Data Mining with R:
learning by case studies

Luis Torgo

LIACC-FEP, University of Porto
R. Campo Alegre, 823 - 4150 Porto, Portugal
email: ltorgo@liacc.up.pt
http://www.liacc.up.pt/∼ltorgo

August 10, 2005

Preface

The main goal of this book is to introduce the reader to the use of R as a tool for performing data mining. R is a freely downloadable[1] language and environment for statistical computing and graphics. Its capabilities and the large set of available packages make this tool an excellent alternative to the existing (and expensive!) data mining tools.

One of the key issues in data mining is size. A typical data mining problem involves a large database from where one seeks to extract useful knowledge. In this book we will use MySQL as the core database management system. MySQL is also freely available[2] for several computer platforms. This means that you will be able to perform "serious" data mining without having to pay any money at all. Moreover, we hope to show you that this comes with no compromise in the quality of the obtained solutions. Expensive tools do not necessarily mean better tools! R together with MySQL form a pair very hard to beat as long as you are willing to spend some time learning how to use them. We think that it is worthwhile, and we hope that you are convinced as well at the end of reading this book.

The goal of this book is not to describe all facets of data mining processes. Many books exist that cover this area. Instead we propose to introduce the reader to the power of R and data mining by means of several case studies. Obviously, these case studies do not represent all possible data mining problems that one can face in the real world. Moreover, the solutions we describe can not be taken as complete solutions. Our goal is more to introduce the reader to the world of data mining using R through pratical examples. As such our analysis of the case studies has the goal of showing examples of knowledge extraction using R, instead of presenting complete reports of data mining case studies. They should be taken as examples of possible paths in any data mining project and can be used as the basis for developping solutions for the reader's own projects. Still, we have tried to cover a diverse set of problems posing different challenges in terms of size, type of data, goals of analysis and tools that are necessary to carry out this analysis.

We do not assume any prior knowledge about R. Readers that are new to R and data mining should be able to follow the case studies. We have tried to make the different case studies self-contained in such a way that the reader can start anywhere in the document. Still, some basic R functionalities are introduced in the first, simpler, case studies, and are not repeated, which means that if you are new to R, then you should at least start with the first case studies

[1]Download it from http://www.R-project.org.
[2]Download it from http://www.mysql.com.

to get acquainted with R. Moreover, the first chapter provides a very short introduction to R and MySQL basics, which should facilitate the understanding of the following chapters. We also do not assume any familiarity with data mining or statistical techniques. Brief introductions to different data mining techniques are provided as they are necessary in the case studies. It is not an objective of this book to provide the reader with full information on the technical and theoretical details of these techniques. Our descriptions of these tools are given to provide basic understanding on their merits, drawbacks and analysis objectives. Other existing books should be considered if further theoretical insights are required. At the end of some sections we provide "Further readings" pointers for the readers interested in following up the topics. In summary, our target readers are more users of data analysis tools than researchers or developers. Still, we hope the latter also find reading this book useful as a form of entering the "world" of R and data mining.

The book is accompanied by a set of freely available R source files that can be obtained at the book Web site[3]. These files include all the code used in the case studies. They facilitate the "do it yourself" philosophy followed in this document. We strongly recommend that readers install R and try the code as they read the book. All data used in the case studies is available at the book Web site as well.

[3] http://www.liacc.up.pt/~ltorgo/DataMiningWithR/.

Contents

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. 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 then any amount of money could ever buy! There are also searchable mailing lists archives[1] that you can (and should!) use before posting a question.

Architectures and operating systems on which R runs

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..

What is Data Mining?

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

[1] http://maths.newcastle.edu.au/~rking/R/.

on large problems. Being faithful to the Open Source philosophy we will use the excellent MySQL database management system[2]. MySQL is also available for a quite large set of computer platforms and operating systems. Moreover, R has a package that enables an easy interface to MySQL (package "RMySQL").

The MySQL DBMS

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 scientists that 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[3] you may 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,

Check the book Web site!

```
> R.version

platform i386-pc-mingw32
arch     i386
os       mingw32
system   i386, mingw32
status
major    2
minor    0.1
year     2004
month    11
day      15
language R
```

R commands are entered at R command prompt, ">". Whenever you see this prompt you may interpret it as R being 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 usually produces 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 may take advantage of the code provided at the book Web site to cut and paste between your browser and the R console, thus avoiding having to type all commands described in the book. This will surely facility your learning, and improve your understanding of its potential.

[2]Free download at http://www.mysql.com.
[3]http://www.liacc.up.pt/~ltorgo/DataMiningWithR/.

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 on this section. Still, if you feel some lack of motivation for continuing 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 which 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. 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

Downloading R

In order to install R in your system the easiest way is to obtain a binary distribution from the R Web site[4] where you may 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 may get instructions on how to do it from CRAN.

Installing R

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 (`rw2001.exe`)[5] 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.

Starting R

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 "`>`".

Quitting R

If you want to quit R you may 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.

Installing add-on packages

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

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

[5] The actual name of the file changes with newer versions. This is the name for version 2.0.1.

be slightly different depending on the gaphical capabilities of your R installation. Still, even without selection from menus, the operation is simple[6]. Suppose you want to download the package that provides functions to connect to MySQL databases. This package name is RMySQL[7]. You just need to type the following two commands 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[8]. The second instruction performs the actual downloading and installation of the package[9].

If you want to know the packages currently installed in your distribution you may issue,

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

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

```
> library()
```

Another useful command is the following, which allows you to check whether there are newer versions of your installed packages at CRAN,

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

Moreover, you may use the following command to update all your installed packages[10],

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

Getting help in R

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 may issue help.start() at the prompt to launch 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 in alternative ?plot). Finally, a quite powerfull alternative, provided you're connected to the Internet, is to use the RSiteSearch() function that searches for key words or phrases in the mailing list archives, and R manuals and help pages.

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

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

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

[9]Please notice that to carry out these tasks on some operating systems you will most surely need to have administration permissions, so the best is to ask you system administrator to do the installation. Still, it is also possible to download and install the packages on your personal home directory (consult the R help facilites to check how).

[10]You need Administrator permissions in some operating system versions to do this.

1.2.2 R objects

R is an object-oriented language. All variables, data, functions, etc. are stored in the memory of the computer in the form of named objects.

The assignment operator

Content may be assigned to objects using the assignment operator. This consists of an angle bracket followed by a minus sign (<-),

```
> x <- 945
```

By simply entering the name of an object at the R prompt one can see its contents,

```
> 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 may find other examples of assignment statements[11],

```
> y <- 39
> y
[1] 39
> y <- 43
> y
[1] 43
```

You may 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
[1] 27.5
```

You do not need to assign the result of an expression to an object. In effect, you may use R prompt as a kind of calculator,

```
> (34 + 90)/12.5
[1] 9.92
```

Listing and deleting objects

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()` commands at the prompt. If you do not need an object you may free some memory space by removing it,

[11]Notice how the assignment is a destructive operation (values previously stored in an object are discarded by new assignments).

```
> ls()
[1] "i" "w" "y" "z"
> rm(y)
> rm(z,w,i)
```

Valid object names

Object names may consist of any upper and lower-case letters, the digits 0-9 (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.

1.2.3 Vectors

Types of 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. For vectors, it can take the values *character*, *logical*, *numeric* or *complex*. Thus you may have vectors of characters, logical values (`T` or `F` or `FALSE` or `TRUE`)[12], numbers, and complex numbers. The length of an object is the number of elements in it, and can be obtained with the function `length()`.

Creating vectors

Most of the times you will be using vectors with length larger than 1. You may create a vector in R , using the `c()` function,

```
> 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"
```

Type coercion

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.

Missing values

All vectors may contain a special value named `NA`. This represents a missing value,

```
> v <- c(NA,"rrr")
> v
[1] NA   "rrr"
> u <- c(4,6,NA,2)
> u
[1]  4  6 NA  2
> k <- c(T,F,NA,TRUE)
```

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

```
> k
[1]  TRUE FALSE    NA  TRUE
```

You may access a particular element of a vector through an index between squared brackets,

Accessing individual elements

```
> v[2]
[1] "rrr"
```

You will learn in Section 1.2.7 that we may use vectors of indeces to obtain more powerful indexing schemes.

You may also change the value of one particular vector element,

Changing an element

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

R allows you to create empty vectors like this,

Empty vectors

```
> v <- vector()
```

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

Adding more elements

```
> v[3] <- 45
> v
[1] NA NA 45
```

Notice how the first two elements got an unknown value, `NA`.

To shrink the size of a vector you can use the assignment operation. For instance,

Removing elements

```
> v <- c(45,243,78,343,445,645,2,44,56,77)
> v
 [1]  45 243  78 343 445 645   2  44  56  77
> v <- c(v[5],v[7])
> v
[1] 445   2
```

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.

Vector arithmetic

You may 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
```

The recycling rule

What if the vectors do not have the same length? R will use a *recycling rule* by repeating the shorter vector till it fills in the size of the larger. 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:
longer object length
        is not a multiple of shorter object length in: v1 + v2
```

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

As mentioned before 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, like arrays and matrices.

1.2.5 Factors

Factors provide an easy and compact form of handling categorical (nominal) data. Factors have *levels* which are the possible values they may take. A factor is stored internally as a numeric vector with values 1,2,...,k, where k is the number of levels of the factor. 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 information.

Let us see how to create factors in R. Suppose you have a vector with the sex of 10 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,

Creating a factor

```
> 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[13]. In this example, we have two levels, 'f' and 'm', which are represented internally as 1 and 2, respectively.

Suppose you have 5 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 may issue the following,

```
> 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').

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

Frequency tables for factors

```
> 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 10 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
```

[13]You may confirm it by typing mode(g).

Notice how we have entered a long command in several lines. If you hit the “return” key before ending some command, R presents a continuation prompt (the “+” sign) for you to complete the instruction.

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

```
> t <- table(a,g)
> margin.table(t,1)
a
   adult juvenile
       6        4
> margin.table(t,2)
g
f m
5 5
```

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 1000, you can simply type,

Integer sequences

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

which creates a vector named `x` containing 1000 elements, the integers from 1 to 1000.

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 like the following,

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

Sequences of real numbers

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()`[14],

```
> 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=.2)
 [1] -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.2
```

Sequences with repeated elements

Another very useful function to generate sequences with a certain pattern is the function ,

```
> rep(5,10)
 [1] 5 5 5 5 5 5 5 5 5 5
> rep('hi',3)
[1] "hi" "hi" "hi"
> rep(1:3,2)
[1] 1 2 3 1 2 3
```

Factor sequences

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

Random sequences

Finally, R has several functions that can be used to generate random sequences according to a large set of probability density functions. The func-

[14]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.

tions 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 10 randomly generated numbers from a normal distribution with zero mean and unit standard deviation, type

```
> rnorm(10)
 [1] -0.306202028  0.335295844  1.199523068  2.034668704  0.273439339
 [6] -0.001529852  1.351941008  1.643033230 -0.927847816 -0.163297158
```

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

```
> rnorm(10,mean=10,sd=3)
 [1]  7.491544 12.360160 12.879259  5.307659 11.103252 18.431678  9.554603
 [8]  9.590276  7.133595  5.498858
```

To get 5 numbers drawn randomly from a Student t distribution with 10 degrees of freedom, type

```
> rt(5,df=10)
[1] -0.46608438 -0.44270650 -0.03921861  0.18618004  2.23085412
```

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 Indexing

We have already seen examples on how to get one element of a vector by indicating its position between 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.

Logical index vectors

```
> x <- c(0,-3,4,-1,45,90,-5)
> x
[1]  0 -3  4 -1 45 90 -5
> x > 0
[1] FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
> y <- x>0
```

The third 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`. We then store this logical vector in object `y`. You can now obtain the positive elements in `x` using this vector `y` as a logical index vector,

```
> x[y]
[1]  4 45 90
```

As the truth elements of vector `y` are in the 3rd, 5th and 6th positions, this corresponds to extracting these elements from `x`.

Incidentally, you could achieve the same result by just issuing,

```
> x[x>0]
[1]  4 45 90
```

Taking advantage of the logical operators available in R you may 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 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.

Integer index vectors

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

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

Negative integer index vectors

Alternatively, you may use a vector with negative indeces, 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 last example due to the precedence of the ":" operator.

Character string index vectors

Indeces may also be formed by a vector of character 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 on 5 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 
```

[15]The 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 elementwise.

The vector pH can now be indexed by using these names,

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

Empty indeces

Finally, indeces may be empty, meaning that all elements are selected. 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 which 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 10 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 2 rows and 5 columns (the dimension we have assigned to `m` using the `dim()` function). Actually, you could simply create the matrix using the simpler instruction,

Creating a matrix

```
> 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, *i.e.* first fill in the first column, then the second, and so on. You may 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
```

Accessing matrix elements

As the visual display of matrices suggests you may access the elements of a matrix through a similar indexing schema as in vectors, but this time with two indeces (the dimensions of a matrix),

```
> m[2,2]
[1] 56
```

You may take advantage of the indexing schemes described in Section 1.2.7 to extract elements of a matrix, as the following examples show,

```
> m[-2,2]
[1] 23
> 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 indexing, you may end up with a vector, as in the two above examples. If you still want the result to be a matrix, eventhough a matrix formed by a single line or column, you may 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
```

Joining vectors and matrices

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(10,50),10,5)
> 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
 [5,]   10   10   10   10   10
 [6,]   10   10   10   10   10
 [7,]   10   10   10   10   10
 [8,]   10   10   10   10   10
 [9,]   10   10   10   10   10
[10,]   10   10   10   10   10
> m3 <- rbind(m1[1,],m2[5,])
> m3
     [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   10   10   10   10   10
```

Giving names to columns and rows of matrices

You may 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
```

Creating an array

Arrays are extensions of matrices to more than two dimensions. This means that they have more than two indeces. 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(1:50,dim=c(2,5,5))
> a
, , 1

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

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   11   13   15   17   19
[2,]   12   14   16   18   20

, , 3

     [,1] [,2] [,3] [,4] [,5]
[1,]   21   23   25   27   29
[2,]   22   24   26   28   30

, , 4

     [,1] [,2] [,3] [,4] [,5]
[1,]   31   33   35   37   39
[2,]   32   34   36   38   40
```

```
, , 5

     [,1] [,2] [,3] [,4] [,5]
[1,]   41   43   45   47   49
[2,]   42   44   46   48   50
```

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

Accessing elements of arrays

```
> a[1,5,2]
[1] 19
> a[1,,4]
[1] 31 33 35 37 39
> a[1,3,]
[1]  5 15 25 35 45
> a[,c(3,4),-4]
, , 1

     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 2

     [,1] [,2]
[1,]   15   17
[2,]   16   18

, , 3

     [,1] [,2]
[1,]   25   27
[2,]   26   28

, , 4

     [,1] [,2]
[1,]   45   47
[2,]   46   48
> a[1,c(1,5),-c(4,5)]
     [,1] [,2] [,3]
[1,]    1   11   21
[2,]    9   19   29
```

The recycling and arithmetic rules also apply to matrices and arrays. See the following small examples of this,

```
> 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
```

Vectorization works in an element by element fashion as mentioned before. If some of the operands is shorter than the others, it is recycled. Still, 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*. These 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,

Creating 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`, other 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
```

Extracting elements of a list

You may 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 have 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!

In the case of lists with named components (as the previous example), we may 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
```

Extending lists

Lists may 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 may check the number of components of a list using the function `lenght()`,

```
> length(my.lst)
[1] 4
```

Concatenating lists

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 may 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. By default 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 which 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

A data frame is similar to a matrix but with named columns. However, contrary to matrices data frames may include data of different type on each column. In this sense they are more similar to lists, and in effect, for R data frames are a special class of lists.

Each row of a data frame is an observation (or case), being described by a set of variables (the named columns of the data frame).

Creating a data frame

You can create a data frame like this,

```
> 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
  site season  pH
```

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

```
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 may use the indexing schemes described in Section 1.2.7 with data frames. Moreover, you may 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
```

Querying data frames

You can perform some simple querying of the data in the data frame taking advantage of the indexing 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
```

You can simplify the typing of these queries by 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[pH > 8,]
  site season  pH
3    A Summer 8.6
5    B   Fall 8.9
> season
[1] Winter Summer Summer Spring Fall
Levels:  Fall Spring Summer Winter
```

[17]Check the help information on the `data.frame()` function to see examples on how you may use the `I()` function to avoid this coercion.

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

```
> detach(my.dataset)
> season
Error: Object "season" not found
```

Adding columns to a data frame

Because data frames are lists, you may 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 6.3 256.6
3    A Summer 8.6 654.1
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 may check the number of rows or columns of a data frame with these two functions,

Number of rows and columns

```
> nrow(my.dataset)
[1] 5
> ncol(my.dataset)
[1] 4
```

Usually you will be reading your data sets 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 different types of possibilities 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())
```

Changing the name of columns

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 7.0 234.5
2    B Summer 6.9 256.6
3    A Summer 8.6 654.1
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 7.0 234.5
2    B Summer 6.9 256.6
3    A Summer 8.6 654.1
4    A Spring 7.2 356.7
5    B   Fall 8.9 776.4
```

Built-in data sets

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 data sets. To obtain information on the available data sets type,

```
> data()
```

To use any of the available data sets you can use its name with the same function,

```
> data(USArrests)
```

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

1.2.11 Some useful functions

This section provides very brief descriptions of some useful functions. This list is far from being exhaustive! It is just a personal sample of different types of functions. You may wish to use R help facilities to get more information on these functions, because we do not describe all their arguments.

Reading and writing data

`read.table(file)`	Reads a table from a file and creates a data frame from the contents of this file, where each row corresponds to a line of the file and each column corresponds to a field in the file.
`write.table(obj,file)`	Converts `obj` into a data frame, and writes the result to `file`.

Some basic statistics

`sum(x)`	Sum of the elements of `x`.
`max(x)`	Largest value of the elements in `x`.
`min(x)`	Smallest value of the elements in `x`.
`which.max(x)`	The index of the largest value in `x`.
`which.min(x)`	The index of the smallest value in `x`.
`range(x)`	The range of values in `x` (gives the same result as `c(min(x),max(x))`).
`length(x)`	The number of elements of `x`.
`mean(x)`	The mean value of the elements of `x`.
`median(x)`	The median value of the elements of `x`.
`sd(x)`	The standard deviation of the elements of `x`.
`var(x)`	The variance of the elements of `x`.
`quantile(x)`	The quantiles of `x`.
`scale(x)`	Standardizes the elements of `x`, *i.e.* subtracts the mean and divides by the standard deviation. Results in a vector with zero mean and unit standard deviation. Also works with data frames (column-wise and only with numeric data!).

Some vector and mathematical functions

`sort(x)`	Sort the elements of `x`.
`rev(x)`	Reverse the order of the elements of `x`.
`rank(x)`	Ranks of the elements of `x`.
`log(x,base)`	The logarithms of all elements of `x` in base `base`.
`exp(x)`	The exponentials of the elements of `x`.
`sqrt(x)`	The square roots of the elements of `x`.
`abs(x)`	The absolute value of the elements of `x`.
`round(x,n)`	Rounds all elements of `x` to `n` decimal places.
`cumsum(x)`	Returns a vector where the ith element is the sum from $x[1]$ to $x[i]$.
`cumprod(x)`	The same for the product.
`match(x,s)`	Returns a vector with the same length as `x`, with the elements of `x` that are contained in `s`. The ones that do not belong to `s` have the value NA.
`union(x,y)`	Returns a vector with the union of vectors `x` and `y`.
`intersect(x,y)`	Returns a vector with the intersection of vectors `x` and `y`.
`setdiff(x,y)`	Returns a vector resulting from removing the elements of `y` from `x`.
`is.element(x,y)`	Return TRUE if `x` is contained in vector `y`.
`choose(n,k)`	Calculates the number of combinations of `k`, `n` to `n`.

Matrix algebra

`diag(x,nrow,ncol)`	Builds a diagonal matrix with `nrow` rows and `ncol` columns, with the number `x`. Can also be used to extract or replace the diagonal elements of a matrix (see the help of the function).
`t(x)`	The transpose of `x`.
`nrow(x)`	Number of rows of `x`.
`ncol(x)`	The number of columns of `x`.
`A %*% B`	Matrix algebraic multiplication of `A` by `B`.
`solve(A,b)`	Solve the system of linear equations $Ax = b$. With a single matrix argument (*e.g.* `solve(A)`) it calculates the inverse of matrix `A`.
`svd(x)`	Singular value decomposition of matrix `x`.
`qr(x)`	QR decomposition of matrix `x`.
`eigen(x)`	Eigen values and vectors of the square matrix `x`.
`det(x)`	The determinant of matrix `x`.

Meta-functions

`apply(x,margin,fun)`	Applies the function `fun` to all rows or columns of matrix `x`. If the parameter `margin` is 1 then the function is applied to each row, if it is 2 it is applied to each column.
`sapply(x,fun)`	Applies the function `fun` to all elements of vector `x`.
`lapply(x,fun)`	The same as previous but the function is applied to all elements of a list `x`.
`aggregate(x,by,fun)`	Applies a summarization function `fun` to all subsets of rows of the data frame `x`. The subsets are formed by using all combinations of the factors in the list `by`.
`by(x,by,fun)`	Applies a function `fun` to all subsets of rows of the data frame `x`. The subsets are formed by using all combinations of the factors in the list `by`.

1.2.12 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, it is better to create a new function containing these instructions, and then simply using it whenever necessary.

R functions are objects similar to the structures that you have seen in 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 we proceed to create the function we should check whether there is already a function with this name in R. If that was the case, and we insisted in creating our new function with that name, the side-effect could be to override the R function which is not probably what you want! 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
```

The body of a function can be written either in different lines (like the example above) or in the same line by separating each instruction by the ';' character.[18]

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]This separator also applies to any instructions that you issue on the R command prompt.

```
> 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
+   }
+   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,

Parameters with default values

```
> basic.stats(c(45,2,4,46,43,65,NA,6,-213,-3,-45))
$n
[1] 11

$nNAs
[1] 1

$mean
[1] -5

$std
[1] 79.87768

$med
[1] 5

> basic.stats(c(45,2,4,46,43,65,NA,6,-213,-3,-45),more=T)
$n
[1] 11

$nNAs
[1] 1

$mean
[1] -5

$std
[1] 79.87768

$med
[1] 5

$skew
[1] -1.638217

$kurt
[1] 1.708149
```

The `if` instruction

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.

The `for` instruction

Another important instruction is the `for()`. This instruction allows us to repeat a set of commands several times. Below you have 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 10 times, each time with `i` set to a different value.

Output to the screen

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

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 file you can use your favorite text editor (like Notepad, Emacs, etc.) or, in case you are using the Windows version of R you may use the script editor available in the File menu. After creating and saving the file, you may 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 named "mycode.R"[19] on 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()` to, respectively, 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 (like for instance some function

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

you have typed in). The following example saves the objects named `f` and `my.dataset` in a file named "mysession.R",

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

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

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

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 quiting R (*c.f.* Section 1.2.1).

Further readings on R

The online manual *An Introduction to R* that comes with every distribution of R is an excelent source of information on R. The "Contributed" sub-section of the "Documentation" section at 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 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 freely downloaded from the Web site http://www.mysql.com. As R, MySQL is available for different operating systems, like for instance Linux and Windows. If you wish to install MySQL on your computer you should download it from MySQL Web site and follow its installation instructions. Alternatively, you may use your local computer network to access a computer server that has MySQL installed.

Assuming you have access to a MySQL server you can use a client program to access MySQL. There are many different MySQL client programs at 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 may install to use MySQL. In particular, the MySQL Query Browser is a freely available and quite nice example of such programs that you may consider installing on your computer.

To access the server using the console-type client you can use the following statement at your operating system prompt,

```
$> mysql -u myuser -p
Password: ********
mysql>
```

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

or, in case of a remote server, something like,

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

We are assuming that you have a user named "myuser" that has access to the MySQL server, 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 for instance the user manual that comes with every installation, or by reading a book (*e.g.* *DuBois, 2000*).

