R Data Analysis Cookbook

Data analytics with R has emerged as a very important focus for organizations of all kinds. R enables even those with only an intuitive grasp of the underlying concepts, without a deep mathematical background, to unleash powerful and detailed examinations of their data.

This book empowers you by showing you ways to use R to generate professional analysis reports. It provides examples for various important analyses and machine-learning tasks that you can try out with associated and ready-to-use data. The book also teaches you how to quickly adapt the example code for your own needs and save yourself the time needed to construct code from scratch.

What this book will do for you...

- Get data into your R environment and prepare it for analysis
- Perform exploratory data analyses and generate meaningful visualizations of the data
- Apply several machine-learning techniques for classification and regression
- Get your hands around large data sets with the help of reduction techniques
- Extract patterns from time-series data and produce forecasts based on them
- Learn how to extract actionable information from social network data
- Implement geospatial analysis
- Present your analysis convincingly through reports and build an infrastructure to enable others to play with your data

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Quick answers to common problems

Over 80 recipes to help you breeze through your data analysis projects using R
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 'Acquire and Prepare the Ingredients – Your Data'
- A synopsis of the book’s content
- More information on R Data Analysis Cookbook

About the Authors

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

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

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

Apart from his deep interest in technical fields such as data analytics, artificial intelligence, computer science, and software engineering, Viswa harbors a deep interest in education, with special emphasis on the roots of learning and methods to foster deeper learning. He has done research in this area and hopes to pursue the subject further.
Viswa would like to express deep gratitude to professors Amitava Bagchi and Anup Sen, who were inspirational forces during his early research career. He is also grateful to several extremely intelligent colleagues, notable among them being Rajesh Venkatesh, Dan Richner, and Sriram Bala, who significantly shaped his thinking. His aunt, Analdavalli; his sister, Sankari; and his wife, Shanthi, taught him much about hard work, and even the little he has absorbed has helped him immensely. His sons, Nitin and Siddarth, have helped with numerous insightful comments on various topics.

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

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

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

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

What This Book Covers

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

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


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

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

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

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

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

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

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

In this chapter, we will cover:

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

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

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

Reading data from CSV files

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

Getting ready

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

How to do it...

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

1. Read the data from auto-mpg.csv, which includes a header row:
   ```
   > auto <- read.csv("auto-mpg.csv", header=TRUE, sep = ",")
   ```

2. Verify the results:
   ```
   > names(auto)
   ```

How it works...

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

```
> names(auto)

[1] "No"        "mpg"        "cylinders"
```
The header and sep parameters allow us to specify whether the .csv file has headers and the character used in the file to separate fields. The header=TRUE and sep="," parameters are the defaults for the read.csv() function—we can omit these in the code example.

There's more...

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

Handling different column delimiters

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

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

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

Handling column headers/variable names

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

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

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

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

```r
> auto  <- read.csv("auto-mpg-noheader.csv")
> head(auto,2)
X1 X28 X4 X140 X90 X2264 X15.5 X71 chevrolet.vega.2300
1  2 19  3  70 97 2330 13.5 72     mazda rx2 coupe
2  3 36  4 107 75 2205 14.5 82        honda accord
```
We can use the optional `col.names` argument to specify the column names. If `col.names` is given explicitly, the names in the header row are ignored even if `header=TRUE` is specified:

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

> head(auto,2)

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

### Handling missing values

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

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

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

### Reading strings as characters and not as factors

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

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

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

### Reading data directly from a website

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

```r
> dat <- read.csv("http://www.exploreddata.net/ftp/WHO.csv")
```
Reading XML data

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

Getting ready

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

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

How to do it...

XML data can be read by following these steps:

1. Load the library and initialize:
   ```r
   > library(XML)
   > url <- "http://www.w3schools.com/xml/cd_catalog.xml"
   ```

2. Parse the XML file and get the root node:
   ```r
   > xmldoc <- xmlParse(url)
   > rootNode <- xmlRoot(xmldoc)
   > rootNode[1]
   ```

3. Extract XML data:
   ```r
   > data <- xmlSApply(rootNode,function(x) xmlSApply(x, xmlValue))
   ```

4. Convert the extracted data into a data frame:
   ```r
   > cd.catalog <- data.frame(t(data),row.names=NULL)
   ```

5. Verify the results:
   ```r
   > cd.catalog[1:2,]
   ```

How it works...

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

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

```r
> rootNode[1]
$CD
```
Acquire and Prepare the Ingredients – Your Data

```xml
<CD>
  <TITLE>Empire Burlesque</TITLE>
  <ARTIST>Bob Dylan</ARTIST>
  <COUNTRY>USA</COUNTRY>
  <COMPANY>Columbia</COMPANY>
  <PRICE>10.90</PRICE>
  <YEAR>1985</YEAR>
</CD>
```

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

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

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

There's more...

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

**Extracting HTML table data from a web page**

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

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

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

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

**Extracting a single HTML table from a web page**

A single table can be extracted using the following command:

```r
> table <- readHTMLTable(url, which=5)
```
Specify which to get data from a specific table. R returns a data frame.

