, or character.

How to do it...

A data frame in R is a collection of variables. A simple way to construct a data frame is using the `data.frame()` function in R:

```
data = data.frame(x = c(1:4), y = c("tom", "jerry", "luke", "brian"))
data
```

Many times, we will encounter plotting functions that require data to be in a data frame. In order to coerce our data into a data frame, we can use the `data.frame()` function. In the following example, we create a matrix and convert it into a data frame:

```
mat = matrix(c(1:10), nrow = 2, ncol = 5)
data.frame(mat)
```

The `data.frame()` function comes with various arguments and can be explored by typing `?data.frame` in the R console window. The code file under the title `Data Frames - 2` provides additional functions that can help in understanding the underlying structure of our data. We can always get additional help by using the R documentation.

Editing a data frame in R

Once we have generated a data and converted it into a data frame, we can edit any row or column of a data frame.

How to do it...

We can add or extract any column of a data frame using the dollar (\$) symbol, as depicted in the following code:

```
data = data.frame(x = c(1:4), y = c("tom", "jerry", "luke", "brian"))
data$age = c(2, 2, 3, 5)
data
```

In the preceding example, we have added a new column called age using the \$ operator. Alternatively, we can also add columns and rows using the rbind() and cbind() functions in R as follows:

```
age = c(2,2,3,5)
data = cbind(data, age)
```

The cbind and rbind functions can also be used to add columns or rows to an existing matrix.

To remove a column or a row from a matrix or data frame, we can simply use the negative sign before the column or row to be deleted, as follows:

```
data = data[,-2]
```

The data[,-2] line will delete the second column from our data.

To re-order the columns of a data frame, we can type the following lines in the R command window:

```
data = data.frame(x = c(1:4), y = c("tom","jerry","luke","brian"))
data = data[c(2,1)]# will reorder the columns
data
```

To view the column names of a data frame, we can use the names() function:

```
names(data)
```

To rename our column names, we can use the colnames() function:

```
colnames(data) = c("Number", "Names")
```

Importing data in R

Data comes in various formats. Most of the data available online can be downloaded in the form of text documents (.txt extension) or as comma-separated values (.csv). We also encounter data in the tab-delimited format, XLS, HTML, JSON, XML, and so on. If you are interested in working with data, either in JSON or XML, refer to the recipe *Constructing a bar plot using XML in R* in Chapter, *Creating Applications in R*.

How to do it...

In order to import a CSV file in R, we can use the read.csv() function:

```
test = read.csv("raw.csv", sep = ",", header = TRUE)
```

Alternatively, `read.table()` function allows us to import data with different separators and formats. Following are some of the methods used to import data in R:

The screenshot shows the RStudio interface. At the top, there's a toolbar with icons for back, forward, and file operations. Below the toolbar is a data frame titled "row.names" with four rows and four columns: age, gender, and score. The data is as follows:

	row.names	age	gender	score
1	tom	18	m	60
2	mary	17	f	60
3	kim	19	f	78
4	jack	18	m	99

Below the data frame is the R console window. It shows the following R code and its output:

```
> setwd("C:/Users/agohil/Book/chapter10/")
> data = read.table("raw.csv", header= TRUE, sep = ",") 
> row.names(data)
[1] "1" "2" "3" "4"
> data = read.table("raw.csv", header= TRUE, sep = ",", row.names = c("Names"))
> row.names(data)
[1] "tom" "mary" "kim" "jack"
> View(data)
>
```

How it works...

The first argument in the `read.csv()` function is the filename, followed by the separator used in the file. The `header = TRUE` argument is used to instruct R that the file contains headers. Please note that R will search for this file in its current directory. We have to specify the directory containing the file using the `setwd()` function. Alternatively, we can navigate and set our working directory by navigating to **Sessions | Set working directory | Choose directory**.

The first argument in the `read.table()` function is the filename that contains the data, the second argument states that the data contains the header, and the third argument is related to the separator. If our data consists of a semi colon (;), a tab delimited, or the @ symbol as a separator, we can specify this under the `sep = ""` argument. Note that, to specify a separator as a tab delimited, users would have to substitute `sep = ", "` with `sep = "\t"` in the `read.table()` function.

One of the other useful arguments is the `row.names` argument. If we omit `row.names`, R will use the column serial numbers as `row.names`. We can assign `row.names` for our data by specifying it as `row.names = c("Name")`.

Exporting data in R

Once we have processed our data, we need to save it to an external device or send it to our colleagues. It is possible to export data in R in many different formats.

How to do it...

To export data from R, we can use the `write.table()` function. Please note that R will export the data to our current directory or the folder we have assigned using the `setwd()` function:

```
write.table(data, "mydata.csv", sep=",")
```

How it works...

The first argument in the `write.table()` function is the data in R that we would like to export. The second argument is the name of the file. We can export data in the `.xls` or `.txt` format, simply by replacing the `mydata.csv` file extension with `mydata.txt` or `mydata.xls` in the `write.table()` function.

Writing a function in R

Most of the tasks in R are performed using functions. A function in R has the same utility as functions in Arithmetic.

Getting ready

In order to write a simple function in R, we must first open a new R script by navigating to **File | New file**.

How to do it...

We write a very simple function that accepts two values and adds them together. Copy and paste the code in the new blank R script:

```
add = function (x,y) {  
  x+y  
}
```

How it works...

A function in R should be defined by `function()`. Once we define our function, we need to save it as a `.r` file. Note that the name of the file should be the same as the function; hence we save our function with name `add.r`.

In order to use the `add()` function in the R command window, we need to source the file by using the `source()` function as follows:

```
source('<your path>/add.R')
```

Now, we can type `add(2, 15)` in the R command window. You get **17** printed as an output.

The function itself takes two arguments in our recipe but, in reality, it can take many arguments. Anything defined inside curly braces gets executed when we call `add()`. In our case, we request the user to input two variables, and the output is a simple sum.

See also

- ▶ Functions can be helpful in performing repetitive tasks such as generating plots or perform complicated calculations. Felix Schönbrodt has implemented visually weighted watercolor plots in R using a function on his blog at <http://www.nicebread.de/visually-weighted-watercolor-plots-new-variants-please-vote/>.
- ▶ We can generate similar plots simply by copying the function created by Felix in our R session and executing it. The plotting function created by Felix also provides users with different ways in which the R function's ability could be leveraged to perform repetitive tasks.

Writing if else statements in R

We often use `if` statements in MS Excel, but we can also write a small code to perform simple tasks in R.

How to do it...

The logic for `if else` statements is very simple and is as follows:

```
if(x>3){  
  print("greater value")  
}else {  
  print("lesser value")  
}
```

We can copy and paste the preceding statement in the R console or write a function that makes use of the `if else` logic.

How it works...

The logic behind `if` `else` statements is very simple. The following lines clearly state the logic:

```
if(condition) {  
  #perform some action  
} else {  
  #perform some other action  
}
```

The preceding code will check whether `x` is greater than or less than 3, and simply print it. In order to get the value, we type the following in the R command window:

```
x = 2
```

Basic loops in R

If we want to perform an action repeatedly in R, we can utilize the loop functionality.

How to do it...

The following lines of code multiply each element of `x` and `y` and store them as a vector `z`:

```
x = c(1:10)  
y = c(1:10)  
for(i in 1:10){  
  z[i] = x[i]*y[i]  
}
```

How it works...

In the preceding code, a calculation is executed 10 times. R performs any calculation specified within `{ }`. We are instructing R to multiply each element of `x` (using the `x[i]` notation) by each element in `y` and store the result in `z`.

Nested loops in R

We can nest loops, as well as `if` statements, to perform some more complicated tasks. In this recipe, we will first define a square matrix and then write a nested for loop to print only those values where `I = J`, namely, the values in the matrix placed in (1,1), (2,2), and so on.

How to do it...

We first define a matrix in R using the following `matrix()` function:

```
mat= matrix(1:25, 5,5)
```

Now, we use the following code to output only those elements where $I = J$:

```
for (i in 1:5){  
  for (j in 1:5){  
    if (i ==j){  
      print(mat[i,j])  
    }  
  }  
}
```

The `if` statement is nested inside two `for` loop statements. As we have a matrix, we have to use two `for` loops instead of just one. The output of the matrix would be values such as 1, 7, 13, and 19.

The `apply`, `lapply`, `sapply`, and `tapply` functions

R has some very handy functions such as `apply`, `sapply`, `tapply`, and `mapply`, that can be used to reduce the task of writing complicated statements. Also, using them makes our code look cleaner. The `apply()` function is similar to writing a loop statement.

The `lapply()` function is very similar to the `apply()` function but can be used on lists; this will return a list. The `sapply()` function is very similar to `lapply()` but returns a vector and not a list.

How to do it...

The `apply()` function can be used as follows:

```
mat= matrix(1:25, 5,5)  
apply(mat,1, sd)
```

The `lapply()` function can be used in the following way:

```
j = list(x = 1:4, b = rnorm(100,1,2))  
lapply(j,mean)
```

The `tapply()` function is useful when we have broken a vector into factors, groups, or categories:

```
tapply(mtcars$mpg,mtcars$gear,mean)
```

How it works...

The first argument in the `apply()` function is the data. The second argument takes two values: 1 and 2; if we state 1, R will perform a row-wise computation; if we mention 2, R will perform a column-wise computation. The third argument is the function. We would like to calculate the standard deviation of each row in R; hence we use the `sd` function as the third argument. Note that we can define our own function and replace it with the `sd` function.

With regard to the `lapply()` function, we have defined `J` as a list and would like to calculate the mean. The first argument in the `lapply()` function is the data and the second argument is the function used to process the data.

The first argument in the `tapply()` function is the data; in our case it is `mpg`. The second argument is the factor or the grouping; in this case it would be `gears`. The last argument is the function used to process the data. We would like to calculate the mean of `mpg` for each unique gear (3, 4, and 5 gears) in the `mtcars` data.

Using `par` to beautify a plot in R

One quick and easy way to edit a plot is by generating the plot in R and then using Inkspace or any other software to edit it. We can save some valuable time if we know some basic edits that can be applied on a plot by setting them in a `par()` function. All the available options to edit a plot can be studied in detail by typing `?par` in the command window.

How to do it...

In the following code, I have highlighted some commonly used parameters:

```
x=c(1:10)
y=c(1:10)
par(bg = "#646989", las = 1, col.lab = "black", col.axis =
    "white", bty = "n", cex.axis = 0.9, cex.lab = 1.5)
plot(x,y, pch = 20, xlab = "fake x data", ylab = "fake y data")
```

How it works...

Under the `par()` function, we have set the background color using the `bg` = argument. The `las` = argument changes the orientation of the labels. The `col.lab` and `col.axis` arguments are used to specify the color of the labels as well as the axis. The `cex` argument is used to specify the size of the labels and axis. The `bty` argument is used to specify the box style in R.

Saving plots

We can save a plot in various formats, such as .jpeg, .svg, .pdf, or .png. I prefer saving a plot as a .png file, as it is easier to edit a plot with Inkspace if saved in the PNG format.

How to do it...

To save a plot in the .png format, we can use the `png()` function as follows:

```
png("TEST.png", width = 300, height = 600)
plot(x,y, xlab = "x axis", ylab = "y axis", cex.lab = 3,col.lab =
      "red", main = "some data", cex.main=1.5, col.main = "red")
dev.off()
```

How it works...

We have used the `png()` function to save the plot as a PNG. To save a plot as a PDF, SVG, or JPEG, we can use the `pdf()`, `svg()`, or `jpeg()` functions, respectively.

The first argument in the `png()` function is the name of the file with the extension, followed by the width and height of the plot. We can now use the `plot()` function to generate a plot; any subsequent plots will also be saved with a .png extension, unless the `dev.off()` function is passed. The `dev.off()` function instructs R that we do not need to save the plots.

2

Practical Machine Learning with R

In this chapter, we will cover the following topics:

- ▶ Downloading and installing R
- ▶ Downloading and installing RStudio
- ▶ Installing and loading packages
- ▶ Reading and writing data
- ▶ Using R to manipulate data
- ▶ Applying basic statistics
- ▶ Visualizing data
- ▶ Getting a dataset for machine learning

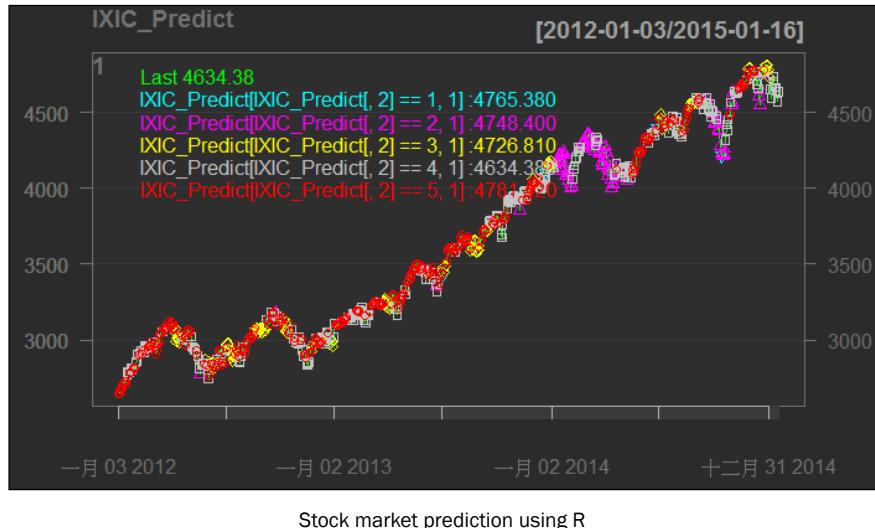
Introduction

The aim of machine learning is to uncover hidden patterns, unknown correlations, and find useful information from data. In addition to this, through incorporation with data analysis, machine learning can be used to perform predictive analysis. With machine learning, the analysis of business operations and processes is not limited to human scale thinking; machine scale analysis enables businesses to capture hidden values in big data.

Machine learning has similarities to the human reasoning process. Unlike traditional analysis, the generated model cannot evolve as data is accumulated. Machine learning can learn from the data that is processed and analyzed. In other words, the more data that is processed, the more it can learn.

R, as a dialect of GNU-S, is a powerful statistical language that can be used to manipulate and analyze data. Additionally, R provides many machine learning packages and visualization functions, which enable users to analyze data on the fly. Most importantly, R is open source and free.

Using R greatly simplifies machine learning. All you need to know is how each algorithm can solve your problem, and then you can simply use a written package to quickly generate prediction models on data with a few command lines. For example, you can either perform Naïve Bayes for spam mail filtering, conduct k-means clustering for customer segmentation, use linear regression to forecast house prices, or implement a hidden Markov model to predict the stock market, as shown in the following screenshot:



Moreover, you can perform nonlinear dimension reduction to calculate the dissimilarity of image data, and visualize the clustered graph, as shown in the following screenshot. All you need to do is follow the recipes provided in this book.



A clustered graph of face image data

This chapter serves as an overall introduction to machine learning and R; the first few recipes introduce how to set up the R environment and integrated development environment, RStudio. After setting up the environment, the following recipe introduces package installation and loading. In order to understand how data analysis is practiced using R, the next four recipes cover data read/write, data manipulation, basic statistics, and data visualization using R. The last recipe in the chapter lists useful data sources and resources.

Downloading and installing R

To use R, you must first install it on your computer. This recipe gives detailed instructions on how to download and install R.

Getting ready

If you are new to the R language, you can find a detailed introduction, language history, and functionality on the official website (<http://www.r-project.org/>). When you are ready to download and install R, please access the following link: <http://cran.r-project.org/>.

How to do it...

Please perform the following steps to download and install R for Windows and Mac users:

1. Go to the R CRAN website, <http://www.r-project.org/>, and click on the **download R** link, that is, <http://cran.r-project.org/mirrors.html>:

The R Project for Statistical Computing

About R
What is R?
Contributors
Screenshots
What's new?

Download, Packages
CRAN

R Project
Foundation
Members & Donors
Mailing Lists
Bug Tracking
Developer Page
Conferences
Search

Documentation
Manuals
FAQs
The R Journal
Wiki
Books
Certification
Other

Getting Started:

- R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and Mac OS. To [download R](#), please choose your preferred CRAN mirror.
- If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

2. You may select the mirror location closest to you:

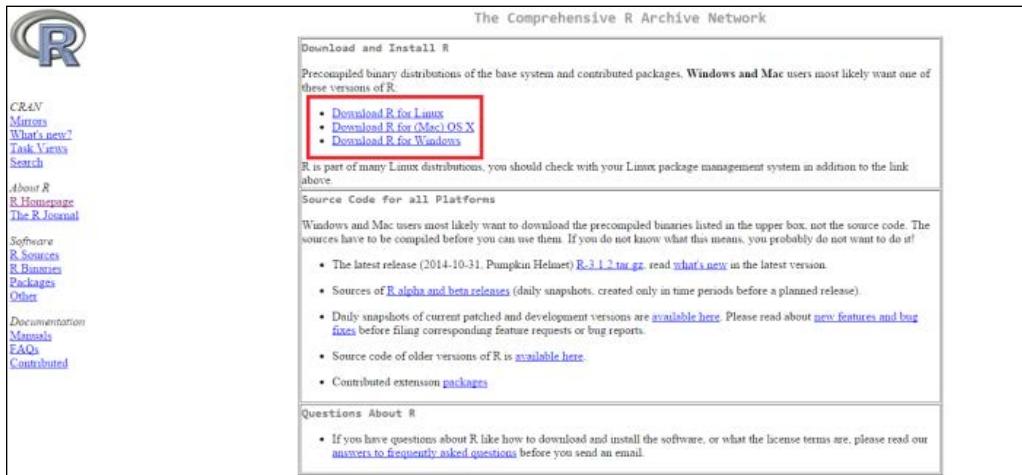
CRAN Mirrors

The Comprehensive R Archive Network is available at the following URLs. Please choose a location close to you. Some statistics on the status of the mirrors can be found here: [main page](#), [windows release](#), [windows old release](#).

O-Cloud http://cran.rstudio.com/	Rotundo, automatic redirection to servers worldwide
Argentina http://mirror.fcaglp.unlp.edu.ar/CRAN/	Universidad Nacional de La Plata
Australia http://cran.csiro.au/	CSIRO
Australia http://cran.ms.unimelb.edu.au/	University of Melbourne
Austria http://cran.at.r-project.org/	Wirtschaftsuniversitaet Wien
Belgium http://www.freestatistics.org/cran/	K.U Leuven Association
Brazil http://abc.gib.uesc.br/mirror/cran/	Center for Comp. Biol. at Universidade Estadual de Santa Cruz
Brazil http://cran-i.csl.ufpr.br/	Universidade Federal do Paraná
Brazil http://cran.fioruz.hu/	Oswaldo Cruz Foundation, Rio de Janeiro
Brazil http://www.vps.fimvz.usp.br/CRAN/	University of Sao Paulo, Sao Paulo
Brazil http://brieger.esalq.usp.br/CRAN/	University of Sao Paulo, Puncicaba
Canada http://cran.stats.ubc.ca/	Simon Fraser University, Burnaby
Canada http://mirror.cts.dal.ca/cran/	Dalhousie University, Halifax
Canada http://cran.umtstat.uregina.ca/	University of Toronto
Canada http://cran.skaakaforyou.com/	iWeb, Montreal
Canada http://cran.parentingamerica.com/	iWeb, Montreal
CRAN Asia http://cran.r-project.org/CRAN/ASIA/	
CRAN Africa http://cran.r-project.org/CRAN/AFRICA/	
CRAN International http://cran.r-project.org/CRAN/INTERNATIONAL/	

CRAN mirrors

3. Select the correct download link based on your operating system:



The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages. Windows and Mac users most likely want one of these versions of R.

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2014-10-31, Pumpkin Helmet) [R-3.1.2.tar.gz](#), read [what's new](#) in the latest version.
- Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are [available here](#). Please read about [new features and bug fixes](#) before filing corresponding feature requests or bug reports.
- Source code of older versions of R is [available here](#).
- Contributed extension [packages](#)

Questions About R

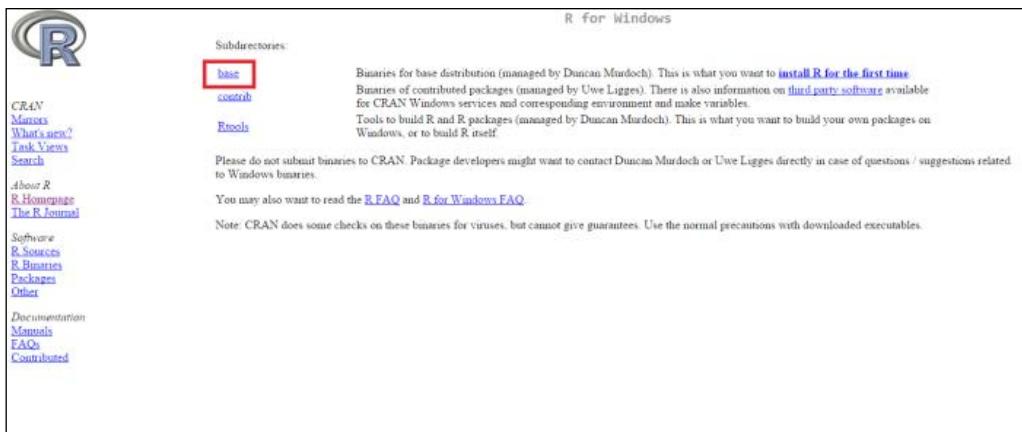
- If you have questions about R, like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

Click on the download link based on your OS

As the installation of R differs for Windows and Mac, the steps required to install R for each OS are provided here.

For Windows users:

1. Click on **Download R for Windows**, as shown in the following screenshot, and then click on **base**:



R for Windows

Subdirectories:

- [base](#) (highlighted with a red box)
- [contrib](#)
- [tools](#)

Binaries for base distribution (managed by Duncan Murdoch). This is what you want to [install R for the first time](#). Binaries of contributed packages (managed by Uwe Ligges). There is also information on [third party software](#) available for CRAN Windows services and corresponding environment and make variables. Tools to build R and R packages (managed by Duncan Murdoch). This is what you want to build your own packages on Windows, or to build R itself.

Please do not submit binaries to CRAN. Package developers might want to contact Duncan Murdoch or Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the [R FAQ](#) and [R for Windows FAQ](#).

Note: CRAN does some checks on these binaries for viruses, but cannot give guarantees. Use the normal precautions with downloaded executables.

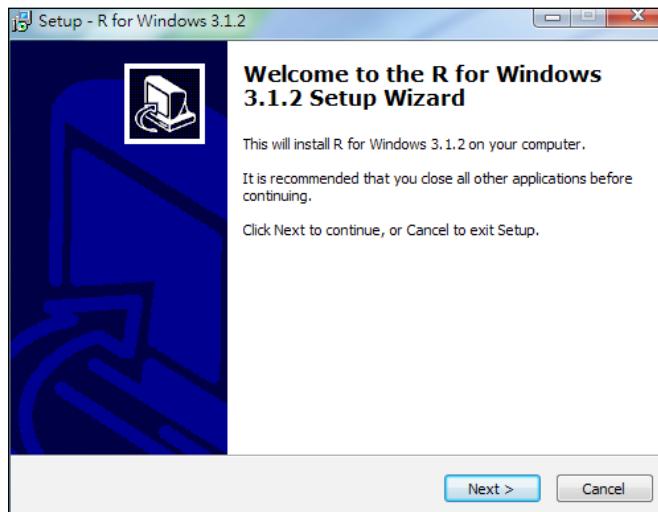
Go to "Download R for Windows" and click "base"

2. Click on Download R 3.x.x for Windows:

The screenshot shows the CRAN Mirrors page for R 3.1.2 for Windows. At the top, there's a large 'R' logo. Below it, a navigation bar with links like 'CRAN', 'Manus', 'What's new?', 'Task Views', 'Search', 'About R', 'R Homepage', 'The R Journal', 'Software', 'R Sources', 'R Binaries', 'Packages', 'Other', 'Documentation', 'Manuals', 'FAQs', and 'Contributed'. The main content area has a title 'R-3.1.2 for Windows (32/64 bit)'. A prominent blue button labeled 'Download R 3.1.2 for Windows' (54 megabytes, 32/64 bit) is highlighted with a red box. Below it are links for 'Installation and other instructions' and 'New features in this version'. A note about file integrity follows, along with a 'Frequently asked questions' section containing three bullet points: 'How do I install R when using Windows Vista?', 'How do I update packages in my previous version of R?', and 'Should I run 32-bit or 64-bit R?'. A note below says 'Please see the [R FAQ](#) for general information about R and the [R Windows FAQ](#) for Windows-specific information.' Another section titled 'Other builds' lists 'Patches to this release are incorporated in the [r-patched snapshot build](#)', 'A build of the development version (which will eventually become the next major release of R) is available in the [r-devel snapshot build](#)', and a link to 'Previous releases'. A note for webmasters at the bottom links to '[CRAN MIRROR-base/windows/base/release.htm](#)'. At the very bottom, it says 'Last change: 2014-10-31, by Duncan Murdoch'.

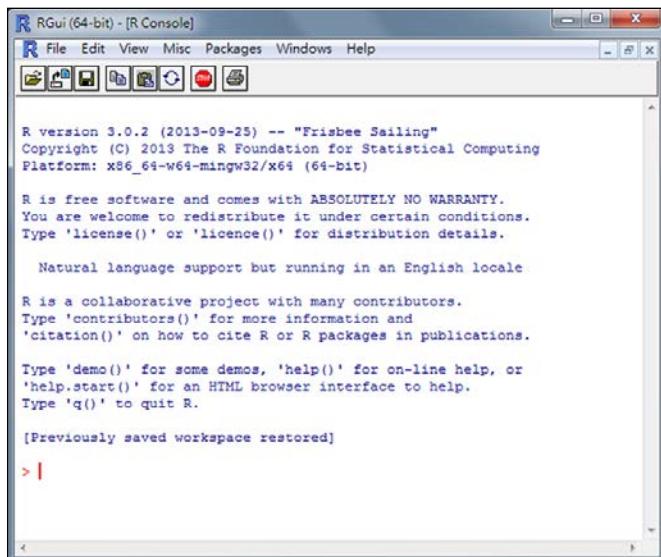
Click "Download R 3.x.x for Windows"

3. The installation file should be downloaded. Once the download is finished, you can double-click on the installation file and begin installing R:



4. The Windows installation of R is quite straightforward; the installation GUI may instruct you on how to install the program step by step (public license, destination location, select components, startup options, startup menu folder, and select additional tasks). Leave all the installation options as the default settings if you do not want to make any changes.

5. After successfully completing the installation, a shortcut to the R application will appear in your Start menu, which will open the R Console:



The Windows R Console

For Mac OS X users:

1. Go to **Download R for (Mac) OS X**, as shown in this screenshot.
2. Click on the latest version (.pkg file extension) according to your Mac OS version:

This screenshot shows the CRAN R for Mac OS X download page. At the top left is the CRAN logo. On the left, there's a sidebar with links: About R, R Homepage, The R Journal, Software, R Sources, R Binaries, Packages, Other, Documentation, Manuals, FAQs, Contributed, and What's new?.

The main content area has a heading "R for Mac OS X". It says: "This directory contains binaries for a base distribution and packages to run on Mac OS X (release 10.6 and above). Mac OS 8.6 to 9.2 (and Mac OS X 10.1) are no longer supported but you can find the last supported release of R for these systems (which is R 3.1.2) [here](#). Releases for old Mac OS X systems (through Mac OS X 10.5) and PowerPC Macs can be found in the [old](#) directory."

Below this, it says: "Note: R does not have Mac OS X systems and cannot check these binaries for viruses. Although we take precautions when assembling binaries, please use the normal precautions with downloaded executables."

It lists two packages:

- R-3.1.2-snowleopard.pkg**
MD5 hash: b6932006f6732192992dceff5defd
SHA1 hash: e3ec5c453d974fae237fb50f0fdef1fe1b6993
(ca. 683MB)
- R-3.1.2-mavericks.pkg**
MD5 hash: dffbe6e20357dd058a1691cf84e091
SHA1 hash: 61c76988f024bf48032009fb19d8413cf2aefba
(ca. 553MB)

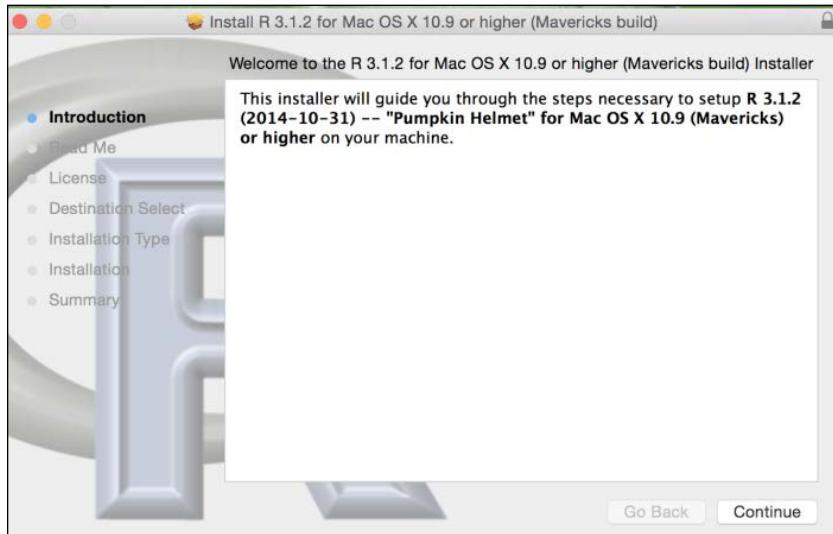
Both packages are described as "R 3.1.2 binary for Mac OS X 10.6 (Snow Leopard) and higher, signed package. Contains R 3.1.2 framework, R.app GUI 1.65 in 64-bit for Intel Macs. The above file is an Installer package which can be installed by double-clicking. Depending on your browser, you may need to press the control key and click on this link to download the file."

The "R-3.1.2-mavericks.pkg" entry continues: "This package contains the R framework, 64-bit GUI (R.app) and Tk/Tk 8.6.0 X11 libraries. The latter component is optional and can be uninstalled when choosing 'custom install', it is only needed if you want to use the tcltk R package. GNU Fortran is NOT included (needed if you want to compile packages from sources that contain FORTRAN code) please see the tools directory."

At the bottom, it notes: "R 3.1.2 binary for Mac OS X 10.9 (Mavericks) and higher, signed package. It contains the same software versions as above, but this R build has been built with Xcode 5 to leverage new compilers and functionalities in Mavericks not available in earlier OS X versions."

Finally, it says: "Note: the use of X11 (including tcltk) requires [XQuartz](#) to be installed since it is no longer part of OS X. Always re-install XQuartz when upgrading your OS X to a new major version."

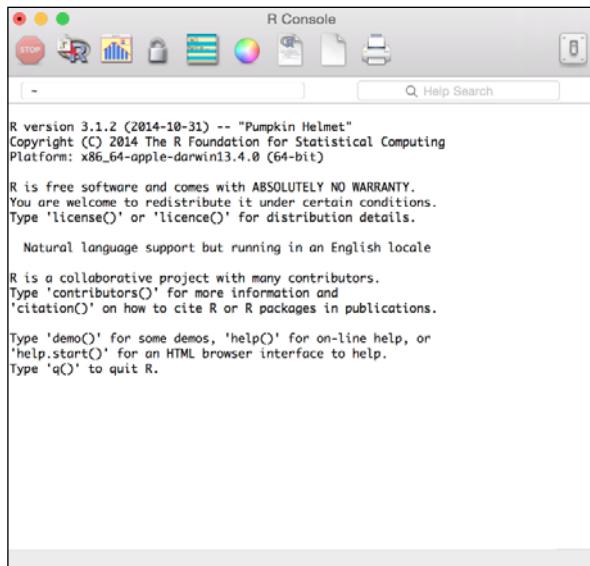
3. Double-click on the downloaded installation file (.pkg extension) and begin to install R. Leave all the installation options as the default settings if you do not want to make any changes:



4. Follow the onscreen instructions, **Introduction**, **Read Me**, **License**, **Destination Select**, **Installation Type**, **Installation**, **Summary**, and click on **continue** to complete the installation.
5. After the file is installed, you can use **Spotlight Search** or go to the application folder to find R:



6. Click on R to open **R Console**:



As an alternative to downloading a Mac .pkg file to install R, Mac users can also install R using Homebrew:

1. Download XQuartz-2.X.X.dmg from <https://xquartz.macosforge.org/> landing/.
2. Double-click on the .dmg file to mount it.
3. Update brew with the following command line:
`$ brew update`
4. Clone the repository and symlink all its formulae to homebrew/science:
`$ brew tap homebrew/science`
5. Install gfortran:
`$ brew install gfortran`
6. Install R:
`$ brew install R`

For Linux users, there are precompiled binaries for Debian, Red Hat, SUSE, and Ubuntu. Alternatively, you can install R from a source code. Besides downloading precompiled binaries, you can install R for Linux through a package manager. Here are the installation steps for CentOS and Ubuntu.

Downloading and installing R on Ubuntu:

1. Add the entry to the /etc/apt/sources.list file:

```
$ sudo sh -c "echo 'deb http://cran.stat.ucla.edu/bin/linux/ubuntu precise/' >> /etc/apt/sources.list"
```

2. Then, update the repository:

```
$ sudo apt-get update
```

3. Install R with the following command:

```
$ sudo apt-get install r-base
```

4. Start R in the command line:

```
$ R
```

Downloading and installing R on CentOS 5:

1. Get rpm CentOS5 RHEL EPEL repository of CentOS5:

```
$ wget http://dl.fedoraproject.org/pub/epel/5/x86_64/epel-release-5-4.noarch.rpm
```

2. Install CentOS5 RHEL EPEL repository:

```
$ sudo rpm -Uvh epel-release-5-4.noarch.rpm
```

3. Update the installed packages:

```
$ sudo yum update
```

4. Install R through the repository:

```
$ sudo yum install R
```

5. Start R in the command line:

```
$ R
```

Downloading and installing R on CentOS 6:

1. Get rpm CentOS5 RHEL EPEL repository of CentOS6:

```
$ wget http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
```

2. Install the CentOS5 RHEL EPEL repository:

```
$ sudo rpm -Uvh epel-release-6-8.noarch.rpm
```

3. Update the installed packages:

```
$ sudo yum update
```

4. Install R through the repository:

```
$ sudo yum install R
```

5. Start R in the command line:

```
$ R
```

How it works...

CRAN provides precompiled binaries for Linux, Mac OS X, and Windows. For Mac and Windows users, the installation procedures are straightforward. You can generally follow onscreen instructions to complete the installation. For Linux users, you can use the package manager provided for each platform to install R or build R from the source code.

See also

- ▶ For those planning to build R from the source code, refer to **R Installation and Administration** (<http://cran.r-project.org/doc/manuals/R-admin.html>), which illustrates how to install R on a variety of platforms.

Downloading and installing RStudio

To write an R script, one can use R Console, R commander, or any text editor (EMACS, VIM, or sublime). However, the assistance of RStudio, an **integrated development environment (IDE)** for R, can make development a lot easier.

RStudio provides comprehensive facilities for software development. Built-in features such as syntax highlighting, code completion, and smart indentation help maximize productivity. To make R programming more manageable, RStudio also integrates the main interface into a four-panel layout. It includes an interactive R Console, a tabbed source code editor, a panel for the currently active objects/history, and a tabbed panel for the file browser/plot window/package install window/R help window. Moreover, RStudio is open source and is available for many platforms, such as Windows, Mac OS X, and Linux. This recipe shows how to download and install RStudio.

Getting ready

RStudio requires a working R installation; when RStudio loads, it must be able to locate a version of R. You must therefore have completed the previous recipe with R installed on your OS before proceeding to install RStudio.

How to do it...

Perform the following steps to download and install RStudio for Windows and Mac users:

1. Access RStudio's official site by using the following URL: <http://www.rstudio.com/products/RStudio/>.

The screenshot shows the RStudio Desktop product page. At the top, there is a navigation bar with links for Products, Resources, Pricing, About Us, Blog, and a search icon. Below the navigation bar, the page title is "RStudio Desktop". There are two main sections: "Open Source Edition" and "Commercial License". The "Open Source Edition" section lists features such as Access RStudio locally, Syntax highlighting, code completion, and smart indentation. The "Commercial License" section adds features like a commercial license for organizations not able to use AGPL software and access to priority support. Below these sections, there are tables for Support (Community forums only), License (AGPL v3), and Pricing (Free). At the bottom, there are two blue buttons: "DOWNLOAD RSTUDIO DESKTOP" and "BUY NOW".

2. For a desktop version installation, click on **Download RStudio Desktop** (<http://www.rstudio.com/products/rstudio/download/>) and choose the RStudio recommended for your system. Download the relevant packages:

Download RStudio

Home / Overview / RStudio / Download RStudio

RStudio is a set of integrated tools designed to help you be more productive with R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management.

If you run R on a Linux server and want to enable users to remotely access RStudio using a web browser please download RStudio Server.

Do you need support or a commercial license?

Check out our commercial offerings

Download RStudio Desktop v0.98.1091 — Release Notes

RStudio requires R 2.11.1 (or higher). If you don't already have R, you can download it here.



Installers for ALL Platforms

Installers	Size	Date	MD5
RStudio 0.98.1091 - Windows XP/Vista/7/8	45 MB	2014-11-06	910fba345c0555597bda498cad1302b0
RStudio 0.98.1091 - Mac OS X 10.6+ (64-bit)	38.4 MB	2014-11-06	9c7d2cea702cf478a4a774b79134b3ee
RStudio 0.98.1091 - Debian 6+/Ubuntu 10.04+ (32-bit)	53 MB	2014-11-06	0bc579cbee43a514e3fb456959a0ada
RStudio 0.98.1091 - Debian 6+/Ubuntu 10.04+ (64-bit)	54.9 MB	2014-11-06	1e88e6775993daa8cf7d4d89f76af7e0
RStudio 0.98.1091 - Fedora 13+/RedHat 7+/openSUSE 11.4+ (32-bit)	53.4 MB	2014-11-06	3ae5923956166f90ecc1cb721b02f90f
RStudio 0.98.1091 - Fedora 13+/RedHat 7+/openSUSE 11.4+ (64-bit)	55 MB	2014-11-06	6d1ac08ceed731f5750f3de9a911511b

Zip/Tarballs

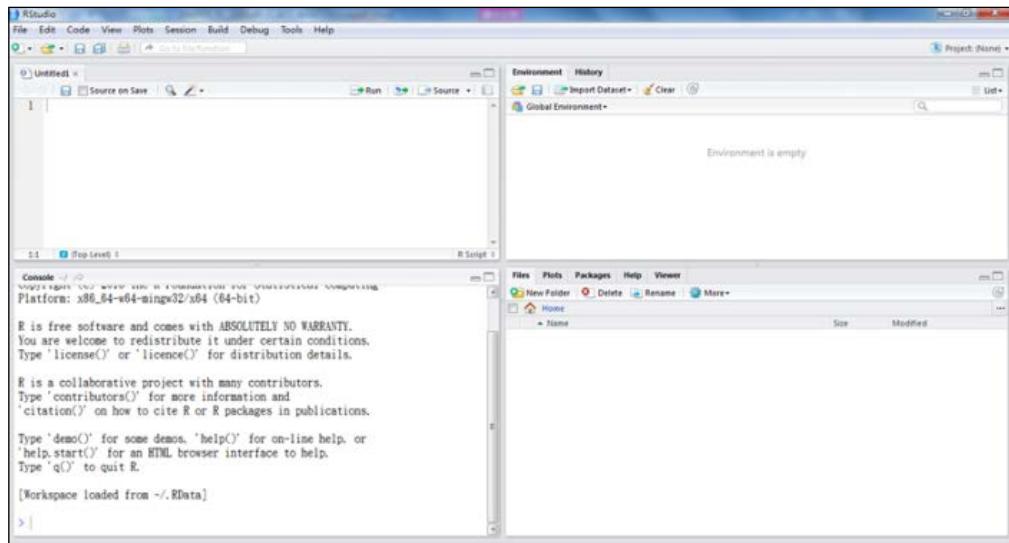
3. Install RStudio by double-clicking on the downloaded packages. For Windows users, follow the onscreen instruction to install the application:



4. For Mac users, simply drag the RStudio icon to the Applications folder:



5. Start RStudio:



The RStudio console

Perform the following steps for downloading and installing RStudio for Ubuntu/Debian and RedHat/Centos users:

For Debian(6+)/Ubuntu(10.04+) 32-bit:

```
$ wget http://download1.rstudio.org/rstudio-0.98.1091-i386.deb  
$ sudo gdebi rstudio-0.98.1091-i386.deb
```

For Debian(6+)/Ubuntu(10.04+) 64-bit:

```
$ wget http://download1.rstudio.org/rstudio-0.98.1091-amd64.deb  
$ sudo gdebi rstudio-0.98.1091-amd64.deb
```

For RedHat/CentOS(5,4+) 32 bit:

```
$ wget http://download1.rstudio.org/rstudio-0.98.1091-i686.rpm
```

```
$ sudo yum install --nogpgcheck rstudio-0.98.1091-i686.rpm
```

For RedHat/CentOS(5,4+) 64 bit:

```
$ wget http://download1.rstudio.org/rstudio-0.98.1091-x86_64.rpm
```

```
$ sudo yum install --nogpgcheck rstudio-0.98.1091-x86_64.rpm
```

How it works

The RStudio program can be run on the desktop or through a web browser. The desktop version is available for Windows, Mac OS X, and Linux platforms with similar operations across all platforms. For Windows and Mac users, after downloading the precompiled package of RStudio, follow the onscreen instructions, shown in the preceding steps, to complete the installation. Linux users may use the package management system provided for installation.

See also

- ▶ In addition to the desktop version, users may install a server version to provide access to multiple users. The server version provides a URL that users can access to use the RStudio resources. To install RStudio, please refer to the following link: <http://www.rstudio.com/ide/download/server.html>. This page provides installation instructions for the following Linux distributions: Debian (6+), Ubuntu (10.04+), RedHat, and CentOS (5.4+).
- ▶ For other Linux distributions, you can build RStudio from the source code.

Installing and loading packages

After successfully installing R, users can download, install, and update packages from the repositories. As R allows users to create their own packages, official and non-official repositories are provided to manage these user-created packages. **CRAN** is the official R package repository. Currently, the CRAN package repository features 6,379 available packages (as of 02/27/2015). Through the use of the packages provided on CRAN, users may extend the functionality of R to machine learning, statistics, and related purposes. CRAN is a network of FTP and web servers around the world that store identical, up-to-date versions of code and documentation for R. You may select the closest CRAN mirror to your location to download packages.

Getting ready

Start an R session on your host computer.

How to do it...

Perform the following steps to install and load R packages:

1. To load a list of installed packages:

```
> library()
```

2. Setting the default CRAN mirror:

```
> chooseCRANmirror()
```

R will return a list of CRAN mirrors, and then ask the user to either type a mirror ID to select it, or enter zero to exit:

1. Install a package from CRAN; take package e1071 as an example:

```
> install.packages("e1071")
```

2. Update a package from CRAN; take package e1071 as an example:

```
> update.packages("e1071")
```

3. Load the package the package:

```
> library(e1071)
```

4. If you would like to view the documentation of the package, you can use the `help` function:

```
> help(package = "e1071")
```

5. If you would like to view the documentation of the function, you can use the `help` function:

```
> help(svm, e1071)
```

6. Alternatively, you can use the help shortcut, `?`, to view the help document for this function:

```
> ?e1071::svm
```

7. If the function does not provide any documentation, you may want to search the supplied documentation for a given keyword. For example, if you wish to search for documentation related to `svm`:

```
> help.search("svm")
```

8. Alternatively, you can use `??` as the shortcut for `help.search`:

```
> ??svm
```

9. To view the argument taken for the function, simply use the `args` function. For example, if you would like to know the argument taken for the `lm` function:

```
> args(lm)
```

10. Some packages will provide examples and demos; you can use `example` or `demo` to view an example or demo. For example, one can view an example of the `lm` package and a demo of the `graphics` package by typing the following commands:

```
> example(lm)  
> demo(graphics)
```

11. To view all the available demos, you may use the `demo` function to list all of them:

```
> demo()
```

How it works

This recipe first introduces how to view loaded packages, install packages from CRAN, and load new packages. Before installing packages, those of you who are interested in the listing of the CRAN package can refer to http://cran.r-project.org/web/packages/available_packages_by_name.html.

When a package is installed, documentation related to the package is also provided. You are, therefore, able to view the documentation or the related help pages of installed packages and functions. Additionally, demos and examples are provided by packages that can help users understand the capability of the installed package.

See also

- ▶ Besides installing packages from CRAN, there are other R package repositories, including Crantastic, a community site for rating and reviewing CRAN packages, and R-Forge, a central platform for the collaborative development of R packages. In addition to this, Bioconductor provides R packages for the analysis of genomic data.
- ▶ If you would like to find relevant functions and packages, please visit the list of task views at <http://cran.r-project.org/web/views/>, or search for keywords at <http://rseek.org>.

Reading and writing data

Before starting to explore data, you must load the data into the R session. This recipe will introduce methods to load data either from a file into the memory or use the predefined data within R.

Getting ready

First, start an R session on your machine. As this recipe involves steps toward the file IO, if the user does not specify the full path, read and write activity will take place in the current working directory.

You can simply type `getwd()` in the R session to obtain the current working directory location. However, if you would like to change the current working directory, you can use `setwd("<path>")`, where `<path>` can be replaced as your desired path, to specify the working directory.

How to do it...

Perform the following steps to read and write data with R:

1. To view the built-in datasets of R, type the following command:

```
> data()
```

2. R will return a list of datasets in a `dataset` package, and the list comprises the name and description of each dataset.

3. To load the dataset `iris` into an R session, type the following command:

```
> data(iris)
```

4. The dataset `iris` is now loaded into the data frame format, which is a common data structure in R to store a data table.

5. To view the data type of `iris`, simply use the `class` function:

```
> class(iris)  
[1] "data.frame"
```

6. The `data.frame` console print shows that the `iris` dataset is in the structure of data frame.

7. Use the `save` function to store an object in a file. For example, to save the loaded `iris` data into `myData.RData`, use the following command:

```
> save(iris, file="myData.RData")
```

8. Use the `load` function to read a saved object into an R session. For example, to load `iris` data from `myData.RData`, use the following command:

```
> load("myData.RData")
```

9. In addition to using built-in datasets, R also provides a function to import data from text into a data frame. For example, the `read.table` function can format a given text into a data frame:

```
> test.data = read.table(header = TRUE, text = "  
+ a b  
+ 1 2  
+ 3 4  
+ ")
```

10. You can also use `row.names` and `col.names` to specify the names of columns and rows:

```
> test.data = read.table(text = "
+ 1 2
+ 3 4",
+ col.names=c("a","b"),
+ row.names = c("first","second"))
```

11. View the class of the `test.data` variable:

```
> class(test.data)
[1] "data.frame"
```

12. The `class` function shows that the `test.data` variable contains a data frame.

13. In addition to importing data by using the `read.table` function, you can use the `write.table` function to export data to a text file:

```
> write.table(test.data, file = "test.txt" , sep = " ")
```

14. The `write.table` function will write the content of `test.data` into `test.txt` (the written path can be found by typing `getwd()`), with a separation delimiter as white space.

15. Similar to `write.table`, `write.csv` can also export data to a file. However, `write.csv` uses a comma as the default delimiter:

```
> write.csv(test.data, file = "test.csv")
```

16. With the `read.csv` function, the `csv` file can be imported as a data frame. However, the last example writes column and row names of the data frame to the `test.csv` file. Therefore, specifying header to `TRUE` and row names as the first column within the function can ensure the read data frame will not treat the header and the first column as values:

```
> csv.data = read.csv("test.csv", header = TRUE, row.names=1)
> head(csv.data)
   a b
1 1 2
2 3 4
```

How it works

Generally, data for collection may be in multiple files and different formats. To exchange data between files and RData, R provides many built-in functions, such as `save`, `load`, `read.csv`, `read.table`, `write.csv`, and `write.table`.

This example first demonstrates how to load the built-in dataset `iris` into an R session. The `iris` dataset is the most famous and commonly used dataset in the field of machine learning. Here, we use the `iris` dataset as an example. The recipe shows how to save RData and load it with the `save` and `load` functions. Furthermore, the example explains how to use `read.table`, `write.table`, `read.csv`, and `write.csv` to exchange data from files to a data frame. The use of the R IO function to read and write data is very important as most of the data sources are external. Therefore, you have to use these functions to load data into an R session.

See also

For the `load`, `read.table`, and `read.csv` functions, the file to be read can also be a complete URL (for supported URLs, use `?url` for more information).

On some occasions, data may be in an Excel file instead of a flat text file. The `WriteXLS` package allows writing an object into an Excel file with a given variable in the first argument and the file to be written in the second argument:

1. Install the `WriteXLS` package:

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

2. Load the `WriteXLS` package:

```
> library("WriteXLS")
```

3. Use the `writeXLS` function to write the data frame `iris` into a file named `iris.xls`:

```
> WriteXLS("iris", ExcelFileName="iris.xls")
```

Using R to manipulate data

This recipe will discuss how to use the built-in R functions to manipulate data. As data manipulation is the most time consuming part of most analysis procedures, you should gain knowledge of how to apply these functions on data.

Getting ready

Ensure you have completed the previous recipes by installing R on your operating system.

How to do it...

Perform the following steps to manipulate the data with R.

Subset the data using the bracelet notation:

1. Load the dataset iris into the R session:

```
> data(iris)
```

2. To select values, you may use a bracket notation that designates the indices of the dataset. The first index is for the rows and the second for the columns:

```
> iris[1,"Sepal.Length"]
```

```
[1] 5.1
```

3. You can also select multiple columns using c():

```
> Sepal.iris = iris[, c("Sepal.Length", "Sepal.Width")]
```

4. You can then use str() to summarize and display the internal structure of Sepal.iris:

```
> str(Sepal.iris)
```

```
'data.frame': 150 obs. of 2 variables:
```

```
 $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
```

```
 $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ..
```

5. To subset data with the rows of given indices, you can specify the indices at the first index with the bracket notation. In this example, we show you how to subset data with the top five records with the Sepal.Length column and the Sepal.Width selected:

```
> Five.Sepal.iris = iris[1:5, c("Sepal.Length", "Sepal.Width")]
```

```
> str(Five.Sepal.iris)
```

```
'data.frame': 5 obs. of 2 variables:
```

```
 $ Sepal.Length: num 5.1 4.9 4.7 4.6 5
```

```
 $ Sepal.Width : num 3.5 3 3.2 3.1 3.6
```

6. It is also possible to set conditions to filter the data. For example, to filter returned records containing the setosa data with all five variables. In the following example, the first index specifies the returning criteria, and the second index specifies the range of indices of the variable returned:

```
> setosa.data = iris[iris$Species=="setosa",1:5]
```

```
> str(setosa.data)
```

```
'data.frame': 50 obs. of 5 variables:
```

```
 $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
```

```
 $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
```

```
 $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
```

```
 $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
```

```
 $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

```
1 1 1 1 1 1 1 1 1 1 ...
```

7. Alternatively, the `which` function returns the indexes of satisfied data. The following example returns indices of the iris data containing species equal to `setosa`:

```
> which(iris$Species=="setosa")
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
[37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

8. The indices returned by the operation can then be applied as the index to select the iris containing the setosa species. The following example returns the setosa with all five variables:

```
> setosa.data = iris[which(iris$Species=="setosa"),1:5]
> str(setosa.data)
'data.frame': 50 obs. of 5 variables:
 $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1
1 1 1 1 1 1 1 1 ...
```

Subset data using the `subset` function:

1. Besides using the bracket notation, R provides a `subset` function that enables users to subset the data frame by observations with a logical statement.
2. First, subset species, sepal length, and sepal width out of the iris data. To select the sepal length and width out of the iris data, one should specify the column to be subset in the `select` argument:

```
> Sepal.data = subset(iris, select=c("Sepal.Length", "Sepal.Width"))
> str(Sepal.data)
'data.frame': 150 obs. of 2 variables:
 $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
```

This reveals that `Sepal.data` contains 150 objects with the `Sepal.Length` variable and `Sepal.Width`.

1. On the other hand, you can use a `subset` argument to get subset data containing `setosa` only. In the second argument of the `subset` function, you can specify the subset criteria:

```

> setosa.data = subset(iris, Species == "setosa")
> str(setosa.data)
'data.frame': 50 obs. of 5 variables:
 $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",...: 1 1
1 1 1 1 1 1 1 ...

```

2. Most of the time, you may want to apply a union or intersect a condition while subsetting data. The OR and AND operations can be further employed for this purpose. For example, if you would like to retrieve data with Petal.Width ≥ 0.2 and Petal.Length ≤ 1.4 :

```

> example.data= subset(iris, Petal.Length <=1.4 & Petal.Width >=
0.2, select=Species )
> str(example.data)
'data.frame': 21 obs. of 1 variable:
 $ Species: Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1
1 1 1 1 1 ...

```

Merging data: merging data involves joining two data frames into a merged data frame by a common column or row name. The following example shows how to merge the flower.type data frame and the first three rows of the iris with a common row name within the Species column:

```

> flower.type = data.frame(Species = "setosa", Flower = "iris")
> merge(flower.type, iris[1:3,], by ="Species")
   Species Flower Sepal.Length Sepal.Width Petal.Length Petal.Width
1  setosa   iris       5.1        3.5       1.4        0.2
2  setosa   iris       4.9        3.0       1.4        0.2
3  setosa   iris       4.7        3.2       1.3        0.2

```

Ordering data: the order function will return the index of a sorted data frame with a specified column. The following example shows the results from the first six records with the sepal length ordered (from big to small) iris data

```

> head(iris[order(iris$Sepal.Length, decreasing = TRUE),])
   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
132         7.9       3.8       6.4       2.0 virginica
118         7.7       3.8       6.7       2.2 virginica

```

119	7.7	2.6	6.9	2.3	virginica
123	7.7	2.8	6.7	2.0	virginica
136	7.7	3.0	6.1	2.3	virginica
106	7.6	3.0	6.6	2.1	virginica

How it works

Before conducting data analysis, it is important to organize collected data into a structured format. Therefore, we can simply use the R data frame to subset, merge, and order a dataset. This recipe first introduces two methods to subset data: one uses the bracket notation, while the other uses the `subset` function. You can use both methods to generate the subset data by selecting columns and filtering data with the given criteria. The recipe then introduces the `merge` function to merge data frames. Last, the recipe introduces how to use `order` to sort the data.

There's more...

The `sub` and `gsub` functions allow using regular expression to substitute a string. The `sub` and `gsub` functions perform the replacement of the first and all the other matches, respectively:

```
> sub("e", "q", names(iris))
[1] "Sqpal.Length" "Sqpal.Width"   "Pqtal.Length" "Pqtal.Width"   "Spqcies"
> gsub("e", "q", names(iris))
[1] "Sqpal.Lqngth" "Sqpal.Width"   "Pqtal.Lqngth" "Pqtal.Width"   "Spqcqqs"
```

Applying basic statistics

R provides a wide range of statistical functions, allowing users to obtain the summary statistics of data, generate frequency and contingency tables, produce correlations, and conduct statistical inferences. This recipe covers basic statistics that can be applied to a dataset.

Getting ready

Ensure you have completed the previous recipes by installing R on your operating system.

How to do it...

Perform the following steps to apply statistics on a dataset:

1. Load the `iris` data into an R session:

```
> data(iris)
```

2. Observe the format of the data:

```
> class(iris)
[1] "data.frame"
```

3. The iris dataset is a data frame containing four numeric attributes: petal length, petal width, sepal width, and sepal length. For numeric values, you can perform descriptive statistics, such as mean, sd, var, min, max, median, range, and quantile. These can be applied to any of the four attributes in the dataset:

```
> mean(iris$Sepal.Length)
[1] 5.843333
> sd(iris$Sepal.Length)
[1] 0.8280661
> var(iris$Sepal.Length)
[1] 0.6856935
> min(iris$Sepal.Length)
[1] 4.3
> max(iris$Sepal.Length)
[1] 7.9
> median(iris$Sepal.Length)
[1] 5.8
> range(iris$Sepal.Length)
[1] 4.3 7.9
> quantile(iris$Sepal.Length)
 0%   25%   50%   75% 100%
4.3   5.1   5.8   6.4   7.9
```

4. The preceding example demonstrates how to apply descriptive statistics on a single variable. In order to obtain summary statistics on every numeric attribute of the data frame, one may use `sapply`. For example, to apply the mean on the first four attributes in the iris data frame, ignore the `na` value by setting `na.rm` as `TRUE`:

```
> sapply(iris[1:4], mean, na.rm=TRUE)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
      5.843333      3.057333      3.758000      1.199333
```

5. As an alternative to using `sapply` to apply descriptive statistics on given attributes, R offers the `summary` function that provides a full range of descriptive statistics. In the following example, the `summary` function provides the mean, median, 25th and 75th quartiles, min, and max of every iris dataset numeric attribute:

```
> summary(iris)
Sepal.Length      Sepal.Width       Petal.Length      Petal.Width
Species
```

Min. :4.300	Min. :2.000	Min. :1.000	Min. :0.100
setosa :50			
1st Qu.:5.100	1st Qu.:2.800	1st Qu.:1.600	1st Qu.:0.300
versicolor:50			
Median :5.800	Median :3.000	Median :4.350	Median :1.300
virginica :50			
Mean :5.843	Mean :3.057	Mean :3.758	Mean :1.199
3rd Qu.:6.400	3rd Qu.:3.300	3rd Qu.:5.100	3rd Qu.:1.800
Max. :7.900	Max. :4.400	Max. :6.900	Max. :2.500

6. The preceding example shows how to output the descriptive statistics of a single variable. R also provides the correlation for users to investigate the relationship between variables. The following example generates a 4x4 matrix by computing the correlation of each attribute pair within the iris:

```
> cor(iris[,1:4])
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Sepal.Length	1.0000000	-0.1175698	0.8717538	0.8179411
Sepal.Width	-0.1175698	1.0000000	-0.4284401	-0.3661259
Petal.Length	0.8717538	-0.4284401	1.0000000	0.9628654
Petal.Width	0.8179411	-0.3661259	0.9628654	1.0000000

7. R also provides a function to compute the covariance of each attribute pair within the iris:

```
> cov(iris[,1:4])
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Sepal.Length	0.6856935	-0.0424340	1.2743154	0.5162707
Sepal.Width	-0.0424340	0.1899794	-0.3296564	-0.1216394
Petal.Length	1.2743154	-0.3296564	3.1162779	1.2956094
Petal.Width	0.5162707	-0.1216394	1.2956094	0.5810063

8. Statistical tests are performed to access the significance of the results; here we demonstrate how to use a t-test to determine the statistical differences between two samples. In this example, we perform a t.test on the petal width of an iris in either the setosa or versicolor species. If we obtain a p-value less than 0.5, we can be certain that the petal width between the setosa and versicolor will vary significantly:

```
> t.test(iris$Petal.Width[iris$Species=="setosa"],
+         iris$Petal.Width[iris$Species=="versicolor"])
```

Welch Two Sample t-test

```
data: iris$Petal.Width[iris$Species == "setosa"] and iris$Petal.Width[iris$Species == "versicolor"]
t = -34.0803, df = 74.755, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-1.143133 -1.016867
sample estimates:
mean of x mean of y
0.246      1.326
```

9. Alternatively, you can perform a correlation test on the sepal length to the sepal width of an iris, and then retrieve a correlation score between the two variables. The stronger the positive correlation, the closer the value is to 1. The stronger the negative correlation, the closer the value is to -1:

```
> cor.test(iris$Sepal.Length, iris$Sepal.Width)
```

Pearson's product-moment correlation

```
data: iris$Sepal.Length and iris$Sepal.Width
t = -1.4403, df = 148, p-value = 0.1519
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
-0.27269325  0.04351158
sample estimates:
cor
-0.1175698
```

How it works...

R has a built-in statistics function, which enables the user to perform descriptive statistics on a single variable. The recipe first introduces how to apply `mean`, `sd`, `var`, `min`, `max`, `median`, `range`, and `quantile` on a single variable. Moreover, in order to apply the statistics on all four numeric variables, one can use the `sapply` function. In order to determine the relationships between multiple variables, one can conduct correlation and covariance. Finally, the recipe shows how to determine the statistical differences of two given samples by performing a statistical test.

There's more...

If you need to compute an aggregated summary statistics against data in different groups, you can use the aggregate and reshape functions to compute the summary statistics of data subsets:

1. Use aggregate to calculate the mean of each iris attribute group by the species:

```
> aggregate(x=iris[,1:4],by=list(iris$Species),FUN=mean)
```

2. Use reshape to calculate the mean of each iris attribute group by the species:

```
> library(reshape)
> iris.melt <- melt(iris,id='Species')
> cast(Species~variable,data=iris.melt,mean,
       subset=Species %in% c('setosa','versicolor'),
       margins='grand_row')
```

For information on reshape and aggregate, refer to the help documents by using `?reshape` or `?aggregate`.

Visualizing data

Visualization is a powerful way to communicate information through graphical means. Visual presentations make data easier to comprehend. This recipe presents some basic functions to plot charts, and demonstrates how visualizations are helpful in data exploration.

Getting ready

Ensure that you have completed the previous recipes by installing R on your operating system.

How to do it...

Perform the following steps to visualize a dataset:

1. Load the iris data into the R session:

```
> data(iris)
```

2. Calculate the frequency of species within the iris using the `table` command:

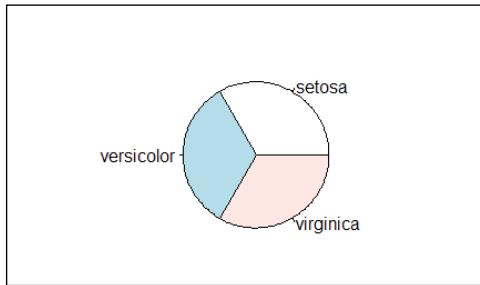
```
> table.iris = table(iris$Species)
> table.iris
```

setosa	versicolor	virginica
--------	------------	-----------

50	50	50
----	----	----

3. As the frequency in the table shows, each species represents 1/3 of the iris data. We can draw a simple pie chart to represent the distribution of species within the iris:

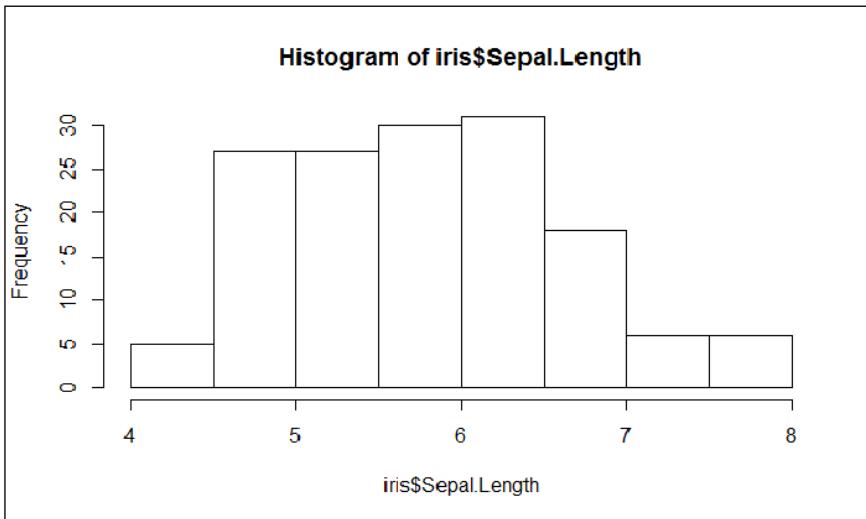
```
> pie(table.iris)
```



The pie chart of species distribution

4. The histogram creates a frequency plot of sorts along the x-axis. The following example produces a histogram of the sepal length:

```
> hist(iris$Sepal.Length)
```

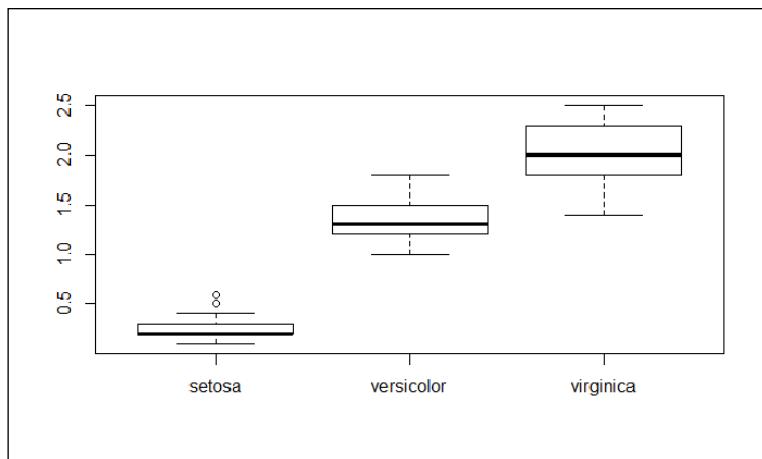


The histogram of the sepal length

5. In the histogram, the x-axis presents the sepal length and the y-axis presents the count for different sepal lengths. The histogram shows that for most irises, sepal lengths range from 4 cm to 8 cm.

6. Boxplots, also named box and whisker graphs, allow you to convey a lot of information in one simple plot. In such a graph, the line represents the median of the sample. The box itself shows the upper and lower quartiles. The whiskers show the range:

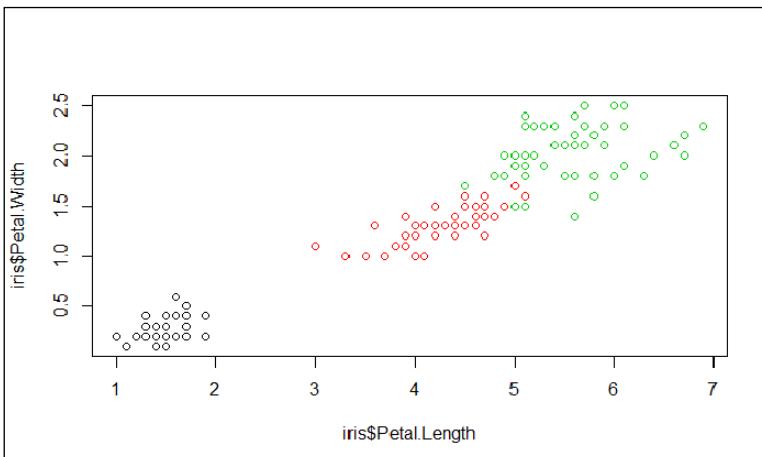
```
> boxplot(Petal.Width ~ Species, data = iris)
```



The boxplot of the petal width

7. The preceding screenshot clearly shows the median and upper range of the petal width of the setosa is much shorter than versicolor and virginica. Therefore, the petal width can be used as a substantial attribute to distinguish iris species.
8. A scatter plot is used when there are two variables to plot against one another. This example plots the petal length against the petal width and color dots in accordance to the species it belongs to:

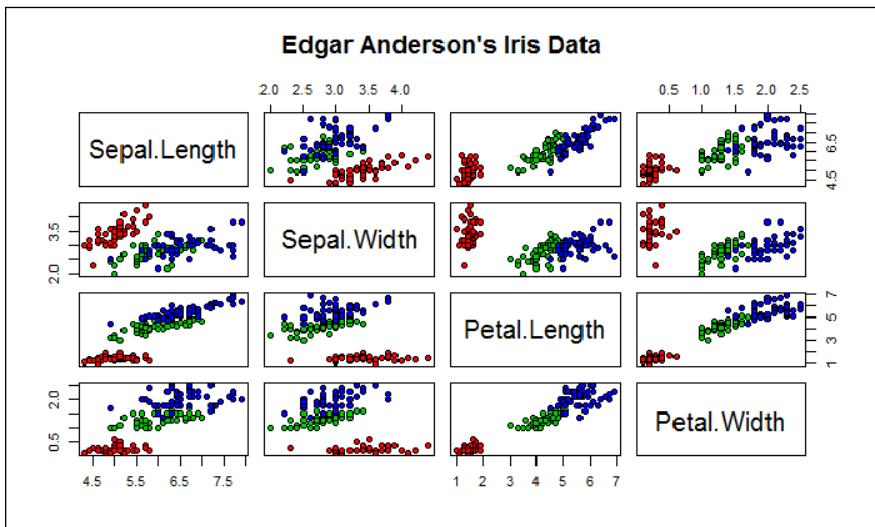
```
> plot(x=iris$Petal.Length, y=iris$Petal.Width, col=iris$Species)
```



The scatter plot of the sepal length

9. The preceding screenshot is a scatter plot of the petal length against the petal width. As there are four attributes within the iris dataset, it takes six operations to plot all combinations. However, R provides a function named `pairs`, which can generate each subplot in one figure:

```
> pairs(iris[1:4], main = "Edgar Anderson's Iris Data", pch = 21,  
bg = c("red", "green3", "blue") [unclass(iris$Species)])
```



Pairs scatterplot of iris data

How it works...

R provides many built-in plot functions, which enable users to visualize data with different kinds of plots. This recipe demonstrates the use of pie charts that can present category distribution. A pie chart of an equal size shows that the number of each species is equal. A histogram plots the frequency of different sepal lengths. A box plot can convey a great deal of descriptive statistics, and shows that the petal width can be used to distinguish an iris species. Lastly, we introduced scatter plots, which plot variables on a single plot. In order to quickly generate a scatter plot containing all the pairs of iris data, one may use the `pairs` command.

See also

- ▶ `ggplot2` is another plotting system for R, based on the implementation of Leland Wilkinson's grammar of graphics. It allows users to add, remove, or alter components in a plot with a higher abstraction. However, the level of abstraction results in slow compared to lattice graphics. For those of you interested in the topic of ggplot, you can refer to this site: <http://ggplot2.org/>.

Getting a dataset for machine learning

While R has a built-in dataset, the sample size and field of application is limited. Apart from generating data within a simulation, another approach is to obtain data from external data repositories. A famous data repository is the UCI machine learning repository, which contains both artificial and real datasets. This recipe introduces how to get a sample dataset from the UCI machine learning repository.

Getting ready

Ensure that you have completed the previous recipes by installing R on your operating system.

How to do it...

Perform the following steps to retrieve data for machine learning:

1. Access the UCI machine learning repository: <http://archive.ics.uci.edu/ml/>.

The screenshot shows the homepage of the UC Irvine Machine Learning Repository. At the top, there's a navigation bar with links for About, Citation Policy, Donate a Data Set, and Contact. Below the navigation is a search bar with a 'Search' button and a link to 'View All Data Sets'. The main header features the 'UCI' logo and a stylized triceratops illustration. The page title is 'Machine Learning Repository' with the subtitle 'Center for Machine Learning and Intelligent Systems'. On the left, there's a 'Latest News' section with a list of recent updates. In the center, there's a 'Newest Data Sets' table showing recent additions like 'NoiseOffice' and 'hEMG for Basic Hand movements'. On the right, there's a 'Most Popular Data Sets' table showing highly used datasets such as 'Iris', 'Adult', 'Wine', 'Car Evaluation', and 'Breast Cancer Wisconsin (Diagnostic)'. The footer contains logos for 'Supported By' and 'In Collaboration With'.

Latest News:		Newest Data Sets:	Most Popular Data Sets (hits since 2007):
2013-04-04: Welcome to the new Repository admins Kevin Bach and Moshe Lichman!		2015-01-03: NoiseOffice	668754: Iris
2010-03-01: Note from donor regarding Netflix data		2014-11-18: hEMG for Basic Hand movements	469530: Adult
2009-10-16: Two new data sets have been added		2014-11-05: Sentence Classification	395279: Wine
2009-09-14: Several data sets have been added		2014-10-23: Dow Jones Index	328162: Car Evaluation
2008-07-23: Repository mirror has been set up		2014-10-18: Geographical Origin of Music	318483: Breast Cancer Wisconsin (Diagnostic)
2008-03-24: New data sets have been added!			
2007-06-25: Two new data sets have been added: UJI Pen Characters, MAGIC Gamma Telescope			
Featured Data Set: Australian.Sign.Language.signs			

UCI data repository

2. Click on **View ALL Data Sets**. Here you will find a list of datasets containing field names, such as **Name**, **Data Types**, **Default Task**, **Attribute Types**, **# Instances**, **# Attributes**, and **Year**:

UCI Machine Learning Repository Center for Machine Learning and Intelligent Systems

About Citation Policy Donate a Data Set Contact Search Repository Web Google+ View ALL Data Sets

Browse Through: 308 Data Sets

	Name	Data Types	Default Task	Attribute Types	# Instances	# Attributes	Year
Default Task							
Classification (221)							
Regression (46)							
Clustering (39)							
Other (50)							
Attribute Type							
Categorical (36)							
Numerical (170)							
Mixed (56)							
Data Type							
Multivariate (235)							
Univariate (15)							
Sequential (27)							
Time-Series (46)							
Text (28)							
Domain-Theory (20)							
Other (21)							
Area							
Life Sciences (76)							
Physical Sciences (41)							
CS & Engineering (64)							
Social Sciences (20)							
Business (16)							
Game (9)							
Other (61)							

© 1998-2018 archive.uci.edu

3. Use **Ctrl + F** to search for **Iris**:

1 of 1 page 1 of 1 item

 ICU	Multivariate, Time-Series						
 Image Segmentation	Multivariate	Classification	Real	2310	19	1990	
 Internet Advertisements	Multivariate	Classification	Categorical, Integer, Real	3279	1558	1998	
 Ionosphere	Multivariate	Classification	Integer, Real	351	34	1989	
 ISOLET	Multivariate	Classification	Real	150	4	1998	
 Kinship	Relational	Relational-Learning	Categorical	104	12	1990	
 Labor Relations	Multivariate		Categorical, Integer, Real	57	16	1988	
 LED Display Domain	Multivariate, Data-Generator	Classification	Categorical		7	1988	

4. Click on **Iris**. This will display the data folder and the dataset description:

UCI 

Machine Learning Repository
Center for Machine Learning and Intelligent Systems

[About](#) [Citation Policy](#) [Donate a Data Set](#) [Contact](#)

[Repository](#) [Web](#) [Google+](#)

[Search](#)

[View ALL Data Sets](#)

Iris Data Set

[Download](#) [Data Folder](#) [Data Set Description](#)

Abstract Famous database, from Fisher, 1936



Data Set Characteristics:	Multivariate	Number of Instances:	150	Area:	Life
Attribute Characteristics:	Real	Number of Attributes:	4	Date Donated	1988-07-01
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	659046

Source:

Creator:
R.A. Fisher

5. Click on **Data Folder**, which will display a directory containing the iris dataset:

Index of /ml/machine-learning-databases/iris

Name	Last modified	Size	Description
 Parent Directory		-	
 Index	03-Dec-1996 04:01	105	
 bezdekIris.data	14-Dec-1999 12:12	4.4K	
 iris.data	08-Mar-1993 16:27	4.4K	
 iris.names	11-Jul-2000 21:30	2.9K	

Apache/2.2.15 (CentOS) Server at archive.ics.uci.edu Port 80

6. You can then either download `iris.data` or use the `read.csv` function to read the dataset:

```
> iris.data = read.csv(url("http://archive.ics.uci.edu/ml/machine-
learning-databases/iris/iris.data"), header = FALSE, col.names =
  c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width",
  "Species"))

> head(iris.data)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
5	5.0	3.6	1.4	0.2	Iris-setosa
6	5.4	3.9	1.7	0.4	Iris-setosa

How it works...

Before conducting data analysis, it is important to collect your dataset. However, to collect an appropriate dataset for further exploration and analysis is not easy. We can, therefore, use the prepared dataset with the UCI repository as our data source. Here, we first access the UCI dataset repository and then use the iris dataset as an example. We can find the iris dataset by using the browser's find function (*Ctrl + F*), and then enter the file directory. Last, we can download the dataset and use the R IO function, `read.csv`, to load the iris dataset into an R session.

See also

- ▶ KDnuggets (<http://www.kdnuggets.com/datasets/index.html>) offers a resourceful list of datasets for data mining and data science. You can explore the list to find the data that satisfies your requirements.

3

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
```

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.explorededata.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(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 = ".")
```

4

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

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



If we select a single variable, then subsetting returns a vector instead of a data frame.

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 qs" "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...

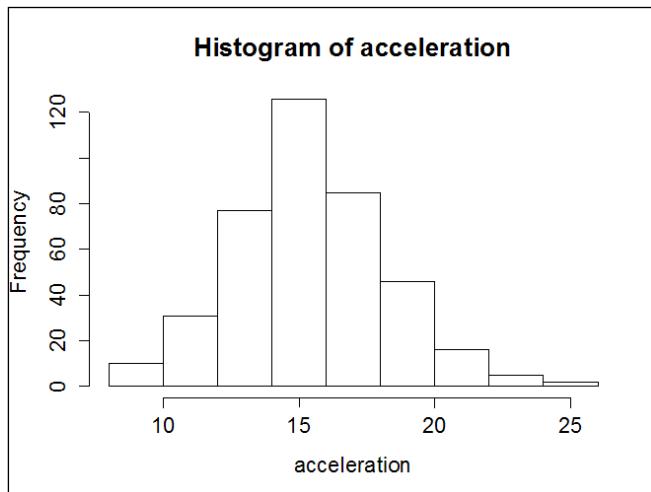
In this recipe, we cover histograms, boxplots, scatterplots and scatterplot matrices.

Histograms

Generate a histogram for acceleration:

```
> hist(acceleration)
```

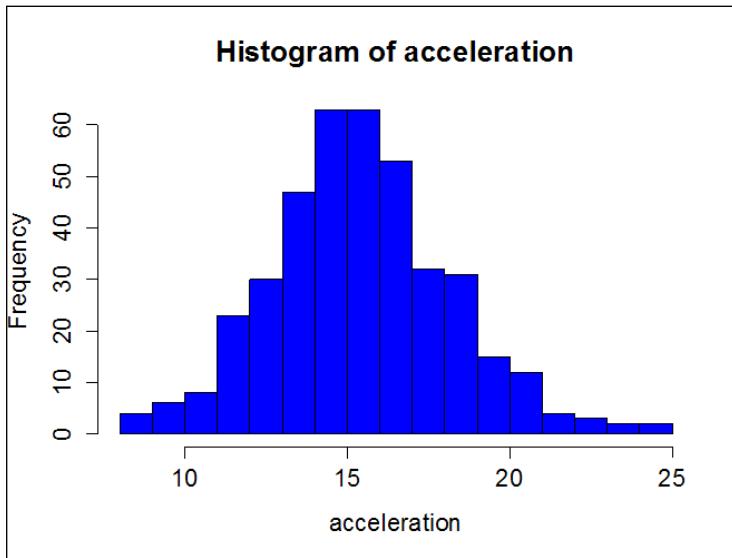
R determines the various properties of the generated graph (like bin sizes, axes scales, axes titles, chart title, bar colors,...) automatically. The following diagram shows the output of the preceding command:



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

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

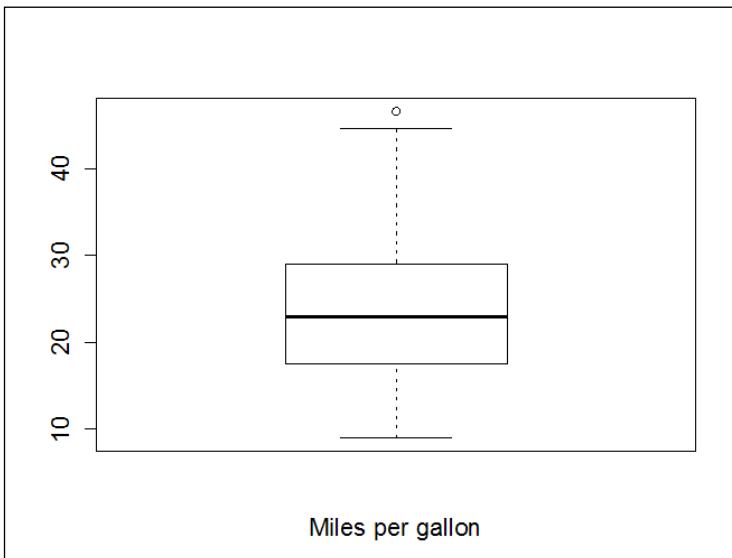
The histogram of acceleration can be seen in the following diagram:



Boxplots

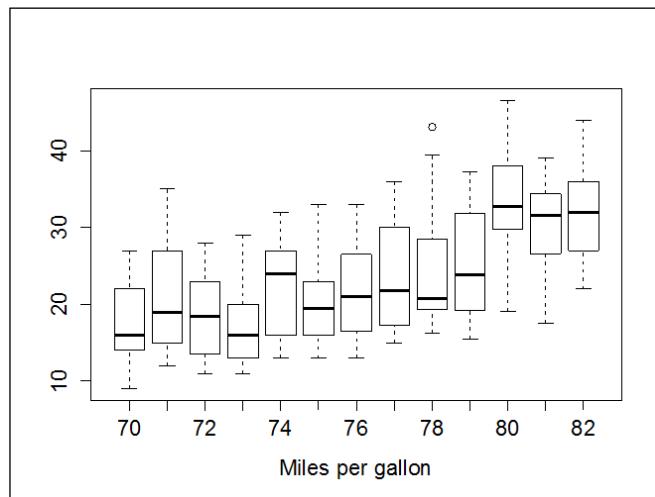
Create a boxplot for mpg using the following command:

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



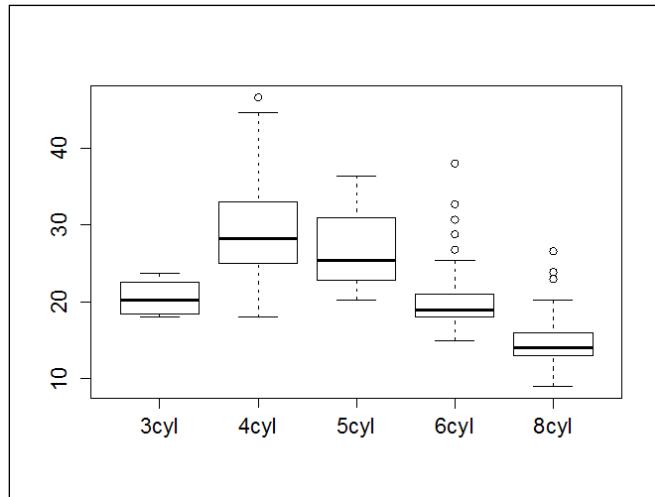
To generate boxplots for subsets within the whole dataset, you can use:

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



Create a boxplot of mpg by cylinders:

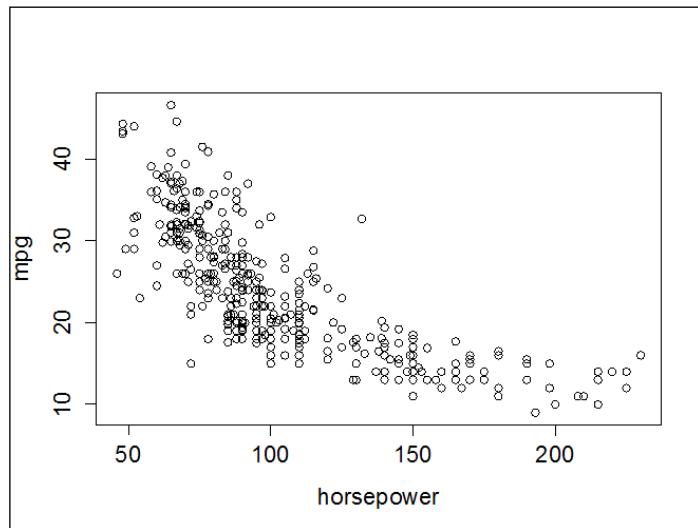
```
> boxplot(mpg ~ cylinders)
```



Scatterplots

Create a scatterplot for mpg by horsepower:

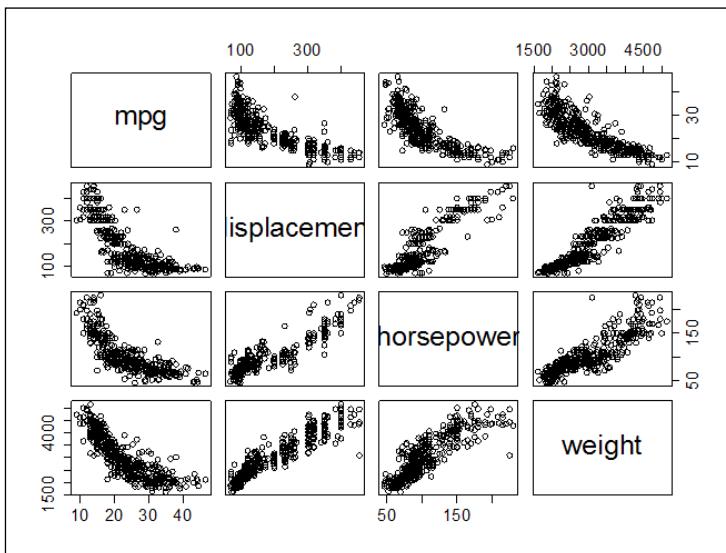
```
> plot(mpg ~ horsepower)
```



Scatterplot matrices

Create pair wise scatterplots for a set of variables:

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



How it works...

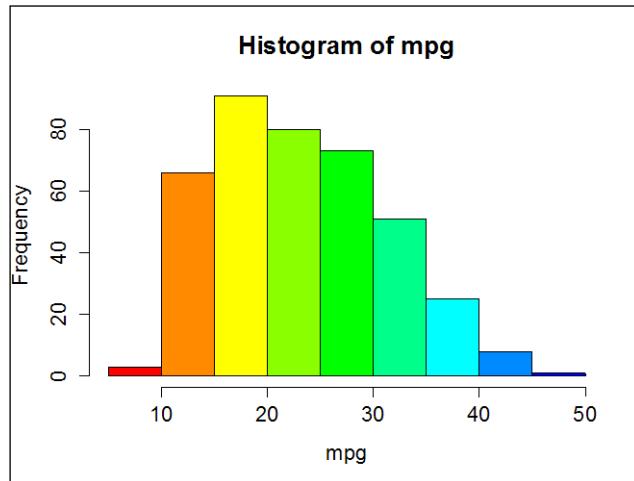
Here, we describe how the preceding lines of code work.

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...

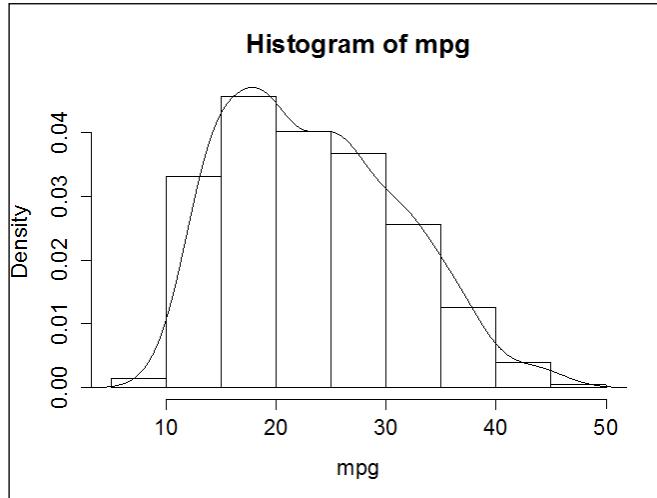
You can overlay plots and color specific points differently. We show some useful options in this section.

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:

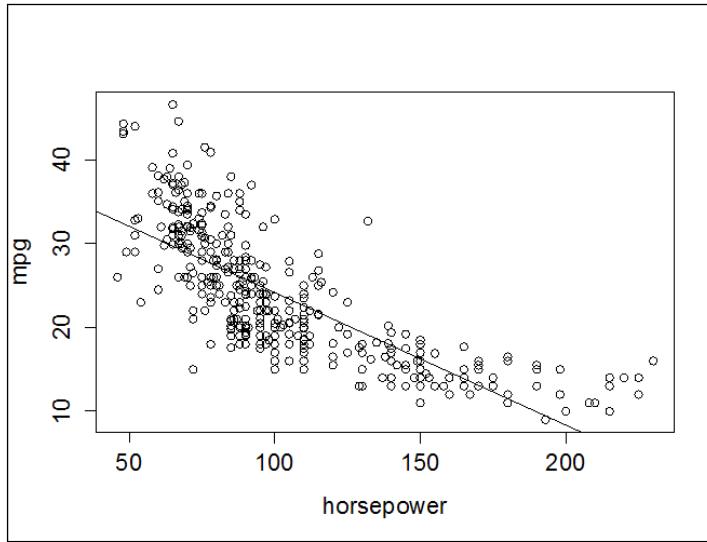
```
hist(mpg, prob=TRUE)
lines(density(mpg))
```



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)
```



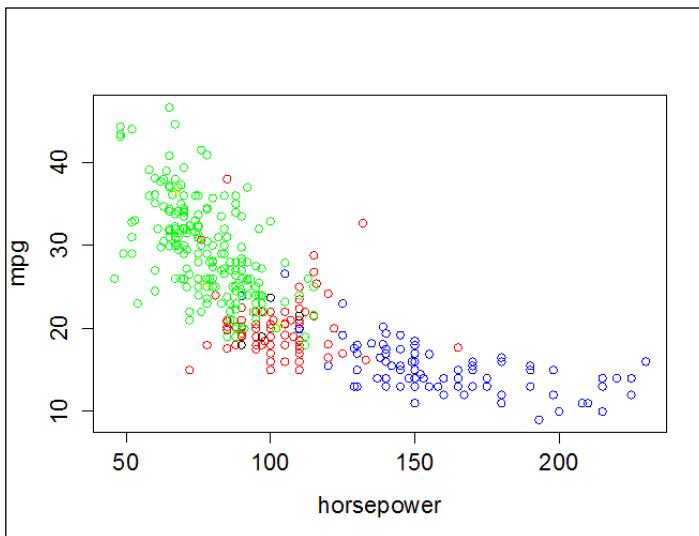
Color specific points on a scatterplot

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

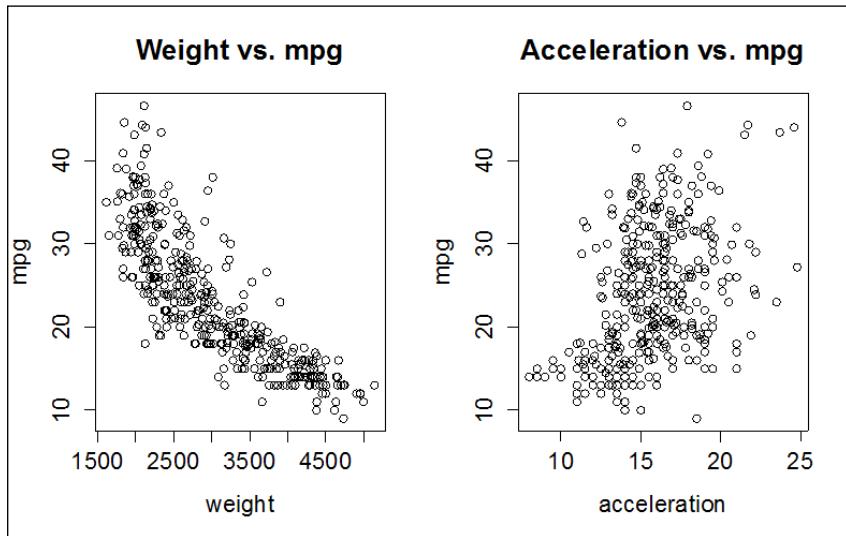
If you have not already done so, download the data files for this chapter and ensure that they are available in your R environment's working directory. Once this is done, run the following commands:

```
> 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:

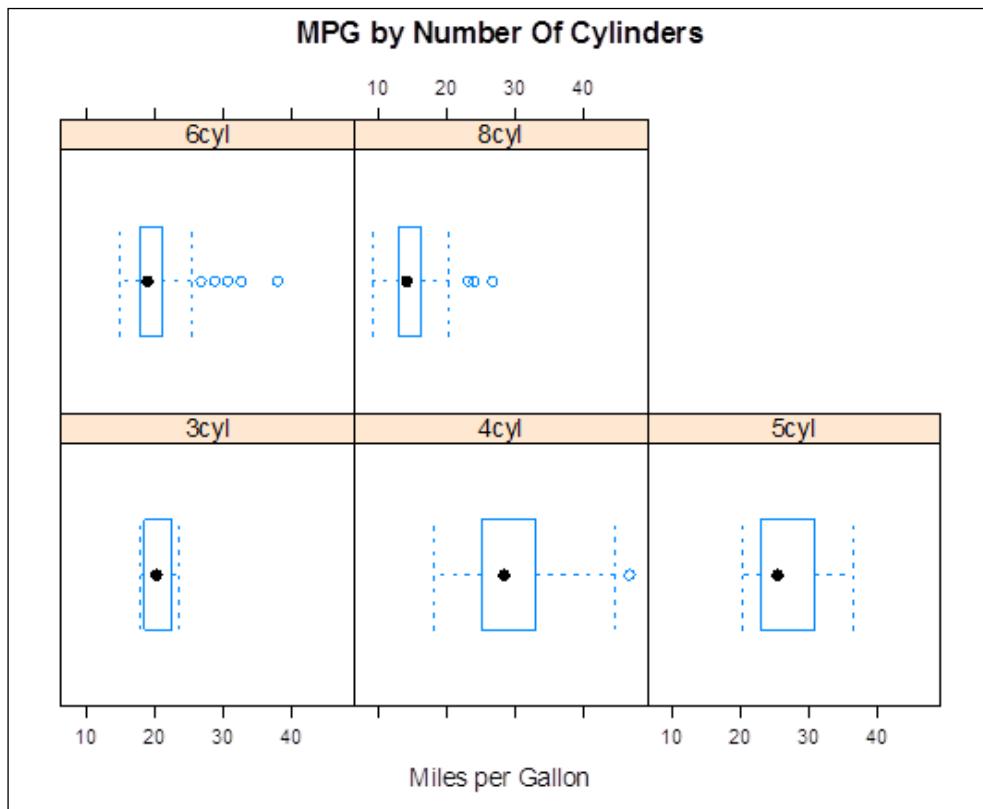
```
> xyplot(mpg~weight|cyl.factor, data=auto,
  main="Weight Vs MPG by Number of Cylinders",
  ylab="Miles per Gallon", xlab="Car Weight")
```

How it works...

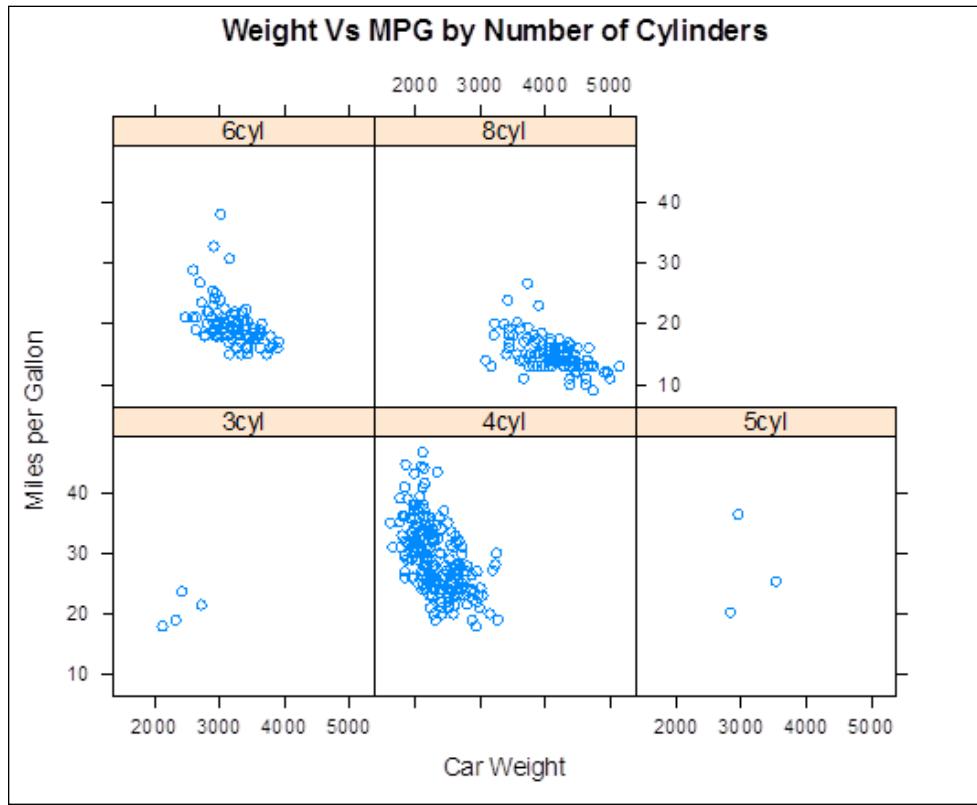
Lattice plot commands comprise the following four parts:

- ▶ **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`.factor 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...

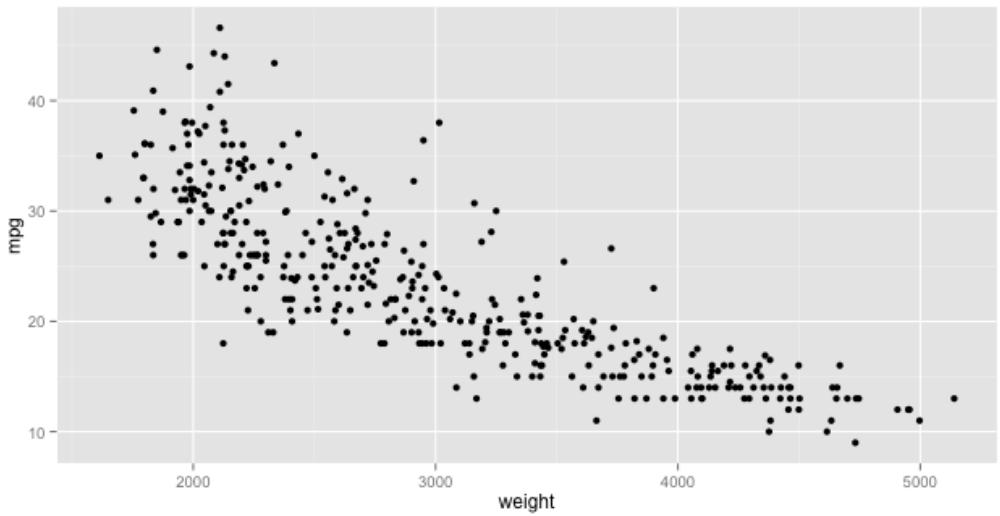
Let's start from the top and discuss some variations:

```
> 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:

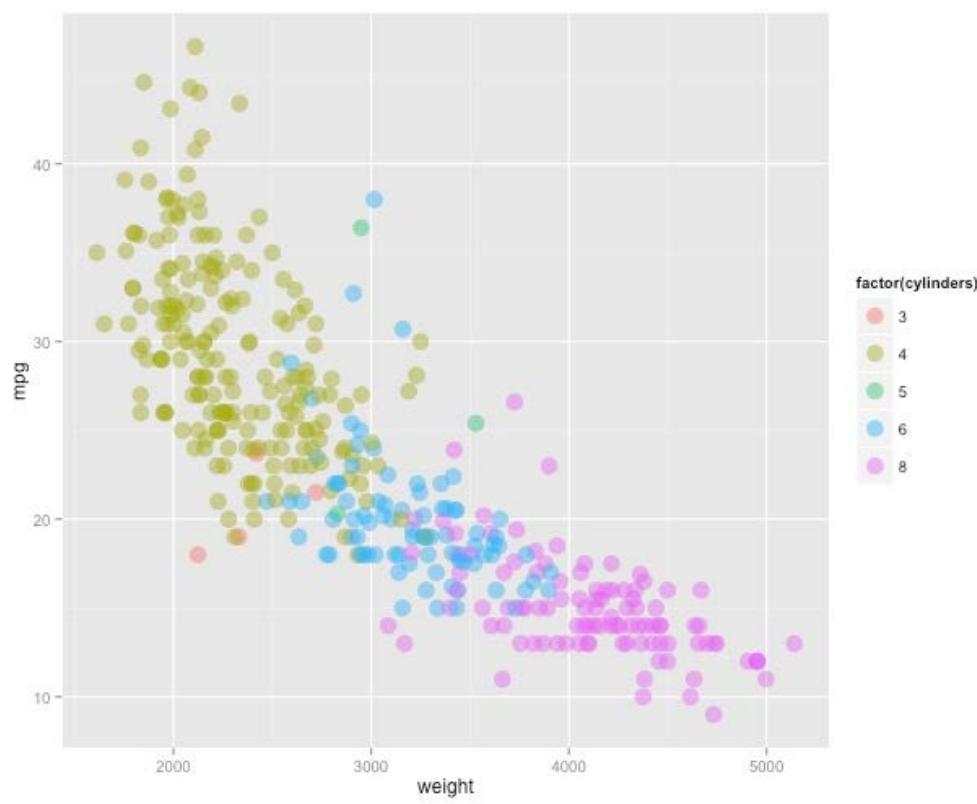
```
> plot + geom_point()
```

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:

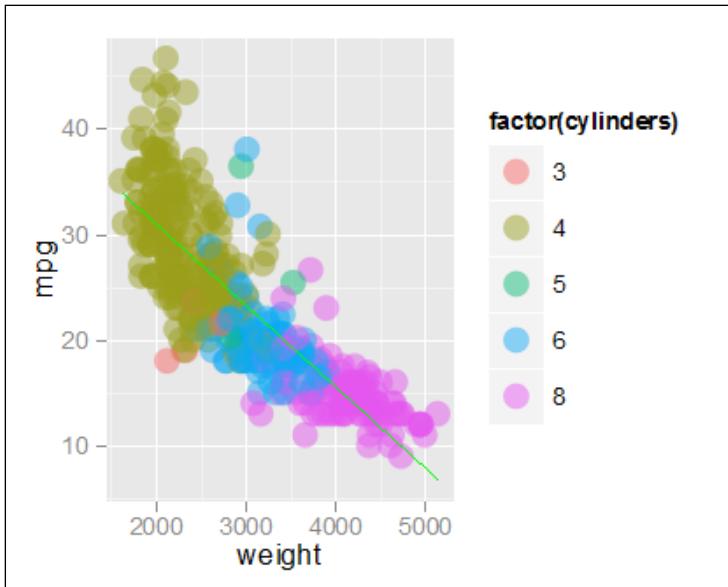
```
> plot + geom_point(alpha=1/2, size=5,  
aes(color=factor(cylinders)))
```



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:

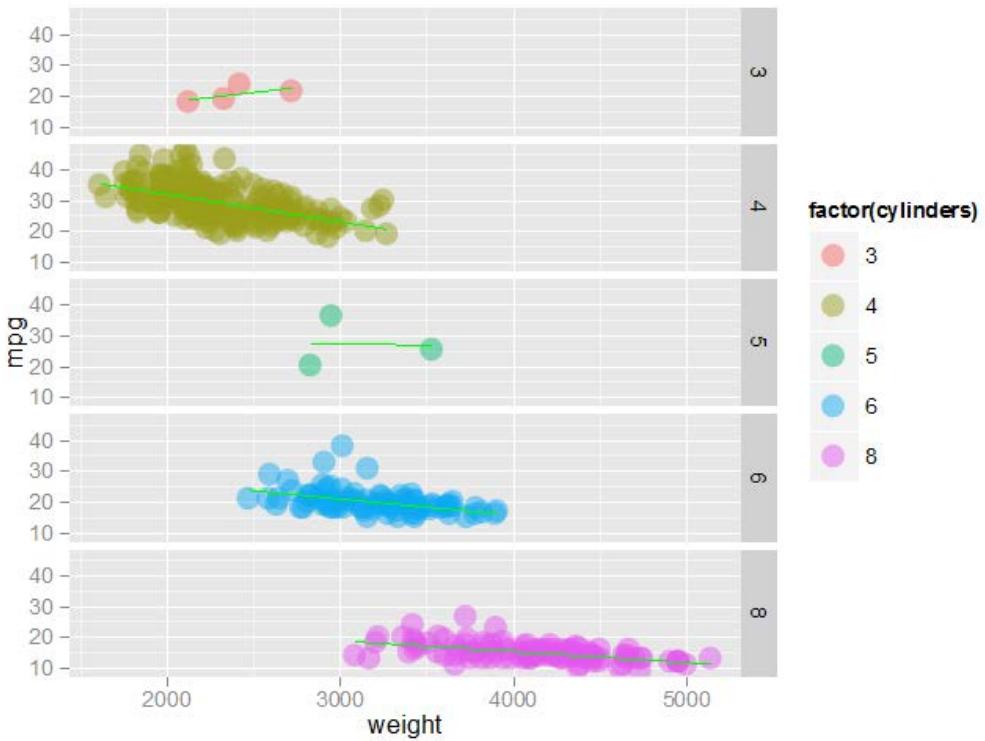


By default, the `geom_smooth` function uses two different smoothing approaches based on the number of observations. If the number of observations exceeds 1000, it uses `gam` smoothing, `loess` otherwise. Given the familiarity with linear models, people mostly use the `lm` smoothing.

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:

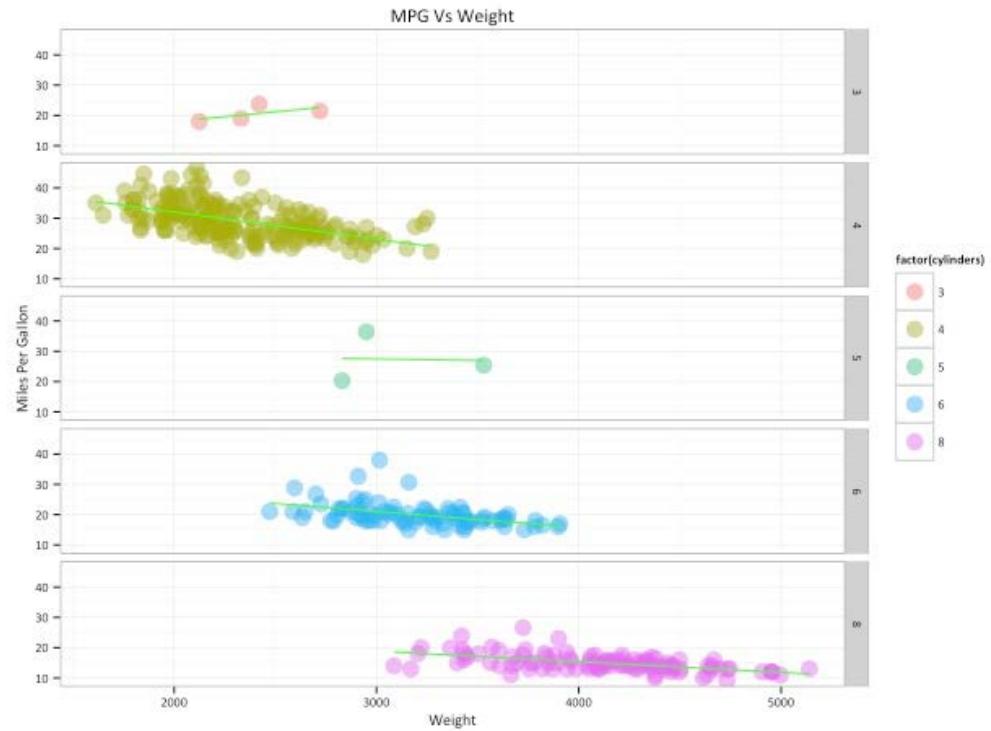


If we change to `facet_grid(~.cylinders)`, the plots for each level of the cylinder are arranged horizontally.

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))
```

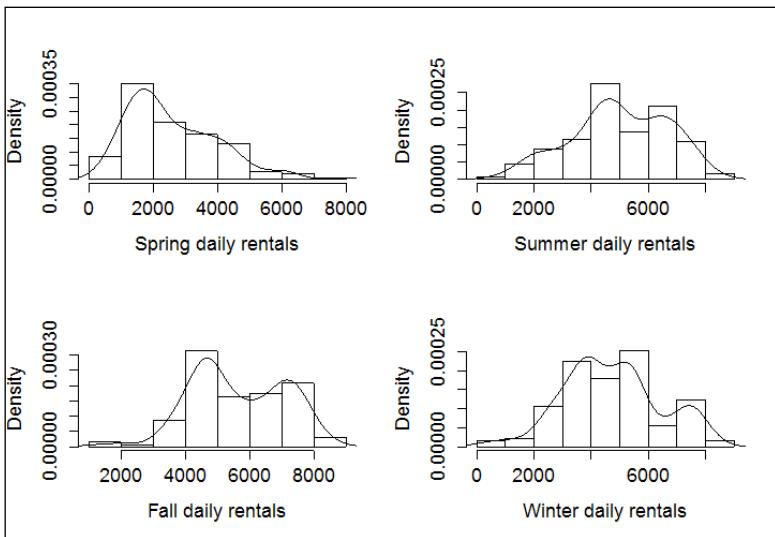
2. Extract data for the seasons:

```
> 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))
```

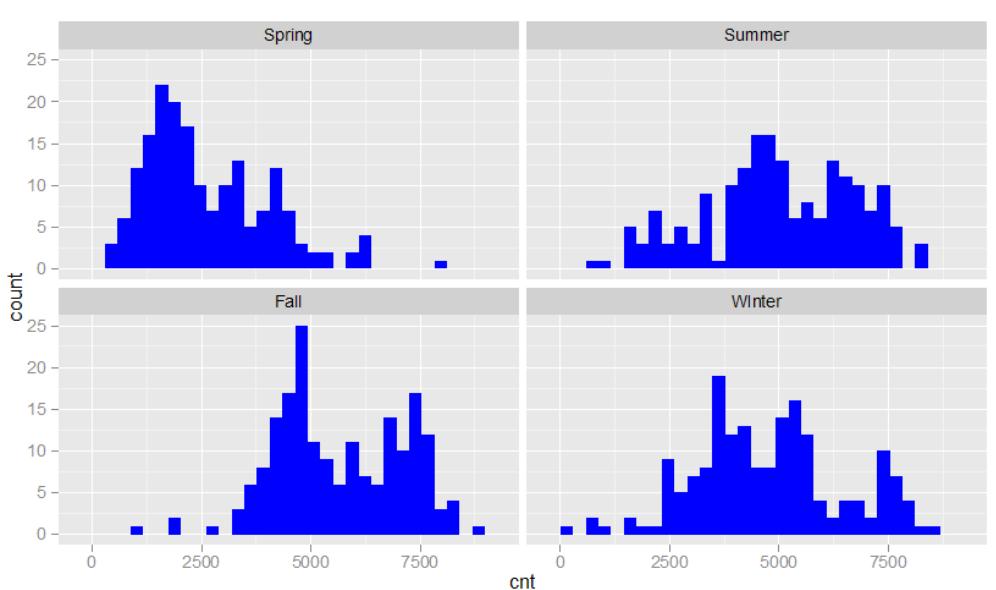
You get the following output that facilitates comparisons across the seasons:



Using ggplot2

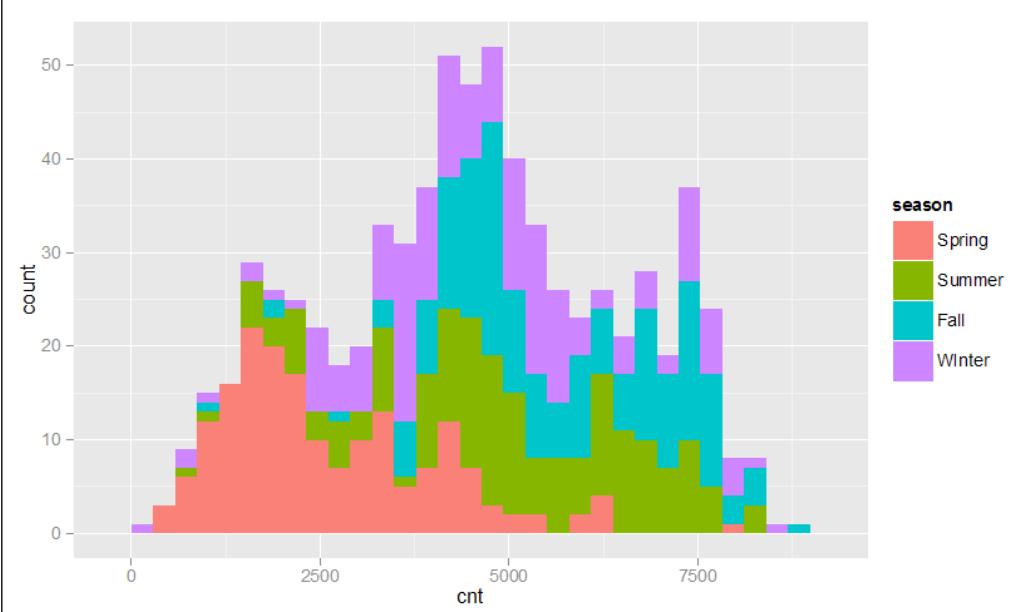
We can achieve much of the preceding results in a single command:

```
> 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...

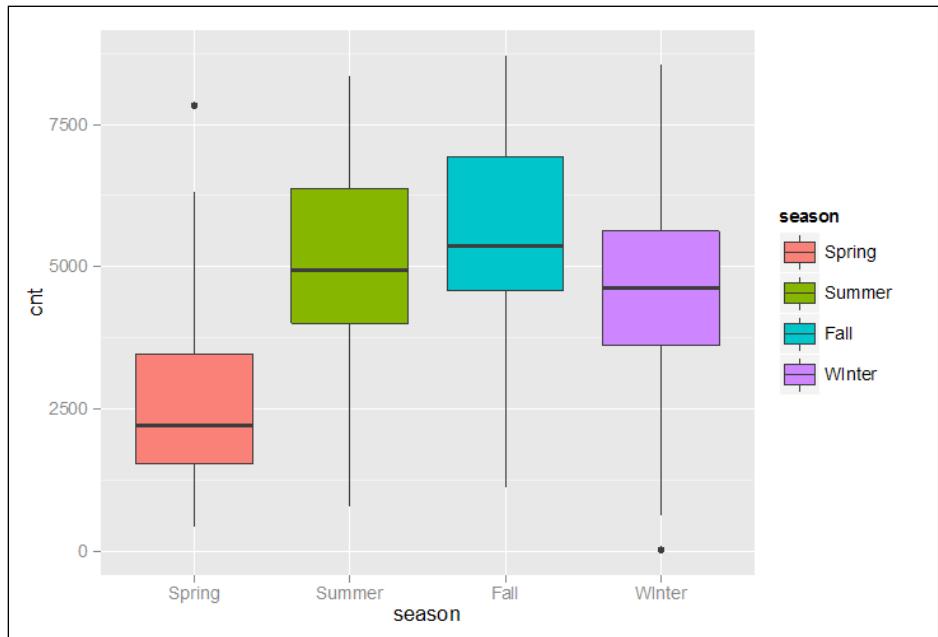
You can use `ggplot2` to generate comparative boxplots as well.

Creating boxplots with ggplot2

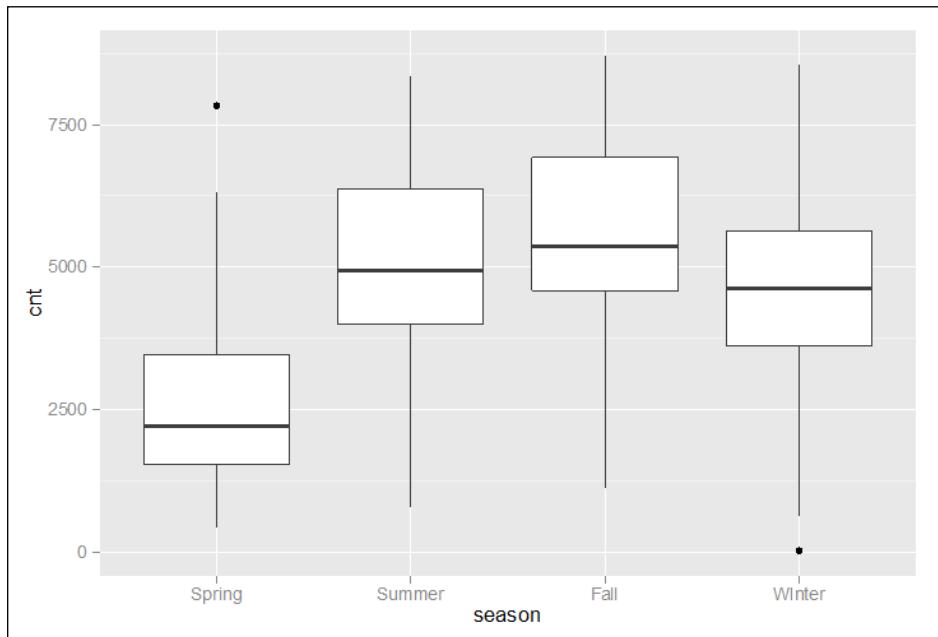
Instead of the default histogram, you can get a boxplot with either of the following two approaches:

```
> 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:

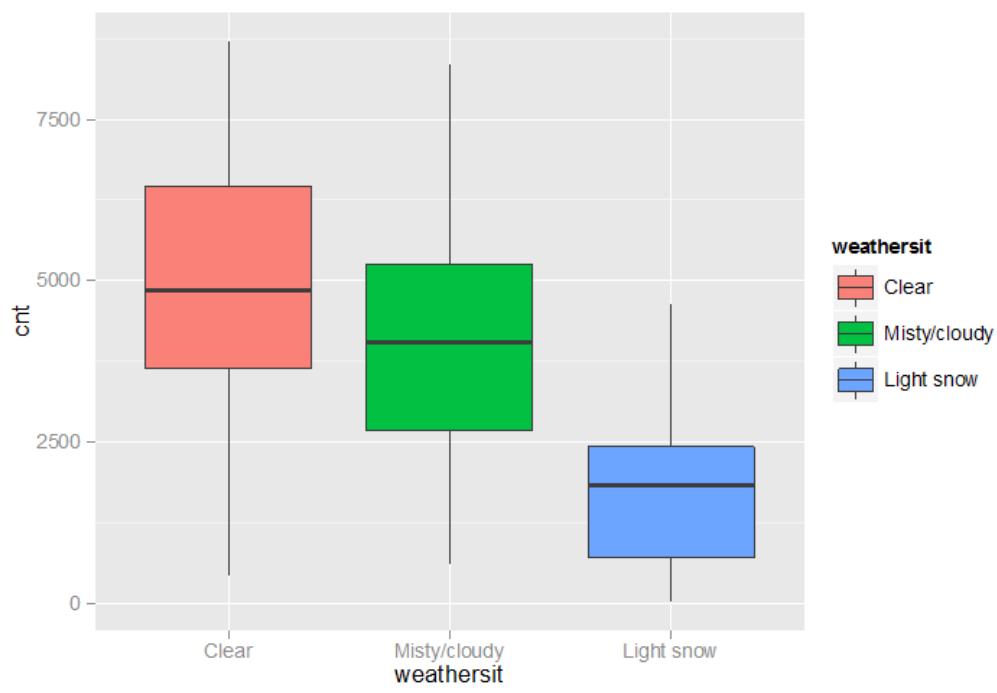
```
> bike <- read.csv("daily-bike-rentals.csv")
> bike$season <- factor(bike$season, levels = c(1,2,3,4),
  labels = c("Spring", "Summer", "Fall", "Winter"))
> bike$weathersit <- factor(bike$weathersit, levels = c(1,2,3),
  labels = c("Clear", "Misty/cloudy", "Light snow"))
> attach(bike)
```

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

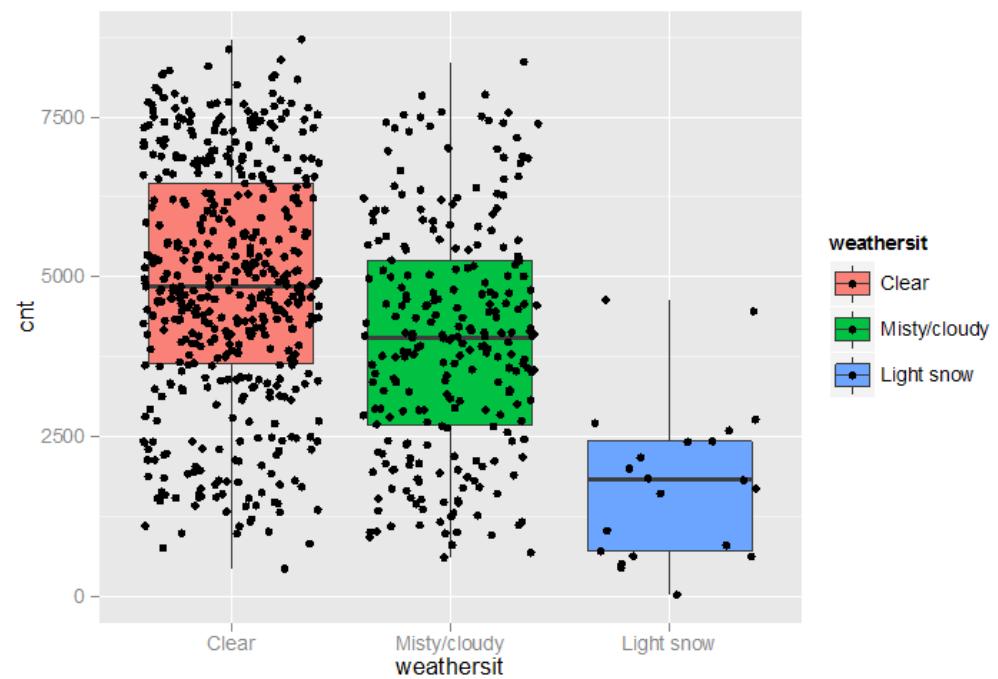
The preceding command produces the following output:



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...

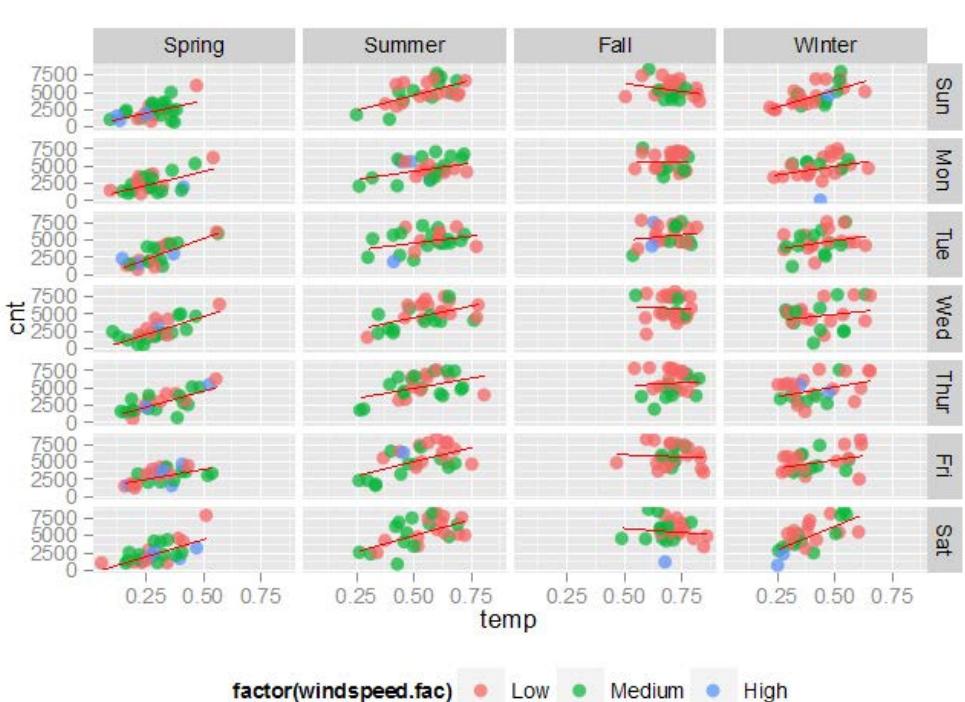
Create a multivariate plot using the following commands:

```

> 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

5

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		
		Low	Medium	High
Actual	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		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		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...

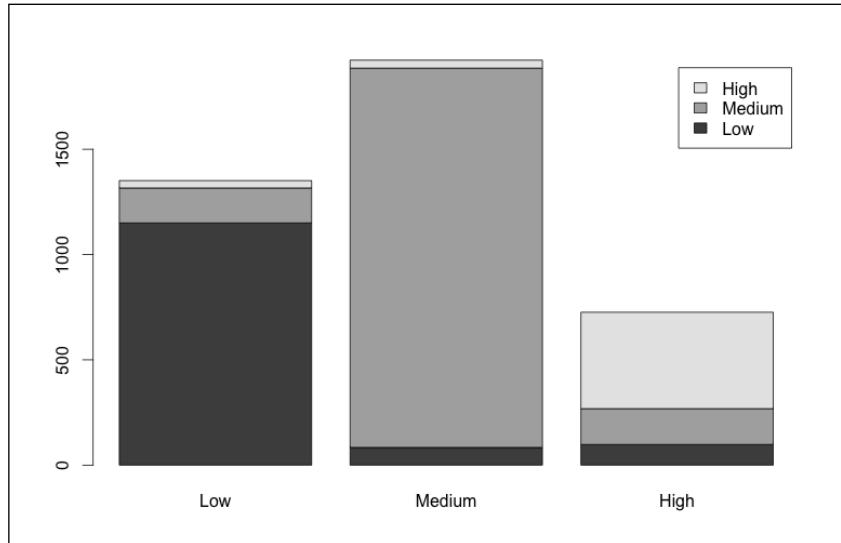
When dealing with more than two or three categories, seeing the error matrix as a chart could be useful to quickly assess the model's performance within various categories.

Visualizing the error/classification confusion matrix

You can create a barplot using the following command :

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

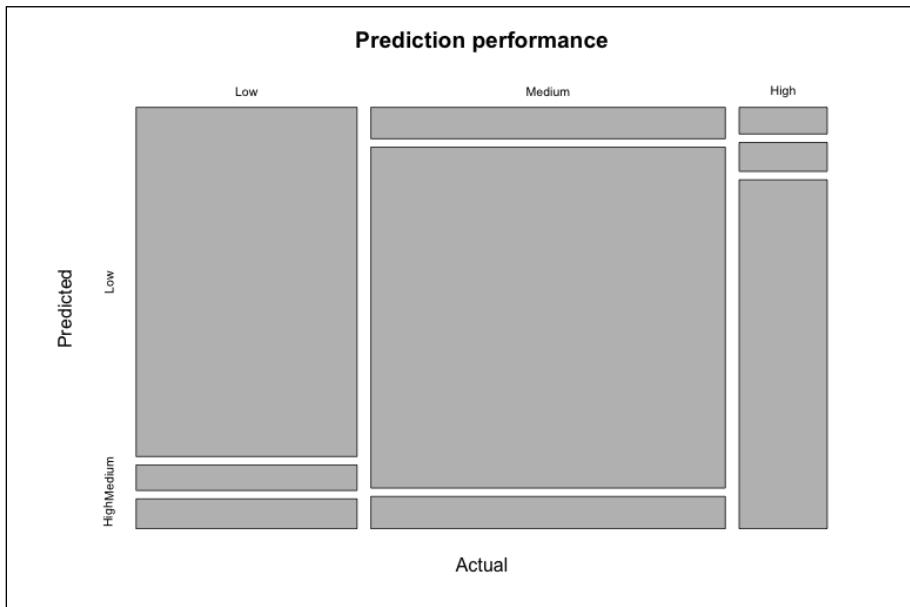
The following output is the result of the preceding command:



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 `rocr-example-1.csv` and `rocr-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)
```

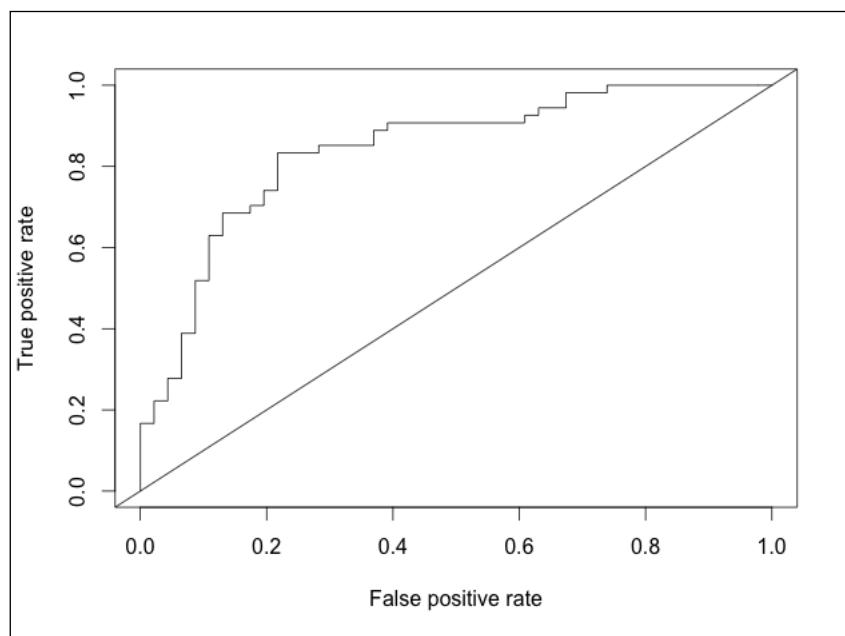
4. Create the performance object:

```
> 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 0.055555556
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

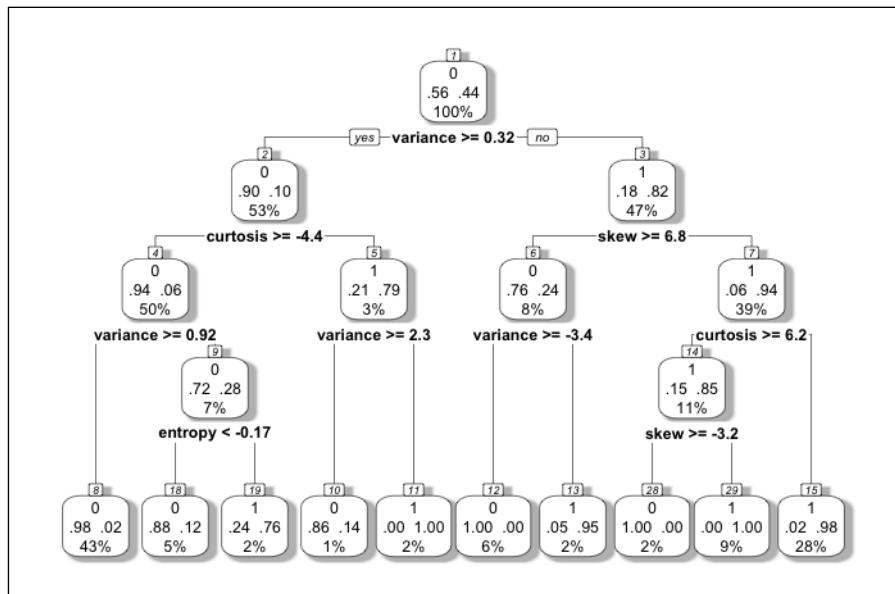
node), split, n, loss, yval, (yprob)
  * denotes terminal node

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:

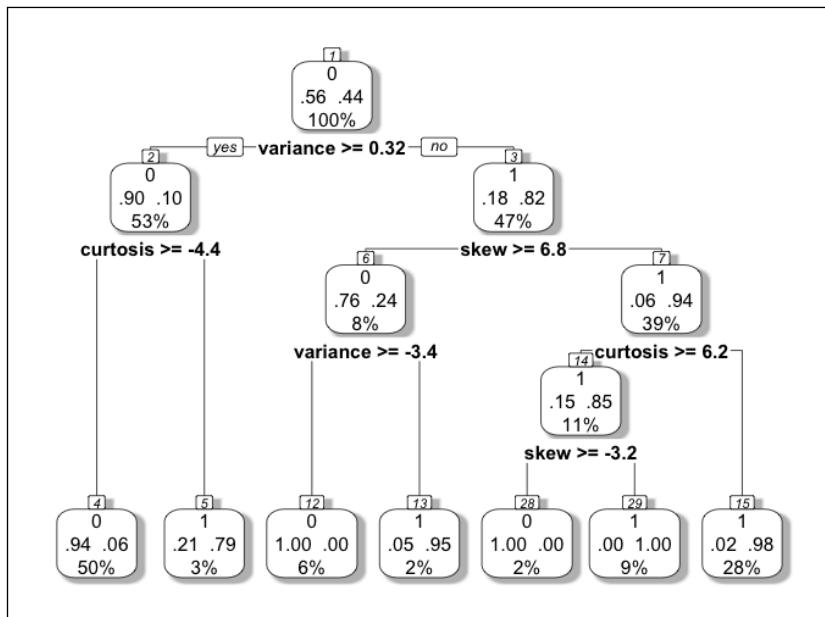
```
> # First see the cptable  
> # !!Note!!: Your table can be different because of the  
> # random aspect in cross-validation  
> mod$cptable
```

	CP	nsplit	rel error	xerror	xstd
1	0.69030733	0	1.00000000	1.00000000	0.03637971
2	0.09456265	1	0.30969267	0.3262411	0.02570025
3	0.04018913	2	0.21513002	0.2387707	0.02247542
4	0.01891253	4	0.13475177	0.1607565	0.01879222
5	0.01182033	6	0.09692671	0.1347518	0.01731090
6	0.01063830	7	0.08510638	0.1323877	0.01716786
7	0.01000000	9	0.06382979	0.1276596	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, *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"`:

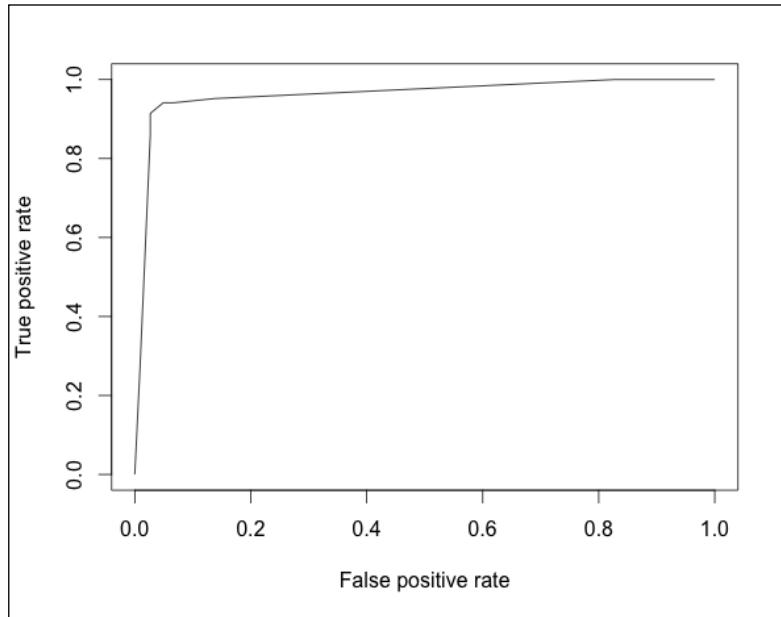
```
> pred.pruned <- predict(mod, bn[-train.idx,], type = "prob")
```

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, *What's in There? – Exploratory Data Analysis*
- ▶ *Generating error/classification-confusion matrices* in this chapter.
- ▶ *Building regression trees* in Chapter, *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:

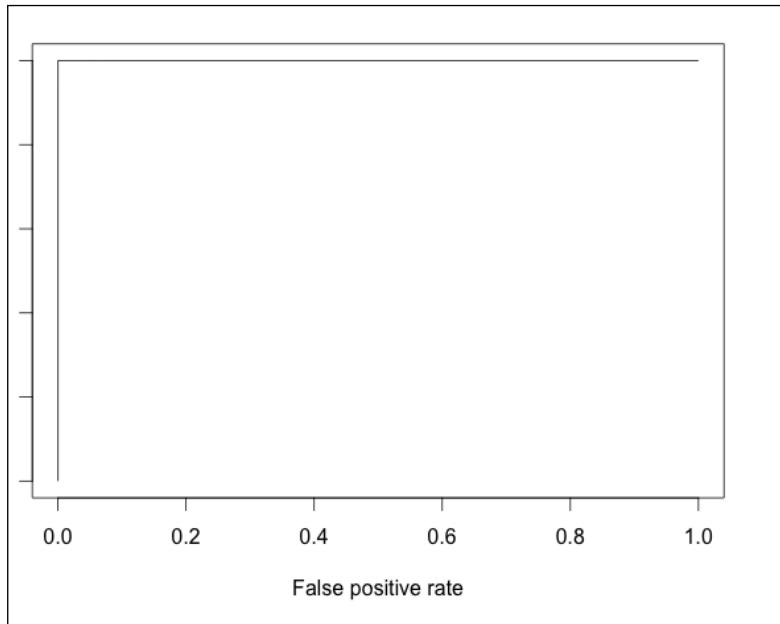
```
> probs <- predict(mod, bn[-sub.idx,], type = "prob")
```

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, *What's in There? – Exploratory Data Analysis*
- ▶ *Generating error/classification-confusion matrices* in this chapter
- ▶ *Building Random Forest models for regression* in Chapter, *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

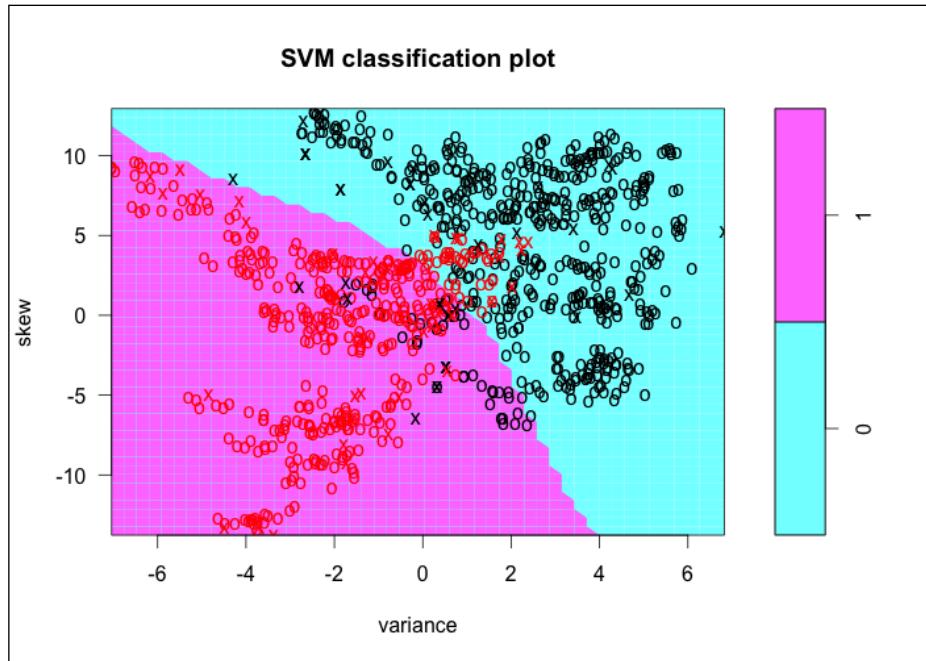
7. Check model performance on the validation partition:

```
> 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)
```

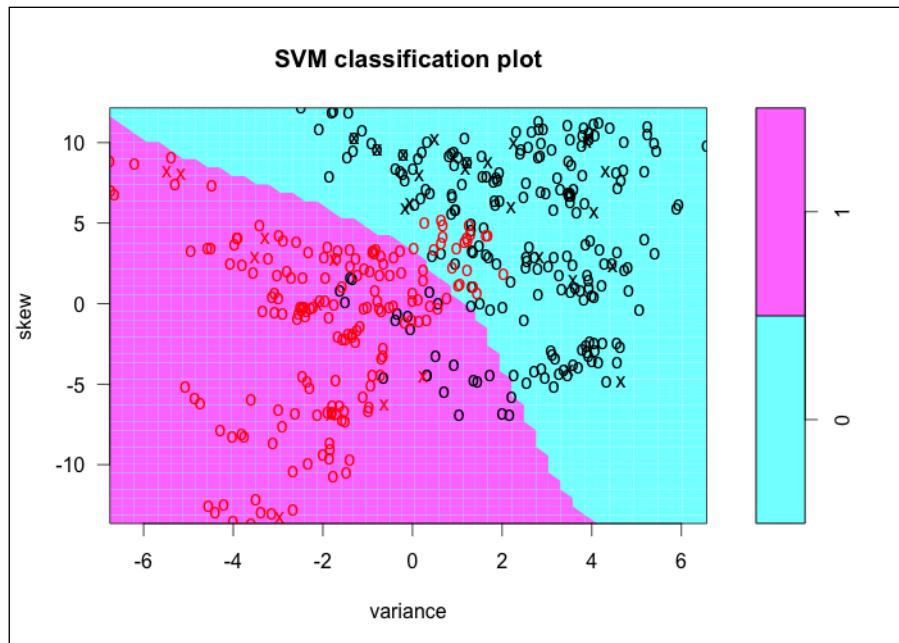
The following plot is the output of the preceding command:



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, 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, *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:

Y	No	Yes
	0.5	0.5

A-priori probabilities of each class

Conditional probabilities:

Education

Y	A		
	B	C	
No	0.5000000	0.3333333	0.1666667
Yes	0.1666667	0.3333333	0.5000000

P(Education=B | Purchase=Yes)

Y	Gender	
	F	M
No	0.5000000	0.5000000
Yes	0.3333333	0.6666667

P(Gender=M | Purchase=No)

Y	Smart_ph	
	N	Y
No	0.5	0.5
Yes	0.5	0.5

Y	Tablet	
	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, 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, *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, 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\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, 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, 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, *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, 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, *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
Actual	0	1
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.

6

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:

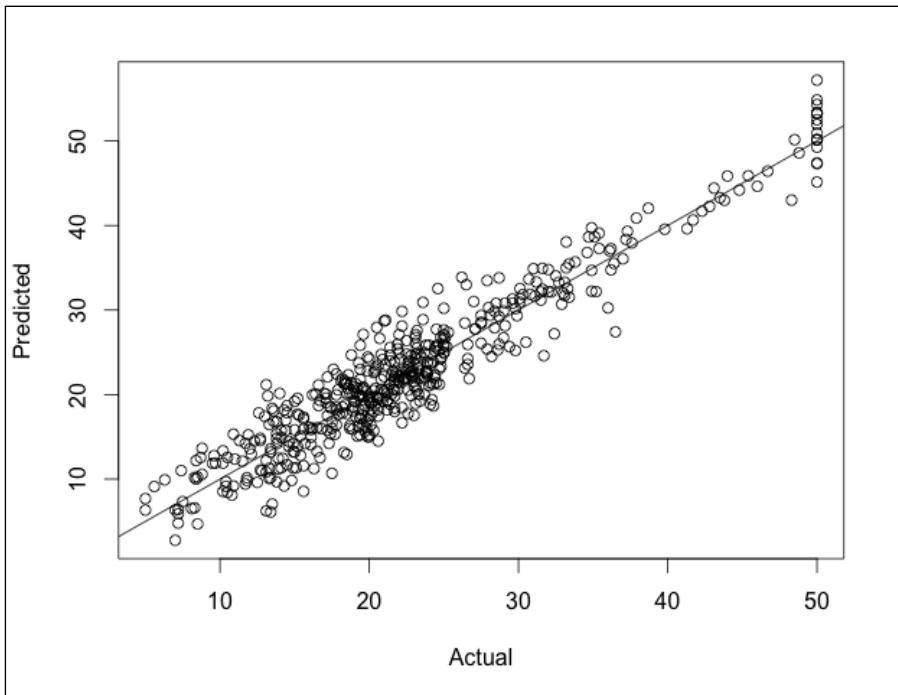
```
> dat <- read.csv("rmse-example.csv")
> rmse <- sqrt(mean((dat$price-dat$pred)^2))
> rmse
```

```
[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 `rdacb.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
```

```

> 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.

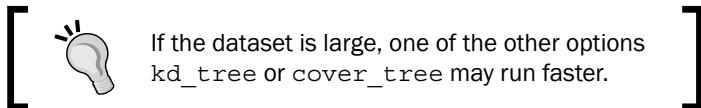
Since KNN requires all the predictors to be numeric, step 3 uses the `dummy` function from the `dummies` package to generate dummies for the categorical variable `region` and then adds the resulting dummy variables to the `educ` data frame.

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, 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.



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:

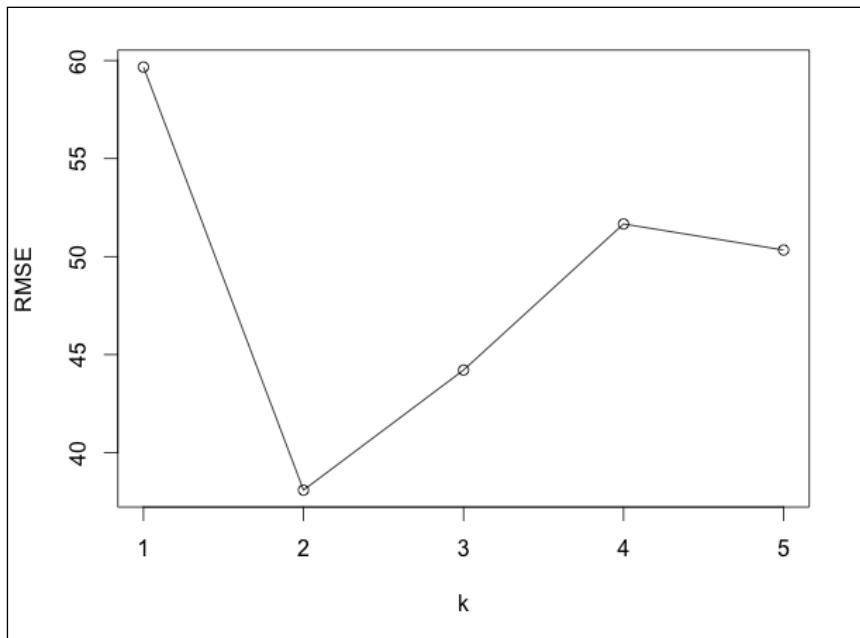
```

> 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, *What's in There? – Exploratory Data Analysis*
- ▶ *Computing the root mean squared error* in this chapter
- ▶ *Classifying using the KNN approach* in Chapter, *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 |
|---------------|-------------|---------------|---------------|---------------|
| cylinders8cyl | 39.450422   | 6.4466511     | 4.769794      | 1.967411      |
| displacement  |             |               |               |               |
| horsepower    |             |               |               |               |
| weight        |             |               |               |               |

```
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)])
```

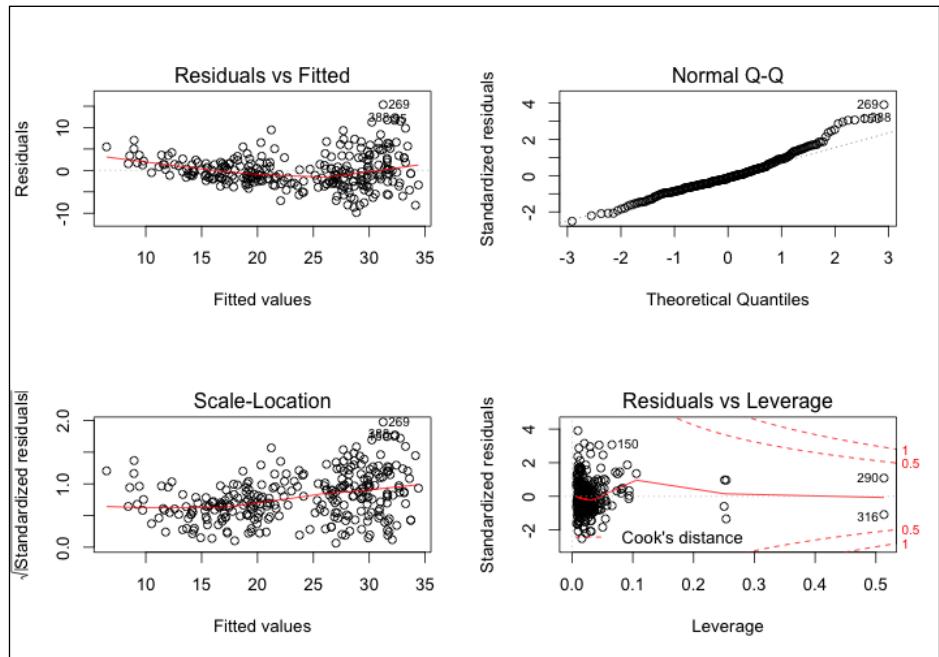
10. Compute the RMS error on the test data (your results can differ):

```
> 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 4, *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.

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:                                            |            |           |         |          |     |
|                                                          |            |           |         |          |     |
| (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  | 0.4706661 | 0.794   | 0.42803  |     |
| cylinders8cyl                                            | 6.29193    | 1.5       | 0.03451 | *        |     |
| displacement                                             | 0.00478    | 0.9       | 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 '***' 0.05 '.' 0.1<br>` ' 1             |            |           |         |          |     |
| Residual standard error: 2.052 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                                               |
| $Y \sim P * Q$<br>or<br>$Y \sim P + Q + P : Q$                          | $\hat{Y} \sim \beta_0 + \beta_1 P + \beta_2 Q + \beta_3 PQ$                                       | 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 I or Identity function is used for this.   |
| $Y \sim (P + Q + R)^{\wedge} 2$<br>or<br>$Y \sim P * Q * R - P : Q : R$ | $\hat{Y} \sim \beta_0 + \beta_1 P + \beta_2 Q + \beta_3 R + \beta_4 PQ + \beta_5 QR + \beta_6 PR$ | 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, 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)])
```

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 |

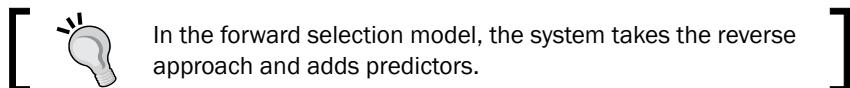
Step: AIC=774.58

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 R<sup>2</sup> has remained almost unchanged, but we got a less complex model.



## See also...

- ▶ *Creating random data partitions* in Chapter, *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 *
```

```

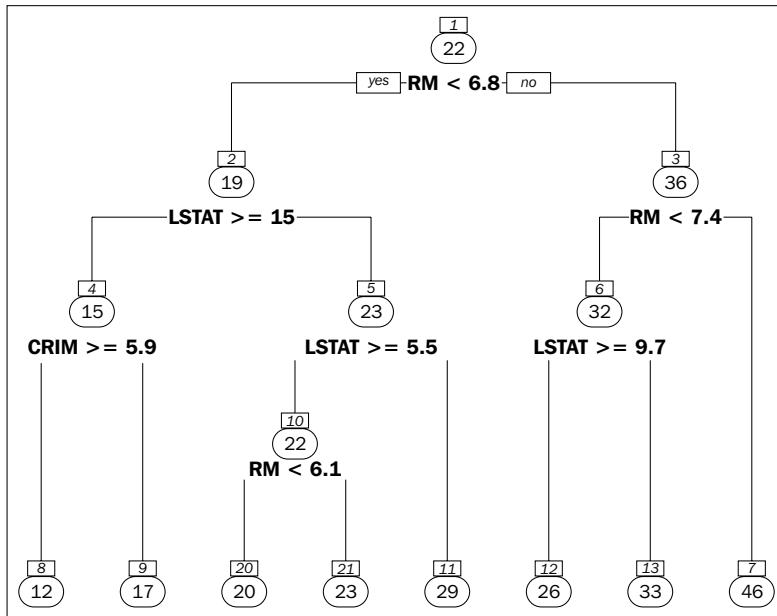
3) LSTAT< 7.865 114 8969.3230 31.93070
6) RM< 7.4525 93 3280.1050 28.70753 *
12) RM< 6.659 46 1022.5320 25.24130 *
13) RM>=6.659 47 1163.9800 32.10000
26) LSTAT>=5.495 17 329.2494 28.59412 *
27) LSTAT< 5.495 30 507.3747 34.08667 *
7) RM>=7.4525 21 444.3295 46.20476 *

```

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
```

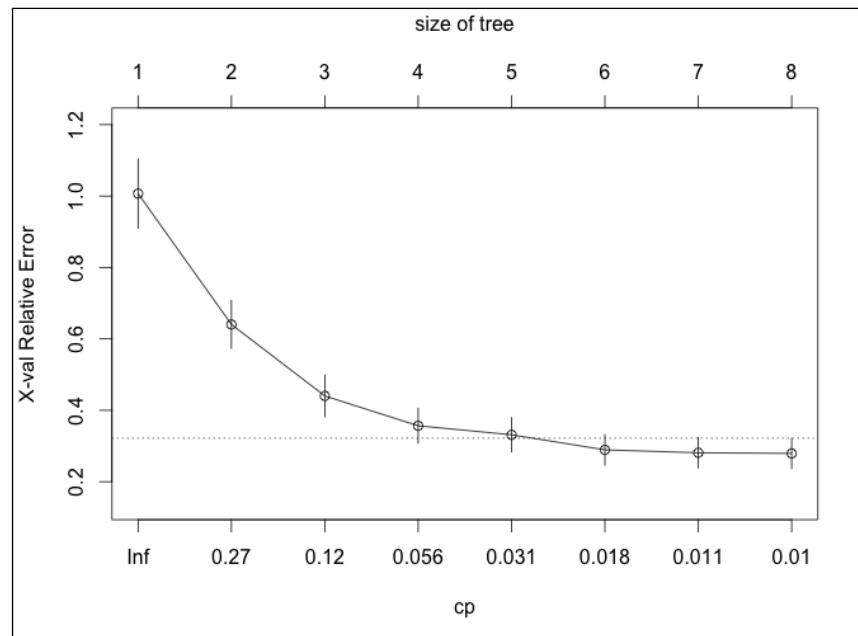
|   | CP         | nsplit | rel error | xerror    | xstd       |
|---|------------|--------|-----------|-----------|------------|
| 1 | 0.45381973 | 0      | 1.0000000 | 1.0068493 | 0.09724445 |
| 2 | 0.16353560 | 1      | 0.5461803 | 0.6403963 | 0.06737452 |
| 3 | 0.09312395 | 2      | 0.3826447 | 0.4402408 | 0.05838413 |

|   |            |   |           |           |            |
|---|------------|---|-----------|-----------|------------|
| 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)
```

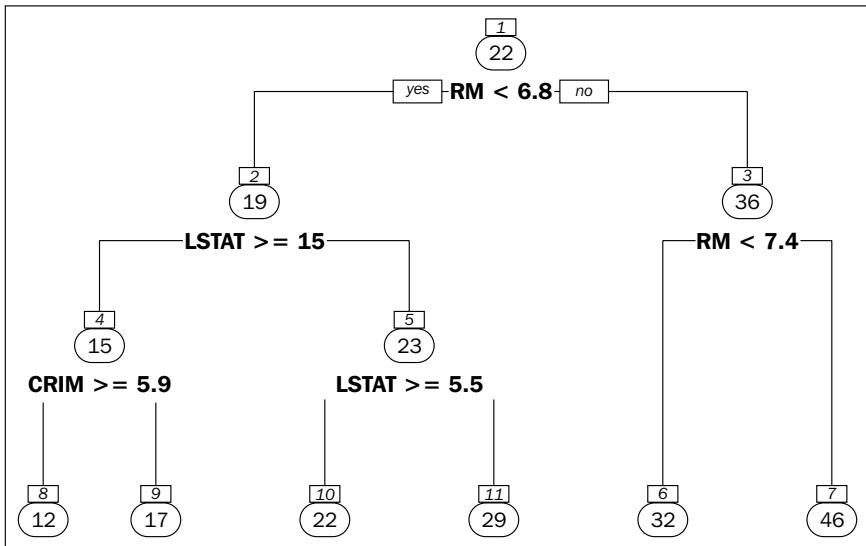
To select the best `cp` value from the plot using the 1 SD rule, pick the leftmost `cp` value for which the cross-validation relative error (y axis) lies below the dashed line. Using this value, we will pick a `cp` value of 0.018:



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, 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.print` 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...

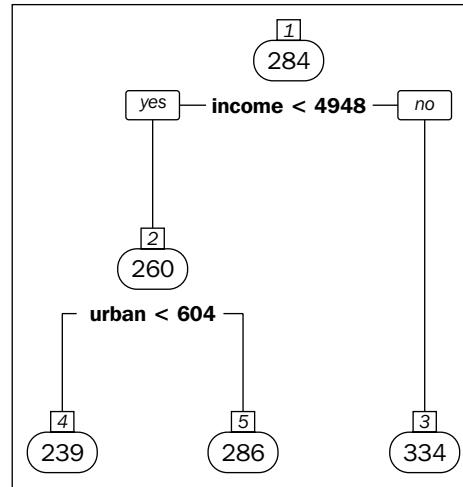
Regression trees can also be built for categorical predictors as explained in this section.

## 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, *What's in There? – Exploratory Data Analysis*
- ▶ *Building, plotting, and evaluating classification trees* in Chapter, *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
```

% Var explained: 86  
Test set MSE: 6.94  
% Var explained: 90.25

6. Examine variable importance:

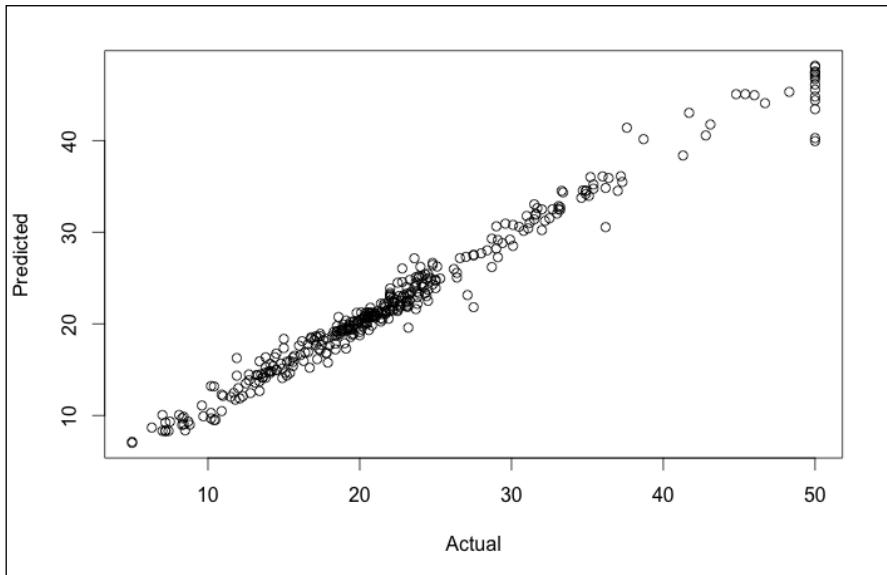
```
> mod$importance
```

|         | %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")
```

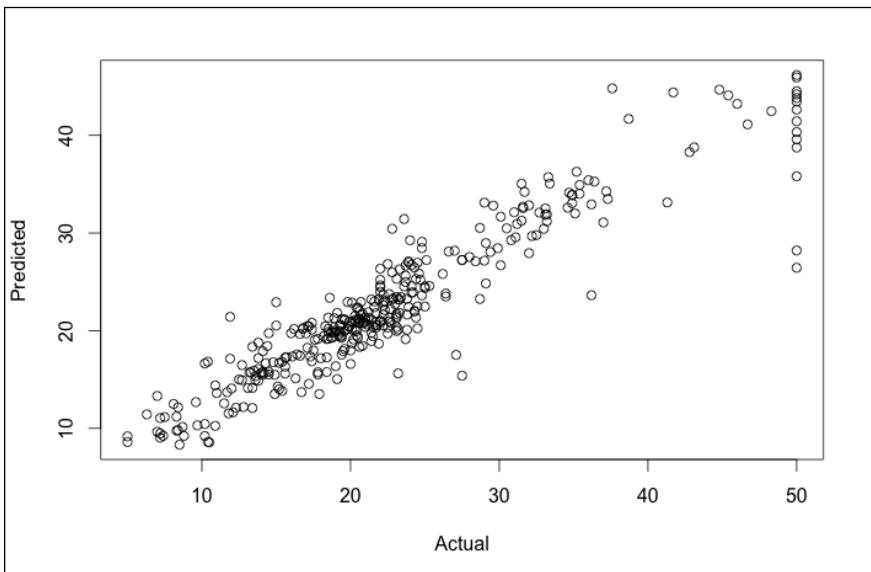
The following output is obtained on executing the preceding command:



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")
```

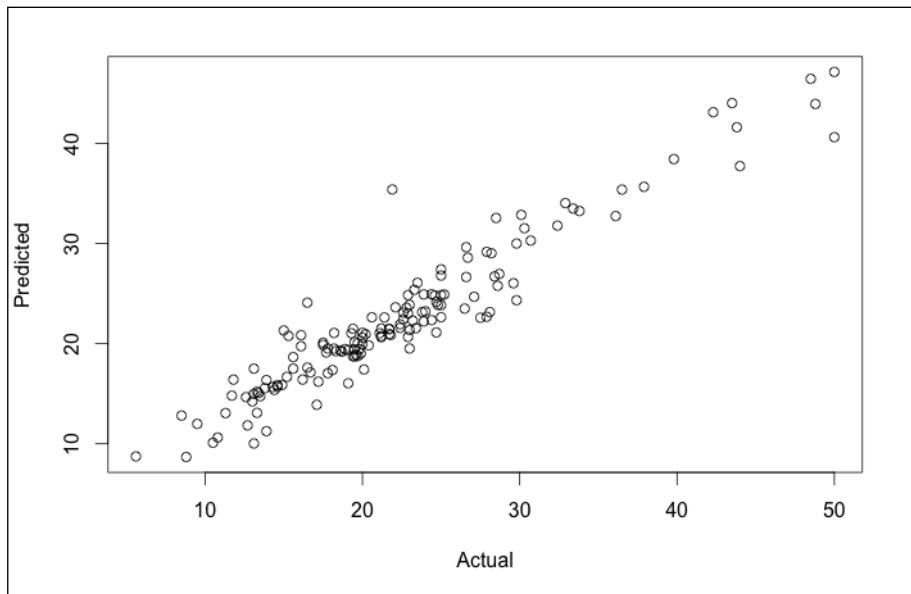
The preceding command produces the following output:



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, *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, What's in There? – Exploratory Data Analysis
- ▶ *Using random forest models for classification* in Chapter, 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)
```

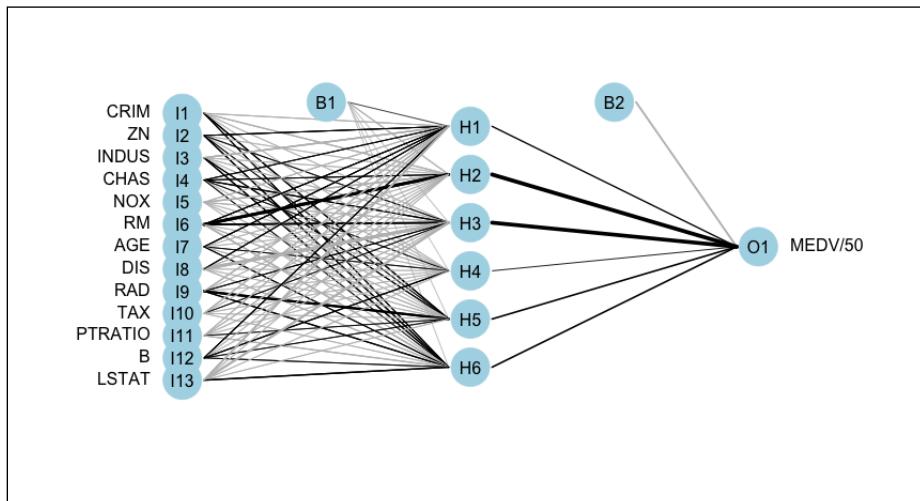
6. In preparation for plotting the network, get the code for the plotting function `plot.nnet` from fawda123's GitHub page. The following loads the function into R:

```
> 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, What's in There?* 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 external `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, What's in There? – Exploratory Data Analysis*
- ▶ *Using neural networks for classification* in *Chapter, 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.iter,
5 df, fold)
4 list("mean_sqr_errs"= mean.sqr errs,
6 "overall_mean_sqr_err" = mean(mean.sqr errs),
7 "std_dev_mean_sqr_err" = sd(mean.sqr errs))
5 }
```

```
6 rdacb.kfold.cval.reg.iter <- 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  $i^{th}$  random number is, say, 3, then the  $i^{th}$  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.

