Quick answers to common problems

R Data Visualization Cookbook

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

Atmajitsinh Gohil

[PACKT] PUBLISHING open source* community experience distilled

R Data Visualization Cookbook

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

Atmajitsinh Gohil

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

R Data Visualization Cookbook

First published: January 2015

Production reference: 1240115

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

ISBN 978-1-78398-950-8

`www.packtpub.com`

Credits

Author
Atmajitsinh Gohil

Reviewers
Sharan Kumar Ravindran
Kannan Kalidasan
Erik M. Rodríguez Pacheco
Arun Padmanabhan
Juan Pablo Zamora
Patric Zhao

Commissioning Editor
Kartikey Pandey

Acquisition Editor
Neha Nagwekar

Content Development Editor
Arun Nadar

Technical Editors
Rohit Kumar Singh
Mitali Somaiya

Copy Editors
Nithya P
Shambhavi Pai
Rashmi Sawant

Project Coordinator
Neha Bhatnagar

Proofreaders
Simran Bhogal
Stephen Copestake
Paul Hindle
Joanna McMahon

Indexer
Priya Sane

Graphics
Disha Haria
Abhinash Sahu

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Atmajitsinh Gohil works as a senior consultant at a consultancy firm in New York City. After graduating, he worked in the financial industry as a Fixed Income Analyst. He writes about data manipulation, data exploration, visualization, and basic R plotting functions on his blog at `http://datavisualizationineconomics.blogspot.com`.

He has a master's degree in financial economics from the State University of New York (SUNY), Buffalo. He also graduated with a master of arts degree in economics from University of Pune, India. He loves to read blogs on data visualization and loves to go out on hikes in his free time.

This book would not have been possible without the help from numerous data visualizers and data scientists around the globe who bring into existence new and innovative ways to transform data into beautiful stories. I would like to sincerely thank the developers of R and R packages who have contributed so generously to the growing R open source community.

I would like to thank Jer Thorpe and Hans Rosling for their inspiring Ted videos on data visualization.

I would also like to thank all the economists and statisticians who have so often inspired me.

I would like to thank my publisher, Packt Publishing, for giving me the opportunity to work on this book. I would also like to thank all the technical reviewers and content development editors at Packt Publishing for their informative comments and suggestions.

Finally, I would like to thank my amazing family and magnificent friends for always encouraging and supporting me.

About the Reviewers

Sharan Kumar Ravindran is a lead data scientist in the fastest growing big data start-up based in Bangalore. His primary interests lie in statistics and machine learning. He has over 4 years of experience and has worked in the domains of e-commerce and IoT.

He has solved several problems on Kaggle and is among the top 10 percent of experts on Kaggle. His blog and social profiles can be found at `www.rsharankumar.com`.

He works for Flutura, which is ranked among the top 20 most promising big data companies across the globe by the leading analyst magazine, *CIO Review*. Flutura also featured on Gigaom reports on big data and M2M in the energy sector. Flutura was also the winner at TechSparks, where 800 innovative start-ups were evaluated.

Kannan Kalidasan is a software developer by profession, an autodidact, and open source evangelist. He has a decade's experience in database computing, data management, open source, and distributed computing. He holds a bachelor's of technology degree in computer science from Pondicherry University. He has played different roles in his career, such as a developer, an architect, a team lead, and a DBA. He currently holds the position of a BI Engineer at Orbitz Worldwide.

He started his own start-up back in 2005 on a part-time basis during his college days, worked with other companies in different open source projects, and provided training. He is passionate about technology and an entrepreneur at heart, and he likes to mentor fellow enthusiasts. His inherent curiosity keeps him occupied with learning new technologies and trying new things. He always believes that "our dreams can be delayed but will never fail if we work hard."

He blogs at `www.kannandreams.wordpress.com` and you can follow him on Twitter at `@kannanpoem`. He loves to take long walks alone, write Tamil poems, paint, and read books.

A big thank you to all who believed in me and supported me. I would like to thank my strong soul for pushing me to achieve my dreams. I would like to express my deepest gratitude to Packt Publishing for giving me this opportunity.

Erik M. Rodríguez Pacheco works as a manager in the Business Intelligence Unit at Banco Improsa in San José, Costa Rica. He has 11 years of experience in the financial industry. He is currently a professor of the Business Intelligence Specialization Program at the Instituto Tecnológico de Costa Rica's Continuing Education Program. Erik is an enthusiast of new technologies, particularly those related to business intelligence, data mining, and data science. He holds a Bachelor's degree in business administration from Universidad de Costa Rica, and has specialized in business intelligence from the Instituto Tecnológico de Costa Rica, data mining from Promidat (Programa Iberoamericano de Formación en Minería de Datos), and business intelligence and data mining from Universidad del Bosque, Colombia. He is currently enrolled in a specialization program in data science from Johns Hopkins University. He can be reached at `cr.linkedin.com/in/erikrodriguezp/`.

Arun Padmanabhan has about 4 years of experience in developing products including mobile, enterprise, statistical, and data mining applications. He graduated with a master's degree in computer applications in 2010. Currently, he is a data scientist at Flutura Decision Science and Analytics, where he is working at saving the world, one data product at a time.

Juan Pablo Zamora holds a bachelor's degree in statistics from the University of Costa Rica (UCR) in 2007. He is currently working on his dissertation in the field of predictive analytics and will obtain an MSc degree in statistics from the University of Costa Rica.

He enjoys teaching and was a tutor of statistics courses at the Business School of UNED of Costa Rica during 2010-2012. He also mentored others in the areas of data processing and analytics as well as the use of statistical analysis tools, to name a few.

Juan has over 7 years of experience in the banking industry, primarily in the credit card business for Central America and Mexico. He began as an analyst, eventually becoming the leader of a team of analysts for Central America's largest credit card issuer and acquirer. During this period, he participated in several predictive analytics projects related to credit risk, account retention, and profitability.

Juan recently joined a large multinational company in the retail sector with the task of building an analytics program to identify and prevent high-risk issues and/or threats to the business in Latin America.

His current interests are R, data visualization, business intelligence, predictive modeling, and big data. He can be reached at `cr.linkedin.com/in/datasciencezamora` or `data.science.zamora@gmail.com`.

Patric Zhao is a senior GPU architect in the High Performance Computing (HPC) field at Nvidia. He has experience in developing scientific and engineering applications and focuses on parallelization, performance modeling, and architecture-specific tuning. Patric is currently working on big data and machine learning areas, including regression, neural network, recommending system design, and implementation in CPU and GPU architectures. Patric has also contributed to accelerate R's applications by CUDA in the GPU ecosystem. You can find related articles on Nvidia's blog at `http://devblogs.nvidia.com/parallelforall/author/patricz/` or write to him at `patric.zhao@gmail.com`.

I would like to really thank my wife Yan Li J for always supporting and encouraging me.

Table of Contents

Preface 1

Chapter 1: A Simple Guide to R 7

Installing packages and getting help in R 8
Data types in R 10
Special values in R 11
Matrices in R 13
Editing a matrix in R 14
Data frames in R 15
Editing a data frame in R 15
Importing data in R 16
Exporting data in R 17
Writing a function in R 18
Writing if else statements in R 19
Basic loops in R 20
Nested loops in R 20
The apply, lapply, sapply, and tapply functions 21
Using par to beautify a plot in R 22
Saving plots 23

Chapter 2: Basic and Interactive Plots 25

Introduction 26
Introducing a scatter plot 26
Scatter plots with texts, labels, and lines 29
Connecting points in a scatter plot 32
Generating an interactive scatter plot 35
A simple bar plot 38
An interactive bar plot 40
A simple line plot 44
Line plot to tell an effective story 46

Generating an interactive Gantt/timeline chart in R 49
Merging histograms 50
Making an interactive bubble plot 53
Constructing a waterfall plot in R 55

Chapter 3: Heat Maps and Dendrograms 57
Introduction 57
Constructing a simple dendrogram 58
Creating dendrograms with colors and labels 62
Creating a heat map 64
Generating a heat map with customized colors 68
Generating an integrated dendrogram and a heat map 70
Creating a three-dimensional heat map and a stereo map 73
Constructing a tree map in R 75

Chapter 4: Maps 79
Introduction 79
Introducing regional maps 80
Introducing choropleth maps 82
A guide to contour maps 85
Constructing maps with bubbles 88
Integrating text with maps 92
Introducing shapefiles 94
Creating cartograms 98

Chapter 5: The Pie Chart and Its Alternatives 101
Introduction 101
Generating a simple pie chart 102
Constructing pie charts with labels 106
Creating donut plots and interactive plots 109
Generating a slope chart 112
Constructing a fan plot 115

Chapter 6: Adding the Third Dimension 117
Introduction 117
Constructing a 3D scatter plot 118
Generating a 3D scatter plot with text 121
A simple 3D pie chart 124
A simple 3D histogram 126
Generating a 3D contour plot 128
Integrating a 3D contour and a surface plot 130
Animating a 3D surface plot 133

Chapter 7: Data in Higher Dimensions 137
Introduction 137
Constructing a sunflower plot 138
Creating a hexbin plot 140
Generating interactive calendar maps 142
Creating Chernoff faces in R 145
Constructing a coxcomb plot in R 147
Constructing network plots 149
Constructing a radial plot 151
Generating a very basic pyramid plot 154
Chapter 8: Visualizing Continuous Data 157
Introduction 157
Generating a candlestick plot 158
Generating interactive candlestick plots 161
Generating a decomposed time series 163
Plotting a regression line 166
Constructing a box and whiskers plot 168
Generating a violin plot 170
Generating a quantile-quantile plot (QQ plot) 171
Generating a density plot 173
Generating a simple correlation plot 176
Chapter 9: Visualizing Text and XKCD-style Plots 181
Introduction 181
Generating a word cloud 182
Constructing a word cloud from a document 185
Generating a comparison cloud 188
Constructing a correlation plot and a phrase tree 191
Generating plots with custom fonts 194
Generating an XKCD-style plot 196
Chapter 10: Creating Applications in R 199
Introduction 199
Creating animated plots in R 200
Creating a presentation in R 202
A basic introduction to API and XML 205
Constructing a bar plot using XML in R 209
Creating a very simple shiny app in R 212
Index 215

Preface

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"*. In reality, the term visualization goes beyond the limits of providing visual images by assisting humans in data recording, revealing pattern, exploration of data, and spreading information in a meaningful way.

Jer Thorpe in an interview with Mashable.com (`http://mashable.com/2012/12/11/data-visualization-jer-thorp/`) introduces the idea of humanizing data:

> *"...And I think that there's a huge possibility for humans, society as a whole—if we could share that data more usefully, for science and for the construction of cities, and for all these kinds of things, then it becomes much more useful. So in my work, I'm really thinking about how we can give people glimpses into that type of future. Giving people an opportunity to think about data ownership or giving people a visualization so that they can see the kinds of things that can be done with data".*

R is an open source platform used to analyze data. It has been widely used as a statistical tool in the past. An individual does not necessarily have to be a programmer to use R. A beginner can use basic R functionalities to manipulate and extract data and create very simple and quick visualizations using the basic graphic tools. An intermediate R user can implement interactive visualizations, perform predictive modeling, or even create animated applications using packages developed by the R community. R will present you with the tools you need to process, manipulate, and communicate with your data, and it is not just limited to statistical analysis.

In this book, 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 book 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.

What this book covers

Chapter 1, *A Simple Guide to R*, is a quick tutorial on getting started with R. You will learn how to install packages, access help on R, construct and edit matrices, create and manipulate data frames, and write and save plots.

Chapter 2, *Basic and Interactive Plots*, introduces some of the basic R plots, such as scatter, line, and bar charts. We will also discuss the basic elements of interactive plots using the googleVis package in R. This chapter is a great resource for understanding the basic R plotting techniques.

Chapter 3, *Heat Maps and Dendrograms*, starts with a simple introduction to dendrograms and further introduces the concept of clustering techniques. The second half of this chapter discusses heat maps and integrating heat maps with dendrograms to get a more complete picture.

Chapter 4, *Maps*, discusses the importance of spatial data and various techniques used to visualize geographic data in R. You will learn how to generate static as well as interactive maps in R. The chapter discusses the topic of shape files and how to use them to generate a cartogram.

Chapter 5, *The Pie Chart and Its Alternatives*, is a detailed discussion on how to generate pie charts in R. You will also learn about the various criticisms of pie charts and how the pie chart is transformed to overcome them. The chapter also provides you with various alternatives used by data scientists and visualization artists to overcome the limitation of a pie chart.

Chapter 6, *Adding the Third Dimension*, dives into constructing 3D plots. This chapter also introduces packages such as rgl and animation, which are used to create interactive 3D plots.

Chapter 7, *Data in Higher Dimensions*, demonstrates the use of visualizations that are used to display data in higher dimension. You will learn the techniques to generate sunflower plots, hexbin plots, Chernoff faces, and so on. This chapter also discusses the usefulness of network, radial, and coxcomb plots, which have been widely used in news.

Chapter 8, *Visualizing Continuous Data*, illustrates the use of visualizations to display time series data. The chapter also discusses some general concepts related to visualizing correlations, the shape of the distribution, and detection of outliers using box and whisker plots.

Chapter 9, *Visualizing Text and XKCD-style Plots*, illustrates the use of text in creating effective visualizations. This chapter focuses mainly on techniques to create word clouds, phase tree, and comparison clouds in R. You will also learn how to use the XKCD package to introduce humor in visualizations.

Chapter 10, *Creating Applications in R*, shows you the techniques to create presentations and R markdown documents for publishing on a blog or a website. The chapter further discusses the XML package used to extract and visualize data as well as using shiny package used to create interactive applications.

What you need for this book

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.

Who this book is for

Having studied economics, I am not a software programmer myself and have written this book for readers new to R and visualization. This book does not delvento complex R code or complex data manipulating techniques, and it is written keeping in mind new and intermediate R users interested in learning about data visualization and data exploration techniques.

The book aims at teaching you the implementation of interactive and animated data visualizations and not just the basic R techniques. However, I have introduced some basic functionalities in *Chapter 1*, *A Simple Guide to R* and *Chapter 2*, *Basic and Interactive Plots*.

Wherever possible, I have provided references to websites, blogs, and journals, which can be explored to learn more about specific functions, graphics, animations, or even basic functionalities in R.

Sections

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

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

Getting ready

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

How to do it...

This section contains the steps required to follow the recipe.

How it works...

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

There's more...

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

See also

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

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We have used the `png()` function to save the plot as a PNG."

Any command line code is written as:

```
k = matrix((1:4),2,2)
l = matrix((5:10),2,3)
dim(k)
dim(l)
```

In R it is a general practice to use `<-` for assignment instead of the = sign. In all the recipes, I have followed the = sign for assignment. You should note that if you refer to blogs or websites related to R, you may encounter the `<-` sign in the code files.

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In order to write a simple function in R, we must first open a new R script by navigating to **File** | **New file**."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

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

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

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

Customer support

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

Downloading the example code

You can download the example code files for 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.

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 would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

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

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

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

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

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.

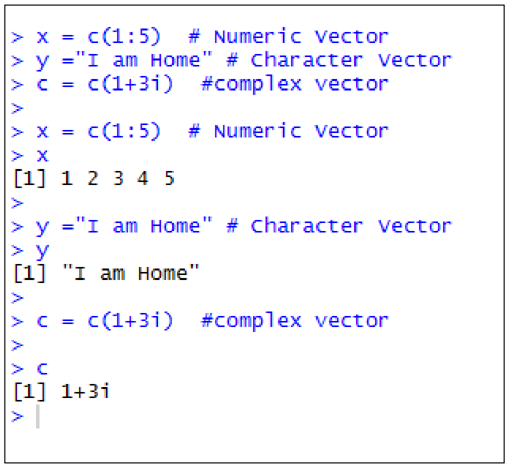
If we are still unable to understand the function or its use and implementation, we could go to Google and type the question or use the Stack Overflow website. I am always able to resolve my errors by searching on the Internet. Remember, every problem has a solution, and the possibilities with R are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

See also

- Flowing Data (`http://flowingdata.com/`): This is a good resource to learn visualization tools and R. The tutorials are based on an annual subscription.
- Stack Overflow (`http://stackoverflow.com/`): This is a great place to get help regarding R functions.
- Inside-R (`http://www.inside-r.org/`): This lists all the packages along with a small description.
- Rblogger (`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/`).
- R journal (`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:

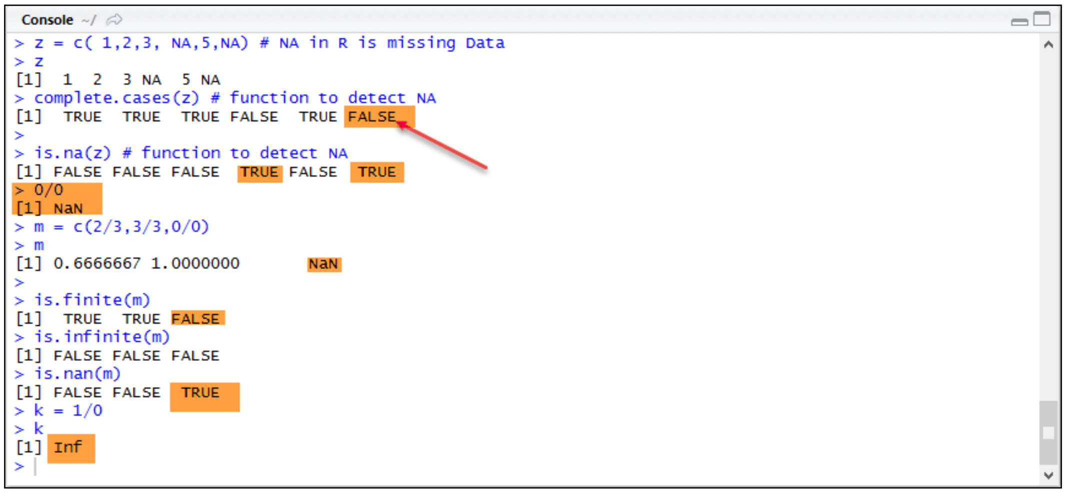
```
c = c(1+3i) #complex vector
```

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.



```
> 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:

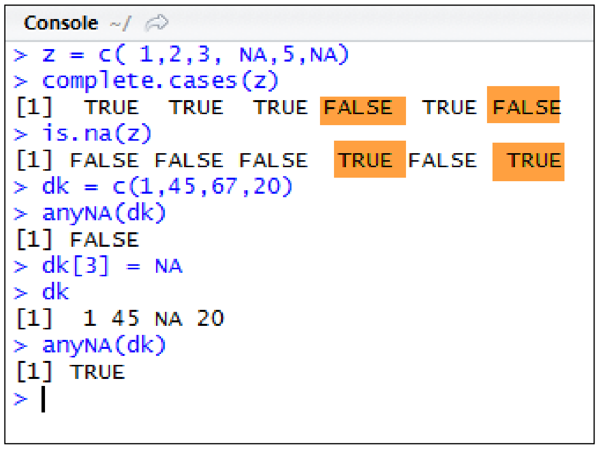
```
## infinite
k = 1/0
```

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.

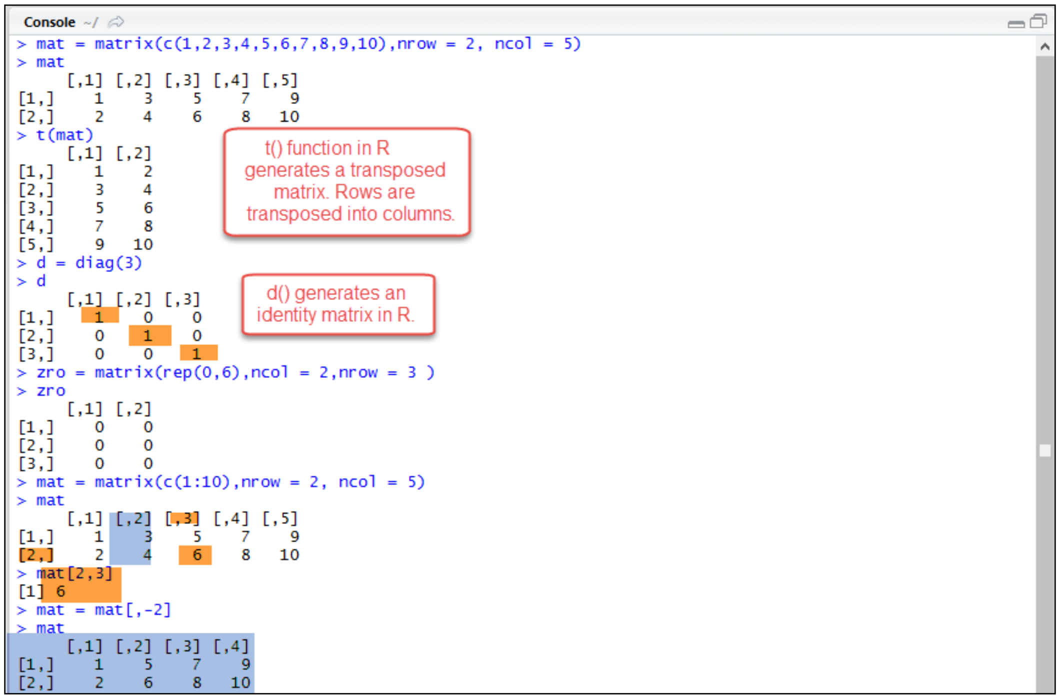


```
> z = c( 1,2,3, NA,5,NA)
> complete.cases(z)
[1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE
> is.na(z)
[1] FALSE FALSE FALSE  TRUE FALSE  TRUE
> dk = c(1,45,67,20)
> anyNA(dk)
[1] FALSE
> dk[3] = NA
> dk
[1]  1 45 NA 20
> anyNA(dk)
[1] TRUE
> 
```

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.



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

Console ~/
t() function in R generates a transposed matrix. Rows are transposed into columns.
d() generates an identity matrix in R.

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 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

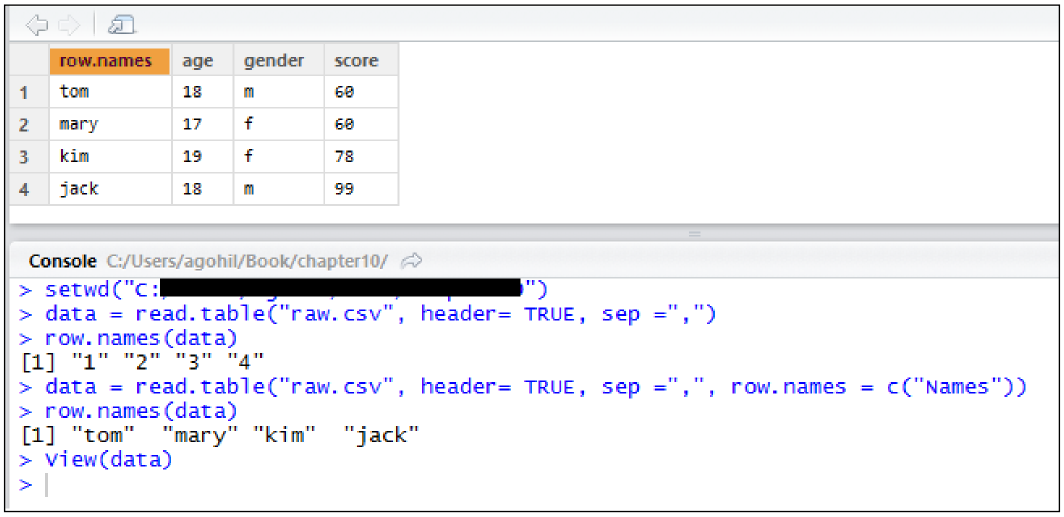
Data comes in various formats. Most of the data available online can be downloaded in the form of text documents (`.txt` extension) or as comma-separated values (`.csv`). We also encounter data in the tab-delimited format, XLS, HTML, JSON, XML, and so on. If you are interested in working with data, either in JSON or XML, refer to the recipe *Constructing a bar plot using XML in R* in *Chapter 10*, *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:



	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

Console C:/Users/agohil/Book/chapter10/

```
> setwd("C:[redacted]")
> 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
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.

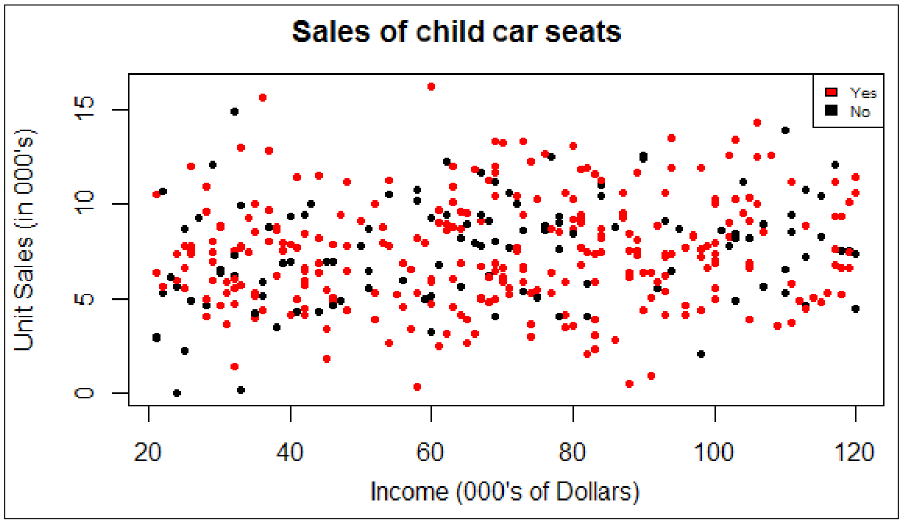
The `googleVis` package is developed by Google and it uses the Google Chart API to create interactive plots. There are a range of plots available with the `googleVis` package and this provides us with an advantage to plot the same data on various plots and select the one that delivers an effective message. 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 1, 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.

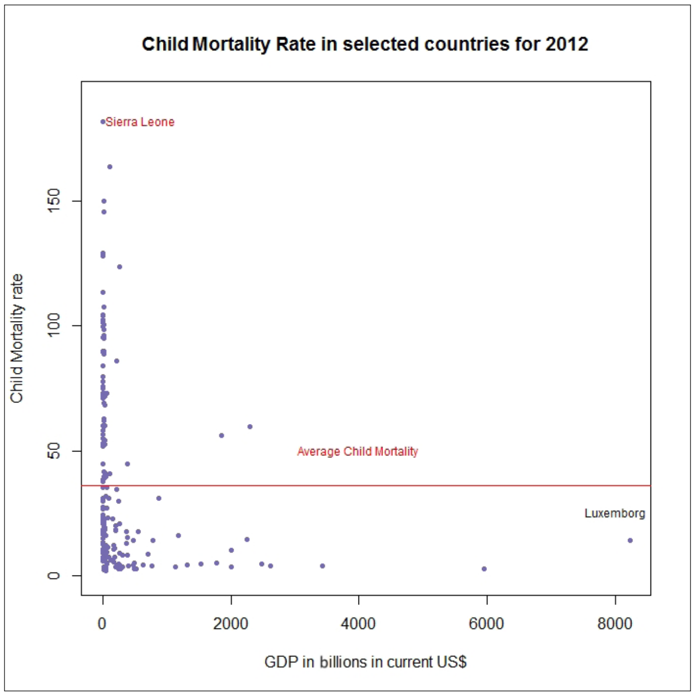
The data can be plotted in R by calling the `plot()` function. The first two arguments in the `plot()` function refer to the data to be displayed on the x-axis (Income) and y-axis (Sales). The `col` argument allows us to assign color to our data points. In this case, we would like to use a qualitative 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 1*, *A Simple Guide to R*.