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

Creating a database in MySQL

```
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 insert 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 issuing the "quit" statement.

Further readings on MySQL

Further information on MySQL can be obtained from the free user's manual coming 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.

Chapter 2

Case Study 1: Predicting Algae Blooms

This case study will introduce you to some basic tasks of data mining: data pre-processing, exploratory data analysis, and predictive model construction. For this initial case study we have selected a small problem by data mining standards. If you are not familiar with the R language and you have not read the small introduction provided in Section 1.2 of Chapter 1, you may feel the need to review that section as you work through this case study.

2.1 Problem description and objectives

High concentrations of certain harmful algae in rivers is a serious ecological problem with a strong impact not only on river lifeforms, but also on water quality. Being able to monitor and perform an early forecast of algae blooms is essential to improve the quality of rivers.

With the goal of addressing this prediction problem several water samples were collected in different European rivers at different times during a period of approximately one year. For each water sample, different chemical properties were measured as well as the frequency of occurrence of 7 harmful algae. Some other characteristics of the water collection process were also stored such as the season of the year, the river size, and the river speed.

One of the main motivations behind this application lies in the fact that chemical monitoring is cheap and easily automated, while the biological analysis of the samples to identify the algae that are present in the water, involves microscopic examination, requires trained manpower and is therefore both expensive and slow. As such, obtaining models that are able to accurately predict the algae frequencies based on chemical properties would facilitate the creation of cheap and automated systems for monitoring harmful algae blooms.

Another objective of this study is to provide a better understanding of the factors influencing the algae frequencies. Namely, we want to understand how these frequencies are related to certain chemical attributes of water samples as well as other characteristics of the samples (like season of the year, type of river, etc.).

2.2 Data Description

The data available for this problem consists of two separate text files ("Analysis.txt" and "Eval.txt"). Each file contains data concerning a set of water samples. The first file contains data regarding 200 water samples collected at different European rivers[1]. Each water sample is described by 11 variables. Three of these variables are nominal and describe the season of the year when the sample was collected, the size of the river, and the water speed of the river. The 8 remaining variables are values of different chemical parameters measured in the water sample. Namely, the measured parameters were:

- Maximum pH value
- Minimum value of O_2 (Oxygen)
- Mean value of Cl (Chloride)
- Mean value of NO_3^- (Nitrates)
- Mean value of NH_4^+ (Ammonium)
- Mean of PO_4^{3-} (Orthophosphate)
- Mean of total PO_4 (Phosphate)
- Mean of Chlorophyll

Associated with each of these water samples are 7 frequency numbers of different harmful algae found in the water samples. No information is given regarding the names of the algae that were identified.

The second data file contains other 140 water samples described by the same variables. These samples can be regarded as a kind of test set. As such, the information concerning the values of the concentrations of the 7 algae was omitted. The main goal of our study is to predict the frequencies of the 7 algae for these 140 water samples. This means that we are facing a predictive task. This is one among the diverse set of problems tackled in data mining. In these predictive tasks our main goal is to obtain a model that allows us to predict the value of a certain target variable given the values of a set of predictor variables. This model may also provide indications on which predictor variables have larger impact on the target variable, *i.e.* the model may provide a comprehensive description of the factors that influence the target variable.

2.3 Loading the data into R

The two data files are available at the book Web site on the Data section. The "Training data" link contains the 200 water samples of the "Analysis.txt" file, while the "Test data" link points to the "Eval.txt" file that contains the 140 test samples. There is an additional link that points to a file ("Sols.txt") that contains the algae frequencies of the 140 test samples. This latter file will be

[1]Actually, each observation in the data files is an aggregation of several water samples carried out over a period of three months, on the same river, and during the same season of the year.

used to check the performance of our predictive models and will be taken as unknown information at the time of model construction. The files have the values for each water sample in a different line. Each line of the training and test files contains the variables values (according to the description given on Section 2.2) separated by spaces. Unknown values are indicated with the string "XXXXXXX".

The first thing to do in order to follow this case study is to download the three files from the book Web site and store them in some directory of your hard disk (preferably on the current working directory of your running R session, which you may check issuing the command `getwd()` at the prompt).

Reading data from text files

After downloading the data files into a local directory, we can start by loading into R the data from the "Analysis.txt" file (the training data,*i.e.* the data that will be used to obtain the predictive models). To read in the data in the file it is sufficient to issue the following command:[2]

```
> algae <- read.table('Analysis.txt',
+          header=F,
+          dec='.',
+          col.names=c('season','size','speed','mxPH','mnO2','Cl',
+          'NO3','NH4','oPO4','PO4','Chla','a1','a2','a3','a4',
+          'a5','a6','a7'),
+          na.strings=c('XXXXXXX'))
```

The parameter `header=F` indicates that the file to be read does not include a first line with the variables names. `dec='.'` states that the numbers use the '.' character to separate decimal places. Both these two previous parameter settings could have been omitted as we are using their default values. `col.names` allows us to provide a vector with the names to give to the variables whose values are being read. Finally, `na.strings` serves to indicate a vector of strings that are to be interpreted as unknown values. These values are represented internally in R by the value NA, as mentioned in Section 1.2.3.

Reading from other types of files

R has several other functions that can be used to read data contained in text files. You may wish to type "`?read.table`" to obtain further information on this and other related functions. Moreover, R has a manual that you may want to browse named 'R Data Import/Export', that describes the different possibilities R includes for reading data from other applications.

The result of this instruction is a data frame, which can be seen as a kind of matrix or table with named columns (in this example with the names we have indicated in the `col.names` parameter). Each line of this data frame contains a water sample. For instance, we can see the first 5 samples using the instruction `algae[1:5,]`[3]. In Section 1.2.7 (page 12) we have described other alternative ways of extracting particular elements of R objects like data frames.

2.4 Data Visualization and Summarization

Given the lack of further information on the problem domain it is wise to investigate some of the statistical properties of the data, so as to get a better grasp

[2]We assume that the data files are in the current working directory of R. If not, use the command "setwd()" to change this, or use the "Change dir..." option in the 'File' menu of Windows versions.

[3]You can get a similar result with `head(algae)`.

of the problem. Even if that was not the case it is always a good idea to start our analysis by some kind of exploratory data analysis similar to the one we will show below.

Getting basic descriptive statistics

A first idea of the statistical properties of the data can be obtained through a summary of its descriptive statistics,

```
> summary(algae)
    season       size        speed          mxPH            mnO2
 autumn:40   large :45   high  :84   Min.   :5.600   Min.   : 1.500
 spring:53   medium:84   low   :33   1st Qu.:7.700   1st Qu.: 7.725
 summer:45   small :71   medium:83   Median :8.060   Median : 9.800
 winter:62                           Mean   :8.012   Mean   : 9.118
                                     3rd Qu.:8.400   3rd Qu.:10.800
                                     Max.   :9.700   Max.   :13.400
                                     NA's   :1.000   NA's   : 2.000
       Cl               NO3              NH4               oPO4
 Min.   :  0.222   Min.   : 0.050   Min.   :    5.00   Min.   :  1.00
 1st Qu.: 10.981   1st Qu.: 1.296   1st Qu.:   38.33   1st Qu.: 15.70
 Median : 32.730   Median : 2.675   Median :  103.17   Median : 40.15
 Mean   : 43.636   Mean   : 3.282   Mean   :  501.30   Mean   : 73.59
 3rd Qu.: 57.824   3rd Qu.: 4.446   3rd Qu.:  226.95   3rd Qu.: 99.33
 Max.   :391.500   Max.   :45.650   Max.   :24064.00   Max.   :564.60
 NA's   : 10.000   NA's   : 2.000   NA's   :    2.00   NA's   :  2.00
      PO4              Chla               a1              a2
 Min.   :  1.00   Min.   :  0.200   Min.   : 0.00   Min.   : 0.000
 1st Qu.: 41.38   1st Qu.:  2.000   1st Qu.: 1.50   1st Qu.: 0.000
 Median :103.29   Median :  5.475   Median : 6.95   Median : 3.000
 Mean   :137.88   Mean   : 13.971   Mean   :16.92   Mean   : 7.458
 3rd Qu.:213.75   3rd Qu.: 18.308   3rd Qu.:24.80   3rd Qu.:11.375
 Max.   :771.60   Max.   :110.456   Max.   :89.80   Max.   :72.600
 NA's   :  2.00   NA's   : 12.000
       a3               a4               a5               a6
 Min.   : 0.000   Min.   : 0.000   Min.   : 0.000   Min.   : 0.000
 1st Qu.: 0.000   1st Qu.: 0.000   1st Qu.: 0.000   1st Qu.: 0.000
 Median : 1.550   Median : 0.000   Median : 1.900   Median : 0.000
 Mean   : 4.309   Mean   : 1.992   Mean   : 5.064   Mean   : 5.964
 3rd Qu.: 4.925   3rd Qu.: 2.400   3rd Qu.: 7.500   3rd Qu.: 6.925
 Max.   :42.800   Max.   :44.600   Max.   :44.400   Max.   :77.600

       a7
 Min.   : 0.000
 1st Qu.: 0.000
 Median : 1.000
 Mean   : 2.495
 3rd Qu.: 2.400
 Max.   :31.600
```

This simple instruction immediately gives us a first overview of the statistical properties of the data[4]. In the case of nominal variables (that are represented by factors in R data frames), it provides frequency counts for each possible value [5]. For instance, we can observe that there are more water samples collected in winter than in other seasons. For numeric variables, R gives us a series of statistics like their mean, median, quartiles information and extreme values. These statistics provide a first idea on the distribution of the variable values (we will return to this issue later on). In the event of a variable having some unknown values, their number is also shown following the string NA's. By observing the

[4]An interesting alternative with similar objectives is the function `describe()` in package 'Hmisc'.

[5]Actually, if there are too many, only the most frequent are shown.

difference between medians and means, as well as the interquartile range (3rd quartile minus the 1st quartile)[6], we can have an idea of the skewness of the distribution and also its spread. Still, most of the times this information is better captured graphically. Let us see an example:

```
> hist(algae$mxPH, prob=T)
```

Histograms

This instruction shows us the histogram of the variable *mxPH*. The result is shown in Figure 2.1. With the parameter `prob=T` we get probabilities instead of frequency counts for each interval of values[7]. Omitting this parameter setting would give us frequency counts of each bar of the histogram.

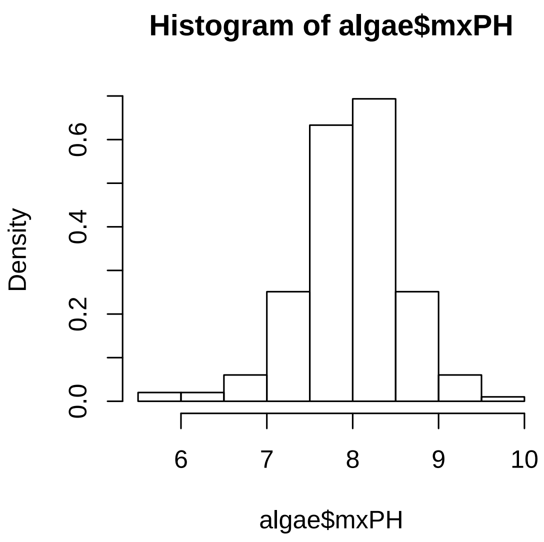

Figure 2.1: The histogram of variable *mxPH*.

Figure 2.1 tells us that the values of variable *mxPH* follow a distribution very near to the normal distribution, with the values nicely clustered around the mean value. We can get further information by using instead the following instructions (the result is shown in Figure 2.2),

```
> hist(algae$mxPH, prob=T, xlab='',
+      main='Histogram of maximum pH value',ylim=0:1)
> lines(density(algae$mxPH,na.rm=T))
> rug(jitter(algae$mxPH))
```

[6]If we order the values of a variable, the 1st quartile is the value below which there are 25% of the data points, while the 3rd quartile is the value below which there are 75% of the cases, thus meaning that between these two values we have 50% of our data. The interquartile range is defined as the 3rd quartile minus the 1st quartile, thus being a measure of the spread of the variable around its central value (larger values indicate larger spread).

[7]The areas of the rectangles should sum up to one (and not the heigth of the rectangles as some people might expect).

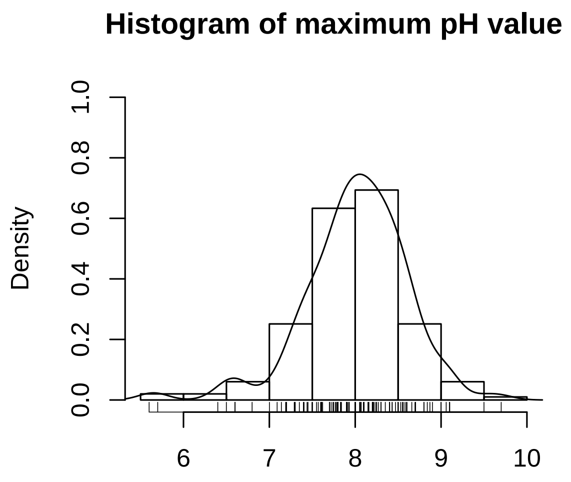

Figure 2.2: An "enriched" version of the histogram of variable *MxPH*.

The first instruction is basically the same as previously, except that we omit the X axis label, we change the title of the graph, and we provide sensible limits for the Y axis. The second instruction plots a smooth version of the histogram (a kernel density estimate[8] of the distribution of the variable), while the third plots the real values of the variable near the X axis, thus allowing easy spotting of outliers[9]. For instance, we can observe that there are two values significantly smaller than all others. This kind of data inspection is very important as it may identify possible errors in the data sample, or even help to locate values that are so awkward that they may only be errors or at least we would be better off by disregarding them in posterior analysis.

Another example (Figure 2.3) showing this kind of data inspection can be achieved with the following instructions, this time for variable *oPO4*:

```
> boxplot(algae$oPO4,boxwex=0.15,ylab='Orthophosphate (oPO4)')
> rug(jitter(algae$oPO4),side=2)
> abline(h=mean(algae$oPO4,na.rm=T),lty=2)
```

Box plots

The first instruction draws a box plot of variable *oPO4*. Box plots provide a quick summarization of some key properties of the variable distribution. Namely, there is a box whose limits are the 1st and 3rd quantiles of the variable. This box has an horizontal line inside that represents the median value of the

[8]The `na.rm=T` parameter setting is used in several functions as a way of indicating that NA values should not be considered in the function calculation. This is necessary in several functions because it is not their default behavior, and otherwise an error would be generated.

[9]Actually, this contains two function calls, the `rug()` function performs the plotting, while the `jitter()` function is used to randomly perturb slightly the original values to plot, so that there will be almost no change that two values are equal, thus avoiding ticks over each other that would "hide" some values from the visual inspection.

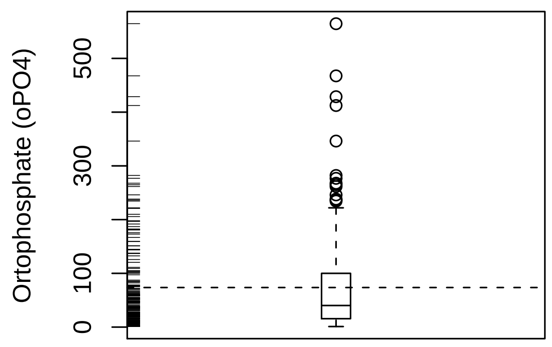

Figure 2.3: An 'enriched' box plot for *Orthophosphate*.

variable. Let r be the interquartile range. The small horizontal dash above the box is the largest observation that is less or equal to the 3rd quartile plus $1.5r$. The small horizontal dash below the box is the smallest observation that is greater than or equal to the 1st quartile minus $1.5r$. The circles below or above these small dashes represent observations that are extremely low (high) compared to all others, and are usually considered outliers. This means that box plots give us plenty of information regarding not only the central value and spread of the variable but also on eventual outliers.

The second instruction was already described before (the only difference being the place where the data is plotted), while the third uses the function `abline()` to draw an horizontal line[10] at the mean value of the variable, which is obtained using the function `mean()`. By comparing this line with the line inside the box indicating the median we can conclude that the presence of several outliers has distorted the value of the mean as a statistic of centrality (i.e. indicating the more common value of the variable).

The analysis of Figure 2.3 shows us that the variable *oPO4* has a distribution of the observed values clearly skeweed to the right (high values). In most of the water samples the value of *oPO4* is low, but there are several observations of high values, and even of extremely high values.

Identifying outlier cases

Sometimes when we encounter outliers, we are interested in identifying the observations that have these "strange" values. We will show two ways of doing this. First, let us do it graphically. If we plot the values of variable *NH4* we notice a very large value. We can identify the respective water sample by doing:

```
> plot(algae$NH4,xlab='')
> abline(h=mean(algae$NH4,na.rm=T),lty=1)
```

[10]The parameter `lty=2` is used to obtain a dashed line.

```
> abline(h=mean(algae$NH4,na.rm=T)+sd(algae$NH4,na.rm=T),lty=2)
> abline(h=median(algae$NH4,na.rm=T),lty=3)
> identify(algae$NH4)
```

The first instruction plots all values of the variable. The calls to the `abline()` function draw three informative lines, one with the mean value, another with the mean plus one standard deviation, and the other with the median. The last instruction is interactive, and allows the user to click on the plotted dots with the left mouse button. For every clicked dot, R will write the respective row number in the `algae` data frame.[11] The user can finish the interaction by clicking the right mouse button.

We can also perform this inspection without graphics, as shown below:[12]

```
> algae[algae$NH4 > 19000,]
```

This instruction illustrates another form of indexing a data frame, by using a logical expression as a row selector (see Section 1.2.7 for more examples on this). The result is showing the rows of the data frame for which the logical expression is true.

Finally, we will explore a few examples of another type of data inspection. These examples use the “lattice” graphics package of R, that provides a set of impressive graphics tools.

Suppose that we would like to study the distribution of the values of say algal *a1*. We could use any of the possibilities discussed before. However, if we wanted to study how this distribution depends on other variables new tools are required.

Conditioned box plots

Conditioned plots are graphical representations that depend on a certain factor. A factor is a nominal variable with a set of finite values. For instance, we can obtain a set of box plots for the variable *a1*, for each value of the variable *size* (*c.f.* Figure 2.4). Each of the box plots was obtained using the subset of water samples that have a certain value of the variable *size*. These graphs allow us to study how this nominal variable may influence the distribution of the values of *a1*. The code to obtain the box plots is,

```
> library(lattice)
> bwplot(size ~ a1, data=algae,ylab='River Size',xlab='Algal A1')
```

The first instruction loads in the ’lattice’ package[13]. The second obtains a box plot using the ’lattice’ version of these plots. The first argument of this

[11]The position where you click relatively to the point determines the side where R writes the row number. For instance, if you click on the right of the dot, the row number will be written on the right.

[12]The output of this instruction may seem a bit strange. This results from the fact that there are some observations with NA values in variable *NH4*, which ”puzzles” R. We may avoid this behavior by issuing instead the instruction `algae[!is.na(algae$NH4) & algae$NH4 > 19000,]`. The ’!’ operator performs the logical negation, the ’&’ operator the logical conjunction, while the function `is.na()` is true whenever its argument has the value NA.

[13]A word of warning on the use of the function `library()` to load packages. This is only possible if the package is installed on your computer. Otherwise a error will be issued by R. If that is the case you will need to install the package using any of the methods described in the Section 1.2.1 of Chapter 1.

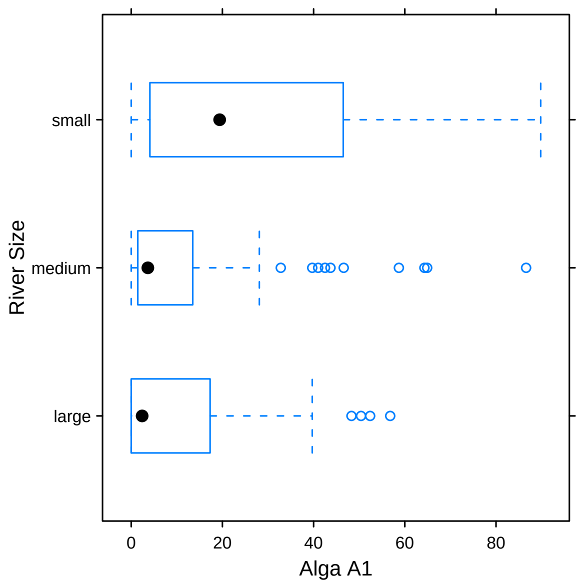

Figure 2.4: A conditioned box plot of Algal *a1*.

instruction can be read as 'plot *a1* for each value of *size'*. The remaining arguments have obvious meanings.

Figure 2.4 allows us to observe that higher frequencies of algal *a1* are expected in smaller rivers, which can be valuable knowledge.

An interesting variant of this type of plots that gives us more information on the distribution of the variable being plotted, are box-percentile plots, which are available in package 'Hmisc'. Let us see an example of its use with the same example of plotting the occurrency of algal *a1* against the size of rivers,

```
> library(Hmisc)
> bwplot(size ~ a1, data=algae,panel=panel.bpplot,
+        probs=seq(.01,.49,by=.01), datadensity=TRUE,
+        ylab='River Size',xlab='Algal A1')
```

The result of this call is shown on Figure 2.5. The dots are the mean value of the frequency of the algal for the different river sizes. Vertical lines represent the 1st quartil, median and 3rd quartil, on this order. The graphs show us the actual values of the data with small dashes, and the information of the distribution of these values is provided by the quantile plots. These graphs thus provide much more information than standard box plots as the one shown in Figure 2.4. For instance, we can confirm our previous observation that smaller rivers have higher frequencies of this alga, but we can also observe that the value of the observed frequencies on these small rivers is much more widespread across the domain of frequencies than on other types of rivers.

This type of conditioned plots is not restricted to nominal variables, neither to a single factor. You can carry out the same kind of conditioning study with continuous variables as long as you previously "discretize" them. Let us see an example by observing the behavior of the frequency of algal *a3* conditioned

Conditioning on continuous variables

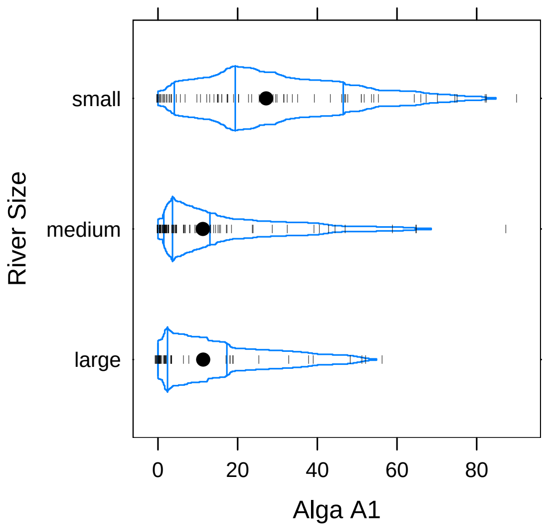

Figure 2.5: A conditioned box percentile plot of Algal *a1*.

by *season* and *mnO2*, this latter being a continuous variable. Figure 2.6 shows such a graph and the code to obtain it is the following:

```
> minO2 <- equal.count(na.omit(algae$mnO2),number=4,overlap=1/5)
> stripplot(season ~ a3|minO2,data=algae[!is.na(algae$mnO2),])
```

The first instruction, uses function `equal.count()` to create a factorized version of the continuous variable *mnO2*. The parameter `number` sets the number of desired bins, while the parameter `overlap` sets the overlap between the bins near their respective boundaries (this means that certain observations will be assigned to adjacent bins). The bins are created such that they contain an equal number of observations. You may have noticed that we did not use `algae$mnO2` directly. The reason is the presence of NA values in this variable. This would cause problems in the subsequent graphics function. We have used the function `na.omit()` that removes any NA value from a vector.[14]

Strip plots

The second line contains the call to the graphics function `stripplot()`. This is another graphical function of the 'lattice' package. It creates a graph containing the actual values of a variable, in different strips depending on another variable (in this case the *season*). Different graphs are them drawn for each bin of the variable *mnO2*. The bins are ordered from left to right and from bottom up. This means that the bottom left plot corresponds to lower values of *mnO2*[15]. The existence of NA values in *mnO2* also has some impact on the data to be used for drawing the graph. Instead of using the parameter `data=algae` (as for creating Figure 2.4), we had to 'eliminate' the rows corresponding to samples with NA values in *mnO2*. This was achieved using the function `is.na()`, which

[14]Later, in Section 2.5 we shall see a better solution to this.
[15]You may check the actual values of the created intervals by printing the created discretized version of the variable.

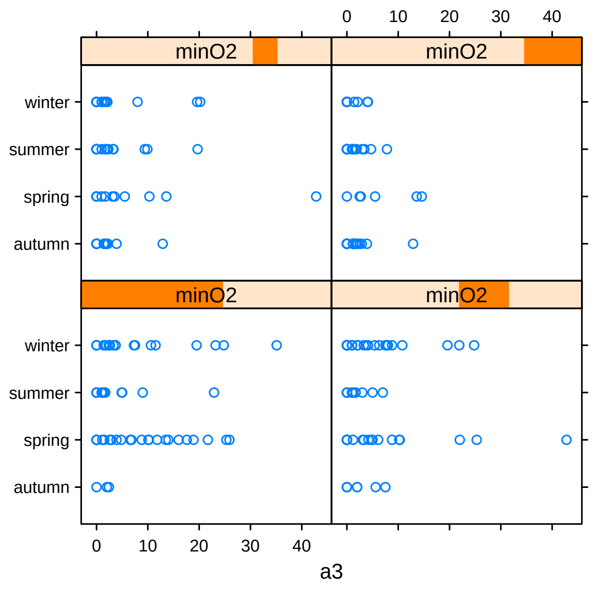

Figure 2.6: A conditioned stripplot plot of Algal *a3* using a continuous variable.

produces a vector of boolean values (TRUE or FALSE). An element of this vector is TRUE when *mnO2* is NA. This vector has as many elements as there are rows in the data frame `algae`. The construction `!is.na(mnO2)` thus returns a vector of boolean values that are TRUE in elements corresponding to rows where *mnO2* is known, because '!' is the logical negation operator. Given the way R handles boolean index vectors (*c.f.* Section 1.2.7) this means we are using the rows of the `algae` data frame, corresponding to water samples not having NA in the value of variable *mnO2*. NA values cause problems to several methods. The next session addresses this issue and suggests forms of overcoming these problems.

Further readings on data summarization and visualization

Most standard statistics books will include some sections on providing summaries of data. A simple and well-written book is *Statistics for technology* by Chatfield (1983). This book has simple examples and is quite informal. Another good source of information is the book *Introductory Statistics with R* by Dalgaard (2002). For data visualization, the book *Visualizing Data* by Cleveland (1993) is definitely a must. This is an outstanding book that is clearly worth its value. A more formal follow-up of this work is the book *The Elements of Graphing Data* (Cleveland, 1995) by the same author.

2.5 Unknown values

There are several water samples with unknown variable values. This situation, rather common in real problems, may preclude the use of certain techniques that are not able to handle missing values.

Whenever we are handling a data set with missing values we can follow several strategies. The most common are:

- Remove the cases with unknowns
- Fill in the unknown values by exploring the correlations between variables
- Fill in the unknown values by exploring the similarity between cases
- Use tools that are able to handle these values.

The last alternative is the most restrictive, as it limits the set of tools one can use. Still, it may be a good option whenever we are confident on the merit of the strategies used by the data mining tools to handle missing values.

In the following sub-sections we will show examples of how to implement these strategies in R. If you decide to try the code given in these sections you should be aware that they are not complementary. This means that as you go into another method of dealing with missing values you should read in again the original data (*c.f.* Section 2.3) to have all the unknown cases again, as each section handles them in a different way.

2.5.1 Removing the observations with unknown values

The option of removing the cases with unknown values is very easy to implement, and can also be a reasonable choice when the proportion of cases with unknowns is small with respect to the size of the available data set.

Inspecting the cases with unknowns

Before eliminating all observations with at least one unknown value in some variable, it is always wise to have a look, or at least count them,