# 7

# 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 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 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 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
```

```
[331] 5 5 3 1 5 3 3 4 1 2 1 5 3 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:

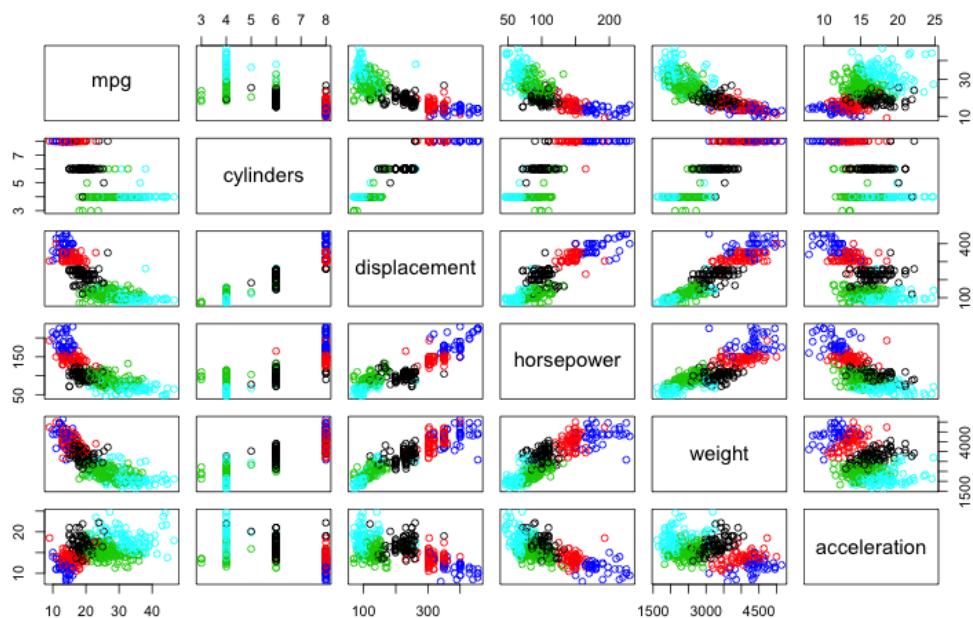
```
[1] 53.49325 134.03814 51.86729 96.53647
[5] 115.59778
(between_SS / total_SS = 81.0 %)
```

Available components:

```
[1] "cluster" "centers" "totss"
[4] "withinss" "tot.withinss" "betweenss"
[7] "size" "iter" "ifault"
```

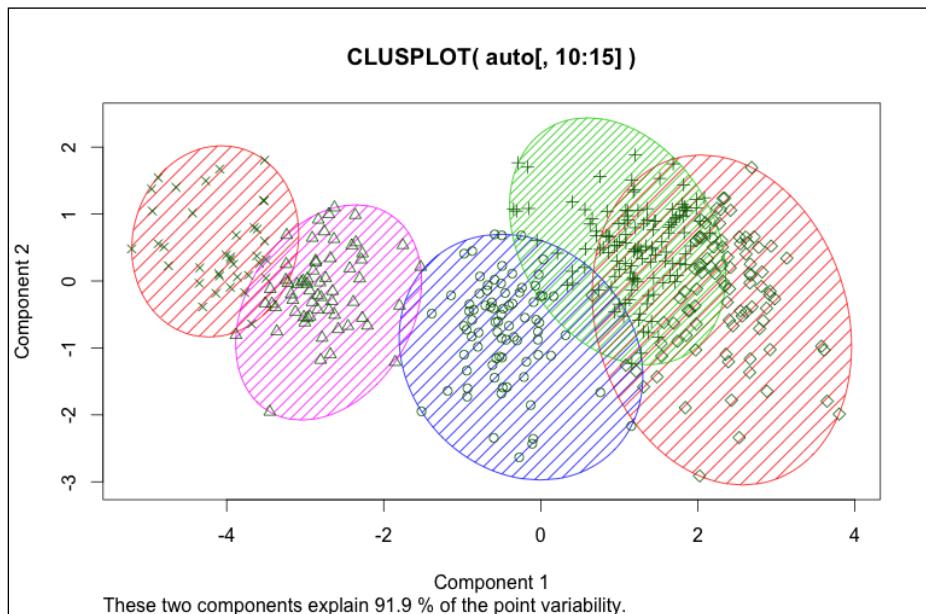
5. We performed cluster analysis on six dimensions and thus cannot visualize the complete analysis. However, we can creatively use a pairwise plot to get an idea of the clustering and visualize the results:

```
> 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                                                                                                                                                        |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| fit\$cluster      | The clustering vector, specifying the cluster to which each case belongs.                                                                                       |
| fit\$centers      | The center of each cluster.                                                                                                                                     |
| fit\$totss        | 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. |
| fit\$withinss     | Within sum of squares for each cluster.                                                                                                                         |
| fit\$tot.withinss | Total of the withinss values.                                                                                                                                   |
| fit\$betweenss    | Total sum of squares if we represent each case by just the center of its cluster.                                                                               |
| fit\$size         | Number of cases in each cluster.                                                                                                                                |
| fit\$iter         | Number of iterations used.                                                                                                                                      |
| fit\$ifault       | 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

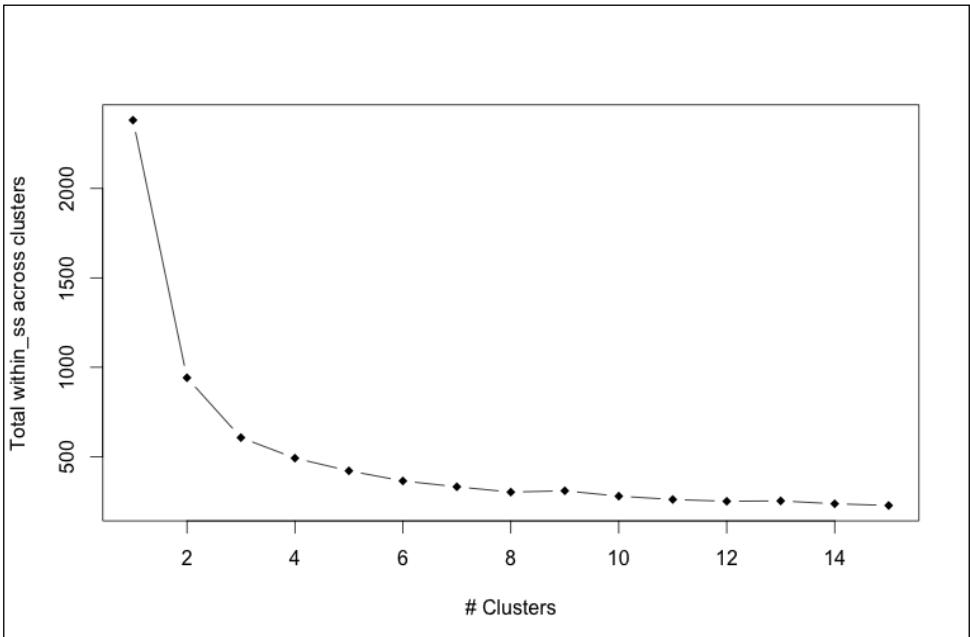
Using what we covered, you can try out various values of K and choose a suitable one. To avoid the repetitive tasks involved in doing this, we suggest the following convenience function:

```
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:

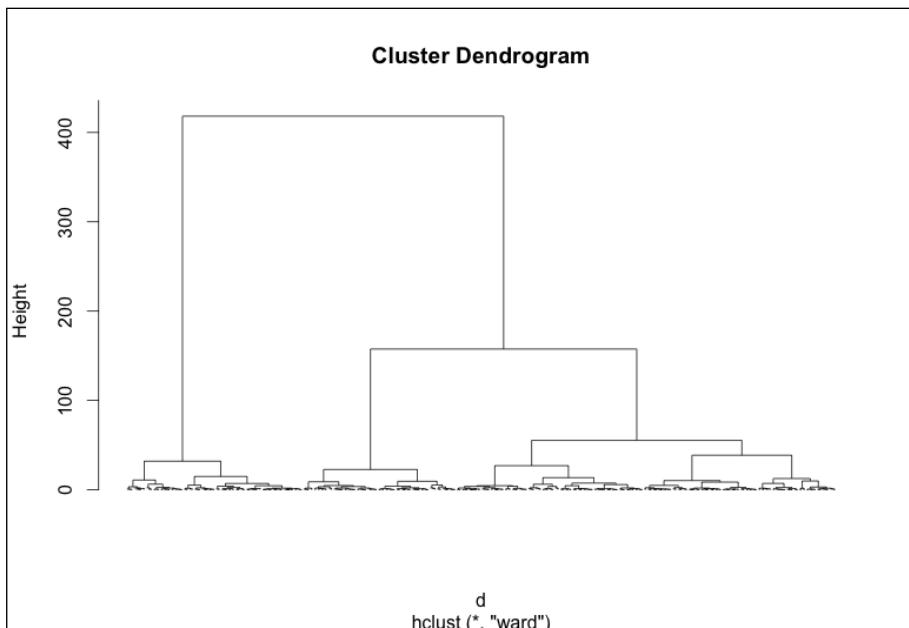
```
> dis <- dist(auto[,10:15], method = "euclidean")
```

5. Use the `hclust` function to perform hierarchical clustering:

```
> fit <- hclust(dis, method = "ward")
```

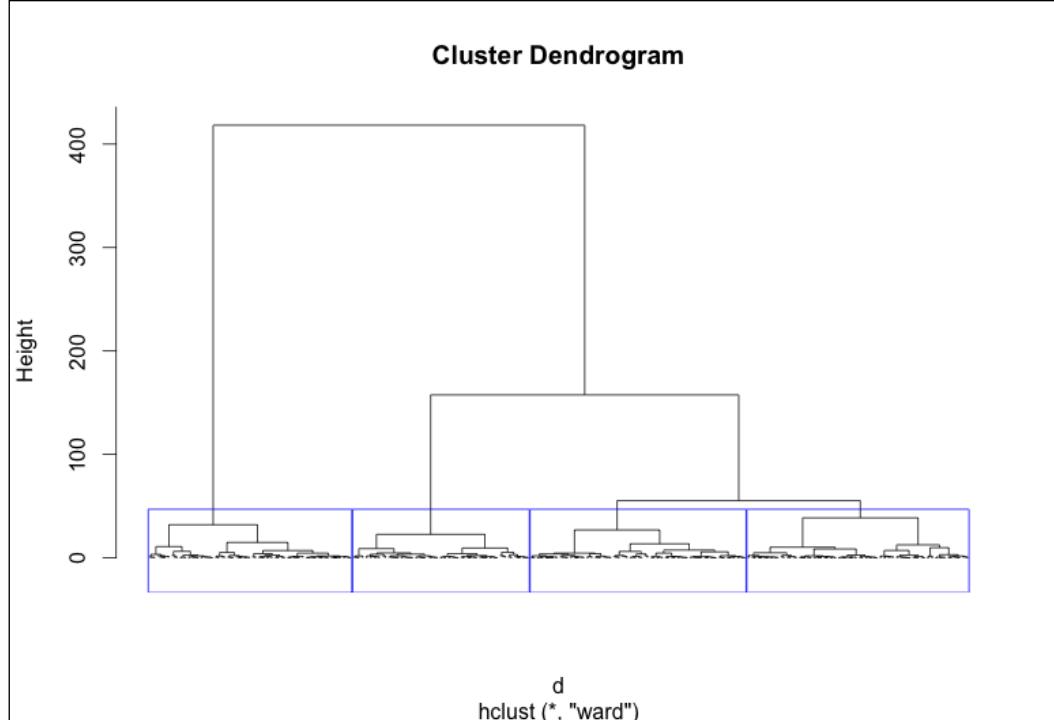
6. Plot the dendrogram representing the result of clustering. The result looks very crowded at the bottom because the function plots every single case in the data:

```
> 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 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 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 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 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 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 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)
```

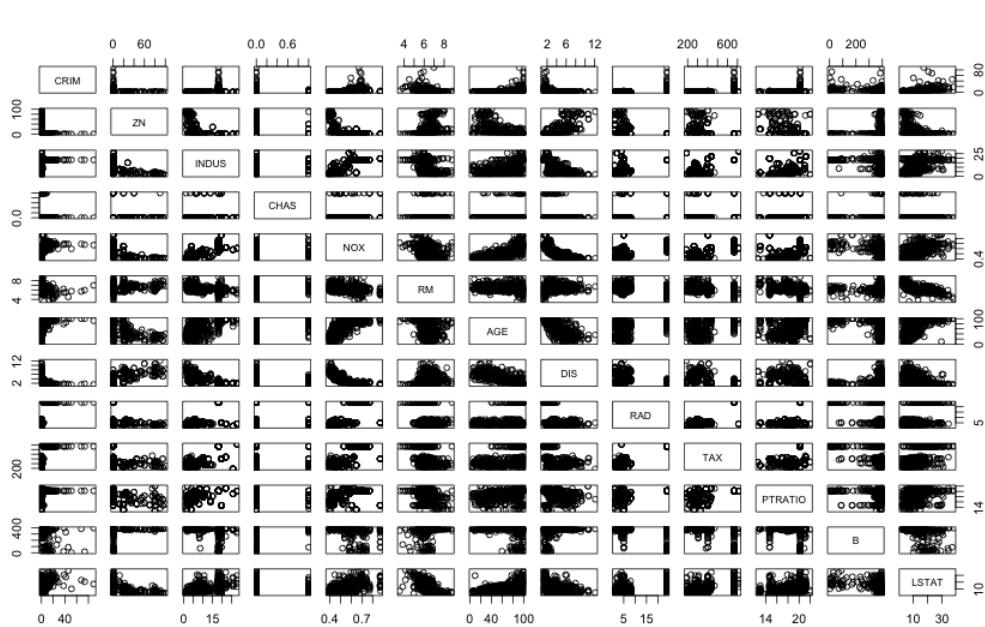
|         | CRIM  | ZN    | INDUS | CHAS    | NOX   | RM    | AGE   |
|---------|-------|-------|-------|---------|-------|-------|-------|
| CRIM    | 1.00  | -0.20 | 0.41  | -0.06   | 0.42  | -0.22 | 0.35  |
| ZN      | -0.20 | 1.00  | -0.53 | -0.04   | -0.52 | 0.31  | -0.57 |
| INDUS   | 0.41  | -0.53 | 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:

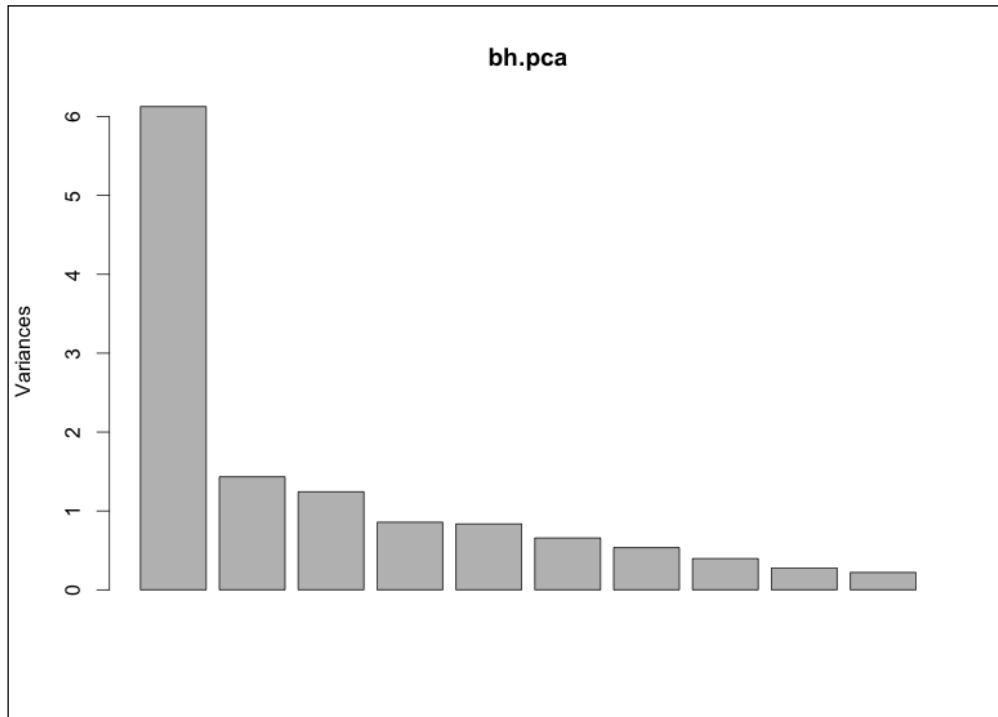
|                        | PC1     | PC2     | PC3     | PC4     |
|------------------------|---------|---------|---------|---------|
| Standard deviation     | 2.4752  | 1.1972  | 1.11473 | 0.92605 |
| Proportion of Variance | 0.4713  | 0.1103  | 0.09559 | 0.06597 |
| Cumulative Proportion  | 0.4713  | 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:

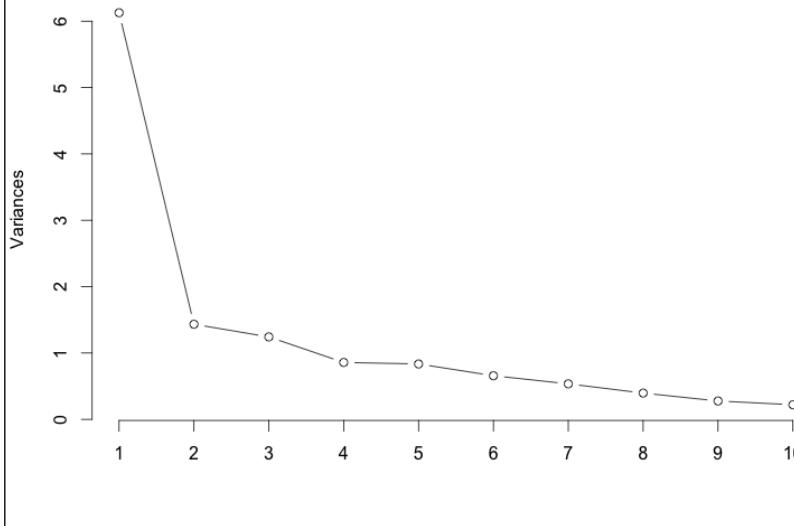
```
> # barplot
> plot(bh.pca)
```



For a scree plot the following command can be used:

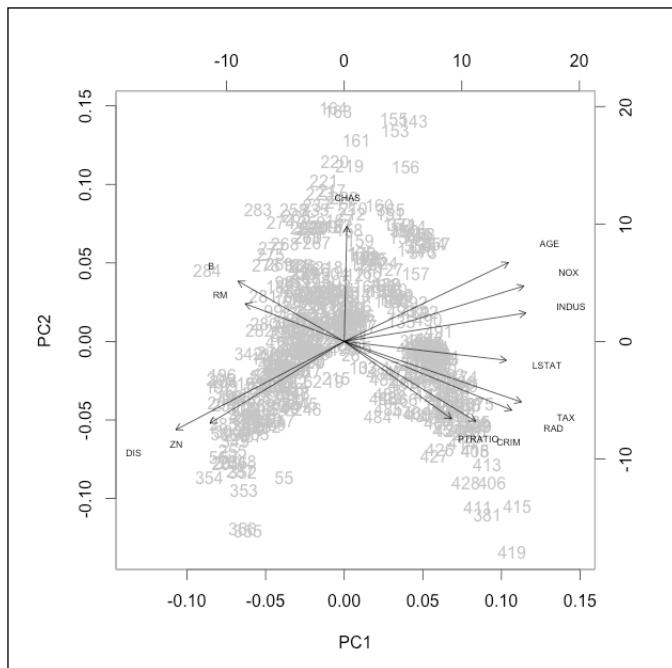
```
> # scree plot
> plot(bh.pca, type = "lines")
```

**bh.pca**



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 = ...)
```

# 8

# 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...

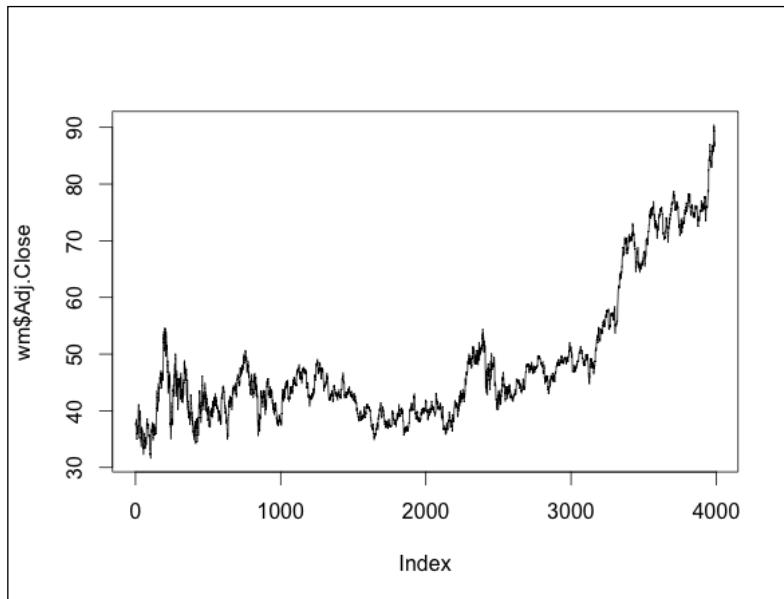
1. Read the file. We will use a data file that has the share prices of Walmart (downloaded from Yahoo Finance) between March 11, 1999 and January 15, 2015:

```
> wm <- read.csv("walmart.csv")
```

2. View the data as a line chart:

```
> plot(wm$Adj.Close, type = "l")
```

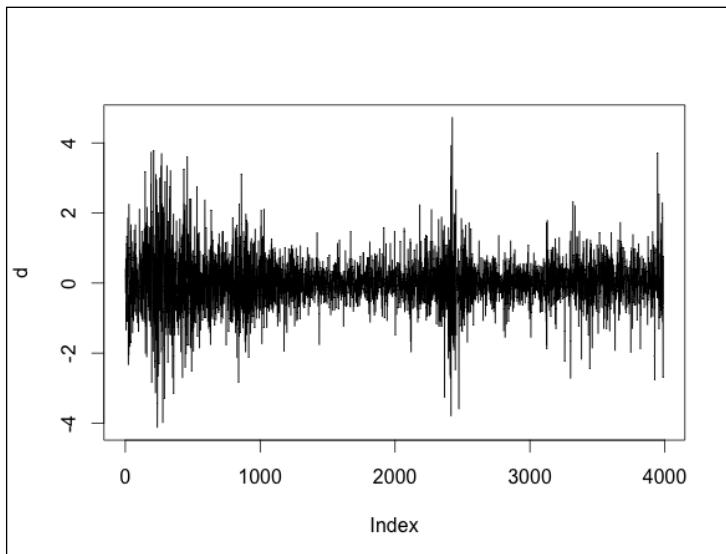
The data can be viewed as a line chart as follows:



3. Compute and plot daily price movements:

```
> d <- diff(wm$Adj.Close)
> plot(d, type = "l")
```

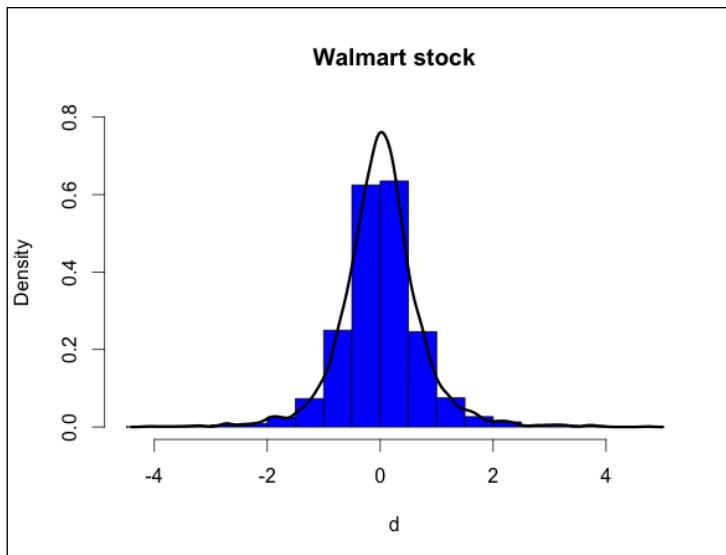
The plotted daily price movements appear as follows:



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)
```

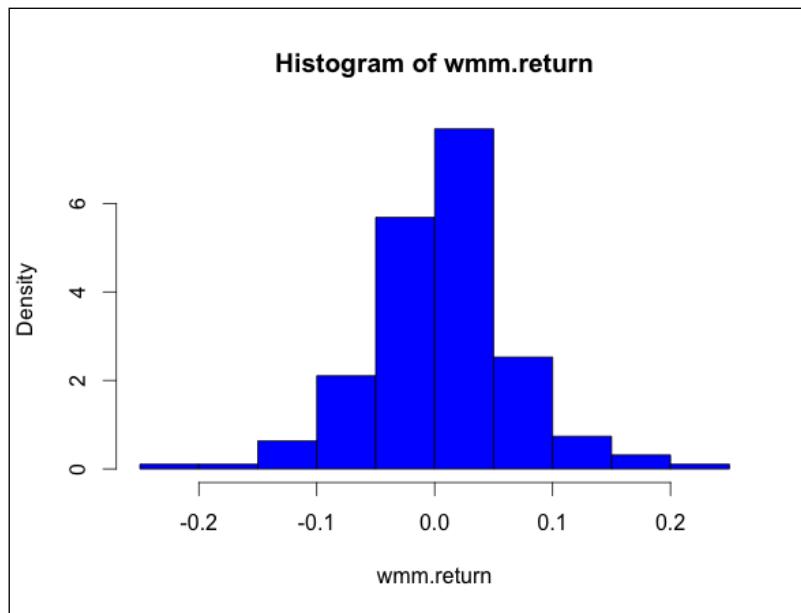
The following histogram shows daily price change:



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.

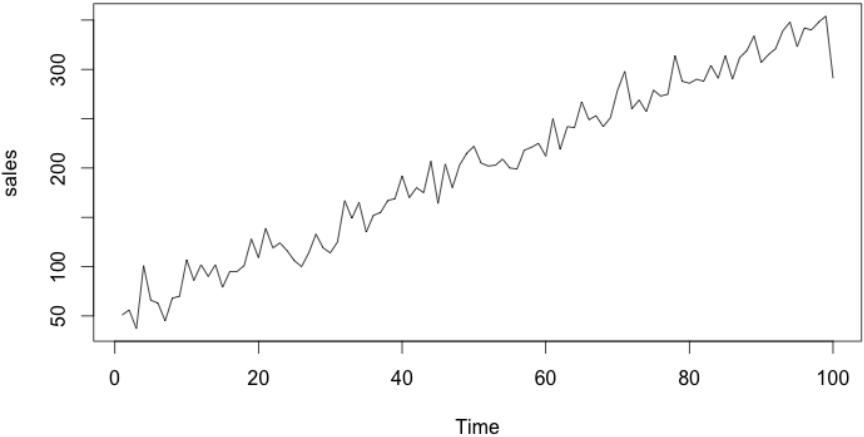
## How to do it...

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:  

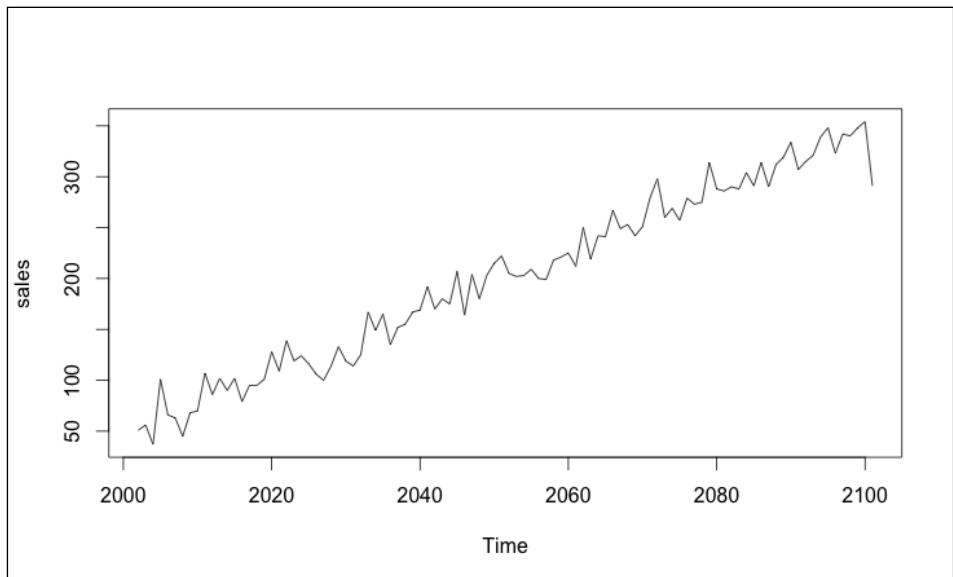
```
> plot(s.ts)
```



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:



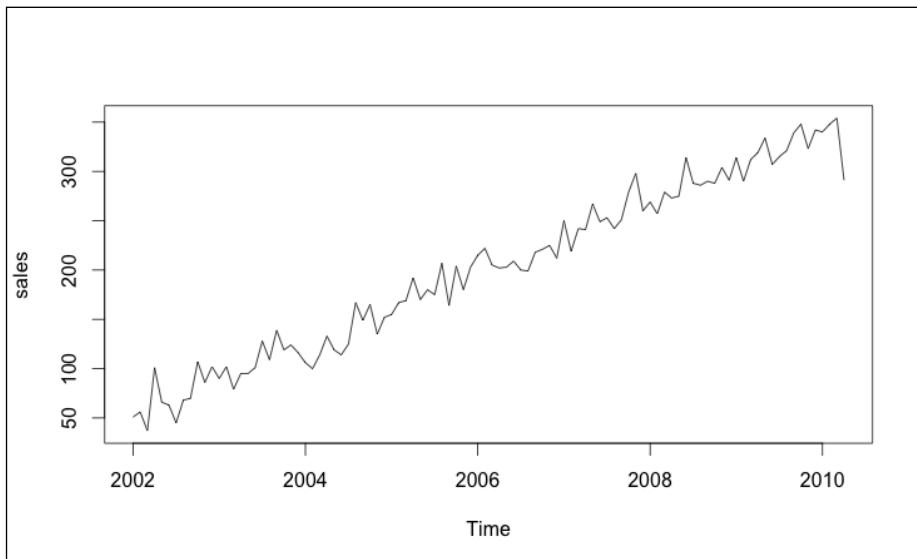
To create a monthly time series run the following command:

```
> # 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
```

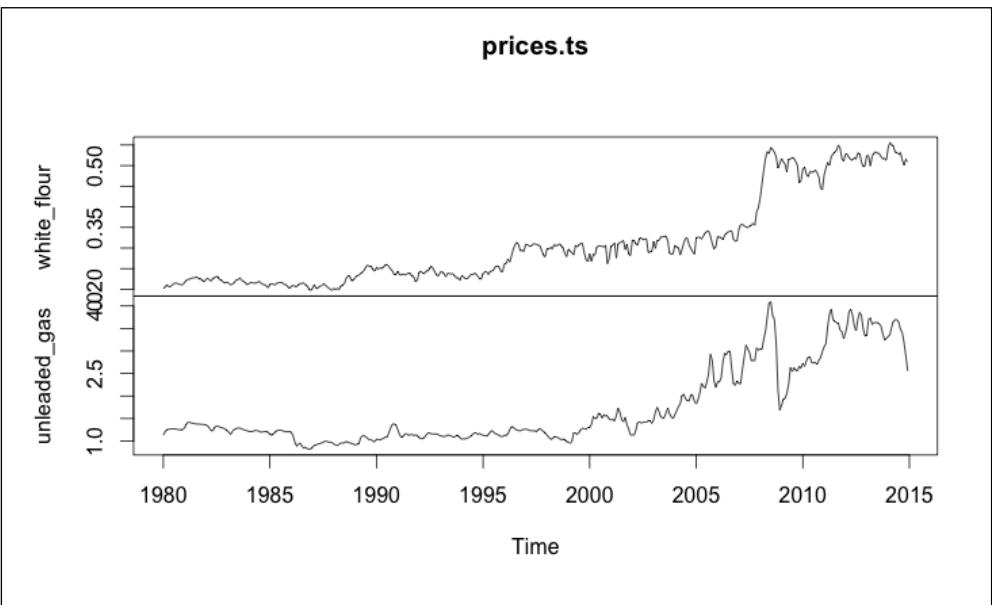
6. Create a time series object with multiple time series. This data file contains US monthly consumer prices for white flour and unleaded gas for the years 1980 through 2014 (downloaded from the website of the US Bureau of Labor Statistics):

```
> 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:

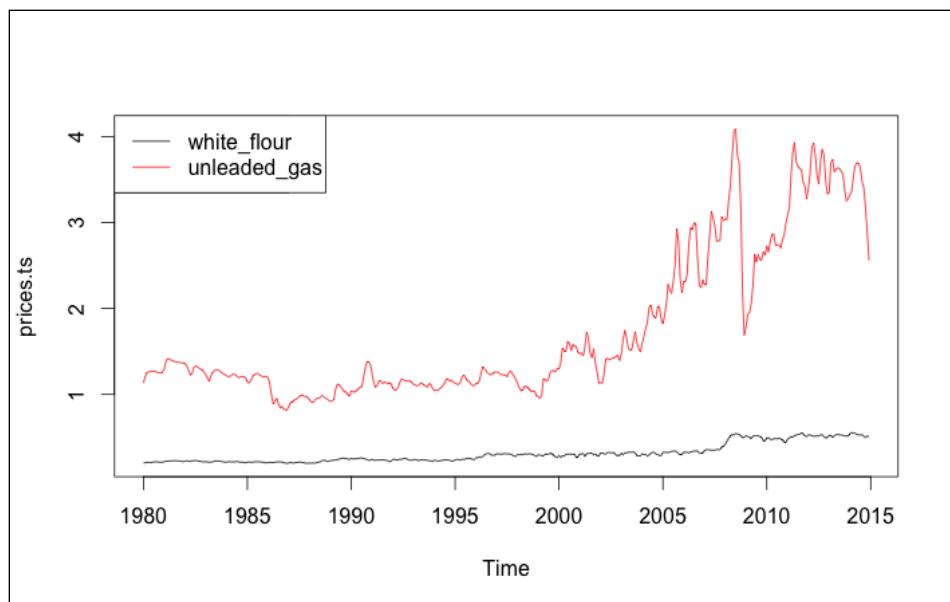
```
> plot(prices.ts)
```

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 R 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])
```

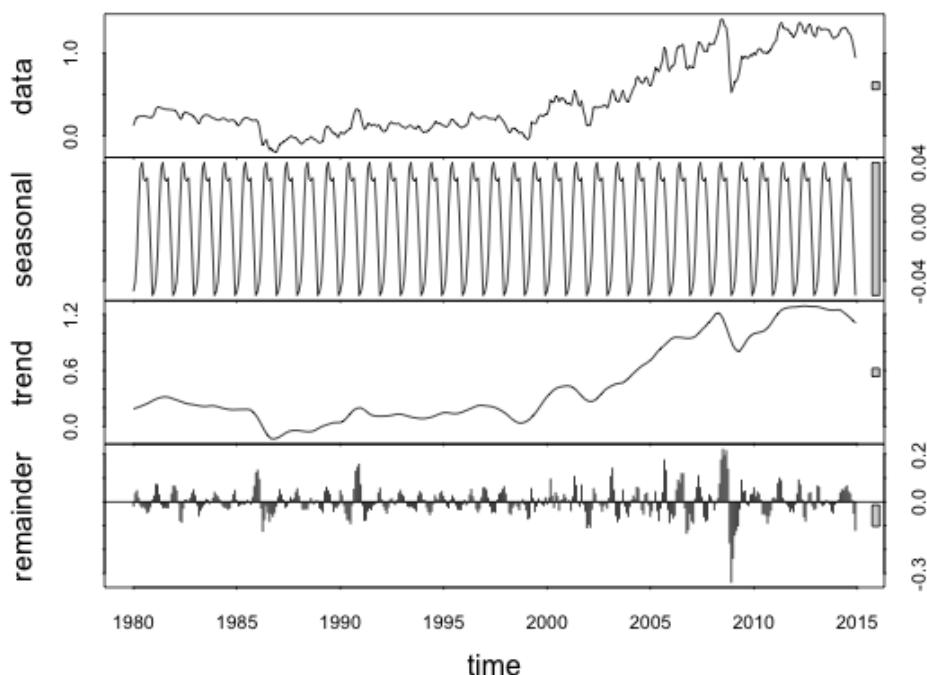
3. The plot shows seasonality in gas prices. The amplitude of fluctuations seems to increase with time and hence this looks like a multiplicative time series. Thus, we will use the log of the prices to make it additive. Use the `stl` function to perform a Loess decomposition of the gas prices:

```
> prices.stl <- stl(log(prices.ts[,1]), s.window =
 "period")
```

4. Plot the results of `stl`:

```
> plot(prices.stl)
```

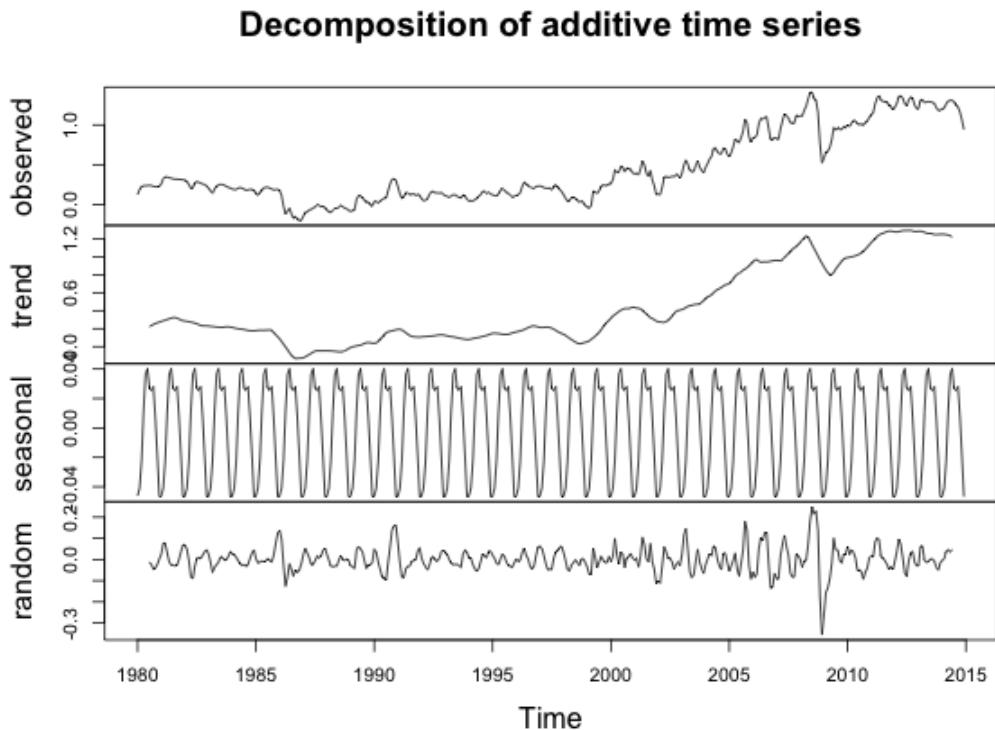
The following plot is the result of the preceding command:



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)
```

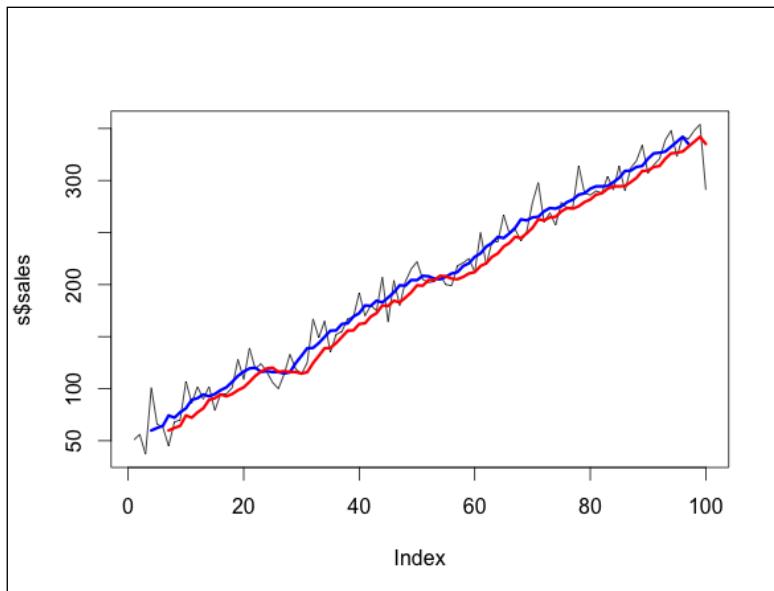
3. Compute the symmetrically filtered values (three past values, one current value, and three future values) and one-sided values (one current and six past values):

```
> 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)
```

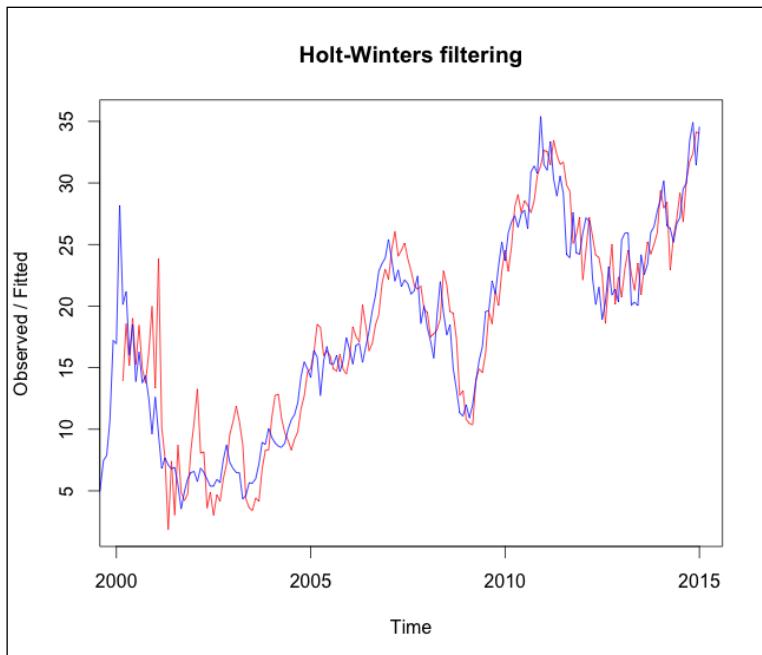
3. Perform Holt-Winters exponential smoothing:

```
> 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:

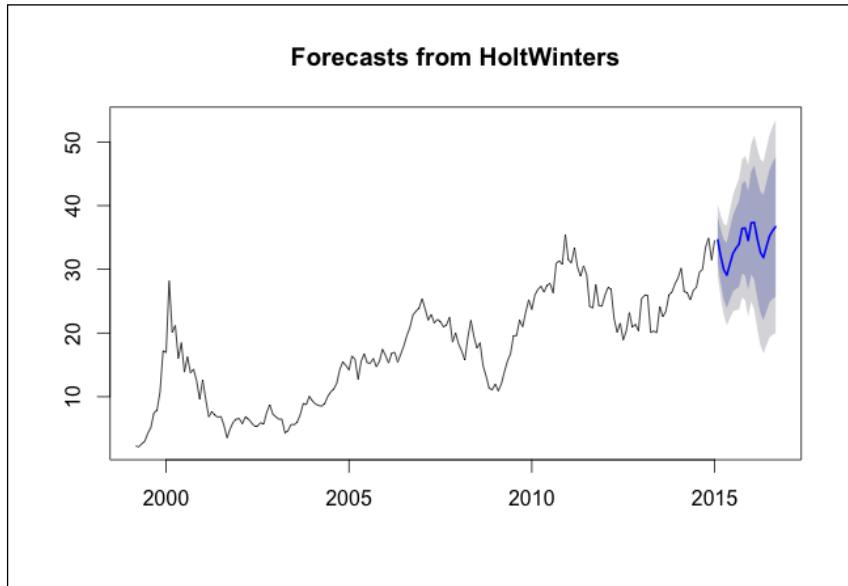
```
> # See the squared errors
> infy.hw$SSE
[1] 1446.232
> # The alpha beta and gamma used for filtering
> infy.hw$alpha
 alpha
0.5658932
> infy.hw$beta
 beta
0.009999868
> infy.hw$gamma
 gamma
1
> # the fitted values
> head(infy.hw$fitted)
 xhat level trend season
[1,] 13.91267 11.00710 0.5904618 2.31510417
[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)
```

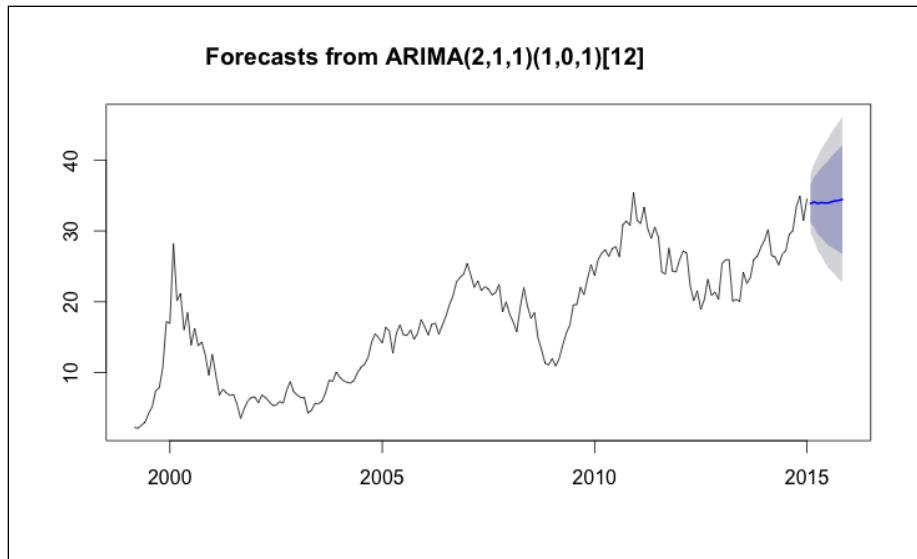
4. Generate the forecast using the ARIMA model:

```
> 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

# 9

# 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, 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, 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, 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, Acquire and Prepare the Ingredients – Your Data](#)
- ▶ [Slicing, dicing, and combining data with data tables in Chapter, 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, user\_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
```

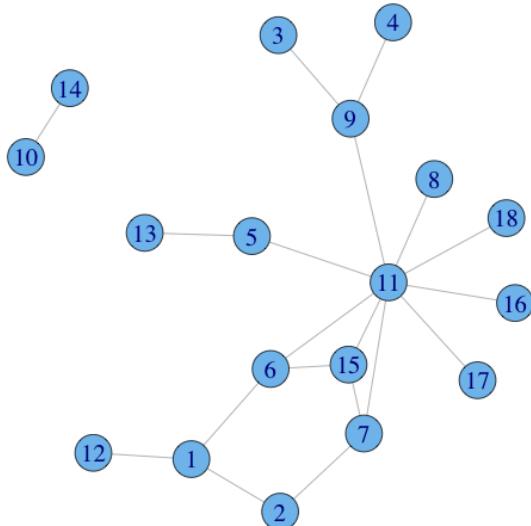
3. Renumber the users. Since we filtered the graph significantly, we have only 18 unique users left, but they retain their original user IDs. We will find it convenient to sequence them with unique numbers from 1 through 18. This step is not strictly needed, but will make the social network graph look 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)
```

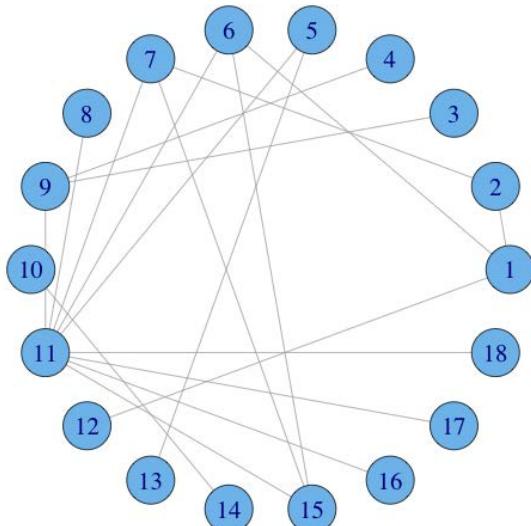
Your plot may look different in terms of layout, but if you look carefully, you will see that the nodes and edges are identical:



5. Plot the graph object with a different layout:

```
> plot.igraph(g,layout=layout.circle, vertex.size = 20)
```

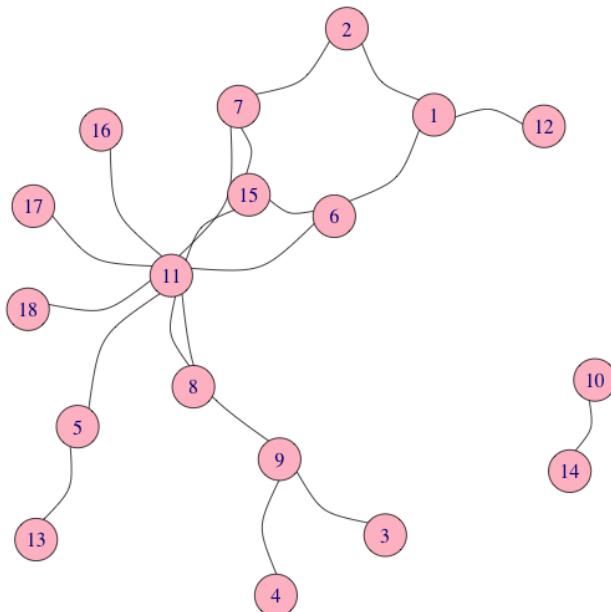
The following graph is the output of the preceding command:



6. Plot the graph object using colors for the vertices and edges:

```
> plot.igraph(g, edge.curved=TRUE, vertex.color="pink",
edge.color="black")
```

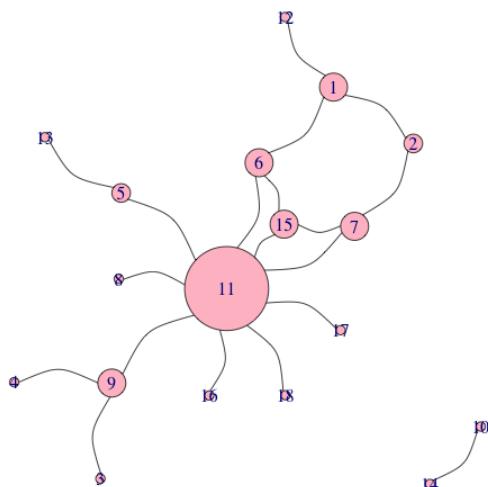
Again, your plot may be laid out differently, but should be functionally identical to the following plot:



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:



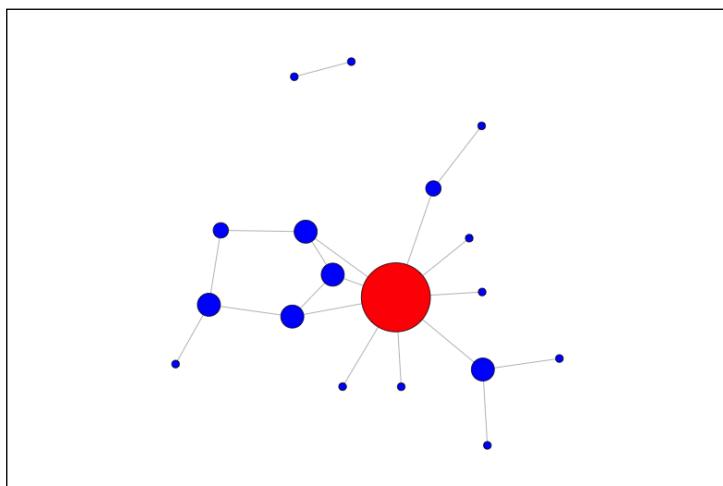
8. Plot the graph with the node size and color based on degree:

```

> 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:



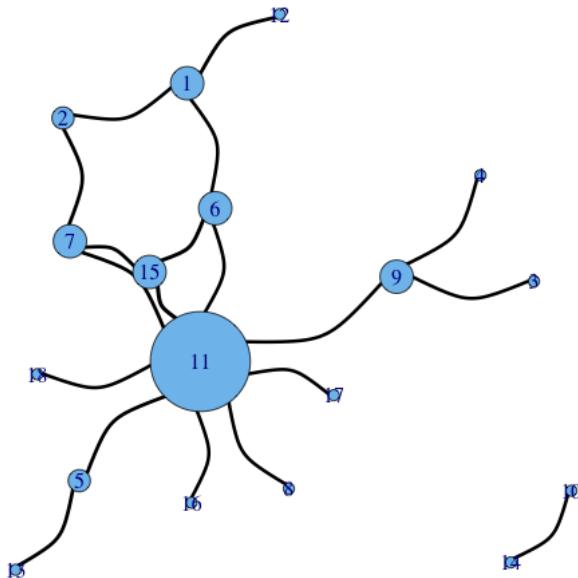
9. Plot the graph with the edge thickness proportional to edge weights:

>  $E(g) \$x$

```
[1] 17 18 17 18 17 17 18 18 17 18 17 17 17 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

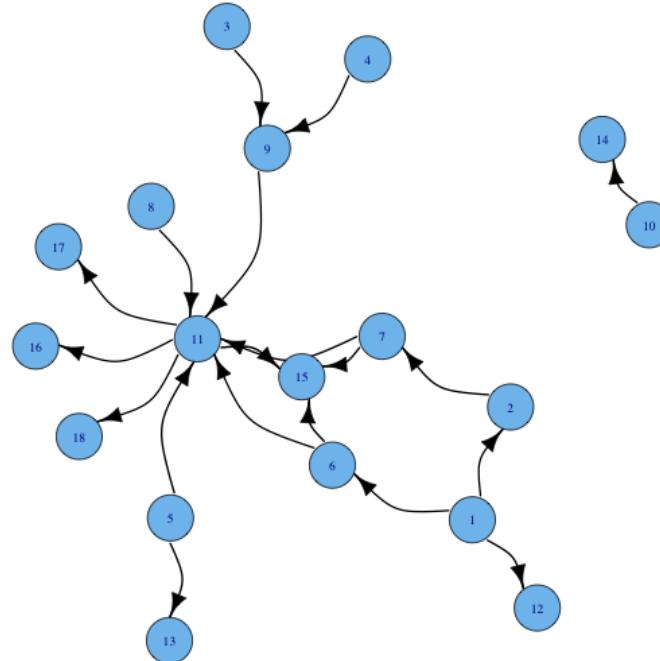
We showed only a few options to control the look of the plot. You have many more options for controlling the look of the edges, nodes, and other aspects. The documentation of `plot.igraph` mentions these options. When using them, remember to prefix the node options with (`vertex.`) and edge options with (`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/l), name (v/c)

> # The B above tells us that this is a bipartite graph
> # Explore bg
```

```
> V(bg)$type
```

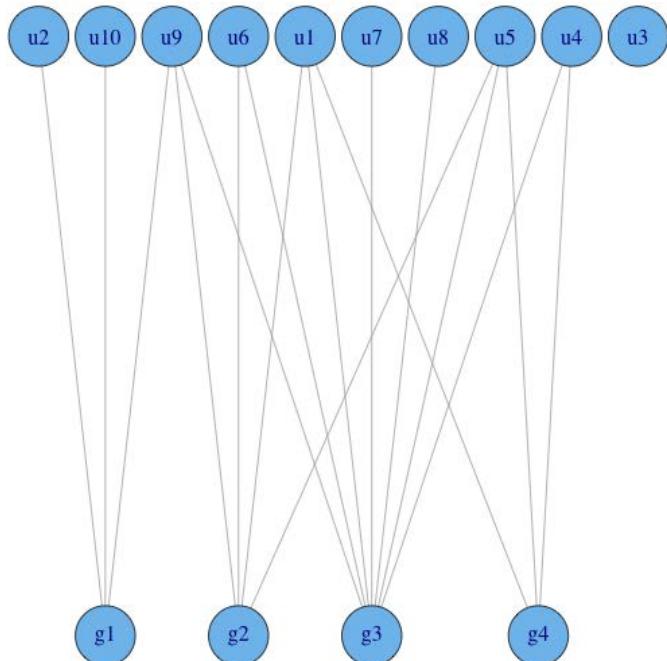
```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[9] FALSE FALSE TRUE TRUE TRUE TRUE
```

```
> # 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:

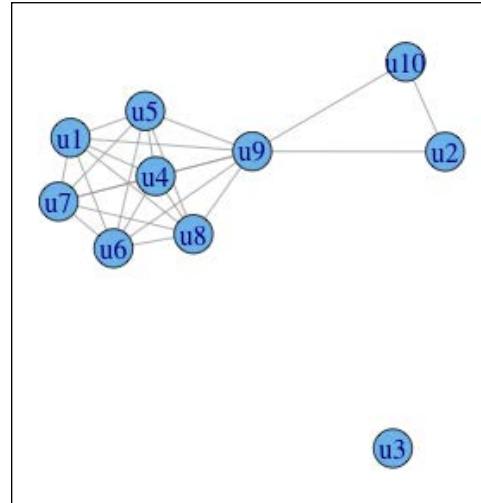
```
> # Generate the two projections
> p <- bipartite.projection(bg)
> p

$pj1
IGRAPH UNW- 10 24 --
+ attr: name (v/c), weight (e/n)

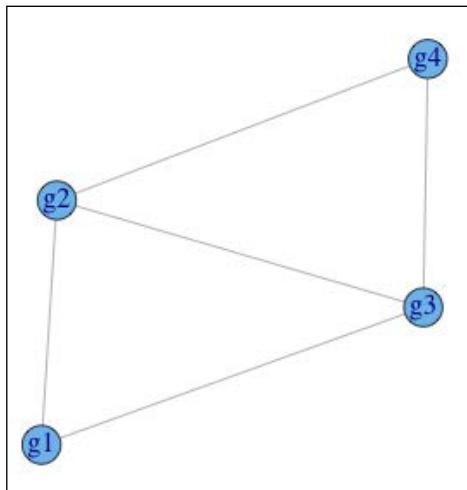
$pj2
IGRAPH UNW- 4 5 --
+ attr: name (v/c), weight (e/n)

> plot(p$pj1, vertex.size = 20)
> plot(p$pj2, 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 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 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
```

```
> 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)
```

# 10

## 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
syntax for authoring HTML, PDF, and MS Word documents. For more
details on using R Markdown see <http://rmarkdown.rstudio.com>.
9
10 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. You can embed an R code chunk like this:
11
12 ``{r}
13 summary(cars)
14 ``
15
16 You can also embed plots, for example:
17
18 ``{r, echo=FALSE}
19 plot(cars)
20 ``
21
22 Note that the `echo = FALSE` parameter was added to the code chunk
to prevent printing of the R code that generated the plot.
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.

- 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.
- The metadata section is between two lines, each with just three hyphens, shown as follows:

```

```

```
title: "Markdown Document"
author: "Shanthy Viswanathan"
date: "December 8, 2014"
output:
 html_document:
 theme: cosmo
 toc: yes

```

# Markdown Document

*Shanthy Viswanathan*

*December 8, 2014*

- [Introduction](#)
- [HTML Content](#)
- [Embed Code](#)
  - [Set Directory](#)
  - [Load data](#)
    - [Plot Data](#)
    - [Plot with format options](#)

- 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")
```

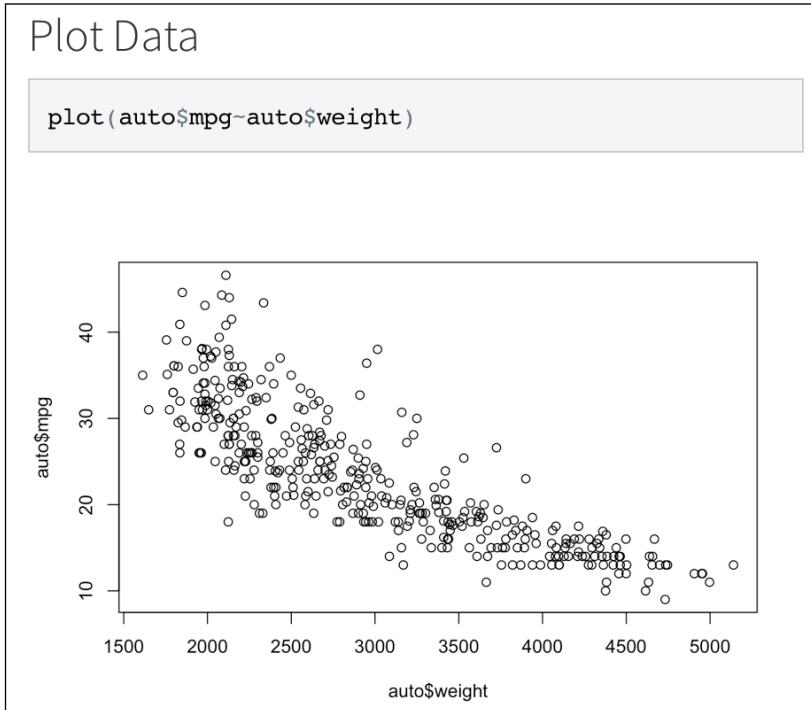
```
```

```

12. Plot the data:

```
```{r plotData }
plot(auto$mpg~auto$weight)
```
```

The output of the preceding command can be seen here:

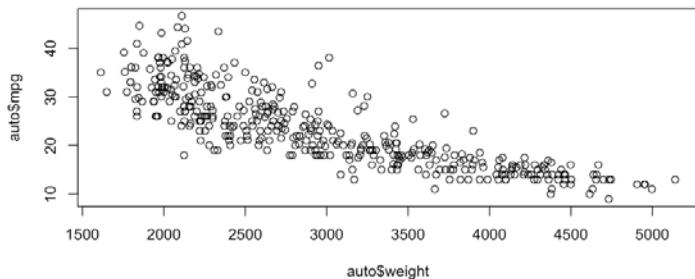


13. Plot with the format options:

```
```{r plotFormatData, echo=FALSE, fig.height=4, fig.width=8}
plot(auto$mpg~auto$weight)
str(auto)
```
```

The following screenshot shows the plotted output:

## Plot with format options



```
'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 : int 4 3 4 4 6 4 8 4 6 8 ...
$ displacement: num 140 70 107 97 199 115 304 1
19 250 318 ...
$ horsepower : int 90 97 75 92 90 95 120 100 1
05 150 ...
$ weight : int 2264 2330 2205 2288 2648 26
94 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 : Factor w/ 305 levels "amc ambass
ador brougham",...: 66 184 165 86 8 18 11 79 42 112 .
..
```

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 `r`, 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:

```
rmarkdown::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%
```

# 11

# 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]
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 | 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      | 4cyl | chevrolet vega        | 2300 29.28676 |
| 2:   | 3   | 36.0      | 4cyl | honda accord          | 29.28676      |
| 3:   | 4   | 28.0      | 4cyl | datsun 510 (sw)       | 29.28676      |
| 4:   | 6   | 23.0      | 4cyl | audi 100ls            | 29.28676      |
| 5:   | 8   | 32.9      | 4cyl | datsun 200sx          | 29.28676      |
| ---  |     |           |      |                       |               |
| 200: | 391 | 32.1      | 4cyl | chevrolet chevette    | 29.28676      |
| 201: | 392 | 23.9      | 4cyl | datsun 200-sx         | 29.28676      |
| 202: | 395 | 34.5      | 4cyl | plymouth horizon tc3  | 29.28676      |
| 203: | 396 | 38.1      | 4cyl | toyota corolla tercel | 29.28676      |
| 204: | 397 | 30.5      | 4cyl | 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 23.00 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        |

# 12

## 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:

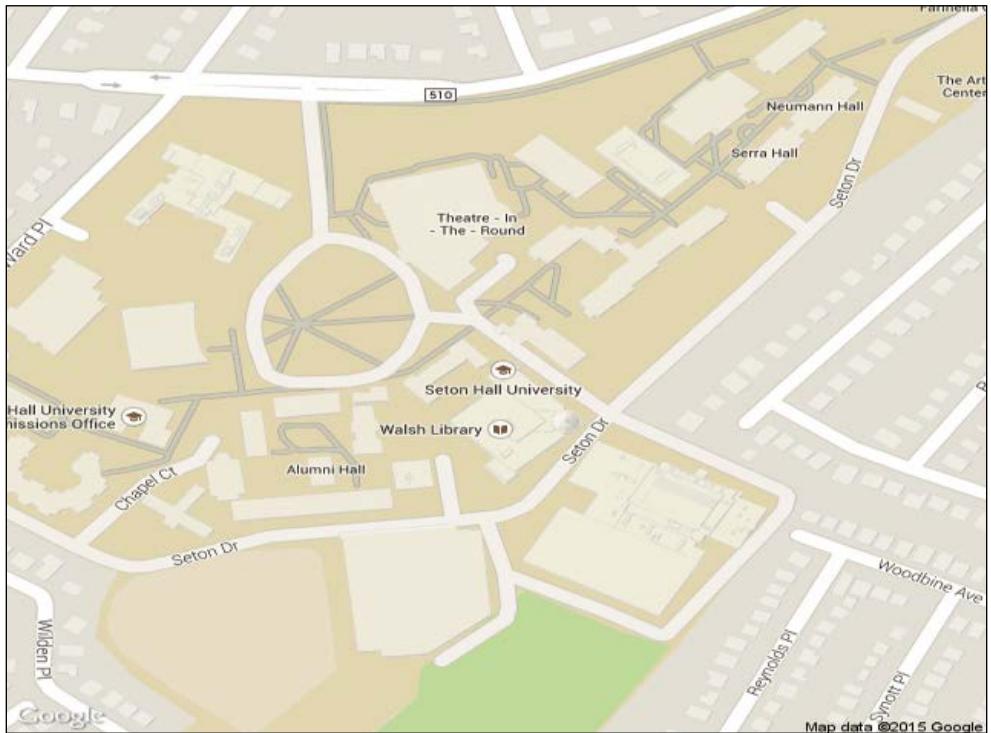
1. Load the `RgoogleMaps` package:

```
> 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)
```

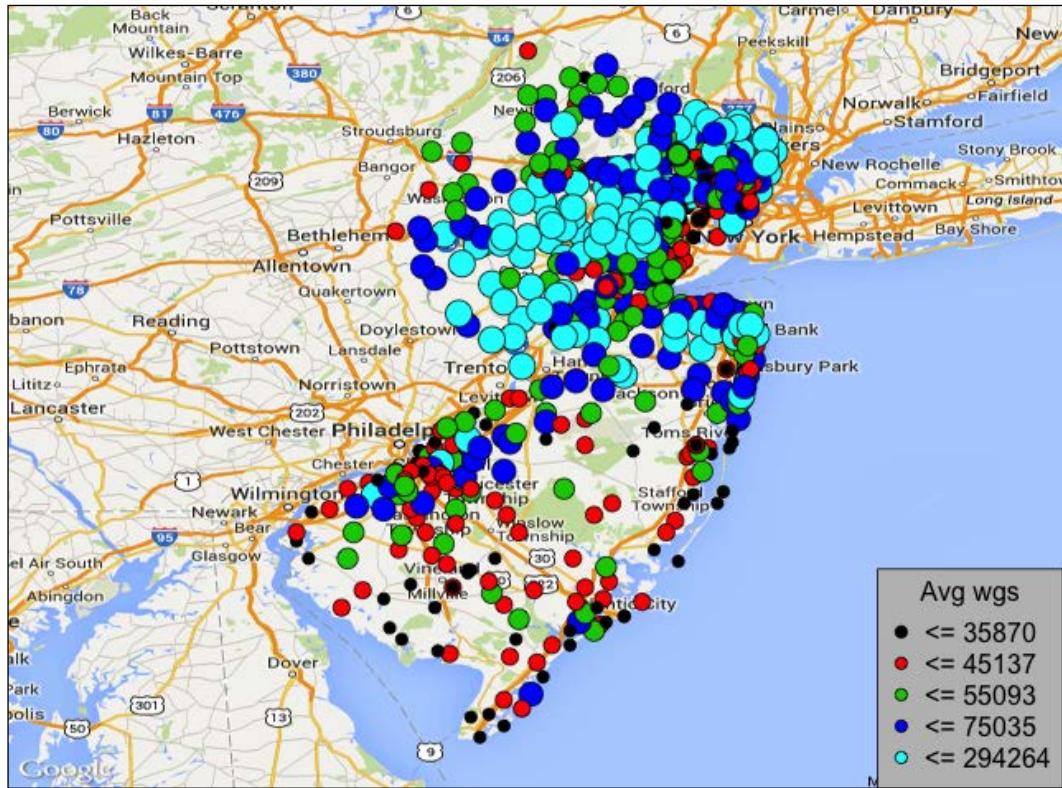
```
[1] "http://maps.google.com/maps/api/staticmap?center=40.115,-
74.715&zoom=8&size=640x640&maptype=mobile&format=png32&sensor=true"
center, zoom: 40.115 -74.715 8
```

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:

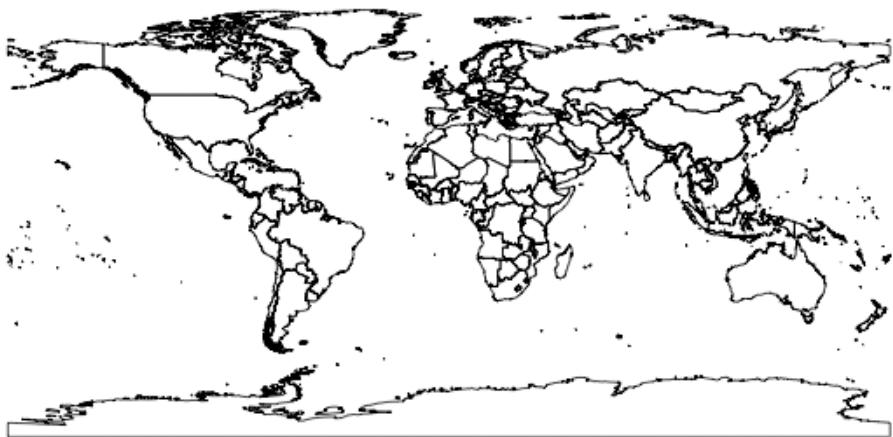
```
> library(sp)
> library(rgdal)
```

2. Read the data:

```
> countries_sp <- readOGR(".", "ne_50m_admin_0_countries")
> airports_sp <- readOGR(".", "ne_50m_airports")
```

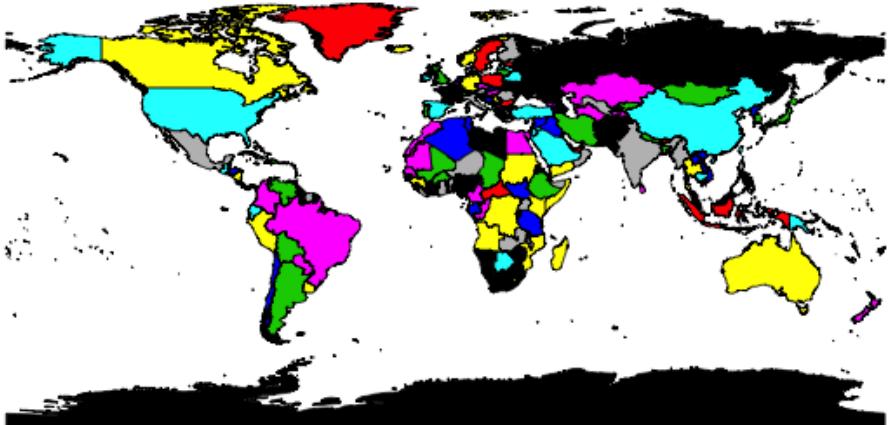
3. Plot the countries without color:

```
> # without color
> plot(countries_sp)
```



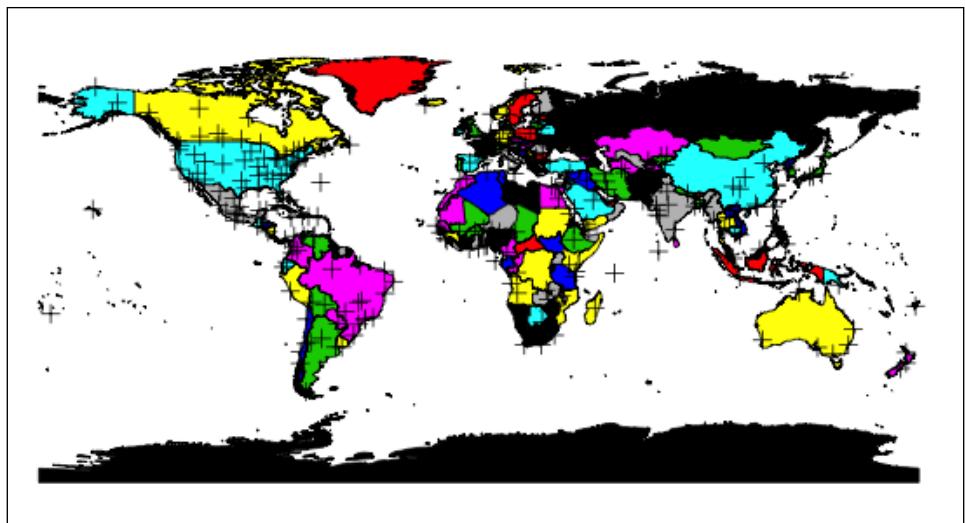
4. Plot the countries with color:

```
> # with color
> plot(countries_sp, col = countries_sp@data$admin)
```



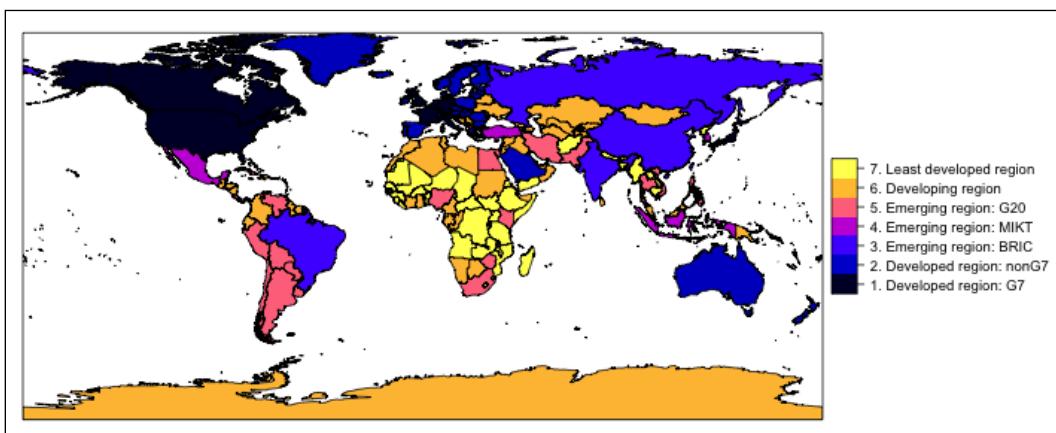
5. Add the airports. Do not close the previous plot:

```
> plot(airports_sp, add=TRUE)
```



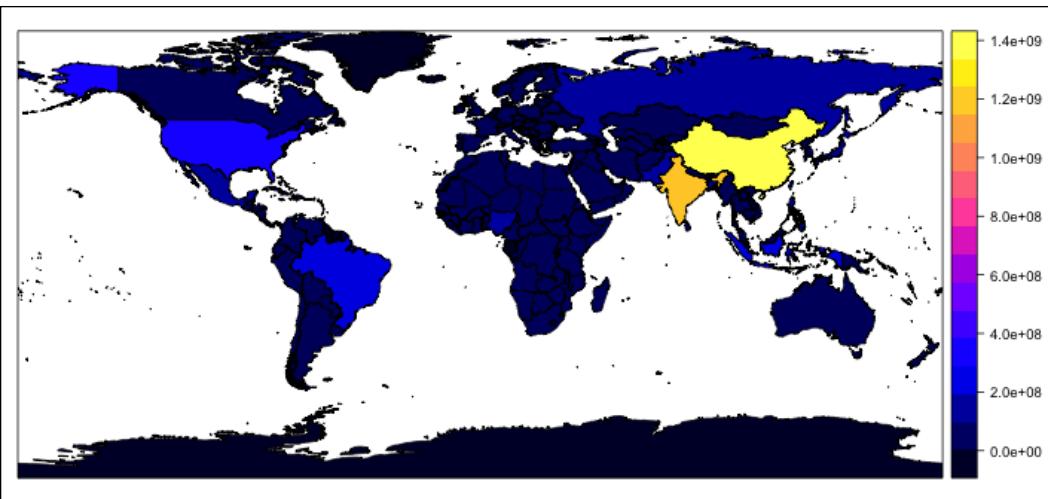
6. Plot the economic level (factor):

```
> 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:

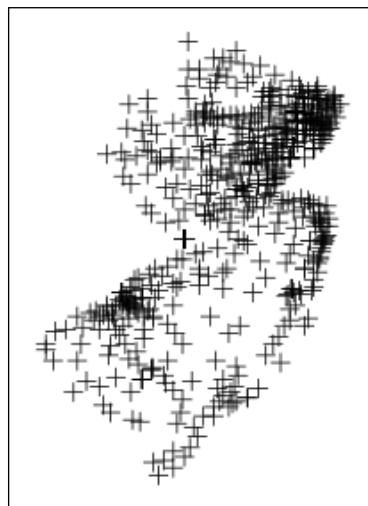
```
> coordinates(nj) <- c("Long", "Lat")
> class(nj)

[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:

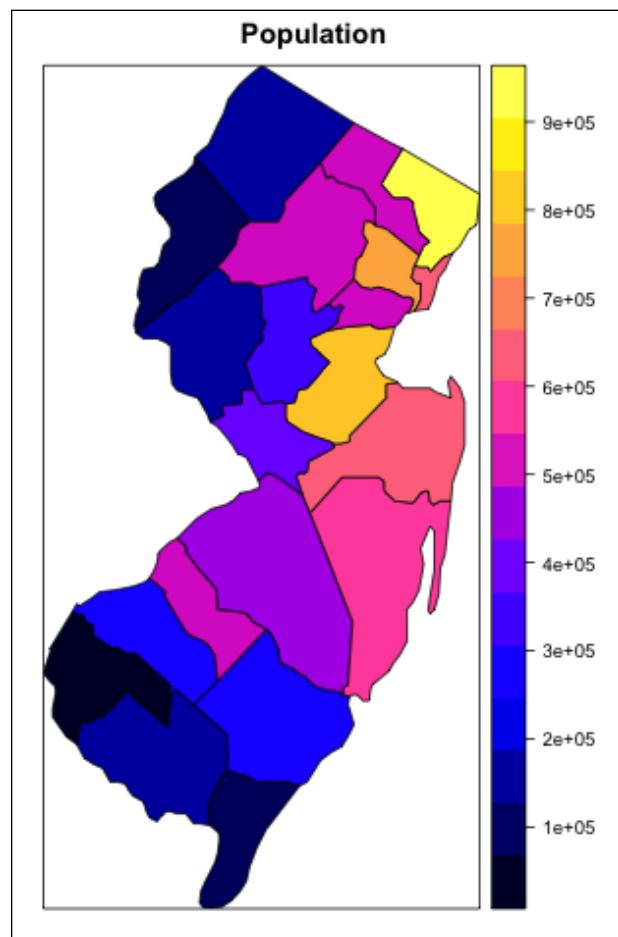
```
> nj.spdf <- SpatialPolygonsDataFrame(nj.sp, nj.dat)
> class(nj.spdf)
```

```
[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
```

8. Plot the map:

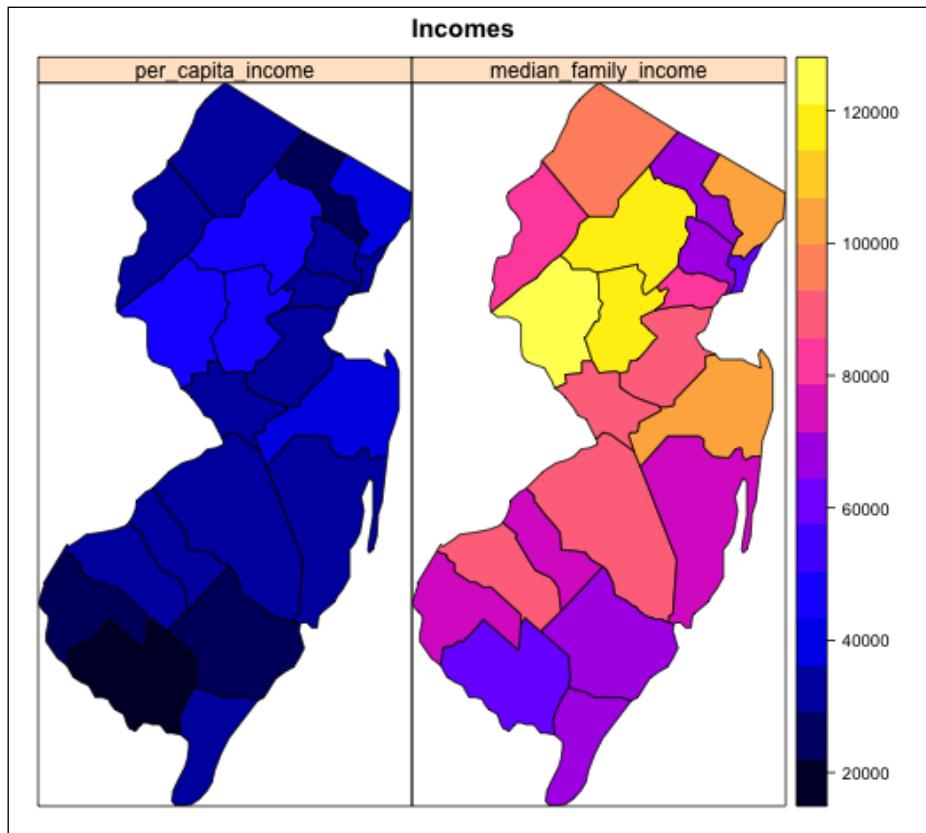
```
> # plain plot of the object
> plot(nj.spdf)
```

```
> # 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
```

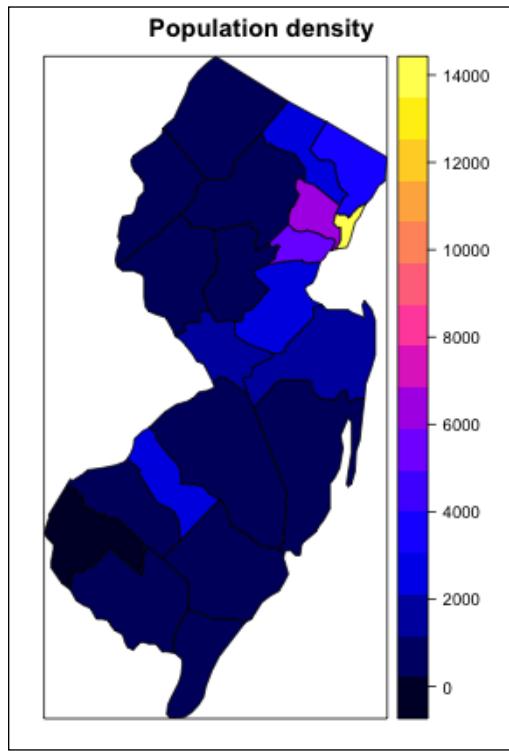
3. Add a new variable to `nj.spdf`:

```
> 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.

# 13

## 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 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;@1ff4689e"

> .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.

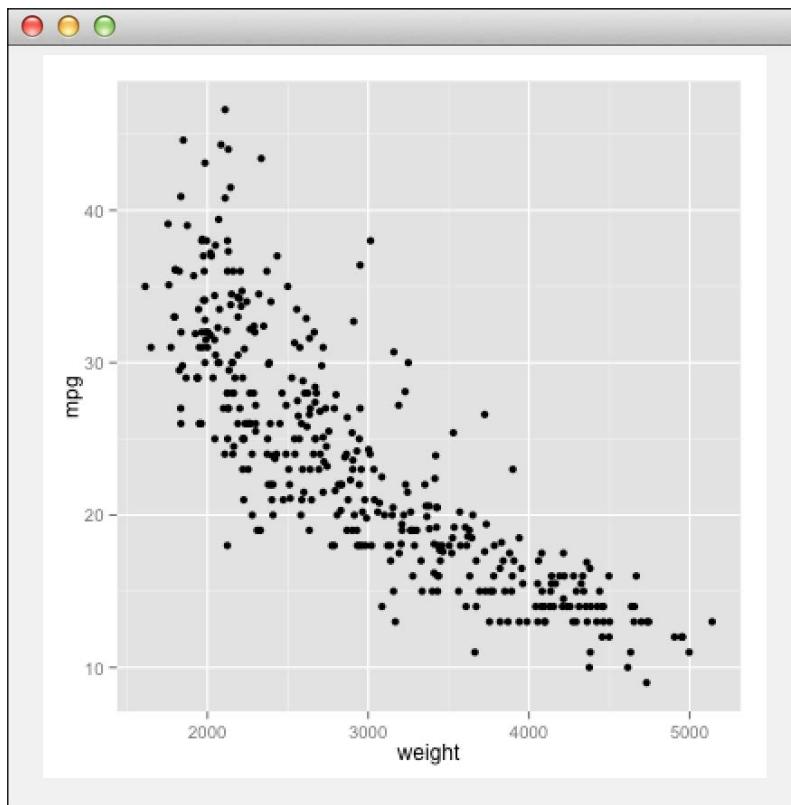
Rserve can run locally or on a remote server accessible by multiple clients. To access the remote Rserve server, provide the hostname or the IP address of the server while creating the RConnection.

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 RServer 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 `javademos` directory—be sure to pass your own R working directory in place of the last part of the command:

```
java -cp ../../lib/* javademos.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/* javademos.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$kmmpg <- auto$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)
```

# Module 2

## R Data Visualization Cookbook

*Over 80 recipes to analyze data and create stunning visualizations with R*

# 1

# Basic and Interactive Plots

In this chapter, we will cover the following recipes:

- ▶ Introducing a scatter plot
- ▶ Scatter plots with texts, labels, and lines
- ▶ Connecting points in a scatter plot
- ▶ Generating an interactive scatter plot
- ▶ A simple bar plot
- ▶ An interactive bar plot
- ▶ A simple line plot
- ▶ Line plot to tell an effective story
- ▶ Generating an interactive Gantt/timeline chart in R
- ▶ Merging histograms
- ▶ Making an interactive bubble plot
- ▶ Constructing a waterfall plot in R

# Introduction

The main motivation behind this chapter is to introduce the basics of plotting in R and an element of interactivity via the `googleVis` package. The basic plots are important as many packages developed in R use basic plot arguments and hence understanding them creates a good foundation for new R users. We will start by exploring the scatter plots in R, which are the most basic plots for exploratory data analysis, and then delve into interactive plots. Every section will start with an introduction to basic R plots and we will build interactive plots thereafter. We will utilize the power of R analytics and implement them using the `googleVis` package to introduce the element of interactivity.

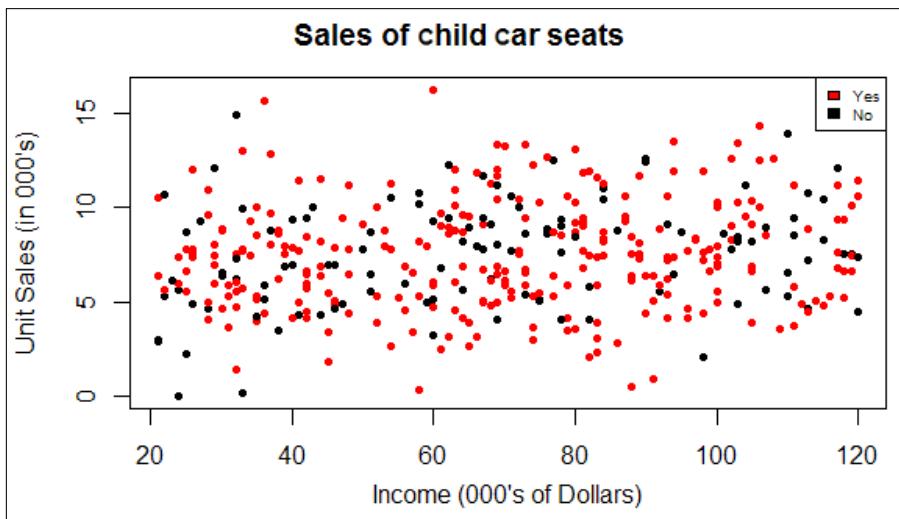
The `googleVis` package is developed by Google and it uses the Google Chart API to create interactive plots. There are a range of plots available with the `googleVis` package and this provides us with an advantage to plot the same data on various plots and select the one that delivers an effective message. The package undergoes regular updates and releases, and new charts are implemented with every release.

The readers should note that there are other alternatives available to create interactive plots in R, but it is not possible to explore all of them and hence I have selected `googleVis` to display interactive elements in a chart. I have selected these purely based on my experience with interactivity in plots. The other good interactive package is offered by `GGobi`.

This chapter is broken down into six major parts. The first part introduces the basics of plotting in R using scatter plots as an example and also introduces the users to interactivity using the `iPlots` package. The second part introduces bar plot functionality in R and further introduces the `googleVis` package to create an interactive bar plot. The third part delves into line plots and how we can make them more meaningful by simply making use of the options available in the line plot functionality in the `googleVis` package. The fourth section of the book discusses interactive histograms. We conclude the chapter by introducing interactive bubble plots and waterfall plots in parts five and six, respectively.

## Introducing a scatter plot

Scatter plots are used primarily to conduct a quick analysis of the relationships among different variables in our data. It is simply plotting points on the x-axis and y-axis. Scatter plots help us detect whether two variables have a positive, negative, or no relationship. In this recipe, we will study the basics of plotting in R using scatter plots. The following screenshot is an example of a scatter plot:



## Getting ready

For implementing the basic scatter plot in R, we would use Carseats data available with the ISLR package in R.

## How to do it...

We will also start this recipe by installing necessary packages using the `install.packages()` function and loading the same in R using the `library()` function:

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

Next, we need to load the data in R. Almost all R packages come with preloaded data and hence we can load the data only after we load the library in R. We can attach the data in R using the `attach()` function. We can view the entire list of datasets along with their respective libraries in R by typing `data()` in the R console window. The `attach()` function attaches the data to our R session. This allows us to access different variables of a database:

```
attach(Carseats)
```

Once we attach the data, it's a good practice to view the data using `head(Carseats)`. The `head()` function will display the first six entries of the dataset and will allow us to know the exact column headings of the data:

```
head(Carseats)
```

The data can be plotted in R by calling the `plot()` function. The `plot()` function in R comes with a variety of options and the best way to know all the options is by simply typing `?plot()` in the R console window:

```
plot(Income, Sales,col = c(Urban),pch = 20, main ="sales of Child
Car Seats", xlab = "Income (000's of Dollars)",
ylab ="Unit Sales (in 000's)")
```

This particular plot requires us to plot the legends as the points have two different color schemes. In R, we can add a legend using the `legend()` function:

```
legend("topright",cex = 0.6, fill = c("red","black"),
legend = c("Yes","No"))
```

## How it works...

Readers who are new to R should definitely read the recipe *Installing packages and getting help in R* in Chapter, *A Simple Guide to R*. The `install.packages()` and `library()` functions are used in most of the recipes in this book.

The `attach()` function is a nice way to reference the data as this allows us to avoid typing the `$` notation. The `$` notation is another way to reference columns in data and is discussed in the next recipe. Once we attach the data, it's a good practice to view the data using `head(Carseats)`. The `head()` function has data as its first argument. To view fewer number of lines in the R console window, we can also type `head(Carseats, 3)`. The `tail(Carseats)` function will display data entries from the bottom of the dataset.

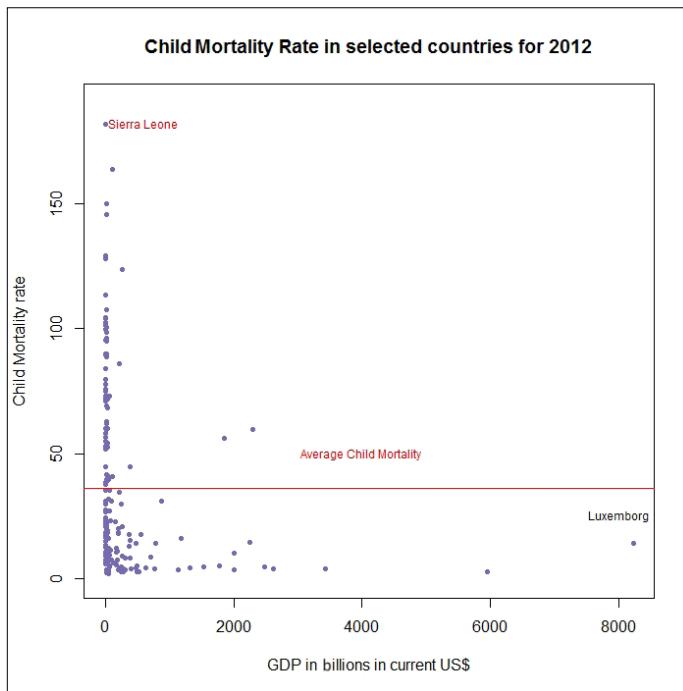
The data can be plotted in R by calling the `plot()` function. The first two arguments in the `plot()` function refer to the data to be displayed on the x-axis (`Income`) and y-axis (`Sales`). The `col` argument allows us to assign color to our data points. In this case, we would like to use a qualitative data column (`Urban`) to color our points. The default color in R is black but we can change this using the `col = "blue"` argument. Please refer to the code file to learn about various other options. The `pch = 20` argument allows us to plot symbols; the value 20 will plot filled circles. To view all the available `pch` values, please type `?par` or `?points` in the R console window. We can also label the heading of the plot using the `main = "Sales"` argument. The `xlab` and `ylab` arguments are used to label the x and y axes in R.

To display a legend is necessary for this scatter plot as we would like to differentiate between sales in urban and rural areas. The first argument in the `legend()` function corresponds to the position of the legend. The `cex` argument is used to size the text, the default value for `cex` is 1. The `fill` argument fills the boxes with the specified colors and the `legend` argument applies the labels to each of the boxes.

# Scatter plots with texts, labels, and lines

In the previous recipe, we studied how to construct a very basic scatter plot. In order for the plot to deliver a strong message, we need to add elements such as text, labels, and lines. The main objective of a visualization is to grab the attention of its audience and make the optimal use of the data available. The audience should be able to get most of its information from the visualization itself.

The following screenshot plots the child mortality rate in selected countries. The story we would like to share with the readers is the relationship between the child mortality rate and Gross Domestic Product (GDP) of a country. We can improve on our understanding of these relationships if the readers can compare extreme scenarios or compare a specific country with a benchmark (average child mortality rate).



## How to do it...

In the previous recipe, we used a dataset from the `ISLR` package. But what if we would like to import our own data in R? We can set a working directory in R using the `setwd()` function. This is a necessary step as R will always search for the datafile in the active/current directory. The `setwd()` function allows us to set our working directory:

```
setwd("D:/book/scatter_Area/data")
```

The `read.csv()` function is used to import the data in R:

```
child = read.csv("chlmort.csv", header = TRUE, sep = ",")
```

The `summary()` function is used to get a general idea about the distribution of variables in our data. The `head()` function allows us to view the actual data:

```
summary(child)
head(child)
```

The following code is used to plot the skeleton of our scatter plot. Some of the arguments may look very familiar to you from the previous recipe. We have used `child$gdp_bil` and `child$child` instead of `gdp_bil` and `child`. This change was necessary as we did not use the `attach()` command:

```
plot(child$gdp_bil, child$child, pch = 20, col = "#756bb1",
xlim=c(0,max(child$gdp_bil)), ylim = c(0,190), xlab = "GDP in
Billions in current US$", ylab ="Child Mortality rate", main =
"child Mortality Rate in selected countries for 2012")
```

In order to plot a horizontal or a vertical line in R, we use the `abline()` function. The `h = ()` argument will draw a horizontal line. The value `36.18` is the world average of child mortality rate and to add this makes it easier to compare the data across countries. The `lwd = 1` argument increases the width of the line and `col = "red"` adds color to the line:

```
abline(h = (36.18), lwd = 1, col = "red")
```

To generate an effective presentation, we add labels to extreme points on our plot. We can immediately observe that GDP and child mortality rate share a negative relationship. We can go a step further and make the plot easy to interpret if we add text, using the `text()` function, to extreme observations in our data:

```
text(8000,25,labels = c("Luxemborg"), cex = 0.75)
text(600,182,labels= c("Sierra Leone"), col = "red", cex = 0.75)
text(4000, 50,labels = c("Average Child Mortality"),
col = "red", cex = 0.75)
```

## How it works...

To import data in R, we need to direct R to the folder where the data is stored. We can either type the command `setwd()` to let R know where to find the file, or we can navigate to the folder via **Session | Set Working Directory | Choose Directory**. Readers can learn more about various methods to import data in R under the recipe *Importing Data in R, Chapter, A Simple Guide to R*.

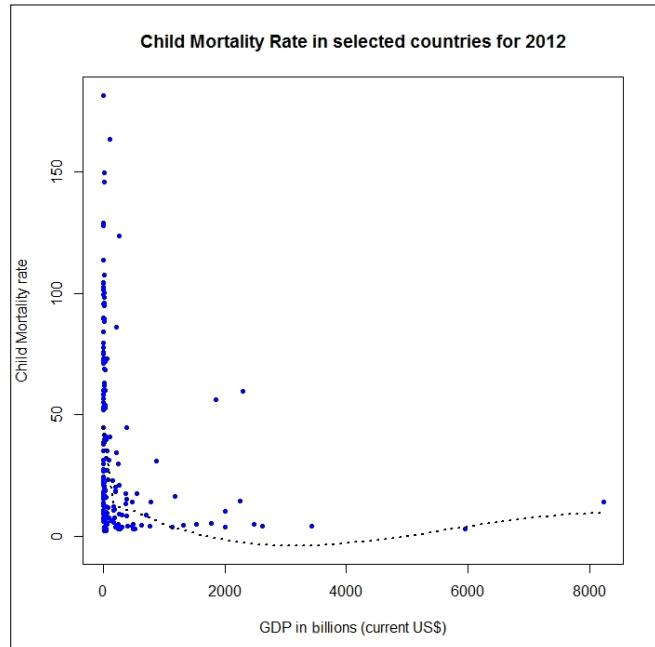
Under the `plot()` function, we have introduced the `$` notation. The name before the `$` sign corresponds to the data and the name after the `$` sign refers to the column (`child$gdp_bill`). We have used `ylim()` to specify the y limit for the plot. We can also assign a hex value to the color instead of simply typing in the name of the color using `col = "#756bb1"`. All the remaining arguments are discussed in detail in the previous recipe.

The `text()` function uses three arguments: x-axis, y-axis, and labels. The x-axis and y-axis arguments inform R as to where exactly to place the labels. The `labels = c()` argument is the actual label to be placed. The `cex = 0.75` argument is used to state the size of the fonts. We can learn about various other text arguments available under R using the command `?text()`.

## There's more...

At times, we would like to add a trend line to our plot. We could achieve this in a variety of ways by fitting a regression line or using the `scatter.smooth()` function that allows users to plot a smooth line using **Locally Weighted Scatterplot Smoothing (LOESS)**. We will use LOESS to study the trend in our data. The idea behind LOESS is to fit a weighted polynomial with more weight to points near the points whose value is being estimated and less weight to points further away. The method was initially proposed by Cleveland. We will avoid going into the mathematical details of how LOESS is calculated, and instead we will assume that R will correctly apply this to our data.

The dashed line in the following screenshot is not the usual regression line but a trend line fitted using LOESS methodology:



Displaying a trend line is not very difficult. The following code would first import the data and then plot the trend line using the LOESS method and using the `scatter.smooth()` function:

```
scatter.smooth(child$gdp_bill, child$child, pch = 20, lwd = 0.75,
 col = "Blue", lpars = list(lty = 3, col = "black", lwd = 2),
 xlab = "GDP in Billions in current US$", ylab = "Child Mortality
 rate", main = "child Mortality Rate in selected countries
 for 2012")
```

We use the `lpars()` function to beautify the trend line. The attributes passed in the `lpars()` function are the same attributes passed in the previous two plots. Readers can learn more about the `scatter.smooth()` function by typing `?scatter.smooth` in the R console window.

## See also

- ▶ *A Long Road Ahead in Regaining Lost Jobs* is a New York Times visualization that uses a text to provide additional information to its audience. It can be accessed at [http://www.nytimes.com/interactive/2010/10/13/business/economy/economy\\_graphic.html?\\_r=0](http://www.nytimes.com/interactive/2010/10/13/business/economy/economy_graphic.html?_r=0).
- ▶ *Narrative Visualization: Telling Stories with Data*, Edward Segel and Jeffrey Heer, 2010, can be accessed at <http://vis.stanford.edu/files/2010-Narrative-InfoVis.pdf>.
- ▶ Nathan Yau has explained the `smooth.scatter()` function and LOESS on his blog, and can be viewed at <http://flowingdata.com/2010/03/29/how-to-make-a-scatterplot-with-a-smooth-fitted-line/>.

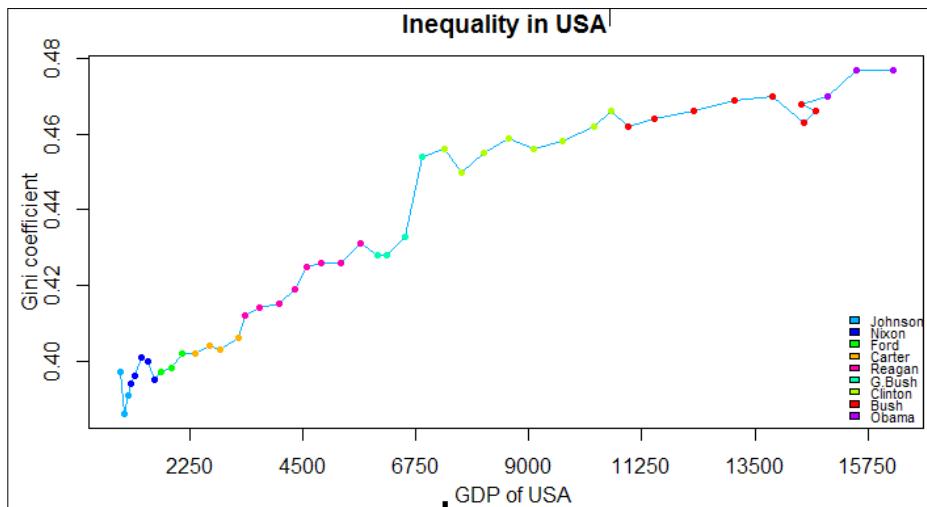
## Connecting points in a scatter plot

The primary objective of this recipe is to understand how we can connect points in a scatter plot. The plot is inspired by Alberto Cairo infographic regarding the Gini coefficient, and the GDP data under various president's tenure in Brazil and connected points based on these three variables. In this recipe, we will apply the same concept to the USA economy.



For all the data visualizations and references used in recipes in this book, please refer to the *See also* section in each recipe





## How to do it...

We will start to import the data in R. The dataset comprises of the Gini coefficient, the GDP data of the USA and USA presidents. The Gini coefficient is used as a measure of inequality in a country:

```
data = read.csv("ginivsgdp1.csv", header = TRUE)
```

The plot can be generated using the `plot()` function:

```
plot(income$gdp_ann, income$Gini, pch = 20, col = c(data$Presidents), type = "o", xlab = "GDP of USA", ylab = "Gini coefficient", main = "Inequality in USA", xaxp = c(0,18000,8))
```

Since we use `col` to distinguish between the periods of various presidents in the USA, we require legends in the plot. The legend is added to our plot using the `legend()` function:

```
legend("bottomright", fill = c(6,7,4,2,9,5,3,1,8), legend = c("Johnson", "Nixon", "Ford", "Carter", "Reagan", "G.Bush", "Clinton", "Bush", "Obama"), bty = "n", cex=0.7)
```

## How it works...

Most of the arguments used in the `plot()` function have been discussed in prior recipes of this chapter. The `type = "o"` argument connects lines in a plot. Readers curious to know more about the various types of option should type `?plot` in the R console window.

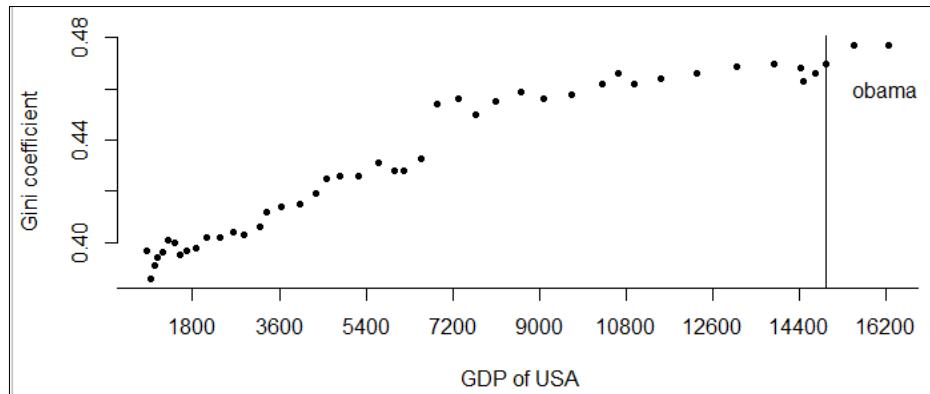
The important point to note is that R used a qualitative variable such as presidents to plot points of different colors. We would require the color names to pass as an argument under the `fill`. R converts the presidents' names into numeric value and uses it to color each point. We can view these numeric values by typing the following lines:

```
cols = as.numeric(income$Presidents)
cols
```

We can simply use these numeric values and pass it as a vector under the `legend()` function. The first argument in the `legend()` function is the position of the legend. The third argument corresponds to the labels. We suppress drawing a box around the label using the `bty = "n"` argument. The `cex` argument allows us to size the labels in R.

## There's more...

We could also add texts and lines to our plot as shown in the following screenshot:



We have implemented the same code as mentioned in the *How to do it...* section of this recipe. But instead of applying different colors, we can also add a line and text to our chart:

```
abline(v = 14958, lwd = 1.5)
text(16200, 0.46, "obama")
```

The `text()` functionality in R will allow us to add a text. The first and second arguments under `text()` represent the x and y coordinates of the plot. The third argument is the actual label to be applied. The `abline()` function is used to apply a vertical line to our plot. To plot labels on all the points, we can use the following code:

```
plot(income$gdp_ann, income$Gini, pch = 20, col = "Black",
 xlab = "GDP of USA", ylab = "Gini coefficient",
 xaxp = c(0, 18000, 10), bty = "n")
text(income$gdp_ann, income$Gini, income$years, cex = 0.7,
 pos = 2, offset = 0)
```

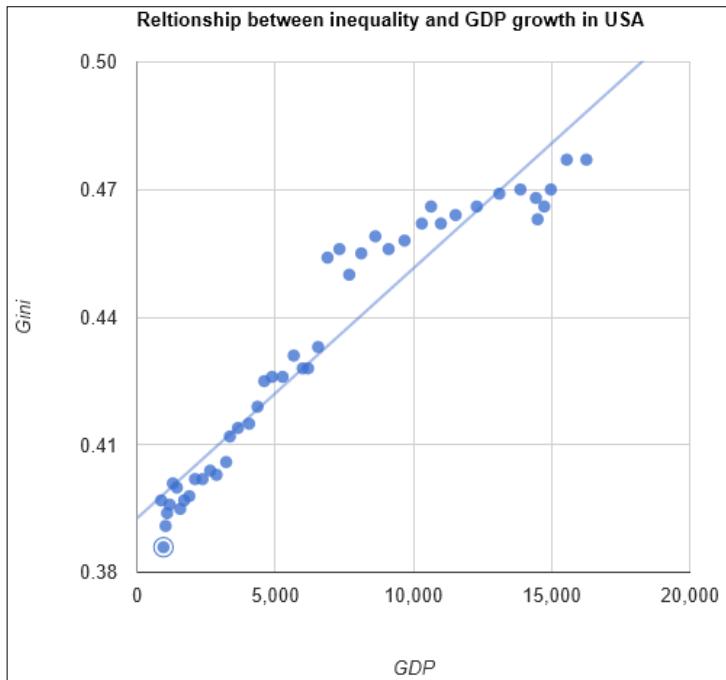
Since we require to plot all the years, we pass `income$years` as our argument for labels. The `pos` argument is used to adjust the position of the label around the point. Readers may observe overlapping labels and they can try to fix this by setting `pos` and `offset`. I would suggest the readers to type `?text` in the R console window to learn more about the `text()` function.

## See also

- ▶ Alberto Cairo visualization on inequality and GDP in Brazil can be accessed at <http://www.thefunctionalart.com/2012/09/in-praise-of-connected-scatter-plots.html>

# Generating an interactive scatter plot

In the previous recipe, we studied how multivariate data can be displayed on a scatter plot. We used color as a visual cue to display information related to different presidents in the USA and how the economy performed. In this recipe, we will build on the same idea and introduce interactive scatter plots. The limitation of a static plot is that they are hard to interpret if the points overlap, and if the gridlines are not present the data may be hard to decipher as well. Interactive plots help us to overcome this limitation. In this recipe, we will plot a simple interactive scatter plot with a trend line as shown in the following screenshot:



# Getting ready

We would plot an interactive scatter plot using the `googleVis` package in R.

## How to do it...

To generate an interactive scatter plot, we will install and load the `googleVis` package in R. We can import the data in R using the `read.csv()` function:

```
install.packages("googleVis")
library(googleVis)
income = read.csv("ginivsgdp1.csv", header = TRUE)
```

By default, the `googleVis` package will use the first column as the x variable. Since our first column is not GDP, we will use the GDP data and Gini data from the imported CSV file to construct a new data frame in R:

```
scater = data.frame(gdp = c(income$gdp_ann), gini= c(income$Gini))
```

Next, we can generate an interactive scatter plot in R. Note that the `googleVis` package will display the scatter plot in a new browser window only when the `plot()` function is executed:

```
scaterp4 = gvisScatterChart(scater, option= list(width = 650,
height = 600, legend = "none",title = "Relationship between
Inequality and GDP growth in USA",
hAxis = "{title : 'GDP'}",
vAxis = "{title : 'Gini'}",
dataOpacity = 0.8,
trendlines="{0:{type : 'linear', visibleInLegend: true,
showR2: true}}"))
plot(scaterp4)
```

## How it works...

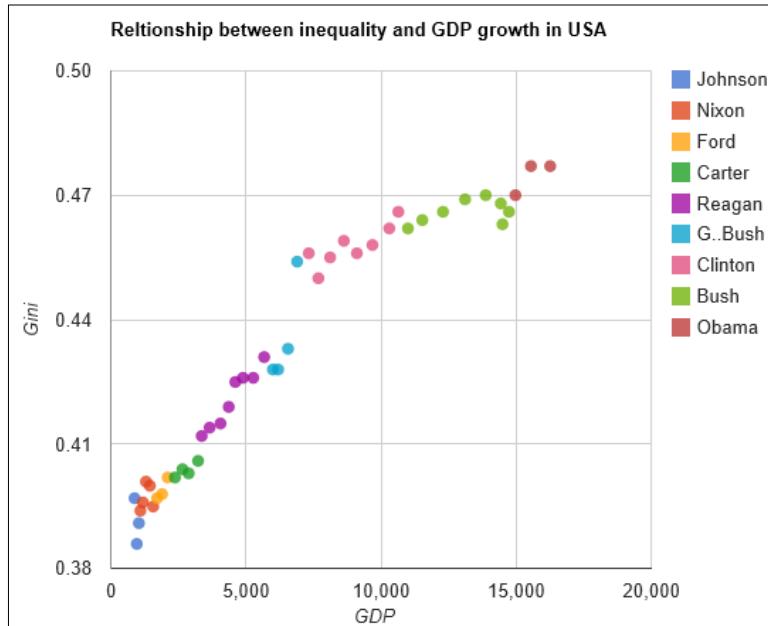
The `data.frame()` function is discussed in the recipe *Data frames in R* in Chapter, *A Simple Guide to R*. Readers new to R can learn about the `data.frame()` function by typing `?data.frame` in the R console window.

The first argument in the `gvisScatterChart()` function is our data frame. We can have more than one column in our data frame but the x-axis will be assigned to the first column. The `googleVis` package comes with some very useful options that allow us to add labels to the x and y axes, and add a title and opacity to our scatter plot. We will discuss these in detail at a later point in this chapter. The options are added to a plot using the `option()` function.

The `trendlines` argument adds a linear trend line to our scatter plot. Note the use of `0` in the `trendline` argument that corresponds to the series. The `visibleInLegend` and `showR2` arguments add the estimates and coefficient of determination to the plots legend section.

## There's more...

In this section, we will learn to plot multiple y-axis values on the same scatter plot. This is shown in the following screenshot. Note that we have processed the data and stored it as a new CSV file.



The code used to generate this plot is exactly the same as the one discussed under the *How to do it...* section of this recipe. But we have altered the data file and stored it as a new CSV file. To learn more about various options available, please refer to the `googleVis` developer website.

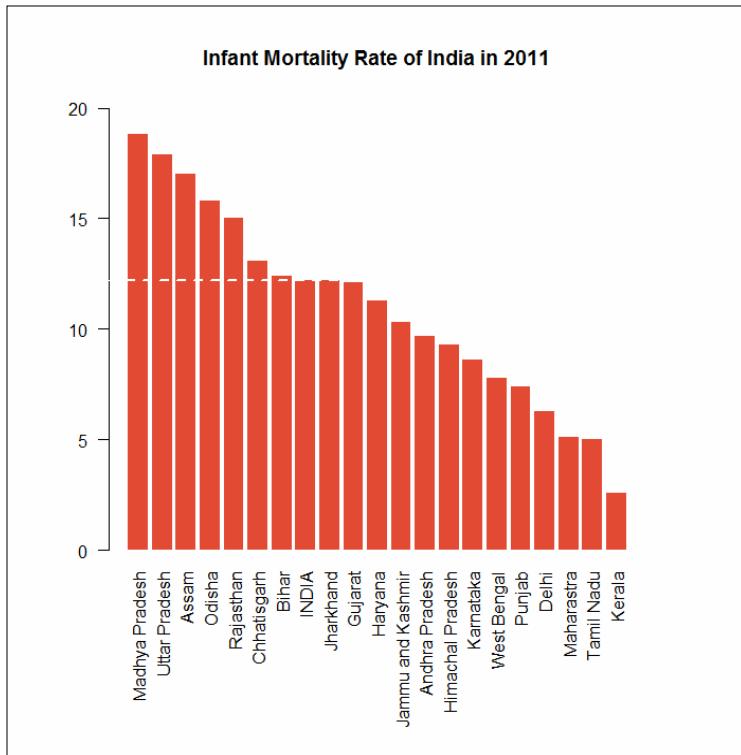
## See also

- ▶ The `googleVis` developer website can be accessed at [https://developers.google.com/chart/interactive/docs/gallery/scatterchart#Configuration\\_Options](https://developers.google.com/chart/interactive/docs/gallery/scatterchart#Configuration_Options)

# A simple bar plot

A bar plot can often be confused with histograms (studied later in this chapter). Histograms are used to study the distribution of data whereas bar plots are used to study categorical data. Both the plots may look similar to the naked eye but the main difference is that the width of a bar plot is not of significance, whereas in histograms the width of the bars signifies the frequency of data.

In this recipe, I have made use of the infant mortality rate in India. The data is made available by the Government of India. The main objective is to study the basics of a bar plot in R as shown in the following screenshot:



## How to do it...

We start the recipe by importing our data in R using the `read.csv()` function. R will search for the data under the current directory, and hence we use the `setwd()` function to set our working directory:

```
setwd("D:/book/scatter_Area/chapter2")
data = read.csv("infant.csv", header = TRUE)
```

Once we import the data, we would like to process the data by ordering it. We order the data using the `order()` function in R. We would like R to order the column `Total2011` in a decreasing order:

```
data = data[order(data$Total2011, decreasing = TRUE),]
```

We use the `ifelse()` function to create a new column. We would utilize this new column to add different colors to bars in our plot. We could also write a loop in R to do this task but we will keep this for later. The `ifelse()` function is quick and easy. We instruct R to assign `yes` if values in the column `Total2011` are more than 12.2 and `no` otherwise. The 12.2 value is not randomly chosen but is the average infant mortality rate of India:

```
new = ifelse(data$Total2011>12.2, "yes", "no")
```

Next, we would like to join the vector of yes and no to our original dataset. In R, we can join columns using the `cbind()` function. Rows can be combined using `rbind()`:

```
data = cbind(data,new)
```

When we initially plot the bar plot, we observe that we need more space at the bottom of the plot. We adjust the margins of a plot in R by passing the `mar()` argument within the `par()` function. The `mar()` function uses four arguments: bottom, left, top, and right spacing:

```
par(mar = c(10,5,5,5))
```

Next, we generate a bar plot in R using the `barplot()` function. The `abline()` function is used to add a horizontal line on the bar plot:

```
barplot(data$Total2011, las = 2, names.arg= data$India,width = 0.80, border = NA,ylim=c(0,20), col = "#e34a33", main = "Infant Mortality Rate of India in 2011")
abline(h = 12.2, lwd =2, col = "white", lty =2)
```

## How it works...

The `order()` function uses permutation to rearrange (decreasing or increasing) the rows based on the variable. We would like to plot the bars from highest to lowest, and hence we require to arrange the data. The `ifelse()` function is used to generate a new column. We would use this column under the *There's more...* section of this recipe. The first argument under the `ifelse()` function is the logical test to be performed. The second argument is the value to be assigned if the test is true, and the third argument is the value to be assigned if the logical test fails.

The first argument in the `barplot()` function defines the height of the bars and `horiz = TRUE` (not used in our code) instructs R to plot the bars horizontally. The default setting in R will plot the bars vertically. The `names.arg` argument is used to label the bars. We also specify `border = NA` to remove the borders and `las = 2` is specified to apply the direction to our labels. Try replacing the `las` values with 1,2,3, or 4 and observe how the orientation of our labels change..

The first argument in the `abline()` function assigns the position where the line is drawn, that is, vertical or horizontal. The `lwd`, `lty`, and `col` arguments are used to define the width, line type, and color of the line.

## There's more...

While plotting a bar plot, it's a good practice to order the data in ascending or descending order. An unordered bar plot does not convey the right message and the plot is hard to read when there are more bars involved. When we observe a plot, we are interested to get the most information out, and ordering the data is the first step toward achieving this objective.

We have not specified how we can use the `ifelse()` and `cbind()` functions in the plot. If we would like to color the plot with different colors to let the readers know which states have high infant mortality above the country level, we can do this by pasting `col = (data$new)` in place of `col = "#e34a33"`.

## See also

- ▶ New York Times has a very interesting implementation of an interactive bar chart and can be accessed at [http://www.nytimes.com/interactive/2007/09/28/business/20070930\\_SAFETY\\_GRAPHIC.html](http://www.nytimes.com/interactive/2007/09/28/business/20070930_SAFETY_GRAPHIC.html)

## An interactive bar plot

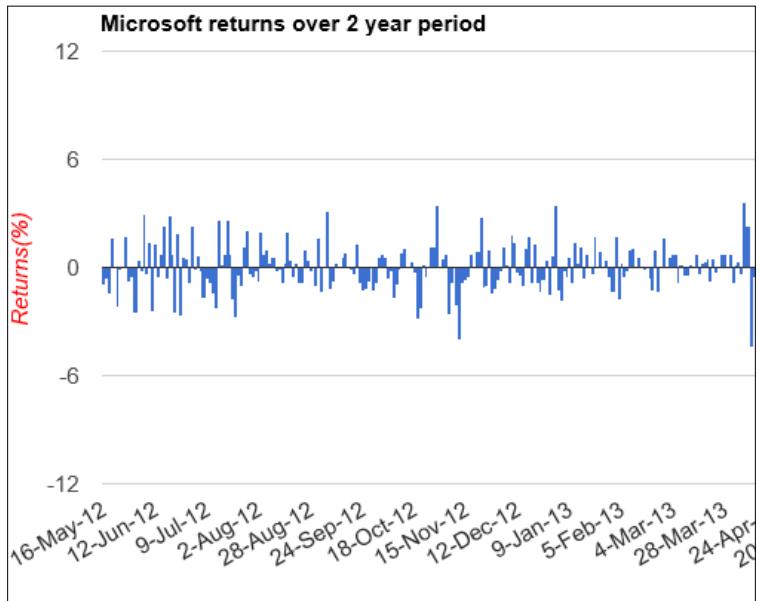
We would like to make the bar plot interactive. The advantage of using the Google Chart API in R is the flexibility it provides in making interactive plots. The `googleVis` package allows us to skip the step to export a plot from R to an illustrator and we can make presentable plots right out of R.

The bar plot functionality in R comes with various options and it is not possible to demonstrate all of the options in this recipe. We will try to explore plot options that are specific to bar plots.

In this recipe, we will learn to plot the returns data of Microsoft over a 2-year period. The data for the exercise was downloaded using Google Finance. We have calculated 1-day returns for Microsoft in MS Excel and imported the CSV in R:

```
Return = ((Pricet - Pricet-1)/pricet-1) *100
```

Readers should note that the following plot is only a part of the actual chart:



## Getting ready

In order to plot a bar plot, we would install the `googleVis` package.

## How to do it...

We would start the recipe by installing the `googleVis` package and loading the same in R:

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

When R loads a library, it loads several messages: we can suppress these messages using `suppressPackageStartupMessages (library (googleVis))`. We can now import our data using the `read.csv()` function. The data file comprises of three variables: date, daily Microsoft prices, and daily returns:

```
stock = read.csv("spq.csv", header = TRUE)
```

We generate an interactive bar plot using the `gvisBarChart()` function:

```
barpt = gvisBarChart(stock, xvar = "Date", yvar = c("Returns"),
 options = list(orientation = "horizontal", width = 1400,
 height = 500, title = "Microsoft returns over 2 year period",
 legend = "none",
 hAxis = "{title : 'Time Period', titleTextStyle :{color: 'red'} }",
 vAxis = "{title : 'Returns(%)', ticks : [-12,-6,0,6,
 12], titleTextStyle :{color: 'red'} }",
 bar = "{groupWidth: '100%'})")
```

The plot is displayed in a new browser window when we execute the `plot()` function:

```
plot(barpt)
```

## How it works...

The `googleVis` Package will generate a plot in a new browser window and requires an Internet connectivity to generate the same in R. The `googleVis` package is a simple communication medium between the Google Charts API and R.

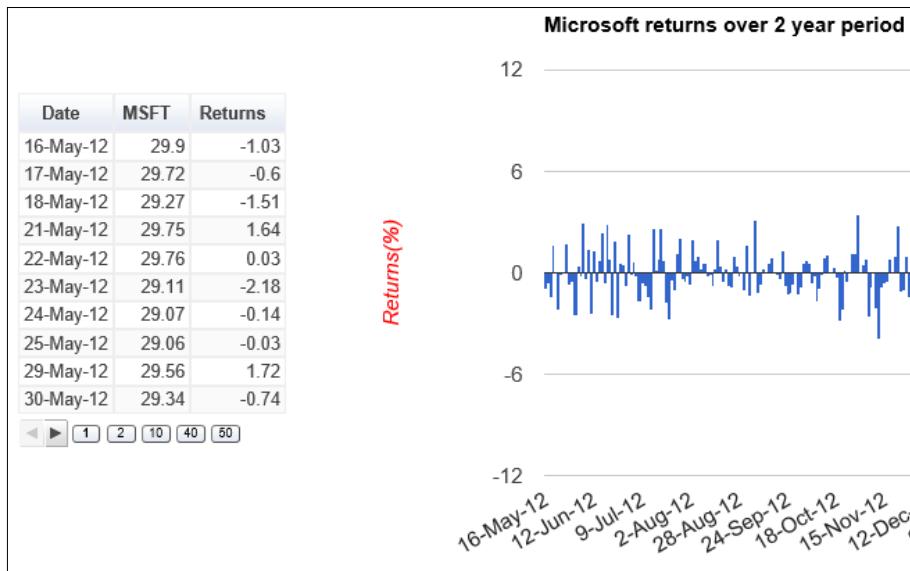
We have defined our `gvisBarChart()` function as `barpt`; we need this in R as all the `googleVis` functions will generate a list that contains the HTML code and reference to a JavaScript function. If you omit `barpt =`, R will display the code generated in your command window.

The first argument under the `gvisBarChart()` function is the `data` argument; we have stored the data in R as a data frame called **stock**. The second and third arguments are the column names of the data to be displayed on the x-axis and y-axis. The `options` argument is used to define a list of options. All the available options related to the bar plot and their descriptions are available on the `googleVis` developer website. Since we are plotting stock returns of Microsoft (only one series), we can avoid legends. The default plot will include legends—this can be overwritten using `legend = "none"`.

To add a title to our plot, we use the `title` attribute. We label our axes using the `vAxis` and `hAxis` attributes. Note the use of `{ }` and `[ ]` within `vAxis` and `hAxis`. We make use of `{ }` to group all the elements related to `hAxis` and `vAxis` instead of typing `vAxis.title` or `vAxis.title.TextStyle`. If readers are familiar with CSS or HTML, this code would be very easy to interpret. We have used the `group.width` attribute and set it to 100 percent in order to eliminate the spacing between bars. Finally, we call the `plot()` function to display our visualization.

## There's more...

In the previous recipe, we constructed a bar plot. The `googleVis` package also allows us to create a table and merge the same with a bar chart. The following screenshot is only part of the plot:



The combined table and bar chart are generated in three different steps. We will first generate a bar chart. The code is exactly the same as the one discussed under the *How to do it...* section of this recipe. We will then generate a table using the following lines of code:

```
table <- gvisTable(stock, options=list(page='enable',
 height='automatic',
 width='automatic'),
 formats = list(Returns ='#.##'))
```

In the final step, we will merge the two chart objects (`barpt` and `table`) using the `gvisMerge()` function:

```
comb = gvisMerge(table,barpt, horizontal = TRUE)
plot(comb)
```

We can display the merged visualization `comb` using the `plot()` function.

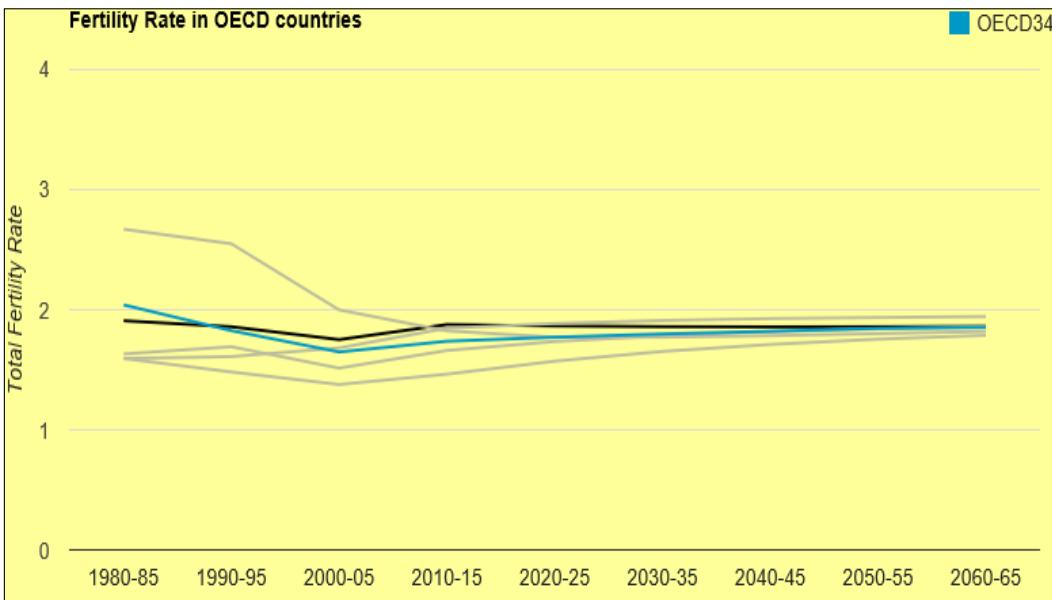
The readers can learn more about exporting the visualization in the `googleVis` package manual. It is also possible to integrate the `googleVis` package plots with `shiny`. Please refer to the recipe *Creating a very simple shiny app in R* in Chapter, *Creating Applications in R*.

## See also

- ▶ The Google Chart API developer website can be accessed at [https://developers.google.com/chart/interactive/docs/gallery/barchart#Configuration\\_Options](https://developers.google.com/chart/interactive/docs/gallery/barchart#Configuration_Options)
- ▶ The googleVis package manual can be accessed at <http://cran.r-project.org/web/packages/googleVis/googleVis.pdf>
- ▶ A blog on integrating shiny with googleVis can be accessed at <http://www.magesblog.com/2013/02/first-steps-of-using-googlevis-on-shiny.html>

## A simple line plot

Line plots are simply lines connecting all the x and y dots. They are very easy to interpret and are widely used to display an upward or downward trend in data. In this recipe, we will use the googleVis package and create an interactive R line plot. We will learn how we can emphasize on certain variables in our data. The following line plot shows fertility rate:



## Getting ready

We will use the googleVis package to generate a line plot.

## How to do it...

In order to construct a line chart, we will install and load the `googleVis` package in R. We would also import the fertility data using the `read.csv()` function:

```
install.packages("googleVis")
library(googleVis)
frt = read.csv("fertility.csv", header = TRUE, sep = ",")
```

The fertility data is downloaded from the OECD website. We can construct our line object using the `gvisLineChart()` function:

```
gvisLineChart(frt, xvar = "Year",
yvar=c("Australia","Austria","Belgium","Canada","Chile","OECD34"),
options = list(width = 1100, height= 500, backgroundColor =
"#FFFF99",title ="Fertility Rate in OECD countries" ,
vAxis = "{title : 'Total Fertility
Rate',gridlines:{color:'#DEDECE',count : 4}, ticks :
[0,1,2,3,4]}",
series = "{0:{color:'black', visibleInLegend :false},
1:{color:'BDBD9D', visibleInLegend :false},
2:{color:'BDBD9D', visibleInLegend :false},
3:{color:'BDBD9D', visibleInLegend :false},
4:{color:'BDBD9D', visibleInLegend :false},
34:{color:'3333FF', visibleInLegend :true}}")
```

We can construct the visualization using the `plot()` function in R:

```
plot(line)
```

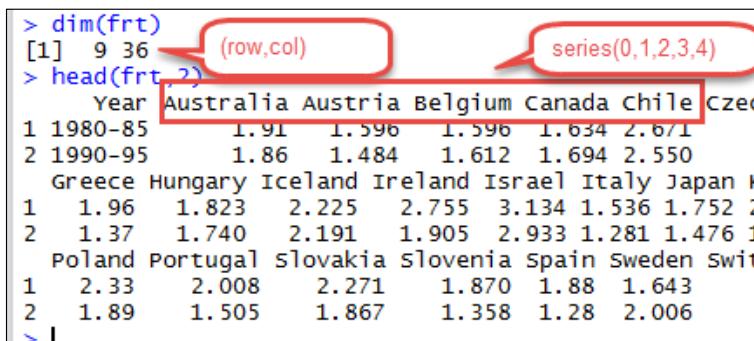
## How it works...

The first three arguments of the `gvisLineChart()` function are the data and the name of the columns to be plotted on the x-axis and y-axis. The `options` argument lists the chart API options to add and modify elements of a chart.

For the purpose of this recipe, we will use part of the dataset. Hence, while we assign the series to be plotted under `yvar = c()`, we will specify the column names that we would like to be plotted in our chart. Note that the series starts at 0, and hence Australia, which is the first column, is in fact series 0 and not 1.

For the purpose of this exercise, let's assume that we would like to demonstrate the mean fertility rate among all OECD economies to our audience. We can achieve this using `series { }` under `option = list()`. The `series` argument will allow us to specify or customize a specific series in our dataset. Under the `gvisLineChart()` function, we instruct the Google Chart API to color OECD series (series 34) and Australia (series 0) with a different color and also make the legend visible only for OECD and not the entire series.

It would be best to display all the legends but we use this to show the flexibility that comes with the Google Chart API. Finally, we can use the `plot()` function to plot the chart in a browser. The following screenshot displays a part of the data. The `dim()` function gives us a general idea about the dimensions of the fertility data:



```
> dim(frt)
[1] 9 36
> head(frt, 2)
 Year Australia Austria Belgium Canada Chile cze
1 1980-85 1.91 1.596 1.596 1.634 2.6/1
2 1990-95 1.86 1.484 1.612 1.694 2.550
 Greece Hungary Iceland Ireland Israel Italy Japan K
1 1.96 1.823 2.225 2.755 3.134 1.536 1.752 2
2 1.37 1.740 2.191 1.905 2.933 1.281 1.476 1
 Poland Portugal slovakia slovenia spain Sweden Swit
1 2.33 2.008 2.271 1.870 1.88 1.643
2 1.89 1.505 1.867 1.358 1.28 2.006
> |
```

The screenshot shows the R console with two annotations. A red box encloses the output of `dim(frt)`, with a callout pointing to it labeled '(row,col)'. Another red box encloses the output of `head(frt, 2)`, with a callout pointing to it labeled 'series(0,1,2,3,4)'.

New York Times Visualization often combines line plots with bar chart and pie charts. Readers should try constructing such visualization. We can use the `gvisMerge()` function to merge plots. The function allows merging of just two plots and hence the readers would have to use multiple `gvisMerge()` functions to create a very similar visualization. The same can also be constructed in R but we will lose the interactive element.

## See also

- ▶ The OECD website provides economic data related to OECD member countries. The data can be freely downloaded from the website <http://www.oecd.org/statistics/>.
- ▶ New York Times Visualization combines bar charts and line charts and can be accessed at [http://www.nytimes.com/imagepages/2009/10/16/business/20091017\\_CHARTS\\_GRAPHIC.html](http://www.nytimes.com/imagepages/2009/10/16/business/20091017_CHARTS_GRAPHIC.html).

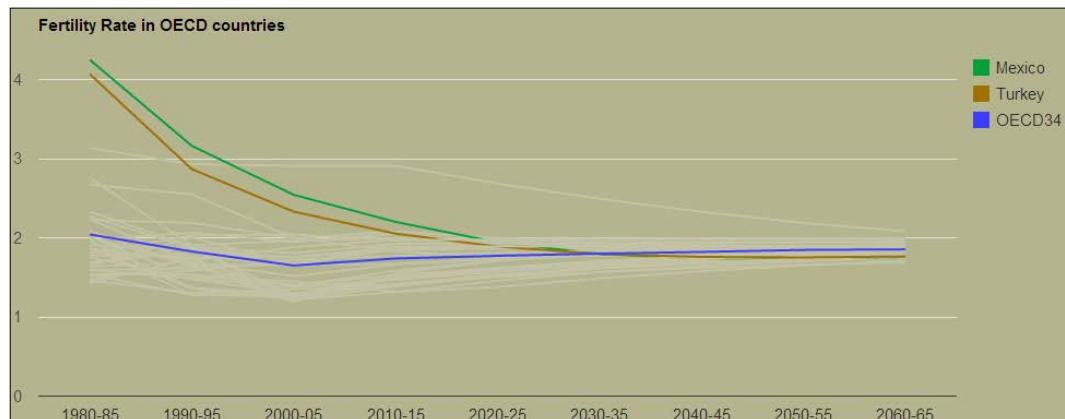
## Line plot to tell an effective story

In the previous recipe, we learned how to plot a very basic line plot and use some of the options. In this recipe, we will go a step further and make use of specific visual cues such as color and line width for easy interpretation.

Line charts are a great tool to visualize time series data. The fertility data is discrete but connecting points over time provides our audience with a direction. The visualization shows the amazing progress countries such as Mexico and Turkey have achieved in reducing their fertility rate.

OECD defines fertility rate as "Refers to the number of children that would be born per woman, assuming no female mortality at child-bearing ages and the age-specific fertility rates of a specified country and reference period".

Line plots have been widely used by New York Times to create very interesting infographics. This recipe is inspired by one of the New York Times visualizations. It is very important to understand that many of the infographics created by professionals are created using D3.js or Processing. We will not go into the detail of the same but it is good to know the working of these softwares and how they can be used to create visualizations.



## Getting ready

We would need to install and load the `googleVis` package to construct a line chart.

## How to do it...

To generate an interactive plot, we will load the fertility data in R using the `read.csv()` function. To generate a line chart that plots the entire dataset, we will use the `gvisLineChart()` function:

```
line = gvisLineChart(frt, xvar = "Year", yvar=c("Australia",
"Austria", "Belgium", "Canada", "Chile", "Czech.Republic",
"Denmark", "Estonia", "Finland", "France", "Germany", "Greece", "Hungary",
",
"Iceland", "Ireland", "Israel", "Italy", "Japan", "Korea", "Luxembourg",
"Mexico",
"Netherlands", "New.Zealand", "Norway", "Poland", "Portugal", "Slovakia",
", "Slovenia",
"Spain", "Sweden", "Switzerland", "Turkey", "United.Kingdom", "United.
States", "OECD34"),
```

```
options = list(width = 1200, backgroundColor = "#ADAD85", title
="Fertility Rate in OECD countries" ,
vAxis = "{gridlines:{color:'#DEDECE',count : 3}, ticks :
[0,1,2,3,4]}",
series = "{0:{color:'BDBD9D', visibleInLegend :false},
20:{color:'009933', visibleInLegend :true},
31:{color:'996600', visibleInLegend :true},
34:{color:'3333FF', visibleInLegend :true}}}")
```

To display our visualization in a new browser, we use the generic R `plot()` function:

```
plot(line)
```

## How it works...

The arguments passed in the `gvisLineChart()` function, stated in the previous section, are exactly the same as discussed under the simple line plot with some minor changes. We would like to plot the entire data for this exercise, and hence we have to state all the column names in `yvar =c()`.

Also, we would like to color all the series with the same color but highlight Mexico, Turkey, and OECD average. We have achieved this in the previous code using `series {}`, and further specify and customize colors and legend visibility for specific countries.

In this particular plot, we have made use of the same color for all the economies but have highlighted Mexico and Turkey to signify the development and growth that took place in the 5-year period. It would also be effective if our audience could compare the OECD average with Mexico and Turkey. This provides the audience with a benchmark they can compare with.

If we plot all the legends, it may make the plot too crowded and 34 legends may not make a very attractive plot. We could avoid this by only making specific legends visible.

## See also

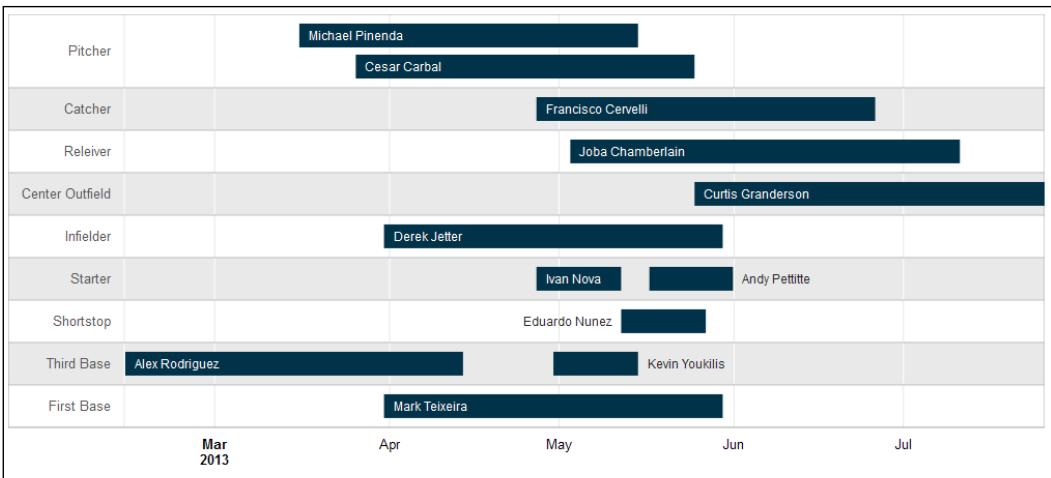
- ▶ D3 is a great tool to develop interactive visualization and this can be accessed at <http://d3js.org/>.
- ▶ Processing is an open source software developed by MIT and can be downloaded from <https://processing.org/>.
- ▶ A good resource to pick colors and use them in our plots is the following link: [http://www.w3schools.com/tags/ref\\_colorpicker.asp](http://www.w3schools.com/tags/ref_colorpicker.asp).
- ▶ I have used New York Times infographics as an inspiration for this plot. You can find a collection of visualization put out by New York Times in 2011 by going to this link, <http://www.smallmeans.com/new-york-times-infographics/>.

# Generating an interactive Gantt/timeline chart in R

Wikipedia describes a Gantt chart as "illustrate the start and finish dates of the terminal elements and summary elements of a project". These charts are used to track the progress of a project displayed against time. The first Gantt chart was developed by Karol Adamiecki in 1890.

Even though the most important application of Gantt charts is in project management, they have been applied in visualization to represent the following:

- ▶ Historical era of artist
- ▶ Periods during which baseball players are disabled
- ▶ Vanishing Wall Street firms



## Getting ready

To generate a timeline plot, we will need to install and load the `googleVis` package in R.

## How to do it...

We would import our data in R using the `read.csv()` function. The `gvisTimeline()` function requires the dates to be in date format in R. Hence, we use the `as.POSIXct()` and `as.character()` functions to re-define a new data frame:

```
base = read.csv("disable.csv")
data = data.frame(position = as.character(base$position), player =
 as.character(base$player), start = as.POSIXct(base$start), end =
 as.POSIXct(base$end))
```

The data was collected by me and is available online. We use the `gvisTimeline()` function to generate an object, which is displayed using the `plot()` function:

```
baseball = gvisTimeline(data = data, rowlabel ="position",start
 ="start", end = "end",barlabel ="player" , option = list(width =
 1000, height = 900,timeline="{singleColor : '#002A3E'}"))
plot(baseball)
```

Many of the arguments used in the `gvisTimeline()` function are self-explanatory. The `start` and `end` arguments refer to the start date and end dates, respectively; in the case of baseball data, they correspond to the length of time for which players are on the disability list. We have passed the `single-color` argument under the options to color all the lines with the same color.

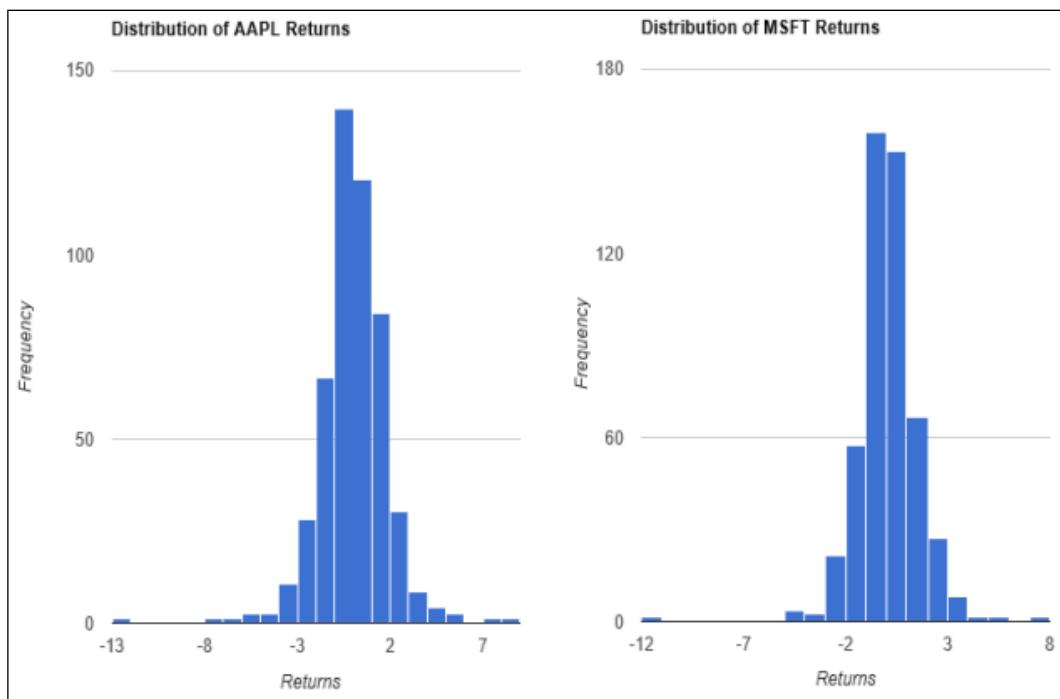
## See also

- ▶ History of Gantt Chart can be accessed at <http://www.gantt.com/>.
- ▶ New York Times visualizes Mets disability and Mets winning percentage in 2009. The timeline is combined with a line plot to add an interesting element to the visualization and is available at <http://www.nytimes.com/interactive/2009/10/01/sports/baseball/mets-injuries.html>.
- ▶ New York Times has a very interesting timeline plot of financial institutions: how they merged or vanished from Wall Street. The text is used to provide its audience with additional information. This is available at <http://www.nytimes.com/imagepages/2008/09/28/business/28lloyd.graf01.ready.html>.
- ▶ Information about injured New York Yankees can be accessed at <http://sports.newsday.com/long-island/baseball/yankees/injured-yankees/>.

## Merging histograms

Histograms help in studying the underlying distribution. It is more useful when we are trying to compare more than one histogram on the same plot; this provides us with greater insight into the skewness and the overall distribution.

In this recipe, we will study how to plot a histogram using the `googleVis` package and how we merge more than one histogram on the same page. We will only merge two plots but we can merge more plots and try to adjust the width of each plot. This makes it easier to compare all the plots on the same page. The following plot shows two merged histograms:



## How to do it...

In order to generate a histogram, we will install the `googleVis` package as well as load the same in R:

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

We have downloaded the prices of two different stocks and have calculated their daily returns over the entire period. We can load the data in R using the `read.csv()` function. Our main aim in this recipe is to plot two different histograms and plot them side by side in a browser. Hence, we require to divide our data in three different data frames. For the purpose of this recipe, we will plot the `aapl` and `msft` data frames:

```
stk = read.csv("stock_cor.csv", header = TRUE, sep = ",")
aapl = data.frame(stk$AAPL)
msft = data.frame(stk$MSFT)
googl = data.frame(stk$GOOGL)
```

To generate the histograms, we implement the `gvisHistogram()` function:

```
al = gvisHistogram(aapl, options = list(histogram = "{bucketSize :1}", legend = "none", title = 'Distribution of AAPL Returns', width = 500, hAxis = "{showTextEvery: 5, title: 'Returns'}", vAxis = "{gridlines : {count:4}, title : 'Frequency'}"))
mft = gvisHistogram(msft, options = list(histogram = "{bucketSize :1}", legend = "none", title = 'Distribution of MSFT Returns', width = 500, hAxis = "{showTextEvery: 5, title: 'Returns'}", vAxis = "{gridlines : {count:4}, title : 'Frequency'}"))
```

We combine the two `gvis` objects in one browser using the `gvisMerge()` function:

```
mrg = gvisMerge(al,mft, horizontal = TRUE)
plot(mrg)
```

## How it works...

The `data.frame()` function is used to construct a data frame in R. We require this step as we do not want to plot all the three histograms on the same plot. Note the use of the `$` notation in the `data.frame()` function.

The first argument in the `gvisHistogram()` function is our data stored as a data frame. We can display individual histograms using the `plot(al)` and `plot(mft)` functions. But in this recipe, we will plot the final output.

We observe that most of the attributes of a histogram function are the same as discussed in previous recipes. The histogram functionality will use an algorithm to create buckets, but we can control this using the `bucketSize` as `histogram = "{bucketSize :1}"`.

Try using different bucket sizes and observe how the buckets in the histograms change. More options related to histograms can also be found in the following link under the *Controlling Buckets* section:

<https://developers.google.com/chart/interactive/docs/gallery/histogram#Buckets>

We have utilized `showTextEvery`, which is also very specific to histograms. This option allows us to specify how many horizontal axis labels we would like to show. We have used 5 to make the histogram more compact. Our main objective is to observe the distribution and the plot serves our purpose. Finally, we will implement `plot()` to plot the chart in our favorite browser.

We do the same steps to plot the return distribution of Microsoft (MSFT). Now, we would like to place both the plots side by side and view the differences in the distribution. We will use the `gvisMerge()` function to generate histograms side by side.

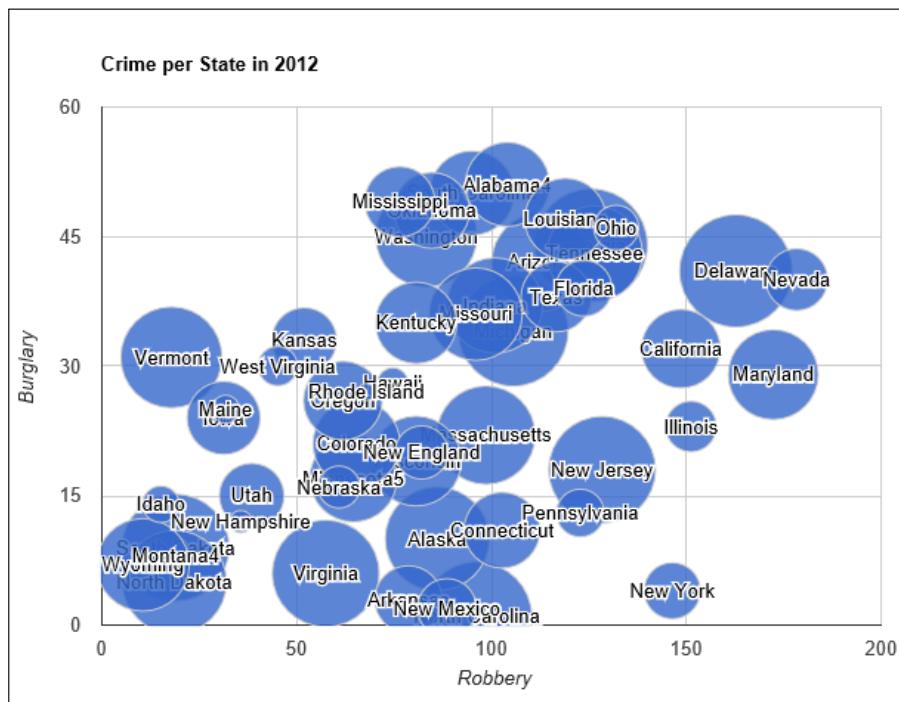
In our recipe, we have two plots for AAPL and MSFT. The default setting plots each chart vertically but we can specify `horizontal = true` to plot charts horizontally.

# Making an interactive bubble plot

My first encounter with a bubble plot was while watching a TED video of Hans Rosling. The video led me to search for creating bubble plots in R; a very good introduction to this is available on the Flowing Data website. The advantage of a bubble plot is that it allows us to visualize a third variable, which in our case would be the size of the bubble.

In this recipe, I have made use of the `googleVis` package to plot a bubble plot but you can also implement this in R. The advantage of the Google Chart API is the interactivity and the ease with which they can be attached to a web page. Also note that we could also use squares instead of circles, but this is not implemented in the Google Chart API yet.

In order to implement a bubble plot, I have downloaded the crime dataset by state. The details regarding the link and definition of crime data are available in the `crime.txt` file and are shown in the following screenshot:



## How to do it...

As with all the plots in this chapter, we will install and load the `googleVis` Package. We will also import our data file in R using the `read.csv()` function:

```
crm = read.csv("crimeusa.csv", header = TRUE, sep =",")
```

We can construct our bubble chart using the `gvisBubbleChart()` function in R:

```
bub1 = gvisBubbleChart(crm,idvar = "States",xvar= "Robbery", yvar= "Burglary", sizevar ="Population", colorvar = "Year", options = list(legend = "none",width = 900, height = 600,title =" Crime per State in 2012", sizeAxis ="{maxSize : 40, minSize : 0.5}",vAxis = "{title : 'Burglary'}",hAxis= "{title : 'Robbery'}"))

bub2 = gvisBubbleChart(crm,idvar = "States",xvar= "Robbery", yvar= "Burglary",sizevar ="Population",
options = list(legend = "none",width = 900, height = 600,title =" Crime per State in 2012", sizeAxis ="{maxSize : 40, minSize : 0.5}",vAxis = "{title : 'Burglary'}",hAxis= "{title : 'Robbery'}"))
```

The `bub2` object does not size the bubbles, but shades them and the scale of shading is automatically displayed on the top of the chart. In order to view the visualization, the readers can type `plot(bub2)` in the R console window. To view both the bubble plots side by side, the readers can use the `gvisMerge()` function in R:

```
bub3 = gvisMerge(bub1,bub2, horizontal = TRUE)
plot(bub3)
```

## How it works...

The `gvisBubbleChart()` function uses six attributes to create a bubble chart, which are as follows:

- ▶ `data`: This is the data defined as a data frame, in our example, `crm`
- ▶ `idvar`: This is the vector that is used to assign IDs to the bubbles, in our example, `states`
- ▶ `xvar`: This is the column in the data to plot on the x-axis, in our example, `Robbery`
- ▶ `yvar`: This is the column in the data to plot on the y-axis, in our example, `Burglary`
- ▶ `sizevar`: This is the column used to define the size of the bubble
- ▶ `colorvar`: This is the column used to define the color

We can define the minimum and maximum sizes of each bubble using `minSize` and `maxSize`, respectively, under `options()`. Note that we have used `gvisMerge` to portray the differences among the bubble plots. In the plot on the right, we have not made use of `colorvar` and hence all the bubbles are of the same size.

## There's more...

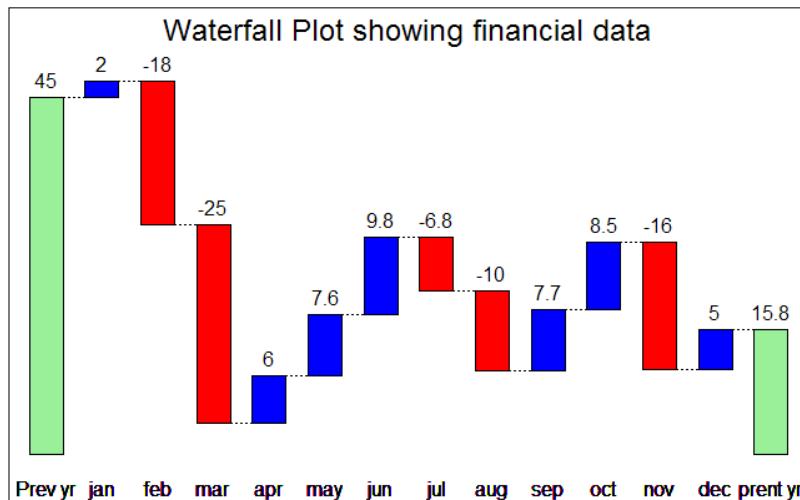
The Google Chart API makes it easier for us to plot a bubble, but the same can be achieved using the R basic plot function. We can make use of the symbols to create a plot. The symbols need not be a bubble; it can be a square as well. By this time, you should have watched Hans' TED lecture and would be wondering how you could create a motion chart with bubbles floating around. The Google Charts API has the ability to create motion charts and the readers can definitely use the googleVis reference manual to learn about this. We have covered the motion chart in the recipe *Creating animated plots in R* in Chapter, *Creating Applications in R*.

## See also

- ▶ TED video by Hans Rosling can be accessed at [http://www.ted.com/talks/hans\\_rosling\\_shows\\_the\\_best\\_stats\\_you\\_ve\\_ever\\_seen](http://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen)
- ▶ The Flowing Data website generates bubble charts using the basic R plot function and can be accessed at <http://flowingdata.com/2010/11/23/how-to-make-bubble-charts/>
- ▶ Animated Bubble Chart by New York Times can be accessed at <http://2010games.nytimes.com/medals/map.html>

## Constructing a waterfall plot in R

The waterfall plots or staircase plots are observed mostly in financial reports. I have not come across a nonfinancial application of these plots. The first and last columns of the plot are usually a total column and the floating columns indicate the incremental change. In this recipe, we generate some fake data of sales figures for every month.



# Getting ready

In order to generate a waterfall plot, we will need to install and load the `plotrix` package in R.

## How to do it...

The data for the same is imported in R using the `read.csv()` function. If we view the data, it begins with a total from last year's sales and we have some gains and some losses in sales, which are indicated by positive and negative values. The last row represents the current year's sales, which is the sum of all the rows:

```
sales = read.csv("waterf.csv")
```

The waterfall plot is constructed in R using the `staircase.plot()` function:

```
staircase.plot(sales$value, totals= sales$logic, labels =
 sales$labels, total.col = c("lightgreen"),inc.col = c("blue",
 "red","red","blue","blue","blue","red","red","blue","blue",
 "red","blue"),main ="Waterfall Plot showing financial data")
```

The first argument in the `staircase.plot()` function refers to the height of the columns. The second argument is a logical vector wherein `TRUE` refers to the total columns in our data and `FALSE` corresponds to the incremental change. The `labels` argument is used to apply the labels to our plot. The `total.col` argument is used to apply color to the total columns (in our case, it is the first and last columns).

We specify the colors for incremental change columns under the `inc.col` argument. For the incremental change columns, we have repeated the blue and red colors. We would prefer to use negative values to have red colored columns and positive incremental changes to have blue colored columns.