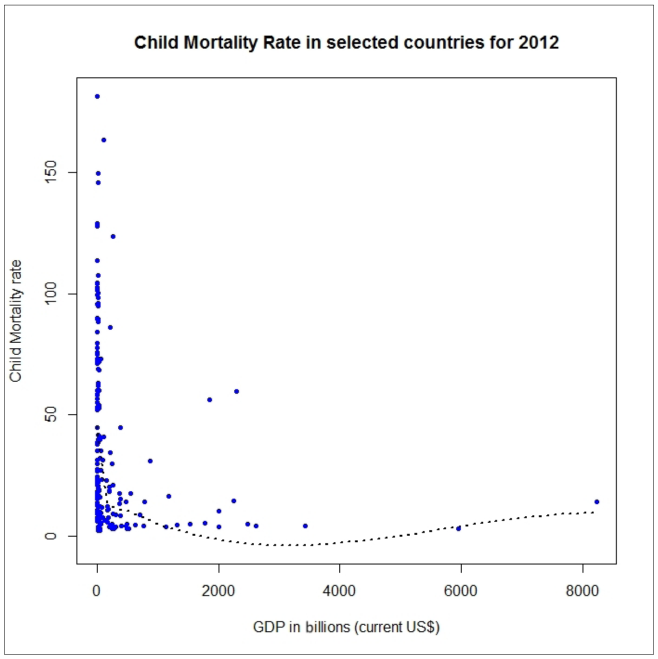
Under the `plot()` function, we have introduced the `$` notation. The name before the `$` sign corresponds to the data and the name after the `$` sign refers to the column (`child$gdp_bil`). We have used `ylim()` to specify the *y* limit for the plot. We can also assign a hex value to the color instead 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$gdp_bil, child$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.

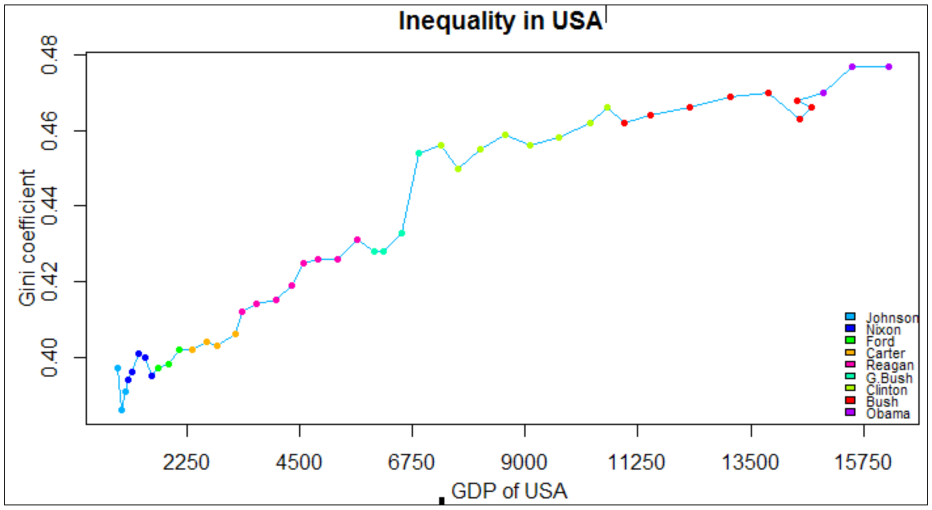
See also

- *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`.
- *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.

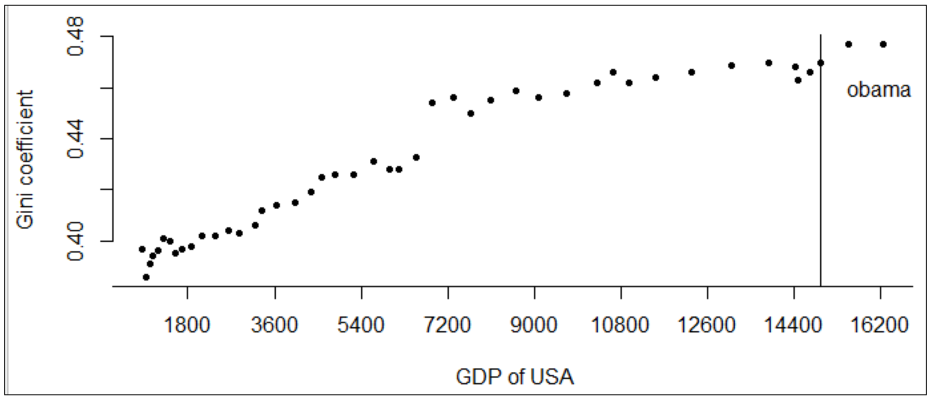
The important point to note is that R used a qualitative variable such as presidents to plot points of different colors. We would require the color names to pass as an argument under the `fill`. R converts the presidents' names into numeric value and uses it to color each point. We can view these numeric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

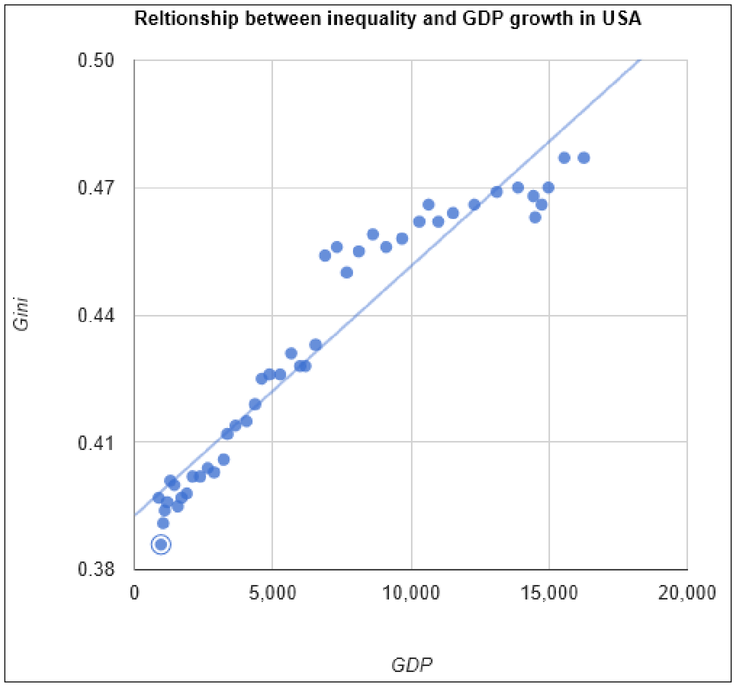
Since we require to plot all the years, we pass `income$years` as our argument for labels. The `pos` argument is used to adjust the position of the label around the point. Readers may observe overlapping labels and they can try to fix this by setting `pos` and `offset`. I would suggest the readers to 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 = "Reltionship 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...

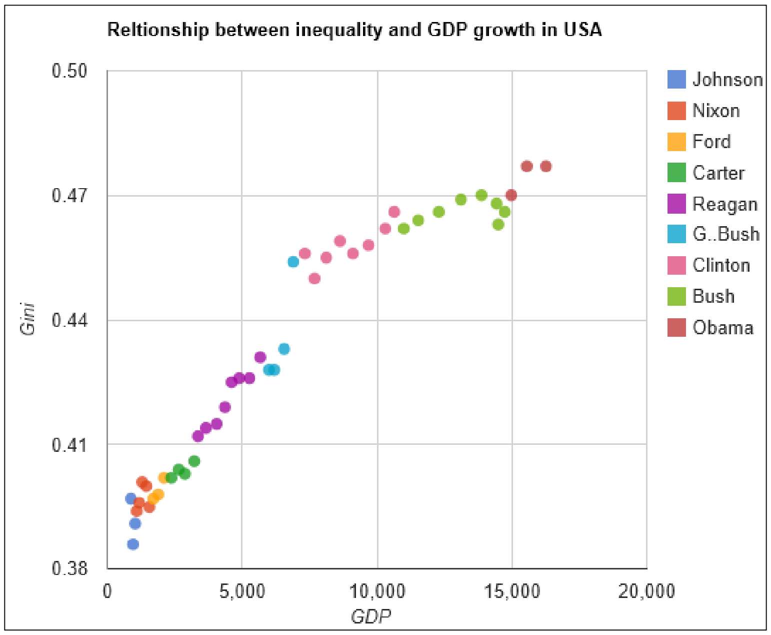
The `data.frame()` function is discussed in the recipe *Data frames in R* in *Chapter 1*, *A Simple Guide to R*. Readers new to R can learn about the `data.frame()` function by typing `?data.frame` in the R console window.

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.

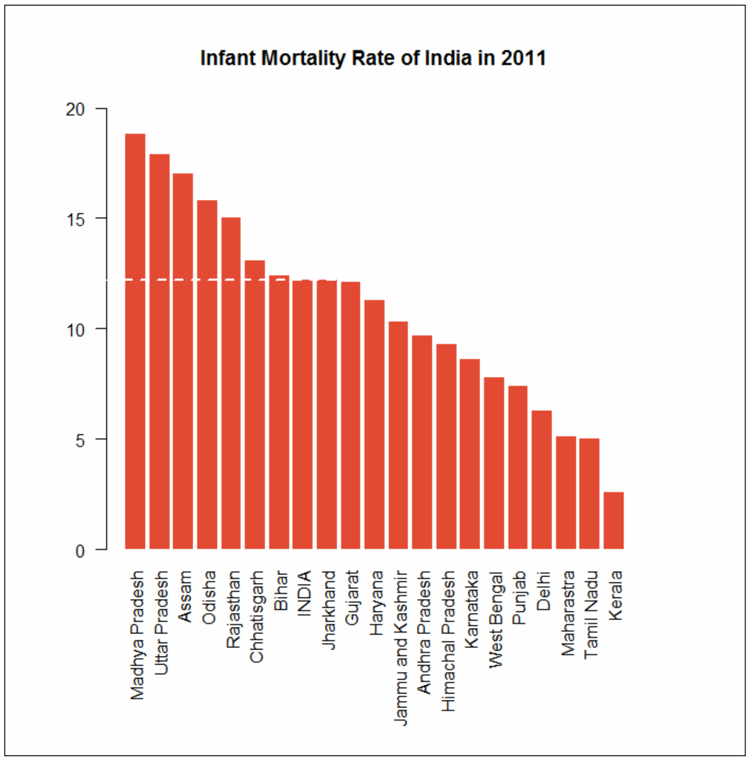
See also

- The `googleVis` developer website can be accessed at `https://developers.google.com/chart/interactive/docs/gallery/scatterchart#Configuration_Options`

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`

An interactive bar plot

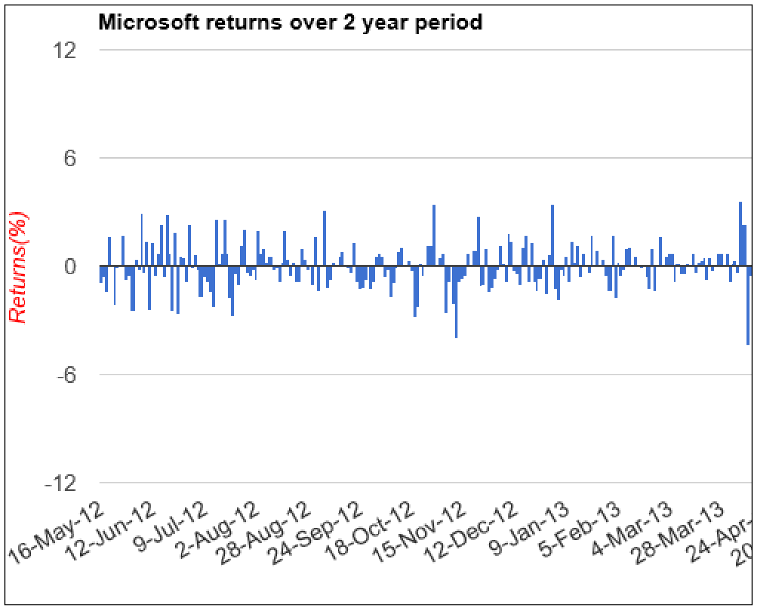
We would like to make the bar plot interactive. The advantage of using the Google Chart API in R is the flexibility it provides in making interactive plots. The `googleVis` package allows us to skip the step to export a plot from R to an illustrator and we can make presentable plots right out of R.

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:

$$\text{Return} = ((\text{Price}_t - \text{Price}_t\text{-1})/\text{price}_{t-1}) * 100$$

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.

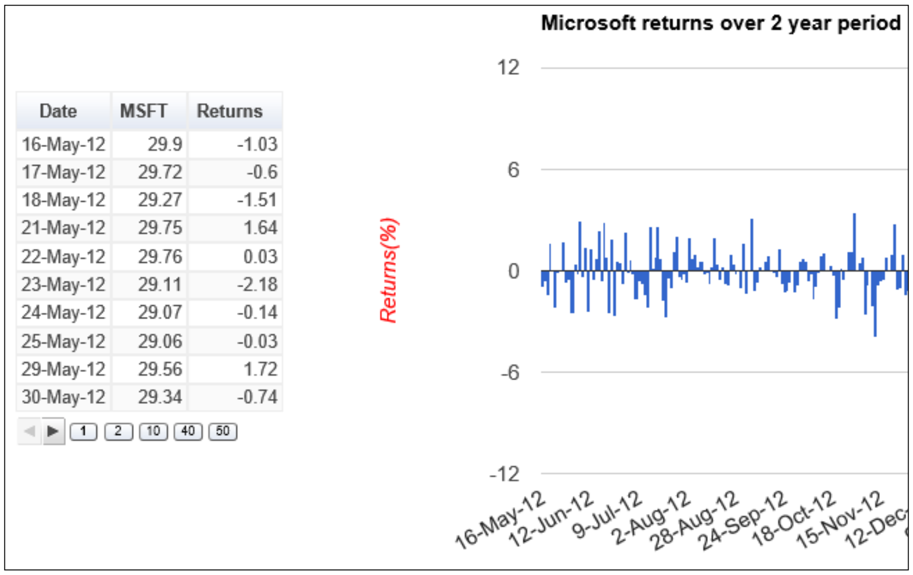
We have defined our `gvisBarChart()` function as `barpt`; we need this in R as all the `googleVis` functions will generate a list that contains the HTML code and reference to a JavaScript function. If you omit `barpt =`, R will display the code generated in your command window.

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:



Date	MSFT	Returns
16-May-12	29.9	-1.03
17-May-12	29.72	-0.6
18-May-12	29.27	-1.51
21-May-12	29.75	1.64
22-May-12	29.76	0.03
23-May-12	29.11	-2.18
24-May-12	29.07	-0.14
25-May-12	29.06	-0.03
29-May-12	29.56	1.72
30-May-12	29.34	-0.74

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.

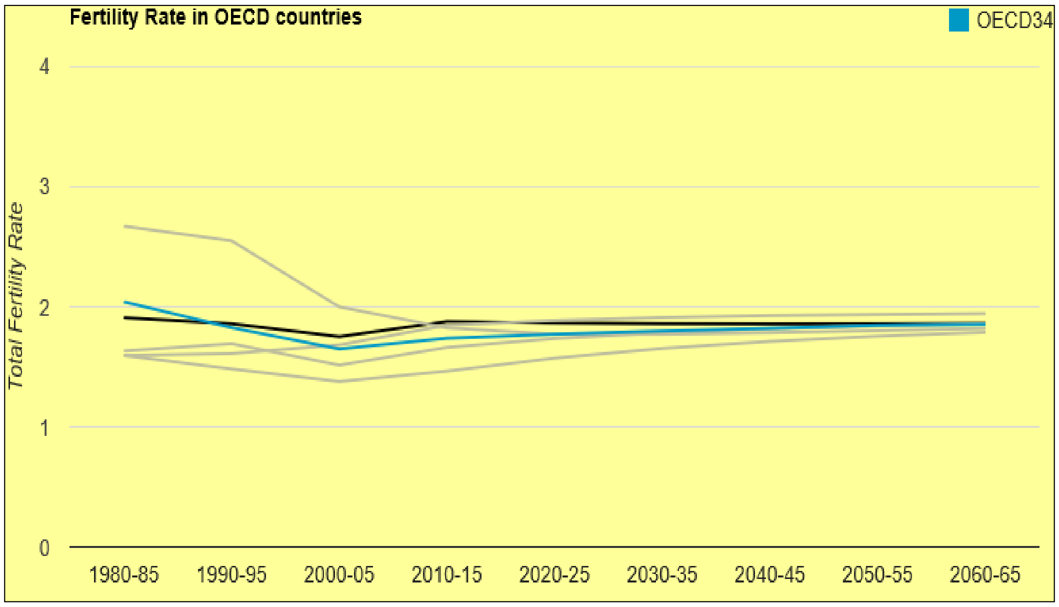
The readers can learn more about exporting the visualization in the `googleVis` package manual. It is also possible to integrate the `googleVis` package plots with `shiny`. Please refer to the recipe *Creating a very simple shiny app in R* in *Chapter 10*, *Creating Applications in R*.

See also

- The Google Chart API developer website can be accessed at `https://developers.google.com/chart/interactive/docs/gallery/barchart#Configuration_Options`
- 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)
```

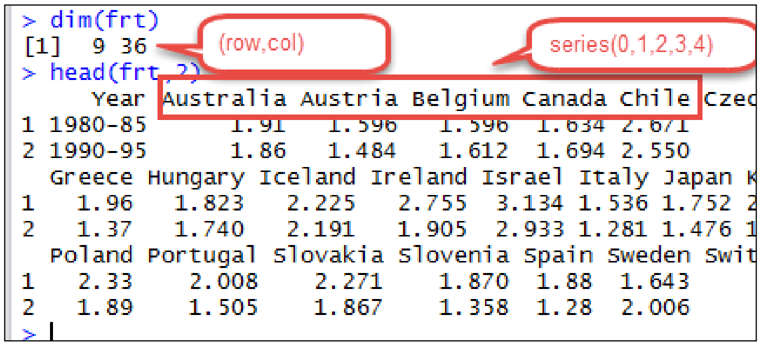
How it works...

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 Czec
1 1980-85      1.91   1.596   1.596  1.634 2.671
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
> |
```

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`.

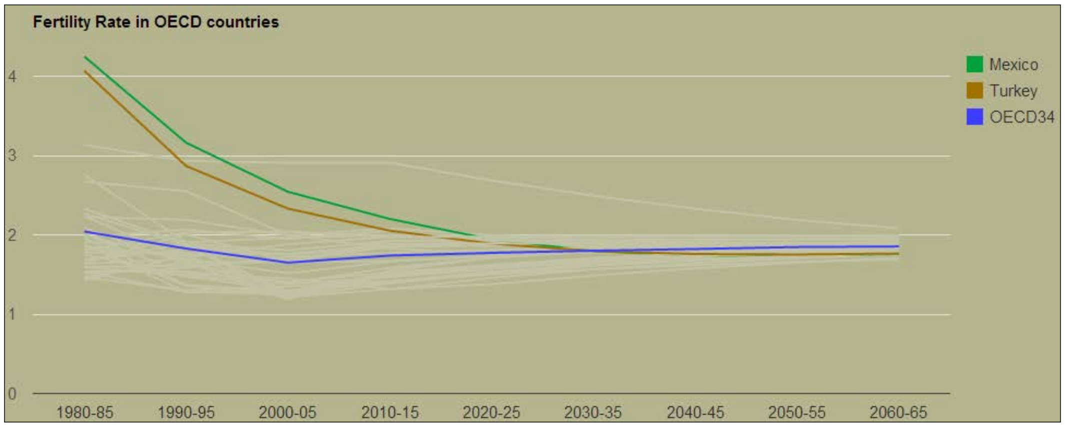
Line plot to tell an effective story

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

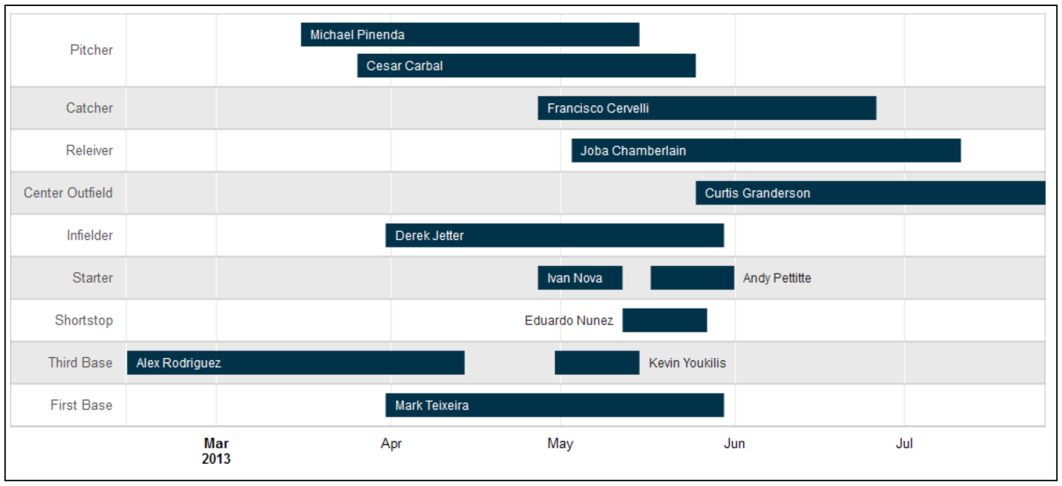
- D3 is a great tool to develop interactive visualization and this can be accessed at `http://d3js.org/`.
- Processing is an open source software developed by MIT and can be downloaded from `https://processing.org/`.
- A good resource to pick colors and use them in our plots is the following link: `http://www.w3schools.com/tags/ref_colorpicker.asp`.
- 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.

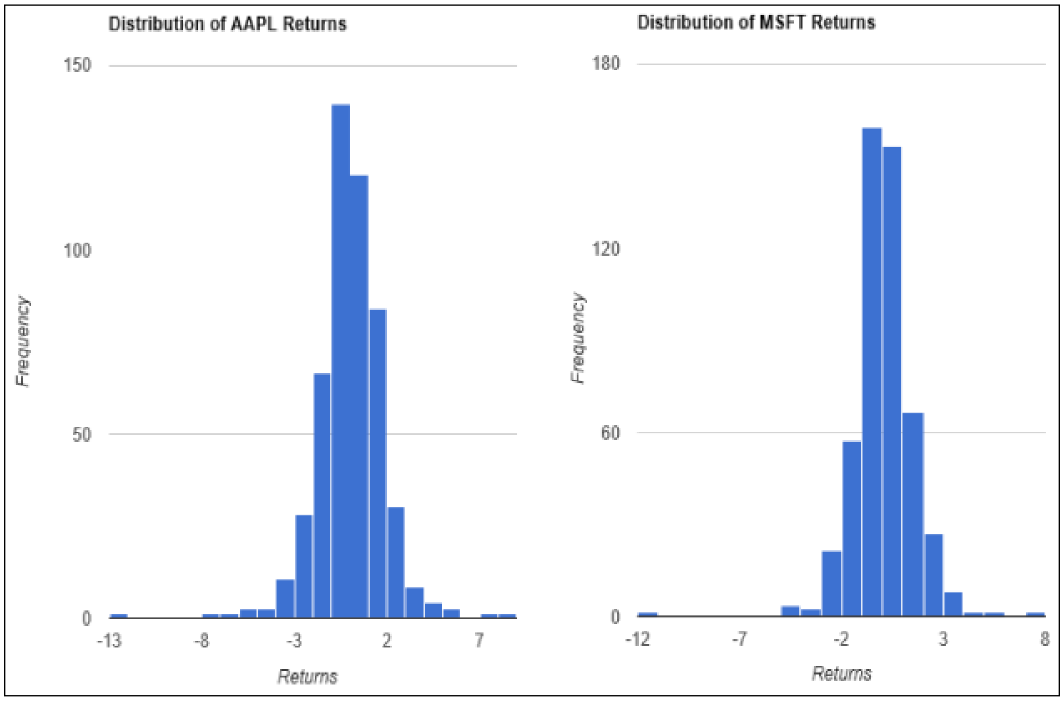
See also

- 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.

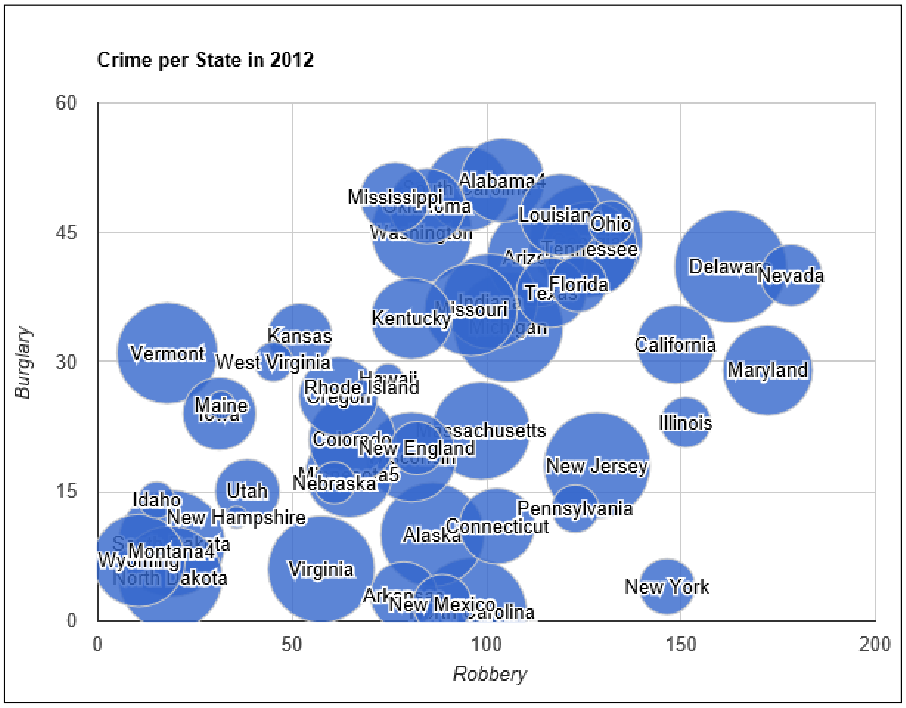
In our recipe, we have two plots for AAPL and MSFT. The default setting plots each chart vertically but we can specify `horizontal = true` to plot charts horizontally.

Making an interactive bubble plot

My first encounter with a bubble plot was while watching a TED video of Hans Roslling. 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...

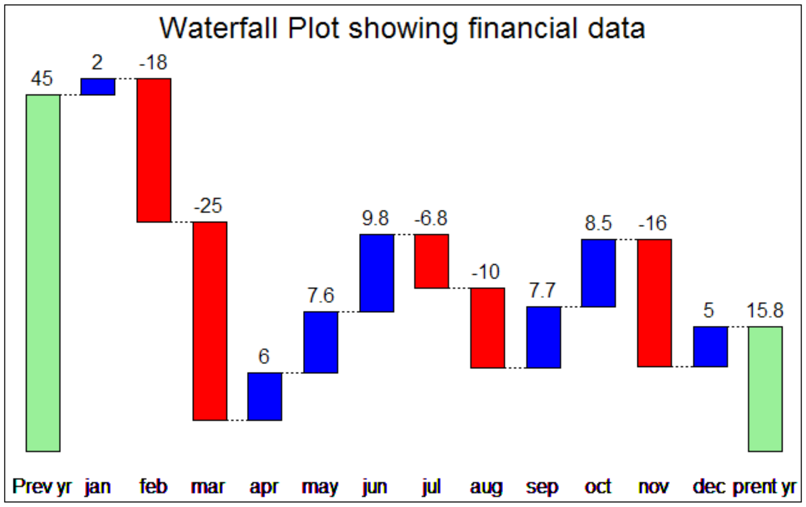
The Google Chart API makes it easier for us to plot a bubble, but the same can be achieved using the R basic plot function. We can make use of the symbols to create a plot. The symbols need not be a bubble; it can be a square as well. By this time, you should have watched Hans' TED lecture and would 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 10*, *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`
- 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.

3
Heat Maps and Dendrograms

In this chapter, we will cover the following recipes:

- Constructing a simple dendrogram
- Creating dendrograms 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 dendrograms 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. Dendrograms, 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.