```
> algae[!complete.cases(algae),]
...
...
> nrow(algae[!complete.cases(algae),])
[1] 16
```

The function `complete.cases()` produces a vector of boolean values with as many elements as there are rows in the `algae` data frame, where an element is TRUE if the respective row is 'clean' of NA values (*i.e.* is a complete observation). Thus the above instruction shows the water samples with some NA values because the '!' operator performs logical negation as it was mentioned before.

Eliminating all cases with unknowns

In order to remove these 16 water samples from our data frame we can simply do,

```
> algae <- na.omit(algae)
```

Even if we decide not to use this drastic method of removing all cases with some unknown value, we may remove some observations because the number of unknown values is so high that they are almost useless, and even complex methods of filling in these values will be too unreliable. Note that if you have executed the previous command you should read in the data again (*c.f.* Section 2.3), as this instruction has removed all unknowns, so the next statements would not make sense! Looking at the cases with unknowns we can see that both the samples 62 and 199 have 6 of the 11 explanatory variables with unknown values. In such cases, it is wise to simply ignore these observations by removing them,

```
> algae <- algae[-c(62,199),]
```

2.5.2 Filling in the unknowns with the most frequent values

An alternative to eliminating the cases with unknown values is to try to find the most probable value for each of these unknowns. Again several strategies can be followed, with different trade-offs between the level of approximation and the computational complexity of the method.

The simplest and fastest way of filling in the unknown values is to use some central value. Central values reflect the most frequent value of a variable distribution, thus they are a natural choice for this strategy. Several statistics of centrality exist, like the mean, the median, etc. The choice of the most adequate value depends on the distribution of the variable. For approximately normal distributions, where all observations are nicely clustered around the mean, this statistic is the best choice. However, for skewed distributions, or for variables with outliers, the mean can be disastrous. Skewed distributions have most values clustered near one of the sides of the range of values of the variable, thus the mean is clearly not representative of the most common value. On the other hand, the presence of outliers (extreme values) may distort the calculation of the mean[16], thus leading to similar representativeness problems. Thus it is not wise to use the mean without a previous inspection of the distribution of the variable using, for instance, some of the graphicall tools of R (*e.g.* Figure 2.2). For skewed distributions or for variables with outliers, the median is a better statistic of centrality.

For instance, the sample `algae[48,]` does not have a value in the variable *mxPH*. As the distribution of this variable is nearly normal (*c.f.* Figure 2.2) we could use its mean value to fill in the "hole". This could be done by,

Filling a single value

```
> algae[48,'mxPH'] <- mean(algae$mxPH,na.rm=T)
```

where the function `mean()` gives the mean value of any vector of numbers, and `na.rm=T` disregards any NA values in this vector from the calculation.[17]

Filling a full column

Most of the times we will be interested in filling in all unknowns of a column instead of working on a case-by-case basis as above. Let us see an example of this with the variable *Chla*. This variable is unknown on 12 water samples. Moreover, this is a situation were the mean is a very poor representative of the most frequent value of the variable. In effect, the distribution of *Chla* is skewed to higher values, and there are a few extreme values that make the mean value (13.971) highly unrepresentative of the most frequent value. Therefore, we will use the median to fill in all unknowns of this column,

```
> algae[is.na(algae$Chla),'Chla'] <- median(algae$Chla,na.rm=T)
```

While the presence of unknown values may impair the use of some methods, filling in their values using a strategy as above is usually considered a bad idea. This simple strategy although extremely fast, and thus appealing for large data sets, may introduce a large bias in the data, which can influence our posterior analysis. However, unbiased methods that find the optimal value to fill in an unknown, are extremely complex and may not be adequate for some large data mining problems.

[16] The mean of the vector `c(1.2,1.3,0.4,0.6,3,15)` is 3.583.

[17] Without this 'detail' the result of the call would be NA because of the presence of NA values in this column.

2.5.3 Filling in the unknown values by exploring correlations

Filling unknowns by exploring correlations

An alternative for getting less biased estimators of the unknown values is to explore the relationships between variables. For instance, using the correlation between the variable values we could discover that a certain variable is highly correlated with *mxPH*, which would enable us to obtain other more probable value for the sample number 48, which has an unknown on this variable. This could be preferable to the use the mean as we did above.

Correlations between numerical variables

To obtain the variables correlation we can issue the command,

```
> cor(algae[,4:18],use="complete.obs")
```

The function `cor()` produces a matrix with the correlation values between the variables (we have avoided the first 3 variables because they are nominal). The `use="complete.obs"` setting tells R to disregard observations with NA values in this calculation. Values near 1 (-1) indicate a strong positive (negative) linear correlation between the values of the two respective variables. Other R functions could then be used to approximate the functional form of this linear correlation, which in turn would allow us to estimate the values of one variable from the values of the correlated variable.

The result of this `cor()` function is not very legible, but we can put it through the function `symnum()` to improve this,

```
> symnum(cor(algae[,4:18],use="complete.obs"))
     mP mO Cl NO NH o P Ch a1 a2 a3 a4 a5 a6 a7
mxPH 1
mnO2    1
Cl         1
NO3           1
NH4           ,  1
oPO4    .  .        1
PO4     .  .        * 1
Chla .                  1
a1         .        . .    1
a2   .                  .     1
a3                               1
a4      .           . .             1
a5                                     1
a6            .  .                     .  1
a7                                           1
attr(,"legend")
[1] 0 ‘ ’ 0.3 ‘.’ 0.6 ‘,’ 0.8 ‘+’ 0.9 ‘*’ 0.95 ‘B’ 1
```

This symbolic representation of the correlation values is more legible, particularly for large correlation matrices.

In our data the correlations are in most cases irrelevant. However, there are two exceptions: between variables *NH4* and *NO3*; and between *PO4* and *oPO4*. These two latter variables are strongly correlated (above 0.9). The correlation between *NH4* and *NO3* is less evident (0.72) and thus it is risky to take advantage of it to fill in the unknowns. Moreover, assuming that you have removed the samples 62 and 199 because they have too many unknowns, there will be no water sample with unknown values on *NH4* and *NO3*. With respect to *PO4* and *oPO4* the discovery of this correlation[18] allows us to fill in the

[18]According to domain experts this was expected because the value of total phosphates (*PO4*) includes the value of orthophosphate (*oPO4*).

unknowns on these variables. In order to achieve this we need to find the form of the linear correlation between these variables. This can be done as follows,

```
> lm(PO4 ~ oPO4,data=algae)

Call:
lm(formula = PO4 ~ oPO4, data = algae)

Coefficients:
(Intercept)         oPO4
     42.897        1.293
```

The function `lm()` can be used to obtain linear models of the form $Y = \beta_0 + \beta_1 X_1 + \ldots + \beta_n X_n$. We will describe this function in detail on Section 2.6. The linear model we have obtained tells us that $PO4 = 42.897 + 1.293 * oPO4$. With this formula we can fill in the unknown values of these variables, provided they are not both unknown.

After removing the sample 62 and 199, we are left with a single observation with an unknown value on the variable *PO4* (sample 28), thus we could simply use the discovered relation to do the following,

```
> algae[28,'PO4'] <- 42.897 + 1.293 * algae[28,'oPO4']
```

However, for illustration purposes, let us assume that there were several samples with unknown values on the variable *PO4*. How could we use the above linear relationship to fill all the unknowns? The best would be to create a function that would return the value of *PO4* given the value of *oPO4*, and then apply this function to all unknown values[19],

```
> fillPO4 <- function(oP) {
+   if (is.na(oP)) return(NA)
+   else return(42.897 + 1.293 * oP)
+ }
> algae[is.na(algae$PO4),'PO4'] <-
+    sapply(algae[is.na(algae$PO4),'oPO4'],fillPO4)
```

The first instruction creates a function named `fillPO4()` with one argument, which is assumed to be the value of *oPO4*. Given a value of *oPO4*, this function returns the value of *PO4* according to the discovered linear relation (try issuing "`fillPO4(6.5)`"). This function is then applied to all samples with unknown value on the variable *PO4*. This is done using the function `sapply()`. This function has a vector as the first argument and a function as the second. The result is another vector with the same length, with the elements being the result of applying the function in the second argument to each element of the given vector[20]. This means that the result of this call to `sapply()` will be a vector with the values to fill in the unknowns of the variable *PO4*.

[19]Because there was a single case with unknown in 'PO4', if you have tried the previous instruction that filled in this unknown, R will complain with these following instructions. If you want to avoid this before trying the code set the 28th observation 'PO4' value back to unknown using for instance `algae[28,'PO4'] <- NA`.

[20]Try, for instance, `sapply(1:10,sqrt)`, where "sqrt" calculates the square root of a number.

The study of the linear correlations enabled us to fill in some new unknown values. Still, there are several observations left with unknown values. We may try to explore the correlations between the variables with unknowns and the nominal variables of this problem. We can use conditioned histograms that are available through the 'lattice' R package with this objective. For instance Figure 2.7, shows an example of such graph. This graph was produced as follows,

Correlation with nominal variables

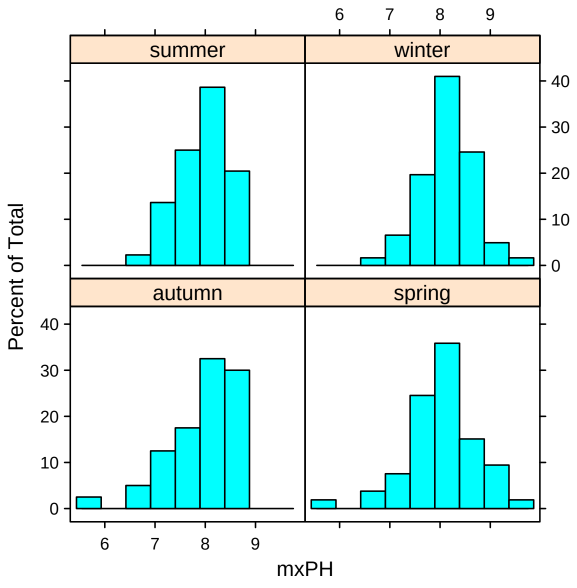

Figure 2.7: An histogram of variable *mxPH* conditioned by *season*.

```
> histogram(~ mxPH | season,data=algae)
```

Conditioned histograms

This instruction obtains an histogram of the values of *mxPH* for the different values of season. Each histogram is built using only the subset of observations with a certain season value. Notice that the histograms are rather similar thus leading us to conclude that the values of *mxPH* are not very influenced by the season of the year when the samples were collected. If we try the same using the size of the river, with `histogram(∼ mxPH | size,data=algae)`, we can observe a tendency for smaller rivers showing lower values of *mxPH*. We may extend our study of these dependencies using several nominal variables. For instance,

```
> histogram(~ mxPH | size*speed,data=algae)
```

shows the variation of *mxPH* for all combinations of size and speed of the rivers. It is curious to note that there is no information regarding small rivers with low speed[21]. The single sample that has these properties is exactly sample 48, the one for which we do not know the value of *mxPH*!

Another alternative to obtain similar information but now with the concrete values of the variable is,

[21] Actually, if you have executed the instruction given before, to fill in the value of *mxPH* with the mean value of this variable, this is not true anymore!

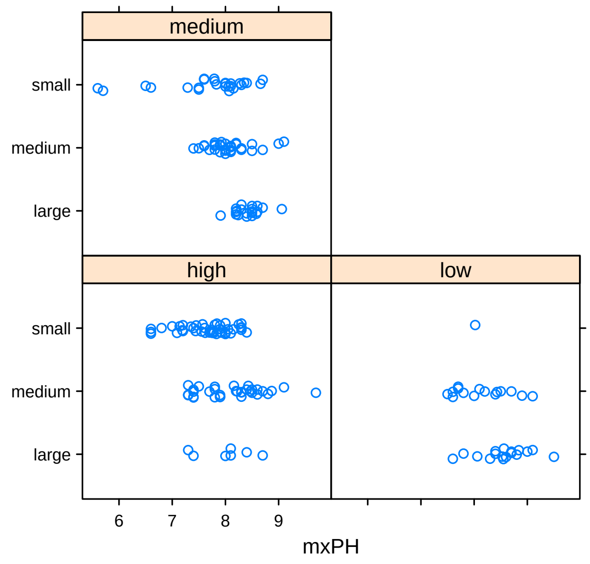

Figure 2.8: The values of variable *mxPH* by river size and speed.

```
> stripplot(size ~ mxPH | speed, data=algae, jitter=T)
```

The result of this instruction is shown in Figure 2.8. The `jitter=T` parameter setting is used to perform a small random permutation of the values in the Y direction to avoid plotting observations with the same values over each other, thus loosing some information on the concentration of observations with some particular value.

This type of analysis could be carried out for the other variables with unknown values. Still, this is a tedious process because there are too many combinations to be analyzed. Nevertheless, this is a method that can be applied in small data sets with few nominal variables.

2.5.4 Filling in the unknown values by exploring similarities between cases

Instead of exploring the correlation between the columns (variables) of a data set, we can try to use the similarities between the rows (observations) to fill in the unknown values. We will illustrate this method to fill in all unknowns with the exception of the two samples with too many NA's. Let us read in again the data to override the code of the previous sections (assuming you have tried it).

```
> algae <- read.table('Analysis.txt',
+            header=F,
+            dec='.',
+            col.names=c('season','size','speed','mxPH','mnO2','Cl','NO3',
+            'NH4','oPO4','PO4','Chla','a1','a2','a3','a4','a5','a6','a7'),
+            na.strings=c('XXXXXXX'))
> algae <- algae[-c(62,199),]
```

The approach described in this section assumes that if two water samples are similar, and one of them has an unknown value in some variable, there is a high probability that this value is similar to the value of the other sample. In order to use this intuitively appealing method we need to define the notion of similarity. This notion is usually defined by using a metric over the multivariate space of the variables used to describe the observations. Many metrics exist in the literature, but a common choice is the euclidean distance. This distance can be informally defined as the sum of the squared differences between the values of any two cases. The method we will describe below will use this metric to find the 10 most similar cases of any water sample with some unknown value in a variable. With these 10 cases, we will calculate their median value on this variable and use this value to fill in the unknown. Let us see how to implement this in R.

R has several packages with functions for calculating distances between cases. Most of them assume that the cases are described by numeric variables (*e.g.* the `dist()` function in the `stats` package). That is not the case of our problem where we have three nominal variables. As such we will use the package `cluster` that includes functions able to calculate distances using mixed mode variables.

Any distance function over a multivariate space will suffer from the existence of different scales of values among the variables. These differences can overweight the differences between the values on some variable over the differences on other variables. To avoid this, we will re-scale all variables to a "normalized" interval where every variable has a zero mean and unit standard deviation. The distance between all water samples can be obtained as follows,

```
> library(cluster)
> dist.mtx <- as.matrix(daisy(algae,stand=T))
```

The second instruction uses the function `daisy()` from the `cluster` package to calculate the distances. We have used the parameter `stand` to indicate that the data should be normalized before the distances are calculated. Moreover, we have converted the output of function `daisy`, which includes several information that we are not interested in this case, into a matrix of distances, using the `as.matrix()` function.

Let us remember which water samples have some unknown values and thus will need to be processed,

```
> which(!complete.cases(algae))
 [1]  28  38  48  55  56  57  58  59  60  61  62 115 160 183
```

Let us play a bit with sample number 38. Line 38 of the matrix `dist.mtx` has the distances between this water sample and all others. We can sort these distances and check the 10 most similar,

```
> sort(dist.mtx[38,])[1:10]
        38         54         22         64         11         30         25
0.00000000 0.02126003 0.05711782 0.05790530 0.06047142 0.06427236 0.06668811
        53         37         24
0.06677694 0.06983926 0.07609126
```

Notice how R has named the columns of the distance matrix so that we know which are the most similar water samples in the original data frame. The most similar observation to the sample 38 is ... the sample 38! After this sample come the samples 54, 22, 64, etc.. We can get a vector with the numbers of the 10 most similar water samples as follows,

```
> as.integer(names(sort(dist.mtx[38,])[2:11]))
 [1] 54 22 64 11 30 25 53 37 24 18
```

Now that we know the most similar water samples let us see which variable(s) we need to fill in,

```
> algae[38,]
   season  size speed mxPH mnO2   Cl  NO3 NH4 oPO4 PO4 Chla   a1 a2 a3 a4 a5 a6
38 spring small  high    8   NA 1.45 0.81  10  2.5   3  0.3 75.8  0  0  0  0  0
   a7
38  0
```

Thus, we want a value for *mnO2* for this water sample. According to the method we have outlined before we can use the median value on this variable in the 10 most similar samples,

```
> median(algae[as.integer(names(sort(dist.mtx[38,])[2:11])),'mnO2'])
[1] 10
```

This means that according to this method we should fill in the value of *mnO2* of the water sample 38 with the value 10, which we can do by,

```
> algae[38,'mnO2'] <-
+     median(algae[as.integer(names(sort(dist.mtx[38,])[2:11])),'mnO2'],
+            na.rm=T)
```

You may check the result with `algae[38,]`!

What if the water sample has more than one unknown value? We can use the function `apply()` to obtain the medians for each of the columns that are unknown. For instance, that is the case of the water sample 55 that has two unknown values,

```
> apply(algae[as.integer(names(sort(dist.mtx[55,])[2:11])),
+             which(is.na(algae[55,]))],
+       2,
+       median,na.rm=T)
    Cl   Chla
6.5835 0.8000
```

The function `apply()` can be used to apply a function to all columns (or rows) of a data frame. The first argument is the data frame. If the second argument has value 2 the function in the third argument is applied to all columns of the data frame. If it is 1, it is applied to all rows. Any argument after the third is passed to the function being applied. Notice that we have used `na.rm=T` as a fourth parameter to avoid problems with unknown values when calculating the medians. We should also remark that this will not work if there are unknowns in the nominal variables of the data frame because you would then try to calculate medians on discrete columns. In that case the best is to create a new function for obtaining the central value to be used, that will depend on whether the column is numeric or a factor,

```
> central.value <- function(x) {
+   if (is.numeric(x)) median(x,na.rm=T)
+   else if (is.factor(x)) levels(x)[which.max(table(x))]
+   else {
+     f <- as.factor(x)
+     levels(f)[which.max(table(f))]
+   }
+ }
```

The function `central.value()` will return the median for any numeric column, while for other type of columns it will transform them into factors and return the value that occurs more frequently. We can use this function instead of the median in the `apply` call mentioned before. Let us now automate this for all samples with some unknown value.

```
> for(r in which(!complete.cases(algae)))
+   algae[r,which(is.na(algae[r,]))] <-
+     apply(algae[as.integer(names(sort(dist.mtx[r,])[2:11])),
+                 which(is.na(algae[r,])),drop=F],
+           2,central.value)
```

There is a small (but important!) detail in the instructions above that deserves further explanations. Namely, when selecting the data frame to use as first argument of the `apply()` function, we have added a third argument `drop=F` to the indexing of the *algae* data frame. The reason for this argument is the fact that the `apply()` function requires that its first argument has more than one dimension, and R sometimes transforms data frames into vectors. This transformation occurs whenever the result of the indexing leads to a single dimension object. This transformation can be avoided using the `drop=F` argument, when indexing the data frame. Check the following examples that should illustrate this behavior[22],

```
> is.data.frame(algae[38,2])
[1] FALSE
> is.data.frame(algae[38,c(1,2,3)])
[1] TRUE
> is.data.frame(algae[38,2,drop=F])
[1] TRUE
```

We use a `for()` statement to go through all cases which have some missing value. This is an iterative statement that allows us to repeat the assignment statement for different values of the `r` variable.[23] With this `for` cycle consisting of a single assignment we are able to fill in all unknown values of our data frame, using the 10 most similar water samples to help finding the most probable value.

In summary, after these simple instructions we have the data frame free of NA values, and are better prepared to take full advantage of several R functions. Regarding which method should be used from the alternatives we have described, the answer is most of the times domain dependent. The method of exploring the similarities between cases seems to be more rational, although it

[22]The function `is.data.frame()` is true whenever its argument is a data frame.

[23]If you are not familiar with this programming constructs maybe it is a good time to review the material presented at Section 1.2.12.

suffers from some problems. These include possible existence of irrelevant variables that may distort the notion of similarity, or even excessive computational complexity for extremely large data sets. Still, for these large problems we can always use random samples to calculate the similarities.

Further readings on handling unknown values

The book *Data preparation for data mining* by Pyle (1999) is an extensive source of information on all issues of preparing data for data mining, which includes handling missing values. The book *Predictive data mining* by Weiss and Indurkhya (1999) is another good source of information on data preparation in general and unknown values in particular.
Hong (1997) and Wilson and Martinez (1997) are good references on distance measures involving variables with different types. Further references can also be found in Torgo (1999a).

2.6 Obtaining prediction models

The main goal of this case study is to obtain predictions for the frequency values of the 7 algae in a set of 140 water samples. Given that these frequencies are numbers, we are facing a regression problem[24]. In simple words, this task consists of trying to obtain a model relating a numerical variable with a set of other explanatory variables. This model can be used either to predict the value of the target variable for future observations of the explanatory variables, or to provide a better understanding of the interactions among the variables in our problem.

Regression problems

In this section we explore two different predictive models that could be applied to the algae domain: multiple linear regression and regression trees. Our choice was mainly guided by illustrative purposes in the context of this book, and not as a consequence of some formal model selection step. Still, these models are two good alternatives for regression problems as they are quite different in terms of their assumptions regarding the "shape" of the regression function being approximated, they are easy to interpret, and fast to run on any computer. This does not mean that in a real data mining scenario we should not try other alternatives and then use some form of model selection (*c.f.* Section 2.7) to select one or more of them for the final predictions on our 140 test samples.

The models we are going to try handle missing values in a different way. While linear regression is not able to use data sets with unknown values, regression trees handle these values naturally[25]. As such, we will follow a different path concerning the preparation of the data before model construction. For linear regression we will use one of the techniques described in Section 2.5 for pre-processing the data so that we can use these models. Regarding regression trees we will use the original 200 water samples.[26]

In the analysis we are going to carry out we will assume that we do not know the true values of the target variables for the 140 test samples. As we have mentioned before, the book Web page also includes a file with these solutions. Still, they are given just for you to get a final opinion on the value of the models we are going to obtain.

[24] Actually, as we want to predict 7 values for each water sample, one can handle this problem as 7 different regression problems.
[25] Obviously, we are referring to the implementations of these methods available in R.
[26] Actually, we will remove two of them because they have too many missing values.

2.6.1 Multiple linear regression

Multiple linear regression is among the most used statistical data analysis techniques. These models obtain an additive function relating a target variable with a set of predictor variables. This additive function is a sum of terms of the form $\beta_i \times X_i$, where X_i is a predictor variable and β_i is a number.

As we have mentioned before, there is no predefined way of handling missing values for this type of modeling techniques. As such we will use the data resulting from applying the method of exploring the similarities among the training cases to fill in the unknowns (*c.f.* Section 2.5.4). Nevertheless, before we apply this method, we will remove the water samples number 62 and 199 because, as mentioned before, they have 6 from the 11 predictor variables missing, which makes the task of filling them by exploring similarities too unreliable. The following code obtains a data frame without missing values,[27]

```
> algae <- read.table('Analysis.txt',
+              header=F,
+              dec='.',
+              col.names=c('season','size','speed','mxPH','mnO2','Cl','NO3',
+              'NH4','oPO4','PO4','Chla','a1','a2','a3','a4','a5','a6','a7'),
+              na.strings=c('XXXXXXX'))
> algae <- algae[-c(62,199),]
> clean.algae <- algae
> for(r in which(!complete.cases(algae)))
+    clean.algae[r,which(is.na(algae[r,]))] <-
+       apply(algae[as.integer(names(sort(dist.mtx[r,])[2:11])),
+                   which(is.na(algae[r,])),drop=F],
+             2,central.value)
```

After executing this code we have a data frame, `clean.algae`, that has no missing variable values.

Obtaining a linear regression model

Let us start by learning how to obtain a linear regression model for predicting the frequency of one of the algae.