# 2

# Heat Maps and Dendograms

In this chapter, we will cover the following recipes:

- ▶ Constructing a simple dendrogram
- ▶ Creating dendograms with colors and labels
- ▶ Creating a heat map
- ▶ Generating a heat map with customized colors
- ▶ Generating an integrated dendrogram and a heat map
- ▶ Creating a three-dimensional heat map and a stereo map
- ▶ Constructing a tree map in R

## Introduction

The main motivation to dedicate an entire chapter on heat maps and dendograms is to introduce the concept of clustering. We have also introduced tree maps toward the end of this chapter. Recently, I have observed the rising popularity of tree maps in news media and on financial websites.

Clustering techniques can be grouped into K mean clustering and hierarchical clustering. Dendograms, a part of hierarchical clustering, are tree-like structures with branches and are used in visualization when we do not know the number of clusters in advance.

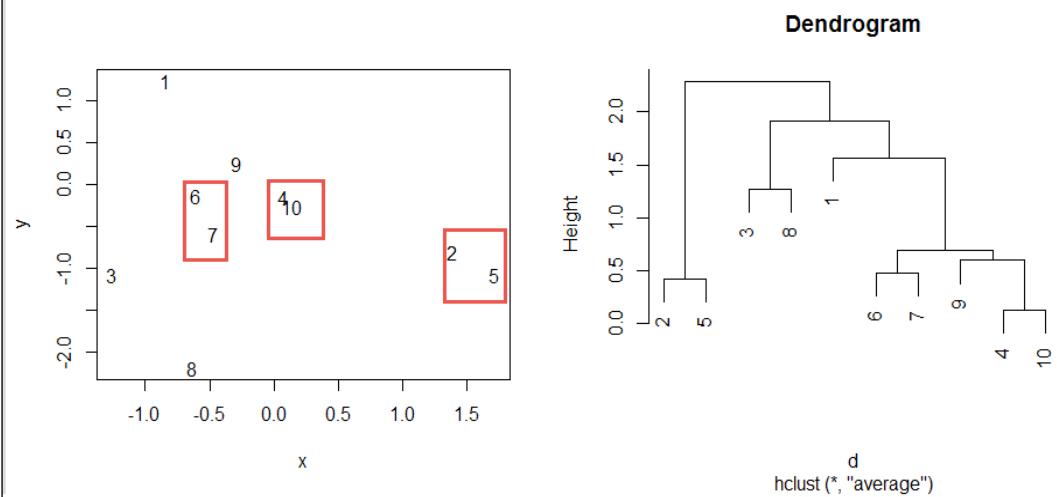
My first encounter with heat maps was an infographic published by the New York Times ([http://www.nytimes.com/interactive/2012/06/11/sports/basketball/nba-shot-analysis.html?\\_r=0](http://www.nytimes.com/interactive/2012/06/11/sports/basketball/nba-shot-analysis.html?_r=0)) about the points scored by basketball players playing for Miami Heat versus Oklahoma City Thunders. The infographic was created by Jeremy White, Joe Ward, and Mathew Ericson. Similar infographics can be constructed in R as well, but for the purpose of this chapter we have only brushed through the concept of heat maps. If readers are interested in building something very similar to the New York Times, I would highly recommend visiting the Flowing Data website (<http://flowingdata.com/2012/10/04/more-on-making-heat-maps-in-r/>). Nathan Yau has a tutorial on how the New York Times infographic can be created using R.

The idea of tree maps is not a new one but this has gained a lot of popularity in recent times. Tree maps are rectangles placed together to form a grid wherein the size of each rectangle is directly proportional to the data that this represents. A higher value would imply a bigger rectangle. The only disadvantage with such a visualization is that humans are not very well trained to distinguish between rectangles of varying sizes unless the difference is very significant. This chapter is divided into four main sections: the first section introduces the idea of dendrograms and further applies the concept to the USArests dataset. The second section introduces heat maps and how the colors in a heat map can be customized to make them look better. The third section combines a heat map with a dendrogram. Finally, the last section introduces the idea of tree maps.

## Constructing a simple dendrogram

The main idea behind clustering is to detect groups of data that share some similarity. We partition the data into distinct groups that share some common traits. Clustering has been applied in fields such as biology (to study genomic data), finance (to study clustering in foreign exchange markets, stocks, different sectors/industries in an index, and so on), and economics (to study crime in various cities and international trade). We have made use of dendrograms, which are tree-like representation of clusters, as they help us to study various cluster formation with much more accuracy. The tree structure allows us to cut trees at various heights to distinguish between clusters with dissimilar characteristics.

In this recipe, we would generate 10 random numbers to introduce the concept of dendrograms. The leaves of a dendrogram merge to become a branch as we move up the tree structure. In the scatter plot on the left side, values 4 and 10 are quite similar as they merge on the lowest point. In the dendrogram, we represent 4 and 10 labeled values as two leaves that merge on the next level with 9. The values labeled 2 and 5 have their own separate cluster (indicating dissimilarity) as the distance between 2 and 5 and the rest of the tree is much greater. The dendrogram is as follows:



## Getting ready

We would generate a random data of 10 observations for this recipe and a sequence of 10 numbers to label the point:

```
set.seed(5)
x = rnorm(10)
y = rnorm(10)
z = seq(1,10, by = 1)
```

## How to do it...

The `cbind()` function binds (column-wise) three vectors together. We use the `data.frame()` function to structure our data in an object of class data frame:

```
mtx = data.frame(cbind(x,y,z))
```

We generate a distance matrix in R using the `dist()` function. The distance matrix is a compulsory argument in the `hclust()` function. The output of a `hclust()` function is a cluster object:

```
d = dist(mtx, method = "euclidean")
clust = hclust(d, "ave")
```

Next, we generate two plots, a scatter plot on the left side and a dendrogram on the right side. The `text()` function is used to apply labels to our scatter plot. The dendrogram is also generated using the `plot()` function, but the arguments are not `x` and `y` axes but the cluster calculated using the `hclust()` function:

```
par(mfrow = c(1,2))
plot(mtxx,mtxy, type = "n", xlab = "x", ylab = "y") # generates
the plot on the left side
text(mtxx,mtxy, labels = z) # applies labels to plot on the left
plot(clust, main = "Dendrogram") # generates plot on the right
```

## How it works...

The `set.seed(5)` function defines the starting state of the random number generator in R. We use the `rnorm()` function to generate 10 random numbers each for `x` and `y` axes. If you omit `set.seed(5)`, R will generate a new set of random values every time we run the `rnorm()` function. We will also generate a sequence of numbers to label our axes: we will call this vector `z`. The `seq()` function takes three arguments from, to, and by. In our example, we want to start the series from 1 to 10 and the difference between the values should be 1.

A very important part of the `hclust()` function is the distance matrix calculated using the `dist()` function. The first argument in the function is the data and the second argument is the method of calculating the distance. We will use Euclidean distance but R provides us with a few other choices. We can read more about the same by typing `?dist` in the R console window.

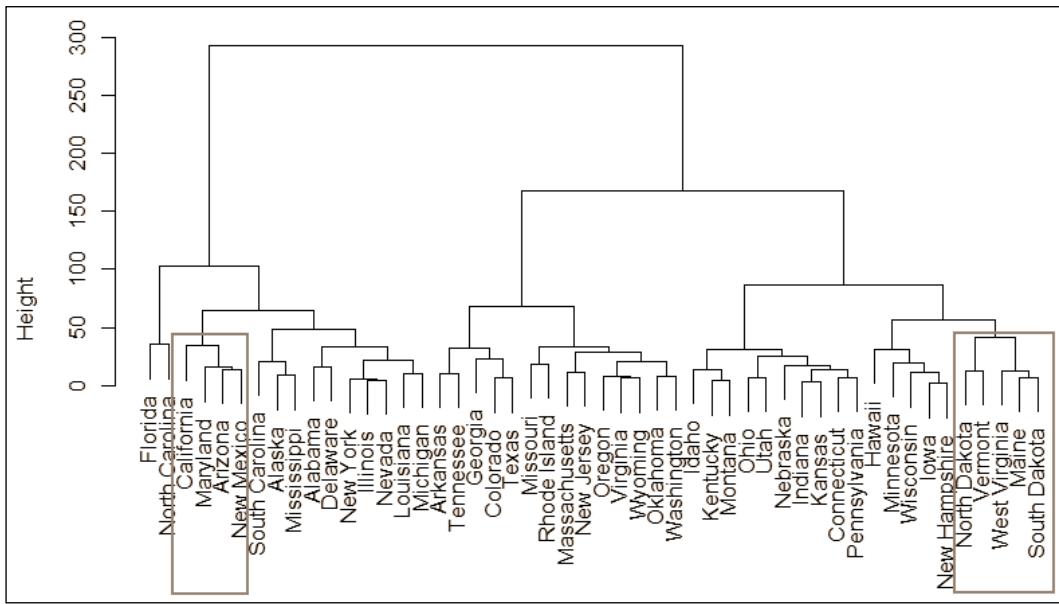
The `hclust()` function is a basic R functionality and is used for hierarchical cluster analysis in R. Hierarchical cluster is a method of clustering in which we do not have any prior knowledge of the underlying cluster. To learn more about hierarchical clustering, readers should refer to *An Introduction to Statistical Learning*. Type `?hclust()` to get more information on this function in R.

The first argument in `hclust()` is the distance: we have already generated the distance matrix in R and stored the values as vector `d` with class `dist`. The second argument is the method: this is the agglomeration method. We will use `average` as a method of agglomeration. Readers can learn more about all the available choices of agglomeration in R by typing `?hclust` in the R console window.

Once we have calculated the hierarchical cluster, we simply plot this in R using the base plotting function. We have constructed a scatter plot and a dendrogram side by side to make a comparison between the two and further study clustering in our data.

## There's more...

In the following dendrogram, I have implemented the same idea discussed in this recipe to actual data in R. However, I have cropped the data to make the interpretation of the plot a bit easier.



We have not shown the scatter plot for the dendrogram of the USArests data. The code.txt file consists of the code for generating the same. Readers can compare both the plots and study the formation of clusters. We have also analyzed the same data using the Principal Component Analysis (PCA) under the recipe *Generating an integrated dendrogram and a heat map*:

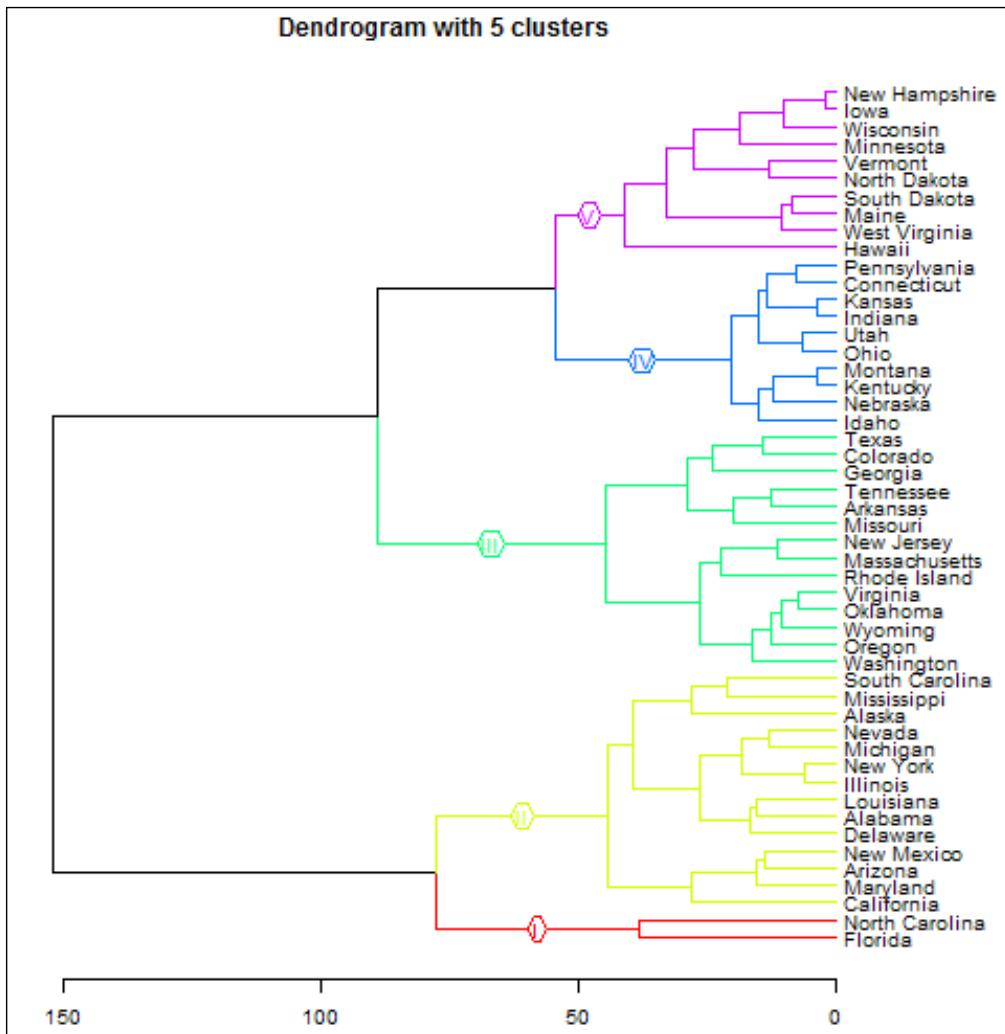
```
data = data.frame(USArests[,1:3])
dt = dist(data, method = "euclidean")
clust = hclust(dt)
plot(data$Murder, data$Assault) # generates a scatter plot
text(USArests$Murder,USArests$Assault, labels =
 row.names(USArests), cex = 0.6) # applies labels
 to a scatter plot.
plot(clust) # generates a dendrogram plot
```

## See also

- ▶ An Introduction to Statistical Learning, J Gareth, D Witten, T Hastie, and R Tibshirani, Springer.statistical. It can be retrieved from <http://www-bcf.usc.edu/~gareth/ISL/>.

# Creating dendrograms with colors and labels

The dendrogram plot in the previous example was all black and white. While making a good presentation, we would like this to be more informative. We would also like our audience to look at the dendrogram and immediately spot clusters and relationships among variables. We will now move a bit away from basic R plots and use a package called `dendroextras`. A sample dendrogram is as follows:



## Getting ready

For creating a plot that is easy to interpret and study, we would use the `dendroextras` package.

For a better understanding on installing packages in R, please refer to the recipe *Installing packages and getting help in R*, in Chapter, *A Simple Guide to R*:

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

## How to do it...

The dendrogram is constructed using the `USArrests` data available in R. The dataset consists of four variables (`Assault`, `Murder`, `Rape`, and `Percent of Urban Population`) in each of 50 states of the USA in 1973. We would load the `USArrests` data using the `data()` function:

```
data(USArrests)
```

In order to avoid any overlap in labels, we will adjust our margins for the plot using the `mar` attribute. The `mar` attribute is passed within the `par()` function. Since we plot all the 50 states, we would adjust the label sizes using the `cex` attribute:

```
par(mar = c(2,10,2,10), cex = 0.6)
```

We have already discussed the use of the `hclust()` and `dist()` functions in the previous recipe. To color the branches and the leaves, we use the `colour_clusters()` function. Note that the function has two levels of nesting `hclust()` and `dist()`:

```
clst1=colour_clusters(hclust(dist(USArrests),
 "ave"),5,groupLabels=as.roman)
```

Next, we generate the dendrogram using the basic R `plot()` function:

```
plot(clst1, main = "Dendrogram with 5 clusters", horiz = TRUE)
```

## How it works...

We do not need the `read.csv()` function as the data is a part of the R datasets and can be referenced directly by calling `data(USArrests)`.

The `par()` function allows us to modify the plotting region as well as the chart itself. This comes with many different arguments and the best way to remember them is to use them. We can always type `?par()` to learn more about various options. The `mar` argument allows us to set the margins of the plot; it has four values each for the bottom, left, top, and right margin. The `cex` allows us to magnify the label's sizes.

Now, to plot a colored dendrogram, we just need a single line of code and the function `colour_clusters()`. The first argument within the function is the cluster. In our case, we compute the distance as well as the hierarchical cluster using the `dist()` and `hclust()` functions. The next argument is the `slice` argument, where we specify R to slice the dendograms into five groups and apply labels to each. We can also use three or two: try this for yourself and see how the dendrogram changes. The number of groups corresponds to the number of major clusters in our data. As we increase the number of groups, we observe that the plot gets divided and subdivided accordingly. We pass `clst1`, which is a dendrogram object as an argument within the `plot()` function. Since the class of `clst1` is a dendrogram, R automatically generates the tree structure.

## There's more...

The dendrogram need not be in a rectangular shape; R provides packages such as `specular`, `dynamicTreeCut`, and so on, which makes use of trees such as dendograms. We can also plot a three-dimensional dendrogram as illustrated in the following code:

```
install.packages ("NeatMap")
library(NeatMap)
clust = hclust(dist(USArrests), method = "complete")
pos<-nMDS(USArrests,metric="euclidean")
draw.dendrogram3d(clust,pos$x,labels=rownames(USArrests),
label.size=0.75)
```

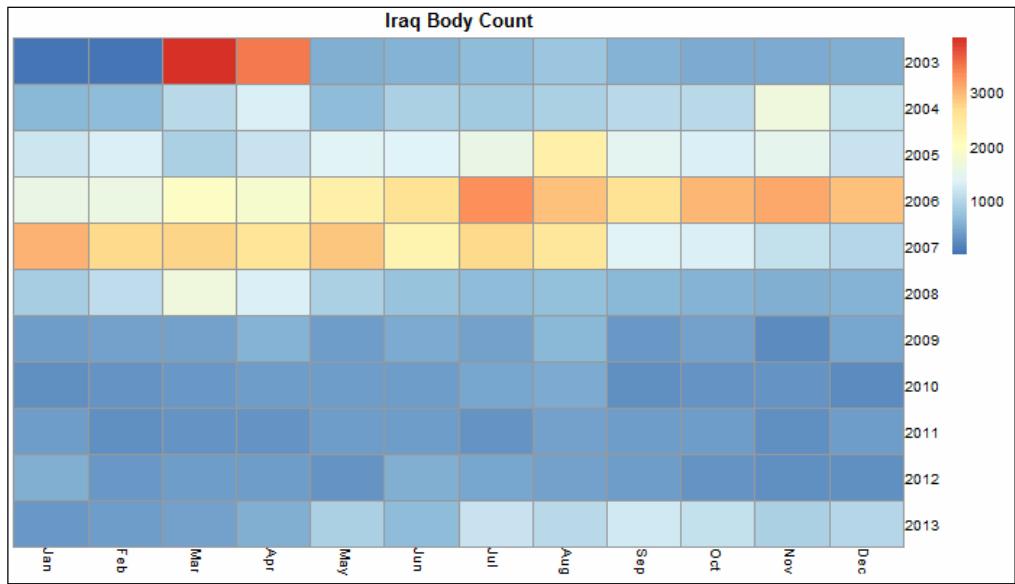
The preceding code uses the `NeatMap` package in R. Hence, we load the package and also load the `USArrests` dataset available in R. For this exercise, we first need to calculate the cluster using the `hclust()` and `dist()` functions in R and then position the list for the labels.

We use the `nMDS()` function to calculate the position: this is necessary as the image is three-dimensional. In the last step, we will plot a three-dimensional dendrogram using the `draw.dendrogram()` function. We will pass `clust` and `pos` as arguments to the `draw.dendrogram3d()` function. Note that the plot is generated in a new window.

## Creating a heat map

Heat maps are a visual representation of data wherein each value in a matrix is represented with a color. It has been widely observed among the visualization community that color as a visual cue is ranked toward the bottom when compared with other tools such as position, length, angle, and so on. The traditional heat map is represented in a rectangular format, but at the very basic level a heat map is representing colors for numbers. An interesting implementation of heat maps is its integration with a calendar. We will study the calendar heat maps in detail in the recipe *Generating interactive calendar maps* in Chapter, *Data in Higher Dimensions*.

The Iraq body count is a database that maintains records of violent civilian deaths since 2003. The heat map clearly shows that the most civilian deaths occurred in the 2006-2008 period and it declined thereafter. Heat maps have a much better application to data such as stock prices, sports data, and biology (to study the levels of expression of many genes).



## Getting ready

In order to plot a heat map, we will need to install and load the `pheatmap` package available in R. Please note that heat maps are available with the basic R package, discussed under the *There's more...* section of this recipe.

## How to do it...

We would first install and load the package in R using the following lines of code:

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

We import our data, and then clean and transform this into a matrix using the `read.csv()` and `data.matrix()` functions. We require to pass the `row.names()` function to label our heat map correctly:

```
irq = read.csv("iraqbdc.csv", header = TRUE, sep =", ")
row.names(irq)=irq$years
irq = data.matrix(irq)
irq = data.matrix(irq[,2:13])
```

We now plot our heat map using the `pheatmap()` function:

```
pheatmap(irq, cluster_row= FALSE, cluster_col = FALSE, main ="Iraq
Body Count")
```

## How it works...

We would like R to use the `Years` column as row names in our dataset while plotting the heat map. If we do not specify which column to use as labels, R will simply use a sequence of numbers. We can assign our row names using the `row.names()` function. The `row.names()` function has only one argument: the column heading that we would like to use as our row names.

The `pheatmap()` function requires that the data be in a matrix format. When we import the data in R, it has a class of `data.frame`. We can convert the data to a matrix form by using the `data.matrix()` function. To learn more about the function, readers should type `?data.matrix` in the R console window.

We observe that our matrix has 14 columns and the `Years` column is not much use to us, since we have already defined row names for R to use. Hence, we will edit our matrix in R and remove the `Years` column using square brackets `[ ]` to manipulate our data, as shown in the following code:

```
irq = data.matrix(irq[,2:13])
```

To learn more about matrix manipulation, please refer to the recipe *Editing a matrix in R* in *Chapter, A Simple Guide to R*. The first argument in the `pheatmap()` function is our data matrix `irq`. The next two arguments `cluster_row` and `cluster_col` are set to `FALSE` to suppress the dendrogram.

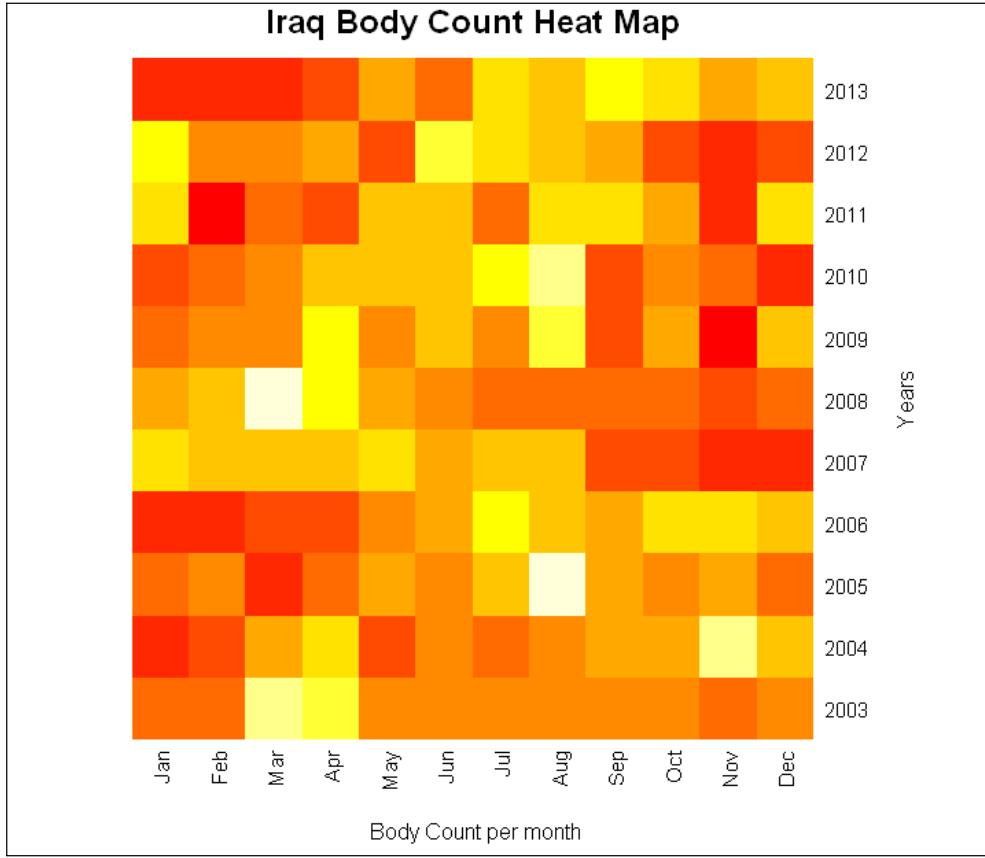
## There's more...

You can generate a heat map using the basic R plot as well. The following image was generated using the basic R plot function called `heatmap()`. We can simply replace the `pheatmap()` function line with the following line of code:

```
heatmap(irq, Rowv = NA, Colv = NA, main = "Iraq Body Count
Heat Map", xlab = " Body Count per month", ylab = "Years")
```

To learn more and understand other options available with the `heatmap()` function, type `?heatmap` in the command window.

Readers may want to change the label's orientation in both the maps. It is not very easy to do this. We would have to manipulate our functions to change the orientation. A good explanation on how to do this is provided in Stack Overflow.

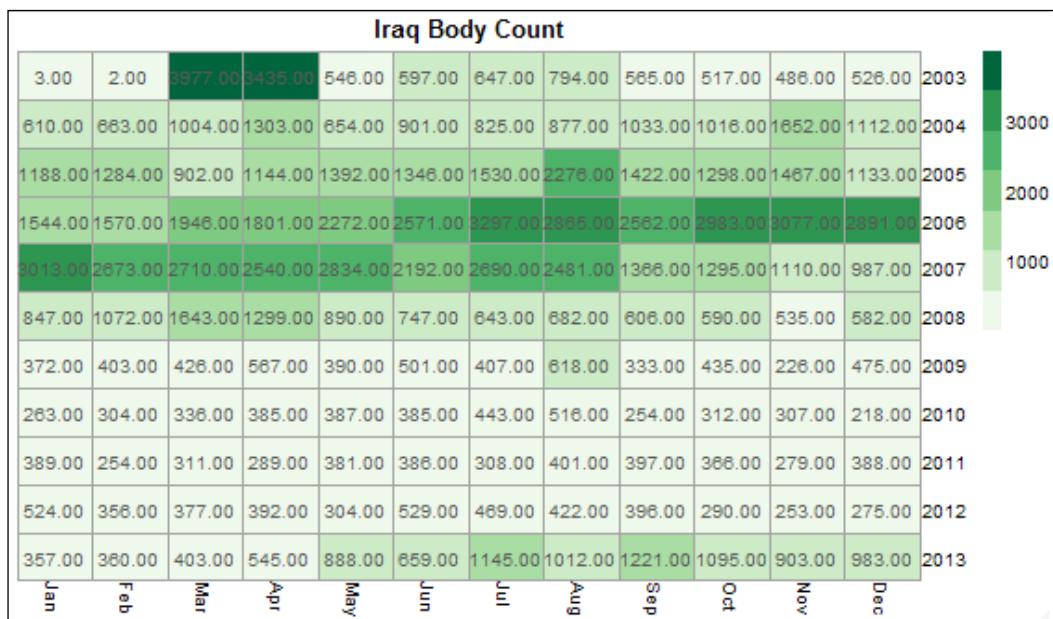


## See also

- ▶ *Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods*, Cleveland and McGill (1983), can be accessed at <http://www.cs.ubc.ca/~tmm/courses/cpsc533c-04-spr/readings/cleveland.pdf>
- ▶ Iraq body count can be accessed at <https://www.iraqbodycount.org/>
- ▶ Editing label orientation can be accessed at <http://stackoverflow.com/questions/15505607/diagonal-labels-orientation-on-x-axis-in-heatmaps>

# Generating a heat map with customized colors

In the previous recipe, we prepared our data and created a heat map. The heat map was constructed using the default color scheme. What if we would like to plot a heat map using a softer colors? This recipe dives into plotting a heat map by customizing colors. The heat map for Iraq body count is as follows:



## Getting ready

For implementing a customized color scheme in our heat map, we require the following two packages available developed for R:

- ▶ `pheatmap`
- ▶ `RColorBrewer`

## How to do it...

We start the recipe by installing the necessary packages and loading the same in our active R session:

```
install.packages(c("pheatmap", "RColorBrewer"))
library(RColorBrewer)
library(pheatmap)
```

Next, we can import the data and process the same. The steps in the following code are the same as discussed in previous recipes:

```
irq = read.csv("iraqbdc.csv", header = TRUE, sep = ",")
row.names(irq) = irq$years
irq = data.matrix(irq)
irq = data.matrix(irq[, 2:13])
```

Since we would like the chart to use our custom colors from the `RColorBrewer` package, we construct a color palette for the same using the `brewer.pal()` function:

```
heatcolor = brewer.pal(7, "Greens")
```

We implement the heat map using the `pheatmap()` function:

```
pheatmap(irq, cluster_row= FALSE, cluster_col = FALSE,
display_numbers = TRUE, color = heatcolor, main =
"Iraq Body Count", fontsize_number = 10)
```

## How it works...

We define our custom color scheme in R using the `brewer.pal()` function. `RcolorBrewer` comes with various useful color schemes and they can be viewed by typing `display.brewer.all()` in the R console window. We would use a very basic color scheme of Greens. The number 7 indicates seven levels of shading:

```
heatcolor = brewer.pal(7, "Greens")
```

The arguments for heat maps are discussed in the previous recipe; the only addition to this code is the `col` argument, which specifies the color scheme defined by us, and the `display_number` argument:

```
pheatmap(irq, cluster_row= FALSE, cluster_col = FALSE,
display_numbers = TRUE, color = heatcolor, main =
"Iraq Body Count", fontsize_number = 10)
```

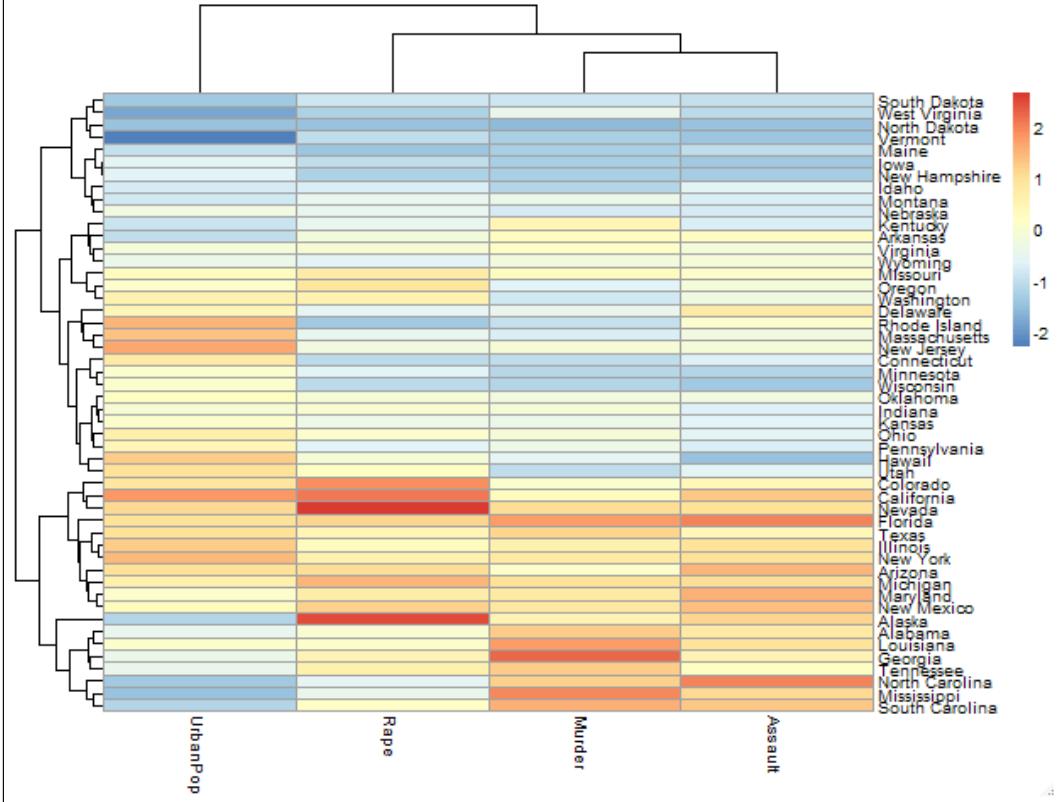
The color shading provides our audience with a visual cue as to the year or month in which the war claimed more innocent lives. The legend key, where a darker shade represents higher deaths, assists our audience in better data interpretation. We can overcome the issue many people face in interpreting colors by adding the actual value in a cell, as shown in this recipe.

## Generating an integrated dendrogram and a heat map

In the beginning of this chapter, we learned how to plot a dendrogram, and in the next section we learned about heat maps. In this recipe, we will integrate these two in a single plot. The advantage of using a heat map with dendograms is that the visualization provides us with more information.

We will observe in this recipe that we are able to view the clustering via dendograms drawn over rows and columns, and the color plotted using the heat map provides us with the information as to the strength of this relationship.

For this recipe, we will use the USArests dataset, which was also used in the prior recipe of this chapter. The techniques of visualization learned in this chapter are a part of a branch of statistics known as unsupervised learning. It is not possible to cover all the techniques (K mean clustering, PCA, and hierarchical clustering), and hence to learn more about this field you should read *Chapter, Unsupervised Learning, An Introduction to Statistical Learning*.



## How to do it...

We install as well as load the `pheatmap()` package in R using the following lines of code:

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

We will first import the data and scale it. In R, we can scale the data using the `scale()` function. The need for scaling is discussed in detail in *Chapter, Unsupervised Learning, of An Introduction to Statistical Learning*:

```
data = as.matrix(scale(USArrests))
clst = hclust(dist(data))
```

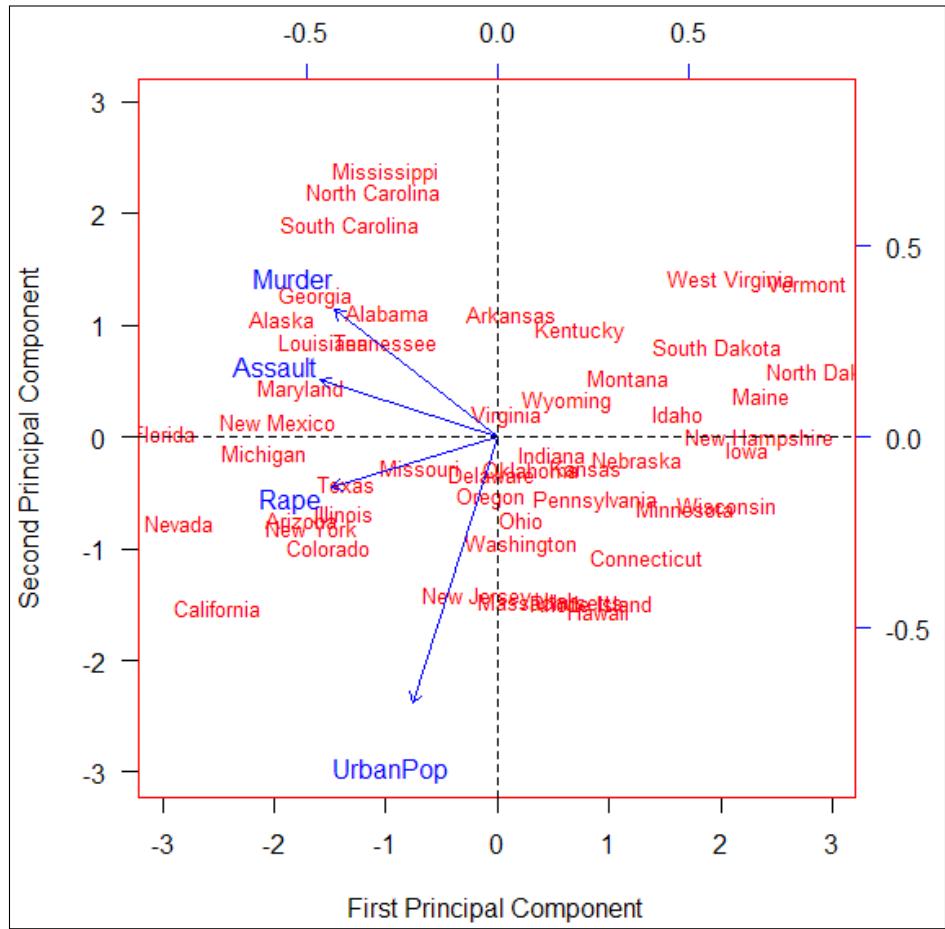
Once we scale the data, we can simply plot the data using the `pheatmap()` function. In this recipe, we will not suppress the dendrogram. Hence, we will use all the default settings to plot a dendrogram and a heat map:

```
pheatmap(data)
```

We do not require using the `clst` object for this plot. In case we are interested to chart a dendrogram, we would require the distance matrix as an argument.

## There's more...

As indicated in the introduction of this recipe, **PCA** can be used to visualize data and decipher information in a multidimensional dataset such as USArests. The PCA reduces the dimension of our data and we can reach the same conclusion as observed in this recipe using the `pheatmap()` function .



The plot was generated using the `biplot()` function in R. The `prcomp()` function allows us to perform the PCA in R:

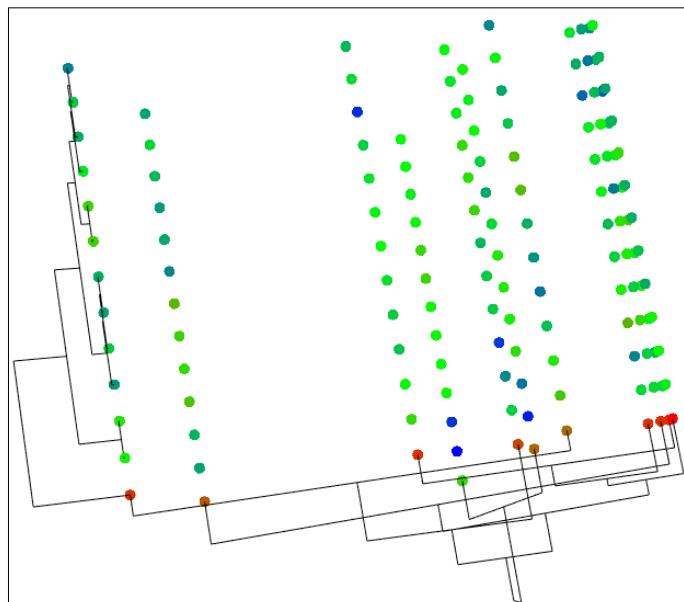
```
pc = prcomp(USArrests, scale = TRUE)
biplot (pr.out , scale =0, col =c("red","blue"), cex = c(0.8,1),
 xlab = "First Principal Component", ylab = "Second Principal
 Component")
abline(h = 0, lty =2)
abline(v = 0, lty =2)
```

## See also

- ▶ *An Introduction to Statistical Learning*, J Gareth, D Witten, T Hastie, and R Tibshirani, Springer, can be accessed at <http://www-bcf.usc.edu/~gareth/ISL/>

## Creating a three-dimensional heat map and a stereo map

We have studied two-dimensional heat maps, but R also allows us to plot three-dimensional interactive heat maps using the `NeatMap` package. The `NeatMap` package manual states the limitation of using the `heatmap()` function as "*The traditional clustered heatmap makes use of cluster analysis to re-order rows and columns such that similar elements are placed together. However, cluster analysis is a poor choice for ordering method since this does not provide a unique ordering*".



# Getting ready

In order to plot a three-dimensional heat map and a stereo map, we would need to install the `NeatMap` package available in R.

## How to do it...

In order to plot a three-dimensional heat map and a stereo map, we would start by loading the `NeatMap` package and load the data. After loading and cleaning the data, we would use `make.profileplot3D()` and `make.stereo.profileplot3D` to plot three-dimensional heat maps and stereo maps, respectively:

```
library(NeatMap)
irq = read.csv("iraqbdc.csv", header = TRUE, sep =",")
irq$years= row.names(irq)
irq = data.matrix(irq)
make.profileplot3d(irq, row.method="PCA", column.method=
 "average.linkage", col = c("red","green","blue"),
 point.size = 10, labels = row.names(irq))
make.stereo.profileplot3d(irq, row.method="PCA", column.method=
 "average.linkage", labels = row.names(irq), label.size = 1)
```

The `make.profileplot3d()` function is implemented to plot a three-dimensional map. The first argument in the function is the data followed by the `row.method` argument, which is constructed using the PCA. The `column.method` argument uses the average linkage. The arguments `point.size` and `labels` can be used to adjust the point sizes and label names, respectively:

```
make.profileplot3d(irq, row.method="PCA", column.method=
 "average.linkage", col = c("red","green","blue"), point.size =
 10, labels = row.names(irq))
```

We can plot a `make.stereo.profileplot3d` by using the following line of code:

```
make.stereo.profileplot3d(irq, row.method="PCA", column.method=
 "average.linkage", labels = row.names(irq), label.size = 1)
```

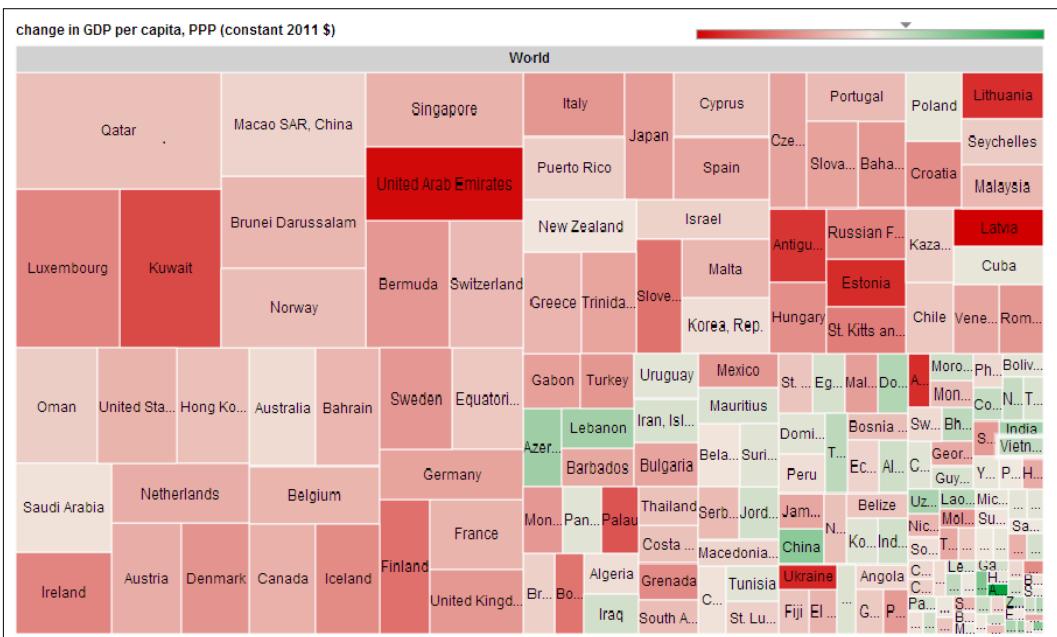
Many of the arguments in the `make.stereo()` function are similar to the function mentioned in `make.profileplot3d()`.

## See also

- ▶ The `NeatMap` manual can be accessed at <http://cran.r-project.org/web/packages/NeatMap/NeatMap.pdf>

# Constructing a tree map in R

Tree maps are basically rectangles placed adjacent to each other. The size of each rectangle is directly proportional to the data being used in the visualization. Tree maps have been used to plot the most watched news on the web by `newsmap.jp`. They have also been applied in financial websites such as smart money to visualize financial market movements. In this recipe, we will implement a tree map using the `googleVis` package.



## Getting ready

We would require to install and load the `googleVis` package for the purpose of the visualization.

## How to do it...

In order to implement a tree map, we would first install and load the `googleVis` library in the R session and import the data in the R session:

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

We import the data using the `read.csv()` function and store the data as a data frame `shk`:

```
shk = read.csv("shrink.csv", header = TRUE, sep = ",")
```

The following code creates a row and adds this to the data; this is necessary to make the code work:

```
shk$Parent [shk$CountryName == "World"] = NA
```

We create a tree object in R using the `gvisTreeMap()` function:

```
tree = gvisTreeMap(shk, idvar = "CountryName", parentvar =
 "Parent", sizevar = "X2009", colorvar = "Change", options =
 list(width = 900, height = 500, showScale = TRUE, maxColor =
 "#009933", minColor = "#CC0000",
 title = "change in GDP per capita, PPP (constant 2011 $)",
 fontColor = "black")
```

We can now construct our plot using the `plot()` function. Running the code will open up a new browser and create an interactive tree map:

```
plot(tree)
```

## How it works...

In order to plot `gvisTreeMAP()` in R, we need at least four arguments. However, at times we would come across situations wherein we would not want to plot `parentvar` in our plot. The `googleVis` package can be forced to accept an `NA` value for the parent variable. We will discuss `parentvar` in the *There's more...* section, but for the current purpose let's assume that we do not want `parentvar` to appear in our plot.

If you open the `shrink.csv` file, you will observe that one of the rows is `World`. This is `parentvar` and the value specified under the column titled `parent` is `NA`. All the other values for parent column are `World`.

In order to instruct R to assign a value of `NA` to the row `world`, we use the following code:

```
shk$Parent [shk$CountryName == "World"] = NA
```

Let's read this line from left to right: we simply instruct R to go to the column `parent` of the dataset titled `shk`. Further, go to the column `CountryName` and if the value in this column equals `World`, assign it a value of `NA`. Note the use of square brackets `[]`.

The advantage of using `googleVis` over the `portfolio` package for tree maps is the flexibility and interactivity that comes along with the Google Chart API.

In the previous code, `gvisTreeMap` takes at least five arguments, which are as follows:

- ▶ `data`: In our example this shnk.
- ▶ `idvar`: This is usually the label of each grid in the plot. In our example, it is the names of the countries.
- ▶ `parentvar`: This argument is used in the next example. We will force this variable to be NA. Note that `parentvar` is needed in a tree map but we are going to force this to be NA.
- ▶ `sizevar`: This argument defines the size of each square in our tree map.
- ▶ `colorvar`: This argument defines the variable to be used to color each square.

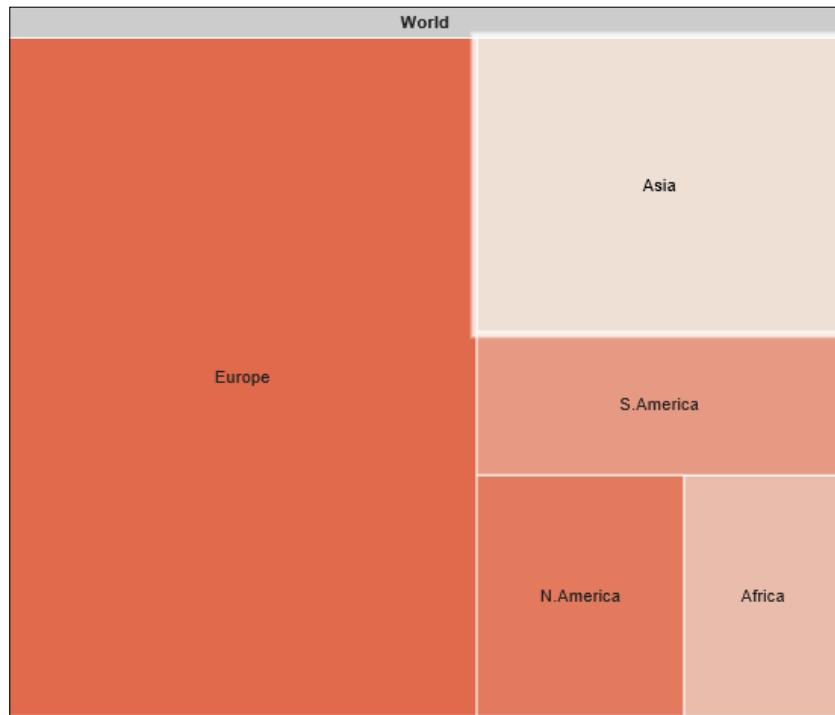
Tree map comes with a few handy options, which can be specified using `options = list()`. In our example, we have made use of `title`, `width`, `height`, and `font color`, which have been explained and used in the previous chapter. The tree maps come with a few other options such as `minColor` and `maxColor`, which are used to specify a range of colors.

The `colorvar` argument works with `minColor` to assign the lowest value to it and assigns the highest value to `maxColor`. We can also apply a scale to our plot using the `showScale = TRUE` argument. This is useful to interpret the plot. Lastly, we use the `plot(tree)` function to plot the tree map in a browser.

## There's more...

The `parentvar` argument in the tree map allows us to create a drill-down effect in our visualization. If you observe in the following image, we have shown continents instead of countries but these rectangles are clickable and will show you the countries that belong to this continent.

In order to generate a drill-down effect, we need to structure the data in a particular manner. If you observe, the data we used in the previous recipe had NA as parentvar. We will remove this NA in our new file. Also, we will add a few more rows that will appear as labels to the initial tree map (in our case, names of continents). To avoid any errors, please be sure to observe and understand the data in a CSV file.



The following code is exactly the same as the code explained under the *How to do it...* section of this recipe:

```
shk = read.csv("shrink1.csv", header = TRUE, sep =",")
tree = gvisTreeMap(shk, idvar = "CountryName", parentvar =
 "Parent", sizevar = "X2009", colorvar = "Change")
plot(tree)
```

## See also

- ▶ History of a tree map can be accessed at <http://www.cs.umd.edu/hcil/treemap-history/>.
- ▶ We can also create similar maps in R using the `map.market()` function of the `portfolio` package. The manual for the same can be accessed at <http://cran.r-project.org/web/packages/portfolio/portfolio.pdf>.

# 3

# Maps

In this chapter, we will cover the following recipes:

- ▶ Introducing regional maps
- ▶ Introducing choropleth maps
- ▶ A guide to contour maps
- ▶ Constructing maps with bubbles
- ▶ Integrating text with maps
- ▶ Introducing shapefiles
- ▶ Creating cartograms

## Introduction

Maps are one of the most widely used forms of data visualization. With the advancement in Geographic Information Systems and technology, it has become possible to collect more precise data and visualize it to reveal trends and conclusions, which seemed impossible before. It is possible to aggregate data based on the geographic location and study the emerging patterns, understand the relationships between two regions, and observe the changing pattern over time.

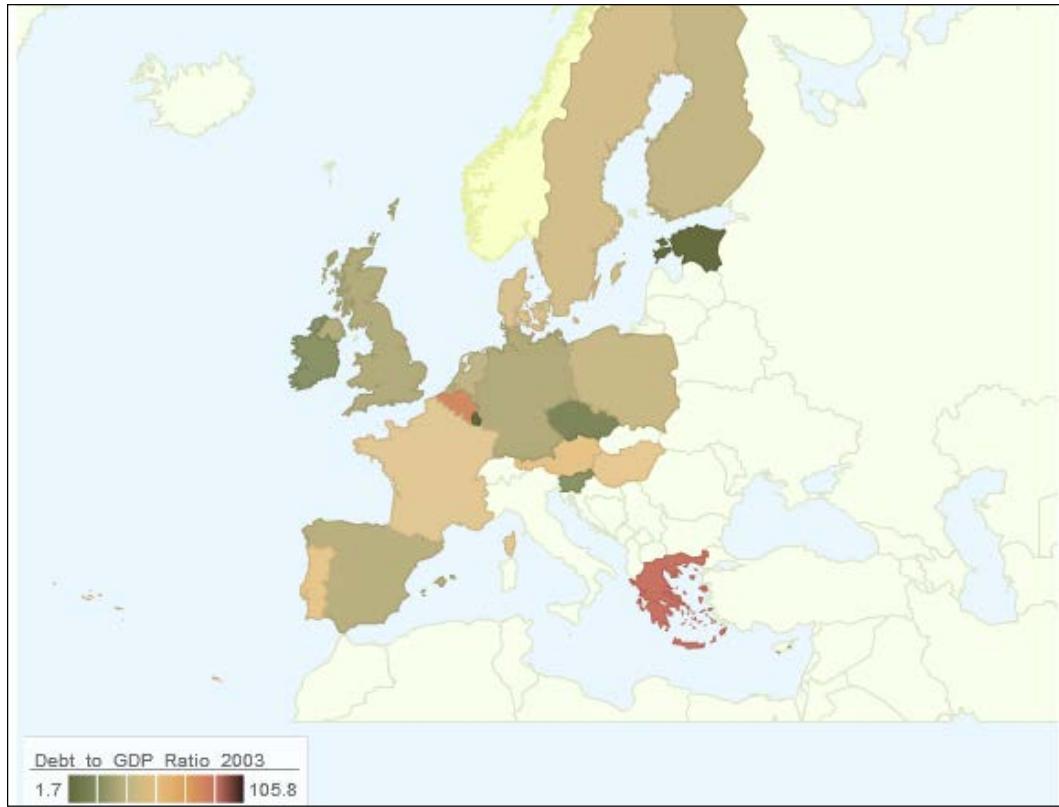
My motive behind this chapter is to introduce the readers to various forms of spatial data visualization. This chapter clearly does not do justice to the visualization of geographic data but it leads you towards the right tools, methods, and procedures. A lot has been researched and written on spatial visualization, and I would encourage readers to go beyond what has been discussed in this chapter.

This chapter helps R users to create interactive maps using the `googleVis` package. The *Introducing choropleth maps* and *A guide to contour maps* recipes go into more detail on generating choropleth maps and contour maps. We also introduce bubble plots and plotting text on maps. Finally, we will also learn to create a shapefile by connecting data and then create a cartogram using the same shapefile.

## Introducing regional maps

We encounter maps on a daily basis, be it for directions or to infer information regarding the distribution of data. Maps have been widely used to plot various types of data in R.

The following visualization relates to the debt to GDP ratio in the European Union. Users can hover over the visualization and get more information related to the data. The visualization is inspired by the New York Times article on the same issue.



# Getting ready

We need to install and load the `googleVis` package in R:

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

## How to do it...

For the purpose of this recipe, we will import the data in R using the `read.csv()` function. The data set comprises the debt to GDP ratio among the European Union nations during the 2003-2010 period. We can check our column headings and data using the `head()` function:

```
debt = read.csv("debt.csv", header = TRUE, sep = ",")
```

The visualization is generated in R using the `gvisGeoMap()` function and the same is displayed in a browser using the `plot()` function:

```
eurdebt <- gvisGeoMap(debt, locationvar="Country", numvar=
 "Debt_to_GDP_Ratio_2003", hovervar="text",
 options = list(width = "600px", height ="700px",
 dataMode = "regions", region = '150',
 colors= "[0xF8DFA7,0x8D9569,0xE9CC99,0xE2AD5A,0xCA7363]"))
plot(eurdebt)
```

## How it works...

The first argument under the `gvisGeoMap()` function relates to the data that is defined as a data frame known as `debt`. The second argument is `locationvar`, which can be specified using many different options. In our recipe, the column containing the country names is passed as `locationvar`. The `numvar` argument corresponds to the column containing our data. The `hovervar` argument displays the text, if specified in our data file. This is a handy tool in case we need to insert additional information to our visualization.

The `options` argument is used to specify many different options related to the visualizations. One of the important attributes passed in the `options` argument is `dataMode = "regions"`. The attribute is required as we would like to plot only the European Union and not the entire world. The second attribute `region = '150'` represents the code to display the European Union. Readers can learn about all the regional codes on the developer website at <https://google-developers.appspot.com/chart/interactive/docs/gallery/geochart>.

The `colors` argument is used to customize the colors. The values passed as `colors` are hex values.

## See also

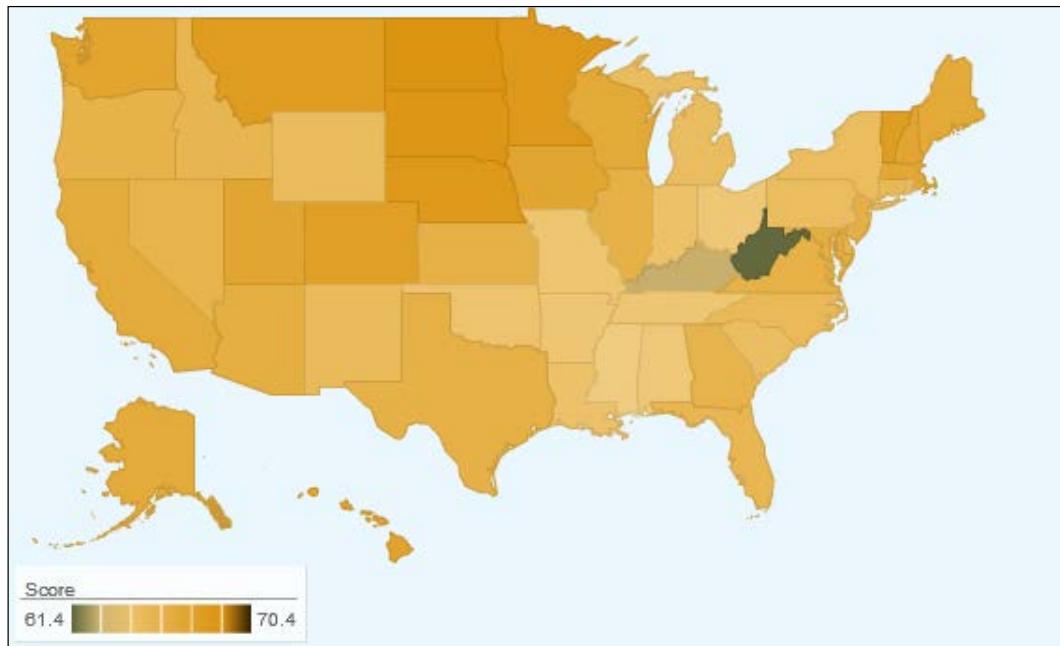
- ▶ The *Debt Rising in Europe* article at <http://www.nytimes.com/interactive/2010/04/06/business/global/european-debt-map.html>
- ▶ The *Dude Map* at <http://qz.com/316906/the-dude-map-how-american-men-refer-to-their-bros/>

## Introducing choropleth maps

Choropleth maps have been extensively used as a medium to represent statistical data. Choropleth maps can be used to represent different data types, such as quantitative, diverging, or qualitative data, using color schemes. The RColorBrewer website provides a great guide to select the color scheme based on the data available to the user. Legends play a very vital role in choropleth maps as we need to state what each color represents in a map.

Choropleth maps can be state level as well as county level. In this recipe, we will plot well-being data on a state level. The data is made available by Gallup (<http://www.gallup.com/home.aspx>).

Please note that there is more than one way to plot a choropleth map. We can use the basic R plotting function along with the maps package or use `ggplot` to plot a choropleth map. The benefit of the `googleVis` package is that it makes the plot interactive and makes it easy to integrate with web pages or blogs.



## Getting ready

We need to install and load the `googleVis` package to construct an interactive choropleth map in R.

## How to do it...

We will import our well-being data using the `read.csv()` function:

```
well = read.csv("wellbeing.csv", header = TRUE, sep = ",")
```

We can now generate the plot using the `gvisGeoMap()` function and display the same in a browser using the `plot()` function:

```
USA <- gvisGeoMap(well, locationvar="State", numvar
="Score", hovervar = "text",
options = list(width = "600px", height ="700px", dataMode =
"regions", region = "US", colors=
" [0x64693D,0xF1CC80,0xEABA58,0xE3A732,0xDC9411] "))
plot(USA)
```

## How it works...

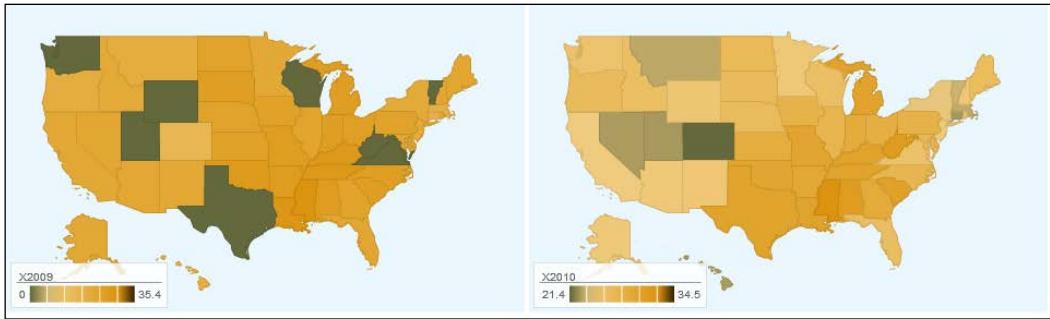
The first three arguments used in the `gvisGeoMap()` function are `data`, `locationvar`, and `numvar`. These arguments are discussed in detail in the previous recipe.

Many of the attributes used under the `options` argument are self-explanatory. The `dataMode` attribute passed in the `options` argument is `regions` as we would like to display the map of a region or a specific country and not the entire world. The `region` attribute is assigned `US`, which instructs the `googleVis` package to plot a map of the USA with boundaries. Readers can display the map of Australia by typing `AU` instead of `US`.

## There's more...

Many times, it is useful to study the spatial data over time rather than at one point in time. As we have geographic data over time, we can compare the change between two periods or even more, and observe how the data has evolved.

In the following visualization, I have displayed the prevalence of obesity for the period 2009-2012 on four separate maps (the code will generate 4 different maps— for instance, two of them are as shown in the following output). We can clearly see that the prevalence of obesity has reduced from 2009 to 2012 and also the reduction is more evident in the western states.



The following code is the same as the one explained in the *How to do it...* section of this recipe. We have used the `gvisMerge()` function to merge the four distinct maps and displayed them using the `plot()` function:

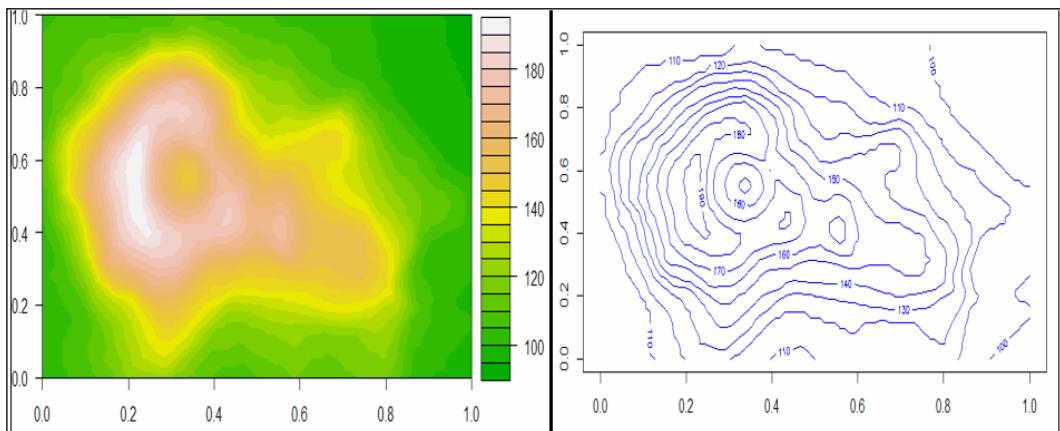
```
obese = read.csv("obesity.csv", header = TRUE, sep = ",")
US_2012 <- gvisGeoMap(obese, locationvar="state",numvar ="X2012",
options = list(width = 500, height =300,dataMode = "regions",
region = "US",colors=
 "[0x64693D,0xF1CC80,0xEABA58,0xE3A732,0xDC9411]"))
US_2011 <- gvisGeoMap(obese, locationvar="state",numvar ="X2011",
options = list(width = 500, height =300,dataMode = "regions",
region = "US",colors=
 "[0x64693D,0xF1CC80,0xEABA58,0xE3A732,0xDC9411]"))
US_2010 <- gvisGeoMap(obese, locationvar="state",numvar ="X2010",
options = list(width = 500, height =300,dataMode = "regions",
region = "US",colors=
 "[0x64693D,0xF1CC80,0xEABA58,0xE3A732,0xDC9411]"))
US_2009 <- gvisGeoMap(obese, locationvar="state",numvar ="X2009",
options = list(width = 500, height =300,dataMode = "regions",
region = "US",colors=
 "[0x64693D,0xF1CC80,0xEABA58,0xE3A732,0xDC9411]"))
merged = gvisMerge(gvisMerge(US_2009,US_2010, horizontal =
 TRUE),gvisMerge(US_2011,US_2012, horizontal = TRUE))
plot(merged)
```

## See also

- ▶ The *Variation in Government Aid Across Nations* article at <http://www.nytimes.com/interactive/2009/05/09/us/0509-safety-net.html>
- ▶ The *MMMap of Baseball Nation* article at <http://www.nytimes.com/interactive/2014/04/24/upshot/facebook-baseball-map.html#3,51.111,-95.503>
- ▶ The *Where are the Hardest places to Live in the US?* article at <http://www.nytimes.com/2014/06/26/upshot/where-are-the-hardest-places-to-live-in-the-us.html?abt=0002&abg=1>

## A guide to contour maps

Contour maps have also been referred to as isometric maps or isopleth maps. Contour maps are used to display data related to temperature or topographic information. Most of the time, contour maps are used to display maps related to geology or mountains. The contour lines represent which sections of the mountains are steeper based on how close the contour lines are drawn from each other.



## How to do it...

We can generate a contour map in R using the `contour()` function and also add a color to the contour using the `filled.contour()` function:

```
contour(volcano, main = "Topographic map of a Volcano",
 col = "blue")
filled.contour(volcano, color.palette = terrain.colors,
 main = "Topographic map of a Volcano")
```

## How it works...

In order to plot a simple contour plot, we will use the `contour()` function available with the basic R plotting functionality. The first argument passed in the `contour()` function is the data. The volcano dataset consists of topographic information on Auckland's Mauga Whau volcano. The dataset consists of 87 rows and 61 columns. All the other arguments relate to the basic R plotting function.

To plot a contour plot with a range of colors, we can use the `filled.contour()` function. The first argument in the `filled.contour()` function is the `volcano` dataset, but we have made use of the `color.palette` argument to assign color to our contours:

```
filled.contour(volcano, color.palette = terrain.colors,
 main = "Topographic map of a Volcano")
```

The `terrain.colors` value is used as an option for the argument within `color.palette`. We can also use some of the predefined color palettes (`topo.colors` or `heat.colors`), as shown in the following code:

```
filled.contour(volcano, color.palette = topo.colors,
 main = "Topographic map of a Volcano")
filled.contour(volcano, color.palette = heat.colors,
 main = "Topographic map of a Volcano")
```

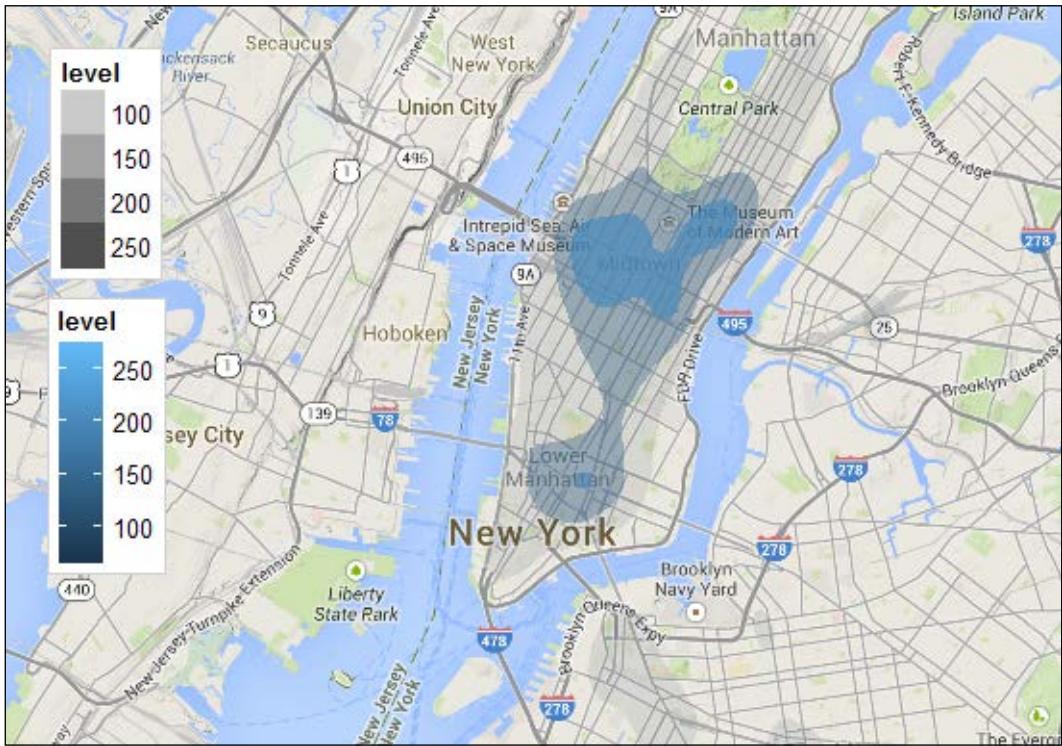
Readers can learn more about how they can generate custom color palettes in R by typing `?colorRamp` in the R console window. We can also specify a range of colors using the `RColorBrewer` package:

```
library(RcolorBrewer)
custom = brewer.pal(9, "BuPu")
filled.contour(volcano, col = custom, main = "Topographic
map of a Volcano")
```

We can define our color palette using the `brewer.pal()` function.

## There's more...

The following map was motivated by an article published in the R Journal. The data for the contour map is downloaded from the New York City open data website. The Excel file is fairly large, so it might slow down your system in case you try to run the code.



The following code is inspired from the article by Kahle and Wickham (2013):

```
install.packages("ggmap", dependencies = TRUE)
library(ggmap)
colide = read.csv("collisions.csv", header = TRUE, sep = ",")
nyc = get_map("newyorkcity", zoom = 12)
nycmap = ggmap(nyc, extent = "device", legend = "topleft")
nycmap +
 stat_density2d(
 aes(x = LONGITUDE, y = LATITUDE, fill = ..level.., alpha =
 ..level..),
 size = 2, bins = 4, data = colide,
 geom = "polygon"
)
```

The `get_map()` function allows us to get the data from Google maps and open streetmaps, stamen maps, and cloudmade maps. The `get_map()` function can be used to get any map based on the address or name of the location. The level of zoom can also be specified in the `get_map()` function. We assign the data to a variable `nyc`.

```
nyc = get_map("newyorkcity", zoom = 12)
```

We would now use the `ggmap()` function to plot the data generated using the `get_map()` function:

```
nycmap <- ggmap(nyc, extent = "device", legend = "topleft")
```

Finally, we use the `stat_density2d()` function to plot the contour map in R. The first argument in the `stat_density2d()` function is `aes` or `aesthetics`, which describes the variables in the data that are mapped. The `geom` argument corresponds to the geometric shape that will be used to construct the contour plot:

```
stat_density2d(
 aes(x = LONGITUDE, y = LATITUDE, fill = ..level.., alpha =
 ..level..),
 size = 2, bins = 4, data = colide,
 geom = "polygon")
```

Readers can learn more about the `stat_density2d()` function as well as the `aes` and `geom` arguments under the `ggplot2` manual available on the CRAN website.

## See also

- ▶ New York City open data at <https://nycopendata.socrata.com/>
- ▶ The *Mapping the Spread of Drought across the USA* article at <http://www.nytimes.com/interactive/2014/upshot/mapping-the-spread-of-drought-across-the-us.html?abt=0002&abg=1>
- ▶ *ggmap: Spatial Visualization with ggplot2*, D Kahle and H (2013), which can be retrieved from <http://journal.r-project.org/archive/2013-1/kahle-wickham.pdf>

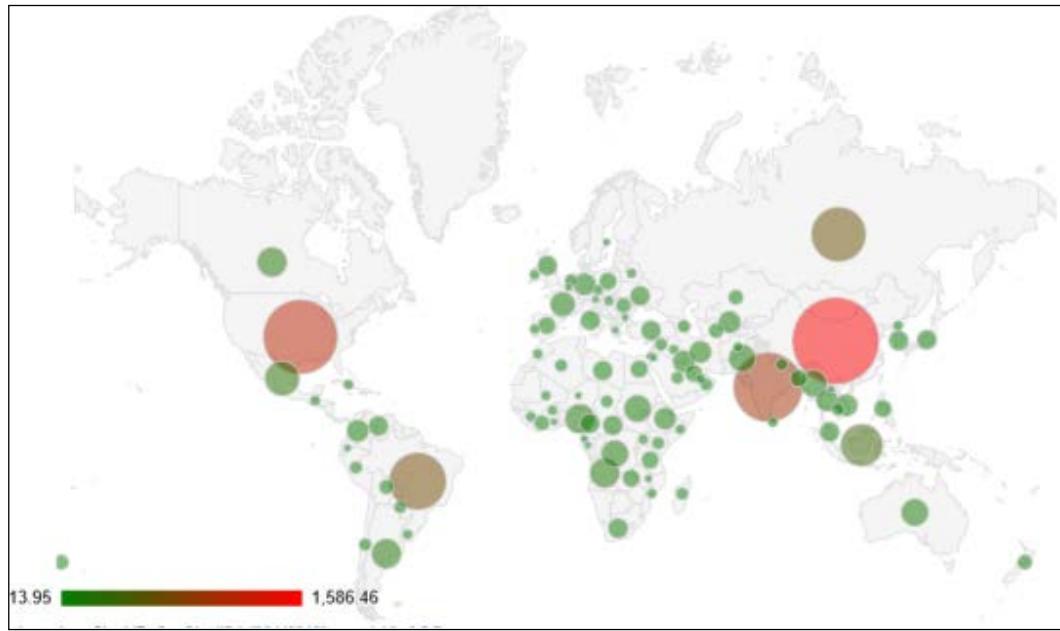
## Constructing maps with bubbles

You might have come across maps that look like choropleth maps, but for each region, there is a bubble or a pie chart that represents percentage. I personally prefer bubbles over pie charts on a map, but in certain cases where we need to identify data based on gender or some other categorical variable, pie charts can come in handy as well. The code to plot a pie on a map is discussed in the recipe *Creating donut plots and interactive plots* in Chapter, *The Pie Chart and Its Alternatives*.

The main motivation behind this plot is also derived from the New York Times infographic on global warming. The visualization displays a line plot along with the bubble plots. The line plot can be accessed by clicking the navigation tab on the chart. We observe that even though on the bubble plot it might look like China is a big contributor to greenhouse gases, the line plot draws a different picture; on a per capita basis, China's contribution is lower than that of the USA.

It is generally a good practice to combine visualizations to make sure that the audience has the correct idea of what the data is trying to portray. The New York Times uses bubbles on maps to depict some interesting analysis on campaign spending and the treasury's TARP spending.

The following visualization shows the 100 most greenhouse gas emitting countries in million metric tons. The colors and size of the bubbles in the map denote the amount of carbon emissions. Alternatively, we could have generated the same plot with size representing a different variable such as population.



## Getting ready

We need to install and load the `googleVis` package in R.

## How to do it...

We will first import the `ghg.csv` file, which contains our data in R, using the `read.csv()` function:

```
emisn = read.csv("ghg.csv", sep = ", ", header = TRUE)
```

We can examine the data using the `head(emisn)` function. The dataset consists of the top 100 greenhouse gas emitting countries in 2013.

We can now generate the visualization using the `gvisGeoChart()` function. We can display our visualization in R using the `plot()` function:

```
emit = gvisGeoChart(emisn, locationvar = "Country",
 sizevar = "Emission_in_2010",
 options = list(displayMode = "markers", width = 900, height
 =500,
 markerOpacity = 0.5, sizeAxis = "{maxSize:'35'}",
 colorAxis = "{colors:['green','red']}"))
plot(emit)
```

## How it works...

The first argument in the `gvisGeoChart()` function corresponds to the data frame that contains the data. The `locationvar` argument comes with many different options and one of the options is the name of the country. In the data frame `emisn`, the country names are stored under the column named `Country`. Readers can also specify latitude and longitude information; the details for the same are discussed in the `googleVis` package manual.

The `sizevar` argument is used to specify the size of the bubble. This argument can be used only when we use `displayMode = 'markers'`.

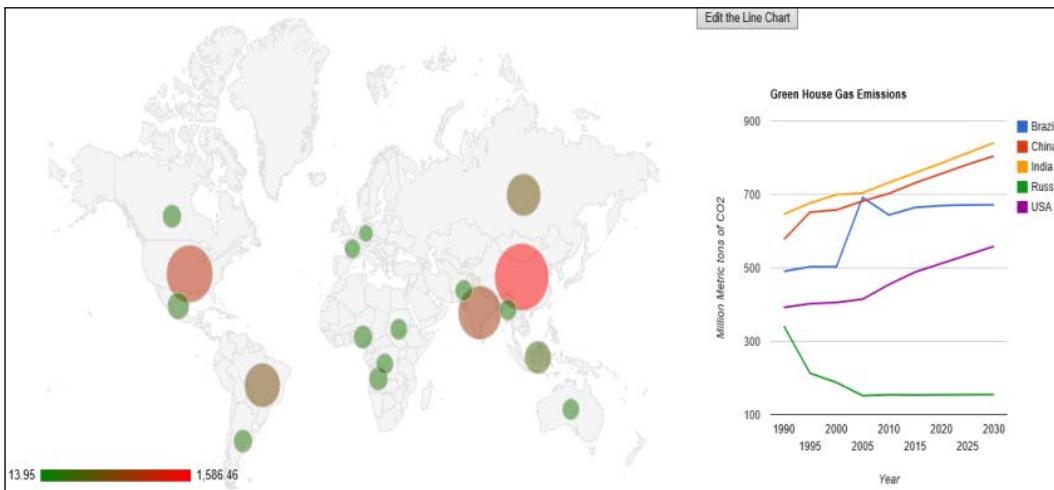
The `options` argument is used to set various options related to the visualization, such as `width`, `height`, and `color`. The `googleVis` package also allows us to set the maximum and minimum radius of the bubbles using the `sizeAxis` option. Readers should note that the `title` option is not implemented under the `gvisGeoChart()` function.

Readers can also include a column that contains text and pass it under the `hovervar` argument. The text will be displayed when the mouse hovers over the bubble.

The `gvis Methods` section under the manual extensively discusses various methods to export the visualization, integrate it with `shiny`, or create a `markdown` file in R.

## There's more...

In the following visualization, we have generated a line chart as well as a bubble plot. We have integrated the two charts using the `gvisMerge()` function.



The following code contains the `gvis.editor` argument. The argument is useful while plotting line charts or histograms, as it allows the audience to click the button and change the chart type or edit various elements within the visualization:

```
library("googleVis")
emisn = read.csv("ghg.csv", sep = ",", header = TRUE)
emit = gvisGeoChart(emisn, locationvar = "Country",
 sizevar = "Emission_in_2010",
 options = list(displayMode = "markers", width = 900,
 height = 500,
 markerOpacity = 0.5, sizeAxis = "{maxSize:'35'}",
 colorAxis = "{colors:['green','red']}"))
emisn1= read.csv("ghg1.csv")
emit1 = gvisLineChart(emisn1, xvar = "Year",
 yvar=c("Brazil","China","India","Russia","USA"),
 options = list(width = 500, height =
 500, title ="Green House Gas Emissions",
 vAxis = "{title:'Million Metric tons of CO2'}",
 hAxis = "{title:'Year'}", gvis.editor = "Edit the Line Chart"))
merge = gvisMerge(emit,emit1, horizontal = TRUE)
plot(merge)
```

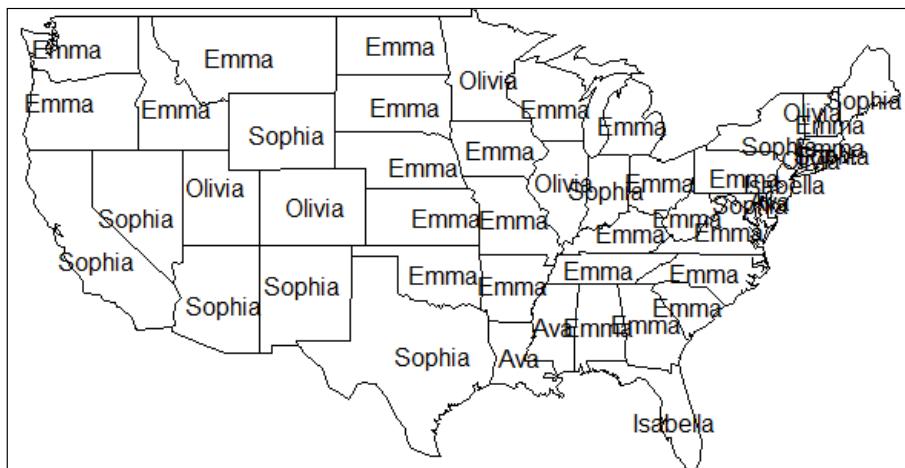
## See also

- ▶ The New York Times visualization on *Copenhagen: Emissions, Treaties, and Impacts* at <http://www.nytimes.com/interactive/2009/12/05/world/climate-graphic-background.html>
  - ▶ The *A Banner Year for Political Spending* article at <http://www.nytimes.com/interactive/2010/10/26/us/politics/campaign-fundraising-roundup.html>
  - ▶ The *Keeping Tabs on the \$700 Billion Bailout* article at [http://www.nytimes.com/imagepages/2008/12/06/business/20081206\\_METRICS\\_GRAPHIC.html](http://www.nytimes.com/imagepages/2008/12/06/business/20081206_METRICS_GRAPHIC.html)
  - ▶ The *Moving To and From New York City* article at <http://www.nytimes.com/interactive/2009/06/14/nyregion/0614-migration.html>

# Integrating text with maps

Overlaying maps with text is not a very prominent medium of displaying information. However, in recent times, with the popularity of social networking sites such as Twitter and Facebook, visualizations that integrate text with maps have become somewhat popular. It is possible to extract real-time information using APIs listed in the *See also* section of this recipe and display them using maps or some other medium. We have discussed the idea of extracting information from the Web using XML in the recipe *A basic introduction to API and XML in Chapter, Creating Applications in R*.

In this recipe, we will briefly discuss how one can plot text over maps. Readers are encouraged to use the `twitteR` package to extract information and follow this recipe to generate similar visualizations.



# Getting ready

To generate the map in R, we need to install and load the `maps` package:

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

## How to do it...

We will import our data in R using the `read.csv()` function. The file contains the most popular female baby names by state for 2013. The data file also contains the average latitude and longitude of each state:

```
names = read.csv("names.csv")
```

Next, we use the `map()` function to display the map of the USA. The argument `state` instructs the `map` package to generate a map of the USA states with boundaries. Readers can learn more about the `map()` function by typing `?map` in the R console window:

```
map("state")
```

We would like to plot the most popular female baby names in each state. The `text()` function allows us to plot text over a plot. The first and second arguments in the `text()` function are `x` and `y` axes. The third argument is the label to be applied. The following code applies the `for` loop in R:

```
for(i in 1: 50){
 text(names$lon[i], names$lat[i], names$name[i], adj= 0.5)
}
```

We are instructing R to loop over the `text()` function and plot the baby names. As we are using a loop function, we will use the `[i]` notation to plot the baby names at the average latitude and longitude.

We can use the following code to place the baby name EMMA over Alabama:

```
text(-86.8073, 32.799, "EMMA")
```

We would simply use a loop to plot all the 50 names instead of typing 50 lines of code.

## See also

- ▶ The *Good Morning!* post at <http://blog.blprnt.com/blog/blprnt/goodmorning>
- ▶ The *Just Landed* article at <http://blog.blprnt.com/blog/blprnt/just-landed-processing-twitter-metacarta-hidden-data>

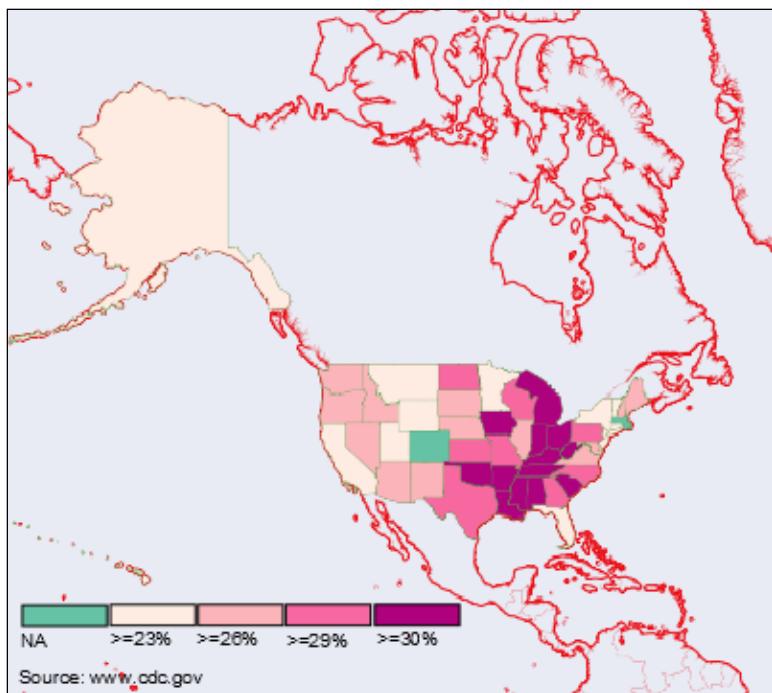
- ▶ The *Newspaper Endorsements: A Final Tally* article at <http://www.nytimes.com/interactive/2008/11/03/us/politics/20081103-endorse-graphic.html>
- ▶ The *Twitter Chatter During Super Bowl* article at [http://www.nytimes.com/interactive/2009/02/02/sports/20090202\\_superbowl\\_twitter.html](http://www.nytimes.com/interactive/2009/02/02/sports/20090202_superbowl_twitter.html)

## Introducing shapefiles

The United States Census Bureau defines a shapefile as "A *shapefile* is a geospatial data format for use in geographic information system (*GIS*) software. Shapefiles spatially describe vector data such as points, lines, and polygons, representing, for instance, landmarks, roads, and lakes". Shapefiles are used extensively to store spatial information and can be used to plot data on maps. We can easily download many different shapefiles for the USA counties or states from the USA Census Bureau website ([www.census.gov](http://www.census.gov)). The shapefiles are downloaded as a folder, which comprises files with various extensions such as .shp, .dbf, .prj, .shx, and .xml. For this recipe, we will only use the .shp extension.

The shapefile package in R can be used to read a shapefile, add the processed data to our shapefile, and then save it in the shapefile format. When you save a shapefile in R or any other software, it will create other files that support the shapefile. However, for this recipe, instead of R we will use QGIS, which is an open source package for the creation of maps.

In this recipe, we will learn to join our data to a shapefile.



# Getting ready

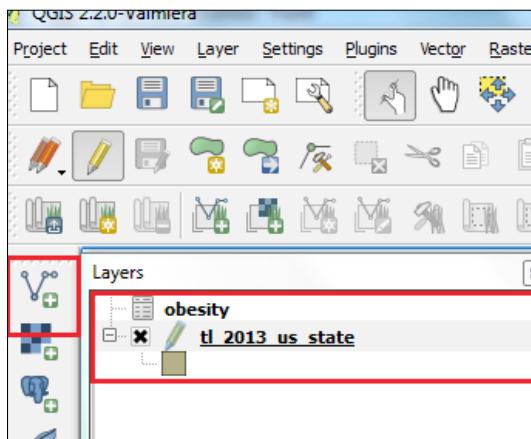
We will use the following open source software for creating the map:

- ▶ QGIS available at <http://www.qgis.org/>
- ▶ Tile Mill available at <https://www.mapbox.com/tilemill/>
- ▶ Inkspace available at <http://www.inkscape.org/>

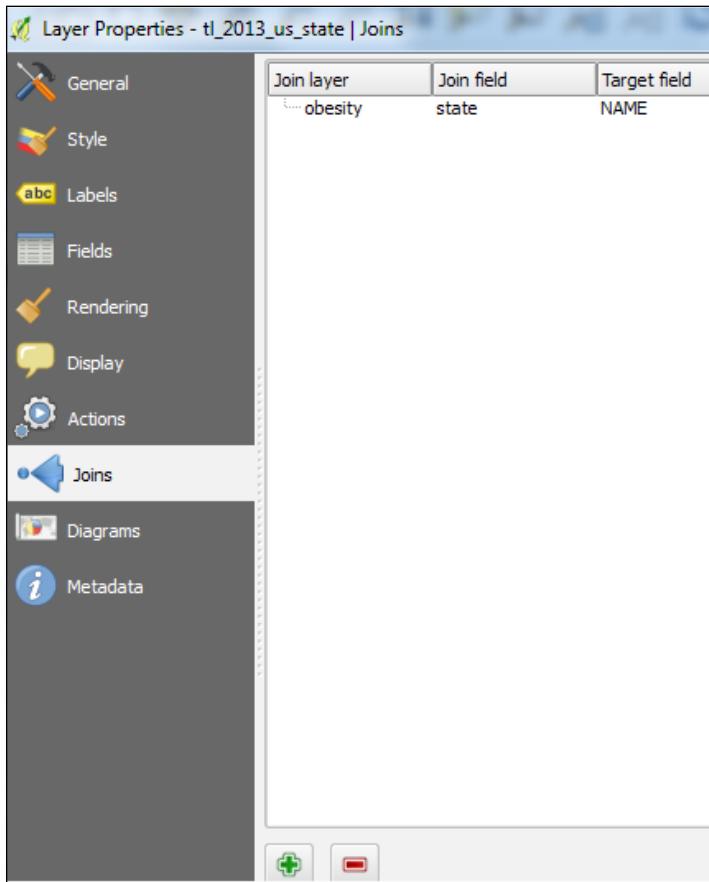
## How to do it...

For this recipe, we download the data from the Center for Disease Control website (<http://www.cdc.gov/>) and join our dataset to the shapefiles. Next, we create a choropleth map using Tile Mill. Finally, we will add legends to our map using Inkspace:

1. We will use the obesity data available in our `Chapter_3` folder. The QGIS will identify the columns with values as strings, but we would like to avoid this. We want QGIS to identify the columns containing our dataset as `Real` and not `Strings`. We will first open our dataset, select all the columns that contain the data, right-click, select the **Format** option, and transform our data to numbers. Then save the dataset in the `.xls` format. The `.csv` format will not recognize the number and, hence, we will use the `.xls` format.
2. We have already downloaded the shapefile from the USA Census Bureau website (<http://www.census.gov/>). These are stored under the `Folder State` under the data folder. Now let's fire up QGIS. In order to join our obesity data to the shapefile, click on the `add vector` image (as shown in the following screenshot). Now browse to the folder and click on the file that contains the `.shp` extension. This will load the USA map with all the states. We will follow the same procedure and load our `.xls` file from the data folder. You will observe the two files on the top-left corner, as shown in the following screenshot:

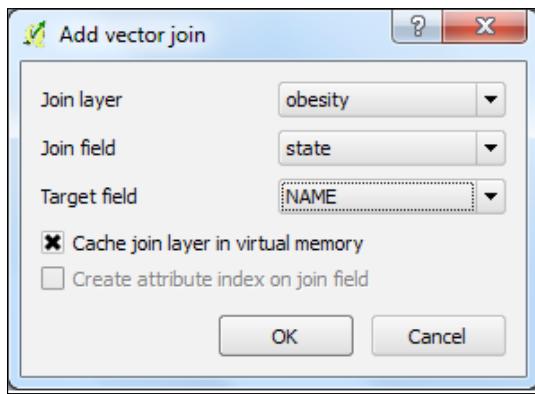


3. You can check your data by right-clicking on the name and selecting **Open Attributes Table**. Now select the shapefile and double click on it; this should open up a layer properties dialogue box, as shown in the following screenshot:



If you click on **Fields**, you will observe that some values are classified as **Strings** and some as **Real**. When we finish joining our data and saving a new shapefile, we would like the values to be identified as **Real** as this makes it easier to plot it in Tile Mill or create a cartogram.

4. Now click on the small + sign at the bottom of the **Joins** option. This will open up another dialogue box, as shown in the following screenshot:



Note that between obesity and shapefile, we need one common column to join the data. In our case, obesity has the names of states and shapefile has this information as well. We can observe this under the **Open Attributes Table**, discussed in step 3. We would like to join the obesity data; hence the **Join layer** dropdown contains the **obesity** layer selected.

Next, we would like to join the column named **state** in the obesity data to the **NAME** column in the shapefile. Once we select this, click on **OK**, click on **Apply**, and then click on **Save**.

If we now click on the shapefile and open the attributes table, you will observe that QGIS has joined the obesity data to our shapefile. Now select the shapefile and right-click to save it as a new file. I have saved this new shapefile under the folder **obesity1**. Note that QGIS will create additional files, but for our purpose we only need the **.shp** extension file. Also, it is very important to go into the layer properties of our new shapefiles and check the fields; all the data should have Real and not String.

5. We can now use the obesity data in R or Tile Mill to add color to the map or do further analysis. We will use the new shapefile to create a cartogram discussed in the next recipe. The image of the USA obesity rate in the beginning of this recipe was created using Tile Mill. Tile Mill is a great tool and very easy to learn. Tile Mill manuals and software can be downloaded from the link <https://www.mapbox.com/tilemill/>.

## See also

- ▶ The United States Census Bureau documentation related to shapefiles at <https://www.census.gov/geo/maps-data/data/tiger-line.html>

# Creating cartograms

Cartograms are distorted shapes of geographical areas. The distortion is based on a set of data encoded in a map. The idea of a cartogram is to show the gravity of the issue or data being studied. Mark Newman at the University of Michigan has created some of the best cartograms. Very interesting implementations of cartograms are the Elemental Cartograms, where the shape of the periodic table is distorted based on the user data.

In the visualization, we have distorted the map of the USA based on the prevalence of obesity in each state. We observe that the boundaries and area are connected but the geographical accuracy is lost as the shape is distorted.

Yau (2013) states the advantage of using cartograms over choropleth maps as "..., the upside of cartograms is that areas fill appropriate amounts of space, but the trade-off is less geographic accuracy. When your data is for larger regions, with wide range of sizes, this trade-off is worth it, but when regions are uniform in size, a choropleth map is most likely a better fit."

For the purpose of the current recipe, we will use an open source tool called ScapeToad. ScapeToad is a cross-platform, open source application written in Java. The tool can be easily downloaded at <http://scapetoad.choros.ch>. The ScapeToad website provides instructions as well as some examples on generating cartograms. I have not been very successful in finding a good package that allows us to generate cartograms using R and, hence, I divert the readers to using ScapeToad.



# Getting ready

In order to create a cartogram, we need to download the ScapeToad software.

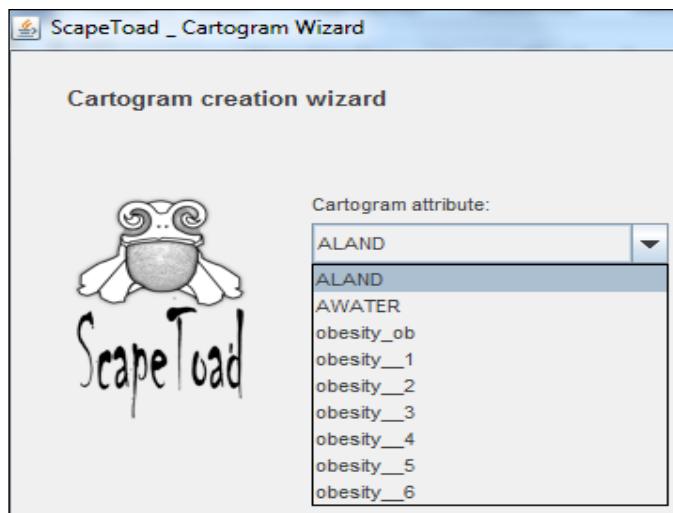
## How to do it...

We already have a shapefile created from the previous recipe; we will use the same shapefile to create the cartogram. Download the ScapeToad tool and open the file titled `scapetoad.exe`.

Click on **Add layer** and navigate to the folder that contains our shapefile, namely, `obesity1.shp`.

Now you should observe the map of the USA in the ScapeToad window. Next, we click on **Create cartogram**. This will open up a wizard. The most important step in this wizard is where the user needs to select the data column to be used to create the cartogram. We will select **obesity\_ob** from the dropdown, which contains the data for 2012.

The reason we observe all the columns (in the drop-down list) is due to the fact that in QGIS, we forced the columns containing our data to be real. If you omit this step, you will not see the data header as a selection.



Under the wizard, you will observe a transformation quality option (I used the **High** option). Once you click on **Compute**, ScapeToad will create a cartogram. Note that ScapeToad is a bit slow and might take some time to compute.

Once computation finishes, you will observe the cartogram in the window with a grid. We can now either save it as a shapefile and edit it further in Tile Mill or save it as an SVG and edit it in Inkspace. The header and source information was added by me using Inkspace.

## See also

- ▶ Maps of the 2012 US presidential election results by Mark Newman at <http://www-personal.umich.edu/~mejn/election/2012/>. The link also provides information and code for the cartogram algorithm.
- ▶ The Worldmapper webpage has many different examples of cartograms; it is available at <http://www.worldmapper.org/index.html>.
- ▶ Even though this does not relate directly to maps, this link provides tools to generate distorted shapes of a periodic table based on data. It is accessible at <http://bsanii.jsd.claremont.edu/ElementalCartograms.html>.
- ▶ The link [http://spatial.ly/2013/06/r\\_activity/](http://spatial.ly/2013/06/r_activity/) uses cartograms to visualize data related to the R activity in the world. The data is processed using R and cartograms are generated using ScapeToad.
- ▶ *Data Points – Visualization that means something, Nathan Yau (2013), Wiley.*

# 4

# The Pie Chart and Its Alternatives

In this chapter, we will cover the following recipes:

- ▶ Generating a simple pie chart
- ▶ Constructing pie charts with labels
- ▶ Creating donut plots and interactive plots
- ▶ Generating a slope chart
- ▶ Constructing a fan plot

## Introduction

Pie charts, as mediums of data representation, have been widely used in news media, corporate presentations, and research papers. Pie charts have been used in the past to compare data between two time periods as well as to represent parts of a whole in a single pie. However, even with such widespread use, they have been criticized by statisticians. The blog post at <http://www.r-chart.com/2010/07/pie-charts-in-ggplot2.html> summarizes the main criticism as follows:

- ▶ *The relative size of each slice is difficult to interpret. Studies have shown that pie charts are hard to read*
- ▶ *Pie charts require too much space to present too little information*
- ▶ *They are frequently rendered in 3D (this makes the previous two issues worse)*

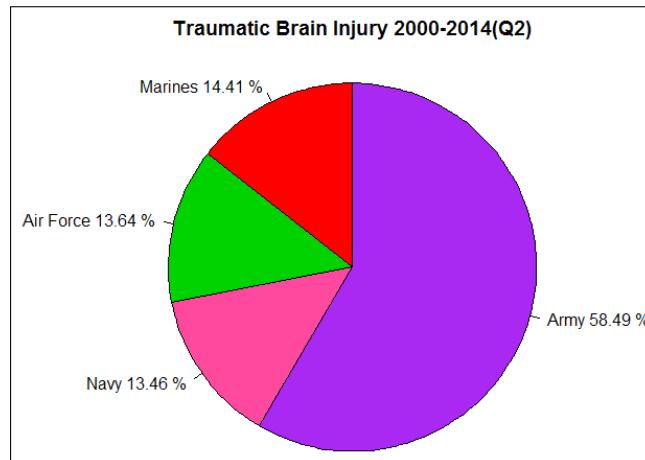
The main criticism of pie charts is that they are difficult to interpret, especially when the difference between two slices of a pie is small. Human beings are not trained well to observe angles; hence, many times, pie charts might even be interpreted wrongly. Even with all their limitations, pie charts are hard to avoid.

The current chapter can be divided into two main sections. The first section deals with pie charts, interactive pie charts, and donut plots. We also introduce the idea of a part of a whole as an alternative to pie charts. The second section delves into some better alternatives such as fan plots, stacked bar charts, and slope charts. The recipes in this chapter are implemented using the plotrix, RColorBrewer, and googleVis packages available in R. If readers get an error while generating the plotrix chart, I would recommend that they clear the plot area, type `plot.new()` in the R console, and re-run the code.

## Generating a simple pie chart

Even after being criticized by many statisticians as a means to deliver information or present data, pie charts have been used widely by companies in their annual reports or by businessmen for presentation purposes. According to Schwabish (2014), *Pie charts force the readers to make comparisons using the area of the slices or the angles formed by the slices—something our visual perception does not accurately support; they are not an effective way to communicate results.*

One way to overcome this criticism is by sorting our data from the highest to the lowest and displaying percentages. The New York Times visualizations implement pie charts to represent data, but the pie charts are also accompanied by bar charts and area charts to provide additional context to the data. The following pie chart shows data on brain injury across different branches of the military:



## How to do it...

The data used for creating the pie chart was extracted from <http://fas.org/sgp/crs/natsec/RS22452.pdf>. The base graphing device in R allows us to generate simple pie charts quickly. We use the `c()` function to create a vector of total traumatic brain injuries reported from 2000-2014 (Q2).

```
data = c(179718, 41370, 41914, 44280)
```

We would prefer to include percentages instead of actual values and hence create another vector `pct` using our data vector. The `sum()` function is similar to the sum function in MS Excel.

```
pct = (data/sum(data))*100
pct = round(pct, 2)
```

The `round()` function simply reduces the decimal places. As we are representing percentages, we use two decimal places. We also would like to add actual labels to our pie and hence we create a character vector using the `c()` notation.

```
labels = c("Army", "Navy", "Air Force", "Marines")
```

The `paste()` function allows us to paste our data to the `labels` vector and add the percentage (%) sign.

```
labels = paste(labels, pct, "%")
```

We also generate a vector of color. This is a character vector similar to `labels`.

```
col = c("purple", "violetred1", "green3", "red", "cyan")
```

To create a simple pie, we use the `pie()` function.

```
pie(pct, col = col, radius = 1, init.angle = 90, clockwise = TRUE,
 labels = labels, main = "Traumatic Brain Injury 2000-2014 (Q2)")
```

## How it works...

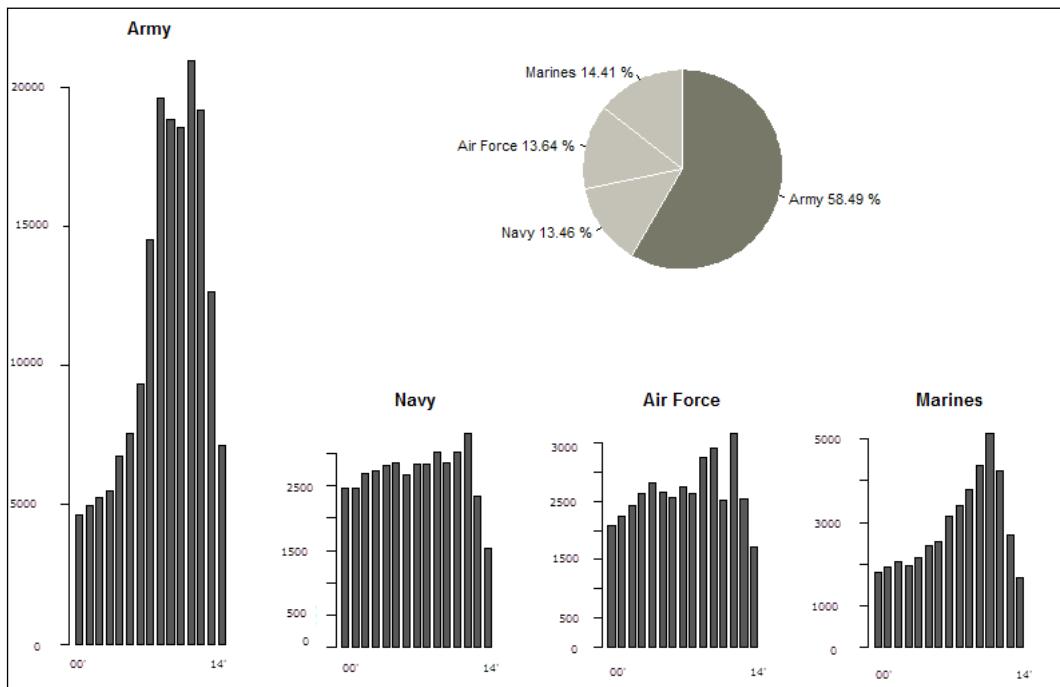
A pie chart will not plot the labels automatically; hence, we need to undertake a few additional steps to generate labels that can be applied to our pie. We would like to create a label with the name of the service, followed by the value, followed by the percentage sign. We convert our actual values to percentages and store them in a vector `pct`, but we would also prefer to plot only two decimals and hence we use the `round()` function so as to have `pct` vector with only two decimal places.

The `paste()` function is very handy to paste everything together and create our final label for the pie chart. The arguments in the `paste` function need to follow the order in which the labels have to be created, that is, the name of the service, value, and sign.

Now we are ready to create our pie chart using the `pie()` function. The first argument in our `pie` function is a vector of values to be used; in our case, it is simply the `pct` vector. The second argument is the color used to fill the slices. The `radius` argument can be adjusted to resize our pie chart. The `init.angle` argument is used to specify the starting angle of the slice. The `clockwise` argument indicates whether the slices are drawn clockwise or counterclockwise. The `labels` arguments is where we would use our `labels` vector generated using the `paste()` function.

## There's more...

Both the New York Times visualizations use pie charts along with bar plots and informative text. This adds more context and facilitates interpretation of the visualization. Note that the code mentioned in the `code.txt` file of this chapter will generate the basic structure of the plot. I have used Inkspace to edit the labels and axis. This can be done in R as well, but using Inkspace saves us a lot of time.



The code used is as follows:

```
data2 = read.table("service.csv", header = TRUE, sep = ",")
army = as.matrix(data2[1,2:16])
navy = as.matrix(data2[2,2:16])
airforce = as.matrix(data2[3,2:16])
marine = as.matrix(data2[4,2:16])
j = layout(matrix(c(1,2,2,2,1,3,4,5),2,4, byrow = TRUE))
layout.show(j)
barplot(army, names.arg= c(2000:2014), space = 0.5, main = "Army")
pie(pct, radius = 1, init.angle = 90, clockwise = TRUE, labels
=labels, border = "white", col=
c("#6C6D5D","#BEBCB0","#BEBCB0","#BEBCB0"))
barplot(navy, names.arg= c(2000:2014), space = 0.5, main = "Navy")
barplot(airforce, names.arg= c(2000:2014), space = 0.5, main =
"Air Force")
barplot(marine, names.arg= c(2000:2014), space = 0.5, main =
"Marines")
```

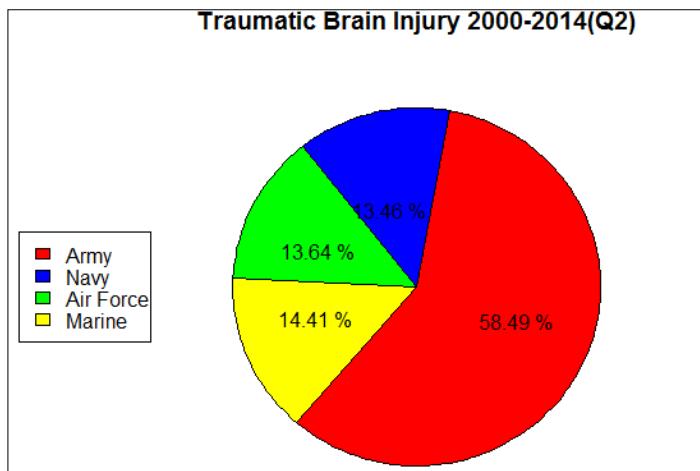
The preceding code is very similar to the code in the *How to do it...* section of this recipe. While implementing the `as.matrix()` function, we have manipulated our data and coerced our series to be in the vector format. To understand the use of the `[]` notation, please refer to the *Editing a Matrix in R* recipe in *Chapter, A Simple Guide to R*. The `barplot()` function has been described in detail in *Chapter, Basic and Interactive Plots*. Readers can either follow the comments in the `code.txt` file or type `?layout()` in the R console window to understand its use in the previous plot.

## See also

- ▶ Jonathan Schwabish, *An Economist's Guide to Visualizing Data* (2014).at <http://pubs.aeaweb.org/doi/pdfplus/10.1257/jep.28.1.209>.
- ▶ The New York Times visualization on consumption of energy using pie charts along with area and bar charts to tell a story. It can be accessed at <http://www.nytimes.com/imagepages/2009/09/19/business/20090920EFFICIENCY-graphic-ready.html>.
- ▶ The New York Times visualization on death by service also uses pie charts as a medium to represent data. It is available at <http://www.nytimes.com/imagepages/2008/08/07/us/07afghanService.GR.ready.html>.

# Constructing pie charts with labels

We go a step further in this recipe and generate a pie chart where the labels are inside the pie rather than outside. I have used the `plotrix` package to construct a pie chart with legends and labels inside the pie. The previous pie chart is represented with labels as shown:



## Getting ready

We must install the `plotrix` package for this recipe. Readers should refer to the recipe *Installing Packages and Getting Help in R* in Chapter, *A Simple Guide to R*.

## How to do it...

We have explored in detail, in the previous recipe, the method to generate labels for our plot using the `paste()` function. The following lines of code have not changed from our previous recipe:

```
data = c(179718,41370,41914,44280)
pct = (data/sum(data))*100
pct = round(pct,2)
labels = c("Army", "Navy", "Air Force", "Marines")
labels1 = paste(pct, "%")
```

We will now create a blank plotting area using the generic `plot()` function. We include this step as we are required to specify the coordinates to construct a pie chart using the `floating.pie()` function.

```
plot(1:5,type="n",main="Traumatic Brain Injury 2000-
2014 (Q2)",xlab="",ylab="",axes=FALSE)
```

We now create a pie using the `floating.pie()` function and assign it to `pie1`. We would like to plot the labels inside the pie, and the `pie1` object will be used as an argument in the `pie.labels()` function. The `pie.labels()` function allows us to add labels to our pie chart.

```
pie1<-floating.pie(3,3,pct, radius=1,
 col=c("red","blue","green","yellow"), startpos = 4)
pie.labels(3,3,pie1, radius=0.4, labels=labels1)
```

We can now add legends to our plot using the basic R `legend()` function:

```
legend("left", fill = c("red", "blue", "green", "yellow"), legend =
 c("Army", "Navy", "Air Force", "Marine"))
```

## How it works...

The `plot()` function is used to create a blank plot. We can suppress the plot using the argument `type = "n"`. All the available type options can be viewed by typing `?plot()` in the R console window.

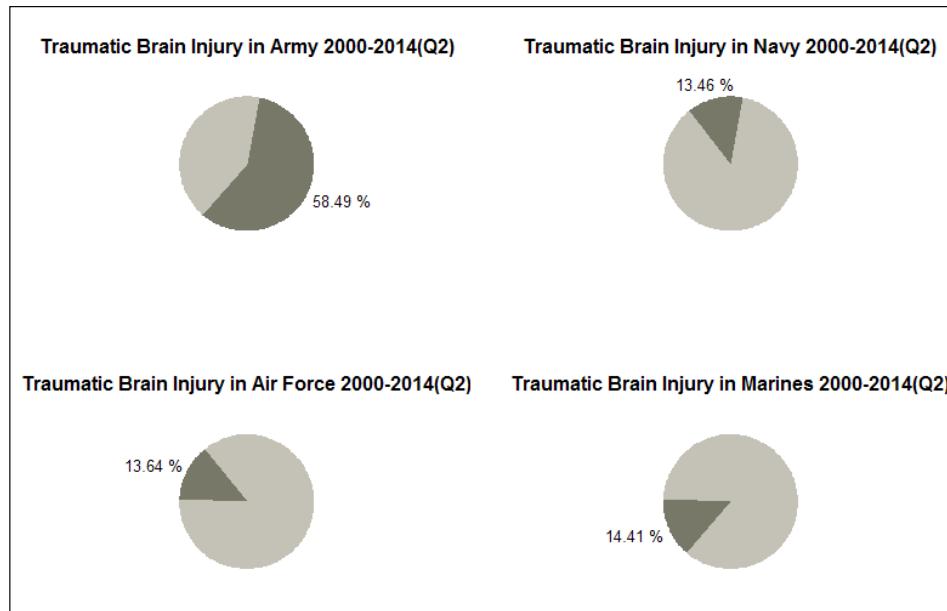
The first two arguments in the `floating.pie()` function are the `x` and `y` coordinates of the center of the pie. Note that this is the reason we generated a blank plot. The third argument `pct` is our vector of data. The `radius` and `col` arguments are the same as in the `pie()` function discussed in the previous recipe. The `startpos = 4` argument is the starting position in radians; it is very similar to the `clockwise` argument in the `pie()` function.

Once we have created a pie chart, we add the labels using the `pie.labels()` function. The first two arguments in the `pie.labels()` function are the `x` and `y` coordinates, and we keep these the same as the `floating.pie()` function. The third argument is the `angles` argument. The `floating.pie()` function returns the angles, and hence, we use the `pie1` object as our angle. The `radius` is lower than 1 unit as we would like the values to be plotted inside the pie and not outside. The `labels` argument is self-explanatory.

As our last step, we would add a legend to our plot. This is necessary as we are not plotting the names of each slice but just the values. The first two arguments in the `legend()` function are the `x` and `y` coordinates. However, we can also specify the position verbally, for example `left`. The `fill` argument fills the colors in the legends and the `legend` argument applies labels to each filled box in the legend section. To explore all the available options for legends, readers should type `?legend` in the R console window.

## There's more...

- The idea behind part-to-whole judgments is very simple. Instead of making one pie chart that makes comparison difficult, we can construct multiple pie charts, where each slice represents the area it occupies compared to other slices in the pie chart.



The code to create multiple pie charts might look complicated at first, but observe that we have generated one pie and repeated the same section of code to create three other pie charts, just by changing some key index values. The following code is used to achieve this:

```
par(mfrow =c(2,2))
plot(1:5,type="n",main="TBI in Army 2000-
2014 (Q2)",xlab="",ylab="",axes=FALSE)
pie1<-floating.pie(3,3,pct, radius=0.8,
col=c("#6C6D5D", "#BEBBCB0", "#BEBBCB0", "#BEBBCB0"),
border = FALSE, startpos = 4)
pie.labels(3,3, 5.837518 ,radius=0.9,labels=label[1])
```

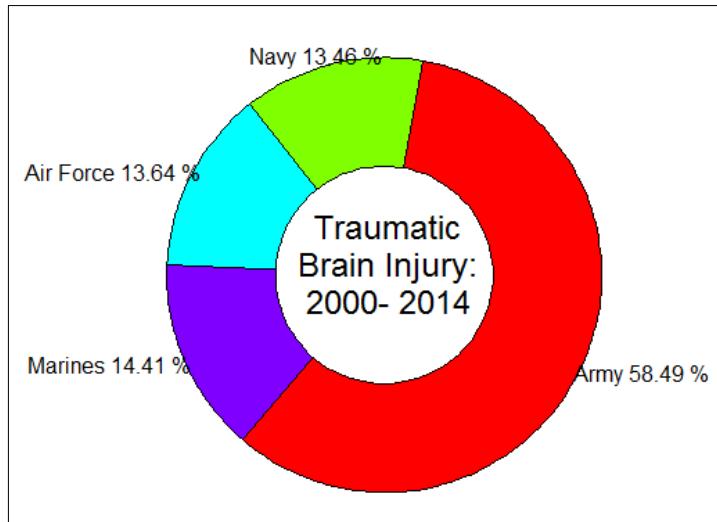
The preceding code is not complete; please refer to the code file to create the entire chart. The first section of the code in the code file is exactly the same as discussed in the previous two recipes. The `par()` function allows us to divide our plotting region into multiple sections. The `par()` function is discussed in detail in the recipe *Introducing a scatter plot*. in *Chapter, Basic and Interactive Plots*.

We use the `plot()` function to create a blank plot and further use the `floating.pie()` function to create a pie chart. Note that we will color the slice we are interested in with a different color and color all the other slices in the pie with the same color. Hence, we have repeated the hex values under the `col` argument for the Navy, Air Force, and Marines data.

As discussed previously, the `floating.pie()` function returns the angle to us and we use this angle as an input in the `pie.labels()` function to plot the label at the exact location where the slice occurs. The second argument in the `pie.labels()` function, `5.837518`, helps us achieve this. The question is, how would we know the angle? The answer is simple: just type `pie1` in the R console window and R will show you the angles returned from the `floating.pie()` function.

## Creating donut plots and interactive plots

Schwabish (2014) describes and outlines a basic limitation of donut charts as, *Donut charts—in which the centre of the pie is punched out—just exacerbate the problem: the empty centre makes the reader estimate the angle and arrive at other qualitative part-to-whole judgments without being able to see the center where the edges meet.* Surprisingly, donut plots have their own followers and we do observe them in business reports or in news media as well. One of the places where donut charts can be useful is where the number of variables to be displayed is small, such as data related to yes versus no or male versus female.



## Getting ready

To create a donut plot in R, we need to install the `plotrix` package in R.

## How to do it...

Donut plots are basically pie charts with a hole punched in the middle. Hence, the first few lines of the code are very similar to the previous recipes and we will not get into the details at this point. We will first install the `plotrix` package and load the library in R as well as create our data and labels vectors.

```
install.packages("plotrix")
library(plotrix)
data = c(179718,41370,41914,44280)
pct = (data/sum(data))*100
pct = round(pct,2)
labels = c("Army", "Navy", "Air Force", "Marines")
labels1 = paste(labels,pct, "%")
```

To create a donut plot, we will first generate a pie chart using the `floating.pie()` function and apply the labels using the `pie.labels()` function.

```
pie1<-floating.pie(3,3,pct,radius=1,col= rainbow(4),startpos = 4)
pie.labels(3,3, pie1,radius=1,labels= labels1)
```

Next, we use the `draw.circle()` function to create a circle in the middle of the pie chart. The circle overlaps the pie chart. We can now add the label for the entire plot using the `textbox()` function.

```
draw.circle(3,3,radius=0.5,col="white")
textbox(c(2.5,3.5),3.5,c("Traumatic Brain Injury: 2000- 2014"),
cex = 1.5, justify ="c",box = FALSE)
```

## How it works...

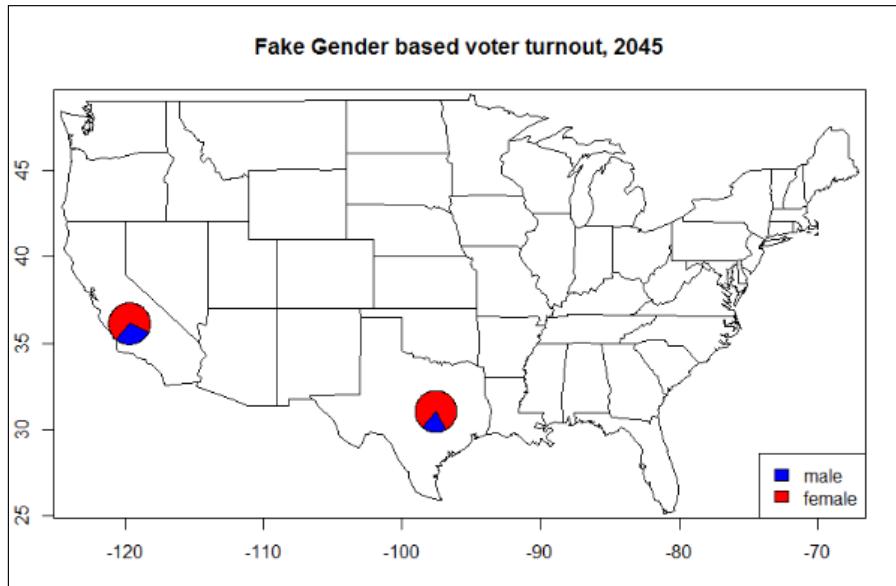
We have introduced two new functions in this recipe, namely: `draw.circle()` and `textbox()`. The first two arguments in the `draw.circle()` are the `x` and `y` coordinates. Note that we would like to draw a circle right in the centre of the pie chart; hence we will use the exact same `x` and `y` coordinates as the pie but the radius will be smaller. The `col` argument specifies the color used to fill the circle.

Now we apply the text to label the plot. The first two arguments in the `textbox()` function are the `x` and `y` coordinates. Readers should note that the `x` coordinate is a vector of two values `c(2.5, 3.5)`; the first value is the minimum and the second is the maximum value on the `x`-axis. The third argument in the function is the character vector of the actual label. The fourth argument, `cex`, controls the size of the fonts, and the `box = FALSE` argument instructs R to not to draw a box around the text.

The labels on the donut plot can be aligned easily by changing the `radius` argument in the `pie.labels()` function.

## There's more...

In order to create an interactive donut plot or a pie chart, readers can refer to the `googleVis` package manual. In the following diagram, we have integrated maps with pie charts. We can also plot a donut chart instead of a pie chart. Note that the data used for this chart is fake. These integrated plots are used to show distribution of voter data related to gender bias. You will observe that, as we plot pie on every state, it will overlap and the image might look too crowded. Also, it might get difficult to read the data when the slices in the pie increase. Plotting a pie over a map is not a new concept and readers interested in using them should keep these limitations in mind.



The following lines of code were used to generate an integrated map:

```
library(maps)
library(plotrix)
map("state")
title("Fake Gender based voter turnout, 2045")
map.axes()
data1 = c(20,80)
data2 = c(30,70)
floating.pie(-97.563461,31.054487,data1, radius=1.5,col=
 c("blue","red"),startpos = 4)
floating.pie(-119.681564,36.116203,data2, radius=1.5,col=
 c("blue","red"),startpos = 4)
legend("bottomright", fill = c("blue","red"), legend =
 c("male","female"))
```

The map of The United States of America, along with all the states, is plotted using the `map()` function available in the `map` library. The `title()` function is used to apply the title for the map. We apply the axes to our map using the `map.axes()` function. This gives us a good estimate as to the latitude and longitude of every state. The `floating.pie()` function is now used to overlay the pie on the map. Note that the x and y coordinates are the average longitude and latitude of Texas and California. We also add a legend to the plot using the `legend()` function. If readers are interested in plotting data over all the states, they can create a loop statement. If readers would like to know more about the `map()` function or `legend()` function, they can type `?map()` or `?legend()` respectively in the R console window.

## See also

- ▶ Jonathan Schwabish, *An Economist's Guide to Visualizing Data* (2014) at <http://pubs.aeaweb.org/doi/pdfplus/10.1257/jep.28.1.209>
- ▶ New York Times visualization, A Newly Contentious Top Court at <http://www.nytimes.com/interactive/2010/02/16/nyregion/0216-lippman.html?r=0>
- ▶ Interactive donut or pie charts can be generated using the `googleVis` package at <http://cran.r-project.org/web/packages/googleVis/googleVis.pdf>

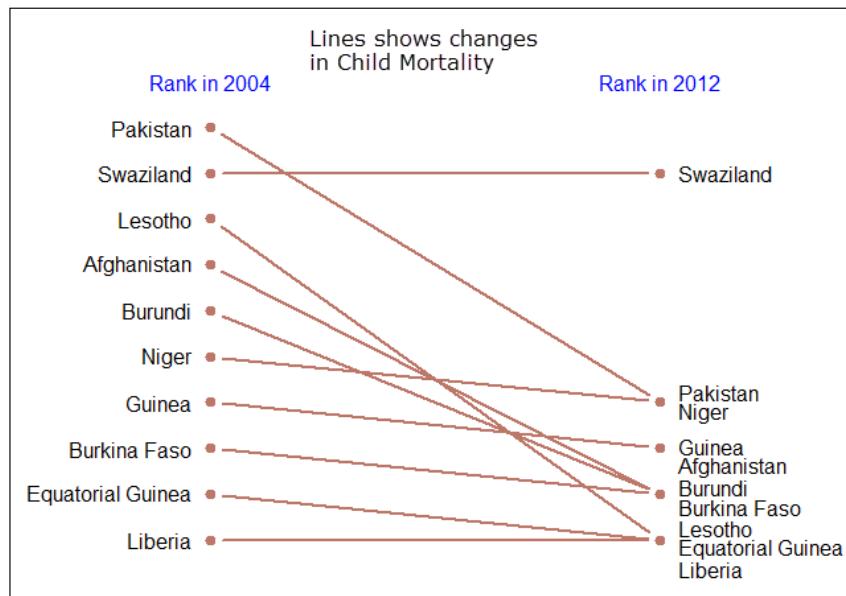
## Generating a slope chart

The idea behind visualizing data is to transfer the information in the right form and use the correct medium. We often observe the use of two different pie charts placed adjacent to each other. These plots are used to compare data between two different time periods. Schwabish (2014) provides slope charts and bar plots as alternatives to using two separate pie charts. The slope charts can be used to study the correlation between variables or to study the change in the same variable between two different time periods.

Some of the applications of slope charts are as follows:

- ▶ Changing obesity rates in the United States
- ▶ Changes in Child Mortality Rate
- ▶ Cancer Survival

The following slope chart implementation has been inspired by the New York Times child mortality infographic:



## Getting ready

We need to install and load the `plotrix` package in R.

## How to do it...

We start our recipe by installing and loading the `plotrix` library in R. The data contained in the `mortality1.csv` file consist of some randomly selected underdeveloped economies, their child mortality rates, and ranks in 2004 as well as 2012. We import the data using the `read.csv()` function. Note that R will look for the file in the current directory.

```
data = read.csv("mortality1.csv")
```

To create a slope chart, we require the use of the `rownames()` function. This instructs the `bumpcharts()` function to use the values listed in that column as the names for our slope chart.

```
rownames(data)=data[,1]
```

Once we assign the row names, we will have one extra column in our dataset. We will delete the country column by using the following manipulation technique:

```
data= data[,-1]
```

Now, we are ready to create our slope chart by using the `bumpchart()` function:

```
bumpchart(data[,3:4], lwd = 2, col= "#BD7769"), top.labels= NA,
rank = FALSE)
```

The labels and title can be added to the chart by using the `boxed.labels()` function:

```
boxed.labels(1,11,labels = c("Rank in 2004"), col = "blue",
border = FALSE)
boxed.labels(2,11,labels = c("Rank in 2012"), col = "blue",
border = FALSE)
```

## How it works...

The first argument in the `bumpchart()` function is the `data` argument. As our data consists of four columns and we would like to construct plots related to the ranks, we will only use the third and fourth columns. Note the use of `[, 3:4]` brackets in the `data` argument. All the other arguments are self-explanatory. The `top.labels` argument is suppressed as the argument plots the column headers as labels and the slope chart looks cluttered.

We have implemented all the labels in the slope chart using the `boxed.labels()` function. The first two arguments in the function relate to the `x` and `y` coordinates. The `labels` argument allows us to plot the labels.

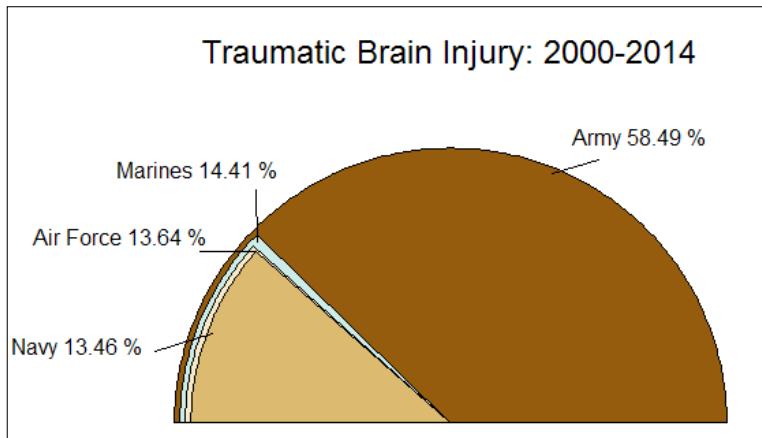
Readers should note that it is possible to construct a slope chart and plot the actual values on the plot. I would urge you to explore the `bumpchart()` function in detail by typing `?bumpcharts()` in the R console window.

## See also

- ▶ Jonathan Schwabish, *An Economist's Guide to Visualizing Data* (2014), at <http://pubs.aeaweb.org/doi/pdfplus/10.1257/jep.28.1.209>
- ▶ New York Times slope chart visualization at [http://www.nytimes.com/imagepages/2009/04/06/health/infant\\_stats.html](http://www.nytimes.com/imagepages/2009/04/06/health/infant_stats.html)
- ▶ Edward Tufte discusses the application of the slope graph at [http://www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg\\_id=0003nk](http://www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=0003nk)

# Constructing a fan plot

Fan plots are very similar to pie charts; the different sectors/slices in a fan plot overlap. The slice with the highest value is placed at the back and subsequent slices are plotted on top of it. In our example, Army has the highest percentage, followed by Marines, Air Force, and Navy. Note that Air Force and Navy overlap each other as the values are very close.



## Getting ready

To construct a fan plot, we use the following two packages:

- ▶ `plotrix`
- ▶ `RColorBrewer`

## How to do it...

The fan plot is implemented using the `plotrix` package; hence we need to install and load this in our R session. We have used the colors from `RColorBrewer` and hence we also need to load it in R.

The data and labels vectors, as well as the `paste()` function, are discussed in detail under the recipe *Generating a simple pie chart* in this chapter. We use the `fan.plot()` function to construct the fan plot in R.

```
data = c(179718, 41370, 41914, 44280)
pct = (data/sum(data))*100
pct = round(pct,2)
labels = c("Army", "Navy", "Air Force", "Marines")
labels = paste(labels,pct, "%")
```

We have used a softer color scheme from RColorBrewer and hence we will assign a set of colors to a variable and call it colors. We can do this by using the `brewer.pal( )` function of RColorBrewer.

```
colors = brewer.pal(6, "BrBG")
```

The `fan.plot()` argument is used to plot the fan plot in R.

```
fan.plot(pct, labels = labels, col = colors, max.span=pi,
align = "left", main ="Traumatic Brain Injury: 2000-2014")
```

## How it works...

The first argument in the `fan.plot()` function is the data. In our case, we have calculated the percentages for each service and stored them as the vector `pct`. The `max.span` argument allows us to create a semicircle. Readers should try to change the `max.span` value and observe how the `fan.plot` function changes. The `labels`, `col`, and `main` arguments are self-explanatory.

# 5

# Adding the Third Dimension

In this chapter, we will cover the following recipes:

- ▶ Constructing a 3D scatter plot
- ▶ Generating a 3D scatter plot with text
- ▶ A simple 3D pie chart
- ▶ A simple 3D histogram
- ▶ Generating a 3D contour plot
- ▶ Integrating a 3D contour and a surface plot
- ▶ Animating a 3D surface plot

## Introduction

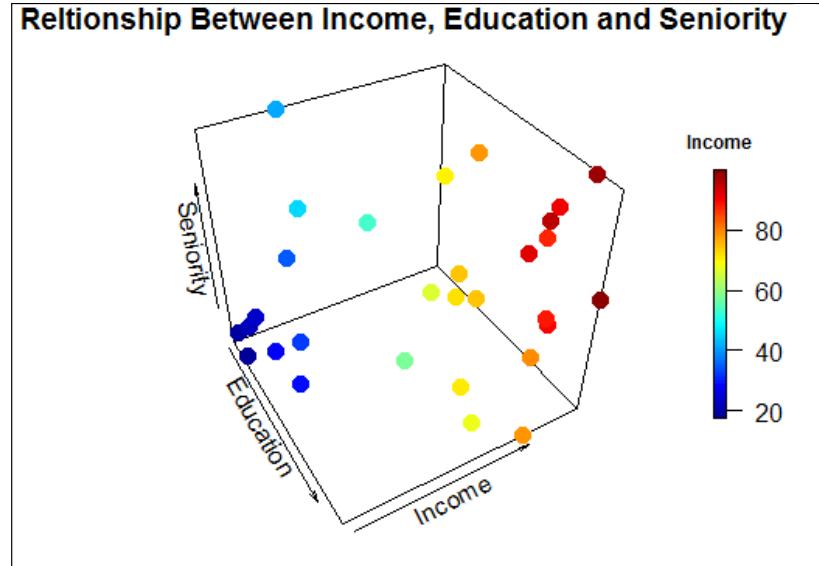
Three-dimensional images (pie charts or histograms), contours, surface plots, and scatter plots are great tools for analyzing multivariate data. Data that comprises more than two variables or densities, or even data with a high volume (medical or geosciences data), can be studied in detail when the third dimension is added. In this chapter, we will study different ways to analyze multivariate datasets, create interactive plots, and also learn to animate plots in R.

This chapter will introduce users to the `scatterplot3d`, `plot3D`, `rgl`, and `animation` packages. The readers should note that new R packages are developed on a regular basis and you might be able to find better packages. At present, these are some of the most useful packages available to plot 3D and interactive plots.

This chapter consists of four main sections. The first section is an introduction to plotting 3D scatter plots, the second section discusses ways to generate a 3D histogram and a pie chart, the third section pertains to 3D contour plots, and the last section describes surface plots and animation in R in detail.

## Constructing a 3D scatter plot

A 3D scatter plot is a great tool for visualizing multivariate data. Adding a third dimension to the existing plot helps in revealing information and portraying data from a newer angle. Even higher dimensions, such as the fourth and fifth, can be visualized by making use of color and shape attributes. We will implement this recipe using the `plot3D` package. The following screenshot shows a 3D scatter plot:



### Getting ready

To implement a scatter plot in 3D, we will need to install and load the `plot3D` package in R. The manual for the package is available on the CRAN website <http://cran.r-project.org/web/packages/plot3D/plot3D.pdf>.

## How to do it...

The package is installed and loaded in R using the `install.packages()` and `library()` functions respectively in R.

```
install.packages("plot3D")
library(plot3D)
```

The data used in generating the scatter plot is part of the book titled *An Introduction to Statistical Learning*, Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, Springer. For the purpose of this chapter, I have modified the data to include two additional columns: names and gender. We use the `read.csv()` function to load the data in our active R session and save it as a data frame called `inc`.

```
inc = read.csv("income2.csv")
```

Next, we generate a scatter plot in 3D using the `scatter3D()` function.

```
scatter3D(x = inc$Education, y = inc$Income, z = inc$Seniority,
colvar = inc$Income,
pch = 16, cex = 1.5, xlab = "Education", ylab = "Income",
zlab = "Seniority", theta = 60, d = 2, clab = c("Income"),
colkey = list(length = 0.5, width = 0.5, cex.clab = 0.75,
dist = -.08, side.clab = 3)
,main = "Relationship Between Income , Education and Seniority")
```

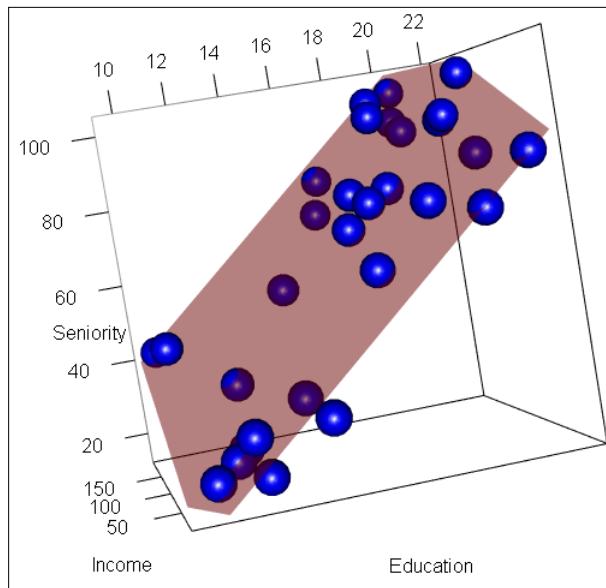
## How it works...

The `scatter3D()` function requires three main arguments, namely: `x`, `y`, and `z`. We supply these arguments from our data frame `inc` using the `$` notation. The `colorvar` argument allows us to color each symbol in the plot using a specific variable and hence, we use `Income` to signify the colors.

The `colkey` argument allows us to modify the color key that appears on the right side of the scatter plot. The manual describes all the attributes and options in detail. Most of the options defined under the `colkey` arguments are self-explanatory. The `cex.clab` argument controls the size of the font used to label the key, and the `dist` argument controls the distance of the key from the plot.

## There's more...

The `rgl` package in R is a very useful package for generating interactive plots. The package has various applications and we have discussed a few in this chapter. The advantage of using the `rgl` package is the interactivity. The plot window allows us to zoom in, zoom out, and rotate the plot.



When we run the following code, it will open up a new window and plot the scatter plot:

```
install.packages("rgl")
library(rgl)
inc = read.csv("income2.csv")
lmin = lm(inc$Income~inc$Education+inc$Seniority)
est = coef(lmin)
a = est["inc$Education"]
b = est["inc$Seniority"]
c=-1
d= est["(Intercept)"]
plot3d(inc$Education,inc$Seniority,inc$Income, type = "s",
 col = "blue", xlab = "Education",ylab = "Income",zlab =
 "Seniority",box = FALSE)
planes3d(a,b,c,d, alpha = 0.5, col = "red")
```

We can plot an interactive scatter plot using the `plot3d()` function. The attributes are very similar to the `scatter3D()` function discussed in the *How to do it...* section of this recipe.

In order to fit a plane, we run a regression using the `lm()` function. We extract the estimates and intercept information from the regression using the following code:

```
a = est["inc$Education"]
b = est["inc$Seniority"]
c=-1
d= est["(Intercept)"]
```

Finally, we will plot the plane in 3D using the following `plane3d()` function:

```
planes3d(a,b,c,d, alpha = 0.5, col = "red")
```

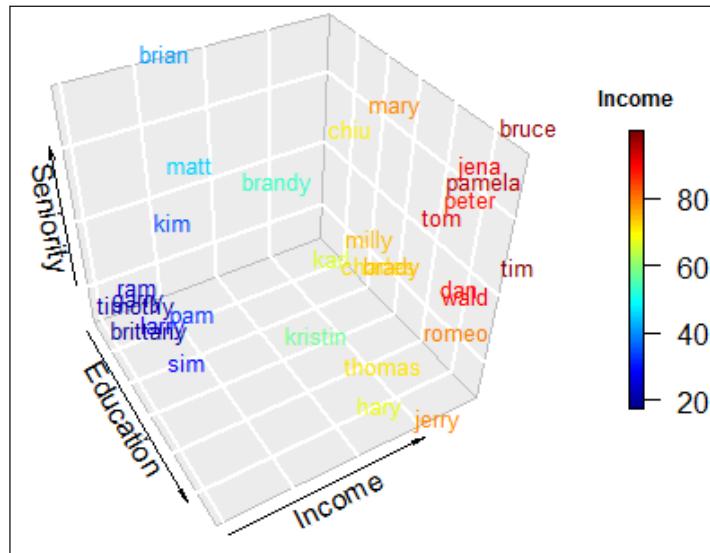
## See also

- ▶ The `plot3D` manual at <http://cran.r-project.org/web/packages/plot3D/plot3D.pdf>
- ▶ *An Introduction to Statistical Learning*, Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, Springer, which can be accessed at <http://www-bcf.usc.edu/~gareth/ISL/ISLR%20Fourth%20Printing.pdf>
- ▶ *Scatterplot3D - An R package for Visualizing Multivariate Data*, Uwe Ligges and Martin Machler, which can be accessed at <http://cran.r-project.org/web/packages/scatterplot3d/vignettes/s3d.pdf>

## Generating a 3D scatter plot with text

The main objective of writing this recipe is to introduce the reader to the additional functionality of the `plot3D` package. Applying text to a plot provides readers with additional information. If the dataset is large, we might encounter the issue of overlapping text, but it is a great tool to explore and present data.

The package under the code title *Two ways to make a scatter 3D of quakes dataset* provides an interesting implementation of the `scatter3D()` function, where the data points in 3D space are projected on a 2D space.



## Getting ready

For the current recipe, we would install and load the `plot3D` package in R.

## How to do it...

We will install and load the `plot3D` package in R using the `install.packages()` and `library()` functions. Next, we load the data using the `read.csv()` function. The dataset is the same data we used in the previous recipe.

```
install.packages("plot3D")
library(plot3D)
inc= read.csv("income2.csv")
```

We will define the column names as row names. This is needed as we would like to plot the actual names.

```
row.names(inc)= inc$names
```

We can now plot the actual text in 3D using the `text3D` function as follows:

```
text3D(x = inc$Education, y = inc$Income, z = inc$Seniority,
 colvar = inc$Income, labels= row.names(inc),
```

```

pch = 16, cex = 0.8, xlab = "Education", ylab = "Income",
zlab = "Seniority", theta = 60, d = 2, clab = c("Income"),
colkey = list(length = 0.5, width = 0.5, cex.clab = 0.75,
dist = -0.08, side.clab = 3)
,bty = "g")

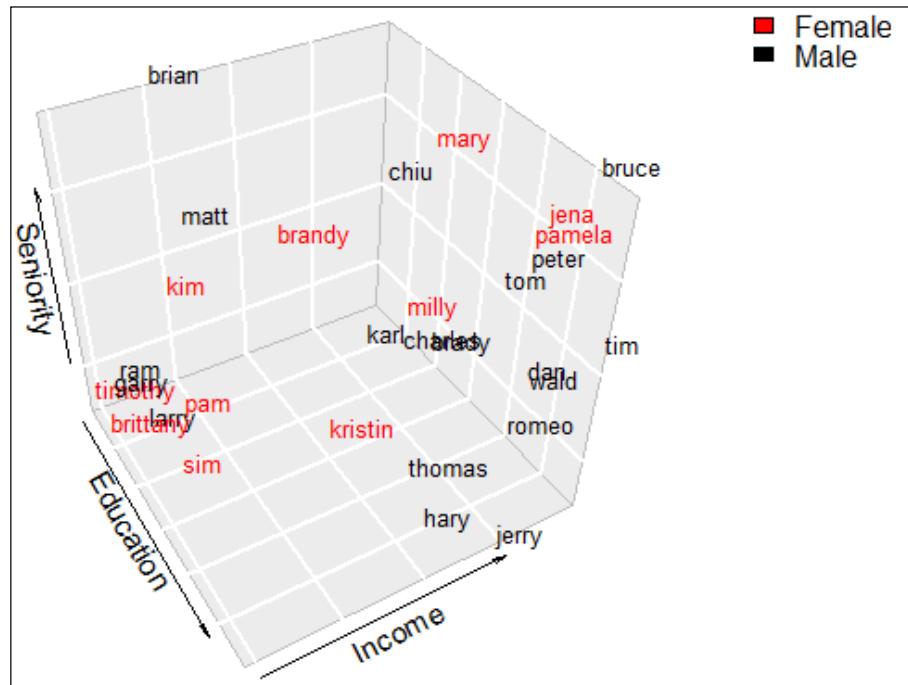
```

## How it works...

The first few arguments in the `text3D()` function are the `x`, `y`, and `z` variables. The `colvar`, `colkey`, `pch`, and `cex` arguments are all self-explanatory. As we would like to plot the actual names of individuals in place of points, the `text3D()` function needs to assign the `labels` argument. If we do not define column names as row names, it will automatically use the first column of the data as row names. The `theta` argument allows us to control the viewing angle of the plot and the `bty` argument controls the background. To learn more about all the available options under the `bty` argument, please refer to the perspective box section in the manual.

**There's more...**

We can tweak an argument in the `text3D()` function and generate a gender-based scatter plot.



We have updated the `colorvar` argument in the `text3D()` function to use the gender column in our database. Readers should keep in mind that the `colorvar` argument requires quantitative variables and hence our dataset includes 0 to indicate female and 1 to indicate male. As an additional step, we will also pass the `col` argument to assign colors to the labels.

```
inc= read.csv("income2.csv")
row.names(inc)= inc$names
text3D(x = inc$Education, y = inc$Income, z = inc$Seniority, colvar
= inc$gender,col = c("red","black"),labels= row.names(inc),
pch = 16, cex = 0.8,xlab = "Education", ylab = "Income",
zlab = "Seniority", theta = 60, d = 2,clab = c("Income"),
bty = "g", colkey = FALSE)
legend("topright", fill = c("red", "black"), legend=
c("Female","Male"), bty = "n")
```

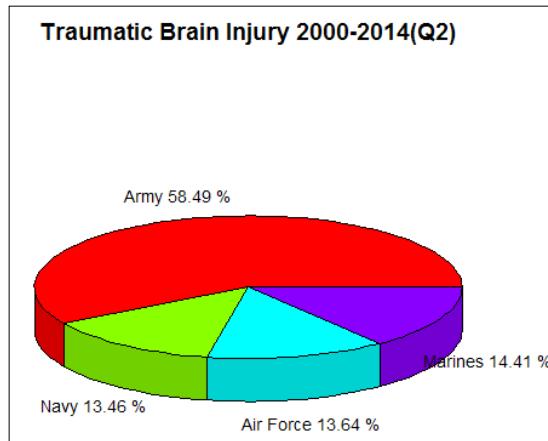
In the code file, you will also find code to generate a scatter plot using the `scatterplot3d` package.

## See also

- ▶ The `plot3D` manual at <http://cran.r-project.org/web/packages/plot3D/plot3D.pdf>
- ▶ *plot3D: Tools for plotting 2D and 3D Data*, Karline Soetaert, which can be accessed at <http://cran.r-project.org/web/packages/plot3D/vignettes/plot3D.pdf>

## A simple 3D pie chart

In *Chapter, The Pie Chart and its Alternatives*, we studied pie charts in great detail. In this recipe, we will focus on generating a 3D pie chart in R using the `plotrix` package.



## Getting ready

In order to generate a 3D pie chart, we will use the `plotrix` package in R.

## How to do it...

In order to plot a 3D pie chart in R, we will first install and load the `plotrix` package in R using the `install.packages()` and `library()` functions respectively:

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

We have generated the 3D pie chart using traumatic brain injury data, also used in *Chapter, The Pie Chart and its Alternatives*. Please refer to the recipe *Generating a simple pie chart* from that chapter to understand the data transformation and further use of `paste()`. The following lines of code are used to construct our data and the code is explained in detail in *Chapter, The Pie Chart and its Alternatives*:

```
data = c(179718, 41370, 41914, 44280)
pct = (data/sum(data))*100
pct = round(pct, 2)
labels = c("Army", "Navy", "Air Force", "Marines")
labels = paste(labels, pct, "%")
```

We have defined our colors by defining a character vector - `col`. We can now generate a 3D pie chart using the `pie3D()` function in R:

```
col = c("purple", "violetred1", "green3", "red", "cyan")
p = pie3D(pct, labels = NA, labelcex= 1, explode = 0,
main = "Traumatic Brain Injury 2000-2014 (Q2)")
```

In order to be certain that the labels are correctly applied to our pie chart, we have used the `pie3D.labels()` function:

```
pie3D.labels(p, labels=labels, labelcex = 0.9)
```

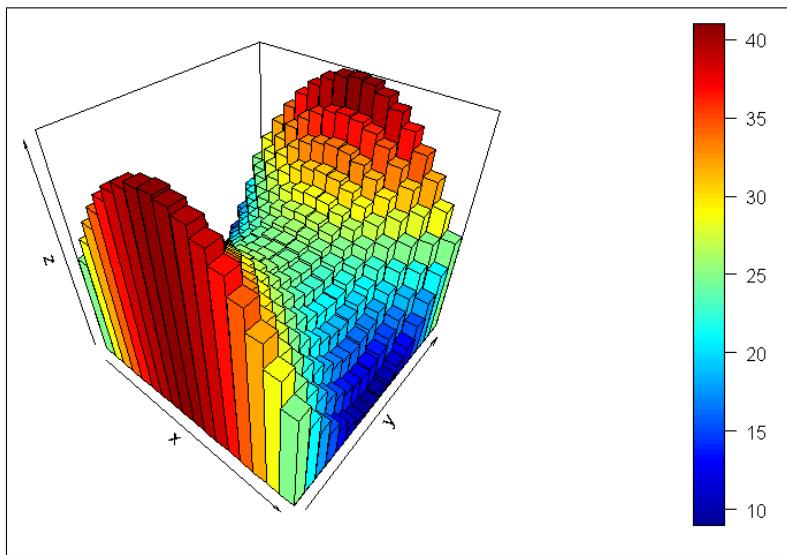
## How it works...

The first argument in the `pie3D` function is our data. We have converted our data into percentages and defined a vector `pct`. We have suppressed the labels using the `labels = NA` argument. We will add the labels to our pie chart using the `pie3D.labels()` functions. The `explode` argument allows us to control the amount by which we would like to explode each slice. Readers can change this value from 0 to 0.5 or 1 and observe the effect in their 3D pies.

The `pie3D.labels()` function is not necessary, but we would like to avoid any overlapping of labels and hence we use this function. The first argument in this function is the position of each label in the radians. The second argument is the `labels` argument, a character vector generated using the `paste()` function.

# A simple 3D histogram

We have studied histograms in *Chapter, A Simple Guide to R*. We will try to plot a 3D histogram in this recipe. The applications of 3D histograms are limited, but they are a great tool for displaying multiple variables in a plot. In order to construct a 3D histogram, as shown in the following screenshot, we will use the `plot3d` package available in R.



## Getting ready

To plot a histogram in 3D, we will use the `plot3D` package available in R.

## How to do it...

We will install as well as load the `plot3D` package in R using the `install.packages()` and `library()` functions, respectively.

```
install.packages("plot3D")
library(plot3D)
```

We will now generate data for the x and y values using the `seq()` function:

```
x = y = seq(-4, 4, by = 0.5)
```

The 3D histogram requires the z variable as well, which is generated using our function f defined using the following line of code:

```
f = function(x,y) {z = (25-(x^2-y^2))}
```

The z values are generated using the outer() function, which consists of the x and y values as well as our f function, defined previously.

```
z = outer(x,y,f)
```

We can now use the hist3D() function to generate the histogram.

```
hist3D(z = z, x = x, y = y, border = "black", image = TRUE)
```

## How it works...

To generate the z values, we need to write a function in R using the following format:

```
f = function() {
}
}
```

To learn more about functions, please refer to the recipe *Writing a function in R* in Chapter, *A Simple Guide to R*. We can define a function f as follows:

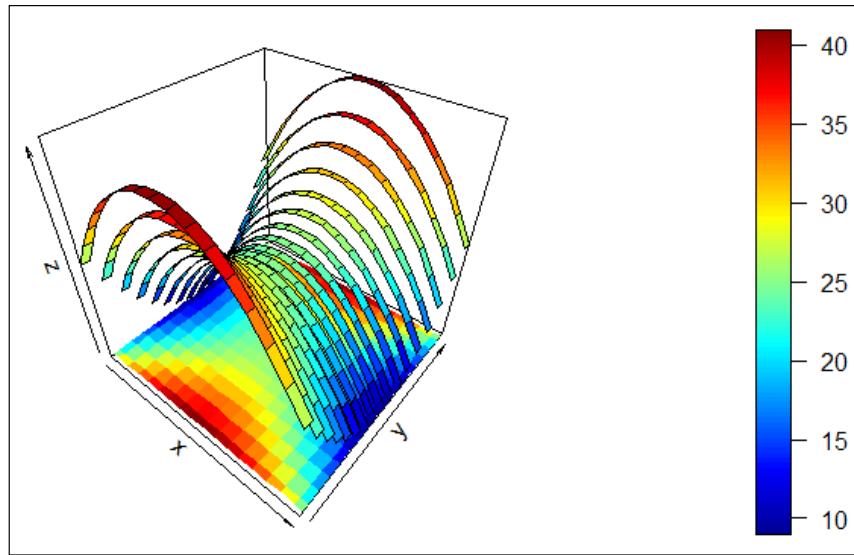
```
f = function(x,y) {
z = (25-(x^2-y^2))
}
```

Once we have established a function, we will generate the x and y values using the seq() function. The first argument in the seq() function is the first value, and the second argument in the seq() function is the last value. The by argument provides the increment from the first to the last value. The outer() function will use the x and y values, generated using the seq() function, as input to the function f defined by us. To learn more about the outer() function, type ?outer() in the R console window.

The first three arguments in the hist3D() function are the x, y, and z variables defined by us. The manual for the plot3D package describes many other options.

## There's more...

The ribbon plots are used to visualize and present data to indicate the direction of flow or twist in angles. The following recipe just skims over the topic of ribbon plots.



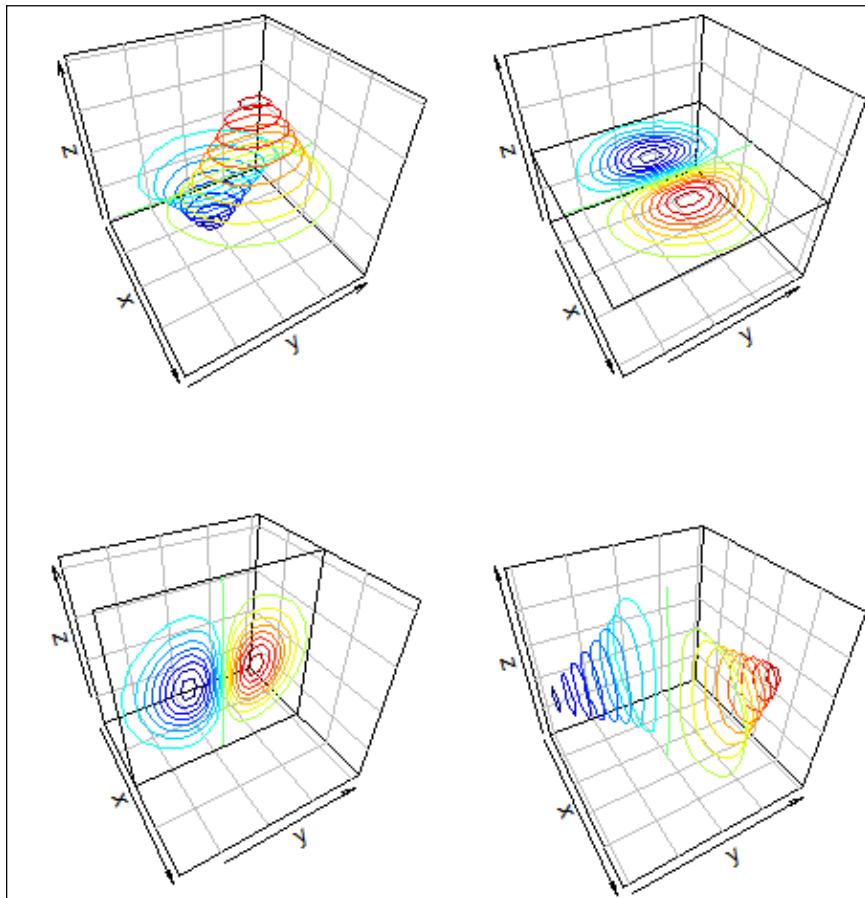
We can also plot a ribbon plot using the same function we defined under this recipe:

```
ribbon3D(z = z, x = x, y = y, border = "black", image = TRUE)
```

The `ribbon3D()` function plots the same data, but now we have a sliced image instead of a histogram.

## Generating a 3D contour plot

The `plot3D` package allows us to construct 3D contours and also slice them. In the current recipe, I have explained the implementation of 3D contours in R. The manual to the package discusses many different examples.



## Getting ready

We will install the `plot3D` package to generate 3D contours in R.

## How to do it...

The implementation of contour 3D is very similar to the histogram discussed in the previous recipe. We will first install and load the `plot3D` package in R using the following lines of code:

```
install.packages("plot3D")
library(plot3D)
```

We will now create a layout for our image. We will generate four different contour plots on the same window. The `layout()` function allows us to divide our plotting window into four sections:

```
par(mar=c(2,1,2,1))
l = layout(matrix(c(1,2,3,4), 2, 2, byrow = TRUE))
```

Next, we generate our dataset for the x and y values and write a function for the z value. The procedure is very similar to the recipe *A simple 3D histogram* discussed in this chapter.

```
x=y= seq(-2,2,by = .2)
fun = function(x,y){z=x*exp(-x^2-y^2)}
z = outer(x,y,fun)
r = 1:nrow(z)
p = 1:ncol(z)
```

We can now construct our four different contour plots using the `contour3D()` function. Each contour plot uses the same data, but we have changed the view points to introduce readers to various options that are available.

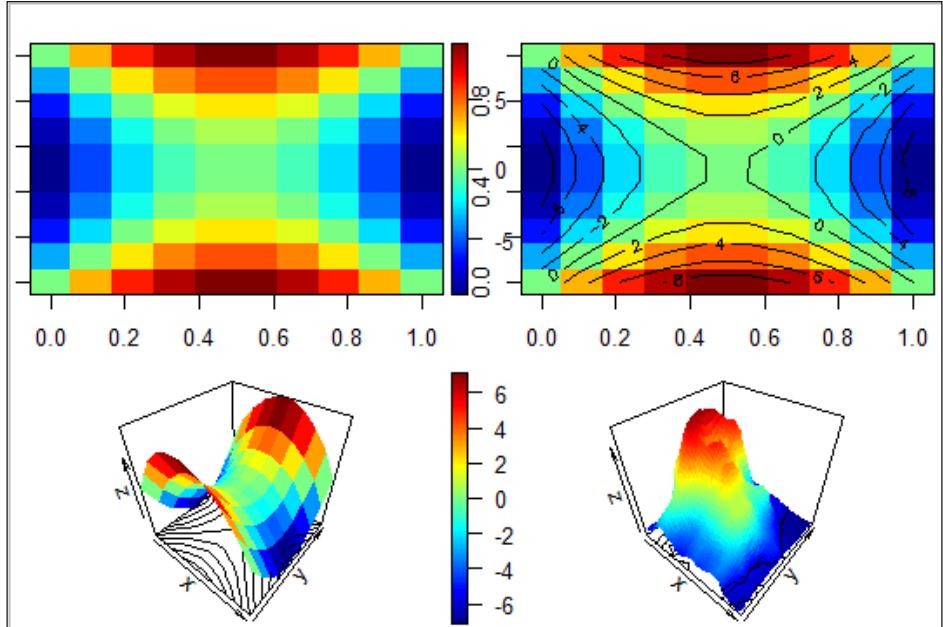
```
contour3D(x=x,y=y,z=z, colvar=z, bty="b2", dDepth =1,
theta=60, nlevels = 20, colkey = FALSE)
contour3D(x=r,y=p,z=5, colvar=z, bty="b2", dDepth =1,
theta=60, nlevels = 20, colkey = FALSE)
contour3D(x=5,y=p,z=r, colvar=z, bty="b2", dDepth =1,
theta=60, nlevels = 20, colkey = FALSE)
contour3D(y=z, colvar=z, bty="b2", dDepth =1, theta=60,
nlevels = 20, colkey = FALSE)
```

## How it works...

The first three arguments in the `contour3D()` functions are the x, y, and z values. The `colvar` argument defines the color of each contour. Note that we have removed the color key from our image by using `colkey= FALSE`. The `nlevels` argument increases the number of rings on the contour and `theta` controls the viewing angle. To learn more about the various options in the `contour3D()` function, readers should type `?contour3D` in the R console window.

## Integrating a 3D contour and a surface plot

We have studied contour plots in the recipe *A guide to contour maps* in *Chapter, Maps*. In this recipe, we will learn to plot a contour map in 3D using the `plot3D` package in R. Readers who are interested in studying contour plots should refer to the *See also* section of this recipe.



## Getting ready

For the purpose of the current recipe, we will install and load the `plot3D` package in R.

## How to do it...

We will install and load the `plot3D` package in R using the following lines of code:

```
install.packages("plot3D")
library(plot3D)
```

We now generate some data to construct our plots. This step is exactly the same step discussed under the *How to do it...* section in the recipe: *A simple 3D histogram*.

```
x = y = seq(-3,3, length.out = 10)
f = function(x,y){ z= (y^2-x^2) }
m = outer(x,y,f)
```

The following lines of code use the `image2D()` and `persp3D()` functions to generate four different plots:

```
image2D(m)
image2D(m, contour = TRUE)
persp3D(z = m, contour = TRUE)
persp3D(z = volcano, contour = TRUE)
```

## How it works...

The values for the plots can be generated using the `seq()` function. We will also define a function `f` to generate the values to be plotted on the z-axis.



Readers are encouraged to refer to the *A simple 3D histogram* recipe for more details on functions such as `seq()` and `outer()`.

The only argument we require to specify in the `image2D()` function is `m`, which is a series of z values generated using the function `f`.

The second attribute, `contour = TRUE`, will plot contour lines on the plot.

We can plot a three-dimensional plot in R using the `persp3D()` function. I would recommend readers to use the `plot3D` manual to better apply various options available with the `persp3D()` function. The `persp3D()` function uses two arguments, data and contour, to create a plot along with the contour image at the bottom of the plot.