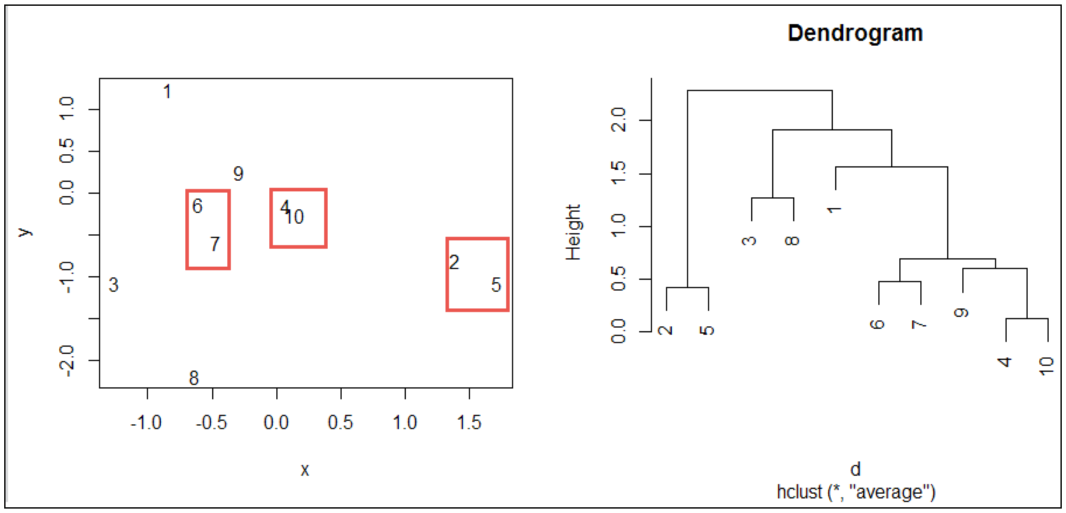
My first encounter with heat maps was an infographic published by the New York Times (`http://www.nytimes.com/interactive/2012/06/11/sports/basketball/nba-shot-analysis.html?_r=0`) about the points scored by basketball players playing for Miami Heat versus Oklahoma City Thunders. The infographic was 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 USArrests 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(mtx$x,mtx$y, type = "n", xlab = "x", ylab = "y") # generates
  the plot on the left side
text(mtx$x,mtx$y, 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.

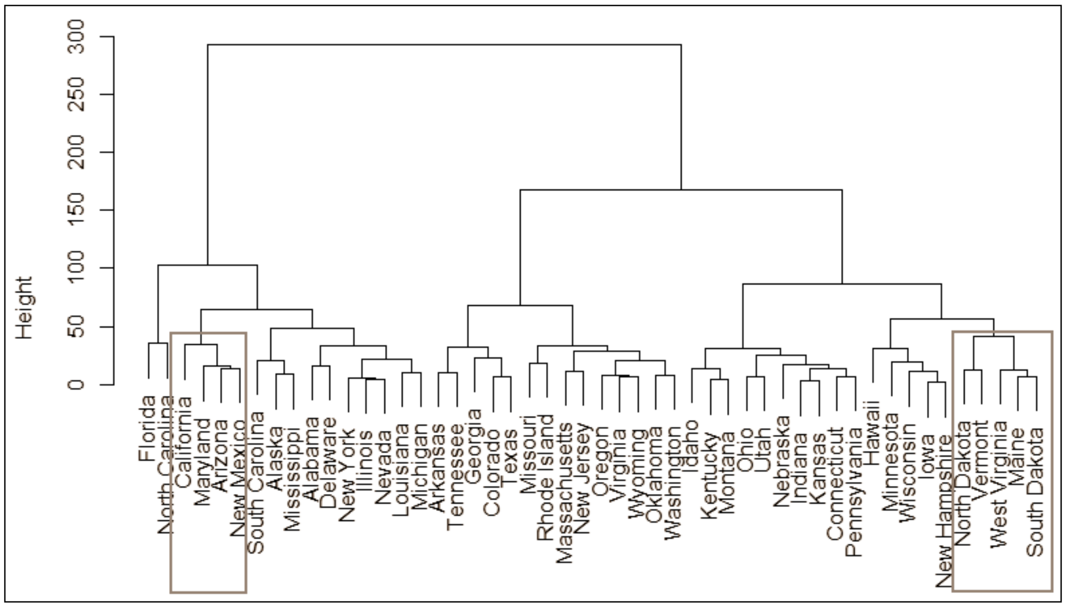
The `hclust()` function is a basic R functionality and is used for hierarchical cluster analysis in R. Hierarchical cluster is a method of clustering in which we do not have any prior knowledge of the underlying cluster. To learn more about hierarchical clustering, readers should refer to *An Introduction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

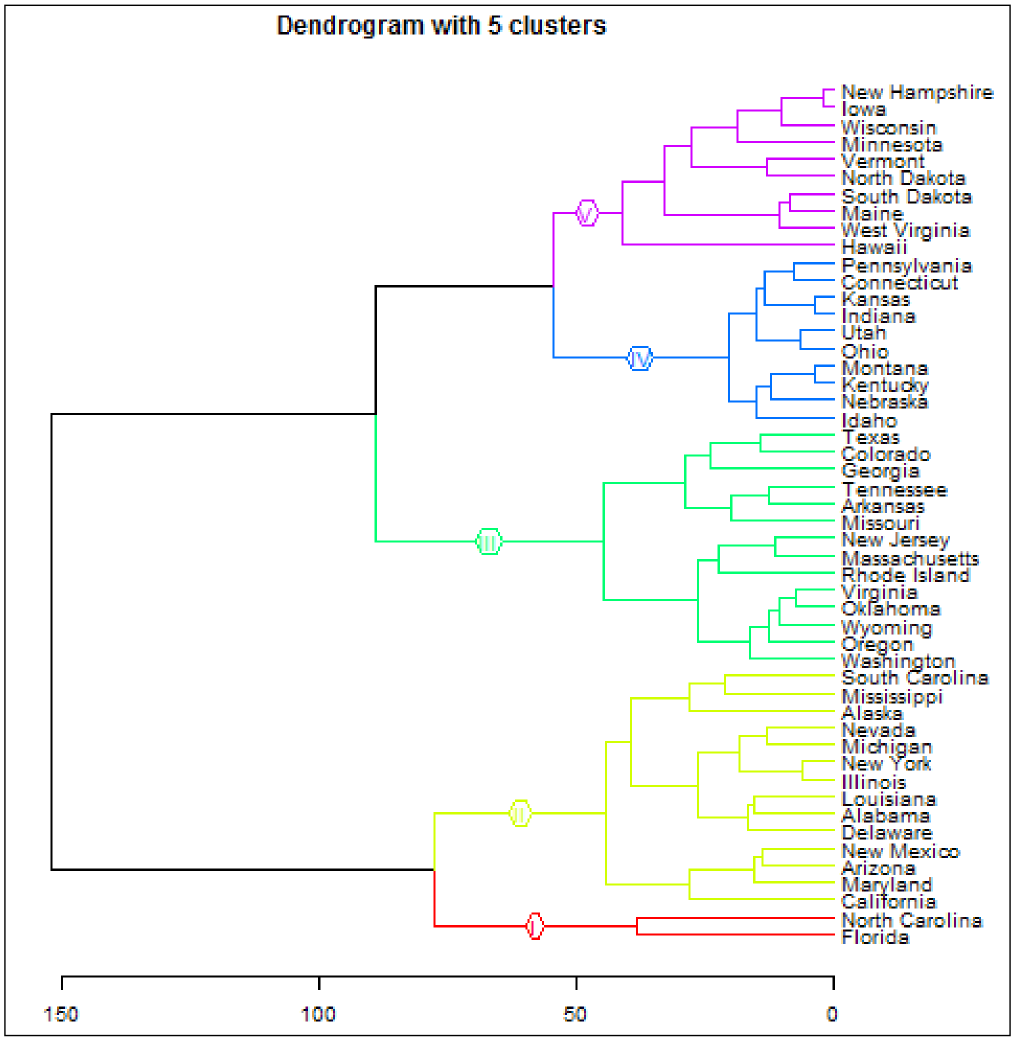
```
data = data.frame(USArrests[,1:3])
dt = dist(data, method = "euclidean")
clust = hclust(dt)
plot(data$Murder, data$Assault) # generates a scatter plot
text(USArrests$Murder,USArrests$Assault, labels =
  row.names(USArrests), cex = 0.6) # applies labels
    to a scatter plot.
plot(clust) # generates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

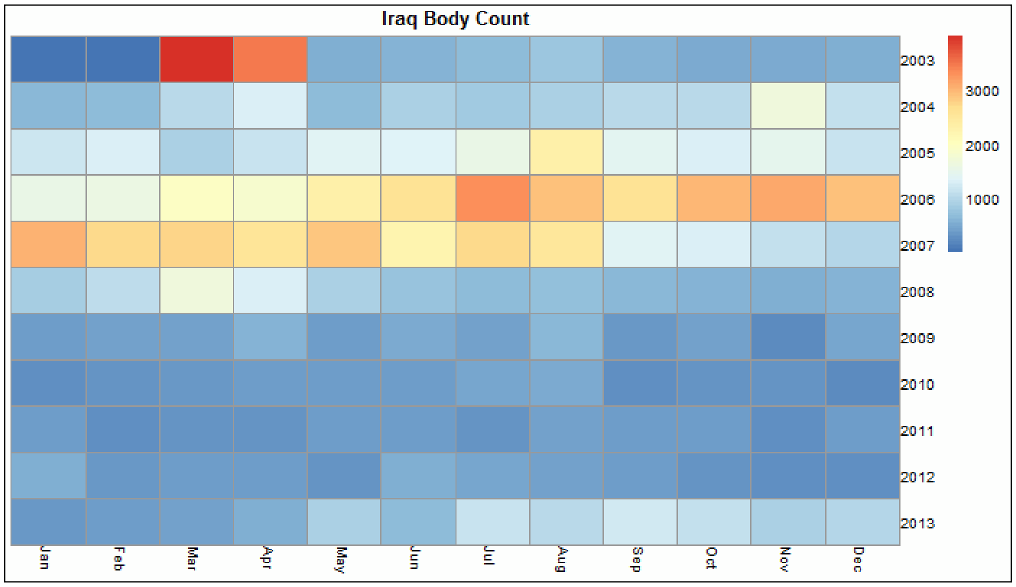
The preceding code uses the `NeatMap` package in R. Hence, we load the package and also load the USArrests dataset available in R. For this exercise, we first need to calculate the cluster using the `hclust()` and `dist()` functions in R and then position the list for the labels.

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 7*, *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 1, 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.

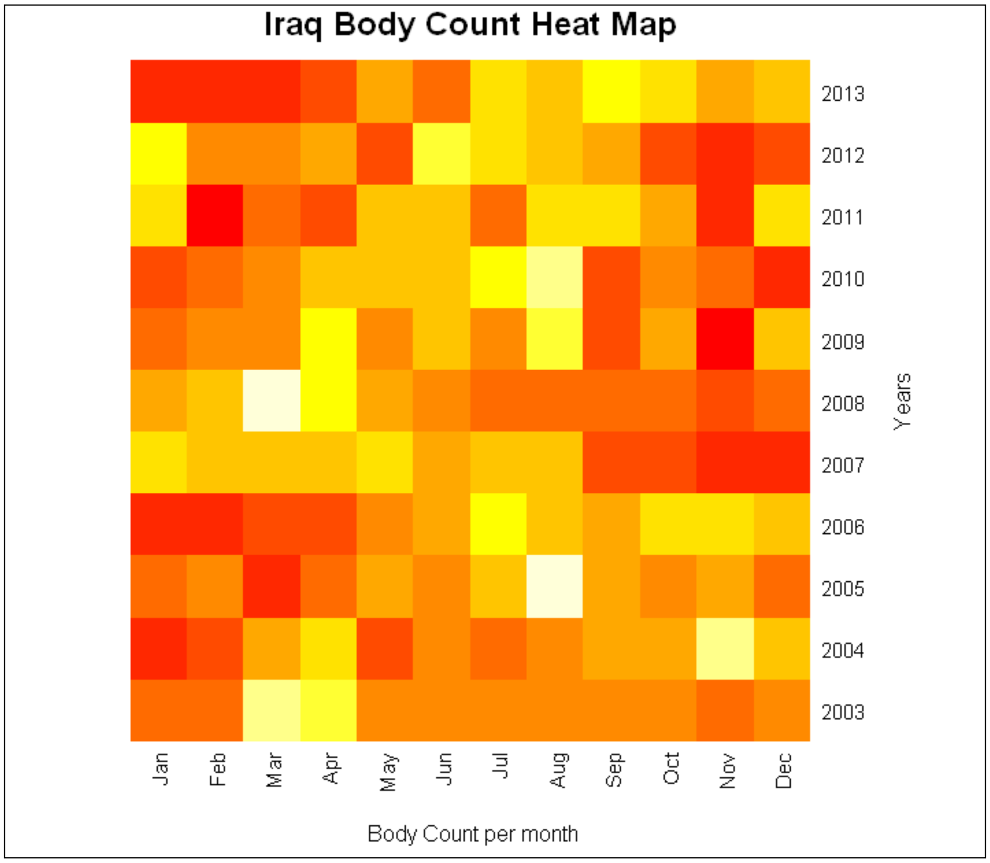
There's more...

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.

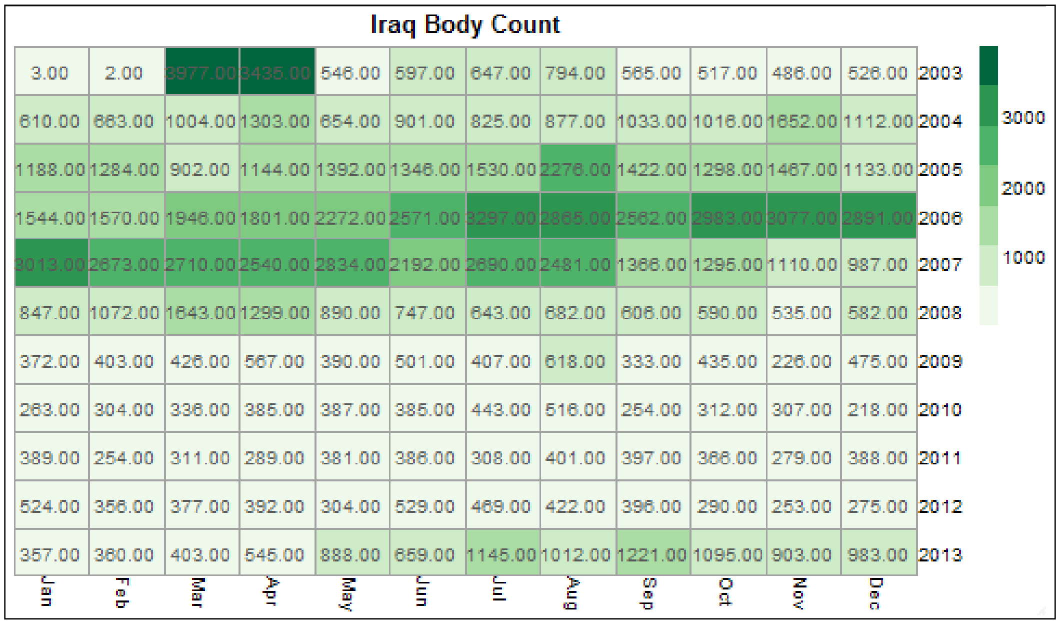


See also

- *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:



3.00	2.00	3977.00	3435.00	546.00	597.00	647.00	794.00	565.00	517.00	486.00	526.00	2003
610.00	663.00	1004.00	1303.00	654.00	901.00	825.00	877.00	1033.00	1016.00	1652.00	1112.00	2004
1188.00	1284.00	902.00	1144.00	1392.00	1346.00	1530.00	2276.00	1422.00	1298.00	1467.00	1133.00	2005
1544.00	1570.00	1946.00	1801.00	2272.00	2571.00	3297.00	2865.00	2562.00	2983.00	3077.00	2891.00	2006
3013.00	2673.00	2710.00	2540.00	2834.00	2192.00	2690.00	2481.00	1366.00	1295.00	1110.00	987.00	2007
847.00	1072.00	1643.00	1299.00	890.00	747.00	643.00	682.00	606.00	590.00	535.00	582.00	2008
372.00	403.00	426.00	567.00	390.00	501.00	407.00	618.00	333.00	435.00	226.00	475.00	2009
263.00	304.00	336.00	385.00	387.00	385.00	443.00	516.00	254.00	312.00	307.00	218.00	2010
389.00	254.00	311.00	289.00	381.00	386.00	308.00	401.00	397.00	366.00	279.00	388.00	2011
524.00	356.00	377.00	392.00	304.00	529.00	469.00	422.00	396.00	290.00	253.00	275.00	2012
357.00	360.00	403.00	545.00	888.00	659.00	1145.00	1012.00	1221.00	1095.00	903.00	983.00	2013
Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	

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.packges(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)
```

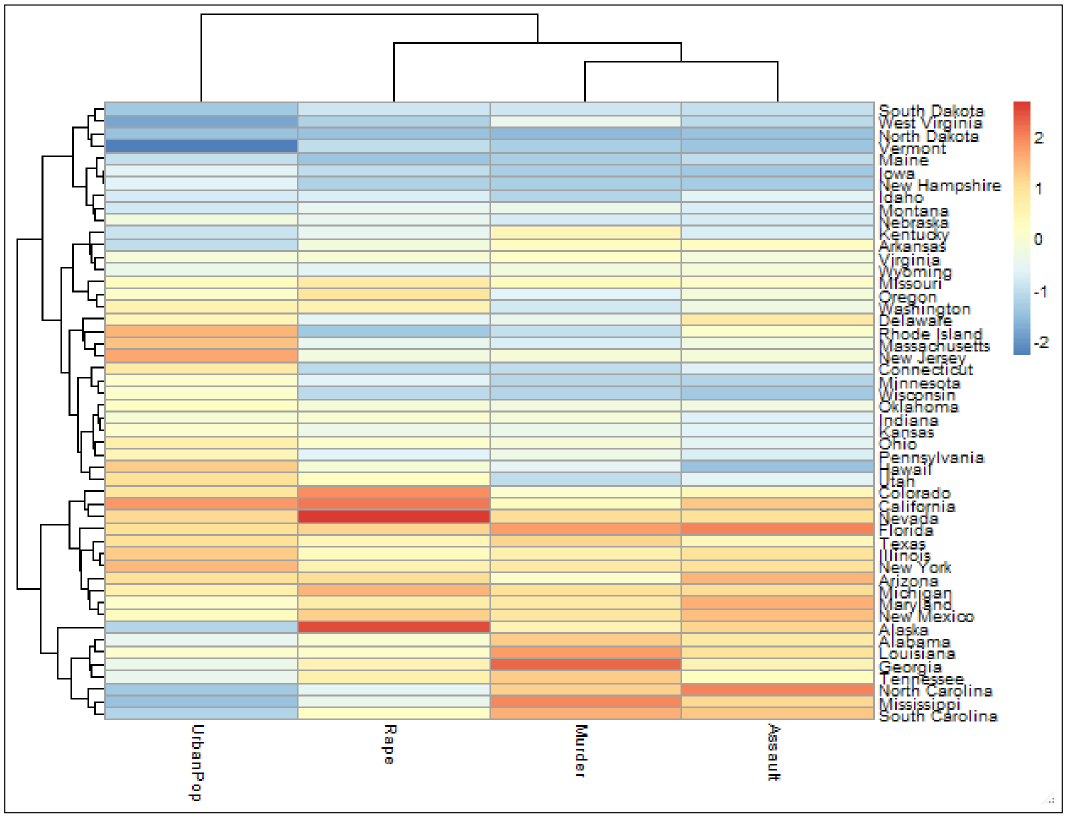
The color shading provides our audience with a visual cue as to the year or month in which the war claimed more innocent lives. The legend key, where a darker shade represents higher deaths, assists our audience in better data interpretation. We can overcome the issue many people face in interpreting 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 dendrograms a is that the visualization provides us with more information.

We will observe in this recipe that we are able to view the clustering via dendrograms 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 USArrests 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 10*, *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 10*, *Unsupervised Learning*, of *An Introduction to Statistical Learning*:

```
data = as.matrix(scale(USArrests))
clst = hclust(dist(data))
```

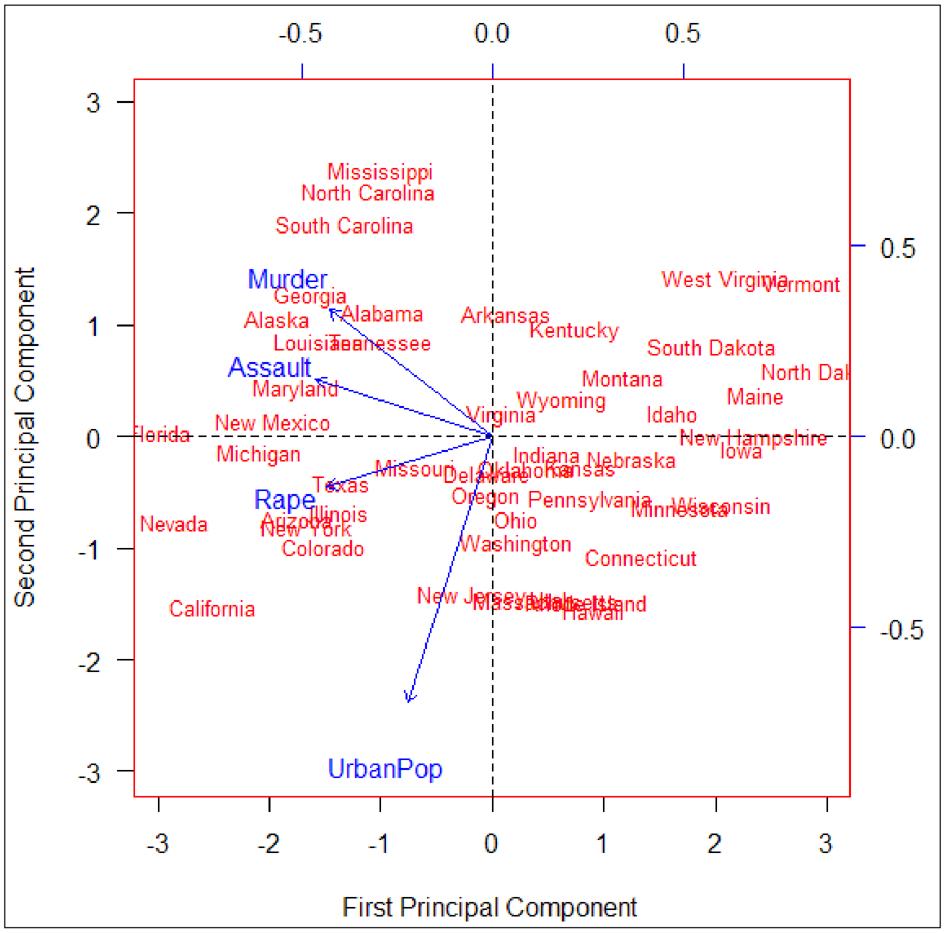
Once we scale the data, we can simply plot the data using the `pheatmap()` function. In this recipe, we will not suppress the dendrogram. Hence, we will use all the default settings to plot a dendrogram and a heat map:

```
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 USArrests. 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:

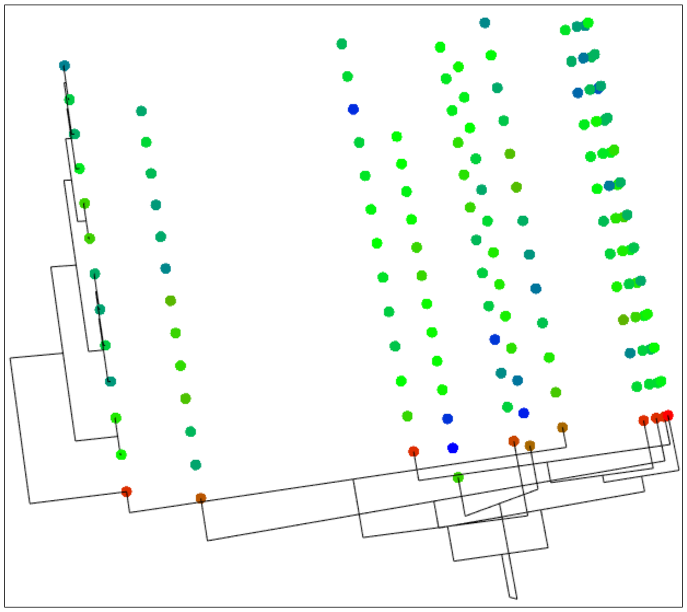
```
pc = prcomp(USArrests, scale = TRUE)
biplot (pr.out , scale =0, col =c("red","blue"), cex = c(0.8,1),
  xlab = "First Principal Component", ylab = "Second Principal
    Component")
abline(h = 0, lty =2)
abline(v = 0, lty =2)
```

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.stere.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)
```

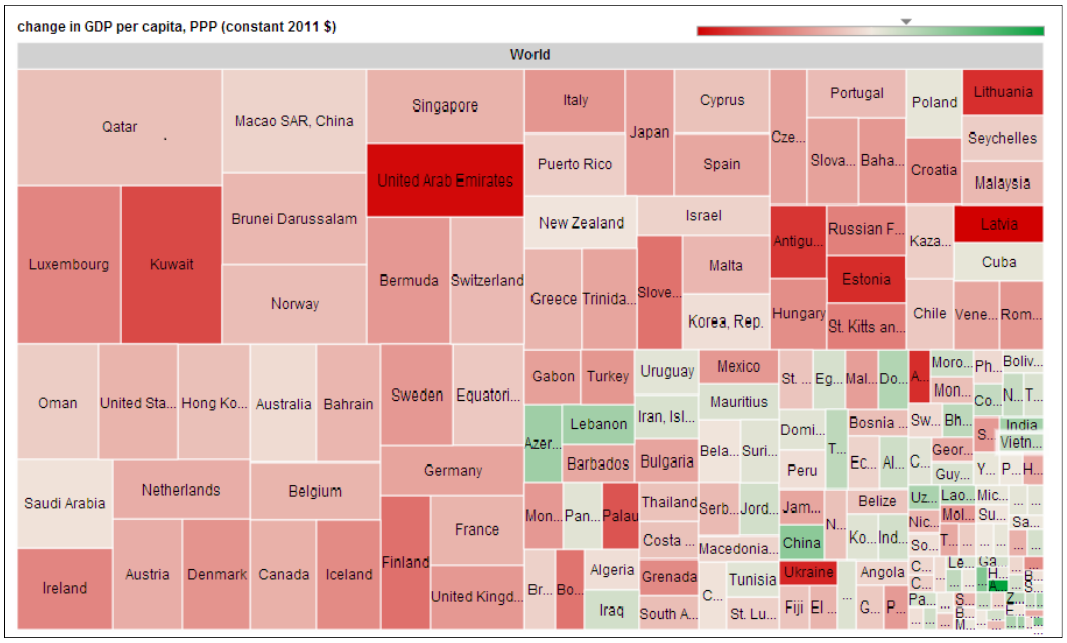
Many of the arguments in the `make.stereo()` function are similar to the function mentioned in `make.profileplot3d()`.

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.

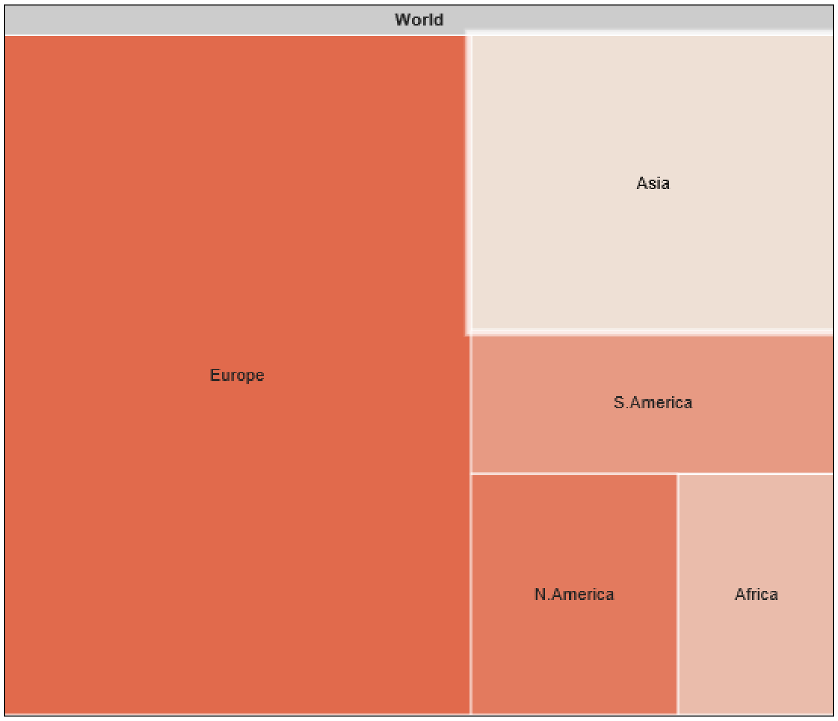
Tree map comes with a few handy options, which can be specified using `options = list()`. In our example, we have made use of title, width, height, and font color, which have been explained and used in the previous chapter. The tree maps come with a few other options such as `minColor` and `maxColor`, 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`.