```
> lm.a1 <- lm(a1 ~ .,data=clean.algae[,1:12])
```

The function `lm()` obtains a linear regression model. The first argument of this function[28] indicates the functional form of the model. In this example, it states that we want a model that predicts the variable *a1* using all other variables present in the data, which is the meaning of the dot character. For instance, if we wanted a model to predict *a1* as a function of the variables *mxPH* and *NH4*, we should have indicated the model as "`a1 ∼ mxPH + NH4`". There are other variants of this model language that we will introduce as necessary. The `data` parameter sets the data sample to be used to obtain the model[29].

We may obtain more information about the linear model with the following instruction,

```
> summary(lm.a1)

Call:
lm(formula = a1 ~ ., data = clean.algae[, 1:12])
```

[27] Reading in again the data is included only because you have probably tried the code of the previous sections that has changed the original data.

[28] Actually, of most functions used to obtain models in R.

[29] We have indicated the 11 explanatory variables plus the column respecting algal *a1*.

```
Residuals:
    Min      1Q  Median      3Q     Max
-37.582 -11.882  -2.741   7.090  62.143

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  43.210622  24.042849   1.797  0.07396 .
seasonspring  3.575474   4.135308   0.865  0.38838
seasonsummer  0.645459   4.020423   0.161  0.87263
seasonwinter  3.572084   3.863941   0.924  0.35647
sizemedium    3.321935   3.797755   0.875  0.38288
sizesmall     9.732162   4.175616   2.331  0.02086 *
speedlow      3.965153   4.709314   0.842  0.40090
speedmedium   0.304232   3.243204   0.094  0.92537
mxPH         -3.570995   2.706612  -1.319  0.18871
mnO2          1.018514   0.704875   1.445  0.15019
Cl           -0.042551   0.033646  -1.265  0.20761
NO3          -1.494145   0.551200  -2.711  0.00736 **
NH4           0.001608   0.001003   1.603  0.11072
oPO4         -0.005235   0.039864  -0.131  0.89566
PO4          -0.052247   0.030737  -1.700  0.09087 .
Chla         -0.090800   0.080015  -1.135  0.25796
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 17.64 on 182 degrees of freedom
Multiple R-Squared: 0.3737,        Adjusted R-squared: 0.3221
F-statistic:  7.24 on 15 and 182 DF,  p-value: 2.273e-12
```

Before we analyze the information provided by the function `summary()` when applied to linear models, let us say something on how R handled the three nominal variables. When using them as shown above, R will create a set of auxiliary variables[30]. Namely, for each factor variable with k levels, R will create $k-1$ auxiliary variables. These variables have the values 0 or 1. A value of 1 means that the associated value of the factor is "present", and that will also mean that the other auxiliary variables will have the value 0. If all $k-1$ variables are 0 then it means that the factor variable has the remaining kth value. Looking at the summary presented above, we can see that R has created three auxiliary variables for the factor *season* (`seasonspring`, `seasonsummer` and `seasonwinter`). This means that if we have a water sample with the value "autumn" in the variable *season*, all these three auxiliary variables will be set to zero.

Nominal variables on linear models

The application of the function `summary()` to a linear model gives some diagnostic information concerning the obtained model. First of all, we have information concerning the residuals (*i.e.* the errors) of the fit of the linear model to the used data. These residuals should have a mean zero and should have a normal distribution (and obviously be as small as possible!).

Diagnostic information regarding linear models

For each coefficient (variable) of the multiple regression equation, R will show its value and also its standard error (an estimate of the variation of these coefficients). In order to check the importance of each coefficient we may test the hypothesis that each of them is null, *i.e.* $H0 : \beta_i = 0$. To test this hypothesis the t test is normally used. R calculates a t value, which is defined as the ratio between the coefficient value and its standard error, *i.e.* $\frac{\beta_i}{s_{\beta_i}}$. R will show us a

Testing for null coefficients

[30]Often named *dummy* variables.

column (`Pr(>|t|)`) associated with each coefficient with the level at which the hypothesis that the coefficient is null is rejected. Thus a value of 0.0001, has the meaning that we are 99.99% confident that the coefficient is not null. R marks each test with a symbol corresponding to a set of common confidence levels used for these tests. In summary, coefficients that do not have any symbol in front of them, cannot be discarded as possibly null with a minimum confidence of at least 90%.

Proportion of explained variance

Another piece of relevant diagnostics information outputted by R, are the R^2 coefficients (Multiple and Adjusted). These indicate the degree of fit of the model to the data, that is the proportion of variance in the data that is explained by the model. Values near 1 are better (almost 100% explained variance), while the smaller the values the larger the lack of fit. The adjusted coefficient is more demanding as it takes into account the number of parameters of the regression model.

Explanation power of the variables

Finally, we can also test the null hypothesis that there is no dependence of the target variable on any of the explanatory variables, *i.e.* $H0 : \beta_1 = \beta_2 = \ldots = \beta_m = 0$. With this goal we use the F-statistic, which is compared to a critical value. R provides the confidence level at which we are sure to reject this null hypothesis. Thus a p-level of 0.0001, means that we are 99.99% confident that the null hypothesis is not true. Usually, if the model fails this test it makes no sense to look at the t-tests on the individual coefficients.

Graphical diagnostics of linear models

Some diagnostics may also be checked by plotting a linear model. In effect, we may issue a command like `plot(lm.a1)` to obtain a series of successive plots that help in understanding the performance of the model. One of the graphs simply plots each fitted target variable value against the respective residual (error) of the model. Larger errors are usually marked by adding the corresponding row number to the dot in the graph, so that you may inspect the observations if you wish. Another graph shown by R is a normal Q-Q plot of the errors that helps you to check if they follow a normal distribution[31] as they should.

The proportion of variance explained by this model is not very impressive (around 32.0%). Still, we can reject the hypothesis that the target variable does not depend on the predictors (the p value of the F test is very small). Looking at the significance of some of the coefficients we may question the inclusion of some of them in the model. There are several methods for simplifying regression models. In this section we will explore a method usually known as *backward elimination*.

Simplification of linear models by backward elimination

We will start our study of simplifying the linear model using the `anova()` function. When applied to a single linear model this function will give us a sequential analysis of variance of the model fit. That is, the reductions in the residual sum of squares (the total error of the model) as each term of the formula is added in turn. The result of this analysis for the model obtained above is shown below,

```
> anova(lm.a1)
Analysis of Variance Table

Response: a1
           Df Sum Sq Mean Sq F value    Pr(>F)
season      3     85      28  0.0906 0.9651499
size        2  11401    5701 18.3253 5.613e-08 ***
```

[31]Ideally, all errors would be in a straight line in this graph.

```
speed       2   3934    1967  6.3236 0.0022126 **
mxPH        1   1322    1322  4.2499 0.0406740 *
mnO2        1   2218    2218  7.1312 0.0082614 **
Cl          1   4451    4451 14.3073 0.0002105 ***
NO3         1   3399    3399 10.9263 0.0011420 **
NH4         1    385     385  1.2376 0.2674000
oPO4        1   4765    4765 15.3168 0.0001283 ***
PO4         1   1423    1423  4.5738 0.0337981 *
Chla        1    401     401  1.2877 0.2579558
Residuals 182  56617     311
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

These results indicate that the variable *season* is the variable that least contributes for the reduction of the fitting error of the model. Let us remove it from the model,

```
> lm2.a1 <- update(lm.a1, . ~ . - season)
```

Updating a model

The `update()` function can be used to perform small changes to an existing linear model. In this case we use it to obtain a new model by removing the variable *season* from the `lm.a1` model. The summary information for this new model is given below,

```
> summary(lm2.a1)

Call:
lm(formula = a1 ~ size + speed + mxPH + mnO2 + Cl + NO3 + NH4 +
    oPO4 + PO4 + Chla, data = clean.algae[, 1:12])

Residuals:
    Min      1Q  Median      3Q     Max
-36.386 -11.899  -2.941   7.338  63.611

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) 44.9587170 23.2659336   1.932  0.05484 .
sizemedium   3.3636189  3.7773655   0.890  0.37437
sizesmall   10.3092317  4.1173665   2.504  0.01315 *
speedlow     3.1460847  4.6155216   0.682  0.49632
speedmedium -0.2146428  3.1839011  -0.067  0.94632
mxPH        -3.2377235  2.6587542  -1.218  0.22487
mnO2         0.7741679  0.6578931   1.177  0.24081
Cl          -0.0409303  0.0333812  -1.226  0.22170
NO3         -1.5126458  0.5475832  -2.762  0.00632 **
NH4          0.0015525  0.0009946   1.561  0.12027
oPO4        -0.0061577  0.0394710  -0.156  0.87620
PO4         -0.0508845  0.0304911  -1.669  0.09684 .
Chla        -0.0879751  0.0794655  -1.107  0.26969
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 17.56 on 185 degrees of freedom
Multiple R-Squared: 0.369,         Adjusted R-squared: 0.3281
F-statistic: 9.016 on 12 and 185 DF,  p-value: 1.581e-13
```

The fit has improved a bit (32.8%) but it is still not too impressive. We may carried out a more formal comparison between the two models by using again the `anova()` function, but this time with both models as arguments,

Comparing two linear models

```
> anova(lm.a1,lm2.a1)
Analysis of Variance Table

Model 1: a1 ~ season + size + speed + mxPH + mnO2 + Cl + NO3 + NH4 + oPO4 +
    PO4 + Chla
Model 2: a1 ~ size + speed + mxPH + mnO2 + Cl + NO3 + NH4 + oPO4 + PO4 +
    Chla
  Res.Df   RSS  Df Sum of Sq      F Pr(>F)
1    182 56617
2    185 57043  -3      -425 0.4559 0.7134
```

This function performs an analysis of variance of the two models using a F-test to assert the significance of the differences. In this case, although the sum of the squared errors has decreased (-425), the comparison shows that the differences are not significant (a value of 0.7134 tells us that with only around 29% confidence we can say they are different). Still, we should recall that this new model is simpler. In order to check if we can remove more coefficients we would use again the `anova()` function, applied to the `lm2.a1` model. This process would continue until we have no candidate coefficients for removal. However, to simplify our backward elimination process R has a function that performs all process for us.

The following code creates a linear model that results from applying the backward elimination method to the initial model we have obtained (`lm.a1`),[32]

```
> final.lm <- step(lm.a1)
Start:  AIC= 1151.85
 a1 ~ season + size + speed + mxPH + mnO2 + Cl + NO3 + NH4 + oPO4 +
    PO4 + Chla

         Df Sum of Sq   RSS   AIC
- season  3       425 57043  1147
- speed   2       270 56887  1149
- oPO4    1         5 56623  1150
- Chla    1       401 57018  1151
- Cl      1       498 57115  1152
- mxPH    1       542 57159  1152
<none>                56617  1152
- mnO2    1       650 57267  1152
- NH4     1       799 57417  1153
- PO4     1       899 57516  1153
- size    2      1871 58488  1154
- NO3     1      2286 58903  1158

Step:  AIC= 1147.33
 a1 ~ size + speed + mxPH + mnO2 + Cl + NO3 + NH4 + oPO4 + PO4 +
    Chla

        Df Sum of Sq   RSS   AIC
- speed  2       213 57256  1144
- oPO4   1         8 57050  1145
- Chla   1       378 57421  1147
- mnO2   1       427 57470  1147
- mxPH   1       457 57500  1147
- Cl     1       464 57506  1147
<none>               57043  1147
- NH4    1       751 57794  1148
- PO4    1       859 57902  1148
- size   2      2184 59227  1151
```

[32] We have omitted some of the output of the `step()` function for space reasons.

```
- NO3    1      2353 59396  1153

...
...

Step:  AIC= 1140.09
 a1 ~ size + mxPH + Cl + NO3 + PO4

       Df Sum of Sq   RSS   AIC
<none>              58432  1140
- mxPH  1       801 59233  1141
- Cl    1       906 59338  1141
- NO3   1      1974 60405  1145
- size  2      2652 61084  1145
- PO4   1      8514 66946  1165
```

The function `step()` uses the Akaike Information Criterion to perform model search. The search uses by default backward elimination, but with the parameter `direction` you may use other algorithms (check the help of this function for further details).

We can obtain the information on the final model by,

```
> summary(final.lm)


Call:
lm(formula = a1 ~ size + mxPH + Cl + NO3 + PO4, data = clean.algae[,
    1:12])

Residuals:
    Min      1Q  Median      3Q     Max
-28.876 -12.681  -3.688   8.393  62.875

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 57.63859   20.93604   2.753  0.00647 **
sizemedium   2.82560    3.39950   0.831  0.40691
sizesmall   10.39431    3.81809   2.722  0.00708 **
mxPH        -4.00980    2.47801  -1.618  0.10728
Cl          -0.05438    0.03160  -1.721  0.08692 .
NO3         -0.89215    0.35124  -2.540  0.01188 *
PO4         -0.05887    0.01116  -5.276 3.57e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 17.49 on 191 degrees of freedom
Multiple R-Squared: 0.3536,        Adjusted R-squared: 0.3333
F-statistic: 17.42 on 6 and 191 DF,  p-value: 4.857e-16
```

The proportion of variance explained by this model is still not very interesting! This kind of proportion is usually considered as a sign that the linearity assumptions of this model are inadequate for the domain.

Further readings on multiple linear regression models

Linear regression is one of the most used statistics techniques. As such, most statistics books will include a chapter on this subject. Still, specialized books should be used for deeper analysis. Two extensive books are the ones by Drapper and Smith (1981) and by Myers (1990). These books should cover most of the topics you will ever want to know about linear regression.

2.6.2 Regression trees

Let us now look at a different kind of regression model available in R. Namely, we will learn how obtain a regression tree (*e.g.* *Breiman et al., 1984*) to predict the value of the frequencies of algal *a1*. As these models handle data sets with missing values we only need to remove the samples 62 and 199 for the reasons mentioned before.

Obtaining a regression tree

The necessary instructions to obtain a regression tree are presented below:

```
> library(rpart)
> algae <- read.table('Analysis.txt',
+               header=F,
+               dec='.',
+               col.names=c('season','size','speed','mxPH','mnO2','Cl','NO3',
+               'NH4','oPO4','PO4','Chla','a1','a2','a3','a4','a5','a6','a7'),
+               na.strings=c('XXXXXXX'))
> algae <- algae[-c(62,199),]
> rt.a1 <- rpart(a1 ~ .,data=algae[,1:12])
```

The first instruction loads the 'rpart' package that implements regression trees in R.[33] The last instruction obtains the tree. Note that this function uses the same schema as the `lm()` function to describe the functional form of the model. The second argument of `rpart()` indicates which data to use to obtain the tree.

The content of the object `rt.a1` object is the following,

```
> rt.a1
n= 198

node), split, n, deviance, yval
      * denotes terminal node

 1) root 198 90401.290 16.996460
   2) PO4>=43.818 147 31279.120  8.979592
     4) Cl>=7.8065 140 21622.830  7.492857
       8) oPO4>=51.118 84  3441.149  3.846429 *
       9) oPO4< 51.118 56 15389.430 12.962500
        18) mnO2>=10.05 24  1248.673  6.716667 *
        19) mnO2< 10.05 32 12502.320 17.646880
          38) NO3>=3.1875 9   257.080  7.866667 *
          39) NO3< 3.1875 23 11047.500 21.473910
            78) mnO2< 8 13  2919.549 13.807690 *
            79) mnO2>=8 10  6370.704 31.440000 *
     5) Cl< 7.8065 7  3157.769 38.714290 *
   3) PO4< 43.818 51 22442.760 40.103920
     6) mxPH< 7.87 28 11452.770 33.450000
      12) mxPH>=7.045 18  5146.169 26.394440 *
      13) mxPH< 7.045 10  3797.645 46.150000 *
     7) mxPH>=7.87 23  8241.110 48.204350
      14) PO4>=15.177 12  3047.517 38.183330 *
      15) PO4< 15.177 11  2673.945 59.136360 *
```

[33]Actually, there is another package that also implements tree-based models, the package `tree`.

How to interpret the tree

A regression tree is a hierarchy of logical tests on some of the explanatory variables. Tree-based models automatically select the more relevant variables, thus not all variables need to appear in the tree. A tree is read from the *root node* that is marked by R with the number 1. R provides some information of the data in this node. Namely, we can observe that we have 198 samples (the overall training data used to obtain the tree) at this node, that these 198 samples have an average value for the frequency of algal *a1* of 16.99, and that the deviance[34] from this average is 90401.29. Each node of a tree has two branches. These are related to the outcome of a test on one of the predictor variables. For instance, from the root node we have a branch (tagged by R with "`2)`") for the cases where the test "$PO4 \geq 43.818$" is true (147 samples); and also a branch for the 51 remaining cases not satisfying this test (marked by R with "`3)`"). From node 2 we have two other branches leading to nodes 4 and 5, depending on the outcome of a test on *Cl*. This testing goes on until a *leaf node* is reached. These nodes are marked with asterisks by R. At these leaves we have the predictions of the tree. This means that if we want to use a tree to obtain a prediction for a particular water sample, we only need to follow a branch from the root node till a leaf, according to the outcome of the tests for this test sample. The average target variable value found at the leaf we have reached is the prediction of the tree for that sample.

Graphical representation of trees

We can also obtain a graphical representation of the tree as follows,

```
> plot(rt.a1,uniform=T,branch=1, margin=0.1, cex=0.9)
> text(rt.a1,cex=0.75)
```

The first instruction draws the tree, while the second labels the nodes of the tree. The other parameters have to do with graphical details of the tree presentation (character size, margins, etc.). They are not essential for the tree visualization, and we may even need to adjust slightly their values to better fit our presentation requirements.

Figure 2.9 shows the obtained tree. On this representation if the test at each node is truth you should follow the left branch, otherwise the right branch should be selected. Manipulating the various parameters of the functions used to draw trees you can obtain much better looking trees.

The `summary()` function can also be applied to tree objects. This will produce a lot of information concerning the tests on the tree, the alternative tests that could be considered and also the surrogate splits. These splits are part of the strategy used in R regression trees to handle unknown values.

Pruning trees

Trees are usually obtained in two steps. Initially, a large tree is grown, and then this tree is pruned by deleting bottom nodes through a process of statistical estimation. This process has the goal of avoiding overfitting. This has to do with the fact that an overly grown tree will fit the training data almost perfectly, but will be capturing spurious relationships of the sample (overfitting), and thus will perform badly when faced with a new data sample for which predictions are necessary. The overfitting problem occurs in many modelling techniques, particularly when the assumptions regarding the function to aproximate are more relaxed. These models although having a wider application range (due to these relaxed criteria), suffer from this overfitting problem, thus demanding for a statistical estimation step which precludes this effect.

[34]The sum of squared differences from the average.

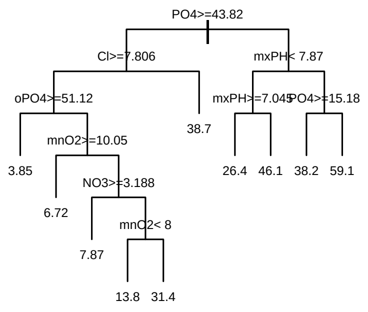

Figure 2.9: A regression tree for predicting algal *a1*.

The function `rpart()` that we have used to obtain our tree only grows the tree, stopping when certain criteria are met. Namely, the tree stops growing whenever the decrease in the deviance goes below a certain threshold; when the number of samples in the node is less than another threshold; or when the tree depth exceeds another value. These thresholds are controled by the parameters `cp`, `minsplit` and `maxdepth`, respectively. Their default values are 0.01, 20 and 30, respectively.

If we want to avoid the overfitting problem we should always check the validity of these default tree growth stopping criteria. This can be carried out through a process of post-pruning of the obtained tree. The `rpart` package implements a pruning method named *cost complexity* pruning. This method uses the values of the parameter `cp` that R calculates for each node of the tree. The pruning method tries to estimate the value of `cp` that ensures the best compromise between predictive accuracy and tree size. Given a tree obtained with the `rpart()` function, R can produce a set of sub-trees of this tree and estimate their predictive performance. This information can be obtained using the function `printcp()`,[35]

```
> printcp(rt.a1)

Regression tree:
rpart(formula = a1 ~ ., data = algae[, 1:12])

Variables actually used in tree construction:
[1] Cl   mnO2 mxPH NO3  oPO4 PO4

Root node error: 90401/198 = 456.57
```

[35]You may obtain similar information in a graphical form using the function `plotcp(rt.a1)`.

```
n= 198

        CP nsplit rel error  xerror    xstd
1 0.405740      0   1.00000 1.00737 0.13075
2 0.071885      1   0.59426 0.65045 0.10913
3 0.030887      2   0.52237 0.65470 0.10912
4 0.030408      3   0.49149 0.69417 0.11537
5 0.027872      4   0.46108 0.70211 0.11682
6 0.027754      5   0.43321 0.70211 0.11682
7 0.018124      6   0.40545 0.68015 0.11488
8 0.016344      7   0.38733 0.71108 0.11552
9 0.010000      9   0.35464 0.70969 0.11522
```

The tree produced by the `rpart()` function is the last tree of this list (tree 9). This tree has a value of `cp` of 0.01 (the default value of this parameter), includes 9 tests and has a relative error (compared to the root node) of 0.354. However, R estimates, using an internal process of 10-fold cross validation, that this tree will have an average relative error[36] 0.70969 ± 011522. Using the information provided by these more reliable estimates of performance, which avoid the overfitting problem, we can observe that we would theoretically be better off with the tree number 2, which has a lower estimated relative error (0.65045). An alternative selection rule is to choose the best tree according to the 1-SE rule. This consists of looking at the cross validation error estimates ("xerror" columns) and their standard deviations ("xstd" column). In this case the 1-SE tree is the smallest tree with error less than 0.65045 + 0.10913 = 0.75958, which in this case is the same as the tree with lowest estimated error (the tree at line 2). If we prefer this tree to the one suggested by R, we can obtain it by using the respective `cp` value[37],

The 1-SE rule

```
> rt2.a1 <- prune(rt.a1,cp=0.08)
> rt2.a1
n= 198

node), split, n, deviance, yval
      * denotes terminal node

1) root 198 90401.29 16.996460
  2) PO4>=43.818 147 31279.12  8.979592 *
  3) PO4< 43.818 51 22442.76 40.103920 *
```

We can automate this grow an prune steps using the following functions,

```
> reliable.rpart <- function(form,data,se=1,cp=0,verbose=T,...) {
+   tree <- rpart(form,data,cp=cp,...)
+   if (verbose && ncol(tree$cptable) < 5)
+     warning("No pruning will be carried out because no estimates were obtained.")
+   rt.prune(tree,se,verbose)
+ }
> rt.prune <- function(tree,se=1,verbose=T,...) {
+   if (ncol(tree$cptable) < 5) tree
```

[36]It is important to note that you may have obtained different numbers on the columns 'xerror' and 'xstd'. The cross validation estimates are obtained using a random sampling process, meaning that your samples will probably be different and thus the results will also differ.

[37]Actually, a value between the `cp`'s of the trees in line 1 and 2.

```
+   else {
+     lin.min.err <- which.min(tree$cptable[,4])
+     if (verbose && lin.min.err == nrow(tree$cptable))
+       warning("Minimal Cross Validation Error is obtained
+               at the largest tree.\n  Further tree growth
+              (achievable through smaller 'cp' parameter value),\n
+               could produce more accurate tree.\n")
+     tol.err <- tree$cptable[lin.min.err,4] + se * tree$cptable[lin.min.err,5]
+     se.lin <- which(tree$cptable[,4] <= tol.err)[1]
+     prune.rpart(tree,cp=tree$cptable[se.lin,1]+1e-9)
+   }
+ }
```

Using this function with its default parameter values we will obtain the 1-SE tree,

```
> (rt.a1 <- reliable.rpart(a1 ~ .,data=algae[,1:12]))
n= 198

node), split, n, deviance, yval
      * denotes terminal node

1) root 198 90401.29 16.996460
  2) PO4>=43.818 147 31279.12  8.979592 *
  3) PO4< 43.818 51 22442.76 40.103920 *
```

The trees are grown with a `cp` value of 0, which ensures a very large initial tree. This avoids stop growing too soon, which has the danger of missing some interesting tree model. In any case, if the chosen tree is the last of the `cp` table obtained with `printcp()`, our functions will issue a warning, suggesting to decrease the `cp` value. You may also add any of the other `rpart()` function parameters to the call of our `reliable.rpart` function, because these will be passed to the `rpart` function. That is the goal of the three dots in the arguments of the function. Their goal is to accept any other parameters apart from the ones presented in the function description, and in this case to pass them to the function `rpart()`. The function also checks whether the user has turned off cross-validation estimates (which is possible though the `xval` parameter). In this case no pruning is carried out and a warning is printed.