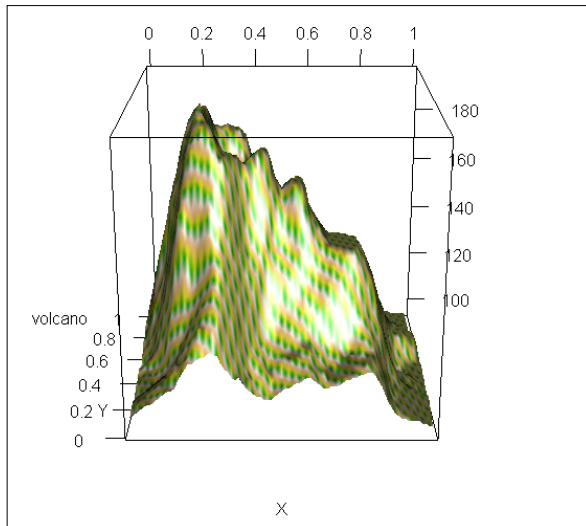
## There's more...

We can also plot an interactive contour map using the `rgl` packages. We will use the `rgl` package to plot the `volcano` dataset in R.

```
library(rgl)
c = terrain.colors(5)
persp3d(z = volcano, contour = TRUE, col = c)
```



Note that the function to plot a 3D interactive plot is identical to the function used to plot a 3D topographic map. We have added `terrain.colors(5)` to define a color range for the plot.

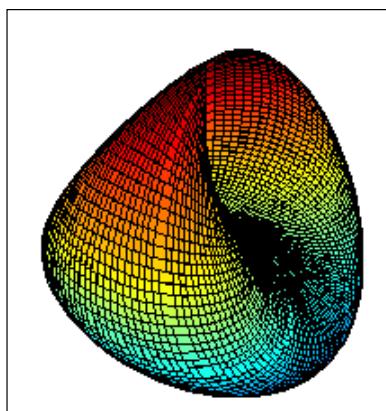


## See also

- ▶ *Fifty ways to draw a volcano using package plot3d*, Karline Soetaert (2013), which can be accessed at <http://cran.r-project.org/web/packages/plot3D/vignettes/volcano.pdf>

## Animating a 3D surface plot

Surface plots are used to generate some great geometrical shapes in R. Readers who have studied calculus will recognize the image generated using the surface plot. In the current recipe, we will introduce readers to surface plots and animation in R. The surface plot is generated using the parametric equations found in any calculus textbook.



## Getting ready

In order to plot a surface plot and animate the same, we need to install and load the following two packages in R:

- ▶ `plotrix3D()`
- ▶ `animation`

## How to do it...

For the purpose of this recipe, we will first generate a simple surface plot. We will then use an animation package to animate the plot. We have discussed the use of the `seq()` function to generate data under a 3D histogram.

```
x = y = seq(0,2*pi, length.out = 100)
z = mesh(x,y)
u = z$x
v = z$y
```

We can now define the `m`, `n`, and `o` variables using the parametric equation:

```
m= (sin(u)*sin(2*v)/2)
n= (sin(2*u)*cos(v)*cos(v))
o= (cos(2*u)*cos(v)*cos(v))
```

We use the `surf3D()` function to construct our surface plot:

```
surf3D(m, n,o, colvar = o, border = "black",colkey = FALSE,
 box = TRUE)
surf3D(m, n,o, colvar = o, border = "black",colkey = TRUE,
 box = TRUE,theta = 60)
surf3D(m, n,o, colvar = o, border = "black",colkey = TRUE,
 box = TRUE,theta = 100)
```

To animate the plot, we have made use of looping in the `saveHTML()` function discussed under the *How it works...* section of this recipe.

## How it works...

We use the `mesh()` function to generate a 2D grid. We assign the variables `u` and `v` the values generated using the `seq()` function.

The `surf3D()` function would use the `m`, `n`, and `o` values as its first three attributes. The `colvar = o` attribute is used to define the range of colors. Note that, as you change the `theta =` values from 10, to 60, to 100, the direction of the image will change. We will take the advantage of this argument to construct a loop to animate the plot.

To animate the image, copy and paste the following lines of code in the R console window. R might be slow to construct the animated image in your browser. The same code can be changed slightly to plot an animated GIF image:

```
library("animation")
saveHTML({
 for (i in 1:100){
 x = y = seq(0,2*pi, length.out = 100)
 z = mesh(x,y)
 u = z$x
 v = z$y
 m= (sin(u)*sin(2*v)/2)
 n = (sin(2*u)*cos(v)*cos(v))
 o = (cos(2*u)*cos(v)*cos(v))
 surf3D(m, n,o, colvar = o, border = "black",colkey = FALSE,
 theta = i, box = TRUE)
 }
},interval = 0.1, ani.width = 500, ani.height = 1000)
```

We observe that most of the code is exactly the same as discussed in the *How to do it...* section of this recipe. To animate the image, we will first load the `animation` library. We will then use the `saveHTML()` function.

Readers should note two very important elements in this code. Firstly, we have the loop statement defined in the `saveHTML()` function that goes from 1 to 100. Secondly, our `surf3D()` function has one additional argument, `theta = i`. The loop function will simply rotate the image in a browser. All the other animation arguments, such as `interval`, `ani.width`, and `ani.height`, are used to define the elements of a browser.

The same code can be used to output a GIF image by simply substituting `saveHTML()` for `saveGIF()`. The `animation` package can be used to plot animated maps, lines, scatter plots, and so on. The readers simply need to make use of the loops in R inside the animation functions.

Depending on your browser setting, you might not see the animation right away. I would suggest you to go to your current R directory. You will observe some extra files generated by the `animation` package; one of them will be `index.html`. If you right-click and open it with Internet Explorer or Google Chrome, you will observe the animation. The files generated by the `animation` package can be used to further import the animation to your website or blog. We can also use the `saveGIF()` function to generate a GIF file. Please refer to the `animation` manual for instructions on using this functionality.

## There's more...

The animation package is a very handy for generating animations quickly. I found the following functions to be great tools to teach and understand some important statistical concepts:

```
brownian.motion()
kmeans.ani()
knn.ani()
least.squares()
lnn.ani()
mar.ani()
newton.mthod()
price.ani()
```

## See also

- ▶ *animation: An R package for Creating Animations and Demonstrating Statistical Methods*, Yihui Xie (2013), which can be accessed at <http://www.jstatsoft.org/v53/i01/paper>
- ▶ *3D-Harmonographs In Motion*, which can be accessed at <https://aschinchon.wordpress.com/2014/11/11/3d-harmonographs-in-motion/>

# 6

# Data in Higher Dimensions

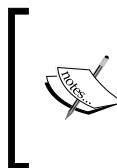
In this chapter, we will cover the following recipes:

- ▶ Constructing a sunflower plot
- ▶ Creating a hexbin plot
- ▶ Generating interactive calendar maps
- ▶ Creating Chernoff faces in R
- ▶ Constructing a coxcomb plot in R
- ▶ Constructing network plots
- ▶ Constructing a radial plot
- ▶ Generating a very basic pyramid plot

## Introduction

Most of the visualizations studied so far have been widely observed in media, magazines, websites, or academic journals. Many of the recipes discussed in this chapter relate to visualizing data in higher dimensions or multivariate data. We might not have encountered some of the visualizations discussed in this chapter due to their limitations, but this does not imply that we cannot utilize them to convey the right information.

In this chapter, we will introduce plots such as sunflower plots, hexbins, calendar maps, coxcombs, and Chernoff faces, which are rarely used but are great tools to explore and present data. We will also explore network plots, pyramid plots, and radial plots, which have been utilized to convey information in a meaningful way.

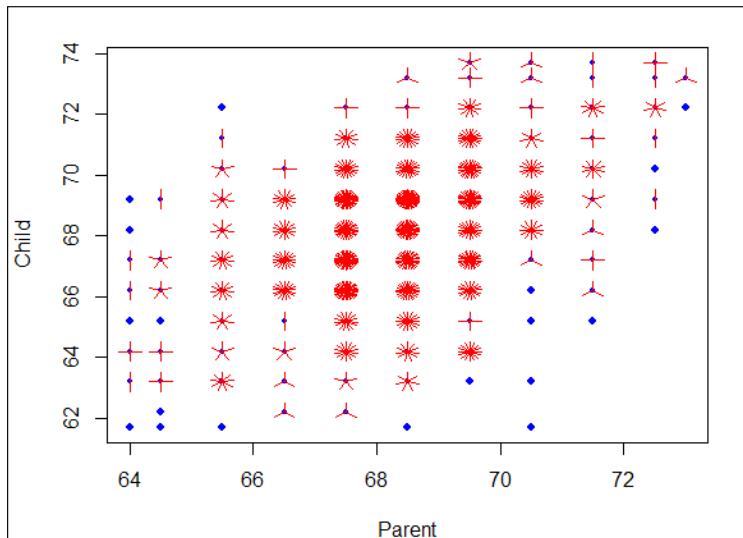


Note that, while running the code, you might encounter issues where the plots do not look right or they are generated over an existing plot. The best way to avoid this is by clearing the plot window using the **clear all** tab and typing `plot.new()` in the R console window before running any code.

## Constructing a sunflower plot

Sunflower plot, as the name suggests, looks like a sunflower drawn in a 2D space. The sunflower plots are used as variants of scatter plots to display bivariate distribution. When the density of data increases in a particular region of a plot, it becomes hard to read. Each petal in a sunflower plot represents an observation; hence, sunflower plots can deal with high-density data. Hexbin plots, discussed later in the chapter, are also an alternative to resolving the issue of overlapping observations in a scatter plot. Dupont and Plummer Jr. (2003) provide an insightful discussion on the advantages of using a sunflower plot over scatter plots in case of high-density datasets.

Sunflower plots are available with the basic R plotting package, and we can learn more about their various arguments by typing `?sunflowerplot` in the R Console window.



## Getting ready

The plot is constructed using the Galton data available with the `HistData` package in R. Galton data comprises two variables: average height of the father and mother, and the height of the child.

## How to do it...

We load the Galton data by installing the package and load it in R by using the `install.packages()` and `library()` functions in R, respectively:

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

To examine the headers and first six observations, we use the `head()` function or `View(head())` function. It is a good practice to partially view the data before starting to plot it; the `head()` function allows us to accomplish this in R:

```
head(Galton)
View(head(Galton, 8))
```

To construct a sunflower plot in R, we utilize the `sunflowerplot()` function:

```
sunflowerplot(Galton$parent, Galton$child, col = "blue",
seg.col = "red", xlab = "Parent", ylab = "Child")
```

## How it works...

The `library()` and `head()` functions are self-explanatory. The number 8 in the `View()` function is the number of data lines to display. The data to be displayed on the x-axis is passed as the first argument in the `sunflowerplot()` function. The second argument consists of data to be displayed on the y-axis. Note the use of the `$` sign; as discussed previously, the `$` sign is placed after the referencing dataset, which is followed by the `$` sign (`data$column name`).

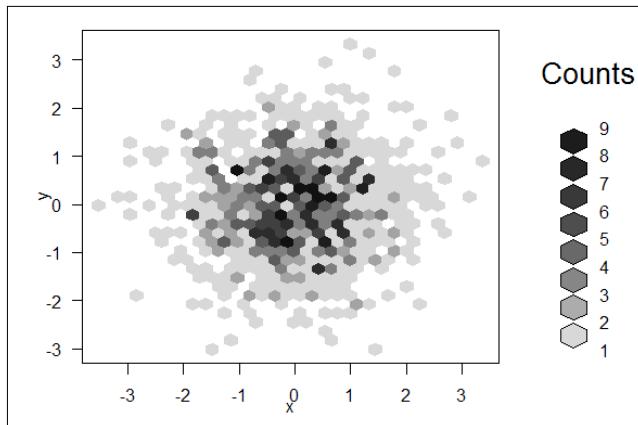
The `col` argument is used to specify the color of the plot and the `seg.col` argument applies color to the leaves. If the data is a single dot, it is just one observation; if the data has two lines, it represents two points; data with three lines represents three points, and so on. The `sunflowerplot()` function comes with additional arguments that can be explored by typing `?sunflowerplot` in the R console window.

## See also

- ▶ *Density Distribution Sunflower Plots, Dupont and Plummer Jr (2013)*, which can be accessed at <http://www.jstatsoft.org/v08/i03/paper>
- ▶ *The Many Faces of a Scatterplot, Cleveland and McGill (1984)*, which can be accessed at [http://moderngraphics11.pbworks.com/w/file/fetch/31401342/cleveland%2526mcgill\\_1984b.pdf](http://moderngraphics11.pbworks.com/w/file/fetch/31401342/cleveland%2526mcgill_1984b.pdf)

## Creating a hexbin plot

Hexbin plots can be viewed as an alternative to scatter plots. The hexagon-shaped bins were introduced to plot densely packed sunflower plots. They can be used to plot scatter plots with high-density data. We will use the `hexbin` package available in R to plot hexagon-shaped bins on a plot instead of a sunflower.



## Getting ready

To generate a hexbin plot, we will use the `hexbin` package in R along with the generic `rnorm()` function.

## How to do it...

The `install.packages()` and `library()` functions assist in installing the package as well as loading it in R:

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

For the purpose of this recipe, we will generate a fake dataset. The `set.seed(356)` code sets the state of random number generation in R:

```
set.seed(356)
```

We will implement the `rnorm()` function to generate 1000 normally distributed random numbers:

```
x = rnorm(1000)
y = rnorm(1000)
```

In order to generate a hexbin plot, we will have to define a hexbin object using the `hexbin()` function:

```
bins = hexbin(x,y)
```

Now, we can simply use the generic R `plot()` function to create a hexbin plot:

```
plot(bins)
plot(bins , border = TRUE)
plot(bins, border = "red")
smb = smooth.hexbin(bins)
plot(smb)
```

## How it works...

The `rnorm()` function consists mainly of three arguments. The first argument is `n`, which represents the number of data points to be generated; the second and third arguments are the mean and standard deviation respectively. By default, R will assume a mean of 0 and a standard deviation of 1 if these arguments are missing. To learn more about all the different distributions available in R, type `?distributions` in the R console window.

One important ingredient required to generate a hexbin plot is generating the bins. We use the `hexbins()` function, available in the `hexbin` package, to accomplish our task. The first and second arguments of the `hexbin()` function are the values generated for variables `x` and `y`. R will calculate the hexbins and store them as objects under the name `bins`. Please refer to the `hexbin` manual available on the CRAN website at <http://cran.r-project.org/web/packages/hexbin/hexbin.pdf> to understand the extra arguments related to the `hexbin()` function.

Finally, we can generate a very simple hexbin plot by passing the `bins` object in the generic `plot()` function in R. We can also apply bin smoothing to our data by creating an object using the `smooth.hexbin(bins)` function and subsequently passing it in the `plot(smb)` function. Smoothing our data assists in better interpretation of data, especially in cases where the data is too large.

## See also

- ▶ It is possible to implement the hexbin plot using the `ggplot2` package. Please refer to the documentation at [http://docs.ggplot2.org/current/stat\\_binhex.html](http://docs.ggplot2.org/current/stat_binhex.html) to understand its implementation.
- ▶ The page at <http://indiemaps.com/blog/2011/10/hexbins/> provides a good explanation regarding hexbin and sunflower plots.
- ▶ New York Times visualization uses hexbin to plot the shooting patterns of two basketball teams. It can be accessed at <http://www.nytimes.com/interactive/2012/06/11/sports/basketball/nba-shot-analysis.html>.

## Generating interactive calendar maps

Calendar maps are used to display continuous data over a period of time. Calendar plots have been used to display data on a daily or monthly basis, where each square represents a data point.

In the past, Nathan Yau, in his fatal car crashes, has used calendar heat maps to study the safest time of the year to travel. Using stock prices as input in our calendar map assists us in conducting event study or holiday and weekend effects. We can also apply calendar maps to study airline delays during the days of the month or crime rates during specific times of the year.

In this recipe, we will use the calendar plot to study the daily stock price movement of Facebook over 2013. In order to extract the stock prices from Yahoo Finance, we will use the `quantmod` package and construct a calendar plot using the `googlevis` package.



## Getting ready

To generate an interactive calendar map, we will install the following packages:

- ▶ The `quantmod` package to download the stock prices
- ▶ The `googleVis` package to generate an interactive calendar plot

## How to do it...

To generate an interactive calendar plot in R, we will install as well as load the `quantmod` and `googleVis` packages in R by typing the following lines in the R console window:

```
install.packages(c("quantmod", "googleVis"))
library(quantmod)
library(googleVis)
```

To extract prices, we will first create a character vector in R using the `c()` notation and use the `getSymbols()` function to extract the prices of Facebook stock using its ticker `FB`:

```
prices = c("FB")
getSymbols(prices, src = "yahoo", from = as.Date ("2013-01-01"),
 to = as.Date("2013-12-31"))
```

To implement a calendar plot using the `googleVis` package, we need the data in a data frame format. Hence, we use the `data.frame()` function as shown:

```
data =data.frame(date = as.Date(index(FB)) ,
 open = as.numeric(FB$FB.Open))
```

Finally, we plot the calendar plot in R by creating a calendar object using the `gvisCalendar()` function and display the plot using the `plot()` function:

```
fb <- gvisCalendar(data, datevar="date",
 numvar="open", options=list(title="Daily Prices of
Facebook stock",height=320,width = 1000, calendar="{yearLabel:
{color: '#FF0000'},cellSize: 14,cellColor :{strokeWidth: 0.5},
monthOutlineColor:{stroke
:'white',strokeWidth: 5}}"))
plot(fb)
```

## How it works...

The first argument in the `getSymbols()` function is the list of tickers. In our recipe, we are only using the Facebook prices, but we can pass multiple tickers as well using the `prices= c("FB", "AMZN", "MSFT")` code. The `src` argument is used to define the source to pull the data. In order to learn more about various other acceptable values, please type `?getSymbols` in the R console window. The `from` and `to` arguments are used to define a date range. The `as.Date()` function is a basic R function used to define dates in R.

By default, the data downloaded using the `getSymbols()` function is in the XTS format. We could also download the data as "zoo", "ts", `data.frame`, or in a time series format. All we need to do is include one more additional argument `return.class = ts` (for time series) under the `getSymbols()` function.

The downloaded data has columns: close, volume, high, low, and so on. We will utilize opening prices and date columns. Note that, in order to plot a calendar plot, we require data in the `data.frame` format. We can extract the dates and opening prices columns from the XTS data file using the `data.frame()` function.

We would like to extract two columns from the FB dataset: date and opening prices. Hence, we pass date as the first argument in the `data.frame()` function and the second argument is `FB.open`, which is just the opening price column from the FB data. Note the use of `as.Date(index(FB))` and `as.numeric(FB$FB.Open)`.

|    | row.names  | FB.Open | FB.High | FB.Low | FB.Close | FB.Volume | FB.Adjusted |
|----|------------|---------|---------|--------|----------|-----------|-------------|
| 1  | 2013-01-02 | 27.44   | 28.18   | 27.42  | 28.00    | 69846400  | 26.00       |
| 2  | 2013-01-03 | 27.88   | 28.47   | 27.59  | 27.77    | 63140600  | 27.77       |
| 3  | 2013-01-04 | 28.01   | 28.93   | 27.83  | 28.76    | 72715400  | 28.76       |
| 4  | 2013-01-07 | 28.69   | 29.79   | 28.65  | 29.42    | 83781800  | 29.42       |
| 5  | 2013-01-08 | 29.51   | 29.60   | 28.86  | 29.06    | 45871500  | 29.06       |
| 6  | 2013-01-09 | 29.67   | 30.60   | 29.49  | 30.59    | 104787700 | 30.55       |
| 7  | 2013-01-10 | 30.60   | 31.45   | 30.28  | 31.30    | 95316400  | 31.30       |
| 8  | 2013-01-11 | 31.28   | 31.96   | 31.10  | 31.72    | 89598000  | 31.72       |
| 9  | 2013-01-14 | 32.08   | 32.21   | 30.62  | 30.95    | 98892800  | 30.95       |
| 10 | 2013-01-15 | 30.64   | 31.71   | 29.88  | 30.10    | 173242600 | 30.10       |
| 11 | 2013-01-16 | 30.21   | 30.35   | 29.53  | 29.85    | 75332700  | 29.85       |
| 12 | 2013-01-17 | 30.08   | 30.42   | 30.03  | 30.14    | 40256700  | 30.14       |
| 13 | 2013-01-18 | 30.31   | 30.44   | 29.27  | 29.66    | 49631500  | 29.66       |
| 14 | 2013-01-22 | 29.75   | 30.89   | 29.74  | 30.73    | 55243300  | 30.73       |
| 15 | 2013-01-23 | 31.10   | 31.50   | 30.80  | 30.82    | 48899800  | 30.82       |

The `gvisCalendar()` function allows us to construct the plot. The first argument in the function is our dataset generated using the `data.frame()` function. The `datevar` as well as `numvar` arguments are used to define the columns to be used to construct the plot. The plot can be customized further by specifying various options such as, width, height, labels, and so on, under the `options = list()` argument. A list of all the available options is specified at <http://cran.r-project.org/web/packages/googleVis/googleVis.pdf> and <https://developers.google.com/chart/interactive/docs/gallery/calendar>.

The `plot()` function can be called to generate the plot in a new browser window. Note that, at the time of writing this chapter, `googleVis` provided limited flexibility to change the color of the cells in a calendar plot. This might change in future releases.

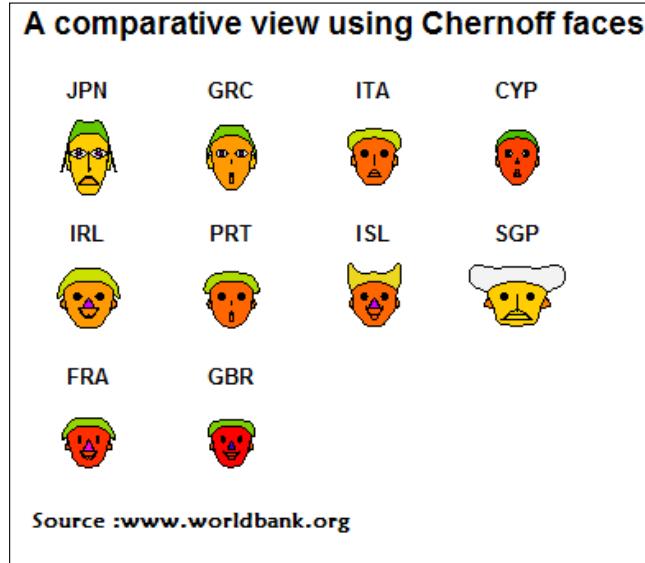
## See also

- ▶ Vehicles involved in Fatal Crashes at <http://flowingdata.com/2012/01/11/vehicles-involved-in-fatal-crashes/>
- ▶ Five Years of Traffic Fatalities at <http://uxblog.idvsolutions.com/2012/12/five-years-of-traffic-fatalities.html>
- ▶ Financial Time series at <http://blog.revolutionanalytics.com/2009/11/charting-time-series-as-calendar-heat-maps-in-r.html>

## Creating Chernoff faces in R

One of the alternative methods to visualize multivariate data is using Chernoff faces. Each variable in the dataset is used to represent a feature of the face. Chernoff used 18 variables to represent different facial features such as head, nose, eyes, eyebrows, mouth, and ears. Kosara (<http://eagereyes.org/criticism/chernoff-faces>) discusses the limitation of using Chernoff faces at length.

In order to construct Chernoff faces, I have downloaded some specific macroeconomic variables for a set of countries from the World Bank website.



## Getting ready

We will implement Chernoff faces in R by using the `faces()` function available under the `aplypack` package in R.

## How to do it...

To create Chernoff faces, we would require to install the `aplypack` package in R and load it in our active R session using the `install.packages()` and `library()` functions:

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

We can load the data in R using the `read.csv()` function. Note that R will search for the datafile in your current directory. Hence, if the data is not present in your current directory, you would have to change your current directory in R using the `setwd()` function:

```
data1 = read.csv("wolddecon.csv", header = TRUE, sep = ",")
```

It is a good practice to view the data once it is imported in R in order to examine the format and column headers. We can explore the data using the `head()` function:

```
head(data1)
```

Finally, we generate Chernoff faces using the `faces()` function available with the `aplypack` package:

```
faces(data1[1:10,3:9], labels = data1$Code, main = "A comparative
view using Chernoff faces")
```

## How it works...

The first argument in the `faces()` function is the dataset that is imported using the `read.csv()` function. The `labels` argument instructs R to use the specified column as labels. The documentation available on the CRAN website lists many other options that we can implement.

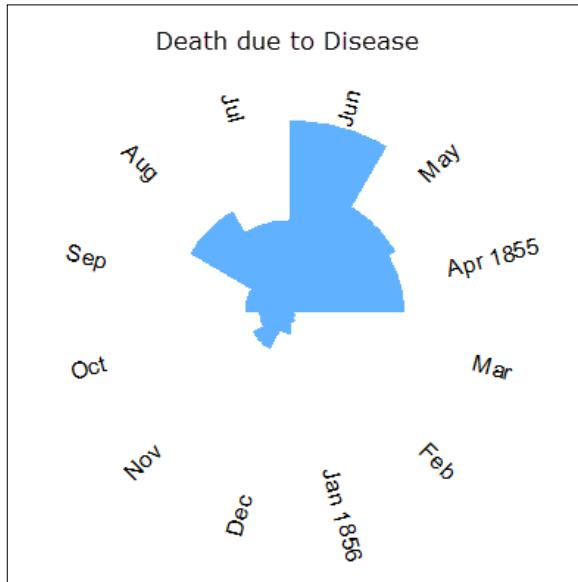
Please note that I have used the first 10 rows to plot the faces. If you use the entire dataset, you might get an error in R. This error can be avoided by simply expanding the R Studio's plot window. When R plots the faces, it will also plot the column headings corresponding to the facial expression. This information can be used as a legend to add more information about our visualization.

One of the criticisms regarding the use of Chernoff faces is that it is very hard to read facial expressions, especially when they are not very extreme. With regard to our data, we observe that Singapore (SGP) has bigger hair, implying high exports compared to other countries. Also, when considering debt, we observe that Great Britain (GBR) has lower debt compared to Japan (JPN), which is evident from the length of the face.

# Constructing a coxcomb plot in R

Coxcomb plots or Polar diagrams were developed by Florence Nightingale to show that most of the deaths of British soldiers were due to sickness rather than actual wounds during the Crimea War. Coxcomb plots are usually viewed as variants of pie charts.

According to Wikipedia, if the count of deaths in each month for a year is to be plotted, then there will be 12 sectors (one per month), all with the same angle of 30 degrees each. The radius of each sector would be proportional to the square root of the death count for the month, so the area of a sector represents the number of deaths in a month. To construct the coxcomb plot, we will use the same dataset that was used by Florence Nightingale.



## Getting ready

In order to construct a coxcomb plot, we will utilize the `HistData` and `plotrix` packages.

## How to do it...

We will install and load the packages in R using the `install.packages()` and `library()` functions:

```
install.packages(c("HistData", "plotrix"))
library(HistData)
library(plotrix)
```

The data used in this recipe is available with the `HistData` package. We can load the Nightingale data in R by typing the following code in R:

```
data = Nightingale[13:24,]
```

To generate the coxcomb plot in R, we will use the `radial.pie()` function available in the `plotrix` package. The labels for the plot are generated by constructing a character vector, `month`:

```
month = c("Apr 1855", "May", "Jun", "Jul", "Aug", "Sep", "Oct",
 "Nov", "Dec", "Jan 1856", "Feb", "Mar")
radial.pie(data$Disease, labels=month, boxed.radial =
 FALSE, show.grid = TRUE, sector.colors =c(rep("#60B1FF",12)),
 grid.col= "white", mar=c(2,10,2,10), show.grid.labels = 0,
 axis = FALSE, label.prop= .9)
```

## How it works...

The `HistData` library consists of many historical datasets; we will use this package to load the Nightingale dataset in R. We are employing two steps in one line of code. As we only require the data for 1855, we will use `[ ]` to extract the data from Nightingale as `Nightingale[13:24, ]` and store it as `data`.

If you go through the `code.txt` file for this chapter, you will observe two additional ways to generate the radial pie. We can manipulate the arguments within the `radial.pie()` function to customize our plot as per our requirements.

The first argument in the `radial.pie()` function is the data that needs to be visualized, and the second argument corresponds to the labels. The `boxed.radial` argument is used to create a box around the radial values. We have displayed the grid around the plot by using the `show.grid = TRUE` argument but have colored it white using the `grid.col` argument.

The `sector.colors` argument allows us to color each sector of the plot differently. However, we would like to apply the same color to all the sectors and hence we use the `rep()` function within the `sector.color` argument.

The `show.grid.labels` argument allows us to display labels in the plot. The `label.prop` argument allows us to position the label; we can change the value and observe the difference. Readers should refer to the `plotrix` documentation on the CRAN website at <http://cran.r-project.org/web/packages/plotrix/plotrix.pdf> for all the extra arguments that can be used along with the `radial.pie()` function.

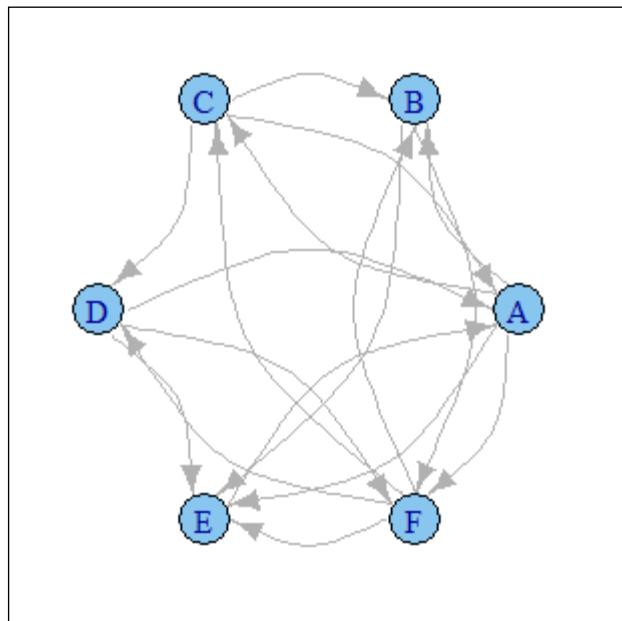
## See also

- ▶ [Spiechart](http://spatial.ly/2014/01/coxcomb-plots/) (<http://spatial.ly/2014/01/coxcomb-plots/>) does a very interesting implementation of spiechart and coxcomb plots. The plot has been generated using the `ggplot2` package. The author also provides R code for this.
- ▶ The *Worth a Thousand Words* article in *The Economist* discusses the coxcomb plot. It can be accessed at <http://www.economist.com/node/10278643>.
- ▶ An animated coxcomb plot is shown at <http://understandinguncertainty.org/coxcombs>.

## Constructing network plots

One of the first visualizations that introduced me to network plots was Visualizing Friendships. Network plots are not just limited to social networks but are observed in finance to study the linkages between Markets; they have been implemented in medicine to study the spread of viruses; and they have also been used to study social dynamics of groups, such as a network of friends. Network is an actively researched field and lots of books and articles have been published on it.

In this recipe, we will study the basics of creating a network plot using a random dataset.



# Getting ready

We will construct a network plot using the `igraph` package in R.

## How to do it...

To construct a network plot, we will install the `igraph` package and also load it in our active R session by typing the following lines of code:

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

Next, we generate fake data and import it in R using the `read.csv()` function:

```
net = read.csv("network.csv", sep = ", ", header = TRUE)
```

We will now create a network graph object using the `graph.data.frame()` function:

```
g = graph.data.frame(net)
```

The network object can be used to plot the network graph by calling the simple R plotting function:

```
plot(g)

V(g)$label = LETTERS[1:6]
plot(g, vertex.size = 25, edge.arrow.size = 0.8)
plot(g, vertex.size = 25, edge.arrow.size = 0.8, edge.curved =
TRUE, layout = layout.circle)
```

## How it works...

In order to create a network graph, we have generated a fake dataset and stored it as a `.csv` file. Once we have loaded the data in R, we can generate a network object by passing our data as an argument in the `graph.data.frame()` function. A very quick and dirty way to visualize a network is by simply passing the network object in the `plot(g)` function.

The `E(g)` and `V(g)` functions will print a list of all the edges and vertices in R. All the vertex and edges options can be declared in the basic `plot()` function, as described in the previous code. The entire list of options is available in the `igraph` library document available on the CRAN website at <http://cran.r-project.org/web/packages/igraph/igraph.pdf>.

## There's more...

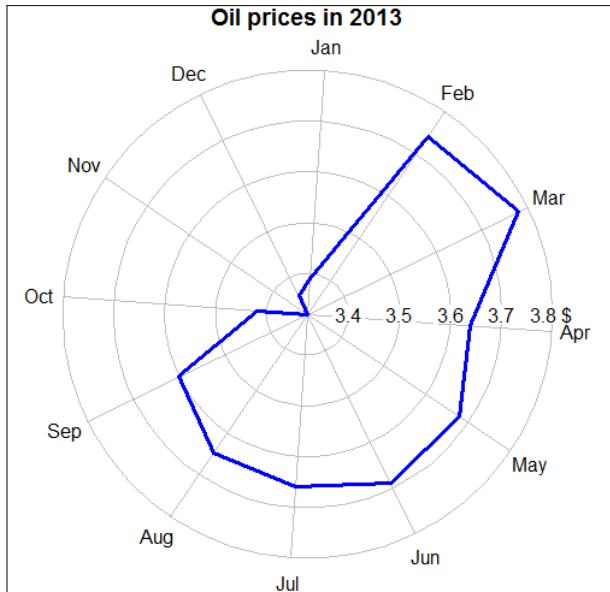
Readers who are interested in creating an interactive network plot can achieve this by using the `tkplot(g)` function in R. The function allows users to interactively change the layout as well as select edges and vertices.

## See also

- ▶ Visualizing Friendships, available at [https://www.facebook.com/note.php?note\\_id=469716398919](https://www.facebook.com/note.php?note_id=469716398919).
- ▶ Financial and Macroeconomic Connectedness provides a way to replicate the network using gephi and the R package. It can be accessed at <http://financialconnectedness.org/Stock.html#>.
- ▶ I personally prefer gephi to generate network plots. The gephi package is an open source package used to plot networks. Users can easily generate, edit, and perform basic calculations. You can refer to <https://gephi.github.io/>.

## Constructing a radial plot

The main idea behind a radial plot is to project the data as a distance from the centre in a circular form. Radial plots are not observed very often but are good tools to visualize monthly time series data. In this recipe, we will use oil prices in USA as an example to construct the radial plot.



## Getting ready

In order to create a radial plot, we need to load the `plotrix` package.

## How to do it...

We can install the package as well as load the library in our active R session by typing the following lines in the R console window:

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

The data consists of 21 years of monthly data for oil prices in USA. We import our data in R using the `read.csv()` function. Note that R will search for the file in our current R directory:

```
oil = read.csv("oil.csv")
```

We can now construct a radial plot by implementing it in R via the `radial.plot()` function:

```
radial.plot(oil[,21], rp.type="p", lwd = 3, line.col="blue",
 labels=oil$Month, clockwise = TRUE, start = 1.5,
 grid.unit = c("$"), main = "Oil prices in 2013")
```

## How it works...

The first argument in the radial plot is the data. Note that square brackets in the code `oil[,21]` instruct R to plot all the rows and column 21. The argument `rp.type` assigns the elements to our plot and the `P` argument creates a polygon. The `rp.type` argument also accepts `l` and `s` to plot radial lines and symbols respectively. The `labels` argument is used to assign the labels to our plot; in this recipe, the labels are months.

The `start` argument allows us to change the orientation of the labels. By default, January is the month drawn at 3 o'clock but, using the `start` argument, we can change it so that January begins at 12 o'clock.

The `radial.plot()` function has many other options to learn about. For these options, users should refer to the manual available on the CRAN website. I have used some of the options in the code file to display the use of various arguments in the `radial.plot()` function.

## There's more...

If we would like to plot all the data points, we do not need to run each line and change the value under `oil[,13]` to `oil[,20]`. In R, we can write a loop that goes to every column and plots the image. The following lines of code are written to demonstrate the looping capability in R:

```
plot.new()
radial.plot(oil[,2], rp.type="p", lwd = 3, line.col="blue",
 labels=oil$Month, clockwise = TRUE, start = 1.5, grid.unit =
 c("$"), main = "Oil prices cycles 1994-2013",
 radial.lim = c(0,4))
for(i in 3:21){

 radial.plot(oil[,i], rp.type="p", lwd = 3, line.col="blue",
 labels=oil$Month, clockwise = TRUE, start = 1.5, add = TRUE) }
```

We have started the code by creating a base radial plot on our plotting area. In this recipe, we would like to plot one series over the other and hence we use a loop. Note that we have used the `add = TRUE` and `radial.lim` arguments. If we omit the `radial.lim` argument, the plot area will not automatically adjust and hence the lines in a radial plot will be all over the plotting region. We determine the limit by using the `summary(oil)` function and observing the max values of each series.

To learn more on how to write a loop, please refer to the recipe *Basic loops in R* in Chapter, *A Simple Guide to R*.

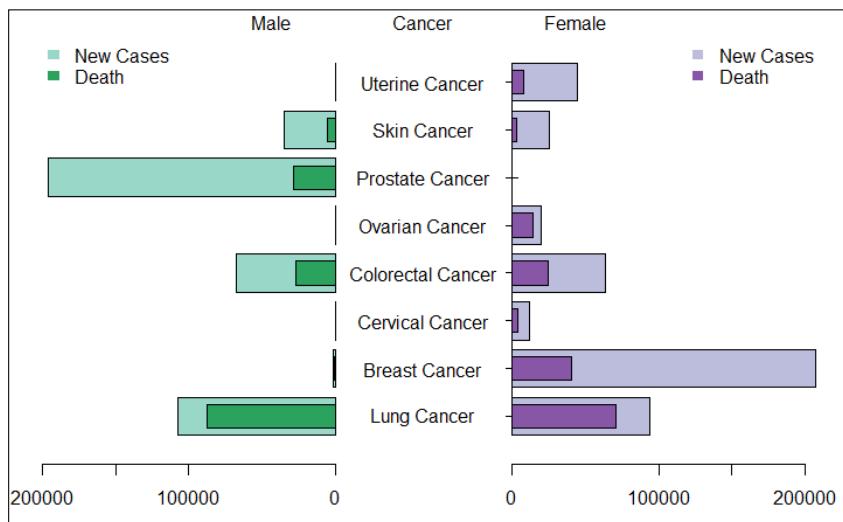
The plot generated does not allow us to understand the cycles. We have used a loop statement to show how new plots can be added to an existing plot. Nathan Yau uses radial plots to visualize cyclical data such as flight data.

## See also

- ▶ The `plotrix` manual at <http://cran.r-project.org/web/packages/plotrix/plotrix.pdf>.
- ▶ I was unable to cover two other very important plots: `StarPie` and `diamond`. Each of these plots is discussed in detail in the `plotrix` manual. The `StarPie` plot can be used to compare similarity or dissimilarity between two different datasets. The `diamond` plot is very similar to radar plots, which we often encounter in business presentations.
- ▶ *Yau, Natha(2013). "Data Points", Wiley.*

# Generating a very basic pyramid plot

You might have seen these plots in news or journal articles and wondered how to create them quickly. This recipe will help you accomplish this task. Pyramid plots are horizontal bar plots and they are often used to display gender differences in a dataset. I have created this plot based on a New York Times infographic discussing the deaths by different types of cancer among men and women. The data was extracted from the Centers of Disease Control and Prevention website.



## Getting ready

We require the following packages:

- ▶ `plotrix`
- ▶ `RColorBrewer`

## How to do it...

The `install.packages()` and `library()` functions can be used to install and load the packages in R:

```
install.packages(c("plotrix", "RColorBrewer"))
library("plotrix")
library("RColorBrewer")
```

The `read.csv()` function is used to load the CSV file in R:

```
data = read.csv("cancer.csv", sep = ", ", header = TRUE)
```

Next, we can create a pyramid plot in R using the `pyramid.plot()` function and add the legends using the `legend()` argument:

```
pyramid.plot(data$Men_g,data$Women_g,labels=data$Causes,
unit = NA, gap = 60000,laxlab=c(0,100000,150000,200000),
raxlab=c(0,100000,150000,200000), top.labels = c("Male",
"Cancer", "Female"),lxcol = "#99d8c9",rxcol = "#bcbddc")
pyramid.plot(data$Men_d,data$Women_d,labels=data$Causes,
unit = NA, gap = 60000,laxlab=c(0,100000,150000,200000),
raxlab=c(0,100000,150000,200000),lxcol = "#2ca25f",rxcol =
"#8856a7", add = TRUE, space = 0.5)
legend("topleft", fill = c("#99d8c9","#2ca25f"),
legend=c("New Cases","Death"), border = FALSE, bty= "n")
legend("topright", fill = c("#bcbddc","#8856a7"),
legend=c("New Cases","Death"), border = FALSE, bty = "n")
```

## How it works...

Our dataset comprises two sets of data. One set titled `_g` consists of all the cases that were diagnosed with cancer in 2012, whereas columns titled `_d` are individuals that died due to cancer. We have utilized both sets of data.

The `pyramid.plot()` function takes the data to be plotted on the left and on the right as its first two arguments. In order to fix the scaling on the plot we have used the `raxlab` and `laxlab` arguments. The argument `top.labels` is used to title each side of the plot. The arguments `rxcol` and `lxcol` are used to color the bars in our plot. Note that we have used the colors from the `RColorBrewer` package.

Finally, we state `add = true` to allow R to plot the second set of data on top of the first set. If the colors are not transparent, you might not see the effect clearly. We have also included the `gap` argument in the second `pyramid.plot()` function in order to make the plot look like the New York Times Visualization.

We add the legends using the `legend()` function. We will require two sets of legends to appear on each side of the plot; hence, we pass the `legend()` function twice. The first argument in the `legend()` function is the position of the legend. The `fill` argument allows the boxes to be filled with colors and the `legend` argument generates the labels to be applied. To learn more about legends, readers should type `?legend()` in the R console window.

## See also

- ▶ New York Times visualization at <http://www.nytimes.com/imagepages/2007/07/29/health/29cancer.graph.web.html>
- ▶ Centers for Disease Control and Prevention, which is a great source for data related to health, at <http://www.cdc.gov/>

# 7

# Visualizing Continuous Data

In this chapter, we will cover the following recipes:

- ▶ Generating a candlestick plot
- ▶ Generating interactive candlestick plots
- ▶ Generating a decomposed time series
- ▶ Plotting a regression line
- ▶ Constructing a box and whiskers plot
- ▶ Generating a violin plot
- ▶ Generating a quantile-quantile plot (QQ plot)
- ▶ Generating a density plot
- ▶ Generating a simple correlation plot

## Introduction

Time is continuous and we observe that, with changes in time, observations change, patterns emerge, and so do our results and conclusions. Our objective with visualizing time series or continuous data is to display visualizations that assist readers in understanding the magnitude and direction of the series. We have discussed interactive bar charts in the recipe *An interactive bar plot* in *Chapter, Basic and Interactive Plots*, and they provide us with details, which are as follows:

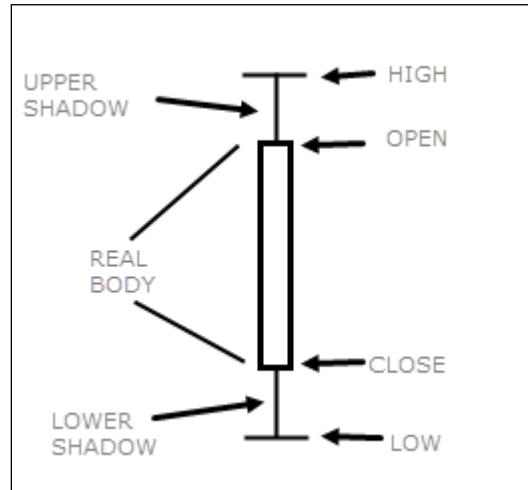
- ▶ When in the past the stock price of Microsoft was the lowest
- ▶ How often the higher prices are followed by higher prices or a dip
- ▶ How a stock's price reacts to different events such as recession, dividend announcements, quarterly results, or significant economic news

Similar information can also be studied using calendar maps and line plots discussed in *Chapter, Data in Higher Dimensions*, and *Chapter, Basic and Interactive Plots*, respectively.

One of the objectives of generating an effective visualization is that it should provide substantial information to its audience so that they can make the correct interpretation of the data. Visualizations, such as box plots, multiple scatter plots, correlation plots, and QQ plots (discussed in this chapter), assist in analyzing data by providing information such as distribution of the data, variation, and where the averages lie relative to the overall distribution, and more.

## Generating a candlestick plot

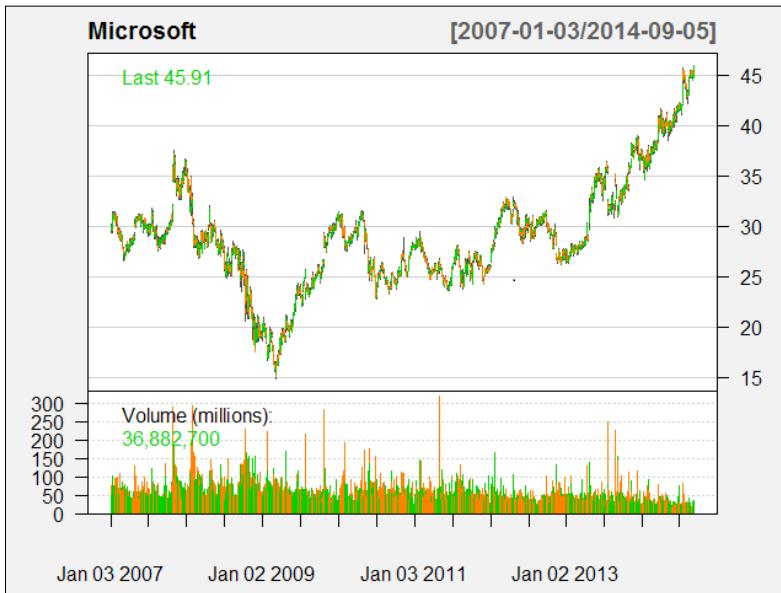
Candlestick plots have been widely used to display time series data related to financial markets. They are a combination of line charts and bar plots. They look very similar to box plots (discussed later in this chapter), but in reality, they do not share any similarity. A single candlestick is used to display the high, low, opening, and closing prices for a single security for a given time.



The difference between the open and close is shown by the size of the body. The highest and lowest traded prices are indicated by the upper and lower wick respectively. A wick is the straight bar at the top and the bottom of the real body.

Candlestick plots are used for technical analysis of the market. They help in understanding trend. Many a time, technical traders would use candlestick plots to observe and predict reversals in trends. If the size of the candlestick starts getting smaller, it may indicate a dying strategy. They are widely used in financial markets, foreign exchange markets, and commodity markets.

The `quantmod` package is implemented to generate candlestick plots in this section. We will also explore some of the useful options available with the packages, such as adding bands, moving average, and volume information.



The candlesticks generated with the `quantmod` package either have the candlesticks filled with red (to show a loss) or green (to show gains). A gain is identified when the daily opening price is higher than the closing price. A loss is identified when the daily opening price is lower than the closing price.

## Getting ready

In order to generate a candlestick plot, we require to install and load the `quantmod` package. Readers can suppress the warning messages by typing `suppressWarnings(library(quantmod))` instead of `library(quantmod)`.

## How to do it...

In the recipes in previous chapters, we imported our data from a CSV file saved in our current directory. However, in this recipe, we will import our data using the `getSymbols()` function in the `quantmod` package. The `getSymbols()` function loads data from various sources, and for the purpose of this recipe, we will define the character vector `prices`, which will hold the actual equity tickers of Microsoft and Facebook:

```
prices = c("MSFT", "FB")
getSymbols(prices)
```

We can now generate a simple candlestick plot using the `chartSeries()` function:

```
chartSeries(MSFT, name = "Microsoft", minor.ticks = FALSE,
theme = chartTheme("white"))
```

## How it works...

The `quantmod` package provides us with the ability to download historical stock prices. The `quantmod` package comes with a variety of functions, and users interested in conducting stock price evaluations should refer to the `quantmod` manual available on the CRAN website.

We will first define a character vector of all the symbols for which we would like to download the equity price data in R. We download the data using the `getSymbols()` function in R. The only argument required by the `getSymbols()` function is a list of symbols defined. Readers interested in exploring this function should type `?getSymbols` in the R console window.

The code will download two sets of data, namely, Facebook and Microsoft. The data downloaded will be in the XTS format. XTS stands for **Extensible Time Series**. Readers can learn more about the XTS format at the sources given in the *See also* section of this recipe. We can explore the data using the `head()` function as `head(MSFT)` or `head(FB)`. The data consists of six columns: open, high, low, close, volume, and adjusted. The data is downloaded from Yahoo Finance.

The `chartSeries()` function uses the XTS format to read the file and plot the candlestick. The first argument in the function is `MSFT`, which is data in the XTS format. The `name` argument can be used to provide a heading to the plot. In order to remove the clutter and make the plot look more readable, we will suppress the tick using `minor.ticks = FALSE`. We have also colored our background in white by using the `theme` argument.

## There's more...

The `chartSeries()` function can be further exploited to bar plots and line plots as well. Try pasting the following lines of code and observe how the chart changes:

```
chartSeries(MSFT, type = c("line"), name = "Microsoft", minor.ticks =
FALSE, theme = chartTheme("white"))

chartSeries(MSFT, type = c("line"), name = "Microsoft", minor.ticks =
FALSE, theme = chartTheme("white"), line.type = "h")

chartSeries(MSFT, type = c("line"), name = "Microsoft", minor.ticks =
FALSE, theme = chartTheme("white"), TA = c("addEMA()"))

chartSeries(MSFT, type = c("line"), name = "Microsoft", minor.ticks =
FALSE, theme = chartTheme("white"), TA = c("addBBands()"))
```

Each line will generate a different plot in R. Readers should note that they can also plot various technical trading tools along with the plot using the `TA` argument. In order to view the entire list of available `TA` options, readers should refer to the technical analysis and `chartSeries()` function available on the `quantmod` website.

## See also

- ▶ The `quantmod` website at <http://www.quantmod.com/examples/data/>. The link provides various examples of using the `chartSeries()` function. The website also provides various examples, manuals, and candlestick examples.
- ▶ XTS at <http://cran.r-project.org/web/packages/xts/vignettes/xts.pdf>.
- ▶ Historical prices for Facebook downloaded from Yahoo Finance at <http://finance.yahoo.com/q/hp?s=FB+Historical+Prices>.

## Generating interactive candlestick plots

We have learned a quick and easy way to plot a candlestick. In this recipe, we will learn an easy way to generate an interactive version of the same plot. The advantage of using an interactive plot is that the audience is able to interact with the visualization and this removes the guess work. Audience can hover over the chart and the values are displayed as a tooltip.



In this visualization, whenever the opening value is less than the closing value, the candlestick is filled with blue (implying a gain). Whenever the opening value is more than the closing value, the candlestick is hollow (implying a loss).

## Getting ready

We will generate an interactive candlestick plot using the `googleVis` package. The data for the same will be downloaded using the `quantmod` package.

## How to do it...

We will download and load the package in R using the `install.packages()` and `library()` functions:

```
install.packages(c("quantmod", "googleVis"))
library(quantmod)
library(googleVis)
```

The data for the interactive candlestick plot is downloaded using the `getSymbols()` function. We have discussed the function in detail in the previous recipe:

```
prices = c("MSFT")
getSymbols(prices)
```

The data downloaded is in XTS format, and to generate a candlestick plot, we require the data to be in a data frame format. We can convert our data to the data frame format using the `data.frame()` function:

```
msft = data.frame(date = as.Date(index(MSFT)), open =
 as.numeric(MSFT$MSFT.Open), close = as.numeric(MSFT$MSFT.Close),
 high = as.numeric(MSFT$MSFT.High), low =
 as.numeric(MSFT$MSFT.Low))
```

For this recipe, we will plot a part of the data. We have defined new data using the `[]` (square brackets) notation. Readers can plot the entire data in a browser by omitting the `msft_n` step:

```
msft_n = msft[1:20,]
```

The candlestick plot object is generated using the `gvisCandlestickChart()` function and the same is displayed using the `plot()` function:

```
mplot = gvisCandlestickChart(msft_n, xvar = "date", low = "low",
 open = "open", close = "close", high = "high", options = list
 (legend = 'none', width = 900, title = "Microsoft stock Price"))
plot(mplot)
```

## How it works...

We have discussed this method of creating the data frame in much detail in the recipe *Generating interactive calendar maps in Chapter, Data in Higher Dimensions*.

The `gvisCandlestickChart()` function uses six arguments to plot the data. The first argument is the data frame defined in our code as `msft_n`. The remaining five arguments are used to plot the data on the x-axis and create a candlestick. We can further modify the plot using the `options = list()` argument in our code. The `googleVis` package comes with many different options available in the manual available on the CRAN website.

# Generating a decomposed time series

A time series is dominated by seasonality as well as trend. The main objective of this section is to introduce the concept of decomposition. We can extract different elements from a time series such as trend or seasonality. The residual series is the series that is free from both, trend and seasonality.

R provides us with the tools to conduct time series analysis very efficiently. It is not possible to cover the topic of time series analysis in detail and I would highly recommend readers to refer to *Introductory Time Series with R* given in the See also section.

In this recipe, we will explore a simple additive decomposition model. The model is represented as follows:

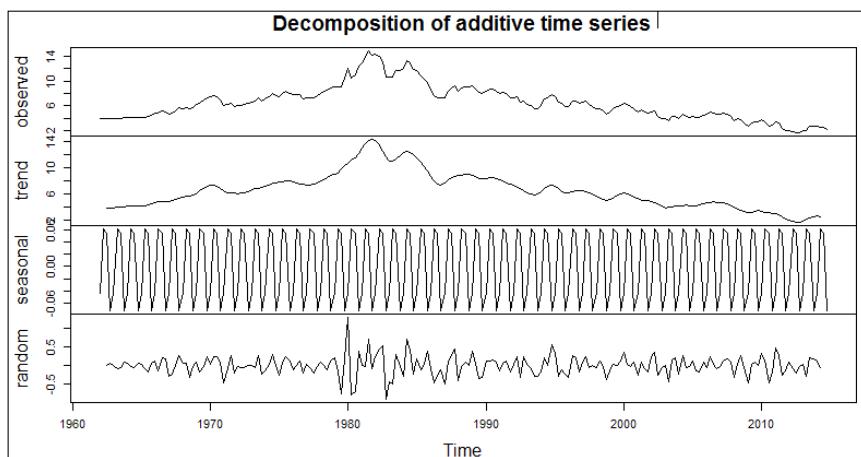
$$X_t = M_t + S_t + Z_t$$

In this notation, the variables at time  $t$  are as follows:

- ▶  $X_t$ : This is an observed series
- ▶  $M_t$ : This is the trend
- ▶  $S_t$ : This is the seasonal effect
- ▶  $Z_t$ : This is the error term

It is also possible to use a multiplicative model and it is necessary that readers interested in decomposing the time series understand both the models. The *Introductory Time Series with R* book is a great introductory resource to understand the time series analysis.

For the purpose of this recipe, the data downloaded is a quarterly 10-year treasury interest rate from FRED. The series comprises data starting from 1962/01 and ending in 2014/10. In the following plot, we observe that our original series is decomposed into seasonal, trend, and random components:



## How to do it...

For the purpose of this recipe, we will import the data as a CSV file using the `read.csv()` function. In order to decompose the series, we require the data to be in the time series format. We can convert a series to time series using the `ts()` function:

```
data = read.csv("fredgraph.csv")
datats = ts(data[,2], frequency = 4, start= c(1962,1))
```

In order to decompose a series, we will use the `decompose()` function in R:

```
d = decompose(datats)
plot(d)
```

R by default will use an additive model to change the model to multiplicative. Readers can use the following line of code:

```
d = decompose (datats, type = multiplicative)
```

## How it works...

The `ts()` function allows us to convert a series into time series. The first argument in the `ts()` function is the data imported using the `read.csv()` function. The `frequency` argument refers to the number of observations per unit in time; in our case, it would be `4` as the data is quarterly. The `start` argument provides R with a starting value or time of the first observation. Readers can learn about the `ts()` function by typing `?ts` in the R console window.

The `decompose()` function uses the time series generated by us in the previous step as its argument. We can implement the `type` argument in case we choose to use the multiplicative model. The output of the `decompose()` function are three series and they can be displayed using the `plot()` function. It is also possible to extract trend, seasonal, and residual series from the result of the `decompose()` function by using the `$` notation as follows:

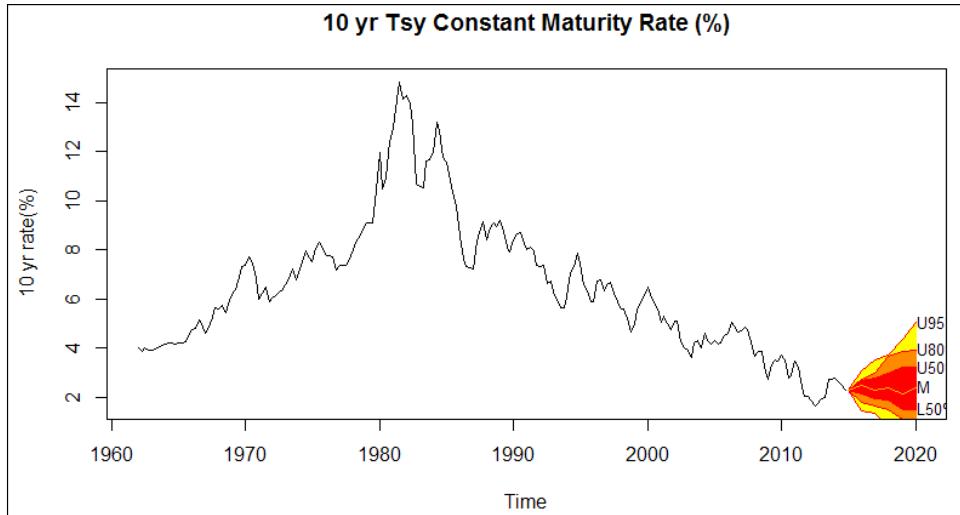
```
trend = d$trend # extracts the trend component
seasonal = d$seasonal # extracts the seasonal component
resid = d$random # extracts the residual
par(mfrow = c(2,2))
plot(datats) #plots the original series
plot(seasonal) # plots the seasonal component
plot(trend) # plots the trend component
plot(resid) # plots the residual
m = na.omit(resid) # removes the na from series
acf(m) # generates a correlogram
```

To generate a correlogram in R we would use the `acf()` function. The argument used within the `acf()` function is the residual series.

## There's more...

We will now go a step further and forecast a series of 10-year treasury rates using the `forecast` package and plot the simulated series using the `fanplot` package in R. The forecasted series is a simulated series using the **Autoregressive Integrated Moving Average (ARIMA)** model. The `fanplot` package utilizes this information to construct a fanplot.

The `fanplot` package is an excellent tool developed by Guy Abel to visualize simulated data in R.



The following code is used to fit a model, simulate the data, and generate the fanplot in R:

```
install.packages(c("forecast", "fanplot"))
library(forecast)
library(fanplot)
fit = auto.arima(datats)
fore = matrix(NA, nrow=20, ncol=5)
for(i in 1:20){
 fore[i,] <- simulate(fit, nsim=5)
}
plot(datats, xlim= c(1962,2020),main = "10 yr Tsy Constant
 Maturity Rate (%)", ylab = "10 yr rate(%)", xlab = "Time")
fan(fore, type = "interval", start = c(2014,3), anchor = 2.28)
```

The `auto.arima()` function uses the time series as its argument to fit the best ARIMA model to a univariate time series model. Readers can type `summary(fit)` to observe the result of the ARIMA model.

Our aim is to forecast for the next four years, and as the data is quarterly, we will generate an empty matrix with 20 rows and 5 columns using the `matrix()` function. We would like to simulate five different simulations. Readers can type `?simulate` in the R console window to learn more about the function.

We simulate the data using the `simulate()` function within the `for` loop. The `simulate()` function is a base R function that uses our fit model to generate five different simulations. We now use the `plot()` function to plot our original data. We overlay our basic plot with the simulated data using the `fan()` function.

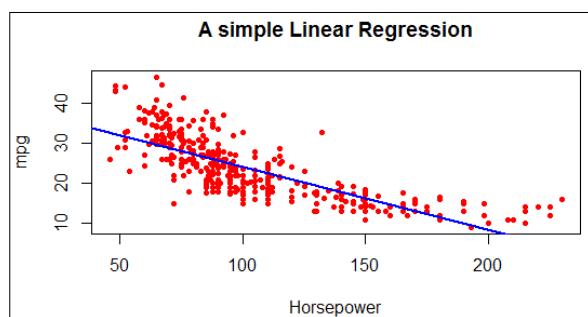
The first argument in the `fan()` function is our matrix that consists of our simulated data. The `type` argument refers to the type of percentiles to plot. Readers can choose between `interval` and `percentile`. The `start` argument refers to the time of the first distribution. The `anchor` argument refers to the last data point in the series.

## See also

- ▶ *Introductory Time Series with R*, P. Cowpertwait and A. Metcalfe, Springer.
- ▶ FRED is a database maintained by the Federal Reserve Bank of St. Louis. The link <http://research.stlouisfed.org/> is a great source to download economic data without any charge.
- ▶ *The fanplot package*, Guy Abel, at <https://gjabel.wordpress.com/2014/04/22/demo-file-for-the-fanplot-package/>.

## Plotting a regression line

All of us have studied regressions or at least heard about them in newspapers or journal articles. Regression lines are a visual representation of the regression equation. Gareth, Witten, Hastie, and Tibshirani in the book *An Introduction to Statistical Learning*, have defined a simple linear regression as, "it is a very straightforward approach for predicting a quantitative response Y on the basis of a single predictor variable X. It assumes that there is approximately a linear relationship between X and Y". Readers should refer to chapter 3 of that book to learn the concept of linear regression and its application using R.



From the preceding plot, we observe that horsepower has a negative relationship with miles per gallon (mpg).

## How to do it...

We will first load our data using the `ISLR` package. We are loading the package to use the data file and this is not required if you would like to use your own imported data:

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

To generate a plot along with the regression line, we will first generate the intercept and the estimate for horsepower using the `lm()` function. We can display our regression results as well as their confidence intervals using the `summary()` function:

```
reg = lm(mpg~horsepower, data = Auto)
summary(reg)
```

To construct the plot, we use the `plot()` function. The regression line is added to the plot using the `abline()` function:

```
plot(Auto$horsepower,Auto$mpg, pch = 20, col = "red", xlab =
 "Horsepower",ylab= "mpg", main = "A simple Linear Regression")
abline(reg, lwd = 2, col = "blue")
```

## How it works...

We would like to test the relationship between miles per gallon and horsepower. This is an example worked out in chapter 3 from the book *An Introduction to Statistical Learning*. To estimate a simple linear regression, we use the `lm()` function, where the first argument defines the formula or the relationship to be tested. The variable on the left side of the tilde sign (~) is the dependent variable (mpg), and the variable on the right side is the independent variable (horsepower). The second argument relates to the name of the data.

Multiple linear regressions can be estimated by adding more independent variables on the right side of tilde sign as `reg = lm(Auto$mpg~Auto$horsepower+Auto$Weight, data = Auto)`.

In order to view the results of the regression, we can use the `summary(reg)` or `coef(reg)` functions. The `plot()` function is a generic R plot function, and the arguments are discussed in the recipe *Introducing a scatter plot in Chapter, Basic and Interactive Plots*.

The `abline()` function uses two arguments, namely, intercept and slope. We specify both these arguments in R simply by referring to the file that contains the information, in our case `reg`.

R also allows us to predict the mpg if the horsepower changes to some other value, such as 30, 34, or 35. The predict () function is used to predict the mpg. The first argument in the predict () function corresponds to the regression object and the second argument is a data frame pr. The pr data frame consists of three values for horsepower that is used to predict the mpg:

```
pr = data.frame(horsepower= c(30,34,45))
predict(reg, pr)
```

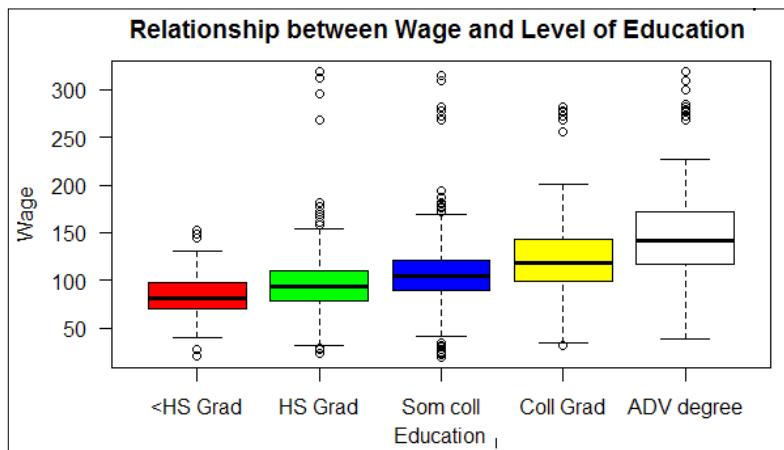
Readers can explore the predict () function by typing ?predict in the R console window. Readers familiar with Generalized Least Squares can type ?glm to learn more about the functionality in R.

## See also

- ▶ An Introduction to Statistical Learning, J Gareth, D Witten, T Hastie, and R Tibshirani, Springer, retrieved from <http://www-bcf.usc.edu/~gareth/ISL/>

## Constructing a box and whiskers plot

Box plots, also known as box and whisker diagrams, assist us in studying the distribution of the data and also provide us with information regarding the outliers. Box plots display the minimum, maximum, median, and interquartile range of each variable in the data. The Flowing Data website provides a very detailed description on how to read a box plot. The following plot shows the distribution and the outliers of wage data. We observe that as the level of education increases, the wage earned also increases.



## Getting ready

Box plots are generated in R using the basic R package. The wage data used to generate the box plot is available with the `ISLR` package and hence needs to be loaded in R. The following lines of code are used to load the package:

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

## How to do it...

We can get a general idea about the distribution of data using the `summary()` function:

```
summary(Wage)
```

We set the margin of the plot using the `mar` argument within the `par()` function. We have discussed the `mar()` function in detail under the recipe *An interactive bar plot in Chapter, Basic and Interactive Plots*:

```
par(mar = c(6, 4, 3, 6))
```

Next, we construct a simple box plot using the `boxplot()` function:

```
boxplot(Wage$wage ~ Wage$education, col =
 c("red", "green", "blue", "yellow", "white"),
 xlab = "Education",
 ylab = "Wage",
 main = "Relationship between Wage and Level of Education",
 las = TRUE,
 names = c("<HS Grad", "HS Grad", "Som coll", "Coll Grad", "ADV
degree"))
```

## How it works...

The first argument in the `boxplot()` function is the formula that states the relationship between the wage earned and the level of education attained by an individual.

The `col` argument is used to fill the box with different colors. Please note that the level of education in the dataset is divided into five categories. We know this because we ran the `summary()` function. As the `boxplot()` function is using the basic plotting functionality of R, we are familiar with many of the arguments. The `names` argument allows us to label the categories on the x-axis. We can alternatively pass the argument `horizontal = TRUE` to plot the box horizontally instead of vertically. We can also type `?boxplot` in the command window to study the options available with `boxplot()`.

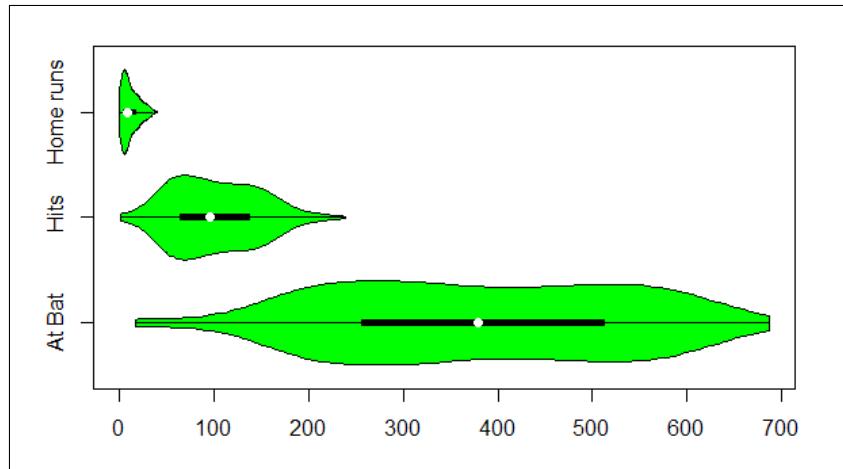
## See also

- ▶ *How to Read a Box-plot* at <http://flowingdata.com/2008/02/15/how-to-read-and-use-a-box-and-whisker-plot/>
- ▶ *How to Visualize and Compare Distributions* at <http://flowingdata.com/2012/05/15/how-to-visualize-and-compare-distributions/>

## Generating a violin plot

Violin plots are very similar to box plots. It is hard to glorify one over the other. I observe that box plots look a little more cluttered when they are plotted along with outliers.

In the following violin plot, we can observe the mean, which is displayed using white dots, and the dispersion of various variables:



## Getting ready

We need to install and load the `ISLR` and `vioplot` packages using the following lines of code:

```
install.packages(c("ISLR", "vioplot"))
library(ISLR)
library(vioplot)
```

We require the `ISLR` package to download the baseball dataset in R. The `vioplot` package is utilized to generate the violin plot.

## How to do it...

The violin plot is constructed using the `vioplot()` function:

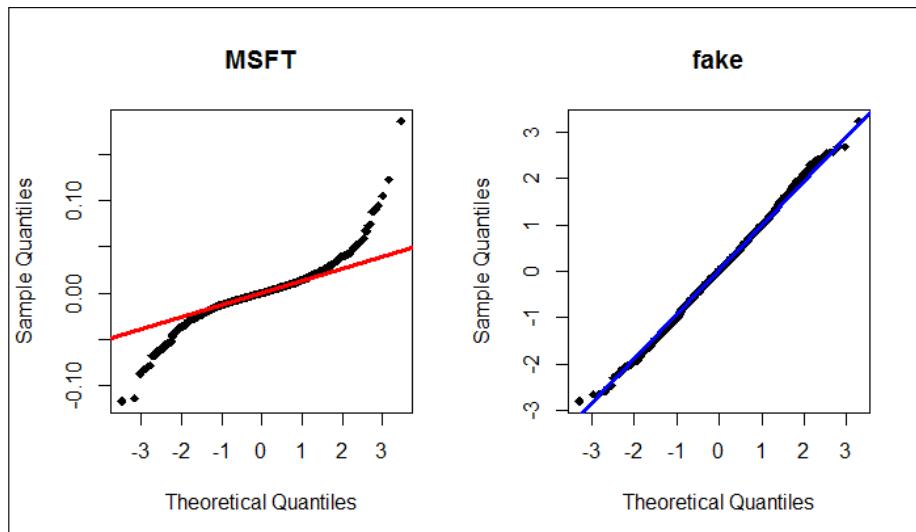
```
vioplot(Hitters$AtBat, Hitters$Hits, Hitters$HmRun, horizontal =
TRUE, col =c("green"), names = c("At Bat", " Hits", "Home runs"))
```

The violin plot can be constructed using the `vioplot()` function. The first argument in the function is the vector of data. In this recipe, we would like to plot the hits, at bats, and home runs. These are the three variables available in the `Hitters` dataset. The `horizontal = TRUE` argument controls the orientation of the plot. The `col` and `names` arguments are used to specify the background color of the violin and labels for the variables, respectively.

## Generating a quantile-quantile plot (QQ plot)

QQ plots are mainly used in academic literature to test for normality. R comes with some basic methods to test for normality, such as the Shapiro test. The Jarque-Bera test is another such normality test that is available with the `tseries` package. Many times, we are interested in understanding how much our data deviates from a normally distributed data.

One of the hot topics in finance is the study of fat tails in stock markets. Researchers have observed that equity prices or stock returns do not have a normal distribution and the actual distribution of returns contains fat tails.



# Getting ready

We need to install and load the `quantmod` package to download the historical prices of Microsoft:

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

## How to do it...

We are familiar with the `getSymbols()` function to load the data in R. All of the functions used previously have been discussed in detail under the recipe *Generating a candlestick plot*:

```
prices = c("MSFT")
getSymbols(prices)
```

The `quantmod` package also comes with some handy functions to calculate daily returns. In order to calculate daily returns, we will use the `dailyReturn()` function as follows:

```
msft = dailyReturn(MSFT)
```

To conduct a Shapiro test, we require the data to be in the form of a numeric vector in R:

```
msft = data.matrix(msft)
```

In order to test for normality, we can use the `shapiro.test()` function available in R:

```
shapiro.test(msft)
```

We are testing the null hypothesis that the sample comes from a normal distribution. An excellent interpretation of the Shapiro test in R is available on Stack Overflow.

In order to generate a simple QQ plot, we use the `qqnorm()` function. The `qqline()` function plots a theoretical line. This line can be used to measure how far our data deviates from normal distribution:

```
fake = rnorm(1000, 0, 1)
shapiro.test(msft)
par(mfrow = c(1, 2))
qqnorm(msft, pch = 18, main = "MSFT")
qqline(msft, col = "red", lwd = 3)
qqnorm(fake, pch = 18, main = "fake")
qqline(fake, col = "blue", lwd = 3)
```

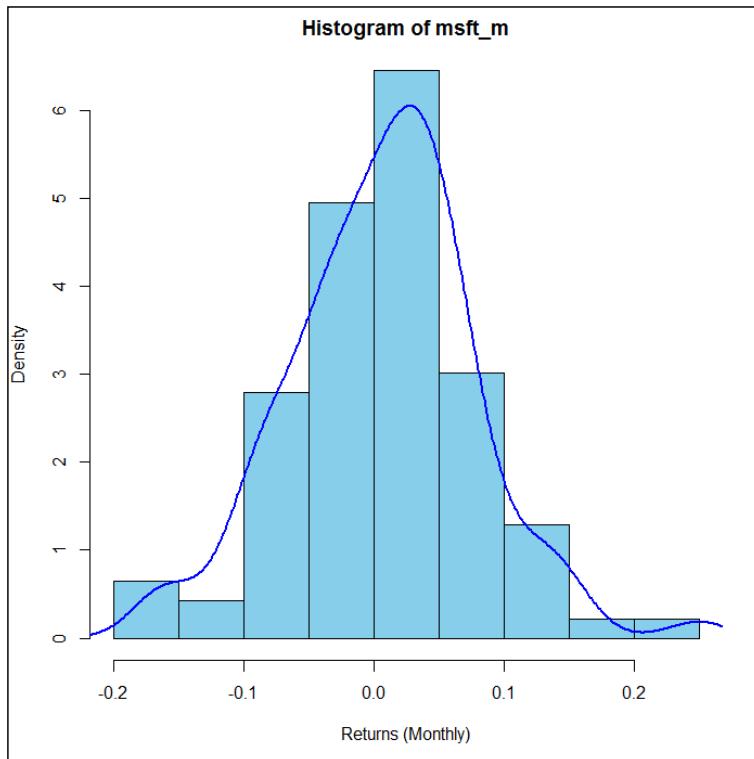
The `qqnorm()` function uses the daily returns on the Microsoft stock as its only argument. In order to show what a QQ plot of a normally distributed data looks like, I have generated fake data with a mean of 0 and standard deviation of 1. We can clearly observe that daily returns on the Microsoft stock deviates from the QQ line. However, for the fake data, all the points lie on the theoretical QQ line.

## See also

- ▶ Interpretation of the Shapiro test in R at <http://stackoverflow.com/questions/15427692/perform-a-shapiro-wilk-normality-test>

# Generating a density plot

The density plot uses the kernel density estimation to generate the distribution. In this recipe, we will utilize the `density()` function to generate a plot. The density plots can be used to study the underlying distribution of the data.



## Getting ready

We will use the `quantmod` package to download the stock prices for Microsoft and also calculate monthly returns:

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

## How to do it...

We will download the data in R using the `getSymbols()` function. Once we have the data, we can calculate the monthly returns using the `monthlyReturns()` function:

```
prices = c("MSFT")
getSymbols(prices)
msft_m = monthlyReturn(MSFT)
```

In order to generate a density plot, we will first estimate the kernel density using the `density()` function in R. Please note that we have plotted a density plot over the histogram and, hence, we need to use the `lines()` function. The `lines()` function will allow us to plot a density plot over the histogram:

```
msft_d = density(msft_m)
hist(msft_m, freq = FALSE, col = "skyblue",
 xlab = "Returns (Monthly)")
lines(msft_d,col= "blue", lwd = 2)
```

## How it works...

Histograms are a good way to visualize distributions. The `hist()` function is a generic function used in R to generate a histogram. Each bar represents the proportion of values in the range. We observe that the Microsoft stock has a slight positive skew.

Most of the arguments in the `hist()` function are self-explanatory. It is also possible to change the number of bars in a histogram by passing the `break` argument. Readers should note that too many bars might make data appear noisy and too few may conceal the underlying shape of the distribution.

## There's more...

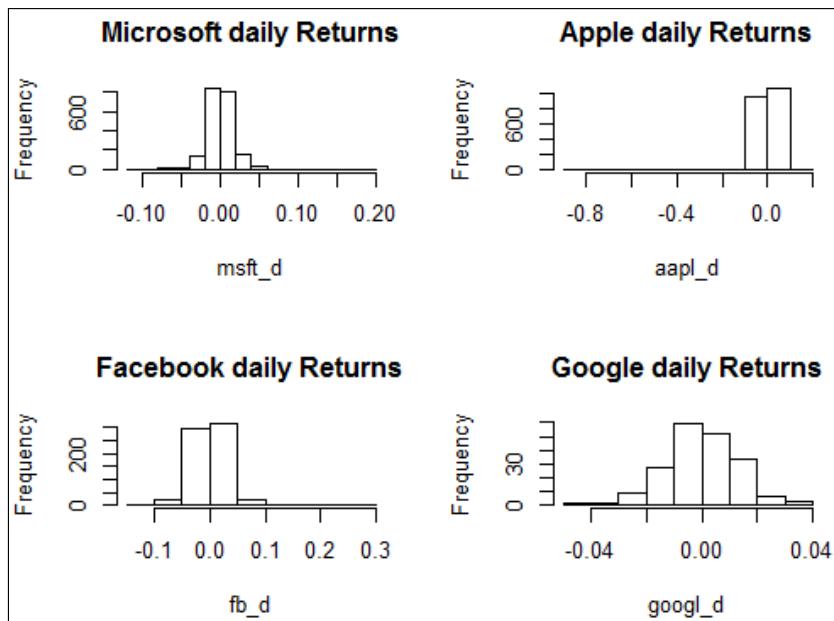
One of the very good implementations of a histogram is when multiple histograms are displayed in the same plot. This allows us to perform a comparative analysis of histograms in the same image. Looking at the visualization gives us a very good idea of how the equity prices have performed in the technology sector. We could also plot the same image using a line plot and it will also lead us to the same conclusion.

The image was generated using the following lines of code:

```
prices = c("FB", "MSFT", "AAPL", "GOOG")
msft_d = dailyReturn(MSFT)
aapl_d = dailyReturn(AAPL)
googl_d = dailyReturn(GOOG)
fb_d = dailyReturn(FB)
par(mfrow = c(2,2))
```

```
hist(msft_d, main = "Microsoft daily Returns")
hist(aapl_d, main = "Apple daily Returns")
hist(fb_d, main = "Facebook daily Returns")
hist(googl_d, main = "Google daily Returns")
```

We have studied all the arguments in the earlier recipes of this chapter. The New York Times visualizations have implemented multiple line plots and histograms to generate informative graphics.

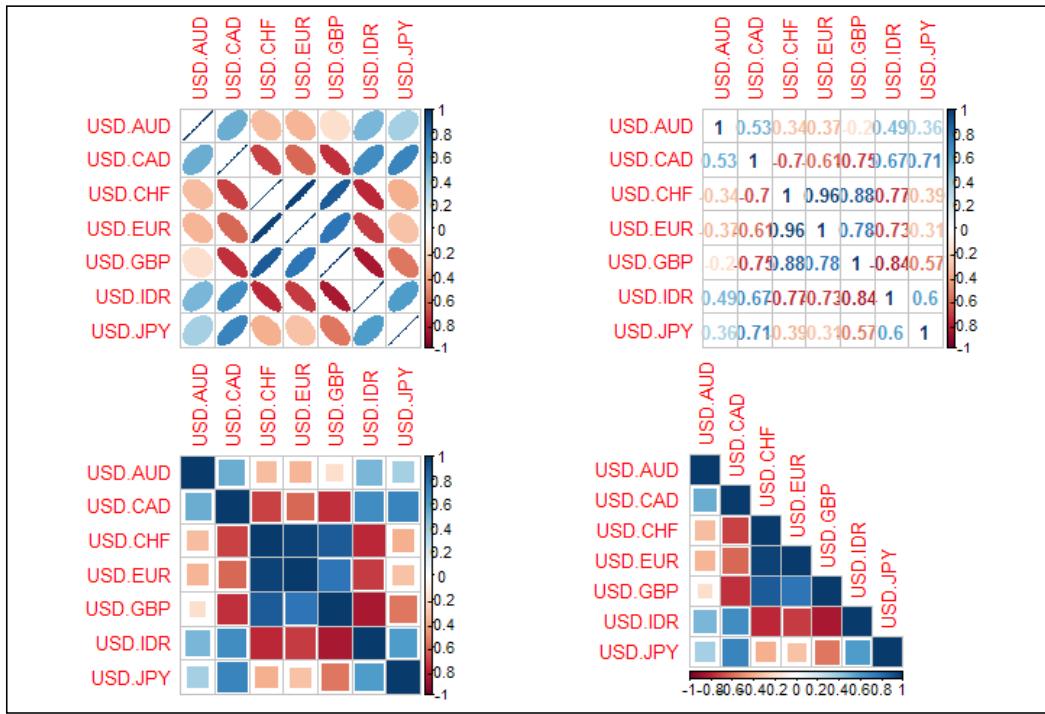


## See also

- ▶ The New York Times visualization uses multiple line plots to compare the quality of Hitters at <http://www.nytimes.com/interactive/2010/06/13/sports/20100613-score-chart.html>
- ▶ The New York Times visualization *A String of Debates* implements multiple histograms to study some of the words used by major presidential candidates, and can be accessed at <http://www.nytimes.com/imagepages/2007/12/15/us/politics/16debates.web.html>

# Generating a simple correlation plot

Correlation plots are a great tool to visualize correlation data. When two sets of data are related to one another, we say they are correlated. Hence, correlation can be negative, positive, or 0 (implying no correlation). The strength of the relationship can be defined by the correlation coefficient, which ranges from -1 (strong negative correlation) to 1 (strong positive correlation). What this implies is that when one series moves, the other series is most likely to move. The direction of the movement depends on the sign and the coefficient. Readers should note that correlation does not imply causation. Google Correlate allows its users to perform correlation on real world data.



## Getting ready

We need to install and load the following two packages in R:

- ▶ `corrplot`
- ▶ `quantmod`

## How to do it...

For the purpose of this recipe, we will download foreign exchange data of various currencies by using the `quantmod` package:

```
rates=c("USD/EUR", "USD/GBP", "USD/CHF", "USD/JPY",
 "USD/CAD", "USD/AUD", "USD/IDR")
getSymbols(rates,src="oanda", return.class = 'data.frame')
fxdata = cbind(USDAUD,USDCAD,USDCHF,USDEUR,USDGBP,USDIDR,USDJPY)
```

Until now, we were using stock price data, but in order to download the foreign exchange data, we will have to use additional arguments in the `getSymbols()` function. The source for the data is `oanda` and this is specified using `src=` attribute. Also note that we have used the `result.class()` function to download the data as a data frame. By default, the `quantmod` package downloads data in the XTS format, but we would like to use the data to generate a correlation matrix and the data frame seems like a very valid choice.

The correlation matrix in R is generated using the `corr()` function:

```
fxcor = cor(fxdata)
```

The `fxcor` object is now passed as an argument under the `corrplot()` function to generate the correlation plots:

```
par(mfrow=c(2,2))
corrplot(fxcor, method = c("ellipse"))
corrplot(fxcor, method = c("number"))
corrplot(fxcor, method = c("square"))
corrplot(fxcor, method = c("square"), type = "lower")
```

## How it works...

When we execute the `getSymbols()` function, it will download seven different datasets. However, to generate a correlation matrix, we need to bind all the data into one single dataset. Hence we use the `cbind()` function to bind all the seven data frames as `fxdata`.

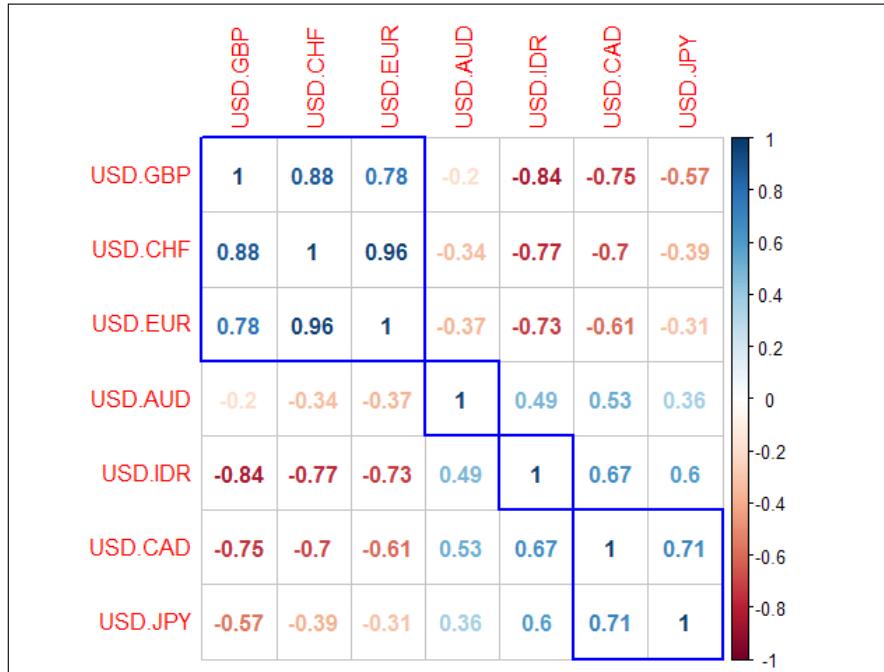
The data frame matrix is passed as an argument to the `corr()` function. Readers new to correlation can view the matrix by typing `fxcor` in the R console window. All the entries in the diagonal of a matrix are 1, as the correlation of a variable within itself is 1. All the off-diagonal entries in the matrix are correlations among currencies.

| > fxcor | USD.AUD   | USD.CAD   | USD.CHF   | USD.EUR   | USD.GBP   | USD.IDR   | USD.JPY   |
|---------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| USD.AUD | 1.0000000 | 0.7531792 | 0.7240081 | 0.7452628 | 0.5717303 | 0.7291265 | 0.8575256 |
| USD.CAD | 0.7531792 | 1.0000000 | 0.4843467 | 0.6007949 | 0.1114390 | 0.6378510 | 0.8122981 |
| USD.CHF | 0.7240081 | 0.4843467 | 1.0000000 | 0.9812093 | 0.7735754 | 0.5600675 | 0.7883815 |
| USD.EUR | 0.7452628 | 0.6007949 | 0.9812093 | 1.0000000 | 0.6789784 | 0.6192859 | 0.8551614 |
| USD.GBP | 0.5717303 | 0.1114390 | 0.7735754 | 0.6789784 | 1.0000000 | 0.1797097 | 0.4621816 |
| USD.IDR | 0.7291265 | 0.6378510 | 0.5600675 | 0.6192859 | 0.1797097 | 1.0000000 | 0.7719414 |
| USD.JPY | 0.8575256 | 0.8122981 | 0.7883815 | 0.8551614 | 0.4621816 | 0.7719414 | 1.0000000 |

The `corrplot()` function comes with many different arguments and many different options. The best place to read about all the options is the manual for the `corrplot` package available on the CRAN website. The first argument in the `corrplot()` function is the correlation matrix generated and stored as `fxcor`. The method specifies the shape to be displayed. In the last line, we have used the `type` argument, which is used to generate the lower part of the matrix, that is, the portion below the diagonal.

## There's more...

We have studied the concept of clustering in *Chapter, Basic and Interactive Plots*. We can apply the same to a correlation plot. We observe that some currencies are closely related or form a cluster and hence they would move in the same direction. If we generate a dendrogram using `fxcor` as our data, we will reach the same conclusion.



To create a correlation plot that also identifies and highlights the cluster, we can use the following line of code with our foreign exchange dataset:

```
corrplot(fxcor, method = c("number"), order="hclust",
 addrect = 4, rect.col = "blue")
```

Note that we have used the `order` = attribute to specify the hierarchical clustering and the `addrect` = argument to identify the four clusters, the rest of the attributes in the `corrplot()` function are self-explanatory.

## See also

- ▶ Google Correlate at <http://www.google.com/trends/correlate>
- ▶ Spurious Correlation is a great site based on the notion that correlation is not the same as causation; it is accessible at <http://www.tylervigen.com/>
- ▶ Wikipedia on correlation at [http://en.wikipedia.org/wiki/Correlation\\_and\\_dependence](http://en.wikipedia.org/wiki/Correlation_and_dependence)

# 8

# Visualizing Text and XKCD-style Plots

In this chapter, we will cover the following recipes:

- ▶ Generating a word cloud
- ▶ Constructing a word cloud from a document
- ▶ Generating a comparison cloud
- ▶ Constructing a correlation plot and a phrase tree
- ▶ Generating plots with custom fonts
- ▶ Generating an XKCD-style plot

## Introduction

Technology has allowed us to digitize text, and this has led to the development of tools to analyze and visualize information stored in text. One of the questions that comes to our mind is how do we utilize the information stored in text. The following are some of the applications and examples of visualization:

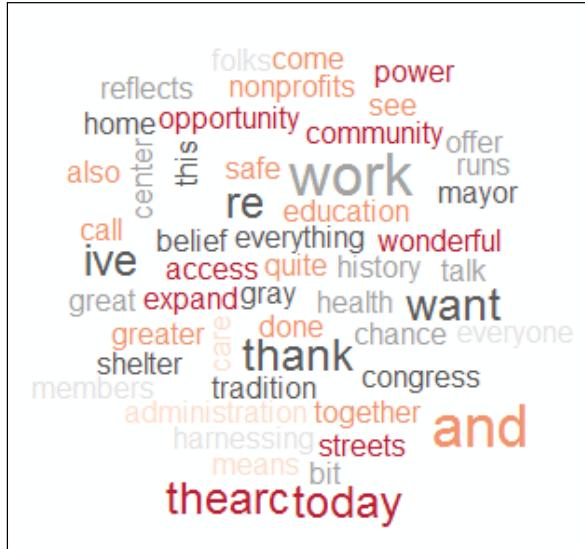
- ▶ Visualizing text provides us with information, trends, and changing patterns in literature
- ▶ Visualizing Twitter feeds can help us in revealing behavioral and social patterns
- ▶ Visualizing books reveals the writing styles of different authors
- ▶ Visualizing political speeches helps us to understand the political view, opinions, and differences among various politicians

In this chapter, we will study a few different ways to visualize text, such as a wordcloud, phrase trees and correlation plots. We will also implement customized fonts in our plots and images, making our visualization not only more appealing, but also humorous.

I would refrain from using XKCD-style fonts, which we will study in this chapter, in a visualization if you want to communicate very significant information via your visualization.

## Generating a word cloud

In this recipe, we will study how to quickly generate a word cloud in R. A word cloud is simply a graphical representation in which the size of the font used for the word corresponds to its frequency relative to others. Bigger the size of the word, higher is its frequency. Color, in this recipe, does not have any interpretation. In this recipe, the text is not formatted or processed using techniques such as stemming or by removal of stop words. We will study these text processing techniques in the next recipe.



## Getting ready

In order to generate a simple word cloud, we will use the following libraries in R:

- ▶ `wordcloud`
- ▶ `tm`
- ▶ `RColorBrewer`

## How to do it...

In order to generate a simple word cloud, we will first install the necessary packages in R using the `install.packages()` and `library()` functions:

```
install.packages(c("wordcloud", "RColorBrewer"))
library(wordcloud)
library(RColorBrewer)
```

The `RColorBrewer` package provides us with a range of color palettes that can be used via the `brewer.pal()` function in our word cloud:

```
pal = brewer.pal(6, "RdGy")
```

Finally, we can generate a word cloud by passing the text as an argument in the `wordcloud()` function:

```
wordcloud("I also want to thank all the members of Congress and my
administration who are here today for the wonderful work that
they do. I want to thank Mayor Gray and everyone here at THEARC
for having me...., min.freq = 1, scale=c(2,0.5), random.color =
TRUE, color = pal)
```

## How it works...

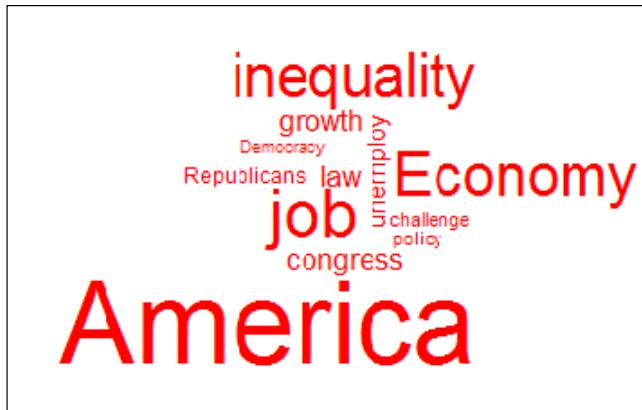
The first step in our recipe is to load the required packages in R. We then create a color palette using the `RcolorBrewer` package. The `brewer.pal()` function has two arguments; the first argument is the number of different colors we would like to use in our visualization and the second argument is the color palette. In order to view a list of all available color palettes, please refer to the `RColorBrewer` package manual at <http://cran.r-project.org/web/packages/RColorBrewer/RColorBrewer.pdf>.

We are now ready to plot a word cloud using the `wordcloud()` function. The first argument in the `wordcloud()` function is the text itself. In the next few recipes, we will learn how we can use an actual document in place of actual text. The second argument is `min.freq`, which allows us to put a lower limit on the number of words to be plotted. We have used `min.freq=1` to plot all the words.

The `scale` argument allows us to provide a range for the size of words, and finally, the `col` argument is used to pass the color palette generated via the `brewer.pal()` function.

## There's more...

We can also use the `wordcloud` function to create a plot, but instead of pasting the entire passage, we can provide the function with an array of words and their frequencies.



We have used the following code to generate the preceding word cloud:

```
wordcloud(c("inequality", "law", "policy", "unemploy", "job",
 "Economy", "Democracy", "Republicans", "challenge", "congress",
 "America", "growth"), freq=c(26, 9, 2, 7, 30, 26, 1, 4, 3, 9, 57, 9),
 min.freq = 0, col = "red")
```

In the preceding code, the first argument, `c()`, is a list of words and the `freq` argument is the number of times the words get repeated. The other arguments are discussed in the *How to do it...* section of this recipe.

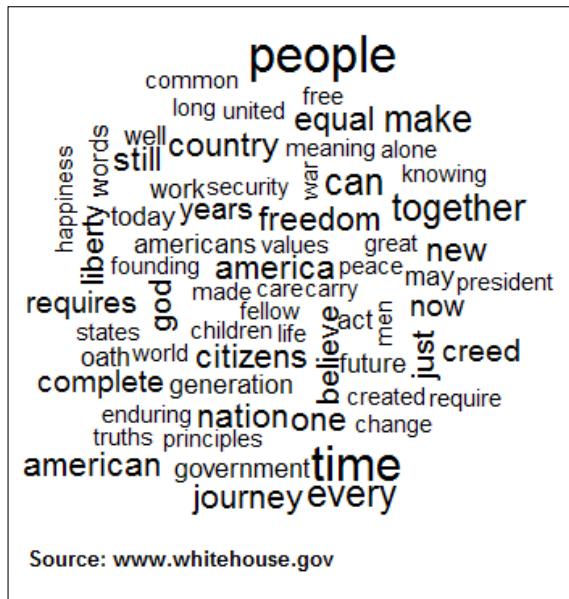
We observe that words such as "inequality" and "economy" are used more often, but words such as "law" or "policy" to decrease inequality are repeated only 9 and 2 times respectively.

## See also

- ▶ [www.wordle.net](http://www.wordle.net) is a website that allows users to copy and paste text and generate beautiful word clouds. Users can change the fonts, colors, and layout.
- ▶ <http://www-958.ibm.com/software/analytics/maneyes/> is a site maintained by IBM that also allows users to upload text and create word clouds.
- ▶ <http://shiny.rstudio.com/gallery/> provides a good example of generating an interactive word cloud using the `shiny` package. We have discussed the `shiny` package in detail in *Chapter, Creating Applications in R*.

# Constructing a word cloud from a document

In the previous recipe, we studied a quick and easy way to generate a word cloud. In this recipe, we will learn how to create a word cloud using an entire document, such as a transcript of the complete inaugural speech by President Obama. We will also learn how to process the text and structure it using the text mining package.



## Getting ready

To generate a word cloud and structure our data, we will use the following packages:

- ▶ `wordcloud`
- ▶ `tm`

## How to do it...

We will start this recipe by installing and loading the required packages in R using the `install.packages()` and `library()` functions:

```
install.packages(c("wordcloud", "tm"))
library(tm)
library(wordcloud)
```

The `readLines()` function allows us to read the file in R that contains our text. The `obama.txt` file should be saved in our current R directory:

```
file = readLines("obama.txt")
```

The text in our file is not well structured. The text file consists of punctuation, numbers, and stop words, which need to be cleaned, as we do not want our word cloud to reflect them. The `tm_map()` function allows us to remove punctuation, stop words, numbers, and specific words:

```
doc = Corpus(VectorSource(file))
doc= tm_map(doc, tolower)
doc= tm_map(doc, removePunctuation)
doc= tm_map(doc, removeNumbers)
doc= tm_map(doc, removeWords, stopwords("english"))
doc= tm_map(doc, removeWords, c("applause", "must", "will", "know"))
```

Once we have cleaned and structured our text document, we can create a word cloud using the `wordcloud()` function. If you have installed `tm` version 0.5, you would be able to create a word cloud by simply using `wordcloud(doc, scale=c(2, 0.5))`. The newer `tm` package requires the following line:

```
wordcloud(as.character(doc), scale= c(2, 0.5))
```

## How it works...

To load the data from a `.txt` file, we use the `readLines()` function. The `readLines()` function comes with other arguments, but for the purpose of this recipe, we only use the filename as its first argument. We can learn more about the `readLines()` functions in R by typing `?readLines()` in the R console window.

The documents in the `tm` package are managed using the `Corpus()` function. The `Corpus` function is a collection of documents. In this recipe, we are analyzing one document. The `VectorSource` function is used to create a vector of words. Feinerer (2014) discusses the `tm` package in detail.

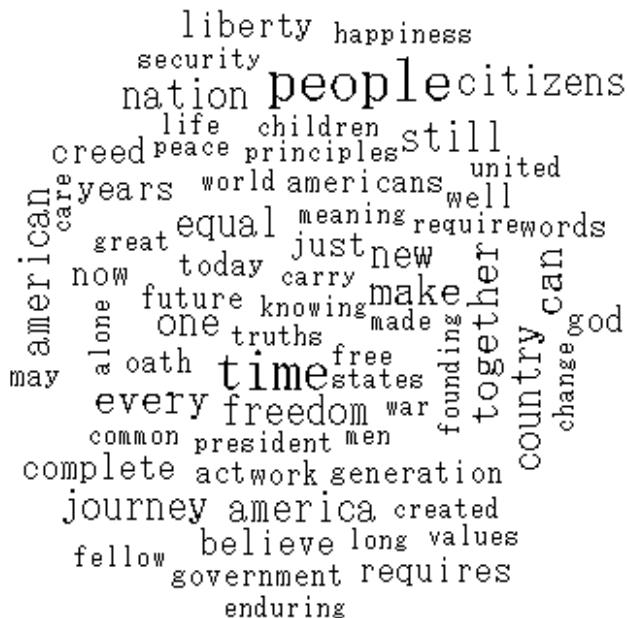
Once we have the document ready, we can modify the document by removing the stop words, punctuation, numbers, and so on. The `tm_map()` function allows us to clean our data by passing various arguments. We can convert all the words in our text document to lowercase by passing the `content_transformer(tolower)` argument. The idea behind converting the words to lowercase is to make the word cloud appear consistent.

The `removePunctuations` and `removeNumbers` arguments are self-explanatory. The `stopwords` argument will delete all the stop words in the English language. Stop words are the most commonly used words in a document. We can observe the list of all the stop words in the `tm` package by typing `stopwords("english")` in the command window. The `tm` package also supports other stop word lists such as German, Romanian, and Catalan. Please refer to the `tm` manual at <http://cran.r-project.org/web/packages/tm/tm.pdf> to understand and implement stop words from many other languages not supported by the `tm` package.

The `removeWords` argument is used not only to remove the stop words, but also to remove specific words that are not mentioned in the list of stop words. Once we have the document transformed into a structure we prefer, we then convert it into a word cloud using the `wordcloud()` function. All the arguments in the word cloud can be studied by typing `?wordcloud()` in the R command window.

**There's more...**

It is very easy to modify the fonts in R and plot them in a word cloud:



The following code will generate a word cloud with custom fonts.

```
windowsFonts(JP1 = windowsFont("MS Mincho"))
par(family = "JP1")
wordcloud(doc, scale= c(2,0.5))
```

R provides a `windowsFonts()` function, which can be used along with the `wordcloud` function. We can define a font we would like to use in the word cloud by first defining it using the following code:

```
windowsFonts(JP1 = windowsFont("MS Mincho"))
```

We can now pass the `fonts` argument in the `wordcloud` function by specifying the fonts in the `par()` function as follows:

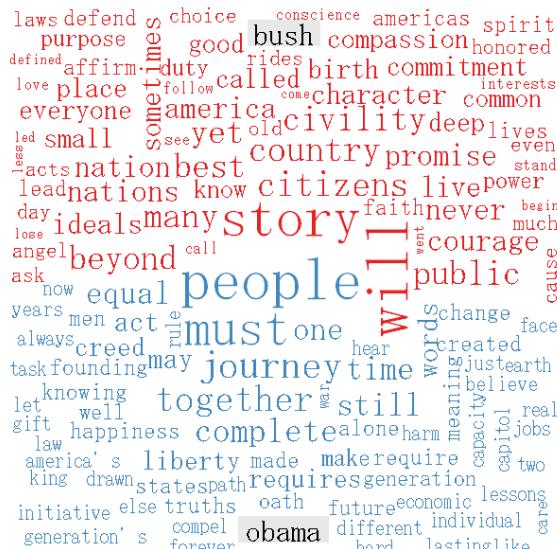
```
par(family = "JP1")
wordcloud(doc, scale= c(2, 0.5))
```

## See also

- ▶ *Introduction to the tm Package, Text Mining in R, Ingo Feinerer (2014)*, available at <http://cran.r-project.org/web/packages/tm/vignettes/tm.pdf>

## Generating a comparison cloud

A comparison cloud works on the same principles as a word cloud. A comparison cloud allows us to study the differences or similarities between two or more individuals' speeches or literature by simply plotting the word cloud of each against the other. In this recipe, we will study the inaugural speeches given by President Obama and former president George Bush. The two clouds provide us with a great contrast on how these individuals perceive the nation and its citizens. We require the `wordcloud` package as well as the `tm` package to generate the cloud. Note that a comparison cloud is not limited to just two individuals.



# Getting ready

In order to create a comparison cloud, we will use the following two packages in R:

- ▶ tm
- ▶ wordcloud

## How to do it...

In order to generate a comparison cloud, we will first install and load the two packages in R using the `install.packages()` function as well as the `library()` function. To create a comparison cloud, we will first create a new folder on our drive, call it speech, and download the two text documents (`bush.txt` and `obama.txt`) to this folder. R requires us to save the speech folder in our current R directory.

We can now load all the documents to R in the speech folder using the `DirSource()` function:

```
files = DirSource("speech/")
```

We can generate a corpus using the `Corpus()` function:

```
data = Corpus(DirSource("speech/"))
```

The rest of the code is very similar to the previous recipe. We are cleaning our text document to generate a document matrix:

```
data= tm_map(data, content_transformer(tolower))
data=tm_map(data, removePunctuation)
data=tm_map(data, removeNumbers)
data=tm_map(data, removeWords, stopwords("english"))
```

Please note that the `stopwords()` function was somehow not successful in removing many of the stop words, and hence I have utilized the `removeWords()` function to eliminate them from our analysis:

```
data=tm_map(data,removeWords,c("applause","Applause","APPLAUSE",
" And","But","will","must"))
```

To construct a comparison cloud, we require the data to be in the form of a term matrix. The `tm` package provides us with the `TermDocumentMatrix()` function that constructs a term document matrix:

```
data = TermDocumentMatrix(data)
data = as.matrix(data)
```

By default, the `tm` package will label each column in the matrix with its filename along with the `.txt` extension. To avoid this, we rename our columns using the `colnames()` function:

```
colnames(data) = c("bush", "obama")
```

We can now generate a comparison cloud in R using the `comparison.cloud()` function:

```
comparison.cloud(data, max.words = 250, title.size = 2,
colors = brewer.pal(3, "Set1"))
```

## How it works...

A corpus, in layman terms, is a collection of text documents. In the previous recipe, we worked with a single document, but we can also work with a collection of documents by harnessing the power of the `tm` package. The argument in the `Corpus()` function is the directory that contains our two text files, `obama.txt` and `bush.txt`.

We would like to plot words that actually convey the underlying interpretation of our data, and hence we clean our document by removing numbers, punctuation, and stop words. The use of all these arguments is discussed in the previous recipe. We also realize that words such as "applause" and "and" still exist; these can be eliminated by passing them as arguments in the `tm_map()` function. I realized that the `stopwords()` function fails to remove the capitalized stop words, and hence this step is necessary.

Next, we generate a term document matrix by passing our clean data as an argument in the `TermDocumentMatrix()` function. Most of the arguments in the `comparison.cloud()` function are similar to the `wordcloud()` function and the `max.words` argument limits the number of words to be displayed. We can display all the words in a document by simply altering the `max.words` argument in R. We might get a warning message after R generates the plot, and this is simply informing us that some of the words could not be fitted in our plot.

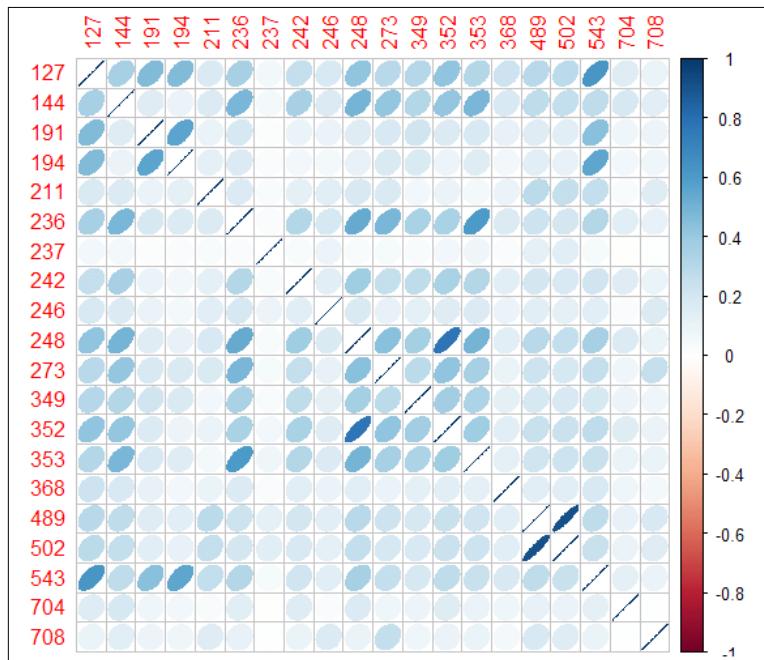
## See also

- ▶ The <http://blog.fellstat.com/?cat=11> link maintained by developers of the `wordcloud` package, provides different examples and applications about it
- ▶ Sentiment analysis using Twitter feeds is available at <http://www.r-bloggers.com/create-twitter-wordcloud-with-sentiments/>
- ▶ The text mining of the complete works of Shakespeare is available at <http://www.exegetic.biz/blog/2013/09/text-mining-the-complete-works-of-william-shakespeare/>

# Constructing a correlation plot and a phrase tree

In the previous recipe, we learned how to create a comparison cloud, which allows us to study the differences or similarities between two documents. In the process, we generated the term document matrix. In this recipe, we will learn some important matrix functions that allow us to further conduct text analysis and also generate a correlation plot.

We will also learn how to generate a phrase tree or a word tree. Wattenberg and Viegas (2008) state that a word tree places a tree structure for the words that follow a particular search term and uses that structure to arrange those words spatially. At the time of writing this book, I was unable to find a package that would allow me to construct a word tree in R. Hence, I have used an external link to demonstrate it. The latest version of Google Charts allows you to generate a word tree; however, the googleVis package in R does not.



## Getting ready

To generate a correlation plot, we will use the following two packages:

- ▶ `corrplot`
- ▶ `tm`

## How to do it...

We will install and load the packages in R using the `install.packages()` and `library()` functions.

For this recipe, we will use the `crude` dataset available in the `tm` package. The `crude` dataset consists of 20 articles related to crude oil news. We use the `data()` function to load our crude data in R:

```
data(crude)
```

We will now clean our text and use the same functions as discussed in the previous two recipes.

```
data= tm_map(crude, content_transformer(tolower))
data=tm_map(data, removePunctuation)
data=tm_map(data, removeNumbers)
data=tm_map(data, removeWords, stopwords("english"))
data = TermDocumentMatrix(data)
```

We can find most words or terms with a particular frequency by using the `findFreqTerms()` function. We can also employ the `findAssocs()` function to generate data related to terms with a lower correlation limit. Note that both these functions have no role to play in our correlation plot but they might be useful for the analysis of text documents in R.

```
findFreqTerms(data, 14)
findAssocs(data, c("oil","crude"), c(0.56))
```

To generate a simple correlation plot, we need to generate a correlation object in R. We can coerce our term document matrix to a matrix using the `as.matrix()` function. Further, we generate a correlation object by using the `cor()` function available with the `corrplot` package. Once we generate a correlation object, we can use the `corrplot()` function to create a correlation plot.

```
data = as.matrix(data)
crf = cor(data)
corrplot(crf, method = c("ellipse"))
```

## How it works...

In the previous recipes, we have discussed at length the various functions used to clean our data and the steps necessary to generate a term document matrix. The `findFreqTerms()` function takes the data as its first argument and the frequency of occurrence as the second argument. The `findFreqTerms()` function will provide users with a list of all the words that have a frequency of 14.

The `findAssocs()` function is another useful function; it uses data as its first argument, a list of words as its second argument, and the lower correlation limit as its third argument. The `findAssocs()` function will provide you all the words with which oil and crude have a correlation of 0.56 or more.

We now need to coerce our data into a matrix, and hence we employ the `as.matrix()` function. We need this step as the `corrplot` package requires data to be in the matrix form and currently our data is in the `TermDocumentMatrix()` form. The `cor()` function uses the data as its argument to generate a correlation object. The `corrplot()` function is used to construct a correlation plot. The first argument in the `corrplot()` function is the data object generated and the second argument is the method. Readers interested in learning more about the correlation plot should refer to the recipe *Generating a simple Correlation Plot in Chapter, Visualizing Continuous Data*.

## **There's more...**

A phrase tree or a word tree provides useful insight into text as it provides a context and not just the frequency of words.



As stated in the introduction of this recipe, currently R does not have a package to generate a word tree. The preceding image is President Obama's inaugural speech. To generate a very similar word tree, please refer to <https://www.jasondavies.com/wordtree/?source=obama.inauguration.2013.txt&prefix=We>.

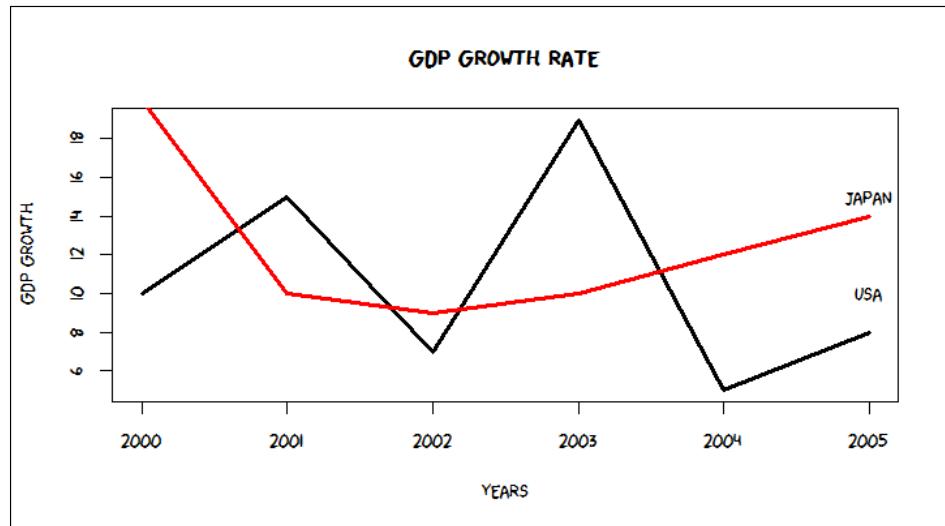
This link allows users to generate a word tree by pasting the data at the bottom of the website.

## See also

- ▶ *The Word Tree, an Interactive Visual Concordance*, Wattenberg and Viegas (2008), which can be accessed at [http://hint.fm/papers/wordtree\\_final2.pdf](http://hint.fm/papers/wordtree_final2.pdf)
- ▶ *Feinerer, Hornik & Meyer (2008), Text Mining Infrastructure in R* can be accessed at <http://www.jstatsoft.org/v25/i05/paper>, provides a very good explanation of text mining in R

## Generating plots with custom fonts

The main aim of this recipe is to introduce you to installing fonts and how to use them to label plots. We have used xkcd fonts for this recipe to introduce humor in our plots. The fonts look very similar to the XKCD cartoon strip.



## Getting ready

To download and generate plots using the `xkcd` package, we require the `sysfonts` package.

## How to do it...

The `install.packages()` and `library()` functions allow us to download the packages as well as load the library in R:

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

Next, we just copy and paste the following lines of code to the R command window. These lines of code are used to download the xkcd fonts to our R working directory. Once you execute these lines of code, you will see a file in your working directory called `xkcd.ttf`. The code is as follows:

```
download.file("http://simonsoftware.se/other/xkcd.ttf",
 dest="xkcd.ttf", mode="wb")
font.paths()
system("mkdir ~/.fonts")
system("cp xkcd.ttf -t ~/.fonts")
font.files()
font.add("xkcd", regular = "xkcd.ttf")
font.families()
windowsFonts(xk = windowsFont("xk"))
par(family = "xk")
```

Manzanera (2014) provides a detailed explanation of the `xkcd` package as well as the method to download the fonts. Once we have the file `xkcd.ttf`, all we have to do is copy and paste the font file to the Windows fonts folder. The Windows fonts folder is usually located under the control panel. We are all set now to use these new fonts in R.

We use the `windowsFonts(xk = windowsFont())` function to load our `xkcd` fonts in R:

```
windowsFonts(xk = windowsFont("xkcd"))
```

We have used the `data.frame()` function to create some fake data for our chart:

```
fake1 = c(2000:2005)
fake_us = c(10,15,7,19,5,8)
fake_jp = c(20,10,9,10,12,14)
data = cbind(fake1,fake_us,fake_jp)
data= data.frame(data)
colnames(data)= c("years", "USA", "Japan")
```

The `par()` function allows us to set many of the graphical parameters, including fonts:

```
par(family = "xk")
```

We can now display the plot using the `plot()` function. We also use the `text()` function to apply labels to our chart:

```
plot(data$years,data$USA, type = "l", lwd = 3,
main = "GDP growth rate", xlab = c("YEARS"),
ylab = c("GDP growth"))
lines(data$years,data$Japan, type = "l", lwd = 3, col = "red")
text(2005,15, "Japan")
text(2005,10, "USA")
```

## How it works...

In order to load custom fonts in R, we pass them as arguments in the `windowsFonts()` function:

```
windowsFonts(xk = windowsFont("xkcd"))
```

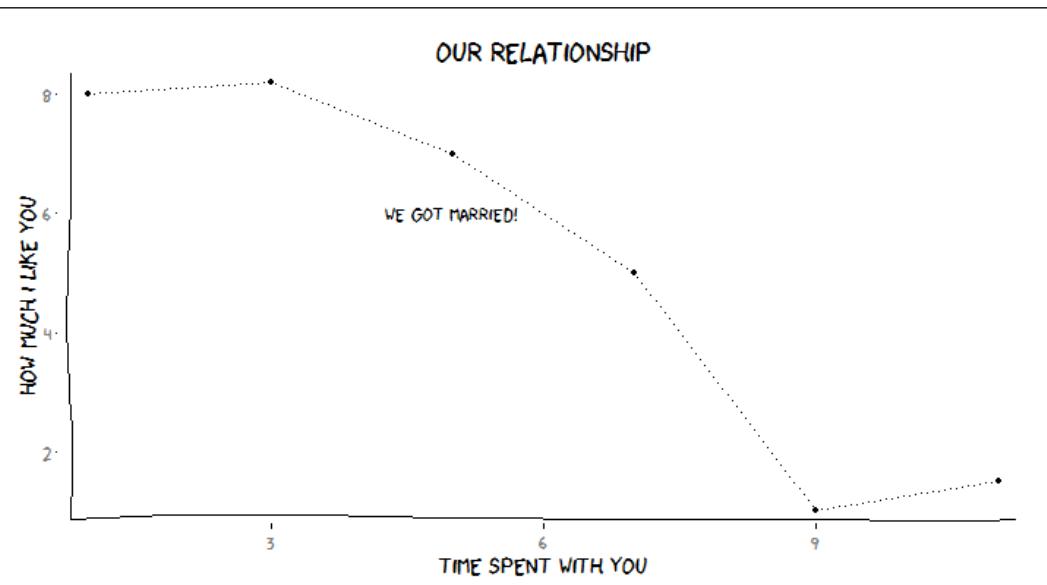
Over the course of the book, we have studied the application of the `cbind()` and `data.frame()` functions. The `par()` function allows us to specify various elements for a plot. The `family` argument in the `par()` function allows us to set a custom font for our chart. In our case, it is `xk`, as defined under the `windowsFonts()` function.

## See also

- ▶ *xkcd: An R Package for Plotting XKCD Graphs*, Emilio Torres-Manzanera, 2014, discusses the code used to download the xkcd fonts in R. The document is available at <http://cran.r-project.org/web/packages/xkcd/vignettes/xkcd-intro.pdf>.

## Generating an XKCD-style plot

If you have never read or seen xkcd cartoon strips, then I would highly recommend that you do. Some of the humor is sarcastic and entertaining. The idea behind generating an XKCD-style plot is to bring the same humor to our plot and try to make our visualization convey an idea or a story which is interesting and entertaining.



# Getting ready

To create a chart, we require to install the following packages:

- ▶ `xkcd`
- ▶ `ggplot2`

## How to do it...

We will install the two required packages and load them in our active R session using the `install.packages()` and `library()` functions:

```
install.packages(c("xkcd", "ggplot2"))
windowsFonts(xk = windowsFont("xkcd"))
library(xkcd)
library(ggplot2)
```

The `windowsFonts()` function allows us to customize our fonts in R:

```
windowsFonts(xk = windowsFont("xkcd"))
```

In order to create a theme, we use the `theme()` function in `ggplot2`. Note that our aim in this recipe is to understand ways to generate `xkcd` plots. Users can refer to the `ggplot2` manual for more clarification on the `ggplot2` functions:

```
theme_xk = theme(text=element_text(family = "xk" , size = 18),
 axis.ticks.x = element_line(linetype = 1),
 line = element_line(linetype = 3))
```

We will further create some fake data for our plot:

```
x = c(1,3,5,7,9,11)
y = c(8,8.2,7,5,1,1.5)
xrange = range(x)
yrange = range(y)
```

The following line uses functions from the `ggplot2` package. The `ggplot2` package is widely used by the R community. My aim in this book was not to concentrate more on `ggplot2` functions, as the plotting functions are very well described in the help manual as well as the `ggplot2` package website.

The `ggplot()` function initializes the `ggplot` object. The `geom_line()` function is used to define the characteristics of a line plot. The `xkcdaxis()` function is available with the `xkcd` package and implements the `xkcd` style of plotting. The `xlab`, `ylab`, and `labs` arguments are self-explanatory:

```
p <- ggplot() + geom_line(aes(x,y),linetype =3) +
 xkcdaxis(xrange,yrange)+geom_point(aes(x,y)) +
 xlab("Time spent with you") + ylab("how much i like you") +
 labs(title = "our relationship")
```

Finally, we can construct our plot by adding the theme to our `ggplot` object `p`. The `annotate()` argument adds the text labels to our plot:

```
p+ theme_xk+annotate("text", x=5, y = 6, label = "We got
married!",family = "xk")
```

Many of the functions used in this recipe are based on the `ggplot2` package. Hence, if you would like to use the `xkcd` package, it would be very helpful if you understand the working of the `ggplot2` package.

## See also

- ▶ The `ggplot2` website has examples and references related to `ggplot2` and it can be accessed at <http://ggplot2.org/>
- ▶ A list of `ggplot2` functions is given at <http://docs.ggplot2.org/current/>

# 9

# Creating Applications in R

In this chapter, we will cover the following recipes:

- ▶ Creating animated plots in R
- ▶ Creating a presentation in R
- ▶ A basic introduction to API and XML
- ▶ Constructing a bar plot using XML in R
- ▶ Creating a very simple shiny app in R

## Introduction

This chapter is specifically kept towards the end as you need to have a good understanding of R and its packages to be able to work through it. My aim behind writing this chapter is to introduce users of R to the concept of interactivity and animation, ways to combine plots, develop a good understanding of API and XML technologies, and how they could be used to better enhance our visualizations.

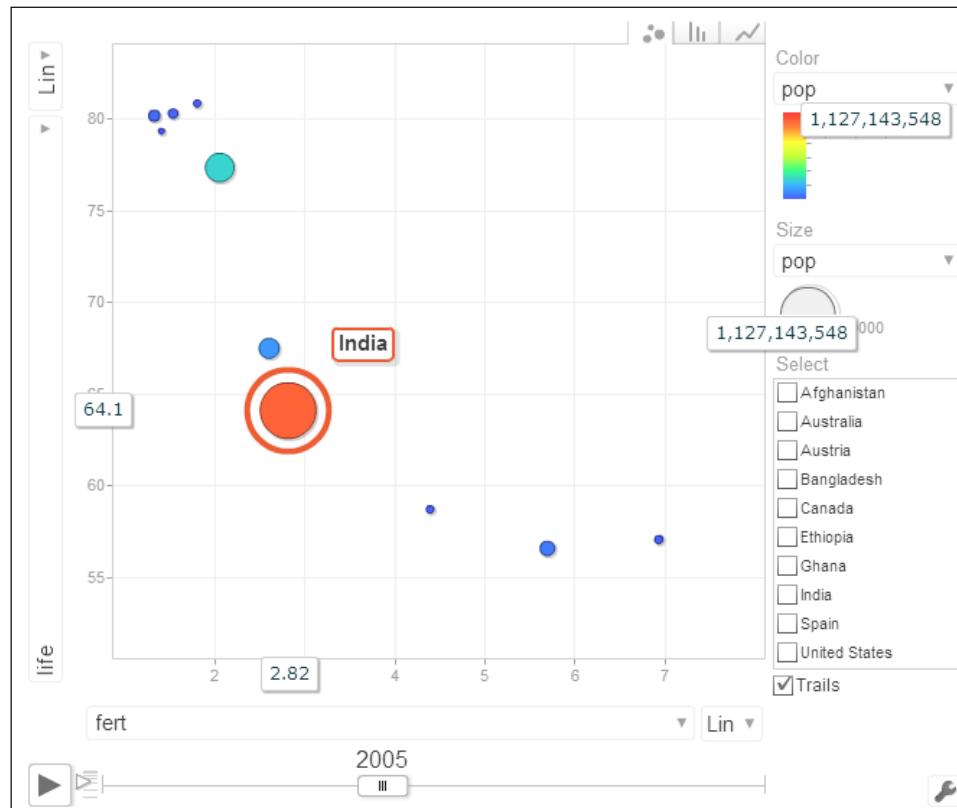
We will learn how to generate powerful and meaningful presentations using the `slidify` package. We will also learn to create web applications and animation in R using the `shiny` and `googleVis` packages. Finally, we learn about API and XML and how they can be effectively used to scrap data from the Web and generate meaningful visualizations. You might find recipes in this chapter a bit more complex, but you might be able to deepen your knowledge of the concepts using the references mentioned under the See also section in each recipe.

# Creating animated plots in R

Readers were exposed to the idea of animation in R using the animation packages in the recipe *Animating a 3D surface plot in chapter 5, Adding the Third Dimension*, but animation in R would seem incomplete without the use of the `googleVis` package. Animating a visualization brings a new dimension to our visualization. It helps to be a good storyteller, but when the story is accompanied by animation or a plot, it adds force to the story. Many of you would agree that the Web is filled with animated visualizations but there are some that have made a great impact. Here are links to some of the really good visualizations I have come across:

- ▶ Gun violence at <http://guns.periscopepic.com/?year=2013>
- ▶ Nytimes at [http://www.nytimes.com/interactive/2009/09/12/business/financial-markets-graphic.html?\\_r=0](http://www.nytimes.com/interactive/2009/09/12/business/financial-markets-graphic.html?_r=0)
- ▶ Hans Rosling's talk and animated visualization at [http://www.ted.com/talks/hans\\_rosling\\_shows\\_the\\_best\\_stats\\_you\\_ve Ever\\_seen](http://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve Ever_seen)

One visualization is shown as follows:



## Getting ready

We would require to install and load the `googleVis` package to generate an animated plot in R.

## How to do it...

For this recipe, we have downloaded data from the World Bank website. It is necessary that the data be in a particular format, hence, once we download the data, we have to structure it in a format that is well interpreted by the `googleVis` package. Please refer to the `final.txt` file as well as `life.csv`, `pop.csv` and `fert.csv` to understand the raw data used in our visualization.

As we have done in the previous recipes, we will install as well as load the `googleVis` package in R using the `install.packages()` and `library()` functions.

The data is imported in our R session via the `read.csv()` function. Note that the file `final.csv` contains the data to be implemented in our animation and this file has to be stored in our current R directory. If you would like to change your current directory in R, you can use the `setwd()` function:

```
data1 = read.csv("final.csv", sep = ",", header = TRUE)
```

We can now generate the `googleVis` object for animated visualizations using the `gvisMotionChart()` function and execute it using the `plot()` function. Note that this visualization opens up a new browser and hence requires Internet connectivity:

```
chart<- gvisMotionChart(data1, idvar="Country",
 timevar="Years", xvar = "fert",yvar = "life")
plot(chart)
```

When we execute the preceding code, R will open a new window with an interactive screen. The `googleVis` package provides its users with the option to switch between bubble, bar, and line charts. The play button at the bottom left will start the animation. Users can also select specific countries from the list and observe how they progressed over time in relation to similar economies. The color drop-down menu will allow users to change the colors of bubbles.

## How it works...

The `gvisMotionchart()` function requires the data to be in a data frame format and it is used as its first argument. In our case, `data1` is passed as the first argument. The second argument is the `idvar` argument, which is the column to be analyzed; in our case, this would be `country`. The third argument, `timevar`, adds the time dimension to our plot.

The `xvar` and `yvar` arguments are self-explanatory. We can also add the `sizevar` as well as the `colorvar` argument. These two arguments, along with the `options` argument are explained in the `googleVis` manual available at <http://cran.r-project.org/web/packages/googleVis/googleVis.pdf>.

The `gvisMotionChart` function will create an object. We would have to utilize the generic `plot()` function to create an animated plot. Note that executing the `plot` function will open a new browser and create the plot.

## Creating a presentation in R

Every individual, irrespective of their professional or academic background, will end up creating a presentation at least once in a lifetime. One of the issues while creating presentations using PowerPoint is that we have to manually update the data, contents, and plot. As this book is related to the topic of generating visualizations in R and narrating an effective story, it becomes necessary that we understand how to use R to generate slides to present the data to our audience. The slides in R are created using the `slidify` package. The package uses HTML and `knitr` to generate slides. In this recipe, we will learn to write sections for code, customize slides to include `googleVis` plots, and write mathematical equations.

### Getting ready

To generate presentations in R, we will require the following two packages:

- ▶ `slidify`
- ▶ `devtools`

### How to do it...

For this recipe, we will not implement the usual `install.packages()` function. The reason being that the `slidify` package is not available on CRAN, but is available on GitHub. In order to install `slidify`, we will use the `install.packages()` function to install the `devtools` package, which helps us connect to GitHub.

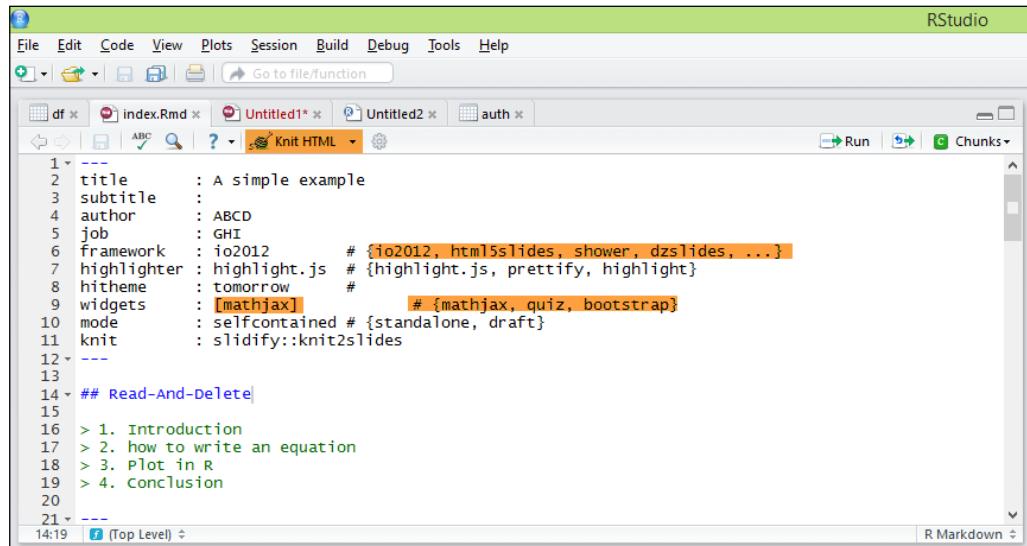
In order to install the package from GitHub, we would use the `install_github('slidify', 'ramnathv')` function. We can now load our `slidify` library in R using the `library(slidify)` function.

To start creating a presentation, we need to type `author("plot")` in the R console window. The `slidify` package will create an `index.rmd` file under a folder called `plot`. The `plot` folder is stored in our current directory. Readers with experience in CSS style sheets and HTML can explore folders inside the `plot` folder, but for the purpose of this recipe, we will not study different folders generated by `slidify`. The folders and files can be edited to generate customized presentations.

We observe that `slidify` will open a file called `index.rmd`. This is the file we require to create presentations. Copy and paste the code from the `code.txt` file to `index.rmd` in your R sessions. You should also be able to open the presentation code (`index.rmd`) by navigating to the `plot` folder. Now click on **knit HTML** to generate the `slidify` presentation. You can go backward and forward using the arrow keys on your keyboard.

## How it works...

The first section of information is related to the first slide / welcome slide. We can change `framework: io2012` to `html5slides` or `dzslides`. The framework controls the transitioning of slides: we can update it and observe the difference. Also note that if we want equations to appear on the slide, we need to include `mathjax` as a widget. The `mathjax` widget is necessary to convert LaTeX-type equations.



```
File Edit Code View Plots Session Build Debug Tools Help
df x index.Rmd x Untitled1* x Untitled2 x auth x
Knit HTML Run Chunks
1 ---
2 title : A simple example
3 subtitle :
4 author : ABCD
5 job : GHI
6 framework : io2012 # {io2012, html5slides, shower, dzslides, ...}
7 highlighter: highlight.js # {highlight.js, prettyify, highlight}
8 hitthem : tomorrow #
9 widgets : [mathjax] # {mathjax, quiz, bootstrap}
10 mode : selfcontained # {standalone, draft}
11 knit : slidify::knit2slides
12 ---
13
14 ## Read-And-Delete
15
16 > 1. Introduction
17 > 2. how to write an equation
18 > 3. Plot in R
19 > 4. Conclusion
20
21 ---
14:19 (Top Level) R Markdown
```

In order to create a new slide, we will use the --- notations and ## are used to apply a heading to the slide. The > notation is used to generate an automated number list. The dollar (\$) sign is used to embed an equation in the R code. The following screenshot shows the use of various special characters:

```
13
14 -## Read-And-Delete
15
16 > 1. Introduction
17 > 2. how to write an equation
18 > 3. Plot in R
19 > 4. Conclusion
20
21 -
22
23 -## Writting an Equations in R
24
25 -# $ CMR = \hat{ \alpha } + \ \hat{ \beta } PGNP + \ \hat{ \gamma } FLR + \ \hat{ \epsilon }
26 where : CMR = Child Mortality Rate
27 PGNP = per capita GNP
28 FMR = Female Literacy Rate
29
```

To embed a code and execute it in our slide, we will begin the slide with `~~{r}` and end with `~~`. Any R code written inside this will be executed and both the code as well as the output will be displayed in the presentation slide. To embed an equation in R in our slide, we will specify the LaTeX format within the \$ sign. There is a lot of material online on how to use the LaTeX format or convert your equation to the LaTeX format.

An image can be embedded by simply using the image as ! [] (<file path>/image1.png). If you would like to embed an image, you need to specify the file path to the folder that contains the image. Please refer to the See also section of this recipe for links related to slidify and R markdown.

To print the presentation, you can open the slides in a browser window and print them to PDF. To open the slides in a browser window, simply go to the plot folder and open the index.html file with Google Chrome. Also, we can publish the slides to <http://rpubs.com/> to access them anywhere, or we can send the link to share them.

## There's more...

To edit a slide in R, we need a basic understanding of the HTML and CSS languages. If you open your current directory, you will observe a folder named plots. This is the folder we generated using the author("plots") function.

Navigate to the slidify.css file located under plots/libraries/frameworks/io2012/css. If we open the file with a text editor, we can edit the CSS elements and customize our slides accordingly. We have to simply find the following code:

```
title-slide {
background-color: #CBE7A5;
```

We need to change this to the following code:

```
title-slide {
background-color: #6b85e1;
```

If you run our presentation again, you can observe that the slide color has changed. You can explore the Stack Overflow website to gain a deeper understanding on how various CSS elements can be edited to customize slidify slides.

## See also

- ▶ R markdown at <http://rmarkdown.rstudio.com/>
- ▶ A quick and easy way to learn all the notations at <http://rmarkdown.rstudio.com/RMarkdownCheatSheet.pdf>
- ▶ The official slidify website at <http://slidify.github.io/>
- ▶ The Stack Overflow website at <http://stackoverflow.com/questions/20875593/how-to-control-the-background-color-of-the-first-slidify-slide>
- ▶ Customizing slide layouts at <http://stackoverflow.com/questions/15259455/customizing-slide-layouts-in-slidify>

## A basic introduction to API and XML

**API** is an abbreviation for **Application Programming Interface**. According to Wikipedia:

*"API is a set of routines, protocols, and tools for building software applications."*

In our case, we will use the web API to connect to different websites and download the data. Here are some examples of API and XML technologies that can be used with R:

- ▶ Most of the news agencies allow users to download data related to articles, news, and so on
- ▶ Many social networking websites such as Twitter and Facebook allow their users to download data related to status updates, friend lists, photos, and links
- ▶ Google API allows its users to download various kinds of data, such as data related to distances, location, books, authors, web searches, and so on

Note that many websites charge for their services and creating a login is necessary. Many websites allow a certain number of calls to the API service free of charge. Readers new to API should also note that some websites have a separate section for their API services and I usually search for a link that says developer to get to the API site. Please refer to the See also section for links related to these API sites.

**XML** stands for **Extensible Markup Language**. When we connect to the API, we can choose the format in which the data should be delivered and one of the methods of delivery is XML. JSON and HTML provide other alternative forms of data delivery, but in this recipe, we will concentrate on the XML format.

## Getting ready

In order to access an API, we need to register with the website. In our recipe, we will use the New York Times API. Once we register on the website, we would be able to access the developer's website at <http://developer.nytimes.com/>.

The screenshot shows the 'My API Keys' section of the New York Times developer portal. On the left, there's a sidebar with links: Overview, Available APIs (which is highlighted), Keys, Forum, Gallery, and API Console. The main area has a title 'My API Keys' and a sub-section 'Article Search API'. Below this, there's a 'Key:' input field and a table with key information:

|              |                      |
|--------------|----------------------|
| Application: | test                 |
| Key:         | <input type="text"/> |
| Status:      | active               |
| Registered:  | 2 years ago          |

Below the table is a section titled 'Key Rate Limits' with two entries:

|        |                  |
|--------|------------------|
| 10     | Calls per second |
| 10,000 | Calls per day    |

We will concentrate on the available API, keys, and API console. The **Available APIs** section, as the name suggests, lists all the APIs made available by the New York Times. Each listed API is clickable and lists examples and methods.

The **Keys** section lists all the registered keys. The New York Times requires all the users to register a key for every API before it allows them to download the data. To register a key, click on the **Real Estate API** and type a name for your API; in our case, I have registered it as test.

The **API Console** section provides us with information to understand how the API works. We will discuss the console under the *How it works...* section.

## How to do it...

Click on the **API Console** section and select **Real Estate API** from the drop-down menu. API is nothing but a link that allows us to communicate with the website. To create a link with all the information, we will use the console.

Real estate API is divided in four parts and we will use the sales count to get an estimate of the sales in the Manhattan area in the fourth quarter of 2008. We see a menu with a lot of available options; fill in the details as per the following screenshot and click on **Try it!**

GET Sales Counts real-estate/v2/sales/count.format

| Parameter           | Value                          | Type         | Description                                                                                                       |
|---------------------|--------------------------------|--------------|-------------------------------------------------------------------------------------------------------------------|
| format              | xml                            | extension    | Select the response format.                                                                                       |
| date-range          | 2008-Q4                        | date         | Specify quarter, week, month or day. See full documentation.                                                      |
| geo-extent-level    | borough                        | string       | Specify extent of results. Use with geo-extent-value parameter.                                                   |
| geo-extent-value    | Manhattan                      | string       | Specify a borough, neighborhood or ZIP code. Use with geo-extent-level parameter. See full documentation.         |
| geo-summary-level   | neighborhood                   | string       | Group results within specified extent with this value. Use with geo-extent-level and geo-extent-value parameters. |
| bedrooms            |                                | integer      | Limit results by number of bedrooms.                                                                              |
| building-built-year |                                | date         | Limit results by the year the property was built, YYYY.                                                           |
| building-type-id    |                                | integer      | Limit results by type of building. See full documentation.                                                        |
| loft                |                                | boolean      | Limit results to lofts.                                                                                           |
| api-key             | <b>paste your API key here</b> | alphanumeric | Switch out the sample key for your API key.                                                                       |

**Try it!** [Clear Results](#)

The API will run and create output right under the **Try it!** section. Please note that the API will not work unless we provide it with a registered API key. If we copy the entire request URL and paste it in a browser, we will observe the entire XML output. Please note that we could have also opted for the JSON format.

## How it works...

The API URL consists of all the necessary data requested by us, including the API key. Readers can explore various options and various other APIs offered by the New York Times. The next recipe explores the options of running the API from R.

Request URI  
`http://api.nytimes.com/svc/real-estate/v2/sales/count.xml?date-range=2008-04&geo-extent-level=borough&geo-extent-value=Manhattan&geo-summary-level=neighborhood&api-key=[REDACTED]`

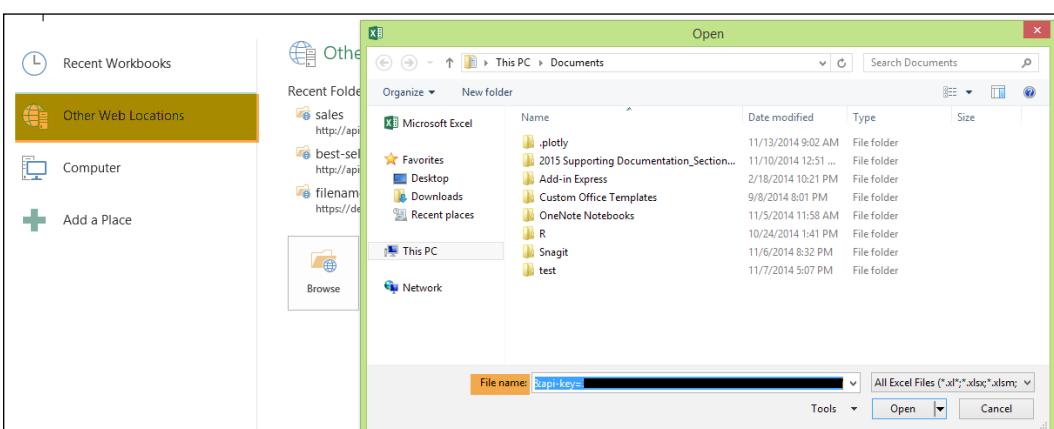
Request Headers [Select content](#)  
X-Originating-Ip: 167.219.88.140

Response Status [Select content](#)  
200 OK

Response Headers [Select content](#)  
X-Mashery-Responder: prod-j-worker-atl-01.mashery.com  
Server: nginx/1.4.1  
Date: Sat, 22 Nov 2014 22:24:00 GMT  
Content-Type: application/xml; charset=UTF-8  
Content-Length: 4327  
X-Powered-By: PHP/5.5.10  
Last-Modified: Sat, 22 Nov 2014 22:24:00 GMT  
Etag: "Sat, 22 Nov 2014 22:24:00 GMT"  
Cache-Control: max-age=14400  
Accept-Ranges: bytes  
X-Varnish: 862437280  
Age: 0  
Via: 1.1 varnish  
X-Cache: MISS

Link to extract the data.

Now, to import the XML data to Excel, open a new Excel sheet. Navigate to **File | Open | Other Web Locations** and paste the API link. MS Excel will identify the format as XML output and import our data directly in the sheet. The final output is available under the filename nytimes.xls.



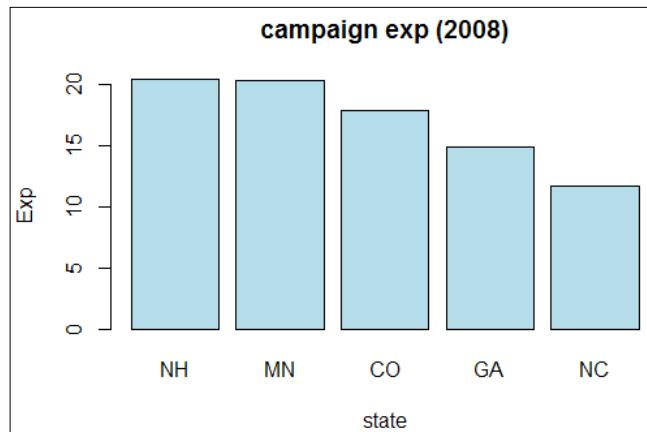
We can now use either Excel or R to generate the charts. To generate a chart in R, we will edit and format the data, save it as a CSV file, and import it in R.

## See also

- ▶ A list of different APIs available on Google can be found at <https://console.developers.google.com/project/myfirstprojectin/apiui/api?authuser=0>. Google requires all users to register. The link also provides documents and examples on all of the APIs.
- ▶ The New York Times website at <http://developer.nytimes.com/>.
- ▶ Facebook has its own API under the name Graph API Explorer. It is accessible at <https://developers.facebook.com/?ref=pf>.
- ▶ The Twitter website at <https://dev.twitter.com/>.

## Constructing a bar plot using XML in R

In the prior recipe, we learned to construct an API, studied various elements of an API, and imported data using the New York Times API. In this section, we will go a step further and import the data directly in R, parse the XML data, construct and structure our data, and plot a bar plot. For this recipe, we will use the New York Times website, but instead of real estate data, we will use the campaign finance data. The bar plot is shown as follows:



## Getting ready

In order to extract data from the New York Times and generate a bar plot, we would use the XML package and the generic `plot` function in R.

## How to do it...

To extract the data from New York Times website, we will register our key with the Campaign Finance API. Please refer to the *Basic introduction to API and XML* recipe to understand how to register.

Once we register our API, we can now install and load the XML package in R using the `install.packages()` and `library()` functions:

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

We copy and paste the link we generated using the New York Times API console. Further, we will use the `readLines()` function in R to extract the XML data from the link. Please note that the data pertains only to 2008:

```
url = "http://api.nytimes.com/svc/elections/us/v3/finances/2008
/independent_expenditures/race_totals/senate.xml?api-key=<your
key>"
data = readLines(url)
```

Please note that you will have to paste your API key in place of `<your key>` in the preceding code.

Next, we parse the XML data using the `xmlParse()` function. You should refer to the links under the See also section of this recipe for a deeper understanding of XML, HTML, and JSON formats and their parsing methods:

```
data = xmlParse(data)
```

We specifically require two sets of data, namely state name and expenditure amount. Hence, we will use the `xpathSApply()` and `xmlValue()` functions:

```
state=xpathSApply(data, "//state", xmlValue)
amount=xpathSApply(data, "//amount", xmlValue)
```

The data extracted is in the form of a character vector. To generate a bar plot, we require that the data is either in a matrix or a vector format, hence we use `as.numeric()` and `as.vector()` to coerce the data into the required formats:

```
amt = as.numeric(amount[1:5])
amt = amt/100000
names= as.vector(state[1:5])
```

Finally, we can construct a plot using the `barplot()` function:

```
barplot(amt,beside = TRUE, col = "lightblue", names.arg = names,
main = "campaign exp (2008)", xlab = "state", ylab = "Exp")
```

## How it works...

When we copy and paste the URL constructed using the API console on the New York Times developer website, we see a XML output. The output contains the state variable and the amount variable.

The `readLines()` function uses the `url` argument to retrieve the data in R. The data needs to be parsed and the XML package provides us with the `xmlParse()` function to parse our data. The data in XML output is stored between the `<state>` and `</state>` tags. To learn more about the child nodes, parent nodes, and XML outputs, please refer to the link in the See also section.

The `xmlValue()` function is used to extract the content of an XML leaf; in our case we do not require the tags but the value itself, for example, NH, CO, and so on. We have used the `xmlValue()` function as an argument within the `xpathSApply()` function. The `xpathSApply()` function is a simplified version of the generic `sapply()` function. We need to extract each value from the XML tree and hence, we need to loop over the entire tree structure to extract all the elements. To understand the `sapply()` function, please refer to the *Apply, Lapply, Sapply, and Tapply* recipe in *Chapter, A Simple Guide to R*.

We also observe `//state` and `//amount` within the `xpathSApply()` function. This is the XPath language, which is used to navigate through the XML tree and extract specific elements. In our case, we are simply looping over the `state` and `amount` XML tags and extracting the value. To learn more about the XPath language and its notations, please refer to the See also section of this recipe.

We have now completely extracted the two variables, namely, the state and expenditure amounts. The data is as a character vector, hence we convert it to a numeric using the `as.numeric()` function. Note that for the purpose of this recipe, we will only utilize the data for the first five states, hence `amount[1:5]` is passed as an argument. Please refer to *Chapter, A Simple Guide to R* to understand the matrix/vector manipulation.

We can now pass the sample data as a first argument to the `barplot()` function. The second argument is `beside = TRUE`, which forces R to plot the bars side by side instead of stacking them. All other arguments in the `barplot()` function are self-explanatory.

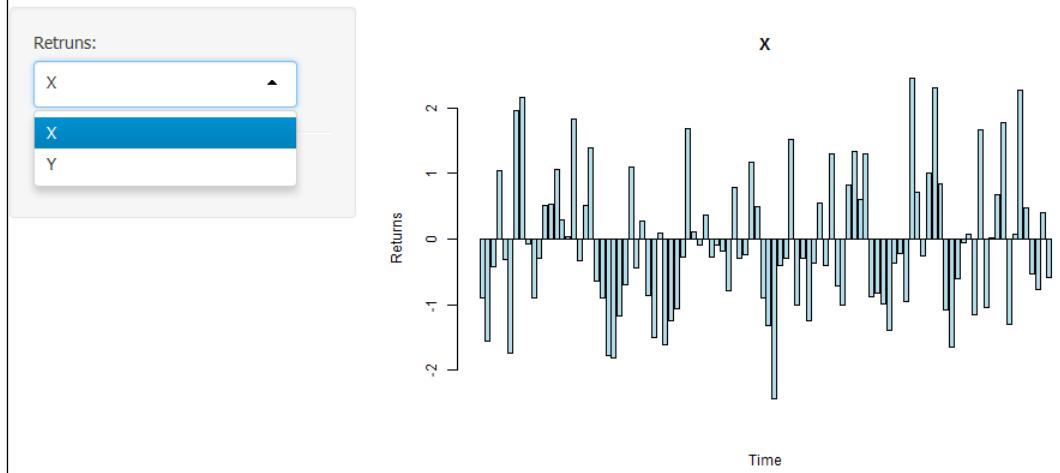
## See also

- ▶ Great presentations on XML and HTML parsing can be found at <http://gastonsanchez.com/teaching/>.
- ▶ Slides with different ways of using R to retrieve data using the XML and JSON formats can be found at <http://quantifyingmemory.blogspot.co.uk/2014/02/web-scraping-basics.html>.
- ▶ The course titled *Getting and Cleaning Data* covers the topic on extracting data from the Web, the API, and databases. The course is regularly offered at <https://www.coursera.org/course/getdata>.

# Creating a very simple shiny app in R

The `shiny` package allows us to create applications in R. The advantage of using the `shiny` package is the flexibility it provides to users. The `googleVis` package provides us with the ability to interact with plot, but the `shiny` package provides its users with the ability to perform very specific tasks, such as uploading/downloading data interactively or creating GUI to calculate analytics.

## Stocks



## Getting ready

We require the `shiny` package to create an application in R. We also need to create a folder in our current directory called `shiny` (or any other name) and further create two text documents named as `server.r` and `ui.r` in the `shiny` folder.

## How to do it...

The `server.r` file contains the code that gets executed in R and `ui.R` contains code that creates the user interface. To launch the application, you can install and load the `shiny` package in R and further execute the `runApp ("shiny")` function. The `runApp()` function will only execute if the `shiny` folder is stored in the current directory referenced in R.

Let's first study the `ui.R` file in detail. We first launch the `shiny` and `ISLR` packages in R using the `library(shiny)` and `library(ISLR)` functions.

The `shinyUI()` function creates the user interface and the `titlePanel(shiny)` function generates the title panel for our application. We would like to construct an application where a user can select between returns for two different stocks and the plot for the same appears on the right side. Hence, we generate a sidebar panel, which allows users to select from a drop-down menu. The `selectInput()` function creates the drop-down box.

The `mainpanel()` function generates the main panel with the output plot and the `plotOutput()` function renders the plot. In our example, we will name our output `plot stock`.

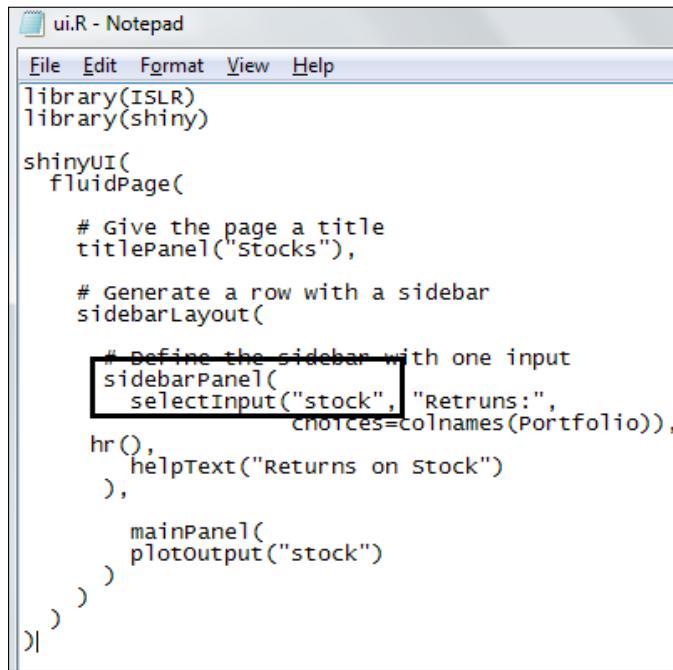
The server-side logic is defined in the `server.R` file under `shinyServer(function (input, output))`. The plot defined under the `renderPlot()` function gets generated in the main panel of the user interface.

## How it works...

The first argument under the `selectInput()` function is the input ID and the second argument is the label for the drop-down menu. The third argument is the `choices` argument, which gives the users of the application with a choice. The `portfolio` data under the `ISLR` package contains returns for two different stocks, X and Y. Hence, the third argument under `selectInput()` should be `colnames` of the portfolio data. In order to learn more about the `selectInput()` function, you should refer to the `shiny` package manual at <http://cran.r-project.org/web/packages/shiny/shiny.pdf>.

Note that under the `ui.R` file, we have named the plot as `stock` under the `plotOutput()` function. This is also referenced in `server.R` under the `shinyServer()` function as `output$stock`. We instruct `shiny` to generate a plot under the `renderPlot()` function and display it in the main panel of the user interface. In order to understand the various functions of `shiny`, it is best to open both, the `shiny.r` and `ui.r` files and read the code simultaneously.

We would like to create a bar plot, hence we generate the same under the `renderPlot()` function via the `barplot()` function. The first argument in the `barplot()` function is the `input` argument; the second third and fourth arguments are self-explanatory. The first argument consists of `[, input$stock]`, which is the input ID, as depicted in the following screenshot:



The screenshot shows a Notepad window titled "ui.R - Notepad". The code is as follows:

```
ui.R - Notepad
File Edit Format View Help
library(ISLR)
library(shiny)

shinyUI(
 fluidPage(
 # Give the page a title
 titlePanel("Stocks"),
 # Generate a row with a sidebar
 sidebarLayout(
 # Define the sidebar with one input
 sidebarPanel(
 selectInput("stock", "Returns:",
 choices=colnames(Portfolio)),
 hr(),
 helpText("Returns on Stock")
),
 mainPanel(
 plotOutput("stock")
)
)
)
)
```

A red rectangular box highlights the line `selectInput("stock", "Returns:",`.

## See also

- ▶ The shiny website consists of some great examples along with the code under the gallery section at <http://shiny.rstudio.com/gallery/>
- ▶ The mages blog that explains methods to integrate the `googleVis` package plots with the `shiny` package is available at <http://www.magesblog.com/2013/02/first-steps-of-using-googlevis-on-shiny.html>

# Module 3

## **Machine Learning with R Cookbook**

*Explore over 110 recipes to analyze data and build predictive models with the simple and easy-to-use R code*

# 1

# Data Exploration with RMS Titanic

In this chapter, we will cover the following recipes:

- ▶ Reading a Titanic dataset from a CSV file
- ▶ Converting types on character variables
- ▶ Detecting missing values
- ▶ Imputing missing values
- ▶ Exploring and visualizing data
- ▶ Predicting passenger survival with a decision tree
- ▶ Validating the power of prediction with a confusion matrix
- ▶ Assessing performance with the ROC curve

## Introduction

Data exploration helps a data consumer to focus on searching for information, with a view to forming a true analysis from the gathered information. Furthermore, with the completion of the steps of data munging, analysis, modeling, and evaluation, users can generate insights and valuable points from their focused data.

In a real data exploration project, there are six steps involved in the exploration process. They are as follows:

1. Asking the right questions.
2. Data collection.
3. Data munging.

4. Basic exploratory data analysis.
5. Advanced exploratory data analysis.
6. Model assessment.

A more detailed explanation of these six steps is provided here:

1. **Asking the right questions:** When the user presents their question, for example "What are my expected findings after the exploration is finished?", or "What kind of information can I extract through the exploration?," different results will be given. Therefore, asking the right question is essential in the first place, for the question itself determines the objective and target of the exploration.
2. **Data collection:** Once the goal of exploration is determined, the user can start collecting or extracting relevant data from the data source, with regard to the exploration target. Mostly, data collected from disparate systems appears unorganized and diverse in format. Clearly, the original data may be from different sources, such as files, databases, or the Internet. To retrieve data from these sources requires the assistance of the file IO function, JDBC/ODBC, web crawler, and so on. This extracted data is called **raw data**, which is because it has not been subjected to processing, or been through any other manipulation. Most raw data is not easily consumed by the majority of analysis tools or visualization programs.
3. **Data munging:** The next phase is data munging (or wrangling), a step to help map raw data into a more convenient format for consumption. During this phase, there are many processes, such as data parsing, sorting, merging, filtering, missing value completion, and other processes to transform and organize the data, and enable it to fit into a consume structure. Later, the mapped data can be further utilized for data aggregation, analysis, or visualization.
4. **Basic exploratory data analysis:** After the data munging phase, users can conduct further analysis toward data processing. The most basic analysis is to perform exploratory data analysis. Exploratory data analysis involves analyzing a dataset by summarizing its characteristics. Performing basic statistical, aggregation, and visual methods are also crucial tasks to help the user understand data characteristics, which are beneficial for the user to capture the majority, trends, and outliers easily through plots.
5. **Advanced exploratory data analysis:** Until now, the descriptive statistic gives a general description of data features. However, one would like to generate an inference rule for the user to predict data features based on input parameters. Therefore, the application of machine learning enables the user to generate an inferential model, where the user can input a training dataset to generate a predictive model. After this, the prediction model can be utilized to predict the output value or label based on given parameters.

6. **Model assessment:** Finally, to assess whether the generating model performs the best in the data estimation of a given problem, one must perform a model selection. The selection method here involves many steps, including data preprocessing, tuning parameters, and even switching the machine learning algorithm. However, one thing that is important to keep in mind is that the simplest model frequently achieves the best results in predictive or exploratory power; whereas complex models often result in over fitting.

For the following example, we would like to perform a sample data exploration based on the dataset of passengers surviving the Titanic shipwreck. The steps we demonstrate here follow how to collect data from the online source, Kaggle; clean data through data munging; perform basic exploratory data analysis to discover important attributes that might give a prediction of the survival rate; perform advanced exploratory data analysis using the classification algorithm to predict the survival rate of the given data; and finally, perform model assessment to generate a prediction model.

## Reading a Titanic dataset from a CSV file

To start the exploration, we need to retrieve a dataset from Kaggle (<https://www.kaggle.com/>). We had look at some of the samples in *Chapter, Practical Machine Learning with R*. Here, we introduce methods to deal with real-world problems.

### Getting ready

To retrieve data from Kaggle, you need to first sign up for a Kaggle account (<https://www.kaggle.com/account/register>). Then, log in to the account for further exploration:

The screenshot shows the main landing page of Kaggle. At the top, there's a dark navigation bar with the 'kaggle' logo, 'Customer Solutions', 'Competitions', 'Community', 'Sign Up', and 'Login' buttons. Below the header, a large blue banner features the text 'We're the global leader in solving business challenges through predictive analytics.' on the left, and two stylized white network graphs on the right. Underneath the banner, there are social media links for Facebook, LinkedIn, and Twitter, along with logos for MasterCard, MERCK, and NASA. A call-to-action button says 'Compete as a data scientist for fortune, fame and fun!'. At the bottom, there's a section titled 'FOCUS INDUSTRY' with a sub-section about the Energy industry using machine learning and big data for high-stakes decisions, with a 'Find out more' button. The footer contains the Kaggle logo and the URL 'Kaggle.com'.