4
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.

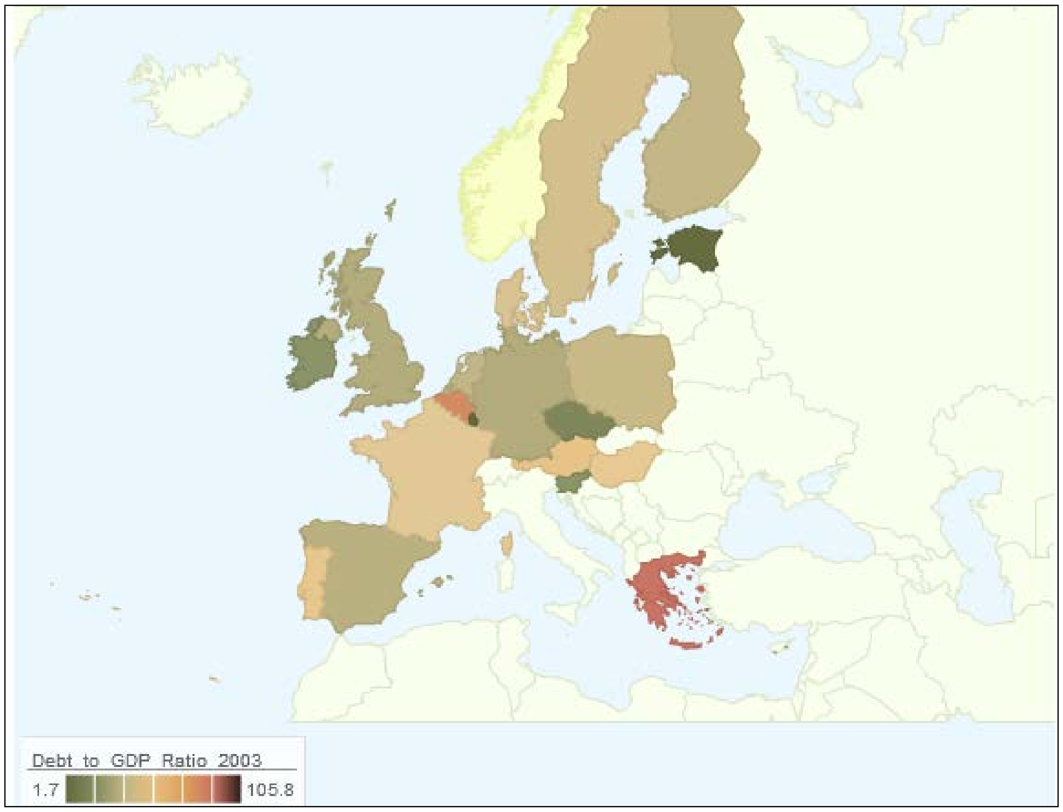
My motive behind this chapter is to introduce the readers to various forms of spatial data visualization. This chapter clearly does not do justice to the visualization of geographic data but it leads you towards the right tools, methods, and procedures. A lot has been researched and written on spatial 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

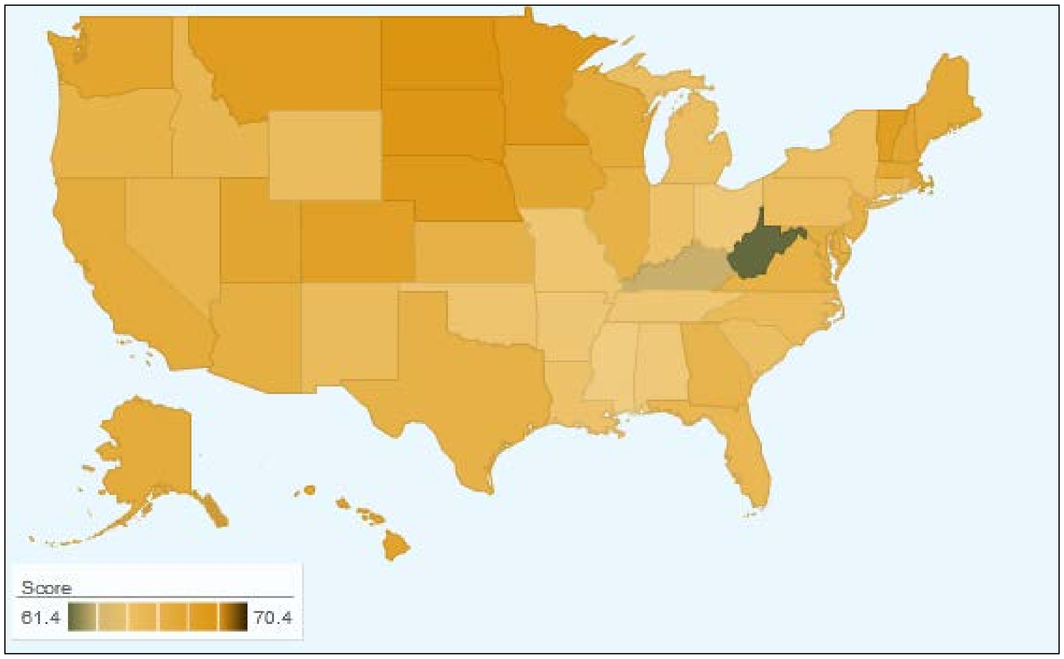
- The *Debt Rising in Europe* article at `http://www.nytimes.com/interactive/2010/04/06/business/global/european-debt-map.html`
- The Dude Map at `http://qz.com/316906/the-dude-map-how-american-men-refer-to-their-bros/`

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...

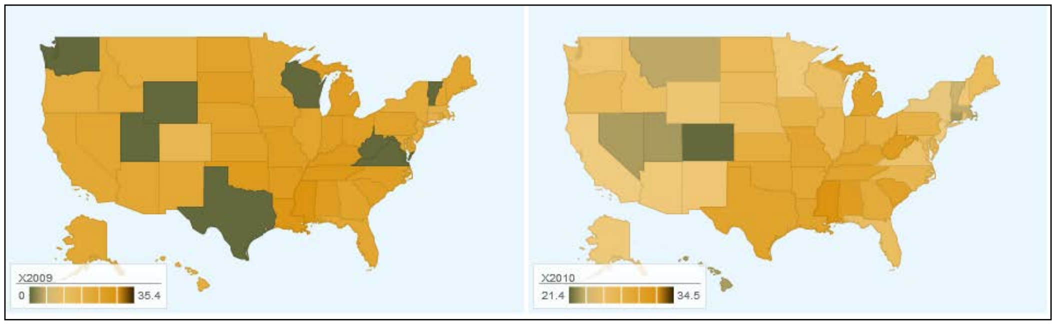
The first three arguments used in the `gvisGeoMap()` function are `data`, `locationvar`, and `numvar`. These arguments are discussed in detail in the previous recipe.

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:

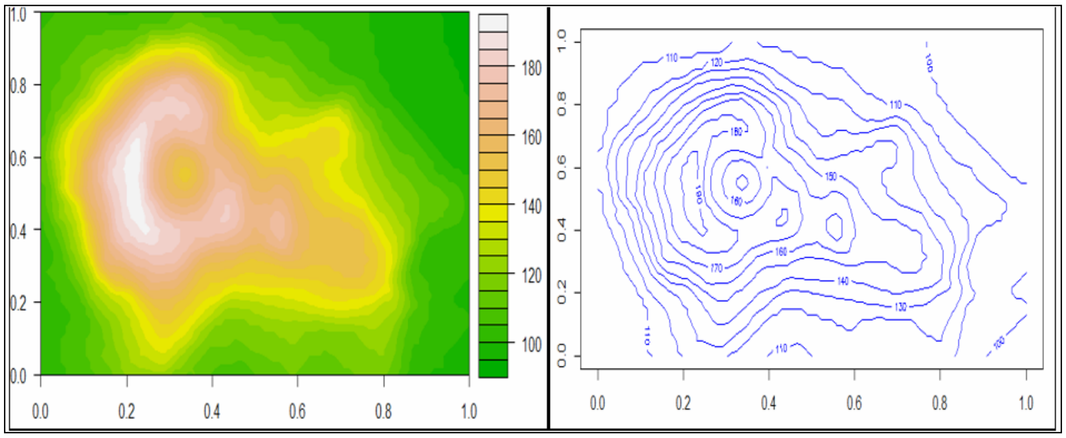
```
obese = read.csv("obesity.csv", header = TRUE, sep =",")
US_2012 <- gvisGeoMap(obese, locationvar="state",numvar ="X2012",
  options = list(width = 500, height =300,dataMode = "regions",
    region = "US",colors=
      "[0x64693D,0xF1CC80,0xEABA58,0xE3A732,0xDC9411]"))
US_2011 <- gvisGeoMap(obese, 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")
```

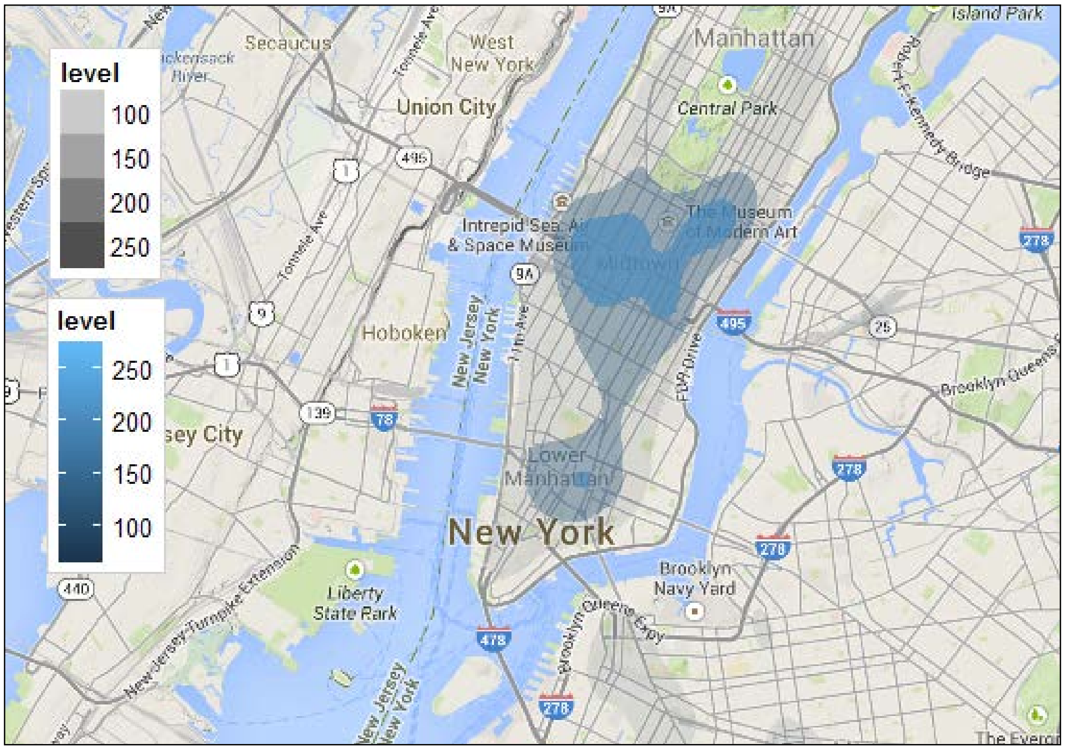
Readers can learn more about how they can generate custom color palettes in R by typing `?colorRamp` in the R console window. We can also specify a range a colors using the `RColorBrewer` package:

```
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`

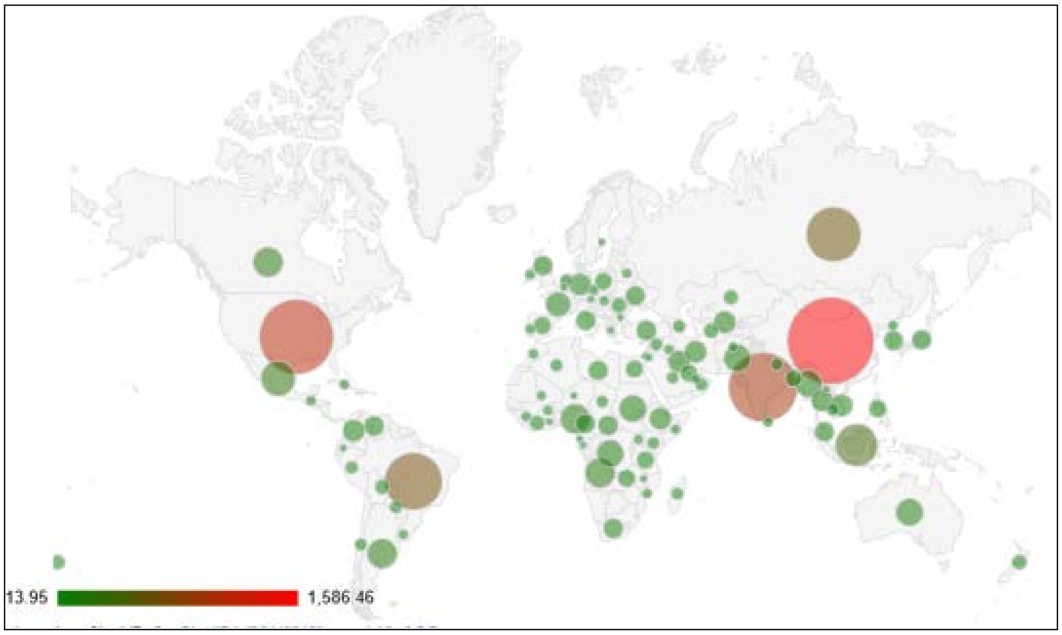
Constructing maps with bubbles

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 5*, *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'`.

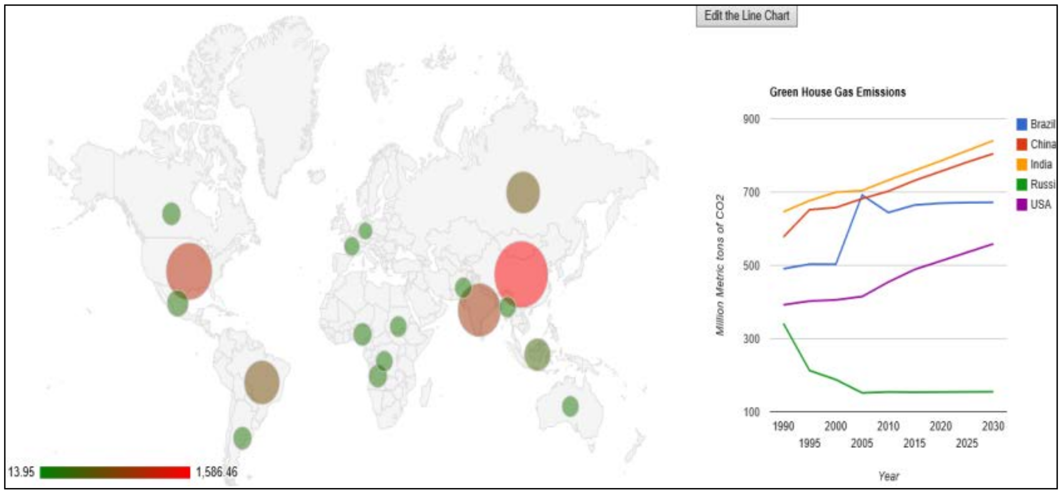
The `options` argument is used to set various options related to the visualization, such as width, height, and color. The `googleVis` package also allows us to set the maximum and minimum radius of the bubbles using the `sizeAxis` option. Readers should note that the `title` option is not implemented 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)
```

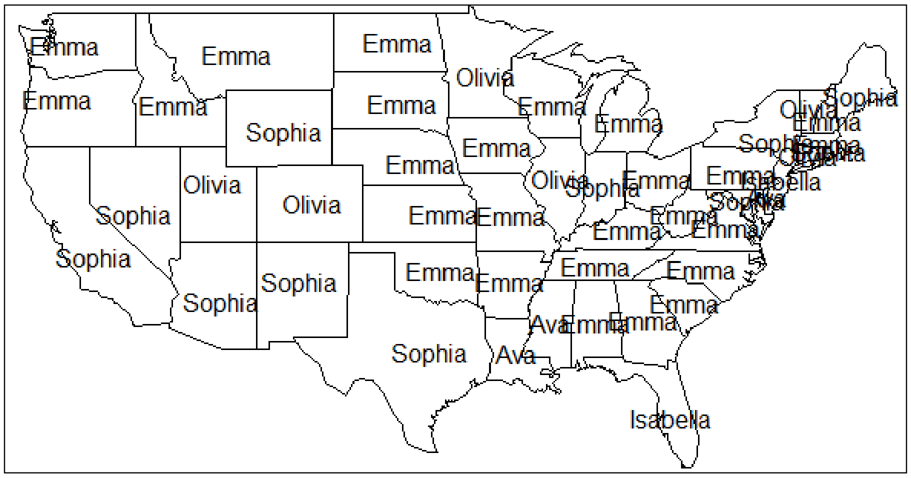
See also

- 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`
- 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 10*, *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`

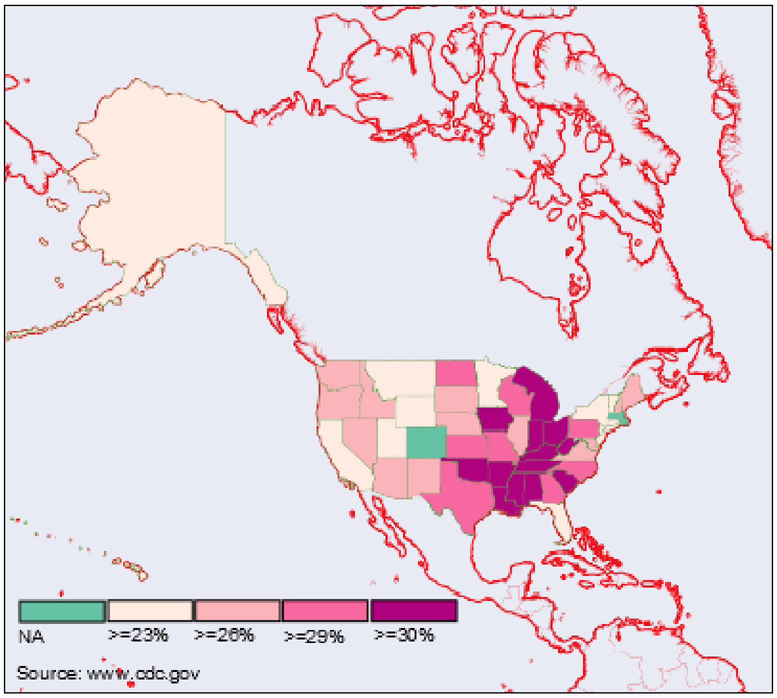
- The *Newspaper Endorsements: A Final Tally* article at `http://www.nytimes.com/interactive/2008/11/03/us/politics/20081103-endorse-graphic.html`
- The *Twitter Chatter During Super Bowl* article at `http://www.nytimes.com/interactive/2009/02/02/sports/20090202_superbowl_twitter.html`

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`). 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

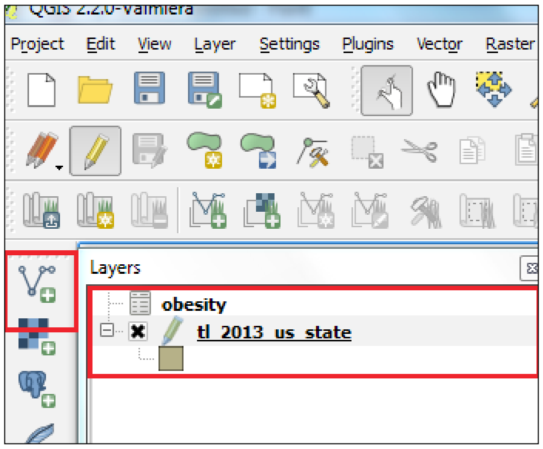
We will use the following open source software for creating the map:

- QGIS available at `http://www.qgis.org/`
- Tile Mill available at `https://www.mapbox.com/tilemill/`
- Inkspace available at `http://www.inkscape.org/`

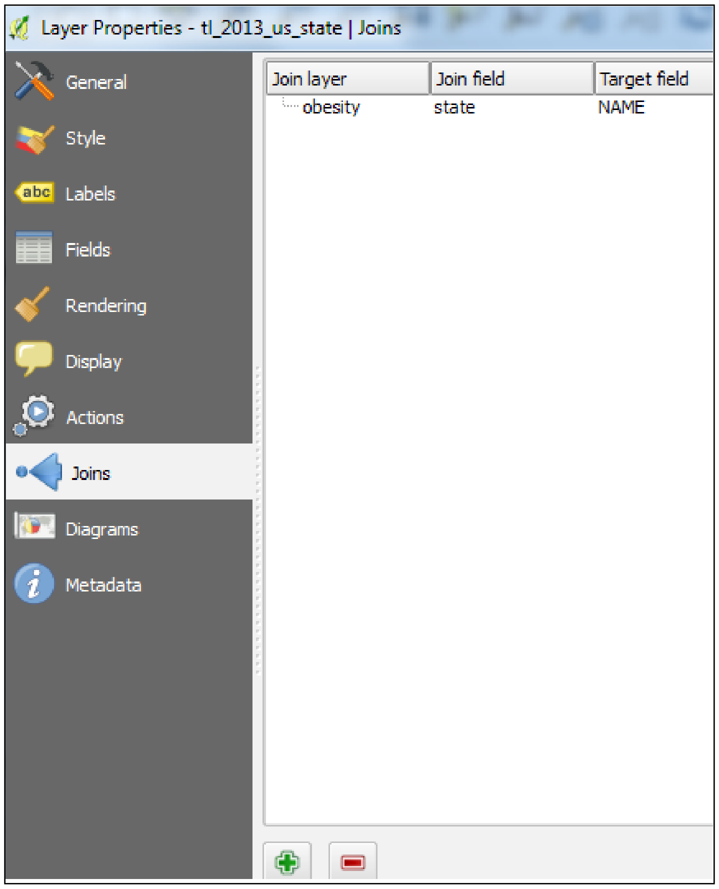
How to do it...

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:



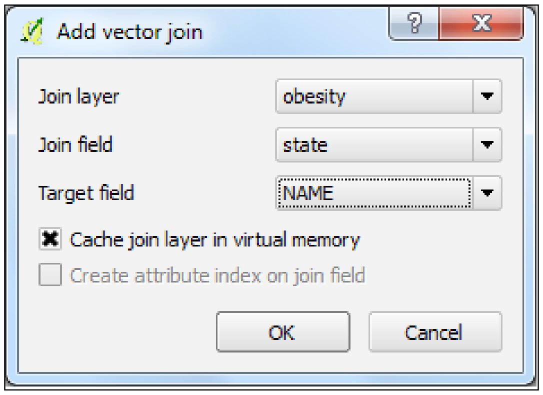
3. You can check your data by right-clicking on the name and selecting **Open Attributes Table**. Now select the shapefile and double click on it; this should open up a layer properties dialogue box, as shown in the following screenshot:



Join layer	Join field	Target field
obesity	state	NAME

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:


Add vector join

Join layer: obesity

Join field: state

Target field: NAME

[✓] Cache join layer in virtual memory

[ ] Create attribute index on join field

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`

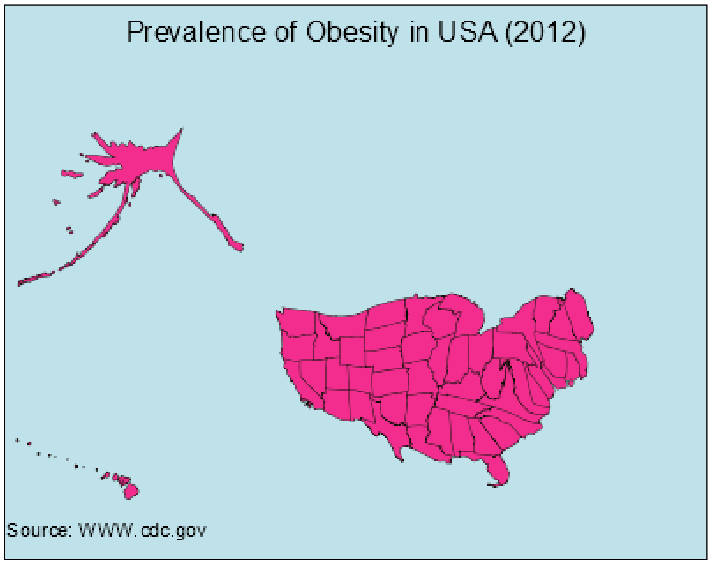
Creating cartograms

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.

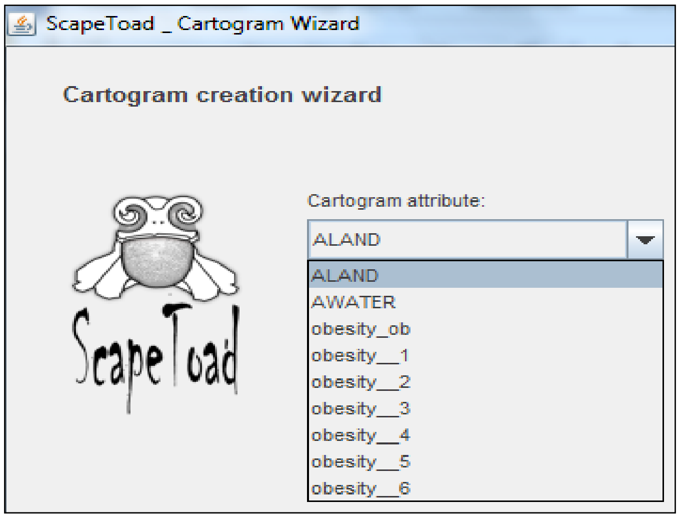
How to do it...

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.



Cartogram attribute:
ALAND
ALAND
AWATER
obesity_ob
obesity__1
obesity__2
obesity__3
obesity__4
obesity__5
obesity__6

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/` 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*.

5
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)*

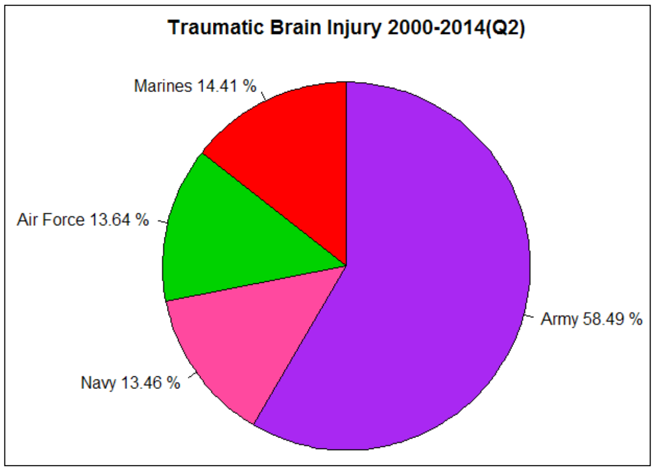
The main criticism of pie charts is that they are difficult to interpret, especially when the difference between two slices of a pie is small. Human beings are not trained well to observe angles; hence, many times, pie charts might even be interpreted wrongly. Even with all their limitations, pie charts 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...

