Chapter 1

Introduction

R is a programming language and an environment for statistical computing. It is similar to the S language developed at AT&T Bell Laboratories by Rick Becker, John Chambers and Allan Wilks. There are versions of R for the Unix, Windows and Mac families of operating systems. Moreover, R runs on different computer architectures like Intel, PowerPC, Alpha systems and Sparc systems. R was initially developed by Ihaka and Gentleman (1996), both from the University of Auckland, New Zealand. The current development of R is carried out by a core team of a dozen people from different institutions around the world. R development takes advantage of a growing community that cooperates in its development due to its open source philosophy. In effect, the source code of every R component is freely available for inspection and/or adaptation. This fact allows you to check and test the reliability of anything you use in R. There are many critics to the open source model. Most of them mention the lack of support as one of the main drawbacks of open source software. It is certainly not the case with R! There are many excellent documents, books and sites that provide free information on R. Moreover, the excellent R-help mailing list is a source of invaluable advice and information, much better than any amount of money could ever buy! There are also searchable mailing lists archives that you can (and should!) use before posting a question. More information on these mailing lists can be obtained at the R Web site in the section "Mailing Lists".

Data mining has to do with the discovery of useful, valid, unexpected, and understandable knowledge from data. These general objectives are obviously shared by other disciplines like statistics, machine learning, or pattern recognition. One of the most important distinguishing issues in data mining is size. With the widespread use of computer technology and information systems, the amount of data available for exploration has increased exponentially. This poses difficult challenges to the standard data analysis disciplines: One has to consider issues like computational efficiency, limited memory resources, interfaces to databases, etc. All these issues turn data mining into a highly interdisciplinary subject involving tasks not only of typical data analysts but also of people working with databases, data visualization on high dimensions, etc.

R has limitations with handling enormous datasets because all computation is carried out in the main memory of the computer. This does not mean that we will not be able to handle these problems. Taking advantage of the highly

flexible database interfaces available in R, we will be able to perform data mining on large problems. Being faithful to the open source philosophy, we will use the excellent MySQL database management system.[1] MySQL is also available for quite a large set of computer platforms and operating systems. Moreover, R has a package that enables an easy interface to MySQL (package `RMySQL` (James and DebRoy, 2009)).

In summary, we hope that at the end of reading this book you are convinced that you can do data mining on large problems without having to spend any money at all! That is only possible due to the generous and invaluable contribution of lots of people who build such wonderful tools as R and MySQL.

1.1 How to Read This Book?

The main spirit behind the book is

Learn by doing it!

The book is organized as a set of case studies. The "solutions" to these case studies are obtained using R. All necessary steps to reach the solutions are described. Using the book Web site[2] and the book-associated R package (`DMwR`), you can get all code included in the document, as well as all data of the case studies. This should facilitate trying them out by yourself. Ideally, you should read this document beside your computer and try every step as it is presented to you in the document. R code is shown in the book using the following font:

```
> R.version

               _
platform       i486-pc-linux-gnu
arch           i486
os             linux-gnu
system         i486, linux-gnu
status
major          2
minor          10.1
year           2009
month          12
day            14
svn rev        50720
language       R
version.string R version 2.10.1 (2009-12-14)
```

[1] Free download at `http://www.mysql.com`

[2] `http://www.liaad.up.pt/~ltorgo/DataMiningWithR/`.

R commands are entered at R command prompt, ">". Whenever you see this prompt you can interpret it as R waiting for you to enter a command. You type in the commands at the prompt and then press the ENTER key to ask R to execute them. This may or may not produce some form of output (the result of the command) and then a new prompt appears. At the prompt you may use the arrow keys to browse and edit previously entered commands. This is handy when you want to type commands similar to what you have done before as you avoid typing them again.

Still, you can take advantage of the code provided at the book Web site to cut and paste between your browser or editor and the R console, thus avoiding having to type all commands described in the book. This will surely facilitate your learning experience and improve your understanding of its potential.

1.2 A Short Introduction to R

The goal of this section is to provide a brief introduction to the key issues of the R language. We do not assume any familiarity with computer programming. Readers should be able to easily follow the examples presented in this section. Still, if you feel some lack of motivation to continue reading this introductory material, do not worry. You may proceed to the case studies and then return to this introduction as you get more motivated by the concrete applications.

R is a functional language for statistical computation and graphics. It can be seen as a dialect of the S language (developed at AT&T) for which John Chambers was awarded the 1998 Association for Computing Machinery (ACM) Software award that mentioned that this language "forever altered how people analyze, visualize and manipulate data".

R can be quite useful just by using it in an interactive fashion at its command line. Still, more advanced uses of the system will lead the user to develop his own functions to systematize repetitive tasks, or even to add or change some functionalities of the existing add-on packages, taking advantage of being open source.

1.2.1 Starting with R

In order to install R in your system, the easiest way is to obtain a binary distribution from the R Web site[3] where you can follow the link that takes you to the CRAN (Comprehensive R Archive Network) site to obtain, among other things, the binary distribution for your particular operating system/architecture. If you prefer to build R directly from the sources, you can get instructions on how to do it from CRAN.

[3] `http://www.R-project.org`.

After downloading the binary distribution for your operating system you just need to follow the instructions that come with it. In the case of the Windows version, you simply execute the downloaded file (`R-2.10.1-win32.exe`)[4] and select the options you want in the following menus. In some operating systems you may need to contact your system administrator to fulfill the installation task due to lack of permissions to install software.

To run R in Windows you simply double-click the appropriate icon on your desktop, while in Unix versions you should type R at the operating system prompt. Both will bring up the R console with its prompt ">".

If you want to quit R you can issue the command `q()` at the prompt. You will be asked if you want to save the current workspace. You should answer yes only if you want to resume your current analysis at the point you are leaving it, later on.

Although the set of tools that comes with R is by itself quite powerful, it is natural that you will end up wanting to install some of the large (and growing) set of add-on packages available for R at CRAN. In the Windows version this is easily done through the "`Packages`" menu. After connecting your computer to the Internet you should select the "`Install package from CRAN...`" option from this menu. This option will present a list of the packages available at CRAN. You select the one(s) you want, and R will download the package(s) and self-install it(them) on your system. In Unix versions, things may be slightly different depending on the graphical capabilities of your R installation. Still, even without selection from menus, the operation is simple.[5] Suppose you want to download the package that provides functions to connect to MySQL databases. This package name is `RMySQL`.[6] You just need to type the following command at R prompt:

```
> install.packages('RMySQL')
```

The `install.packages()` function has many parameters, among which there is the `repos` argument that allows you to indicate the nearest CRAN mirror.[7] Still, the first time you run the function in an R session, it will prompt you for the repository you wish to use.

One thing that you surely should do is install the package associated with this book, which will give you access to several functions used throughout the book as well as datasets. To install it you proceed as with any other package:

```
> install.packages('DMwR')
```

---

[4]The actual name of the file changes with newer versions. This is the name for version 2.10.1.