## How to do it...

Perform the following steps to read the Titanic dataset from the CSV file:

1. Go to <http://www.kaggle.com/c/titanic-gettingStarted/data> to retrieve the data list.
2. You can see a list of data files for download, as shown in the following table:

| Filename         | Available formats |
|------------------|-------------------|
| train            | .csv (59.76 kb)   |
| genderclassmodel | .py (4.68 kb)     |
| myfirstforest    | .csv (3.18 kb)    |
| myfirstforest    | .py (3.83 kb)     |
| gendermodel      | .csv (3.18 kb)    |
| genderclassmodel | .csv (3.18 kb)    |
| test             | .csv (27.96 kb)   |
| gendermodel      | .py (3.58 kb)     |

3. Download the training data (<https://www.kaggle.com/c/titanic-gettingStarted/download/train.csv>) to a local disk.
4. Then, make sure the downloaded file is placed under the current directory. You can use the `getwd` function to check the current working directory. If the downloaded file is not located in the working directory, move the file to the current working directory. Or, you can use `setwd()` to set the working directory to where the downloaded files are located:

```
> getwd()
[1] "C:/Users/guest"
```

5. Next, one can use `read.csv` to load data into the data frame. Here, one can use the `read.csv` function to read `train.csv` to frame the data with the variable names set as `train.data`. However, in order to treat the blank string as NA, one can specify that `na.strings` equals either "NA" or an empty string:

```
> train.data = read.csv("train.csv", na.strings=c("NA", ""))
```

6. Then, check the loaded data with the `str` function:

```
> str(train.data)
'data.frame': 891 obs. of 12 variables:
 $ PassengerId: int 1 2 3 4 5 6 7 8 9 10 ...
 $ Survived : int 0 1 1 1 0 0 0 0 1 1 ...
 $ Pclass : int 3 1 3 1 3 3 1 3 3 2 ...
```

```
$ Name : Factor w/ 891 levels "Abbing, Mr. Anthony",...: 109
191 358 277 16 559 520 629 417 581 ...

$ Sex : Factor w/ 2 levels "female","male": 2 1 1 1 2 2 2
2 1 1 ...

$ Age : num 22 38 26 35 35 NA 54 2 27 14 ...

$ SibSp : int 1 1 0 1 0 0 0 3 0 1 ...

$ Parch : int 0 0 0 0 0 0 0 1 2 0 ...

$ Ticket : Factor w/ 681 levels "110152","110413",...: 524 597
670 50 473 276 86 396 345 133 ...

$ Fare : num 7.25 71.28 7.92 53.1 8.05 ...

$ Cabin : Factor w/ 148 levels "", "A10", "A14",...: 1 83 1 57
1 1 131 1 1 1 ...

$ Embarked : Factor w/ 4 levels "", "C", "Q", "S": 4 2 4 4 4 3 4 4
4 2 ...
```

## How it works...

To begin the data exploration, we first downloaded the Titanic dataset from Kaggle, a website containing many data competitions and datasets. To load the data into the data frame, this recipe demonstrates how to apply the `read.csv` function to load the dataset with the `na.strings` argument, for the purpose of converting blank strings and "NA" to NA values. To see the structure of the dataset, we used the `str` function to compactly display `train.data`; you can find the dataset contains demographic information and survival labels of the passengers. The data collected here is good enough for beginners to practice how to process and analyze data.

## There's more...

On Kaggle, much of the data on science is related to competitions, which mostly refer to designing a machine learning method to solve real-world problems.

Most competitions on Kaggle are held by either academia or corporate bodies, such as Amazon or Facebook. In fact, they create these contests and provide rewards, such as bonuses, or job prospects (see <https://www.kaggle.com/competitions>). Thus, there are many data scientists who are attracted to registering for a Kaggle account to participate in competitions. A beginner in a pilot exploration can participate in one of these competitions, which will help them gain experience by solving real-world problems with their machine learning skills.

To create a more challenging learning environment as a competitor, a participant needs to submit their output answer and will receive the assessment score, so that each one can assess their own rank on the leader board.

# Converting types on character variables

In R, since nominal, ordinal, interval, and ratio variable are treated differently in statistical modeling, we have to convert a nominal variable from a character into a factor.

## Getting ready

You need to have the previous recipe completed by loading the Titanic training data into the R session, with the `read.csv` function and assigning an argument of `na.strings` equal to `NA` and the blank string (""). Then, assign the loaded data from `train.csv` into the `train.data` variables.

## How to do it...

Perform the following steps to convert the types on character variables:

1. Use the `str` function to print the overview of the Titanic data:

```
> str(train.data)

'data.frame': 891 obs. of 12 variables:
 $ PassengerId: int 1 2 3 4 5 6 7 8 9 10 ...
 $ Survived : int 0 1 1 1 0 0 0 0 1 1 ...
 $ Pclass : int 3 1 3 1 3 3 1 3 3 2 ...
 $ Name : Factor w/ 891 levels "Abbing, Mr. Anthony",...: 109
191 358 277 16 559 520 629 417 581 ...
$ Sex : Factor w/ 2 levels "female","male": 2 1 1 1 2 2 2
2 1 1 ...
$ Age : num 22 38 26 35 35 NA 54 2 27 14 ...
$ SibSp : int 1 1 0 1 0 0 0 3 0 1 ...
$ Parch : int 0 0 0 0 0 0 0 1 2 0 ...
$ Ticket : Factor w/ 681 levels "110152","110413",...: 524 597
670 50 473 276 86 396 345 133 ...
$ Fare : num 7.25 71.28 7.92 53.1 8.05 ...
$ Cabin : Factor w/ 147 levels "A10","A14","A16",...: NA 82
NA 56 NA NA 130 NA NA NA ...
$ Embarked : Factor w/ 3 levels "C","Q","S": 3 1 3 3 3 2 3 3 3
1 ...
```

2. To transform the variable from the `int` numeric type to the `factor` categorical type, you can cast `factor`:

```
> train.data$Survived = factor(train.data$Survived)
> train.data$Pclass = factor(train.data$Pclass)
```

3. Print out the variable with the `str` function and again, you can see that `Pclass` and `Survived` are now transformed into the factor as follows:

```
> str(train.data)
'data.frame': 891 obs. of 12 variables:
 $ PassengerId: int 1 2 3 4 5 6 7 8 9 10 ...
 $ Survived : Factor w/ 2 levels "0","1": 1 2 2 2 1 1 1 1 1 2 2
 ...
 $ Pclass : Factor w/ 3 levels "1","2","3": 3 1 3 1 3 3 1 3 3
 2 ...
 $ Name : Factor w/ 891 levels "Abbing, Mr. Anthony",...: 109
 191 358 277 16 559 520 629 417 581 ...
 $ Sex : Factor w/ 2 levels "female","male": 2 1 1 1 2 2 2
 2 1 1 ...
 $ Age : num 22 38 26 35 35 NA 54 2 27 14 ...
 $ SibSp : int 1 1 0 1 0 0 0 3 0 1 ...
 $ Parch : int 0 0 0 0 0 0 0 1 2 0 ...
 $ Ticket : Factor w/ 681 levels "110152","110413",...: 524 597
 670 50 473 276 86 396 345 133 ...
 $ Fare : num 7.25 71.28 7.92 53.1 8.05 ...
 $ Cabin : Factor w/ 147 levels "A10","A14","A16",...: NA 82
 NA 56 NA NA 130 NA NA NA ...
 $ Embarked : Factor w/ 3 levels "C","Q","S": 3 1 3 3 3 2 3 3 3
 1 ...
```

## How it works...

Talking about statistics, there are four measurements: nominal, ordinal, interval, and ratio. Nominal variables are used to label variables, such as gender and name; ordinal variables, and are measures of non-numeric concepts, such as satisfaction and happiness. Interval variables shows numeric scales, which tell us not only the order but can also show the differences between the values, such as temperatures in Celsius. A ratio variable shows the ratio of a magnitude of a continuous quantity to a unit magnitude. Ratio variables provide order, differences between the values, and a true zero value, such as weight and height. In R, different measurements are calculated differently, so you should perform a type conversion before applying descriptive or inferential analytics toward the dataset.

In this recipe, we first display the structure of the train data using the `str` function. From the structure of data, you can find the attribute name, data type, and the first few values contained in each attribute. From the `Survived` and `Pclass` attribute, you can see the data type as `int`. As the variable description listed in Chart 1 (Preface), you can see that `Survived` (`0 = No; 1 = Yes`) and `Pclass` (`1 = 1st; 2 = 2nd; 3 = 3rd`) are categorical variables. As a result, we transform the data from a character to a factor type via the `factor` function.

## There's more...

Besides `factor`, there are more type conversion functions. For numeric types, there are `is.numeric()` and `as.numeric()`; for character, there are: `is.character()` and `as.character()`. For vector, there are: `is.vector()` and `as.vector()`; for matrix, there are `is.matrix()` and `as.matrix()`. Finally, for data frame, there are: `is.data.frame()` and `as.data.frame()`.

## Detecting missing values

Missing values reduce the representativeness of the sample, and furthermore, might distort inferences about the population. This recipe will focus on detecting missing values within the Titanic dataset.

### Getting ready

You need to have completed the previous recipes by the `Pclass` attribute and `Survived` to a factor type.

In R, a missing value is noted with the symbol **NA (not available)**, and an impossible value is **NaN (not a number)**.

### How to do it...

Perform the following steps to detect the missing value:

1. The `is.na` function is used to denote which index of the attribute contains the NA value. Here, we apply it to the `Age` attribute first:  

```
> is.na(train.data$Age)
```
2. The `is.na` function indicates the missing value of the `Age` attribute. To get a general number of how many missing values there are, you can perform a `sum` to calculate this:  

```
> sum(is.na(train.data$Age) == TRUE)
[1] 177
```

3. To calculate the percentage of missing values, one method adopted is to count the number of missing values against nonmissing values:

```
> sum(is.na(train.data$Age) == TRUE) / length(train.data$Age)
[1] 0.1986532
```

4. To get a percentage of the missing value of the attributes, you can use `sapply` to calculate the percentage of all the attributes:

```
> sapply(train.data, function(df) {
+ sum(is.na(df)==TRUE) / length(df) ;
+ })

PassengerId Survived Pclass Name Sex
Age
0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
0.198653199

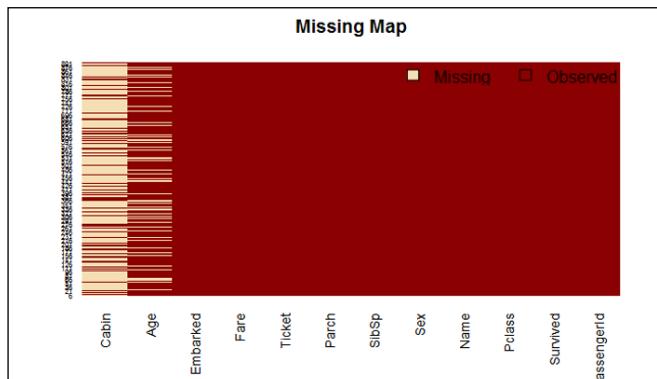
 SibSp Parch Ticket Fare Cabin
Embarked
0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.771043771
0.002244669
```

5. Besides simply viewing the percentage of missing data, one may also use the `Amelia` package to visualize the missing values. Here, we use `install.packages` and `require` to install `Amelia` and load the package. However, before the installation and loading of the `Amelia` package, you are required to install `Rcpp`, beforehand:

```
> install.packages("Amelia")
> require(Amelia)
```

6. Then, use the `missmap` function to plot the missing value map:

```
> missmap(train.data, main="Missing Map")
```



Missing map of the Titanic dataset

## How it works...

In R, a missing value is often noted with the "NA" symbol, which stands for not available. Most functions (such as `mean` or `sum`) may output NA while encountering an NA value in the dataset. Though you can assign an argument such as `na.rm` to remove the effect of NA, it is better to impute or remove the missing data in the dataset to prevent propagating the effect of the missing value. To find out the missing value in the Titanic dataset, we first sum up all the NA values and divide them by the number of values within each attribute. Then, we apply the calculation to all the attributes with `sapply`.

In addition to this, to display the calculation results using a table, you can utilize the `Amelia` package to plot the missing value map of every attribute on one chart. The visualization of missing values enables users to get a better understanding of the missing percentage within each dataset. From the preceding screenshot, you may have observed that the missing value is beige colored, and its observed value is dark red. The x-axis shows different attribute names, and the y-axis shows the recorded index. Clearly, most of the cabin shows missing data, and it also shows that about 19.87 percent of the data is missing when counting the `Age` attribute, and two values are missing in the `Embarked` attribute.

## There's more...

To handle the missing values, we introduced `Amelia` to visualize them. Apart from typing console commands, you can also use the interactive GUI of `Amelia` and `AmeliaView`, which allows users to load datasets, manage options, and run `Amelia` from a windowed environment.

To start running `AmeliaView`, simply type `AmeliaView()` in the R Console:

```
> AmeliaView()
```



AmeliaView

# Imputing missing values

After detecting the number of missing values within each attribute, we have to impute the missing values since they might have a significant effect on the conclusions that can be drawn from the data.

## Getting ready

This recipe will require `train.data` loaded in the R session and have the previous recipe completed by converting `Pclass` and `Survived` to a factor type.

## How to do it...

Perform the following steps to impute the missing values:

1. First, list the distribution of **Port of Embarkation**. Here, we add the `useNA = "always"` argument to show the number of NA values contained within `train.data`:

```
> table(train.data$Embarked, useNA = "always")
```

| C   | Q  | S   | <NA> |
|-----|----|-----|------|
| 168 | 77 | 644 | 2    |

2. Assign the two missing values to a more probable port (that is, the most counted port), which is Southampton in this case:

```
> train.data$Embarked[which(is.na(train.data$Embarked))] = 'S';
> table(train.data$Embarked, useNA = "always")
```

| C   | Q  | S   | <NA> |
|-----|----|-----|------|
| 168 | 77 | 646 | 0    |

3. In order to discover the types of titles contained in the names of `train.data`, we first tokenize `train.data>Name` by blank (a regular expression pattern as "`\s+`"), and then count the frequency of occurrence with the `table` function. After this, since the name title often ends with a period, we use the regular expression to grep the word containing the period. In the end, sort the table in decreasing order:

```
> train.data$name = as.character(train.data$name)
> table_words = table(unlist(strsplit(train.data$name, "\s+")))
> sort(table_words [grep('.\.',names(table_words))],
decreasing=TRUE)
```

| Mr.       | Miss. | Mrs.      | Master. |
|-----------|-------|-----------|---------|
| 517       | 182   | 125       | 40      |
| Dr.       | Rev.  | Col.      | Major.  |
| 7         | 6     | 2         | 2       |
| Mlle.     | Capt. | Countess. | Don.    |
| 2         | 1     | 1         | 1       |
| Jonkheer. | L.    | Lady .    | Mme.    |
| 1         | 1     | 1         | 1       |
| Ms.       | Sir.  |           |         |
| 1         | 1     |           |         |

4. To obtain which title contains missing values, you can use `str_match` provided by the `stringr` package to get a substring containing a period, then bind the column together with `cbind`. Finally, by using the `table` function to acquire the statistics of missing values, you can work on counting each title:

```
> library(stringr)
> tb = cbind(train.data$Age, str_match(train.data$name, "[a-zA-Z]+\\"."))
> table(tb[is.na(tb[,1]),2])
```

| Dr. | Master. | Miss. | Mr. | Mrs. |
|-----|---------|-------|-----|------|
| 1   | 4       | 36    | 119 | 17   |

5. For a title containing a missing value, one way to impute data is to assign the mean value for each title (not containing a missing value):

```
> mean.mr = mean(train.data$Age[grep1(" Mr\\\".", train.data$name) & !is.na(train.data$Age)])
> mean.mrs = mean(train.data$Age[grep1(" Mrs\\\".", train.data$name) & !is.na(train.data$Age)])
> mean.dr = mean(train.data$Age[grep1(" Dr\\\".", train.data$name) & !is.na(train.data$Age)])
> mean.miss = mean(train.data$Age[grep1(" Miss\\\".", train.data$name) & !is.na(train.data$Age)])
> mean.master = mean(train.data$Age[grep1(" Master\\\".", train.data$name) & !is.na(train.data$Age)])
```

6. Then, assign the missing value with the mean value of each title:

```
> train.data$Age[grep1(" Mr\\\".", train.data$name) & is.na(train.data$Age)] = mean.mr
> train.data$Age[grep1(" Mrs\\\".", train.data$name) & is.na(train.data$Age)] = mean.mrs
```

```
> train.data$Age[grep1(" Dr\\.", train.data$Name) & is.na(train.
data$Age)] = mean.dr
> train.data$Age[grep1(" Miss\\.", train.data$Name) & is.na(train.
data$Age)] = mean.miss
> train.data$Age[grep1(" Master\\.", train.data$Name) &
is.na(train.data$Age)] = mean.master
```

## How it works...

To impute the missing value of the `Embarked` attribute, we first produce the statistics of the embarked port with the `table` function. The `table` function counts two NA values in `train.data`. From the dataset description, we recognize C, Q, and S(C = Cherbourg, Q = Queenstown, S = Southampton). Since we do not have any knowledge about which category these two missing values are in, one possible way is to assign the missing value to the most likely port, which is Southampton.

As for another attribute, `Age`, though about 20 percent of the value is missing, users can still infer the missing value with the title of each passenger. To discover how many titles there are within the name of the dataset, we suggest the method of counting segmented words in the `Name` attribute, which helps to calculate the number of missing values of each given title. The resultant word table shows common titles such as `Mr`, `Mrs`, `Miss`, and `Master`. You may reference an English honorific entry from Wikipedia to get the description of each title.

Considering the missing data, we reassign the mean value of each title to the missing value with the corresponding title. However, for the `Cabin` attribute, there are too many missing values, and we cannot infer the value from any referencing attribute. Therefore, we find it does not work by trying to use this attribute for further analysis.

## There's more...

Here we list the honorific entry from Wikipedia for your reference. According to it ([http://en.wikipedia.org/wiki/English\\_honorific](http://en.wikipedia.org/wiki/English_honorific)):

- ▶ **Mr:** This is used for a man, regardless of his marital status
- ▶ **Master:** This is used for young men or boys, especially used in the UK
- ▶ **Miss:** It is usually used for unmarried women, though also used by married female entertainers
- ▶ **Mrs:** It is used for married women
- ▶ **Dr:** It is used for a person in the US who owns his first professional degree

# Exploring and visualizing data

After imputing the missing values, one should perform an exploratory analysis, which involves using a visualization plot and an aggregation method to summarize the data characteristics. The result helps the user gain a better understanding of the data in use. The following recipe will introduce how to use basic plotting techniques with a view to help the user with exploratory analysis.

## Getting ready

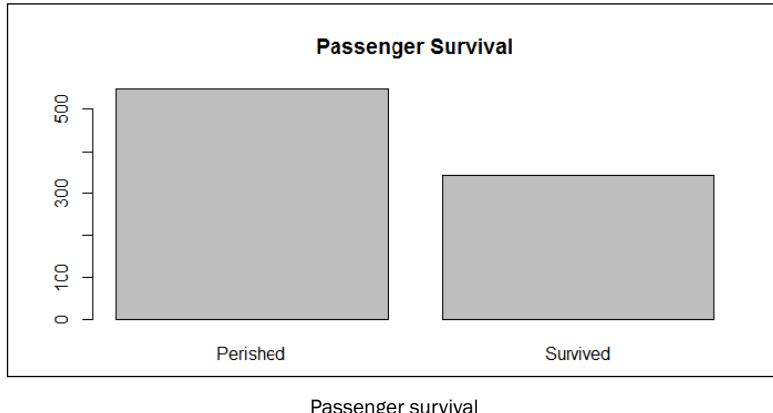
This recipe needs the previous recipe to be completed by imputing the missing value in the age and Embarked attribute.

## How to do it...

Perform the following steps to explore and visualize data:

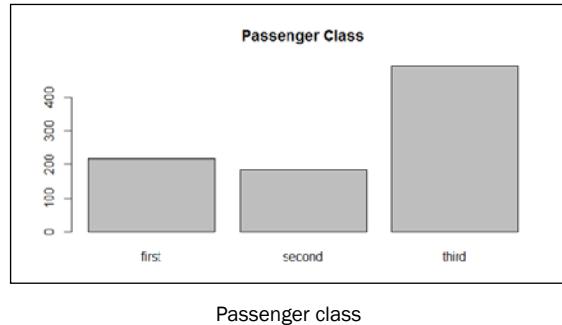
1. First, you can use a bar plot and histogram to generate descriptive statistics for each attribute, starting with passenger survival:

```
> barplot(table(train.data$Survived), main="Passenger Survival",
names= c("Perished", "Survived"))
```



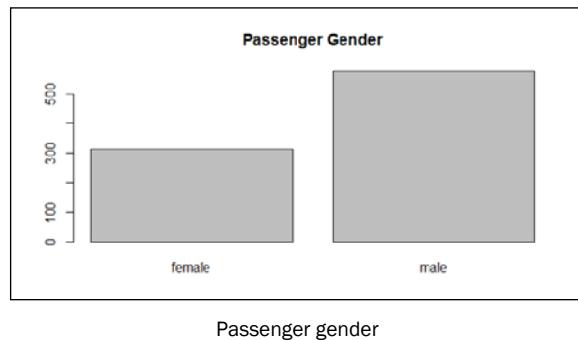
2. We can generate the bar plot of passenger class:

```
> barplot(table(train.data$Pclass), main="Passenger Class",
names= c("first", "second", "third"))
```



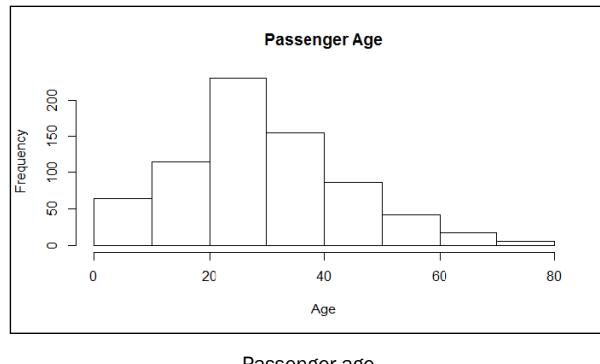
3. Next, we outline the gender data with the bar plot:

```
> barplot(table(train.data$Sex), main="Passenger Gender")
```



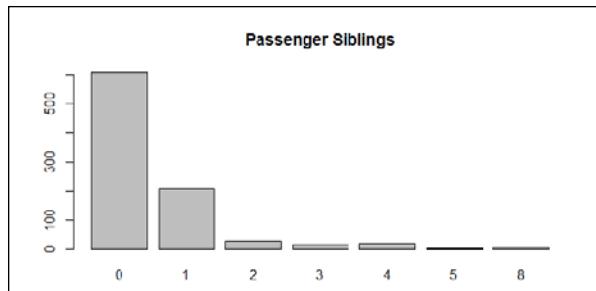
4. We then plot the histogram of the different ages with the `hist` function:

```
> hist(train.data$Age, main="Passenger Age", xlab = "Age")
```



5. We can plot the bar plot of sibling passengers to get the following:

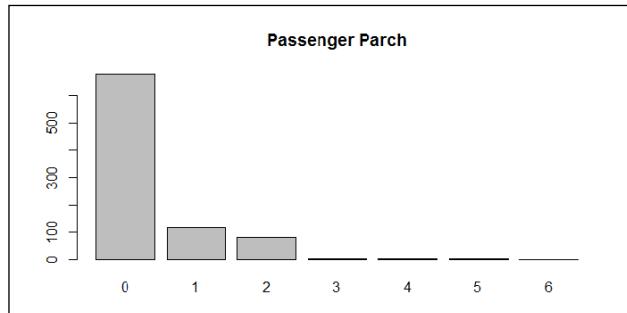
```
> barplot(table(train.data$SibSp), main="Passenger Siblings")
```



Passenger siblings

6. Next, we can get the distribution of the passenger parch:

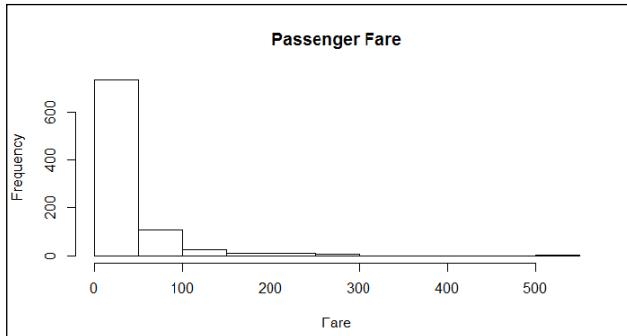
```
> barplot(table(train.data$Parch), main="Passenger Parch")
```



Passenger parch

7. Next, we plot the histogram of the passenger fares:

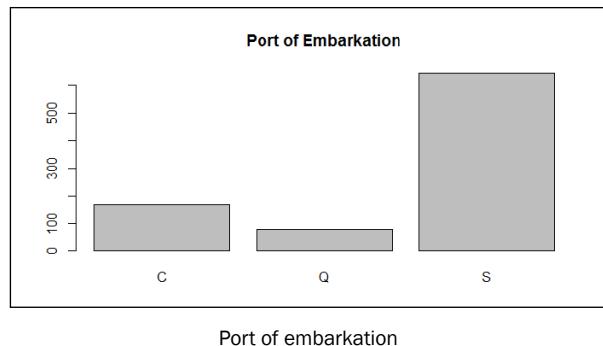
```
> hist(train.data$Fare, main="Passenger Fare", xlab = "Fare")
```



Passenger fares

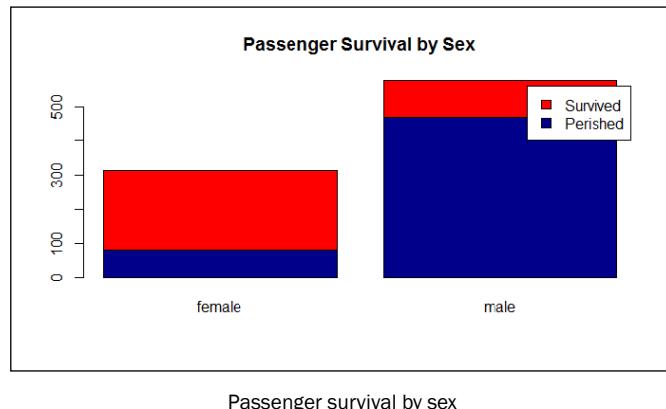
8. Finally, one can look at the port of embarkation:

```
> barplot(table(train.data$Embarked), main="Port of Embarkation")
```



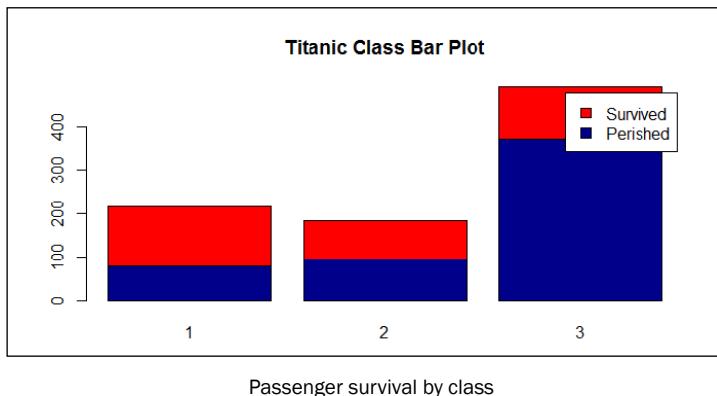
9. Use barplot to find out which gender is more likely to perish during shipwrecks:

```
> counts = table(train.data$Survived, train.data$Sex)
> barplot(counts, col=c("darkblue","red"), legend = c("Perished",
"Survived"), main = "Passenger Survival by Sex")
```



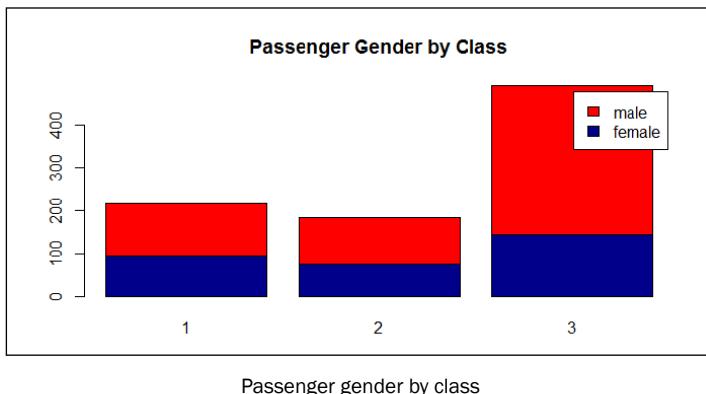
10. Next, we should examine whether the Pclass factor of each passenger may affect the survival rate:

```
> counts = table(train.data$Survived, train.data$Pclass)
> barplot(counts, col=c("darkblue","red"), legend =c("Perished",
 "Survived"), main= "Titanic Class Bar Plot")
```



11. Next, we examine the gender composition of each Pclass:

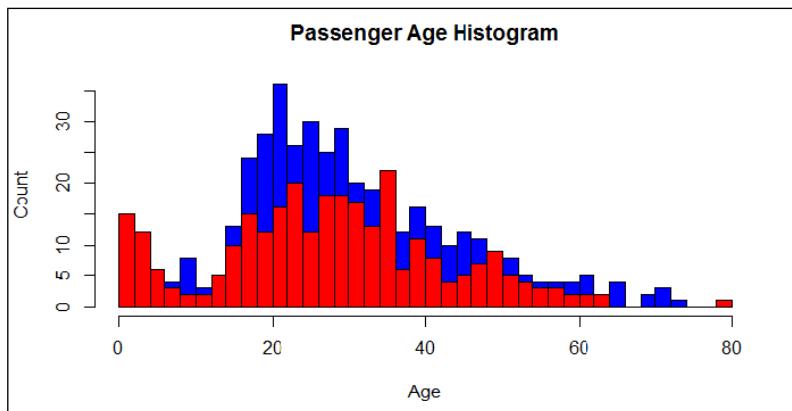
```
> counts = table(train.data$Sex, train.data$Pclass)
> barplot(counts, col=c("darkblue","red"), legend =
rownames(counts), main= "Passenger Gender by Class")
```



12. Furthermore, we examine the histogram of passenger ages:

```
> hist(train.data$Age[which(train.data$Survived == "0")], main="Passenger Age Histogram", xlab="Age", ylab="Count", col ="blue", breaks=seq(0,80,by=2))

> hist(train.data$Age[which(train.data$Survived == "1")], col ="red", add = T, breaks=seq(0,80,by=2))
```



Passenger age histogram

13. To examine more details about the relationship between the age and survival rate, one can use a boxplot:

```
> boxplot(train.data$Age ~ train.data$Survived,
+ main="Passenger Survival by Age",
+ xlab="Survived", ylab="Age")
```



Passenger survival by age

14. To categorize people with different ages into different groups, such as children (below 13), youths (13 to 19), adults (20 to 65), and senior citizens (above 65), execute the following commands:

```
> train.child = train.data$Survived[train.data$Age < 13]
> length(train.child[which(train.child == 1)]) / length(train.
child)
[1] 0.5797101

> train.youth = train.data$Survived[train.data$Age >= 15 & train.
data$Age < 25]
> length(train.youth[which(train.youth == 1)]) / length(train.
youth)
[1] 0.4285714

> train.adult = train.data$Survived[train.data$Age >= 20 & train.
data$Age < 65]
> length(train.adult[which(train.adult == 1)]) / length(train.
adult)
[1] 0.3659218

> train.senior = train.data$Survived[train.data$Age >= 65]
> length(train.senior[which(train.senior == 1)]) / length(train.
senior)
[1] 0.09090909
```

## How it works...

Before we predict the survival rate, one should first use the aggregation and visualization method to examine how each attribute affects the fate of the passengers. Therefore, we begin the examination by generating a bar plot and histogram of each attribute.

The plots from the screenshots in the preceding list give one an outline of each attribute of the Titanic dataset. As per the first screenshot, more passengers perished than survived during the shipwreck. Passengers in the third class made up the biggest number out of the three classes on board, which also reflects the truth that the third class was the most economical class on the Titanic (step 2). For the sex distribution, there were more male passengers than female (step 3). As for the age distribution, the screenshot in step 4 shows that most passengers were aged between 20 to 40. According to the screenshot in step 5, most passengers had one or fewer siblings. The screenshot in step 6 shows that most of the passengers have 0 to 2 parch.

In the screenshot in step 7, the fare histogram shows there were fare differences, which may be as a result of the different passenger classes on the Titanic. At last, the screenshot in step 8 shows that the boat made three stops to pick up passengers.

As we began the exploration from the `sex` attribute, and judging by the resulting bar plot, it clearly showed that female passengers had a higher rate of survival than males during the shipwreck (step 9). In addition to this, the Wikipedia entry for RMS Titanic ([http://en.wikipedia.org/wiki/RMS\\_Titanic](http://en.wikipedia.org/wiki/RMS_Titanic)) explains that '*A disproportionate number of men were left aboard because of a "women and children first" protocol followed by some of the officers loading the lifeboats*'. Therefore, it is reasonable that the number of female survivors outnumbered the male survivors. In other words, simply using `sex` can predict whether a person will survive with a high degree of accuracy.

Then, we examined whether the passenger class affected the survival rate (step 10). Here, from the definition of `Pclass`, the fares for each class were priced accordingly with the quality; high fares for first class, and low fares for third class. As the class of each passenger seemed to indicate their social and financial status, it is fair to assume that the wealthier passengers may have had more chances to survive.

Unfortunately, there was no correlation between the class and survival rate, so the result does not show the phenomenon we assumed. Nevertheless, after we examined `sex` in the composition of `pclass` (step 11), the results revealed that most third-class passengers were male; the assumption of wealthy people tending to survive more may not be that concrete.

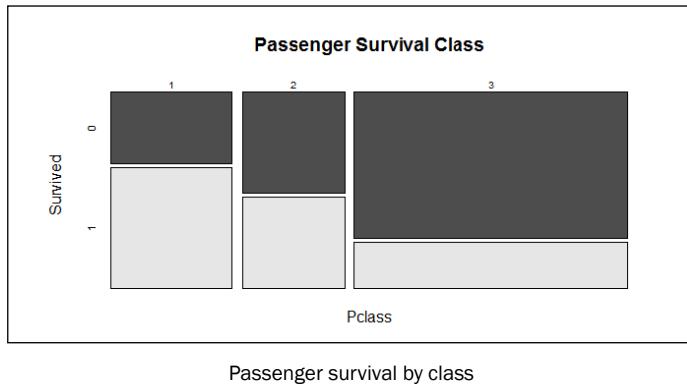
Next, we examined the relationship between the age and passenger fate through a histogram and box plot (step 12). The bar plot shows the age distribution with horizontal columns in which red columns represent the passengers that survived, while blue columns represent those who perished. It is hard to tell the differences in the survival rate from the ages of different groups. The bar plots that we created did not prove that passengers in different age groups were more likely to survive. On the other hand, the plots showed that most people on board were aged between 20 to 40, but does not show whether this group was more likely to survive compared to elderly or young children (step 13). Here, we introduced a box plot, which is a standardized plotting technique that displays the distribution of data with information, such as minimum, first quartile, median, third quartile, maximum, and outliers.

Later, we further examined whether age groups have any relation to passenger fates, by categorizing passenger ages into four groups. The statistics show the the children group (below 13) was more likely to survive than the youths (13 to 20), adults (20 to 65), and senior citizens (above 65). The results showed that people in the younger age groups were more likely to survive the shipwreck. However, we noted that this possibly resulted from the 'women and children first' protocol.

## There's more...

Apart from using bar plots, histograms, and boxplots to visualize data, one can also apply `mosaicplot` in the `vcd` package to examine the relationship between multiple categorical variables. For example, when we examine the relationship between the `Survived` and `Pclass` variables, the application is performed as follows:

```
> mosaicplot(train.data$Pclass ~ train.data$Survived,
+ main="Passenger Survival Class", color=TRUE,
+ xlab="Pclass", ylab="Survived")
```



Passenger survival by class

## See also

- ▶ For more information about the shipwreck, one can read the history of RMS Titanic (please refer to the entry *Sinking of the RMS Titanic* in Wikipedia [http://en.wikipedia.org/wiki/Sinking\\_of\\_the\\_RMS\\_Titanic](http://en.wikipedia.org/wiki/Sinking_of_the_RMS_Titanic)), as some of the protocol practiced at that time may have substantially affected the passenger survival rate.

## Predicting passenger survival with a decision tree

The exploratory analysis helps users gain insights into how single or multiple variables may affect the survival rate. However, it does not determine what combinations may generate a prediction model, so as to predict the passengers' survival. On the other hand, machine learning can generate a prediction model from a training dataset, so that the user can apply the model to predict the possible labels from the given attributes. In this recipe, we will introduce how to use a decision tree to predict passenger survival rates from the given variables.

# Getting ready

We will use the data, `train.data`, that we have already used in our previous recipes.

## How to do it...

Perform the following steps to predict the passenger survival with the decision tree:

1. First, we construct a data split `split.data` function with three input parameters: `data`, `p`, and `s`. The `data` parameter stands for the input dataset, the `p` parameter stands for the proportion of generated subset from the input dataset, and the `s` parameter stands for the random seed:

```
> split.data = function(data, p = 0.7, s = 666){
+ set.seed(s)
+ index = sample(1:dim(data)[1])
+ train = data[index[1:floor(dim(data)[1] * p)],]
+ test = data[index[((ceiling(dim(data)[1] * p)) +
1):dim(data)[1]],]
+ return(list(train = train, test = test))
+ }
```

2. Then, we split the data, with 70 percent assigned to the training dataset and the remaining 30 percent for the testing dataset:

```
> allset= split.data(train.data, p = 0.7)
> trainset = allset$train
> testset = allset$test
```

3. For the condition tree, one has to use the `ctree` function from the `party` package; therefore, we install and load the `party` package:

```
> install.packages('party')
> require('party')
```

4. We then use `Survived` as a label to generate the prediction model in use. After that, we assign the classification tree model into the `train.ctree` variable:

```
> train.ctree = ctree(Survived ~ Pclass + Sex + Age + SibSp + Fare
+ Parch + Embarked, data=trainset)
> train.ctree
```

Conditional inference tree with 7 terminal nodes

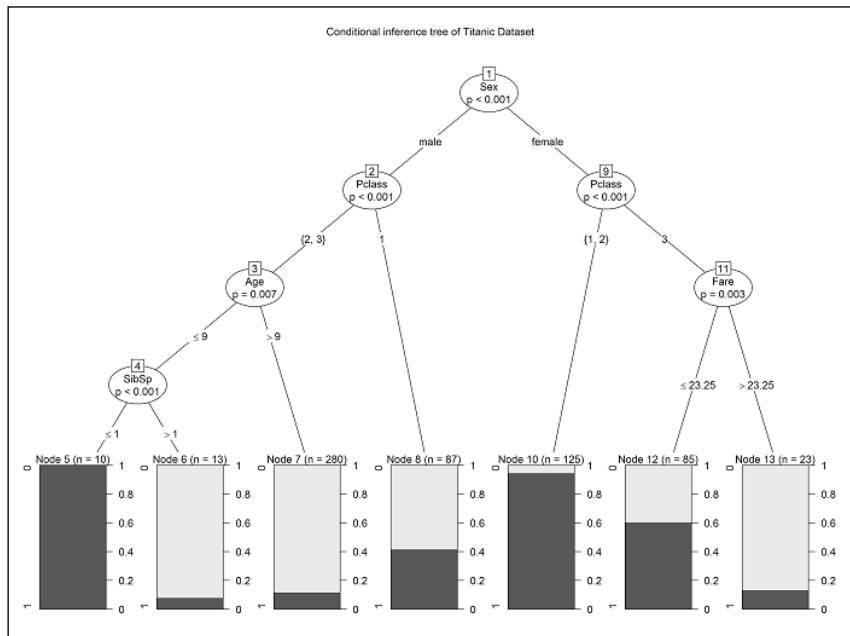
Response: Survived

Inputs: Pclass, Sex, Age, SibSp, Fare, Parch, Embarked  
Number of observations: 623

```
1) Sex == {male}; criterion = 1, statistic = 173.672
 2) Pclass == {2, 3}; criterion = 1, statistic = 30.951
 3) Age <= 9; criterion = 0.997, statistic = 12.173
 4) SibSp <= 1; criterion = 0.999, statistic = 15.432
 5)* weights = 10
 4) SibSp > 1
 6)* weights = 11
 3) Age > 9
 7)* weights = 282
 2) Pclass == {1}
 8)* weights = 87
1) Sex == {female}
 9) Pclass == {1, 2}; criterion = 1, statistic = 59.504
 10)* weights = 125
 9) Pclass == {3}
 11) Fare <= 23.25; criterion = 0.997, statistic = 12.456
 12)* weights = 85
 11) Fare > 23.25
 13)* weights = 23
```

5. We use a plot function to plot the tree:

```
> plot(train.ctree, main="Conditional inference tree of Titanic Dataset")
```



Conditional inference tree of the Titanic dataset

## How it works...

This recipe introduces how to use a conditional inference tree, `ctree`, to predict passenger survival. While the conditional inference tree is not the only method to solve the classification problem, it is an easy method to comprehend the decision path to predict passenger survival.

We first split the data into a training and testing set by using our implemented function, `split.data`. So, we can then use the training set to generate a prediction model and later employ the prediction model on the testing dataset in the recipe of the model assessment. Then, we install and load the `party` package, and use `ctree` to build a prediction model, with `Survived` as its label. Without considering any particular attribute, we put attributes such as `Pclass`, `Sex`, `Age`, `SibSp`, `Parch`, `Embarked`, and `Fare` as training attributes, except for `Cabin`, as most of this attribute's values are missing.

After constructing the prediction model, we can either print out the decision path and node in a text mode, or use a plot function to plot the decision tree. From the decision tree, the user can see what combination of variables may be helpful in predicting the survival rate. As per the preceding screenshot, users can find a combination of Pclass and Sex, which served as a good decision boundary (node 9) to predict the survival rates. This shows female passengers who were in first and second class mostly survived the shipwreck. Male passengers, those in second and third class and aged over nine, almost all perished during the shipwreck. From the tree, one may find that attributes such as Embarked and Parch are missing. This is because the conditional inference tree regards these attributes as less important during classification.

From the decision tree, the user can see what combination of variables may be helpful in predicting the survival rate. Furthermore, a conditional inference tree is helpful in selecting important attributes during the classification process; one can examine the built tree to see whether the selected attribute matches one's presumption.

## There's more...

This recipe covers issues relating to classification algorithms and conditional inference trees. Since we do not discuss the background knowledge of the adapted algorithm, it is better for the user to use the `help` function to view the documents related to `ctree` in the `party` package, if necessary.

There is a similar decision tree based package, named `rpart`. The difference between `party` and `rpart` is that `ctree` in the `party` package avoids the following variable selection bias of `rpart` and `ctree` in the `party` package, tending to select variables that have many possible splits or many missing values. Unlike the others, `ctree` uses a significance testing procedure in order to select variables, instead of selecting the variable that maximizes an information measure.

Besides `ctree`, one can also use `svm` to generate a prediction model. To load the `svm` function, load the `e1071` package first, and then use the `svm` build to generate this prediction:

```
> install.packages('e1071')
> require('e1071')

> svm.model = svm(Survived ~ Pclass + Sex + Age + SibSp + Fare + Parch +
Embarked, data = trainset, probability = TRUE)
```

Here, we use `svm` to show how easy it is that you can immediately use different machine learning algorithms on the same dataset when using R. For further information on how to use `svm`, please refer to *Chapter, Classification (II) – Neural Network, SVM*.

# Validating the power of prediction with a confusion matrix

After constructing the prediction model, it is important to validate how the model performs while predicting the labels. In the previous recipe, we built a model with `ctree` and pre-split the data into a training and testing set. For now, users will learn to validate how well `ctree` performs in a survival prediction via the use of a confusion matrix.

## Getting ready

Before assessing the prediction model, first be sure that the generated training set and testing dataset are within the R session.

## How to do it...

Perform the following steps to validate the prediction power:

1. We start using the constructed `train.ctree` model to predict the survival of the testing set:  

```
> ctree.predict = predict(train.ctree, testset)
```
2. First, we install the `caret` package, and then load it:  

```
> install.packages("caret")
> require(caret)
```
3. After loading `caret`, one can use a confusion matrix to generate the statistics of the output matrix:

```
> confusionMatrix(ctree.predict, testset$Survived)
Confusion Matrix and Statistics
```

### Reference

| Prediction | 0   | 1  |
|------------|-----|----|
| 0          | 160 | 25 |
| 1          | 16  | 66 |

```
Accuracy : 0.8464
95% CI : (0.7975, 0.8875)
No Information Rate : 0.6592
P-Value [Acc > NIR] : 4.645e-12
```

```
Kappa : 0.6499
McNemar's Test P-Value : 0.2115

Sensitivity : 0.9091
Specificity : 0.7253
Pos Pred Value : 0.8649
Neg Pred Value : 0.8049
Prevalence : 0.6592
Detection Rate : 0.5993
Detection Prevalence : 0.6929
Balanced Accuracy : 0.8172

'Positive' Class : 0
```

## How it works...

After building the prediction model in the previous recipe, it is important to measure the performance of the constructed model. The performance can be assessed by whether the prediction result matches the original label contained in the testing dataset. The assessment can be done by using the confusion matrix provided by the caret package to generate a confusion matrix, which is one method to measure the accuracy of predictions.

To generate a confusion matrix, a user needs to install and load the `caret` package first. The confusion matrix shows that purely using `cTree` can achieve accuracy of up to 84 percent. One may generate a better prediction model by tuning the attribute used, or by replacing the classification algorithm to SVM, `glm`, or random forest.

## There's more...

A caret package (*Classification and Regression Training*) helps make iterating and comparing different predictive models very convenient. The package also contains several functions, including:

- ▶ Data splits
- ▶ Common preprocessing: creating dummy variables, identifying zero- and near-zero-variance predictors, finding correlated predictors, centering, scaling, and so on
- ▶ Training (using cross-validation)
- ▶ Common visualizations (for example, `featurePlot`)

# Assessing performance with the ROC curve

Another measurement is by using the ROC curve (this requires the ROCR package), which plots a curve according to its true positive rate against its false positive rate. This recipe will introduce how we can use the ROC curve to measure the performance of the prediction model.

## Getting ready

Before applying the ROC curve to assess the prediction model, first be sure that the generated training set, testing dataset, and built prediction model, `ctree.predict`, are within the R session.

## How to do it...

Perform the following steps to assess prediction performance:

1. Prepare the probability matrix:

```
> train.ctree.pred = predict(train.ctree, testset)
> train.ctree.prob = 1- unlist(treeresponse(train.ctree,
testset), use.names=F) [seq(1,nrow(testset)*2,2)]
```

2. Install and load the ROCR package:

```
> install.packages("ROCR")
> require(ROCR)
```

3. Create an ROCR prediction object from probabilities:

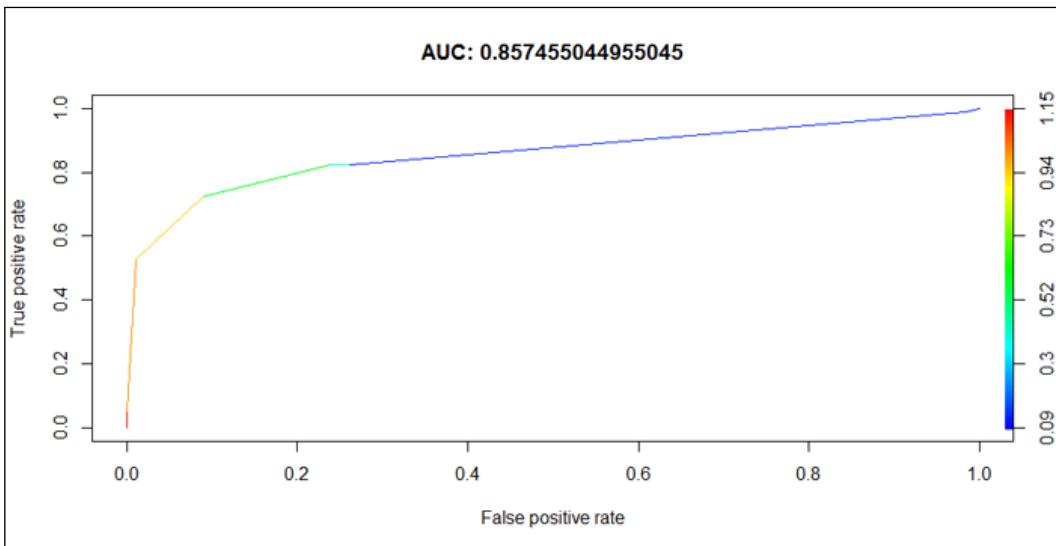
```
> train.ctree.prob.rocr = prediction(train.ctree.prob,
testset$Survived)
```

4. Prepare the ROCR performance object for the ROC curve (`tpr=true positive rate`, `fpr=false positive rate`) and the area under curve (AUC):

```
> train.ctree.perf = performance(train.ctree.prob.rocr,
"tpr", "fpr")
> train.ctree.auc.perf = performance(train.ctree.prob.rocr,
measure = "auc", x.measure = "cutoff")
```

5. Plot the ROC curve, with colorize as TRUE, and put AUC as the title:

```
> plot(train.ctree.perf, col=2,colorize=T, main=paste("AUC:",
train.ctree.auc.perf@y.values))
```



ROC of the prediction model

## How it works...

Here, we first create the prediction object from the probabilities matrix, and then prepare the ROCR performance object for the ROC curve (`tpr=true positive rate`, `fpr=false positive rate`) and the AUC. Lastly, we use the `plot` function to draw the ROC curve.

The result drawn in the preceding screenshot is interpreted in the following way: the larger under the curve (a perfect prediction will make AUC equal to 1), the better the prediction accuracy of the model. Our model returns a value of 0.857, which suggests that the simple conditional inference tree model is powerful enough to make survival predictions.

## See also

- ▶ To get more information on the ROCR, you can read the paper *Sing, T., Sander, O., Berenwinkel, N., and Lengauer, T. (2005). ROCR: visualizing classifier performance in R. Bioinformatics, 21(20), 3940-3941.*

# 2

# R and Statistics

In this chapter, we will cover the following topics:

- ▶ Understanding data sampling in R
- ▶ Operating a probability distribution in R
- ▶ Working with univariate descriptive statistics in R
- ▶ Performing correlations and multivariate analysis
- ▶ Operating linear regression and a multivariate analysis
- ▶ Conducting an exact binomial test
- ▶ Performing student's t-test
- ▶ Performing the Kolmogorov-Smirnov test
- ▶ Understanding the Wilcoxon Rank Sum and Signed Rank test
- ▶ Working with Pearson's Chi-squared test
- ▶ Conducting a one-way ANOVA
- ▶ Performing a two-way ANOVA

## Introduction

The R language, as the descendent of the statistics language, S, has become the preferred computing language in the field of statistics. Moreover, due to its status as an active contributor in the field, if a new statistical method is discovered, it is very likely that this method will first be implemented in the R language. As such, a large quantity of statistical methods can be fulfilled by applying the R language.

To apply statistical methods in R, the user can categorize the method of implementation into descriptive statistics and inferential statistics:

- ▶ **Descriptive statistics:** These are used to summarize the characteristics of the data. The user can use mean and standard deviation to describe numerical data, and use frequency and percentages to describe categorical data.
- ▶ **Inferential statistics:** Based on the pattern within a sample data, the user can infer the characteristics of the population. The methods related to inferential statistics are for hypothesis testing, data estimation, data correlation, and relationship modeling. Inference can be further extended to forecasting, prediction, and estimation of unobserved values either in or associated with the population being studied.

In the following recipe, we will discuss examples of data sampling, probability distribution, univariate descriptive statistics, correlations and multivariate analysis, linear regression and multivariate analysis, Exact Binomial Test, student's t-test, Kolmogorov-Smirnov test, Wilcoxon Rank Sum and Signed Rank test, Pearson's Chi-squared Test, One-way ANOVA, and Two-way ANOVA.

## Understanding data sampling in R

Sampling is a method to select a subset of data from a statistical population, which can use the characteristics of the population to estimate the whole population. The following recipe will demonstrate how to generate samples in R.

### Getting ready

Make sure that you have an R working environment for the following recipe.

### How to do it...

Perform the following steps to understand data sampling in R:

1. In order to generate random samples of a given population, the user can simply use the `sample` function:  

```
> sample(1:10)
```
2. To specify the number of items returned, the user can set the assigned value to the `size` argument:  

```
> sample(1:10, size = 5)
```
3. Moreover, the sample can also generate Bernoulli trials by specifying `replace = TRUE` (default is `FALSE`):  

```
> sample(c(0,1), 10, replace = TRUE)
```

## How it works...

As we saw in the preceding demonstration, the `sample` function can generate random samples from a specified population. The returned number from records can be designated by the user simply by specifying the argument of `size`. Assigning the `replace` argument to `TRUE`, you can generate Bernoulli trials (a population with 0 and 1 only).

## See also

- In R, the default package provides another sample method, `sample.int`, where both `n` and `size` must be supplied as integers:

```
> sample.int(20, 12)
```

# Operating a probability distribution in R

Probability distribution and statistics analysis are closely related to each other. For statistics analysis, analysts make predictions based on a certain population, which is mostly under a probability distribution. Therefore, if you find that the data selected for prediction does not follow the exact assumed probability distribution in experiment design, the upcoming results can be refuted. In other words, probability provides the justification for statistics. The following examples will demonstrate how to generate probability distribution in R.

## Getting ready

Since most distribution functions originate from the `stats` package, make sure the library `stats` are loaded.

## How to do it...

Perform the following steps:

- For a normal distribution, the user can use `dnorm`, which will return the height of a normal curve at 0:

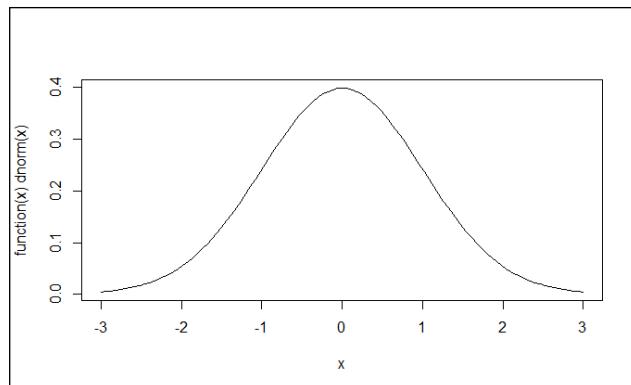
```
> dnorm(0)
[1] 0.3989423
```

- Then, the user can change the mean and the standard deviation in the argument:

```
> dnorm(0,mean=3, sd=5)
[1] 0.06664492
```

3. Next, plot the graph of a normal distribution with the `curve` function:

```
> curve(dnorm, -3, 3)
```



Standard normal distribution

4. In contrast to `dnorm`, which returns the height of a normal curve, the `pnorm` function can return the area under a given value:

```
> pnorm(1.5)
```

```
[1] 0.9331928
```

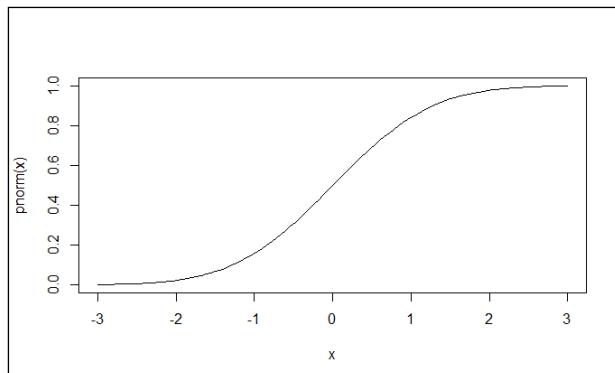
5. Alternatively, to get the area above a certain value, you can specify the option, `lower.tail`, to FALSE:

```
> pnorm(1.5, lower.tail=FALSE)
```

```
[1] 0.0668072
```

6. To plot the graph of `pnorm`, the user can employ a `curve` function:

```
> curve(pnorm(x), -3, 3)
```



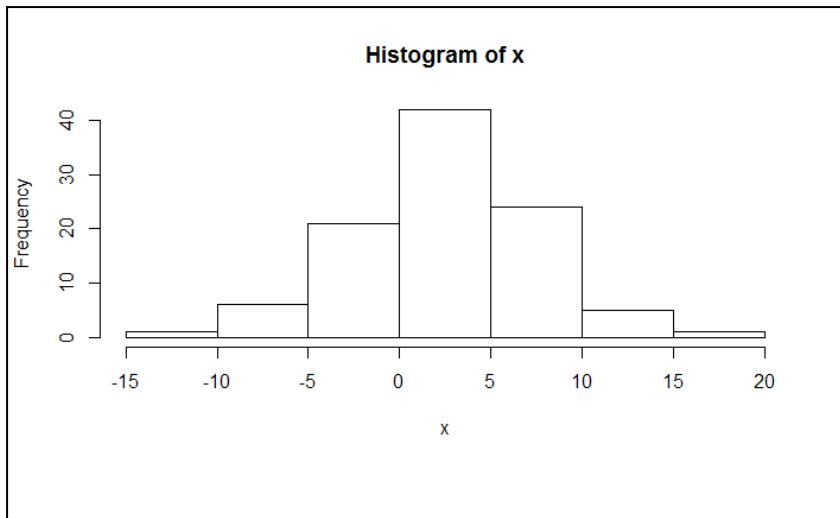
Cumulative density function (pnorm)

7. To calculate the quantiles for a specific distribution, you can use `qnorm`. The function, `qnorm`, can be treated as the inverse of `pnorm`, which returns the Z-score of a given probability:

```
> qnorm(0.5)
[1] 0
> qnorm(pnorm(0))
[1] 0
```

8. To generate random numbers from a normal distribution, one can use the `rnorm` function and specify the number of generated numbers. Also, one can define optional arguments, such as the mean and standard deviation:

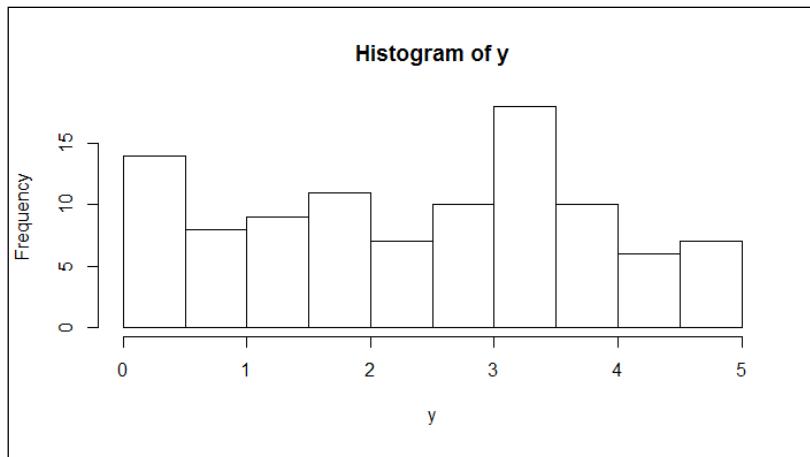
```
> set.seed(50)
> x = rnorm(100,mean=3, sd=5)
> hist(x)
```



Histogram of a normal distribution

9. To calculate the uniform distribution, the `runif` function generates random numbers from a uniform distribution. The user can specify the range of the generated numbers by specifying variables, such as the minimum and maximum. For the following example, the user generates 100 random variables from 0 to 5:

```
> set.seed(50)
> y = runif(100,0,5)
> hist(y)
```



Histogram of a uniform distribution

10. Lastly, if you would like to test the normality of the data, the most widely used test for this is the Shapiro-Wilks test. Here, we demonstrate how to perform a test of normality on both samples from the normal and uniform distributions, respectively:

```
> shapiro.test(x)
```

Shapiro-Wilk normality test

```
data: x
W = 0.9938, p-value = 0.9319
> shapiro.test(y)
```

Shapiro-Wilk normality test

```
data: y
W = 0.9563, p-value = 0.002221
```

## How it works...

In this recipe, we first introduce `dnorm`, a probability density function, which returns the height of a normal curve. With a single input specified, the input value is called a standard score or a z-score. Without any other arguments specified, it is assumed that the normal distribution is in use with a mean of zero and a standard deviation of one. We then introduce three ways to draw standard and normal distributions.

After this, we introduce `pnorm`, a cumulative density function. The function, `pnorm`, can generate the area under a given value. In addition to this, `pnorm` can be also used to calculate the p-value from a normal distribution. One can get the p-value by subtracting 1 from the number, or assigning `True` to the option, `lower.tail`. Similarly, one can use the `plot` function to plot the cumulative density.

In contrast to `pnorm`, `qnorm` returns the z-score of a given probability. Therefore, the example shows that the application of a `qnorm` function to a `pnorm` function will produce the exact input value.

Next, we show you how to use the `rnorm` function to generate random samples from a normal distribution, and the `runif` function to generate random samples from the uniform distribution. In the function, `rnorm`, one has to specify the number of generated numbers and we may also add optional augments, such as the mean and standard deviation. Then, by using the `hist` function, one should be able to find a bell-curve in figure 3. On the other hand, for the `runif` function, with the minimum and maximum specifications, one can get a list of sample numbers between the two. However, we can still use the `hist` function to plot the samples. It is clear that the output figure (shown in the preceding figure) is not in a bell-shape, which indicates that the sample does not come from the normal distribution.

Finally, we demonstrate how to test data normality with the Shapiro-Wilks test. Here, we conduct the normality test on both the normal and uniform distribution samples, respectively. In both outputs, one can find the p-value in each test result. The p-value shows the changes, which show that the sample comes from a normal distribution. If the p-value is higher than 0.05, we can conclude that the sample comes from a normal distribution. On the other hand, if the value is lower than 0.05, we conclude that the sample does not come from a normal distribution.

## There's more...

Besides the normal distribution, you can obtain a t distribution, binomial distribution, and Chi-squared distribution by using the built-in functions of R. You can use the `help` function to access further information about this:

- ▶ For a t distribution:  
  > `help(TDist)`

- ▶ For a binomial distribution:  
`>help(Binomial)`
- ▶ For the Chi-squared distribution:  
`>help(Chisquare)`

To learn more about the distributions in the package, the user can access the `help` function with the keyword `distributions` to find all related documentation on this:

```
> help(distributions)
```

## Working with univariate descriptive statistics in R

Univariate descriptive statistics describes a single variable for unit analysis, which is also the simplest form of quantitative analysis. In this recipe, we introduce some basic functions used to describe a single variable.

### Getting ready

We need to apply descriptive statistics to a sample data. Here, we use the built-in `mtcars` data as our example.

### How to do it...

Perform the following steps:

1. First, load the `mtcars` data into a data frame with a variable named `mtcars`:  
`> data(mtcars)`
2. To obtain the vector range, the `range` function will return the lower and upper bound of the vector:  
`> range(mtcars$mpg)`  
[1] 10.4 33.9
3. Compute the length of the variable:  
`> length(mtcars$mpg)`  
[1] 32
4. Obtain the mean of mpg:  
`> mean(mtcars$mpg)`  
[1] 20.09062

5. Obtain the median of the input vector:

```
> median(mtcars$mpg)
[1] 19.2
```

6. To obtain the standard deviation of the input vector:

```
> sd(mtcars$mpg)
[1] 6.026948
```

7. To obtain the variance of the input vector:

```
> var(mtcars$mpg)
[1] 36.3241
```

8. The variance can also be computed with the square of standard deviation:

```
> sd(mtcars$mpg) ^ 2
[1] 36.3241
```

9. To obtain the **Interquartile Range (IQR)**:

```
> IQR(mtcars$mpg)
[1] 7.375
```

10. To obtain the quantile:

```
> quantile(mtcars$mpg, 0.67)
67%
21.4
```

11. To obtain the maximum of the input vector:

```
> max(mtcars$mpg)
[1] 33.9
```

12. To obtain the minima of the input vector:

```
> min(mtcars$mpg)
[1] 10.4
```

13. To obtain a vector with elements that are the cumulative maxima:

```
> cummax(mtcars$mpg)
[1] 21.0 21.0 22.8 22.8 22.8 22.8 22.8 24.4 24.4 24.4 24.4 24.4
24.4 24.4 24.4 24.4
[17] 24.4 32.4 32.4 33.9 33.9 33.9 33.9 33.9 33.9 33.9 33.9 33.9
33.9 33.9 33.9 33.9
```

14. To obtain a vector with elements that are the cumulative minima:

```
> cummin(mtcars$mpg)
[1] 21.0 21.0 21.0 21.0 18.7 18.1 14.3 14.3 14.3 14.3 14.3 14.3
14.3 14.3 10.4 10.4
[17] 10.4 10.4 10.4 10.4 10.4 10.4 10.4 10.4 10.4 10.4 10.4 10.4 10.4
10.4 10.4 10.4 10.4
```

15. To summarize the dataset, you can apply the `summary` function:

```
> summary(mtcars)
```

16. To obtain a frequency count of the categorical data, take `cyl` of `mtcars` as an example:

```
> table(mtcars$cyl)
```

```
 4 6 8
11 7 14
```

17. To obtain a frequency count of numerical data, you can use a stem plot to outline the data shape; `stem` produces a stem-and-leaf plot of the given values:

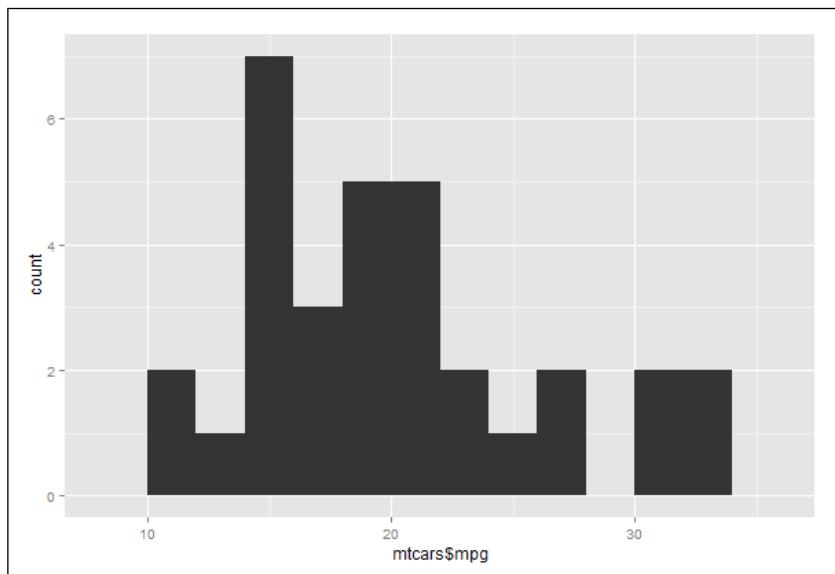
```
> stem(mtcars$mpg)
```

The decimal point is at the |

```
10 | 44
12 | 3
14 | 3702258
16 | 438
18 | 17227
20 | 00445
22 | 88
24 | 4
26 | 03
28 |
30 | 44
32 | 49
```

18. You can use a histogram of ggplot to plot the same stem-and-leaf figure:

```
> library(ggplot2)
> qplot(mtcars$mpg, binwidth=2)
```



Histogram of mpg of mtcars

## How it works...

Univariate descriptive statistics generate the frequency distribution of datasets. Moreover, they can be used to identify the obvious patterns in the data and the characteristics of the variates to provide a better understanding of the data from a holistic viewpoint. Additionally, they can provide information about the central tendency and descriptors of the skewness of individual cases. Therefore, it is common to see that univariate analysis is conducted at the beginning of the data exploration process.

To begin the exploration of data, we first load the dataset, `mtcars`, to an R session. From the data, we apply `range`, `length`, `mean`, `median`, `sd`, `var`, `IQR`, `quantile`, `min`, `max`, `cumin`, and `cumax` to obtain the descriptive statistic of the attribute, `mpg`. Then, we use the `summary` function to obtain summary information about `mtcars`.

Next, we obtain a frequency count of the categorical data (`cyl`). To obtain a frequency count of the numerical data, we use a stem plot to outline the data shape. Lastly, we use a histogram with the `binwidth` argument in 2 to generate a plot similar to the stem-and-leaf plot.

## There's more...

One common scenario in univariate descriptive statistics is to find the mode of a vector. In R, there is no built-in function to help the user obtain the mode of the data. However, one can implement the mode function by using the following code:

```
> mode = function(x) {
+ temp = table(x)
+ names(temp)[temp == max(temp)]
+ }
```

By applying the `mode` function on the vector, `mtcars$mpg`, you can find the most frequently occurring numeric value or category of a given vector:

```
> x = c(1,2,3,3,3,4,4,5,5,5,6)
> mode(x)
[1] "3" "5"
```

## Performing correlations and multivariate analysis

To analyze the relationship of more than two variables, you would need to conduct multivariate descriptive statistics, which allows the comparison of factors. Additionally, it prevents the effect of a single variable from distorting the analysis. In this recipe, we will discuss how to conduct multivariate descriptive statistics using a correlation and covariance matrix.

### Getting ready

Ensure that `mtcars` has already been loaded into a data frame within an R session.

### How to do it...

Perform the following steps:

1. Here, you can get the covariance matrix by inputting the first three variables in `mtcars` to the `cov` function:

```
> cov(mtcars[1:3])
```

|      | mpg         | cyl        | disp       |
|------|-------------|------------|------------|
| mpg  | 36.324103   | -9.172379  | -633.0972  |
| cyl  | -9.172379   | 3.189516   | 199.6603   |
| disp | -633.097208 | 199.660282 | 15360.7998 |

2. To obtain a correlation matrix of the dataset, we input the first three variables of `mtcars` to the `cor` function:

```
> cor(mtcars[1:3])
```

|      | mpg        | cyl        | disp       |
|------|------------|------------|------------|
| mpg  | 1.0000000  | -0.8521620 | -0.8475514 |
| cyl  | -0.8521620 | 1.0000000  | 0.9020329  |
| disp | -0.8475514 | 0.9020329  | 1.0000000  |

## How it works...

In this recipe, we have demonstrated how to apply correlation and covariance to discover the relationship between multiple variables.

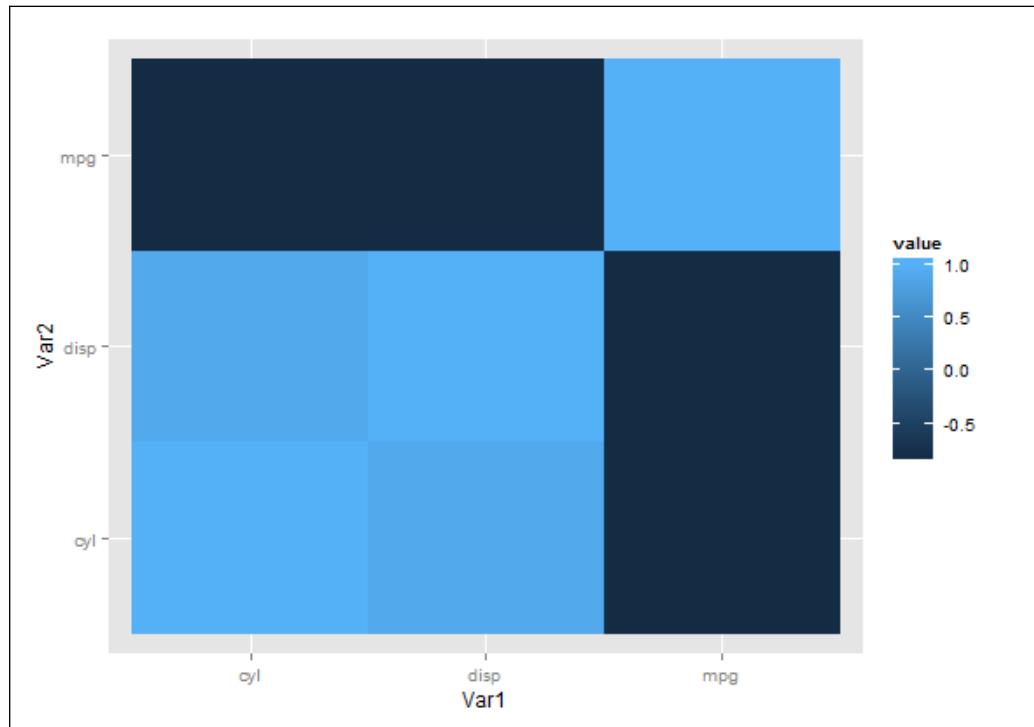
First, we compute a covariance matrix of the first three `mtcars` variables. Covariance can measure how variables are linearly related. Thus, a positive covariance (for example, `cyl` versus `mpg`) indicates that the two variables are positively linearly related. On the other hand, a negative covariance (for example, `mpg` versus `disp`) indicates the two variables are negatively linearly related. However, due to the variance of different datasets, the covariance score of these datasets is not comparable. As a result, if you would like to compare the strength of the linear relation between two variables in a different dataset, you should use the normalized score, that is, the correlation coefficient instead of covariance.

Next, we apply a `cor` function to obtain a correlation coefficient matrix of three variables within the `mtcars` dataset. In the correlation coefficient matrix, the numeric element of the matrix indicates the strength of the relationship between the two variables. If the correlation coefficient of a variable against itself scores 1, the variable has a positive relationship against itself. The `cyl` and `mpg` variables have a correlation coefficient of -0.85, which means they have a strong, negative relationship. On the other hand, the `disp` and `cyl` variables score 0.90, which may indicate that they have a strong, positive relationship.

## See also

- You can use `ggplot` to plot the heatmap of the correlation coefficient matrix:

```
> library(reshape2)
> qplot(x=Var1, y=Var2, data=melt(cor(mtcars[1:3])), fill=value,
 geom="tile")
```



The correlation coefficient matrix heatmap

## Operating linear regression and multivariate analysis

Linear regression is a method to assess the association between dependent and independent variables. In this recipe, we will cover how to conduct linear regression for multivariate analysis.

### Getting ready

Ensure that `mtcars` has already been loaded into a data frame within an R session.

# How to do it...

Perform the following steps:

1. To fit variables into a linear model, you can use the `lm` function:

```
> lmfit = lm(mtcars$mpg ~ mtcars$cyl)
> lmfit
```

Call:

```
lm(formula = mtcars$mpg ~ mtcars$cyl)
```

Coefficients:

| (Intercept) | mtcars\$cyl |
|-------------|-------------|
| 37.885      | -2.876      |

2. To get detailed information on the fitted model, you can use the `summary` function:

```
> summary(lmfit)
```

Call:

```
lm(formula = mtcars$mpg ~ mtcars$cyl)
```

Residuals:

| Min     | 1Q      | Median | 3Q     | Max    |
|---------|---------|--------|--------|--------|
| -4.9814 | -2.1185 | 0.2217 | 1.0717 | 7.5186 |

Coefficients:

|                | Estimate | Std. Error | t value | Pr(> t )     |       |   |
|----------------|----------|------------|---------|--------------|-------|---|
| (Intercept)    | 37.8846  | 2.0738     | 18.27   | < 2e-16 ***  |       |   |
| mtcars\$cyl    | -2.8758  | 0.3224     | -8.92   | 6.11e-10 *** |       |   |
| ---            |          |            |         |              |       |   |
| Signif. codes: | 0 ****   | 0.001 ***  | 0.01 ** | 0.05 *       | 0.1 . | 1 |

Residual standard error: 3.206 on 30 degrees of freedom

Multiple R-squared: 0.7262, Adjusted R-squared: 0.7171

F-statistic: 79.56 on 1 and 30 DF, p-value: 6.113e-10

3. To create an analysis of a variance table, one can employ the anova function:

```
> anova(lmfit)
```

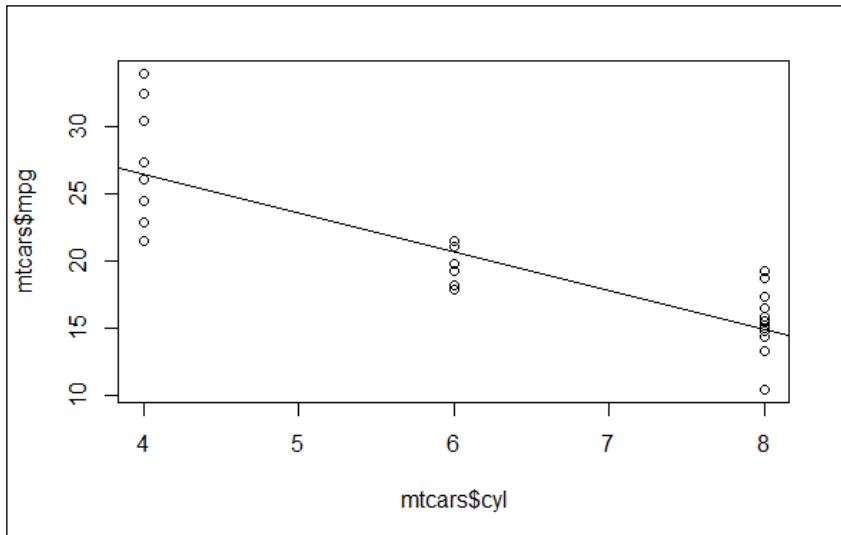
#### Analysis of Variance Table

Response: mtcars\$mpg

|                | Df  | Sum Sq | Mean Sq | F value | Pr(>F)        |     |      |      |     |     |   |
|----------------|-----|--------|---------|---------|---------------|-----|------|------|-----|-----|---|
| mtcars\$cyl    | 1   | 817.71 | 817.71  | 79.561  | 6.113e-10 *** |     |      |      |     |     |   |
| Residuals      | 30  | 308.33 | 10.28   |         |               |     |      |      |     |     |   |
|                | --- |        |         |         |               |     |      |      |     |     |   |
| Signif. codes: | 0   | '****' | 0.001   | '**'    | 0.01          | '*' | 0.05 | '..' | 0.1 | ' ' | 1 |

4. To plot the regression line on a scatter plot of two variables, you first plot cyl against mpg in it, then use the abline function to add a regression line on the plot:

```
> lmfit = lm(mtcars$mpg ~ mtcars$cyl)
> plot(mtcars$cyl, mtcars$mpg)
> abline(lmfit)
```



The regression plot of cyl against mpg

## How it works...

In this recipe, we apply the linear model function, `lm`, which builds a linear fitted model of two variables and returns the formula and coefficient. Next, we apply the `summary` function to retrieve the detailed information (including F-statistic and P-value) of the model. The purpose of F-statistic is to test the statistical significance of the model. It produces an F-value, which is the ratio of the model mean square to the error mean square. Thus, a large F-value indicates that more of the total variability is accounted for by the regression model. Then, we can use the F-value to support or reject the null hypothesis that all of the regression coefficients are equal to zero. In other words, the null hypothesis is rejected if the F-value is large and shows that the regression model has a predictive capability. On the other hand, P-values of each attribute test the null hypothesis that the coefficient is equal to zero (no effect on the response variable). In other words, a low p-value can reject a null hypothesis and indicates that a change in the predictor's value is related to the value of the response variable.

Next, we apply the `anova` function on the fitted model to determine the variance. The function outputs the sum of squares, which stands for the variability of the model's predicted value. Further, to visualize the linear relationship between two variables, the `abline` function can add a regression line on a scatter plot of `mpg` against `cyl`. From the preceding figure, it is obvious that the `mpg` and `cyl` variables are negatively related.

## See also

- ▶ For more information on how to perform linear and nonlinear regression analysis, please refer to the Chapter, *Understanding Regression Analysis*

## Conducting an exact binomial test

While making decisions, it is important to know whether the decision error can be controlled or measured. In other words, we would like to prove that the hypothesis formed is unlikely to have occurred by chance, and is statistically significant. In hypothesis testing, there are two kinds of hypotheses: null hypothesis and alternative hypothesis (or research hypothesis). The purpose of hypothesis testing is to validate whether the experiment results are significant. However, to validate whether the alternative hypothesis is acceptable, it is deemed to be true if the null hypothesis is rejected.

In the following recipes, we will discuss some common statistical testing methods. First, we will cover how to conduct an exact binomial test in R.

## Getting ready

Since the `binom.test` function originates from the `stats` package, make sure the `stats` library is loaded.

## How to do it...

Perform the following step:

1. Assume there is a game where a gambler can win by rolling the number-six-on a dice. As part of the rules, gamblers can bring their own dice. If a gambler tried to cheat in a game, he would use a loaded dice to increase his chance of winning. Therefore, if we observe that the gambler won 92 out of 315 games, we could determine whether the dice was fair by conducting an exact binomial test:

```
> binom.test(x=92, n=315, p=1/6)
```

```
Exact binomial test

data: 92 and 315
number of successes = 92, number of trials = 315, p-value =
3.458e-08
alternative hypothesis: true probability of success is not equal
to 0.1666667
95 percent confidence interval:
0.2424273 0.3456598
sample estimates:
probability of success
0.2920635
```

## How it works...

A binomial test uses the binomial distribution to find out whether the true success rate is likely to be  $P$  for  $n$  trials with the binary outcome. The formula of the probability,  $P$ , can be defined in following equation:

$$P(X = k) = \binom{n}{k} p^k q^{n-k}$$

Here,  $X$  denotes the random variables, counting the number of outcomes of the interest;  $n$  denotes the number of trials;  $k$  indicates the number of successes;  $p$  indicates the probability of success; and  $q$  denotes the probability of failure.

After we have computed the probability,  $P$ , we can then perform a sign test to determine whether the success probability is similar to what we expected. If the probability is not equal to what we expected, we can reject the null hypothesis.

By definition, the null hypothesis is a skeptical perspective or a statement about the population parameter that will be tested. The null hypothesis is denoted by  $H_0$ . An alternative hypothesis is represented by a range of population values, which are not included in the null hypothesis. The alternative hypothesis is denoted by  $H_1$ . In this case, the null and alternative hypothesis, respectively, are illustrated as:

- ▶  **$H_0$  (null hypothesis):** The true probability of success is equal to what we expected
- ▶  **$H_1$  (alternative hypothesis):** The true probability of success is not equal to what we expected

In this example, we demonstrate how to use a binomial test to determine the number of times the dice is rolled, the frequency of rolling the number six, and the probability of rolling a six from an unbiased dice. The result of the t-test shows that the p-value = 3.458e-08 (lower than 0.05). For significance, at the five percent level, the null hypothesis (the dice is unbiased) is rejected as too many sixes were rolled (the probability of success = 0.2920635).

## See also

- ▶ To read more about the usage of the exact binomial test, please use the `help` function to view related documentation on this:

```
> ?binom.test
```

## Performing student's t-test

A one sample t-test enables us to test whether two means are significantly different; a two sample t-test allows us to test whether the means of two independent groups are different. In this recipe, we will discuss how to conduct one sample t-test and two sample t-tests using R.

## Getting ready

Ensure that `mtcars` has already been loaded into a data frame within an R session. As the `t.test` function originates from the `stats` package, make sure the library, `stats`, is loaded.

# How to do it...

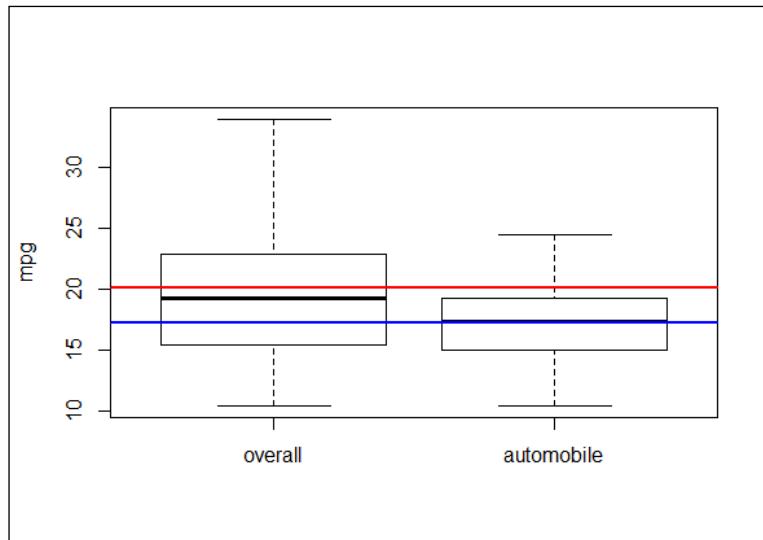
Perform the following steps:

1. First, we visualize the attribute, mpg, against am using a boxplot:

```
> boxplot(mtcars$mpg, mtcars$mpg[mtcars$am==0], ylab = "mpg", name
s=c("overall", "automobile"))

> abline(h=mean(mtcars$mpg), lwd=2, col="red")

> abline(h=mean(mtcars$mpg[mtcars$am==0]), lwd=2, col="blue")
```



The boxplot of mpg of the overall population and automobiles

2. We then perform a statistical procedure to validate whether the average mpg of automobiles is lower than the average of the overall mpg:

```
> mpg_mu = mean(mtcars$mpg)

> mpg_am = mtcars$mpg[mtcars$am == 0]

> t.test(mpg_am, mu = mpg_mu)
```

## One Sample t-test

```
data: mpg_am
t = -3.3462, df = 18, p-value = 0.003595
alternative hypothesis: true mean is not equal to 20.09062
95 percent confidence interval:
```

```
15.29946 18.99528
```

sample estimates:

mean of x

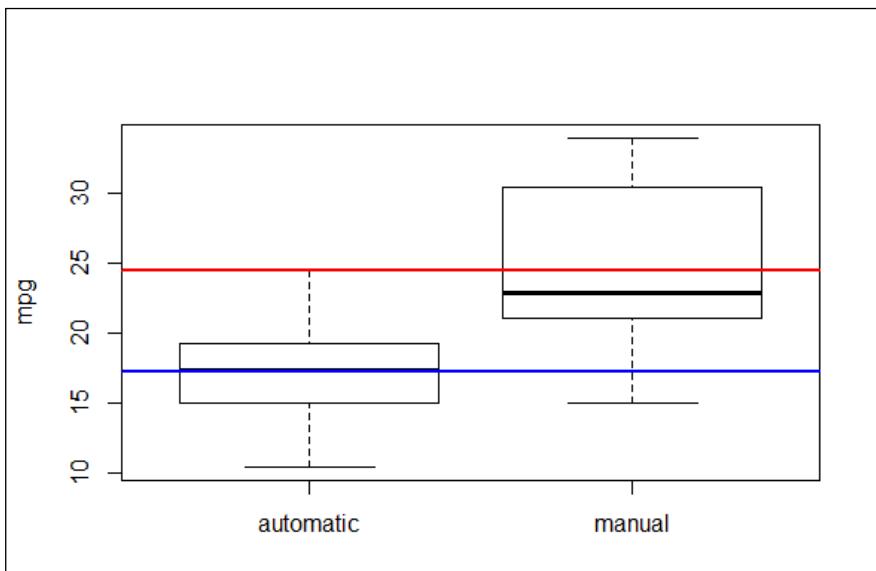
```
17.14737
```

3. We begin visualizing the data by plotting a boxplot:

```
> boxplot(mtcars$mpg~mtcars$am, ylab='mpg', names=c('automatic', 'manual'))

> abline(h=mean(mtcars$mpg[mtcars$am==0]), lwd=2, col="blue")

> abline(h=mean(mtcars$mpg[mtcars$am==1]), lwd=2, col="red")
```



The boxplot of mpg of automatic and manual transmission cars

4. The preceding figure reveals that the mean mpg of automatic transmission cars is lower than the average mpg of manual transmission vehicles:

```
> t.test(mtcars$mpg~mtcars$am)
```

Welch Two Sample t-test

```
data: mtcars$mpg by mtcars$am
```

```
t = -3.7671, df = 18.332, p-value = 0.001374
```

```
alternative hypothesis: true difference in means is not equal to 0
```

```
95 percent confidence interval:
```

```
-11.280194 -3.209684
```

```
sample estimates:
```

```
mean in group 0 mean in group 1
```

```
17.14737
```

```
24.39231
```

## How it works...

Student's t-test is where the test statistic follows a normal distribution (the student's t distribution) if the null hypothesis is true. It can be used to determine whether there is a difference between two independent datasets. Student's t-test is best used with the problems associated with an inference based on small samples.

In this recipe, we discuss one sample student's t-test and two sample student's t-tests. In the one sample student's t-test, a research question often asked is, "Is the mean of the population different from the null hypothesis?" Thus, in order to test whether the average mpg of automobiles is lower than the overall average mpg, we first use a boxplot to view the differences between populations without making any assumptions. From the preceding figure, it is clear that the mean of mpg of automobiles (the blue line) is lower than the average mpg (red line) of the overall population. Then, we apply one sample t-test; the low p-value of 0.003595 ( $< 0.05$ ) suggests that we should reject the null hypothesis that the mean mpg for automobiles is less than the average mpg of the overall population.

As a one sample t-test enables us to test whether two means are significantly different, a two sample t-test allows us to test whether the means of two independent groups are different. Similar to a one sample t-test, we first use a boxplot to see the differences between populations and then apply a two-sample t-test. The test results shows the p-value = 0.01374 ( $p < 0.05$ ). In other words, the test provides evidence that rejects the null hypothesis, which shows the mean mpg of cars with automatic transmission differs from the cars with manual transmission.

## See also

- ▶ To read more about the usage of student's t-test, please use the `help` function to view related documents:  
  
▶ `?t.test`

# Performing the Kolmogorov-Smirnov test

A one-sample Kolmogorov-Smirnov test is used to compare a sample with a reference probability. A two-sample Kolmogorov-Smirnov test compares the cumulative distributions of two datasets. In this recipe, we will demonstrate how to perform the Kolmogorov-Smirnov test with R.

## Getting ready

Ensure that `mtcars` has already been loaded into a data frame within an R session. As the `ks.test` function is originated from the `stats` package, make sure the `stats` library is loaded.

## How to do it...

Perform the following steps:

1. Validate whether the dataset, `x` (generated with the `rnorm` function), is distributed normally with a one-sample Kolmogorov-Smirnov test:

```
> x = rnorm(50)
> ks.test(x, "pnorm")
```

**One-sample Kolmogorov-Smirnov test**

```
data: x
D = 0.1698, p-value = 0.0994
alternative hypothesis: two-sided
```

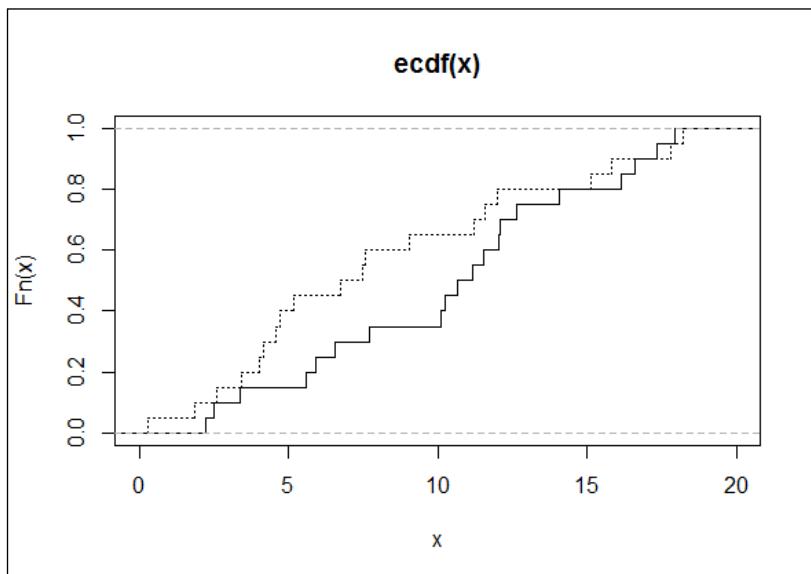
2. Next, you can generate uniformly distributed sample data:

```
> set.seed(3)
> x = runif(n=20, min=0, max=20)

> y = runif(n=20, min=0, max=20)
```

3. We first plot ecdf of two generated data samples:

```
> plot(ecdf(x), do.points = FALSE, verticals=T, xlim=c(0, 20))
> lines(ecdf(y), lty=3, do.points = FALSE, verticals=T)
```



The ecdf plot of two generated data samples

4. Finally, we apply a two-sample Kolmogorov-Smirnov test on two groups of data:

```
> ks.test(x,y)
```

Two-sample Kolmogorov-Smirnov test

```
data: x and y
D = 0.3, p-value = 0.3356
alternative hypothesis: two-sided
```

## How it works...

The **Kolmogorov-Smirnov test (K-S test)** is a nonparametric and statistical test, used for the equality of continuous probability distributions. It can be used to compare a sample with a reference probability distribution (a one sample K-S test), or it can directly compare two samples (a two sample K-S test). The test is based on the empirical distribution function (ECDF). Let  $x_1, x_2 \dots x_n$  be a random sample of size, n; the empirical distribution function,  $F_n(x)$ , is defined as:

$$F_n(x) = \frac{1}{n} \sum_{i=1}^n I\{x_i \leq x\}$$

Here,  $I\{x_i \leq x\}$  is the indicator function. If  $x_i \leq x$ , the function equals to 1. Otherwise, the function equals to 0.

The Kolmogorov-Smirnov statistic (D) is based on the greatest (where  $\sup_x$  denotes the supremum) vertical difference between  $F(x)$  and  $F_n(x)$ . It is defined as:

$$D_n = \sup_x |F_n(x) - F(x)|$$

$H_0$  is the sample follows the specified distribution.  $H_1$  is the sample does not follow the specified distribution.

If  $D_n$  is greater than the critical value obtained from a table, then we reject  $H_0$  at the level of significance  $\alpha$ .

We first test whether a random number generated from a normal distribution is normally distributed. At the 5 percent significance level, the p-value of 0.0994 indicates that the input is normally distributed.

Then, we plot an empirical cumulative distribution function (`ecdf`) plot to show how a two-sample test calculates the maximum distance D (showing 0.3), and apply the two-sample Kolmogorov-Smirnov test to discover whether the two input datasets possibly come from the same distribution.

The p-value is above 0.05, which does not reject the null hypothesis. In other words, it means the two datasets are possibly from the same distribution.

## See also

- ▶ To read more about the usage of the Kolmogorov-Smirnov test, please use the `help` function to view related documents:  
`> ?ks.test`
- ▶ As for the definition of an empirical cumulative distribution function, please refer to the help page of `ecdf`:  
`> ?ecdf`

# Understanding the Wilcoxon Rank Sum and Signed Rank test

The Wilcoxon Rank Sum and Signed Rank test (or Mann-Whitney-Wilcoxon) is a nonparametric test of the null hypothesis, which shows that the population distribution of two different groups are identical without assuming that the two groups are normally distributed. This recipe will show how to conduct the Wilcoxon Rank Sum and Signed Rank test in R.

## Getting ready

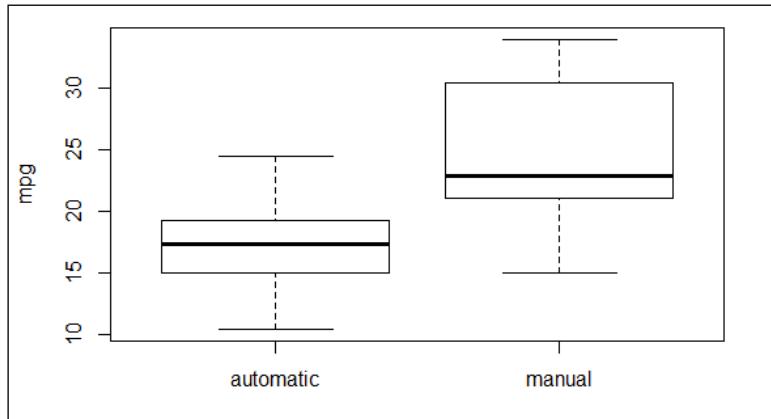
Ensure that `mtcars` has already been loaded into a data frame within an R session. As the `wilcox.test` function is originated from the `stats` package, make sure the library, `stats`, is loaded.

## How to do it...

Perform the following steps:

1. We first plot the data of `mtcars` with the `boxplot` function:

```
> boxplot(mtcars$mpg ~ mtcars$am, ylab='mpg', names=c('automatic', 'manual'))
```



The boxplot of mpg of automatic cars and manual transmission cars

2. Next, we still perform a Wilcoxon Rank Sum test to validate whether the distribution of automatic transmission cars is identical to that of manual transmission cars:

```
> wilcox.test(mpg ~ am, data=mtcars)
```

Wilcoxon rank sum test with continuity correction

```
data: mpg by am
W = 42, p-value = 0.001871
alternative hypothesis: true location shift is not equal to 0

Warning message:
In wilcox.test.default(x = c(21.4, 18.7, 18.1, 14.3, 24.4, 22.8,
:
cannot compute exact p-value with ties
```

## How it works...

In this recipe, we discuss a nonparametric test method, the Wilcoxon Rank Sum test (also known as the Mann-Whitney U-test). For student's t-test, it is assumed that the differences between the two samples are normally distributed (and it also works best when the two samples are normally distributed). However, when the normality assumption is uncertain, one can adopt the Wilcoxon Rank Sum Test to test a hypothesis.

Here, we used a Wilcoxon Rank Sum test to determine whether the mpg of automatic and manual transmission cars in the dataset, `mtcars`, is distributed identically. From the test result, we see that the p-value = 0.001871 (< 0.05) rejects the null hypothesis, and also reveals that the distribution of mpg in automatic and manual transmission cars is not identical. When performing this test, you may receive the warning message, "cannot compute exact p-value with ties", which indicates that there are duplicate values within the dataset. The warning message will be cleared once the duplicate values are removed.

## See also

- ▶ To read more about the usage of the Wilcoxon Rank Sum and Signed Rank Test, please use the `help` function to view the concerned documents:  
`> ? wilcox.test`

## Working with Pearson's Chi-squared test

In this recipe, we introduce Pearson's Chi-squared test, which is used to examine whether the distributions of categorical variables of two groups differ. We will discuss how to conduct Pearson's Chi-squared Test in R.

## Getting ready

Ensure that `mtcars` has already been loaded into a data frame within an R session. Since the `chisq.test` function is originated from the `stats` package, make sure the library, `stats`, is loaded.

## How to do it

Perform the following steps:

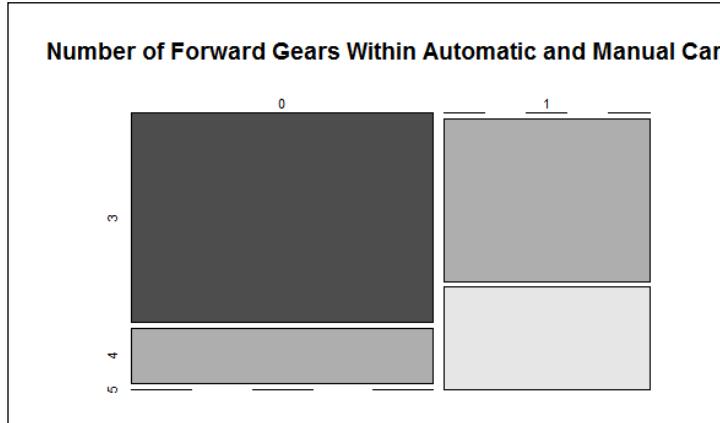
1. To make the counting table, we first use the contingency table built with the inputs of the transmission type and number of forward gears:

```
> ftable = table(mtcars$am, mtcars$gear)
> ftable
```

|   |    |   |
|---|----|---|
| 3 | 4  | 5 |
| 0 | 15 | 4 |
| 1 | 0  | 8 |

2. We then plot the mosaic plot of the contingency table:

```
> mosaicplot(ftable, main="Number of Forward Gears Within
Automatic and Manual Cars", color = TRUE)
```



Number of forward gears in automatic and manual cars

3. Next, we perform the Pearson's Chi-squared test on the contingency table to test whether the numbers of gears in automatic and manual transmission cars is the same:

```
> chisq.test(ftable)
```

```
Pearson's Chi-squared test
```

```
data: ftable
```

```
X-squared = 20.9447, df = 2, p-value = 2.831e-05
```

```
Warning message:
```

```
In chisq.test(ftable) : Chi-squared approximation may be incorrect
```

## How it works...

Pearson's Chi-squared test is a statistical test used to discover whether there is a relationship between two categorical variables. It is best used for unpaired data from large samples. If you would like to conduct Pearson's Chi-squared test, you need to make sure that the input samples satisfy two assumptions: firstly, the two input variables should be categorical. Secondly, the variable should include two or more independent groups.

In Pearson's Chi-squared test, the assumption is that we have two variables, A and B; we can illustrate the null and alternative hypothesis in the following statements:

- ▶  $H_0$ : Variable A and variable B are independent
- ▶  $H_1$ : Variable A and variable B are not independent

To test whether the null hypothesis is correct or incorrect, the Chi-squared test takes these steps.

It calculates the Chi-squared test statistic,  $X^2$ :

$$X^2 = \sum_{i=1}^r \sum_{j=1}^c \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}}$$

Here, r is the number of rows in the contingency table, c is the number of columns in the contingency table,  $O_{i,j}$  is the observed frequency count,  $E_{i,j}$  is the expected frequency count.

It determines the degrees of freedom,  $df$ , of that statistic. The degree of freedom is equal to:

$$df = (r - 1) \times (c - 1)$$

Here,  $r$  is the number of levels for one variable, and  $c$  is the number of levels for another variable.

It compares  $X^2$  to the critical value from the Chi-squared distribution with the degrees of freedom.

In this recipe, we use a contingency table and mosaic plot to illustrate the differences in count numbers. It is obvious that the number of forward gears is less in automatic transmission cars than in manual transmission cars.

Then, we perform the Pearson's Chi-squared test on the contingency table to determine whether the gears in automatic and manual transmission cars are the same. The output,  $p\text{-value} = 2.831e-05 (< 0.05)$ , refutes the null hypothesis and shows the number of forward gears is different in automatic and manual transmission cars. However, the output message contains a warning message that Chi-squared approximation may be incorrect, which is because the number of samples in the contingency table is less than five.

## There's more...

To read more about the usage of the Pearson's Chi-squared test, please use the `help` function to view the related documents:

```
> ? chisq.test
```

Besides some common hypothesis testing methods mentioned in previous examples, there are other hypothesis methods provided by R:

- ▶ The Proportional test (`prop.test`): It is used to test whether the proportions in different groups are the same
- ▶ The Z-test (`simple.z.test` in the `UsingR` package): It compares the sample mean with the population mean and standard deviation
- ▶ The Bartlett Test (`bartlett.test`): It is used to test whether the variance of different groups is the same
- ▶ The Kruskal-Wallis Rank Sum Test (`kruskal.test`): It is used to test whether the distribution of different groups is identical without assuming that they are normally distributed
- ▶ The Shapiro-Wilk test (`shapiro.test`): It is used test for normality

# Conducting a one-way ANOVA

**Analysis of variance (ANOVA)** investigates the relationship between categorical independent variables and continuous dependent variables. It can be used to test whether the means of several groups are equal. If there is only one categorical variable as an independent variable, you can perform a one-way ANOVA. On the other hand, if there are more than two categorical variables, you should perform a two-way ANOVA. In this recipe, we discuss how to conduct a one-way ANOVA with R.

## Getting ready

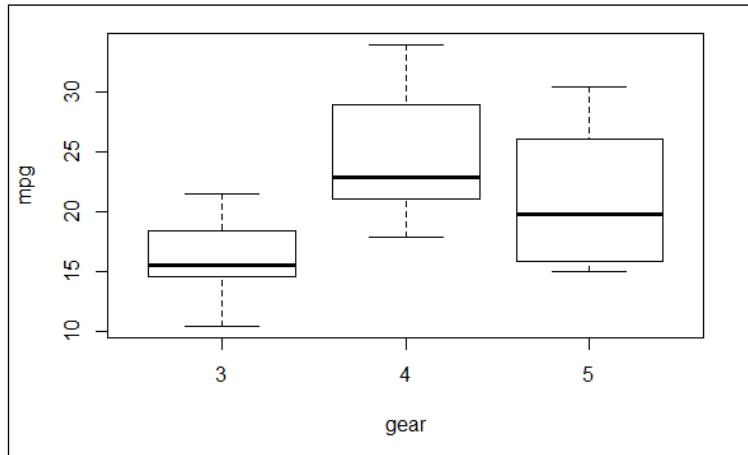
Ensure that `mtcars` has already been loaded into a data frame within an R session. Since the `oneway.test` and `TukeyHSD` functions originated from the `stats` package, make sure the library, `stats`, is loaded.

## How to do it...

Perform the following steps:

1. We begin exploring by visualizing the data with a boxplot:

```
> boxplot(mtcars$mpg~factor(mtcars$gear),xlab='gear',ylab='mpg')
```



Comparison of mpg of different numbers of forward gears

2. Next, we conduct a one-way ANOVA to examine whether the mean of mpg changes with different numbers of forward gears. We use the function, oneway.test:

```
> oneway.test(mtcars$mpg~factor(mtcars$gear))
```

One-way analysis of means (not assuming equal variances)

```
data: mtcars$mpg and factor(mtcars$gear)
```

```
F = 11.2848, num df = 2.000, denom df = 9.508, p-value = 0.003085
```

3. In addition to oneway.test, a standard function, aov, is used for the ANOVA analysis:

```
> mtcars.aov = aov(mtcars$mpg ~ as.factor(mtcars$gear))
```

```
> summary(mtcars.aov)
```

|                         | Df             | Sum Sq                                  | Mean Sq | F value | Pr(>F)       |
|-------------------------|----------------|-----------------------------------------|---------|---------|--------------|
| as.factor(mtcars\$gear) | 2              | 483.2                                   | 241.62  | 10.9    | 0.000295 *** |
| Residuals               | 29             | 642.8                                   | 22.17   |         |              |
|                         | ---            |                                         |         |         |              |
|                         | Signif. codes: | 0 **** 0.001 *** 0.01 ** 0.05 * 0.1 . 1 |         |         |              |

4. The model generated by the aov function can also generate a summary as a fitted table:

```
> model.tables(mtcars.aov, "means")
```

Tables of means

Grand mean

20.09062

```
as.factor(mtcars$gear)
```

|   |   |   |
|---|---|---|
| 3 | 4 | 5 |
|---|---|---|

|       |       |       |
|-------|-------|-------|
| 16.11 | 24.53 | 21.38 |
|-------|-------|-------|

```
rep 15.00 12.00 5.00
```

5. For the aov model, one can use TukeyHSD for a post hoc comparison test:

```
> mtcars_posthoc =TukeyHSD(mtcars.aov)
```

```
> mtcars_posthoc
```

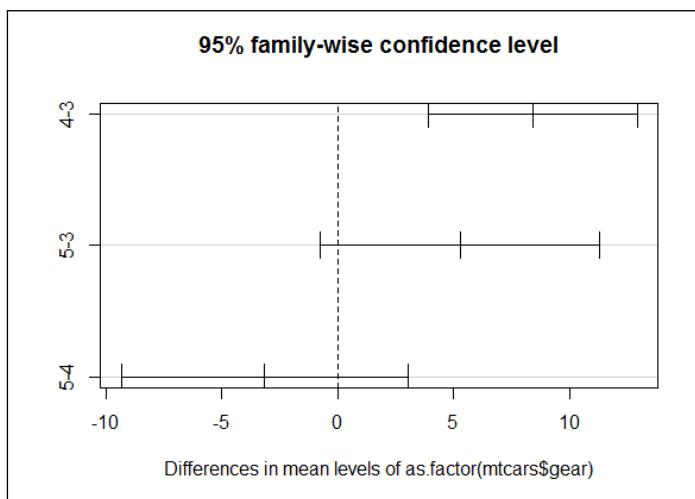
Tukey multiple comparisons of means

95% family-wise confidence level

```
Fit: aov(formula = mtcars$mpg ~ as.factor(mtcars$gear))
```

```
$`as.factor(mtcars$gear)`
diff lwr upr p adj
4-3 8.426667 3.9234704 12.929863 0.0002088
5-3 5.273333 -0.7309284 11.277595 0.0937176
5-4 -3.153333 -9.3423846 3.035718 0.4295874
```

6. Further, we can visualize the differences in mean level with a `plot` function:



The Tukey mean-difference plot of groups with different numbers of gears

## How it works...

In order to understand whether cars with a different number of forward gears have different means in mpg, we first plot the boxplot of mpg by the numbers of forward gears. This offers a simple indication if cars with a different number of forward gears have different means of mpg. We then perform the most basic form of ANOVA, a one-way ANOVA, to test whether the populations have different means.

In R, there are two functions to perform the ANOVA test: `oneway.test` and `aov`. The advantage of `oneway.test` is that the function applies a Welch correction to address the nonhomogeneity of a variance. However, it does not provide as much information as `aov`, and it does not offer a post hoc test. Next, we perform `oneway.test` and `aov` on the independent variable, `gear`, with regard to the dependent variable, `mpg`. Both test results show a small p-value, which rejects the null hypothesis that the mean between cars with a different number of forward gears have the same `mpg` mean.

As the results of ANOVA only suggest that there is a significant difference in the means within overall populations, you may not know which two populations differ in terms of their mean. Therefore, we apply the TukeyHSD post hoc comparison test on our ANOVA model. The result shows that cars with four forward gears and cars with three gears have the largest difference, as their confidence interval is the furthest to the right within the plot.

## There's more...

ANOVA relies on an F-distribution as the basis of all probability distribution. An F score is obtained by dividing the between-group variance by the in-group variance. If the overall F test was significant, you can conduct a post hoc test (or multiple comparison tests) to measure the differences between groups. The most commonly used post hoc tests are Scheffé's method, the Tukey-Kramer method, and the Bonferroni correction.

In order to interpret the output of ANOVA, you need to have a basic understanding of certain terms, including the degrees of freedom, the sum of square total, the sum of square groups, the sum of square errors, the mean square errors, and the F statistic. If you require more information about these terms, you may refer to *Using multivariate statistics* (Fidell, L. S., & Tabachnick, B. G. (2006) Boston: Allyn & Bacon.), or refer to the Wikipedia entry of Analysis of variance ([http://en.wikipedia.org/wiki/Analysis\\_of\\_variance#cite\\_ref-31](http://en.wikipedia.org/wiki/Analysis_of_variance#cite_ref-31)).

## Performing a two-way ANOVA

A two-way ANOVA can be viewed as the extension of a one-way ANOVA, for the analysis covers more than two categorical variables rather than one. In this recipe, we will discuss how to conduct a two-way ANOVA in R.

### Getting ready

Ensure that `mtcars` has already been loaded into a data frame within an R session. Since the `twoway.test`, `TukeyHSD` and `interaction.plot` functions are originated from the `stats` package, make sure the library, `stats`, is loaded.

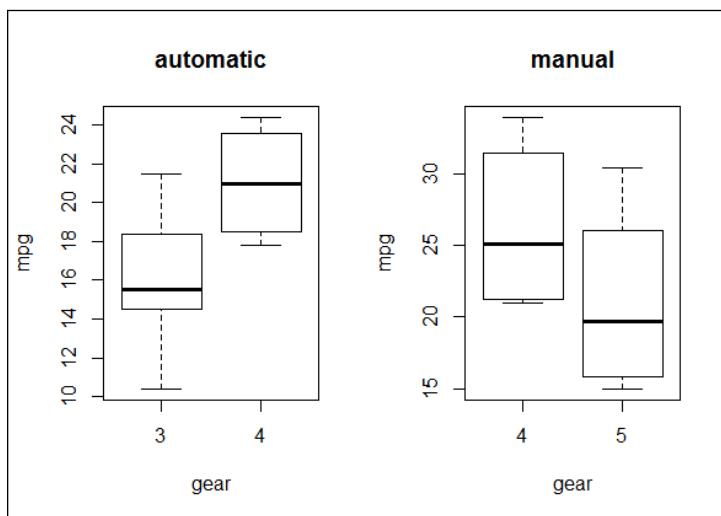
### How to do it...

Perform the following steps:

1. First we plot the two boxplots of factor gears in regard to mpg, with the plot separated from the transmission type:

```
> par(mfrow=c(1,2))
> boxplot(mtcars$mpg~mtcars$gear,subset=(mtcars$am==0),xlab='ge
ar', ylab = "mpg",main='automatic')
```

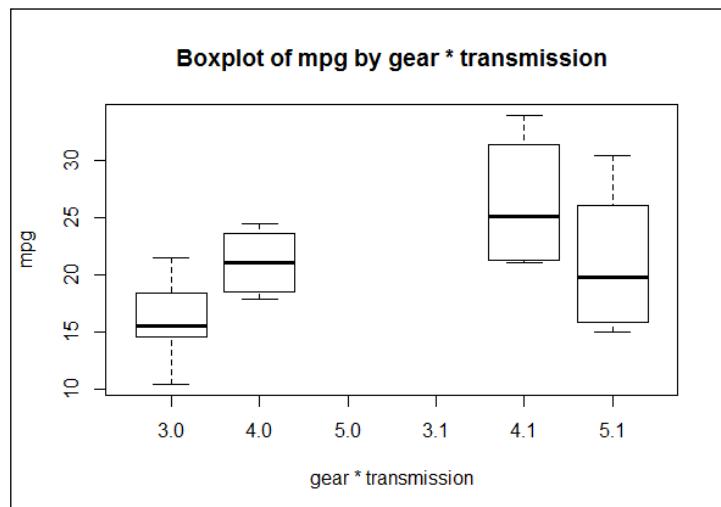
```
> boxplot(mtcars$mpg~mtcars$gear, subset=(mtcars$am==1), xlab='gear', ylab = "mpg", main='manual')
```



The boxplots of mpg by the gear group and the transmission type

2. Also, you may produce a boxplot of mpg by the number of forward gears \* transmission type, with the use of the \* operation in the boxplot function:

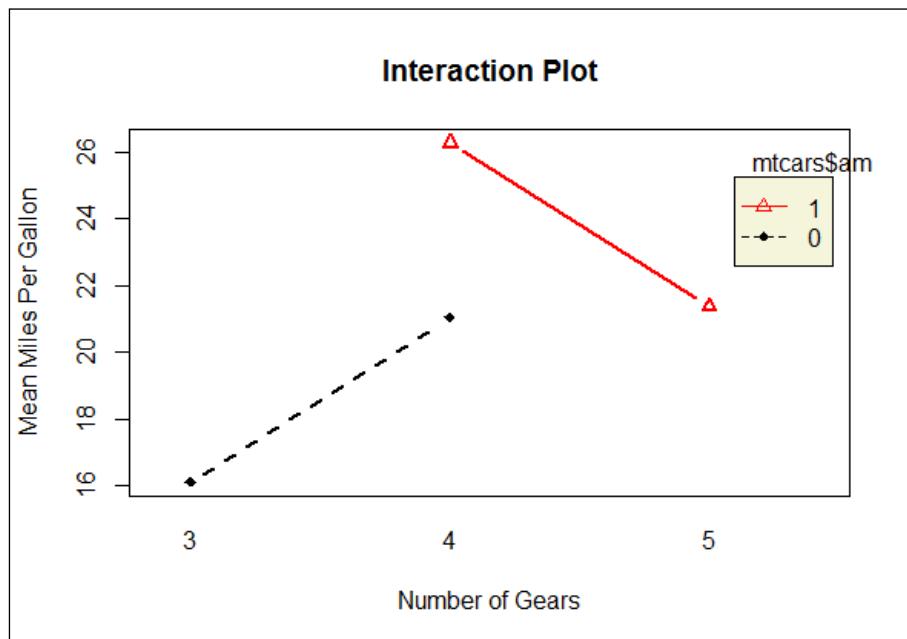
```
> boxplot(mtcars$mpg~factor(mtcars$gear)*
factor(mtcars$am), xlab='gear * transmission', ylab =
"mpg", main='Boxplot of mpg by gear * transmission')
```



The boxplot of mpg by the gear \* transmission type

3. Next, we use an interaction plot to characterize the relationship between variables:

```
> interaction.plot(mtcars$gear, mtcars$am, mtcars$mpg, type="b",
 col=c(1:3), leg.bty="o", leg.bg="beige", lwd=2, pch=c(18,24,22),
 xlab="Number of Gears", ylab="Mean Miles Per Gallon",
 main="Interaction Plot")
```



Interaction between the transmission type and the number of gears with the main effects, mpg

4. We then perform a two-way ANOVA on mpg with a combination of the gear and transmission-type factors:

```
> mpg_anova2 = aov(mtcars$mpg~factor(mtcars$gear)*factor(mtcars$am))
> summary(mpg_anova2)
Df Sum Sq Mean Sq F value Pr(>F)
factor(mtcars$gear) 2 483.2 241.62 11.869 0.000185 ***
factor(mtcars$am) 1 72.8 72.80 3.576 0.069001 .
Residuals 28 570.0 20.36

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

5. Similar to a one-way ANOVA, we can perform a post hoc comparison test to see the results of the two-way ANOVA model:

```
> TukeyHSD(mpg_anova2)
```

```
Tukey multiple comparisons of means
```

```
95% family-wise confidence level
```

```
Fit: aov(formula = mtcars$mpg ~ factor(mtcars$gear) *
factor(mtcars$am))
```

```
$`factor(mtcars$gear)`
```

|     | diff      | lwr        | upr       | p adj     |
|-----|-----------|------------|-----------|-----------|
| 4-3 | 8.426667  | 4.1028616  | 12.750472 | 0.0001301 |
| 5-3 | 5.273333  | -0.4917401 | 11.038407 | 0.0779791 |
| 5-4 | -3.153333 | -9.0958350 | 2.789168  | 0.3999532 |

```
$`factor(mtcars$am)`
```

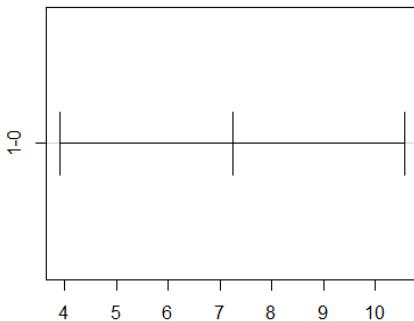
|     | diff     | lwr       | upr     | p adj     |
|-----|----------|-----------|---------|-----------|
| 1-0 | 1.805128 | -1.521483 | 5.13174 | 0.2757926 |

6. We then visualize the differences in mean levels with the `plot` function:

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

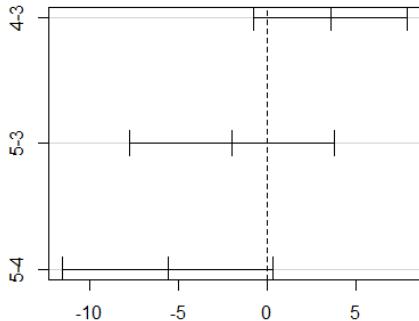
```
> plot(TukeyHSD(mpg_anova2))
```

95% family-wise confidence level



Differences in mean levels of factor(mtcars\$am)

95% family-wise confidence level



Differences in mean levels of factor(mtcars\$gear)

The comparison plot of differences in mean levels by the transmission type and the number of gears

## How it works...

In this recipe, we perform a two-way ANOVA to examine the influences of the independent variables, `gear` and `am`, on the dependent variable, `mpg`. In the first step, we use a boxplot to examine the mean of mpg by the number of gears and the transmission type. Secondly, we apply an interaction plot to visualize the change in mpg through the different numbers of gears with lines separated by the transmission type.

The resulting plot shows that the number of gears does have an effect on the mean of mpg, but does not show a positive relationship either. Thirdly, we perform a two-way ANOVA with the `aov` function. The output shows that the p-value of the `gear` factor rejects the null hypothesis, while the factor, `transmission type`, does not reject the null hypothesis. In other words, cars with different numbers of gears are more likely to have different means of mpg. Finally, in order to examine which two populations have the largest differences, we perform a post hoc analysis, which reveals that cars with four gears and three gears, respectively, have the largest difference in terms of the mean, mpg.

## See also

- ▶ For multivariate analysis of variances, the function, `manova`, can be used to examine the effect of multiple independent variables on multiple dependent variables. Further information about MANOVA is included within the `help` function in R:  
`> ?MANOVA`

# 3

# Understanding Regression Analysis

In this chapter, we will cover the following recipes:

- ▶ Fitting a linear regression model with lm
- ▶ Summarizing linear model fits
- ▶ Using linear regression to predict unknown values
- ▶ Generating a diagnostic plot of a fitted model
- ▶ Fitting a polynomial regression model with lm
- ▶ Fitting a robust linear regression model with rlm
- ▶ Studying a case of linear regression on SLID data
- ▶ Applying the Gaussian model for generalized linear regression
- ▶ Applying the Poisson model for generalized linear regression
- ▶ Applying the Binomial model for generalized linear regression
- ▶ Fitting a generalized additive model to data
- ▶ Visualizing a generalized additive model
- ▶ Diagnosing a generalized additive model

## Introduction

Regression is a supervised learning method, which is employed to model and analyze the relationship between a dependent (response) variable and one or more independent (predictor) variables. One can use regression to build a prediction model, which can first be used to find the best fitted model with minimum squared errors of the fitted values. The fitted model can then be further applied to data for continuous value predictions.

There are many types of regression. If there is only one predictor variable, and the relationship between the response variable and independent variable is linear, we can apply a linear model. However, if there is more than one predictor variable, a multiple linear regression method should be used. When the relationship is nonlinear, one can use a nonlinear model to model the relationship between the predictor and response variables.

In this chapter, we introduce how to fit a linear model into data with the `lm` function. Next, for distribution in other than the normal Gaussian model (for example, Poisson or Binomial), we use the `glm` function with an appropriate link function correspondent to the data distribution. Finally, we cover how to fit a generalized additive model into data using the `gam` function.

## Fitting a linear regression model with lm

The simplest model in regression is linear regression, which is best used when there is only one predictor variable, and the relationship between the response variable and the independent variable is linear. In R, we can fit a linear model to data with the `lm` function.

### Getting ready

We need to prepare data with one predictor and response variable, and with a linear relationship between the two variables.

### How to do it...

Perform the following steps to perform linear regression with `lm`:

1. You should install the `car` package and load its library:

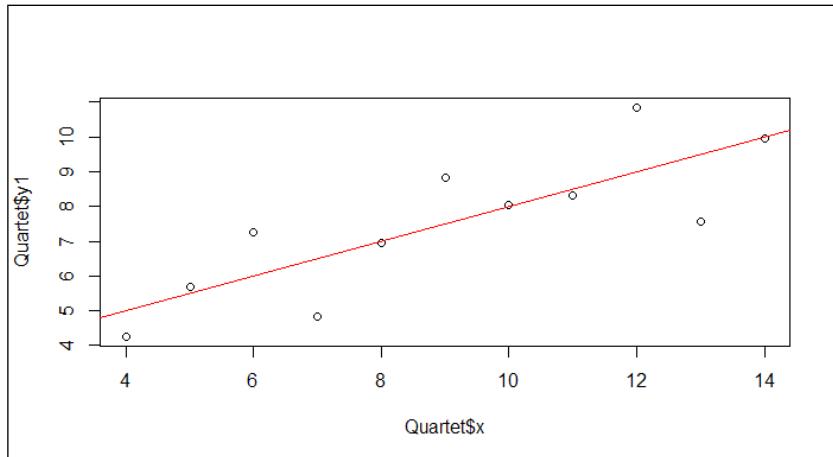
```
> install.packages("car")
> library(car)
```
2. From the package, you can load the `Quartet` dataset:

```
> data(Quartet)
```
3. You can use the `str` function to display the structure of the `Quartet` dataset:

```
> str(Quartet)
'data.frame': 11 obs. of 6 variables:
 $ x : int 10 8 13 9 11 14 6 4 12 7 ...
 $ y1: num 8.04 6.95 7.58 8.81 8.33 ...
 $ y2: num 9.14 8.14 8.74 8.77 9.26 8.1 6.13 3.1 9.13 7.26 ...
 $ y3: num 7.46 6.77 12.74 7.11 7.81 ...
 $ x4: int 8 8 8 8 8 8 19 8 8 ...
 $ y4: num 6.58 5.76 7.71 8.84 8.47 7.04 5.25 12.5 5.56 7.91 ...
```

4. Draw a scatter plot of the x and y variables with `plot`, and append a fitted line through the `lm` and `abline` function:

```
> plot(Quartet$x, Quartet$y1)
> lmfit = lm(y1~x, Quartet)
> abline(lmfit, col="red")
```



A simple regression plot fitted by lm

5. To view the fit model, execute the following:

```
> lmfit

Call:
lm(formula = y1 ~ x, data = Quartet)

Coefficients:
```

| (Intercept) | x      |
|-------------|--------|
| 3.0001      | 0.5001 |

## How it works...

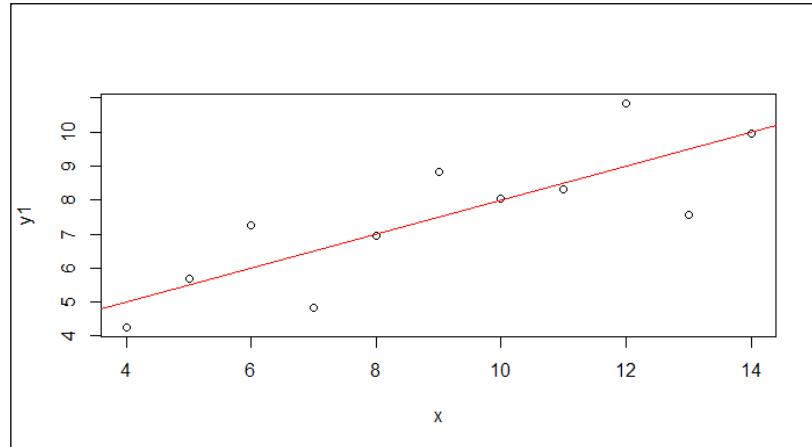
The regression model has the `response ~ terms` form, where `response` is the response vector, and `terms` is a series of terms that specifies a predictor. We can illustrate a simple regression model in the formula  $y = \alpha + \beta x$ , where  $\alpha$  is the intercept while the slope,  $\beta$ , describes the change in  $y$  when  $x$  changes. By using the least squares method, we can estimate  $\beta = \frac{\text{cov}[x, y]}{\text{var}[x]}$  and  $\alpha = \bar{y} - \beta \bar{x}$  (where  $\bar{y}$  indicates the mean value of  $y$  and  $\bar{x}$  denotes the mean value of  $x$ ).

To perform linear regression, we first prepare the data that has a linear relationship between the predictor variable and response variable. In this example, we load Anscombe's quartet dataset from the package car. Within the dataset, the x and y1 variables have a linear relationship, and we prepare a scatter plot of these variables. To generate the regression line, we use the lm function to generate a model of the two variables. Further, we use abline to plot a regression line on the plot. As per the previous screenshot, the regression line illustrates the linear relationship of x and y1 variables. We can see that the coefficient of the fitted model shows the intercept equals 3.0001 and coefficient equals 0.5001. As a result, we can use the intercept and coefficient to infer the response value. For example, we can infer the response value when x at 3 is equal to 4.5103 ( $3 * 0.5001 + 3.0001$ ).

## There's more...

Besides the lm function, you can also use the lsfit function to perform simple linear regression. For example:

```
> plot(Quartet$x, Quartet$y1)
> lmfit2 = lsfit(Quartet$x, Quartet$y1)
> abline(lmfit2, col="red")
```



A simple regression fitted by the lsfit function.

## Summarizing linear model fits

The summary function can be used to obtain the formatted coefficient, standard errors, degree of freedom, and other summarized information of a fitted model. This recipe introduces how to obtain overall information on a model through the use of the summary function.

## Getting ready

You need to have completed the previous recipe by computing the linear model of the x and y1 variables from the quartet, and have the fitted model assigned to the lmfit variable.

## How to do it...

Perform the following step to summarize linear model fits:

1. Compute a detailed summary of the fitted model:

```
> summary(lmfit)
```

Call:

```
lm(formula = y1 ~ x)
```

Residuals:

| Min      | 1Q       | Median   | 3Q      | Max     |
|----------|----------|----------|---------|---------|
| -1.92127 | -0.45577 | -0.04136 | 0.70941 | 1.83882 |

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t )   |
|-------------|----------|------------|---------|------------|
| (Intercept) | 3.0001   | 1.1247     | 2.667   | 0.02573 *  |
| Quartet\$x  | 0.5001   | 0.1179     | 4.241   | 0.00217 ** |
| ---         |          |            |         |            |

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

Residual standard error: 1.237 on 9 degrees of freedom

Multiple R-squared: 0.6665, Adjusted R-squared: 0.6295

F-statistic: 17.99 on 1 and 9 DF, p-value: 0.00217

## How it works...

The `summary` function is a generic function used to produce summary statistics. In this case, it computes and returns a list of the summary statistics of the fitted linear model. Here, it will output information such as residuals, coefficient standard error R-squared, f-statistic, and a degree of freedom. In the Call section, the function called to generate the fitted model is displayed. In the Residuals section, it provides a quick summary (min, 1Q, median, 3Q, max) of the distribution.

In the Coefficients section, each coefficient is a Gaussian random variable. Within this section, Estimate represents the mean distribution of the variable; Std. Error displays the standard error of the variable; the t value is Estimate divided by Std. Error and the p value indicates the probability of getting a value larger than the t value. In this sample, the p value of both intercepts (0.002573) and x (0.00217) have a 95 percent level of confidence.

Residual standard error outputs the standard deviation of residuals, while the degree of freedom indicates the differences between the observation in training samples and the number used in the model. Multiple R-squared is obtained by dividing the sum of squares. One can use R-squared to measure how close the data is to fit into the regression line. Mostly, the higher the R-squared, the better the model fits your data. However, it does not necessarily indicate whether the regression model is adequate. This means you might get a good model with a low R-squared or you can have a bad model with a high R-squared. Since multiple R-squared ignore a degree of freedom, the calculated score is biased. To make the calculation fair, an adjusted R-squared (0.6295) uses an unbiased estimate, and will be slightly less than multiple R-squared (0.6665). F-statistic is retrieved by performing an f-test on the model. A p value equal to 0.00217 (< 0.05) rejects the null hypothesis (no linear correlation between variables) and indicates that the observed F is greater than the critical F value. In other words, the result shows that there is a significant positive linear correlation between the variables.

## See also

- ▶ For more information on the parameters used for obtaining a summary of the fitted model, you can use the `help` function or `?lm` to view the help page:  

```
> ?summary.lm
```
- ▶ Alternatively, you can use the following functions to display the properties of the model:  

```
> coefficients(lmfit) # Extract model coefficients
> confint(lmfit, level=0.95) # Computes confidence intervals for
model parameters.
> fitted(lmfit) # Extract model fitted values
> residuals(lmfit) # Extract model residuals
> anova(lmfit) # Compute analysis of variance tables for fitted
model object
> vcov(lmfit) # Calculate variance-covariance matrix for a fitted
model object
> influence(lmfit) # Diagnose quality of regression fits
```

# Using linear regression to predict unknown values

With a fitted regression model, we can apply the model to predict unknown values. For regression models, we can express the precision of prediction with a prediction interval and a confidence interval. In the following recipe, we introduce how to predict unknown values under these two measurements.

## Getting ready

You need to have completed the previous recipe by computing the linear model of the `x` and `y1` variables from the `quartet` dataset.

## How to do it...

Perform the following steps to predict values with linear regression:

1. Fit a linear model with the `x` and `y1` variables:

```
> lmfit = lm(y1~x, Quartet)
```

2. Assign values to be predicted into `newdata`:

```
> newdata = data.frame(x = c(3,6,15))
```

3. Compute the prediction result using the confidence interval with `level` set as 0.95:

```
> predict(lmfit, newdata, interval="confidence", level=0.95)
```

|  | fit | lwr | upr |
|--|-----|-----|-----|
|--|-----|-----|-----|

|   |          |          |          |
|---|----------|----------|----------|
| 1 | 4.500364 | 2.691375 | 6.309352 |
|---|----------|----------|----------|

|   |          |          |          |
|---|----------|----------|----------|
| 2 | 6.000636 | 4.838027 | 7.163245 |
|---|----------|----------|----------|

|   |           |          |           |
|---|-----------|----------|-----------|
| 3 | 10.501455 | 8.692466 | 12.310443 |
|---|-----------|----------|-----------|

4. Compute the prediction result using this prediction interval:

```
> predict(lmfit, newdata, interval="predict")
```

|  | fit | lwr | upr |
|--|-----|-----|-----|
|--|-----|-----|-----|

|   |          |          |          |
|---|----------|----------|----------|
| 1 | 4.500364 | 1.169022 | 7.831705 |
|---|----------|----------|----------|

|   |          |          |          |
|---|----------|----------|----------|
| 2 | 6.000636 | 2.971271 | 9.030002 |
|---|----------|----------|----------|

|   |           |          |           |
|---|-----------|----------|-----------|
| 3 | 10.501455 | 7.170113 | 13.832796 |
|---|-----------|----------|-----------|

## How it works...

We first build a linear fitted model with  $x$  and  $y1$  variables. Next, we assign values to be predicted into a data frame, `newdata`. It is important to note that the generated model is in the form of  $y1 \sim x$ .

Next, we compute the prediction result using a confidence interval by specifying `confidence` in the argument interval. From the output of row 1, we get fitted  $y1$  of the  $x=3$  input, which equals to 4.500364, and a 95 percent confidence interval (set 0.95 in the `level` argument) of the  $y1$  mean for  $x=3$  is between 2.691375 and 6.309352. In addition to this, row 2 and 3 give the prediction result of  $y1$  with an input of  $x=6$  and  $x=15$ .

Next, we compute the prediction result using a prediction interval by specifying `prediction` in the argument interval. From the output of row 1, we can see fitted  $y1$  of the  $x=3$  input equals to 4.500364, and a 95 percent prediction interval of  $y1$  for  $x=3$  is between 1.169022 and 7.831705. Row 2 and 3 output the prediction result of  $y1$  with an input of  $x=6$  and  $x=15$ .

## See also

- ▶ For those who are interested in the differences between prediction intervals and confidence intervals, you can refer to the Wikipedia entry *contrast with confidence intervals* at [http://en.wikipedia.org/wiki/Prediction\\_interval#Contrast\\_with\\_confidence\\_intervals](http://en.wikipedia.org/wiki/Prediction_interval#Contrast_with_confidence_intervals).

## Generating a diagnostic plot of a fitted model

Diagnostics are methods to evaluate assumptions of the regression, which can be used to determine whether a fitted model adequately represents the data. In the following recipe, we introduce how to diagnose a regression model through the use of a diagnostic plot.

## Getting ready

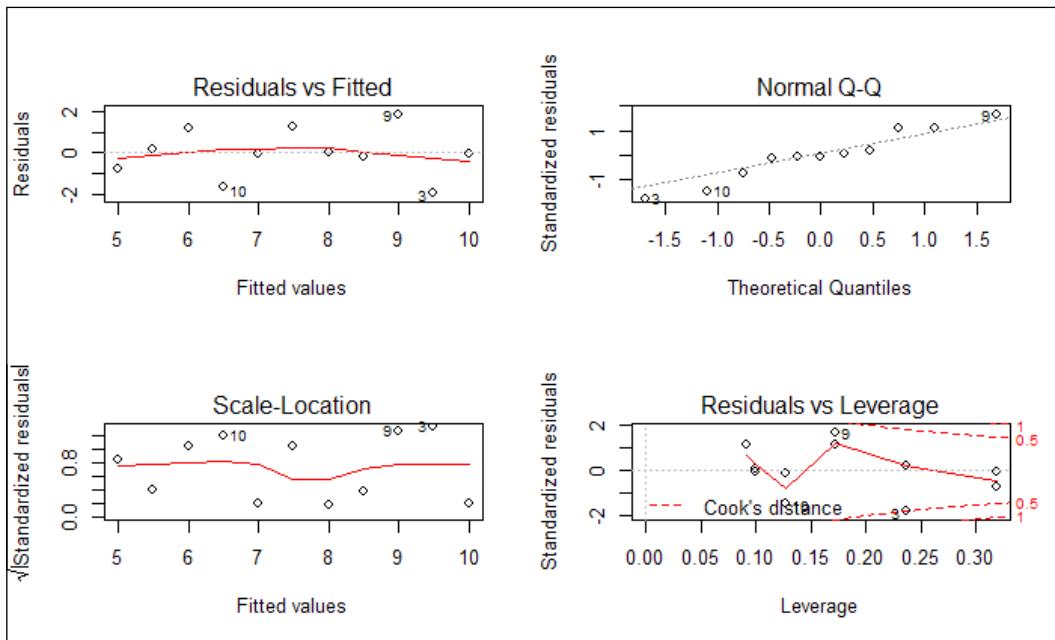
You need to have completed the previous recipe by computing a linear model of the  $x$  and  $y1$  variables from the quartet, and have the model assigned to the `lmfit` variable.

## How to do it...

Perform the following step to generate a diagnostic plot of the fitted model:

1. Plot the diagnostic plot of the regression model:

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



Diagnostic plots of the regression model

## How it works...

The `plot` function generates four diagnostic plots of a regression model:

- ▶ The upper-left plot shows residuals versus fitted values. Within the plot, residuals represent the vertical distance from a point to the regression line. If all points fall exactly on the regression line, all residuals will fall exactly on the dotted gray line. The red line within the plot is a smooth curve with regard to residuals, and if all the dots fall exactly on the regression line, the position of the red line should exactly match the dotted gray line.
- ▶ The upper-right shows the normal of residuals. This plot verifies the assumption that residuals were normally distributed. Thus, if the residuals were normally distributed, they should lie exactly on the gray dash line.

- ▶ The **Scale-Location** plot on the bottom-left is used to measure the square root of the standardized residuals against the fitted value. Therefore, if all dots lie on the regression line, the value of  $y$  should be close to zero. Since it is assumed that the variance of residuals does not change the distribution substantially, if the assumption is correct, the red line should be relatively flat.
- ▶ The bottom-right plot shows standardized residuals versus leverage. The leverage is a measurement of how each data point influences the regression. It is a measurement of the distance from the centroid of regression and level of isolation (measured by whether it has neighbors). Also, you can find the contour of Cook's distance, which is affected by high leverage and large residuals. You can use this to measure how regression would change if a single point is deleted. The red line is smooth with regard to standardized residuals. For a perfect fit regression, the red line should be close to the dashed line with no points over 0.5 in Cook's distance.

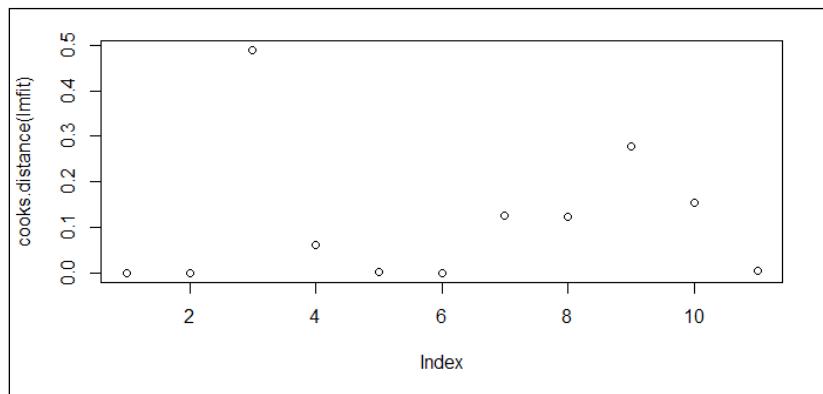
## There's more...

To see more of the diagnostic plot function, you can use the `help` function to access further information:

```
> ?plot.lm
```

In order to discover whether there are points with large Cook's distance, one can use the `cooks.distance` function to compute the Cook's distance of each point, and analyze the distribution of distance through visualization:

```
> plot(cooks.distance(lmfit))
```



A plot of Cook's distance

In this case, where the point on index 3 shows greater Cook's distance than other points, one can investigate whether this point might be an outlier.

# Fitting a polynomial regression model with lm

Some predictor variables and response variables may have a non-linear relationship, and their relationship can be modeled as an  $n$ th order polynomial. In this recipe, we introduce how to deal with polynomial regression using the `lm` and `poly` functions.

## Getting ready

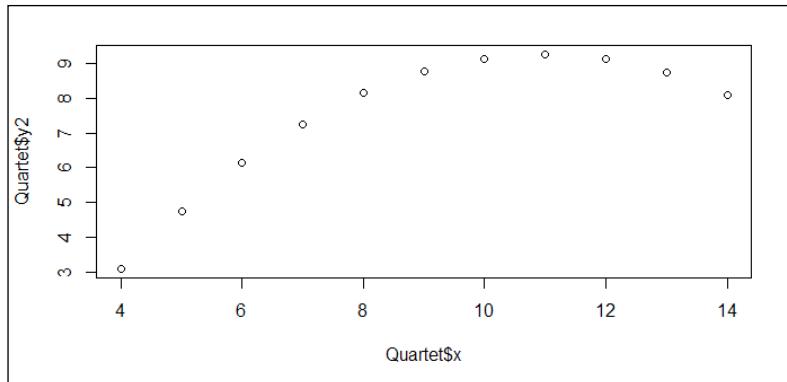
Prepare the dataset that includes a relationship between the predictor and response variable that can be modeled as an  $n$ th order polynomial. In this recipe, we will continue to use the Quartet dataset from the `car` package.

## How to do it...

Perform the following steps to fit the polynomial regression model with `lm`:

1. First, we make a scatter plot of the `x` and `y2` variables:

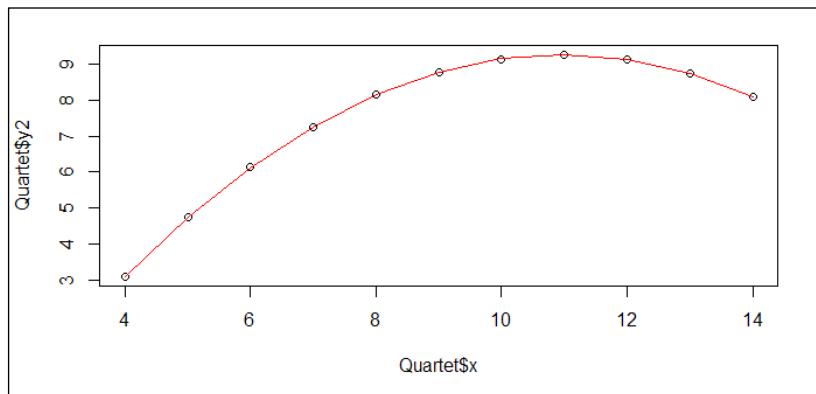
```
> plot(Quartet$x, Quartet$y2)
```



Scatter plot of variables x and y2

2. You can apply the `poly` function by specifying 2 in the argument:

```
> lmfit = lm(Quartet$y2~poly(Quartet$x, 2))
> lines(sort(Quartet$x), lmfit$fit[order(Quartet$x)], col = "red")
```



A quadratic fit example of the regression plot of variables x and y2

## How it works

We can illustrate the second order polynomial regression model in formula,  $y = \alpha + \beta x + \gamma x^2$ , where  $\alpha$  is the intercept while  $\beta$ ,  $\gamma$  illustrates regression coefficients.

In the preceding screenshot (step 1), the scatter plot of the `x` and `y2` variables does not fit in a linear relationship, but shows a concave downward curve (or convex upward) with the turning point at  $x=11$ . In order to model the nonlinear relationship, we apply the `poly` function with an argument of 2 to fit the independent `x` variable and the dependent `y2` variable. The red line in the screenshot shows that the model perfectly fits the data.

## There's more...

You can also fit a second order polynomial model with an independent variable equal to the formula of the combined first order `x` variable and the second order `x` variable:

```
> plot(Quartet$x, Quartet$y2)
> lmfit = lm(Quartet$y2 ~ I(Quartet$x) + I(Quartet$x^2))
```

# Fitting a robust linear regression model with rlm

An outlier in the dataset will move the regression line away from the mainstream. Apart from removing it, we can apply a robust linear regression to fit datasets containing outliers. In this recipe, we introduce how to apply `rlm` to perform robust linear regression to datasets containing outliers.

## Getting ready

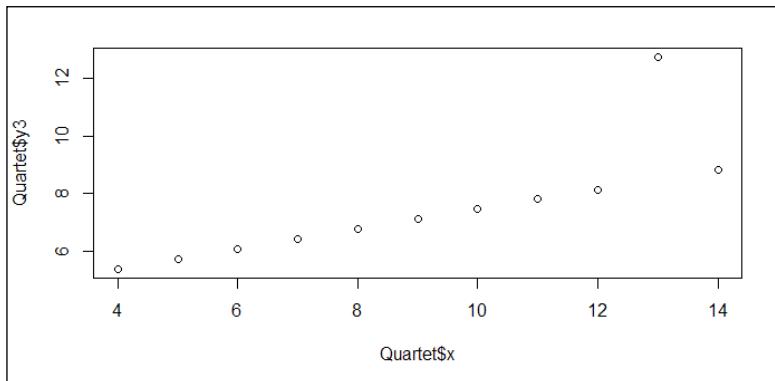
Prepare the dataset that contains an outlier that may move the regression line away from the mainstream. Here, we use the `Quartet` dataset loaded from the previous recipe.

## How to do it...

Perform the following steps to fit the robust linear regression model with `rlm`:

1. Generate a scatter plot of the `x` variable against `y3`:

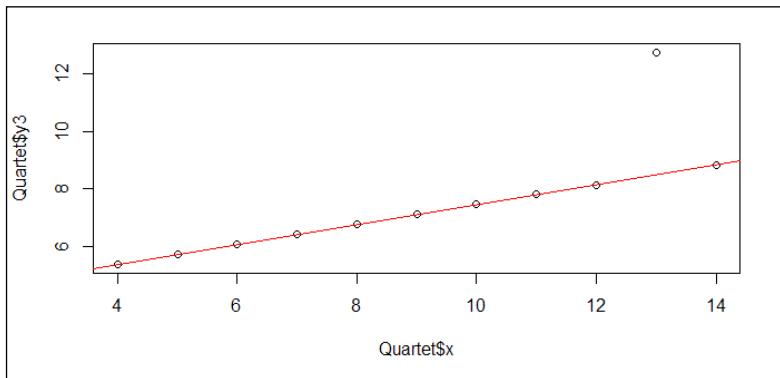
```
> plot(Quartet$x, Quartet$y3)
```



Scatter plot of variables x and y3

2. Next, you should import the MASS library first. Then, you can apply the `rlm` function to fit the model, and visualize the fitted line with the `abline` function:

```
> library(MASS)
> lmfit = rlm(Quartet$y3~Quartet$x)
> abline(lmfit, col="red")
```



Robust linear regression to variables x and y3

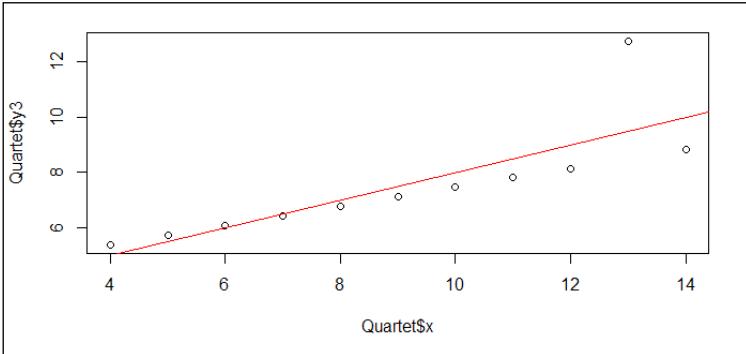
## How it works

As per the preceding screenshot (step 1), you may encounter datasets that include outliers away from the mainstream. To remove the effect of an outlier, we demonstrate how to apply a robust linear regression (`rlm`) to fit the data. In the second screenshot (step 2), the robust regression line ignores the outlier and matches the mainstream.

## There's more...

To see the effect of how an outlier can move the regression line away from the mainstream, you may replace the `rlm` function used in this recipe to `lm`, and replot the graph:

```
> plot(Quartet$x, Quartet$y3)
> lmfit = lm(Quartet$y3~Quartet$x)
> abline(lmfit, col="red")
```



Linear regression on variables x and y3

It is obvious that outlier ( $x=13$ ) moves the regression line away from the mainstream.

## Studying a case of linear regression on SLID data

To summarize the contents of the previous section, we explore more complex data with linear regression. In this recipe, we demonstrate how to apply linear regression to analyze the **Survey of Labor and Income Dynamics (SLID)** dataset.

### Getting ready

Check whether the `car` library is installed and loaded, as it is required to access the dataset SLID.

### How to do it...

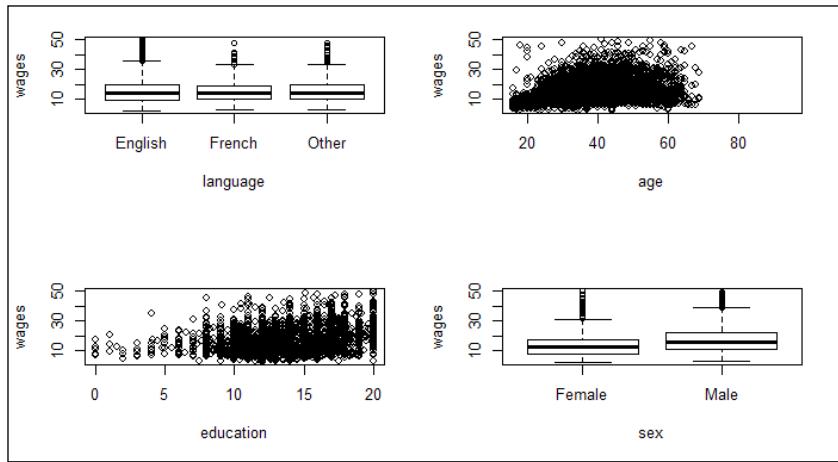
Follow these steps to perform linear regression on SLID data:

1. You can use the `str` function to get an overview of the data:

```
> str(SLID)
'data.frame': 7425 obs. of 5 variables:
 $ wages : num 10.6 11 NA 17.8 NA ...
 $ education: num 15 13.2 16 14 8 16 12 14.5 15 10 ...
 $ age : int 40 19 49 46 71 50 70 42 31 56 ...
 $ sex : Factor w/ 2 levels "Female","Male": 2 2 2 2 2 1 1 1
2 1 ...
 $ language : Factor w/ 3 levels "English","French",...: 1 1 3 3 1
1 1 1 1 1 ..
```

2. First, we visualize the variable wages against language, age, education, and sex:

```
> par(mfrow=c(2, 2))
> plot(SLID$wages ~ SLID$language)
> plot(SLID$wages ~ SLID$age)
> plot(SLID$wages ~ SLID$education)
> plot(SLID$wages ~ SLID$sex)
```



Plot of wages against multiple combinations

3. Then, we can use `lm` to fit the model:

```
> lmfit = lm(wages ~ ., data = SLID)
```

4. You can examine the summary of the fitted model through the `summary` function:

```
> summary(lmfit)
```

**Call:**

```
lm(formula = wages ~ ., data = SLID)
```

**Residuals:**

| Min     | 1Q     | Median | 3Q    | Max    |
|---------|--------|--------|-------|--------|
| -26.062 | -4.347 | -0.797 | 3.237 | 35.908 |

**Coefficients:**

|             | Estimate  | Std. Error | t value | Pr(> t )   |
|-------------|-----------|------------|---------|------------|
| (Intercept) | -7.888779 | 0.612263   | -12.885 | <2e-16 *** |
| education   | 0.916614  | 0.034762   | 26.368  | <2e-16 *** |

```
age 0.255137 0.008714 29.278 <2e-16 ***
sexMale 3.455411 0.209195 16.518 <2e-16 ***
languageFrench -0.015223 0.426732 -0.036 0.972
languageOther 0.142605 0.325058 0.439 0.661

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

Residual standard error: 6.6 on 3981 degrees of freedom  
(3438 observations deleted due to missingness)

Multiple R-squared: 0.2973, Adjusted R-squared: 0.2964

F-statistic: 336.8 on 5 and 3981 DF, p-value: < 2.2e-16

5. Drop the language attribute, and refit the model with the lm function:

```
> lmfit = lm(wages ~ age + sex + education, data = SLID)
> summary(lmfit)
```

Call:

```
lm(formula = wages ~ age + sex + education, data = SLID)
```

Residuals:

| Min     | 1Q     | Median | 3Q    | Max    |
|---------|--------|--------|-------|--------|
| -26.111 | -4.328 | -0.792 | 3.243 | 35.892 |

Coefficients:

|             | Estimate  | Std. Error | t value | Pr(> t )   |
|-------------|-----------|------------|---------|------------|
| (Intercept) | -7.905243 | 0.607771   | -13.01  | <2e-16 *** |
| age         | 0.255101  | 0.008634   | 29.55   | <2e-16 *** |
| sexMale     | 3.465251  | 0.208494   | 16.62   | <2e-16 *** |
| education   | 0.918735  | 0.034514   | 26.62   | <2e-16 *** |

---

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

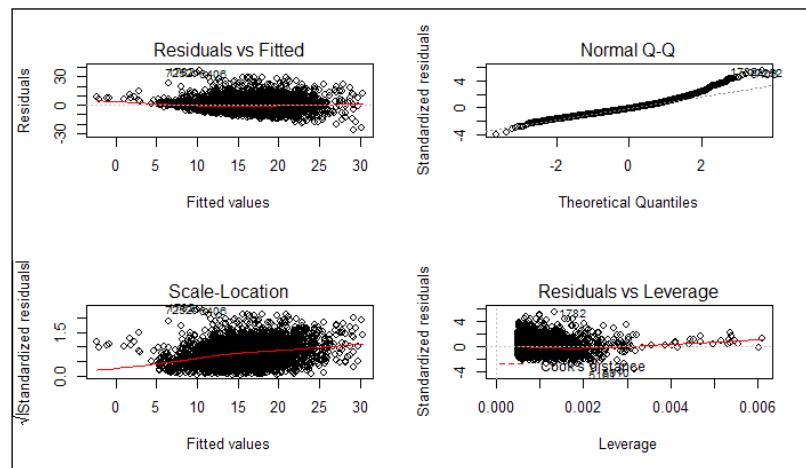
Residual standard error: 6.602 on 4010 degrees of freedom  
(3411 observations deleted due to missingness)

Multiple R-squared: 0.2972, Adjusted R-squared: 0.2967

F-statistic: 565.3 on 3 and 4010 DF, p-value: < 2.2e-16

6. We can then draw a diagnostic plot of lmfit:

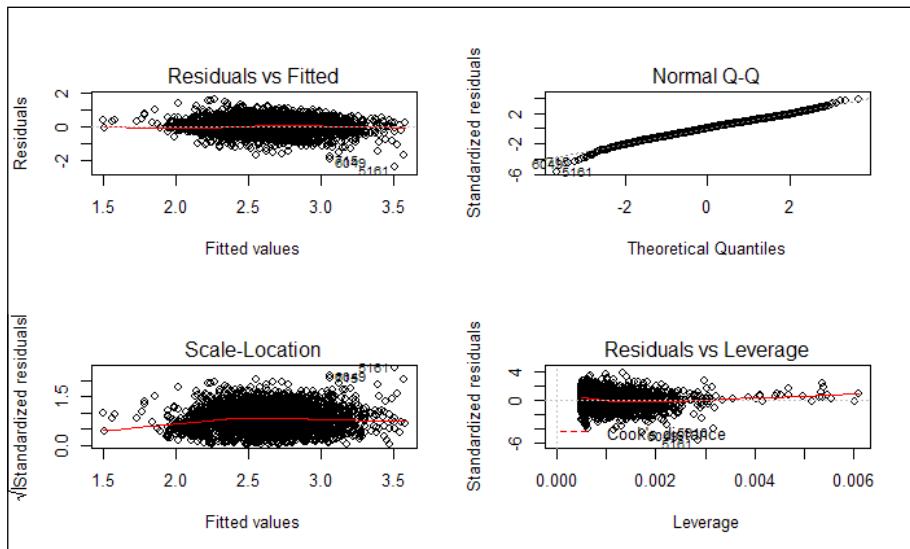
```
> par(mfrow=c(2,2))
> plot(lmfit)
```



Diagnostic plot of fitted model

7. Next, we take the log of wages and replot the diagnostic plot:

```
> lmfit = lm(log(wages) ~ age + sex + education, data = SLID)
> plot(lmfit)
```



Diagnostic plot of adjusted fitted model

8. Next, you can diagnose the multi-collinearity of the regression model using the `vif` function:

```
> vif(lmfit)
 age sex education
1.011613 1.000834 1.012179
> sqrt(vif(lmfit)) > 2
 age sex education
 FALSE FALSE FALSE
```

9. Then, you can install and load the `lmtest` package and diagnose the heteroscedasticity of the regression model with the `bptest` function:

```
> install.packages("lmtest")
> library(lmtest)
> bptest(lmfit)
```

studentized Breusch-Pagan test

```
data: lmfit
BP = 29.0311, df = 3, p-value = 2.206e-06
```

10. Finally, you can install and load the `rms` package. Then, you can correct standard errors with `robcov`:

```
> install.packages("rms")
> library(rms)
> olsfit = ols(log(wages) ~ age + sex + education, data= SLID, x=TRUE, y= TRUE)
> robcov(olsfit)
```

Linear Regression Model

```
ols(formula = log(wages) ~ age + sex + education, data = SLID,
x = TRUE, y = TRUE)
```

Frequencies of Missing Values Due to Each Variable

| log(wages) | age | sex | education |
|------------|-----|-----|-----------|
| 3278       | 0   | 0   | 249       |

|       |        | Model Likelihood |        | Discrimination |        |       |
|-------|--------|------------------|--------|----------------|--------|-------|
|       |        | Ratio Test       |        | Indexes        |        |       |
| Obs   | 4014   | LR               | chi2   | 1486.08        | R2     | 0.309 |
| sigma | 0.4187 | d.f.             |        | 3              | R2 adj | 0.309 |
| d.f.  | 4010   | Pr(> chi2)       | 0.0000 | g              |        | 0.315 |

### Residuals

| Min      | 1Q       | Median  | 3Q      | Max     |
|----------|----------|---------|---------|---------|
| -2.36252 | -0.27716 | 0.01428 | 0.28625 | 1.56588 |

|           | Coef   | S.E.   | t     | Pr(> t ) |
|-----------|--------|--------|-------|----------|
| Intercept | 1.1169 | 0.0387 | 28.90 | <0.0001  |
| age       | 0.0176 | 0.0006 | 30.15 | <0.0001  |
| sex=Male  | 0.2244 | 0.0132 | 16.96 | <0.0001  |
| education | 0.0552 | 0.0022 | 24.82 | <0.0001  |

## How it works...

This recipe demonstrates how to conduct linear regression analysis on the SLID dataset. First, we load the SLID data and display its structure through the use of the `str` function. From the structure of the data, we know that there are four independent variables that will affect the wages of the dependent variable.

Next, we explore the relationship of each independent variable to the dependent variable, wages, through visualization; the visualization result is shown in the preceding screenshot (step 2). In the upper-left section of this screenshot, you can find the box plot of three different languages against wages; the correlation between the languages and wages is not obvious. The upper-right section of the screenshot shows that the age appears to have a positive relationship with the dependent variable, wages. In the bottom-left of the screenshot, it is shown that education also appears to have a positive relationship with wages. Finally, the box plot in the bottom-right section of the screenshot shows that the wages of males are slightly higher than females.

Next, we fit all the attributes except for wages to the model as predictor variables. By summarizing the model, it is shown that education, age, and sex show a significance (*p-value* < 0.05). As a result, we drop the insignificant language attribute (which has a *p-value* greater than 0.05) and fit the three independent variables (education, sex, and age) with regard to the dependent variable (wages) in the linear model. This accordingly raises the f-statistic from 336.8 to 565.3.

Next, we generate the diagnostic plot of the fitted model. Within the diagnostic plot, all the four plots indicate that the regression model follows the regression assumption. However, from residuals versus fitted and scale-location plot, residuals of smaller fitted values are biased toward the regression model. Since wages range over several orders of magnitude, to induce the symmetry, we apply a log transformation to wages and refit the data into a regression model. The red line of residuals versus fitted values plot and the Scale-Location plot are now closer to the gray dashed line.

Next, we would like to test whether multi-collinearity exists in the model. Multi-collinearity takes place when a predictor is highly correlated with others. If multi-collinearity exists in the model, you might see some variables have a high R-squared value but are shown as variables insignificant. To detect multi-collinearity, we can calculate the variance inflation and generalized variance inflation factors for linear and generalized linear models with the `vif` function. If multi-collinearity exists, we should find predictors with the square root of variance inflation factor above 2. Then, we may remove redundant predictors or use a principal component analysis to transform predictors to a smaller set of uncorrelated components.

Finally, we would like to test whether **heteroscedasticity** exists in the model. Before discussing the definition of heteroscedasticity, we first have to know that in classic assumptions, the ordinary regression model assumes that the variance of the error is constant or homogeneous across observations. On the contrary, heteroscedasticity means that the variance is unequal across observations. As a result, heteroscedasticity may be biased toward the standard errors of our estimates and, therefore, mislead the testing of the hypotheses. To detect and test heteroscedasticity, we can perform the **Breusch-Pagan** test for heteroscedasticity with the `bptest` function within the `lmtest` package. In this case, the p-value shows 2.206e-06 (<0.5), which rejects the null hypothesis of homoscedasticity (no heteroscedasticity). Here, it implies that the standard errors of the parameter estimates are incorrect. However, we can use robust standard errors to correct the standard error (do not remove the heteroscedasticity) and increase the significance of truly significant parameters with `robcov` from the `rms` package. However, since it only takes the fitted model from the `rms` series as an input, we have to fit the ordinary least squares model beforehand.

## See also

- ▶ For more information about the SLID dataset, you can use the `help` function to view the related documentation:
  - > `?SLID`

# Applying the Gaussian model for generalized linear regression

**Generalized linear model (GLM)** is a generalization of linear regression, which can include a link function to make a linear prediction. As a default setting, the family object for `glm` is Gaussian, which makes the `glm` function perform exactly the same as `lm`. In this recipe, we first demonstrate how to fit the model into the data using the `glm` function, and then show that `glm` with a Gaussian model performs exactly the same as `lm`.

## Getting ready

Check whether the `car` library is installed and loaded as we require the SLID dataset from this package.