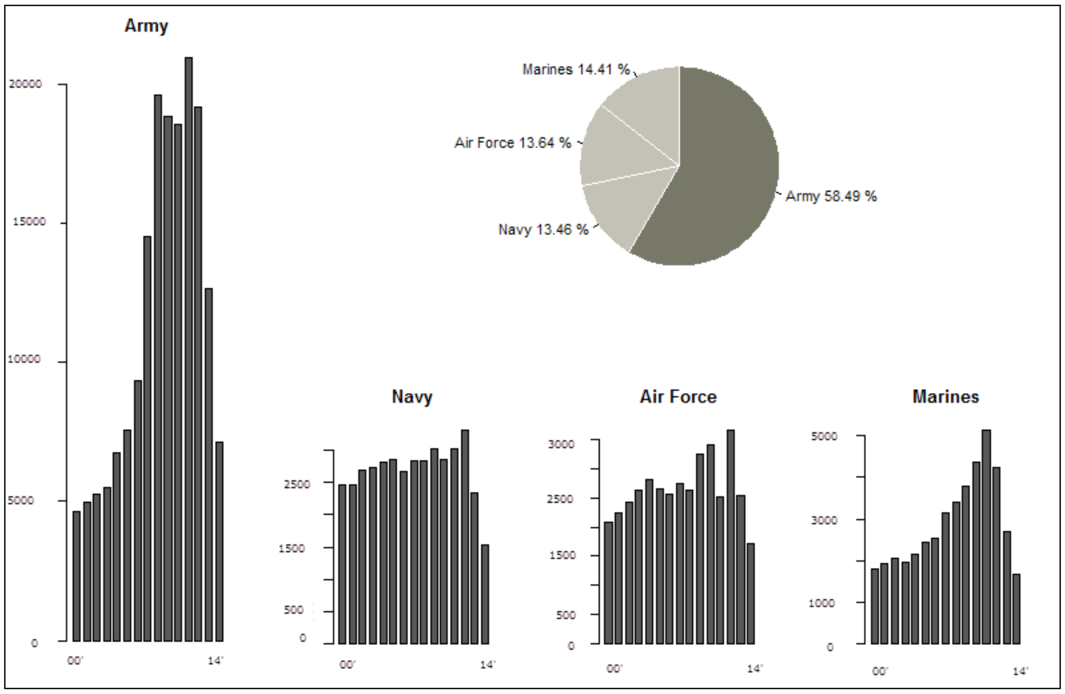
A pie chart will not plot the labels automatically; hence, we need to undertake a few additional steps to generate labels that can be applied to our pie. We would like to create a label with the name of the service, followed by the value, followed by the percentage sign. We convert our actual values 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")
```

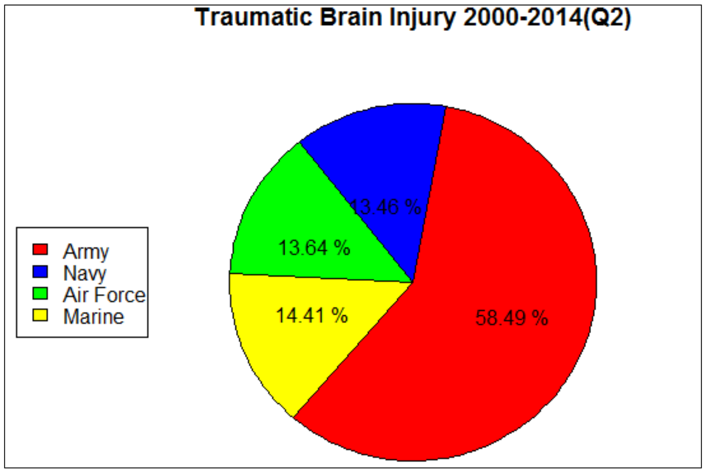
The preceding code is very similar to the code in the *How to do it…* section of this recipe. While implementing the `as.matrix()` function, we have manipulated our data and coerced our series to be in the vector format. To understand the use of the `[]` notation, please refer to the *Editing a Matrix in R* recipe in *Chapter 1*, *A Simple Guide to R*. The `barplot()` function has been described in detail in *Chapter 2*, *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 1*, *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.

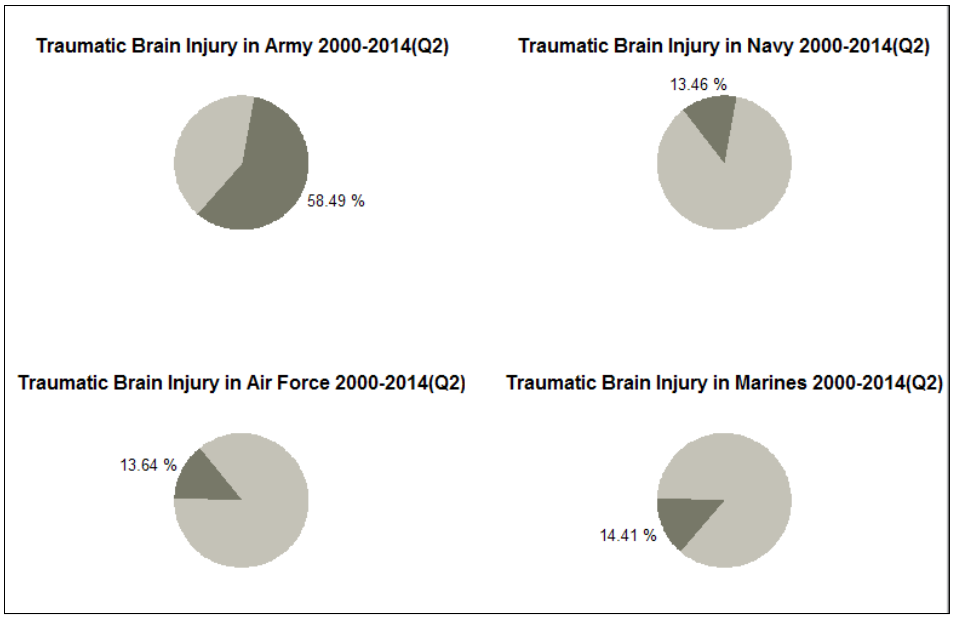
The first two arguments in the `floating.pie()` function are the *x* and *y* coordinates of the center of the pie. Note that this is the reason we generated a blank plot. The third argument `pct` is our vector of data. The `radius` and `col` arguments are the same as in the `pie()` function discussed 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","#BEBCB0","#BEBCB0","#BEBCB0"),
    border = FALSE, startpos = 4)
pie.labels(3,3, 5.837518 ,radius=0.9,labels=label[1])
```

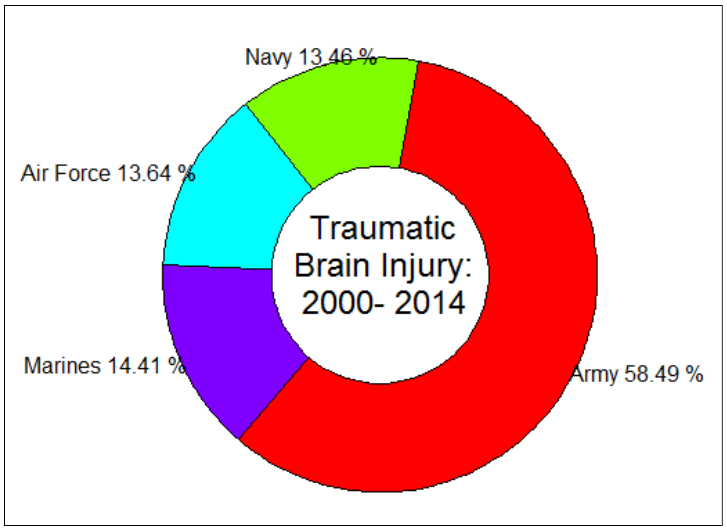
The preceding code is not complete; please refer to the code file to create the entire chart. The first section of the code in the code file is exactly the same as discussed in the previous two recipes. The `par()` function allows us to divide our plotting region into multiple sections. The `par()` function is discussed in detail in the recipe *Introducing a scatter plot.* in *Chapter 2, 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 rady

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...

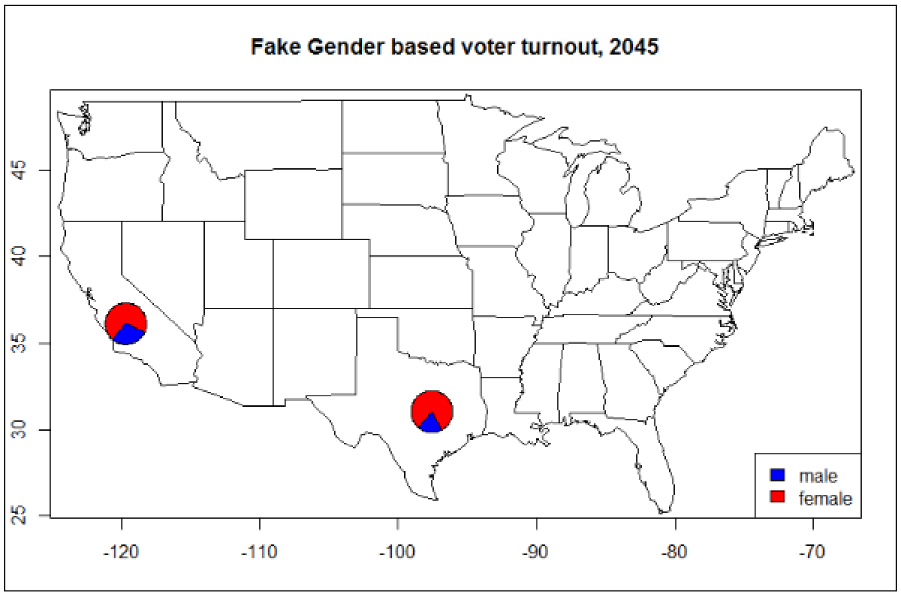
We have introduced two new functions in this recipe, namely: `draw.circle()` and `textbox()`. The first two arguments in the `draw.circle()` are the *x* and *y* coordinates. Note that we would like to draw a circle right in the centre of the pie chart; hence we will use the exact same *x* and *y* coordinates 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`

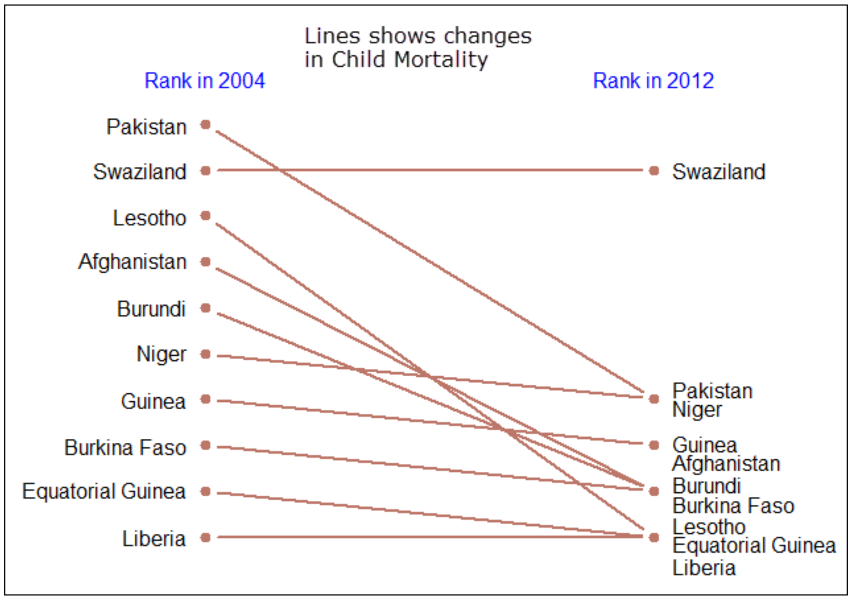
Generating a slope chart

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.

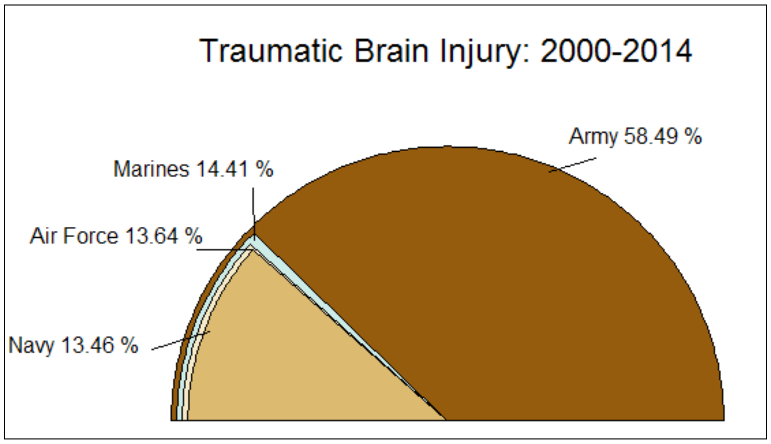
Readers should note that it is possible to construct a slope chart and plot the actual values on the plot. I would urge you to explore the `bumpchart()` function in detail by typing `?bumpcharts()` in the R console window.

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`
- Edward Tufte discusses the application of the slope graph at `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.

6
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

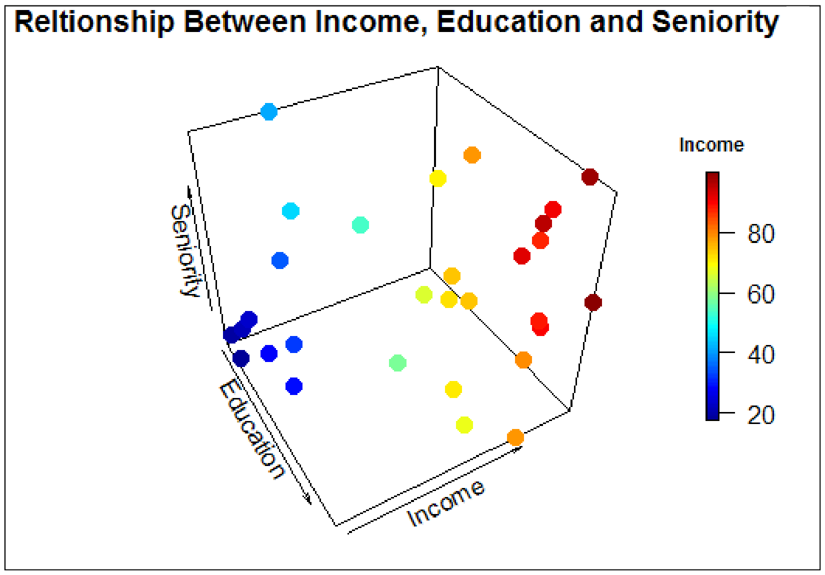
Three-dimensional images (pie charts or histograms), contours, surface plots, and scatter plots are great tools for analyzing multivariate data. Data that comprises more than two variables or densities, or even data with a high volume (medical or geosciences data), can be studied in detail when the third 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 = "Reltionship Between Income , Education and Seniority")
```

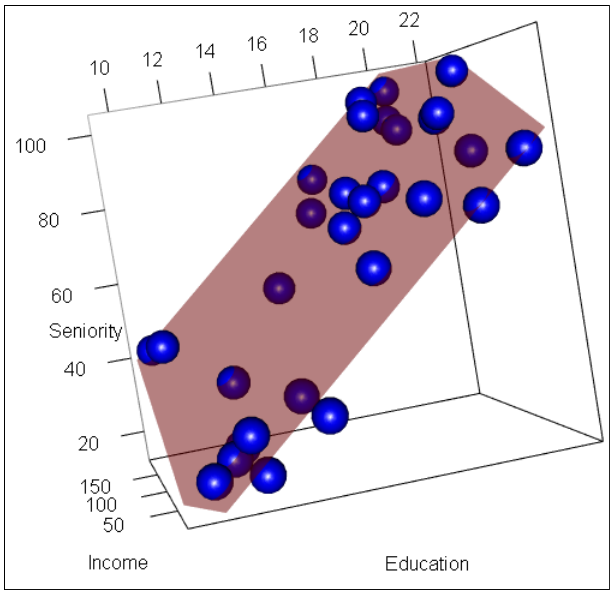
How it works...

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")
```

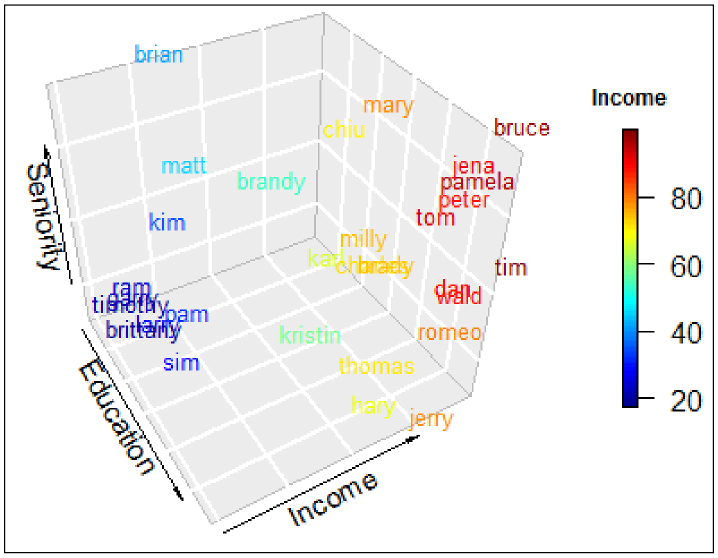
See also

- 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:

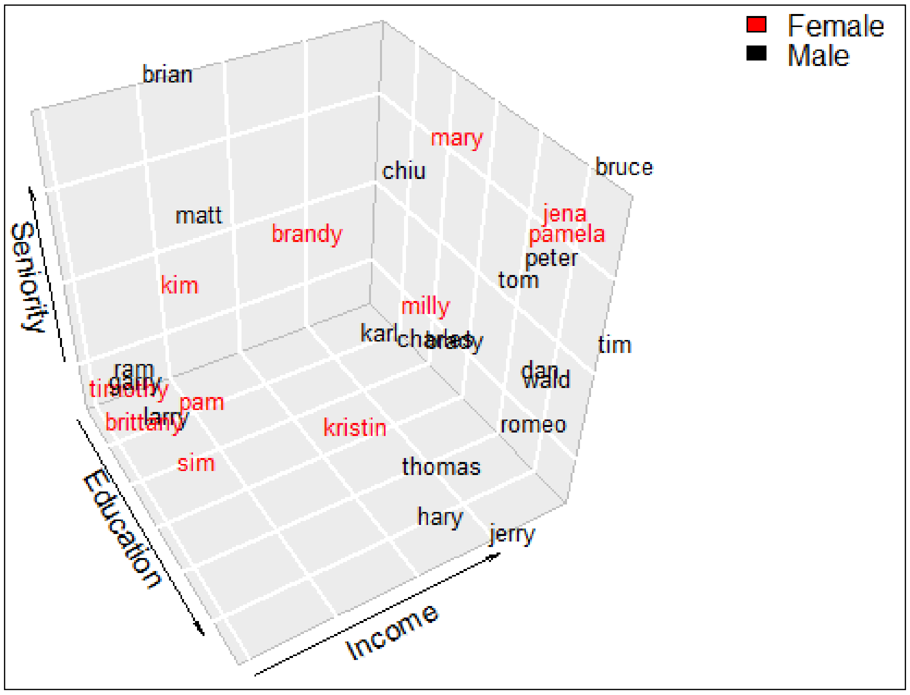
```
text3D(x = inc$Education, y = inc$Income, z =inc$Seniority,
  colvar = inc$Income,labels= row.names(inc),
  pch = 16, cex = 0.8, xlab = "Education", ylab = "Income",
  zlab = "Seniority", theta = 60, d = 2,clab = c("Income"),
  colkey = list(length = 0.5, width = 0.5, cex.clab = 0.75,
  dist = -.08, 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")
```

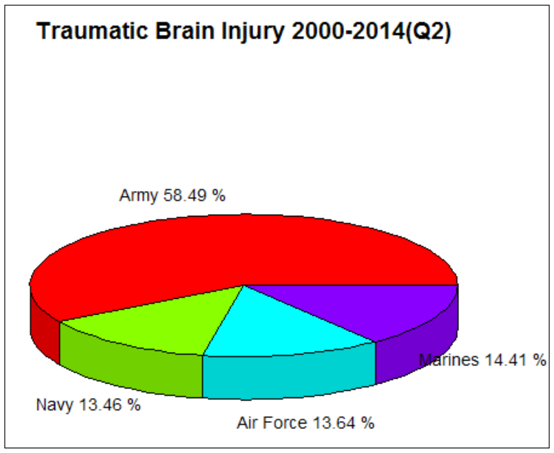
In the code file, you will also find code to generate a scatter plot using the `scatterplot3d` package.

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 5*, *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 5*, *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 5*, *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)
```

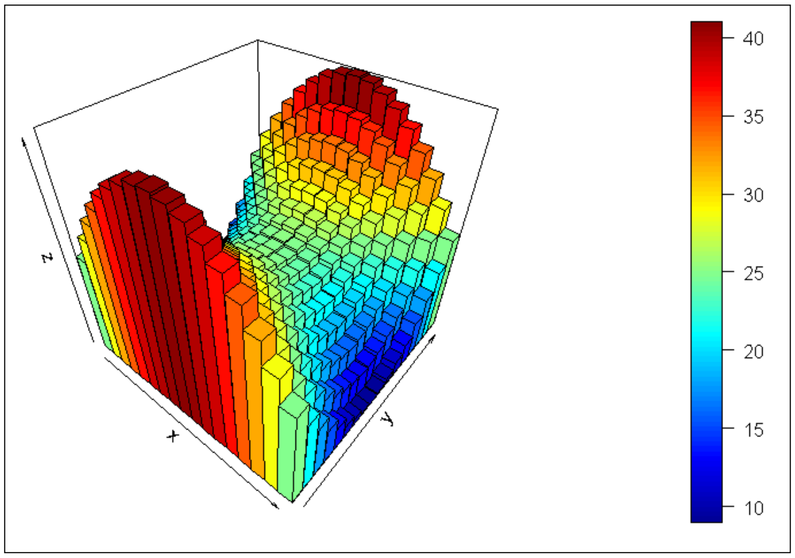
How it works...

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 1*, *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 1, A Simple Guide to R*. We can define a function `f` as follows:

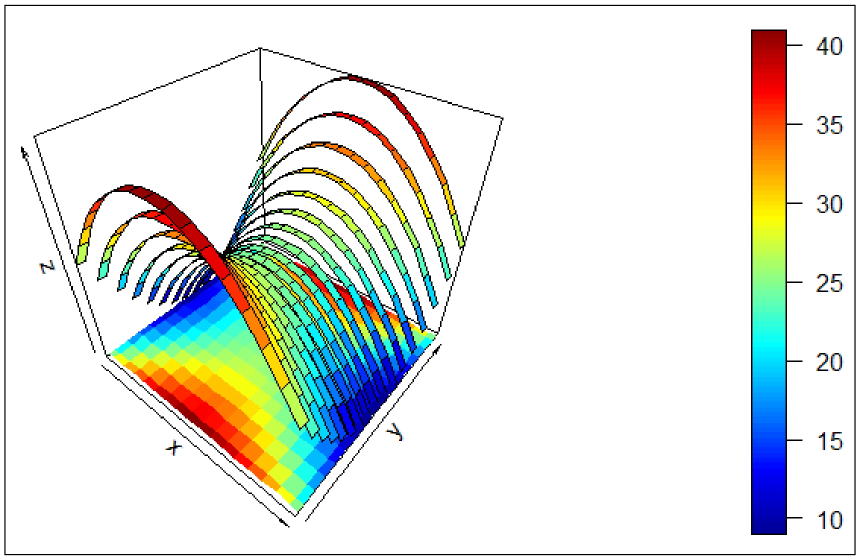
```
    f = function(x,y){
    z = (25-(x^2-y^2))
    }
```

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.



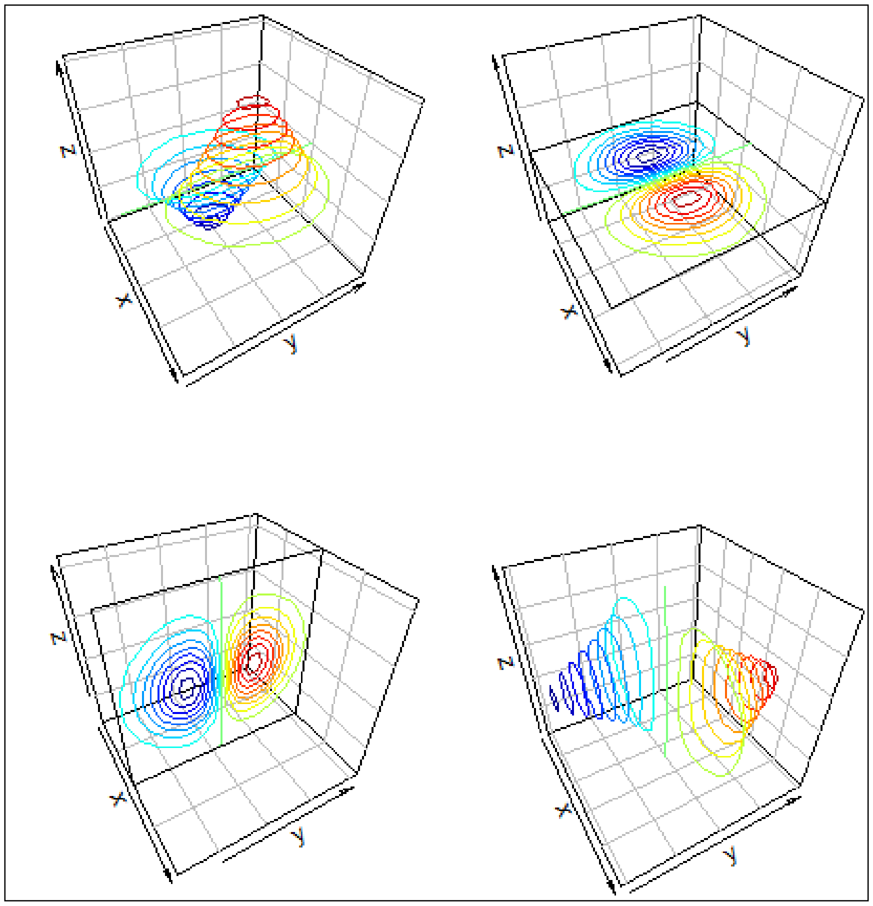
We can also plot a ribbon plot using the same function we defined under this recipe:

```
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.