[5]Please note that the following code also works in Windows versions, although you may find the use of the menu more practical.

[6]You can get an idea of the functionalities of each of the R packages in the R FAQ (frequently asked questions) at CRAN.

[7]The list of available mirrors can be found at `http://cran.r-project.org/mirrors.html`.

If you want to know the packages currently installed in your computer, you can issue

```
> installed.packages()
```

This produces a long output with each line containing a package, its version information, the packages it depends, and so on. A more user-friendly, although less complete, list of the installed packages can be obtained by issuing

```
> library()
```

The following command can be very useful as it allows you to check whether there are newer versions of your installed packages at CRAN:

```
> old.packages()
```

Moreover, you can use the following command to update all your installed packages:

```
> update.packages()
```

R has an integrated help system that you can use to know more about the system and its functionalities. Moreover, you can find extra documentation at the R site. R comes with a set of HTML files that can be read using a Web browser. On Windows versions of R, these pages are accessible through the HELP menu. Alternatively, you can issue `help.start()` at the prompt to launch a browser showing the HTML help pages. Another form of getting help is to use the `help()` function. For instance, if you want some help on the `plot()` function, you can enter the command "`help(plot)`" (or alternatively, `?plot`). A quite powerful alternative, provided you are connected to the Internet, is to use the `RSiteSearch()` function that searches for key words or phrases in the mailing list archives, R manuals, and help pages; for example,

```
> RSiteSearch('neural networks')
```

Finally, there are several places on the Web that provide help on several facets of R, such as the site `http://www.rseek.org/`.

1.2.2 R Objects

There are two main concepts behind the R language: objects and functions. An object can be seen as a storage space with an associated name. Everything in R is stored in an object. All variables, data, functions, etc. are stored in the memory of the computer in the form of named objects.

Functions are a special type of R objects designed to carry out some operation. They usually take some arguments and produce a result by means of executing some set of operations (themselves usually other function calls). R

already comes with an overwhelming set of functions available for us to use, but as we will see later, the user can also create new functions.

Content may be stored in objects using the assignment operator. This operator is denoted by an angle bracket followed by a minus sign (<-):[8]

```
> x <- 945
```

The effect of the previous instruction is thus to store the number 945 on an object named **x**.

By simply entering the name of an object at the R prompt one can see its contents:[9]

```
> x

[1] 945
```

The rather cryptic "[1]" in front of the number 945 can be read as "this line is showing values starting from the first element of the object." This is particularly useful for objects containing several values, like vectors, as we will see later.

Below you will find other examples of assignment statements. These examples should make it clear that this is a destructive operation as any object can only have a single content at any time t. This means that by assigning some new content to an existing object, you in effect lose its previous content:

```
> y <- 39
> y

[1] 39

> y <- 43
> y

[1] 43
```

You can also assign numerical expressions to an object. In this case the object will store the result of the expression:

```
> z <- 5
> w <- z^2
> w

[1] 25

> i <- (z * 2 + 45)/2
> i
```

[8] You may actually also use the = sign but this is not recommended as it may be confused with testing for equality.

[9] Or an error message if we type the name incorrectly, a rather frequent error!

```
[1] 27.5
```

This means that we can think of the assignment operation as "calculate whatever is given on the right side of the operator, and assign (store) the result of this calculation to the object whose name is given on the left side".

If you only want to know the result of some arithmetic operation, you do not need to assign the result of an expression to an object. In effect, you can use R prompt as a kind of calculator:

```
> (34 + 90)/12.5
```

```
[1] 9.92
```

Every object you create will stay in the computer memory until you delete it. You may list the objects currently in the memory by issuing the `ls()` or `objects()` command at the prompt. If you do not need an object, you may free some memory space by removing it:

```
> ls()
```

```
[1] "i" "w" "x" "y" "z"
```

```
> rm(y)
> rm(z, w, i)
```

Object names may consist of any upper- and lower-case letters, the digits 0 to 9 (except in the beginning of the name), and also the period, ".", which behaves like a letter. Note that names in R are *case sensitive*, meaning that `Color` and `color` are two distinct objects. This is in effect a frequent cause of frustration for beginners who keep getting "object not found" errors. If you face this type of error, start by checking the correctness of the name of the object causing the error.

1.2.3 Vectors

The most basic data object in R is a vector. Even when you assign a single number to an object (like in `x <- 45.3`), you are creating a vector containing a single element. All objects have a *mode* and a *length*. The mode determines the kind of data stored in the object. Vectors are used to store a set of elements of the same atomic data type. The main atomic types are *character*,[10] *logical*, *numeric*, or *complex*. Thus you may have vectors of characters, logical values (`T` or `F` or `FALSE` or `TRUE`),[11] numbers, and complex numbers. The length of an object is the number of elements in it, and can be obtained with the function `length()`.

[10] The *character* type is in effect a set of characters, which are usually known as strings in some programming languages, and not a single character as you might expect.

[11] Recall that R is case sensitive; thus, for instance, `True` is not a valid logical value.

Most of the time you will be using vectors with length larger than 1. You can create a vector in R, using the `c()` function, which combines its arguments to form a vector:

```
> v <- c(4, 7, 23.5, 76.2, 80)
> v

[1]  4.0  7.0 23.5 76.2 80.0

> length(v)

[1] 5

> mode(v)

[1] "numeric"
```

All elements of a vector must belong to the same mode. If that is not true, R will force it by type coercion. The following is an example of this:

```
> v <- c(4, 7, 23.5, 76.2, 80, "rrt")
> v

[1] "4"    "7"    "23.5" "76.2" "80"   "rrt"
```

All elements of the vector have been converted to character mode. Character values are strings of characters surrounded by either single or double quotes.

All vectors may contain a special value called `NA`. This represents a missing value:

```
> u <- c(4, 6, NA, 2)
> u

[1]  4  6 NA  2

> k <- c(T, F, NA, TRUE)
> k

[1]  TRUE FALSE    NA  TRUE
```

You can access a particular element of a vector through an index between square brackets:

```
> v[2]

[1] "7"
```

The example above gives you the second element of the vector **v**. You will learn in Section 1.2.7 that we may use vectors of indexes to obtain more powerful indexing schemes.

You can also change the value of one particular vector element by using the same indexing strategies:

```
> v[1] <- "hello"
> v

[1] "hello" "7"     "23.5"  "76.2"  "80"    "rrt"
```

R allows you to create empty vectors like this:

```
> x <- vector()
```

The length of a vector can be changed by simply adding more elements to it using a previously nonexistent index. For instance, after creating the empty vector `x`, you could type

```
> x[3] <- 45
> x

[1] NA NA 45
```

Notice how the first two elements have an unknown value, `NA`. This sort of flexibility comes with a cost. Contrary to other programming languages, in R you will not get an error if you use a position of a vector that does not exists:

```
> length(x)

[1] 3

> x[10]

[1] NA

> x[5] <- 4
> x

[1] NA NA 45 NA  4
```

To shrink the size of a vector, you can take advantage of the fact that the assignment operation is destructive, as we have mentioned before. For instance,

```
> v <- c(45, 243, 78, 343, 445, 44, 56, 77)
> v

[1]  45 243  78 343 445  44  56  77

> v <- c(v[5], v[7])
> v

[1] 445  56
```

Through the use of more powerful indexing schemes to be explored in Section 1.2.7, you will be able delete particular elements of a vector in an easier way.