Interactive pruning

R also allows a kind of interactive pruning of a tree through the function `snip.rpart()`. This function can be used to generate a pruned tree in two ways. The first consists of indicating the number of the nodes (you can obtain these numbers by printing a tree object) at which you want to prune the tree,

```
> first.tree <- rpart(a1 ~ .,data=algae[,1:12])
> snip.rpart(first.tree,c(4,7))
n= 198

node), split, n, deviance, yval
      * denotes terminal node

 1) root 198 90401.290 16.996460
   2) PO4>=43.818 147 31279.120  8.979592
     4) Cl>=7.8065 140 21622.830  7.492857 *
     5) Cl< 7.8065 7  3157.769 38.714290 *
   3) PO4< 43.818 51 22442.760 40.103920
     6) mxPH< 7.87 28 11452.770 33.450000
      12) mxPH>=7.045 18  5146.169 26.394440 *
      13) mxPH< 7.045 10  3797.645 46.150000 *
     7) mxPH>=7.87 23  8241.110 48.204350 *
```

Note that the function returns a tree object like the one returned by the `rpart()` function, which means that you can store your pruned tree using something like `my.tree <- snip.rpart(first.tree,c(4,7))`.

Alternatively, you may use `snip.rpart()` in a graphical way. First, you plot the tree, and then you call the function without the second argument. If you click with the mouse at some node, R prints on its console some information about the node. If you click again on that node, R prunes the tree at that node[38]. You can go on pruning nodes in this graphical way. You finish the interaction by clicking the right mouse button. The result of the call is again a tree object,

```
> plot(first.tree)
> text(first.tree)
> snip.rpart(first.tree)
node number: 2  n= 147
    response= 8.979592
    Error (dev) =  31279.12
node number: 6  n= 28
    response= 33.45
    Error (dev) =  11452.77
n= 198

node), split, n, deviance, yval
      * denotes terminal node

 1) root 198 90401.290 16.996460
   2) PO4>=43.818 147 31279.120  8.979592 *
   3) PO4< 43.818 51 22442.760 40.103920
     6) mxPH< 7.87 28 11452.770 33.450000 *
     7) mxPH>=7.87 23  8241.110 48.204350
      14) PO4>=15.177 12  3047.517 38.183330 *
      15) PO4< 15.177 11  2673.945 59.136360 *
```

In this example, I have clicked and pruned nodes 2 and 6.

Further readings on regression trees

The more complete study on regression trees is probably the book by Breiman et al. (1984). This is the standard reference on both classification and regression trees. It provides a deep study of these two types of models. The approach can be seen as a bit formal (at least in some chapters) for some readers. Nevertheless, it is definitely a good reference although slightly biased towards statistical literature. The book on the system C4.5 by Quinlan (1993) is a good reference on classification trees from the machine learning community perspective. My Ph.D thesis (Torgo, 1999a), which you can freely download from my home page, should provide you with a good

[38]Note that the plot of the tree is not updated, so you will not see the pruning being carried out in the graphics window.

introduction, references and advanced topics on regression trees. It will also introduce you to other types of tree-based models that have the goal of improving the accuracy of regression trees by using more sophisticated models at the leaves (see also Torgo (2000)).

2.7 Model evaluation and selection

In Section 2.6 we have seen two examples of prediction models that could be used on this case study. The obvious question is which one should we use for obtaining the predictions for the 7 algae of the 140 test samples. To answer this question one needs to specify some preference criteria over the space of possible models, *i.e.* we need to specify how we will evaluate the performance of the models.

Model selection criteria

Several criteria exist for evaluating (and thus comparing) models. Among the most popular are criteria that calculate the predictive performance of the models. Still, other criteria exist like for instance the model interpretability, or even the model computational efficiency that can be important for very large data mining problems.

Predictive performance of regression models

The predictive performance of regression models is obtained by comparing the predictions of the models with the real values of the target variables, and calculating some average error measure from this comparison. One of such measures is the mean absolute error (MAE). Let us see how to obtain this measure for our two models (linear regression and regression trees). The first step is to obtain the model predictions for the set of cases where we want to evaluate it. To obtain the predictions of any model in R, one uses the function `predict()`. This general function peeks a model and a set of data and retrieves the model predictions,

Obtaining model predictions

```
> lm.predictions.a1 <- predict(final.lm,clean.algae)
> rt.predictions.a1 <- predict(rt.a1,algae)
```

These two statements collect the predictions of the models obtained in Section 2.6 for algal *a1*. Note that we have used the `clean.algae` data frame with linear models, because of the missing values.

Mean absolute error

Having the predictions of the models we can calculate their mean absolute error as follows,

```
> (mae.a1.lm <- mean(abs(lm.predictions.a1-algae[,'a1'])))
[1] 13.10279
> (mae.a1.rt <- mean(abs(rt.predictions.a1-algae[,'a1'])))
[1] 11.61717
```

Mean squared error

Another popular error measure is the mean squared error (MSE). This measure can be obtained as follows,

```
> (mse.a1.lm <- mean((lm.predictions.a1-algae[,'a1'])^2))
[1] 295.1097
> (mse.a1.rt <- mean((rt.predictions.a1-algae[,'a1'])^2))
[1] 271.3226
```

This latter statistic has the disadvantage of not being measured in the same units as the target variable, and thus being less interpretable from the user perspective. Even if we use the MAE statistic we can ask ourselves the question whether the scores obtained by the models are good or bad. An alternative statistic that provides a reasonable answer to this question is the normalized mean squared error (NMSE). This statistic calculates a ratio between the performance of our models and that of a baseline predictor, usually taken as the mean value of the target variable,

Normalized mean squared error

```
> (nmse.a1.lm <- mean((lm.predictions.a1-algae[,'a1'])^2)/
+                mean((mean(algae[,'a1'])-algae[,'a1'])^2))
[1] 0.6463594
> (nmse.a1.rt <- mean((rt.predictions.a1-algae[,'a1'])^2)/
+                mean((mean(algae[,'a1'])-algae[,'a1'])^2))
[1] 0.5942601
```

The NMSE is a unit-less error measure with values usually ranging from 0 to 1. If your model is performing better than this very simple baseline predictor then the NMSE should be clearly below 1. The smaller the NMSE, the better. Values above 1 mean that your model is performing worse than simply predicting always the average for all cases!

Errors scaterplot

Occasionally, it may also be interesting to have some kind of visual inspection of the predictions of the models, using a scaterplot of the errors. The following is an example using the predictions of our two models (*c.f.* the result in Figure 2.10),

```
> old.par <- par(mfrow=c(2,1))
> plot(lm.predictions.a1,algae[,'a1'],main="Linear Model",
+      xlab="Predictions",ylab="True Values")
> abline(0,1,lty=2)
> plot(rt.predictions.a1,algae[,'a1'],main="Regression Tree",
+      xlab="Predictions",ylab="True Values")
> abline(0,1,lty=2)
> par(old.par)
```

The first instruction sets one of the many parameters of the graphics system of R. The `mfrow` parameter allows us to divide the figure region in a kind of matrix of plots, providing means for presenting several plots in the same figure. In this case we have set it to a matrix with one row and two columns. After setting this value, any subsequent calls to the `plot()` function will fill each of these matrix elements in turn. Finally, the last call to the `par()` function sets the graphical parameters to what they were before our changes.

Looking at Figure 2.10 we can observe that the models have a rather poor performance in several cases. In the ideal scenario that they make correct predictions for all cases, all the circles in the plots should lie on the dashed lines, that were obtained with the `abline(0,1,lty=2)` calls. These lines have a 45 degrees slope. Given that each circle in the plots gets its coordinates from the predicted and truth values of the target variable, if these values were equal the circles would all lie on this ideal line. As we can observe that is not the case at all! We can check which is the sample number where a particularly bad prediction is made with the function `identify()`,

Interactive identification of samples

```
> plot(lm.predictions.a1,algae[,'a1'],main="Linear Model",
+      xlab="Predictions",ylab="True Values")
```

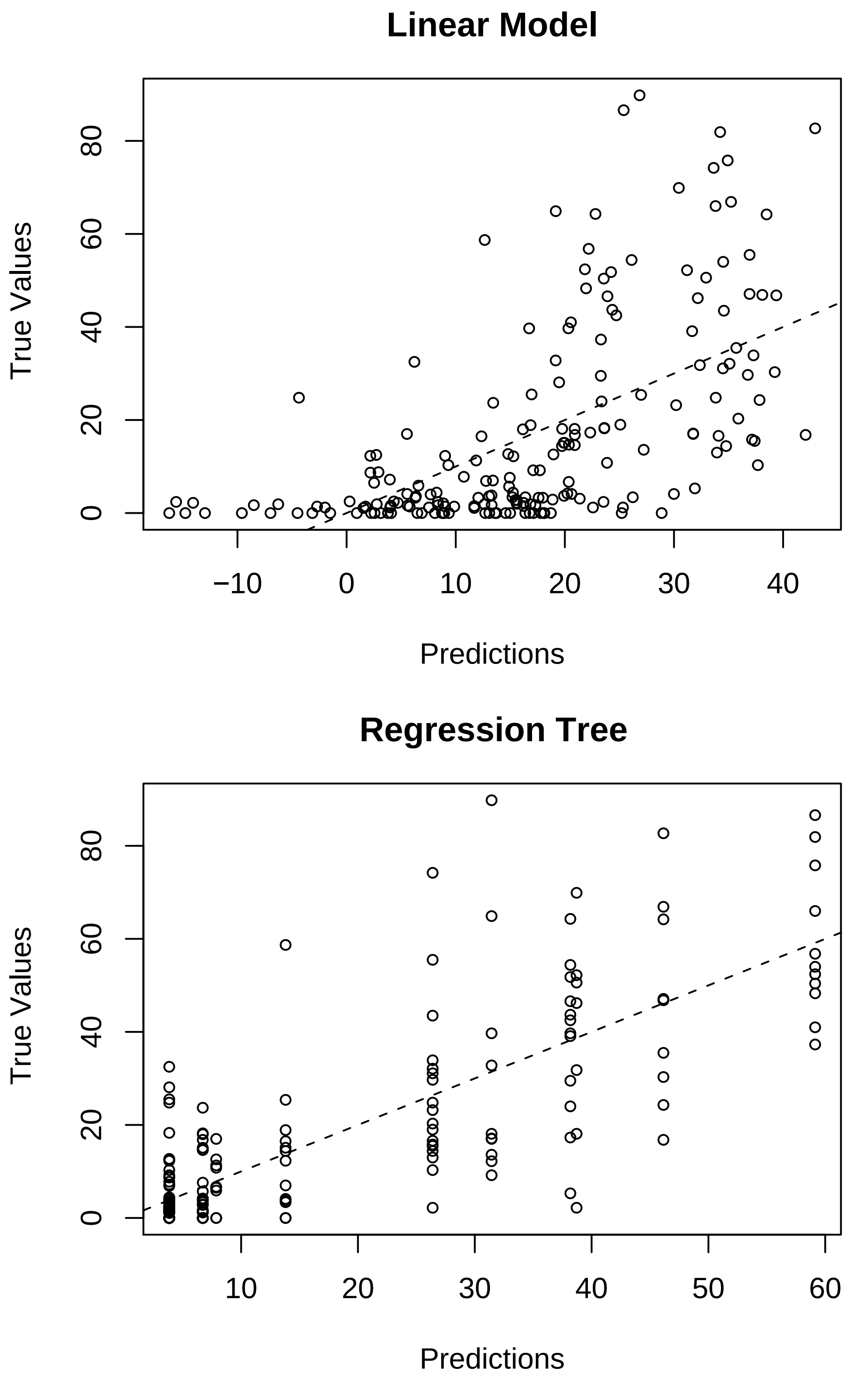

Figure 2.10: Errors scaterplot.

```
> abline(0,1,lty=2)
> identify(lm.predictions.a1,algae[,'a1'])
```

After the call to the function `identify()`, R enters in an interactive mode where the user is allowed to click with the left mouse button on any of the circles in the plot. For each clicked circle a number appears. This number is the row number in the `algae` data frame corresponding to that particular prediction. Try clicking the worse predictions. To end this interactive mode click on the right button of your mouse. R returns a vector with the row numbers of the clicked circles. Taking advantage of this, we could see the complete information regarding these water samples by issuing the following command instead of the `identify()` call used above,

```
> plot(lm.predictions.a1,algae[,'a1'],main="Linear Model",
+      xlab="Predictions",ylab="True Values")
> abline(0,1,lty=2)
> algae[identify(lm.predictions.a1,algae[,'a1']),]
```

Using this alternative, after finishing the interaction with the graphics window, you should see the rows of the `algae` data frame corresponding to the clicked circles, because we are using the vector returned by the `identify()` function to index the `algae` data frame.

Looking at the graph with the predictions of the linear model we can see that this model predicts negative algae frequencies for some cases. In this application domain it makes no sense to say that the occurrence of an algal in a water sample is negative (at most it can be zero). As such, we can take advantage of this domain knowledge and use this minimum value as a form of improving the linear model performance,

```
> sensible.lm.predictions.a1 <- ifelse(lm.predictions.a1 < 0,0,lm.predictions.a1)
> (mae.a1.lm <- mean(abs(sensible.lm.predictions.a1-algae[,'a1'])))
[1] 12.47114
> (nmse.a1.lm <- mean((sensible.lm.predictions.a1-algae[,'a1'])^2)/
+                mean((mean(algae[,'a1'])-algae[,'a1'])^2))
[1] 0.6257973
```

We have used the function `ifelse()` to achieve this effect. This function has 3 arguments. The first is a logical condition, the second is the result of the function call when the condition is true, while the third argument is the result when the condition is false. Notice how this small detail has increased the performance of our model!

According to the performance measures calculated above one should prefer the regression tree to obtain the predictions for the 140 test samples. However, there is a trap on this reasoning. Our goal is to choose the best model for obtaining the predictions on the 140 test samples. As we do not know the target variables values for those samples, we have to estimate which of our models will perform better on these test samples. The key issue here is to obtain a reliable estimate of a model performance on data for which we do not know the true target value. The measures calculated using the training data (as the ones obtained above) are unreliable, because they are biased. In effect, there are models that can easily obtain zero prediction error on the training

Reliable performance estimates

data. However, this performance will hardly generalize over new samples for which the target variable value is unknown. This phenomenon is usually known as *overfitting* the training data, as we have mentioned before. Thus to select a model one needs to obtain more reliable estimates of the models performance on unseen data. K-fold Cross Validation is among the most frequently used methods of obtaining these reliable estimates for small data sets like our case study. This method can be briefly described as follows. Obtain K equally sized and random sub-sets of the training data. For each of these K sub-sets, build a model using the remaining K-1 sets and evaluate this model on the Kth sub-set. Store the performance of the model and repeat this process for all remaining sub-sets. In the end we have K performance measures, all obtained by testing a model on data not used for its construction. The K-fold Cross Validation estimate is the average of these K measures. A common choice for K is 10. The following code puts these ideas in practice for our two models,

Overfitting

Cross validation

```
> cross.validation <- function(all.data,clean.data,n.folds=10) {
+
+   n <- nrow(all.data)
+   idx <- sample(n,n)
+   all.data <- all.data[idx,]
+   clean.data <- clean.data[idx,]
+
+   n.each.part <- n %/% n.folds
+
+   perf.lm <- vector()
+   perf.rt <- vector()
+
+   for(i in 1:n.folds) {
+     cat('Fold ',i,'\n')
+     out.fold <- ((i-1)*n.each.part+1):(i*n.each.part)
+
+     l.model <- lm(a1 ~ .,clean.data[-out.fold,1:12])
+     l.model <- step(l.model)
+     l.model.preds <- predict(l.model,clean.data[out.fold,1:12])
+     l.model.preds <- ifelse(l.model.preds < 0,0,l.model.preds)
+
+     r.model <- reliable.rpart(a1 ~ .,all.data[-out.fold,1:12])
+     r.model.preds <- predict(r.model,all.data[out.fold,1:12])
+
+     perf.lm[i] <- mean((l.model.preds-all.data[out.fold,'a1'])^2) /
+        mean((mean(all.data[-out.fold,'a1'])-all.data[out.fold,'a1'])^2)
+     perf.rt[i] <- mean((r.model.preds-all.data[out.fold,'a1'])^2) /
+        mean((mean(all.data[-out.fold,'a1'])-all.data[out.fold,'a1'])^2)
+   }
+
+   list(lm=list(avg=mean(perf.lm),std=sd(perf.lm),fold.res=perf.lm),
+        rt=list(avg=mean(perf.rt),std=sd(perf.rt),fold.res=perf.rt))
+ }

> cv10.res <- cross.validation(algae,clean.algae)
...
...
```

```
> cv10.res
$lm
$lm$avg
[1] 0.7719678

$lm$std
[1] 0.3045998

$lm$fold.res
 [1] 0.6603702 0.3752920 0.8724026 0.6825784 0.5895715 0.9561979 1.5006679
 [8] 0.6767427 0.5797774 0.8260770


$rt
$rt$avg
[1] 0.7538853

$rt$std
[1] 0.3368459

$rt$fold.res
 [1] 0.9575477 0.3213817 1.0339134 0.9735010 0.6873993 0.7501364 1.3070644
 [8] 0.2072840 0.5173976 0.7832269
```

The function `cross.validation()` implements the K-fold Cross Validation process outlined above, for the two models and for algal *a1*. The result of the function is a list with two components that are also lists. Each component contains the performance of one of the models (linear regression and regression trees). The performance of the models is described by the average performance (measured using the NMSE) over the K folds, the standard deviation of this performance and also a vector containing the performance on each individual fold.

As we can see from the output of the function, regression trees have a slightly better score (0.75 against 0.77)[39]. However, you may also note that there is a large variation of the scores on the different folds, which is also captured by the large standard deviations of both methods. We can carry out a formal statistical test to check whether the difference between the two means is statistically significant with some degree of confidence. The appropriate test for this situation is the paired *Wilcoxon* test. With this test we will check the hypothesis that the difference between the means of the two methods is zero. The following code performs the test for the results of our methods in the 10 folds,

Paired *Wilcoxon* tests

```
> wilcox.test(cv10.res$lm$fold.res,cv10.res$rt$fold.res,paired=T)

        Wilcoxon signed rank test

data:  cv10.res$lm$fold.res and cv10.res$rt$fold.res
V = 29, p-value = 0.9219
alternative hypothesis: true mu is not equal to 0
```

[39]You may obtain a different score if you try this code as there is a random component in the cross validation function (the call to the `sample()` function).

As we can see from the large *p-value* we cannot be confident that there is a significant difference between the performance of the two models. Thus the observed difference of mean error cannot be regarded as statisticaly significant.

In summary, according to these results we have no reasons to prefer one model over the other, in the task of predicting the concentrations of algal *a1*.

2.8 Predictions for the 7 algae

In this section we will see how to obtain the predictions for the 7 algae on the 140 test samples. Section 2.7 described how to proceed to choose a model to obtain these predictions. A similar procedure can be followed for all 7 algae. This process would lead us to the choice of a model to obtain the predictions for each of the 7 algae.

An alternative procedure is to carry out some kind of averaging over the models, instead of selecting one of them to obtain the predictions. This mixture of different models is frequently used as a form of reducing the final prediction error, by incorporating "different views" of the same training data. As such, we will include this mixture model as a third alternative to obtain the predictions of each algal.

In summary, for each algal, we will compare three alternative ways of obtaining the predictions for the 140 test samples: using a linear model; using a regression tree; or using a combination of the two models. The comparison will be carried out using a 10-fold cross validation process designed to estimate the performance of these alternatives in predicting the frequencies of the 7 algae. The estimated performance for the two "basic" models (linear models and regression trees) will be used to calculate a weight. This weight will enter the averaging process of the two models to obtain the predictions of the mixture model. The idea is to give more weight on this combination to the model which we estimate to have better performance.

In the end of this comparison process we will have information to decide which of the 3 models should be used for each of the 7 algae.

2.8.1 Preparing the test data

In this case study we have a separate file with test data, for which we want to obtain predictions for the 7 algae frequencies. Let us start by loading the test data, following a similar procedure as described in Section 2.3,

```
> test.algae <- read.table('Eval.txt',
+          header=F,
+          dec='.',
+          col.names=c('season','size','speed','mxPH','mnO2','Cl',
+          'NO3','NH4','oPO4','PO4','Chla'),
+          na.strings=c('XXXXXXX'))
```

Note that the test data does not contain the seven columns of the algae frequencies, which is reflected in the `col.names` parameter.

We will use the same filling method as the one used in Section 2.6.1 to prepare the test data for the linear regression models. As mentioned before, regression trees do not need any preparation as they handle unknown values.

```
> data4dist <- rbind(algae[,1:11],test.algae[,1:11])
> dist.mtx <- as.matrix(daisy(data4dist,stand=T))
> clean.test.algae <- test.algae
> for(r in which(!complete.cases(test.algae)))
+    clean.test.algae[r,which(is.na(test.algae[r,]))] <-
+       apply(data4dist[as.integer(names(sort(dist.mtx[r+198,])[2:11])),
+                        which(is.na(test.algae[r,])),drop=F],
+             2,central.value)
```

2.8.2 Comparing the alternative models

In this section we carry out a process of performance estimation for three alternative models: linear regression; a regression tree; and a combination of the predictions of both models. As mentioned before, we use a 10-fold cross validation method to estimate the NMSE of these models for each of the 7 algae. The code is quite similar to the one given in Section 2.7 with the main difference being the fact that we are estimating the accuracy for all 7 algae and using a third alternative model,

```
> cv.all <- function(all.data,clean.data,n.folds=10) {
+
+   n <- nrow(all.data)
+   idx <- sample(n,n)
+   all.data <- all.data[idx,]
+   clean.data <- clean.data[idx,]
+
+   n.each.part <- n %/% n.folds
+
+   perf.lm <- matrix(nrow=n.folds,ncol=7)
+   perf.rt <- matrix(nrow=n.folds,ncol=7)
+   perf.comb <- matrix(nrow=n.folds,ncol=7)
+
+   for(i in 1:n.folds) {
+     cat('Fold ',i,'\n')
+     out.fold <- ((i-1)*n.each.part+1):(i*n.each.part)
+
+     for(a in 1:7) {
+
+       form <- as.formula(paste(names(all.data)[11+a],"~."))
+       l.model <- lm(form,clean.data[-out.fold,c(1:11,11+a)])
+       l.model <- step(l.model)
+       l.model.preds <- predict(l.model,clean.data[out.fold,c(1:11,11+a)])
+       l.model.preds <- ifelse(l.model.preds < 0,0,l.model.preds)
+
+       r.model <- reliable.rpart(form,all.data[-out.fold,c(1:11,11+a)])
+       r.model.preds <- predict(r.model,all.data[out.fold,c(1:11,11+a)])
+
+       perf.lm[i,a] <- mean((l.model.preds-all.data[out.fold,11+a])^2) /
+             mean((mean(all.data[-out.fold,11+a])-all.data[out.fold,11+a])^2)
+       perf.rt[i,a] <- mean((r.model.preds-all.data[out.fold,11+a])^2) /
+             mean((mean(all.data[-out.fold,11+a])-all.data[out.fold,11+a])^2)
+
+       wl <- 1-perf.lm[i,a]/(perf.lm[i,a]+perf.rt[i,a])
+       wr <- 1-wl
+       comb.preds <- wl*l.model.preds + wr*r.model.preds
+       perf.comb[i,a] <- mean((comb.preds-all.data[out.fold,11+a])^2) /
+           mean((mean(all.data[-out.fold,11+a])-all.data[out.fold,11+a])^2)
+
+       cat(paste("Algal a",a,sep=""),"\tlm=",perf.lm[i,a],"\trt=",
+                 perf.rt[i,a],"\tcomb=",perf.comb[i,a],"\n")
+     }
+   }
+
+   lm.res <- apply(perf.lm,2,mean)
+   names(lm.res) <- paste("a",1:7,sep="")
+   rt.res <- apply(perf.rt,2,mean)
+   names(rt.res) <- paste("a",1:7,sep="")
+   comb.res <- apply(perf.comb,2,mean)
+   names(comb.res) <- paste("a",1:7,sep="")
+   list(lm=lm.res,rt=rt.res,comb=comb.res)
+ }

> all.res <- cv.all(algae,clean.algae)
...
...

> all.res
$lm
       a1        a2        a3        a4        a5        a6        a7
0.8001161 1.0645198 1.0554538 2.7642420 1.1138268 0.8801750 1.1648678

$rt
       a1        a2        a3        a4        a5        a6        a7
0.8967313 1.0000000 1.0047853 1.0000000 1.0000000 1.0000000 0.9500312

$comb
       a1        a2        a3        a4        a5        a6        a7
0.7360687 0.8641726 0.9463703 0.8990338 0.8936548 0.8401665 0.9491620
```

This may take a little while to run on slower computers and will surely produce lots of output!