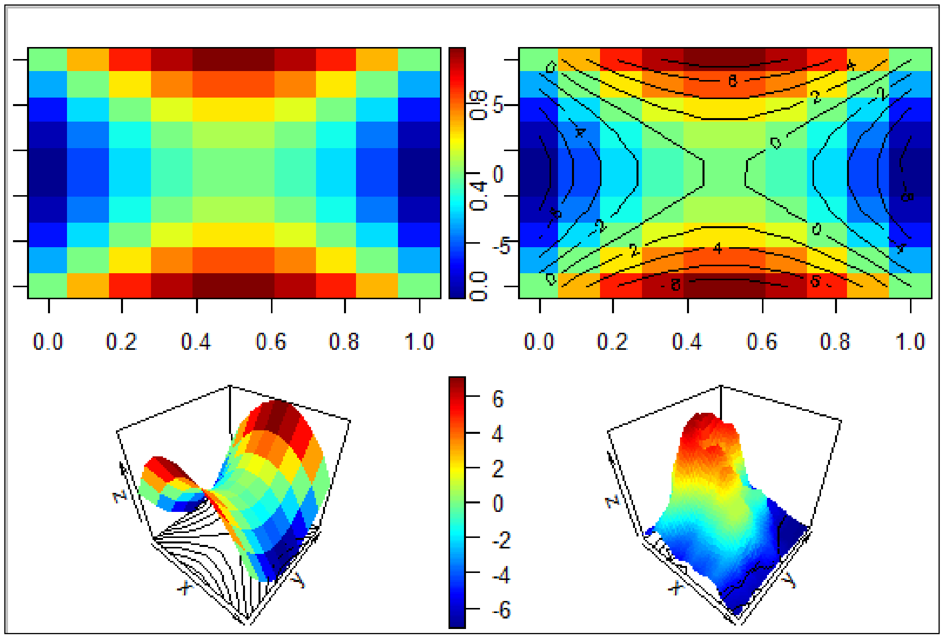
```
contour3D(x=x,y=y,z=z, colvar=z, bty="b2",dDepth =1,
  theta=60, nlevels = 20, colkey = FALSE)
contour3D(x=r,y=p,z=5, colvar=z, bty="b2",dDepth =1,
  theta=60, nlevels = 20, colkey = FALSE)
contour3D(x=5,y=p,z=r, colvar=z, bty="b2",dDepth =1,
  theta=60, nlevels = 20, colkey = FALSE)
contour3D(y=z, 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 4*, *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.

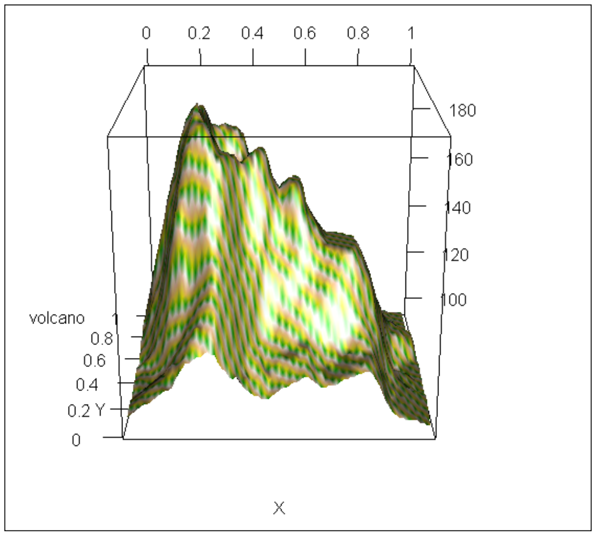
We can plot a three-dimensional plot in R using the `persp3D()` function. I would recommend readers to use the plot3D manual to better apply various options available with the `persp3D()` function. The `persp3D()` function uses two arguments, data and contour, to create a plot along with the contour 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.

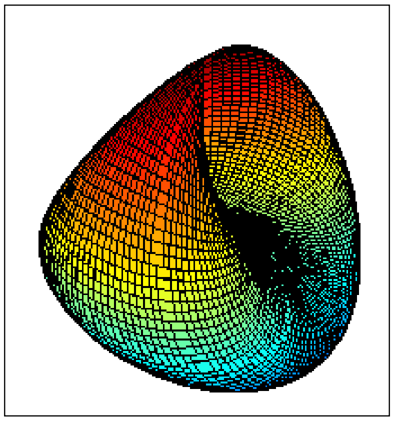


See also

- *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.hieght`, 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/`

7
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.

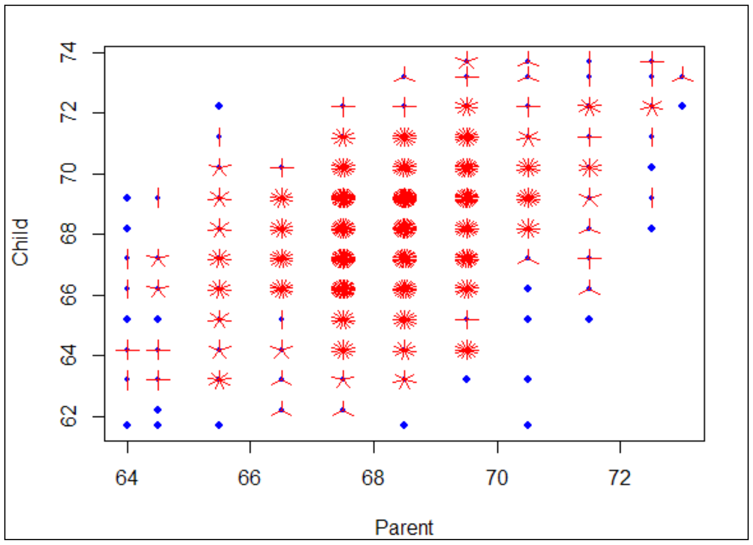
In this chapter, we will introduce plots such as sunflower plots, hexbins, calendar maps, coxcombs, and Chernoff faces, which are rarely used but are great tools to explore and present data. We will also explore network plots, pyramid plots, and radial plots, which have been utilized to convey information 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).

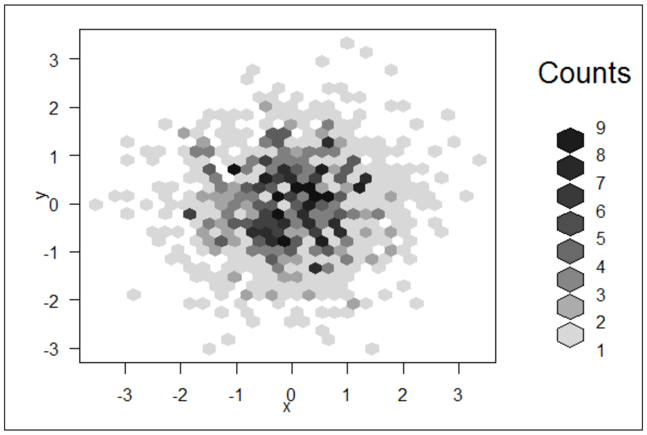
The `col` argument is used to specify the color of the plot and the `seg.col` argument applies color to the leaves. If the data is a single dot, it is just one observation; if the data has two lines, it represents two points; data with three lines represents three points, and so on. The `sunflowerplot()` 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`

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

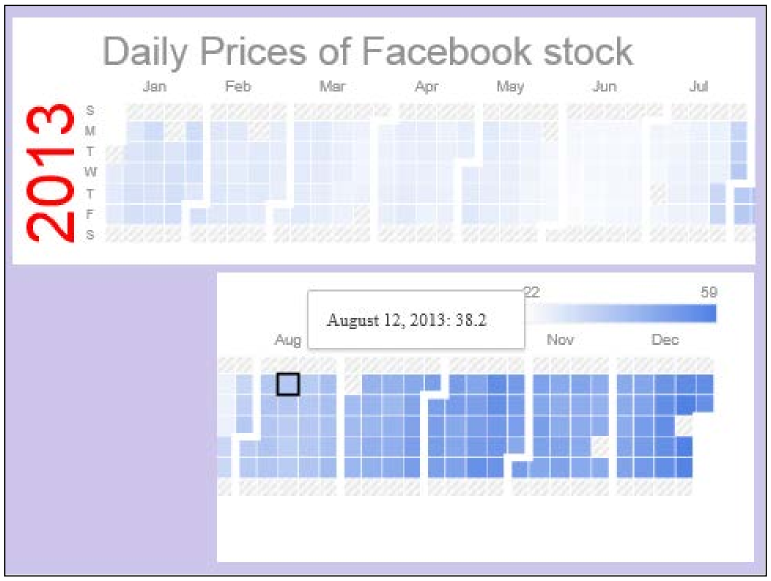
- It is possible to implement the hexbin plot using the `ggplot2` package. Please refer to the documentation at `http://docs.ggplot2.org/current/stat_binhex.html` to understand its implementation.
- The page at `http://indiemaps.com/blog/2011/10/hexbins/` provides a good explanation regarding hexbin 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)
```

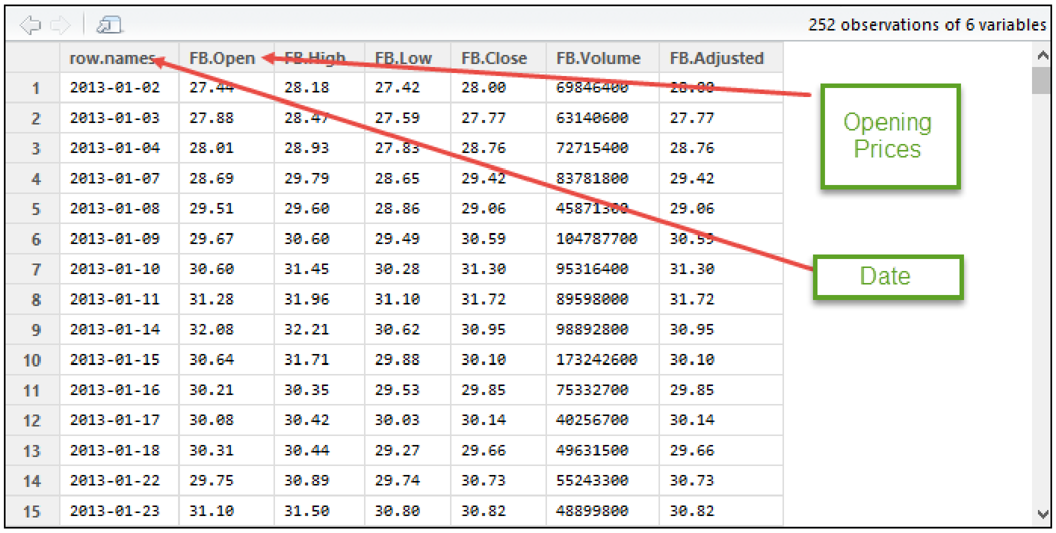
How it works...

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)`.



252 observations of 6 variables

	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	28.00
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	45871300	29.06
6	2013-01-09	29.67	30.60	29.49	30.59	104787700	30.59
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.

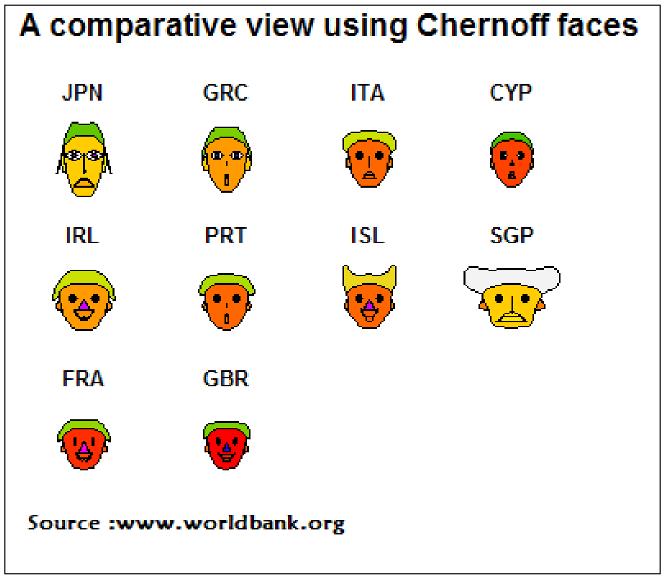
See also

- *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 `aplpack` package in R.

How to do it...

To create Chernoff faces, we would require to install the `aplpack` package in R and load it in our active R session using the `install.packages()` and `library()` functions:

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

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("worldecon.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 `aplpack` 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.

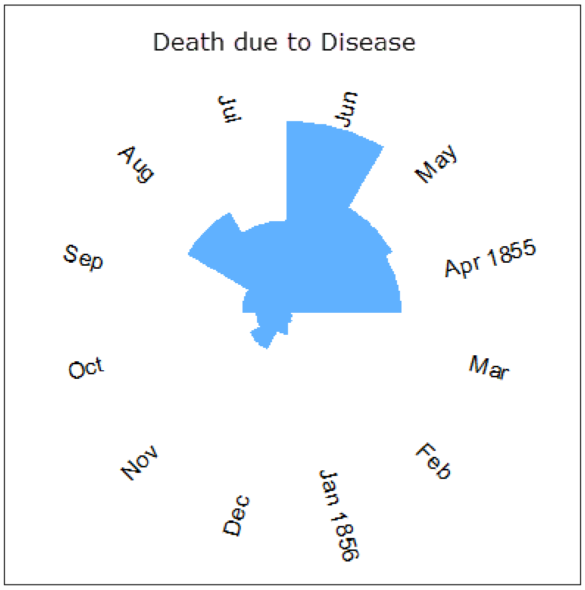
Please note that I have used the first 10 rows to plot the faces. If you use the entire dataset, you might get an error in R. This error can be avoided by simply expanding the R Studio's plot window. When R plots the faces, it will also plot the column headings corresponding to the facial expression. 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.

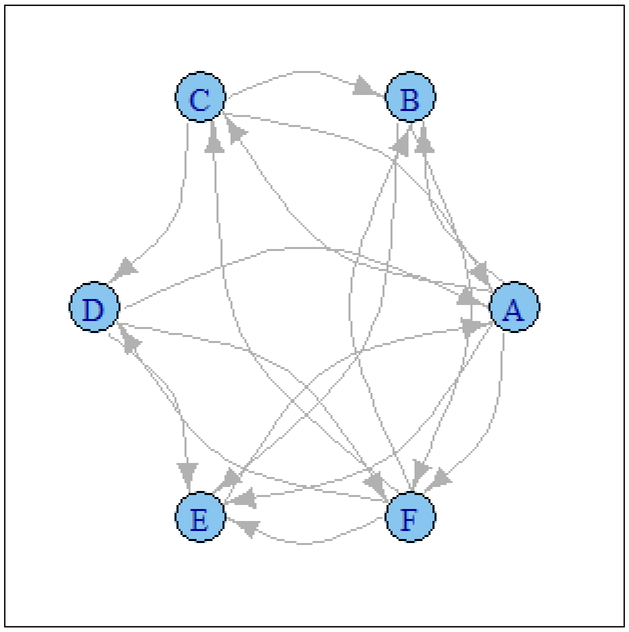
See also

- Spiechart (`http://spatial.ly/2014/01/coxcomb-plots/`) 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...

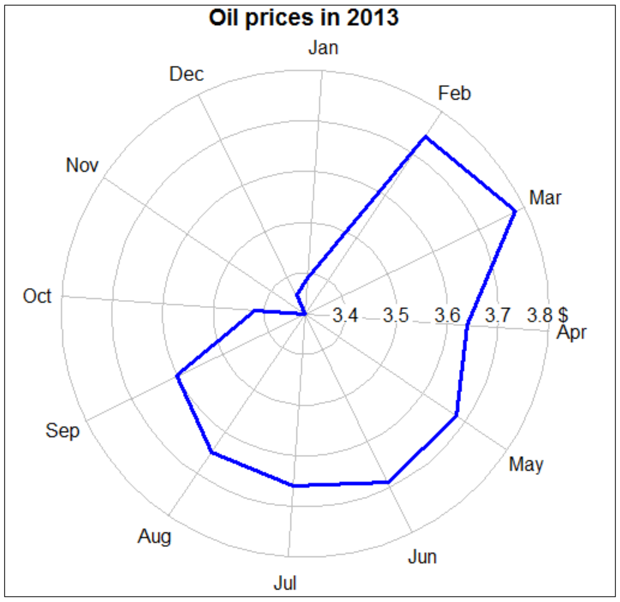
Readers who are interested in creating an interactive network plot can achieve this by using the `tkplot(g)` function in R. The function allows users to interactively change the layout as well as select edges and vertices.

See also

- Visualizing Friendships, available at `https://www.facebook.com/note.php?note_id=469716398919`.
- 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 cysles 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 1, A Simple Guide to R*.

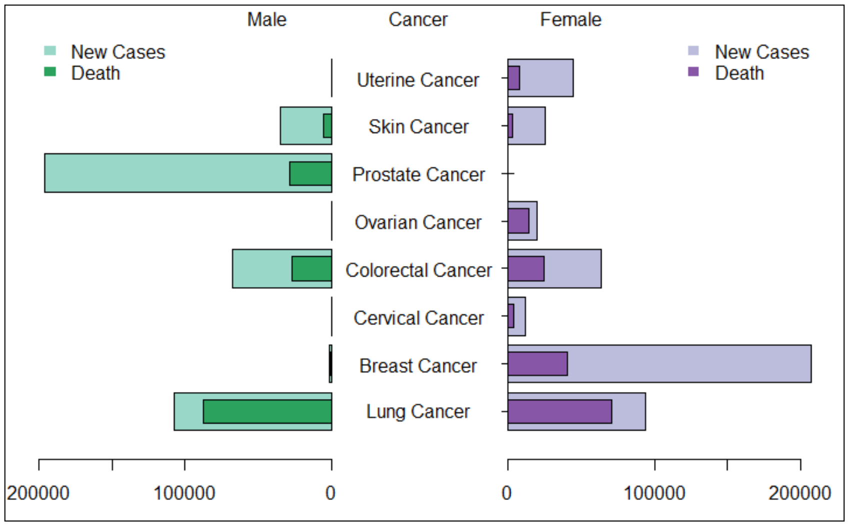
The plot generated does not allow us to understand the cycles. We have used a loop statement to show how new plots can be added to an existing plot. Nathan Yau uses radial plots to visualize cyclical data such as flight data.

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/`

8
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 2*, *Basic and Interactive Plots*, and they provide us with details, which are as follows:

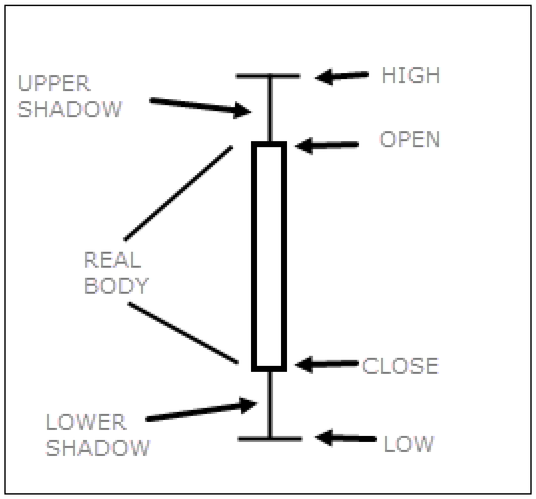
- When in the past the stock price of Microsoft was the lowest
- How often the higher prices are followed by higher prices or a dip
- How a stock's price reacts to different events such as recession, dividend announcements, quarterly results, or significant economic news

Similar information can also be studied using calendar maps and line plots discussed in *Chapter 7*, *Data in Higher Dimensions*, and *Chapter 2*, *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

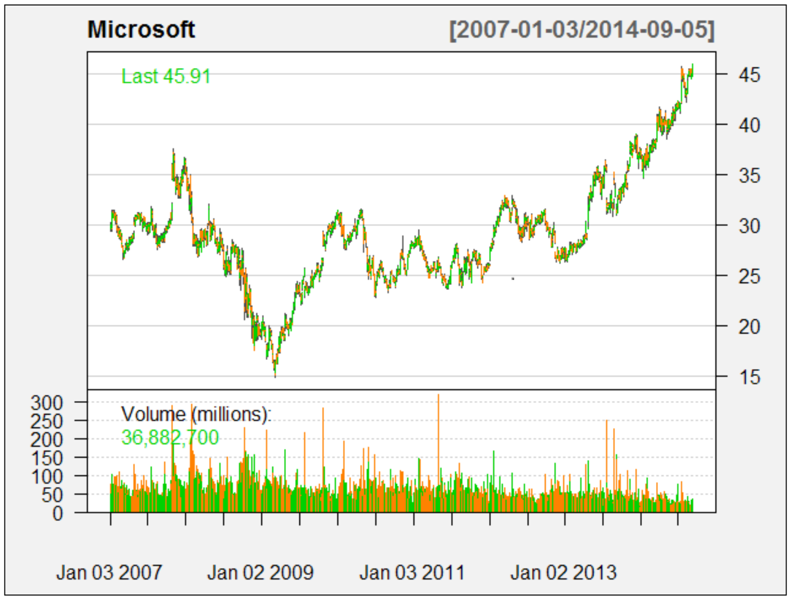
Candlestick plots have been widely used to display time series data related to financial markets. They are a combination of line charts and bar plots. They look very similar to box plots (discussed later in this chapter), but in reality, they do not share any similarity. A single candlestick is used 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()"))
```

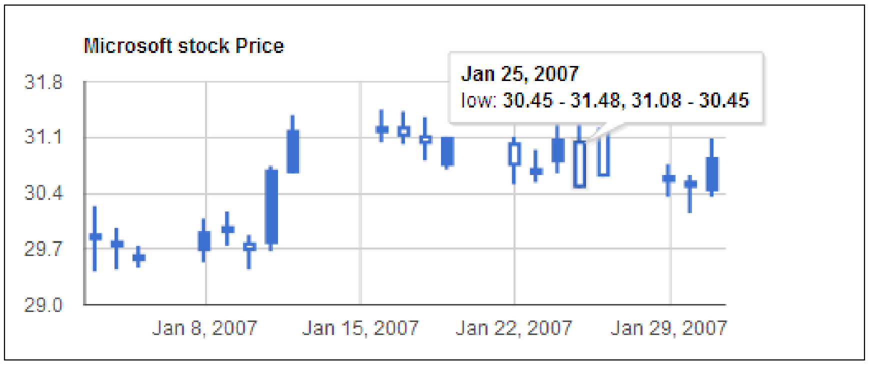
Each line will generate a different plot in R. Readers should note that they can also plot various technical trading tools along with the plot using the `TA` argument. In order to view the entire list of available `TA` options, readers should refer to the technical analysis and `chartSeries()` function 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 7*, *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:

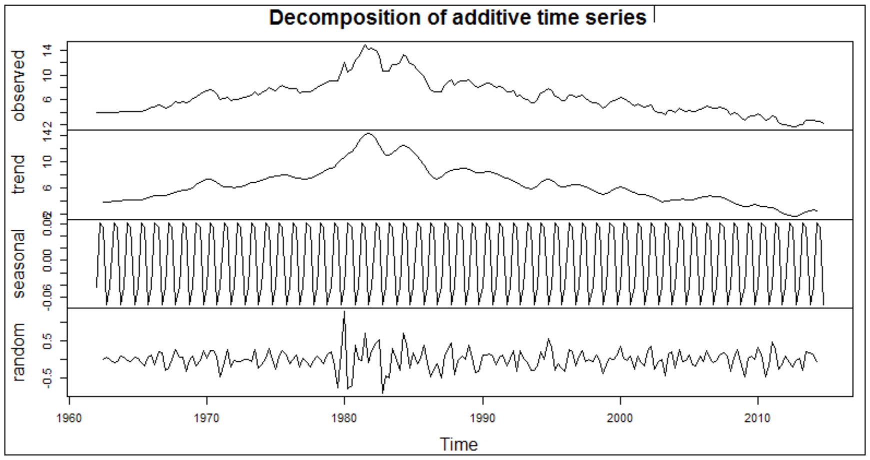
Xt = Mt+St+Zt

In this notation, the variables at time *t* are as follows:

- Xt: This is an observed series
- Mt: This is the trend
- St: This is the seasonal effect
- Zt: 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:

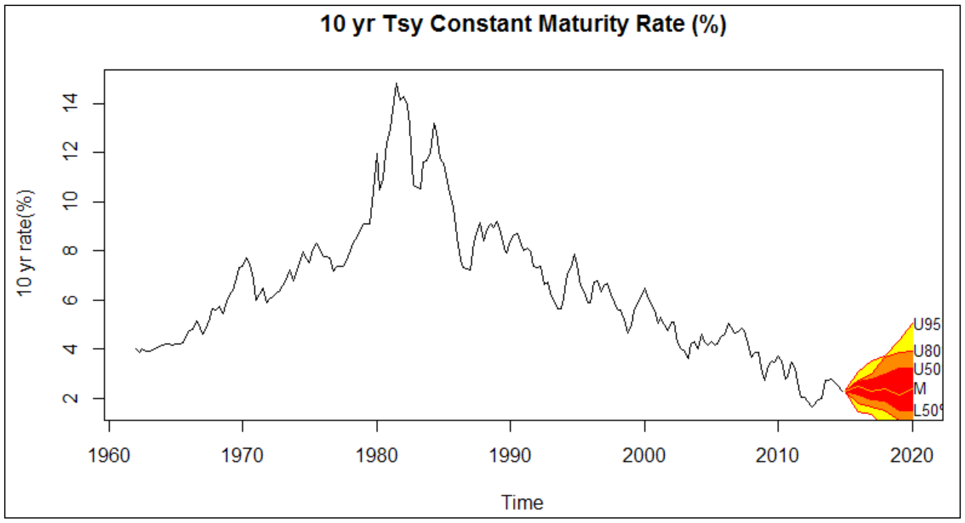
```
trend = d$trend # extracts the trend component
seasonal = d$seasonal # extracts the seasonal component
resid = d$random # extracts the residual
par(mfrow = c(2,2))
plot(datats) #plots the original series
plot(seasonal) # plots the seasonal component
plot(trend)# plots the trend component
plot(resid) # 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.

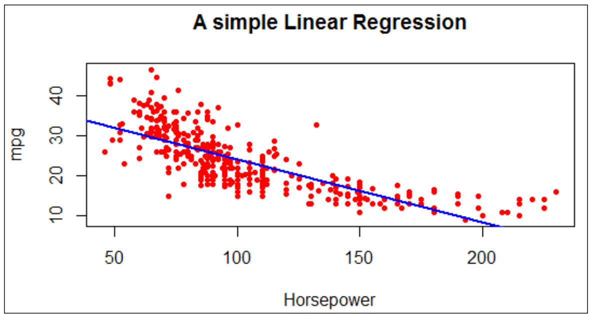
The first argument in the `fan()` function is our matrix that consists of our simulated data. The `type` argument refers to the type of percentiles to plot. Readers can choose between `interval` and `percentile`. The `start` argument refers to the time of the first distribution. The `anchor` argument 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 2*, *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)
```

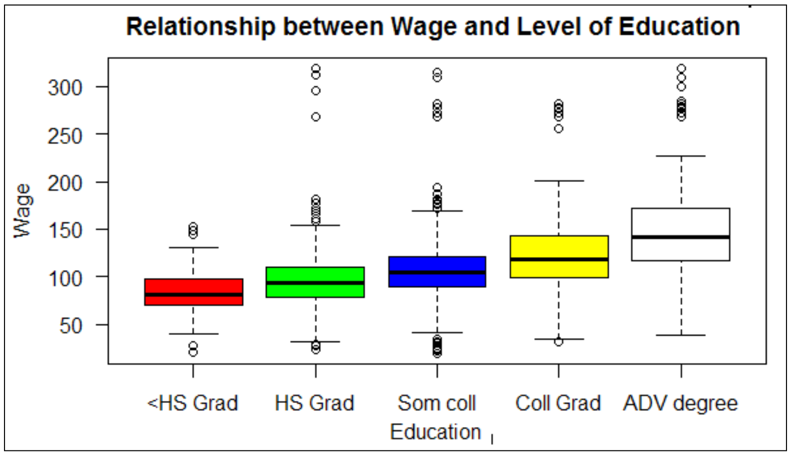
Readers can explore the `predict()` function by typing `?predict` in the R console window. Readers familiar with Generalized Least Squares can type `?glm` to learn more about the functionality in R.

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 2*, *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()`.

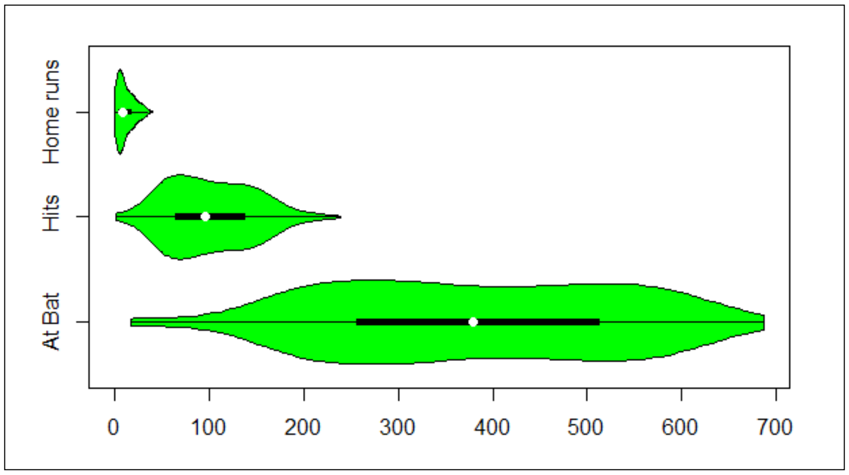
See also

- *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:

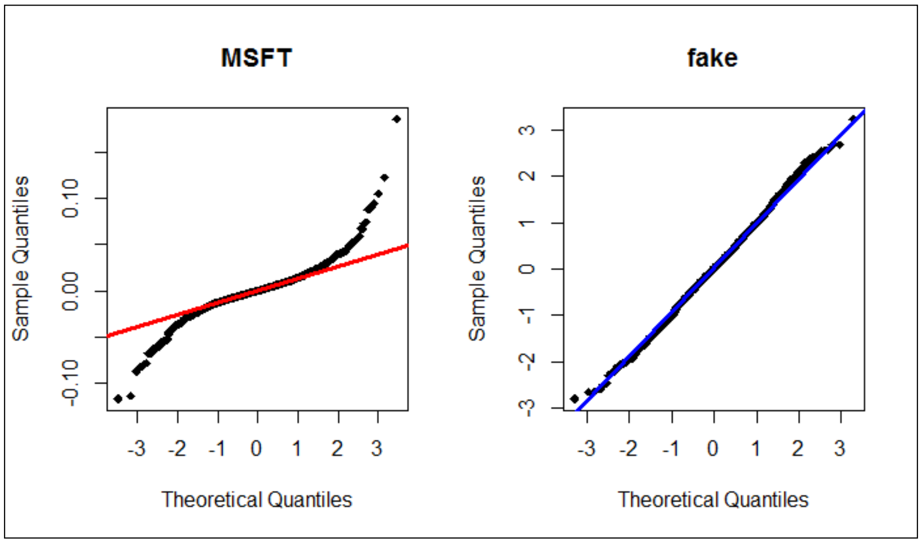
```
vioplot(Hitters$AtBat,Hitters$Hits,Hitters$HmRun, horizontal =
  TRUE, col =c("green"), names = c("At Bat"," Hits", "Home runs"))
```

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 `qqlines()` 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)
```

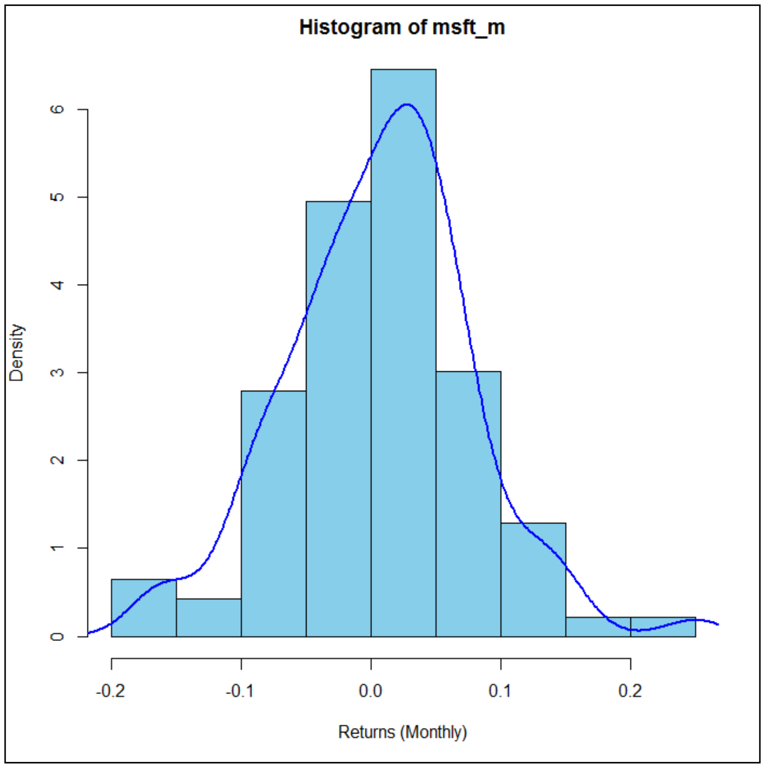
The `qqnorm()` function uses the daily returns on the Microsoft stock as its only argument. In order to show what a QQ plot of a normally distributed data looks like, I have generated fake data with a mean of 0 and standard deviation of 1. We can clearly observe that daily returns on the Microsoft stock 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...