## How to do it...

Perform the following steps to fit a generalized linear regression model with the Gaussian model:

1. Fit the independent variables, age, sex, and education, and dependent variable wages to `glm`:

```
> lmfit1 = glm(wages ~ age + sex + education, data = SLID,
family=gaussian)
> summary(lmfit1)
```

Call:

```
glm(formula = wages ~ age + sex + education, family = gaussian,
data = SLID)
```

Deviance Residuals:

| Min     | 1Q     | Median | 3Q    | Max    |
|---------|--------|--------|-------|--------|
| -26.111 | -4.328 | -0.792 | 3.243 | 35.892 |

Coefficients:

|             | Estimate  | Std. Error | t value | Pr(> t )   |
|-------------|-----------|------------|---------|------------|
| (Intercept) | -7.905243 | 0.607771   | -13.01  | <2e-16 *** |
| age         | 0.255101  | 0.008634   | 29.55   | <2e-16 *** |
| sexMale     | 3.465251  | 0.208494   | 16.62   | <2e-16 *** |

```
education 0.918735 0.034514 26.62 <2e-16 ***

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

(Dispersion parameter for Gaussian family taken to be 43.58492)

Null deviance: 248686 on 4013 degrees of freedom
Residual deviance: 174776 on 4010 degrees of freedom
(3411 observations deleted due to missingness)
AIC: 26549
```

Number of Fisher Scoring iterations: 2

2. Fit the independent variables, age, sex, and education, and the dependent variable wages to lm:

```
> lmfit2 = lm(wages ~ age + sex + education, data = SLID)
> summary(lmfit2)
```

Call:

```
lm(formula = wages ~ age + sex + education, data = SLID)
```

Residuals:

| Min     | 1Q     | Median | 3Q    | Max    |
|---------|--------|--------|-------|--------|
| -26.111 | -4.328 | -0.792 | 3.243 | 35.892 |

Coefficients:

|             | Estimate  | Std. Error | t value | Pr(> t )   |
|-------------|-----------|------------|---------|------------|
| (Intercept) | -7.905243 | 0.607771   | -13.01  | <2e-16 *** |
| age         | 0.255101  | 0.008634   | 29.55   | <2e-16 *** |
| sexMale     | 3.465251  | 0.208494   | 16.62   | <2e-16 *** |
| education   | 0.918735  | 0.034514   | 26.62   | <2e-16 *** |

---

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

Residual standard error: 6.602 on 4010 degrees of freedom  
(3411 observations deleted due to missingness)

Multiple R-squared: 0.2972, Adjusted R-squared: 0.2967  
F-statistic: 565.3 on 3 and 4010 DF, p-value: < 2.2e-16

3. Use anova to compare the two fitted models:

```
> anova(lmfit1, lmfit2)
```

#### Analysis of Deviance Table

Model: gaussian, link: identity

Response: wages

Terms added sequentially (first to last)

|           | Df | Deviance | Resid. Df | Resid. Dev |
|-----------|----|----------|-----------|------------|
| NULL      |    |          | 4013      | 248686     |
| age       | 1  | 31953    | 4012      | 216733     |
| sex       | 1  | 11074    | 4011      | 205659     |
| education | 1  | 30883    | 4010      | 174776     |

## How it works...

The `glm` function fits a model to the data in a similar fashion to the `lm` function. The only difference is that you can specify a different link function in the parameter, `family` (you may use `?family` in the console to find different types of link functions). In this recipe, we first input the independent variables, `age`, `sex`, and `education`, and the dependent `wages` variable to the `glm` function, and assign the built model to `lmfit1`. You can use the built model for further prediction.

Next, to determine whether `glm` with a Gaussian model is exactly the same as `lm`, we fit the independent variables, `age`, `sex`, and `education`, and the dependent variable, `wages`, to the `lm` model. By applying the `summary` function to the two different models, it reveals that the residuals and coefficients of the two output summaries are exactly the same.

Finally, we further compare the two fitted models with the `anova` function. The result of the `anova` function shows that the two models are similar, with the same **residual degrees of freedom (Res.DF)** and **residual sum of squares (RSS Df)**.

## See also

- ▶ For a comparison of generalized linear models with linear models, you can refer to Venables, W. N., & Ripley, B. D. (2002). *Modern applied statistics with S*. Springer.

# Applying the Poisson model for generalized linear regression

Generalized linear models allow response variables that have error distribution models other than a normal distribution (Gaussian). In this recipe, we demonstrate how to apply Poisson as a family object within `glm` with regard to count data.

## Getting ready

The prerequisite of this task is to prepare the count data, with all the input data values as integers.

## How to do it...

Perform the following steps to fit the generalized linear regression model with the Poisson model:

1. Load the `warpbreaks` data, and use `head` to view the first few lines:

```
> data(warpbreaks)
> head(warpbreaks)
 breaks wool tension
1 26 A L
2 30 A L
3 54 A L
4 25 A L
5 70 A L
6 52 A L
```

2. We apply Poisson as a family object for the independent variable, `tension`, and the dependent variable, `breaks`:

```
> rsl = glm(breaks ~ tension, data=warpbreaks, family="poisson")
> summary(rsl)
```

Call:

```
glm(formula = breaks ~ tension, family = "poisson", data =
warpbreaks)
```

Deviance Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----|----|--------|----|-----|
|-----|----|--------|----|-----|

-4.2464 -1.6031 -0.5872 1.2813 4.9366

Coefficients:

|                | Estimate | Std. Error | z value | Pr(> z )     |
|----------------|----------|------------|---------|--------------|
| (Intercept)    | 3.59426  | 0.03907    | 91.988  | < 2e-16 ***  |
| tensionM       | -0.32132 | 0.06027    | -5.332  | 9.73e-08 *** |
| tensionH       | -0.51849 | 0.06396    | -8.107  | 5.21e-16 *** |
| ---            |          |            |         |              |
| Signif. codes: | 0 ****   | 0.001 ***  | 0.01 ** | 0.05 *       |
|                | 0.1 .    | 0.1 .      | 0.1 .   | 1            |

(Dispersion parameter for Poisson family taken to be 1)

Null deviance: 297.37 on 53 degrees of freedom  
Residual deviance: 226.43 on 51 degrees of freedom  
AIC: 507.09

Number of Fisher Scoring iterations: 4

## How it works...

Under the assumption of a Poisson distribution, the count data can be fitted to a log-linear model. In this recipe, we first loaded a sample count data from the `warpbreaks` dataset, which contained data regarding the number of warp breaks per loom. Next, we applied the `glm` function with `breaks` as a dependent variable, `tension` as an independent variable, and Poisson as a family object. Finally, we viewed the fitted log-linear model with the `summary` function.

## See also

- ▶ To understand more on how a Poisson model is related to count data, you can refer to Cameron, A. C., & Trivedi, P. K. (2013). *Regression analysis of count data* (No. 53). Cambridge university press.

## Applying the Binomial model for generalized linear regression

For a binary dependent variable, one may apply a binomial model as the family object in the `glm` function.

# Getting ready

The prerequisite of this task is to prepare a binary dependent variable. Here, we use the `vs` variable (V engine or straight engine) as the dependent variable.

## How to do it...

Perform the following steps to fit a generalized linear regression model with the Binomial model:

1. First, we examine the first six elements of `vs` within `mtcars`:

```
> head(mtcars$vs)
[1] 0 0 1 1 0 1
```

2. We apply the `glm` function with `binomial` as the family object:

```
> lm1 = glm(vs ~ hp+mpg+gear, data=mtcars, family=binomial)
> summary(lm1)
```

Call:

```
glm(formula = vs ~ hp + mpg + gear, family = binomial, data =
mtcars)
```

Deviance Residuals:

| Min      | 1Q       | Median   | 3Q      | Max     |
|----------|----------|----------|---------|---------|
| -1.68166 | -0.23743 | -0.00945 | 0.30884 | 1.55688 |

Coefficients:

|                | Estimate | Std. Error | z value | Pr(> z ) |
|----------------|----------|------------|---------|----------|
| (Intercept)    | 11.95183 | 8.00322    | 1.493   | 0.1353   |
| hp             | -0.07322 | 0.03440    | -2.129  | 0.0333 * |
| mpg            | 0.16051  | 0.27538    | 0.583   | 0.5600   |
| gear           | -1.66526 | 1.76407    | -0.944  | 0.3452   |
| ---            |          |            |         |          |
| Signif. codes: | 0 ****   | 0.001 ***  | 0.01 ** | 0.05 *.  |

(Dispersion parameter for binomial family taken to be 1)

```
Null deviance: 43.860 on 31 degrees of freedom
Residual deviance: 15.651 on 28 degrees of freedom
AIC: 23.651
```

Number of Fisher Scoring iterations: 7

## How it works...

Within the binary data, each observation of the response value is coded as either 0 or 1. Fitting into the regression model of the binary data requires a binomial distribution function. In this example, we first load the binary dependent variable, `vs`, from the `mtcars` dataset. The `vs` is suitable for the binomial model as it contains binary data. Next, we fit the model into the binary data using the `glm` function by specifying `binomial` as the family object. Last, by referring to the summary, we can obtain the description of the fitted model.

## See also

- ▶ If you specify the family object in parameters only, you will use the default link to fit the model. However, to use an alternative link function, you can add a link argument. For example:

```
> lml = glm(vs ~ hp+mpg+gear,data=mtcars,
 family=binomial(link="probit"))
```
- ▶ If you would like to know how many alternative links you can use, please refer to the family document via the help function:

```
> ?family
```

## Fitting a generalized additive model to data

**Generalized additive model (GAM)**, which is used to fit generalized additive models, can be viewed as a semiparametric extension of GLM. While GLM holds the assumption that there is a linear relationship between dependent and independent variables, GAM fits the model on account of the local behavior of data. As a result, GAM has the ability to deal with highly nonlinear relationships between dependent and independent variables. In the following recipe, we introduce how to fit regression using a generalized additive model.

## Getting ready

We need to prepare a data frame containing variables, where one of the variables is a response variable and the others may be predictor variables.

## How to do it...

Perform the following steps to fit a generalized additive model into data:

1. First, load the mgcv package, which contains the gam function:

```
> install.packages("mgcv")
> library(mgcv)
```

2. Then, install the MASS package and load the Boston dataset:

```
> install.packages("MASS")
> library(MASS)
> attach(Boston)
> str(Boston)
```

3. Fit the regression using gam:

```
> fit = gam(dis ~ s(nox))
```

4. Get the summary information of the fitted model:

```
> summary(fit)
```

Family: gaussian

Link function: identity

Formula:

dis ~ s(nox)

Parametric coefficients:

|                | Estimate | Std. Error | t value | Pr(> t )   |
|----------------|----------|------------|---------|------------|
| (Intercept)    | 3.79504  | 0.04507    | 84.21   | <2e-16 *** |
| ---            |          |            |         |            |
| Signif. codes: | 0 ****   | 0.001 ***  | 0.01 ** | 0.05 *.    |

Approximate significance of smooth terms:

|                | edf    | Ref.df    | F       | p-value    |
|----------------|--------|-----------|---------|------------|
| s(nox)         | 8.434  | 8.893     | 189     | <2e-16 *** |
| ---            |        |           |         |            |
| Signif. codes: | 0 **** | 0.001 *** | 0.01 ** | 0.05 *.    |

R-sq.(adj) = 0.768 Deviance explained = 77.2%
GCV = 1.0472 Scale est. = 1.0277 n = 506

## How it works

GAM is designed to maximize the prediction of a dependent variable,  $y$ , from various distributions by estimating the nonparametric functions of the predictors that link to the dependent variable through a link function. The notion of GAM is  $g(E(y)) = \beta + f_1(x_1) + f_2(x_2) + \dots + f_n(x_n)$ , where an exponential family,  $E$ , is specified for  $y$ , along with the  $g$  link function;  $f$  denotes the link function of predictors.

The `gam` function is contained in the `mgcv` package, so, install this package first and load it into an R session. Next, load the Boston dataset (*Housing Values in the Suburbs of Boston*) from the `MASS` package. From the dataset, we use `dis` (the weighted mean of the distance to five Boston employment centers) as the dependent variable, and `nox` (nitrogen oxide concentration) as the independent variable, and then input them into the `gam` function to generate a fitted model.

Similar to `glm`, `gam` allows users to summarize the `gam` fit. From the summary, one can find the parametric parameter, significance of smoothed terms, and other useful information.

## See also

- ▶ Apart from `gam`, the `mgcv` package provides another generalized additive model, `bam`, for large datasets. The `bam` package is very similar to `gam`, but uses less memory and is relatively more efficient. Please use the `help` function for more information on this model:  
`> ? bam`
- ▶ For more information about generalized additive models in R, please refer to Wood, S. (2006). *Generalized additive models: an introduction with R*. CRC press.

## Visualizing a generalized additive model

In this recipe, we demonstrate how to add a `gam` fitted regression line to a scatter plot. In addition, we visualize the `gam` fit using the `plot` function.

## Getting ready

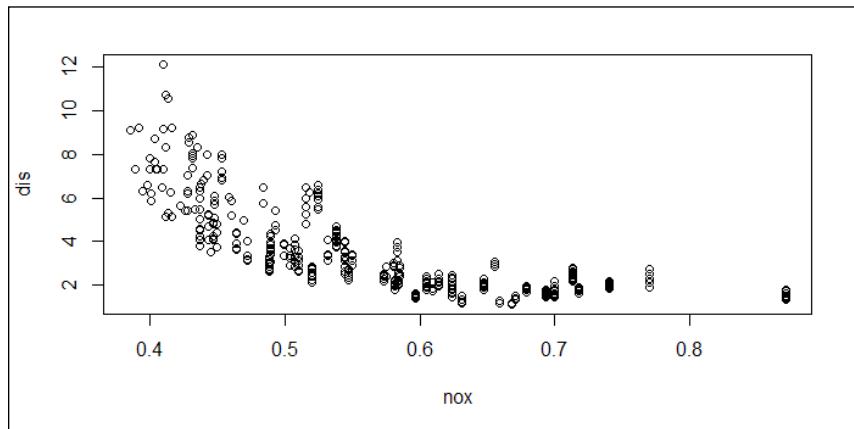
Complete the previous recipe by assigning a `gam` fitted model to the `fit` variable.

## How to do it...

Perform the following steps to visualize the generalized additive model:

1. Generate a scatter plot using the nox and dis variables:

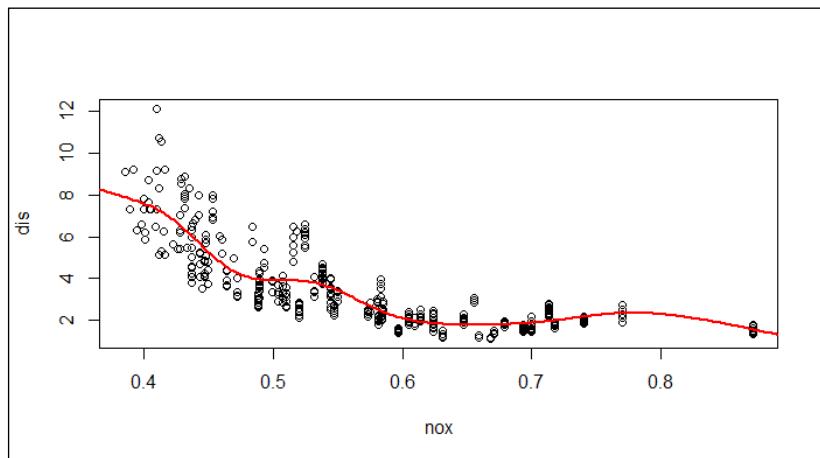
```
> plot(nox, dis)
```



Scatter plot of variable nox against dis

2. Add the regression to the scatter plot:

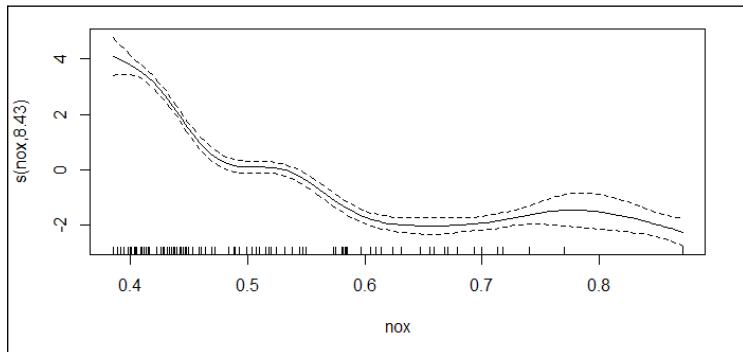
```
> x = seq(0, 1, length = 500)
> y = predict(fit, data.frame(nox = x))
> lines(x, y, col = "red", lwd = 2)
```



Fitted regression of gam on a scatter plot

3. Alternatively, you can plot the fitted model using the `plot` function:

```
> plot(fit)
```



Plot of fitted gam

## How it works...

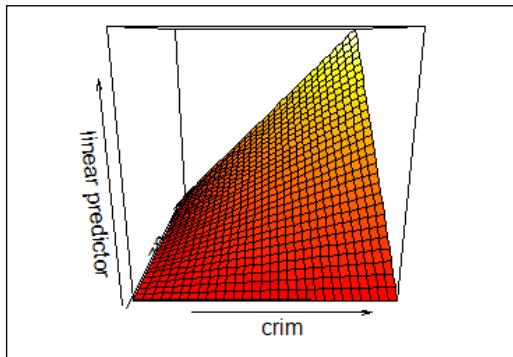
To visualize the fitted regression, we first generate a scatter plot using the `dis` and `nox` variables. Then, we generate the sequence of x-axis, and respond `y` through the use of the `predict` function on the fitted model, `fit`. Finally, we use the `lines` function to add the regression line to the scatter plot.

Besides using the `lines` to add fitted regression lines on the scatter plot, `gam` has a `plot` function to visualize the fitted regression lines containing the confidence region. To shade the confidence region, we assign `shade = TRUE` within the function.

## There's more...

The `vis.gam` function is used to produce perspective or contour plot views of the `gam` model predictions. It is helpful to observe how response variables interact with two predictor variables. The following is an example of a contour plot on the `Boston` dataset:

```
> fit2=gam(medv~crim+zn+crim:zn, data=Boston)
> vis.gam(fit2)
```



A sample contour plot produced by vis.gam

## Diagnosing a generalized additive model

GAM also provides diagnostic information about the fitting procedure and results of the generalized additive model. In this recipe, we demonstrate how to plot diagnostic plots through the `gam.check` function.

### Getting ready

Ensure that the previous recipe is completed with the `gam` fitted model assigned to the `fit` variable.

### How to do it...

Perform the following step to diagnose the generalized additive model:

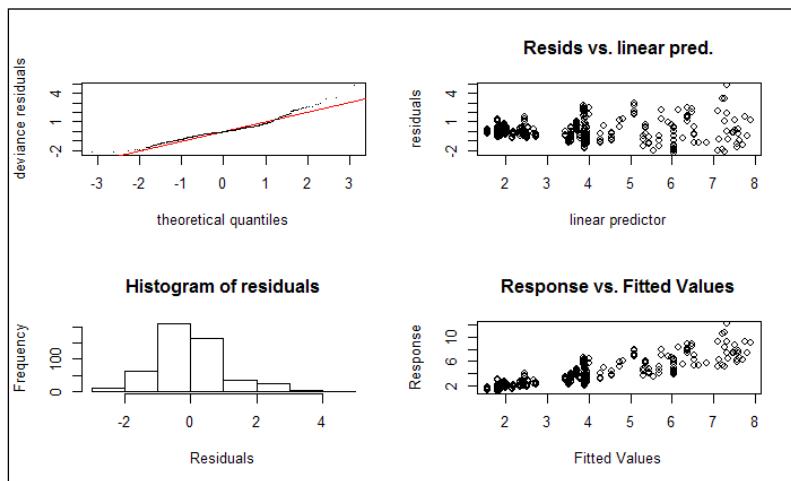
1. Generate the diagnostic plot using `gam.check` on the fitted model:

```
> gam.check(fit)
```

```
Method: GCV Optimizer: magic
Smoothing parameter selection converged after 7 iterations.
The RMS GCV score gradient at convergence was 8.79622e-06 .
The Hessian was positive definite.
The estimated model rank was 10 (maximum possible: 10)
Model rank = 10 / 10
```

Basis dimension (k) checking results. Low p-value (k-index<1) may indicate that k is too low, especially if edf is close to k'.

|         | k'    | edf   | k-index | p-value |
|---------|-------|-------|---------|---------|
| s (nox) | 9.000 | 8.434 | 0.397   | 0       |



Diagnostic plot of fitted gam

## How it works...

The `gam.check` function first produces the smoothing parameter estimation convergence information. In this example, the smoothing parameter, **GCV/UBRE (Generalized Cross Validation/ Unbiased Risk Estimator)** score converges after seven iterations. The mean absolute gradient of the GCV/UBRE function at the minimum is 8.79622e-06 and the estimated rank is 10. The dimension check is to test whether the basis dimension for a smooth function is adequate. From this example, the low p-value indicates that the k is set too low. One may adjust the dimension choice for smooth by specifying the argument, `k`, by fitting `gam` to the data.

In addition to providing information regarding smoothing parameter estimation convergence, the function returns four diagnostic plots. The upper-left section of the plot in the screenshot shows a **quantile-comparison** plot. This plot is useful to identify outliers and heavy tails. The upper-right section of the plot shows residuals versus linear predictors, which are useful in finding nonconstant error variances. The bottom-left section of the plot shows a histogram of the residuals, which is helpful in detecting non-normality. The bottom-right section shows response versus the fitted value.

## There's more...

You can access the `help` function for more information on `gam.check`. In particular, this includes a detailed illustration of smoothing parameter estimation convergence and four returned plots:

```
> ?gam.check
```

In addition, more information for `choose.k` can be accessed by the following command:

```
> ?choose.k
```

# 4

# Classification (I) – Tree, Lazy, and Probabilistic

In this chapter, we will cover the following recipes:

- ▶ Preparing the training and testing datasets
- ▶ Building a classification model with recursive partitioning trees
- ▶ Visualizing a recursive partitioning tree
- ▶ Measuring the prediction performance of a recursive partitioning tree
- ▶ Pruning a recursive partitioning tree
- ▶ Building a classification model with a conditional inference tree
- ▶ Visualizing a conditional inference tree
- ▶ Measuring the prediction performance of a conditional inference tree
- ▶ Classifying data with a k-nearest neighbor classifier
- ▶ Classifying data with logistic regression
- ▶ Classifying data with the Naïve Bayes classifier

## Introduction

Classification is used to identify a category of new observations (testing datasets) based on a classification model built from the training dataset, of which the categories are already known. Similar to regression, classification is categorized as a supervised learning method as it employs known answers (label) of a training dataset to predict the answer (label) of the testing dataset. The main difference between regression and classification is that regression is used to predict continuous values.

In contrast to this, classification is used to identify the category of a given observation. For example, one may use regression to predict the future price of a given stock based on historical prices. However, one should use the classification method to predict whether the stock price will rise or fall.

In this chapter, we will illustrate how to use R to perform classification. We first build a training dataset and a testing dataset from the churn dataset, and then apply different classification methods to classify the churn dataset. In the following recipes, we will introduce the tree-based classification method using a traditional classification tree and a conditional inference tree, lazy-based algorithm, and a probabilistic-based method using the training dataset to build up a classification model, and then use the model to predict the category (class label) of the testing dataset. We will also use a confusion matrix to measure the performance.

## Preparing the training and testing datasets

Building a classification model requires a training dataset to train the classification model, and testing data is needed to then validate the prediction performance. In the following recipe, we will demonstrate how to split the telecom churn dataset into training and testing datasets, respectively.

### Getting ready

In this recipe, we will use the telecom churn dataset as the input data source, and split the data into training and testing datasets.

### How to do it...

Perform the following steps to split the churn dataset into training and testing datasets:

1. You can retrieve the churn dataset from the C50 package:

```
> install.packages("C50")
> library(C50)
> data(churn)
```
2. Use `str` to read the structure of the dataset:

```
> str(churnTrain)
```
3. We can remove the `state`, `area_code`, and `account_length` attributes, which are not appropriate for classification features:

```
> churnTrain = churnTrain[!, ! names(churnTrain) %in% c("state",
 "area_code", "account_length")]
```

4. Then, split 70 percent of the data into the training dataset and 30 percent of the data into the testing dataset:

```
> set.seed(2)

> ind = sample(2, nrow(churnTrain), replace = TRUE, prob=c(0.7,
0.3))

> trainset = churnTrain[ind == 1,]

> testset = churnTrain[ind == 2,]
```

5. Lastly, use `dim` to explore the dimensions of both the training and testing datasets:

```
> dim(trainset)
[1] 2315 17

> dim(testset)
[1] 1018 17
```

## How it works...

In this recipe, we use the telecom churn dataset as our example data source. The dataset contains 20 variables with 3,333 observations. We would like to build a classification model to predict whether a customer will churn, which is very important to the telecom company as the cost of acquiring a new customer is significantly more than retaining one.

Before building the classification model, we need to preprocess the data first. Thus, we load the churn data from the `C50` package into the R session with the variable name as `churn`. As we determined that attributes such as `state`, `area_code`, and `account_length` are not useful features for building the classification model, we remove these attributes.

After preprocessing the data, we split it into training and testing datasets, respectively. We then use a `sample` function to randomly generate a sequence containing 70 percent of the training dataset and 30 percent of the testing dataset with a size equal to the number of observations. Then, we use a generated sequence to split the churn dataset into the training dataset, `trainset`, and the testing dataset, `testset`. Lastly, by using the `dim` function, we found that 2,315 out of the 3,333 observations are categorized into the training dataset, `trainset`, while the other 1,018 are categorized into the testing dataset, `testset`.

## There's more...

You can combine the split process of the training and testing datasets into the `split.data` function. Therefore, you can easily split the data into the two datasets by calling this function and specifying the proportion and seed in the parameters:

```
> split.data = function(data, p = 0.7, s = 666) {
+ set.seed(s)
+ index = sample(1:dim(data)[1])
```

```
+ train = data[index[1:floor(dim(data)[1] * p)],]
+ test = data[index[((ceiling(dim(data)[1] * p)) + 1):dim(data)[1]],]
+ return(list(train = train, test = test))
+ }
```

## Building a classification model with recursive partitioning trees

A classification tree uses a split condition to predict class labels based on one or multiple input variables. The classification process starts from the root node of the tree; at each node, the process will check whether the input value should recursively continue to the right or left sub-branch according to the split condition, and stops when meeting any leaf (terminal) nodes of the decision tree. In this recipe, we will introduce how to apply a recursive partitioning tree on the customer churn dataset.

### Getting ready

You need to have completed the previous recipe by splitting the churn dataset into the training dataset (`trainset`) and testing dataset (`testset`), and each dataset should contain exactly 17 variables.

### How to do it...

Perform the following steps to split the churn dataset into training and testing datasets:

1. Load the `rpart` package:  
`> library(rpart)`
2. Use the `rpart` function to build a classification tree model:  
`> churn.rp = rpart(churn ~ ., data=trainset)`
3. Type `churn.rp` to retrieve the node detail of the classification tree:  
`> churn.rp`
4. Next, use the `printcp` function to examine the complexity parameter:  
`> printcp(churn.rp)`

**Classification tree:**

```
rpart(formula = churn ~ ., data = trainset)
```

**Variables actually used in tree construction:**

```
[1] international_plan number_customer_service_calls
[3] total_day_minutes total_eve_minutes
[5] total_intl_calls total_intl_minutes
[7] voice_mail_plan
```

Root node error: 342/2315 = 0.14773

n= 2315

|   | CP       | nsplit | rel error | xerror  | xstd     |
|---|----------|--------|-----------|---------|----------|
| 1 | 0.076023 | 0      | 1.00000   | 1.00000 | 0.049920 |
| 2 | 0.074561 | 2      | 0.84795   | 0.99708 | 0.049860 |
| 3 | 0.055556 | 4      | 0.69883   | 0.76023 | 0.044421 |
| 4 | 0.026316 | 7      | 0.49415   | 0.52632 | 0.037673 |
| 5 | 0.023392 | 8      | 0.46784   | 0.52047 | 0.037481 |
| 6 | 0.020468 | 10     | 0.42105   | 0.50877 | 0.037092 |
| 7 | 0.017544 | 11     | 0.40058   | 0.47076 | 0.035788 |
| 8 | 0.010000 | 12     | 0.38304   | 0.47661 | 0.035993 |

5. Next, use the `plotcp` function to plot the cost complexity parameters:

```
> plotcp(churn.rp)
```

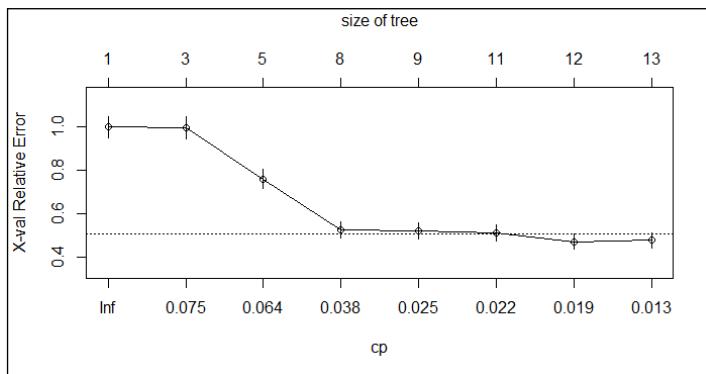


Figure 1: The cost complexity parameter plot

6. Lastly, use the `summary` function to examine the built model:

```
> summary(churn.rp)
```

## How it works...

In this recipe, we use a recursive partitioning tree from the `rpart` package to build a tree-based classification model. The recursive portioning tree includes two processes: recursion and partitioning. During the process of decision induction, we have to consider a statistic evaluation question (or simply a yes/no question) to partition the data into different partitions in accordance with the assessment result. Then, as we have determined the child node, we can repeatedly perform the splitting until the stop criteria is satisfied.

For example, the data (shown in the following figure) in the root node can be partitioned into two groups with regard to the question of whether  $f_1$  is smaller than  $X$ . If so, the data is divided into the left-hand side. Otherwise, it is split into the right-hand side. Then, we can continue to partition the left-hand side data with the question of whether  $f_2$  is smaller than  $Y$ :

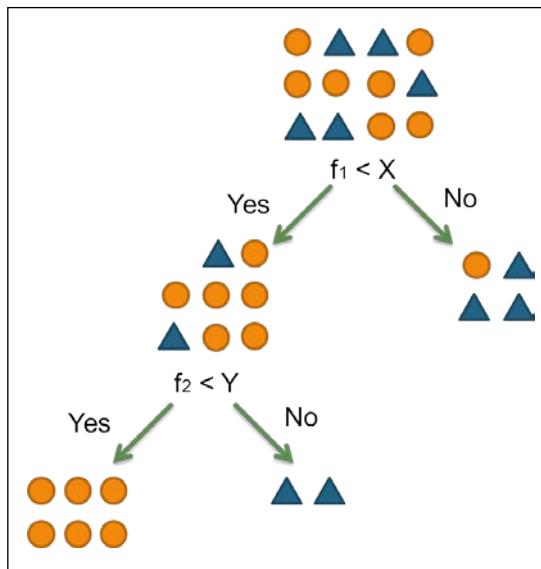


Figure 2: Recursive partitioning tree

In the first step, we load the `rpart` package with the `library` function. Next, we build a classification model using the `churn` variable as a classification category (class label) and the remaining variables as input features.

After the model is built, you can type the variable name of the built model, `churn.rp`, to display the tree node details. In the printed node detail, `n` indicates the sample size, `loss` indicates the misclassification cost, `yval` stands for the classified membership (no or yes, in this case), and `yprob` stands for the probabilities of two classes (the left value refers to the probability reaching label no, and the right value refers to the probability reaching label yes).

Then, we use the `printcp` function to print the complexity parameters of the built tree model. From the output of `printcp`, one should find the value of CP, a complexity parameter, which serves as a penalty to control the size of the tree. In short, the greater the CP value, the fewer the number of splits there are (`nsplit`). The output value (the `rel error`) represents the average deviance of the current tree divided by the average deviance of the null tree. A `xerror` value represents the relative error estimated by a 10-fold classification. `xstd` stands for the standard error of the relative error.

To make the **CP (cost complexity parameter)** table more readable, we use `plotcp` to generate an information graphic of the CP table. As per the screenshot (step 5), the x-axis at the bottom illustrates the `cp` value, the y-axis illustrates the relative error, and the upper x-axis displays the size of the tree. The dotted line indicates the upper limit of a standard deviation. From the screenshot, we can determine that minimum cross-validation error occurs when the tree is at a size of 12.

We can also use the `summary` function to display the function call, complexity parameter table for the fitted tree model, variable importance, which helps identify the most important variable for the tree classification (summing up to 100), and detailed information of each node.

The advantage of using the decision tree is that it is very flexible and easy to interpret. It works on both classification and regression problems, and more; it is nonparametric. Therefore, one does not have to worry about whether the data is linear separable. As for the disadvantage of using the decision tree, it is that it tends to be biased and over-fitted. However, you can conquer the bias problem through the use of a conditional inference tree, and solve the problem of over-fitting through a random forest method or tree pruning.

## See also

- ▶ For more information about the `rpart`, `printcp`, and `summary` functions, please use the `help` function:

```
> ?rpart
> ?printcp
> ?summary.rpart
```
- ▶ `C50` is another package that provides a decision tree and a rule-based model. If you are interested in the package, you may refer to the document at <http://cran.r-project.org/web/packages/C50/C50.pdf>.

## Visualizing a recursive partitioning tree

From the last recipe, we learned how to print the classification tree in a text format. To make the tree more readable, we can use the `plot` function to obtain the graphical display of a built classification tree.

## Getting ready

One needs to have the previous recipe completed by generating a classification model, and assign the model into the `churn_rp` variable.

## **How to do it...**

Perform the following steps to visualize the classification tree:

1. Use the `plot` function and the `text` function to plot the classification tree:

```
> plot(churn.rp, margin= 0.1)
> text(churn.rp, all=TRUE, use.n = TRUE)
```

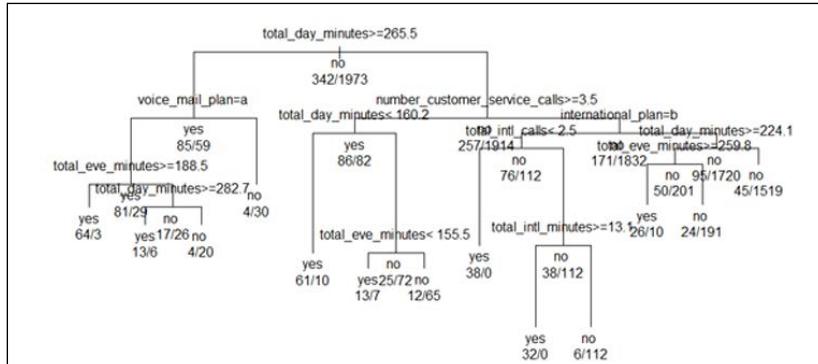


Figure 3: The graphical display of a classification tree

2. You can also specify the uniform, branch, and margin parameter to adjust the layout:

```
> plot(churn.rp, uniform=TRUE, branch=0.6, margin=0.1)
> text(churn.rp, all=TRUE, use.n = TRUE)
```

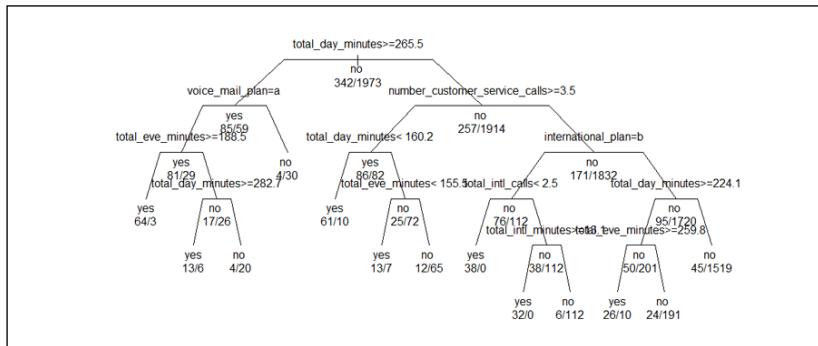


Figure 4: Adjust the layout of the classification tree

## How it works...

Here, we demonstrate how to use the `plot` function to graphically display a classification tree. The `plot` function can simply visualize the classification tree, and you can then use the `text` function to add text to the plot.

In *Figure 3*, we assign `margin = 0.1` as a parameter to add extra white space around the border to prevent the displayed text being truncated by the margin. It shows that the length of the branches displays the relative magnitude of the drop in deviance. We then use the `text` function to add labels for the nodes and branches. By default, the `text` function will add a split condition on each split, and add a category label in each terminal node. In order to add extra information on the tree plot, we set the parameter as all equal to `TRUE` to add a label to all the nodes. In addition to this, we add a parameter by specifying `use.n = TRUE` to add extra information, which shows that the actual number of observations fall into two different categories (no and yes).

In *Figure 4*, we set the option `branch` to `0.6` to add a shoulder to each plotted branch. In addition to this, in order to display branches of an equal length rather than relative magnitude of the drop in deviance, we set the option `uniform` to `TRUE`. As a result, *Figure 4* shows a classification tree with short shoulders and branches of equal length.

## See also

- ▶ You may use `?plot.rpart` to read more about the plotting of the classification tree. This document also includes information on how to specify the parameters, `uniform`, `branch`, `compress`, `nspac`, `margin`, and `minbranch`, to adjust the layout of the classification tree.

## Measuring the prediction performance of a recursive partitioning tree

Since we have built a classification tree in the previous recipes, we can use it to predict the category (class label) of new observations. Before making a prediction, we first validate the prediction power of the classification tree, which can be done by generating a classification table on the testing dataset. In this recipe, we will introduce how to generate a predicted label versus a real label table with the `predict` function and the `table` function, and explain how to generate a confusion matrix to measure the performance.

## Getting ready

You need to have the previous recipe completed by generating the classification model, `churn.rp`. In addition to this, you have to prepare the training dataset, `trainset`, and the testing dataset, `testset`, generated in the first recipe of this chapter.

## How to do it...

Perform the following steps to validate the prediction performance of a classification tree:

1. You can use the predict function to generate a predicted label of testing the dataset:

```
> predictions = predict(churn.rp, testset, type="class")
```

2. Use the table function to generate a classification table for the testing dataset:

```
> table(testset$churn, predictions)

predictions
 yes no
yes 100 41
no 18 859
```

3. One can further generate a confusion matrix using the confusionMatrix function provided in the caret package:

```
> library(caret)
> confusionMatrix(table(predictions, testset$churn))

Confusion Matrix and Statistics
```

```
predictions yes no
 yes 100 18
 no 41 859
```

```
Accuracy : 0.942
95% CI : (0.9259, 0.9556)
```

```
No Information Rate : 0.8615
```

```
P-Value [Acc > NIR] : < 2.2e-16
```

```
Kappa : 0.7393
```

```
McNemar's Test P-Value : 0.004181
```

```
Sensitivity : 0.70922
```

```
Specificity : 0.97948
```

```
Pos Pred Value : 0.84746
```

```
Neg Pred Value : 0.95444
```

```
Prevalence : 0.13851
```

```
Detection Rate : 0.09823
Detection Prevalence : 0.11591
Balanced Accuracy : 0.84435

'Positive' Class : yes
```

## How it works...

In this recipe, we use a predict function and built up classification model, `churn.rp`, to predict the possible class labels of the testing dataset, `testset`. The predicted categories (class labels) are coded as either no or yes. Then, we use the `table` function to generate a classification table on the testing dataset. From the table, we discover that there are 859 correctly predicted as no, while 18 are misclassified as yes. 100 of the yes predictions are correctly predicted, but 41 observations are misclassified into no. Further, we use the `confusionMatrix` function from the `caret` package to produce a measurement of the classification model.

## See also

- ▶ You may use `?confusionMatrix` to read more about the performance measurement using the confusion matrix
- ▶ For those who are interested in the definition output by the confusion matrix, please refer to the Wikipedia entry, **Confusion\_matrix** ([http://en.wikipedia.org/wiki/Confusion\\_matrix](http://en.wikipedia.org/wiki/Confusion_matrix))

## Pruning a recursive partitioning tree

In previous recipes, we have built a complex decision tree for the churn dataset. However, sometimes we have to remove sections that are not powerful in classifying instances to avoid over-fitting, and to improve the prediction accuracy. Therefore, in this recipe, we introduce the cost complexity pruning method to prune the classification tree.

## Getting ready

You need to have the previous recipe completed by generating a classification model, and assign the model into the `churn.rp` variable.

# How to do it...

Perform the following steps to prune the classification tree:

1. Find the minimum cross-validation error of the classification tree model:

```
> min(churn.rp$cptable[, "xerror"])
[1] 0.4707602
```

2. Locate the record with the minimum cross-validation errors:

```
> which.min(churn.rp$cptable[, "xerror"])
7
```

3. Get the cost complexity parameter of the record with the minimum cross-validation errors:

```
> churn.cp = churn.rp$cptable[7, "CP"]
> churn.cp
[1] 0.01754386
```

4. Prune the tree by setting the `cp` parameter to the CP value of the record with minimum cross-validation errors:

```
> prune.tree = prune(churn.rp, cp= churn.cp)
```

5. Visualize the classification tree by using the `plot` and `text` function:

```
> plot(prune.tree, margin= 0.1)
> text(prune.tree, all=TRUE , use.n=TRUE)
```

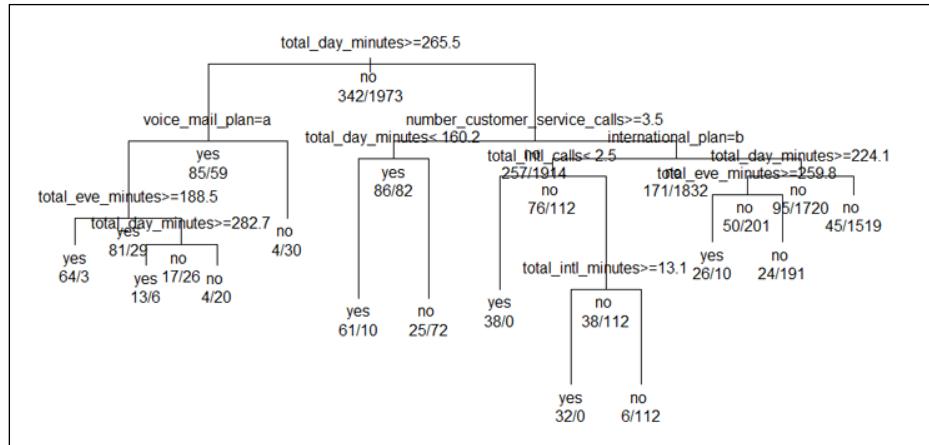


Figure 5: The pruned classification tree

6. Next, you can generate a classification table based on the pruned classification tree model:

```
> predictions = predict(prune.tree, testset, type="class")
> table(testset$churn, predictions)

 predictions
 yes no
 yes 95 46
 no 14 863
```

7. Lastly, you can generate a confusion matrix based on the classification table:

```
> confusionMatrix(table(predictions, testset$churn))
Confusion Matrix and Statistics
```

```
predictions yes no
 yes 95 14
 no 46 863
```

```
Accuracy : 0.9411
95% CI : (0.9248, 0.9547)
```

```
No Information Rate : 0.8615
P-Value [Acc > NIR] : 2.786e-16
```

```
Kappa : 0.727
```

```
Mcnemar's Test P-Value : 6.279e-05
```

```
Sensitivity : 0.67376
```

```
Specificity : 0.98404
```

```
Pos Pred Value : 0.87156
```

```
Neg Pred Value : 0.94939
```

```
Prevalence : 0.13851
```

```
Detection Rate : 0.09332
```

```
Detection Prevalence : 0.10707
```

```
Balanced Accuracy : 0.82890
```

```
'Positive' Class : yes
```

## How it works...

In this recipe, we discussed pruning a classification tree to avoid over-fitting and producing a more robust classification model. We first located the record with the minimum cross-validation errors within the `cptable`, and we then extracted the CP of the record and assigned the value to `churn.cp`. Next, we used the `prune` function to prune the classification tree with `churn.cp` as the parameter. Then, by using the `plot` function, we graphically displayed the pruned classification tree. From *Figure 5*, it is clear that the split of the tree is less than the original classification tree (*Figure 3*). Lastly, we produced a classification table and used the confusion matrix to validate the performance of the pruned tree. The result shows that the accuracy (0.9411) is slightly lower than the original model (0.942), and also suggests that the pruned tree may not perform better than the original classification tree as we have pruned some split conditions (Still, one should examine the change in sensitivity and specificity). However, the pruned tree model is more robust as it removes some split conditions that may lead to over-fitting.

## See also

- ▶ For those who would like to know more about cost complexity pruning, please refer to the Wikipedia article for **Pruning (decision\_trees)**: [http://en.wikipedia.org/wiki/Pruning\\_\(decision\\_trees\)](http://en.wikipedia.org/wiki/Pruning_(decision_trees))

## Building a classification model with a conditional inference tree

In addition to traditional decision trees (`rpart`), conditional inference trees (`ctree`) are another popular tree-based classification method. Similar to traditional decision trees, conditional inference trees also recursively partition the data by performing a univariate split on the dependent variable. However, what makes conditional inference trees different from traditional decision trees is that conditional inference trees adapt the significance test procedures to select variables rather than selecting variables by maximizing information measures (`rpart` employs a Gini coefficient). In this recipe, we will introduce how to adapt a conditional inference tree to build a classification model.

## Getting ready

You need to have the first recipe completed by generating the training dataset, `trainset`, and the testing dataset, `testset`.

## How to do it...

Perform the following steps to build the conditional inference tree:

1. First, we use `ctree` from the `party` package to build the classification model:

```
> library(party)
> ctree.model = ctree(churn ~ . , data = trainset)
```

2. Then, we examine the built tree model:

```
> ctree.model
```

## How it works...

In this recipe, we used a conditional inference tree to build a classification tree. The use of `ctree` is similar to `rpart`. Therefore, you can easily test the classification power using either a traditional decision tree or a conditional inference tree while confronting classification problems. Next, we obtain the node details of the classification tree by examining the built model. Within the model, we discover that `ctree` provides information similar to a split condition, criterion (1 - p-value), statistics (test statistics), and weight (the case weight corresponding to the node). However, it does not offer as much information as `rpart` does through the use of the `summary` function.

## See also

- ▶ You may use the `help` function to refer to the definition of **Binary Tree Class** and read more about the properties of binary trees:  
`> help("BinaryTree-class")`

## Visualizing a conditional inference tree

Similar to `rpart`, the `party` package also provides a visualization method for users to plot conditional inference trees. In the following recipe, we will introduce how to use the `plot` function to visualize conditional inference trees.

## Getting ready

You need to have the first recipe completed by generating the conditional inference tree model, `ctree.model`. In addition to this, you need to have both, `trainset` and `testset`, loaded in an R session.

# How to do it...

Perform the following steps to visualize the conditional inference tree:

1. Use the `plot` function to plot `ctree.model` built in the last recipe:

```
> plot(ctree.model)
```

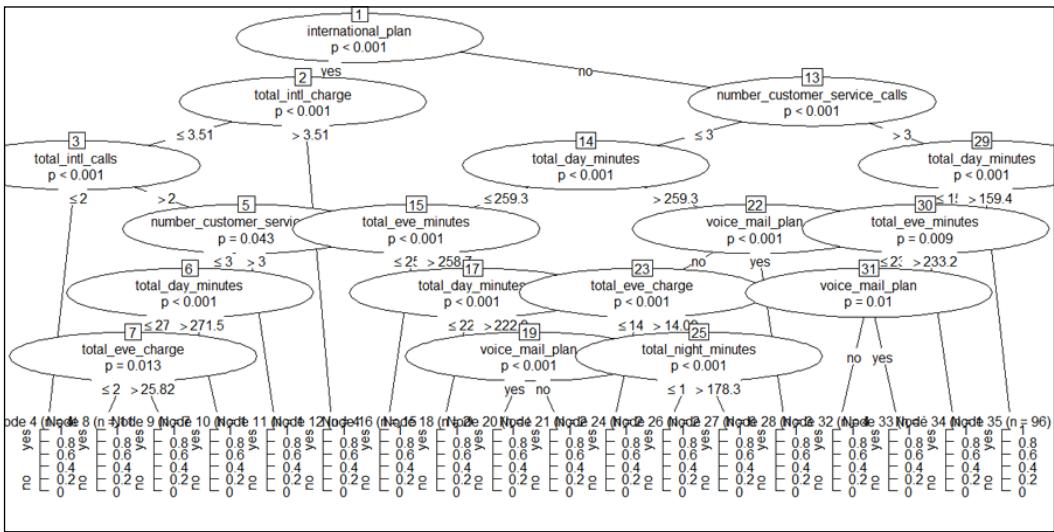


Figure 6: A conditional inference tree of churn data

2. To obtain a simple conditional inference tree, one can reduce the built model with less input features, and redraw the classification tree:

```
> daycharge.model = ctree(churn ~ total_day_charge, data = trainset)
> plot(daycharge.model)
```

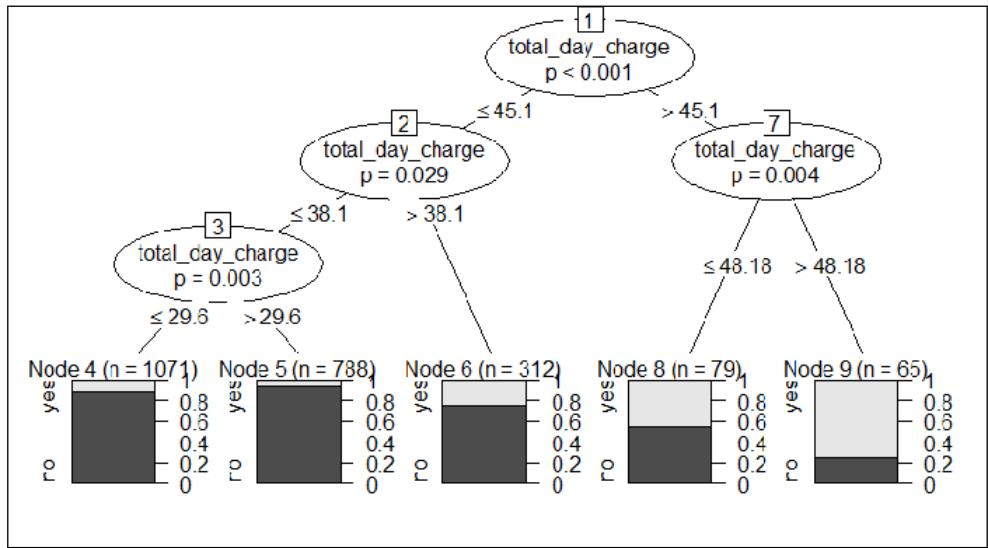


Figure 7: A conditional inference tree using the `total_day_charge` variable as only split condition

## How it works...

To visualize the node detail of the conditional inference tree, we can apply the `plot` function on a built classification model. The output figure reveals that every intermediate node shows the dependent variable name and the p-value. The split condition is displayed on the left and right branches. The terminal nodes show the number of categorized observations,  $n$ , and the probability of a class label of either 0 or 1.

Taking Figure 7 as an example, we first build a classification model using `total_day_charge` as the only feature and `churn` as the class label. The built classification tree shows that when `total_day_charge` is above 48.18, the lighter gray area is greater than the darker gray in node 9, which indicates that the customer with a day charge of over 48.18 has a greater likelihood to churn (label = yes).

## See also

- ▶ The visualization of the conditional inference tree comes from the `plot.BinaryTree` function. If you are interested in adjusting the layout of the classification tree, you may use the `help` function to read the following document:

```
> ?plot.BinaryTree
```

# Measuring the prediction performance of a conditional inference tree

After building a conditional inference tree as a classification model, we can use the `treeresponse` and `predict` functions to predict categories of the testing dataset, `testset`, and further validate the prediction power with a classification table and a confusion matrix.

## Getting ready

You need to have the previous recipe completed by generating the conditional inference tree model, `ctree.model`. In addition to this, you need to have both `trainset` and `testset` loaded in an R session.

## How to do it...

Perform the following steps to measure the prediction performance of a conditional inference tree:

1. You can use the `predict` function to predict the category of the testing dataset, `testset`:

```
> ctree.predict = predict(ctree.model , testset)
> table(ctree.predict, testset$churn)
```

| ctree.predict | yes | no  |
|---------------|-----|-----|
| yes           | 99  | 15  |
| no            | 42  | 862 |

2. Furthermore, you can use `confusionMatrix` from the `caret` package to generate the performance measurements of the prediction result:

```
> confusionMatrix(table(ctree.predict, testset$churn))
Confusion Matrix and Statistics
```

| ctree.predict | yes | no  |
|---------------|-----|-----|
| yes           | 99  | 15  |
| no            | 42  | 862 |

Accuracy : 0.944

95% CI : (0.9281, 0.9573)

```
No Information Rate : 0.8615
P-Value [Acc > NIR] : < 2.2e-16
```

```
Kappa : 0.7449
McNemar's Test P-Value : 0.0005736
```

```
Sensitivity : 0.70213
Specificity : 0.98290
Pos Pred Value : 0.86842
Neg Pred Value : 0.95354
Prevalence : 0.13851
Detection Rate : 0.09725
Detection Prevalence : 0.11198
Balanced Accuracy : 0.84251
```

```
'Positive' Class : yes
```

3. You can also use the treeresponse function, which will tell you the list of class probabilities:

```
> tr = treeresponse(ctree.model, newdata = testset[1:5,])
```

```
> tr
```

```
[[1]]
```

```
[1] 0.03497409 0.96502591
```

```
[[2]]
```

```
[1] 0.02586207 0.97413793
```

```
[[3]]
```

```
[1] 0.02586207 0.97413793
```

```
[[4]]
```

```
[1] 0.02586207 0.97413793
```

```
[[5]]
```

```
[1] 0.03497409 0.96502591
```

## How it works...

In this recipe, we first demonstrate that one can use the `prediction` function to predict the category (class label) of the testing dataset, `testset`, and then employ a `table` function to generate a classification table. Next, you can use the `confusionMatrix` function built into the `caret` package to determine the performance measurements.

In addition to the `predict` function, `treeresponse` is also capable of estimating the class probability, which will often classify labels with a higher probability. In this example, we demonstrated how to obtain the estimated class probability using the top five records of the testing dataset, `testset`. The `treeresponse` function returns a list of five probabilities. You can use the list to determine the label of instance.

## See also

- ▶ For the `predict` function, you can specify the type as `response`, `prob`, or `node`. If you specify the type as `prob` when using the `predict` function (for example, `predict(... type="prob")`), you will get exactly the same result as what `treeresponse` returns.

# Classifying data with the k-nearest neighbor classifier

**K-nearest neighbor (knn)** is a nonparametric lazy learning method. From a nonparametric view, it does not make any assumptions about data distribution. In terms of lazy learning, it does not require an explicit learning phase for generalization. The following recipe will introduce how to apply the k-nearest neighbor algorithm on the churn dataset.

## Getting ready

You need to have the previous recipe completed by generating the training and testing datasets.

## How to do it...

Perform the following steps to classify the churn data with the k-nearest neighbor algorithm:

1. First, one has to install the `class` package and have it loaded in an R session:  

```
> install.packages("class")
> library(class)
```

2. Replace yes and no of the voice\_mail\_plan and international\_plan attributes in both the training dataset and testing dataset to 1 and 0:

```
> levels(trainset$international_plan) = list("0"="no", "1"="yes")
> levels(trainset$voice_mail_plan) = list("0"="no", "1"="yes")
> levels(testset$international_plan) = list("0"="no", "1"="yes")
> levels(testset$voice_mail_plan) = list("0"="no", "1"="yes")
```

3. Use the knn classification method on the training dataset and the testing dataset:

```
> churn.knn = knn(trainset[, ! names(trainset) %in% c("churn")], testset[, ! names(testset) %in% c("churn")], trainset$churn, k=3)
```

4. Then, you can use the summary function to retrieve the number of predicted labels:

```
> summary(churn.knn)
yes no
 77 941
```

5. Next, you can generate the classification matrix using the table function:

```
> table(testset$churn, churn.knn)
churn.knn
yes no
yes 44 97
no 33 844
```

6. Lastly, you can generate a confusion matrix by using the confusionMatrix function:

```
> confusionMatrix(table(testset$churn, churn.knn))
Confusion Matrix and Statistics
```

```
churn.knn
yes no
yes 44 97
no 33 844
```

```
Accuracy : 0.8723
95% CI : (0.8502, 0.8922)
No Information Rate : 0.9244
P-Value [Acc > NIR] : 1
```

Kappa : 0.339

McNemar's Test P-Value : 3.286e-08

Sensitivity : 0.57143  
Specificity : 0.89692  
Pos Pred Value : 0.31206  
Neg Pred Value : 0.96237  
Prevalence : 0.07564  
Detection Rate : 0.04322  
Detection Prevalence : 0.13851  
Balanced Accuracy : 0.73417

'Positive' Class : yes

## How it works...

**knn** trains all samples and classifies new instances based on a similarity (distance) measure. For example, the similarity measure can be formulated as follows:

- ▶ **Euclidian Distance:**  $\sqrt{\sum_{i=1}^k (x_i - y_i)^2}$
- ▶ **Manhattan Distance:**  $\sum_{i=1}^k |(x_i - y_i)|$

In knn, a new instance is classified to a label (class) that is common among the k-nearest neighbors. If  $k = 1$ , then the new instance is assigned to the class where its nearest neighbor belongs. The only required input for the algorithm is k. If we give a small k input, it may lead to over-fitting. On the other hand, if we give a large k input, it may result in under-fitting. To choose a proper k-value, one can count on cross-validation.

The advantages of knn are:

- ▶ The cost of the learning process is zero
- ▶ It is nonparametric, which means that you do not have to make the assumption of data distribution
- ▶ You can classify any data whenever you can find similarity measures of given instances

The main disadvantages of knn are:

- ▶ It is hard to interpret the classified result.
- ▶ It is an expensive computation for a large dataset.
- ▶ The performance relies on the number of dimensions. Therefore, for a high dimension problem, you should reduce the dimension first to increase the process performance.

The use of knn does not vary significantly from applying a tree-based algorithm mentioned in the previous recipes. However, while a tree-based algorithm may show you the decision tree model, the output produced by knn only reveals classification category factors. However, before building a classification model, one should replace the attribute with a string type to an integer since the k-nearest neighbor algorithm needs to calculate the distance between observations. Then, we build up a classification model by specifying  $k=3$ , which means choosing the three nearest neighbors. After the classification model is built, we can generate a classification table using predicted factors and the testing dataset label as the input. Lastly, we can generate a confusion matrix from the classification table. The confusion matrix output reveals an accuracy result of (0.8723), which suggests that both the tree-based methods mentioned in previous recipes outperform the accuracy of the k-nearest neighbor classification method in this case. Still, we cannot determine which model is better depending merely on accuracy, one should also examine the specificity and sensitivity from the output.

## See also

- ▶ There is another package named kknn, which provides a weighted k-nearest neighbor classification, regression, and clustering. You can learn more about the package by reading this document: <http://cran.r-project.org/web/packages/kknn/kknn.pdf>.

# Classifying data with logistic regression

Logistic regression is a form of probabilistic statistical classification model, which can be used to predict class labels based on one or more features. The classification is done by using the `logit` function to estimate the outcome probability. One can use logistic regression by specifying the family as a binomial while using the `glm` function. In this recipe, we will introduce how to classify data using logistic regression.

## Getting ready

You need to have completed the first recipe by generating training and testing datasets.

## How to do it...

Perform the following steps to classify the churn data with logistic regression:

1. With the specification of family as a binomial, we apply the `glm` function on the dataset, `trainset`, by using `churn` as a class label and the rest of the variables as input features:

```
> fit = glm(churn ~ ., data = trainset, family=binomial)
```

2. Use the `summary` function to obtain summary information of the built logistic regression model:

```
> summary(fit)
```

Call:

```
glm(formula = churn ~ ., family = binomial, data = trainset)
```

Deviance Residuals:

| Min     | 1Q     | Median | 3Q     | Max    |
|---------|--------|--------|--------|--------|
| -3.1519 | 0.1983 | 0.3460 | 0.5186 | 2.1284 |

Coefficients:

|                       | Estimate   | Std. Error | z value         |
|-----------------------|------------|------------|-----------------|
| Pr(> z )              |            |            |                 |
| (Intercept)           | 8.3462866  | 0.8364914  | 9.978 < 2e-16   |
| international_plan    | -2.0534243 | 0.1726694  | -11.892 < 2e-16 |
| voice_mail_plan       | 1.3445887  | 0.6618905  | 2.031           |
| 0.042211              |            |            |                 |
| number_vmail_messages | -0.0155101 | 0.0209220  | -0.741          |
| 0.458496              |            |            |                 |
| total_day_minutes     | 0.2398946  | 3.9168466  | 0.061           |
| 0.951163              |            |            |                 |
| total_day_calls       | -0.0014003 | 0.0032769  | -0.427          |
| 0.669141              |            |            |                 |
| total_day_charge      | -1.4855284 | 23.0402950 | -0.064          |
| 0.948592              |            |            |                 |
| total_eve_minutes     | 0.3600678  | 1.9349825  | 0.186           |
| 0.852379              |            |            |                 |

|                               |            |            |                 |
|-------------------------------|------------|------------|-----------------|
| total_eve_calls               | -0.0028484 | 0.0033061  | -0.862          |
| 0.388928                      |            |            |                 |
| total_eve_charge              | -4.3204432 | 22.7644698 | -0.190          |
| 0.849475                      |            |            |                 |
| total_night_minutes           | 0.4431210  | 1.0478105  | 0.423           |
| 0.672367                      |            |            |                 |
| total_night_calls             | 0.0003978  | 0.0033188  | 0.120           |
| 0.904588                      |            |            |                 |
| total_night_charge            | -9.9162795 | 23.2836376 | -0.426          |
| 0.670188                      |            |            |                 |
| total_intl_minutes            | 0.4587114  | 6.3524560  | 0.072           |
| 0.942435                      |            |            |                 |
| total_intl_calls              | 0.1065264  | 0.0304318  | 3.500           |
| 0.000464                      |            |            |                 |
| total_intl_charge             | -2.0803428 | 23.5262100 | -0.088          |
| 0.929538                      |            |            |                 |
| number_customer_service_calls | -0.5109077 | 0.0476289  | -10.727 < 2e-16 |

|                               |     |
|-------------------------------|-----|
| (Intercept)                   | *** |
| international_planyes         | *** |
| voice_mail_planyes            | *   |
| number_vmail_messages         |     |
| total_day_minutes             |     |
| total_day_calls               |     |
| total_day_charge              |     |
| total_eve_minutes             |     |
| total_eve_calls               |     |
| total_eve_charge              |     |
| total_night_minutes           |     |
| total_night_calls             |     |
| total_night_charge            |     |
| total_intl_minutes            |     |
| total_intl_calls              | *** |
| total_intl_charge             |     |
| number_customer_service_calls | *** |
| ---                           |     |

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

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 1938.8 on 2314 degrees of freedom  
Residual deviance: 1515.3 on 2298 degrees of freedom  
AIC: 1549.3

Number of Fisher Scoring iterations: 6

3. Then, we find that the built model contains insignificant variables, which would lead to misclassification. Therefore, we use significant variables only to train the classification model:

```
> fit = glm(churn ~ international_plan + voice_mail_plan+total_
 intl_calls+number_customer_service_calls, data = trainset,
 family=binomial)
> summary(fit)
```

Call:

```
glm(formula = churn ~ international_plan + voice_mail_plan +
 total_intl_calls + number_customer_service_calls, family =
binomial,
 data = trainset)
```

Deviance Residuals:

| Min     | 1Q     | Median | 3Q     | Max    |
|---------|--------|--------|--------|--------|
| -2.7308 | 0.3103 | 0.4196 | 0.5381 | 1.6716 |

Coefficients:

|                               | Estimate | Std. Error | z value |
|-------------------------------|----------|------------|---------|
| (Intercept)                   | 2.32304  | 0.16770    | 13.852  |
| international_planyes         | -2.00346 | 0.16096    | -12.447 |
| voice_mail_planyes            | 0.79228  | 0.16380    | 4.837   |
| total_intl_calls              | 0.08414  | 0.02862    | 2.939   |
| number_customer_service_calls | -0.44227 | 0.04451    | -9.937  |
|                               | Pr(> z ) |            |         |

```
(Intercept) < 2e-16 ***
international_planyes < 2e-16 ***
voice_mail_planyes 1.32e-06 ***
total_intl_calls 0.00329 **
number_customer_service_calls < 2e-16 ***

Signif. codes:
0 es: des: **rvice_calls < '. es: de
```

(Dispersion parameter for binomial family taken to be 1)

```
Null deviance: 1938.8 on 2314 degrees of freedom
Residual deviance: 1669.4 on 2310 degrees of freedom
AIC: 1679.4
```

Number of Fisher Scoring iterations: 5

4. Then, you can then use a fitted model, `fit`, to predict the outcome of `testset`. You can also determine the class by judging whether the probability is above 0.5:

```
> pred = predict(fit,testset, type="response")
> Class = pred >.5
```

5. Next, the use of the `summary` function will show you the binary outcome count, and reveal whether the probability is above 0.5:

```
> summary(Class)

 Mode FALSE TRUE NA's
logical 29 989 0
```

6. You can generate the counting statistics based on the testing dataset label and predicted result:

```
> tb = table(testset$churn,Class)
> tb

 Class
 FALSE TRUE
yes 18 123
no 11 866
```

7. You can turn the statistics of the previous step into a classification table, and then generate the confusion matrix:

```
> churn.mod = ifelse(testset$churn == "yes", 1, 0)
> pred_class = churn.mod
> pred_class[pred<=.5] = 1- pred_class[pred<=.5]
> ctb = table(churn.mod, pred_class)
> ctb
 pred_class
churn.mod 0 1
 0 866 11
 1 18 123
> confusionMatrix(ctb)
Confusion Matrix and Statistics

 pred_class
churn.mod 0 1
 0 866 11
 1 18 123

 Accuracy : 0.9715
 95% CI : (0.9593, 0.9808)
No Information Rate : 0.8684
P-Value [Acc > NIR] : <2e-16

 Kappa : 0.8781
McNemar's Test P-Value : 0.2652

 Sensitivity : 0.9796
 Specificity : 0.9179
Pos Pred Value : 0.9875
Neg Pred Value : 0.8723
 Prevalence : 0.8684
Detection Rate : 0.8507
Detection Prevalence : 0.8615
Balanced Accuracy : 0.9488

'Positive' Class : 0
```

## How it works...

Logistic regression is very similar to linear regression; the main difference is that the dependent variable in linear regression is continuous, but the dependent variable in logistic regression is dichotomous (or nominal). The primary goal of logistic regression is to use logit to yield the probability of a nominal variable is related to the measurement variable. We can formulate logit in following equation:  $\ln(P/(1-P))$ , where P is the probability that certain event occurs.

The advantage of logistic regression is that it is easy to interpret, it directs model logistic probability, and provides a confidence interval for the result. Unlike the decision tree, which is hard to update the model, you can quickly update the classification model to incorporate new data in logistic regression. The main drawback of the algorithm is that it suffers from multicollinearity and, therefore, the explanatory variables must be linear independent. `glm` provides a generalized linear regression model, which enables specifying the model in the option family. If the family is specified to a binomial logistic, you can set the family as a binomial to classify the dependent variable of the category.

The classification process begins by generating a logistic regression model with the use of the training dataset by specifying `Churn` as the class label, the other variables as training features, and family set as binomial. We then use the `summary` function to generate the model's summary information. From the summary information, we may find some insignificant variables ( $p$ -values  $> 0.05$ ), which may lead to misclassification. Therefore, we should consider only significant variables for the model.

Next, we use the `fit` function to predict the categorical dependent variable of the testing dataset, `testset`. The `fit` function outputs the probability of a class label, with a result equal to 0.5 and below, suggesting that the predicted label does not match the label of the testing dataset, and a probability above 0.5 indicates that the predicted label matches the label of the testing dataset. Further, we can use the `summary` function to obtain the statistics of whether the predicted label matches the label of the testing dataset. Lastly, in order to generate a confusion matrix, we first generate a classification table, and then use `confusionMatrix` to generate the performance measurement.

### See also

- ▶ For more information of how to use the `glm` function, please refer to *Chapter, Understanding Regression Analysis*, which covers how to interpret the output of the `glm` function

# Classifying data with the Naïve Bayes classifier

The Naïve Bayes classifier is also a probability-based classifier, which is based on applying the Bayes theorem with a strong independent assumption. In this recipe, we will introduce how to classify data with the Naïve Bayes classifier.

## Getting ready

You need to have the first recipe completed by generating training and testing datasets.

## How to do it...

Perform the following steps to classify the churn data with the Naïve Bayes classifier:

1. Load the e1071 library and employ the naiveBayes function to build the classifier:

```
> library(e1071)
> classifier=naiveBayes(trainset[, !names(trainset) %in%
c("churn")], trainset$churn)
```

2. Type classifier to examine the function call, a-priori probability, and conditional probability:

```
> classifier
```

```
Naive Bayes Classifier for Discrete Predictors
```

```
Call:
```

```
naiveBayes.default(x = trainset[, !names(trainset) %in%
c("churn")],
y = trainset$churn)
```

```
A-priori probabilities:
```

```
trainset$churn
```

```
 yes no
```

```
0.1477322 0.8522678
```

```
Conditional probabilities:
```

```
international_plan
```

```
trainset$churn no yes
 yes 0.70467836 0.29532164
 no 0.93512418 0.06487582
```

3. Next, you can generate a classification table for the testing dataset:

```
> bayes.table = table(predict(classifier, testset[, names(testset) %in% c("churn")]), testset$churn)
> bayes.table
```

|     | yes | no  |
|-----|-----|-----|
| yes | 68  | 45  |
| no  | 73  | 832 |

4. Lastly, you can generate a confusion matrix from the classification table:

```
> confusionMatrix(bayes.table)
Confusion Matrix and Statistics
```

|     | yes | no  |
|-----|-----|-----|
| yes | 68  | 45  |
| no  | 73  | 832 |

```
Accuracy : 0.8841
95% CI : (0.8628, 0.9031)
No Information Rate : 0.8615
P-Value [Acc > NIR] : 0.01880
```

```
Kappa : 0.4701
McNemar's Test P-Value : 0.01294
```

```
Sensitivity : 0.4823
Specificity : 0.9487
Pos Pred Value : 0.6018
Neg Pred Value : 0.9193
Prevalence : 0.1385
Detection Rate : 0.0668
Detection Prevalence : 0.1110
Balanced Accuracy : 0.7155
```

```
'Positive' Class : yes
```

## How it works...

Naive Bayes assumes that features are conditionally independent, which the effect of a predictor(x) to class (c) is independent of the effect of other predictors to class(c). It computes the posterior probability,  $P(c|x)$ , as the following formula:

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

Where  $P(x|c)$  is called likelihood,  $p(x)$  is called the marginal likelihood, and  $p(c)$  is called the prior probability. If there are many predictors, we can formulate the posterior probability as follows:

$$P(c|x) = P(x_1|c) \times P(x_2|c) \times \dots \times P(x_n|c) \times P(c)$$

The advantage of Naïve Bayes is that it is relatively simple and straightforward to use. It is suitable when the training set is relative small, and may contain some noisy and missing data. Moreover, you can easily obtain the probability for a prediction. The drawbacks of Naïve Bayes are that it assumes that all features are independent and equally important, which is very unlikely in real-world cases.

In this recipe, we use the Naïve Bayes classifier from the `e1071` package to build a classification model. First, we specify all the variables (excluding the `churn` class label) as the first input parameters, and specify the `churn` class label as the second parameter in the `naiveBayes` function call. Next, we assign the classification model into the variable `classifier`. Then, we print the variable `classifier` to obtain information, such as function call, A-priori probabilities, and conditional probabilities. We can also use the `predict` function to obtain the predicted outcome and the `table` function to retrieve the classification table of the testing dataset. Finally, we use a confusion matrix to calculate the performance measurement of the classification model.

At last, we list a comparison table of all the mentioned algorithms in this chapter:

| Algorithm                     | Advantage                                                                                                                                                                                                                                                                  | Disadvantage                                                                                                                                                                                                               |
|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Recursive partitioning tree   | <ul style="list-style-type: none"><li>▶ Very flexible and easy to interpret</li><li>▶ Works on both classification and regression problems</li><li>▶ Nonparametric</li></ul>                                                                                               | <ul style="list-style-type: none"><li>▶ Prone to bias and overfitting</li></ul>                                                                                                                                            |
| Conditional inference tree    | <ul style="list-style-type: none"><li>▶ Very flexible and easy to interpret</li><li>▶ Works on both classification and regression problems</li><li>▶ Nonparametric</li><li>▶ Less prone to bias than a recursive partitioning tree</li></ul>                               | <ul style="list-style-type: none"><li>▶ Prone to over-fitting</li></ul>                                                                                                                                                    |
| K-nearest neighbor classifier | <ul style="list-style-type: none"><li>▶ The cost of the learning process is zero</li><li>▶ Nonparametric</li><li>▶ You can classify any data whenever you can find similarity measures of any given instances</li></ul>                                                    | <ul style="list-style-type: none"><li>▶ Hard to interpret the classified result</li><li>▶ Computation is expensive for a large dataset</li><li>▶ The performance relies on the number of dimensions</li></ul>              |
| Logistic regression           | <ul style="list-style-type: none"><li>▶ Easy to interpret</li><li>▶ Provides model logistic probability</li><li>▶ Provides confidence interval</li><li>▶ You can quickly update the classification model to incorporate new data</li></ul>                                 | <ul style="list-style-type: none"><li>▶ Suffers multicollinearity</li><li>▶ Does not handle the missing value of continuous variables</li><li>▶ Sensitive to extreme values of continuous variables</li></ul>              |
| Naïve Bayes                   | <ul style="list-style-type: none"><li>▶ Relatively simple and straightforward to use</li><li>▶ Suitable when the training set is relative small</li><li>▶ Can deal with some noisy and missing data</li><li>▶ Can easily obtain the probability for a prediction</li></ul> | <ul style="list-style-type: none"><li>▶ Assumes all features are independent and equally important, which is very unlikely in real-world cases</li><li>▶ Prone to bias when the number of training sets increase</li></ul> |

## See also

- ▶ To learn more about the Bayes theorem, you can refer to the following Wikipedia article: [http://en.wikipedia.org/wiki/Bayes'\\_theorem](http://en.wikipedia.org/wiki/Bayes'_theorem)

# 5

# Classification (II) – Neural Network and SVM

In this chapter, we will cover the following recipes:

- ▶ Classifying data with a support vector machine
- ▶ Choosing the cost of a support vector machine
- ▶ Visualizing an SVM fit
- ▶ Predicting labels based on a model trained by a support vector machine
- ▶ Tuning a support vector machine
- ▶ Training a neural network with neuralnet
- ▶ Visualizing a neural network trained by neuralnet
- ▶ Predicting labels based on a model trained by neuralnet
- ▶ Training a neural network with nnet
- ▶ Predicting labels based on a model trained by nnet

## Introduction

Most research has shown that **support vector machines (SVM)** and **neural networks (NN)** are powerful classification tools, which can be applied to several different areas. Unlike tree-based or probabilistic-based methods that were mentioned in the previous chapter, the process of how support vector machines and neural networks transform from input to output is less clear and can be hard to interpret. As a result, both support vector machines and neural networks are referred to as black box methods.

The development of a neural network is inspired by human brain activities. As such, this type of network is a computational model that mimics the pattern of the human mind. In contrast to this, support vector machines first map input data into a high dimension feature space defined by the kernel function, and find the optimum hyperplane that separates the training data by the maximum margin. In short, we can think of support vector machines as a linear algorithm in a high dimensional space.

Both these methods have advantages and disadvantages in solving classification problems. For example, support vector machine solutions are the global optimum, while neural networks may suffer from multiple local optima. Thus, choosing between either depends on the characteristics of the dataset source. In this chapter, we will illustrate the following:

- ▶ How to train a support vector machine
- ▶ Observing how the choice of cost can affect the SVM classifier
- ▶ Visualizing the SVM fit
- ▶ Predicting the labels of a testing dataset based on the model trained by SVM
- ▶ Tuning the SVM

In the neural network section, we will cover:

- ▶ How to train a neural network
- ▶ How to visualize a neural network model
- ▶ Predicting the labels of a testing dataset based on a model trained by `neuralnet`
- ▶ Finally, we will show how to train a neural network with `nnet`, and how to use it to predict the labels of a testing dataset

## Classifying data with a support vector machine

The two most well known and popular support vector machine tools are `libsvm` and `SVMLite`. For R users, you can find the implementation of `libsvm` in the `e1071` package and `SVMLite` in the `klaR` package. Therefore, you can use the implemented function of these two packages to train support vector machines. In this recipe, we will focus on using the `svm` function (the `libsvm` implemented version) from the `e1071` package to train a support vector machine based on the telecom customer churn data training dataset.

### Getting ready

In this recipe, we will continue to use the telecom churn dataset as the input data source to train the support vector machine. For those who have not prepared the dataset, please refer to *Chapter, Classification (I) – Tree, Lazy, and Probabilistic*, for details.

## How to do it...

Perform the following steps to train the SVM:

1. Load the e1071 package:

```
> library(e1071)
```

2. Train the support vector machine using the `svm` function with `trainset` as the input dataset, and use `churn` as the classification category:

```
> model = svm(churn~, data = trainset, kernel="radial", cost=1,
gamma = 1/ncol(trainset))
```

3. Finally, you can obtain overall information about the built model with `summary`:

```
> summary(model)
```

**Call:**

```
svm(formula = churn ~ ., data = trainset, kernel = "radial", cost
= 1, gamma = 1/ncol(trainset))
```

**Parameters:**

```
SVM-Type: C-classification
SVM-Kernel: radial
cost: 1
gamma: 0.05882353
```

**Number of Support Vectors:** 691

```
(394 297)
```

**Number of Classes:** 2

**Levels:**

```
yes no
```

## How it works...

The support vector machine constructs a hyperplane (or set of hyperplanes) that maximize the margin width between two classes in a high dimensional space. In these, the cases that define the hyperplane are support vectors, as shown in the following figure:

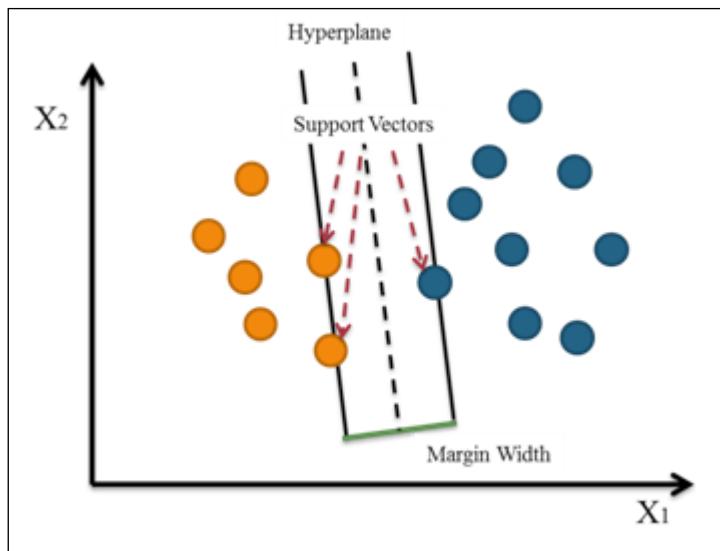


Figure 1: Support Vector Machine

Support vector machine starts from constructing a hyperplane that maximizes the margin width. Then, it extends the definition to a nonlinear separable problem. Lastly, it maps the data to a high dimensional space where the data can be more easily separated with a linear boundary.

The advantage of using SVM is that it builds a highly accurate model through an engineering problem-oriented kernel. Also, it makes use of the regularization term to avoid over-fitting. It also does not suffer from local optimal and multicollinearity. The main limitation of SVM is its speed and size in the training and testing time. Therefore, it is not suitable or efficient enough to construct classification models for data that is large in size. Also, since it is hard to interpret SVM, how does the determination of the kernel take place? Regularization is another problem that we need tackle.

In this recipe, we continue to use the telecom `churn` dataset as our example data source. We begin training a support vector machine using `libsvm` provided in the `e1071` package. Within the training function, `svm`, one can specify the `kernel` function, `cost`, and the `gamma` function. For the `kernel` argument, the default value is `radial`, and one can specify the `kernel` to a `linear`, `polynomial`, `radial basis`, and `sigmoid`. As for the `gamma` argument, the default value is equal to `(1/data dimension)`, and it controls the shape of the separating hyperplane. Increasing the `gamma` argument usually increases the number of support vectors.

As for the cost, the default value is set to 1, which indicates that the regularization term is constant, and the larger the value, the smaller the margin is. We will discuss more on how the cost can affect the SVM classifier in the next recipe. Once the support vector machine is built, the `summary` function can be used to obtain information, such as calls, parameters, number of classes, and the types of label.

## See also

Another popular support vector machine tool is `SVMLight`. Unlike the `e1071` package, which provides the full implementation of `libsvm`, the `klaR` package simply provides an interface to `SVMLight` only. To use `SVMLight`, one can perform the following steps:

1. Install the `klaR` package:

```
> install.packages("klaR")
> library(klaR)
```

2. Download the `SVMLight` source code and binary for your platform from <http://svmlight.joachims.org/>. For example, if your guest OS is Windows 64-bit, you should download the file from [http://download.joachims.org/svm\\_light/current/svm\\_light\\_windows64.zip](http://download.joachims.org/svm_light/current/svm_light_windows64.zip).
3. Then, you should unzip the file and put the workable binary in the working directory; you may check your working directory by using the `getwd` function:

```
> getwd()
```

4. Train the support vector machine using the `svmlight` function:

```
> model.light = svmlight(churn~, data = trainset,
 kernel="radial", cost=1, gamma = 1/ncol(trainset))
```

## Choosing the cost of a support vector machine

The support vector machines create an optimum hyperplane that separates the training data by the maximum margin. However, sometimes we would like to allow some misclassifications while separating categories. The SVM model has a cost function, which controls training errors and margins. For example, a small cost creates a large margin (a soft margin) and allows more misclassifications. On the other hand, a large cost creates a narrow margin (a hard margin) and permits fewer misclassifications. In this recipe, we will illustrate how the large and small cost will affect the SVM classifier.

# Getting ready

In this recipe, we will use the `iris` dataset as our example data source.

## How to do it...

Perform the following steps to generate two different classification examples with different costs:

1. Subset the `iris` dataset with columns named as `Sepal.Length`, `Sepal.Width`, `Species`, with species in `setosa` and `virginica`:

```
> iris.subset = subset(iris, select=c("Sepal.Length", "Sepal.Width", "Species"), Species %in% c("setosa", "virginica"))
```

2. Then, you can generate a scatter plot with `Sepal.Length` as the x-axis and the `Sepal.Width` as the y-axis:

```
> plot(x=iris.subset$Sepal.Length, y=iris.subset$Sepal.Width, col=iris.subset$Species, pch=19)
```

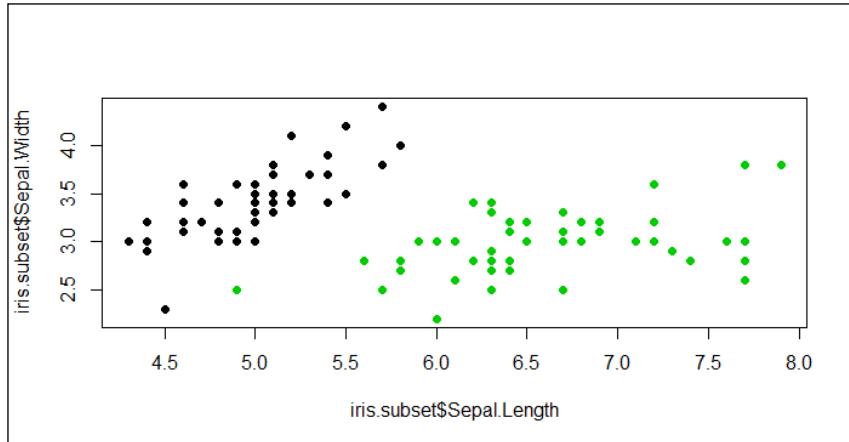


Figure 2: Scatter plot of Sepal.Length and Sepal.Width with subset of iris dataset

3. Next, you can train SVM based on `iris.subset` with the cost equal to 1:

```
> svm.model = svm(Species ~ ., data=iris.subset, kernel='linear', cost=1, scale=FALSE)
```

4. Then, we can circle the support vector with blue circles:

```
> points(iris.subset[svm.model$index,c(1,2)], col="blue", cex=2)
```

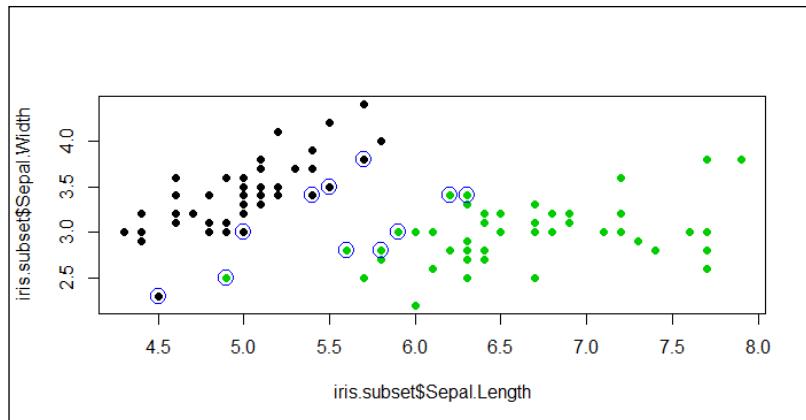


Figure 3: Circling support vectors with blue ring

5. Lastly, we can add a separation line on the plot:

```
> w = t(svm.model$coefs) %*% svm.model$SV
> b = -svm.model$rho
> abline(a=-b/w[1,2], b=-w[1,1]/w[1,2], col="red", lty=5)
```

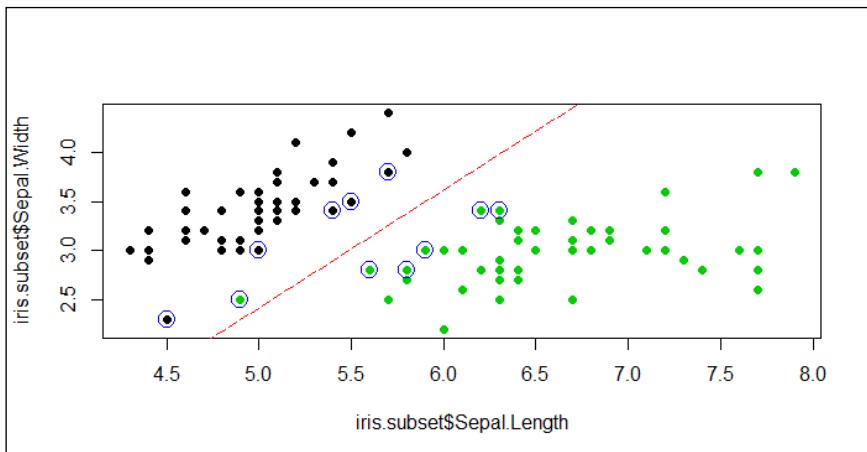


Figure 4: Add separation line to scatter plot

6. In addition to this, we create another SVM classifier where cost = 10,000:

```
> plot(x=iris.subset$Sepal.Length,y=iris.subset$Sepal.Width,
col=iris.subset$Species, pch=19)

> svm.model = svm(Species ~ ., data=iris.subset, type='C-
classification', kernel='linear', cost=10000, scale=FALSE)

> points(iris.subset[svm.model$index,c(1,2)], col="blue", cex=2)

> w = t(svm.model$coefs) %*% svm.model$SV

> b = -svm.model$rho

> abline(a=-b/w[1,2], b=-w[1,1]/w[1,2], col="red", lty=5)
```

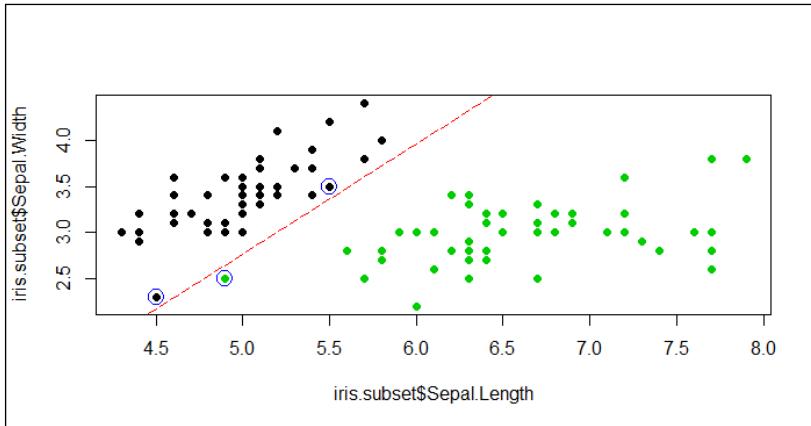


Figure 5: A classification example with large cost

## How it works...

In this recipe, we demonstrate how different costs can affect the SVM classifier. First, we create an iris subset with the columns, Sepal.Length, Sepal.Width, and Species containing the species, setosa and virginica. Then, in order to create a soft margin and allow some misclassification, we use an SVM with small cost (where cost = 1) to train the support of the vector machine. Next, we circle the support vectors with blue circles and add the separation line. As per Figure 5, one of the green points (virginica) is misclassified (it is classified to setosa) to the other side of the separation line due to the choice of the small cost.

In addition to this, we would like to determine how a large cost can affect the SVM classifier. Therefore, we choose a large cost (where cost = 10,000). From Figure 5, we can see that the margin created is narrow (a hard margin) and no misclassification cases are present. As a result, the two examples show that the choice of different costs may affect the margin created and also affect the possibilities of misclassification.

## See also

- The idea of soft margin, which allows misclassified examples, was suggested by Corinna Cortes and Vladimir N. Vapnik in 1995 in the following paper: Cortes, C., and Vapnik, V. (1995). *Support-vector networks*. *Machine learning*, 20(3), 273-297.

# Visualizing an SVM fit

To visualize the built model, one can first use the plot function to generate a scatter plot of data input and the SVM fit. In this plot, support vectors and classes are highlighted through the color symbol. In addition to this, one can draw a contour filled plot of the class regions to easily identify misclassified samples from the plot.

## Getting ready

In this recipe, we will use two datasets: the `iris` dataset and the `telecom churn` dataset. For the `telecom churn` dataset, one needs to have completed the previous recipe by training a support vector machine with SVM, and to have saved the SVM fit model.

## How to do it...

Perform the following steps to visualize the SVM fit object:

1. Use SVM to train the support vector machine based on the `iris` dataset, and use the `plot` function to visualize the fitted model:

```
> data(iris)
> model.iris = svm(Species~, iris)
> plot(model.iris, iris, Petal.Width ~ Petal.Length, slice =
list(Sepal.Width = 3, Sepal.Length = 4))
```

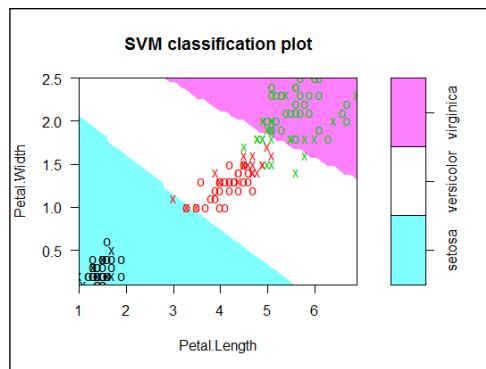


Figure 6: The SVM classification plot of trained SVM fit based on iris dataset

2. Visualize the SVM fit object, `model`, using the `plot` function with the dimensions of `total_day_minutes` and `total_intl_charge`:

```
> plot(model, trainset, total_day_minutes ~ total_intl_charge)
```

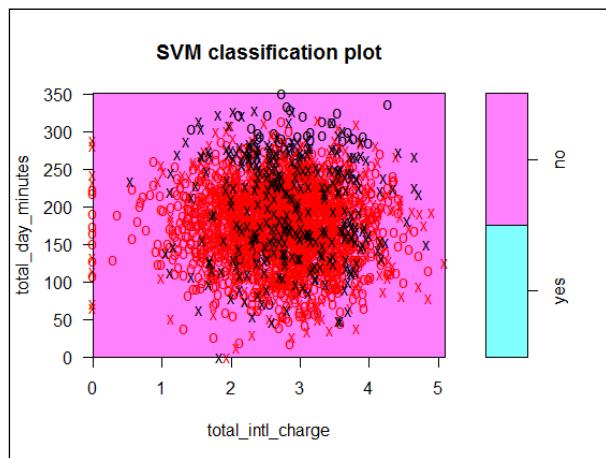


Figure 7: The SVM classification plot of trained SVM fit based on churn dataset

## How it works...

In this recipe, we demonstrate how to use the `plot` function to visualize the SVM fit. In the first plot, we train a support vector machine using the `iris` dataset. Then, we use the `plot` function to visualize the fitted SVM.

In the argument list, we specify the fitted model in the first argument and the dataset (this should be the same data used to build the model) as the second parameter. The third parameter indicates the dimension used to generate the classification plot. By default, the `plot` function can only generate a scatter plot based on two dimensions (for the x-axis and y-axis). Therefore, we select the variables, `Petal.Length` and `Petal.Width` as the two dimensions to generate the scatter plot.

From *Figure 6*, we find `Petal.Length` assigned to the x-axis, `Petal.Width` assigned to the y-axis, and data points with `x` and `o` symbols scattered on the plot. Within the scatter plot, the `x` symbol shows the support vector and the `o` symbol represents the data points. These two symbols can be altered through the configuration of the `svSymbol` and `dataSymbol` options. Both the support vectors and true classes are highlighted and colored depending on their label (green refers to `viginica`, red refers to `versicolor`, and black refers to `setosa`). The last argument, `slice`, is set when there are more than two variables. Therefore, in this example, we use the additional variables, `Sepal.width` and `Sepal.length`, by assigning a constant of 3 and 4.

Next, we take the same approach to draw the SVM fit based on customer churn data. In this example, we use `total_day_minutes` and `total_intl_charge` as the two dimensions used to plot the scatterplot. As per *Figure 7*, the support vectors and data points in red and black are scattered closely together in the central region of the plot, and there is no simple way to separate them.

## See also

- ▶ There are other parameters, such as `fill`, `grid`, `symbolPalette`, and so on, that can be configured to change the layout of the plot. You can use the `help` function to view the following document for further information:

```
> ?svm.plot
```

# Predicting labels based on a model trained by a support vector machine

In the previous recipe, we trained an SVM based on the training dataset. The training process finds the optimum hyperplane that separates the training data by the maximum margin. We can then utilize the SVM fit to predict the label (category) of new observations. In this recipe, we will demonstrate how to use the `predict` function to predict values based on a model trained by SVM.

## Getting ready

You need to have completed the previous recipe by generating a fitted SVM, and save the fitted model in `model`.

## How to do it...

Perform the following steps to predict the labels of the testing dataset:

1. Predict the label of the testing dataset based on the fitted SVM and attributes of the testing dataset:

```
> svm.pred = predict(model, testset[, !names(testset) %in%
c("churn")])
```

2. Then, you can use the `table` function to generate a classification table with the prediction result and labels of the testing dataset:

```
> svm.table=table(svm.pred, testset$churn)
> svm.table
```

```
svm.pred yes no
yes 70 12
no 71 865
```

3. Next, you can use `classAgreement` to calculate coefficients compared to the classification agreement:

```
> classAgreement(svm.table)
```

```
$diag
```

```
[1] 0.9184676
```

```
$kappa
```

```
[1] 0.5855903
```

```
$rand
```

```
[1] 0.850083
```

```
$crand
```

```
[1] 0.5260472
```

4. Now, you can use `confusionMatrix` to measure the prediction performance based on the classification table:

```
> library(caret)
```

```
> confusionMatrix(svm.table)
```

```
Confusion Matrix and Statistics
```

```
svm.pred yes no
yes 70 12
no 71 865
```

```
Accuracy : 0.9185
```

```
95% CI : (0.8999, 0.9345)
```

```
No Information Rate : 0.8615
```

```
P-Value [Acc > NIR] : 1.251e-08
```

```
Kappa : 0.5856
```

```
McNemar's Test P-Value : 1.936e-10
```

```
Sensitivity : 0.49645
Specificity : 0.98632
Pos Pred Value : 0.85366
Neg Pred Value : 0.92415
Prevalence : 0.13851
Detection Rate : 0.06876
Detection Prevalence : 0.08055
Balanced Accuracy : 0.74139
```

'Positive' Class : yes

## How it works...

In this recipe, we first used the `predict` function to obtain the predicted labels of the testing dataset. Next, we used the `table` function to generate the classification table based on the predicted labels of the testing dataset. So far, the evaluation procedure is very similar to the evaluation process mentioned in the previous chapter.

We then introduced a new function, `classAgreement`, which computes several coefficients of agreement between the columns and rows of a two-way contingency table. The coefficients include `diag`, `kappa`, `rand`, and `crand`. The `diag` coefficient represents the percentage of data points in the main diagonal of the classification table, `kappa` refers to `diag`, which is corrected for an agreement by chance (the probability of random agreements), `rand` represents the Rand index, which measures the similarity between two data clusters, and `crand` indicates the Rand index, which is adjusted for the chance grouping of elements.

Finally, we used `confusionMatrix` from the `caret` package to measure the performance of the classification model. The accuracy of 0.9185 shows that the trained support vector machine can correctly classify most of the observations. However, accuracy alone is not a good measurement of a classification model. One should also reference sensitivity and specificity.

## There's more...

Besides using SVM to predict the category of new observations, you can use SVM to predict continuous values. In other words, one can use SVM to perform regression analysis.

In the following example, we will show how to perform a simple regression prediction based on a fitted SVM with the type specified as `eps-regression`.

Perform the following steps to train a regression model with SVM:

1. Train a support vector machine based on a Quartet dataset:

```
> library(car)
> data(Quartet)
> model.regression = svm(Quartet$y1~Quartet$x, type="eps-
regression")
```

2. Use the predict function to obtain prediction results:

```
> predict.y = predict(model.regression, Quartet$x)
> predict.y
```

|          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|
| 1        | 2        | 3        | 4        | 5        | 6        | 7        |
| 8        |          |          |          |          |          |          |
| 8.196894 | 7.152946 | 8.807471 | 7.713099 | 8.533578 | 8.774046 | 6.186349 |
| 5.763689 |          |          |          |          |          |          |
| 9        | 10       | 11       |          |          |          |          |
| 8.726925 | 6.621373 | 5.882946 |          |          |          |          |

3. Plot the predicted points as squares and the training data points as circles on the same plot:

```
> plot(Quartet$x, Quartet$y1, pch=19)
> points(Quartet$x, predict.y, pch=15, col="red")
```

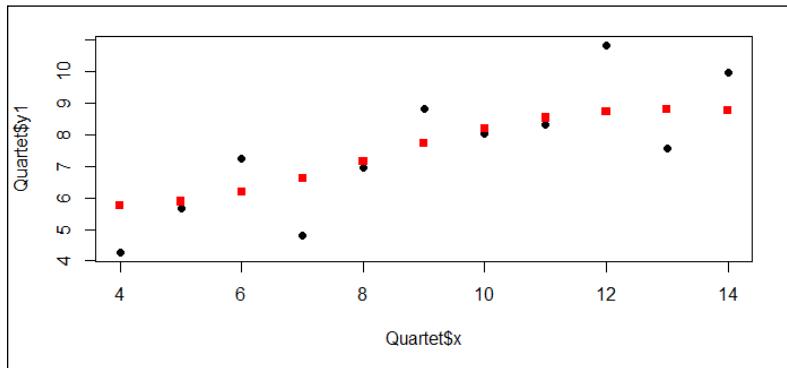


Figure 8: The scatter plot contains predicted data points and training data points

# Tuning a support vector machine

Besides using different feature sets and the `kernel` function in support vector machines, one trick that you can use to tune its performance is to adjust the gamma and cost configured in the argument. One possible approach to test the performance of different gamma and cost combination values is to write a `for` loop to generate all the combinations of gamma and cost as inputs to train different support vector machines. Fortunately, SVM provides a tuning function, `tune.svm`, which makes the tuning much easier. In this recipe, we will demonstrate how to tune a support vector machine through the use of `tune.svm`.

## Getting ready

You need to have completed the previous recipe by preparing a training dataset, `trainset`.

## How to do it...

Perform the following steps to tune the support vector machine:

1. First, tune the support vector machine using `tune.svm`:

```
> tuned = tune.svm(churn~, data = trainset, gamma = 10^(-6:-1),
cost = 10^(1:2))
```

2. Next, you can use the `summary` function to obtain the tuning result:

```
> summary(tuned)
```

**Parameter tuning of 'svm':**

- sampling method: 10-fold cross validation

- best parameters:

gamma cost

0.01 100

- best performance: 0.08077885

- Detailed performance results:

| gamma | cost | error | dispersion |
|-------|------|-------|------------|
|-------|------|-------|------------|

|   |       |    |            |            |
|---|-------|----|------------|------------|
| 1 | 1e-06 | 10 | 0.14774780 | 0.02399512 |
|---|-------|----|------------|------------|

|   |       |    |            |            |
|---|-------|----|------------|------------|
| 2 | 1e-05 | 10 | 0.14774780 | 0.02399512 |
|---|-------|----|------------|------------|

```
3 1e-04 10 0.14774780 0.02399512
4 1e-03 10 0.14774780 0.02399512
5 1e-02 10 0.09245223 0.02046032
6 1e-01 10 0.09202306 0.01938475
7 1e-06 100 0.14774780 0.02399512
8 1e-05 100 0.14774780 0.02399512
9 1e-04 100 0.14774780 0.02399512
10 1e-03 100 0.11794484 0.02368343
11 1e-02 100 0.08077885 0.01858195
12 1e-01 100 0.12356135 0.01661508
```

3. After retrieving the best performance parameter from tuning the result, you can retrain the support vector machine with the best performance parameter:

```
> model.tuned = svm(churn~, data = trainset, gamma = tuned$best.parameters$gamma, cost = tuned$best.parameters$cost)
> summary(model.tuned)
```

Call:

```
svm(formula = churn ~ ., data = trainset, gamma = 10^-2, cost =
100)
```

Parameters:

```
SVM-Type: C-classification
SVM-Kernel: radial
cost: 100
gamma: 0.01
```

Number of Support Vectors: 547

```
(304 243)
```

Number of Classes: 2

Levels:

```
yes no
```

4. Then, you can use the predict function to predict labels based on the fitted SVM:
- ```
> svm.tuned.pred = predict(model.tuned, testset[, !names(testset)
%in% c("churn")])
```

5. Next, generate a classification table based on the predicted and original labels of the testing dataset:

```
> svm.tuned.table=table(svm.tuned.pred, testset$churn)
> svm.tuned.table
```

```
svm.tuned.pred yes no
yes    95   24
no     46 853
```

6. Also, generate a class agreement to measure the performance:

```
> classAgreement(svm.tuned.table)
$diag
[1] 0.9312377
```

```
$kappa
[1] 0.691678
```

```
$rand
[1] 0.871806
```

```
$crand
[1] 0.6303615
```

7. Finally, you can use a confusion matrix to measure the performance of the retrained model:

```
> confusionMatrix(svm.tuned.table)
Confusion Matrix and Statistics
```

```
svm.tuned.pred yes no
yes    95   24
no     46 853
```

Accuracy : 0.9312

95% CI : (0.9139, 0.946)

No Information Rate : 0.8615

P-Value [Acc > NIR] : 1.56e-12

Kappa : 0.6917

Mcnemar's Test P-Value : 0.01207

Sensitivity : 0.67376

Specificity : 0.97263

Pos Pred Value : 0.79832

Neg Pred Value : 0.94883

Prevalence : 0.13851

Detection Rate : 0.09332

Detection Prevalence : 0.11690

Balanced Accuracy : 0.82320

'Positive' Class : yes

How it works...

To tune the support vector machine, you can use a trial and error method to find the best gamma and cost parameters. In other words, one has to generate a variety of combinations of gamma and cost for the purpose of training different support vector machines.

In this example, we generate different gamma values from 10^{-6} to 10^{-1} , and cost with a value of either 10 or 100. Therefore, you can use the tuning function, `svm.tune`, to generate 12 sets of parameters. The function then makes 10 cross-validations and outputs the error dispersion of each combination. As a result, the combination with the least error dispersion is regarded as the best parameter set. From the summary table, we found that gamma with a value of 0.01 and cost with a value of 100 are the best parameters for the SVM fit.

After obtaining the best parameters, we can then train a new support vector machine with gamma equal to 0.01 and cost equal to 100. Additionally, we can obtain a classification table based on the predicted labels and labels of the testing dataset. We can also obtain a confusion matrix from the classification table. From the output of the confusion matrix, you can determine the accuracy of the newly trained model in comparison to the original model.

See also

- ▶ For more information about how to tune SVM with `svm.tune`, you can use the `help` function to access this document:

```
> ?svm.tune
```

Training a neural network with neuralnet

The neural network is constructed with an interconnected group of nodes, which involves the input, connected weights, processing element, and output. Neural networks can be applied to many areas, such as classification, clustering, and prediction. To train a neural network in R, you can use `neuralnet`, which is built to train multilayer perceptron in the context of regression analysis, and contains many flexible functions to train forward neural networks. In this recipe, we will introduce how to use `neuralnet` to train a neural network.

Getting ready

In this recipe, we will use an `iris` dataset as our example dataset. We will first split the `iris` dataset into a training and testing datasets, respectively.

How to do it...

Perform the following steps to train a neural network with `neuralnet`:

1. First load the `iris` dataset and split the data into training and testing datasets:

```
> data(iris)
> ind = sample(2, nrow(iris), replace = TRUE, prob=c(0.7, 0.3))
> trainset = iris[ind == 1,]
> testset = iris[ind == 2,]
```

2. Then, install and load the `neuralnet` package:

```
> install.packages("neuralnet")
> library(neuralnet)
```

3. Add the columns `versicolor`, `setosa`, and `virginica` based on the name matched value in the `Species` column:

```
> trainset$setosa = trainset$Species == "setosa"
> trainset$virginica = trainset$Species == "virginica"
> trainset$versicolor = trainset$Species == "versicolor"
```

4. Next, train the neural network with the `neuralnet` function with three hidden neurons in each layer. Notice that the results may vary with each training, so you might not get the same result. However, you can use `set.seed` at the beginning, so you can get the same result in every training process

```
> network = neuralnet(versicolor + virginica + setosa ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width, trainset, hidden=3)

> network

Call: neuralnet(formula = versicolor + virginica + setosa ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width, data = trainset, hidden = 3)
```

1 repetition was calculated.

Error Reached Threshold Steps

```
1 0.8156100175 0.009994274769 11063
```

5. Now, you can view the summary information by accessing the `result.matrix` attribute of the built neural network model:

```
> network$result.matrix
```

	1
error	0.815610017474
reached.threshold	0.009994274769
steps	11063.000000000000
Intercept.to.1layhid1	1.686593311644
Sepal.Length.to.1layhid1	0.947415215237
Sepal.Width.to.1layhid1	-7.220058260187
Petal.Length.to.1layhid1	1.790333443486
Petal.Width.to.1layhid1	9.943109233330
Intercept.to.1layhid2	1.411026063895
Sepal.Length.to.1layhid2	0.240309549505
Sepal.Width.to.1layhid2	0.480654059973
Petal.Length.to.1layhid2	2.221435192437
Petal.Width.to.1layhid2	0.154879347818
Intercept.to.1layhid3	24.399329878242
Sepal.Length.to.1layhid3	3.313958088512
Sepal.Width.to.1layhid3	5.845670010464
Petal.Length.to.1layhid3	-6.337082722485
Petal.Width.to.1layhid3	-17.990352566695
Intercept.to.versicolor	-1.959842102421
1layhid.1.to.versicolor	1.010292389835

1layhid.2.to.versicolor	0.936519720978
1layhid.3.to.versicolor	1.023305801833
Intercept.to.virginica	-0.908909982893
1layhid.1.to.virginica	-0.009904635231
1layhid.2.to.virginica	1.931747950462
1layhid.3.to.virginica	-1.021438938226
Intercept.to.setosa	1.500533827729
1layhid.1.to.setosa	-1.001683936613
1layhid.2.to.setosa	-0.498758815934
1layhid.3.to.setosa	-0.001881935696

6. Lastly, you can view the generalized weight by accessing it in the network:

```
> head(network$generalized.weights[[1]])
```

How it works...

The neural network is a network made up of artificial neurons (or nodes). There are three types of neurons within the network: input neurons, hidden neurons, and output neurons. In the network, neurons are connected; the connection strength between neurons is called weights. If the weight is greater than zero, it is in an excitation status. Otherwise, it is in an inhibition status. Input neurons receive the input information; the higher the input value, the greater the activation. Then, the activation value is passed through the network in regard to weights and transfer functions in the graph. The hidden neurons (or output neurons) then sum up the activation values and modify the summed values with the transfer function. The activation value then flows through hidden neurons and stops when it reaches the output nodes. As a result, one can use the output value from the output neurons to classify the data.

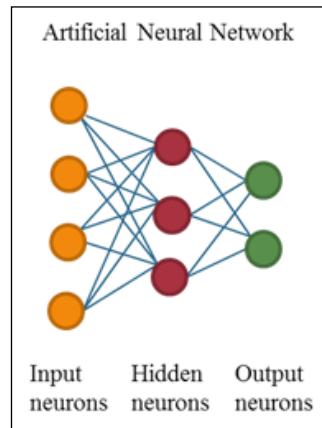


Figure 9: Artificial Neural Network

The advantages of a neural network are: first, it can detect nonlinear relationships between the dependent and independent variable. Second, one can efficiently train large datasets using the parallel architecture. Third, it is a nonparametric model so that one can eliminate errors in the estimation of parameters. The main disadvantages of a neural network are that it often converges to the local minimum rather than the global minimum. Also, it might over-fit when the training process goes on for too long.

In this recipe, we demonstrate how to train a neural network. First, we split the `iris` dataset into training and testing datasets, and then install the `neuralnet` package and load the library into an R session. Next, we add the columns `versicolor`, `setosa`, and `virginica` based on the name matched value in the `Species` column, respectively. We then use the `neuralnet` function to train the network model. Besides specifying the label (the column where the name equals to `versicolor`, `virginica`, and `setosa`) and training attributes in the function, we also configure the number of hidden neurons (vertices) as three in each layer.

Then, we examine the basic information about the training process and the trained network saved in the network. From the output message, it shows the training process needed 11,063 steps until all the absolute partial derivatives of the error function were lower than 0.01 (specified in the threshold). The error refers to the likelihood of calculating **Akaike**

Information Criterion (AIC). To see detailed information on this, you can access the `result`. matrix of the built neural network to see the estimated weight. The output reveals that the estimated weight ranges from -18 to 24.40; the intercepts of the first hidden layer are 1.69, 1.41 and 24.40, and the two weights leading to the first hidden neuron are estimated as 0.95 (`Sepal.Length`), -7.22 (`Sepal.Width`), 1.79 (`Petal.Length`), and 9.94 (`Petal.Width`). We can lastly determine that the trained neural network information includes generalized weights, which express the effect of each covariate. In this recipe, the model generates 12 generalized weights, which are the combination of four covariates (`Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`) to three responses (`setosa`, `virginica`, `versicolor`).

See also

- ▶ For a more detailed introduction on `neuralnet`, one can refer to the following paper: Günther, F., and Fritsch, S. (2010). *neuralnet: Training of neural networks*. *The R journal*, 2(1), 30-38.

Visualizing a neural network trained by neuralnet

The package, `neuralnet`, provides the `plot` function to visualize a built neural network and the `gwpplot` function to visualize generalized weights. In following recipe, we will cover how to use these two functions.

Getting ready

You need to have completed the previous recipe by training a neural network and have all basic information saved in the network.

How to do it...

Perform the following steps to visualize the neural network and the generalized weights:

1. You can visualize the trained neural network with the `plot` function:

```
> plot(network)
```

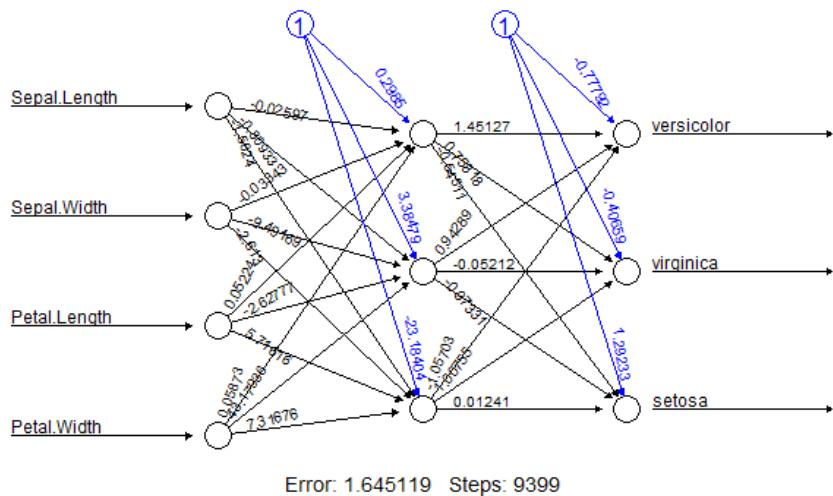


Figure 10: The plot of the trained neural network

2. Furthermore, you can use gwplot to visualize the generalized weights:

```
> par(mfrow=c(2, 2))  
> gwplot(network, selected.covariate="Petal.Width")  
> gwplot(network, selected.covariate="Sepal.Width")  
> gwplot(network, selected.covariate="Petal.Length")  
> gwplot(network, selected.covariate="Petal.Width")
```

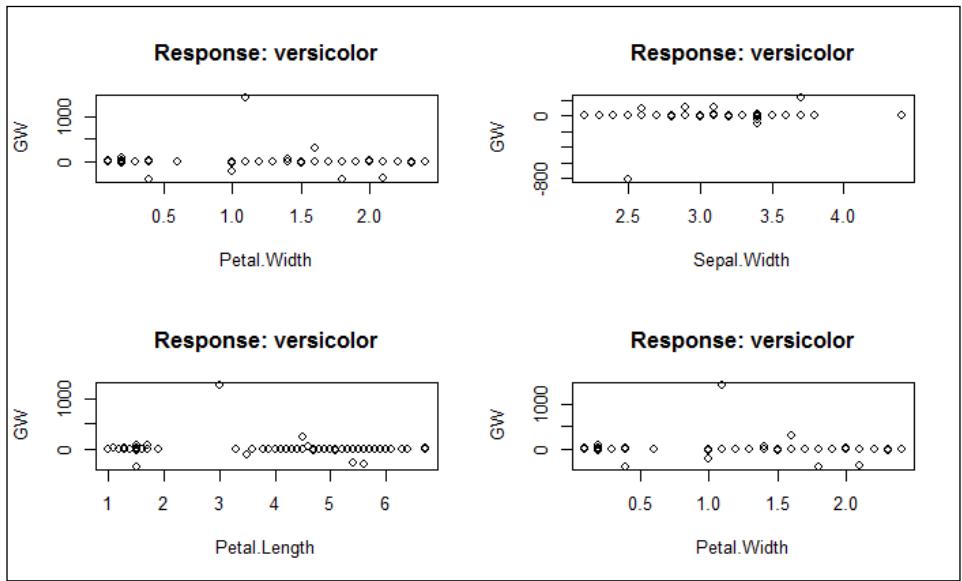


Figure 11: The plot of generalized weights

How it works...

In this recipe, we demonstrate how to visualize the trained neural network and the generalized weights of each trained attribute. As per *Figure 10*, the plot displays the network topology of the trained neural network. Also, the plot includes the estimated weight, intercepts and basic information about the training process. At the bottom of the figure, one can find the overall error and number of steps required to converge.

Figure 11 presents the generalized weight plot in regard to `network$generalized.weights`. The four plots in *Figure 11* display the four covariates: `Petal.Width`, `Sepal.Width`, `Petal.Length`, and `Petal.Width`, in regard to the `versicolor` response. If all the generalized weights are close to zero on the plot, it means the covariate has little effect. However, if the overall variance is greater than one, it means the covariate has a nonlinear effect.

See also

- ▶ For more information about `gwplot`, one can use the `help` function to access the following document:

```
> ?gwplot
```

Predicting labels based on a model trained by neuralnet

Similar to other classification methods, we can predict the labels of new observations based on trained neural networks. Furthermore, we can validate the performance of these networks through the use of a confusion matrix. In the following recipe, we will introduce how to use the `compute` function in a neural network to obtain a probability matrix of the testing dataset labels, and use a table and confusion matrix to measure the prediction performance.

Getting ready

You need to have completed the previous recipe by generating the training dataset, `trainset`, and the testing dataset, `testset`. The trained neural network needs to be saved in the network.

How to do it...

Perform the following steps to measure the prediction performance of the trained neural network:

1. First, generate a prediction probability matrix based on a trained neural network and the testing dataset, `testset`:

```
> net.predict = compute(network, testset[-5])$net.result
```

2. Then, obtain other possible labels by finding the column with the greatest probability:

```
> net.prediction = c("versicolor", "virginica", "setosa")
[apply(net.predict, 1, which.max)]
```

3. Generate a classification table based on the predicted labels and the labels of the testing dataset:

```
> predict.table = table(testset$Species, net.prediction)
```

```
> predict.table
```

```
prediction
```

```
setosa versicolor virginica
```

setosa	20	0	0
versicolor	0	19	1
virginica	0	2	16

4. Next, generate classAgreement from the classification table:

```
> classAgreement(predict.table)
```

```
$diag
```

```
[1] 0.94444444444
```

```
$kappa
```

```
[1] 0.9154488518
```

```
$rand
```

```
[1] 0.9224318658
```

```
$crand
```

```
[1] 0.8248251737
```

5. Finally, use confusionMatrix to measure the prediction performance:

```
> confusionMatrix(predict.table)
```

```
Confusion Matrix and Statistics
```

	prediction		
	setosa	versicolor	virginica
setosa	20	0	0
versicolor	0	19	1
virginica	0	2	16

```
Overall Statistics
```

```
Accuracy : 0.9482759
```

```
95% CI : (0.8561954, 0.9892035)
```

```
No Information Rate : 0.362069
```

```
P-Value [Acc > NIR] : < 0.000000000000000022204
```

```
Kappa : 0.922252
```

```
McNemar's Test P-Value : NA
```

Statistics by Class:

	Class: setosa	Class: versicolor	Class: virginica
Sensitivity	1.0000000	0.9047619	0.9411765
Specificity	1.0000000	0.9729730	0.9512195
Pos Pred Value	1.0000000	0.9500000	0.8888889
Neg Pred Value	1.0000000	0.9473684	0.9750000
Prevalence	0.3448276	0.3620690	0.2931034
Detection Rate	0.3448276	0.3275862	0.2758621
Detection Prevalence	0.3448276	0.3448276	0.3103448
Balanced Accuracy	1.0000000	0.9388674	0.9461980

How it works...

In this recipe, we demonstrate how to predict labels based on a model trained by neuralnet. Initially, we use the compute function to create an output probability matrix based on the trained neural network and the testing dataset. Then, to convert the probability matrix to class labels, we use the which.max function to determine the class label by selecting the column with the maximum probability within the row. Next, we use a table to generate a classification matrix based on the labels of the testing dataset and the predicted labels. As we have created the classification table, we can employ a confusion matrix to measure the prediction performance of the built neural network.

See also

- In this recipe, we use the `net.result` function, which is the overall result of the neural network, used to predict the labels of the testing dataset. Apart from examining the overall result by accessing `net.result`, the `compute` function also generates the output from neurons in each layer. You can examine the output of neurons to get a better understanding of how `compute` works:

```
> compute(network, testset[-5])
```

Training a neural network with nnet

The `nnet` package is another package that can deal with artificial neural networks. This package provides the functionality to train feed-forward neural networks with traditional back propagation. As you can find most of the neural network function implemented in the `neuralnet` package, in this recipe we provide a short overview of how to train neural networks with `nnet`.

Getting ready

In this recipe, we do not use the `trainset` and `testset` generated from the previous step; please reload the `iris` dataset again.

How to do it...

Perform the following steps to train the neural network with `nnet`:

1. First, install and load the `nnet` package:

```
> install.packages("nnet")
> library(nnet)
```

2. Next, split the dataset into training and testing datasets:

```
> data(iris)
> set.seed(2)
> ind = sample(2, nrow(iris), replace = TRUE, prob=c(0.7, 0.3))
> trainset = iris[ind == 1,]
> testset = iris[ind == 2,]
```

3. Then, train the neural network with `nnet`:

```
> iris.nn = nnet(Species ~ ., data = trainset, size = 2, rang =
0.1, decay = 5e-4, maxit = 200)
# weights:  19
initial  value 165.086674
iter   10 value 70.447976
iter   20 value 69.667465
iter   30 value 69.505739
iter   40 value 21.588943
iter   50 value 8.691760
iter   60 value 8.521214
iter   70 value 8.138961
```

```
iter  80 value 7.291365
iter  90 value 7.039209
iter 100 value 6.570987
iter 110 value 6.355346
iter 120 value 6.345511
iter 130 value 6.340208
iter 140 value 6.337271
iter 150 value 6.334285
iter 160 value 6.333792
iter 170 value 6.333578
iter 180 value 6.333498
final  value 6.333471
converged
```

4. Use the `summary` to obtain information about the trained neural network:

```
> summary(iris.nn)
a 4-2-3 network with 19 weights
options were - softmax modelling decay=0.0005
b->h1 i1->h1 i2->h1 i3->h1 i4->h1
-0.38 -0.63 -1.96 3.13 1.53
b->h2 i1->h2 i2->h2 i3->h2 i4->h2
 8.95 0.52 1.42 -1.98 -3.85
b->o1 h1->o1 h2->o1
 3.08 -10.78 4.99
b->o2 h1->o2 h2->o2
-7.41 6.37 7.18
b->o3 h1->o3 h2->o3
 4.33 4.42 -12.16
```

How it works...

In this recipe, we demonstrate steps to train a neural network model with the `nnet` package. We first use `nnet` to train the neural network. With this function, we can set the classification formula, source of data, number of hidden units in the `size` parameter, initial random weight in the `rang` parameter, parameter for weight decay in the `decay` parameter, and the maximum iteration in the `maxit` parameter. As we set `maxit` to 200, the training process repeatedly runs till the value of the fitting criterion plus the decay term converge. Finally, we use the `summary` function to obtain information about the built neural network, which reveals that the model is built with 4-2-3 networks with 19 weights. Also, the model shows a list of weight transitions from one node to another at the bottom of the printed message.

See also

For those who are interested in the background theory of `nnet` and how it is made, please refer to the following articles:

- ▶ Ripley, B. D. (1996) *Pattern Recognition and Neural Networks*. Cambridge
- ▶ Venables, W. N., and Ripley, B. D. (2002). *Modern applied statistics with S*. Fourth edition. Springer

Predicting labels based on a model trained by `nnet`

As we have trained a neural network with `nnet` in the previous recipe, we can now predict the labels of the testing dataset based on the trained neural network. Furthermore, we can assess the model with a confusion matrix adapted from the `caret` package.

Getting ready

You need to have completed the previous recipe by generating the training dataset, `trainset`, and the testing dataset, `testset`, from the `iris` dataset. The trained neural network also needs to be saved as `iris.nn`.

How to do it...

Perform the following steps to predict labels based on the trained neural network:

1. Generate the predictions of the testing dataset based on the model, `iris.nn`:

```
> iris.predict = predict(iris.nn, testset, type="class")
```
2. Generate a classification table based on the predicted labels and labels of the testing dataset:

```
> nn.table = table(testset$Species, iris.predict)
```

	setosa	versicolor	virginica
setosa	17	0	0
versicolor	0	14	0
virginica	0	1	14

3. Lastly, generate a confusion matrix based on the classification table:

```
> confusionMatrix(nn.table)
Confusion Matrix and Statistics

iris.predict
             setosa versicolor virginica
setosa          17        0        0
versicolor       0       14        0
virginica        0        1       14
```

Overall Statistics

Accuracy : 0.9782609

95% CI : (0.8847282, 0.9994498)

No Information Rate : 0.3695652

P-Value [Acc > NIR] : < 0.00000000000000022204

Kappa : 0.9673063

McNemar's Test P-Value : NA

Statistics by Class:

	Class: setosa	Class: versicolor
Sensitivity	1.0000000	0.9333333
Specificity	1.0000000	1.0000000
Pos Pred Value	1.0000000	1.0000000
Neg Pred Value	1.0000000	0.9687500
Prevalence	0.3695652	0.3260870
Detection Rate	0.3695652	0.3043478
Detection Prevalence	0.3695652	0.3043478
Balanced Accuracy	1.0000000	0.9666667

	Class: virginica
Sensitivity	1.0000000
Specificity	0.9687500
Pos Pred Value	0.9333333

Neg Pred Value	1.0000000
Prevalence	0.3043478
Detection Rate	0.3043478
Detection Prevalence	0.3260870
Balanced Accuracy	0.9843750

How it works...

Similar to other classification methods, one can also predict labels based on the neural networks trained by `nnet`. First, we use the `predict` function to generate the predicted labels based on a testing dataset, `testset`. Within the `predict` function, we specify the `type` argument to the `class`, so the output will be class labels instead of a probability matrix. Next, we use the `table` function to generate a classification table based on predicted labels and labels written in the testing dataset. Finally, as we have created the classification table, we can employ a confusion matrix from the `caret` package to measure the prediction performance of the trained neural network.

See also

- ▶ For the `predict` function, if the `type` argument to `class` is not specified, by default, it will generate a probability matrix as a prediction result, which is very similar to `net.result` generated from the `compute` function within the `neuralnet` package:

```
> head(predict(iris.nn, testset))
```

6

Model Evaluation

In this chapter, we will cover the following topics:

- ▶ Estimating model performance with k-fold cross-validation
- ▶ Performing cross-validation with the e1071 package
- ▶ Performing cross-validation with the caret package
- ▶ Ranking the variable importance with the caret package
- ▶ Ranking the variable importance with the rminer package
- ▶ Finding highly correlated features with the caret package
- ▶ Selecting features using the caret package
- ▶ Measuring the performance of a regression model
- ▶ Measuring the prediction performance with the confusion matrix
- ▶ Measuring the prediction performance using ROCR
- ▶ Comparing an ROC curve using the caret package
- ▶ Measuring performance differences between models with the caret package

Introduction

Model evaluation is performed to ensure that a fitted model can accurately predict responses for future or unknown subjects. Without model evaluation, we might train models that over-fit in the training data. To prevent overfitting, we can employ packages, such as `caret`, `rminer`, and `rocr` to evaluate the performance of the fitted model. Furthermore, model evaluation can help select the optimum model, which is more robust and can accurately predict responses for future subjects.

In the following chapter, we will discuss how one can implement a simple R script or use one of the packages (for example, caret or rminer) to evaluate the performance of a fitted model.

Estimating model performance with k-fold cross-validation

The k-fold cross-validation technique is a common technique used to estimate the performance of a classifier as it overcomes the problem of over-fitting. For k-fold cross-validation, the method does not use the entire dataset to build the model, instead it splits the data into a training dataset and a testing dataset. Therefore, the model built with a training dataset can then be used to assess the performance of the model on the testing dataset. By performing n repeats of the k-fold validation, we can then use the average of n accuracies to truly assess the performance of the built model. In this recipe, we will illustrate how to perform a k-fold cross-validation.

Getting ready

In this recipe, we will continue to use the telecom churn dataset as the input data source to train the support vector machine. For those who have not prepared the dataset, please refer to *Chapter, Classification (I) – Tree, Lazy, and Probabilistic*, for detailed information.

How to do it...

Perform the following steps to cross-validate the telecom churn dataset:

1. Split the index into 10 fold using the cut function:

```
> ind = cut(1:nrow(churnTrain), breaks=10, labels=F)
```

2. Next, use for loop to perform a 10 fold cross-validation, repeated 10 times:

```
> accuracies = c()
> for (i in 1:10) {
+   fit = svm(churn ~., churnTrain[ind != i,])
+   predictions = predict(fit, churnTrain[ind == i, !
+ names(churnTrain) %in% c("churn")))
+   correct_count = sum(predictions == churnTrain[ind ==
+ i,c("churn")])
+   accuracies = append(correct_count / nrow(churnTrain[ind ==
+ i,]), accuracies)
+ }
```

3. You can then print the accuracies:

```
> accuracies  
[1] 0.9341317 0.8948949 0.8978979 0.9459459 0.9219219 0.9281437  
0.9219219 0.9249249 0.9189189 0.9251497
```

4. Lastly, you can generate average accuracies with the `mean` function:

```
> mean(accuracies)  
[1] 0.9213852
```

How it works...

In this recipe, we implement a simple script performing 10-fold cross-validations. We first generate an index with 10 fold with the `cut` function. Then, we implement a `for` loop to perform a 10-fold cross-validation 10 times. Within the loop, we first apply `svm` on 9 folds of data as the training set. We then use the fitted model to predict the label of the rest of the data (the testing dataset). Next, we use the sum of the correctly predicted labels to generate the accuracy. As a result of this, the loop stores 10 generated accuracies. Finally, we use the `mean` function to retrieve the average of the accuracies.

There's more...

If you wish to perform the k-fold validation with the use of other models, simply replace the line to generate the variable `fit` to whatever classifier you prefer. For example, if you would like to assess the Naïve Bayes model with a 10-fold cross-validation, you just need to replace the calling function from `svm` to `naiveBayes`:

```
> for (i in 1:10) {  
+   fit = naiveBayes(churn ~., churnTrain[ind != i,])  
+   predictions = predict(fit, churnTrain[ind == i, ! names(churnTrain)  
%in% c("churn")])  
+   correct_count = sum(predictions == churnTrain[ind == i,c("churn")])  
+   accuracies = append(correct_count / nrow(churnTrain[ind == i,]),  
accuracies)  
+ }
```

Performing cross-validation with the e1071 package

Besides implementing a `loop` function to perform the k-fold cross-validation, you can use the `tuning` function (for example, `tune.nnet`, `tune.randomForest`, `tune.rpart`, `tune.svm`, and `tune.knn`) within the `e1071` package to obtain the minimum error value. In this recipe, we will illustrate how to use `tune.svm` to perform the 10-fold cross-validation and obtain the optimum classification model.

Getting ready

In this recipe, we continue to use the `telecom churn` dataset as the input data source to perform 10-fold cross-validation.

How to do it...

Perform the following steps to retrieve the minimum estimation error using cross-validation:

1. Apply `tune.svm` on the training dataset, `trainset`, with the 10-fold cross-validation as the tuning control. (If you find an error message, such as `could not find function predict.func`, please clear the workspace, restart the R session and reload the `e1071` library again):

```
> tuned = tune.svm(churn~, data = trainset, gamma = 10^-2, cost = 10^2, tunecontrol=tune.control(cross=10))
```

2. Next, you can obtain the summary information of the model, `tuned`:

```
> summary(tuned)
```

```
Error estimation of 'svm' using 10-fold cross validation:  
0.08164651
```

3. Then, you can access the performance details of the tuned model:

```
> tuned$performances  
    gamma cost      error dispersion  
1 0.01 100 0.08164651 0.02437228
```

4. Lastly, you can use the optimum model to generate a classification table:

```
> svmfit = tuned$best.model  
> table(trainset[,c("churn")], predict(svmfit))
```

	yes	no
yes	234	108
no	13	1960

How it works...

The `e1071` package provides miscellaneous functions to build and assess models, therefore, you do not need to reinvent the wheel to evaluate a fitted model. In this recipe, we use the `tune.svm` function to tune the `svm` model with the given formula, dataset, gamma, cost, and control functions. Within the `tune.control` options, we configure the option `as cross=10`, which performs a 10-fold cross validation during the tuning process. The tuning process will eventually return the minimum estimation error, performance detail, and the best model during the tuning process. Therefore, we can obtain the performance measures of the tuning and further use the optimum model to generate a classification table.

See also

- In the `e1071` package, the `tune` function uses a grid search to tune parameters. For those interested in other tuning functions, use the `help` function to view the `tune` document:

```
> ?e1071::tune
```

Performing cross-validation with the caret package

The `Caret` (classification and regression training) package contains many functions in regard to the training process for regression and classification problems. Similar to the `e1071` package, it also contains a function to perform the k-fold cross validation. In this recipe, we will demonstrate how to the perform k-fold cross validation using the `caret` package.

Getting ready

In this recipe, we will continue to use the `telecom churn` dataset as the input data source to perform the k-fold cross validation.

How to do it...

Perform the following steps to perform the k-fold cross-validation with the `caret` package:

- First, set up the control parameter to train with the 10-fold cross validation in 3 repetitions:

```
> control = trainControl(method="repeatedcv", number=10,  
  repeats=3)
```

2. Then, you can train the classification model on telecom churn data with `rpart`:

```
> model = train(churn~., data=trainset, method="rpart",
+ preProcess="scale", trControl=control)
```

3. Finally, you can examine the output of the generated model:

```
> model
```

```
CART
```

```
2315 samples
```

```
16 predictor
```

```
2 classes: 'yes', 'no'
```

```
Pre-processing: scaled
```

```
Resampling: Cross-Validated (10 fold, repeated 3 times)
```

```
Summary of sample sizes: 2084, 2083, 2082, 2084, 2083, 2084, ...
```

```
Resampling results across tuning parameters:
```

cp	Accuracy	Kappa	Accuracy SD	Kappa SD
0.0556	0.904	0.531	0.0236	0.155
0.0746	0.867	0.269	0.0153	0.153
0.0760	0.860	0.212	0.0107	0.141

```
Accuracy was used to select the optimal model using the largest value.
```

```
The final value used for the model was cp = 0.05555556.
```

How it works...

In this recipe, we demonstrate how convenient it is to conduct the k-fold cross-validation using the `caret` package. In the first step, we set up the training control and select the option to perform the 10-fold cross-validation in three repetitions. The process of repeating the k-fold validation is called repeated k-fold validation, which is used to test the stability of the model. If the model is stable, one should get a similar test result. Then, we apply `rpart` on the training dataset with the option to scale the data and to train the model with the options configured in the previous step.

After the training process is complete, the model outputs three resampling results. Of these results, the model with `cp=0.05555556` has the largest accuracy value (0.904), and is therefore selected as the optimal model for classification.

See also

- ▶ You can configure the resampling function in `trainControl`, in which you can specify `boot`, `boot632`, `cv`, `repeatedcv`, `LOOCV`, `LGOCV`, `none`, `oob`, `adaptive_cv`, `adaptive_boot`, or `adaptive_LGOCV`. To view more detailed information of how to choose the resampling method, view the `trainControl` document:
 > `?trainControl`

Ranking the variable importance with the caret package

After building a supervised learning model, we can estimate the importance of features. This estimation employs a sensitivity analysis to measure the effect on the output of a given model when the inputs are varied. In this recipe, we will show you how to rank the variable importance with the `caret` package.

Getting ready

You need to have completed the previous recipe by storing the fitted `rpart` object in the `model` variable.

How to do it...

Perform the following steps to rank the variable importance with the `caret` package:

1. First, you can estimate the variable importance with the `varImp` function:

```
> importance = varImp(model, scale=FALSE)
> importance
rpart variable importance
```

	Overall
<code>number_customer_service_calls</code>	116.015
<code>total_day_minutes</code>	106.988
<code>total_day_charge</code>	100.648
<code>international_planyes</code>	86.789

voice_mail_planyes	25.974
total_eve_charge	23.097
total_eve_minutes	23.097
number_vmail_messages	19.885
total_intl_minutes	6.347
total_eve_calls	0.000
total_day_calls	0.000
total_night_charge	0.000
total_intl_calls	0.000
total_intl_charge	0.000
total_night_minutes	0.000
total_night_calls	0.000

2. Then, you can generate the variable importance plot with the `plot` function:

```
> plot(importance)
```

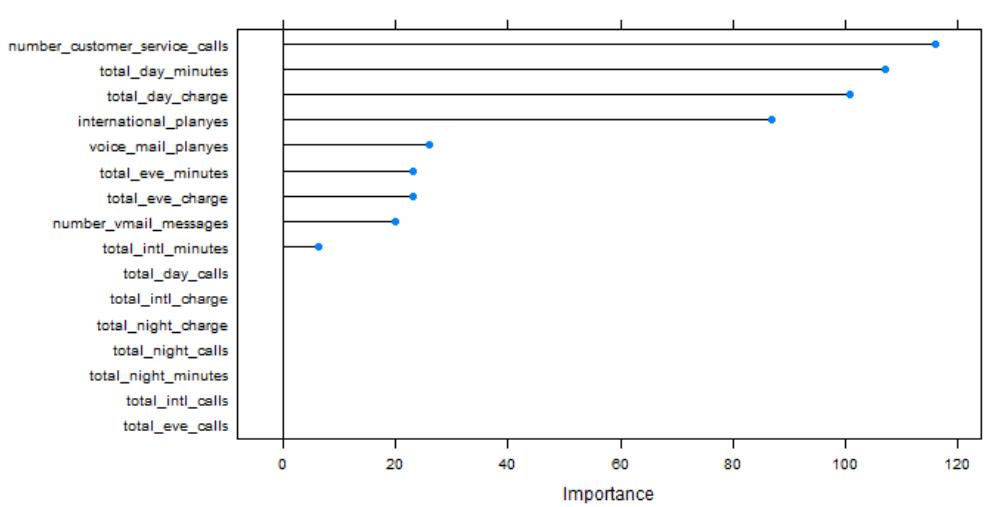


Figure 1: The visualization of variable importance using the caret package

How it works...

In this recipe, we first use the `varImp` function to retrieve the variable importance and obtain the summary. The overall results show the sensitivity measure of each attribute. Next, we plot the variable importance in terms of rank, which shows that the `number_customer_service_calls` attribute is the most important variable in the sensitivity measure.

There's more...

In some classification packages, such as `rpart`, the object generated from the training model contains the variable importance. We can examine the variable importance by accessing the output object:

```
> library(rpart)
> model.rp = rpart(churn~, data=trainset)
> model.rp$variable.importance
```

total_day_minutes	111.645286	total_day_charge	110.881583
number_customer_service_calls	58.486651	total_intl_minutes	48.283228
total_intl_charge	47.698379	total_eve_charge	47.166646
total_eve_minutes	47.166646	international_plan	42.194508
total_intl_calls	36.730344	number_vmail_messages	19.884863
voice_mail_plan	19.884863	total_night_calls	7.195828
total_eve_calls	3.553423	total_night_charge	1.754547
total_night_minutes	1.754547	total_day_calls	1.494986

Ranking the variable importance with the rminer package

Besides using the `caret` package to generate variable importance, you can use the `rminer` package to generate the variable importance of a classification model. In the following recipe, we will illustrate how to use `rminer` to obtain the variable importance of a fitted model.

Getting ready

In this recipe, we will continue to use the telecom churn dataset as the input data source to rank the variable importance.

How to do it...

Perform the following steps to rank the variable importance with rminer:

1. Install and load the package, rminer:

```
> install.packages("rminer")
> library(rminer)
```

2. Fit the svm model with the training set:

```
> model=fit(churn~,trainset,model="svm")
```

3. Use the Importance function to obtain the variable importance:

```
> VariableImportance=Importance(model,trainset,method="sensv")
```

4. Plot the variable importance ranked by the variance:

```
> L=list(runs=1,sen=t(VariableImportance$imp),sresponses=VariableImportance$sresponses)
> mgraph(L,graph="IMP",leg=names(trainset),col="gray",Grid=10)
```

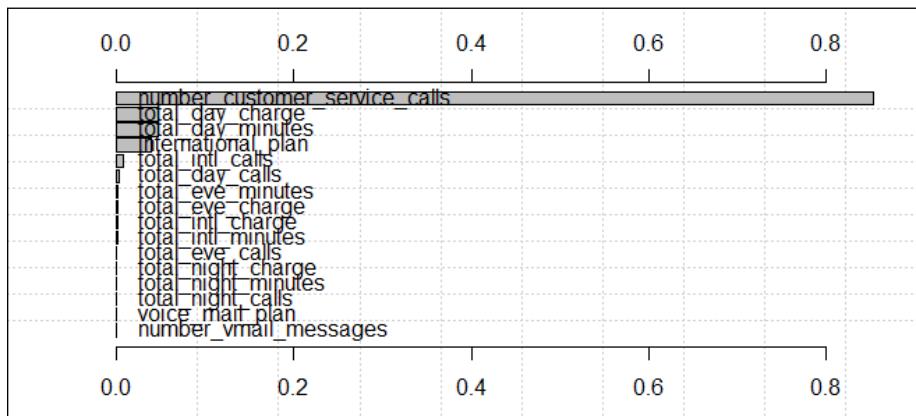


Figure 2: The visualization of variable importance using the rminer package

How it works...

Similar to the caret package, the rminer package can also generate the variable importance of a classification model. In this recipe, we first train the svm model on the training dataset, trainset, with the fit function. Then, we use the Importance function to rank the variable importance with a sensitivity measure. Finally, we use mgraph to plot the rank of the variable importance. Similar to the result obtained from using the caret package, number_customer_service_calls is the most important variable in the measure of sensitivity.

See also

- ▶ The `rminer` package provides many classification models for one to choose from. If you are interested in using models other than `svm`, you can view these options with the following command:

```
> ?rminer::fit
```

Finding highly correlated features with the caret package

When performing regression or classification, some models perform better if highly correlated attributes are removed. The `caret` package provides the `findCorrelation` function, which can be used to find attributes that are highly correlated to each other. In this recipe, we will demonstrate how to find highly correlated features using the `caret` package.

Getting ready

In this recipe, we will continue to use the `telecom churn` dataset as the input data source to find highly correlated features.

How to do it...

Perform the following steps to find highly correlated attributes:

1. Remove the features that are not coded in numeric characters:

```
> new_train = trainset[, ! names(churnTrain) %in% c("churn",  
"international_plan", "voice_mail_plan")]
```

2. Then, you can obtain the correlation of each attribute:

```
>cor_mat = cor(new_train)
```

3. Next, we use `findCorrelation` to search for highly correlated attributes with a cutoff equal to 0.75:

```
> highlyCorrelated = findCorrelation(cor_mat, cutoff=0.75)
```

4. We then obtain the name of highly correlated attributes:

```
> names(new_train)[highlyCorrelated]  
[1] "total_intl_minutes" "total_day_charge"      "total_eve_  
minutes"    "total_night_minutes"
```

How it works...

In this recipe, we search for highly correlated attributes using the `caret` package. In order to retrieve the correlation of each attribute, one should first remove nonnumeric attributes. Then, we perform correlation to obtain a correlation matrix. Next, we use `findCorrelation` to find highly correlated attributes with the cut off set to 0.75. We finally obtain the names of highly correlated (with a correlation coefficient over 0.75) attributes, which are `total_intl_minutes`, `total_day_charge`, `total_eve_minutes`, and `total_night_minutes`. You can consider removing some highly correlated attributes and keep one or two attributes for better accuracy.

See also

- ▶ In addition to the `caret` package, you can use the `leaps`, `genetic`, and `anneal` functions in the `subselect` package to achieve the same goal

Selecting features using the caret package

The feature selection method searches the subset of features with minimized predictive errors. We can apply feature selection to identify which attributes are required to build an accurate model. The `caret` package provides a recursive feature elimination function, `rfe`, which can help automatically select the required features. In the following recipe, we will demonstrate how to use the `caret` package to perform feature selection.

Getting ready

In this recipe, we will continue to use the telecom `churn` dataset as the input data source for feature selection.

How to do it...

Perform the following steps to select features:

1. Transform the feature named as `international_plan` of the training dataset, `trainset`, to `intl_yes` and `intl_no`:

```
> intl_plan = model.matrix(~ trainset.international_plan - 1,  
  data=data.frame(trainset$international_plan))  
  
> colnames(intl_plan) = c("trainset.international_planno"="intl_no", "trainset.international_planyes"= "intl_yes")
```

2. Transform the feature named as voice_mail_plan of the training dataset, trainset, to voice_yes and voice_no:

```
> voice_plan = model.matrix(~ trainset.voice_mail_plan - 1,  
  data=data.frame(trainset$voice_mail_plan))  
  
> colnames(voice_plan) = c("trainset.voice_mail_planno" = "voice_  
 no", "trainset.voice_mail_planyes"= "voidec_yes")
```

3. Remove the international_plan and voice_mail_plan attributes and combine the training dataset, trainset with the data frames, intl_plan and voice_plan:

```
> trainset$international_plan = NULL  
  
> trainset$voice_mail_plan = NULL  
  
> trainset = cbind(intl_plan,voice_plan, trainset)
```

4. Transform the feature named as international_plan of the testing dataset, testset, to intl_yes and intl_no:

```
> intl_plan = model.matrix(~ testset.international_plan - 1,  
  data=data.frame(testset$international_plan))  
  
> colnames(intl_plan) = c("testset.international_planno"="intl_  
 no", "testset.international_planyes"= "intl_yes")
```

5. Transform the feature named as voice_mail_plan of the training dataset, trainset, to voice_yes and voice_no:

```
> voice_plan = model.matrix(~ testset.voice_mail_plan - 1,  
  data=data.frame(testset$voice_mail_plan))  
  
> colnames(voice_plan) = c("testset.voice_mail_planno" = "voice_  
 no", "testset.voice_mail_planyes"= "voidec_yes")
```

6. Remove the international_plan and voice_mail_plan attributes and combine the testing dataset, testset with the data frames, intl_plan and voice_plan:

```
> testset$international_plan = NULL  
  
> testset$voice_mail_plan = NULL  
  
> testset = cbind(intl_plan,voice_plan, testset)
```

7. We then create a feature selection algorithm using linear discriminant analysis:

```
> ldaControl = rfeControl(functions = ldaFuncs, method = "cv")
```

8. Next, we perform a backward feature selection on the training dataset, trainset using subsets from 1 to 18:

```
> ldaProfile = rfe(trainset[, !names(trainset) %in% c("churn")],  
trainset[,c("churn")], sizes = c(1:18), rfeControl = ldaControl)  
> ldaProfile
```

Recursive feature selection

Outer resampling method: Cross-Validated (10 fold)

Resampling performance over subset size:

Variables	Accuracy	Kappa	AccuracySD	KappaSD	Selected
1	0.8523	0.0000	0.001325	0.00000	
2	0.8523	0.0000	0.001325	0.00000	
3	0.8423	0.1877	0.015468	0.09787	
4	0.8462	0.2285	0.016593	0.09610	
5	0.8466	0.2384	0.020710	0.09970	
6	0.8466	0.2364	0.019612	0.09387	
7	0.8458	0.2315	0.017551	0.08670	
8	0.8458	0.2284	0.016608	0.09536	
9	0.8475	0.2430	0.016882	0.10147	
10	0.8514	0.2577	0.014281	0.08076	
11	0.8518	0.2587	0.014124	0.08075	
12	0.8544	0.2702	0.015078	0.09208	*
13	0.8544	0.2721	0.015352	0.09421	
14	0.8531	0.2663	0.018428	0.11022	
15	0.8527	0.2652	0.017958	0.10850	
16	0.8531	0.2684	0.017897	0.10884	
17	0.8531	0.2684	0.017897	0.10884	
18	0.8531	0.2684	0.017897	0.10884	

The top 5 variables (out of 12):

```
total_day_charge, total_day_minutes, intl_no, number_customer_  
service_calls, total_eve_charge
```

9. Next, we can plot the selection result:

```
> plot(ldaProfile, type = c("o", "g"))
```

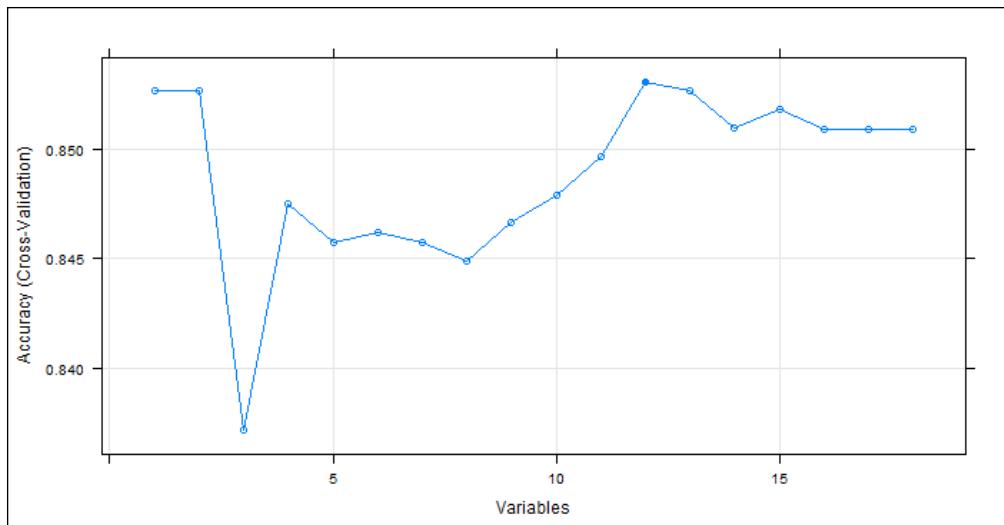


Figure 3: The feature selection result

10. We can then examine the best subset of the variables:

```
> ldaProfile$optVariables  
[1] "total_day_charge"  
[2] "total_day_minutes"  
[3] "intl_no"  
[4] "number_customer_service_calls"  
[5] "total_eve_charge"  
[6] "total_eve_minutes"  
[7] "voidce_yes"  
[8] "total_intl_calls"  
[9] "number_vmail_messages"  
[10] "total_intl_charge"  
[11] "total_intl_minutes"  
[12] "total_night_minutes"
```

11. Now, we can examine the fitted model:

```
> ldaProfile$fit
```

Call:

```
lda(x, y)
```

Prior probabilities of groups:

yes no

0.1477322 0.8522678

Group means:

	total_day_charge	total_day_minutes	intl_no
yes	35.00143	205.8877	0.7046784
no	29.62402	174.2555	0.9351242
	number_customer_service_calls	total_eve_charge	
yes		2.204678	18.16702
no		1.441460	16.96789
	total_eve_minutes	voidce_yes	total_intl_calls
yes	213.7269	0.1666667	4.134503
no	199.6197	0.2954891	4.5144445
	number_vmail_messages	total_intl_charge	
yes	5.099415		2.899386
no	8.674607		2.741343
	total_intl_minutes	total_night_minutes	
yes	10.73684		205.4640
no	10.15119		201.4184

Coefficients of linear discriminants:

	LD1
total_day_charge	0.715025524
total_day_minutes	-0.130486470
intl_no	2.259889324
number_customer_service_calls	-0.421997335
total_eve_charge	-2.390372793
total_eve_minutes	0.198406977
voidce_yes	0.660927935
total_intl_calls	0.066240268
number_vmail_messages	-0.003529233
total_intl_charge	2.315069869
total_intl_minutes	-0.693504606
total_night_minutes	-0.002127471

12. Finally, we can calculate the performance across resamples:

```
> postResample(predict(ldaProfile, testset[, !names(testset) %in%  
c("churn")]), testset[,c("churn")])  
Accuracy      Kappa  
0.8605108  0.2672027
```

How it works...

In this recipe, we perform feature selection using the `caret` package. As there are factor-coded attributes within the dataset, we first use a function called `model.matrix` to transform the factor-coded attributes into multiple binary attributes. Therefore, we transform the `international_plan` attribute to `intl_yes` and `intl_no`. Additionally, we transform the `voice_mail_plan` attribute to `voice_yes` and `voice_no`.

Next, we set up control parameters for training using the cross-validation method, `cv`, with the linear discriminant function, `ldaFuncs`. Then, we use the recursive feature elimination, `rfe`, to perform feature selection with the use of the `control` function, `ldaFuncs`. The `rfe` function generates the summary of feature selection, which contains resampling a performance over the subset size and top variables.

We can then use the obtained model information to plot the number of variables against accuracy. From Figure 3, it is obvious that using 12 features can obtain the best accuracy. In addition to this, we can retrieve the best subset of the variables in (12 variables in total) the fitted model. Lastly, we can calculate the performance across resamples, which yields an accuracy of 0.86 and a kappa of 0.27.

See also

- ▶ In order to specify the algorithm used to control feature selection, one can change the control function specified in `rfeControl`. Here are some of the options you can use:

<code>caretFuncs</code>	<code>SVM (caret)</code>
<code>lmFuncs</code>	<code>lm (base)</code>
<code>rfFuncs</code>	<code>RF (randomForest)</code>
<code>treebagFuncs</code>	<code>DT (ipred)</code>
<code>ldaFuncs</code>	<code>lda (base)</code>
<code>nbFuncs</code>	<code>NB (klaR)</code>
<code>gamFuncs</code>	<code>gam (gam)</code>

Measuring the performance of the regression model

To measure the performance of a regression model, we can calculate the distance from predicted output and the actual output as a quantifier of the performance of the model. Here, we often use the **root mean square error (RMSE)**, **relative square error (RSE)** and R-Square as common measurements. In the following recipe, we will illustrate how to compute these measurements from a built regression model.

Getting ready

In this recipe, we will use the Quartet dataset, which contains four regression datasets, as our input data source.

How to do it...

Perform the following steps to measure the performance of the regression model:

1. Load the Quartet dataset from the `car` package:

```
> library(car)  
> data(Quartet)
```

2. Plot the attribute, `y3`, against `x` using the `lm` function:

```
> plot(Quartet$x, Quartet$y3)  
> lmfit = lm(Quartet$y3~Quartet$x)  
> abline(lmfit, col="red")
```

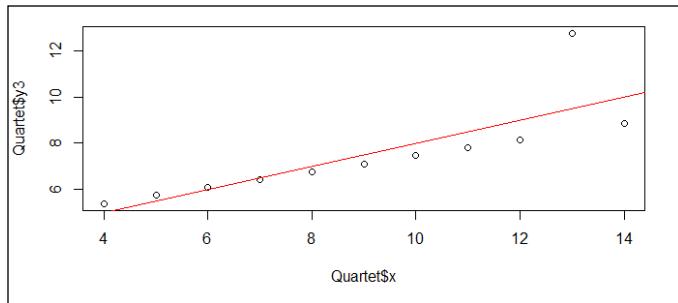


Figure 4: The linear regression plot

3. You can retrieve predicted values by using the `predict` function:

```
> predicted= predict(lmfit, newdata=Quartet[c("x")])
```

4. Now, you can calculate the root mean square error:

```
> actual = Quartet$y3  
> rmse = (mean((predicted - actual)^2))^0.5  
> rmse  
[1] 1.118286
```

5. You can calculate the relative square error:

```
> mu = mean(actual)  
> rse = mean((predicted - actual)^2) / mean((mu - actual)^2)  
> rse  
[1] 0.333676
```

6. Also, you can use R-Square as a measurement:

```
> rsquare = 1 - rse  
> rsquare  
[1] 0.666324
```

7. Then, you can plot attribute, y3, against x using the `rlm` function from the MASS package:

```
> library(MASS)  
> plot(Quartet$x, Quartet$y3)  
> rlmfit = rlm(Quartet$y3~Quartet$x)  
> abline(rlmfit, col="red")
```

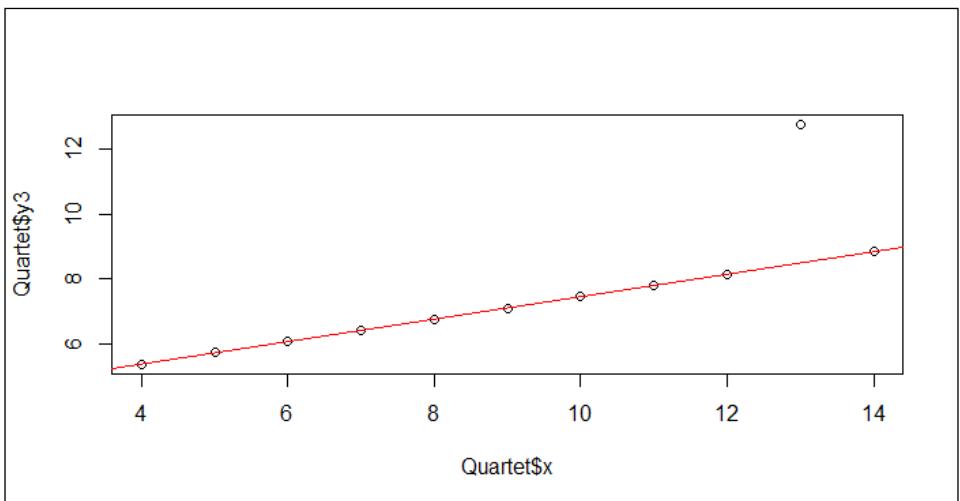


Figure 5: The robust linear regression plot on the Quartet dataset

8. You can then retrieve the predicted value using the predict function:

```
> predicted = predict(rlmfit, newdata=Quartet[c("x")])
```

9. Next, you can calculate the root mean square error using the distance of the predicted and actual value:

```
> actual = Quartet$y3  
> rmse = (mean((predicted - actual)^2)) ^0.5  
> rmse  
[1] 1.279045
```

10. Calculate the relative square error between the predicted and actual labels:

```
> mu = mean(actual)  
> rse = mean((predicted - actual)^2) / mean((mu - actual)^2)  
> rse  
[1] 0.4365067
```

11. Now, you can calculate the R-Square value:

```
> rsquare = 1 - rse  
> rsquare  
[1] 0.5634933
```

How it works...

The measurement of the performance of the regression model employs the distance between the predicted value and the actual value. We often use these three measurements, root mean square error, relative square error, and R-Square, as the quantifier of the performance of regression models. In this recipe, we first load the `Quartet` data from the `car` package. We then use the `lml` function to fit the linear model, and add the regression line on a scatter plot of the `x` variable against the `y3` variable. Next, we compute the predicted value using the `predict` function, and begin to compute the **root mean square error (RMSE)**, **relative square error (RSE)**, and R-Square for the built model.

As this dataset has an outlier at `x=13`, we would like to quantify how the outlier affects the performance measurement. To achieve this, we first train a regression model using the `rlm` function from the `MASS` package. Similar to the previous step, we then generate a performance measurement of the root square mean error, relative error and R-Square. From the output measurement, it is obvious that the mean square error and the relative square errors of the `lml` model are smaller than the model built by `rlm`, and the score of R-Square shows that the model built with `lml` has a greater prediction power. However, for the actual scenario, we should remove the outlier at `x=13`. This comparison shows that the outlier may be biased toward the performance measure and may lead us to choose the wrong model.

There's more...

If you would like to perform cross-validation on a linear regression model, you can use the `tune` function within the `e1071` package:

```
> tune(lm, y3~x, data = Quartet)
Error estimation of 'lm' using 10-fold cross validation: 2.33754
```

Other than the `e1071` package, you can use the `train` function from the `caret` package to perform cross-validation. In addition to this, you can also use `cv.lm` from the `DAAG` package to achieve the same goal.

Measuring prediction performance with a confusion matrix

To measure the performance of a classification model, we can first generate a classification table based on our predicted label and actual label. Then, we can use a confusion matrix to obtain performance measures such as precision, recall, specificity, and accuracy. In this recipe, we will demonstrate how to retrieve a confusion matrix using the `caret` package.

Getting ready

In this recipe, we will continue to use the `telecom churn` dataset as our example dataset.

How to do it...

Perform the following steps to generate a classification measurement:

1. Train an `svm` model using the training dataset:

```
> svm.model= train(churn ~ .,
+                   data = trainset,
+                   method = "svmRadial")
```

2. You can then predict labels using the fitted model, `svm.model`:

```
> svm.pred = predict(svm.model, testset[,- names(testset) %in%
c("churn")])
```

3. Next, you can generate a classification table:

```
> table(svm.pred, testset[,c("churn")])

svm.pred yes no
      yes   73  16
      no    68 861
```

4. Lastly, you can generate a confusion matrix using the prediction results and the actual labels from the testing dataset:

```
> confusionMatrix(svm.pred, testset[,c("churn")])
```

```
Confusion Matrix and Statistics
```

		Reference
Prediction	yes	no
yes	73	16
no	68	861

Accuracy : 0.9175

95% CI : (0.8989, 0.9337)

No Information Rate : 0.8615

P-Value [Acc > NIR] : 2.273e-08

Kappa : 0.5909

McNemar's Test P-Value : 2.628e-08

Sensitivity : 0.51773

Specificity : 0.98176

Pos Pred Value : 0.82022

Neg Pred Value : 0.92680

Prevalence : 0.13851

Detection Rate : 0.07171

Detection Prevalence : 0.08743

Balanced Accuracy : 0.74974

'Positive' Class : yes

How it works...