1.2.4 Vectorization

One of the most powerful aspects of the R language is the vectorization of several of its available functions. These functions operate directly on each element of a vector. For instance,

```
> v <- c(4, 7, 23.5, 76.2, 80)
> x <- sqrt(v)
> x

[1] 2.000000 2.645751 4.847680 8.729261 8.944272
```

The function `sqrt()` calculates the square root of its argument. In this case we have used a vector of numbers as its argument. Vectorization leads the function to produce a vector of the same length, with each element resulting from applying the function to the respective element of the original vector.

You can also use this feature of R to carry out vector arithmetic:

```
> v1 <- c(4, 6, 87)
> v2 <- c(34, 32.4, 12)
> v1 + v2

[1] 38.0 38.4 99.0
```

What if the vectors do not have the same length? R will use a *recycling rule* by repeating the shorter vector until it fills in the size of the larger vector. For example,

```
> v1 <- c(4, 6, 8, 24)
> v2 <- c(10, 2)
> v1 + v2

[1] 14  8 18 26
```

It is just as if the vector `c(10,2)` was `c(10,2,10,2)`. If the lengths are not multiples, then a warning is issued:

```
> v1 <- c(4, 6, 8, 24)
> v2 <- c(10, 2, 4)
> v1 + v2

[1] 14  8 12 34
Warning message:
In v1 + v2 :
  longer object length is not a multiple of shorter object length
```

Yet, the recycling rule has been used, and the operation was carried out (it is a warning, not an error!).

As mentioned, single numbers are represented in R as vectors of length 1. This is very handy for operations like the one shown below:

```
> v1 <- c(4, 6, 8, 24)
> 2 * v1

[1]  8 12 16 48
```

Notice how the number 2 (actually the vector `c(2)`!) was recycled, resulting in multiplying all elements of `v1` by 2. As we will see, this recycling rule is also applied with other objects, such as arrays and matrices.

1.2.5 Factors

Factors provide an easy and compact form of handling categorical (nominal) data. Factors have *levels* that are the possible values they can take. Factors are particularly useful in datasets where you have nominal variables with a fixed number of possible values. Several graphical and summarization functions that we will explore in the following chapters take advantage of this type of information. Factors allow you to use and show the values of your nominal variables as they are, which is clearly more interpretable for the user, while internally R stores these values as numeric codes that are considerably more memory efficient.

Let us see how to create factors in R. Suppose you have a vector with the sex of ten individuals:

```
> g <- c("f", "m", "m", "m", "f", "m", "f", "m", "f", "f")
> g

 [1] "f" "m" "m" "m" "f" "m" "f" "m" "f" "f"
```

You can transform this vector into a factor by entering

```
> g <- factor(g)
> g

 [1] f m m m f m f m f f
Levels: f m
```

Notice that you do not have a character vector anymore. Actually, as mentioned above, factors are represented internally as numeric vectors.[12] In this example, we have two levels, 'f' and 'm', which are represented internally as 1 and 2, respectively. Still, you do not need to bother about this as you can use the "original" character values, and R will also use them when showing you the factors. So the coding translation, motivated by efficiency reasons, is transparent to you.

Suppose you have five extra individuals whose sex information you want to store in another factor object. Suppose that they are all males. If you still want the factor object to have the same two levels as object `g`, you must use the following:

[12] You can confirm it by typing `mode(g)`.

```
> other.g <- factor(c("m", "m", "m", "m", "m"), levels = c("f",
+     "m"))
> other.g

[1] m m m m m
Levels: f m
```

Without the `levels` argument; the factor `other.g` would have a single level ('m').

As a side note, this is one of the first examples of one of the most common things in a functional programming language like R, which is function composition. In effect, we are applying one function (`factor()`) to the result of another function (`c()`). Obviously, we could have first assigned the result of the `c()` function to an object and then call the function `factor()` with this object. However, this is much more verbose and actually wastes some memory by creating an extra object, and thus one tends to use function composition quite frequently, although we incur the danger of our code becoming more difficult to read for people not so familiarized with this important notion of function composition.

One of the many things you can do with factors is to count the occurrence of each possible value. Try this:

```
> table(g)

g
f m
5 5

> table(other.g)

other.g
f m
0 5
```

The `table()` function can also be used to obtain cross-tabulation of several factors. Suppose that we have in another vector the age category of the ten individuals stored in vector `g`. You could cross-tabulate these two vectors as follows:

```
> a <- factor(c('adult','adult','juvenile','juvenile','adult','adult',
+               'adult','juvenile','adult','juvenile'))
> table(a,g)

          g
a          f m
  adult    4 2
  juvenile 1 3
```

A short side note: You may have noticed that sometimes we have a line starting with a "+" sign. This occurs when a line is getting too big and you decide to change to a new line (by hitting the ENTER key) before the command you are entering finishes. As the command is incomplete, R starts the new line with the continuation prompt, the "+" sign. You should remember that these signs are not to be entered by you! They are automatically printed by R (as is the normal prompt ">").

Sometimes we wish to calculate the marginal and relative frequencies for this type of contingency tables. The following gives you the totals for both the sex and the age factors of this dataset:

```
> t <- table(a, g)
> margin.table(t, 1)
a
   adult juvenile
       6        4

> margin.table(t, 2)

g
f m
5 5
```

The "1" and "2" in the functions represent the first and second dimensions of the table, that is, the rows and columns of `t`.

For relative frequencies with respect to each margin and overall, we do

```
> prop.table(t, 1)

          g
a                   f         m
  adult    0.6666667 0.3333333
  juvenile 0.2500000 0.7500000

> prop.table(t, 2)

          g
a              f   m
  adult    0.8 0.4
  juvenile 0.2 0.6

> prop.table(t)

          g
a              f   m
  adult    0.4 0.2
  juvenile 0.1 0.3
```

Notice that if we wanted percentages instead, we could simply multiply these function calls by 100.

1.2.6 Generating Sequences

R has several facilities to generate different types of sequences. For instance, if you want to create a vector containing the integers between 1 and 1,000, you can simply type

```
> x <- 1:1000
```

which creates a vector called `x` containing 1,000 elements—the integers from 1 to 1,000.

You should be careful with the precedence of the operator ":". The following examples illustrate this danger:

```
> 10:15 - 1
[1]  9 10 11 12 13 14

> 10:(15 - 1)

[1] 10 11 12 13 14
```

Please make sure you understand what happened in the first command (remember the recycling rule!).