The `cv.all()` function returns a list containing three vectors: one with the estimated NMSE of the linear models for all 7 algae; another with the same estimates for regression trees; and the third with these estimates for the combination strategy. We will use these estimates to decide which model to use for obtaining the predictions for the 140 test samples.

The function has some degree of complexity and it is worth spending some time trying to understand its functioning. Note how we have built the formula in the calls to `lm()` and `reliable.rpart()`. The `cv.all()` function obtains several models for each algal, and for each of the 10 iterations of the cross validation process. This is accomplished by using two `for()` cycles inside each other. The first iterates through the 10 repetitions of the 10-fold cross validation process. On each of these iterations the `i` variable takes a different value leading to a different set of data being left out for testing the models (*c.f.* the `out.fold` variable that depends on `i`). For each i iteration, the function obtains seven models, one for each algal. This is accomplished with another `for()` cycle. On each repetition of this inner cycle what varies is the target variable, which is successively the algal 1 to 7. As such, the single difference when calling the modeling functions is on the target variable, *i.e.* the formula argument. This means that on the first model we want the formula to be "a1 ∼ .", on the second "a2 ∼ .", and so on. In order to achieve this we have to "build" the formula at running time as the inner `for()` iterates from 1 to 7. This is accomplished with the call to the function `as.formula()` that can be used to transform a string into a model formula.

As we can observe from the results of the estimation process, the combination strategy, in spite of some poor NMSE scores (algae *a3* and *a7*), is the alternative with better estimated predictive accuracy for all algae. One can question the statistical significance of these observed differences. This could be asserted through paired *Wilcoxon* tests as the ones carried out in Section 2.7. However, even if some of the differences are not statistically significant we need to make a decision regards which model is going to be used for obtaining the predictions for the 140 test samples. As such, even on those cases we will choose the model with better estimated performance (even if the difference to the others is not statistically significant).

In summary, we will use the mixture model in all algae, when obtaining predictions for the 140 test samples.

2.8.3 Obtaining the prediction for the test samples

In this section we will obtain two regression models using the 200 training samples, for each of the 7 algae. These models will then be used to obtain predictions for the 140 test samples. The predictions of each of the models will then be weighed using the prediction accuracy estimates obtained in Section 2.8.2. These weighed predictions will be our final "bets" for the 140 test samples as a result of our predictive performance estimation process described in Section 2.8.2

The following code obtains 7 different linear models for the algae, constructing a list with several information concerning these models that we may later inspect or use in any way,

```
> lm.all <- function(train,test) {
+   results <- list()
+   results$models <- list()
+   results$preds <- list()
+   for (alg in 1:7) {
+     results$models[[alg]] <- step(lm(as.formula(paste(names(train)[11+alg],'~ .')),
+                                    data=train[,c(1:11,11+alg)]))
+     p <- predict(results$models[[alg]],test)
+     results$preds[[alg]] <- ifelse(p<0,0,p)
+   }
+   results
+ }

> lm.models <- lm.all(clean.algae,clean.test.algae)
```

The `lm.all()` function produces a list containing two sub-lists: one with the 7 linear models; and the other with the respective predictions of these models for the 140 test samples. This way of working is one of the advantages of R with respect to other statistical software. All models obtained in R are objects, and as such can be stored in variables for later inspection. For instance, if we were curious about the linear model obtained for algae *a5*, we can check its details at any time by simply doing,

```
> summary(lm.models$models[[5]])

Call:
lm(formula = a5 ~ season + size + speed + mnO2 + NO3 + NH4 +
    PO4, data = train[, c(1:11, 11 + alg)])

Residuals:
    Min      1Q  Median      3Q     Max
-12.945  -3.503  -0.975   2.143  35.770

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)  -5.0245574  3.4407693  -1.460   0.1459
seasonspring -1.9868754  1.5455275  -1.286   0.2002
seasonsummer  0.9967160  1.4982003   0.665   0.5067
seasonwinter -1.6702756  1.4389373  -1.161   0.2472
sizemedium    3.4485731  1.3323818   2.588   0.0104 *
sizesmall     0.1520604  1.4172271   0.107   0.9147
speedlow     -3.4456928  1.6615247  -2.074   0.0395 *
speedmedium  -0.3032838  1.1892515  -0.255   0.7990
mnO2          0.7030887  0.2622467   2.681   0.0080 **
NO3           0.4885728  0.1949781   2.506   0.0131 *
NH4          -0.0008542  0.0003647  -2.342   0.0202 *
PO4           0.0184992  0.0045891   4.031 8.09e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 6.608 on 186 degrees of freedom
Multiple R-Squared: 0.2695,         Adjusted R-squared: 0.2263
F-statistic: 6.238 on 11 and 186 DF,  p-value: 1.022e-08
```

One of the nicest things of having the model construction functions sharing a similar syntax, is that we can create a similar function to obtain regression trees by only having to change the function `lm()` into `reliable.rpart()`! Everything else stays the same[40] as you can see below,

```
> rt.all <- function(train,test) {
+   results <- list()
+   results$models <- list()
+   results$preds <- list()
+   for (alg in 1:7) {
+     results$models[[alg]] <- reliable.rpart(as.formula(paste(names(train)[11+alg],'~ .')),
+                                  data=train[,c(1:11,11+alg)])
+     results$preds[[alg]] <- predict(results$models[[alg]],test)
+   }
+   results
+ }

> rt.models <- rt.all(algae,test.algae)
```

Having obtained the predictions from the models considered in this case study, we are now ready to obtain the weighed average of these predictions that lead to our final predictions,

[40]Actually, we also remove the correction for negative predicted frequencies, as that does not happen with regression trees.

```
> final.preds <- function(lm.preds,rt.preds,ws) {
+   final <- matrix(nrow=140,ncol=7)
+   for (alg in 1:7) {
+     wl <- 1-ws$lm[alg]/(ws$lm[alg]+ws$rt[alg])
+     wr <- 1-wl
+     final[,alg] <- wl*lm.preds[[alg]] + wr*rt.preds[[alg]]
+   }
+   colnames(final) <- paste('a',1:7,sep='')
+   final
+ }
> final <- final.preds(lm.models$preds,rt.models$preds,all.res)
> head(final)
            a1       a2       a3       a4       a5       a6       a7
 [1,]  8.010443 7.659476 4.046844 2.968524 6.398159 3.801033 2.481505
 [2,] 13.092888 7.929399 3.754411 1.488492 5.184258 8.104343 2.585820
 [3,] 15.774637 7.748229 3.544444 2.191713 4.490437 5.437407 1.957751
 [4,] 14.933346 6.463206 5.465103 1.788305 3.737262 4.704890 1.709891
 [5,] 35.908769 7.717396 2.998125 1.682552 3.995119 2.810936 1.370024
 [6,] 36.034865 9.180016 2.220470 1.466830 5.023223 4.380623 1.370024
```

The matrix `final` contains our predictions for the 7 algae of the 140 test samples (above we are only showing the predictions for the first few algae, which can be easily achieved with the function `head()`).

We may be curious about how good (or bad) are these predictions. As mentioned in the beginning of this chapter we have the "solutions" for these 140 test samples in a text file at the book web page. We may compare our predictions with these true values just to check how far we got from the perfect predictions.

```
> algae.sols <- read.table('Sols.txt',
+                          header=F,dec='.',
+                          col.names=c('a1','a2','a3','a4','a5','a6','a7'))
> sq.errs <- (final-algae.sols)^2
> abs.errs <- abs(final-algae.sols)
> apply(sq.errs,2,mean)
        a1         a2         a3         a4         a5         a6         a7
229.572679 101.150321  28.111477   5.782261  82.443397 154.842940  21.871590
> apply(abs.errs,2,mean)
       a1        a2        a3        a4        a5        a6        a7
11.298151  7.139530  4.078631  1.811488  5.487225  7.543154  2.700523
> baseline.preds <- apply(algae[,paste('a',1:7,sep='')],2,mean)
> base.sq.errs <- (matrix(rep(baseline.preds,nrow(algae.sols)),byrow=T,ncol=T)
+                - algae.sols)^2
> apply(sq.errs,2,mean)/apply(base.sq.errs,2,mean)
       a1        a2        a3        a4        a5        a6        a7
0.4172368 0.7931353 0.5327987 0.1082848 0.7276379 0.7750680 0.3779473
```

The code presented above calculates the MSE, MAD and NMSE for all 7 algae. As we can observe from the NMSE results (the last calculation), the scores obtained with our models are quite good for some algae, when compared to the baseline predictor (predicting the average target value on the training data). Still, we can observe some relatively large average errors in the predictions of some of the algae (*e.g.* $a1$).

Now that we know the true values of the target variables for the 140 test samples, we can check whether our model selection strategy "did a good job", by comparing the accuracy we have obtained with our strategy, with the accuracy we would have obtained if we used only the linear models or the regression trees,

```
> rt.preds <- matrix(nrow=140,ncol=7)
> lm.preds <- matrix(nrow=140,ncol=7)
> for(a in 1:7) {
+       rt.preds[,a] <- rt.models$preds[[a]]
+       lm.preds[,a] <- lm.models$preds[[a]]
+ }
> rt.sq.errs <- (rt.preds-algae.sols)^2
> lm.sq.errs <- (lm.preds-algae.sols)^2
> apply(rt.sq.errs,2,mean)/apply(sq.errs,2,mean)
       a1        a2        a3        a4        a5        a6        a7
1.0843925 1.0616313 1.1254209 1.3584554 1.1186903 1.1580331 0.9889851
> apply(lm.sq.errs,2,mean)/apply(sq.errs,2,mean)
       a1        a2        a3        a4        a5        a6        a7
1.1361735 1.0120873 1.0004431 1.2750399 0.9679453 0.9349201 1.0688560
```

As we can observe, most NMSE's resulting from comparing the performance of regression trees and linear models against the combination strategy, are above 1. This means that the model selection strategy based on cross validation estimates performed well. Still, there are exceptions to this (algae $a5$ and $a6$ for linear models, and algal $a7$ for regression trees), which means that in these particular cases we would obtain more accurate predictions if we have used these models instead of the combination of their predictions. This serves as an alert to the risks we are taking whenever we are relying on processes of accuracy estimation: they have an associated estimation error and thus may be different from the truth value of the parameter we are estimating! Still, these methodologies are designed to be right on average, and in any case they are the best way to proceed whenever we do not know the true value of the target variables of our test cases and we have different models that can be used to obtain the predictions for these cases.

Further readings on combination of models

Combining different models (sometimes known as *ensemble learning*) is a hot topic in data mining. Techniques like *bagging* (Breiman, 1996) or *boosting* (Freund and Shapire, 1996; Shapire, 1990) are quite frequently used to increase the performance of base models. A good overview of research on this topics can be found in Dietterich (2000).

2.9 Summary

The main goal of this first case study was to familiarize the reader with R. With this purpose we have used a small problem at least by data mining standards. Our goal in this chapter was to introduce the reader to some of the existing techniques in R.

In case you are interested in knowing more about the international data analysis competition that was behind the data used in this chapter, you may browse through the competition Web page[41], or read some of the papers of the winning solutions (Bontempi et al., 1999; Chan, 1999; Devogelaere et al., 1999; Torgo, 1999b) to compare the data analysis strategies followed by these authors.

We hope that by now you are more acquainted with the interaction with R, and also familiarized with some of its features. Namely, you should have learned some techniques for:

[41] http://www.erudit.de/erudit/competitions/ic-99/.

- loading data from text files,
- descriptive statistics of data sets,
- basic visualization of data,
- handling data sets with unknown values,
- linear regression models,
- regression trees,
- model selection and comparison,
- and model combination.

Further cases studies will give you more details on these and other data mining techniques.

Chapter 3

Case Study 2: Predicting Stock Market Returns

The second case study addresses the problem of trying to build a stock trading system based on prediction models obtained with daily stock quotes data. We will apply different models to predict the future returns of the S&P 500 stock index. These predictions will be used together with a trading strategy to reach a decision regarding the market orders to generate. This chapter addresses several new data mining issues: (1) how to use R to analyze data stored in a database; (2) how to handle prediction problems where there is a time ordering among training cases (also known as time series); (3) and an example of the difficulties of translating model predictions into decisions/actions in real world applications.

3.1 Problem description and objectives

Stock market trading is an application domain with a big potential for data mining. In effect, the existence of an enormous amount of historical data suggests that data mining can provide a competitive advantage over human inspection of this data. On the other hand there are researchers claiming that the markets adapt so rapidly in terms of price adjustments that there is no space to obtain profits in a consistent way. This is usually known as the *efficient markets hypothesis*. This theory has been successively replaced by more relaxed versions that leave some space for trading opportunities due to market inefficiencies.

The general goal of stock trading is to maintain a portfolio of stocks based on buy and sell orders. The long term objective is to achieve as much profit as possible from these trading actions. In the context of this chapter we will constrain a bit more this general scenario. Namely, we will only "trade" a single security, actually a stock index. Given this security and an initial capital, we will try to maximize our profit over a future testing period by means of trading actions (Buy, Sell, Hold). Our trading strategy will use as basis for decision making the indications provided by the result of a data mining process.

This process will consist of trying to predict the future returns of the index based on a model obtained with historical quotes data. Thus our prediction model will be incorporated in a trading system that generates its decisions based on the predictions of the model. Our overall evaluation criteria will be the performance of this trading system, *i.e.* the profit/loss resulting from the actions of the system as well as some other statistics that are of interest to investors. This means that our main evaluation criteria will be the results of applying the knowledge discovered by our data mining process and not the predictive accuracy of the models developed during this process.

3.2 The available data

In our case study we will concentrate on trading the S&P 500 stock index. Daily data concerning the quotes of this security are freely available in many places, like for instance the Yahoo finance site[1].

The data that will be used in this case study is provided in two different formats at the book Web site[2]. The first is a comma separated values (CSV) file that can be read into R in the same way as the data used in Chapter 2. The book site also provides the data as a MySQL database dump file which we can use to create a database with the index quotes in MySQL. In this section, we will illustrate how to load this data into R for these two alternative ways of storing it. It is up to you to decide which alternative you will download. The rest of the chapter (*i.e.* the analysis after reading the data) is independent of the storage schema you decide to use.

Whichever the format you choose to download, the daily stock quotes data includes information regarding the following properties:

- Date of the stock exchange session.
- Open price at the begining of the session.
- Highest price during the session.
- Lowest price.
- Closing price of the session.

We will also illustrate a third alternative way of getting this and other stocks data directly. This alternative consists of taking advantage of R `tseries` package that includes a function to get the quotes of many stocks and stock indices directly from the Yahoo finance site.[3]

3.2.1 Handling time dependent data in R

The data available for this case study is dependent on time. This means that each observation of our data set has a time tag attached to it. This type of data is frequently known as time series data. The main distinguishing feature of this kind of data is that order between cases matters, due to their attached

Time series

[1] `http://finance.yahoo.com`.
[2] `http://www.liacc.up.pt/~ltorgo/DataMiningWithR`
[3] As long as you have your computer connected to the Internet.

time tags. Generally speaking a time series is a set of observations of a variable Y,

$$y_1, y_2, \ldots, y_{t-1}, y_t, y_{t+1}, \ldots, y_n \tag{3.1}$$

where, y_t is the value of the series variable Y at time t.

The main goal of time series analysis is to obtain a model based on past observations of the variable, $y_1, y_2, \ldots, y_{t-1}, y_t$, which allows us to make predictions regarding future observations of the variable, $y_{t+1}, \ldots, y_n$.

In the case of our stocks data we have what is usually known as a multivariate time series, because we have several variables being recorded at the same time tags, namely the *Open*, *High*, *Low* and *Close*.[4]

Time series objects

R has several packages devoted to the analysis of this type of data, and in effect it has special objects that are used to store type-dependent data. Namely, objects of class "ts" or "mts" can be used to store time series or multivariate time series, respectively. R has many functions specially tunned for this type of objects, like special plotting functions, etc.

Time series objects are vectors or matrices that have additional time-related attributes, like the starting time, the end time, etc. An important attribute is the sampling frequency, which determines the number of times the series is sampled in each unit of time. For instance, you could have a time series whose natural time unit is a year but which is sampled every quarter. The following is an example of such a series,

```
> ts(rnorm(25), frequency = 4, start = c(1959, 2))
            Qtr1        Qtr2        Qtr3        Qtr4
1959             -0.94949037 -0.09679323  0.28514212
1960 -0.28858372 -1.05443436 -0.89808154  0.99313512
1961 -0.52661363 -0.29520804 -2.84444432 -0.04904597
1962 -0.57365502 -0.32814304  1.54520560 -0.35873328
1963  0.33722051  0.01313544  2.98275830  0.22094523
1964 -0.51842271  0.91633143  0.32806589 -0.67932168
1965  1.85276121  2.26321250
```

The function `ts()` can be used to create time series objects. In this example we create an object from 25 normally distributed random numbers. This time series starts at the 2nd quarter of 1959, and as it is a quarterly sampled variable we set the frequency to 4. If we had used the value 12 for the frequency parameter we would get a monthly sampled time series,

```
> ts(rnorm(25), frequency = 12, start = c(1959, 2))
            Jan        Feb        Mar        Apr        May        Jun
1959            -1.1666875  0.0730924  0.7219297 -0.3368549 -0.3132447
1960 -0.1059628 -0.4633219 -1.2088039 -0.6267457 -0.4261878 -1.0771319
1961  0.9317941 -1.0811400
            Jul        Aug        Sep        Oct        Nov        Dec
1959 -1.0946693 -0.3296631  0.1893661  2.5913124  1.3165223  0.4877449
1960  0.3252741 -0.7814481 -0.8461871  0.2978047 -0.1522421 -0.6137481
1961
```

[4]Actually, if we wanted to be more precise we would have to say that we have only a single time series, because the quotes are actually the same variable (*Price*) sampled at different times of a day.

In spite of all their advantages, this type of time series objects are not particularly adequate to our data. In effect, one could be tempted to think of our data as a daily time series and thus store it as a "mts" object with frequency 7, which is used with daily sampled data. However, this would lead to an object with lots of "holes", the holidays and weekends where the markets are closed and thus we have no data! What we need is a data structure that can be used to store irregular time series, that is data that is not sampled at constant rates. R has several packages that can be used for this purpose. Namely, packages "zoo" and "its" provide this type of objects. Moreover, on package "tseries" there is also means to store this type of time series through the function `irts()`. We will use the package "its", and the class of objects it creates (objects of class "its"), to store our data, as we will see in the next sections. This is an extra package that you need to download from the CRAN site and install in R (*c.f.* Section 1.2.1).

Irregular time series

We can create an object of class "its" as follows,

```
> library(its)
> its(rnorm(5),dates=as.POSIXct(c('2003-01-01','2003-01-04',
+                                 '2003-01-05','2003-01-06',
+                                 '2003-02-16')))
                     1
2003-01-01  0.54026294
2003-01-04  1.18044581
2003-01-05  0.05492627
2003-01-06 -0.84049009
2003-02-16 -1.43544393
```

The function `its()` receives the time series data in the first argument. This can either be a vector, or a matrix if we have a multivariate time series. In the latter case each column of the matrix is interpreted as a variable being sampled at each time tag (*i.e.* each row). The time tags are indicated through the `dates` argument. This should contain a vector of POSIX dates, one for each element of the vector or row of the matrix. POSIX dates are one of the most common ways of representing dates in R. There are many functions associated with these objects for manipulating dates information, that you may want to check using the help facilities of R. In our case we have used function `as.POSIXct()` to create a vector of POSIX dates from a vector of strings representing the time tags.

In the case of multiple time series stored in a matrix the function `its()` can use the matrix dimension names to obtain both the time tags and the series names, as shown in the following example,

```
> mts.vals <- matrix(rnorm(25),5,5)
> rownames(mts.vals) <- c('2003-01-01','2003-01-04','2003-01-05',
+                         '2003-01-06','2003-02-16')
> colnames(mts.vals) <- paste('ts',1:5,sep='')
> obj <- its(mts.vals)
> obj
                  ts1         ts2        ts3        ts4         ts5
2003-01-01  0.4979519 -0.76274645  2.6698256  0.5219371  0.04234734
2003-01-04  0.7192498 -0.69294540  1.0902582  0.2752827 -1.41858946
2003-01-05 -0.4681299  0.55959956 -0.2785585 -2.1201813 -0.24449859
```

```
2003-01-06 -0.6092108 -0.00693051 -0.3824503 -2.0698096  0.41484473
2003-02-16 -1.3394441 -0.49645794 -0.5763984 -0.8137900  0.38754320
```

The objects of class “its” store the data and the time tags in two different slots. This information is seldom needed when using these objects but we will see in the next sections some examples where accessing the value of a particular slot is important. The following examples show some functions and operators available in R to find the names of the slots of any object and to access the values stored in the slots,

Slots of an object

Manipulating slots information

```
> slotNames(obj)
[1] ".Data" "dates"
> class(obj)
[1] "its"
> class(obj@.Data)
[1] "matrix"
> class(obj@dates)
[1] "POSIXt"  "POSIXct"
> obj@.Data
                 ts1         ts2        ts3        ts4         ts5
2003-01-01  0.4979519 -0.76274645  2.6698256  0.5219371  0.04234734
2003-01-04  0.7192498 -0.69294540  1.0902582  0.2752827 -1.41858946
2003-01-05 -0.4681299  0.55959956 -0.2785585 -2.1201813 -0.24449859
2003-01-06 -0.6092108 -0.00693051 -0.3824503 -2.0698096  0.41484473
2003-02-16 -1.3394441 -0.49645794 -0.5763984 -0.8137900  0.38754320
> obj@dates
[1] "2003-01-01 WET" "2003-01-04 WET" "2003-01-05 WET" "2003-01-06 WET"
[5] "2003-02-16 WET"
```

As we can see “its” objects store the time series values in a slot named “.Data” and the time tags in another slot named “dates”. The operator @ can be used to extract the slot values of any object.

In summary, “its” objects seem adequated to store stock quotes data, as they allow to store multiple time series with irregular time tags.

3.2.2 Reading the data from the CSV file

As we have mentioned before, at the book Web site you can find different sources containing the data to use in this case study. If you decide to use the CSV file, you will download a file whose first lines look like this,

```
# Query Results
# Connection: ltorgo@localhost:mysql.sock
# Host:
# Saved: 2003-10-22 16:46:43
#
# Query:
# SELECT Date, Open, High, Low, Close
# FROM ‘quotes‘ where Ticker='^GSPC'
#
'Date','Open','High','Low','Close',
'1970-01-02','92.06','93.54','91.79','93',
'1970-01-05','93','94.25','92.53','93.46',
'1970-01-06','93.46','93.81','92.13','92.82',
'1970-01-07','92.82','93.38','91.93','92.63',
'1970-01-08','92.63','93.47','91.99','92.68',
```

This CSV file was created from a MySQL database. The software that generated it has attached a header with some comments regarding the creation process, which obviously are not of interest to us in R. These comment lines all start with the character "#". All data values are within single quotes and all lines end with an extra comma. Let us see how can we overcome these "difficulties", and read in the data storing it in an "its" object,

```
> sp500 <- readcsvIts('sp500.txt',
+     col.names=c('Date','Open','High','Low','Close','X'),
+     quote = "'",
+     as.is=T,
+     comment.char='#',
+     header=T)
> sp500 <- its(sp500[,1:4])
> head(sp500)
            Open  High   Low Close
1970-01-02 92.06 93.54 91.79 93.00
1970-01-05 93.00 94.25 92.53 93.46
1970-01-06 93.46 93.81 92.13 92.82
1970-01-07 92.82 93.38 91.93 92.63
1970-01-08 92.63 93.47 91.99 92.68
1970-01-09 92.68 93.25 91.82 92.40
```

Reading an irregular time series from a file

A few comments on these instructions. We have used the function `readcsvIts()`, from the package "its", which is basically a wrapper for `read.table()` used in Chapter 2, but "tunned" for data files with values separated by commas and date information tagging each line, as it is the case of our CSV file. The fact that the file has a comma at the end of each line leads R to expect that each line has one extra field. Because of this we have added another variable to the column names ('X'). This column will be empty because there are no real values in the file (just an extra comma), and thus we do not use it when creating the "its" object with the `its()` function[5]. The `quote` parameter is used to tell R the character that is used to quote values (in this case the single quote). The `as.is` parameter is necessary because by default R transforms character columns into factors. Because all our values are quoted in the CSV file, the columns would be transformed into factors, which does not make sense for this data. This parameter is used to avoid this transformation. In case you only want to avoid this transformation for particular columns you may specify a vector with the numbers of these columns. The `comment.char` parameter is used to tell R that every line starting with a certain character should be disregarded as a comment line. We then use the `its()` function to transform the matrix obtained by the function `readcsvIts()` into an "its" object. Finally, we use the function `head()` to show the first lines of the object.

3.2.3 Reading the data from a MySQL database

The other alternative form of storing the data used in this case study is in a MySQL database. At the book Web site you have a file containing SQL statements that can be downloaded and executed within MySQL to upload S&P 500 quotes into a database table. Information on the use and creation of MySQL databases can be found at Section 1.3.

[5]Notice that the `readcsvIts()` function returns a matrix object without the date column. The date information is used to give names to the rows of the returned matrix.