In this recipe, we demonstrate how to obtain a confusion matrix to measure the performance of a classification model. First, we use the `train` function from the `caret` package to train an `svm` model. Next, we use the `predict` function to extract the predicted labels of the `svm` model using the testing dataset. Then, we perform the `table` function to obtain the classification table based on the predicted and actual labels. Finally, we use the `confusionMatrix` function from the `caret` package to generate a confusion matrix to measure the performance of the classification model.

See also

- ▶ If you are interested in the available methods that can be used in the `train` function, you can refer to this website: <http://topepo.github.io/caret/modelList.html>

Measuring prediction performance using ROCR

A **receiver operating characteristic (ROC)** curve is a plot that illustrates the performance of a binary classifier system, and plots the true positive rate against the false positive rate for different cut points. We most commonly use this plot to calculate the **area under curve (AUC)** to measure the performance of a classification model. In this recipe, we will demonstrate how to illustrate an ROC curve and calculate the AUC to measure the performance of a classification model.

Getting ready

In this recipe, we will continue using the `telecom churn` dataset as our example dataset.

How to do it...

Perform the following steps to generate two different classification examples with different costs:

1. First, you should install and load the `ROCR` package:

```
> install.packages("ROCR")
> library(ROCR)
```
2. Train the `svm` model using the training dataset with a probability equal to `TRUE`:

```
> svmfit=svm(churn~ ., data=trainset, prob=TRUE)
```
3. Make predictions based on the trained model on the testing dataset with the probability set as `TRUE`:

```
> pred=predict(svmfit,testset[, !names(testset) %in% c("churn")], probability=TRUE)
```
4. Obtain the probability of labels with `yes`:

```
> pred.prob = attr(pred, "probabilities")
> pred.to.roc = pred.prob[, 2]
```

5. Use the prediction function to generate a prediction result:

```
> pred.rocr = prediction(pred.to.roc, testset$churn)
```

6. Use the performance function to obtain the performance measurement:

```
> perf.rocr = performance(pred.rocr, measure = "auc", x.measure =  
"cutoff")
```

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

7. Visualize the ROC curve using the plot function:

```
> plot(perf.tpr.rocr, colorize=T, main=paste("AUC:", (perf.rocr@y.  
values)))
```

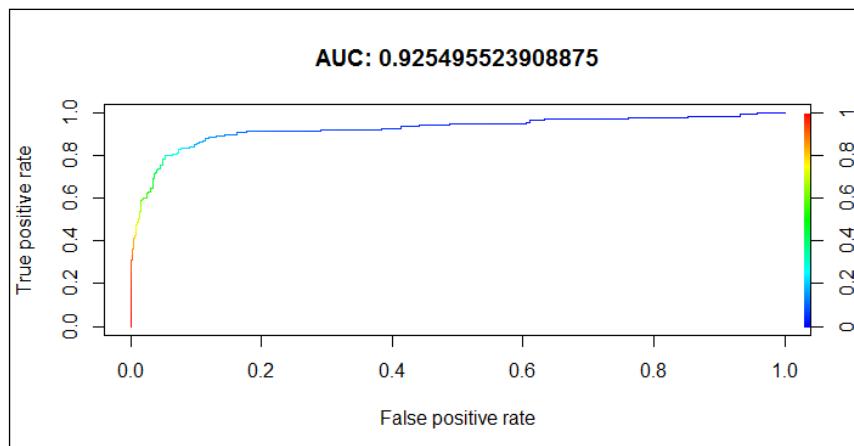


Figure 6: The ROC curve for the svm classifier performance

How it works...

In this recipe, we demonstrated how to generate an ROC curve to illustrate the performance of a binary classifier. First, we should install and load the library, ROCR. Then, we use `svm`, from the `e1071` package, to train a classification model, and then use the model to predict labels for the testing dataset. Next, we use the `prediction` function (from the package, `ROCR`) to generate prediction results. We then adapt the `performance` function to obtain the performance measurement of the true positive rate against the false positive rate. Finally, we use the `plot` function to visualize the ROC plot, and add the value of AUC on the title. In this example, the AUC value is 0.92, which indicates that the `svm` classifier performs well in classifying telecom user churn datasets.

See also

- For those interested in the concept and terminology of ROC, you can refer to http://en.wikipedia.org/wiki/Receiver_operating_characteristic

Comparing an ROC curve using the caret package

In previous chapters, we introduced many classification methods; each method has its own advantages and disadvantages. However, when it comes to the problem of how to choose the best fitted model, you need to compare all the performance measures generated from different prediction models. To make the comparison easy, the caret package allows us to generate and compare the performance of models. In this recipe, we will use the function provided by the caret package to compare different algorithm trained models on the same dataset.

Getting ready

Here, we will continue to use telecom dataset as our input data source.

How to do it...

Perform the following steps to generate an ROC curve of each fitted model:

1. Install and load the library, pROC:

```
> install.packages("pROC")
> library("pROC")
```

2. Set up the training control with a 10-fold cross-validation in 3 repetitions:

```
> control = trainControl(method = "repeatedcv",
+                           number = 10,
+                           repeats = 3,
+                           classProbs = TRUE,
+                           summaryFunction = twoClassSummary)
```

3. Then, you can train a classifier on the training dataset using glm:

```
> glm.model= train(churn ~.,
+                     data = trainset,
+                     method = "glm",
+                     metric = "ROC",
+                     trControl = control)
```

4. Also, you can train a classifier on the training dataset using svm:

```
> svm.model= train(churn ~.,
+                     data = trainset,
```

```
+           method = "svmRadial",
+
+           metric = "ROC",
+
+           trControl = control)
```

5. To see how rpart performs on the training data, we use the rpart function:

```
> rpart.model= train(churn ~.,
+
+                     data = trainset,
+
+                     method = "rpart",
+
+                     metric = "ROC",
+
+                     trControl = control)
```

6. You can make predictions separately based on different trained models:

```
> glm.probs = predict(glm.model, testset[,- names(testset) %in%
c("churn")], type = "prob")
> svm.probs = predict(svm.model, testset[,- names(testset) %in%
c("churn")], type = "prob")
> rpart.probs = predict(rpart.model, testset[,- names(testset)
%in% c("churn")], type = "prob")
```

7. You can generate the ROC curve of each model, and plot the curve on the same figure:

```
> glm.ROC = roc(response = testset[,c("churn")],
+
+                  predictor =glm.probs$yes,
+
+                  levels = levels(testset[,c("churn")]))
> plot(glm.ROC, type="S", col="red")
```

Call:

```
roc.default(response = testset[, c("churn")], predictor = glm.
probs$yes,      levels = levels(testset[, c("churn")]))
```

Data: glm.probs\$yes in 141 controls (testset[, c("churn")] yes) >
877 cases (testset[, c("churn")] no).

Area under the curve: 0.82

```
> svm.ROC = roc(response = testset[,c("churn")],
+
+                  predictor =svm.probs$yes,
+
+                  levels = levels(testset[,c("churn")]))
> plot(svm.ROC, add=TRUE, col="green")
```

Call:

```
roc.default(response = testset[, c("churn")], predictor = svm.  
probs$yes,      levels = levels(testset[, c("churn")]))
```

Data: svm.probs\$yes in 141 controls (testset[, c("churn")] yes) >
877 cases (testset[, c("churn")] no).

Area under the curve: 0.9233

```
> rpart.ROC = roc(response = testset[,c("churn")],  
+                  predictor =rpart.probs$yes,  
+                  levels = levels(testset[,c("churn")]))  
> plot(rpart.ROC, add=TRUE, col="blue")
```

Call:

```
roc.default(response = testset[, c("churn")], predictor = rpart.  
probs$yes,      levels = levels(testset[, c("churn")]))
```

Data: rpart.probs\$yes in 141 controls (testset[, c("churn")] yes)
> 877 cases (testset[, c("churn")] no).

Area under the curve: 0.7581

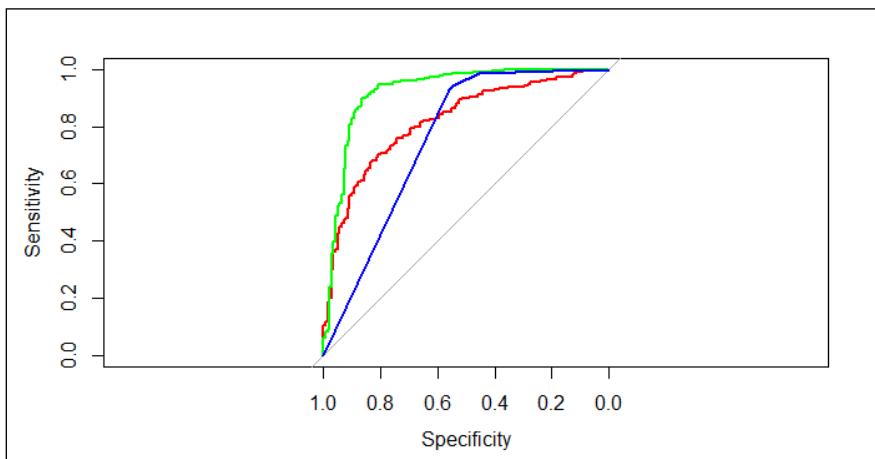


Figure 7: The ROC curve for the performance of three classifiers

How it works...

Here, we demonstrate how we can compare fitted models by illustrating their ROC curve in one figure. First, we set up the control of the training process with a 10-fold cross validation in 3 repetitions with the performance evaluation in `twoClassSummary`. After setting up control of the training process, we then apply `glm`, `svm`, and `rpart` algorithms on the training dataset to fit the classification models. Next, we can make a prediction based on each generated model and plot the ROC curve, respectively. Within the generated figure, we find that the model trained by `svm` has the largest area under curve, which is 0.9233 (plotted in green), the AUC of the `glm` model (red) is 0.82, and the AUC of the `rpart` model (blue) is 0.7581. From *Figure 7*, it is obvious that `svm` performs the best among all the fitted models on this training dataset (without requiring tuning).

See also

- ▶ We use another ROC visualization package, `pROC`, which can be employed to display and analyze ROC curves. If you would like to know more about the package, please use the `help` function:

`> help(package="pROC")`

Measuring performance differences between models with the `caret` package

In the previous recipe, we introduced how to generate ROC curves for each generated model, and have the curve plotted on the same figure. Apart from using an ROC curve, one can use the resampling method to generate statistics of each fitted model in ROC, sensitivity and specificity metrics. Therefore, we can use these statistics to compare the performance differences between each model. In the following recipe, we will introduce how to measure performance differences between fitted models with the `caret` package.

Getting ready

One needs to have completed the previous recipe by storing the `glm` fitted model, `svm` fitted model, and the `rpart` fitted model into `glm.model`, `svm.model`, and `rpart.model`, respectively.

How to do it...

Perform the following steps to measure performance differences between each fitted model:

1. Resample the three generated models:

```
> cv.values = resamples(list(glm = glm.model, svm=svm.model, rpart = rpart.model))
```

2. Then, you can obtain a summary of the resampling result:

```
> summary(cv.values)
```

Call:

```
summary.resamples(object = cv.values)
```

Models: glm, svm, rpart

Number of resamples: 30

ROC

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	NA's
glm	0.7206	0.7847	0.8126	0.8116	0.8371	0.8877	0
svm	0.8337	0.8673	0.8946	0.8929	0.9194	0.9458	0
rpart	0.2802	0.7159	0.7413	0.6769	0.8105	0.8821	0

Sens

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	NA's
glm	0.08824	0.2000	0.2286	0.2194	0.2517	0.3529	0
svm	0.44120	0.5368	0.5714	0.5866	0.6424	0.7143	0
rpart	0.20590	0.3742	0.4706	0.4745	0.5929	0.6471	0

Spec

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	NA's
glm	0.9442	0.9608	0.9746	0.9701	0.9797	0.9949	0
svm	0.9442	0.9646	0.9746	0.9740	0.9835	0.9949	0
rpart	0.9492	0.9709	0.9797	0.9780	0.9848	0.9949	0

3. Use dotplot to plot the resampling result in the ROC metric:

```
> dotplot(cv.values, metric = "ROC")
```

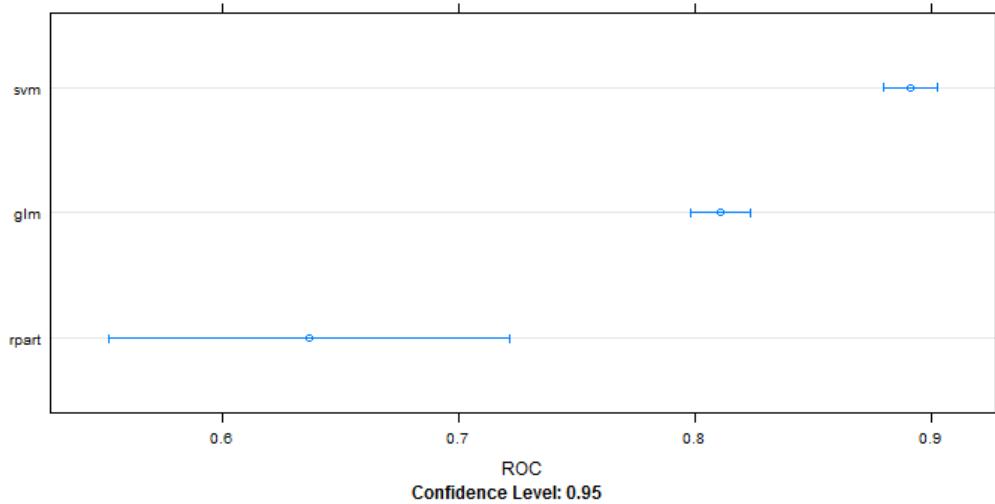


Figure 8: The dotplot of resampling result in ROC metric

4. Also, you can use a box-whisker plot to plot the resampling result:

```
> bwplot(cv.values, layout = c(3, 1))
```

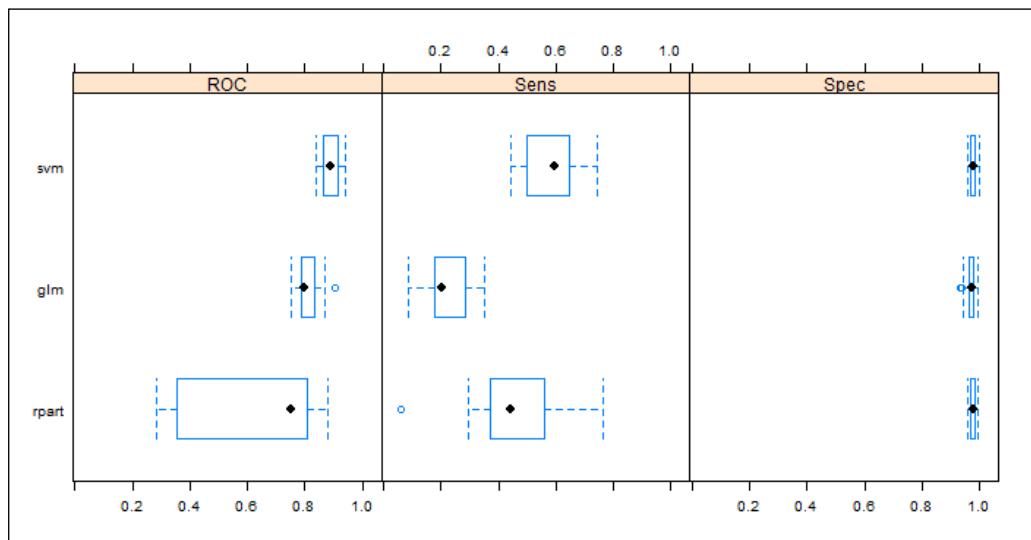


Figure 9: The box-whisker plot of resampling result

How it works...

In this recipe, we demonstrate how to measure the performance differences among three fitted models using the resampling method. First, we use the `resample` function to generate the statistics of each fitted model (`svm.model`, `glm.model`, and `rpart.model`). Then, we can use the `summary` function to obtain the statistics of these three models in the ROC, sensitivity and specificity metrics. Next, we can apply a `dotplot` on the resampling result to see how ROC varied between each model. Last, we use a box-whisker plot on the resampling results to show the box-whisker plot of different models in the ROC, sensitivity and specificity metrics on a single plot.

See also

- ▶ Besides using `dotplot` and `bwplot` to measure performance differences, one can use `densityplot`, `splom`, and `xyplot` to visualize the performance differences of each fitted model in the ROC, sensitivity, and specificity metrics.

7

Ensemble Learning

In this chapter, we will cover the following topics:

- ▶ Classifying data with the bagging method
- ▶ Performing cross-validation with the bagging method
- ▶ Classifying data with the boosting method
- ▶ Performing cross-validation with the boosting method
- ▶ Classifying data with gradient boosting
- ▶ Calculating the margins of a classifier
- ▶ Calculating the error evolution of the ensemble method
- ▶ Classifying the data with random forest
- ▶ Estimating the prediction errors of different classifiers

Introduction

Ensemble learning is a method to combine results produced by different learners into one format, with the aim of producing better classification results and regression results. In previous chapters, we discussed several classification methods. These methods take different approaches but they all have the same goal, that is, finding an optimum classification model. However, a single classifier may be imperfect, which may misclassify data in certain categories. As not all classifiers are imperfect, a better approach is to average the results by voting. In other words, if we average the prediction results of every classifier with the same input, we may create a superior model compared to using an individual method.

In ensemble learning, bagging, boosting, and random forest are the three most common methods:

- ▶ Bagging is a voting method, which first uses Bootstrap to generate a different training set, and then uses the training set to make different base learners. The bagging method employs a combination of base learners to make a better prediction.
- ▶ Boosting is similar to the bagging method. However, what makes boosting different is that it first constructs the base learning in sequence, where each successive learner is built for the prediction residuals of the preceding learner. With the means to create a complementary learner, it uses the mistakes made by previous learners to train the next base learner.
- ▶ Random forest uses the classification results voted from many classification trees. The idea is simple; a single classification tree will obtain a single classification result with a single input vector. However, a random forest grows many classification trees, obtaining multiple results from a single input. Therefore, a random forest will use the majority of votes from all the decision trees to classify data or use an average output for regression.

In the following recipes, we will discuss how to use bagging and boosting to classify data. We can then perform cross-validation to estimate the error rate of each classifier. In addition to this, we'll introduce the use of a margin to measure the certainty of a model. Next, we cover random forests, similar to the bagging and boosting methods, and introduce how to train the model to classify data and use margins to estimate the model certainty. Lastly, we'll demonstrate how to estimate the error rate of each classifier, and use the error rate to compare the performance of different classifiers.

Classifying data with the bagging method

The `adabag` package implements both boosting and bagging methods. For the bagging method, the package implements Breiman's Bagging algorithm, which first generates multiple versions of classifiers, and then obtains an aggregated classifier. In this recipe, we will illustrate how to use the bagging method from `adabag` to generate a classification model using the `telecom churn` dataset.

Getting ready

In this recipe, we continue to use the `telecom churn` dataset as the input data source for the bagging method. For those who have not prepared the dataset, please refer to *Chapter, Classification (I) – Tree, Lazy, and Probabilistic*, for detailed information.

How to do it...

Perform the following steps to generate a classification model for the telecom churn dataset:

1. First, you need to install and load the adabag package (it might take a while to install adabag):

```
> install.packages("adabag")
> library(adabag)
```

2. Next, you can use the bagging function to train a training dataset (the result may vary during the training process):

```
> set.seed(2)
> churn.bagging = bagging(churn ~ ., data=trainset, mfinal=10)
```

3. Access the variable importance from the bagging result:

```
> churn.bagging$importance
```

international_plan	number_customer_service_calls
10.4948380	16.4260510
number_vmail_messages	total_day_calls
0.5319143	0.3774190
total_day_charge	total_day_minutes
0.0000000	28.7545042
total_eve_calls	total_eve_charge
0.1463585	0.0000000
total_eve_minutes	total_intl_calls
14.2366754	8.7733895
total_intl_charge	total_intl_minutes
0.0000000	9.7838256
total_night_calls	total_night_charge
0.4349952	0.0000000
total_night_minutes	voice_mail_plan
2.3379622	7.7020671

4. After generating the classification model, you can use the predicted results from the testing dataset:

```
> churn.predbagging= predict.bagging(churn.bagging,
newdata=testset)
```

5. From the predicted results, you can obtain a classification table:

```
> churn.predbagging$confusion
```

		Observed Class	
		Predicted Class	yes no
Predicted Class	yes	no	
no	35	866	
yes	106	11	

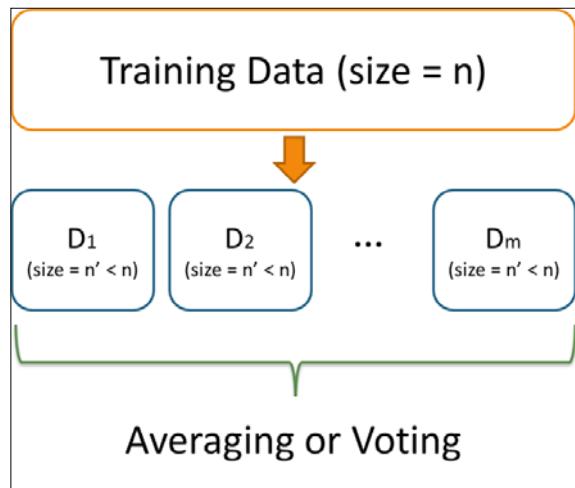
6. Finally, you can retrieve the average error of the bagging result:

```
> churn.predbagging$error
```

```
[1] 0.0451866
```

How it works...

Bagging is derived from the name Bootstrap aggregating, which is a stable, accurate, and easy to implement model for data classification and regression. The definition of bagging is as follows: given a training dataset of size n , bagging performs Bootstrap sampling and generates m new training sets, D_i , each of size n . Finally, we can fit m Bootstrap samples to m models and combine the result by averaging the output (for regression) or voting (for classification):



An illustration of bagging method

The advantage of using bagging is that it is a powerful learning method, which is easy to understand and implement. However, the main drawback of this technique is that it is hard to analyze the result.

In this recipe, we use the boosting method from adabag to classify the telecom churn data. Similar to other classification methods discussed in previous chapters, you can train a boosting classifier with a formula and a training dataset. Additionally, you can set the number of iterations to 10 in the `mfinal` argument. Once the classification model is built, you can examine the importance of each attribute. Ranking the attributes by importance reveals that the number of customer service calls play a crucial role in the classification model.

Next, with a fitted model, you can apply the `predict.bagging` function to predict the labels of the testing dataset. Therefore, you can use the labels of the testing dataset and predicted results to generate a classification table and obtain the average error, which is 0.045 in this example.

There's more...

Besides adabag, the ipred package provides a bagging method for a classification tree. We demonstrate here how to use the bagging method of the ipred package to train a classification model:

1. First, you need to install and load the ipred package:

```
> install.packages("ipred")
> library(ipred)
```

2. You can then use the bagging method to fit the classification method:

```
> churn.bagging = bagging(churn ~ ., data = trainset, coob = T)
> churn.bagging
```

```
Bagging classification trees with 25 bootstrap replications
```

```
Call: bagging.data.frame(formula = churn ~ ., data = trainset,
coob = T)
```

```
Out-of-bag estimate of misclassification error:  0.0605
```

3. Obtain an out of bag estimate of misclassification of the errors:

```
> mean(predict(churn.bagging) != trainset$churn)
[1] 0.06047516
```

4. You can then use the `predict` function to obtain the predicted labels of the testing dataset:

```
> churn.prediction = predict(churn.bagging, newdata=testset,
type="class")
```

5. Obtain the classification table from the labels of the testing dataset and prediction result:

```
> prediction.table = table(churn.prediction, testset$churn)
```

```
churn.prediction yes no
no      31 869
yes    110   8
```

Performing cross-validation with the bagging method

To assess the prediction power of a classifier, you can run a cross-validation method to test the robustness of the classification model. In this recipe, we will introduce how to use `bagging.cv` to perform cross-validation with the bagging method.

Getting ready

In this recipe, we continue to use the telecom `churn` dataset as the input data source to perform a k-fold cross-validation with the bagging method.

How to do it...

Perform the following steps to retrieve the minimum estimation errors by performing cross-validation with the bagging method:

1. First, we use `bagging.cv` to make a 10-fold classification on the training dataset with 10 iterations:

```
> churn.baggingcv = bagging.cv(churn ~ ., v=10, data=trainset,
mfinal=10)
```

2. You can then obtain the confusion matrix from the cross-validation results:

```
> churn.baggingcv$confusion
```

		Observed Class	
		yes	no
Predicted Class	yes	242	35
	no	100	1938

3. Lastly, you can retrieve the minimum estimation errors from the cross-validation results:

```
> churn.baggingcv$error
```

```
[1] 0.05831533
```

How it works...

The adabag package provides a function to perform the k-fold validation with either the bagging or boosting method. In this example, we use `bagging.cv` to make the k-fold cross-validation with the bagging method. We first perform a 10-fold cross validation with 10 iterations by specifying `v=10` and `mfinal=10`. Please note that this is quite time consuming due to the number of iterations. After the cross-validation process is complete, we can obtain the confusion matrix and average errors (0.058 in this case) from the cross-validation results.

See also

- ▶ For those interested in tuning the parameters of `bagging.cv`, please view the `bagging.cv` document by using the `help` function:
`> help(bagging.cv)`

Classifying data with the boosting method

Similar to the bagging method, boosting starts with a simple or weak classifier and gradually improves it by reweighting the misclassified samples. Thus, the new classifier can learn from previous classifiers. The adabag package provides implementation of the **AdaBoost.M1** and **SAMME** algorithms. Therefore, one can use the boosting method in adabag to perform ensemble learning. In this recipe, we will use the boosting method in adabag to classify the telecom churn dataset.

Getting ready

In this recipe, we will continue to use the telecom churn dataset as the input data source to perform classifications with the boosting method. Also, you need to have the adabag package loaded in R before commencing the recipe.

How to do it...

Perform the following steps to classify the telecom churn dataset with the boosting method:

1. You can use the boosting function from the adabag package to train the classification model:

```
> set.seed(2)  
> churn.boost = boosting(churn ~., data=trainset, mfinal=10,  
  coeflearn="Freund", boos=FALSE , control=rpart.  
  control(maxdepth=3))
```

2. You can then make a prediction based on the boosted model and testing dataset:

```
> churn.boost.pred = predict.boosting(churn.boost,newdata=testset)
```

3. Next, you can retrieve the classification table from the predicted results:

```
> churn.boost.pred$confusion
```

		Observed Class	
		Predicted Class	yes no
Predicted Class	no	41 858	
	yes	100 19	

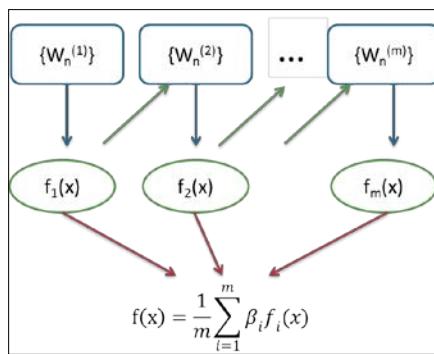
4. Finally, you can obtain the average errors from the predicted results:

```
> churn.boost.pred$error
```

```
[1] 0.0589391
```

How it works...

The idea of boosting is to "boost" weak learners (for example, a single decision tree) into strong learners. Assuming that we have n points in our training dataset, we can assign a weight, W_i ($0 \leq i < n$), for each point. Then, during the iterative learning process (we assume the number of iterations is m), we can reweigh each point in accordance with the classification result in each iteration. If the point is correctly classified, we should decrease the weight. Otherwise, we increase the weight of the point. When the iteration process is finished, we can then obtain the m fitted model, $f_i(x)$ ($0 \leq i < n$). Finally, we can obtain the final prediction through the weighted average of each tree's prediction, where the weight, β_i , is based on the quality of each tree:



An illustration of boosting method

Both bagging and boosting are ensemble methods, which combine the prediction power of each single learner into a strong learner. The difference between bagging and boosting is that the bagging method combines independent models, but boosting performs an iterative process to reduce the errors of preceding models by predicting them with successive models.

In this recipe, we demonstrate how to fit a classification model within the boosting method. Similar to bagging, one has to specify the formula and the training dataset used to train the classification model. In addition, one can specify parameters, such as the number of iterations (`mfinal`), the weight update coefficient (`coeflearn`), the weight of how each observation is used (`boos`), and the control for `rpart` (a single decision tree). In this recipe, we set the iteration to 10, using `Freund` (the AdaBoost.M1 algorithm implemented method) as `coeflearn`, `boos` set to false and max depth set to 3 for `rpart` configuration.

We use the boosting method to fit the classification model and then save it in `churn.boost`. We can then obtain predicted labels using the `prediction` function. Furthermore, we can use the `table` function to retrieve a classification table based on the predicted labels and testing the dataset labels. Lastly, we can get the average errors of the predicted results.

There's more...

In addition to using the `boosting` function in the `adabag` package, one can also use the `caret` package to perform a classification with the boosting method:

1. First, load the `mboost` and `pROC` package:

```
> library(mboost)
> install.packages("pROC")
> library(pROC)
```

2. We can then set the training control with the `trainControl` function and use the `train` function to train the classification model with `adaboost`:

```
> set.seed(2)
> ctrl = trainControl(method = "repeatedcv", repeats = 1,
  classProbs = TRUE, summaryFunction = twoClassSummary)
> ada.train = train(churn ~ ., data = trainset, method = "ada",
  metric = "ROC", trControl = ctrl)
```

3. Use the `summary` function to obtain the details of the classification model:

```
> ada.train$result
      nu maxdepth iter      ROC      Sens      Spec      ROCSD
SensSD      SpecSD
1 0.1          1    50 0.8571988 0.9152941 0.012662155 0.03448418
0.04430519 0.007251045
2 0.1          2    50 0.8905514 0.7138655 0.006083679 0.03538445
0.10089887 0.006236741
3 0.1          3    50 0.9056456 0.4036134 0.007093780 0.03934631
0.09406015 0.006407402
4 0.1          1   100 0.8550789 0.8918487 0.015705276 0.03434382
0.06190546 0.006503191
5 0.1          2   100 0.8907720 0.6609244 0.009626724 0.03788941
0.11403364 0.006940001
```

```

8 0.1      3 100 0.9077750 0.3832773 0.005576065 0.03601187
0.09630026 0.003738978

3 0.1      1 150 0.8571743 0.8714286 0.016720505 0.03481526
0.06198773 0.006767313

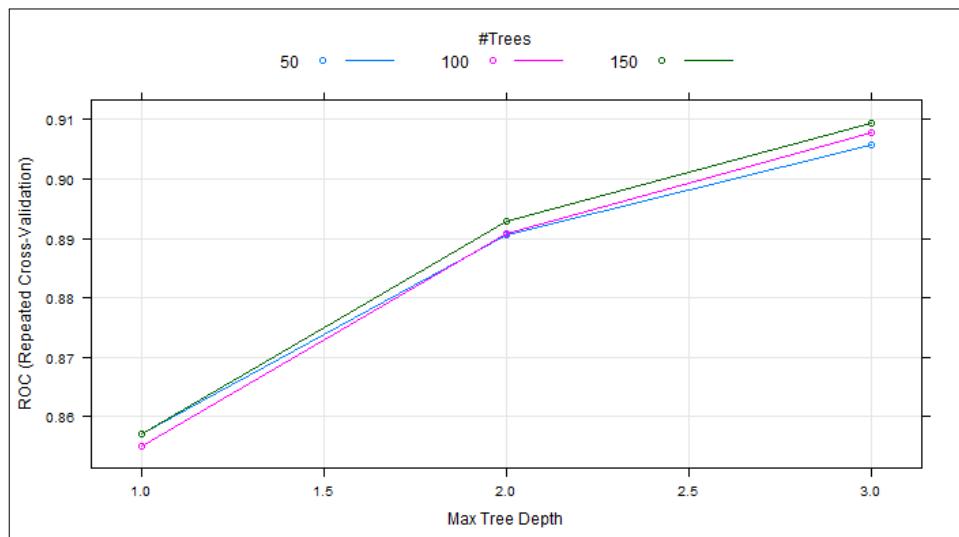
6 0.1      2 150 0.8929524 0.6171429 0.011654617 0.03638272
0.11383803 0.006777465

9 0.1      3 150 0.9093921 0.3743697 0.007093780 0.03258220
0.09504202 0.005446136

```

4. Use the `plot` function to plot the ROC curve within different iterations:

```
> plot(ada.train)
```



The repeated cross validation plot

5. Finally, we can make predictions using the `predict` function and view the classification table:

```

> ada.predict = predict(ada.train, testset, "prob")
> ada.predict.result = ifelse(ada.predict[1] > 0.5, "yes", "no")

> table(testset$churn, ada.predict.result)

ada.predict.result
  no yes
yes 40 101
no 872   5

```

Performing cross-validation with the boosting method

Similar to the bagging function, adabag provides a cross-validation function for the boosting method, named `boosting.cv`. In this recipe, we will demonstrate how to perform cross-validation using `boosting.cv` from the package, `adabag`.

Getting ready

In this recipe, we continue to use the telecom `churn` dataset as the input data source to perform a k-fold cross-validation with the `boosting` method.

How to do it...

Perform the following steps to retrieve the minimum estimation errors via cross-validation with the `boosting` method:

1. First, you can use `boosting.cv` to cross-validate the training dataset:

```
> churn.boostcv = boosting.cv(churn ~ ., v=10, data=trainset,
  mfinal=5,control=rpart.control(cp=0.01))
```

2. You can then obtain the confusion matrix from the boosting results:

```
> churn.boostcv$confusion
```

		Observed Class	
		Predicted Class	yes no
Predicted Class	yes	119	1940
	no	223	33

3. Finally, you can retrieve the average errors of the boosting method:

```
> churn.boostcv$error
```

```
[1] 0.06565875
```

How it works...

Similar to `bagging.cv`, we can perform cross-validation with the `boosting` method using `boosting.cv`. If `v` is set to 10 and `mfinal` is set to 5, the `boosting` method will perform 10-fold cross-validations with five iterations. Also, one can set the control of the `rpart` fit within the parameter. We can set the complexity parameter to 0.01 in this example. Once the training is complete, the confusion matrix and average errors of the boosted results will be obtained.

See also

- ▶ For those who require more information on tuning the parameters of `boosting.cv`, please view the `boosting.cv` document by using the `help` function:
`> help(boosting.cv)`

Classifying data with gradient boosting

Gradient boosting ensembles weak learners and creates a new base learner that maximally correlates with the negative gradient of the loss function. One may apply this method on either regression or classification problems, and it will perform well in different datasets. In this recipe, we will introduce how to use `gbm` to classify a telecom churn dataset.

Getting ready

In this recipe, we continue to use the telecom `churn` dataset as the input data source for the bagging method. For those who have not prepared the dataset, please refer to *Chapter, Classification (I) – Tree, Lazy, and Probabilistic*, for detailed information.

How to do it...

Perform the following steps to calculate and classify data with the gradient boosting method:

1. First, install and load the package, `gbm`:

```
> install.packages("gbm")
> library(gbm)
```

2. The `gbm` function only uses responses ranging from 0 to 1; therefore, you should transform yes/no responses to numeric responses (0/1):

```
> trainset$churn = ifelse(trainset$churn == "yes", 1, 0)
```

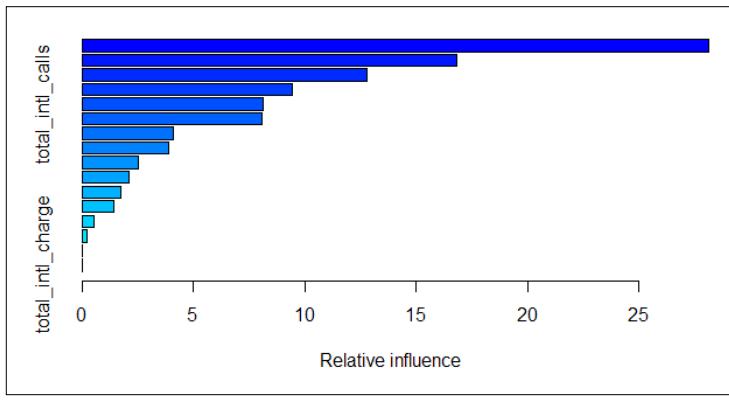
3. Next, you can use the `gbm` function to train a training dataset:

```
> set.seed(2)
> churn.gbm = gbm(formula = churn ~ .,distribution =
  "bernoulli",data = trainset,n.trees = 1000,interaction.depth =
  7,shrinkage = 0.01, cv.folds=3)
```

4. Then, you can obtain the summary information from the fitted model:

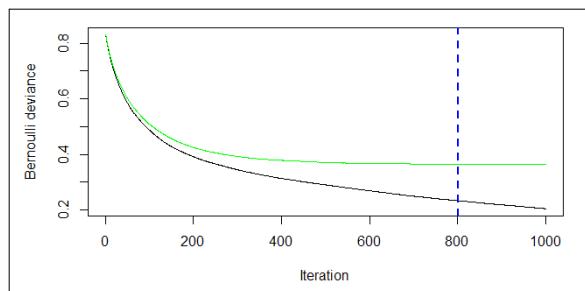
```
> summary(churn.gbm)
```

var	rel.inf
total_day_minutes	total_day_minutes 28.1217147
total_eve_minutes	total_eve_minutes 16.8097151
number_customer_service_calls	number_customer_service_calls
12.7894464	
total_intl_minutes	total_intl_minutes 9.4515822
total_intl_calls	total_intl_calls 8.1379826
international_plan	international_plan 8.0703900
total_night_minutes	total_night_minutes 4.0805153
number_vmail_messages	number_vmail_messages 3.9173515
voice_mail_plan	voice_mail_plan 2.5501480
total_night_calls	total_night_calls 2.1357970
total_day_calls	total_day_calls 1.7367888
total_eve_calls	total_eve_calls 1.4398047
total_eve_charge	total_eve_charge 0.5457486
total_night_charge	total_night_charge 0.2130152
total_day_charge	total_day_charge 0.0000000
total_intl_charge	total_intl_charge 0.0000000



5. You can obtain the best iteration using cross-validation:

```
> churn.iter = gbm.perf(churn.gbm, method="cv")
```



The performance measurement plot

6. Then, you can retrieve the odd value of the log returned from the Bernoulli loss function:

```
> churn.predict = predict(churn.gbm, testset, n.trees = churn.iter)  
> str(churn.predict)  
num [1:1018] -3.31 -2.91 -3.16 -3.47 -3.48 ...
```

7. Next, you can plot the ROC curve and get the best cut off that will have the maximum accuracy:

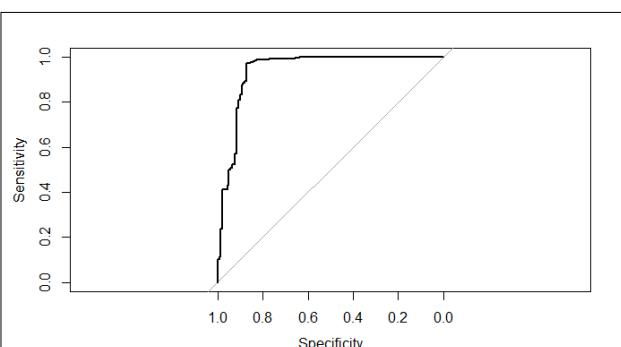
```
> churn.roc = roc(testset$churn, churn.predict)  
> plot(churn.roc)
```

Call:

```
roc.default(response = testset$churn, predictor = churn.predict)
```

```
Data: churn.predict in 141 controls (testset$churn yes) > 877  
cases (testset$churn no).
```

```
Area under the curve: 0.9393
```



The ROC curve of fitted model

8. You can retrieve the best cut off with the coords function and use this cut off to obtain the predicted label:

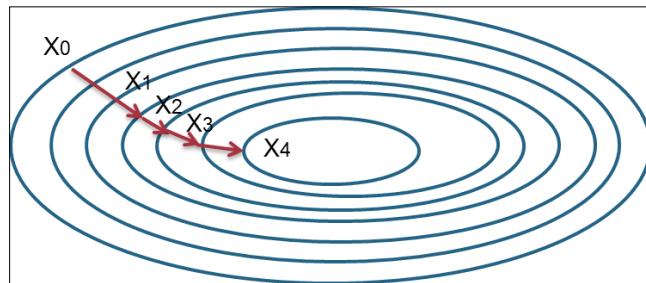
```
> coords(churn.roc, "best")  
threshold specificity sensitivity  
-0.9495258 0.8723404 0.9703535  
> churn.predict.class = ifelse(churn.predict > coords(churn.roc,  
"best") ["threshold"], "yes", "no")
```

9. Lastly, you can obtain the classification table from the predicted results:

```
> table( testset$churn, churn.predict.class)  
  
churn.predict.class  
  
no yes  
yes 18 123  
no 851 26
```

How it works...

The algorithm of gradient boosting involves, first, the process computes the deviation of residuals for each partition, and then, determines the best data partitioning in each stage. Next, the successive model will fit the residuals from the previous stage and build a new model to reduce the residual variance (an error). The reduction of the residual variance follows the functional gradient descent technique, in which it minimizes the residual variance by going down its derivative, as show here:



Gradient descent method

In this recipe, we use the gradient boosting method from `gbm` to classify the telecom churn dataset. To begin the classification, we first install and load the `gbm` package. Then, we use the `gbm` function to train the classification model. Here, as our prediction target is the `churn` attribute, which is a binary outcome, we therefore set the distribution as `bernoulli` in the distribution argument. Also, we set the 1000 trees to fit in the `n.tree` argument, the maximum depth of the variable interaction to 7 in `interaction.depth`, the learning rate of the step size reduction to 0.01 in `shrinkage`, and the number of cross-validations to 3 in `cv.folds`. After the model is fitted, we can use the `summary` function to obtain the relative influence information of each variable in the table and figure. The relative influence shows the reduction attributable to each variable in the sum of the square error. Here, we can find `total_day_minutes` is the most influential one in reducing the loss function.

Next, we use the `gbm.perf` function to find the optimum iteration. Here, we estimate the optimum number with cross-validation by specifying the `method` argument to `cv`. The function further generates two plots, where the black line plots the training error and the green one plots the validation error. The error measurement here is a `bernoulli` distribution, which we have defined earlier in the training stage. The blue dash line on the plot shows where the optimum iteration is.

Then, we use the `predict` function to obtain the odd value of a log in each testing case returned from the Bernoulli loss function. In order to get the best prediction result, one can set the `n.trees` argument to an optimum iteration number. However, as the returned value is an odd value log, we still have to determine the best cut off to determine the label. Therefore, we use the `roc` function to generate an ROC curve and get the cut off with the maximum accuracy.

Finally, we can use the function, `coords`, to retrieve the best cut off threshold and use the `ifelse` function to determine the class label from the odd value of the log. Now, we can use the `table` function to generate the classification table and see how accurate the classification model is.

There's more...

In addition to using the boosting function in the `gbm` package, one can also use the `mboost` package to perform classifications with the gradient boosting method:

1. First, install and load the `mboost` package:

```
> install.packages("mboost")
> library(mboost)
```

2. The `mboost` function only uses numeric responses; therefore, you should transform yes/no responses to numeric responses (0/1):

```
> trainset$churn = ifelse(trainset$churn == "yes", 1, 0)
```

3. Also, you should remove nonnumerical attributes, such as `voice_mail_plan` and `international_plan`:

```
> trainset$voice_mail_plan = NULL  
> trainset$international_plan = NULL
```

4. We can then use `mboost` to train the classification model:

```
> churn.mboost = mboost(churn ~ ., data=trainset, control =  
  boost_control(mstop = 10))
```

5. Use the `summary` function to obtain the details of the classification model:

```
> summary(churn.mboost)
```

Model-based Boosting

Call:

```
mboost(formula = churn ~ ., data = trainset, control = boost_  
control(mstop = 10))
```

Squared Error (Regression)

Loss function: $(y - f)^2$

Number of boosting iterations: `mstop = 10`

Step size: 0.1

Offset: 1.147732

Number of baselearners: 14

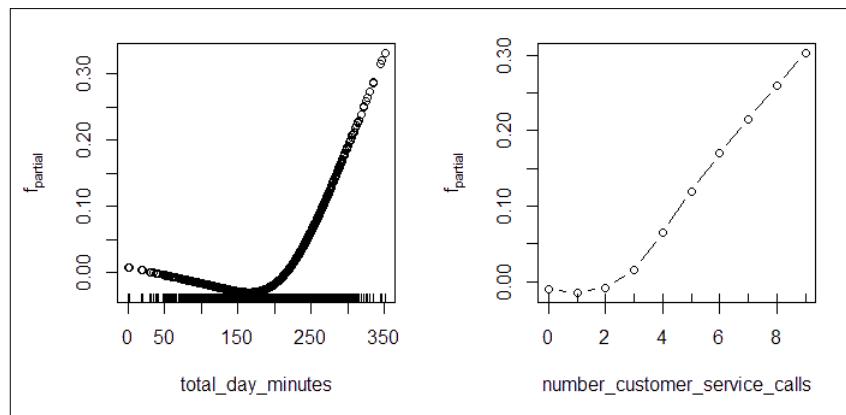
Selection frequencies:

	bbs(total_day_minutes)	bbs(number_customer_service_calls)
--	------------------------	------------------------------------

0.6	0.4
-----	-----

6. Lastly, use the plot function to draw a partial contribution plot of each attribute:

```
> par(mfrow=c(1,2))  
> plot(churn.mboost)
```



The partial contribution plot of important attributes

Calculating the margins of a classifier

A margin is a measure of the certainty of classification. This method calculates the difference between the support of a correct class and the maximum support of an incorrect class. In this recipe, we will demonstrate how to calculate the margins of the generated classifiers.

Getting ready

You need to have completed the previous recipe by storing a fitted bagging model in the variables, churn.bagging and churn.predbagging. Also, put the fitted boosting classifier in both churn.boost and churn.boost.pred.

How to do it...

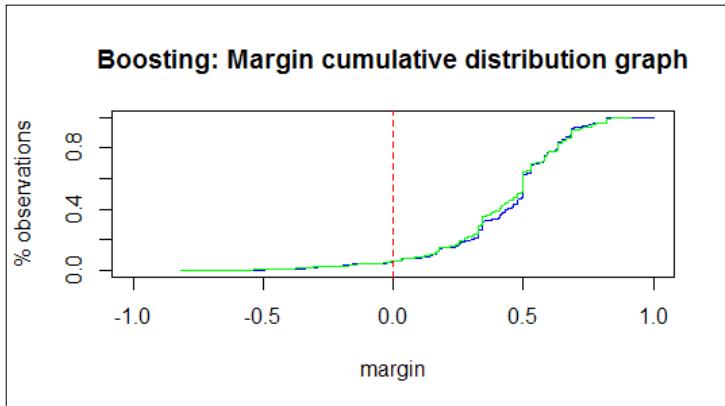
Perform the following steps to calculate the margin of each ensemble learner:

1. First, use the margins function to calculate the margins of the boosting classifiers:

```
> boost.margins = margins(churn.boost, trainset)  
> boost.pred.margins = margins(churn.boost.pred, testset)
```

2. You can then use the `plot` function to plot a marginal cumulative distribution graph of the boosting classifiers:

```
> plot(sort(boost.margins[[1]]), (1:length(boost.margins[[1]]))/length(boost.margins[[1]]), type="l", xlim=c(-1,1), main="Boosting:  
Margin cumulative distribution graph", xlab="margin", ylab="%  
observations", col = "blue")  
  
> lines(sort(boost.pred.margins[[1]]), (1:length(boost.pred.  
margins[[1]]))/length(boost.pred.margins[[1]]), type="l", col =  
"green")  
  
> abline(v=0, col="red", lty=2)
```



The margin cumulative distribution graph of using the boosting method

3. You can then calculate the percentage of negative margin matches training errors and the percentage of negative margin matches test errors:

```
> boosting.training.margin = table(boost.margins[[1]] > 0)  
  
> boosting.negative.training = as.numeric(boosting.training.  
margin[1]/boosting.training.margin[2])  
  
> boosting.negative.training  
  
[1] 0.06387868  
  
  
> boosting.testing.margin = table(boost.pred.margins[[1]] > 0)  
  
> boosting.negative.testing = as.numeric(boosting.testing.  
margin[1]/boosting.testing.margin[2])  
  
> boosting.negative.testing  
  
[1] 0.06263048
```

4. Also, you can calculate the margins of the bagging classifiers. You might see the warning message showing "no non-missing argument to min". The message simply indicates that the min/max function is applied to the numeric of the 0 length argument:

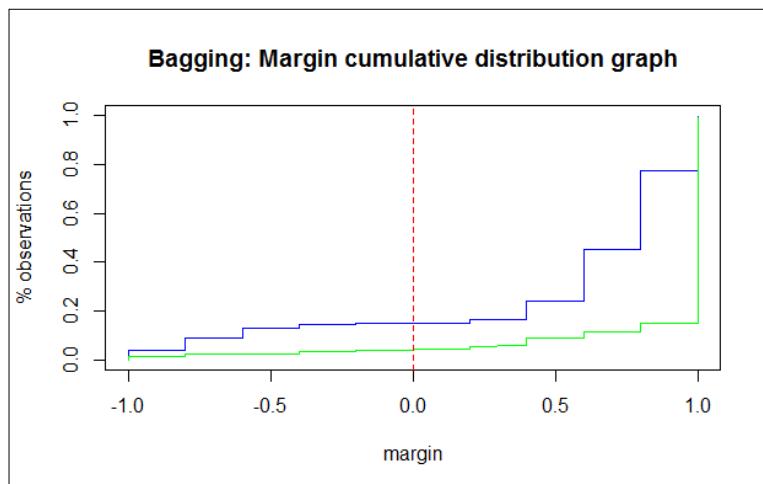
```
> bagging.margins = margins(churn.bagging, trainset)
> bagging.pred.margins = margins(churn.predbagging, testset)
```

5. You can then use the `plot` function to plot a margin cumulative distribution graph of the bagging classifiers:

```
> plot(sort(bagging.margins[[1]]), (1:length(bagging.
margins[[1]]))/length(bagging.margins[[1]]), type="l", xlim=c(-
1,1), main="Bagging: Margin cumulative distribution graph",
xlab="margin", ylab="% observations", col = "blue")

> lines(sort(bagging.pred.margins[[1]]), (1:length(bagging.pred.
margins[[1]]))/length(bagging.pred.margins[[1]]), type="l", col =
"green")

> abline(v=0, col="red", lty=2)
```



The margin cumulative distribution graph of the bagging method

6. Finally, you can then compute the percentage of negative margin matches training errors and the percentage of negative margin matches test errors:

```
> bagging.training.margin = table(bagging.margins[[1]] > 0)
> bagging.negative.training = as.numeric(bagging.training.
margin[1]/bagging.training.margin[2])
> bagging.negative.training
[1] 0.1733401
```

```

> bagging.testing.margin = table(bagging.pred.margins[[1]] > 0)
> bagging.negative.testing = as.numeric(bagging.testing.margin[1]/
bagging.testing.margin[2])
> bagging.negative.testing
[1] 0.04303279

```

How it works...

A margin is the measurement of certainty of the classification; it is computed by the support of the correct class and the maximum support of the incorrect class. The formula of margins can be formulated as:

$$\text{margin}(x_i) = \text{support}_c(x_i) - \max_{i \neq c} \text{support}_j(x_i)$$

Here, the margin of the x_i sample equals the support of a correctly classified sample (c denotes the correct class) minus the maximum support of a sample that is classified to class j (where $j \neq c$ and $j=1\dots k$). Therefore, correctly classified examples will have positive margins and misclassified examples will have negative margins. If the margin value is close to one, it means that correctly classified examples have a high degree of confidence. On the other hand, examples of uncertain classifications will have small margins.

The `margins` function calculates the margins of AdaBoost.M1, AdaBoost-SAMME, or the bagging classifier, which returns a vector of a margin. To visualize the margin distribution, one can use a margin cumulative distribution graph. In these graphs, the x-axis shows the margin and the y-axis shows the percentage of observations where the margin is less than or equal to the margin value of the x-axis. If every observation is correctly classified, the graph will show a vertical line at the margin equal to 1 (where margin = 1).

For the margin cumulative distribution graph of the boosting classifiers, we can see that there are two lines plotted on the graph, in which the green line denotes the margin of the testing dataset, and the blue line denotes the margin of the training set. The figure shows about 6.39 percent of negative margins match the training error, and 6.26 percent of negative margins match the test error. On the other hand, we can find that 17.33% of negative margins match the training error and 4.3 percent of negative margins match the test error in the margin cumulative distribution graph of the bagging classifiers. Normally, the percentage of negative margins matching the training error should be close to the percentage of negative margins that match the test error. As a result of this, we should then examine the reason why the percentage of negative margins that match the training error is much higher than the negative margins that match the test error.

See also

- ▶ If you are interested in more details on margin distribution graphs, please refer to the following source: *Kuncheva LI (2004), Combining Pattern Classifiers: Methods and Algorithms, John Wiley & Sons.*

Calculating the error evolution of the ensemble method

The adabag package provides the `errorevol` function for a user to estimate the ensemble method errors in accordance with the number of iterations. In this recipe, we will demonstrate how to use `errorevol` to show the evolution of errors of each ensemble classifier.

Getting ready

You need to have completed the previous recipe by storing the fitted bagging model in the variable, `churn.bagging`. Also, put the fitted boosting classifier in `churn.boost`.

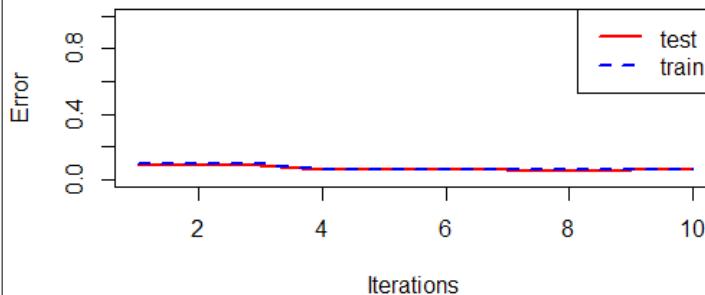
How to do it...

Perform the following steps to calculate the error evolution of each ensemble learner:

1. First, use the `errorevol` function to calculate the error evolution of the boosting classifiers:

```
> boosting.evol.train = errorevol(churn.boost, trainset)
> boosting.evol.test = errorevol(churn.boost, testset)
> plot(boosting.evol.test$error, type = "l", ylim = c(0, 1),
+       main = "Boosting error versus number of trees", xlab =
"Iterations",
+       ylab = "Error", col = "red", lwd = 2)
> lines(boosting.evol.train$error, cex = .5, col = "blue", lty =
2, lwd = 2)
> legend("topright", c("test", "train"), col = c("red", "blue"),
lty = 1:2, lwd = 2)
```

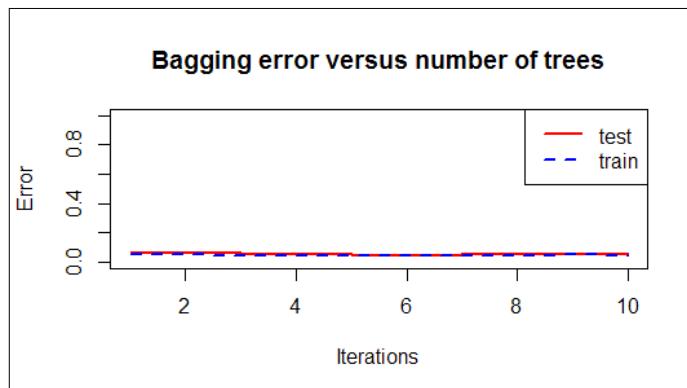
Boosting error versus number of trees



Boosting error versus number of trees

2. Next, use the errorevol function to calculate the error evolution of the bagging classifiers:

```
> bagging.evol.train = errorevol(churn.bagging, trainset)
> bagging.evol.test = errorevol(churn.bagging, testset)
> plot(bagging.evol.test$error, type = "l", ylim = c(0, 1),
+       main = "Bagging error versus number of trees", xlab =
"Iterations",
+       ylab = "Error", col = "red", lwd = 2)
> lines(bagging.evol.train$error, cex = .5, col = "blue", lty = 2,
lwd = 2)
> legend("topright", c("test", "train"), col = c("red", "blue"),
lty = 1:2, lwd = 2)
```



Bagging error versus number of trees

How it works...

The `errorest` function calculates the error evolution of AdaBoost.M1, AdaBoost-SAMME, or the bagging classifiers and returns a vector of error evolutions. In this recipe, we use the boosting and bagging models to generate error evolution vectors and graph the error versus number of trees.

The resulting graph reveals the error rate of each iteration. The trend of the error rate can help measure how fast the errors reduce, while the number of iterations increases. In addition to this, the graphs may show whether the model is over-fitted.

See also

- ▶ If the ensemble model is over-fitted, you can use the `predict.bagging` and `predict.boosting` functions to prune the ensemble model. For more information, please use the `help` function to refer to `predict.bagging` and `predict.boosting`:
 > `help(predict.bagging)`
 > `help(predict.boosting)`

Classifying data with random forest

Random forest is another useful ensemble learning method that grows multiple decision trees during the training process. Each decision tree will output its own prediction results corresponding to the input. The forest will use the voting mechanism to select the most voted class as the prediction result. In this recipe, we will illustrate how to classify data using the `randomForest` package.

Getting ready

In this recipe, we will continue to use the `telecom churn` dataset as the input data source to perform classifications with the random forest method.

How to do it...

Perform the following steps to classify data with random forest:

1. First, you have to install and load the `randomForest` package;

```
> install.packages("randomForest")
> library(randomForest)
```

2. You can then fit the random forest classifier with a training set:

```
> churn.rf = randomForest(churn ~ ., data = trainset, importance = T)
> churn.rf
```

Call:

```
randomForest(formula = churn ~ ., data = trainset, importance = T)
```

Type of random forest: classification

Number of trees: 500

No. of variables tried at each split: 4

OOB estimate of error rate: 4.88%

Confusion matrix:

	yes	no	class.error
yes	247	95	0.2777777778
no	18	1955	0.009123163

3. Next, make predictions based on the fitted model and testing dataset:

```
> churn.prediction = predict(churn.rf, testset)
```

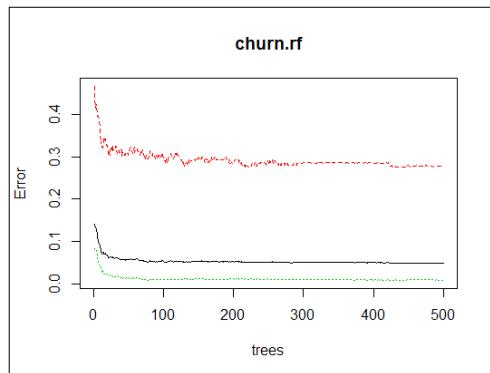
4. Similar to other classification methods, you can obtain the classification table:

```
> table(churn.prediction, testset$churn)
```

churn.prediction	yes	no
yes	110	7
no	31	870

5. You can use the `plot` function to plot the mean square error of the forest object:

```
> plot(churn.rf)
```



The mean square error of the random forest

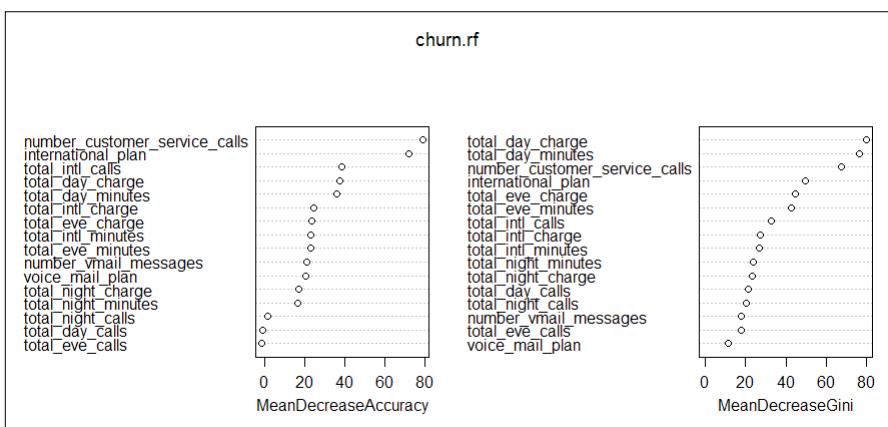
6. You can then examine the importance of each attribute within the fitted classifier:

```
> importance(churn.rf)
```

	yes	no
international_plan	66.55206691	56.5100647
voice_mail_plan	19.98337191	15.2354970
number_vmail_messages	21.02976166	14.0707195
total_day_minutes	28.05190188	27.7570444

7. Next, you can use the `varImpPlot` function to obtain the plot of variable importance:

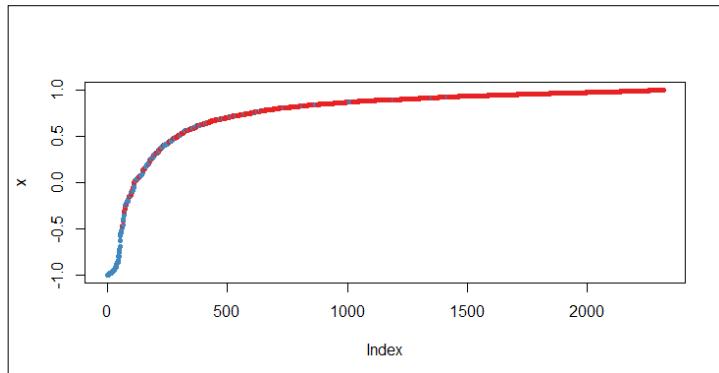
```
> varImpPlot(churn.rf)
```



The visualization of variable importance

8. You can also use the `margin` function to calculate the margins and plot the margin cumulative distribution:

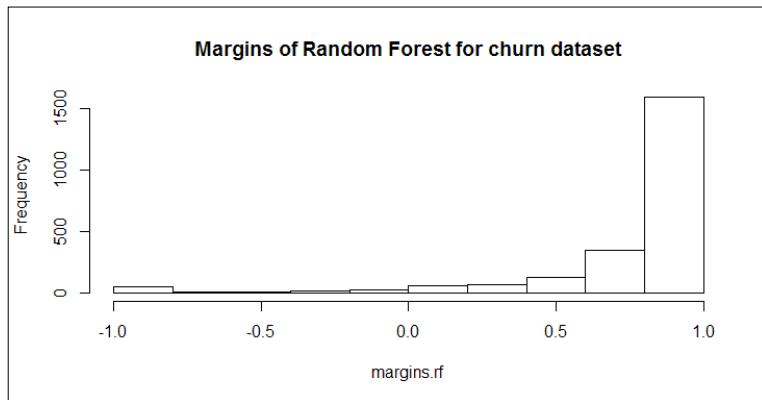
```
> margins.rf=margin(churn.rf,trainset)  
> plot(margins.rf)
```



The margin cumulative distribution graph for the random forest method

9. Furthermore, you can use a histogram to visualize the margin distribution of the random forest:

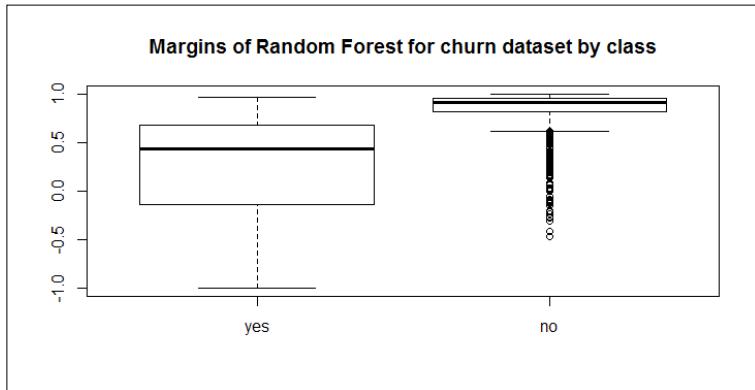
```
> hist(margins.rf,main="Margins of Random Forest for churn dataset")
```



The histogram of margin distribution

10. You can also use boxplot to visualize the margins of the random forest by class:

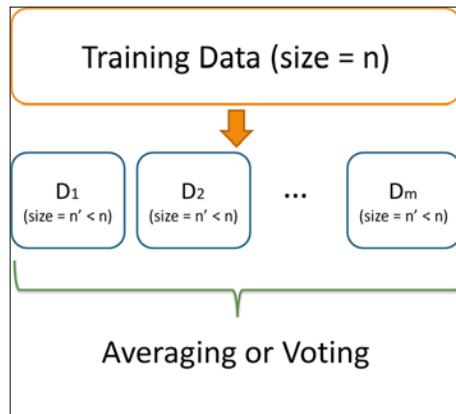
```
> boxplot(margins.rf~trainset$churn, main="Margins of Random Forest for churn dataset by class")
```



Margins of the random forest by class

How it works...

The purpose of random forest is to ensemble weak learners (for example, a single decision tree) into a strong learner. The process of developing a random forest is very similar to the bagging method, assuming that we have a training set containing N samples with M features. The process first performs bootstrap sampling, which samples N cases at random, with the replacement as the training dataset of each single decision tree. Next, in each node, the process first randomly selects m variables (where $m \ll M$), then finds the predictor variable that provides the best split among m variables. Next, the process grows the full tree without pruning. In the end, we can obtain the predicted result of an example from each single tree. As a result, we can get the prediction result by taking an average or weighted average (for regression) of an output or taking a majority vote (for classification):



A random forest uses two parameters: **ntree** (the number of trees) and **mtry** (the number of features used to find the best feature), while the bagging method only uses ntree as a parameter. Therefore, if we set mtry equal to the number of features within a training dataset, then the random forest is equal to the bagging method.

The main advantages of random forest are that it is easy to compute, it can efficiently process data, and is fault tolerant to missing or unbalanced data. The main disadvantage of random forest is that it cannot predict the value beyond the range of a training dataset. Also, it is prone to over-fitting of noisy data.

In this recipe, we employ the random forest method adapted from the `randomForest` package to fit a classification model. First, we install and load `randomForest` into an R session. We then use the random forest method to train a classification model. We set `importance = T`, which will ensure that the importance of the predictor is assessed.

Similar to the bagging and boosting methods, once the model is fitted, one can perform predictions using a fitted model on the testing dataset, and furthermore, obtain the classification table.

In order to assess the importance of each attribute, the `randomForest` package provides the `importance` and `varImpPlot` functions to either list the importance of each attribute in the fitted model or visualize the importance using either mean decrease accuracy or mean decrease gini.

Similar to `adabag`, which contains a method to calculate the margins of the bagging and boosting methods, `randomForest` provides the `margin` function to calculate the margins of the forest object. With the `plot`, `hist`, and `boxplot` functions, you can visualize the margins in different aspects to the proportion of correctly classified observations.

There's more...

Apart from the `randomForest` package, the `party` package also provides an implementation of random forest. In the following steps, we illustrate how to use the `cforest` function within the `party` package to perform classifications:

1. First, install and load the `party` package:

```
> install.packages("party")
> library(party)
```

2. You can then use the `cforest` function to fit the classification model:

```
> churn.cforest = cforest(churn ~ ., data = trainset,
  controls=cforest_unbiased(ntree=1000, mtry=5))
> churn.cforest
```

Number of trees: 1000

Response: churn

Inputs: international_plan, voice_mail_plan, number_vmail_messages, total_day_minutes, total_day_calls, total_day_charge, total_eve_minutes, total_eve_calls, total_eve_charge, total_night_minutes, total_night_calls, total_night_charge, total_intl_minutes, total_intl_calls, total_intl_charge, number_customer_service_calls

Number of observations: 2315

3. You can make predictions based on the built model and the testing dataset:

```
> churn.cforest.prediction = predict(churn.cforest, testset,  
  OOB=TRUE, type = "response")
```

4. Finally, obtain the classification table from the predicted labels and the labels of the testing dataset:

```
> table(churn.cforest.prediction, testset$churn)
```

churn.cforest.prediction	yes	no
yes	91	3
no	50	874

Estimating the prediction errors of different classifiers

At the beginning of this chapter, we discussed why we use ensemble learning and how it can improve the prediction performance compared to using just a single classifier. We now validate whether the ensemble model performs better than a single decision tree by comparing the performance of each method. In order to compare the different classifiers, we can perform a 10-fold cross-validation on each classification method to estimate test errors using erroreset from the ipred package.

Getting ready

In this recipe, we will continue to use the telecom churn dataset as the input data source to estimate the prediction errors of the different classifiers.

How to do it...

Perform the following steps to estimate the prediction errors of each classification method:

1. You can estimate the error rate of the bagging model:

```
> churn.bagging= errorest(churn ~ ., data = trainset, model =  
bagging)  
> churn.bagging
```

Call:

```
errorest.data.frame(formula = churn ~ ., data = trainset, model =  
bagging)
```

```
10-fold cross-validation estimator of misclassification error
```

```
Misclassification error: 0.0583
```

2. You can then estimate the error rate of the boosting method:

```
> install.packages("ada")  
> library(ada)  
> churn.boosting= errorest(churn ~ ., data = trainset, model =  
ada)  
> churn.boosting
```

Call:

```
errorest.data.frame(formula = churn ~ ., data = trainset, model =  
ada)
```

```
10-fold cross-validation estimator of misclassification error
```

```
Misclassification error: 0.0475
```

3. Next, estimate the error rate of the random forest model:

```
> churn.rf= errorest(churn ~ ., data = trainset, model =  
randomForest)  
> churn.rf
```

Call:

```
errorest.data.frame(formula = churn ~ ., data = trainset, model =  
randomForest)
```

10-fold cross-validation estimator of misclassification error

Misclassification error: 0.051

4. Finally, make a prediction function using `churn.predict`, and then use the function to estimate the error rate of the single decision tree:

```
> churn.predict = function(object, newdata) {predict(object,  
newdata = newdata, type = "class")}  
> churn.tree= errorest(churn ~ ., data = trainset, model =  
rpart,predict = churn.predict)  
> churn.tree
```

Call:

```
errorest.data.frame(formula = churn ~ ., data = trainset, model =  
rpart,  
predict = churn.predict)
```

10-fold cross-validation estimator of misclassification error

Misclassification error: 0.0674

How it works...

In this recipe, we estimate the error rates of four different classifiers using the `errorest` function from the `ipred` package. We compare the boosting, bagging, and random forest methods, and the single decision tree classifier. The `errorest` function performs a 10-fold cross-validation on each classifier and calculates the misclassification error. The estimation results from the four chosen models reveal that the boosting method performs the best with the lowest error rate (0.0475). The random forest method has the second lowest error rate (0.051), while the bagging method has an error rate of 0.0583. The single decision tree classifier, `rpart`, performs the worst among the four methods with an error rate equal to 0.0674. These results show that all three ensemble learning methods, boosting, bagging, and random forest, outperform a single decision tree classifier.

See also

- ▶ In this recipe we mentioned the `ada` package, which contains a method to perform stochastic boosting. For those interested in this package, please refer to: *Additive Logistic Regression: A Statistical View of Boosting* by Friedman, et al. (2000).

8

Clustering

In this chapter, we will cover the following topics:

- ▶ Clustering data with hierarchical clustering
- ▶ Cutting a tree into clusters
- ▶ Clustering data with the k-means method
- ▶ Drawing a bivariate cluster plot
- ▶ Comparing clustering methods
- ▶ Extracting silhouette information from clustering
- ▶ Obtaining optimum clusters for k-means
- ▶ Clustering data with the density-based method
- ▶ Clustering data with the model-based method
- ▶ Visualizing a dissimilarity matrix
- ▶ Validating clusters externally

Introduction

Clustering is a technique used to group similar objects (close in terms of distance) together in the same group (cluster). Unlike supervised learning methods (for example, classification and regression) covered in the previous chapters, a clustering analysis does not use any label information, but simply uses the similarity between data features to group them into clusters.

Clustering can be widely adapted in the analysis of businesses. For example, a marketing department can use clustering to segment customers by personal attributes. As a result of this, different marketing campaigns targeting various types of customers can be designed.

The four most common types of clustering methods are hierarchical clustering, k-means clustering, model-based clustering, and density-based clustering:

- ▶ **Hierarchical clustering:** It creates a hierarchy of clusters, and presents the hierarchy in a dendrogram. This method does not require the number of clusters to be specified at the beginning.
- ▶ **k-means clustering:** It is also referred to as flat clustering. Unlike hierarchical clustering, it does not create a hierarchy of clusters, and it requires the number of clusters as an input. However, its performance is faster than hierarchical clustering.
- ▶ **Model-based clustering:** Both hierarchical clustering and k-means clustering use a heuristic approach to construct clusters, and do not rely on a formal model. Model-based clustering assumes a data model and applies an EM algorithm to find the most likely model components and the number of clusters.
- ▶ **Density-based clustering:** It constructs clusters in regard to the density measurement. Clusters in this method have a higher density than the remainder of the dataset.

In the following recipes, we will discuss how to use these four clustering techniques to cluster data. We discuss how to validate clusters internally, using within clusters the sum of squares, average silhouette width, and externally, with ground truth.

Clustering data with hierarchical clustering

Hierarchical clustering adopts either an agglomerative or divisive method to build a hierarchy of clusters. Regardless of which approach is adopted, both first use a distance similarity measure to combine or split clusters. The recursive process continues until there is only one cluster left or you cannot split more clusters. Eventually, we can use a dendrogram to represent the hierarchy of clusters. In this recipe, we will demonstrate how to cluster customers with hierarchical clustering.

Getting ready

In this recipe, we will perform hierarchical clustering on customer data, which involves segmenting customers into different groups. You can download the data from this Github page: https://github.com/ywchiu/ml_R_cookbook/tree/master/CH9.

How to do it...

Perform the following steps to cluster customer data into a hierarchy of clusters:

1. First, you need to load data from `customer.csv` and save it into `customer`:

```
> customer= read.csv('customer.csv', header=TRUE)
> head(customer)
```

ID	Visit.Time	Average.Expense	Sex	Age
1	1	3	5.7	0 10
2	2	5	14.5	0 27
3	3	16	33.5	0 32
4	4	5	15.9	0 30
5	5	16	24.9	0 23
6	6	3	12.0	0 15

2. You can then examine the dataset structure:

```
> str(customer)

'data.frame': 60 obs. of 5 variables:
 $ ID           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Visit.Time   : int  3 5 16 5 16 3 12 14 6 3 ...
 $ Average.Expense: num  5.7 14.5 33.5 15.9 24.9 12 28.5 18.8 23.8
5.3 ...
 $ Sex          : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Age          : int  10 27 32 30 23 15 33 27 16 11 ...
```

3. Next, you should normalize the customer data into the same scale:

```
> customer = scale(customer[, -1])
```

4. You can then use agglomerative hierarchical clustering to cluster the customer data:

```
> hc = hclust(dist(customer, method = "euclidean"), method = "ward.D2")
> hc
```

Call:

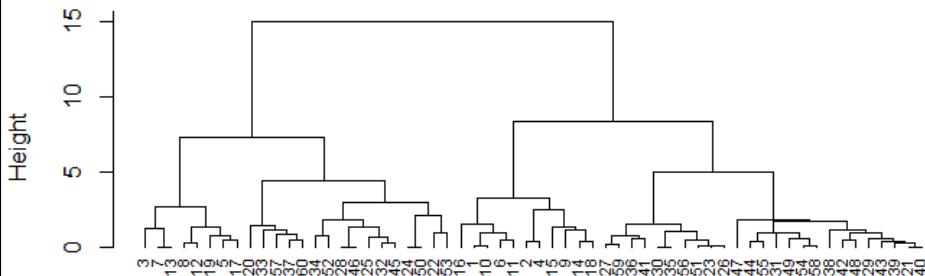
```
hclust(d = dist(customer, method = "euclidean"), method = "ward.D2")
```

```
Cluster method  : ward.D2
Distance       : euclidean
Number of objects: 60
```

5. Lastly, you can use the `plot` function to plot the dendrogram:

```
> plot(hc, hang = -0.01, cex = 0.7)
```

Cluster Dendrogram



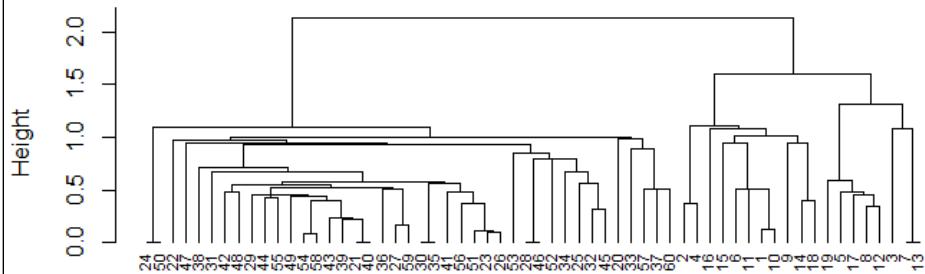
```
dist(customer, method = "euclidean")  
hclust (*, "ward.D2")
```

The dendrogram of hierarchical clustering using "ward.D2"

6. Additionally, you can use the `single` method to perform hierarchical clustering and see how the generated dendrogram differs from the previous:

```
> hc2 = hclust(dist(customer), method="single")  
> plot(hc2, hang = -0.01, cex = 0.7)
```

Cluster Dendrogram



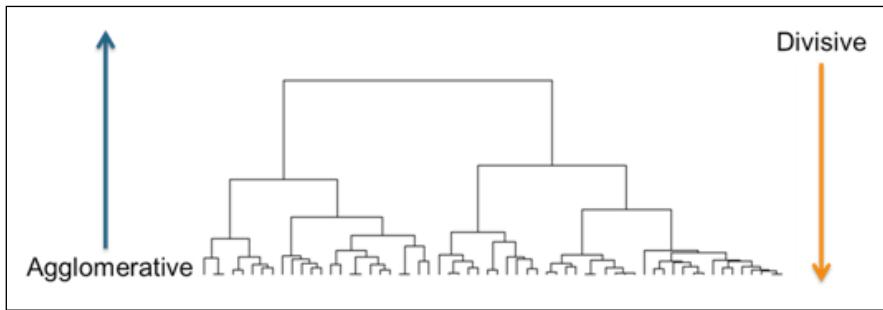
```
dist(customer)  
hclust (*, "single")
```

The dendrogram of hierarchical clustering using "single"

How it works...

Hierarchical clustering is a clustering technique that tries to build a hierarchy of clusters iteratively. Generally, there are two approaches to build hierarchical clusters:

- ▶ **Agglomerative hierarchical clustering:** This is a bottom-up approach. Each observation starts in its own cluster. We can then compute the similarity (or the distance) between each cluster and then merge the two most similar ones at each iteration until there is only one cluster left.
- ▶ **Divisive hierarchical clustering:** This is a top-down approach. All observations start in one cluster, and then we split the cluster into the two least dissimilar clusters recursively until there is one cluster for each observation:



An illustration of hierarchical clustering

Before performing hierarchical clustering, we need to determine how similar the two clusters are. Here, we list some common distance functions used for the measurement of similarity:

- ▶ **Single linkage:** This refers to the shortest distance between two points in each cluster:

$$\text{dist}(C_i, C_j) = \min_{a \in C_i, b \in C_j} \text{dist}(a, b)$$

- ▶ **Complete linkage:** This refers to the longest distance between two points in each cluster:

$$\text{dist}(C_i, C_j) = \max_{a \in C_i, b \in C_j} \text{dist}(a, b)$$

- ▶ **Average linkage:** This refers to the average distance between two points in each cluster (where $|C_i|$ is the size of cluster C_i and $|C_j|$ is the size of cluster C_j):

$$\text{dist}(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{a \in C_i, b \in C_j} \text{dist}(a, b)$$

- ▶ **Ward method:** This refers to the sum of the squared distance from each point to the mean of the merged clusters (where μ is the mean vector of $C_i \cup C_j$):

$$\text{dist}(C_i, C_j) = \sum_{a \in C_i \cup C_j} \|a - \mu\|$$

In this recipe, we perform hierarchical clustering on customer data. First, we load the data from `customer.csv`, and then load it into the customer data frame. Within the data, we find five variables of customer account information, which are ID, number of visits, average expense, sex, and age. As the scale of each variable varies, we use the `scale` function to normalize the scale.

After the scales of all the attributes are normalized, we perform hierarchical clustering using the `hclust` function. We use the Euclidean distance as distance metrics, and use Ward's minimum variance method to perform agglomerative clustering.

Finally, we use the `plot` function to plot the dendrogram of hierarchical clusters. We specify `hang` to display labels at the bottom of the dendrogram, and use `cex` to shrink the label to 70 percent of the normal size. In order to compare the differences using the `ward.D2` and `single` methods to generate a hierarchy of clusters, we draw another dendrogram using `single` in the preceding figure (step 6).

There's more...

You can choose a different distance measure and method while performing hierarchical clustering. For more details, you can refer to the documents for `dist` and `hclust`:

```
> ? dist
> ? hclust
```

In this recipe, we use `hclust` to perform agglomerative hierarchical clustering; if you would like to perform divisive hierarchical clustering, you can use the `diana` function:

1. First, you can use `diana` to perform divisive hierarchical clustering:

```
> dv = diana(customer, metric = "euclidean")
```

2. Then, you can use `summary` to obtain the summary information:

```
> summary(dv)
```

3. Lastly, you can plot a dendrogram and banner with the `plot` function:

```
> plot(dv)
```

If you are interested in drawing a horizontal dendrogram, you can use the `dendextend` package. Use the following procedure to generate a horizontal dendrogram:

1. First, install and load the `dendextend` and `magrittr` packages (if your R version is 3.1 and above, you do not have to install and load the `magrittr` package):

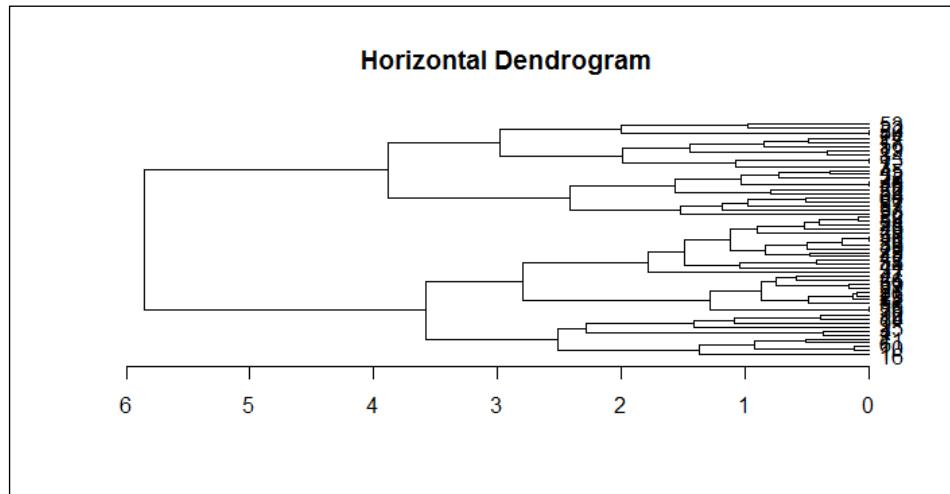
```
> install.packages("dendextend")
> library(dendextend)
> install.packages("magrittr")
> library(magrittr)
```

2. Set up the dendrogram:

```
> dend = customer %>% dist %>% hclust %>% as.dendrogram
```

3. Finally, plot the horizontal dendrogram:

```
dend %>% plot(horiz=TRUE, main = "Horizontal Dendrogram")
```



The horizontal dendrogram

Cutting trees into clusters

In a dendrogram, we can see the hierarchy of clusters, but we have not grouped data into different clusters yet. However, we can determine how many clusters are within the dendrogram and cut the dendrogram at a certain tree height to separate the data into different groups. In this recipe, we demonstrate how to use the `cutree` function to separate the data into a given number of clusters.

Getting ready

In order to perform the `cutree` function, you need to have the previous recipe completed by generating the `hclust` object, `hc`.

How to do it...

Perform the following steps to cut the hierarchy of clusters into a given number of clusters:

1. First, categorize the data into four groups:

```
> fit = cutree(hc, k = 4)
```

2. You can then examine the cluster labels for the data:

```
> fit  
[1] 1 1 2 1 2 1 2 2 1 1 1 2 2 1 1 1 2 1 2 3 4 3 4 3 3 4 4 4 3 4 3 3 4 4 4 4 3 4  
[30] 4 4 3 3 3 4 4 3 4 4 4 4 4 4 4 4 3 3 4 4 4 4 3 4 3 3 4 4 4 4 3 4 3 4  
[59] 4 3
```

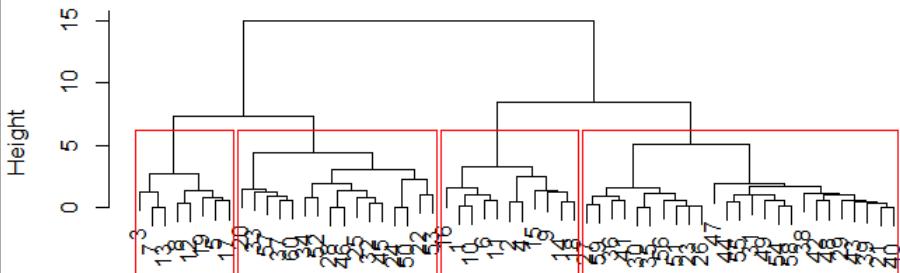
3. Count the number of data within each cluster:

```
> table(fit)  
fit  
 1 2 3 4  
11 8 16 25
```

4. Finally, you can visualize how data is clustered with the red rectangle border:

```
> plot(hc)  
> rect.hclust(hc, k = 4 , border="red")
```

Cluster Dendrogram



```
dist(customer, method = "euclidean")
hclust (*, "ward.D2")
```

Using the red rectangle border to distinguish different clusters within the dendrogram

How it works...

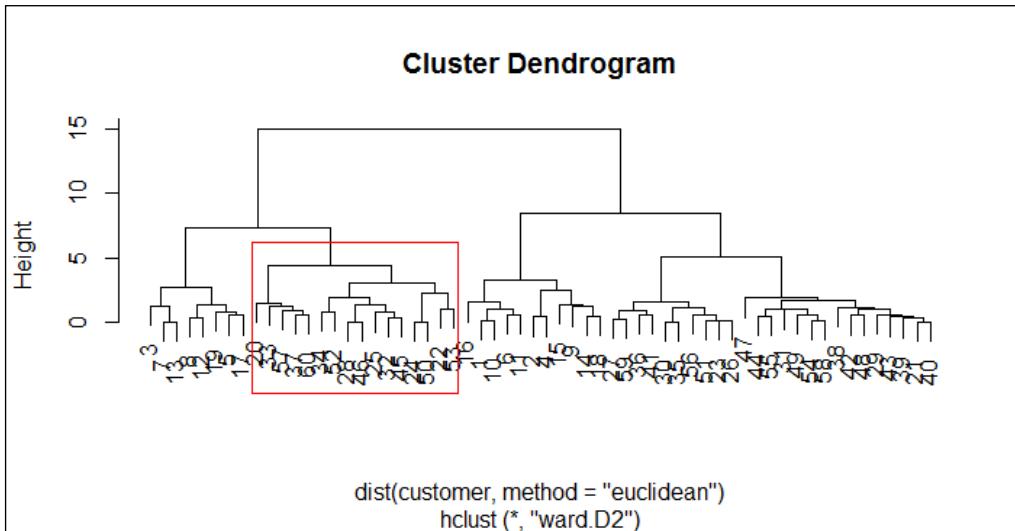
We can determine the number of clusters from the dendrogram in the preceding figure. In this recipe, we determine there should be four clusters within the tree. Therefore, we specify the number of clusters as 4 in the `cutree` function. Besides using the number of clusters to cut the tree, you can specify the height as the `cut` tree parameter.

Next, we can output the cluster labels of the data and use the `table` function to count the number of data within each cluster. From the counting table, we find that most of the data is in cluster 4. Lastly, we can draw red rectangles around the clusters to show how data is categorized into the four clusters with the `rect.hclust` function.

There's more...

Besides drawing rectangles around all hierarchical clusters, you can place a red rectangle around a certain cluster:

```
> rect.hclust(hc, k = 4 , which =2, border="red")
```



Drawing a red rectangle around a certain cluster.

Also, you can color clusters in different colors with a red rectangle around the clusters by using the dendextend package. You have to complete the instructions outlined in the *There's more* section of the previous recipe and perform the following steps:

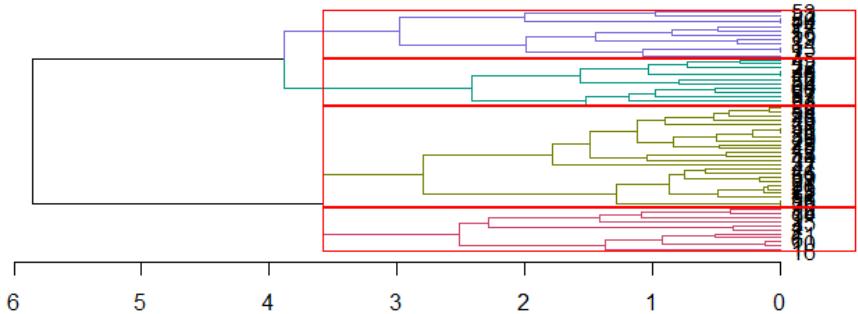
1. Color the branch according to the cluster it belongs to:

```
> dend %>% color_branches(k=4) %>% plot(horiz=TRUE, main =  
"Horizontal Dendrogram")
```

2. You can then add a red rectangle around the clusters:

```
> dend %>% rect.dendrogram(k=4,horiz=TRUE)
```

Horizontal Dendrogram

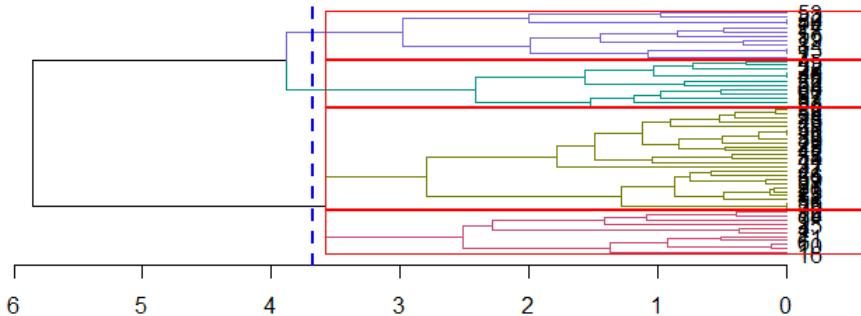


Drawing red rectangles around clusters within a horizontal dendrogram

- Finally, you can add a line to show the tree cutting location:

```
> abline(v = heights_per_k.dendrogram(dend) ["4"] + .1, lwd = 2,  
lty = 2, col = "blue")
```

Horizontal Dendrogram



Drawing a cutting line within a horizontal dendrogram

Clustering data with the k-means method

k-means clustering is a flat clustering technique, which produces only one partition with k clusters. Unlike hierarchical clustering, which does not require a user to determine the number of clusters at the beginning, the k-means method requires this to be determined first. However, k-means clustering is much faster than hierarchical clustering as the construction of a hierarchical tree is very time consuming. In this recipe, we will demonstrate how to perform k-means clustering on the customer dataset.

Getting ready

In this recipe, we will continue to use the customer dataset as the input data source to perform k-means clustering.

How to do it...

Perform the following steps to cluster the `customer` dataset with the k-means method:

1. First, you can use `kmeans` to cluster the customer data:

```
> set.seed(22)
> fit = kmeans(customer, 4)
> fit
K-means clustering with 4 clusters of sizes 8, 11, 16, 25
```

Cluster means:

	Visit.Time	Average.Expense	Sex	Age
1	1.3302016	1.0155226	-1.4566845	0.5591307
2	-0.7771737	-0.5178412	-1.4566845	-0.4774599
3	0.8571173	0.9887331	0.6750489	1.0505015
4	-0.6322632	-0.7299063	0.6750489	-0.6411604

Clustering vector:

```
[1] 2 2 1 2 1 2 1 1 2 2 2 1 1 2 2 2 1 2 1 3 4 3 4 3 3 4 4 4 3
[29] 4 4 4 3 3 3 4 4 3 4 4 4 4 4 4 3 3 4 4 4 3 4 3 3 4 4 4 4 3
[57] 3 4 4 3
```

Within cluster sum of squares by cluster:

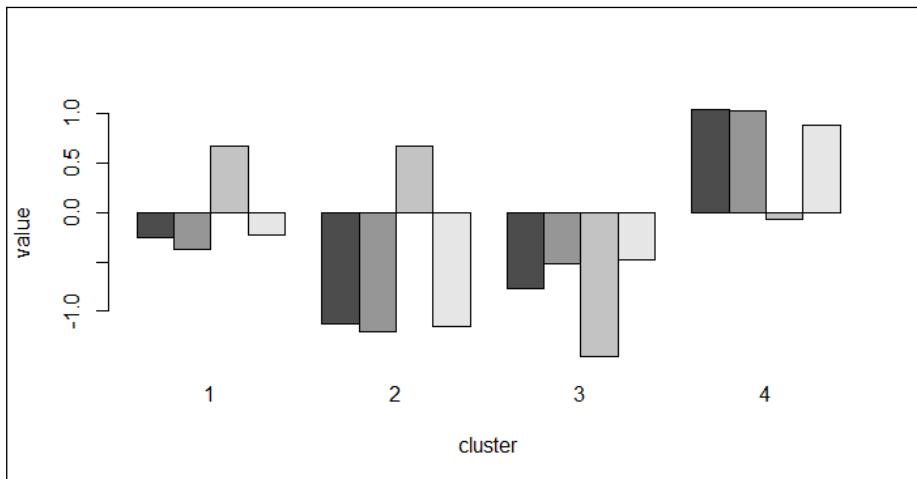
```
[1] 5.90040 11.97454 22.58236 20.89159
(between_SS / total_SS = 74.0 %)
```

Available components:

```
[1] "cluster"      "centers"       "totss"
[4] "withinss"     "tot.withinss" "betweenss"
[7] "size"         "iter"          "ifault
```

2. You can then inspect the center of each cluster using barplot:

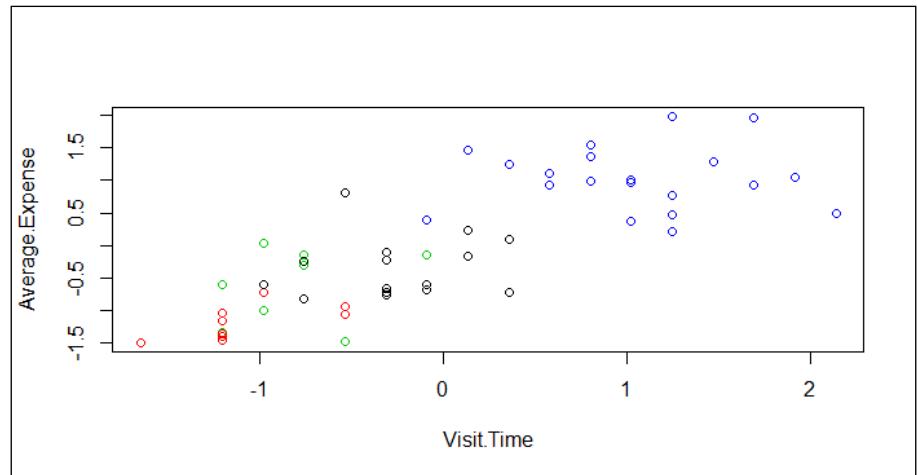
```
> barplot(t(fit$centers), beside = TRUE, xlab="cluster",  
y whole="value")
```



The barplot of centers of different attributes in four clusters

3. Lastly, you can draw a scatter plot of the data and color the points according to the clusters:

```
> plot(customer, col = fit$cluster)
```



The scatter plot showing data colored with regard to its cluster label

How it works...

k-means clustering is a method of partitioning clustering. The goal of the algorithm is to partition n objects into k clusters, where each object belongs to the cluster with the nearest mean. The objective of the algorithm is to minimize the **within-cluster sum of squares (WCSS)**. Assuming x is the given set of observations, $S = \{S_1, S_2 \dots S_k\}$ denotes k partitions, and μ_i is the mean of S_i , then we can formulate the WCSS function as follows:

$$f = \sum_{i=1}^k \sum_{x \in S_i} \|x - \mu_i\|^2$$

The process of k-means clustering can be illustrated by the following five steps:

1. Specify the number of k clusters.
2. Randomly create k partitions.
3. Calculate the center of the partitions.
4. Associate objects closest to the cluster center.
5. Repeat steps 2, 3, and 4 until the WCSS changes very little (or is minimized).

In this recipe, we demonstrate how to use k-means clustering to cluster customer data. In contrast to hierarchical clustering, k-means clustering requires the user to input the number of K . In this example, we use $K=4$. Then, the output of a fitted model shows the size of each cluster, the cluster means of four generated clusters, the cluster vectors with regard to each data point, the within cluster sum of squares by the clusters, and other available components.

Further, you can draw the centers of each cluster in a bar plot, which will provide more details on how each attribute affects the clustering. Lastly, we plot the data point in a scatter plot and use the fitted cluster labels to assign colors with regard to the cluster label.

See also

- ▶ In k-means clustering, you can specify the algorithm used to perform clustering analysis. You can specify either Hartigan-Wong, Lloyd, Forgy, or MacQueen as the clustering algorithm. For more details, please use the `help` function to refer to the document for the `kmeans` function:

```
>help(kmeans)
```

Drawing a bivariate cluster plot

In the previous recipe, we employed the k-means method to fit data into clusters. However, if there are more than two variables, it is impossible to display how data is clustered in two dimensions. Therefore, you can use a bivariate cluster plot to first reduce variables into two components, and then use components, such as axis and circle, as clusters to show how data is clustered. In this recipe, we will illustrate how to create a bivariate cluster plot.

Getting ready

In this recipe, we will continue to use the `customer` dataset as the input data source to draw a bivariate cluster plot.

How to do it...

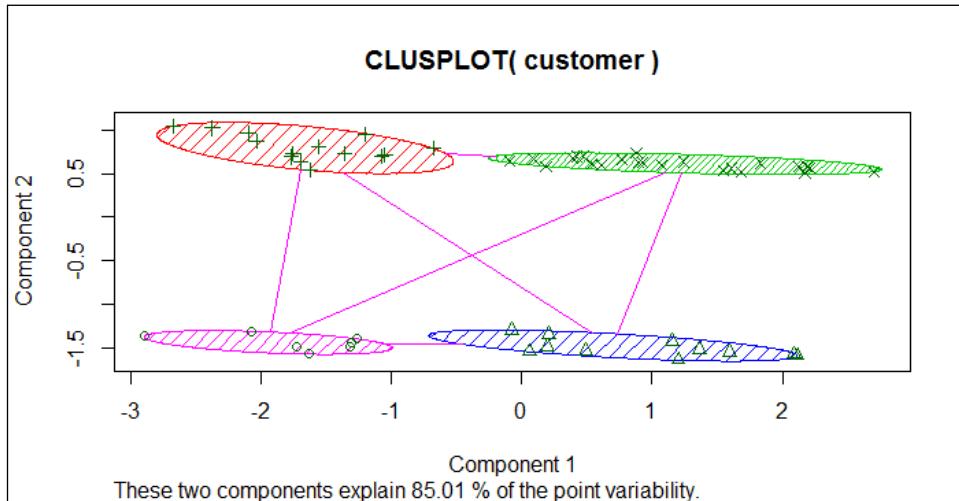
Perform the following steps to draw a bivariate cluster plot:

1. Install and load the cluster package:

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

2. You can then draw a bivariate cluster plot:

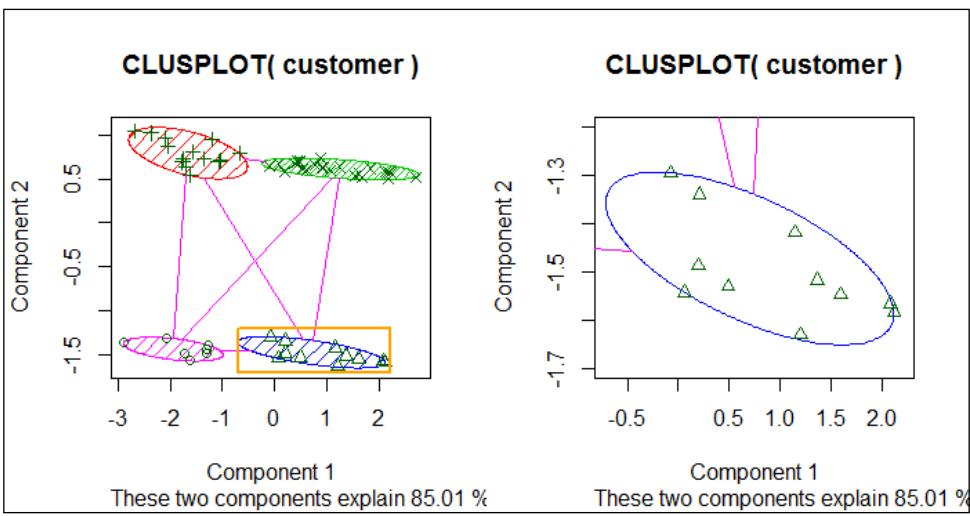
```
> clusplot(customer, fit$cluster, color=TRUE, shade=TRUE)
```



The bivariate clustering plot of the customer dataset

3. You can also zoom into the bivariate cluster plot:

```
> par(mfrow= c(1,2))
> clusplot(customer, fit$cluster, color=TRUE, shade=TRUE)
> rect(-0.7,-1.7, 2.2,-1.2, border = "orange", lwd=2)
> clusplot(customer, fit$cluster, color = TRUE, xlim = c(-0.7,2.2), ylim = c(-1.7,-1.2))
```



The zoom-in of the bivariate clustering plot

How it works...

In this recipe, we draw a bivariate cluster plot to show how data is clustered. To draw a bivariate cluster plot, we first need to install the `cluster` package and load it into R. We then use the `clusplot` function to draw a bivariate cluster plot from a customer dataset. In the `clusplot` function, we can set `shade` to `TRUE` and `color` to `TRUE` to display a cluster with colors and shades. As per the preceding figure (step 2) we found that the bivariate uses two components, which explains 85.01 percent of point variability, as the x-axis and y-axis. The data points are then scattered on the plot in accordance with component 1 and component 2. Data within the same cluster is circled in the same color and shade.

Besides drawing the four clusters in a single plot, you can use `rect` to add a rectangle around a specific area within a given x-axis and y-axis range. You can then zoom into the plot to examine the data within each cluster by using `xlim` and `ylim` in the `clusplot` function.

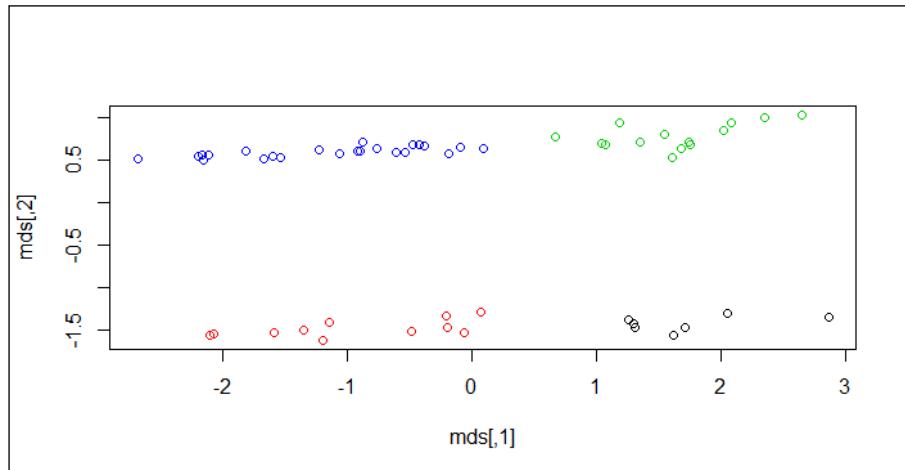
There's more

The `clusplot` function uses `princomp` and `cmdscale` to reduce the original feature dimension to the principal component. Therefore, one can see how data is clustered in a single plot with these two components as the x-axis and y-axis. To learn more about `princomp` and `cmdscale`, one can use the `help` function to view related documents:

```
> help(cmdscale)
> help(princomp)
```

For those interested in how to use `cmdscale` to reduce the dimensions, please perform the following steps:

```
> mds = cmdscale(dist(customer), k = 2)
> plot(mds, col = fit$cluster)
```



The scatter plot of data with regard to scaled dimensions

Comparing clustering methods

After fitting data into clusters using different clustering methods, you may wish to measure the accuracy of the clustering. In most cases, you can use either intracluster or intercluster metrics as measurements. We now introduce how to compare different clustering methods using `cluster.stat` from the `fpc` package.

Getting ready

In order to perform a clustering method comparison, one needs to have the previous recipe completed by generating the `customer` dataset.

How to do it...

Perform the following steps to compare clustering methods:

1. First, install and load the `fpc` package:

```
> install.packages("fpc")
> library(fpc)
```

2. You then need to use hierarchical clustering with the `single` method to cluster customer data and generate the object `hc_single`:

```
> single_c = hclust(dist(customer), method="single")
> hc_single = cutree(single_c, k = 4)
```

3. Use hierarchical clustering with the `complete` method to cluster customer data and generate the object `hc_complete`:

```
> complete_c = hclust(dist(customer), method="complete")
> hc_complete = cutree(complete_c, k = 4)
```

4. You can then use k-means clustering to cluster customer data and generate the object `km`:

```
> set.seed(22)
> km = kmeans(customer, 4)
```

5. Next, retrieve the cluster validation statistics of either clustering method:

```
> cs = cluster.stats(dist(customer), km$cluster)
```

6. Most often, we focus on using `within.cluster.ss` and `avg.silwidth` to validate the clustering method:

```
> cs[c("within.cluster.ss", "avg.silwidth")]
$within.cluster.ss
[1] 61.3489

$avg.silwidth
[1] 0.4640587
```

7. Finally, we can generate the cluster statistics of each clustering method and list them in a table:

```
> sapply(list(kmeans = km$cluster, hc_single = hc_single, hc_
  complete = hc_complete), function(c) cluster.stats(dist(customer),
  c) [c("within.cluster.ss", "avg.silwidth")])
      kmeans    hc_single hc_complete
within.cluster.ss 61.3489   136.0092  65.94076
avg.silwidth      0.4640587 0.2481926 0.4255961
```

How it works...

In this recipe, we demonstrate how to validate clusters. To validate a clustering method, we often employ two techniques: intercluster distance and intracluster distance. In these techniques, the higher the intercluster distance, the better it is, and the lower the intracluster distance, the better it is. In order to calculate related statistics, we can apply `cluster.stat` from the `fpc` package on the fitted clustering object.

From the output, the `within.cluster.ss` measurement stands for the within clusters sum of squares, and `avg.silwidth` represents the average silhouette width. The `within.cluster.ss` measurement shows how closely related objects are in clusters; the smaller the value, the more closely related objects are within the cluster. On the other hand, a silhouette is a measurement that considers how closely related objects are within the cluster and how clusters are separated from each other. Mathematically, we can define the silhouette width for each point x as follows:

$$\text{Silhouette}(x) = \frac{b(x) - a(x)}{\max([b(x), a(x)])}$$

In the preceding equation, $a(x)$ is the average distance between x and all other points within the cluster, and $b(x)$ is the minimum of the average distances between x and the points in the other clusters. The silhouette value usually ranges from 0 to 1; a value closer to 1 suggests the data is better clustered.

The summary table generated in the last step shows that the complete hierarchical clustering method outperforms a single hierarchical clustering method and k-means clustering in `within.cluster.ss` and `avg.silwidth`.

See also

- ▶ The `kmeans` function also outputs statistics (for example, `withinss` and `betweenss`) for users to validate a clustering method:

```
> set.seed(22)
> km = kmeans(customer, 4)
> km$withinss
[1] 5.90040 11.97454 22.58236 20.89159
> km$betweenss
[1] 174.6511
```

Extracting silhouette information from clustering

Silhouette information is a measurement to validate a cluster of data. In the previous recipe, we mentioned that the measurement of a cluster involves the calculation of how closely the data is clustered within each cluster, and measures how far different clusters are apart from each other. The silhouette coefficient combines the measurement of the intracluster and intercluster distance. The output value typically ranges from 0 to 1; the closer to 1, the better the cluster is. In this recipe, we will introduce how to compute silhouette information.

Getting ready

In order to extract the silhouette information from a cluster, you need to have the previous recipe completed by generating the `customer` dataset.

How to do it...

Perform the following steps to compute the silhouette information:

1. Use `kmeans` to generate a k-means object, `km`:

```
> set.seed(22)
> km = kmeans(customer, 4)
```

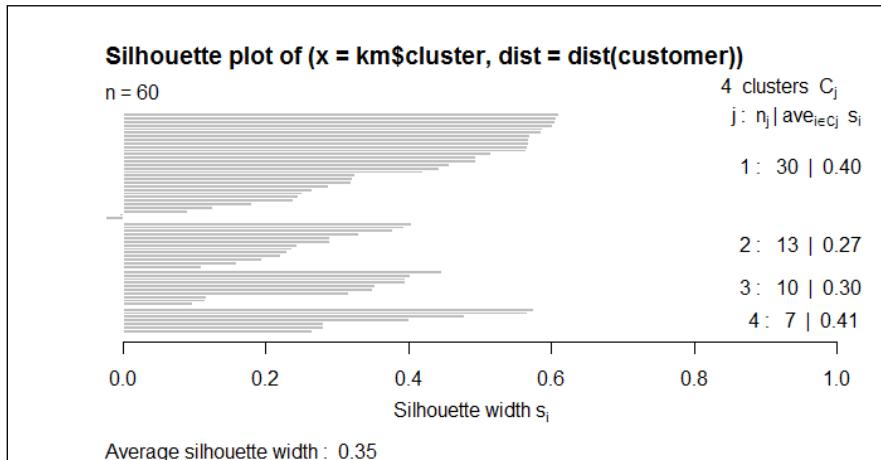
2. You can then compute the silhouette information:

```
> kms = silhouette(km$cluster,dist(customer))
> summary(kms)

Silhouette of 60 units in 4 clusters from silhouette.default(x =
  km$cluster, dist = dist(customer)) :
  Cluster sizes and average silhouette widths:
    8          11          16          25
  0.5464597  0.4080823  0.3794910  0.5164434
  Individual silhouette widths:
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
  0.1931  0.4030  0.4890  0.4641  0.5422  0.6333
```

3. Next, you can plot the silhouette information:

```
> plot(kms)
```



The silhouette plot of the k-means clustering result

How it works...

In this recipe, we demonstrate how to use the silhouette plot to validate clusters. You can first retrieve the silhouette information, which shows cluster sizes, the average silhouette widths, and individual silhouette widths. The silhouette coefficient is a value ranging from 0 to 1; the closer to 1, the better the quality of the cluster.

Lastly, we use the `plot` function to draw a silhouette plot. The left-hand side of the plot shows the number of horizontal lines, which represent the number of clusters. The right-hand column shows the mean similarity of the plot of its own cluster minus the mean similarity of the next similar cluster. The average silhouette width is presented at the bottom of the plot.

See also

- For those interested in how silhouettes are computed, please refer to the Wikipedia entry for **Silhouette Value**: http://en.wikipedia.org/wiki/Silhouette_%28clustering%29

Obtaining the optimum number of clusters for k-means

While k-means clustering is fast and easy to use, it requires k to be the input at the beginning. Therefore, we can use the sum of squares to determine which k value is best for finding the optimum number of clusters for k-means. In the following recipe, we will discuss how to find the optimum number of clusters for the k-means clustering method.

Getting ready

In order to find the optimum number of clusters, you need to have the previous recipe completed by generating the *customer* dataset.

How to do it...

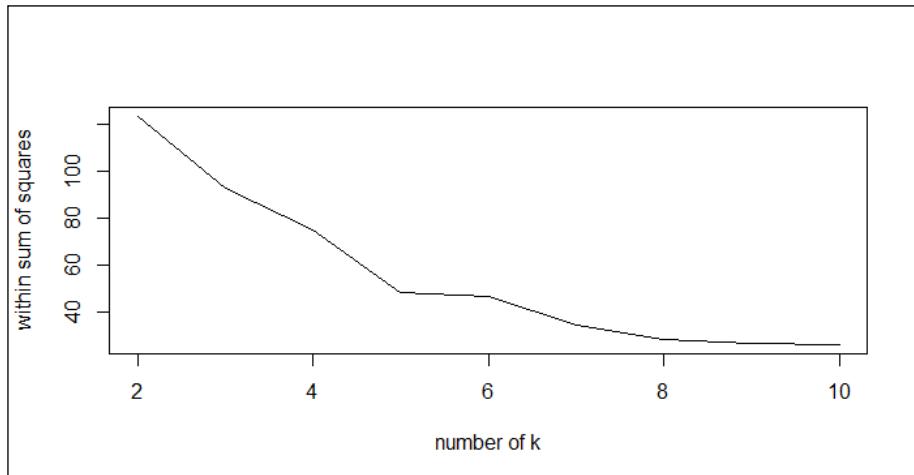
Perform the following steps to find the optimum number of clusters for the k-means clustering:

1. First, calculate the within sum of squares (withinss) of different numbers of clusters:

```
> nk = 2:10
> set.seed(22)
> WSS = sapply(nk, function(k) {
+   kmeans(customer, centers=k)$tot.withinss
+ })
> WSS
[1] 123.49224  88.07028  61.34890  48.76431  47.20813
[6] 45.48114  29.58014  28.87519  23.21331
```

2. You can then use a line plot to plot the within sum of squares with a different number of k:

```
> plot(nk, WSS, type="l", xlab= "number of k", ylab="within sum of squares")
```



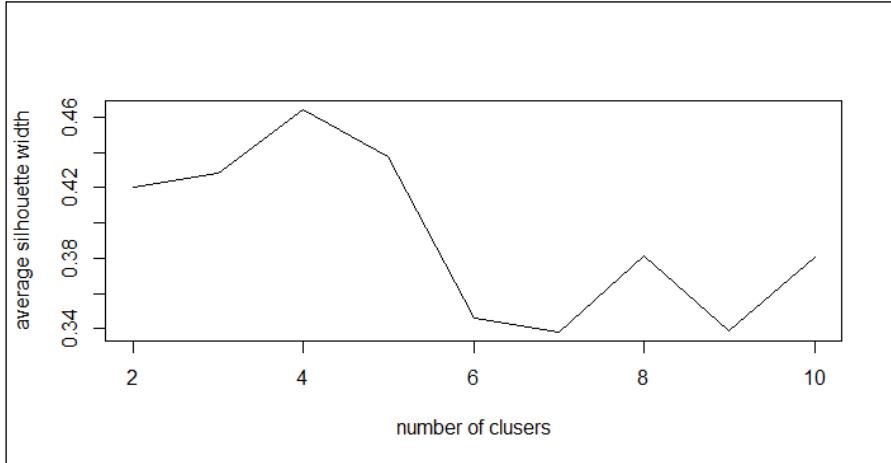
The line plot of the within sum of squares with regard to the different number of k

3. Next, you can calculate the average silhouette width (avg.silwidth) of different numbers of clusters:

```
> SW = sapply(nk, function(k) {  
+   cluster.stats(dist(customer), kmeans(customer,  
centers=k)$cluster)$avg.silwidth  
+ })  
> SW  
[1] 0.4203896 0.4278904 0.4640587 0.4308448 0.3481157  
[6] 0.3320245 0.4396910 0.3417403 0.4070539
```

4. You can then use a line plot to plot the average silhouette width with a different number of k:

```
> plot(nk, SW, type="l", xlab="number of clusters", ylab="average  
silhouette width")
```



The line plot of average silhouette width with regard to the different number of k

5. Retrieve the maximum number of clusters:

```
> nk[which.max(SW)]  
[1] 4
```

How it works...

In this recipe, we demonstrate how to find the optimum number of clusters by iteratively getting within the sum of squares and the average silhouette value. For the within sum of squares, lower values represent clusters with better quality. By plotting the within sum of squares in regard to different number of k, we find that the elbow of the plot is at k=4.

On the other hand, we also compute the average silhouette width based on the different numbers of clusters using `cluster.stats`. Also, we can use a line plot to plot the average silhouette width with regard to the different numbers of clusters. The preceding figure (step 4) shows the maximum average silhouette width appears at $k=4$. Lastly, we use `which.max` to obtain the value of k to determine the location of the maximum average silhouette width.

See also

- ▶ For those interested in how the within sum of squares is computed, please refer to the Wikipedia entry of **K-means clustering**: http://en.wikipedia.org/wiki/K-means_clustering

Clustering data with the density-based method

As an alternative to distance measurement, you can use a density-based measurement to cluster data. This method finds an area with a higher density than the remaining area. One of the most famous methods is DBSCAN. In the following recipe, we will demonstrate how to use DBSCAN to perform density-based clustering.

Getting ready

In this recipe, we will use simulated data generated from the `mlbench` package.

How to do it...

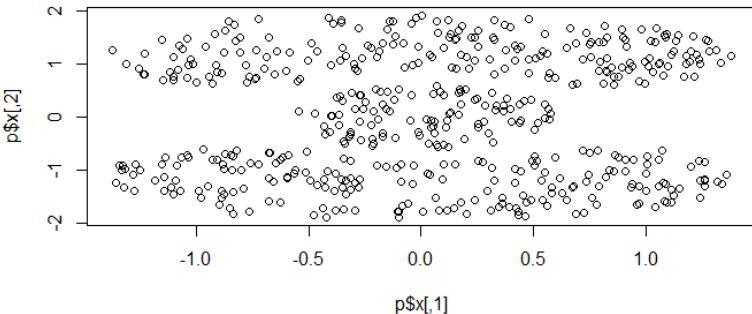
Perform the following steps to perform density-based clustering:

1. First, install and load the `fpc` and `mlbench` packages:

```
> install.packages("mlbench")
> library(mlbench)
> install.packages("fpc")
> library(fpc)
```

2. You can then use the `mlbench` library to draw a Cassini problem graph:

```
> set.seed(2)
> p = mlbench.cassini(500)
> plot(p$x)
```



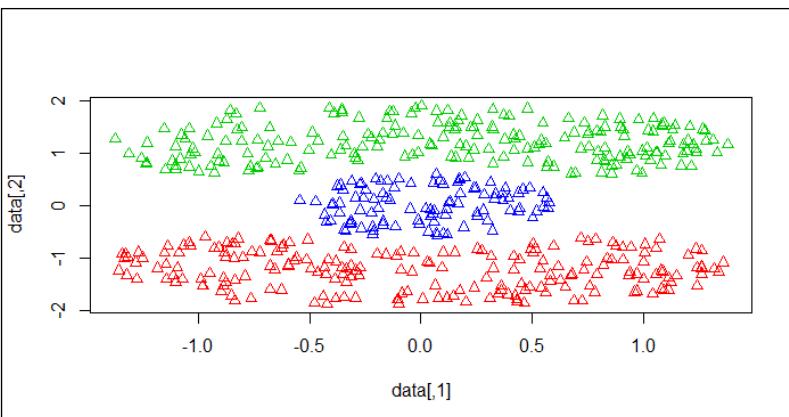
The Cassini problem graph

3. Next, you can cluster data with regard to its density measurement:

```
> ds = dbscan(dist(p$x), 0.2, 2, countmode=NULL, method="dist")
> ds
dbscan Pts=500 MinPts=2 eps=0.2
      1   2   3
seed  200 200 100
total 200 200 100
```

4. Plot the data in a scatter plot with different cluster labels as the color:

```
> plot(ds, p$x)
```



The data scatter plot colored with regard to the cluster label

5. You can also use `dbSCAN` to predict which cluster the data point belongs to. In this example, first make three inputs in the matrix `p`:

```
> y = matrix(0,nrow=3,ncol=2)
> y[1,] = c(0,0)
> y[2,] = c(0,-1.5)
> y[3,] = c(1,1)
> y
     [,1]  [,2]
[1,]    0   0.0
[2,]    0  -1.5
[3,]    1   1.0
```

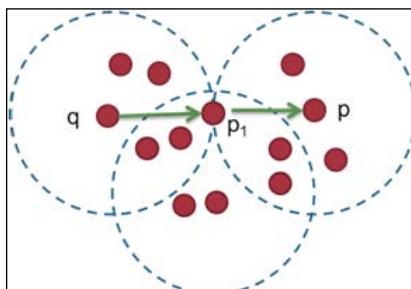
6. You can then predict which cluster the data belongs to:

```
> predict(ds, p$x, y)
[1] 3 1 2
```

How it works...

Density-based clustering uses the idea of density reachability and density connectivity, which makes it very useful in discovering a cluster in nonlinear shapes. Before discussing the process of density-based clustering, some important background concepts must be explained. Density-based clustering takes two parameters into account: `eps` and `MinPts`. `eps` stands for the maximum radius of the neighborhood; `MinPts` denotes the minimum number of points within the `eps` neighborhood. With these two parameters, we can define the core point as having points more than `MinPts` within `eps`. Also, we can define the border point as having points less than `MinPts`, but is in the neighborhood of the core points. Then, we can define the core object as if the number of points in the `eps`-neighborhood of `p` is more than `MinPts`.

Furthermore, we have to define the reachability between two points. We can say that a point, `p`, is directly density reachable from another point, `q`, if `q` is within the `eps`-neighborhood of `p` and `p` is a core object. Then, we can define that a point, `p`, is generic and density reachable from the point `q`, if there exists a chain of points, p_1, p_2, \dots, p_n , where $p_1 = q$, $p_n = p$, and p_{i+1} is directly density reachable from p_i with regard to `Eps` and `MinPts` for $1 \leq i \leq n$:



With a preliminary concept of density-based clustering, we can then illustrate the process of DBSCAN, the most popular density-based clustering, as shown in these steps:

1. Randomly select a point, p .
2. Retrieve all the points that are density-reachable from p with regard to Eps and MinPts .
3. If p is a core point, then a cluster is formed. Otherwise, if it is a border point and no points are density reachable from p , the process will mark the point as noise and continue visiting the next point.
4. Repeat the process until all points have been visited.

In this recipe, we demonstrate how to use the DBSCAN density-based method to cluster customer data. First, we have to install and load the `mlbench` and `fpc` libraries. The `mlbench` package provides many methods to generate simulated data with different shapes and sizes. In this example, we generate a Cassini problem graph.

Next, we perform `dbscan` on a Cassini dataset to cluster the data. We specify the reachability distance as 0.2, the minimum reachability number of points to 2, the progress reporting as null, and use distance as a measurement. The clustering method successfully clusters data into three clusters with sizes of 200, 200, and 100. By plotting the points and cluster labels on the plot, we see that three sections of the Cassini graph are separated in different colors.

The `fpc` package also provides a `predict` function, and you can use this to predict the cluster labels of the input matrix. Point $c(0,0)$ is classified into cluster 3, point $c(0, -1.5)$ is classified into cluster 1, and point $c(1,1)$ is classified into cluster 2.

See also

- ▶ The `fpc` package contains flexible procedures of clustering, and has useful clustering analysis functions. For example, you can generate a discriminant projection plot using the `plotcluster` function. For more information, please refer to the following document:
`> help(plotcluster)`

Clustering data with the model-based method

In contrast to hierarchical clustering and k-means clustering, which use a heuristic approach and do not depend on a formal model. Model-based clustering techniques assume varieties of data models and apply an EM algorithm to obtain the most likely model, and further use the model to infer the most likely number of clusters. In this recipe, we will demonstrate how to use the model-based method to determine the most likely number of clusters.

Getting ready

In order to perform a model-based method to cluster customer data, you need to have the previous recipe completed by generating the customer dataset.

How to do it...

Perform the following steps to perform model-based clustering:

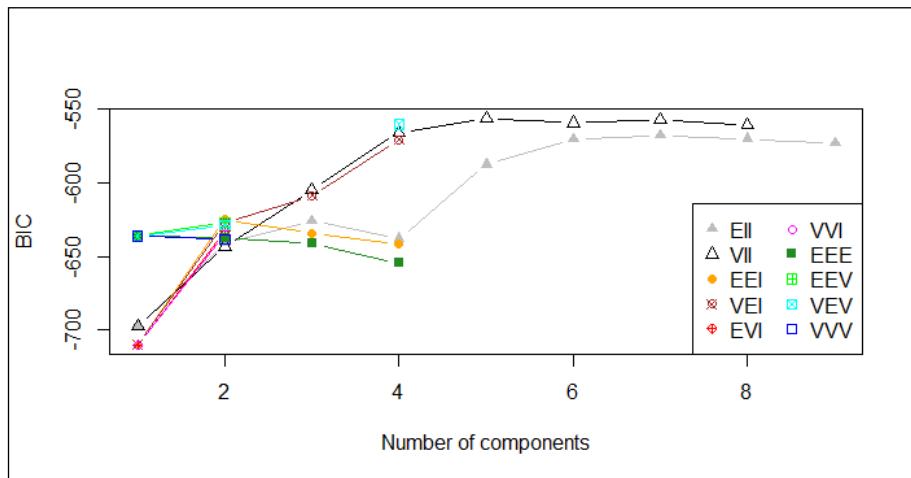
1. First, please install and load the library `mclust`:

```
> install.packages("mclust")
> library(mclust)
```

2. You can then perform model-based clustering on the `customer` dataset:

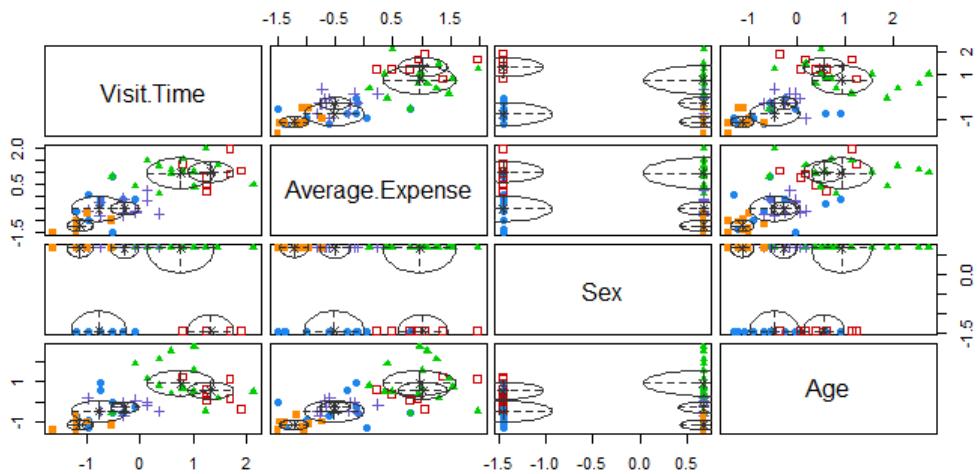
```
> mb = Mclust(customer)
> plot(mb)
```

3. Then, you can press 1 to obtain the BIC against a number of components:



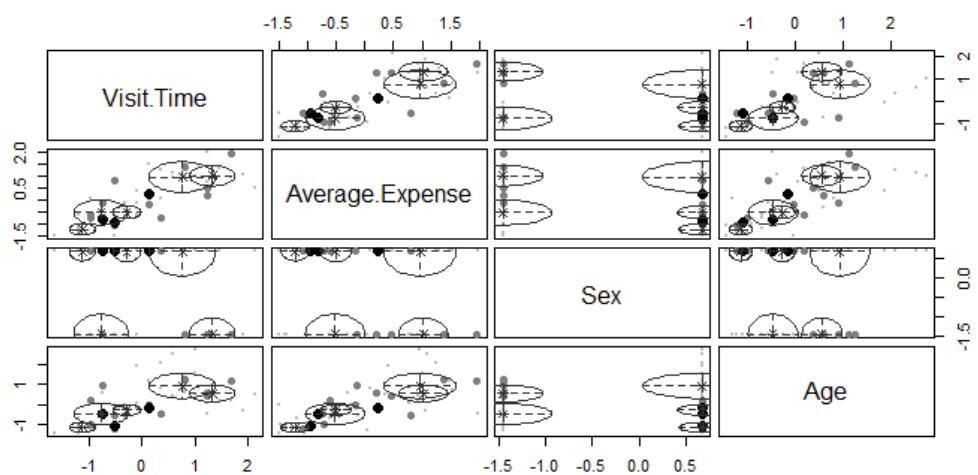
Plot of BIC against number of components

4. Then, you can press 2 to show the classification with regard to different combinations of features:



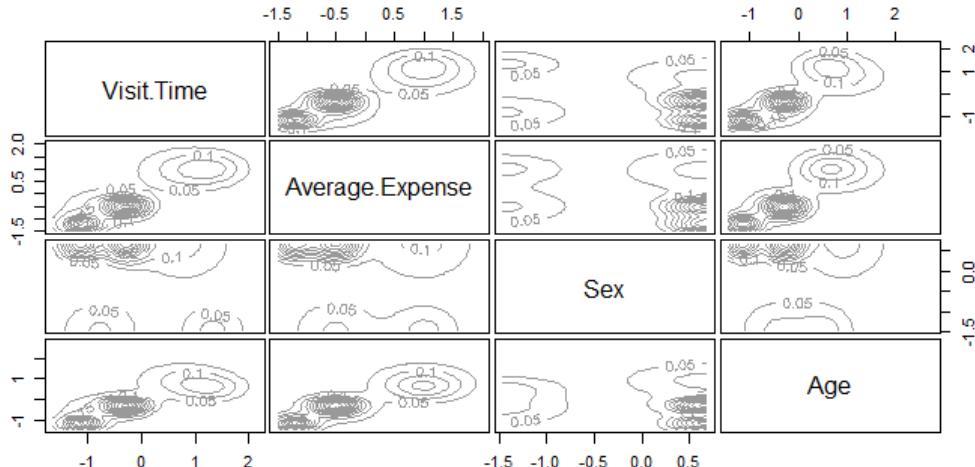
Plot showing classification with regard to different combinations of features

5. Press 3 to show the classification uncertainty with regard to different combinations of features:



Plot showing classification uncertainty with regard to different combinations of features

6. Next, press 4 to plot the density estimation:



A plot of density estimation

7. Then, you can press 0 to plot density to exit the plotting menu.
8. Lastly, use the `summary` function to obtain the most likely model and number of clusters:

```
> summary(mb)
```

```
-----  
Gaussian finite mixture model fitted by EM algorithm  
-----
```

```
Mclust VII (spherical, varying volume) model with 5 components:
```

```
log.likelihood  n df          BIC          ICL  
-218.6891  60 29 -556.1142 -557.2812
```

```
Clustering table:
```

	1	2	3	4	5
11	8	17	14	10	

How it works...

Instead of taking a heuristic approach to build a cluster, model-based clustering uses a probability-based approach. Model-based clustering assumes that the data is generated by an underlying probability distribution and tries to recover the distribution from the data. One common model-based approach is using finite mixture models, which provide a flexible modeling framework for the analysis of the probability distribution. Finite mixture models are a linearly weighted sum of component probability distribution. Assume the data $y=(y_1, y_2, \dots, y_n)$ contains n independent and multivariable observations; G is the number of components; the likelihood of finite mixture models can be formulated as:

$$L_{MIX}(\theta_1, \dots, \theta_G | y) = \prod_{i=1}^n \prod_{k=1}^G \tau_k f_k(y_i | \theta_k)$$

Where f_k and θ_k are the density and parameters of the k th component in the mixture, and τ_k ($\tau_k \geq 0$ and $\sum_{k=1}^G \tau_k = 1$) is the probability that an observation belongs to the k th component.

The process of model-based clustering has several steps: First, the process selects the number and types of component probability distribution. Then, it fits a finite mixture model and calculates the posterior probabilities of a component membership. Lastly, it assigns the membership of each observation to the component with the maximum probability.

In this recipe, we demonstrate how to use model-based clustering to cluster data. We first install and load the `Mclust` library into R. We then fit the customer data into the model-based method by using the `Mclust` function.

After the data is fit into the model, we plot the model based on clustering results. There are four different plots: BIC, classification, uncertainty, and density plots. The BIC plot shows the BIC value, and one can use this value to choose the number of clusters. The classification plot shows how data is clustered in regard to different dimension combinations. The uncertainty plot shows the uncertainty of classifications in regard to different dimension combinations. The density plot shows the density estimation in contour.

You can also use the `summary` function to obtain the most likely model and the most possible number of clusters. For this example, the most possible number of clusters is five, with a BIC value equal to -556.1142.

See also

- For those interested in detail on how `Mclust` works, please refer to the following source: C. Fraley, A. E. Raftery, T. B. Murphy and L. Scrucca (2012). *mclust Version 4 for R: Normal Mixture Modeling for Model-Based Clustering, Classification, and Density Estimation*. Technical Report No. 597, Department of Statistics, University of Washington.

Visualizing a dissimilarity matrix

A dissimilarity matrix can be used as a measurement for the quality of a cluster. To visualize the matrix, we can use a heat map on a distance matrix. Within the plot, entries with low dissimilarity (or high similarity) are plotted darker, which is helpful to identify hidden structures in the data. In this recipe, we will discuss some techniques that are useful to visualize a dissimilarity matrix.

Getting ready

In order to visualize the dissimilarity matrix, you need to have the previous recipe completed by generating the customer dataset. In addition to this, a k-means object needs to be generated and stored in the variable `km`.

How to do it...

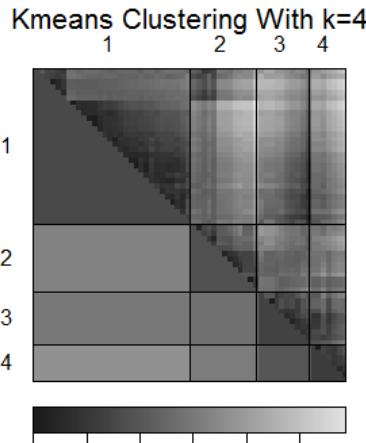
Perform the following steps to visualize the dissimilarity matrix:

1. First, install and load the `seriation` package:

```
> install.packages("seriation")
> library(seriation)
```

2. You can then use `dissplot` to visualize the dissimilarity matrix in a heat map:

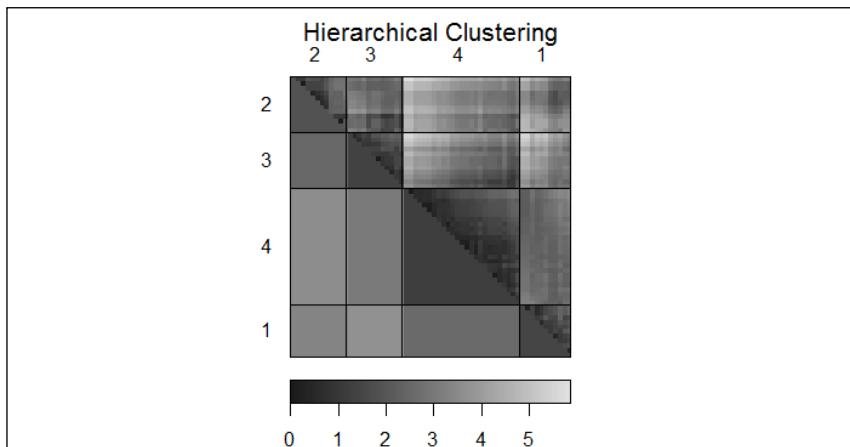
```
> dissplot(dist(customer), labels=km$cluster,
  options=list(main="Kmeans Clustering With k=4"))
```



A dissimilarity plot of k-means clustering

3. Next, apply `dissplot` on hierarchical clustering in the heat map:

```
> complete_c = hclust(dist(customer), method="complete")
> hc_complete = cutree(complete_c, k = 4)
> dissplot(dist(customer), labels=hc_complete,
options=list(main="Hierarchical Clustering"))
```



A dissimilarity plot of hierarchical clustering

How it works...

In this recipe, we use a dissimilarity plot to visualize the dissimilarity matrix. We first install and load the package `seriation`, and then apply the `dissplot` function on the k-means clustering output, generating the preceding figure (step 2).

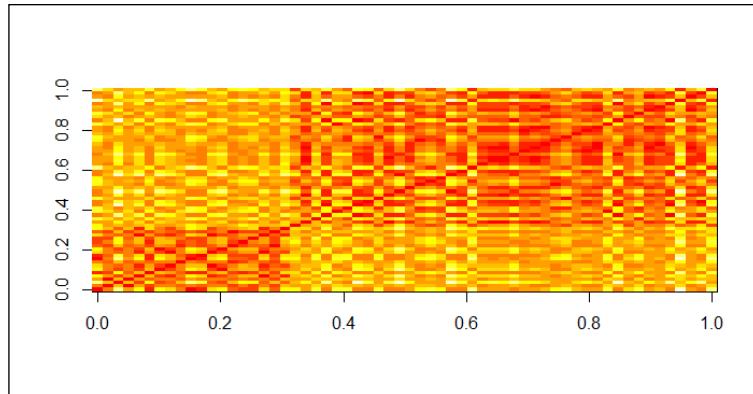
It shows that clusters similar to each other are plotted darker, and dissimilar combinations are plotted lighter. Therefore, we can see clusters against their corresponding clusters (such as cluster 4 to cluster 4) are plotted diagonally and darker. On the other hand, clusters dissimilar to each other are plotted lighter and away from the diagonal.

Likewise, we can apply the `dissplot` function on the output of hierarchical clustering. The generated plot in the figure (step 3) shows the similarity of each cluster in a single heat map.

There's more...

Besides using `dissplot` to visualize the dissimilarity matrix, one can also visualize a distance matrix by using the `dist` and `image` functions. In the resulting graph, closely related entries are plotted in red. Less related entries are plotted closer to white:

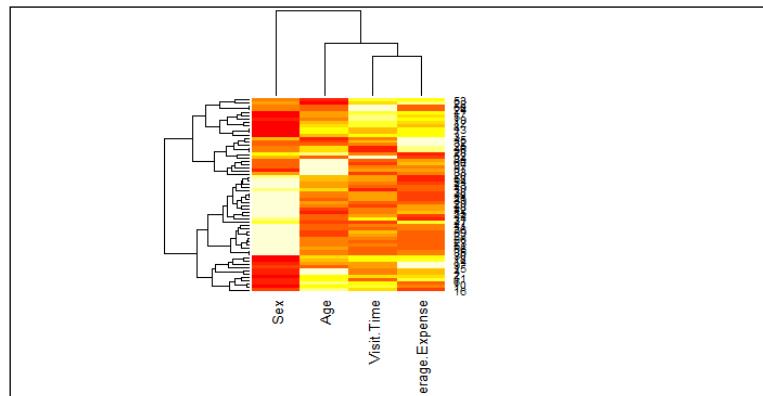
```
> image(as.matrix(dist(customer)))
```



A distance matrix plot of customer dataset

In order to plot both a dendrogram and heat map to show how data is clustered, you can use the `heatmap` function:

```
> cd=dist(customer)
> hc=hclust(cd)
> cdt=dist(t(customer))
> hcc=hclust(cdt)
> heatmap(customer, Rowv=as.dendrogram(hc), Colv=as.dendrogram(hcc))
```



A heat map with dendrogram on the column and row side

Validating clusters externally

Besides generating statistics to validate the quality of the generated clusters, you can use known data clusters as the ground truth to compare different clustering methods. In this recipe, we will demonstrate how clustering methods differ with regard to data with known clusters.

Getting ready

In this recipe, we will continue to use handwriting digits as clustering inputs; you can find the figure on the author's Github page: https://github.com/ywchiu/ml_R_cookbook/tree/master/CH9.

How to do it...

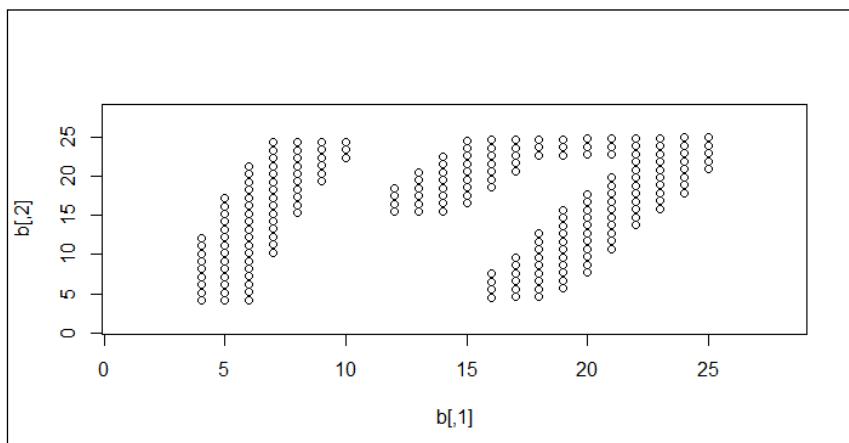
Perform the following steps to cluster digits with different clustering techniques:

1. First, you need to install and load the package `png`:

```
> install.packages("png")
> library(png)
```

2. Then, please read images from `handwriting.png` and transform the read data into a scatter plot:

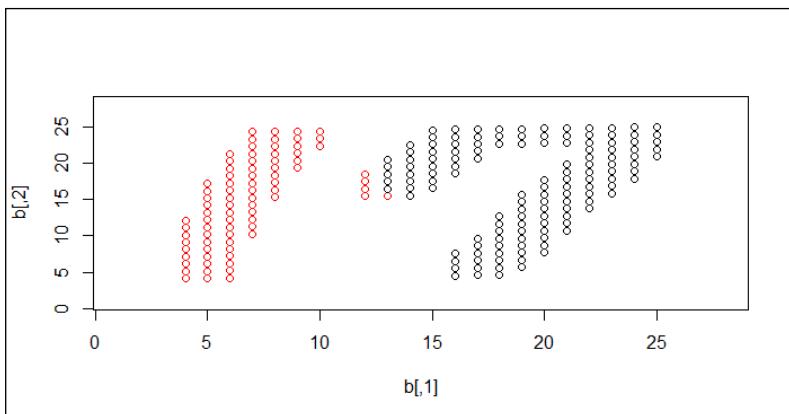
```
> img2 = readPNG("handwriting.png", TRUE)
> img3 = img2[,nrow(img2):1]
> b = cbind(as.integer(which(img3 < -1) %% 28), which(img3 < -1) / 28)
> plot(b, xlim=c(1,28), ylim=c(1,28))
```



A scatter plot of handwriting digits

3. Perform a k-means clustering method on the handwriting digits:

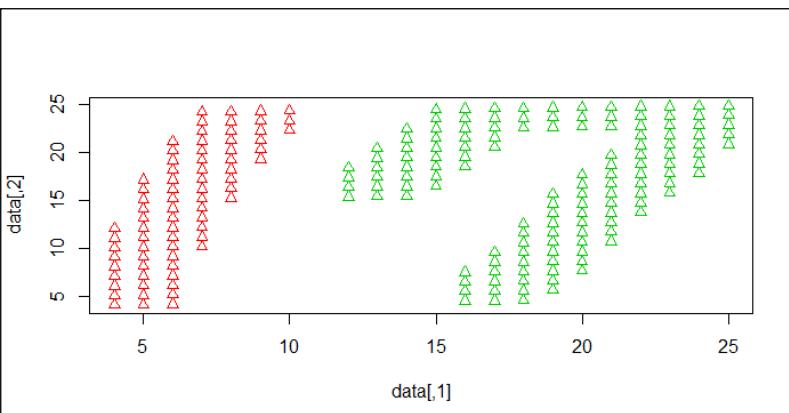
```
> set.seed(18)
> fit = kmeans(b, 2)
> plot(b, col=fit$cluster)
> plot(b, col=fit$cluster, xlim=c(1,28), ylim=c(1,28))
```



k-means clustering result on handwriting digits

4. Next, perform the dbscan clustering method on the handwriting digits:

```
> ds = dbscan(b, 2)
> ds
dbscan Pts=212 MinPts=5 eps=2
      1   2
seed  75 137
total 75 137
> plot(ds, b, xlim=c(1,28), ylim=c(1,28))
```



DBSCAN clustering result on handwriting digits

How it works...

In this recipe, we demonstrate how different clustering methods work in regard to a handwriting dataset. The aim of the clustering is to separate 1 and 7 into different clusters. We perform different techniques to see how data is clustered in regard to the k-means and DBSCAN methods.

To generate the data, we use the Windows application `paint.exe` to create a PNG file with dimensions of 28 x 28 pixels. We then read the PNG data using the `readPNG` function and transform the read PNG data points into a scatter plot, which shows the handwriting digits in 17.

After the data is read, we perform clustering techniques on the handwriting digits. First, we perform k-means clustering, where $k=2$ on the dataset. Since k-means clustering employs distance measures, the constructed clusters cover the area of both the 1 and 7 digits. We then perform DBSCAN on the dataset. As DBSCAN is a density-based clustering technique, it successfully separates digit 1 and digit 7 into different clusters.

See also

- ▶ If you are interested in how to read various graphic formats in R, you may refer to the following document:
`> help(package="png")`

9

Association Analysis and Sequence Mining

In this chapter, we will cover the following topics:

- ▶ Transforming data into transactions
- ▶ Displaying transactions and associations
- ▶ Mining associations with the Apriori rule
- ▶ Pruning redundant rules
- ▶ Visualizing associations rules
- ▶ Mining frequent itemsets with Eclat
- ▶ Creating transactions with temporal information
- ▶ Mining frequent sequential patterns with cSPADE

Introduction

Enterprises accumulate a large amount of transaction data (for example, sales orders from retailers, invoices, and shipping documentations) from daily operations. Finding hidden relationships in the data can be useful, such as, "What products are often bought together?" or "What are the subsequent purchases after buying a cell phone?" To answer these two questions, we need to perform association analysis and frequent sequential pattern mining on a transaction dataset.

Association analysis is an approach to find interesting relationships within a transaction dataset. A famous association between products is that *customers who buy diapers also buy beer*. While this association may sound unusual, if retailers can use this kind of information or rule to cross-sell products to their customers, there is a high likelihood that they can increase their sales.

Association analysis is used to find a correlation between **itemsets**, but what if you want to find out the order in which items are frequently purchased? To achieve this, you can adopt frequent sequential pattern mining to find frequent subsequences from transaction datasets with temporal information. You can then use the mined frequent subsequences to predict customer shopping sequence orders, web click streams, biological sequences, and usages in other applications.

In this chapter, we will cover recipes to create and inspect transaction datasets, performing association analysis with an Apriori algorithm, visualizing associations in various graph formats, and finding frequent itemsets using the Eclat algorithm. Lastly, we will create transactions with temporal information and use the cSPADE algorithm to discover frequent sequential patterns.

Transforming data into transactions

Before creating a mining association rule, you need to transform the data into transactions. In the following recipe, we will introduce how to transform either a list, matrix, or data frame into transactions.

Getting ready

In this recipe, we will generate three different datasets in a list, matrix, or data frame. We can then transform the generated dataset into transactions.

How to do it...

Perform the following steps to transform different formats of data into transactions:

1. First, you have to install and load the package arule:

```
> install.packages("arules")
> library(arules)
```

2. You can then make a list with three vectors containing purchase records:

```
> tr_list = list(c("Apple", "Bread", "Cake"),
+                  c("Apple", "Bread", "Milk"),
+                  c("Bread", "Cake", "Milk"))
> names(tr_list) = paste("Tr",c(1:3), sep = "")
```

3. Next, you can use the as function to transform the data frame into transactions:

```
> trans = as(tr_list, "transactions")
> trans
transactions in sparse format with
3 transactions (rows) and
4 items (columns)
```

4. You can also transform the matrix format data into transactions:

```
> tr_matrix = matrix(  
+   c(1,1,1,0,  
+     1,1,0,1,  
+     0,1,1,1), ncol = 4)  
> dimnames(tr_matrix) = list(  
+   paste("Tr",c(1:3), sep = ""),  
+   c("Apple","Bread","Cake", "Milk")  
+ )  
> trans2 = as(tr_matrix, "transactions")  
> trans2  
transactions in sparse format with  
3 transactions (rows) and  
4 items (columns)
```

5. Lastly, you can transform the data frame format datasets into transactions:

```
> Tr_df = data.frame(  
+   TrID= as.factor(c(1,2,1,1,2,3,2,3,2,3)),  
+   Item = as.factor(c("Apple","Milk","Cake","Bread",  
+                     "Cake","Milk","Apple","Cake",  
+                     "Bread","Bread"))  
+ )  
> trans3 = as(split(Tr_df[, "Item"], Tr_df[, "TrID"]),  
"transactions")  
> trans3  
transactions in sparse format with  
3 transactions (rows) and  
4 items (columns)
```

How it works...

Before mining frequent itemsets or using the association rule, it is important to prepare the dataset by the class of transactions. In this recipe, we demonstrate how to transform a dataset from a list, matrix, and data frame format to transactions. In the first step, we generate the dataset in a list format containing three vectors of purchase records. Then, after we have assigned a transaction ID to each transaction, we transform the data into transactions using the `as` function.

Next, we demonstrate how to transform the data from the matrix format into transactions. To denote how items are purchased, one should use a binary incidence matrix to record the purchase behavior of each transaction with regard to different items purchased. Likewise, we can use an `as` function to transform the dataset from the matrix format into transactions.

Lastly, we illustrate how to transform the dataset from the data frame format into transactions. The data frame contains two factor-type vectors: one is a transaction ID named `TrID`, while the other shows purchased items (named in `Item`) with regard to different transactions. Also, one can use the `as` function to transform the data frame format data into transactions.

See also

- ▶ The `transactions` class is used to represent transaction data for rules or frequent pattern mining. It is an extension of the `itemMatrix` class. If you are interested in how to use the two different classes to represent transaction data, please use the `help` function to refer to the following documents:

```
> help(transactions)
> help(itemMatrix)
```

Displaying transactions and associations

The `arule` package uses its own `transactions` class to store transaction data. As such, we must use the generic function provided by `arule` to display transactions and association rules. In this recipe, we will illustrate how to display transactions and association rules via various functions in the `arule` package.

Getting ready

Ensure that you have completed the previous recipe by generating transactions and storing these in the variable, `trans`.

How to do it...

Perform the following steps to display transactions and associations:

1. First, you can obtain a LIST representation of the transaction data:

```
> LIST(trans)
$Tr1
[1] "Apple" "Bread" "Cake"

$Tr2
[1] "Apple" "Bread" "Milk"

$Tr3
[1] "Bread" "Cake"   "Milk"
```

2. Next, you can use the `summary` function to show a summary of the statistics and details of the transactions:

```
> summary(trans)
transactions as itemMatrix in sparse format with
 3 rows (elements/itemsets/transactions) and
 4 columns (items) and a density of 0.75

most frequent items:
  Bread   Apple    Cake    Milk (Other)
      3        2        2        2        0

element (itemset/transaction) length distribution:
sizes
3
3

Min. 1st Qu. Median     Mean 3rd Qu.     Max.
3       3       3       3       3       3
```

includes extended item information - examples:

```
labels
1 Apple
2 Bread
3 Cake
```

includes extended transaction information - examples:

```
transactionID
1           Tr1
2           Tr2
3           Tr3
```

3. You can then display transactions using the `inspect` function:

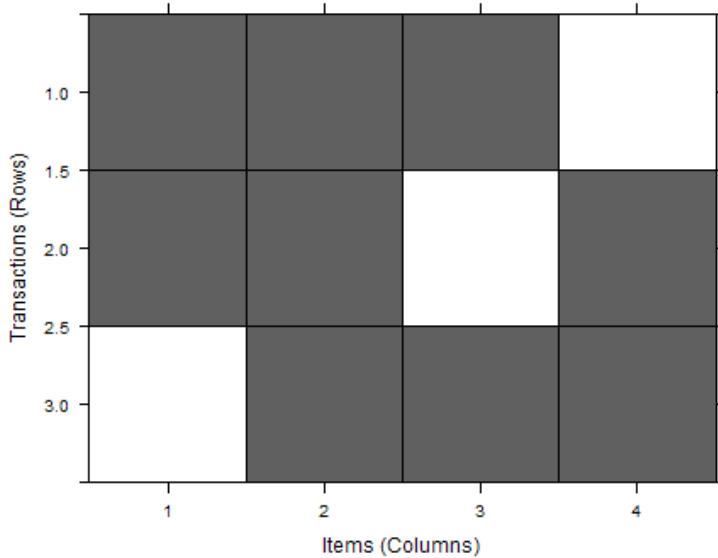
```
> inspect(trans)
  items  transactionID
1 {Apple,
  Bread,
  Cake}          Tr1
2 {Apple,
  Bread,
  Milk}          Tr2
3 {Bread,
  Cake,
  Milk}          Tr3
```

4. In addition to this, you can filter the transactions by size:

```
> filter_trains = trans[size(trans) >=3]
> inspect(filter_trains)
  items    transactionID
1 {Apple,
  Bread,
  Cake}           Tr1
2 {Apple,
  Bread,
  Milk}           Tr2
3 {Bread,
  Cake,
  Milk}           Tr3
```

5. Also, you can use the image function to visually inspect the transactions:

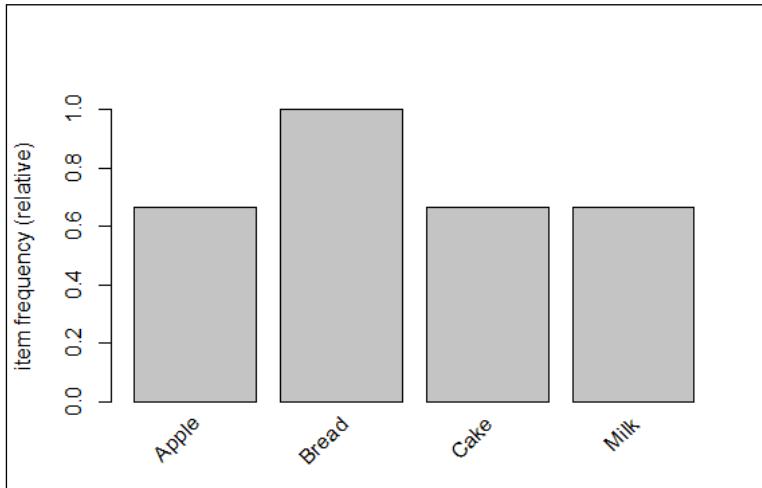
```
> image(trans)
```



Visual inspection of transactions

6. To visually show the frequency/support bar plot, one can use itemFrequencyPlot:

```
> itemFrequencyPlot (trans)
```



Item frequency bar plot of transactions

How it works...

As the transaction data is the base for mining associations and frequent patterns, we have to learn how to display the associations to gain insights and determine how associations are built. The `arules` package provides various methods to inspect transactions. First, we use the `LIST` function to obtain the list representation of the transaction data. We can then use the `summary` function to obtain information, such as basic descriptions, most frequent items, and the transaction length distribution.

Next, we use the `inspect` function to display the transactions. Besides displaying all transactions, one can first filter the transactions by size and then display the associations by using the `inspect` function. Furthermore, we can use the `image` function to visually inspect the transactions. Finally, we illustrate how to use the frequency/support bar plot to display the relative item frequency of each item.

See also

- ▶ Besides using `itemFrequencyPlot` to show the frequency/bar plot, you can use the `itemFrequency` function to show the support distribution. For more details, please use the `help` function to view the following document:
 > `help(itemFrequency)`

Mining associations with the Apriori rule

Association mining is a technique that can discover interesting relationships hidden in transaction datasets. This approach first finds all frequent itemsets, and generates strong association rules from frequent itemsets. Apriori is the most well-known association mining algorithm, which identifies frequent individual items first and then performs a breadth-first search strategy to extend individual items to larger itemsets until larger frequent itemsets cannot be found. In this recipe, we will introduce how to perform association analysis using the Apriori rule.

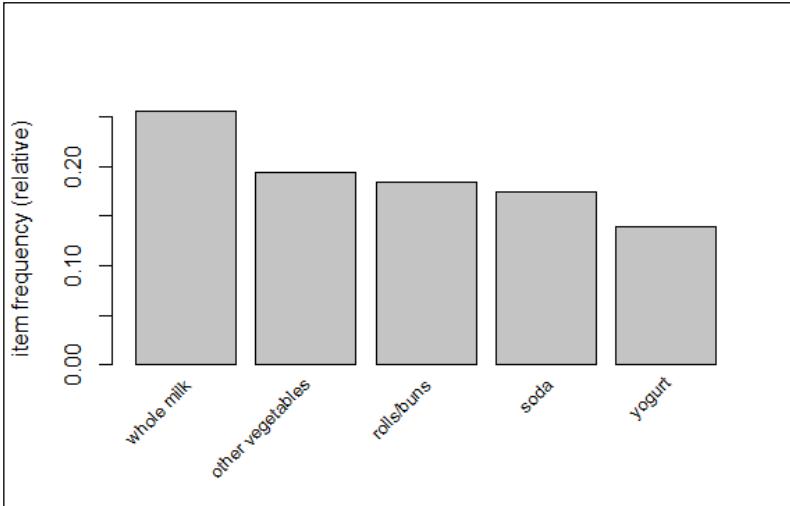
Getting ready

In this recipe, we will use the built-in transaction dataset, `Groceries`, to demonstrate how to perform association analysis with the Apriori algorithm in the `arules` package. Please make sure that the `arules` package is installed and loaded first.

How to do it...