You may also generate decreasing sequences such as the following:

```
> 5:0

[1] 5 4 3 2 1 0
```

To generate sequences of real numbers, you can use the function `seq()`. The instruction

```
> seq(-4, 1, 0.5)

 [1] -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0
```

generates a sequence of real numbers between -4 and 1 in increments of 0.5. Here are a few other examples of the use of the function `seq()`:[13]

```
> seq(from = 1, to = 5, length = 4)

[1] 1.000000 2.333333 3.666667 5.000000

> seq(from = 1, to = 5, length = 2)

[1] 1 5

> seq(length = 10, from = -2, by = 0.2)
```

[13] You may want to have a look at the help page of the function (typing, for instance, '`?seq`'), to better understand its arguments and variants.

```
[1] -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.2
```

You may have noticed that in the above examples the arguments used in the function calls were specified in a different way—by first indicating the name of the parameter and then the value we want to use for that specific parameter. This is very handy when we have functions with lots of parameters, most with default values. These defaults allow us to avoid having to specify them in our calls if the values suit our needs. However, if some of these defaults do not apply to our problem, we need to provide alternative values. Without the type of specification by name shown in the above examples, we would need to use the specification by position. If the parameter whose default we want to change is one of the last parameters of the function, the call by position would require the specification of all previous parameters values, even though we want to use their default values.[14] With the specification by name we avoid this trouble as this allows us to change the order of the parameters in our function calls, as they are being specified by their names.

Another very useful function to generate sequences with a certain pattern is the function `rep()`:

```
> rep(5, 10)

 [1] 5 5 5 5 5 5 5 5 5 5

> rep("hi", 3)

[1] "hi" "hi" "hi"

> rep(1:2, 3)

[1] 1 2 1 2 1 2

> rep(1:2, each = 3)

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

The function `gl()` can be used to generate sequences involving factors. The syntax of this function is `gl(k,n)`, where `k` is the number of levels of the factor, and `n` is the number of repetitions of each level. Here are two examples,

```
> gl(3, 5)

 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3

> gl(2, 5, labels = c("female", "male"))
```

[14] Actually, we can simply use commas with empty values until we reach the wanted position, as in `seq(1,4,40)`.

```
 [1] female female female female female male   male   male   male   male
Levels: female male
```

Finally, R has several functions that can be used to generate random sequences according to different probability density functions. The functions have the generic structure `r`*func*`(n, par1, par2, ...)`, where *func* is the name of the probability distribution, `n` is the number of data to generate, and `par1, par2, ...` are the values of some parameters of the density function that may be required. For instance, if you want ten randomly generated numbers from a normal distribution with zero mean and unit standard deviation, type

```
> rnorm(10)

 [1] -0.74350857  1.14875838  0.26971256  1.06230562 -0.46296225
 [6] -0.89086612 -0.12533888 -0.08887182  1.27165411  0.86652581
```

while if you prefer a mean of 10 and a standard deviation of 3, you should use

```
> rnorm(4, mean = 10, sd = 3)

[1]  5.319385 15.133113  8.449766 10.817147
```

To get five numbers drawn randomly from a Student t distribution with 10 degrees of freedom, type

```
> rt(5, df = 10)

[1] -1.2697062  0.5467355  0.7979222  0.4949397  0.2497204
```

R has many more probability functions, as well as other functions for obtaining the probability densities, the cumulative probability densities, and the quantiles of these distributions.

1.2.7 Sub-Setting

We have already seen examples of how to get one element of a vector by indicating its position inside square brackets. R also allows you to use vectors within the brackets. There are several types of index vectors. Logical index vectors extract the elements corresponding to true values. Let us see a concrete example:

```
> x <- c(0, -3, 4, -1, 45, 90, -5)
> x > 0

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

The second instruction of the code shown above is a logical condition. As `x` is a vector, the comparison is carried out for all elements of the vector (remember the famous recycling rule!), thus producing a vector with as many logical values as there are elements in `x`. If we use this vector of logical values to index `x`, we get as a result the positions of `x` that correspond to the true values:

```
> x[x > 0]

[1]  4 45 90
```

This reads as follows: Give me the positions of `x` for which the following logical expression is true. Notice that this is another example of the notion of function composition, which we will use rather frequently. Taking advantage of the logical operators available in R, you can use more complex logical index vectors, as for instance,

```
> x[x <= -2 | x > 5]

[1] -3 45 90 -5

> x[x > 40 & x < 100]

[1] 45 90
```

As you may have guessed, the "|" operator performs logical disjunction, while the "&" operator is used for logical conjunction.[15] This means that the first instruction shows us the elements of `x` that are either less than or equal to -2, or greater than 5. The second example presents the elements of `x` that are both greater than 40 and less than 100.

R also allows you to use a vector of integers to extract several elements from a vector. The numbers in the vector of indexes indicate the positions in the original vector to be extracted:

```
> x[c(4, 6)]

[1] -1 90

> x[1:3]

[1]  0 -3  4

> y <- c(1, 4)
> x[y]

[1]  0 -1
```

[15] There are also other operators, `&&` and `||`, to perform these operations. These alternatives evaluate expressions from left to right, examining only the first element of the vectors, while the single character versions work element-wise.

Alternatively, you can use a vector with negative indexes to indicate which elements are to be excluded from the selection:

```
> x[-1]
[1] -3  4 -1 45 90 -5
> x[-c(4, 6)]
[1]  0 -3  4 45 -5
> x[-(1:3)]
[1] -1 45 90 -5
```

Note the need for parentheses in the previous example due to the precedence of the ":" operator.

Indexes can also be formed by a vector of strings, taking advantage of the fact that R allows you to name the elements of a vector, through the function `names()`. Named elements are sometimes preferable because their positions are easier to memorize. For instance, imagine you have a vector of measurements of a chemical parameter obtained at five different places. You could create a named vector as follows:

```
> pH <- c(4.5, 7, 7.3, 8.2, 6.3)
> names(pH) <- c("area1", "area2", "mud", "dam", "middle")
> pH
 area1  area2    mud    dam middle
   4.5    7.0    7.3    8.2    6.3
```

In effect, if you already know the names of the positions in the vector at the time of its creation, it is easier to proceed this way:

```
> pH <- c(area1 = 4.5, area2 = 7, mud = 7.3, dam = 8.2, middle = 6.3)
```

The vector `pH` can now be indexed using the names shown above:

```
> pH["mud"]
mud
7.3
> pH[c("area1", "dam")]
area1   dam
  4.5   8.2
```

Finally, indexes may be empty, meaning that all elements are selected. An empty index represents the absence of a restriction on the selection process. For instance, if you want to fill in a vector with zeros, you could simply do "`x[] <- 0`". Please notice that this is different from doing "`x <- 0`". This latter case would assign to `x` a vector with one single element (zero), while the former (assuming that `x` exists before, of course!) will fill in all current elements of `x` with zeros. Try both!