After creating a database to store stock quotes we are ready to execute the SQL statements of the file downloaded from the book site. Assuming that this file is in the same directory from where you have entered MySQL, and that the database you have created is named `stocks`, you can type,

```
mysql> use stocks;
mysql> source sp500_db.sql;
```

The SQL statements contained in the file "`sp500_db.sql`" (the file downloaded from the book Web site) will create a table named "sp500" and insert several records in this table containing the data available for this case study. You may confirm that everything is OK by executing the following statements at the MySQL prompt,

```
mysql> show tables;
+------------------+
| Tables_in_stocks |
+------------------+
| sp500            |
+------------------+
1 row in set (0.03 sec)

mysql> select * from sp500;
```

The last SQL statement should print a large set of records, namely the quotes of S&P 500.

General interface to databases

R has a package named DBI, which implements a series of database interface functions. These functions are independent of the database server lying on the other end of the communication path. The user only needs to indicate which communication interface he will use at the first step when he establishes a connection to the database. This means that if you change your database management system (DBMS), you will only need to change a single instruction (the one which specifies the DBMS to which you wish to communicate). In order to achieve this independence the user also needs to install other packages that take care of the communication details for each different DBMS. R has many DBMS-specific packages for major DBMS's. Specifically, for communication with a MySQL database stored in some server you have two options: either to use the ODBC protocol via the RODBC package, or communicate directly with the server via the RMySQL package. The best choice depends on the operating system on which you are running R. Figure 3.1 shows the overall communication architecture between R and MySQL.

Loading the data into R running on Windows

If you are running R on Windows, independently of whether the MySQL database server resides on this same PC or in another computer (eventually running other operating system), the simplest way to connect to the database from R is through the ODBC protocol. In order to use this protocol in R you need to install the RODBC package apart from the DBI package.

Before you are able to connect to any MySQL database for the first time using the ODBC protocol, a few extra steps are necessary. Namely, you need

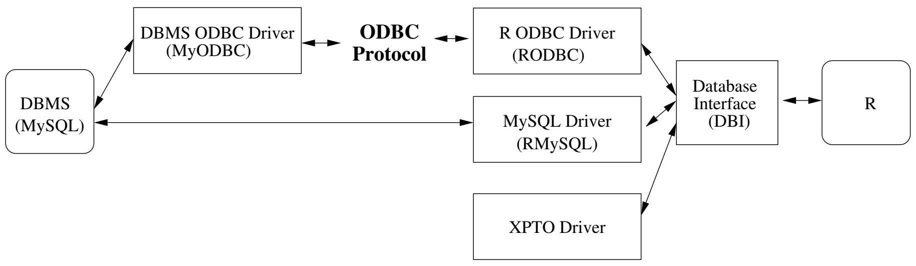

Figure 3.1: The communication architecture between R and MySQL.

also to install the MySQL ODBC driver on your Windows system, which is named "`myodbc`" and can be downloaded from the MySQL site. This only needs to be done the first time you use ODBC to connect to MySQL. After installing this driver you can create ODBC connections to MySQL databases residing on your computer or any other system to which you have access through your local network. According to the ODBC protocol every database connection you create has a name (the *Data Source Name*, or DSN according to the ODBC jargon). This name will be used to access the MySQL database from R. To create an ODBC connection on a Windows PC you have to use a program named "ODBC data sources" available at the Windows control panel. After running this program you have to create a new User Data Source using the MySQL ODBC driver (`myodbc`) that you are supposed to have previously installed. During this creation process you will be asked several things like the MySQL server address (`localhost` if it is your own computer, or e.g. `myserver.xpto.pt` if it is a remote server), the name of the database to which you wish to establish a connection (`stocks` in our previous example), and the name you wish to give to this connection (the DSN). Once you have completed this process, which you only have to do for the first time, you are ready to connect to this MySQL database from R.

The following R code establishes a connection to the `stocks` database from R, and loads in the S&P 500 quotes data into a data frame,

```
> library(RODBC)
> library(DBI)
> drv <- dbDriver("ODBC")
> ch <- dbConnect(drv,"stocksDSN","myusername","mypassword")
> sp500 <- dbGetQuery(ch,"select * from sp500")
> dbDisconnect(ch)
> dbUnloadDriver(drv)
> sp500 <- its(as.matrix(sp500[,2:5]),dates=as.POSIXct(sp500[,1]))
> sp500[1:4,]
            Open  High   Low Close
1970-01-02 92.06 93.54 91.79 93.00
1970-01-05 93.00 94.25 92.53 93.46
1970-01-06 93.46 93.81 92.13 92.82
1970-01-07 92.82 93.38 91.93 92.63
```

The first instructions load the packages that provide the functions allowing R to communicate with the database through the ODBC protocol. The following two instructions load the ODBC protocol driver and then establish a connection using this driver, with the DSN connection that you are supposed to have created before[6]. For this to be successful you need to provide your MySQL server username and password. Then comes the workhorse function, `dbGetQuery`, which allows you to send a query to the DBMS and get the results collected in a data frame. After closing the connection and unloading the driver, we then transform the obtained data frame into an "its" object using the `its()` function. The data to be stored in this object is a matrix of daily session quotes and the time tags for each session result from transforming the strings containing the dates on the first column of the data frame into a vector of POSIX dates, using the function `as.POSIXct()`. The result of this process is a similar object as the one you would get if loading the data from the CSV file.

A brief note on working with extremely large databases. If your query generates a result too large to fit in your computer main memory then you have to use some other strategy. The database interface package of R has several functions that allow you to send the query to the DBMS, but get the results in smaller chunks. Obviously, this means that you will also need to adjust your posterior data analysis steps since you will get the data in several steps and not in a single data frame.

Loading the data into R running on Linux

In case you are running R from a Linux box the easiest way to communicate to your MySQL database is through the RMySQL package. With this package you do not need any preparatory stages as with RODBC. After installing the package you can start using it as shown by the following example,

```
> library(RMySQL)
> library(DBI)
> drv <- dbDriver("MySQL")
> ch <- dbConnect(drv,dbname="stocks","myusername","mypassword")
> sp500 <- dbGetQuery(ch,"select * from sp500")
> dbDisconnect(ch)
[1] TRUE
> dbUnloadDriver(drv)
> sp500 <- its(as.matrix(sp500[,2:5]),dates=as.POSIXct(sp500[,1]))
> sp500[1:4,]
            Open  High   Low Close
1970-01-02 92.06 93.54 91.79 93.00
1970-01-05 93.00 94.25 92.53 93.46
1970-01-06 93.46 93.81 92.13 92.82
1970-01-07 92.82 93.38 91.93 92.63
```

Notice the advantages of the DBI package which leads to a code almost identical to the use of the ODBC protocol. In effect, you only need to change the driver that is loaded, and also change some parameters in the `dbConnect` function. Instead of specifying the DSN connection id, you provide the name of the database to which you want to communicate.

[6] Here you should substitute by whichever DSN name you have used when creating the data source in the Windows control panel.

3.2.4 Getting the data from the Web

Another alternative way of getting the S&P 500 quotes is to use the free service provided by Yahoo finance, which allows you to download a CSV file with the quotes you want. The `tseries` R package has a function (`get.hist.quote()`) that can be used to download the quotes into a R data structure. This is an example of the use of this function to get the quotes of S&P 500:

```
> library(tseries)
> sp500.web <- get.hist.quote("^GSPC",start="1970-01-02",end="2003-08-07",
+                              quote=c("Open", "High", "Low", "Close"))
> sp500.web[1:4,]
       Open   High    Low  Close
[1,] 139.23 139.23 139.23 139.23
[2,] 139.06 139.06 139.06 139.06
[3,] 138.87 138.87 138.87 138.87
[4,]     NA     NA     NA     NA
```

The `get.hist.quote()` function returns a time series object (or multiple time series as in this example). As we have seen in Section 3.2.1 this is not particularly adequate for this type of data. As a result the object **sp500.web** has several rows filled with **NA**'s, corresponding to the days where the markets were closed (weekends and holidays).

One of the beauties of Open Source software is that if you do not like something about a function, then you just need to look at its code and change it to better suite your needs. That is what we have done with the `get.hist.quote()` function. At the book Web site, in the code section for this chapter, you have a function called `its.get.hist.quote()`, which basically was written by pasting the code of the function `get.hist.quote()` and changing it slightly so that it returns an "its" object as we want. After pasting the code of this function into your R console you may use it as follows,

```
> sp500.web <- its.get.hist.quote("^GSPC",start="1970-01-02",end="2003-08-07",
+                              quote=c("Open", "High", "Low", "Close"))
> sp500.web[1:4,]
             Open   High    Low  Close
1982-10-20 139.23 139.23 139.23 139.23
1982-10-21 139.06 139.06 139.06 139.06
1982-10-22 138.87 138.87 138.87 138.87
1982-10-25 133.32 133.32 133.32 133.32
```

As you may have noticed the Yahoo site does not have the quotes since 1970-01-02, and thus the first date is 1982-10-20. As such, we recommend that you use for the rest of the chapter the data contained in either the CSV file or the MySQL database.

3.3 Data pre-processing

If we want to obtain a prediction model that can help us to trade the S&P 500 security we need to start by answering some basic questions like: what do we want to predict; or which predictor variables to use. Our overall objective is to take the correct decision at each time step from a set of three possibilities: sell, hold or buy. In order to help making the correct decision we can follow different paths in terms of model construction. Namely, we can try to obtain a model whose target variable is the decision, or alternatively we can construct a model that simply predicts the future evolution of the price time series, leaving the translation of these predictions into a trading action to a posterior stage. We will follow this latter alternative as it allows an easier integration of other important factors in developing a trading system, like risk management, as we will see later on this chapter.

In order to obtain a model to predict the future value of the prices we will use as basic information the previous values of these prices. The usual approach in financial time series analysis is to focus on predicting the closing prices of a security. Moreover, to avoid trend effects, it is common to use the percentage differences of the closing prices (or log transformations) as the basic time series being modeled, instead of the absolute values. Below we define these two alternative target variables,

$$Y_t = 100 \times \frac{Close_t - Close_{t-1}}{Close_{t-1}} \tag{3.2}$$

$$Y_t = \log \frac{Close_t}{Close_{t-1}} \tag{3.3}$$

We will adopt the first alternative because it has a more intuitive interpretation for the user. Namely, we will concentrate on developing models for a time series consisting of the h-days returns on closing prices defined as,

h-days returns

$$R_h(t) = \frac{Close_t - Close_{t-h}}{Close_{t-h}} \tag{3.4}$$

Summarizing, our series data will consist of the observations,

$$R_h(1), R_h(2), \ldots, R_h(t-1), R_h(t), R_h(t+1), \ldots, R_h(n) \tag{3.5}$$

The following R function obtains the h-days returns of a vector of values (for instance the closing prices of a stock). Actually, the function is implemented with “its” objects in mind, so it receives an “its” vector of quote prices and produces an “its” vector of the respective returns,

```
> h.returns <- function(x,h=1) {
+   its(100*c(rep(NA,h),diff(x,lag=h)),
+       dates=x@dates,
+       names=paste(dimnames(x)[[2]],'.',h,'-d.Rets',sep='')
+      )/x
+   }
> h.returns(sp500[1:6,'Close'],h=3)
           Close.3-d.Rets
1970-01-02             NA
1970-01-05             NA
1970-01-06             NA
1970-01-07     -0.3994386
1970-01-08     -0.8416055
1970-01-09     -0.4545455
> sp500[1:6,'Close']
           Close
```

```
1970-01-02 93.00
1970-01-05 93.46
1970-01-06 92.82
1970-01-07 92.63
1970-01-08 92.68
1970-01-09 92.40
```

Lagged differences

To create this function we have used the function `diff()`. This R function calculates lagged differences of a vector, *i.e.* $x_t - x_{t-lag}$.[7]

Notice how the first three sessions got the value NA. This occurs because there was no information on the closing prices necessary to calculate their 3 days past returns.

Using this function we will create a data set, which will then be used to obtain a model to predict the future h-days returns of the closing price of the S&P 500 security. The most common approach to obtain models for predicting the future values of a time series variable is to use the recent past values of the series as the input variables of the model. Thus our model will try to predict the h-days future returns of the closing price of S&P 500 based on the most recent values of these returns. This data preparation technique is usually known as time delay embedding.

The `its` package includes several functions that are very useful for constructing this type of lagged data sets. For instance, this package includes the function `lagdistIts()`, which can be used to create a matrix with a time delay embedding of a time series. The following example illustrates how to use this function:

```
> lagdistIts(sp500[1:10,'Close'],1,3)
           Close lag 1 Close lag 2 Close lag 3
1970-01-02          NA          NA          NA
1970-01-05       93.00          NA          NA
1970-01-06       93.46       93.00          NA
1970-01-07       92.82       93.46       93.00
1970-01-08       92.63       92.82       93.46
1970-01-09       92.68       92.63       92.82
1970-01-12       92.40       92.68       92.63
1970-01-13       91.70       92.40       92.68
1970-01-14       91.92       91.70       92.40
1970-01-15       91.65       91.92       91.70
```

As we can observe from this example we can use the second and third parameters of this function to set the range of sizes (or dimensions) of the embed.

We present below another example of its use together with other very useful function of the `its` package, the function `union()`, which can used to join two "its" objects,

```
> union(sp500[1:10,'Close'],lagdistIts(sp500[1:10,'Close'],1,3))
           Close Close lag 1 Close lag 2 Close lag 3
1970-01-02 93.00          NA          NA          NA
1970-01-05 93.46       93.00          NA          NA
1970-01-06 92.82       93.46       93.00          NA
```

[7]Try for instance `diff(c(4,6,2,4))`.

```
1970-01-07 92.63       92.82       93.46       93.00
1970-01-08 92.68       92.63       92.82       93.46
1970-01-09 92.40       92.68       92.63       92.82
1970-01-12 91.70       92.40       92.68       92.63
1970-01-13 91.92       91.70       92.40       92.68
1970-01-14 91.65       91.92       91.70       92.40
1970-01-15 91.68       91.65       91.92       91.70
```

The function `lagIts()` calculates the lagged values of a vector. The function `lagdistIts()` that we have seen before, calls this function iteratively for a range of lags. We can use `lagIts()` with negative lags to get the values lagged into the future, which is useful to construct variables with the future values of a time series. Let us see an example of this,

```
> union(lagIts(sp500[1:10,'Close'],-1),
+       union(sp500[1:10,'Close'],lagdistIts(sp500[1:10,'Close'],1,3)))
           Close lag -1 Close Close lag 1 Close lag 2 Close lag 3
1970-01-02        93.46 93.00          NA          NA          NA
1970-01-05        92.82 93.46       93.00          NA          NA
1970-01-06        92.63 92.82       93.46       93.00          NA
1970-01-07        92.68 92.63       92.82       93.46       93.00
1970-01-08        92.40 92.68       92.63       92.82       93.46
1970-01-09        91.70 92.40       92.68       92.63       92.82
1970-01-12        91.92 91.70       92.40       92.68       92.63
1970-01-13        91.65 91.92       91.70       92.40       92.68
1970-01-14        91.68 91.65       91.92       91.70       92.40
1970-01-15           NA 91.68       91.65       91.92       91.70
```

Putting all this together we can write down a function to create a data set with different lagged values of a time series,

```
> embeded.dataset <- function(data,quote='Close',fret.lag=1,ret.lags=1:3,emb=3) {
+   tmp <- lagIts(h.returns(data[,quote],fret.lag),-fret.lag)
+   for(i in ret.lags)
+     tmp <- union(tmp,lagdistIts(h.returns(data[,quote],i),0,emb-1))
+   its(tmp[(max(ret.lags)+emb):(NROW(data)-fret.lag),],
+       dates=tmp@dates[(max(ret.lags)+emb):(NROW(data)-fret.lag)],
+       names=c('fr',paste(rep(paste('pr',ret.lags,sep=''),each=emb),
+                          '.t',0:(emb-1),sep="")))
+ }
> sp500.data <- embeded.dataset(sp500,fret.lag=1,ret.lags=1:3,emb=2)
> colnames(sp500.data)
[1] "fr"     "pr1.t0" "pr1.t1" "pr2.t0" "pr2.t1" "pr3.t0" "pr3.t1"
> sp500.data[1:5,]
                    fr      pr1.t0      pr1.t1      pr2.t0     pr2.t1
1970-01-08 -0.30303030  0.05394907 -0.20511713 -0.15105740 -0.8960380
1970-01-09 -0.76335878 -0.30303030  0.05394907 -0.24891775 -0.1510574
1970-01-12  0.23933856 -0.76335878 -0.30303030 -1.06870229 -0.2489177
1970-01-13 -0.29459902  0.23933856 -0.76335878 -0.52219321 -1.0687023
1970-01-14  0.03272251 -0.29459902  0.23933856 -0.05455537 -0.5221932
               pr3.t0     pr3.t1
1970-01-08 -0.8416055 -0.3994386
1970-01-09 -0.4545455 -0.8416055
1970-01-12 -1.0141767 -0.4545455
1970-01-13 -0.8268059 -1.0141767
1970-01-14 -0.8183306 -0.8268059
```

This function receives an "its" object as first argument and produces another "its" containing several h-days returns of one of the quotes. By default the quote which is used is the closing price of each session. One of the columns of the returned object contains the future returns of the quote `fret.lag` days ahead. The other columns are controled by the parameters `ret.lags` and `emb`. The first of these parameters receives a vector of values for the *h*-days past returns. By default the function calculates the 1-, 2- and 3-days past returns of the selected quote. For each of these returns we can lag them several days back, *i.e.* we can calculate todays 1-day past returns, but we can also calculate this same value lagged one day back. The parameter `emb` controls how many lagged values are calculated for each of the *h*-days returns. In the example call we create an "its" object with the 1-day ahead returns (column "fr"), the 1-day back returns for two lags ("pr1.t0" for the value of today, and "pr1.t1" for yesterday's 1-day returns), and the same values for 2- and 3-days past returns.

In order to avoid the existence of rows with NA values in the returned "its" (which could cause difficulties with some modelling techniques), we have eliminated some of the first and last sessions in the resulting "its". The number of removed sessions depends on the parameters of the function.

Our data set includes the previous values of the *h*-days returns as the unique information which will be used to obtain a model to predict the next value of this time series. When creating a data set for obtaining a predictive model the obvious question is: how many days back should I consider? The answer is that we should use the previous returns that we believe may influence the value of the target variable, *i.e.* the returns in the future. This problem is usually known as feature selection, and its main goal is to search for the best set of predictor variables for obtaining a predictive model. The answer is not an easy one and it usually involves trying several alternatives and estimate their predictive power through some error estimation strategy.

Autocorrelation and autocorrelograms

When we are using modelling techniques assuming some linearity relation between the target variable and the predictors, like many "classical" time series models, we can use the notion of *autocorrelation* to help in answering the question of the number of lagged values we should use. This statistical notion captures how the values of a time series variable are correlated to the values on the previous time steps (different time *lags*). The function `acf()` computes and by default plots the autocorrelation of a variable along a set of time lags,

```
> acf(sp500.data[,'fr'],main='',ylim=c(-0.1,0.1))
```

Figure 3.2 shows the autocorrelation plot of the variable 'fr'. The dotted lines represent a 95% confidence threshold on the significance of the autocorrelation values. We have limited the Y-axis to eliminate the scaling effects of an extremely large correlation value for the lag 0.

As we can see there are few significant autocorrelation values[8], which does not provide good indications on the possibility of predicting the variable 'fr' using only the lagged returns. However, we should recall that these autocorrelations refer to linear correlations and, as we will not use linear models for this case study, we may still get reasonable results using only this information.

The interpretation of the autocorrelogram of Figure 3.2 provides important information (e.g. Chatfield, 1989, sec. 2.7.2) regarding which variables to use in our modeling task. Still, we should mention that given the linearity assumptions behind these correlations, this information is mainly useful if we are using linear models.

[8]The ones above (below) the dotted lines.

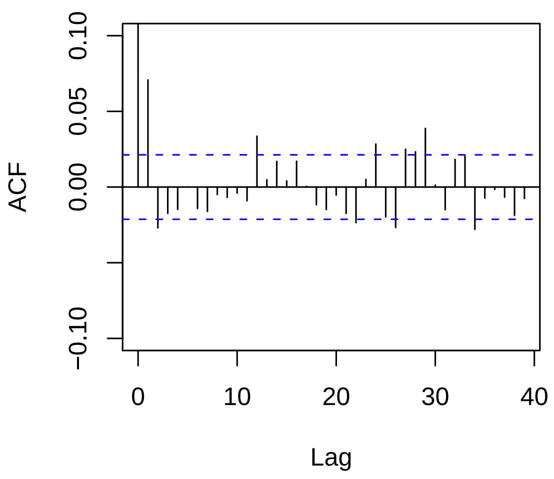

Figure 3.2: The autocorrelation of the 'fr' returns variable.

The data set we have obtained uses as predictor variables the previous values of the target variable. Can we add some extra information that could help in predicting the future returns? What kind of information and how to obtain it?

Given that we are predicting the future returns of an index one could try to add further economic indicators that we have reasons to belive that may influence the value of the index. Moreover, we could even add some fundamental data concerning the companies forming the index. However, this would complicate our task because we would probably end-up with information with different sampling frequencies and of rather diverse sources. For this case study we will assume that we only have access to the quotes data load at the start of the chapter. Using this information we can try to build some extra predictors by trying to capture some dynamic properties of the price time series that could be useful in predicting their future behavior. One example of such type of variables is what is usually known as technical indicators within the financial research are. These are values calculated solely on the basis of the prices that can capture some specific characteristic of the prices time series, like for instance their volatility, their momentum, etc. There is a large number of technical indicators. that can be used to enrich our set of variables. We will select a few just for illustrative purposes. Notice that most of these indicators are used by traders to generate trading signals. That will not be our approach. Instead we use them as input values for model construction. This means that they will be among a set of predictor variables that can be used by the model to reach a prediction.

Technical indicators

Moving averages are among the simplest technical indicators. A moving average is an indicator that shows the average value of a security's price over a period of time. The most popular method of using this indicator is to compare the moving average of the security's price with the security's price itself. A

Moving averages

buy signal is generated when the security's price rises above its moving average and a sell signal is generated when the security's price falls below this moving average.

A moving average can be easily obtained with the following function,

```
> ma <- function(x,size=1) {
+   require('ts')
+   its(as.numeric(filter(x@.Data,rep(1/size,size),sides=1)),
+       dates=x@dates
+       )
+ }
```

As the function needs the `filter()` function which is available by the package `ts`, we need to ensure that this package is loaded when the user calls our function. This is accomplished through the function `require()` which loads it if it is not loaded yet. The `filter()` function provides means to obtain several types of filters for time series, among which are moving averages. You may obtain the details at the help of the function.

If we want to obtain a moving average of the last 10 days for the closing prices of our index, we can type the following command:

```
> ma10.sp <- ma(sp500[,'Close'],size=10)
```

We can get a better feel for the use of moving averages as technical indicators by plotting a few of them with different time lags. An example of such plot is given in Figure 3.3, which shows the closing prices of S&P 500 together with two different moving averages for the first 1000 stock sessions. The code to generate the figure is the following,

```
> plot(union(sp500[1:1000,'Close'],
+            union(ma(sp500[1:1000,'Close'],50),
+                  ma(sp500[1:1000,'Close'],100))),
+      main='',type='l',ylab='Value',xlab='')
> legend(1.18, 120, c("Close", "MA(50)", "MA(100)"), col = 1:3, lty=1)
```

As it can be seen from Figure 3.3 the larger the moving average window the smoother the resulting curve.

Please remark how R has plotted an "its" object formed by several time series (the close prices and two moving averages). Due to its object-oriented features, R has used a special purpose (though in a transparentt way for the user) function for plotting "its" objects. This function among other things chooses proper labels for the X-axis as you may notice in the figure.

As mentioned above we can compare the moving average line and the closing prices to obtain trading signals. Instead of looking at the graph, we can build a function to obtain the signals,

```
> ma.indicator <- function(prices,size=20) {
+   its(c(NA,diff(sign(prices@.Data-ma(prices,size)@.Data))),dates=prices@dates)
+ }
```

Positive values of this indicator are usually interpreted by traders as buy opportunities, while negative values are interpreted as sell signals.

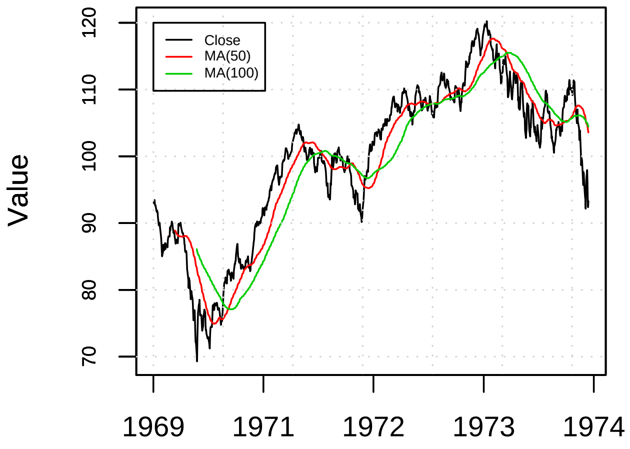

Figure 3.3: Two moving averages of the closing prices of S&P 500.