Perform the following steps to analyze the association rules:

1. First, you need to load the dataset `Groceries`:
`> data(Groceries)`
2. You can then examine the summary of the `Groceries` dataset:
`> summary(Groceries)`
3. Next, you can use `itemFrequencyPlot` to examine the relative item frequency of itemsets:
`> itemFrequencyPlot(Groceries, support = 0.1, cex.names=0.8,
topN=5)`



The top five item frequency bar plot of groceries transactions

4. Use *apriori* to discover rules with the support over 0.001 and confidence over 0.5:

```
> rules = apriori(Groceries, parameter = list(supp = 0.001, conf = 0.5, target= "rules"))
> summary(rules)
set of 5668 rules
```

```
rule length distribution (lhs + rhs):sizes
 2      3      4      5      6
11 1461 3211   939    46
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
2.00	3.00	4.00	3.92	4.00	6.00

summary of quality measures:

support	confidence	lift
Min. :0.001017	Min. :0.5000	Min. : 1.957
1st Qu.:0.001118	1st Qu.:0.5455	1st Qu.: 2.464
Median :0.001322	Median :0.6000	Median : 2.899
Mean :0.001668	Mean :0.6250	Mean : 3.262
3rd Qu.:0.001729	3rd Qu.:0.6842	3rd Qu.: 3.691
Max. :0.022267	Max. :1.0000	Max. :18.996

mining info:

	ntransactions	support	confidence
Groceries	9835	0.001	0.5

5. We can then inspect the first few rules:

```
> inspect(head(rules))
   lhs                      rhs          support confidence
lift
1 {honey}           => {whole milk} 0.001118454  0.7333333
2.870009
2 {tidbits}        => {rolls/buns} 0.001220132  0.5217391
2.836542
3 {cocoa drinks}  => {whole milk} 0.001321810  0.5909091
2.312611
4 {pudding powder} => {whole milk} 0.001321810  0.5652174
2.212062
5 {cooking chocolate} => {whole milk} 0.001321810  0.5200000
2.035097
6 {cereals}         => {whole milk} 0.003660397  0.6428571
2.515917
```

6. You can sort rules by confidence and inspect the first few rules:

```
> rules=sort(rules, by="confidence", decreasing=TRUE)
> inspect(head(rules))
   lhs                      rhs          support
confidence      lift
1 {rice,
  sugar}           => {whole milk} 0.001220132
1 3.913649
2 {canned fish,
  hygiene articles} => {whole milk} 0.001118454
1 3.913649
3 {root vegetables,
  butter,
  rice}            => {whole milk} 0.001016777
1 3.913649
4 {root vegetables,
  whipped/sour cream,
  flour}           => {whole milk} 0.001728521
1 3.913649
5 {butter,
  soft cheese,
  domestic eggs}  => {whole milk} 0.001016777
1 3.913649
6 {citrus fruit,
  root vegetables,
  soft cheese}    => {other vegetables} 0.001016777
1 5.168156
```

How it works...

The purpose of association mining is to discover associations among items from the transactional database. Typically, the process of association mining proceeds by finding itemsets that have the support greater than the minimum support. Next, the process uses the frequent itemsets to generate strong rules (for example, milk => bread; a customer who buys milk is likely to buy bread) that have the confidence greater than minimum the confidence. By definition, an association rule can be expressed in the form of $X \Rightarrow Y$, where X and Y are disjointed itemsets. We can measure the strength of associations between two terms: support and confidence. Support shows how much of the percentage of a rule is applicable within a dataset, while confidence indicates the probability of both X and Y appearing in the same transaction:

- ▶ $\text{Support} = \frac{\sigma(x \cup y)}{N}$
- ▶ $\text{Confidence} = \frac{\sigma(x \cup y)}{\sigma(x)}$

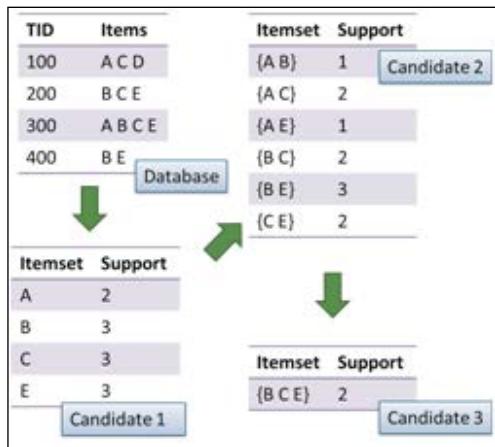
Here, σ refers to the frequency of a particular itemset; N denotes the populations.

As support and confidence are metrics for the strength rule only, you might still obtain many redundant rules with a high support and confidence. Therefore, we can use the third measure, lift, to evaluate the quality (ranking) of the rule. By definition, lift indicates the strength of a rule over the random co-occurrence of X and Y, so we can formulate lift in the following form:

$$\text{Lift} = \frac{\sigma(x \cup y)}{\sigma(x) \times \sigma(y)}$$

Apriori is the best known algorithm for mining associations, which performs a level-wise, breadth-first algorithm to count the candidate itemsets. The process of Apriori starts by finding frequent itemsets (a set of items that have minimum support) level-wisely. For example, the process starts with finding frequent 1-itemsets. Then, the process continues by using frequent 1-itemsets to find frequent 2-itemsets. The process iteratively discovers new frequent k+1-itemsets from frequent k-itemsets until no frequent itemsets are found.

Finally, the process utilizes frequent itemsets to generate association rules:



An illustration of Apriori algorithm (Where support = 2)

In this recipe, we use the Apriori algorithm to find association rules within transactions. We use the built-in Groceries dataset, which contains one month of real-world point-of-sale transaction data from a typical grocery outlet. We then use the `summary` function to obtain the summary statistics of the Groceries dataset. The summary statistics shows that the dataset contains 9,835 transactions, which are categorized into 169 categories. In addition to this, the summary shows information, such as most frequent items, itemset distribution, and example extended item information within the dataset. We can then use `itemFrequencyPlot` to visualize the five most frequent items with support over 0.1.

Next, we apply the Apriori algorithm to search for rules with support over 0.001 and confidence over 0.5. We then use the `summary` function to inspect detailed information on the generated rules. From the output summary, we find the Apriori algorithm generates 5,668 rules with support over 0.001 and confidence over 0.5. Further, we can find the rule length distribution, summary of quality measures, and mining information. In the summary of the quality measurement, we find descriptive statistics of three measurements, which are support, confidence, and lift. Support is the proportion of transactions containing a certain itemset. Confidence is the correctness percentage of the rule. Lift is the response target association rule divided by the average response.

To explore some generated rules, we can use the `inspect` function to view the first six rules of the 5,668 generated rules. Lastly, we can sort rules by confidence and list rules with the most confidence. Therefore, we find that `rich sugar` associated to `whole milk` is the most confident rule with the support equal to 0.001220132, confidence equal to 1, and lift equal to 3.913649.

See also

For those interested in the research results using the Groceries dataset, and how the support, confidence, and lift measurement are defined, you can refer to the following papers:

- ▶ Michael Hahsler, Kurt Hornik, and Thomas Reutterer (2006) *Implications of probabilistic data modeling for mining association rules*. In M. Spiliopoulou, R. Kruse, C. Borgelt, A. Nuernberger, and W. Gaul, editors, *From Data and Information Analysis to Knowledge Engineering, Studies in Classification, Data Analysis, and Knowledge Organization*, pages 598–605. Springer-Verlag

Also, in addition to using the `summary` and `inspect` functions to inspect association rules, you can use `interestMeasure` to obtain additional interest measures:

```
> head(interestMeasure(rules, c("support", "chiSquare", "confidence",  
"conviction", "cosine", "coverage", "leverage", "lift", "oddsRatio"),  
Groceries))
```

Pruning redundant rules

Among the generated rules, we sometimes find repeated or redundant rules (for example, one rule is the super rule or subset of another rule). In this recipe, we will show you how to prune (or remove) repeated or redundant rules.

Getting ready

In this recipe, you have to complete the previous recipe by generating rules and have it stored in the variable `rules`.

How to do it...

Perform the following steps to prune redundant rules:

1. First, follow these steps to find redundant rules:

```
> rules.sorted = sort(rules, by="lift")  
> subset.matrix = is.subset(rules.sorted, rules.sorted)  
> subset.matrix[lower.tri(subset.matrix, diag=T)] = NA  
> redundant = colSums(subset.matrix, na.rm=T) >= 1
```

2. You can then remove redundant rules:

```
> rules.pruned = rules.sorted[!redundant]  
> inspect(head(rules.pruned))
```

lhs	rhs	support
confidence lift		
1 {Instant food products, soda}	=> {hamburger meat} 0.001220132	
0.6315789 18.99565		
2 {soda, popcorn}	=> {salty snack} 0.001220132	
0.6315789 16.69779		
3 {flour, baking powder}	=> {sugar} 0.001016777	
0.5555556 16.40807		
4 {ham, processed cheese}	=> {white bread} 0.001931876	
0.6333333 15.04549		
5 {whole milk, Instant food products}	=> {hamburger meat} 0.001525165	
0.5000000 15.03823		
6 {other vegetables, curd, yogurt, whipped/sour cream}	=> {cream cheese } 0.001016777	
0.5882353 14.83409		

How it works...

The two main constraints of association mining are to choose between the support and confidence. For example, if you use a high support threshold, you might remove rare item rules without considering whether these rules have a high confidence value. On the other hand, if you choose to use a low support threshold, the association mining can produce huge sets of redundant association rules, which make these rules difficult to utilize and analyze. Therefore, we need to prune redundant rules so we can discover meaningful information from these generated rules.

In this recipe, we demonstrate how to prune redundant rules. First, we search for redundant rules. We sort the rules by a lift measure, and then find subsets of the sorted rules using the `is.subset` function, which will generate an `itemMatrix` object. We can then set the lower triangle of the matrix to NA. Lastly, we compute `colSums` of the generated matrix, of which `colSums >=1` indicates that the specific rule is redundant.

After we have found the redundant rules, we can prune these rules from the sorted rules. Lastly, we can examine the pruned rules using the `inspect` function.

See also

- In order to find subsets or supersets of rules, you can use the `is.superset` and `is.subset` functions on the association rules. These two methods may generate an `itemMatrix` object to show which rule is the superset or subset of other rules. You can refer to the `help` function for more information:

```
> help(is.superset)  
> help(is.subset)
```

Visualizing association rules

Besides listing rules as text, you can visualize association rules, making it easier to find the relationship between itemsets. In the following recipe, we will introduce how to use the `aruleViz` package to visualize the association rules.

Getting ready

In this recipe, we will continue using the `Groceries` dataset. You need to have completed the previous recipe by generating the pruned rule `rules.pruned`.

How to do it...

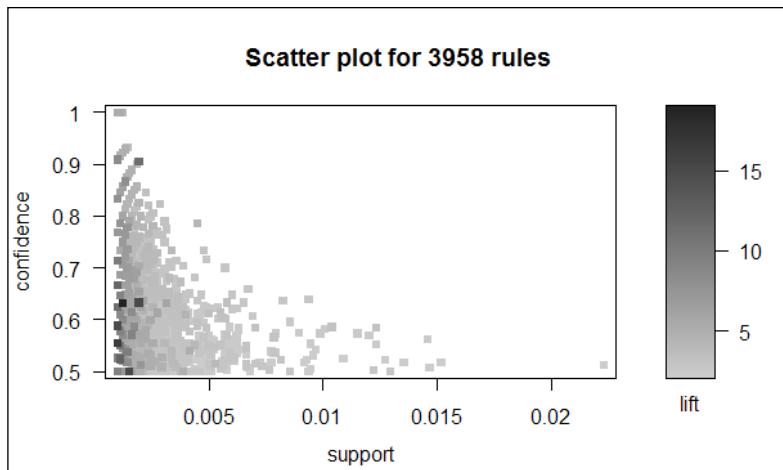
Perform the following steps to visualize the association rule:

- First, you need to install and load the package `arulesViz`:

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

2. You can then make a scatter plot from the pruned rules:

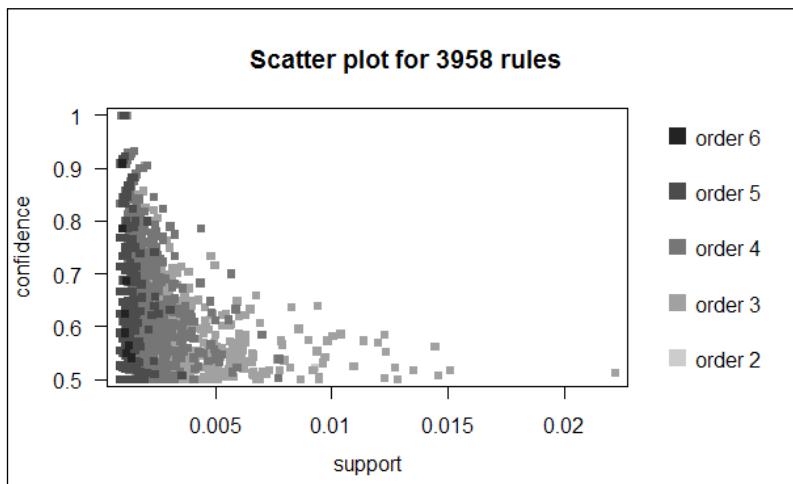
```
> plot(rules.pruned)
```



The scatter plot of pruned association rules

3. Additionally, to prevent overplotting, you can add jitter to the scatter plot:

```
> plot(rules.pruned, shading="order", control=list(jitter=6))
```



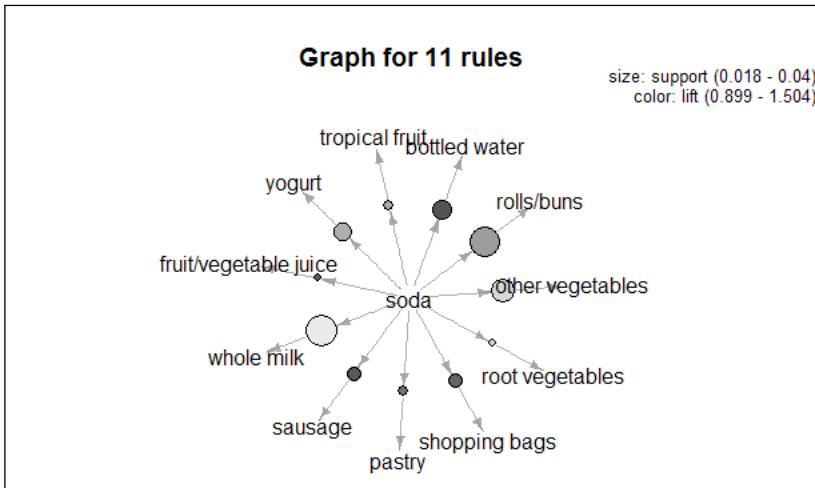
The scatter plot of pruned association rules with jitters

4. We then produce new rules with soda on the left-hand side using the Apriori algorithm:

```
> soda_rule=apriori(data=Groceries, parameter=list(supp=0.001,conf = 0.1, minlen=2), appearance = list(default="rhs",lhs="soda"))
```

5. Next, you can plot soda_rule in a graph plot:

```
> plot(sort(soda_rule, by="lift"), method="graph",
control=list(type="items"))
```

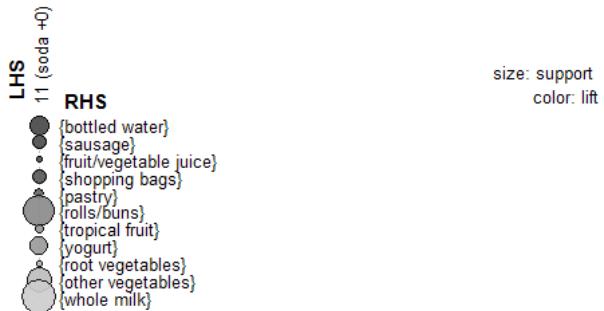


Graph plot of association rules

6. Also, the association rules can be visualized in a balloon plot:

```
> plot(soda_rule, method="grouped")
```

Grouped matrix for 11 rules



Balloon plot of association rules

How it works...

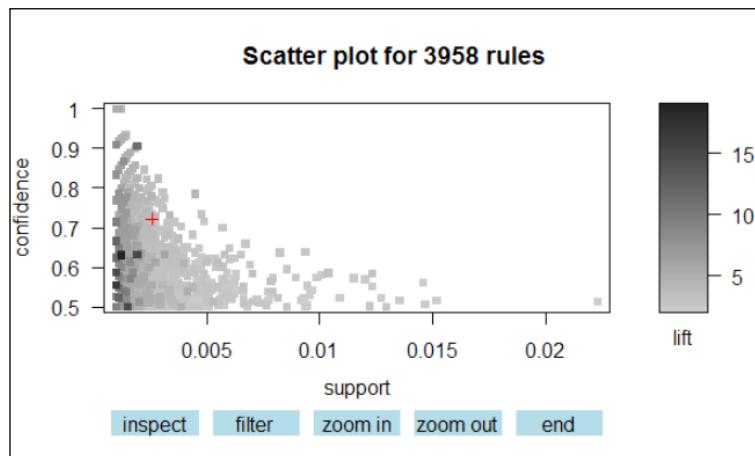
Besides presenting association rules as text, one can use `arulesViz` to visualize association rules. The `arulesViz` is an `arules` extension package, which provides many visualization techniques to explore association rules. To start using `arulesViz`, first install and load the package `arulesViz`. We then use the pruned rules generated in the previous recipe to make a scatter plot. As per the figure in step 2, we find the rules are shown as points within the scatter plot, with the x-axis in support and y-axis in confidence. The shade of color shows the lift of the rule; the darker the shade, the higher the lift. Next, in order to prevent overplotting points, we can include the jitter as an argument in the control list. The plot with the jitter added is provided in the figure in step 3.

In addition to plotting the rules in a scatter plot, `arulesViz` enables you to plot rules in a graph and grouped matrix. Instead of printing all the rules on a single plot, we choose to produce new rules with `soda` on the left-hand side. We then sort the rules by using the lift and visualize the rules in the graph in the figure in step 4. From the graph, every itemset is presented in a vertex and their relationship is presented in an edge. The figure (step 4) shows it is clear that the rule with `soda` on the left-handside to `whole milk` on the right-handside has the maximum support, for the size of the node is greatest. Also, the rule shows that `soda` on the left-hand side to `bottled water` on the right-hand side has the maximum lift as the shade of color in the circle is the darkest. We can then use the same data with `soda` on the left-handside to generate a grouped matrix, which is a balloon plot shown in the figure in step 5, with the left-handside rule as column labels and the right-handside as row labels. Similar to the graph plot in the figure in step 4, the size of the balloon in the figure in step 5 shows the support of the rule, and the color of the balloon shows the lift of the rule.

See also

- ▶ In this recipe, we introduced three visualization methods to plot association rules. However, `arulesViz` also provides features to plot parallel coordinate plots, double-decker plots, mosaic plots, and other related charts. For those who are interested in how these plots work, you may refer to: Hahsler, M., and Chelluboina, S. (2011). *Visualizing association rules: Introduction to the R-extension package arulesViz*. R project module.
- ▶ In addition to generating a static plot, you can generate an interactive plot by setting `interactive` equal to `TRUE` through the following steps:

```
> plot(rules.pruned, interactive=TRUE)
```



The interactive scatter plots

Mining frequent itemsets with Eclat

In addition to the Apriori algorithm, you can use the Eclat algorithm to generate frequent itemsets. As the Apriori algorithm performs a breadth-first search to scan the complete database, the support counting is rather time consuming. Alternatively, if the database fits into the memory, you can use the Eclat algorithm, which performs a depth-first search to count the supports. The Eclat algorithm, therefore, performs quicker than the Apriori algorithm. In this recipe, we introduce how to use the Eclat algorithm to generate frequent itemsets.

Getting ready

In this recipe, we will continue using the dataset `Groceries` as our input data source.

How to do it...

Perform the following steps to generate a frequent itemset using the Eclat algorithm:

- Similar to the Apriori method, we can use the `eclat` function to generate the frequent itemset:

```
> frequentsets=eclat(Groceries,parameter=list(support=0.05,maxL
en=10))
```
- We can then obtain the summary information from the generated frequent itemset:

```
> summary(frequentsets)
set of 31 itemsets
```

most frequent items:

whole milk	other vegetables	yogurt
4	2	2
rolls/buns	frankfurter	(Other)
2	1	23

element (itemset/transaction) length distribution:sizes

1 2
28 3

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.000	1.000	1.000	1.097	1.000	2.000

summary of quality measures:

support

Min.	:	0.05236
1st Qu.	:	0.05831
Median	:	0.07565
Mean	:	0.09212
3rd Qu.	:	0.10173
Max.	:	0.25552

includes transaction ID lists: FALSE

mining info:

data	ntransactions	support
Groceries	9835	0.05

3. Lastly, we can examine the top ten support frequent itemsets:

> inspect(sort(frequentsets,by="support") [1:10])

	items	support
1	{whole milk}	0.25551601
2	{other vegetables}	0.19349263
3	{rolls/buns}	0.18393493
4	{soda}	0.17437722
5	{yogurt}	0.13950178
6	{bottled water}	0.11052364
7	{root vegetables}	0.10899847
8	{tropical fruit}	0.10493137
9	{shopping bags}	0.09852567
10	{sausage}	0.09395018

How it works...

In this recipe, we introduce another algorithm, Eclat, to perform frequent itemset generation. Though Apriori is a straightforward and easy to understand association mining method, the algorithm has the disadvantage of generating huge candidate sets and performs inefficiently in support counting, for it takes multiple scans of databases. In contrast to Apriori, Eclat uses equivalence classes, depth-first searches, and set intersections, which greatly improves the speed in support counting.

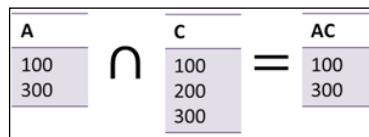
In Apriori, the algorithm uses a horizontal data layout to store transactions. On the other hand, Eclat uses a vertical data layout to store a list of transaction IDs (tid) for each item. Then, Eclat determines the support of any k+1-itemset by intersecting tid-lists of two k-itemsets. Lastly, Eclat utilizes frequent itemsets to generate association rules:

Horizontal Data Layout		Vertical Data Layout				
TID	Items	A	B	C	D	E
100	A C D	100	200	100	100	200
200	B C E	300	300	200	300	300
300	A B C E	400		300		400
400	B E					

↓
TID-list

An illustration of Eclat algorithm

Similar to the recipe using the Apriori algorithm, we can use the `eclat` function to generate a frequent itemset with a given support (assume support = 2 in this case) and maximum length.



Generating frequent itemset

We can then use the `summary` function to obtain summary statistics, which include: most frequent items, itemset length distributions, summary of quality measures, and mining information. Finally, we can sort frequent itemsets by the support and inspect the top ten support frequent itemsets.

See also

- ▶ Besides Apriori and Eclat, another popular association mining algorithm is **FP-Growth**. Similar to Eclat, this takes a depth-first search to count supports. However, there is no existing R package that you can download from CRAN that contains this algorithm. However, if you are interested in knowing how to apply the FP-growth algorithm in your transaction dataset, you can refer to Christian Borgelt's page at <http://www.borgelt.net/fpgrowth.html> for more information.

Creating transactions with temporal information

In addition to mining interesting associations within the transaction database, we can mine interesting sequential patterns using transactions with temporal information. In the following recipe, we demonstrate how to create transactions with temporal information.

Getting ready

In this recipe, we will generate transactions with temporal information. We can use the generated transactions as the input source for frequent sequential pattern mining.

How to do it...

Perform the following steps to create transactions with temporal information:

1. First, you need to install and load the package arulesSequences:

```
> install.packages("arulesSequences")
> library(arulesSequences)
```

2. You can first create a list with purchasing records:

```
> tmp_data=list(c("A"),
+                 c("A", "B", "C"),
+                 c("A", "C"),
+                 c("D"),
+                 c("C", "F"),
+                 c("A", "D"),
+                 c("C"),
+                 c("B", "C"),
+                 c("A", "E"),
+                 c("E", "F"),
+                 c("A", "B"),
+                 c("D", "F"),
+                 c("C"),
+                 c("B"),
+                 c("E"),
+                 c("G"),
+                 c("A", "F"),
+                 c("C"),
+                 c("B"),
+                 c("C"))
```

3. You can then turn the list into transactions and add temporal information:

```
>names(tmp_data) = paste("Tr",c(1:20), sep = "")  
>trans = as(tmp_data,"transactions")  
>transactionInfo(trans)$sequenceID  
=c(1,1,1,1,1,2,2,2,3,3,3,3,4,4,4,4,4)  
>transactionInfo(trans)$eventID=c(10,20,30,40,50,10,20,30,40,10,20  
,30,40,50,10,20,30,40,50,60)  
> trans  
transactions in sparse format with  
20 transactions (rows) and  
7 items (columns)
```

4. Next, you can use the inspect function to inspect the transactions:

```
> inspect(head(trans))  
  items transactionID sequenceID eventID  
1 {A}           Tr1         1       10  
2 {A,  
  B,  
  C}           Tr2         1       20  
3 {A,  
  C}           Tr3         1       30  
4 {D}           Tr4         1       40  
5 {C,  
  F}           Tr5         1       50  
6 {A,  
  D}           Tr6         2       10
```

5. You can then obtain the summary information of the transactions with the temporal information:

```
> summary(trans)  
transactions as itemMatrix in sparse format with  
20 rows (elements/itemsets/transactions) and  
7 columns (items) and a density of 0.2214286
```

most frequent items:

C	A	B	F	D (Other)
8	7	5	4	3 4

element (itemset/transaction) length distribution:

sizes

1	2	3
10	9	1

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
------	---------	--------	------	---------	------

1.00 1.00 1.50 1.55 2.00 3.00

includes extended item information - examples:

labels
1 A
2 B
3 C

includes extended transaction information - examples:

transactionID sequenceID eventID
1 Tr1 1 10
2 Tr2 1 20
3 Tr3 1 30

6. You can also read the transaction data in a basket format:

```
> zaki=read_baskets(con = system.file("misc", "zaki.txt", package
= "arulesSequences"), info = c("sequenceID","eventID","SIZE"))

> as(zaki, "data.frame")

  transactionID.sequenceID transactionID.eventID transactionID.
SIZE      items

1           {C,D}          1           10
2           {A,B,C}         1           15
3           {A,B,F}         1           20
3           {A,C,D,F}       1           25
4           {A,C,D,F}       2           15
3           {A,B,F}         2           20
6           {E}              3           10
3           {A,B,F}         4           10
3           {D,G,H}         4           20
2           {B,F}            4           25
10          {A,G,H}
```

How it works...

Before mining frequent sequential patterns, you are required to create transactions with the temporal information. In this recipe, we introduce two methods to obtain transactions with temporal information. In the first method, we create a list of transactions, and assign a transaction ID for each transaction. We use the `as` function to transform the list data into a transaction dataset. We then add `eventID` and `sequenceID` as temporal information; `sequenceID` is the sequence that the event belongs to, and `eventID` indicates when the event occurred. After generating transactions with temporal information, one can use this dataset for frequent sequential pattern mining.

In addition to creating your own transactions with temporal information, if you already have data stored in a text file, you can use the `read_basket` function from `arulesSequences` to read the transaction data into the basket format. We can also read the transaction dataset for further frequent sequential pattern mining.

See also

- ▶ The `arulesSequences` function provides two additional data structures, `sequences` and `timedsequences`, to present pure sequence data and sequence data with the time information. For those who are interested in these two collections, please use the `help` function to view the following documents:

```
> help("sequences-class")
> help("timedsequences-class")
```

Mining frequent sequential patterns with cSPADE

In contrast to association mining, which only discovers relationships between itemsets, we may be interested in exploring patterns shared among transactions where a set of itemsets occurs sequentially.

One of the most famous frequent sequential pattern mining algorithms is the **Sequential PAtern Discovery using Equivalence classes (SPADE)** algorithm, which employs the characteristics of a vertical database to perform an intersection on an ID list with an efficient lattice search and allows us to place constraints on mined sequences. In this recipe, we will demonstrate how to use cSPADE to mine frequent sequential patterns.

Getting ready

In this recipe, you have to complete the previous recipes by generating transactions with the temporal information and have it stored in the variable `trans`.

How to do it...

Perform the following steps to mine the frequent sequential patterns:

1. First, you can use the cspade function to generate frequent sequential patterns:

```
> s_result=cspade(trans,parameter = list(support = 0.75),control =  
list(verbose = TRUE))
```

2. You can then examine the summary of the frequent sequential patterns:

```
> summary(s_result)  
set of 14 sequences with
```

most frequent items:

C	A	B	D	E (Other)
8	5	5	2	1 1

most frequent elements:

{C}	{A}	{B}	{D}	{E} (Other)
8	5	5	2	1 1

element (sequence) size distribution:

sizes
1 2 3
6 6 2

sequence length distribution:

lengths
1 2 3
6 6 2

summary of quality measures:

support
Min. :0.7500
1st Qu.:0.7500
Median :0.7500
Mean :0.8393
3rd Qu.:1.0000

```
Max. :1.0000
```

```
includes transaction ID lists: FALSE
```

```
mining info:
```

	data	ntransactions	nsequences	support
trans		20	4	0.75

3. Transform a generated sequence format data back to the data frame:

```
> as(s_result, "data.frame")
```

	sequence	support
1	<{A}>	1.00
2	<{B}>	1.00
3	<{C}>	1.00
4	<{D}>	0.75
5	<{E}>	0.75
6	<{F}>	0.75
7	<{A},{C}>	1.00
8	<{B},{C}>	0.75
9	<{C},{C}>	0.75
10	<{D},{C}>	0.75
11	<{A},{C},{C}>	0.75
12	<{A},{B}>	1.00
13	<{C},{B}>	0.75
14	<{A},{C},{B}>	0.75

How it works...

The object of sequential pattern mining is to discover sequential relationships or patterns in transactions. You can use the pattern mining result to predict future events, or recommend items to users.

One popular method of sequential pattern mining is SPADE. SPADE uses a vertical data layout to store a list of IDs. In these, each input sequence in the database is called SID, and each event in a given input sequence is called EID. The process of SPADE is performed by generating patterns level-wisely by an Apriori candidate generation. In detail, SPADE generates subsequent n-sequences from joining (n-1)-sequences from the intersection of ID lists. If the number of sequences is greater than the **minimum support (minsup)**, we can consider the sequence to be frequent enough. The algorithm stops until the process cannot find more frequent sequences:

Database			Frequent Sequences (minsup =2)		
SID	EID	Items	Frequent 1-Sequences		Frequent 3-Sequences
1	100	C D	A	4	ABF 3
1	150	A B C	B	4	BF->A 2
1	200	A B F	D	2	D->BF 2
1	250	A C D F	F	4	D->B->A 2
			Frequent 2-Sequences		D->F->A 2
2	150	A B F	AB	3	
2	200	E	AF	3	
3	100	A B F	B->A	2	
4	100	D G H	BF	4	
4	200	B F	D->A	2	
4	250	A G H	D->B	2	
			D->F	2	
			F->A	2	

An illustration of SPADE algorithm

In this recipe, we illustrate how to use a frequent sequential pattern mining algorithm, cSPADE, to mine frequent sequential patterns. First, as we have transactions with temporal information loaded in the variable `trans`, we can use the `cspade` function with the support over 0.75 to generate frequent sequential patterns in the `sequences` format. We can then obtain summary information, such as most frequent items, sequence size distributions, a summary of quality measures, and mining information. Lastly, we can transform the generated sequence information back to the data frame format, so we can examine the sequence and support of frequent sequential patterns with the support over 0.75.

See also

- If you are interested in the concept and design of the SPADE algorithm, you can refer to the original published paper: M. J. Zaki. (2001). *SPADE: An Efficient Algorithm for Mining Frequent Sequences*. *Machine Learning Journal*, 42, 31–60.

10

Dimension Reduction

In this chapter, we will cover the following topics:

- ▶ Performing feature selection with FSelector
- ▶ Performing dimension reduction with PCA
- ▶ Determining the number of principal components using a scree test
- ▶ Determining the number of principal components using the Kaiser method
- ▶ Visualizing multivariate data using biplot
- ▶ Performing dimension reduction with MDS
- ▶ Reducing dimensions with SVD
- ▶ Compressing images with SVD
- ▶ Performing nonlinear dimension reduction with ISOMAP
- ▶ Performing nonlinear dimension deduction with Local Linear Embedding

Introduction

Most datasets contain features (such as attributes or variables) that are highly redundant. In order to remove irrelevant and redundant data to reduce the computational cost and avoid overfitting, you can reduce the features into a smaller subset without a significant loss of information. The mathematical procedure of reducing features is known as dimension reduction.

The reduction of features can increase the efficiency of data processing. Dimension reduction is, therefore, widely used in the fields of pattern recognition, text retrieval, and machine learning. Dimension reduction can be divided into two parts: feature extraction and feature selection. Feature extraction is a technique that uses a lower dimension space to represent data in a higher dimension space. Feature selection is used to find a subset of the original variables.

The objective of feature selection is to select a set of relevant features to construct the model. The techniques for feature selection can be categorized into feature ranking and feature selection. Feature ranking ranks features with a certain criteria and then selects features that are above a defined threshold. On the other hand, feature selection searches the optimal subset from a space of feature subsets.

In feature extraction, the problem can be categorized as linear or nonlinear. The linear method searches an affine space that best explains the variation of data distribution. In contrast, the nonlinear method is a better option for data that is distributed on a highly nonlinear curved surface. Here, we list some common linear and nonlinear methods.

Here are some common linear methods:

- ▶ **PCA:** Principal component analysis maps data to a lower dimension, so that the variance of the data in a low dimension representation is maximized.
- ▶ **MDS:** Multidimensional scaling is a method that allows you to visualize how near (pattern proximities) objects are to each other and can produce a representation of your data with lower dimension space. PCA can be regarded as the simplest form of MDS if the distance measurement used in MDS equals the covariance of data.
- ▶ **SVD:** Singular value decomposition removes redundant features that are linearly correlated from the perspective of linear algebra. PCA can also be regarded as a specific case of SVD.

Here are some common nonlinear methods:

- ▶ **ISOMAP:** ISOMAP can be viewed as an extension of MDS, which uses the distance metric of geodesic distances. In this method, geodesic distance is computed by graphing the shortest path distances.
- ▶ **LLE:** Locally linear embedding performs local PCA and global eigen-decomposition. LLE is a local approach, which involves selecting features for each category of the class feature. In contrast, ISOMAP is a global approach, which involves selecting features for all features.

In this chapter, we will first discuss how to perform feature ranking and selection. Next, we will focus on the topic of feature extraction and cover recipes in performing dimension reduction with both linear and nonlinear methods. For linear methods, we will introduce how to perform PCA, determine the number of principal components, and its visualization. We then move on to MDS and SVD. Furthermore, we will introduce the application of SVD to compress images. For nonlinear methods, we will introduce how to perform dimension reduction with ISOMAP and LLE.

Performing feature selection with FSelector

The FSelector package provides two approaches to select the most influential features from the original feature set. Firstly, rank features by some criteria and select the ones that are above a defined threshold. Secondly, search for optimum feature subsets from a space of feature subsets. In this recipe, we will introduce how to perform feature selection with the FSelector package.

Getting ready

In this recipe, we will continue to use the telecom `churn` dataset as the input data source to train the support vector machine. For those who have not prepared the dataset, please refer to *Chapter, Classification (I) – Tree, Lazy, and Probabilistic*, for detailed information.

How to do it...

Perform the following steps to perform feature selection on a `churn` dataset:

1. First, install and load the package, FSelector:

```
> install.packages("FSelector")
> library(FSelector)
```

2. Then, we can use `random森林.importance` to calculate the weight for each attribute, where we set the importance type to 1:

```
> weights = randomForest.importance(churn~, trainset,
  importance.type = 1)
> print(weights)
```

	attr_importance
international_plan	96.3255882
voice_mail_plan	24.8921239
number_vmail_messages	31.5420332
total_day_minutes	51.9365357
total_day_calls	-0.1766420
total_day_charge	53.7930096
total_eve_minutes	33.2006078
total_eve_calls	-2.2270323
total_eve_charge	32.4317375
total_night_minutes	22.0888120
total_night_calls	0.3407087
total_night_charge	21.6368855

total_intl_minutes	32.4984413
total_intl_calls	51.1154046
total_intl_charge	32.4855194
number_customer_service_calls	114.2566676

3. Next, we can use the cutoff function to obtain the attributes of the top five weights:

```
> subset = cutoff.k(weights, 5)
> f = as.simple.formula(subset, "Class")
> print(f)
Class ~ number_customer_service_calls + international_plan +
      total_day_charge + total_day_minutes + total_intl_calls
<environment: 0x00000000269a28e8>
```

4. Next, we can make an evaluator to select the feature subsets:

```
> evaluator = function(subset) {
+   k = 5
+   set.seed(2)
+   ind = sample(5, nrow(trainset), replace = TRUE)
+   results = sapply(1:k, function(i) {
+     train = trainset[ind == i,]
+     test = trainset[ind != i,]
+     tree = rpart(as.simple.formula(subset, "churn"), trainset)
+     error.rate = sum(test$churn != predict(tree, test,
+ type="class")) / nrow(test)
+     return(1 - error.rate)
+   })
+   return(mean(results))
+ }
```

5. Finally, we can find the optimum feature subset using a hill climbing search:

```
> attr.subset = hill.climbing.search(names(trainset)
[!names(trainset) %in% "churn"], evaluator)
> f = as.simple.formula(attr.subset, "churn")
> print(f)
churn ~ international_plan + voice_mail_plan + number_vmail_
messages +
      total_day_minutes + total_day_calls + total_eve_minutes +
```

```
total_eve_charge + total_intl_minutes + total_intl_calls +
total_intl_charge + number_customer_service_calls
<environment: 0x000000002224d3d0>
```

How it works...

In this recipe, we present how to use the `FSelector` package to select the most influential features. We first demonstrate how to use the feature ranking approach. In the feature ranking approach, the algorithm first employs a weight function to generate weights for each feature. Here, we use the random forest algorithm with the mean decrease in accuracy (`importance.type = 1`) as the importance measurement to gain the weights of each attribute. Besides the random forest algorithm, you can select other feature ranking algorithms (for example, `chi.squared`, `information.gain`) from the `FSelector` package. Then, the process sorts attributes by their weight. At last, we can obtain the top five features from the sorted feature list with the `cutoff` function. In this case, `number_customer_service_calls`, `international_plan`, `total_day_charge`, `total_day_minutes`, and `total_intl_calls` are the five most important features.

Next, we illustrate how to search for optimum feature subsets. First, we need to make a five-fold cross-validation function to evaluate the importance of feature subsets. Then, we use the hill climbing searching algorithm to find the optimum feature subsets from the original feature sets. Besides the hill-climbing method, one can select other feature selection algorithms (for example, `forward.search`) from the `FSelector` package. Lastly, we can find that `international_plan + voice_mail_plan + number_vmail_messages + total_day_minutes + total_day_calls + total_eve_minutes + total_eve_charge + total_intl_minutes + total_intl_calls + total_intl_charge + number_customer_service_calls` are optimum feature subsets.

See also

- ▶ You can also use the `caret` package to perform feature selection. As we have discussed related recipes in the model assessment chapter, you can refer to *Chapter, Model Evaluation*, for more detailed information.
- ▶ For both feature ranking and optimum feature selection, you can explore the package, `FSelector`, for more related functions:

```
> help(package="FSelector")
```

Performing dimension reduction with PCA

Principal component analysis (PCA) is the most widely used linear method in dealing with dimension reduction problems. It is useful when data contains many features, and there is redundancy (correlation) within these features. To remove redundant features, PCA maps high dimension data into lower dimensions by reducing features into a smaller number of principal components that account for most of the variance of the original features. In this recipe, we will introduce how to perform dimension reduction with the PCA method.

Getting ready

In this recipe, we will use the `swiss` dataset as our target to perform PCA. The `swiss` dataset includes standardized fertility measures and socio-economic indicators from around the year 1888 for each of the 47 French-speaking provinces of Switzerland.

How to do it...

Perform the following steps to perform principal component analysis on the `swiss` dataset:

1. First, load the `swiss` dataset:

```
> data(swiss)
```

2. Exclude the first column of the `swiss` data:

```
> swiss = swiss[, -1]
```

3. You can then perform principal component analysis on the `swiss` data:

```
> swiss.pca = prcomp(swiss,
```

```
+ center = TRUE,
```

```
+ scale = TRUE)
```

```
> swiss.pca
```

Standard deviations:

```
[1] 1.6228065 1.0354873 0.9033447 0.5592765 0.4067472
```

Rotation:

	PC1	PC2	PC3	PC4
PC5				
Agriculture	0.52396452	-0.25834215	0.003003672	-0.8090741
0.06411415				
Examination	-0.57185792	-0.01145981	-0.039840522	-0.4224580
-0.70198942				

Education	-0.49150243	0.19028476	0.539337412	-0.3321615
0.56656945				
Catholic	0.38530580	0.36956307	0.725888143	0.1007965
-0.42176895				
Infant.Mortality	0.09167606	0.87197641	-0.424976789	-0.2154928
0.06488642				

4. Obtain a summary from the PCA results:

```
> summary(swiss.pca)
```

Importance of components:

	PC1	PC2	PC3	PC4	PC5
Standard deviation	1.6228	1.0355	0.9033	0.55928	0.40675
Proportion of Variance	0.5267	0.2145	0.1632	0.06256	0.03309
Cumulative Proportion	0.5267	0.7411	0.9043	0.96691	1.00000

5. Lastly, you can use the `predict` function to output the value of the principal component with the first row of data:

```
> predict(swiss.pca, newdata=head(swiss, 1))
```

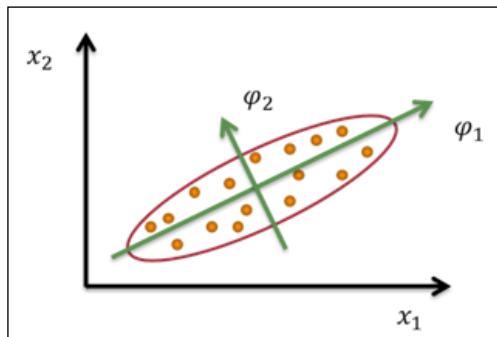
	PC1	PC2	PC3	PC4	PC5
Courtelary	-0.9390479	0.8047122	-0.8118681	1.000307	0.4618643

How it works...

Since the feature selection method may remove some correlated but informative features, you have to consider combining these correlated features into a single feature with the feature extraction method. PCA is one of the feature extraction methods, which performs orthogonal transformation to convert possibly correlated variables into principal components. Also, you can use these principal components to identify the directions of variance.

The process of PCA is carried on by the following steps: firstly, find the mean vector, $\mu = \frac{1}{n} \sum_{i=1}^n x_i$, where x_i indicates the data point, and n denotes the number of points. Secondly, compute the covariance matrix by the equation, $C = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)(x_i - \mu)^T$. Thirdly, compute the eigenvectors, φ , and the corresponding eigenvalues. In the fourth step, we rank and choose the top k eigenvectors. In the fifth step, we construct a $d \times k$ dimensional eigenvector matrix, U . Here, d is the number of original dimensions and k is the number of eigenvectors. Finally, we can transform data samples to a new subspace in the equation, $y = U^T \cdot x$.

In the following figure, it is illustrated that we can use two principal components, φ_1 , and φ_2 , to transform the data point from a two-dimensional space to new two-dimensional subspace:



A sample illustration of PCA

In this recipe we use the `prcomp` function from the `stats` package to perform PCA on the `swiss` dataset. First, we remove the standardized fertility measures and use the rest of the predictors as input to the function, `prcomp`. In addition to this, we set `swiss` as an input dataset; the variable should be shifted to the zero center by specifying `center=TRUE`; scale variables into the unit variance with the option, `scale=TRUE`, and store the output in the variable, `swiss.pca`.

Then, as we print out the value stored in `swiss.pca`, we can find the standard deviation and rotation of the principal component. The standard deviation indicates the square root of the eigenvalues of the covariance/correlation matrix. On the other hand, the rotation of the principal components shows the coefficient of the linear combination of the input features. For example, PC1 equals *Agriculture* * 0.524 + *Examination* * -0.572 + *Education* * -0.492 + *Catholic** 0.385 + *Infant.Mortality* * 0.092. Here, we can find that the attribute, *Agriculture*, contributes the most for PC1, for it has the highest coefficient.

Additionally, we can use the `summary` function to obtain the importance of components. The first row shows the standard deviation of each principal component, the second row shows the proportion of variance explained by each component, and the third row shows the cumulative proportion of the explained variance. Finally, you can use the `predict` function to obtain principal components from the input features. Here, we input the first row of the dataset, and retrieve five principal components.

There's more...

Another principal component analysis function is `princomp`. In this function, the calculation is performed by using `eigen` on a correlation or covariance matrix instead of a single value decomposition used in the `prcomp` function. In general practice, using `prcomp` is preferable; however, we cover how to use `princomp` here:

1. First, use princomp to perform PCA:

```
> swiss.princomp = princomp(swiss,
+ center = TRUE,
+ scale  = TRUE)
> swiss.princomp
Call:
princomp(x = swiss, center = TRUE, scale = TRUE)
```

Standard deviations:

Comp.1	Comp.2	Comp.3	Comp.4	Comp.5
42.896335	21.201887	7.587978	3.687888	2.721105

5 variables and 47 observations.

2. You can then obtain the summary information:

```
> summary(swiss.princomp)
```

Importance of components:

Comp.4	Comp.5	Comp.1	Comp.2	Comp.3
Standard deviation	3.687888330	42.8963346	21.2018868	7.58797830
Cumulative Proportion	0.996873399	0.7770024	0.1898152	0.02431275
Proportion of Variance	0.005742983	0.003126601	0.9668177	0.99113042
	1.000000000			

3. You can use the predict function to obtain principal components from the input features:

```
> predict(swiss.princomp, swiss[1,])
Comp.1     Comp.2     Comp.3     Comp.4     Comp.5
Courtelary -38.95923 -20.40504 12.45808  4.713234 -1.46634
```

In addition to the prcomp and princomp functions from the stats package, you can use the principal function from the psych package:

1. First, install and load the psych package:

```
> install.packages("psych")
> install.packages("GPArotation")
> library(psych)
```

2. You can then use the principal function to retrieve the principal components:

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

Principal Components Analysis

```
Call: principal(r = swiss, nfactors = 5, rotate = "none")
```

```
Standardized loadings (pattern matrix) based upon correlation
matrix
```

	PC1	PC2	PC3	PC4	PC5	h2	u2
Agriculture	-0.85	-0.27	0.00	0.45	-0.03	1	-6.7e-16
Examination	0.93	-0.01	-0.04	0.24	0.29	1	4.4e-16
Education	0.80	0.20	0.49	0.19	-0.23	1	2.2e-16
Catholic	-0.63	0.38	0.66	-0.06	0.17	1	-2.2e-16
Infant.Mortality	-0.15	0.90	-0.38	0.12	-0.03	1	-8.9e-16

	PC1	PC2	PC3	PC4	PC5
SS loadings	2.63	1.07	0.82	0.31	0.17
Proportion Var	0.53	0.21	0.16	0.06	0.03
Cumulative Var	0.53	0.74	0.90	0.97	1.00
Proportion Explained	0.53	0.21	0.16	0.06	0.03
Cumulative Proportion	0.53	0.74	0.90	0.97	1.00

Test of the hypothesis that 5 components are sufficient.

The degrees of freedom for the null model are 10 and the objective function was 2.13

The degrees of freedom for the model are -5 and the objective function was 0

The total number of observations was 47 with MLE Chi Square = 0 with prob < NA

Fit based upon off diagonal values = 1

Determining the number of principal components using the scree test

As we only need to retain the principal components that account for most of the variance of the original features, we can either use the Kaiser method, scree test, or the percentage of variation explained as the selection criteria. The main purpose of a scree test is to graph the component analysis results as a scree plot and find where the obvious change in the slope (elbow) occurs. In this recipe, we will demonstrate how to determine the number of principal components using a scree plot.

Getting ready

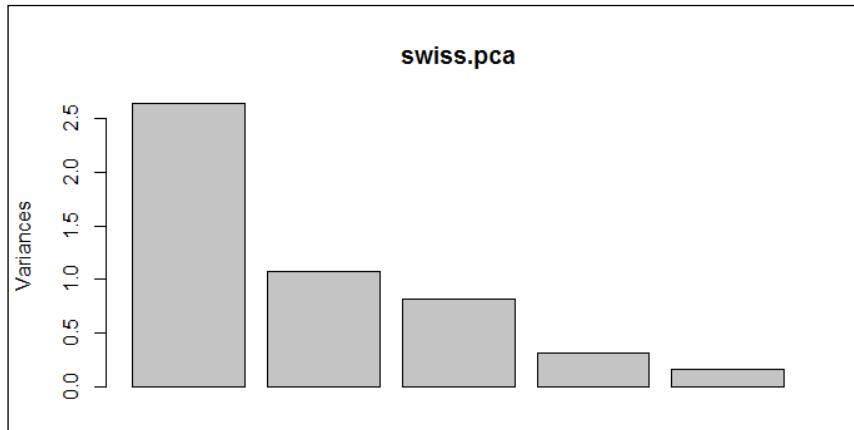
Ensure that you have completed the previous recipe by generating a principal component object and save it in the variable, `swiss.pca`.

How to do it...

Perform the following steps to determine the number of principal components with the scree plot:

1. First, you can generate a bar plot by using `screeplot`:

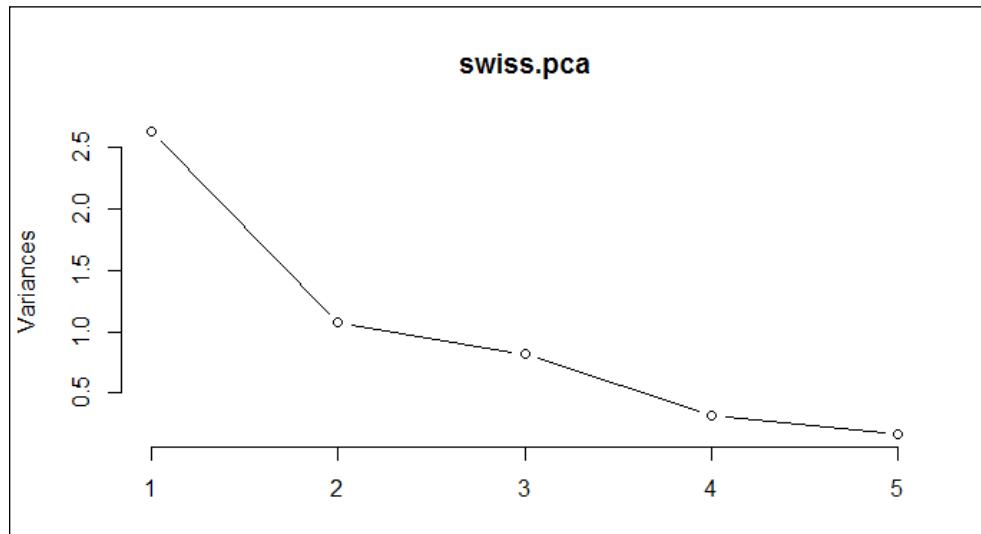
```
> screeplot(swiss.pca, type="barplot")
```



The scree plot in bar plot form

2. You can also generate a line plot by using `screeplot`:

```
> screeplot(swiss.pca, type="line")
```



The scree plot in line plot form

How it works...

In this recipe, we demonstrate how to use a scree plot to determine the number of principal components. In a scree plot, there are two types of plots, namely, bar plots and line plots. As both generated scree plots reveal, the obvious change in slope (the so-called elbow or knee) occurs at component 2. As a result, we should retain component 1, where the component is in a steep curve before component 2, which is where the flat line trend commences. However, as this method can be ambiguous, you can use other methods (such as the Kaiser method) to determine the number of components.

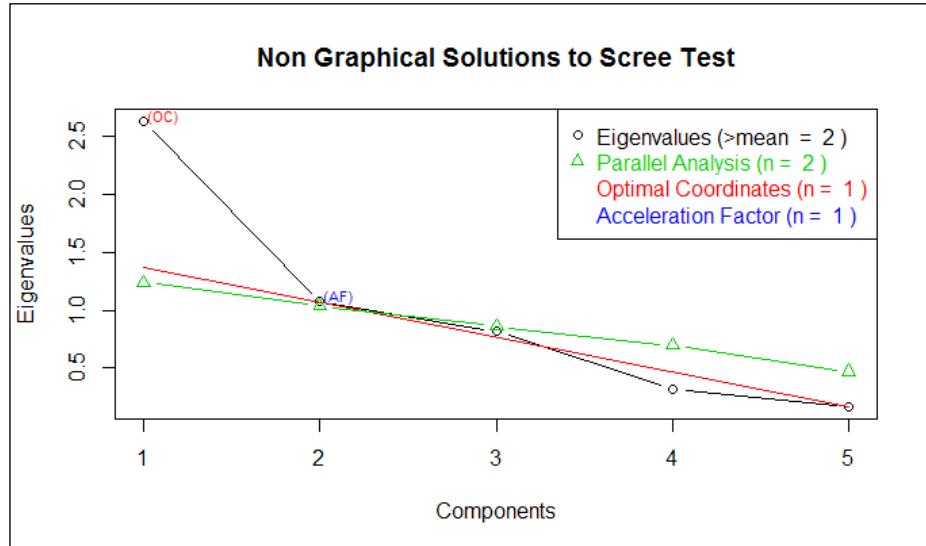
There's more...

By default, if you use the `plot` function on a generated principal component object, you can also retrieve the scree plot. For more details on `screeplot`, please refer to the following document:

```
> help(screeplot)
```

You can also use `nFactors` to perform parallel analysis and nongraphical solutions to the Cattell scree test:

```
> install.packages("nFactors")
> library(nFactors)
> ev = eigen(cor(swiss))
> ap = parallel(subject=nrow(swiss), var=ncol(swiss), rep=100, cent=.05)
> nS = nScree(x=ev$values, aparallel=ap$eigen$qevpea)
> plotnScree(nS)
```



Non-graphical solution to scree test

Determining the number of principal components using the Kaiser method

In addition to the scree test, you can use the Kaiser method to determine the number of principal components. In this method, the selection criteria retains eigenvalues greater than 1. In this recipe, we will demonstrate how to determine the number of principal components using the Kaiser method.

Getting ready

Ensure that you have completed the previous recipe by generating a principal component object and save it in the variable, `swiss.pca`.

How to do it...

Perform the following steps to determine the number of principal components with the Kaiser method:

1. First, you can obtain the standard deviation from `swiss.pca`:

```
> swiss.pca$sdev  
[1] 1.6228065 1.0354873 0.9033447 0.5592765 0.4067472
```

2. Next, you can obtain the variance from `swiss.pca`:

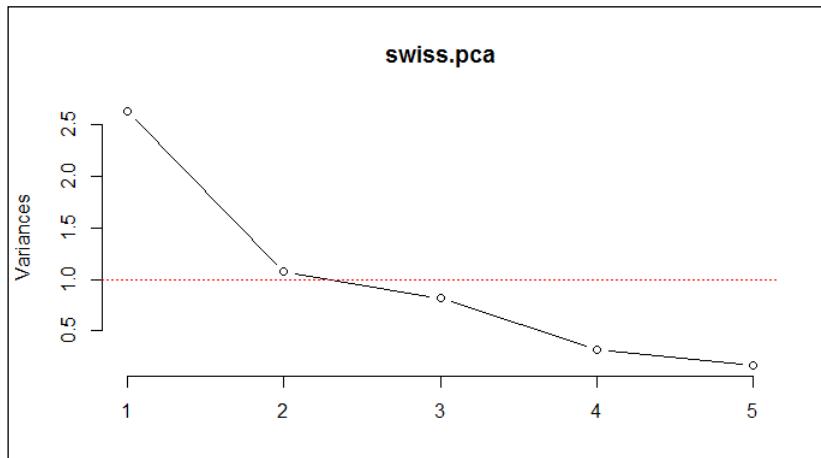
```
> swiss.pca$sdev ^ 2  
[1] 2.6335008 1.0722340 0.8160316 0.3127902 0.1654433
```

3. Select components with a variance above 1:

```
> which(swiss.pca$sdev ^ 2 > 1)  
[1] 1 2
```

4. You can also use the scree plot to select components with a variance above 1:

```
> screeplot(swiss.pca, type="line")  
> abline(h=1, col="red", lty= 3)
```



Select component with variance above 1

How it works...

You can also use the Kaiser method to determine the number of components. As the computed principal component object contains the standard deviation of each component, we can compute the variance as the standard deviation, which is the square root of variance. From the computed variance, we find both component 1 and 2 have a variance above 1. Therefore, we can determine the number of principal components as 2 (both component 1 and 2). Also, we can draw a red line on the scree plot (as shown in the preceding figure) to indicate that we need to retain component 1 and 2 in this case.

See also

In order to determine which principal components to retain, please refer to:

- ▶ Ledesma, R. D., and Valero-Mora, P. (2007). *Determining the Number of Factors to Retain in EFA: an easy-to-use computer program for carrying out Parallel Analysis*. *Practical Assessment, Research & Evaluation*, 12(2), 1-11.

Visualizing multivariate data using biplot

In order to find out how data and variables are mapped in regard to the principal component, you can use `biplot`, which plots data and the projections of original features on to the first two components. In this recipe, we will demonstrate how to use `biplot` to plot both variables and data on the same figure.

Getting ready

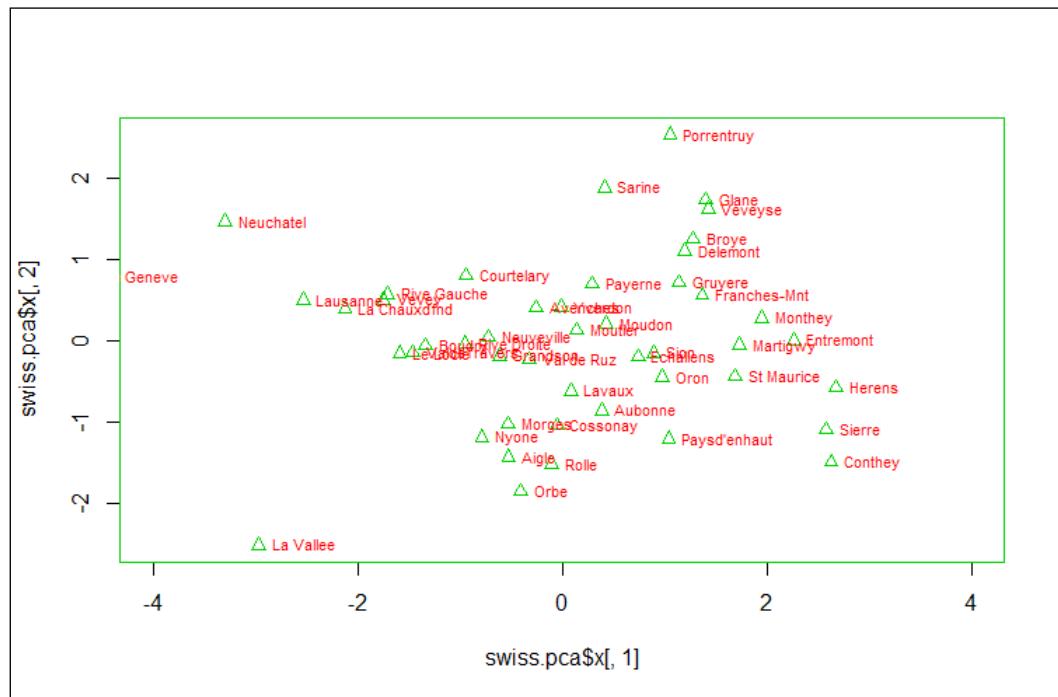
Ensure that you have completed the previous recipe by generating a principal component object and save it in the variable, `swiss.pca`.

How to do it...

Perform the following steps to create a biplot:

1. You can create a scatter plot using component 1 and 2:

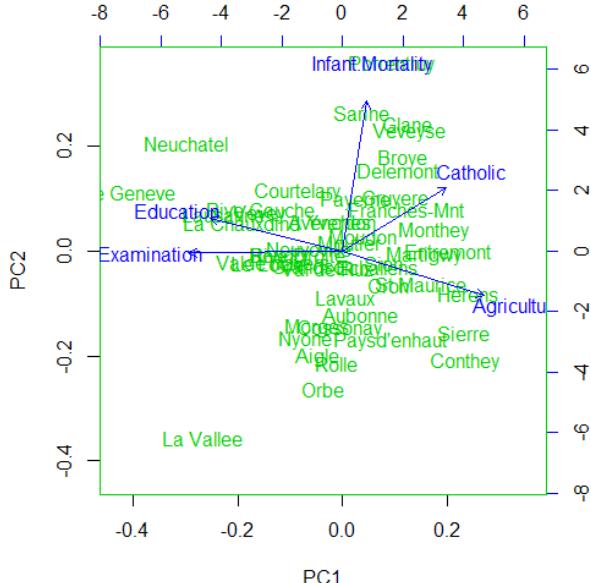
```
> plot(swiss.pca$x[,1], swiss.pca$x[,2], xlim=c(-4,4))  
> text(swiss.pca$x[,1], swiss.pca$x[,2], rownames(swiss.pca$x),  
cex=0.7, pos=4, col="red")
```



The scatter plot of first two components from PCA result

2. If you would like to add features on the plot, you can create biplot using the generated principal component object:

```
> biplot(swiss.pca)
```



The biplot using PCA result

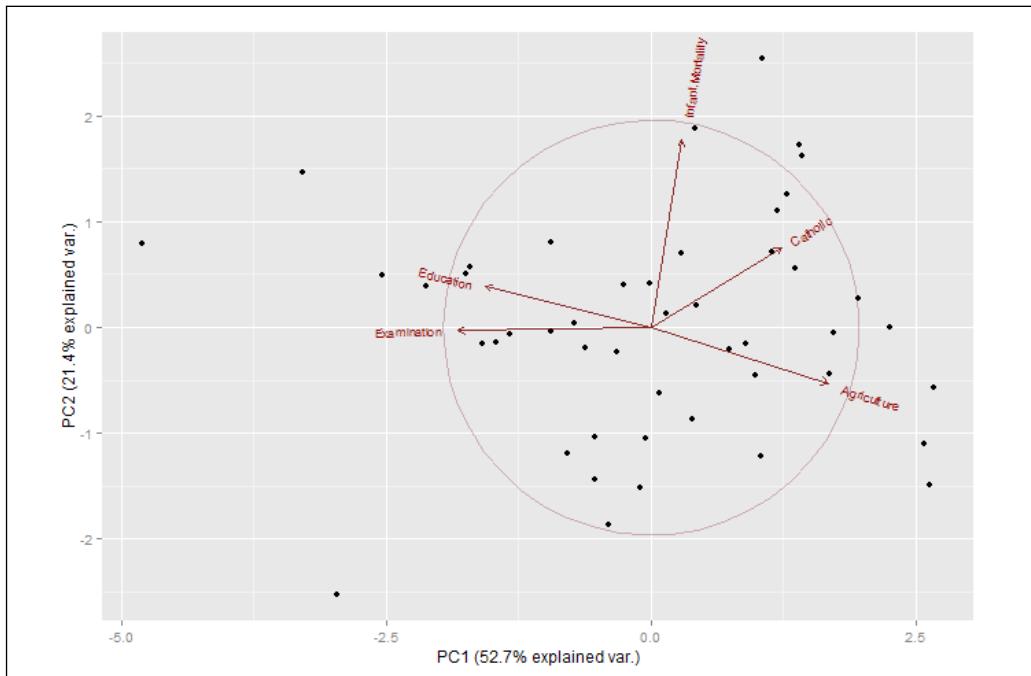
How it works...

In this recipe, we demonstrate how to use `biplot` to plot data and projections of original features on to the first two components. In the first step, we demonstrate that we can actually use the first two components to create a scatter plot. Furthermore, if you want to add variables on the same plot, you can use `biplot`. In `biplot`, you can see the provinces with higher indicators in the agriculture variable, lower indicators in the education variable, and examination variables scores that are higher in PC1. On the other hand, the provinces with higher infant mortality indicators and lower agriculture indicators score higher in PC2.

There's more...

Besides biplot in the stats package, you can also use ggbiplots. However, you may not find this package from CRAN; you have to first install devtools and then install ggbiplots from GitHub:

```
> install.packages("devtools")  
  
> library(ggbiplots)  
> g = ggbiplots(swiss.pca, obs.scale = 1, var.scale = 1,  
+ ellipse = TRUE,  
+ circle = TRUE)  
> print(g)
```



The ggbiplots using PCA result

Performing dimension reduction with MDS

Multidimensional scaling (MDS) is a technique to create a visual presentation of similarities or dissimilarities (distance) of a number of objects. The *multi* prefix indicates that one can create a presentation map in one, two, or more dimensions. However, we most often use MDS to present the distance between data points in one or two dimensions.

In MDS, you can either use a metric or a nonmetric solution. The main difference between the two solutions is that metric solutions try to reproduce the original metric, while nonmetric solutions assume that the ranks of the distance are known. In this recipe, we will illustrate how to perform MDS on the swiss dataset.

Getting ready

In this recipe, we will continue using the `swiss` dataset as our input data source.

How to do it...

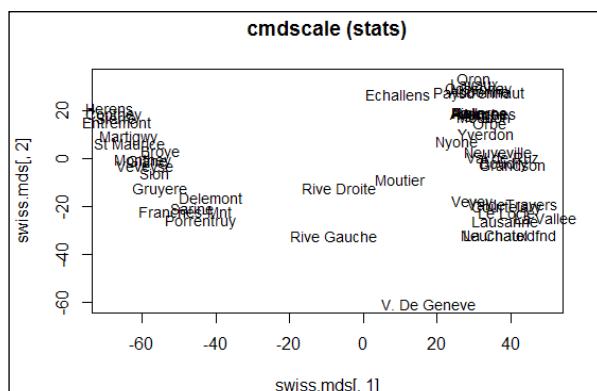
Perform the following steps to perform multidimensional scaling using the metric method:

1. First, you can perform metric MDS with a maximum of two dimensions:

```
> swiss.dist = dist(swiss)  
> swiss.mds = cmdscale(swiss.dist, k=2)
```

2. You can then plot the `swiss` data in a two-dimension scatter plot:

```
> plot(swiss.mds[,1], swiss.mds[,2], type = "n", main = "cmdscale  
(stats)")  
  
> text(swiss.mds[,1], swiss.mds[,2], rownames(swiss), cex = 0.9,  
xpd = TRUE)
```



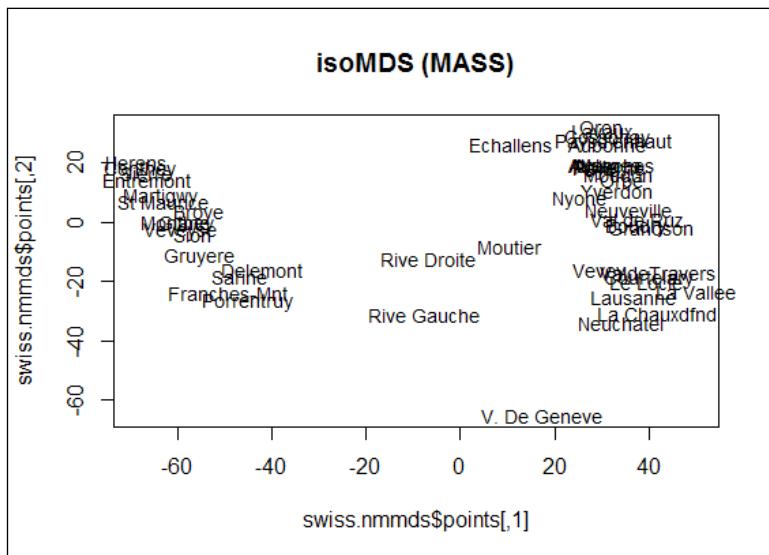
The 2-dimension scatter plot from cmdscale object

3. In addition, you can perform nonmetric MDS with isoMDS:

```
> library(MASS)  
  
> swiss.nmmds = isoMDS(swiss.dist, k=2)  
  
initial value 2.979731  
iter 5 value 2.431486  
iter 10 value 2.343353  
final value 2.338839  
converged
```

4. You can also plot the data points in a two-dimension scatter plot:

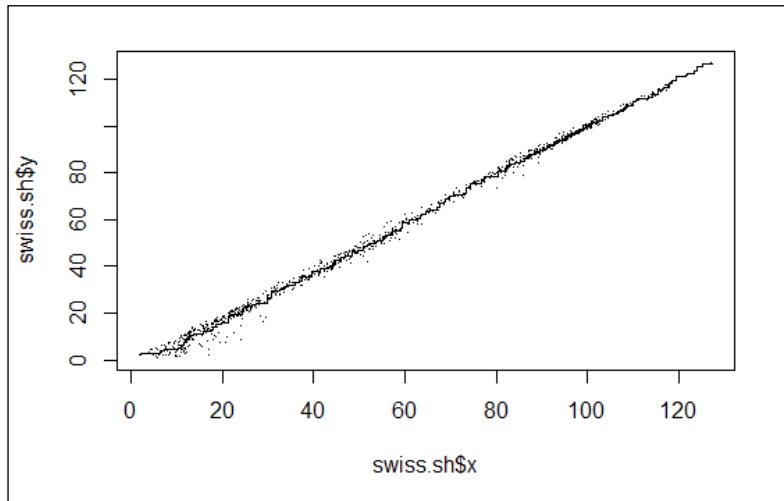
```
> plot(swiss.nmmds$points, type = "n", main = "isoMDS (MASS)")  
> text(swiss.nmmds$points, rownames(swiss), cex = 0.9, xpd = TRUE)
```



The 2-dimension scatter plot from isoMDS object

5. You can then plot the data points in a two-dimension scatter plot:

```
> swiss.sh = Shepard(swiss.dist, swiss.mds)  
> plot(swiss.sh, pch = ".")  
> lines(swiss.sh$x, swiss.sh$yf, type = "S")
```



The Shepard plot from isoMDS object

How it works...

MDS reveals the structure of the data by providing a visual presentation of similarities among a set of objects. In more detail, MDS places an object in an n-dimensional space, where the distances between pairs of points corresponds to the similarities among the pairs of objects. Usually, the dimensional space is a two-dimensional Euclidean space, but it may be non-Euclidean and have more than two dimensions. In accordance with the meaning of the input matrix, MDS can be mainly categorized into two types: metric MDS, where the input matrix is metric-based, nonmetric MDS, where the input matrix is nonmetric-based.

Metric MDS is also known as principal coordinate analysis, which first transforms a distance into similarities. In the simplest form, the process linearly projects original data points to a subspace by performing principal components analysis on similarities. On the other hand, the process can also perform a nonlinear projection on similarities by minimizing the stress value, $S = \sum_{k \neq l} [d(k,l) - d'(k,l)]^2$, where $d(k,l)$ is the distance measurement between the two points, x_k and x_l , and $d'(k,l)$ is the similarity measure of two projected points, x'_k and x'_l . As a result, we can represent the relationship among objects in the Euclidean space.

In contrast to metric MDS, which use a metric-based input matrix, a nonmetric-based MDS is used when the data is measured at the ordinal level. As only the rank order of the distances between the vectors is meaningful, nonmetric MDS applies a monotonically increasing function, f , on the original distances and projects the distance to new values that preserve the rank order. The normalized equation can be formulated as

$$S = \frac{1}{\sum_{k \neq l} [d'(k,l)]^2} \sum_{k \neq l} [f(d(k,l)) - d'(k,l)]^2$$

In this recipe, we illustrate how to perform metric and nonmetric MDS on the `swiss` dataset. To perform metric MDS, we first need to obtain the distance metric from the `swiss` data. In this step, you can replace the distance measure to any measure as long as it produces a similarity/dissimilarity measure of data points. You can use `cmdscale` to perform metric multidimensional scaling. Here, we specify `k = 2`, so the maximum generated dimensions equals 2. You can also visually present the distance of the data points on a two-dimensional scatter plot.

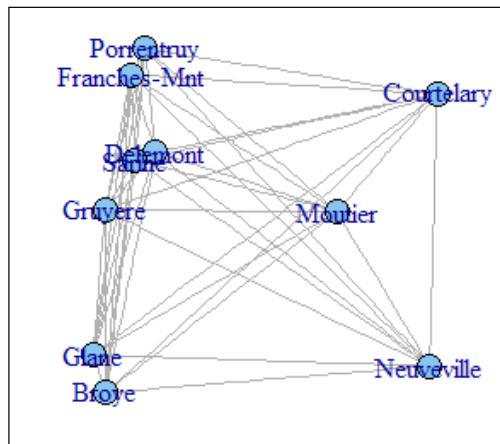
Next, you can perform nonmetric MDS with `isoMDS`. In nonmetric MDS, we do not match the distances, but only arrange them in order. We also set `swiss` as an input dataset with maximum dimensions of two. Similar to the metric MDS example, we can plot the distance between data points on a two-dimensional scatter plot. Then, we use a Shepard plot, which shows how well the projected distances match those in the distance matrix. As per the figure in step 4, the projected distance matches well in the distance matrix.

There's more...

Another visualization method is to present an MDS object as a graph. A sample code is listed here:

```
> library(igraph)
> swiss.sample = swiss[1:10,]

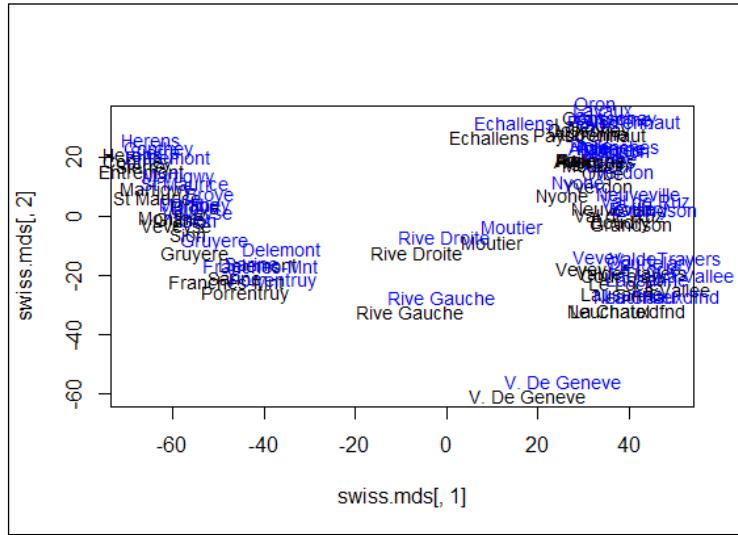
> g = graph.full(nrow(swiss.sample))
> V(g)$label = rownames(swiss.sample)
> layout = layout.mds(g, dist = as.matrix(dist(swiss.sample)))
> plot(g, layout = layout, vertex.size = 3)
```



The graph presentation of MDS object

You can also compare differences between the generated results from MDS and PCA. You can compare their differences by drawing the projected dimensions on the same scatter plot. If you use a Euclidean distance on MDS, the projected dimensions are exactly the same as the ones projected from PCA:

```
> swiss.dist = dist(swiss)
> swiss.mds = cmdscale(swiss.dist, k=2)
> plot(swiss.mds[,1], swiss.mds[,2], type="n")
> text(swiss.mds[,1], swiss.mds[,2], rownames(swiss), cex = 0.9, xpd = TRUE)
>
> swiss.pca = prcomp(swiss)
> text(-swiss.pca$x[,1], -swiss.pca$x[,2], rownames(swiss),
+       col="blue", adj = c(0.2, -0.5), cex = 0.9, xpd = TRUE)
```



The comparison between MDS and PCA

Reducing dimensions with SVD

Singular value decomposition (SVD) is a type of matrix factorization (decomposition), which can factorize matrices into two orthogonal matrices and diagonal matrices. You can multiply the original matrix back using these three matrices. SVD can reduce redundant data that is linear dependent from the perspective of linear algebra. Therefore, it can be applied to feature selection, image processing, clustering, and many other fields. In this recipe, we will illustrate how to perform dimension reduction with SVD.

Getting ready

In this recipe, we will continue using the dataset, `swiss`, as our input data source.

How to do it...

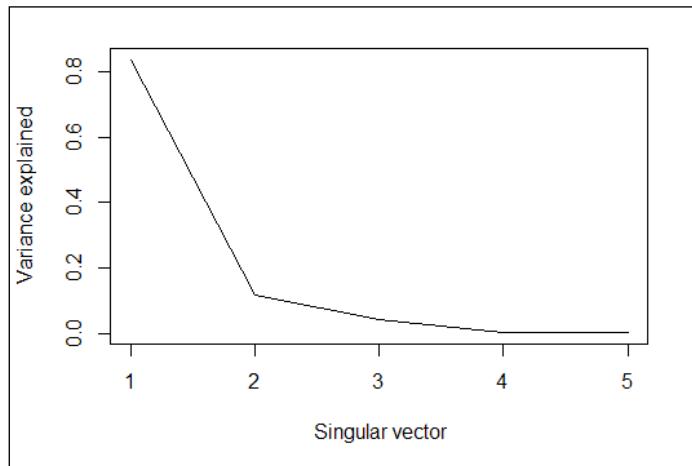
Perform the following steps to perform dimension reduction using SVD:

1. First, you can perform `svd` on the `swiss` dataset:

```
> swiss.svd = svd(swiss)
```

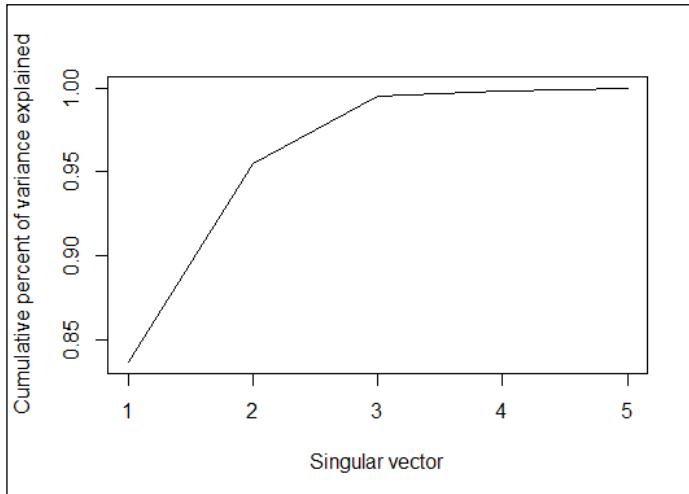
2. You can then plot the percentage of variance explained and the cumulative variance explained in accordance with the SVD column:

```
> plot(swiss.svd$d^2/sum(swiss.svd$d^2), type="l", xlab=" Singular vector", ylab = "Variance explained")
```



The percent of variance explained

```
> plot(cumsum(swiss.svd$d^2/sum(swiss.svd$d^2)), type="l",  
xlab="Singular vector", ylab = "Cumulative percent of variance  
explained")
```



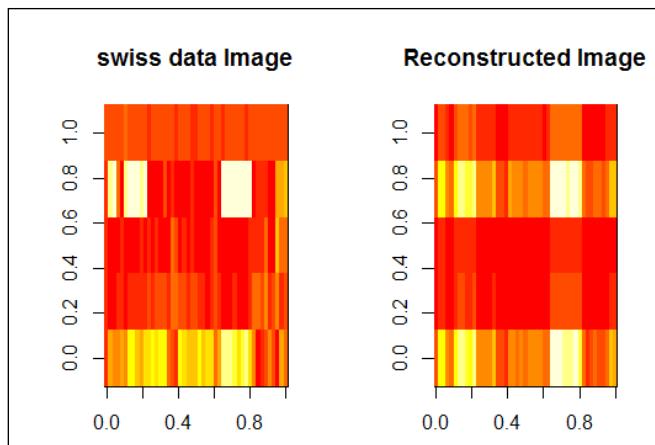
Cumulative percent of variance explained

3. Next, you can reconstruct the data with only one singular vector:

```
> swiss.recon = swiss.svd$u[,1] %*% diag(swiss.svd$d[1],  
length(1), length(1)) %*% t(swiss.svd$v[,1])
```

4. Lastly, you can compare the original dataset with the constructed dataset in an image:

```
> par(mfrow=c(1,2))  
> image(as.matrix(swiss), main="swiss data Image")  
> image(swiss.recon, main="Reconstructed Image")
```



The comparison between original dataset and re-constructed dataset

How it works...

SVD is a factorization of a real or complex matrix. In detail, the SVD of $m \times n$ matrix, A, is the factorization of A into the product of three matrices, $A = UDV^T$. Here, U is an $m \times m$ orthonormal matrix, D has singular values and is an $m \times n$ diagonal matrix, and V^T is an $n \times n$ orthonormal matrix.

In this recipe, we demonstrate how to perform dimension reduction with SVD. First, you can apply the `svd` function on the `swiss` dataset to obtain factorized matrices. You can then generate two plots: one shows the variance explained in accordance to a singular vector, the other shows the cumulative variance explained in accordance to a singular vector.

The preceding figure shows that the first singular vector can explain 80 percent of variance. We now want to compare the differences from the original dataset and the reconstructed dataset with a single singular vector. We, therefore, reconstruct the data with a single singular vector and use the `image` function to present the original and reconstructed datasets side-by-side and see how they differ from each other. The next figure reveals that these two images are very similar.

See also

- As we mentioned earlier, PCA can be regarded as a specific case of SVD. Here, we generate the orthogonal vector from the `swiss` data from SVD and obtained the rotation from `prcomp`. We can see that the two generated matrices are the same:

```
> svd.m = svd(scale(swiss))
> svd.m$v
      [,1]      [,2]      [,3]      [,4]      [,5]
[1,]  0.52396452 -0.25834215  0.003003672 -0.8090741  0.06411415
[2,] -0.57185792 -0.01145981 -0.039840522 -0.4224580 -0.70198942
[3,] -0.49150243  0.19028476  0.539337412 -0.3321615  0.56656945
[4,]  0.38530580  0.36956307  0.725888143  0.1007965 -0.42176895
[5,]  0.09167606  0.87197641 -0.424976789 -0.2154928  0.06488642
> pca.m = prcomp(swiss,scale=TRUE)
> pca.m$rotation
          PC1        PC2        PC3        PC4
PC5
Agriculture      0.52396452 -0.25834215  0.003003672 -0.8090741
0.06411415
Examination     -0.57185792 -0.01145981 -0.039840522 -0.4224580
-0.70198942
```

```
Education      -0.49150243  0.19028476  0.539337412 -0.3321615
0.56656945

Catholic       0.38530580  0.36956307  0.725888143  0.1007965
-0.42176895

Infant.Mortality 0.09167606  0.87197641 -0.424976789 -0.2154928
0.06488642
```

Compressing images with SVD

In the previous recipe, we demonstrated how to factorize a matrix with SVD and then reconstruct the dataset by multiplying the decomposed matrix. Furthermore, the application of matrix factorization can be applied to image compression. In this recipe, we will demonstrate how to perform SVD on the classic image processing material, Lenna.

Getting ready

In this recipe, you should download the image of Lenna beforehand (refer to <http://www.ece.rice.edu/~wakin/images/lena512.bmp> for this), or you can prepare an image of your own to see how image compression works.

How to do it...

Perform the following steps to compress an image with SVD:

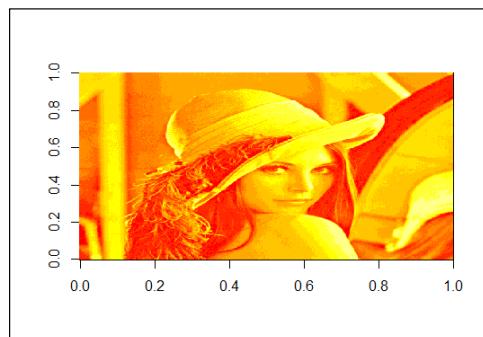
1. First, install and load `bmp`:

```
> install.packages("bmp")
> library(bmp)
```
2. You can then read the image of Lenna as a numeric matrix with the `read.bmp` function. When the reader downloads the image, the default name is `lena512.bmp`:

```
> lenna = read.bmp("lena512.bmp")
```

3. Rotate and plot the image:

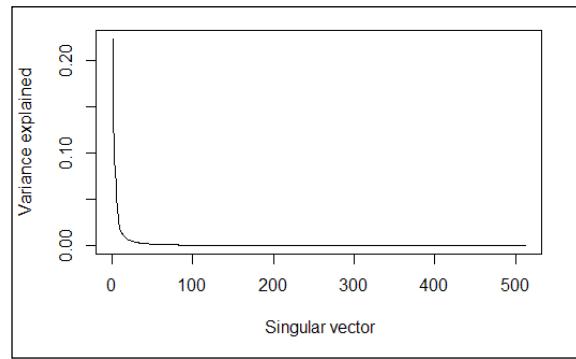
```
> lenna = t(lenna) [,nrow(lenna):1]  
> image(lenna)
```



The picture of Lenna

4. Next, you can perform SVD on the read numeric matrix and plot the percentage of variance explained:

```
> lenna.svd = svd(scale(lenna))  
> plot(lenna.svd$d^2/sum(lenna.svd$d^2), type="l", xlab=" Singular  
vector", ylab = "Variance explained")
```



The percentage of variance explained

5. Next, you can obtain the number of dimensions to reconstruct the image:

```
> length(lenna.svd$d)  
[1] 512
```

6. Obtain the point at which the singular vector can explain more than 90 percent of the variance:

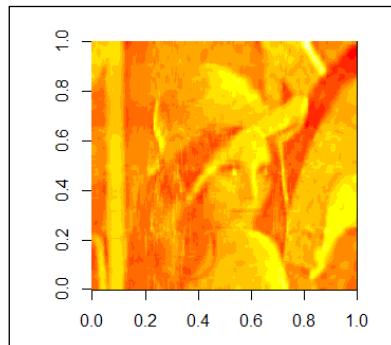
```
> min(which(cumsum(lenna.svd$d^2/sum(lenna.svd$d^2)) > 0.9))  
[1] 18
```

7. You can also wrap the code into a function, `lenna_compression`, and you can then use this function to plot compressed Lena:

```
> lenna_compression = function(dim){  
+   u=as.matrix(lenna.svd$u[, 1:dim])  
+   v=as.matrix(lenna.svd$v[, 1:dim])  
+   d=as.matrix(diag(lenna.svd$d) [1:dim, 1:dim])  
+   image(u%*%d%*%t(v))  
+ }
```

8. Also, you can use 18 vectors to reconstruct the image:

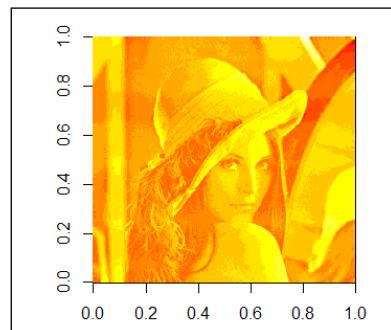
```
> lenna_compression(18)
```



The reconstructed image with 18 components

9. You can obtain the point at which the singular vector can explain more than 99 percent of the variance;

```
> min(which(cumsum(lenna.svd$d^2/sum(lenna.svd$d^2)) > 0.99))  
[1] 92  
> lenna_compression(92)
```



The reconstructed image with 92 components

How it works...

In this recipe, we demonstrate how to compress an image with SVD. In the first step, we use the package, `bmp`, to load the image, Lenna, to an R session. Then, as the read image is rotated, we can rotate the image back and use the `plot` function to plot Lenna in R (as shown in the figure in step 3). Next, we perform SVD on the image matrix to factorize the matrix. We then plot the percentage of variance explained in regard to the number of singular vectors.

Further, as we discover that we can use 18 components to explain 90 percent of the variance, we then use these 18 components to reconstruct Lenna. Thus, we make a function named `lenna_compression` with the purpose of reconstructing the image by matrix multiplication. As a result, we enter 18 as the input to the function, which returns a rather blurry Lenna image (as shown in the figure in step 8). However, we can at least see an outline of the image. To obtain a clearer picture, we discover that we can use 92 components to explain 99 percent of the variance. We, therefore, set the input to the function, `lenna_compression`, as 92. The figure in step 9 shows that this generates a clearer picture than the one constructed using merely 18 components.

See also

- ▶ The Lenna picture is one of the most widely used standard test images for compression algorithms. For more details on the Lenna picture, please refer to <http://www.cs.cmu.edu/~chuck/lennapg/>.

Performing nonlinear dimension reduction with ISOMAP

ISOMAP is one of the approaches for manifold learning, which generalizes linear framework to nonlinear data structures. Similar to MDS, ISOMAP creates a visual presentation of similarities or dissimilarities (distance) of a number of objects. However, as the data is structured in a nonlinear format, the Euclidian distance measure of MDS is replaced by the geodesic distance of a data manifold in ISOMAP. In this recipe, we will illustrate how to perform a nonlinear dimension reduction with ISOMAP.

Getting ready

In this recipe, we will use the `digits` data from `RnavGraphImageData` as our input source.

How to do it...

Perform the following steps to perform nonlinear dimension reduction with ISOMAP:

1. First, install and load the RnavGraphImageData and vegan packages:

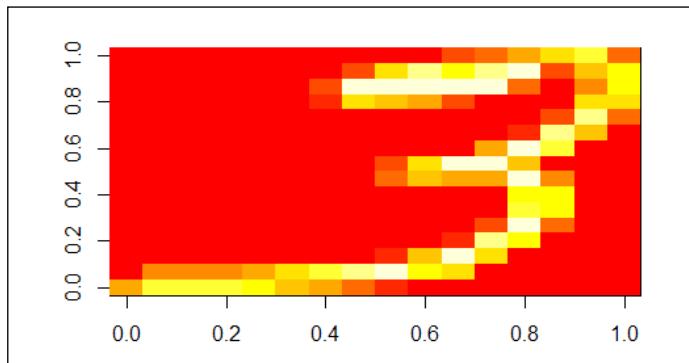
```
> install.packages("RnavGraphImageData")
> install.packages("vegan")
> library(RnavGraphImageData)
> library(vegan)
```

2. You can then load the dataset, digits:

```
> data(digits)
```

3. Rotate and plot the image:

```
> sample.digit = matrix(digits[,3000], ncol = 16, byrow=FALSE)
> image(t(sample.digit) [,nrow(sample.digit):1])
```



A sample image from the digits dataset

4. Next, you can randomly sample 300 digits from the population:

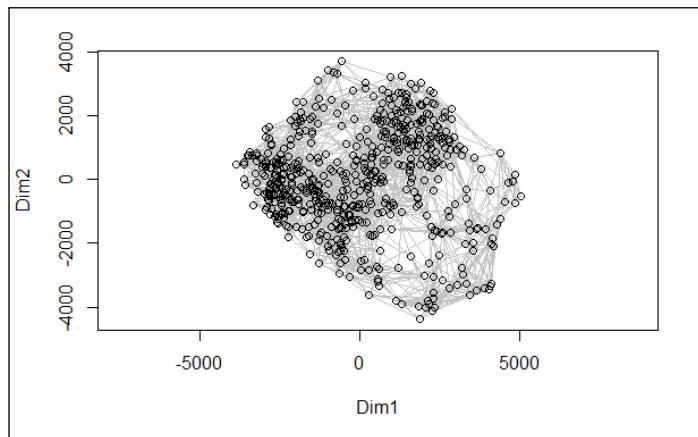
```
> set.seed(2)
> digit.idx = sample(1:ncol(digits), size = 600)
> digit.select = digits[,digit.idx]
```

5. Transpose the selected digit data and then compute the dissimilarity between objects using vegdist:

```
> digits.Transpose = t(digit.select)
> digit.dist = vegdist(digits.Transpose, method="euclidean")
```

6. Next, you can use isomap to perform dimension reduction:

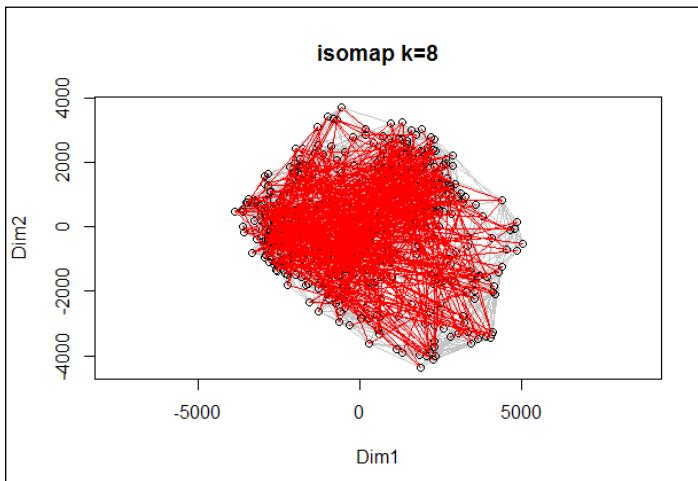
```
> digit.isomap = isomap(digit.dist, k = 8, ndim=6, fragmentedOK =  
TRUE)  
> plot(digit.isomap)
```



A 2-dimension scatter plot from ISOMAP object

7. Finally, you can overlay the scatter plot with the minimum spanning tree, marked in red;

```
> digit.st = spantree(digit.dist)  
> digit.plot = plot(digit.isomap, main="isomap k=8")  
> lines(digit.st, digit.plot, col="red")
```



A 2-dimension scatter plot overlay with minimum spanning tree

How it works...

ISOMAP is a nonlinear dimension reduction method and a representative of isometric mapping methods. ISOMAP can be regarded as an extension of the metric MDS, where pairwise the Euclidean distance among data points is replaced by geodesic distances induced by a neighborhood graph.

The description of the ISOMAP algorithm is shown in four steps. First, determine the neighbor of each point. Secondly, construct a neighborhood graph. Thirdly, compute the shortest distance path between two nodes. At last, find a low dimension embedding of the data by performing MDS.

In this recipe, we demonstrate how to perform a nonlinear dimension reduction using ISOMAP. First, we load the digits data from `RnavGraphImageData`. Then, after we select one digit and plot its rotated image, we can see an image of the handwritten digit (the numeral 3, in the figure in step 3).

Next, we randomly sample 300 digits as our input data to ISOMAP. We then transpose the dataset to calculate the distance between each image object. Once the data is ready, we calculate the distance between each object and perform a dimension reduction. Here, we use `vegdist` to calculate the dissimilarities between each object using a Euclidean measure. We then use ISOMAP to perform a nonlinear dimension reduction on the digits data with the dimension set as 6, number of shortest dissimilarities retained for a point as 8, and ensure that you analyze the largest connected group by specifying `fragmentedOK` as TRUE.

Finally, we can use the generated ISOMAP object to make a two-dimension scatter plot (figure in step 6), and also overlay the minimum spanning tree with lines in red on the scatter plot (figure in step 7).

There's more...

You can also use the `RnavGraph` package to visualize high dimensional data (digits in this case) using graphs as a navigational infrastructure. For more information, please refer to <http://www.icesi.edu.co/CRAN/web/packages/RnavGraph/vignettes/RnavGraph.pdf>.

Here is a description of how you can use `RnavGraph` to visualize high dimensional data in a graph:

1. First, install and load the `RnavGraph` and `graph` packages:

```
> install.packages("RnavGraph")
> source("http://bioconductor.org/biocLite.R")
> biocLite("graph")
> library(RnavGraph)
```

2. You can then create an NG_data object from the digit data:

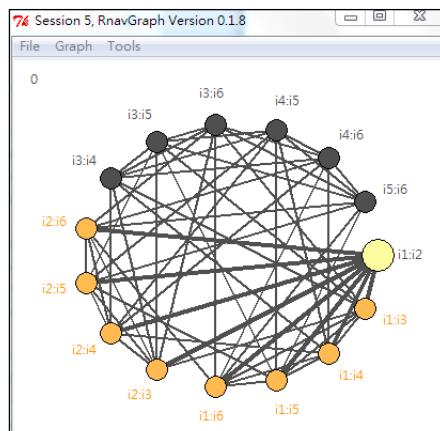
```
> digit.group = rep(c(1:9,0), each = 1100)
> digit.ng_data = ng_data(name = "ISO_digits",
+ data = data.frame(digit.isomap$points),
+ shortnames = paste('i',1:6, sep = ''),
+ group = digit.group[digit.idx],
+ labels = as.character(digits.group[digit.idx]))
```

3. Create an NG_graph object from NG_data:

```
> V = shortnames(digit.ng_data)
> G = completegraph(V)
> LG = linegraph(G)
> LGnot = complement(LG)
> ng.LG = ng_graph(name = "3D Transition", graph = LG)
> ng.LGnot = ng_graph(name = "4D Transition", graph = LGnot)
```

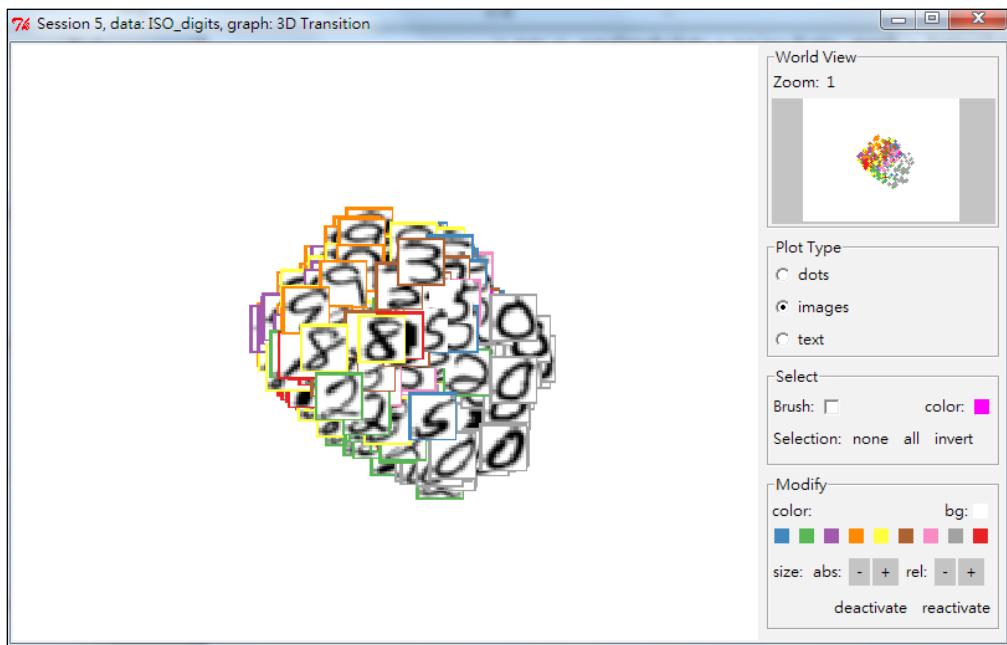
4. Finally, you can visualize the graph in the tk2d plot:

```
> ng.i.digits = ng_image_array_gray('USPS Handwritten Digits',
+ digit.select,16,16,invert = TRUE,
+ img_in_row = FALSE)
> vizDigits1 = ng_2d(data = digit.ng_data, graph = ng.LG, images =
ng.i.digits)
> vizDigits2 = ng_2d(data = digit.ng_data, graph = ng.LGnot,
images = ng.i.digits)
> nav = navGraph(data = digit.ng_data, graph = list(ng.LG,
ng.LGnot), viz = list(vizDigits1, vizDigits2))
```



A 3-D Transition graph plot

5. One can also view a 4D transition graph plot:



A 4D transition graph plot

Performing nonlinear dimension reduction with Local Linear Embedding

Locally linear embedding (LLE) is an extension of PCA, which reduces data that lies on a manifold embedded in a high dimensional space into a low dimensional space. In contrast to ISOMAP, which is a global approach for nonlinear dimension reduction, LLE is a local approach that employs a linear combination of the k-nearest neighbor to preserve local properties of data. In this recipe, we will give a short introduction of how to use LLE on an s-curve data.

Getting ready

In this recipe, we will use digit data from `lle_scurve_data` within the `lle` package as our input source.

How to do it...

Perform the following steps to perform nonlinear dimension reduction with LLE:

1. First, you need to install and load the package, lle:

```
> install.packages("lle")
> library(lle)
```

2. You can then load lle_scurve_data from lle:

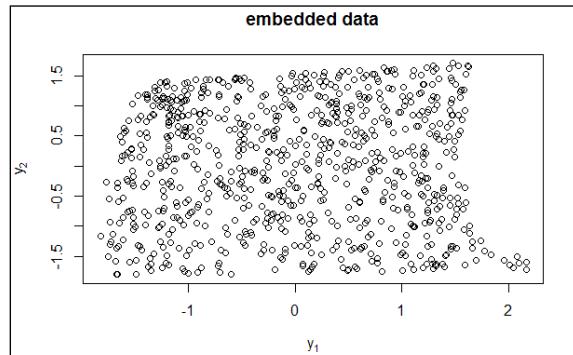
```
> data( lle_scurve_data )
```

3. Next, perform lle on lle_scurve_data:

```
> X = lle_scurve_data
> results = lle( X=X , m=2, k=12, id=TRUE)
finding neighbours
calculating weights
intrinsic dim: mean=2.47875, mode=2
computing coordinates
```

4. Examine the result with the str and plot function:

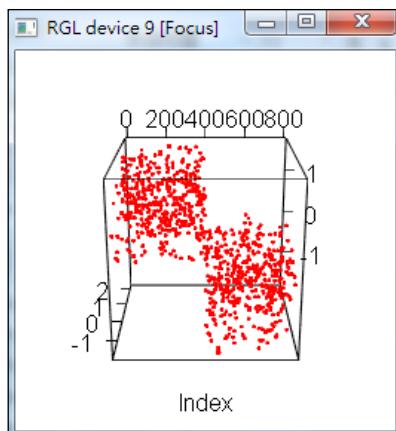
```
> str( results )
List of 4
$ Y      : num [1:800, 1:2] -1.586 -0.415 0.896 0.513 1.477 ...
$ X      : num [1:800, 1:3] 0.955 -0.66 -0.983 0.954 0.958 ...
$ choise: NULL
$ id     : num [1:800] 3 3 2 3 2 2 2 3 3 3 ...
>plot( results$Y, main="embedded data", xlab=expression(y[1]),
ylab=expression(y[2]) )
```



A 2-D scatter plot of embedded data

5. Lastly, you can use `plot_lle` to plot the LLE result:

```
> plot_lle( results$Y, X, FALSE, col="red", inter=TRUE )
```



A LLE plot of LLE result

How it works...

LLE is a nonlinear dimension reduction method, which computes a low dimensional, neighborhood, preserving embeddings of high dimensional data. The algorithm of LLE can be illustrated in these steps: first, LLE computes the k-neighbors of each data point, x_i . Secondly, it computes a set of weights for each point, which minimizes the residual sum of errors, which can best reconstruct each data point from its neighbors. The residual sum of errors can be described as $RSS(w) = \sum_{i=1}^n \left\| x_i - \sum_{j \neq i} w_{ij} x_j \right\|^2$, where $w_{ij} = 0$ if x_j is not one of x_i 's k-nearest neighbor, and for each i , $\sum_j w_{ij} = 1$. Finally, find the vector, Y , which is best reconstructed by the weight, W . The cost function can be illustrated as $\varphi(Y) = \sum_{i=1}^n \left\| y_i - \sum_{j \neq i} w_{ij} y_j \right\|^2$, with the constraint that $\sum_i Y_{ij} = 0$, and $Y^T Y = I$.

In this recipe, we demonstrate how to perform nonlinear dimension reduction using LLE. First, we load `lle_scurve_data` from `lle`. We then perform `lle` with two dimensions and 12 neighbors, and list the dimensions for every data point by specifying `id = TRUE`. The LLE has three steps, including: building a neighborhood for each point in the data, finding the weights for linearly approximating the data in that neighborhood, and finding the low dimensional coordinates.

Next, we can examine the data using the `str` and `plot` functions. The `str` function returns X , Y , choice, and ID. Here, X represents the input data, Y stands for the embedded data, choice indicates the index vector of the kept data, while subset selection and ID show the dimensions of every data input. The `plot` function returns the scatter plot of the embedded data. Lastly, we use `plot_lle` to plot the result. Here, we enable the interaction mode by setting the `inter` equal to `TRUE`.

See also

- ▶ Another useful package for nonlinear dimension reduction is RDRTToolbox, which is a package for nonlinear dimension reduction with ISOMAP and LLE. You can use the following command to install RDRTToolbox:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("RDRTToolbox")
> library(RDRTToolbox)
```

11

Big Data Analysis (R and Hadoop)

In this chapter, we will cover the following topics:

- ▶ Preparing the RHadoop environment
- ▶ Installing rmr2
- ▶ Installing rhdfs
- ▶ Operating HDFS with rhdfs
- ▶ Implementing a word count problem with RHadoop
- ▶ Comparing the performance between an R MapReduce program and a standard R program
- ▶ Testing and debugging the rmr2 program
- ▶ Installing plyrnr
- ▶ Manipulating data with plyrnr
- ▶ Conducting machine learning with RHadoop
- ▶ Configuring RHadoop clusters on Amazon EMR

Introduction

RHadoop is a collection of R packages that enables users to process and analyze big data with Hadoop. Before understanding how to set up RHadoop and put it in to practice, we have to know why we need to use machine learning to big-data scale.

In the previous chapters, we have mentioned how useful R is when performing data analysis and machine learning. In traditional statistical analysis, the focus is to perform analysis on historical samples (small data), which may ignore rarely occurring but valuable events and results to uncertain conclusions.

The emergence of Cloud technology has made real-time interaction between customers and businesses much more frequent; therefore, the focus of machine learning has now shifted to the development of accurate predictions for various customers. For example, businesses can provide real-time personal recommendations or online advertisements based on personal behavior via the use of a real-time prediction model.

However, if the data (for example, behaviors of all online users) is too large to fit in the memory of a single machine, you have no choice but to use a supercomputer or some other scalable solution. The most popular scalable big-data solution is Hadoop, which is an open source framework able to store and perform parallel computations across clusters. As a result, you can use RHadoop, which allows R to leverage the scalability of Hadoop, helping to process and analyze big data. In RHadoop, there are five main packages, which are:

- ▶ `rmr`: This is an interface between R and Hadoop MapReduce, which calls the Hadoop streaming MapReduce API to perform MapReduce jobs across Hadoop clusters. To develop an R MapReduce program, you only need to focus on the design of the map and reduce functions, and the remaining scalability issues will be taken care of by Hadoop itself.
- ▶ `rhdfs`: This is an interface between R and HDFS, which calls the HDFS API to access the data stored in HDFS. The use of `rhdfs` is very similar to the use of the Hadoop shell, which allows users to manipulate HDFS easily from the R console.
- ▶ `rbase`: This is an interface between R and HBase, which accesses Hbase and is distributed in clusters through a Thrift server. You can use `rbase` to read/write data and manipulate tables stored within HBase.
- ▶ `plyr`: This is a higher-level abstraction of MapReduce, which allows users to perform common data manipulation in a plyr-like syntax. This package greatly lowers the learning curve of big-data manipulation.
- ▶ `avro`: This allows users to read `avro` files in R, or write `avro` files. It allows R to exchange data with HDFS.

In this chapter, we will start by preparing the Hadoop environment, so that you can install RHadoop. We then cover the installation of three main packages: `rnr`, `rhdfs`, and `plyr`. Next, we will introduce how to use `rnr` to perform MapReduce from R, operate an HDFS file through `rhdfs`, and perform a common data operation using `plyr`. Further, we will explore how to perform machine learning using RHadoop. Lastly, we will introduce how to deploy multiple RHadoop clusters on Amazon EC2.

Preparing the RHadoop environment

As RHadoop requires an R and Hadoop integrated environment, we must first prepare an environment with both R and Hadoop installed. Instead of building a new Hadoop system, we can use the **Cloudera QuickStart VM** (the VM is free), which contains a single node Apache Hadoop Cluster and R. In this recipe, we will demonstrate how to download the Cloudera QuickStart VM.

Getting ready

To use the Cloudera QuickStart VM, it is suggested that you should prepare a 64-bit guest OS with either VMWare or VirtualBox, or the KVM installed.

If you choose to use VMWare, you should prepare a player compatible with WorkStation 8.x or higher: Player 4.x or higher, ESXi 5.x or higher, or Fusion 4.x or higher.

Note, 4 GB of RAM is required to start VM, with an available disk space of at least 3 GB.

How to do it...

Perform the following steps to set up a Hadoop environment using the Cloudera QuickStart VM:

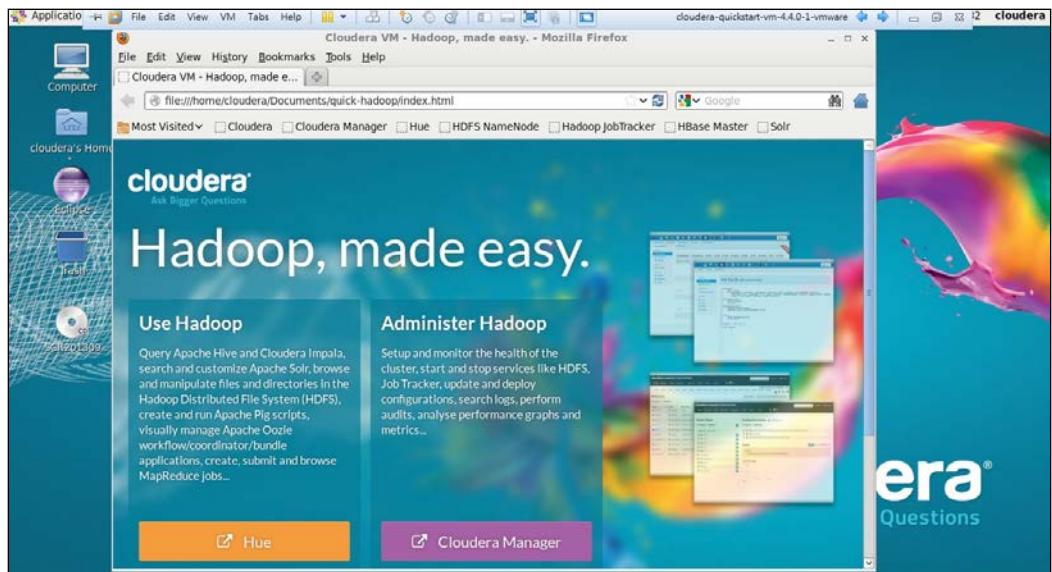
1. Visit the Cloudera QuickStart VM download site (you may need to update the link as Cloudera upgrades its VMs , the current version of CDH is 5.3) at http://www.cloudera.com/content/cloudera/en/downloads/quickstart_vms/cdh-5-3-x.html.

The screenshot shows the Cloudera website with the URL http://www.cloudera.com/content/cloudera/en/downloads/quickstart_vms/cdh-5-3-x.html. The page displays the 'QuickStart VMs for CDH 5.2.x' section, which is described as 'A Single-Node Hadoop Cluster and Examples for Easy Learning!'. It includes a note about running the VMs on CentOS 6.2 and a requirement for a 64-bit host OS. A dropdown menu shows 'Version: Quick Start VM with CDH 5.2.x'. Below this, a note states: 'Please Note: Cloudera QuickStart VMs are for demo purposes only and are not to be used as a starting point for clusters.' To the right, a 'Create New Virtual Machine' wizard window is open, titled 'Welcome to the New Virtual Machine Wizard!'. The window provides instructions for using the wizard and includes 'Go Back' and 'Next Step' buttons.

A screenshot of the Cloudera QuickStart VM download site

2. Depending on the virtual machine platform installed on your OS, choose the appropriate link (you may need to update the link as Cloudera upgrades its VMs) to download the VM file:
 - ❑ **To download VMWare:** You can visit https://downloads.cloudera.com/demo_vm/vmware/cloudera-quickstart-vm-5.2.0-0-vmware.7z
 - ❑ **To download KVM:** You can visit https://downloads.cloudera.com/demo_vm/kvm/cloudera-quickstart-vm-5.2.0-0-kvm.7z
 - ❑ **To download VirtualBox:** You can visit https://downloads.cloudera.com/demo_vm/virtualbox/cloudera-quickstart-vm-5.2.0-0-virtualbox.7z

3. Next, you can start the QuickStart VM using the virtual machine platform installed on your OS. You should see the desktop of Centos 6.2 in a few minutes.



The screenshot of Cloudera QuickStart VM.

4. You can then open a terminal and type hadoop, which will display a list of functions that can operate a Hadoop cluster.

```
[cloudera@quickstart ~]$ hadoop
Usage: hadoop [--config confdir] COMMAND
    where COMMAND is one of:
      fs                  run a generic filesystem user client
      version             print the version
      jar <jar>            run a jar file
      checknative [-a|-h]  check native hadoop and compression libraries availability
      distcp <srcurl> <desturl> copy file or directories recursively
      archive -archiveName NAME -p <parent path> <src>* <dest> create a hadoop archive
      classpath           prints the class path needed to get the Hadoop jar and the required libraries
      daemonlog           get/set the log level for each daemon
      or
      CLASSNAME          run the class named CLASSNAME

Most commands print help when invoked w/o parameters.
[cloudera@quickstart ~]$ █
```

The terminal screenshot after typing hadoop

5. Open a terminal and type R. Access an R session and check whether version 3.1.1 is already installed in the Cloudera QuickStart VM. If you cannot find R installed in the VM, please use the following command to install R:

```
$ yum install R R-core R-core-devel R-devel
```

How it works...

Instead of building a Hadoop system on your own, you can use the Hadoop VM application provided by Cloudera (the VM is free). The QuickStart VM runs on CentOS 6.2 with a single node Apache Hadoop cluster, Hadoop Ecosystem module, and R installed. This helps you to save time, instead of requiring you to learn how to install and use Hadoop.

The QuickStart VM requires you to have a computer with a 64-bit guest OS, at least 4 GB of RAM, 3 GB of disk space, and either VMWare, VirtualBox, or KVM installed. As a result, you may not be able to use this version of VM on some computers. As an alternative, you could consider using Amazon's Elastic MapReduce instead. We will illustrate how to prepare a RHadoop environment in EMR in the last recipe of this chapter.

Setting up the Cloudera QuickStart VM is simple. Download the VM from the download site and then open the built image with either VMWare, VirtualBox, or KVM. Once you can see the desktop of CentOS, you can then access the terminal and type hadoop to see whether Hadoop is working; then, type R to see whether R works in the QuickStart VM.

See also

- ▶ Besides using the Cloudera QuickStart VM, you may consider using a Sandbox VM provided by Hortonworks or MapR. You can find Hortonworks Sandbox at <http://hortonworks.com/products/hortonworks-sandbox/#install> and mapR Sandbox at <https://www.mapr.com/products/mapr-sandbox-hadoop/download>.

Installing rmr2

The `rmr2` package allows you to perform big data processing and analysis via MapReduce on a Hadoop cluster. To perform MapReduce on a Hadoop cluster, you have to install R and `rmr2` on every task node. In this recipe, we will illustrate how to install `rmr2` on a single node of a Hadoop cluster.

Getting ready

Ensure that you have completed the previous recipe by starting the Cloudera QuickStart VM and connecting the VM to the Internet, so that you can proceed with downloading and installing the `rmr2` package.

How to do it...

Perform the following steps to install `rmr2` on the QuickStart VM:

1. First, open the terminal within the Cloudera QuickStart VM.
2. Use the permission of the root to enter an R session:

```
$ sudo R
```

3. You can then install dependent packages before installing `rmr2`:

```
> install.packages(c("codetools", "Rcpp", "RJSONIO", "bitops",
  "digest", "functional", "stringr", "plyr", "reshape2", "rJava",
  "caTools"))
```

4. Quit the R session:
 5. Next, you can download `rmr-3.3.0` to the QuickStart VM. You may need to update the link if Revolution Analytics upgrades the version of `rmr2`:
- ```
$ wget --no-check-certificate https://raw.githubusercontent.com/
RevolutionAnalytics/rmr2/3.3.0/build/rmr2_3.3.0.tar.gz
```

6. You can then install `rmr-3.3.0` to the QuickStart VM:

```
$ sudo R CMD INSTALL rmr2_3.3.0.tar.gz
```

7. Lastly, you can enter an R session and use the `library` function to test whether the library has been successfully installed:

```
$ R
> library(rmr2)
```

## How it works...

In order to perform MapReduce on a Hadoop cluster, you have to install R and RHadoop on every task node. Here, we illustrate how to install `rmr2` on a single node of a Hadoop cluster. First, open the terminal of the Cloudera QuickStart VM. Before installing `rmr2`, we first access an R session with root privileges and install dependent R packages.

Next, after all the dependent packages are installed, quit the R session and use the `wget` command in the Linux shell to download `rnr-3.3.0` from GitHub to the local filesystem. You can then begin the installation of `rmr2`. Lastly, you can access an R session and use the `library` function to validate whether the package has been installed.

## See also

- ▶ To see more information and read updates about RHadoop, you can refer to the RHadoop wiki page hosted on GitHub: <https://github.com/RevolutionAnalytics/RHadoop/wiki>

## Installing rhdfs

The `rhdfs` package is the interface between R and HDFS, which allows users to access HDFS from an R console. Similar to `rmr2`, one should install `rhdfs` on every task node, so that one can access HDFS resources through R. In this recipe, we will introduce how to install `rhdfs` on the Cloudera QuickStart VM.

## Getting ready

Ensure that you have completed the previous recipe by starting the Cloudera QuickStart VM and connecting the VM to the Internet, so that you can proceed with downloading and installing the `rhdfs` package.

## How to do it...

Perform the following steps to install `rhdfs`:

1. First, you can download `rhdfs` 1.0.8 from GitHub. You may need to update the link if Revolution Analytics upgrades the version of `rhdfs`:

```
$ wget --no-check-certificate https://raw.github.com/
RevolutionAnalytics/rhdfs/master/build/rhdfs_1.0.8.tar.gz
```

2. Next, you can install `rhdfs` under the command-line mode:

```
$ sudo HADOOP_CMD=/usr/bin/hadoop R CMD INSTALL rhdfs_1.0.8.tar.
gz
```

3. You can then set up `JAVA_HOME`. The configuration of `JAVA_HOME` depends on the installed Java version within the VM:

```
$ sudo JAVA_HOME=/usr/java/jdk1.7.0_67-cloudera R CMD javareconf
```

4. Last, you can set up the system environment and initialize `rhdfs`. You may need to update the environment setup if you use a different version of QuickStart VM:

```
$ R
> Sys.setenv(HADOOP_CMD="/usr/bin/hadoop")
> Sys.setenv(HADOOP_STREAMING="/usr/lib/hadoop-mapreduce/hadoop-
streaming-2.5.0-cdh5.2.0.jar")
> library(rhdfs)
> hdfs.init()
```

## How it works...

The package, `rhdfs`, provides functions so that users can manage HDFS using R. Similar to `rmr2`, you should install `rhdfs` on every task node, so that one can access HDFS through the R console.

To install `rhdfs`, you should first download `rhdfs` from GitHub. You can then install `rhdfs` in R by specifying where the `HADOOP_CMD` is located. You must configure R with Java support through the command, `javareconf`.

Next, you can access R and configure where `HADOOP_CMD` and `HADOOP_STREAMING` are located. Lastly, you can initialize `rhdfs` via the `rhdfs.init` function, which allows you to begin operating HDFS through `rhdfs`.

## See also

- ▶ To find where `HADOOP_CMD` is located, you can use the `which hadoop` command in the Linux shell. In most Hadoop systems, `HADOOP_CMD` is located at `/usr/bin/hadoop`.
- ▶ As for the location of `HADOOP_STREAMING`, the streaming JAR file is often located in `/usr/lib/hadoop-mapreduce/`. However, if you cannot find the directory, `/usr/lib/Hadoop-mapreduce`, in your Linux system, you can search the streaming JAR by using the `locate` command. For example:

```
$ sudo updatedb
$ locate streaming | grep jar | more
```

## Operating HDFS with rhdfs

The `rhdfs` package is an interface between Hadoop and R, which can call an HDFS API in the backend to operate HDFS. As a result, you can easily operate HDFS from the R console through the use of the `rhdfs` package. In the following recipe, we will demonstrate how to use the `rhdfs` function to manipulate HDFS.

### Getting ready

To proceed with this recipe, you need to have completed the previous recipe by installing `rhdfs` into R, and validate that you can initial HDFS via the `hdfs.init` function.

### How to do it...

Perform the following steps to operate files stored on HDFS:

1. Initialize the `rhdfs` package:

```
> Sys.setenv(HADOOP_CMD="/usr/bin/hadoop")
> Sys.setenv(HADOOP_STREAMING="/usr/lib/hadoop-mapreduce/hadoop-streaming-2.5.0-cdh5.2.0.jar")
> library(rhdfs)
> hdfs.init ()
```

2. You can then manipulate files stored on HDFS, as follows:

- `hdfs.put`: Copy a file from the local filesystem to HDFS:  

```
> hdfs.put('word.txt', './')
```
- `hdfs.ls`: Read the list of directory from HDFS:  

```
> hdfs.ls('./')
```

- ❑ `hdfs.copy`: Copy a file from one HDFS directory to another:  
`> hdfs.copy('word.txt', 'wordcnt.txt')`
- ❑ `hdfs.move` : Move a file from one HDFS directory to another:  
`> hdfs.move('wordcnt.txt', './data/wordcnt.txt')`
- ❑ `hdfs.delete`: Delete an HDFS directory from R:  
`> hdfs.delete('./data/')`
- ❑ `hdfs.rm`: Delete an HDFS directory from R:  
`> hdfs.rm('./data/')`
- ❑ `hdfs.get`: Download a file from HDFS to a local filesystem:  
`> hdfs.get('word.txt', '/home/cloudera/word.txt')`
- ❑ `hdfs.rename`: Rename a file stored on HDFS:  
`hdfs.rename('./test/q1.txt','./test/test.txt')`
- ❑ `hdfs.chmod`: Change the permissions of a file or directory:  
`> hdfs.chmod('test', permissions= '777')`
- ❑ `hdfs.file.info`: Read the meta information of the HDFS file:  
`> hdfs.file.info('./')`

### 3. Also, you can write stream to the HDFS file:

```
> f = hdfs.file("iris.txt", "w")
> data(iris)
> hdfs.write(iris,f)
> hdfs.close(f)
```

### 4. Lastly, you can read stream from the HDFS file:

```
> f = hdfs.file("iris.txt", "r")
> dfserialized = hdfs.read(f)
> df = unserialize(dfserialized)
> df
> hdfs.close(f)
```

## How it works...

In this recipe, we demonstrate how to manipulate HDFS using the `rhdfs` package. Normally, you can use the Hadoop shell to manipulate HDFS, but if you would like to access HDFS from R, you can use the `rhdfs` package.

Before you start using `rhdfs`, you have to initialize `rhdfs` with `hdfs.init()`. After initialization, you can operate HDFS through `hdfs.read` and `hdfs.write`. We, therefore, demonstrate how to write a data frame in R to an HDFS file, `iris.txt`, using `hdfs.write`. Lastly, you can recover the written file back to the data frame using the `hdfs.read` function and the  `unserialize` function.

## See also

- ▶ To initialize `rhdfs`, you have to set `HADOOP_CMD` and `HADOOP_STREAMING` in the system environment. Instead of setting the configuration each time you're using `rhdfs`, you can put the configurations in the `.rprofile` file. Therefore, every time you start an R session, the configuration will be automatically loaded.

# Implementing a word count problem with RHadoop

To demonstrate how MapReduce works, we illustrate the example of a word count, which counts the number of occurrences of each word in a given input set. In this recipe, we will demonstrate how to use `rmr2` to implement a word count problem.

## Getting ready

In this recipe, we will need an input file as our word count program input. You can download the example input from [https://github.com/ywchiu/ml\\_R\\_cookbook/tree/master/CH12](https://github.com/ywchiu/ml_R_cookbook/tree/master/CH12).

## How to do it...

Perform the following steps to implement the word count program:

1. First, you need to configure the system environment, and then load `rmr2` and `rhdfs` into an R session. You may need to update the use of the JAR file if you use a different version of QuickStart VM:

```
> Sys.setenv(HADOOP_CMD="/usr/bin/hadoop")
> Sys.setenv(HADOOP_STREAMING="/usr/lib/hadoop-mapreduce/hadoop-streaming-2.5.0-cdh5.2.0.jar ")
> library(rmr2)
> library(rhdfs)
> hdfs.init()
```

2. You can then create a directory on HDFS and put the input file into the newly created directory:

```
> hdfs.mkdir("/user/cloudera/wordcount/data")
> hdfs.put("wc_input.txt", "/user/cloudera/wordcount/data")
```

3. Next, you can create a map function:

```
> map = function(.,lines) { keyval(
+ unlist(
+ strsplit(
+ x = lines,
+ split = " +")),
+ 1)}
```

4. Create a reduce function:

```
> reduce = function(word, counts) {
+ keyval(word, sum(counts))
+ }
```

5. Call the MapReduce program to count the words within a document:

```
> hdfs.root = 'wordcount'
> hdfs.data = file.path(hdfs.root, 'data')
> hdfs.out = file.path(hdfs.root, 'out')
> wordcount = function (input, output=NULL) {
+ mapreduce(input=input, output=output, input.format="text",
map=map,
+ reduce=reduce)
+ }
> out = wordcount(hdfs.data, hdfs.out)
```

6. Lastly, you can retrieve the top 10 occurring words within the document:

```
> results = from.dfs(out)
> results$key[order(results$val, decreasing = TRUE)][1:10]
```

## How it works...

In this recipe, we demonstrate how to implement a word count using the `rmr2` package. First, we need to configure the system environment and load `rhdfs` and `rnr2` into R. Then, we specify the input of our word count program from the local filesystem into the HDFS directory, `/user/cloudera/wordcount/data`, via the `hdfs.put` function.

Next, we begin implementing the MapReduce program. Normally, we can divide the MapReduce program into the `map` and `reduce` functions. In the `map` function, we first use the `strsplit` function to split each line into words. Then, as the `strsplit` function returns a list of words, we can use the `unlist` function to character vectors. Lastly, we can return key-value pairs with each word as a key and the value as one. As the `reduce` function receives the key-value pair generated from the `map` function, the `reduce` function sums the count and returns the number of occurrences of each word (or key).

After we have implemented the `map` and `reduce` functions, we can submit our job via the `mapreduce` function. Normally, the `mapreduce` function requires four inputs, which are the HDFS input path, the HDFS output path, the `map` function, and the `reduce` function. In this case, we specify the input as `wordcount/data`, output as `wordcount/out`, `map` function as `map`, `reduce` function as `reduce`, and wrap the `mapreduce` call in function, `wordcount`. Lastly, we call the function, `wordcount` and store the output path in the variable, `out`.

We can use the `from.dfs` function to load the HDFS data into the `results` variable, which contains the mapping of words and number of occurrences. We can then generate the top 10 occurring words from the `results` variable.

## See also

- ▶ In this recipe, we demonstrate how to write an R MapReduce program to solve a word count problem. However, if you are interested in how to write a native Java MapReduce program, you can refer to <http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>.

# Comparing the performance between an R MapReduce program and a standard R program

Those not familiar with how Hadoop works may often see Hadoop as a remedy for big data processing. Some might believe that Hadoop can return the processed results for any size of data within a few milliseconds. In this recipe, we will compare the performance between an R MapReduce program and a standard R program to demonstrate that Hadoop does not perform as quickly as some may believe.

## Getting ready

In this recipe, you should have completed the previous recipe by installing `rmr2` into the R environment.

## How to do it...

Perform the following steps to compare the performance of a standard R program and an R MapReduce program:

1. First, you can implement a standard R program to have all numbers squared:

```
> a.time = proc.time()
> small.ints2=1:100000
> result.normal = sapply(small.ints2, function(x) x^2)
> proc.time() - a.time
```

2. To compare the performance, you can implement an R MapReduce program to have all numbers squared:

```
> b.time = proc.time()
> small.ints= to.dfs(1:100000)
> result = mapreduce(input = small.ints, map = function(k,v)
cbind(v,v^2))
> proc.time() - b.time
```

## How it works...

In this recipe, we implement two programs to square all the numbers. In the first program, we use a standard R function, `sapply`, to square the sequence from 1 to 100,000. To record the program execution time, we first record the processing time before the execution in `a.time`, and then subtract `a.time` from the current processing time after the execution. Normally, the execution takes no more than 10 seconds. In the second program, we use the `rmr2` package to implement a program in the R MapReduce version. In this program, we also record the execution time. Normally, this program takes a few minutes to complete a task.

The performance comparison shows that a standard R program outperforms the MapReduce program when processing small amounts of data. This is because a Hadoop system often requires time to spawn daemons, job coordination between daemons, and fetching data from data nodes. Therefore, a MapReduce program often takes a few minutes to a couple of hours to finish the execution. As a result, if you can fit your data in the memory, you should write a standard R program to solve the problem. Otherwise, if the data is too large to fit in the memory, you can implement a MapReduce solution.

## See also

- In order to check whether a job will run smoothly and efficiently in Hadoop, you can run a MapReduce benchmark, MRBench, to evaluate the performance of the job:

```
$ hadoop jar /usr/lib/hadoop-0.20-mapreduce/hadoop-test.jar
mrbench -numRuns 50
```

# Testing and debugging the rmr2 program

Since running a MapReduce program will require a considerable amount of time, varying from a few minutes to several hours, testing and debugging become very important. In this recipe, we will illustrate some techniques you can use to troubleshoot an R MapReduce program.

## Getting ready

In this recipe, you should have completed the previous recipe by installing `rmr2` into an R environment.

## How to do it...

Perform the following steps to test and debug an R MapReduce program:

- First, you can configure the backend as local in `rmr.options`:  

```
> rmr.options(backend = 'local')
```
- Again, you can execute the number squared MapReduce program mentioned in the previous recipe:  

```
> b.time = proc.time()
> small.ints= to.dfs(1:100000)
> result = mapreduce(input = small.ints, map = function(k,v)
cbind(v,v^2))
> proc.time() - b.time
```
- In addition to this, if you want to print the structure information of any variable in the MapReduce program, you can use the `rmr.str` function:  

```
> out = mapreduce(to.dfs(1), map = function(k, v) rmr.str(v))
Dotted pair list of 14
```

```
$: language mapreduce(to.dfs(1), map = function(k, v) rmr.
str(v))
```

```

$: language mr(map = map, reduce = reduce, combine =
combine, vectorized.reduce, in.folder = if (is.list(input)) {
lapply(input, to.dfs.path) ...

$: language c.keyval(do.call(c, lapply(in.folder,
function(fname) { kv = get.data(fname) ...

$: language do.call(c, lapply(in.folder, function(fname) {
kv = get.data(fname) ...

$: language lapply(in.folder, function(fname) { kv = get.
data(fname) ...

$: language FUN("/tmp/Rtmp813BFJ/file25af6e85cfde"[[1L]], ...)

$: language uname(tapply(1:1kv, ceiling((1:1kv)/(1kv/(object.
size(kv)/10^6))), function(r) { kvr = slice.keyval(kv, r) ...

$: language tapply(1:1kv, ceiling((1:1kv)/(1kv/(object.
size(kv)/10^6))), function(r) { kvr = slice.keyval(kv, r) ...

$: language lapply(X = split(X, group), FUN = FUN, ...)

$: language FUN(X[[1L]], ...)

$: language as.keyval(map(keys(kvr), values(kvr)))

$: language is.keyval(x)

$: language map(keys(kvr), values(kvr))

$: length 2 rmr.str(v)

... attr(*, "srcref")=Class 'srcref' atomic [1:8] 1 34 1 58 34
58 1 1

... attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile'
<environment: 0x3f984f0>

v

num 1

```

## How it works...

In this recipe, we introduced some debugging and testing techniques you can use while implementing the MapReduce program. First, we introduced the technique to test a MapReduce program in a local mode. If you would like to run the MapReduce program in a pseudo distributed or fully distributed mode, it would take you a few minutes to several hours to complete the task, which would involve a lot of wastage of time while troubleshooting your MapReduce program. Therefore, you can set the backend to the local mode in `rmr.options` so that the program will be executed in the local mode, which takes lesser time to execute.

Another debugging technique is to list the content of the variable within the `map` or `reduce` function. In an R program, you can use the `str` function to display the compact structure of a single variable. In `rmr2`, the package also provides a function named `rmr.str`, which allows you to print out the content of a single variable onto the console. In this example, we use `rmr.str` to print the content of variables within a MapReduce program.

## See also

- ▶ For those who are interested in the option settings for the `rmr2` package, you can refer to the help document of `rmr.options`:

```
> help(rmr.options)
```

# Installing plyrMr

The `plyrMr` package provides common operations (as found in `plyr` or `reshape2`) for users to easily perform data manipulation through the MapReduce framework. In this recipe, we will introduce how to install `plyrMr` on the Hadoop system.

## Getting ready

Ensure that you have completed the previous recipe by starting the Cloudera QuickStart VM and connecting the VM to the Internet. Also, you need to have the `rmr2` package installed beforehand.

## How to do it...

Perform the following steps to install `plyrMr` on the Hadoop system:

1. First, you should install `libxml2-devel` and `curl-devel` in the Linux shell:

```
$ yum install libxml2-devel
$ sudo yum install curl-devel
```

2. You can then access R and install the dependent packages:

```
$ sudo R
> Install.packages(c("Rcurl", "httr"), dependencies = TRUE)
> Install.packages("devtools", dependencies = TRUE)
> library(devtools)
> install_github("pryr", "hadley")
> install.packages(c("R.methodsS3", "hydroPSO"), dependencies = TRUE)
> q()
```

3. Next, you can download `plyrMr` 0.5.0 and install it on Hadoop VM. You may need to update the link if Revolution Analytics upgrades the version of `plyrMr`:

```
$ wget -no-check-certificate https://raw.github.com/
RevolutionAnalytics/plyrMr/master/build/plyrMr_0.5.0.tar.gz
$ sudo R CMD INSTALL plyrMr_0.5.0.tar.gz
```

#### 4. Lastly, validate the installation:

```
$ R
> library(plyrmr)
```

## How it works...

Besides writing an R MapReduce program using the `rmr2` package, you can use the `plyrmr` to manipulate data. The `plyrmr` package is similar to `hive` and `pig` in the Hadoop ecosystem, which is the abstraction of the MapReduce program. Therefore, we can implement an R MapReduce program in `plyr` style instead of implementing the `map` f and `reduce` functions.

To install `plyrnr`, first install the package of `libxml2-devel` and `curl-devel`, using the `yum install` command. Then, access R and install the dependent packages. Lastly, download the file from GitHub and install `plyrnr` in R.

## See also

- ▶ To read more information about `plyrnr`, you can use the `help` function to refer to the following document:

```
> help(package=plyrnr)
```

## Manipulating data with `plyrnr`

While writing a MapReduce program with `rnr2` is much easier than writing a native Java version, it is still hard for nondevelopers to write a MapReduce program. Therefore, you can use `plyrnr`, a high-level abstraction of the MapReduce program, so that you can use `plyr`-like operations to manipulate big data. In this recipe, we will introduce some operations you can use to manipulate data.

## Getting ready

In this recipe, you should have completed the previous recipes by installing `plyrnr` and `rnr2` in R.

## How to do it...

Perform the following steps to manipulate data with `plyrnr`:

1. First, you need to load both `plyrnr` and `rnr2` into R:

```
> library(rnr2)
> library(plyrnr)
```

2. You can then set the execution mode to the local mode:

```
> plyr$mr.options(backend="local")
```

3. Next, load the Titanic dataset into R:

```
> data(Titanic)
> titanic = data.frame(Titanic)
```

4. Begin the operation by filtering the data:

```
> where(
+ Titanic,
+ Freq >=100)
```

5. You can also use a pipe operator to filter the data:

```
> titanic %|% where(Freq >=100)
```

6. Put the Titanic data into HDFS and load the path of the data to the variable, `tidata`:

```
> tidata = to.dfs(data.frame(Titanic), output = '/tmp/titanic')
> tidata
```

7. Next, you can generate a summation of the frequency from the Titanic data:

```
> input(tidata) %|% transmute(sum(Freq))
```

8. You can also group the frequency by sex:

```
> input(tidata) %|% group(Sex) %|% transmute(sum(Freq))
```

9. You can then sample 10 records out of the population:

```
> sample(input(tidata), n=10)
```

10. In addition to this, you can use `plyr$mr` to join two datasets:

```
> convert_tb = data.frame(Label=c("No", "Yes"), Symbol=c(0,1))
ctb = to.dfs(convert_tb, output = 'convert')
> as.data.frame(plyr$mr::merge(input(tidata), input(ctb),
by.x="Survived", by.y="Label"))
> file.remove('convert')
```

## How it works...

In this recipe, we introduce how to use `plyr$mr` to manipulate data. First, we need to load the `plyr$mr` package into R. Then, similar to `rmr2`, you have to set the backend option of `plyr$mr` as the local mode. Otherwise, you will have to wait anywhere between a few minutes to several hours if `plyr$mr` is running on Hadoop mode (the default setting).

Next, we can begin the data manipulation with data filtering. You can choose to call the function nested inside the other function call in step 4. On the other hand, you can use the pipe operator, `% | %`, to chain multiple operations. Therefore, we can filter data similar to step 4, using pipe operators in step 5.

Next, you can input the dataset into either the HDFS or local filesystem, using `to .dfs` in accordance with the current running mode. The function will generate the path of the dataset and save it in the variable, `tidata`. By knowing the path, you can access the data using the `input` function. Next, we illustrate how to generate a summation of the frequency from the Titanic dataset with the `transmute` and `sum` functions. Also, `plyrivr` allows users to sum up the frequency by gender.

Additionally, in order to sample data from a population, you can also use the `sample` function to select 10 records out of the Titanic dataset. Lastly, we demonstrate how to join two datasets using the `merge` function from `plyrivr`.

## See also

Here we list some functions that can be used to manipulate data with `plyrivr`. You may refer to the `help` function for further details on their usage and functionalities:

- ▶ Data manipulation:
  - `bind.cols`: This adds new columns
  - `select`: This is used to select columns
  - `where`: This is used to select rows
  - `transmute`: This uses all of the above plus their summaries
- ▶ From `reshape2`:
  - `melt` and `dcast`: It converts long and wide data frames
- ▶ Summary:
  - `count`
  - `quantile`
  - `sample`
- ▶ Extract:
  - `top.k`
  - `bottom.k`

# Conducting machine learning with RHadoop

In the previous chapters, we have demonstrated how powerful R is when used to solve machine learning problems. Also, we have shown that the use of Hadoop allows R to process big data in parallel. At this point, some may believe that the use of RHadoop can easily solve machine learning problems of big data via numerous existing machine learning packages. However, you cannot use most of these to solve machine learning problems as they cannot be executed in the MapReduce mode. In the following recipe, we will demonstrate how to implement a MapReduce version of linear regression and compare this version with the one using the `lm` function.

## Getting ready

In this recipe, you should have completed the previous recipe by installing `rmr2` into the R environment.

## How to do it...

Perform the following steps to implement a linear regression in MapReduce:

1. First, load the `cats` dataset from the `MASS` package:

```
> library(MASS)
> data(cats)
> X = matrix(cats$Bwt)
> y = matrix(cats$Hwt)
```

2. You can then generate a linear regression model by calling the `lm` function:

```
> model = lm(y~X)
> summary(model)
```

**Call:**

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

**Residuals:**

| Min     | 1Q      | Median  | 3Q     | Max    |
|---------|---------|---------|--------|--------|
| -3.5694 | -0.9634 | -0.0921 | 1.0426 | 5.1238 |

**Coefficients:**

| Estimate | Std. Error | t value | Pr(> t ) |
|----------|------------|---------|----------|
|----------|------------|---------|----------|

```

(Intercept) -0.3567 0.6923 -0.515 0.607
X 4.0341 0.2503 16.119 <2e-16 ***

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

```

```

Residual standard error: 1.452 on 142 degrees of freedom
Multiple R-squared: 0.6466, Adjusted R-squared: 0.6441
F-statistic: 259.8 on 1 and 142 DF, p-value: < 2.2e-16

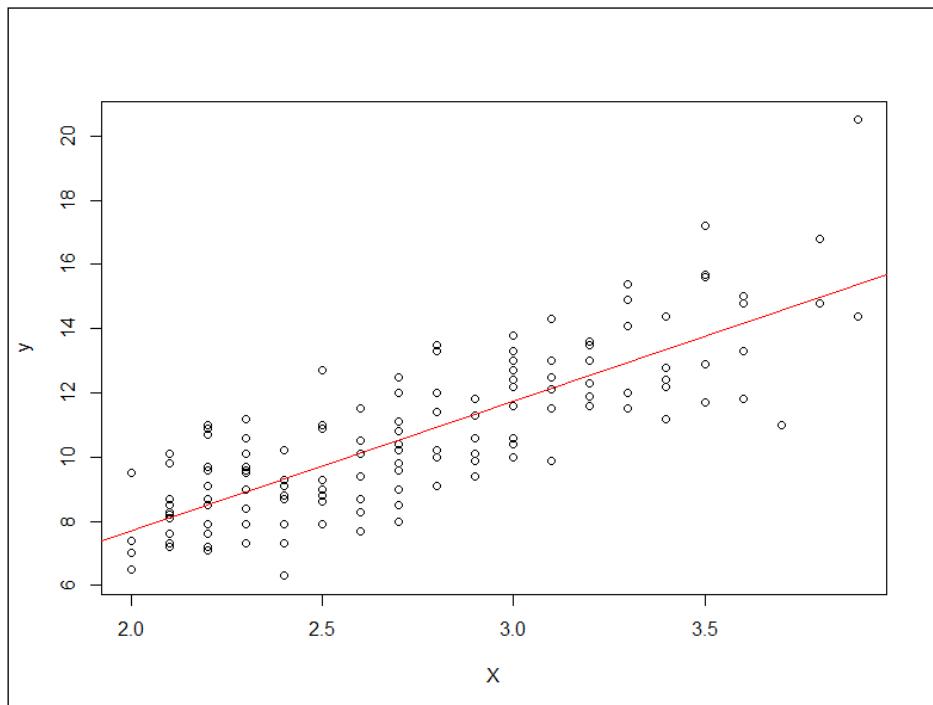
```

3. You can now make a regression plot with the given data points and model:

```

> plot(y~X)
> abline(model, col="red")

```



Linear regression plot of cats dataset

4. Load rmr2 into R:

```
> Sys.setenv(HADOOP_CMD="/usr/bin/hadoop")
> Sys.setenv(HADOOP_STREAMING="/usr/lib/hadoop-mapreduce/hadoop->
streaming-2.5.0-cdh5.2.0.jar")
> library(rmr2)
> rmr.options(backend="local")
```

5. You can then set up x and y values:

```
> X = matrix(cats$Bwt)
> X.index = to.dfs(cbind(1:nrow(X), X))
> y = as.matrix(cats$Hwt)
```

6. Make a Sum function to sum up the values:

```
> Sum =
+ function(., YY)
+ keyval(1, list(Reduce('+', YY)))
```

7. Compute XtX in MapReduce, Job1:

```
> XtX =
+ values(
+ from.dfs(
+ mapreduce(
+ input = X.index,
+ map =
+ function(., Xi) {
+ Xi = Xi[,-1]
+ keyval(1, list(t(Xi) %*% Xi))},
+ reduce = Sum,
+ combine = TRUE))))[[1]]
```

8. You can then compute  $XtY$  in MapReduce, Job2:

```
XtY =
+ values(
+ from.dfs(
+ mapreduce(
+ input = X.index,
+ map = function(., Xi) {
+ yi = y[Xi[,1],]
+ Xi = Xi[,-1]
+ keyval(1, list(t(Xi) %*% yi))},
+ reduce = Sum,
+ combine = TRUE))))[[1]]
```

9. Lastly, you can derive the coefficient from  $XtX$  and  $XtY$ :

```
> solve(XtX, XtY)
 [,1]
 [1,] 3.907113
```

## How it works...

In this recipe, we demonstrate how to implement linear logistic regression in a MapReduce fashion in R. Before we start the implementation, we review how traditional linear models work. We first retrieve the `cats` dataset from the `MASS` package. We then load `x` as the body weight (`Bwt`) and `y` as the heart weight (`Hwt`).

Next, we begin to fit the data into a linear regression model using the `lm` function. We can then compute the fitted model and obtain the summary of the model. The summary shows that the coefficient is 4.0341 and the intercept is -0.3567. Furthermore, we draw a scatter plot in accordance with the given data points and then draw a regression line on the plot.

As we cannot perform linear regression using the `lm` function in the MapReduce form, we have to rewrite the regression model in a MapReduce fashion. Here, we would like to implement a MapReduce version of linear regression in three steps, which are: calculate the  $XtX$  value with the MapReduce, job1, calculate the  $XtY$  value with MapReduce, job2, and then derive the coefficient value:

- ▶ In the first step, we pass the matrix, `x`, as the input to the `map` function. The `map` function then calculates the cross product of the transposed matrix, `x`, and, `x`. The `reduce` function then performs the sum operation defined in the previous section.

- ▶ In the second step, the procedure of calculating  $\mathbf{x}^T \mathbf{y}$  is similar to calculating  $\mathbf{x}^T \mathbf{x}$ . The procedure calculates the cross product of the transposed matrix,  $\mathbf{x}$ , and,  $\mathbf{y}$ . The `reduce` function then performs the sum operation.
- ▶ Lastly, we use the `solve` function to derive the coefficient, which is 3.907113.

As the results show, the coefficients computed by `lm` and MapReduce differ slightly. Generally speaking, the coefficient computed by the `lm` model is more accurate than the one calculated by MapReduce. However, if your data is too large to fit in the memory, you have no choice but to implement linear regression in the MapReduce version.

## See also

- ▶ You can access more information on machine learning algorithms at: <https://github.com/RevolutionAnalytics/rmr2/tree/master/pkg/tests>

# Configuring RHadoop clusters on Amazon EMR

Until now, we have only demonstrated how to run a RHadoop program in a single Hadoop node. In order to test our RHadoop program on a multi-node cluster, the only thing you need to do is to install RHadoop on all the task nodes (nodes with either task tracker for mapreduce version 1 or node manager for map reduce version 2) of Hadoop clusters. However, the deployment and installation is time consuming. On the other hand, you can choose to deploy your RHadoop program on Amazon EMR, so that you can deploy multi-node clusters and RHadoop on every task node in only a few minutes. In the following recipe, we will demonstrate how to configure RHadoop cluster on an Amazon EMR service.

## Getting ready

In this recipe, you must register and create an account on AWS, and you also must know how to generate a EC2 key-pair before using Amazon EMR.

For those who seek more information on how to start using AWS, please refer to the tutorial provided by Amazon at [http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2\\_GetStarted.html](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html).

# How to do it...

Perform the following steps to configure RHadoop on Amazon EMR:

1. First, you can access the console of the Amazon Web Service (refer to <https://us-west-2.console.aws.amazon.com/console/>) and find EMR in the analytics section. Then, click on **EMR**.

The screenshot shows the Amazon Web Services console with the 'Services' dropdown open. The 'Analytics' section is highlighted with a red box, containing the 'EMR Managed Hadoop Framework' service.

| Amazon Web Services                                                       |                                                                |                                                              |
|---------------------------------------------------------------------------|----------------------------------------------------------------|--------------------------------------------------------------|
| <b>Compute</b>                                                            | <b>Administration &amp; Security</b>                           | <b>Application Services</b>                                  |
| EC2 Virtual Servers in the Cloud                                          | Directory Service Managed Directories in the Cloud             | SQS Message Queue Service                                    |
| Lambda PREVIEW Run Code in Response to Events                             | Identity & Access Management Access Control and Key Management | SWF Workflow Service for Coordinating Application Components |
| <b>Storage &amp; Content Delivery</b>                                     | Trusted Advisor AWS Cloud Optimization Expert                  | AppStream Low Latency Application Streaming                  |
| S3 Scalable Storage in the Cloud                                          | CloudTrail User Activity and Change Tracking                   | Elastic Transcoder Easy-to-use Scalable Media Transcoding    |
| Storage Gateway Integrates On-Premises IT Environments with Cloud Storage | Config PREVIEW Resource Configurations and Inventory           | SES Email Sending Service                                    |
| Glacier Archive Storage in the Cloud                                      | CloudWatch Resource and Application Monitoring                 | CloudSearch Managed Search Service                           |
| CloudFront Global Content Delivery Network                                |                                                                |                                                              |
| <b>Database</b>                                                           | <b>Deployment &amp; Management</b>                             | <b>Mobile Services</b>                                       |
| RDS MySQL, Postgres, Oracle, SQL Server, and Amazon Aurora                | Elastic Beanstalk AWS Application Container                    | Cognito User Identity and App Data Synchronization           |
| DynamoDB Predictable and Scalable NoSQL Data Store                        | OpsWorks DevOps Application Management Service                 | Mobile Analytics Understand App Usage Data at Scale          |
| ElastiCache In-Memory Cache                                               | CloudFormation Templated AWS Resource Creation                 | SNS Push Notification Service                                |
| Redshift Managed Petabyte-Scale Data Warehouse Service                    | CodeDeploy Automated Deployments                               |                                                              |
| <b>Networking</b>                                                         | <b>Analytics</b>                                               | <b>Enterprise Applications</b>                               |
|                                                                           | EMR Managed Hadoop Framework                                   | WorkSpaces Desktops in the Cloud                             |
|                                                                           | Kinesis Real-time Processing of Streaming Big Data             | Zocalo Secure Enterprise Storage and Sharing Service         |

Access EMR service from AWS console.

2. You should find yourself in the cluster list of the EMR dashboard (refer to [https://us-west-2.console.aws.amazon.com/elasticmapreduce/home?region=us-west-2#cluster-list:::\)](https://us-west-2.console.aws.amazon.com/elasticmapreduce/home?region=us-west-2#cluster-list:::); click on **Create cluster**.

The screenshot shows the 'Cluster List' page for the Elastic MapReduce service. The 'Create cluster' button is highlighted with a red box.

| Name | ID | Status | Creation time (UTC+8) | Elapsed time | Normalized instance hours |
|------|----|--------|-----------------------|--------------|---------------------------|
|      |    |        |                       |              |                           |

Cluster list of EMR

3. Then, you should find yourself on the **Create Cluster** page (refer to [- 4. Next, you should specify \*\*Cluster name\*\* and \*\*Log folder S3 location\*\* in the cluster configuration.](https://us-west-2.console.aws.amazon.com/elasticmapreduce/home?region=us-west-2#create-cluster:)

Cluster Configuration

|                        |                                                                            |
|------------------------|----------------------------------------------------------------------------|
| Cluster name           | <input type="text" value="rhadoop"/>                                       |
| Termination protection | <input checked="" type="radio"/> Yes<br><input type="radio"/> No           |
| Logging                | <input checked="" type="checkbox"/> Enabled                                |
| Log folder S3 location | <input type="text" value="s3://rhadoop/"/><br>s3://<bucket-name>/<folder>/ |
| Debugging              | <input checked="" type="checkbox"/> Enabled                                |

Cluster configuration in the create cluster page

5. You can then configure the Hadoop distribution on **Software Configuration**.

Software Configuration

| Hadoop distribution          | <input checked="" type="radio"/> Amazon              | Use Amazon's Hadoop distribution. <a href="#">Learn more</a>                                                                                                                                                                                                  |
|------------------------------|------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AMI version                  | <input type="text" value="3.3.1"/>                   | Determines the base configuration of the instances in your cluster, including the Hadoop version. <a href="#">Learn more</a>                                                                                                                                  |
| MapR                         |                                                      | Use MapR's Hadoop distribution. <a href="#">Learn more</a>                                                                                                                                                                                                    |
| Applications to be installed | Version                                              |                                                                                                                                                                                                                                                               |
| Hive                         | 0.13.1                                               |    |
| Pig                          | 0.12.0                                               |    |
| Hue                          | 3.6.0                                                |    |
| Additional applications      | <input type="button" value="Select an application"/> |                                                                                                                                                                                                                                                               |
|                              | <input type="button" value="Configure and add"/>     |                                                                                                                                                                                                                                                               |

Configure the software and applications

6. Next, you can configure the number of nodes within the Hadoop cluster.

#### Hardware Configuration

Specify the networking and hardware configuration for your cluster. If you need more than 20 EC2 instances, complete this form. Request Spot instances (unused EC2 capacity) to save money.

| Type   | Name                    | EC2 instance type | Count | Request spot             | Bid price | ? |
|--------|-------------------------|-------------------|-------|--------------------------|-----------|---|
| Master | Master instance group - | m3.xlarge         | 1     | <input type="checkbox"/> |           |   |
| Core   | Core instance group - 2 | m1.large          | 2     | <input type="checkbox"/> |           |   |
| Task   | Task instance group - 3 | m1.medium         | 0     | <input type="checkbox"/> |           |   |

**Add task instance group**

Configure the hardware within Hadoop cluster

7. You can then specify the EC2 key-pair for the master node login.

#### Security and Access

|                 |                                                                                                  |                                                                                                                |
|-----------------|--------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| EC2 key pair    | ec2-startup                                                                                      | Use an existing EC2 key pair to SSH into the master node of the Amazon EMR cluster. <a href="#">Learn more</a> |
| IAM user access | <input checked="" type="radio"/> All other IAM users<br><input type="radio"/> No other IAM users | Control the visibility of this cluster to other IAM users. <a href="#">Learn more</a>                          |

Security and access to the master node of the EMR cluster

8. To set up RHadoop, one has to perform bootstrap actions to install RHadoop on every task node. Please write a file named `bootstrapRHadoop.sh`, and insert the following lines within the file:

```
echo 'install.packages(c("codetools", "Rcpp", "RJSONIO", "bitops",
"digest", "functional", "stringr", "plyr", "reshape2", "rJava",
"caTools"), repos="http://cran.us.r-project.org")' > /home/hadoop/
installPackage.R
sudo Rscript /home/hadoop/installPackage.R
wget --no-check-certificate https://raw.githubusercontent.com/
RevolutionAnalytics/rmr2/master/build/rmr2_3.3.0.tar.gz
sudo R CMD INSTALL rmr2_3.3.0.tar.gz
wget --no-check-certificate https://raw.githubusercontent.com/
RevolutionAnalytics/rhdfs/master/build/rhdfs_1.0.8.tar.gz
sudo HADOOP_CMD=/home/hadoop/bin/hadoop R CMD INSTALL
rhdfs_1.0.8.tar.gz
```

9. You should upload bootstrapRHadoop.sh to S3.
10. You now need to add the bootstrap action with Custom action, and add s3://<location>/bootstrapRHadoop.sh within the S3 location.

### Bootstrap Actions

**i** Bootstrap actions are scripts that are executed during setup before Hadoop starts on every cluster node. You can use them to install additional software and customize your applications. [Learn more](#)

| Bootstrap action type | Name          | S3 location                         | Optional arguments                                                                                                                                                     |
|-----------------------|---------------|-------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Custom action         | Custom action | s3://pccproject/bootstrapRHadoop.sh |   |
| Add bootstrap action  | Custom action |                                     |                                                                                       |

Set up the bootstrap action

11. Next, you can click on **Create cluster** to launch the Hadoop cluster.

### Steps

**i** A step is a unit of work you submit to the cluster. A step might contain one or more Hadoop jobs, or contain instructions to install or configure an application. You can submit up to 256 steps to a cluster. [Learn more](#)

| Name     | Action on failure                                                                 | JAR location | Arguments |
|----------|-----------------------------------------------------------------------------------|--------------|-----------|
| Add step | Select a step                                                                     |              |           |
|          |  |              |           |

Auto-terminate  Yes      Automatically terminate cluster after the last step is completed.  
 No      Keep cluster running until you terminate it.

[Cancel](#) Create cluster

Create the cluster

12. Lastly, you should see the master public DNS when the cluster is ready. You can now access the terminal of the master node with your EC2-key pair:

The screenshot shows the AWS Elastic MapReduce Cluster Details page. At the top, it says "Cluster: rhadoop Waiting Waiting after step completed". Below that, under "Connections", it lists "Master public DNS: ec2-54-68-153-263.us-west-2.compute.amazonaws.com SSH". The "Tags" section has a link to "View All / Edit". The "Summary" section includes the cluster ID "j-2CL38XT296NL2", creation date "2014-12-17 19:41 (UTC+8)", elapsed time "18 minutes", and auto-terminate status "No". The "Termination On Change protection:" field is set to "Change". The "Configuration Details" section shows AMI version "3.3.1", distribution "Hadoop Amazon 2.4.0", applications "Hive 0.13.1, Pig 0.12.0, Hue", log URI "s3://hadoop/", and EMRFS status "Disabled". It also shows "consistent view". The "Security/Network" section includes "Availability us-west-2a", "zone: --", "Subnet ID: subnet-7d04f818", "Key name: DSP-EC2", "EC2 instance -- profile: --", "EMR role: --", and "Visible to all All Change users: --". The "Hardware" section shows "Master: Running 1 m3.xlarge", "Core: --", and "Task: --".

A screenshot of the created cluster

## How it works...

In this recipe, we demonstrate how to set up RHadoop on Amazon EMR. The benefit of this is that you can quickly create a scalable, on demand Hadoop with just a few clicks within a few minutes. This helps save you time from building and deploying a Hadoop application. However, you have to pay for the number of running hours for each instance. Before using Amazon EMR, you should create an AWS account and know how to set up the EC2 key-pair and the S3. You can then start installing RHadoop on Amazon EMR.

In the first step, access the EMR cluster list and click on **Create cluster**. You can see a list of configurations on the **Create cluster** page. You should then set up the cluster name and log folder in the S3 location in the cluster configuration.

Next, you can set up the software configuration and choose the Hadoop distribution you would like to install. Amazon provides both its own distribution and the MapR distribution. Normally, you would skip this section unless you have concerns about the default Hadoop distribution.

You can then configure the hardware by specifying the master, core, and task node. By default, there is only one master node, and two core nodes. You can add more core and task nodes if you like. You should then set up the key-pair to login to the master node.

You should next make a file containing all the start scripts named `bootstrapRHadoop.sh`. After the file is created, you should save the file in the S3 storage. You can then specify custom action in **Bootstrap Action** with `bootstrapRHadoop.sh` as the Bootstrap script. Lastly, you can click on **Create cluster** and wait until the cluster is ready. Once the cluster is ready, one can see the master public DNS and can use the EC2 key-pair to access the terminal of the master node.

Beware! Terminate the running instance if you do not want to continue using the EMR service. Otherwise, you will be charged per instance for every hour you use.

## See also

- ▶ Google also provides its own cloud solution, the Google compute engine. For those who would like to know more, please refer to <https://cloud.google.com/compute/>.

# A

## Resources for R and Machine Learning

The following table lists all the resources for R and machine learning:

| R introduction                                                  |                                                                                                                                                                         |                                                  |
|-----------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------|
| Title                                                           | Link                                                                                                                                                                    | Author                                           |
| R in Action                                                     | <a href="http://www.amazon.com/R-Action-Robert-Kabacoff/dp/1935182390">http://www.amazon.com/R-Action-Robert-Kabacoff/dp/1935182390</a>                                 | Robert Kabacoff                                  |
| The Art of R Programming: A Tour of Statistical Software Design | <a href="http://www.amazon.com/The-Art-Programming-Statistical-Software/dp/1593273843">http://www.amazon.com/The-Art-Programming-Statistical-Software/dp/1593273843</a> | Norman Matloff                                   |
| An Introduction to R                                            | <a href="http://cran.r-project.org/doc/manuals/R-intro.pdf">http://cran.r-project.org/doc/manuals/R-intro.pdf</a>                                                       | W. N. Venables, D. M. Smith, and the R Core Team |
| Quick-R                                                         | <a href="http://www.statmethods.net/">http://www.statmethods.net/</a>                                                                                                   | Robert I. Kabacoff, PhD                          |
| Online courses                                                  |                                                                                                                                                                         |                                                  |
| Title                                                           | Link                                                                                                                                                                    | Instructor                                       |
| Computing for Data Analysis (with R)                            | <a href="https://www.coursera.org/course/compdata">https://www.coursera.org/course/compdata</a>                                                                         | Roger D. Peng, Johns Hopkins University          |
| Data Analysis                                                   | <a href="https://www.coursera.org/course/dataanalysis">https://www.coursera.org/course/dataanalysis</a>                                                                 | Jeff Leek, Johns Hopkins University              |
| Data Analysis and Statistical Inference                         | <a href="https://www.coursera.org/course/statistics">https://www.coursera.org/course/statistics</a>                                                                     | Mine Çetinkaya-Rundel, Duke University           |

## Machine learning

| Title                        | Link                                                                                                                            | Author                           |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------|----------------------------------|
| Machine Learning for Hackers | <a href="http://www.amazon.com/dp/1449303714?tag=inspiredalgor-20">http://www.amazon.com/dp/1449303714?tag=inspiredalgor-20</a> | Drew Conway and John Myles White |
| Machine Learning with R      | <a href="http://www.packtpub.com/machine-learning-with-r/book">http://www.packtpub.com/machine-learning-with-r/book</a>         | Brett Lantz                      |

## Online blog

| Title         | Link                                                                      |
|---------------|---------------------------------------------------------------------------|
| R-bloggers    | <a href="http://www.r-bloggers.com/">http://www.r-bloggers.com/</a>       |
| The R Journal | <a href="http://journal.r-project.org/">http://journal.r-project.org/</a> |

## CRAN task view

| Title                                                     | Link                                                                                                                            |
|-----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| CRAN Task View: Machine Learning and Statistical Learning | <a href="http://cran.r-project.org/web/views/MachineLearning.html">http://cran.r-project.org/web/views/MachineLearning.html</a> |

# B

## **Dataset – Survival of Passengers on the Titanic**

Before the exploration process, we would like to introduce the example adopted here. It is the demographic information on passengers aboard the RMS Titanic, provided by Kaggle (<https://www.kaggle.com/>, a platform for data prediction competitions). The result we are examining is whether passengers on board would survive the shipwreck or not.

There are two reasons to apply this dataset:

- ▶ RMS Titanic is considered as the most infamous shipwreck in history, with a death toll of up to 1,502 out of 2,224 passengers and crew. However, after the ship sank, the passengers' chance of survival was not by chance only; actually, the cabin class, sex, age, and other factors might also have affected their chance of survival.
- ▶ The dataset is relatively simple; you do not need to spend most of your time on data munging (except when dealing with some missing values), but you can focus on the application of exploratory analysis.

The following chart is the variables' descriptions of the target dataset:

Variable descriptions:

|                       |                                                                         |
|-----------------------|-------------------------------------------------------------------------|
| <code>survival</code> | Survival<br>(0 = No; 1 = Yes)                                           |
| <code>pclass</code>   | Passenger class<br>(1 = 1st; 2 = 2nd; 3 = 3rd)                          |
| <code>name</code>     | Name                                                                    |
| <code>sex</code>      | Sex                                                                     |
| <code>age</code>      | Age                                                                     |
| <code>sibsp</code>    | Number of siblings/spouses aboard                                       |
| <code>parch</code>    | Number of parents/children aboard                                       |
| <code>ticket</code>   | Ticket number                                                           |
| <code>fare</code>     | Passenger fare                                                          |
| <code>cabin</code>    | Cabin                                                                   |
| <code>embarked</code> | Port of embarkation<br>(C = Cherbourg; Q = Queenstown; S = Southampton) |

Special notes:

`Pclass` is a proxy for **socio-economic status (SES)**

1st ~ Upper; 2nd ~ Middle; 3rd ~ Lower

Age is in years; it is fractional if the age is less than one (1), and if the age is estimated, it is in the form, `xx . xx`.

With respect to the family relation variables (that is, `sibsp` and `parch`), some relations were ignored. The following are the definitions used for `sibsp` and `parch`:

**Sibling:** Brother, sister, stepbrother, or stepsister of a passenger aboard the Titanic

**Spouse:** Husband or wife of a passenger aboard the Titanic (mistresses and fiancés are ignored)

**Parent:** Mother or father of a passenger aboard the Titanic

**Child:** Son, daughter, stepson, or stepdaughter of a passenger aboard the Titanic

Other family relatives excluded from this study include, cousins, nephews/nieces, aunts/uncles, and in-laws. Some children travelled only with a nanny, therefore, `parch=0` for them. Also, some travelled with very close friends or neighbors from the same village; however, the definitions do not support such relations.

Judging from the description of the variables, one might have some questions in mind, such as, "Are there any missing values in this dataset?", "What was the average age of the passengers on the Titanic?", "What proportion of the passengers survived the disaster?", "What social class did most passengers on board belong to?". All these questions presented here will be answered in *Chapter, Data Exploration with RMS Titanic*.

Beyond questions relating to descriptive statistics, the eventual object of *Chapter, Data Exploration with RMS Titanic*, is to generate a model to predict the chance of survival given by the input parameters. In addition to this, we will assess the performance of the generated model to determine whether the model is suited for the problem.

# Bibliography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *R Data Analysis Cookbook, Viswa Viswanathan and Shanthi Viswanathan*
- *R Data Visualization Cookbook, Atmajitsinh Gohil*
- *Machine Learning with R Cookbook, Yu-Wei Chiu (David Chiu)*