1.2.8 Matrices and Arrays

Data elements can be stored in an object with more than one dimension. This may be useful in several situations. Arrays store data elements in several dimensions. Matrices are a special case of arrays with two single dimensions. Arrays and matrices in R are nothing more than vectors with a particular attribute that is the *dimension*. Let us see an example. Suppose you have the vector of numbers `c(45,23,66,77,33,44,56,12,78,23)`. The following would "organize" these ten numbers as a matrix:

```
> m <- c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23)
> m

 [1] 45 23 66 77 33 44 56 12 78 23
```

```
> dim(m) <- c(2, 5)
> m

     [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   23   77   44   12   23
```

Notice how the numbers were "spread" through a matrix with two rows and five columns (the dimension we have assigned to `m` using the `dim()` function). Actually, you could simply create the matrix using the simpler instruction:

```
> m <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2,
+     5)
```

You may have noticed that the vector of numbers was spread in the matrix by columns; that is, first fill in the first column, then the second, and so on. You can fill the matrix by rows using the following parameter of the function `matrix()`:

```
> m <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2,
+     5, byrow = T)
> m

     [,1] [,2] [,3] [,4] [,5]
[1,]   45   23   66   77   33
[2,]   44   56   12   78   23
```

As the visual display of matrices suggests, you can access the elements of a matrix through a similar indexing scheme as in vectors, but this time with two indexes (the dimensions of a matrix):

```
> m[2, 3]

[1] 12
```

You can take advantage of the sub-setting schemes described in Section 1.2.7 to extract elements of a matrix, as the following examples show:

```
> m[-2, 1]

[1] 45

> m[1, -c(3, 5)]

[1] 45 23 77
```

Moreover, if you omit any dimension, you obtain full columns or rows of the matrix:

```
> m[1, ]

[1] 45 23 66 77 33

> m[, 4]

[1] 77 78
```

Notice that, as a result of sub-setting, you may end up with a vector, as in the two above examples. If you still want the result to be a matrix, even though it is a matrix formed by a single line or column, you can use the following instead:

```
> m[1, , drop = F]

     [,1] [,2] [,3] [,4] [,5]
[1,]   45   23   66   77   33

> m[, 4, drop = F]

     [,1]
[1,]   77
[2,]   78
```

Functions **cbind()** and **rbind()** may be used to join together two or more vectors or matrices, by columns or by rows, respectively. The following examples should illustrate this:

```
> m1 <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2,
+     5)
> m1

     [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   23   77   44   12   23

> cbind(c(4, 76), m1[, 4])
```

```
     [,1] [,2]
[1,]    4   56
[2,]   76   12

> m2 <- matrix(rep(10, 20), 4, 5)
> m2

     [,1] [,2] [,3] [,4] [,5]
[1,]   10   10   10   10   10
[2,]   10   10   10   10   10
[3,]   10   10   10   10   10
[4,]   10   10   10   10   10

> m3 <- rbind(m1[1, ], m2[3, ])
> m3

     [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   10   10   10   10   10
```

You can also give names to the columns and rows of matrices, using the functions `colnames()` and `rownames()`, respectively. This facilitates memorizing the data positions.

```
> results <- matrix(c(10, 30, 40, 50, 43, 56, 21, 30), 2, 4,
+     byrow = T)
> colnames(results) <- c("1qrt", "2qrt", "3qrt", "4qrt")
> rownames(results) <- c("store1", "store2")
> results

       1qrt 2qrt 3qrt 4qrt
store1   10   30   40   50
store2   43   56   21   30

> results["store1", ]

1qrt 2qrt 3qrt 4qrt
  10   30   40   50

> results["store2", c("1qrt", "4qrt")]

1qrt 4qrt
  43   30
```

Arrays are extensions of matrices to more than two dimensions. This means that they have more than two indexes. Apart from this they are similar to matrices and can be used in the same way. Similar to the `matrix()` function, there is an `array()` function to facilitate the creation of arrays. The following is an example of its use:

```
> a <- array(1:24, dim = c(4, 3, 2))
> a

, , 1

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

You can use the same indexing schemes to access elements of an array. Make sure you understand the following examples.

```
> a[1, 3, 2]

[1] 21

> a[1, , 2]

[1] 13 17 21

> a[4, 3, ]

[1] 12 24

> a[c(2, 3), , -2]

     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    3    7   11
```

The recycling and arithmetic rules also apply to matrices and arrays, although they are tricky to understand at times. Below are a few examples:

```
> m <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2,
+     5)
> m

     [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   23   77   44   12   23
```

```
> m * 3

     [,1] [,2] [,3] [,4] [,5]
[1,]  135  198   99  168  234
[2,]   69  231  132   36   69

> m1 <- matrix(c(45, 23, 66, 77, 33, 44), 2, 3)
> m1

     [,1] [,2] [,3]
[1,]   45   66   33
[2,]   23   77   44

> m2 <- matrix(c(12, 65, 32, 7, 4, 78), 2, 3)
> m2

     [,1] [,2] [,3]
[1,]   12   32    4
[2,]   65    7   78

> m1 + m2

     [,1] [,2] [,3]
[1,]   57   98   37
[2,]   88   84  122
```

R also includes operators and functions for standard matrix algebra that have different rules. You may obtain more information on this by looking at Section 5 of the document "An Introduction to R" that comes with R.

1.2.9 Lists

R lists consist of an ordered collection of other objects known as their *components*. Unlike the elements of vectors, list components do not need to be of the same type, mode, or length. The components of a list are always numbered and may also have a name attached to them. Let us start by seeing a simple example of how to create a list:

```
> my.lst <- list(stud.id=34453,
+                stud.name="John",
+                stud.marks=c(14.3,12,15,19))
```

The object `my.lst` is formed by three components. One is a number and has the name `stud.id`, the second is a character string having the name `stud.name`, and the third is a vector of numbers with name `stud.marks`.

To show the contents of a list you simply type its name as any other object:

```
> my.lst
```

```
$stud.id
[1] 34453

$stud.name
[1] "John"

$stud.marks
[1] 14.3 12.0 15.0 19.0
```

You can extract individual elements of lists using the following indexing schema:

```
> my.lst[[1]]

[1] 34453

> my.lst[[3]]

[1] 14.3 12.0 15.0 19.0
```

You may have noticed that we have used double square brackets. If we had used `my.lst[1]` instead, we would obtain a different result:

```
> my.lst[1]

$stud.id
[1] 34453
```

This latter notation extracts a sub-list formed by the first component of `my.lst`. On the contrary, `my.lst[[1]]` extracts the value of the first component (in this case, a number), which is not a list anymore, as you can confirm by the following:

```
> mode(my.lst[1])
[1] "list"

> mode(my.lst[[1]])

[1] "numeric"
```

In the case of lists with named components (as the previous example), we can use an alternative way of extracting the value of a component of a list:

```
> my.lst$stud.id

[1] 34453
```

The names of the components of a list are, in effect, an attribute of the list, and can be manipulated as we did with the names of elements of vectors:

```
> names(my.lst)

[1] "stud.id"    "stud.name"  "stud.marks"

> names(my.lst) <- c("id", "name", "marks")
> my.lst

$id
[1] 34453

$name
[1] "John"

$marks
[1] 14.3 12.0 15.0 19.0
```

Lists can be extended by adding further components to them:

```
> my.lst$parents.names <- c("Ana", "Mike")
> my.lst

$id
[1] 34453

$name
[1] "John"

$marks
[1] 14.3 12.0 15.0 19.0

$parents.names
[1] "Ana"  "Mike"
```

You can check the number of components of a list using the function `length()`:

```
> length(my.lst)

[1] 4
```

You can remove components of a list as follows:

```
> my.lst <- my.lst[-5]
```

You can concatenate lists using the `c()` function:

```
> other <- list(age = 19, sex = "male")
> lst <- c(my.lst, other)
> lst
```

```
$id
[1] 34453

$name
[1] "John"

$marks
[1] 14.3 12.0 15.0 19.0

$parents.names
[1] "Ana"  "Mike"

$age
[1] 19

$sex
[1] "male"
```

Finally, you can unflatten all data in a list using the function `unlist()`. This will create a vector with as many elements as there are data objects in a list. This will coerce different data types to a common data type,[16] which means that most of the time you will end up with everything being character strings. Moreover, each element of this vector will have a name generated from the name of the list component that originated it:

```
> unlist(my.lst)
```

```
          id           name         marks1         marks2         marks3
    "34453"         "John"         "14.3"           "12"           "15"
     marks4 parents.names1 parents.names2
       "19"          "Ana"         "Mike"
```

1.2.10 Data Frames

Data frames are the data structure most indicated for storing data tables in R. They are similar to matrices in structure as they are also bi-dimensional. However, contrary to matrices, data frames may include data of a different type in each column. In this sense they are more similar to lists, and in effect, for R, data frames are a special class of lists.

We can think of each row of a data frame as an observation (or case), being described by a set of variables (the named columns of the data frame).

You can create a data frame as follows:

```
> my.dataset <- data.frame(site=c('A','B','A','A','B'),
+ season=c('Winter','Summer','Summer','Spring','Fall'),
+ pH = c(7.4,6.3,8.6,7.2,8.9))
> my.dataset
```

[16] Because vector elements must have the same type (*c.f.* Section 1.2.3).

```
  site season  pH
1    A Winter 7.4
2    B Summer 6.3
3    A Summer 8.6
4    A Spring 7.2
5    B   Fall 8.9
```

Elements of data frames can be accessed like a matrix:

```
> my.dataset[3, 2]

[1] Summer
Levels: Fall Spring Summer Winter
```

Note that the "season" column has been coerced into a factor because all its elements are character strings. Similarly, the "site" column is also a factor. This is the default behavior of the `data.frame()` function.[17]

You can use the indexing schemes described in Section 1.2.7 with data frames. Moreover, you can use the column names for accessing full columns of a data frame:

```
> my.dataset$pH

[1] 7.4 6.3 8.6 7.2 8.9
```

You can perform some simple querying of the data in the data frame, taking advantage of the sub-setting possibilities of R, as shown on these examples:

```
> my.dataset[my.dataset$pH > 7, ]

  site season  pH
1    A Winter 7.4
3    A Summer 8.6
4    A Spring 7.2
5    B   Fall 8.9

> my.dataset[my.dataset$site == "A", "pH"]

[1] 7.4 8.6 7.2

> my.dataset[my.dataset$season == "Summer", c("site", "pH")]

  site  pH
2    B 6.3
3    A 8.6
```

[17] Check the help information on the `data.frame()` function to see examples of how you can use the `I()` function, or the `stringsAsFactors` parameter to avoid this coercion.

You can simplify the typing of these queries using the function `attach()`, which allows you to access the columns of a data frame directly without having to use the name of the respective data frame. Let us see some examples of this:

```
> attach(my.dataset)
> my.dataset[site=='B', ]

  site season  pH
2    B Summer 6.3
5    B   Fall 8.9

> season

[1] Winter Summer Summer Spring Fall
Levels: Fall Spring Summer Winter
```

The inverse of the function `attach()` is the function `detach()` that disables these facilities:

```
> detach(my.dataset)
> season

Error: Object "season" not found
```

Whenever you are simply querying the data frame, you may find it simpler to use the function `subset()`:

```
> subset(my.dataset, pH > 8)

  site season  pH
3    A Summer 8.6
5    B   Fall 8.9

> subset(my.dataset, season == "Summer", season:pH)

  season  pH
2 Summer 6.3
3 Summer 8.6
```

Notice however that, contrary to the other examples seen above, you may not use this sub-setting strategy to change values in the data. So, for instance, if you want to sum 1 to the pH values of all summer rows, you can only do it this way:

```
> my.dataset[my.dataset$season == 'Summer','pH'] <-
+       my.dataset[my.dataset$season == 'Summer','pH'] + 1
```

You can add new columns to a data frame in the same way you did with lists:

```
> my.dataset$NO3 <- c(234.5, 256.6, 654.1, 356.7, 776.4)
> my.dataset

  site season  pH   NO3
1    A Winter 7.4 234.5
2    B Summer 7.3 256.6
3    A Summer 9.6 654.1
4    A Spring 7.2 356.7
5    B   Fall 8.9 776.4
```

The only restriction to this addition is that new columns must have the same number of rows as the existing data frame; otherwise R will complain. You can check the number of rows or columns of a data frame with these two functions:

```
> nrow(my.dataset)

[1] 5

> ncol(my.dataset)

[1] 4
```

Usually you will be reading your datasets into a data frame, either from some file or from a database. You will seldom type the data using the `data.frame()` function as above, particularly in a typical data mining scenario. In the next chapters describing our data mining case studies, you will see how to import this type of data into data frames. In any case, you may want to browse the "R Data Import/Export" manual that comes with R to check all the different possibilities that R has.

R has a simple spreadsheet-like interface that can be used to enter small data frames. You can edit an existent data frame by typing

```
> my.dataset <- edit(my.dataset)
```

or you may create a new data frame with,

```
> new.data <- edit(data.frame())
```

You can use the names vector to change the name of the columns of a data frame:

```
> names(my.dataset)

[1] "site"   "season" "pH"     "NO3"

> names(my.dataset) <- c("area", "season", "pH", "NO3")
> my.dataset
```

```
  area season  pH   NO3
1    A Winter 7.4 234.5
2    B Summer 7.3 256.6
3    A Summer 9.6 654.1
4    A Spring 7.2 356.7
5    B   Fall 8.9 776.4
```

As the names attribute is a vector, if you just want to change the name of one particular column, you can type