**Reading JSON data**

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

**Getting ready**

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

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

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

**How to do it...**

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

1. Load the library:
   ```r
   > library(jsonlite)
   ```

2. Load the JSON data from files:
   ```r
   > dat.1 <- fromJSON("students.json")
   > dat.2 <- fromJSON("student-courses.json")
   ```

3. Load the JSON document from the Web:
   ```r
   > jsonDoc <- fromJSON(url)
   ```

4. Extract data into data frames:
   ```r
   > dat <- jsonDoc$list$resources$resource$fields
   > dat.1 <- jsonDoc$list$resources$resource$fields
   > dat.2 <- jsonDoc$list$resources$resource$fields
   ```

5. Verify the results:
   ```r
   > dat[1:2,]
   > dat.1[1:3,]
   > dat.2[,c(1,2,4:5)]
   ```
How it works...

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

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

Depending on the structure of the JSON document, loading the data can vary in complexity. If given a URL, the `fromJSON` function returns a list object. In the preceding list, in step 4, we see how to extract the enclosed data frame.

Reading data from fixed-width formatted files

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

Getting ready

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

How to do it...

Read the fixed-width formatted file as follows:

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

How it works...

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

We can use the optional `col.names` argument to supply our own variable names.
There's more...

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

**Files with headers**

Files with headers use the following command:

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

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

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

**Excluding columns from data**

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

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

**Reading data from R files and R libraries**

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

**Getting ready**

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

```r
> customer <- c("John", "Peter", "Jane")
> orderdate <- as.Date(c('2014-10-1','2014-1-2','2014-7-6'))
> orderamount <- c(280, 100.50, 40.25)
> order <- data.frame(customer,orderdate,orderamount)
> names <- c("John", "Joan")
> save(order, names, file="test.Rdata")
> saveRDS(order,file="order.rds")
> remove(order)
```
After saving the preceding code, the `remove()` function deletes the object from the current session.

**How to do it...**

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

1. Load data from R data files into memory:
   ```r
   > load("test.Rdata")
   > ord <- readRDS("order.rds")
   ```

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

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

**How it works...**

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

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

**There's more...**

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

**To save all objects in a session**

The following command can be used to save all objects:

```r
> save.image(file = "all.RData")
```
To selectively save objects in a session
To save objects selectively use the following commands:

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

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

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

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

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

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

  order
```

Listing all datasets in loaded packages
All the loaded packages can be listed using the following command:

```r
> data()
```

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

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

```r
> dat <- read.csv("missing-data.csv", na.strings="")
```
Acquire and Prepare the Ingredients – Your Data

**How to do it...**

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

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

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

**How it works...**

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

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

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

**There’s more...**

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

**Eliminating cases with NA for selected variables**

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

```r
> dat.income.cleaned <- dat[!is.na(dat$Income),]
> nrow(dat.income.cleaned)
[1] 25
```
Finding cases that have no missing values

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

```r
> complete.cases(dat)
[1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE
[10]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE
[19]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

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

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

Converting specific values to NA

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

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

Excluding NA values from computations

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

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

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

Replacing missing values with the mean

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

Getting ready

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

How to do it...

Read data and replace missing values:

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

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

How it works...

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

There's more...

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

Imputing random values sampled from nonmissing values

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

```r
rand.impute <- function(a) {
  missing <- is.na(a)
  n.missing <- sum(missing)
  a.obs <- a[!missing]
  imputed <- a
  imputed[missing] <- sample(a.obs, n.missing, replace=TRUE)
  return (imputed)
}

random.impute.data.frame <- function(dat, cols) {
  nms <- names(dat)
  for(col in cols) {
    name <- paste(nms[col],".imputed", sep = "")
    dat[name] <- rand.impute(dat[,col])
  }
```
With these two functions in place, you can use the following to impute random values for both Income and Phone_type:

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

## Removing duplicate cases

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

### Getting ready

Create a sample data frame:

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

### How to do it...

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

Get unique values:

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

### How it works...

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

There's more...

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

**Identifying duplicates (without deleting them)**

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

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

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

To list the duplicate cases, use the following code:

```
> prospect[duplicated(prospect),]

   salary family.size car
5 30000           3 Compact
7 30000           3 Compact
```

**Rescaling a variable to [0,1]**

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

**Getting ready**

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

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

**How to do it...**

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

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

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

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

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

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

There's more...

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

Rescaling many variables at once

Use the following function:

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

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

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

See also...

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

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

Getting ready

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

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

How to do it...

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

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

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

How it works...

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

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

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

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

**Standardizing several variables simultaneously**

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

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

With this function, you can now do things like:

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

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

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

See also...

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

---

**Downloading the example code and data**

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

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

Getting ready

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

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

How to do it...

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

1. Create a vector of break points:
   ```r
   > b <- c(-Inf, 10000, 31000, Inf)
   ```

2. Create a vector of names for break points:
   ```r
   > names <- c("Low", "Medium", "High")
   ```

3. Cut the vector using the break points:
   ```r
   > students$Income.cat <- cut(students$Income, breaks = b, labels = names)
   ```

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

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

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

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

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

There's more...

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

Creating a specified number of intervals automatically

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

```r
> students$Income.cat2 <- cut(students$Income, breaks = 4, labels = c("Level1", "Level2", "Level3","Level4"))
```
Creating dummies for categorical variables

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

Getting ready

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

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

How to do it...

Create dummies for all factors in the data frame:

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

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

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

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

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

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

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

There's more...

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

Choosing which variables to create dummies for

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

```r
> students.new1 <- dummy.data.frame(students,
   names = c("State","Gender") , sep = ".")
```
Where to buy this book

You can buy R Data Analysis Cookbook from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.