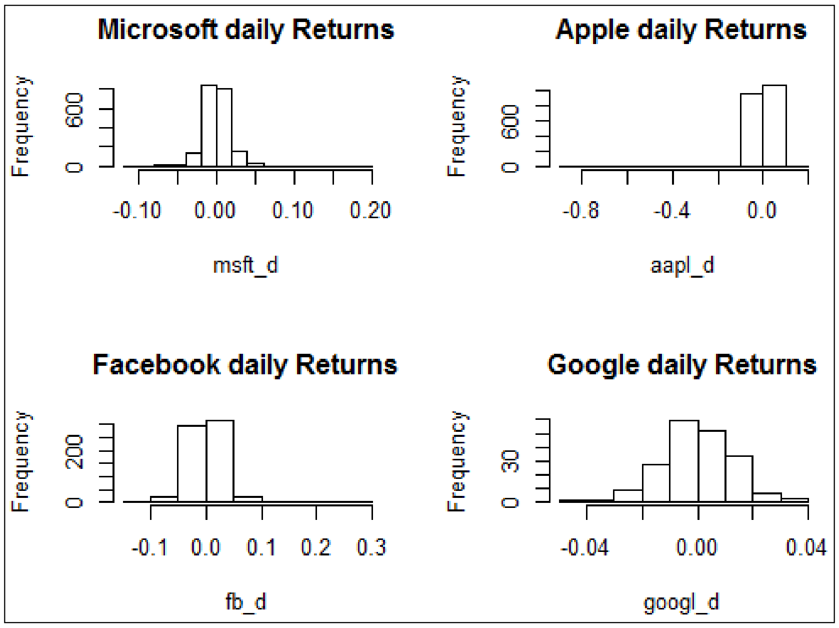
One of the very good implementations of a histogram is when multiple histograms are displayed in the same plot. This allows us to perform a comparative analysis of histograms in the same image. Looking at the visualization gives us a very good idea of how the equity prices have performed in the technology 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.

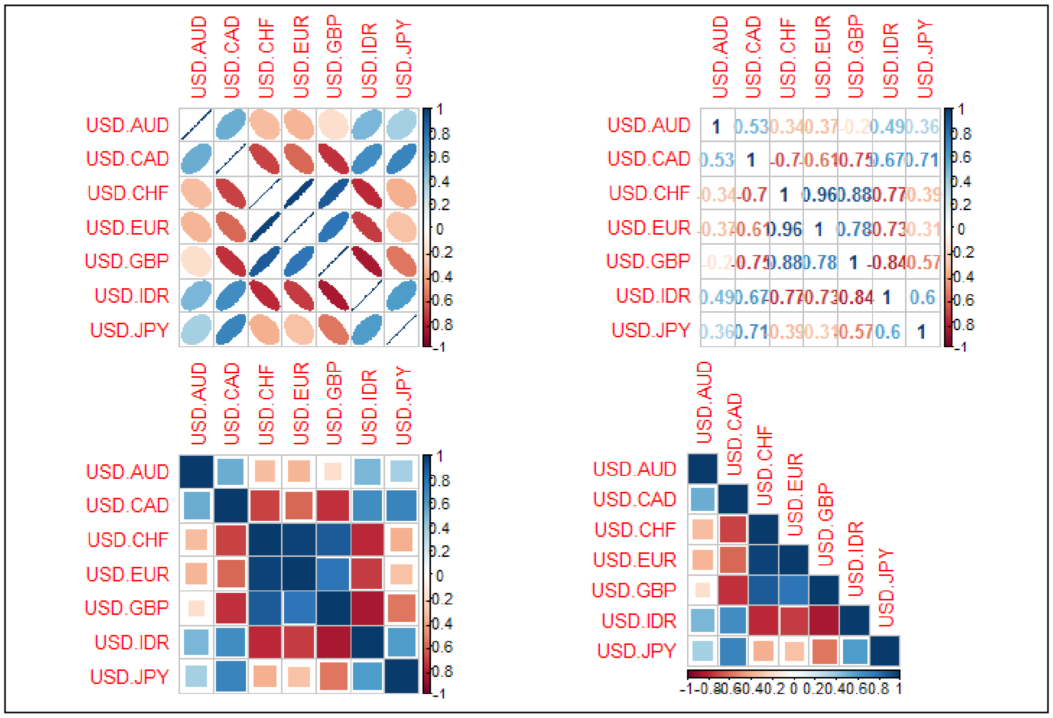


See also

- 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)
```

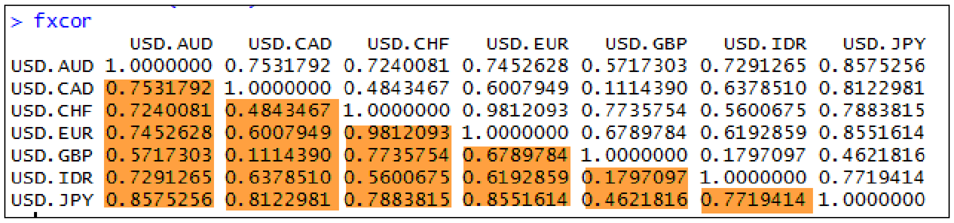
The `fxcor` object is now passed as an argument under the `corrplot()` function to generate the correlation plots:

```
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.

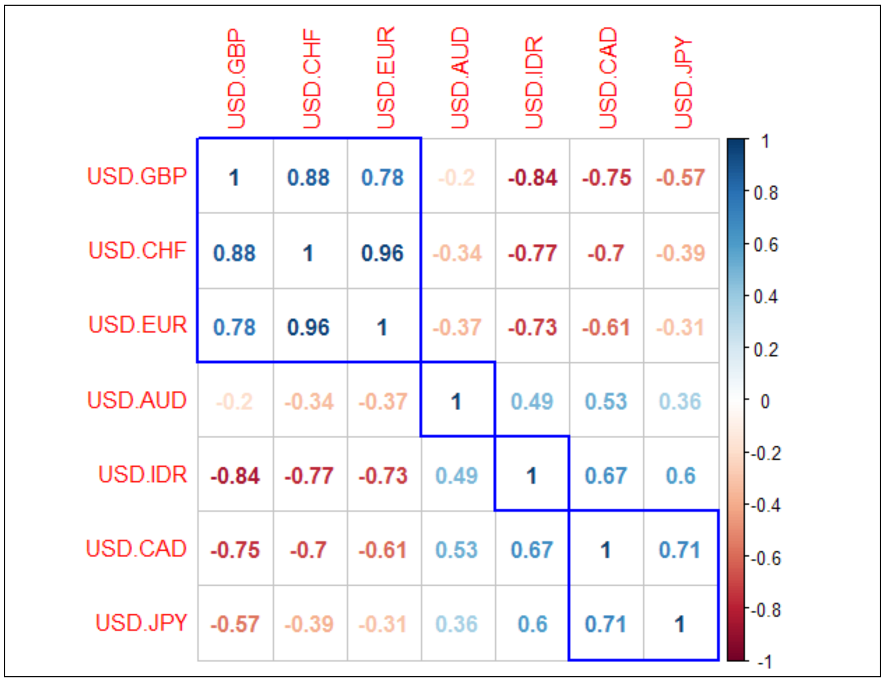


```
> fxcor
          USD.AUD   USD.CAD   USD.CHF   USD.EUR   USD.GBP   USD.IDR   USD.JPY
USD.AUD 1.0000000 0.7531792 0.7240081 0.7452628 0.5717303 0.7291265 0.8575256
USD.CAD 0.7531792 1.0000000 0.4843467 0.6007949 0.1114390 0.6378510 0.8122981
USD.CHF 0.7240081 0.4843467 1.0000000 0.9812093 0.7735754 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 2, 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.



	USD.GBP	USD.CHF	USD.EUR	USD.AUD	USD.IDR	USD.CAD	USD.JPY
USD.GBP	1	0.88	0.78	-0.2	-0.84	-0.75	-0.57
USD.CHF	0.88	1	0.96	-0.34	-0.77	-0.7	-0.39
USD.EUR	0.78	0.96	1	-0.37	-0.73	-0.61	-0.31
USD.AUD	-0.2	-0.34	-0.37	1	0.49	0.53	0.36
USD.IDR	-0.84	-0.77	-0.73	0.49	1	0.67	0.6
USD.CAD	-0.75	-0.7	-0.61	0.53	0.67	1	0.71
USD.JPY	-0.57	-0.39	-0.31	0.36	0.6	0.71	1

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`

9
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:

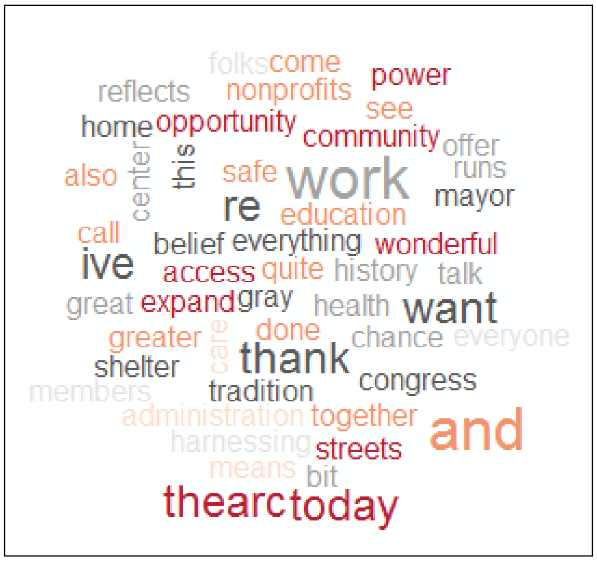
- Visualizing text provides us with information, trends, and changing patterns in literature
- Visualizing Twitter feeds can help us in revealing behavioral and social patterns
- Visualizing books reveals the writing styles of different authors
- Visualizing political speeches helps us to understand 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...

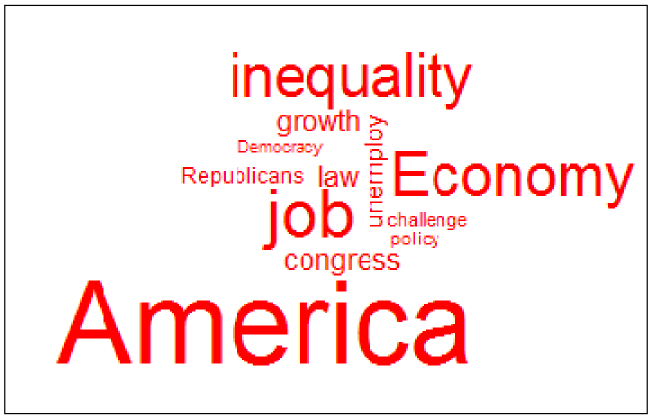
The first step in our recipe is to load the required packages in R. We then create a color palette using the `RcolorBrewer` package. The `brewer.pal()` function has two arguments; the first argument is the number of different colors we would like to use in our visualization and the second argument is 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.

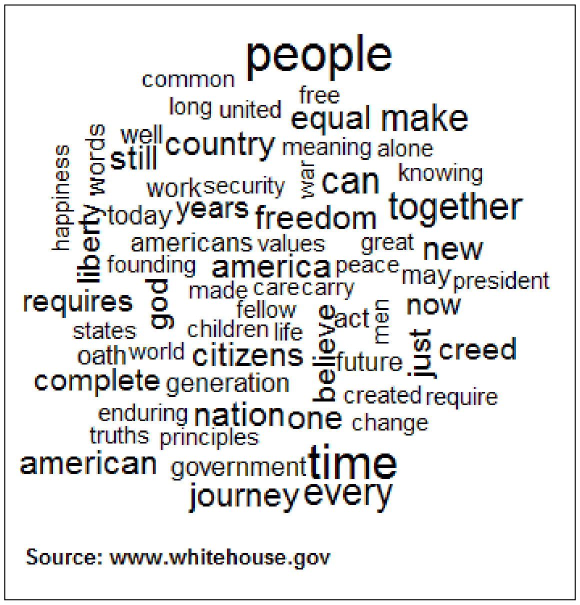
We observe that words such as "inequality" and "economy" are used more often, but words such as "law" or "policy" to decrease inequality are repeated only 9 and 2 times respectively.

See also

- `www.wordle.net` 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/manyeyes/` 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 10*, *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 `readlLines()` 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.

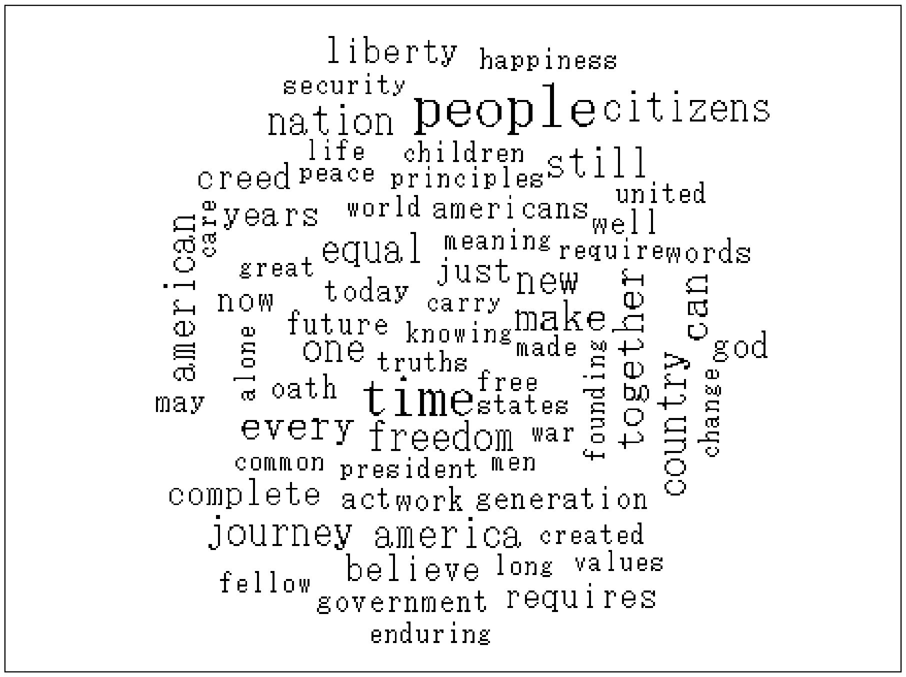
Once we have the document ready, we can modify the document by removing the stop words, punctuation, numbers, and so on. The `tm_map()` function allows us to clean our data by passing various arguments. We can convert all the words in our text document to lowercase by passing the `content_transformer(tolower)` 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:

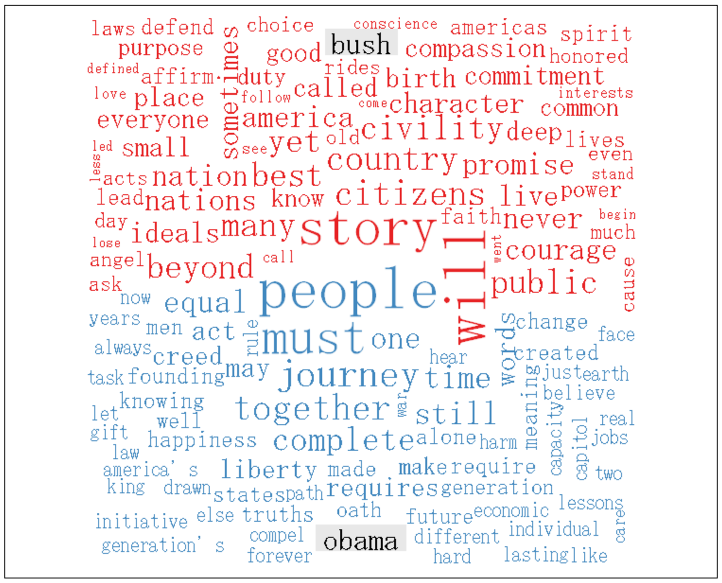
```
par(family = "JP1")
wordcloud(doc, scale= c(2,0.5))
```

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.

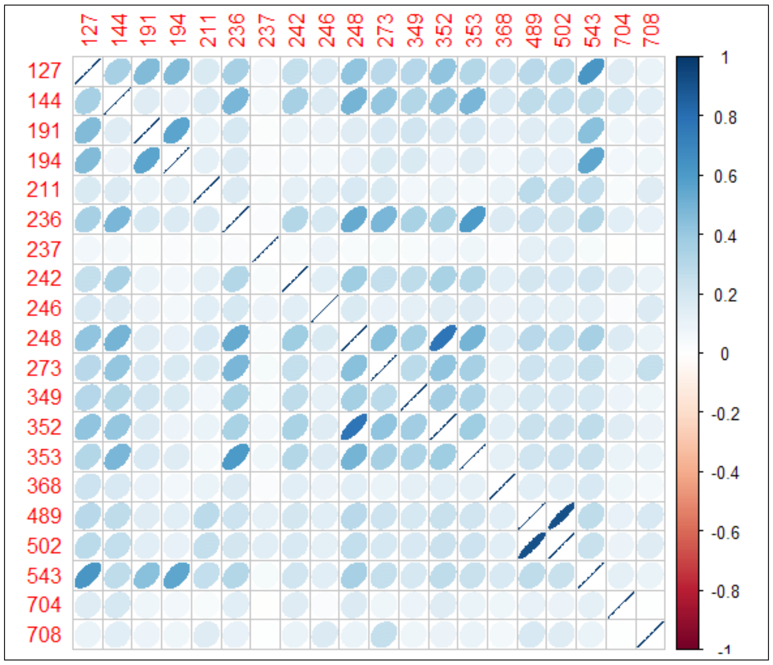
See also

- 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...

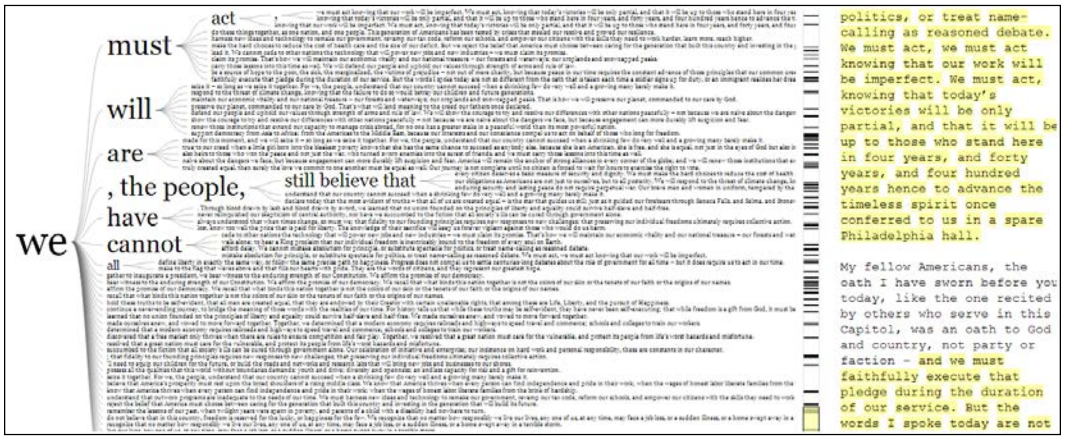
In the previous recipes, we have discussed at length the various functions used to clean our data and the steps necessary to generate a term document matrix. The `findFreqTerms()` function takes the data as its first argument and the frequency of occurrence as the second argument. The `findFreqTerms()` 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 8*, *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.



politics, or treat name-
calling as reasoned debate.
We must act, we must act
knowing that our work will
be imperfect. We must act,
knowing that today's
victories will be only
partial, and that it will be
up to those who stand here
in four years, and forty
years, and four hundred
years hence to advance the
timeless spirit once
conferred to us in a spare
Philadelphia hall.

My fellow Americans, the
oath I have sworn before you
today, like the one recited
by others who serve in this
Capitol, was an oath to God
and country, not party or
faction - and we must
faithfully execute that
pledge during the duration
of our service. But the
words I spoke today are not

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`.

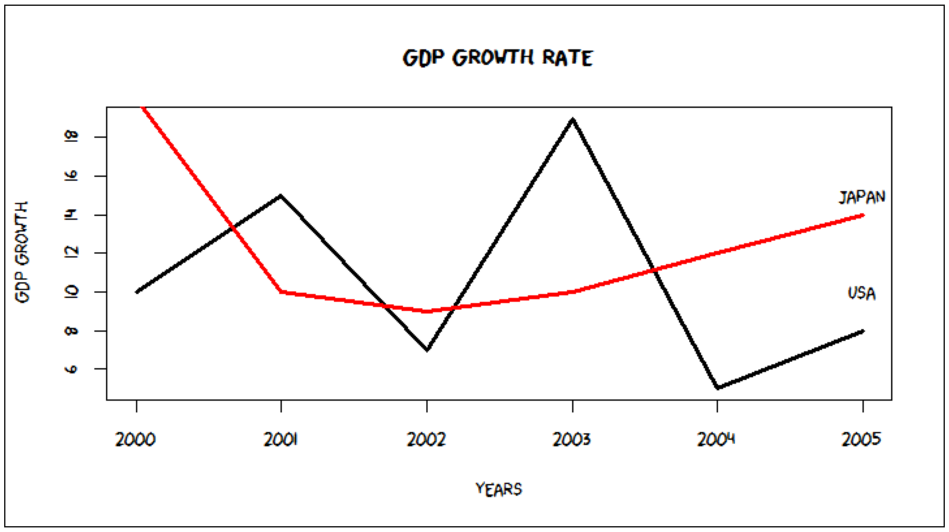
This link allows users to generate a word tree by pasting the data at the bottom of the website.

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`
- *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.farme()` 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"))
```

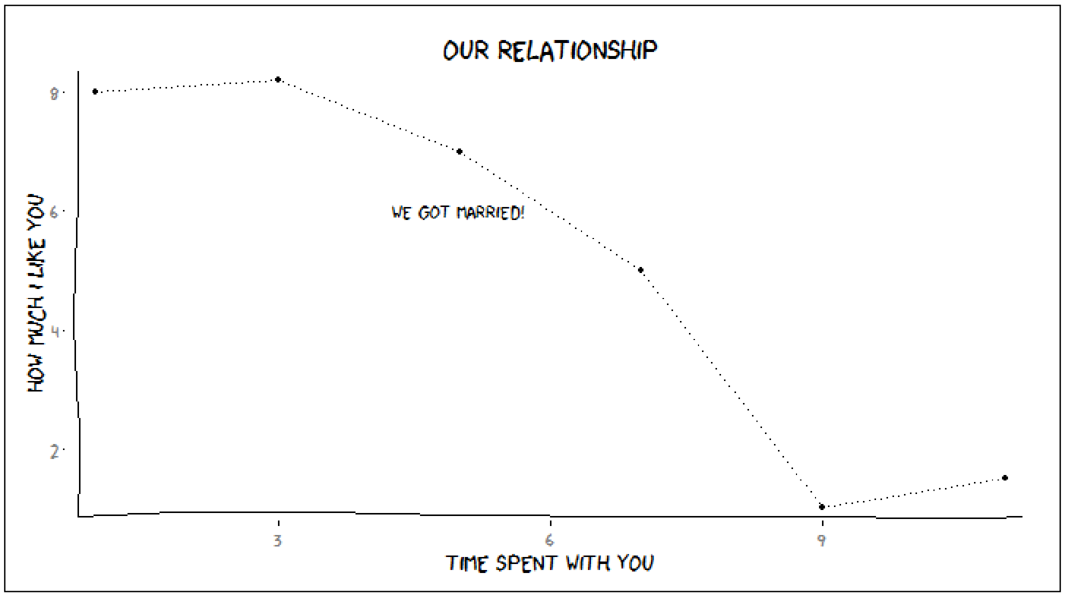
Over the course of the book, we have studied the application of the `cbind()` and `data.frame()` functions. The `par()` function allows us to specify various elements for a plot. The `family` argument in the `par()` function allows us to set a custom font for our chart. In our case, it is `xk`, as defined 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/`

10 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.

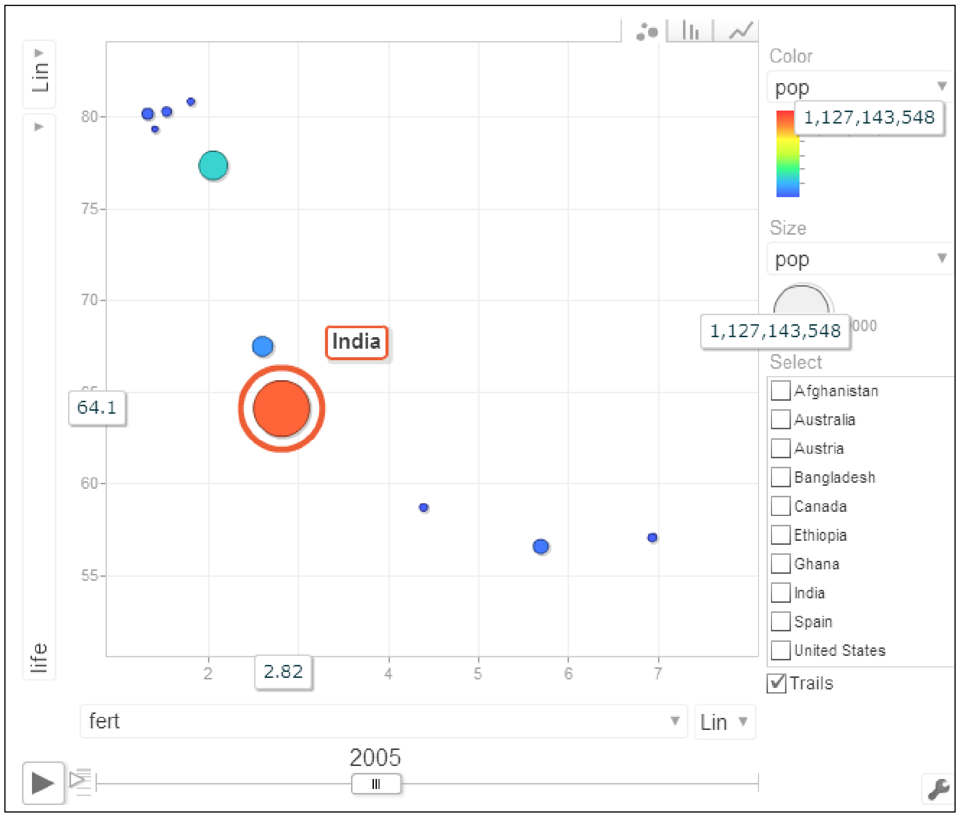
We will learn how to generate powerful and meaningful presentations using the `slidify` package. We will also learn to create web applications and animation in R using the `shiny` and `googleVis` packages. Finally, we learn about API and XML and how they can be effectively used to scrap data from the 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 6*, *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.periscopic.com/?year=2013`
- Nytimes at `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`

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.