```
> names(my.dataset)[4] <- "PO4"
> my.dataset
```

```
  area season  pH   PO4
1    A Winter 7.4 234.5
2    B Summer 7.3 256.6
3    A Summer 9.6 654.1
4    A Spring 7.2 356.7
5    B   Fall 8.9 776.4
```

Finally, R comes with some "built-in" data sets that you can use to explore some of its potentialities. Most of the add-on packages also come with datasets. To obtain information on the available datasets, type

```
> data()
```

To use any of the available datasets, you can proceed as follows:

```
> data(USArrests)
```

This instruction "creates" a data frame called `USArrests`, containing the data of this problem that comes with R.

1.2.11 Creating New Functions

R allows the user to create new functions. This is a useful feature, particularly when you want to automate certain tasks that you have to repeat over and over. Instead of writing the instructions that perform this task every time you want to execute it, you encapsulate them in a new function and then simply use it whenever necessary.

R functions are objects as the structures that you have seen in the previous sections. As an object, a function can store a value. The "value" stored in a function is the set of instructions that R will execute when you call this function. Thus, to create a new function, one uses the assignment operator to store the contents of the function in an object name (the name of the function).

Let us start with a simple example. Suppose you often want to calculate the standard error of a mean associated to a set of values. By definition, the standard error of a sample mean is given by

$$\text{standard error} = \sqrt{\frac{s^2}{n}}$$

where s^2 is the sample variance and n the sample size.

Given a vector of values, we want a function to calculate the respective standard error. Let us call this function `se`. Before proceeding to create the function we should check whether there is already a function with this name in R. If that is the case, then it would be better to use another name, not to "hide" the other R function from the user.[18] To check the existence of that function, it is sufficient to type its name at the prompt:

```
> se

Error: Object "se" not found
```

The error printed by R indicates that we are safe to use that name. If a function (or any other object) existed with the name "se", R would have printed its content instead of the error.

The following is a possible way to create our function:

```
> se <- function(x) {
+   v <- var(x)
+   n <- length(x)
+   return(sqrt(v/n))
+ }
```

Thus, to create a function object, you assign to its name something with the general form

```
function(<set of parameters>) { <set of R instructions> }
```

After creating this function, you can use it as follows:

```
> se(c(45,2,3,5,76,2,4))
[1] 11.10310
```

If we need to execute several instructions to implement a function, like we did for the function `se()`, we need to have a form of telling R when the function body starts and when it ends. R uses the curly braces as the syntax elements that start and finish a group of instructions.

The value returned by any function can be "decided" using the function `return()` or, alternatively, R returns the result of the last expression that was evaluated within the function. The following function illustrates this and also the use of parameters with default values,

[18] You do not have to worry about overriding the definition of the R function. It will continue to exist, although your new function with the same name will be on top of the search path of R, thus "hiding" the other standard function.

```
> basic.stats <- function(x,more=F) {
+   stats <- list()
+
+   clean.x <- x[!is.na(x)]
+
+   stats$n <- length(x)
+   stats$nNAs <- stats$n-length(clean.x)
+
+   stats$mean <- mean(clean.x)
+   stats$std <- sd(clean.x)
+   stats$med <- median(clean.x)
+   if (more) {
+     stats$skew <- sum(((clean.x-stats$mean)/stats$std)^3) /
+                   length(clean.x)
+     stats$kurt <- sum(((clean.x-stats$mean)/stats$std)^4) /
+                   length(clean.x) - 3
+   }
+   unlist(stats)
+ }
```

This function has a parameter (`more`) that has a default value (`F`). This means that you can call the function with or without setting this parameter. If you call it without a value for the second parameter, the default value will be used. Below are examples of these two alternatives:

```
> basic.stats(c(45, 2, 4, 46, 43, 65, NA, 6, -213, -3, -45))

       n     nNAs     mean      std      med
11.00000  1.00000 -5.00000 79.87768  5.00000

> basic.stats(c(45, 2, 4, 46, 43, 65, NA, 6, -213, -3, -45),
+     more = T)

         n      nNAs      mean       std       med      skew      kurt
11.000000  1.000000 -5.000000 79.877684  5.000000 -1.638217  1.708149
```

The function `basic.stats()` also introduces a new instruction of R: the instruction `if()`. As the name indicates this instruction allows us to condition the execution of certain instructions to the truth value of a logical test. In the case of this function, the two instructions that calculate the kurtosis and skewness of the vector of values are only executed if the variable `more` is true; otherwise they are skipped.

Another important instruction is the `for()`. This instruction allows us to repeat a set of commands several times. Below is an example of the use of this instruction:

```
> f <- function(x) {
+   for(i in 1:10) {
+     res <- x*i
```

```
+     cat(x,'*',i,'=',res,'\n')
+   }
+ }
```

Try to call `f()` with some number (e.g. `f(5)`). The instruction `for` in this function says to R that the instructions "inside of it" (delimited by the curly braces) are to be executed several times. Namely, they should be executed with the variable "`i`" taking different values at each repetition. In this example, "`i`" should take the values in the set `1:10`, that is, 1, 2, 3, ..., 10. This means that the two instructions inside the `for` are executed ten times, each time with `i` set to a different value. The set of values specified in front of the word **`in`** can be any vector, and the values need not be a sequence or numeric.

The function `cat()` can be used to output the contents of several objects to the screen. Namely, character strings are written as themselves (try `cat('hello!')`), while other objects are written as their content (try `y <- 45` and then `cat(y)`). The string "\n" makes R change to the next line.

1.2.12 Objects, Classes, and Methods

One of the design goals of R is to facilitate the manipulation of data so that we can easily perform the data analysis tasks we have. In R, data is stored on objects. As mentioned, everything in R is an object, from simple numbers to functions or more elaborate data structures. Every R object belongs to a *class*. Classes define the abstract characteristics of the objects that belong to them. Namely, they specify the attributes or properties of these objects and also their behaviors (or methods). For instance, the `matrix` class has specific properties like the dimension of the matrices and it also has specific behavior for some types of operations. In effect, when we ask R the content of a matrix, R will show it with a specific format on the screen. This happens because there is a specific `print` method associated with all objects of the class `matrix`. In summary, the class of an object determines (1) the methods that are used by some general functions when applied to these objects, and also (2) the representation of the objects of that class. This representation consists of the information that is stored by the objects of this class.

R has many predefined classes of objects, together with associated methods. On top of this we can also extend this list by creating new classes of objects or new methods. These new methods can be both for these new classes or for existing classes. New classes are normally created after existing classes, usually by adding some new pieces of information to their representation.

The representation of a class consists of a set of *slots*. Each slot has a name and an associated class that determines the information that it stores. The operator "`@`" can be used to access the information stored in a slot of an object. This means that `x@y` is the value of the slot `y` of the object `x`. This obviously assumes that the class of objects to which `x` belongs has a slot of information named `y`.

Another important notion related to classes is the notion of inheritance

between classes. This notion establishes relationships between the classes that allow us to indicate that a certain new class extends an existing one by adding some extra information. This extension also implies that the new class inherits all the methods of the previous class, which facilitates the creation of new classes, as we do not start from scratch. In this context, we only need to worry about implementing the methods for the operations where the new class of objects differs from the existing one that it extends.

Finally, another very important notion is that of polymorphism. This notion establishes that some functions can be applied to different classes of objects, producing the results that are adequate for the respective class. In R, this is strongly related to the notion of generic functions. Generic functions implement a certain, very general, high-level operation. For instance, as we will see, the function `plot()` can be used to obtain a graphical representation of an object. This is its general goal. However, this graphical representation may actually be different depending on the type of object. It is different to plot a set of numbers, than to plot a linear regression model, for instance. Polymorphism is the key to implementing this without disturbing the user. The user only needs to know that there is a function that provides a graphical representation of objects. R and its inner mechanisms handle the job of *dispatching* these general tasks for the class-specific functions that provide the graphical representation for each class of objects. All this method-dispatching occurs in the background without the user needing to know the "dirty" details of it. What happens, in effect, is that as R knows that `plot()` is a generic function, it will search for a plot method that is specific for the class of objects that were included in the `plot()` function call. If such a method exists, it will use it; otherwise it will resort to some default plotting method. When the user decides to create a new class of objects he needs to decide if he wants to have specific methods for his new class of objects. So if he wants to be able to plot objects of the new class, then he needs to provide a specific plot method for this new class of objects that "tells" R how to plot these new objects.

These are the basic details on classes and methods in R. The creation of new classes and respective methods is outside the scope of this book. More details can be obtained in many existing books on programming with R, such as, the excellent book *Software for Data Analysis* by Chambers (2008).

1.2.13 Managing Your Sessions

When you are using R for more complex tasks, the command line typing style of interaction becomes a bit limited. In these situations it is more practical to write all your code in a text file and then ask R to execute it. To produce such a file, you can use your favorite text editor (like Notepad, Emacs, etc.) or, in case you are using the Windows version of R, you can use the script editor available in the File menu. After creating and saving the file, you can issue the following command at R prompt to execute all commands in the file:

```
> source('mycode.R')
```

This assumes that you have a text file called "mycode.R"[19] in the current working directory of R. In Windows versions the easiest way to change this directory is through the option "Change directory" of the "File" menu. In Unix versions you may use the functions `getwd()` and `setwd()` respectively, to, check and change the current working directory.

When you are using the R prompt in an interactive fashion you may wish to save some of the objects you create for later use (such as some function you have typed in). The following example saves the objects named `f` and `my.dataset` in a file named "mysession.RData":

```
> save(f,my.dataset,file='mysession.RData')
```

Later, for instance in a new R session, you can load these objects by issuing

```
> load('mysession.RData')
```

You can also save all objects currently in R workspace,[20] by issuing

```
> save.image()
```

This command will save the workspace in a file named ".RData" in the current working directory. This file is automatically loaded when you run R again from this directory. This kind of effect can also be achieved by answering Yes when quitting R (*see* Section 1.2.1).

Further readings on R

The online manual *An Introduction to R* that comes with every distribution of R is an excellent source of information on the R language. The "Contributed" subsection of the "Documentation" section at the R Web site, includes several free books on different facets of R.

1.3 A Short Introduction to MySQL

This section provides a very brief introduction to MySQL. MySQL is not necessary to carry out all the case studies in this book. Still, for larger data mining projects, the use of a database management system like MySQL can be crucial.

MySQL can be downloaded at no cost from the Web site `http://www.mysql.com`. As R, MySQL is available for different operating systems, such as Linux and Windows. If you wish to install MySQL on your computer, you should download it from the MySQL Web site and follow its installation instructions. Alternatively, you can also access any MySQL server that is installed on another computer to which you have network access.

[19] The extension ".R" is not mandatory.

[20] These can be listed issuing `ls()`, as mentioned before.

You can use a client program to access MySQL on your local computer or over the network. There are many different MySQL client programs at the MySQL Web site. MySQL comes with a console-type client program, which works in a command-by-command fashion, like the R console. Alternatively, you have graphical client programs that you can install to use MySQL. In particular, the MySQL Query Browser is a freely available and quite a nice example of such programs that you may consider installing on your computer.

To access a MySQL server installed on your computer using the console-type client, you can issue the following command at your operating system prompt:

```
$> mysql -u myuser -p

Password: ********

mysql>
```

or, in case of a remote server, something like

```
$> mysql -h myserver.xpto.pt -u myuser -p

Password: ********

mysql>
```

We are assuming that the server has a user named "myuser" and that the server is password protected. If all this sounds strange to you, you should either talk with your system administrator about MySQL, or learn a bit more about this software using the user manual that comes with every installation, or by reading a book (e.g., DuBois, 2000).

After entering MySQL, you can either use existent database or create a new one. The latter can be done as follows in the MySQL console-type client:

```
mysql> create database contacts;

Query OK, 1 row affected (0.09 sec)
```

To use this newly created database or any other existing database, you issue

```
mysql> use contacts;

Database changed
```

A database is formed by a set of tables containing the data concerning some entities. You can create a table as follows:

```
mysql> create table people(
    -> id INT primary key,
    -> name CHAR(30),
    -> address CHAR(60));
```

```
Query OK, 1 row affected (0.09 sec)
```

Note the continuation prompt of MySQL ("->").

To populate a table with data, you can either insert each record by hand or use one of the MySQL import statements to read in data contained, for instance, in a text file.

A record can be inserted in a table as follows:

```
mysql> insert into people
    -> values(1,'John Smith','Strange Street, 34, Unknown City');

Query OK, 1 row affected (0.35 sec)
```

You can list the records in a given table using the SELECT statement, of which we provide a few examples below.

```
mysql> select * from people;

+----+------------+-----------------------------------+
| id | name       | address                           |
+----+------------+-----------------------------------+
|  1 | John Smith | Strange Street, 34, Unknown City |
+----+------------+-----------------------------------+
1 row in set (0.04 sec)


mysql> select name, address from people;

+------------+-----------------------------------+
| name       | address                           |
+------------+-----------------------------------+
| John Smith | Strange Street, 34, Unknown City |
+------------+-----------------------------------+
1 row in set (0.00 sec)


mysql> select name from people where id >= 1 and id < 10;

+------------+
| name       |
+------------+
| John Smith |
+------------+
1 row in set (0.00 sec)
```

After you finish working with MySQL, you can leave the console-type client by issuing the "`quit`" statement.

Further readings on MySQL

Further information on MySQL can be obtained from the free user's manual that comes with MySQL. This manual illustrates all aspects of MySQL, from installation to the technical specifications of the SQL language used in MySQL. The book *MySQL* by DuBois (2000), one of the active developers of MySQL, is also a good general reference on this DBMS.