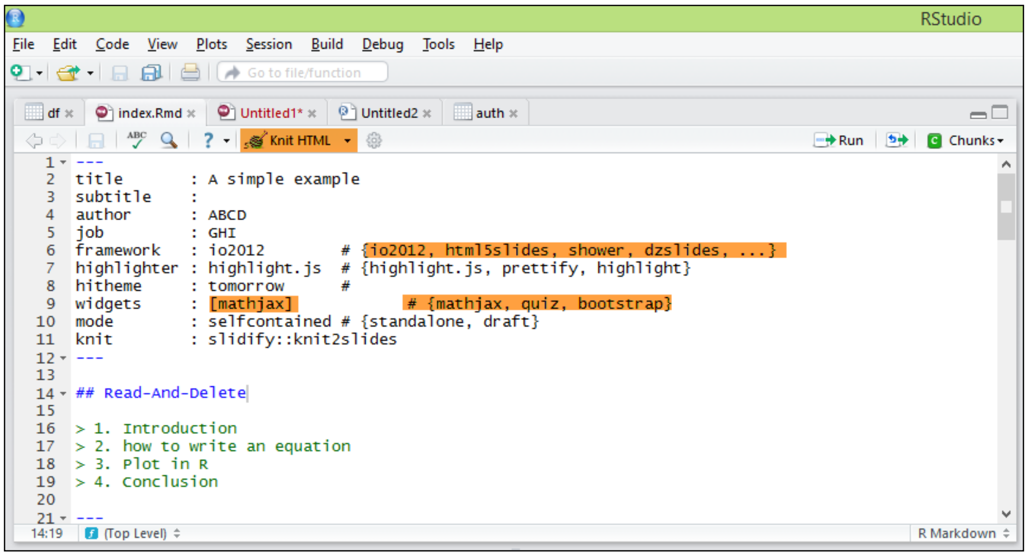
In order to install the package from GitHub, we would use the install_github('slidify','ramnathv') function. We can now load our slidify library in R using the library(slidify) function.

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.


RStudio
File Edit Code View Plots Session Build Debug Tools Help
Go to file/function
df index.Rmd Untitled1* Untitled2 auth
Knit HTML
Run
Chunks
14:19 (Top Level)
R Markdown

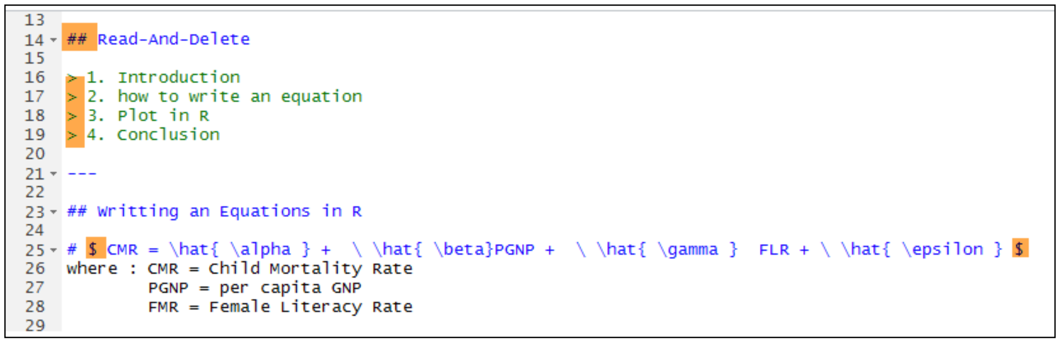
```
---
title       : A simple example
subtitle    :
author      : ABCD
job         : GHI
framework   : io2012        # {io2012, html5slides, shower, dzslides, ...}
highlighter : highlight.js  # {highlight.js, prettify, highlight}
hitheme     : tomorrow      #
widgets     : [mathjax]            # {mathjax, quiz, bootstrap}
mode        : selfcontained # {standalone, draft}
knit        : slidify::knit2slides
---

## Read-And-Delete

> 1. Introduction
> 2. how to write an equation
> 3. Plot in R
> 4. Conclusion

---
```

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:



```

## Read-And-Delete

> 1. Introduction
> 2. how to write an equation
> 3. Plot in R
> 4. Conclusion

---

## Writting an Equations in R

# $ CMR = \hat{ \alpha } +  \ \hat{ \beta}PGNP +  \ \hat{ \gamma }  FLR + \ \hat{ \epsilon } $
where : CMR = Child Mortality Rate
        PGNP = per capita GNP
        FMR = Female Literacy Rate

```

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:

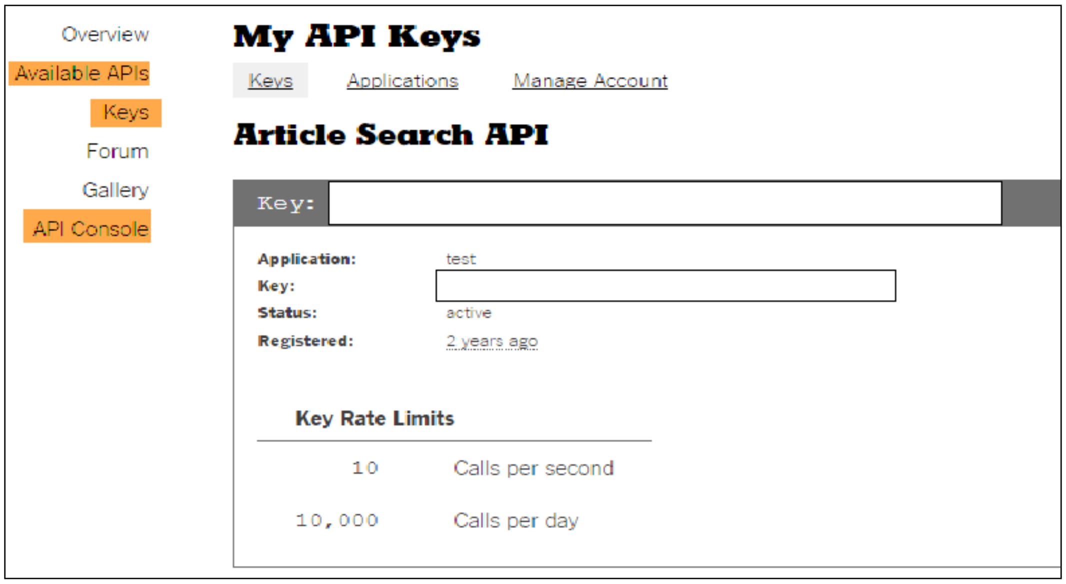
- Most of the news agencies allow users to download data related to articles, news, and so on
- Many social networking websites such as Twitter and Facebook allow their users to download data related to status updates, friend lists, photos, and links
- Google API allows its users to download various 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/`.



My API Keys

Article Search API

Key:

Application:	test
Key:	
Status:	active
Registered:	2 years ago

Key Rate Limits

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.

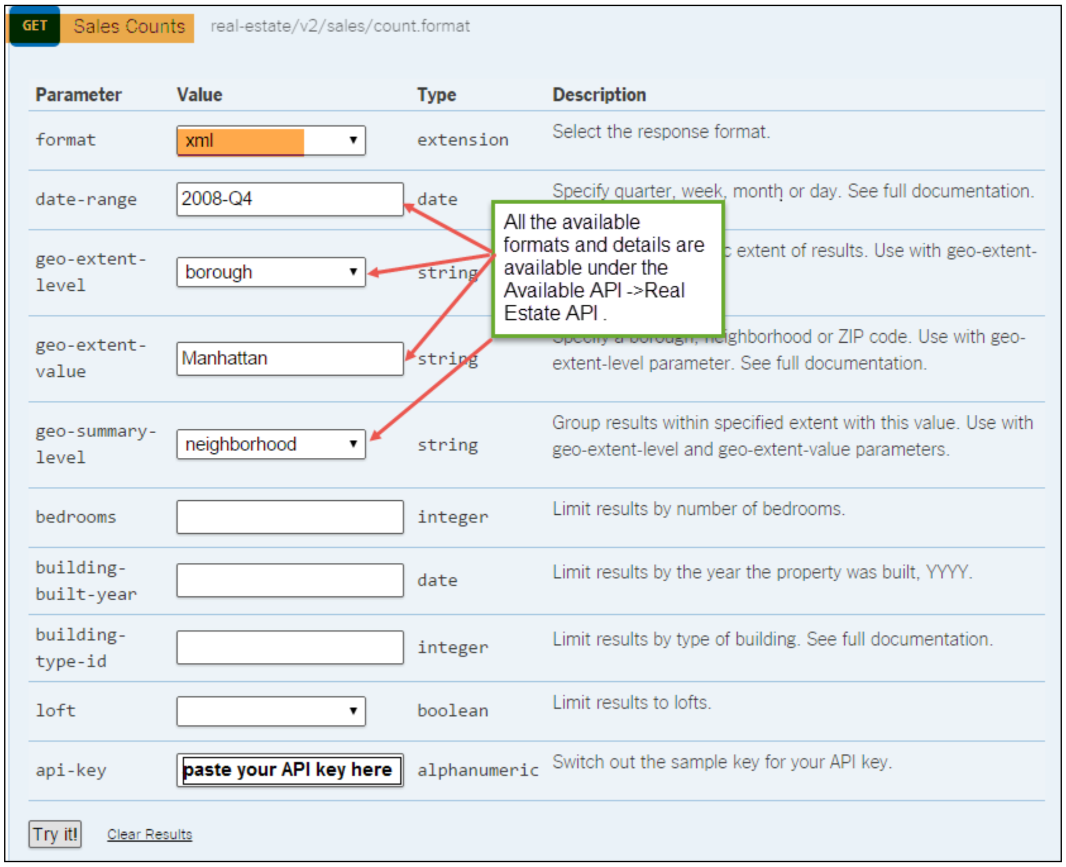
The **Keys** section lists all the registered keys. The New York Times requires all the users to register a key for every API before it allows them to download the data. To register a key, click on the **Real Estate API** and type a name for your API; in our case, I have registered it as `test`.

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**!

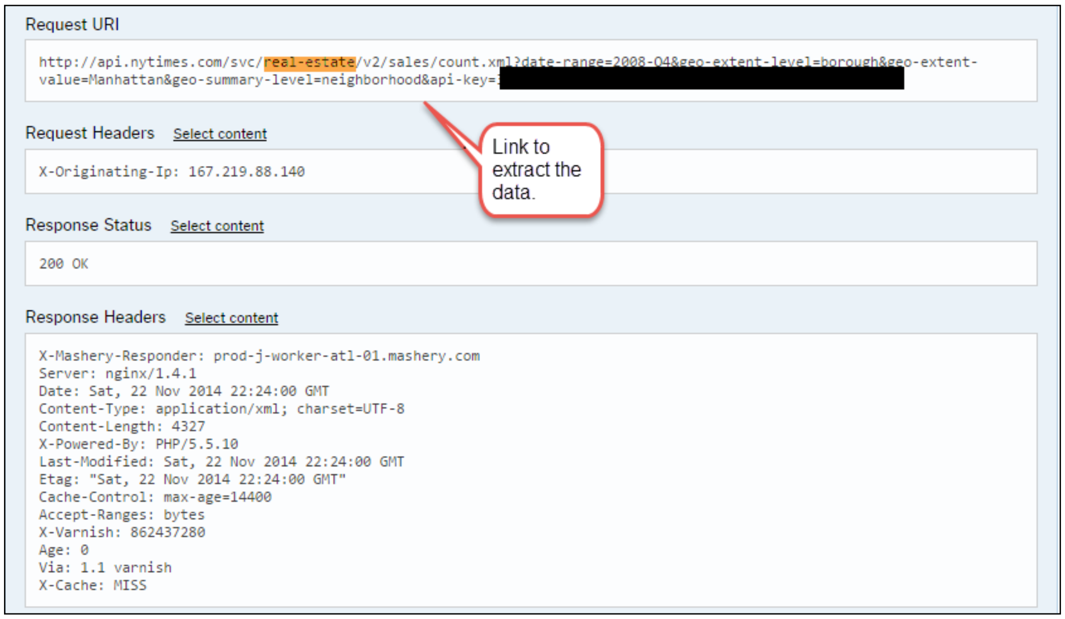

GET
Sales Counts
real-estate/v2/sales/count.format
All the available formats and details are available under the Available API ->Real Estate API .
Try it!
Clear Results

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	extent of results. Use with geo-extent-
geo-extent-value	Manhattan	string	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	paste your API key here	alphanumeric	Switch out the sample key for your API key.

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
```

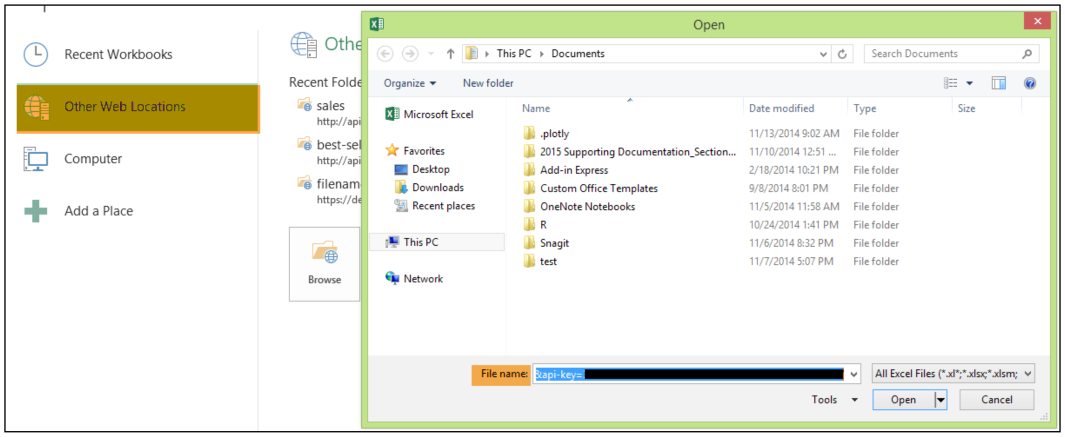
Response Status Select content

```
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
```

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`.



Name	Date modified	Type	Size
.plotly	11/13/2014 9:02 AM	File folder	
2015 Supporting Documentation_Section...	11/10/2014 12:51 ...	File folder	
Add-in Express	2/18/2014 10:21 PM	File folder	
Custom Office Templates	9/8/2014 8:01 PM	File folder	
OneNote Notebooks	11/5/2014 11:58 AM	File folder	
R	10/24/2014 1:41 PM	File folder	
Snagit	11/6/2014 8:32 PM	File folder	
test	11/7/2014 5:07 PM	File folder	

File name: &api-key=[redacted]

All Excel Files (*.xl*;*.xlsx;*.xlsm;

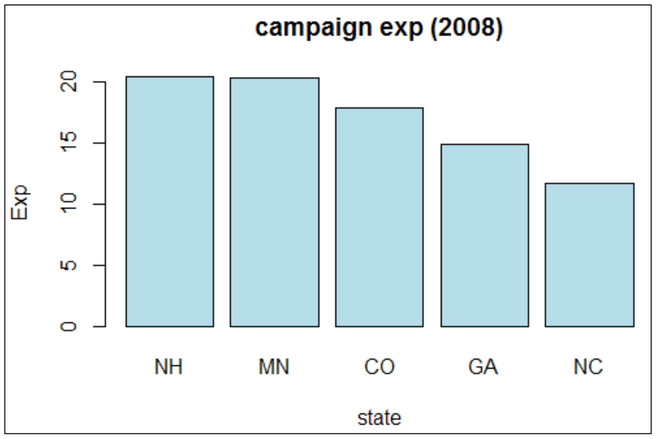
We can now use either Excel or R to generate the charts. To generate a chart in R, we will edit and format the data, save it as a CSV file, and import it in R.

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 1*, *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 1*, *A Simple Guide to R* to understand the matrix/vector manipulation.

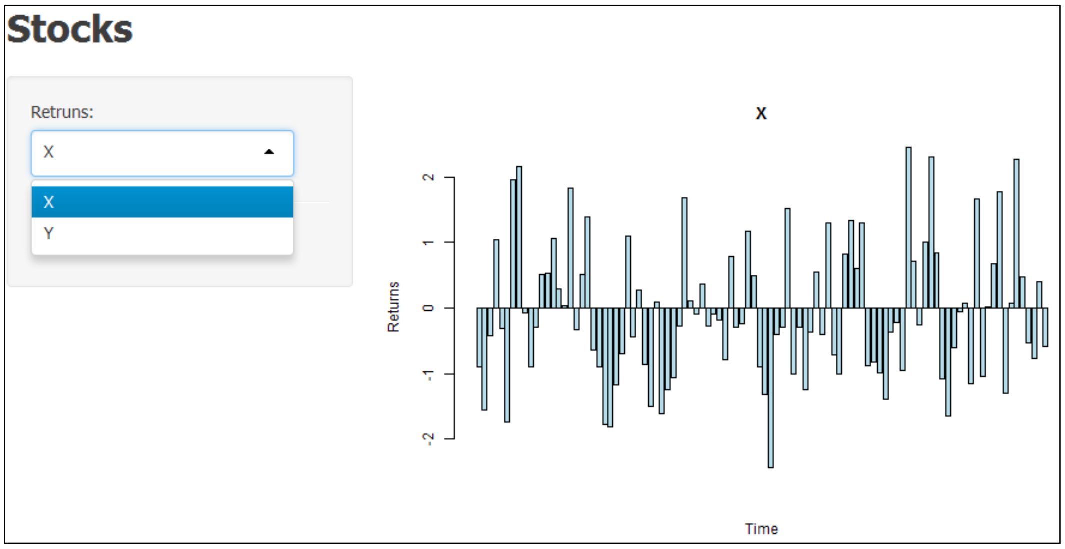
We can now pass the sample data as a first argument to the `barplot()` function. The second argument is `beside = TRUE`, which forces R to plot the bars side by side instead of stacking them. All other arguments in the `barplot()` function are self-explanatory.

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.



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`.

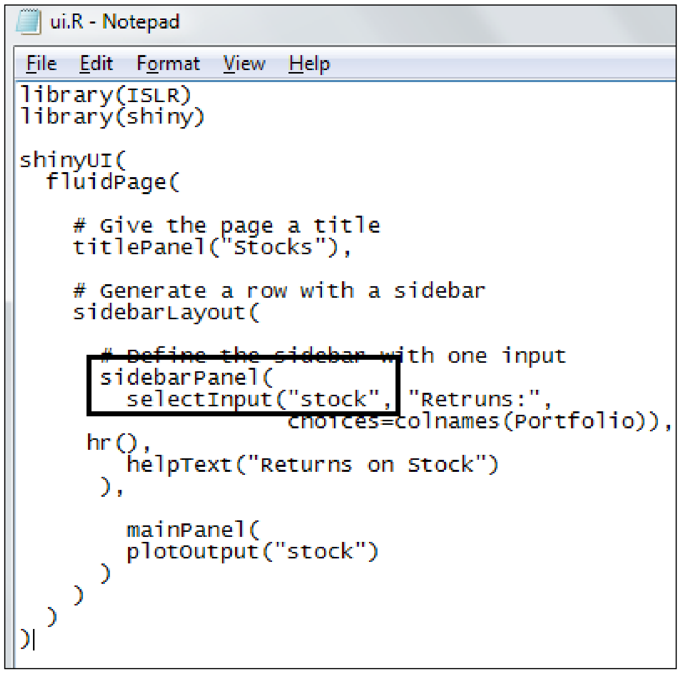
The server-side logic is defined in the `server.R` file under `shinyServer(function (input,output))`. The plot defined under the `renderPlot()` function gets generated in the main panel of the user interface.

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:



```
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", "Retruns:",
                    choices=colnames(Portfolio)),
     hr(),
        helpText("Returns on Stock")
      ),

        mainPanel(
        plotOutput("stock")
      )
    )
  )
)
```

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`

Index

Symbols

3D contour plot
  surface plot, integrating 130-133
  generating 128-130
3D histogram 126-128
3D pie chart 124, 125
3D scatter plot
  constructing 118-121
  generating, with text 121-124
3D surface plot
  animating 133-135

A

abline() function 34
animated coxcomb plot
  URL 149
animated plots
  creating, in R 200-202
API 205-209
apply() function 21, 22

B

bar plot
  about 38-40
  constructing, XML used 209-211
box plot
  constructing 168, 169
bubbles
  maps, constructing with 88-91

C

candlestick plot
  generating 158-161
cartograms
  creating 98-100
cbind function 16
cex argument 28
chartSeries() function 160
Chernoff faces
  creating, in R 145, 146
choropleth maps 82, 83
clockwise argument 104
col argument 139
colkey argument 119
colors
  dendrograms, creating with 62-64
colors argument 81
comparison cloud
  generating 188-190
contour() function 86
contour maps
  defining 85-88
cor() function 193
correlation plot
  constructing 191-194
  generating 176-178
corrplot() function 193
coxcomb plots
  constructing, in R 147, 148
custom fonts
  plots, generating with 194-196
customized colors
  heat map, generating with 68, 69

D

D3
  URL 48
data
  exporting, in R 17

importing, in R 16, 17
data frame
editing, in R 15
data.frame() function 52
data types, R 10
decomposed time series
generating 163-166
decompose() function 164
dendrogram
constructing 58-61
creating, with colors 62-64
creating, with labels 62-64
density plot
generating 173, 174
document
word cloud, constructing from 185-187
donut plots
creating 109-112

E

Extensible Markup Language. *See* **XML**
Extensible Time Series. *See* **XTS**

F

fan plot
constructing 115
packages 115
fill argument 28, 155
findAssocs() function 193
findFreqTerms() function 192
floating.pie() function 107, 112
Flowing Data
URL 9
function
writing, in R 18

G

Gallup
URL 82
Gantt chart
generating, in R 49, 50
gephi package
URL 151
get_map() function 87
ggplot2
URL 142, 198
Google Correlate
URL 179
googleVis developer
URL 37
googleVis package
about 26, 40, 90, 143, 201
URL 44
Gross Domestic Product (GDP) 29
Gun violence
URL 200
gvisBubbleChart function 54
gvisBubbleChart function, attributes
colorvar 54
data 54
idvar 54
sizevar 54
xvar 54
yvar 54
gvisCalendar() function 144
gvisMotionchart() function 201
gvisTreeMap, arguments
colorvar 77
data 77
idvar 77
parentvar 77
sizevar 77

H

heat map
creating 64-67
generating 70-73
generating, with customized colors 68, 69
heatmap() function 66
hexbin manual
URL 141
hexbin plot
creating 140, 141
hist() function 174
histograms
merging 50-52

I

if else statements
writing, in R 19

Inkspace
URL 95
install.packages() function 154, 195
integrated dendrogram
generating 70-73
interactive bar plot 40-43
interactive bubble plot
creating 53-55
interactive calendar maps
generating 142-144
interactive candlestick plots
generating 161, 162
interactive plots
advantages 161
creating 109-112
interactive scatter plot
generating 35-37

K

Kosara
URL 145

L

label orientation
URL 67
label.prop argument 148
labels
dendrograms, creating with 62-64
pie chart, constructing with 106-108
lapply() function 21, 22
layout() function 130
library() function 8, 154, 195
line plot
about 44-46
defining 46-48
Locally Weighted Scatterplot Smoothing (LOESS) 31
loops, R 20

M

mainpanel() function 213
map() function 93
maps
about 79
constructing, with bubbles 88-91
text, integrating with 92, 93
matrix
editing, in R 14

N

NeatMap manual
URL 74
nested loops, R 20
network plots
constructing 149, 150
notations
URL 205
Nytimes
URL 200

O

OECD
URL 46
options argument
about 42, 81, 90
URL 202
order() function 39

P

packages
installing, in R 8, 9
parentvar argument 77
par() function
about 63, 108, 195
used, in R 22
paste() function 103
pheatmap() function 66
phrase tree
constructing 191-194
pie chart
constructing, with labels 106-108
generating 102-105
URL 101
pie.labels() function 107
plot3D manual
URL 118, 121, 124
plot() function 28, 107, 144
plotrix manual 153
plots
generating, with custom fonts 194-196

saving 23
png() function 23
points
connecting, in scatter plot 32-35
presentation
creating, in R 202-205
processing
URL 48
pyramid plot
generating 154, 155
pyramid.plot() function 155

Q

QGIS
URL 95
QQ plot
generating 171, 172
quantmod package 143
quantmod website
URL 161

R

R
animated plots, creating in 200-202
Chernoff faces, creating 145, 146
coxcomb plots, constructing 147, 148
data, exporting 17
data frame, editing 15
data, importing 16, 17
data types 10
function, writing 18
Gantt chart, generating 49, 50
help, obtaining 8, 9
if else statements, writing 19
loops 20
matrix, editing 14
nested loops 20
packages, installing in 8, 9
par() function, using 22
presentation, creating in 202-205
special values 11, 12
timeline chart, generating 49, 50
tree map, constructing 75-78
waterfall plot, constructing 55, 56
radial plot
constructing 151-153
radius argument 104
rbind function 16
Rblogger
URL 9
RColorBrewer package
about 183
URL 183
read.csv() function 154
readLines() function 186, 211
read.table() function 17
regional maps 80-82
regression line
plotting 166, 167
removeWords argument 187
R forge
URL 9
ribbon3D() function 128
R journal
URL 9
R markdown
URL 205
rnorm() function 141
round() function 103

S

sapply() function 21, 22
scale argument 183
ScapeToad
about 98
URL 98
scatter plot
defining 26-28
points, connecting 32-34
with labels 29-32
with lines 29-32
with texts 29-32
sector.colors argument 148
sentiment analysis
URL 190
shapefiles
defining 94-97
URL 97
Shapiro test
URL 173
shiny app, R
creating 212-214

shiny package
about 212
URL 213
shinyUI() function 213
shiny website
URL 214
show.grid.labels argument 148
sizevar argument 90
slide layouts
URL 205
slidify package 203
slope chart
generating 112-114
smooth.scatter() function
URL 32
special values, R 11, 12
spiechart
URL 149
Stack Overflow
URL 9, 205
staircase.plot() function 56
start argument 152
stereo map
creating 73, 74
stock 42
summary() function 30
sunflower plot
constructing 138, 139
surf3D() function 134
surface plot
and 3D contour plot, integrating 130-133

T

tapply() function 21, 22
text
3D scatter plot, generating with 121-124
integrating, with maps 92, 93
text() function 31, 60, 93
text mining
URL 190
theta argument 123
three-dimensional heat map
creating 73, 74
Tile Mill
URL 95, 97
timeline chart
generating, in R 49, 50
title() function 112
tm_map() function 186
tree map
about 58
constructing, in R 75-78
URL 78
Twitter website
URL 209

V

violin plot
generating 170, 171
vioplot() function 171

W

watercolor plots
URL 19
waterfall plot
constructing, in R 55, 56
windowsFonts() function 197
word cloud
constructing, from document 185-187
generating 182-184
Worldmapper webpage
URL 100

X

XKCD style plot
generating 196, 197
XML
about 205-209
used, for constructing bar plot 209-211
XML and JSON formats
URL 211
xmlValue() function 211
xpathSApply() function 211
XTS
URL 161