The RSI indicator

Another popular indicator is the Relative Strength Index (RSI). This is a price-following oscillator that ranges between 0 and 100. It is calculated as,

$$\text{RSI} = 100 - \left(\frac{100}{1 + \left(\frac{U}{D}\right)} \right) \tag{3.6}$$

where, U is the percentage of positive returns in a certain time lag, and D is the percentage of negative returns during that period.

This indicator can be implemented by the following code:

```
> moving.function <- function(x, lag, FUN, ...) {
+   require('ts')
+   FUN <- match.fun(FUN)
+   c(rep(NA,lag),apply(embed(x,lag+1),1,FUN,...))
+ }
> rsi.aux <- function(diffs,lag) {
+   u <- length(which(diffs > 0))/lag
+   d <- length(which(diffs < 0))/lag
+   ifelse(d==0,100,100-(100/(1 + u/d)))
+ }
> rsi <- function(x,lag=20) {
+   d <- c(0,diff(x))
+   moving.function(d,lag,rsi.aux,lag)
+ }
```

The RSI-based trading rule is usually the following: As the RSI indicator crosses the value 70 coming from above, generated a sell signal; when the indicator crosses the value 30 coming from below generate a sell signal. This rule is implemented by the following function,

```
> rsi.indicator <- function(x,lag=20) {
+   r <- rsi(x,lag)
+   d <- diff(ifelse(r > 70,3,ifelse(r<30,2,1)))
+   f <- cut(c(rep(0,lag),d[!is.na(d)]),breaks=c(-3,-2,-1,10),
+            labels=c('sell','buy','hold'),right=T)
+   factor(f,levels=c('sell','hold','buy'))
+ }
```

Finally, we will also use the Chaikin oscillator to generate more information for our predictive models. This indicator includes information regarding the volume of transactions besides the stock prices. It can be calculated with the following two functions,

```
> ad.line <- function(df) {
+   df$Volume*((df$Close-df$Low) - (df$High-df$Close))/(df$High-df$Low)
+ }
> chaikin.oscillator <- function(df,short=3,long=10) {
+   ad <- ad.line(df)
+   ewma(ad,lambda=1/(short+1))-ewma(ad,lambda=1/(long+1))
+ }
```

Note that to obtain the values for this indicator you need to provide the overall data frame since the indicator uses several quotes of the stock. The following is an example call of the function,

```
> chaikin <- chaikin.oscillator(ibm[1:first12y,])
```

[Describe the signal generation from this oscillator]

3.4 Time series prediction models

Given an "its" like the one obtained previously, we can use a multiple regression method to build a model that predicts the value of the future returns given the past observed returns. Before obtaining such models let us study some properties of this data set.

Figure 3.4 shows a box-plot of the 1-day returns of the closing price, obtained with,

```
> boxplot(sp500.data[,'fr']@.Data,boxwex=0.15,ylab='1-day ahead returns of Closing price'
> rug(jitter(sp500.data[,'fr']@.Data),side=2)
```

This graph reveals a symmetric distribution with very large tails (*i.e.* a clear presence of rare events). From the 8479 rows of the `sp500.data` object, only 191 have future 1-day returns larger than 2.5% in absolute values.[9] This would not be a problem if they were not the situations that we are really interested in being accurate! One of the more interesting features of this domain is that

[9]Practice your R knowledge by trying to obtain this number.

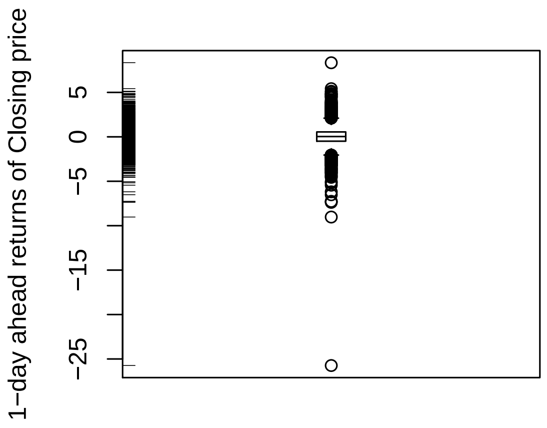

Figure 3.4: The box plot of the 1-day returns.

most of the times the market varies in a way which is not interesting for trading because the changes are too small to compensate for the trading costs. Thus, the majority of the data we have is not interesting from a trading perspective. The interesting cases are exactly those with large variations. We want our models to be extremely accurate at predicting these large movements. The other movements are more or less irrelevant. The characteristics of this domain are problematic for most modeling tools. Standard regression methods try to minimize some error estimate, for instance, the mean squared error. While there is nothing wrong with this, in this domain we would be willing to give up some accuracy on small movements in exchange for correct predictions of large market movements. Moreover, we are not particularly interested in being accurate at the precise value of a large change, as long as we predict a value that leads us to the correct trading decision (buy, sell or hold).

Neural Networks

The first model we will try with the goal of predicting the 1-day ahead returns of closing prices will be a neural network. Neural networks are among the most frequently used models in financial predictions experiments (Deboeck, 1994), because of their ability to deal with highly non-linear problems. The package `nnet` implements feed forward neural nets in R. This type of neural networks are among the most frequently.

A neural network is formed by a network of computing units (the neurons) linked to each other. Each of these connections has an associated weight. Constructing a neural network consists of using an algorithm to find the weights of the connections between the neurons. A neural network has its neurons organized in layers. The first layer contains the input neurons of the network. The cases of the problem we are addressing are presented to the network through these input neurons. The final layer contains the predictions of the neural network for the case presented at its input neurons. In between we have one or more "hidden" layers of neurons. The weight updating algorithms, like for instance the back-propagation method, try to obtain the connection weights that optimize a certain error criterion, that is the weights which ensure that the network output is in accordance to the cases presented to the model.

We will now illustrate how to obtain a neural network in R, and also how to use these models to obtain predictions. To achieve these goals we will split our data set in two time windows, one used to obtain the neural network and the other to evaluate it, *i.e.* to test how well the model predicts the target variable values. In Section **??** we will address more thoroughly the issue of evaluating time series models. For now let us simply split our data in two time windows, one consisting of the first 20 years of quotes data, and the other with the remaining 12 years,

```
> ibm.train <- ibm.data[ibm.data$Date < '1990-01-01',]
> ibm.test <- ibm.data[ibm.data$Date > '1989-12-31',]
```

Neural networks usually obtain better results with normalized data. Still, in this particular application all the variables have the same scale and they have roughly a normal distribution with zero mean.[10] So we will avoid this normalization step.

To obtain a neural network to predict the 1-day ahead future returns we can use the function `nnet()` as follows,

```
> library(nnet)
> nn <- nnet(r1.f1 ~ .,data=ibm.train[,-ncol(ibm.train)],
+             linout=T,size=10,decay=0.01,maxit=1000)
```

This function call will build a neural network with a single hidden layer, in this case formed by 10 hidden units(the parameter `size`). Moreover, the weights will be learned with a weight updating rate of 0.01 (the parameter `decay`). The parameter `linout` indicates that the target variable is continuous. The `maxit` parameter sets the maximum number of iterations of the weight convergence algorithm. Notice that we have removed the last column of the `ibm.data` data frame which contains the *Date* information. This type of information is useless for model construction as its value is different for all cases. Still, we could eventually consider to use part of this information, like the month number or the day of the week, to try to capture eventual seasonal effects.

The `nnet()` function uses the back-propagation algorithm as the basis of an iterative process of updating the weights of the neural network, up to a maximum of `maxit` cycles. This iterative process may take a long time to compute for large datasets.

The above function call creates a neural network with 24 input units (the number of predictor variables of this problem) connected to 10 hidden units, which will then be linked to a single output unit. This leads to a total number of 261 connections. We can see the final weights of these connections by issuing,

```
> summary(nn)
```

[10]Check this doing `summary(ibm.data)`.

This neural net can be used to make predictions for our test period,

```
> nn.preds <- predict(nn,ibm.test)
```

The following code gives as a graph with the predictions plotted against the true values (*c.f.* Figure 3.5),[11]

```
>  plot(ibm.test[,1],nn.preds,ylim=c(-0.01,0.01),
+       main='Neural Net Results',xlab='True',ylab='NN predictions')
> abline(h=0,v=0); abline(0,1,lty=2)
```

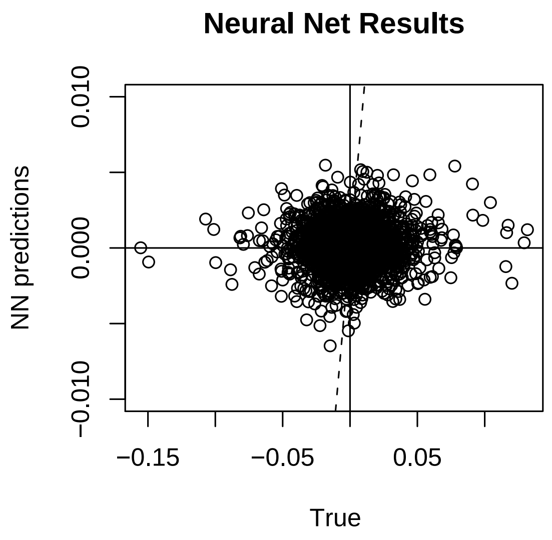

Figure 3.5: The 1-day ahead returns predictions of the Neural Network.

In Section ?? we will address the issue of evaluating time series models so that we may have an idea of the relative value of these neural network predictions. Still, ideally all dots in Figure 3.5 should be on the dotted line (representing zero error), which is clearly not the case.

Further readings on neural networks

The book by Rojas (1996) is a reasonable general reference on Neural Networks. For more financially-oriented readings the book by Zirilli (1997) is a good and easy reading reference. The collection of papers entitled "Artificial neural networks forecasting time series" (Rogers and Vemuri, 1994) is another example of a good source of references. Part I of the book by Deboeck (1994) provides several chapters devoted to the application of neural networks to trading. The work of McCulloch and Pitts (1943) presents the first model of an artificial neuron. This work was generalized by Ronsenblatt (1958) and Minsky and Papert (1969). The back-propagation algorithm, the most frequently used weight updating method, although usually attributed to Rumelhart et al. (1986), was, according to Rojas (1996), invented by Werbos (1974, 1996).

[11] A word of warning concerning the results you obtain if you are trying this same code. The `nnet()` function has a random effect in its starting point (the initial weights) which does not ensure that we will always obtain the same network even if using the same data. This means that your results or graphs may be slightly different from the ones presented here.

Projection pursuit regression

Let us try another multiple regression technique, namely projection pursuit regression (Friedman, 1981).

A projection pursuit model can be seen as a kind of additive model (Hastie and Tibshirani, 1990) where each additive term is a linear combination of the original variables of the problem. This means that a projection pursuit model is an addition of terms formed by linear combinations of the original variables.

Projection pursuit regression is implemented in package `modreg`. We can obtain such a model with the following code,

```
> library(modreg)
> pp <- ppr(r1.f1 ~ ., data=ibm.train[,-ncol(ibm.train)],nterms=5)
> pp.preds <- predict(pp,ibm.test)
```

After loading the respective library, we can obtain a projection pursuit model using the function `ppr()`. This function has several parameters[12]. The parameter `nterms` sets the number of linear combinations that will be included in the projection pursuit model. You may also use the parameter `max.terms` to set a maximum number of terms that will be used. After adding this number of maximum terms the "worse" terms will be iteratively removed until a model with `nterms` is reached.

As usual with all models in R, we can use the function `predict()` to obtain the predictions of our model.

We can have a look at the coefficients of the linear combinations that form the terms of the additive model by issuing,

```
> summary(pp)
Call:
ppr.formula(formula = r1.f1 ~ ., data = ibm.train[, -ncol(ibm.train)],
    nterms = 5)

Goodness of fit:
5 terms
      0

Projection direction vectors:
       term 1       term 2       term 3       term 4       term 5
r1.t0  -0.274593836 -0.003047629  0.000000000  0.000000000  0.000000000
r1.t1  -0.232145592  0.303103072  0.000000000  0.000000000  0.000000000
r1.t2  -0.125882592  0.042927734  0.000000000  0.000000000  0.000000000
r1.t3  -0.277068070 -0.015256559  0.000000000  0.000000000  0.000000000
r1.t4   0.237179308  0.137114309  0.000000000  0.000000000  0.000000000
r1.t5   0.100427485 -0.119630099  0.000000000  0.000000000  0.000000000
r1.t6   0.193712788 -0.255578319  0.000000000  0.000000000  0.000000000
...
...
r1.t20  0.008716963 -0.014020849  0.000000000  0.000000000  0.000000000
r1.t21 -0.287156429  0.138665388  0.000000000  0.000000000  0.000000000
r1.t22 -0.196584111  0.181707111  0.000000000  0.000000000  0.000000000
r1.t23  0.271527937  0.224461896  0.000000000  0.000000000  0.000000000

Coefficients of ridge terms:
     term 1      term 2      term 3      term 4      term 5
0.001805114 0.001025505 0.000000000 0.000000000 0.000000000
```

[12]You may want to try '? ppr' to read more about them.

As we can see from the result of the `summary()` function, the `ppr()` routine obtained a model consisting only of two terms. These terms are linear combinations of the original variables as we can see from this output. This means that the model we have obtained can be described by the equation,

$$\begin{aligned} r_1.f_1 &= 0.001805 \times (-0.2746 \times r_1.t_0 - 0.23215 \times r_1.t_1 \ldots) + \\ &\quad +0.001026 \times (-0.00305 \times r_1.t_0 + 0.30310 \times r_1.t_1 \ldots) \end{aligned}$$

Multivariate adaptive regression splines (MARS)

Finally, we will try multivariate adaptive regression splines (Friedman, 1991). This is another example of an additive regression model (Hastie and Tibshirani, 1990). This model is a bit complex and it can be seen as the following sum of sums of functions,

$$mars\,(\mathbf{x}) = c_0 + \sum f_m\,(X_m) + \sum f_{m,n}\,(X_{m,n}) + \sum f_{m,n,o}\,(X_{m,n,o}) + \ldots \quad (3.7)$$

where the first sum is over a set of basis functions that involve a single predictor variable, the second sum is over basis function involving two variables, and so on.

MARS models are implemented in R in package `mda`, which basically is a re-implementation of the original MARS code done by Trevor Hastie and Robert Tibshirani. The following code, obtains a MARS model and evaluates it on the test period,

```
> library(mda)
> m <- mars(ibm.train[,2:10],ibm.train[,1])
> m.preds <- predict(m,ibm.test[,2:10])
```

Unfortunately, the `mars` function does not use the standard R formula syntax. In effect, the first argument of this function is the data frame containing the input variables, while the second contains the data referring to the respective target variable values.[13]

We can also apply the `summary()` function to the model produced by the `mars()` function. However, the information returned is not very complete. In the help for the `mars()` function there is further information on inspecting these models.

Further readings on multivariate adaptive regression splines

The definitive reference on MARS is the original journal article by Friedman (1991). This is a very well-written article providing all details concerning the motivation for the development of MARS as well as the techniques used in the system. The article also includes a quite interesting discussion section by other scientists that provides other views of this work.

[13]This is known as the matrix syntax and most model functions also accept this syntax apart from the more convenient formula syntax.

3.5 Evaluating time series models

Due to the time dependence between observations the evaluation procedures for time series prediction models are different from standard methods. The latter are usually based on resampling strategies (for instance bootstrap or cross validation), which work by obtaining random samples from the original unordered data. The use of these methodologies with time series could lead to undesirable situations like using future observations of the variable for training purposes[14] , and evaluating models with past data. In order to avoid these problems we usually split the available time series data into time windows, obtaining the models with past data and testing it on subsequent time slices.

The main purpose of any evaluation strategy is to obtain a reliable value of the expected predictive accuracy of a model. If our estimate is reliable we can be reasonably confident that the predictive performance of our model will not deviate a lot from our estimate when we apply the model to new data from the same domain. In Section 2.7 (page 66) we have seen that the key issue to obtain reliable estimates is to evaluate the models on a sample independent from the data used to obtain them.

In our case study we have quotes data from 1970 till mid 2002. We will set a time t as the start of the testing period. Data before t will be used to obtain the prediction models, while data occurring after t will be used only to test them.

Within the scenario described above there are some alternatives one may consider. The first is to obtain a single model using the data before time t and test it on each case occurring after t. Alternatively, we can use windowing strategies. The first windowing strategy we describe is the *growing window*,

Windowing strategies

```
1. Given a series $R_h(1), R_h(2), \ldots, R_h(n)$ and a time $t$ $(< n)$
2. Obtain a prediction model with training data $R_h(1), R_h(2), \ldots, R_h(t-1)$
3. REPEAT
4.    Obtain a prediction for observation $R_h(t)$
5.    Record the prediction error
6.    Add $R_h(t)$ to the training data
7.    Obtain a new model with the new training set
8.    Let $t = t + 1$
9. UNTIL $t = n$
```

An alternative is to use a *sliding window*,

```
1. Given a series $R_h(1), R_h(2), \ldots, R_h(n)$, a time $t$ and a window size $w$
2. Obtain a prediction model with training data $R_h(t-w-1), \ldots, R_h(t-1)$
3. REPEAT
```

[14] The data used to obtain the models.

```
4.    Obtain a prediction for $R_h(t)$
5.    Record the prediction error
6.    Add $R_h(t)$ to the training data and remove $R_h(t-w-1)$
7.    Obtain a new prediction model with the new training data
8.    Let $t = t+1$
9. UNTIL $t = n$
```

The windowing approaches seem more reasonable as they incorporate new information into the model (by modifying it) as time goes by. Still, they require obtaining a large number of models.[15] Although intermediate alternatives exist, we will adopt the single model strategy in this case study to simplify our experiments. Nevertheless, better results could eventually be obtained with windowing strategies, thus for real applications these should not be discarded.

Our experiments with this data set consist of obtaining a prediction model with the data up to 1-Jan-1990, and testing it with the remaining data. This means that we will obtain our model using around 20 years of quotes data, and test it on the next 12 years. This large test set (around 3100 sessions) ensures a reasonable degree of statistical significance for our accuracy estimates. Moreover, this 12 years period includes a large variety of market conditions, which increases the confidence of the estimated accuracy.

Having decided on a experimental setup, we still need to choose which error measures we will use to evaluate our models. In Section 2.6 (page 53) we have seen a few examples of error measures that can be used in multiple regression experiments, like for instance the mean squared error. We have also mentioned the advantages of relative error measures like the normalized mean squared error, which provides an indication of the relative "value" of the models. Within time series analysis it is common to use (e.g. Gershenfeld and Weigend, 1994) the Theil U coefficient(Theil, 1966) with this purpose. This statistic is basically the normalized mean squared error adapted for time series problems. In these domains we use the following simple model to normalize our model predictions,

The Theil coefficient

$$\hat{Y}(t+h) = Y(t) \tag{3.8}$$

where $\hat{Y}(t+h)$ represents the predicted value of Y for time $t+h$.

In our problem this corresponds to predicting that the next value of the h-days ahead returns will be the same as the value observed today. This leads to the following definition of the Theil coefficient,

$$U = \frac{\sqrt{\sum_{t=1}^{N_{test}} \left(R_h(t+h) - \hat{R}_h(t+h)\right)^2}}{\sqrt{\sum_{t=1}^{N_{test}} \left(R_h(t+h) - R_h(t)\right)^2}} \tag{3.9}$$

where, $\hat{R}_h(t+h)$ is the prediction of our model for time $t+h$, *i.e.* the h-days ahead returns. As with the normalized mean squared error our goal is to have models with U values significantly lower than 1.

[15]This may not be completely true if the techniques we are using are "incremental", in the sense that we can perform slight modifications to the model as new data becomes available, instead of building a new model from scratch.

Let us see an example of calculating the Theil coefficient using the predictions of our neural network (*c.f.* Section 3.4),

```
> naive.returns <- c(ibm.train[nrow(ibm.train),1],
+                    ibm.test[1:(nrow(ibm.test)-1),1])
> theil <- function(preds,naive,true) {
+   sqrt(sum((true-preds)^2))/sqrt(sum((true-naive)^2))
+ }
> theil(nn.preds,naive.returns,ibm.test[,1])
[1] 0.7006378
```

The first instruction obtains the simplest model predictions. We then define a function to calculate the Theil coefficient according to the definition in Equation 3.9. Finally, we call this function to obtain the Theil coefficient of the neural network predictions.

Evaluating financial time series

When evaluating financial time series predictions several other statistics are of interest to assert the model quality (Hellstrom and Holmstrom, 1998). Namely, we are usually interested in looking at the accuracy in predicting the direction of the market, *i.e.* whether the next day returns are positive or negative. We will use the hit rate at predicting the non-zero returns, and also the positive and negative hit rates (which assess the accuracy of the model at predicting the raises and falls of the market). The following R code implements these statistics,

```
> hit.rate <- function(preds,true) {
+   length(which(preds*true > 0))/length(which(true != 0))
+ }
> positive.hit.rate <- function(preds,true) {
+   length(which(preds > 0 & true > 0))/length(which(true > 0))
+ }
> negative.hit.rate <- function(preds,true) {
+   length(which(preds < 0 & true < 0))/length(which(true < 0))
+ }
> hit.rate(nn.preds,ibm.test[,1])
[1] 0.5051172
> positive.hit.rate(nn.preds,ibm.test[,1])
[1] 0.5753247
> negative.hit.rate(nn.preds,ibm.test[,1])
[1] 0.432505
```

We can join all these evaluating statistics into a single structure through the following function,

```
> timeseries.eval <- function(preds,naive,true) {
+    th <- sqrt(sum((true-preds)^2))/sqrt(sum((true-naive)^2))
+    hr <- length(which(preds*true > 0))/length(which(true != 0))
+    pr <- length(which(preds > 0 & true > 0))/length(which(true > 0))
+    nr <- length(which(preds < 0 & true < 0))/length(which(true < 0))
+    n.sigs <- length(which(preds != 0))
+    perc.up <- length(which(preds > 0)) / n.sigs
+    perc.down <- length(which(preds < 0)) / n.sigs
+    round(data.frame(N=length(true),Theil=th,HitRate=hr,PosRate=pr,NegRate=nr,
+                     Perc.Up=perc.up,Perc.Down=perc.down),
+          3)
+  }
> timeseries.eval(nn.preds,naive.returns,ibm.test[,1])
     N Theil HitRate PosRate NegRate Perc.Up Perc.Down
1 3122 0.701   0.505   0.575   0.433   0.573     0.427
```

This code uses the function `round()` to set the number of significant digits used to show the results. `Perc.Up` and `Perc.Down` are the percentage of times the model predicts positive or negative returns, respectively.

In the context of financial time series, it is interesting to consider annualized results. Our testing period goes over 12 years. We want to obtain not only the results shown above, but also the same statistics on each of these years. The following code achieves this,

```
> annualized.timeseries.eval <- function(preds,naive,test) {
+   res <- timeseries.eval(preds,naive,test[,1])
+
+   years <- unique(substr(test[,'Date'],1,4))
+   for(y in years) {
+     idx <- which(substr(test[,'Date'],1,4)==y)
+     res <- rbind(res,timeseries.eval(preds[idx],naive[idx],test[idx,1]))
+   }
+
+   row.names(res) <- c('avg',years)
+   res
+ }
> annualized.timeseries.eval(nn.preds,naive.returns,ibm.test)
        N Theil HitRate PosRate NegRate Perc.Up Perc.Down
avg  3122 0.701   0.505   0.575   0.433   0.573     0.427
1990  253 0.737   0.463   0.550   0.363   0.597     0.403
1991  253 0.713   0.445   0.561   0.344   0.605     0.395
1992  254 0.758   0.559   0.661   0.469   0.587     0.413
1993  253 0.713   0.494   0.535   0.454   0.549     0.451
1994  252 0.674   0.502   0.569   0.437   0.583     0.417
1995  252 0.705   0.485   0.575   0.384   0.583     0.417
1996  254 0.714   0.565   0.642   0.468   0.591     0.409
1997  253 0.712   0.480   0.549   0.402   0.577     0.423
1998  252 0.715   0.500   0.575   0.414   0.579     0.421
1999  252 0.715   0.518   0.540   0.496   0.524     0.476
2000  252 0.642   0.530   0.605   0.462   0.575     0.425
2001  248 0.716   0.500   0.526   0.469   0.528     0.472
2002   94 0.725   0.553   0.628   0.490   0.564     0.436
```

This function shows us that the performance of the neural network is not constant over the 12 years. Moreover, with this table of results we can clearly observe the difference between accuracy from a regression perspective and the more trading oriented indicators. We have cases where a better score in terms of Theil coefficient leads to a worse score in terms of hit rate. For instance, the best score in terms of the Theil coefficient is on 2000, where the hit rate is 53%. The worst Theil score, is obtained on 1992, but the hit rate is higher (55.9%).

3.5.1 Model selection

In the previous sections we have obtained several regression models that can be used to predict the returns of the closing price of IBM, 1 day ahead. We have also shown some measures that can be used to evaluate these models. The purpose of this section is to answer the question:

What model should I use?

This is usually known as the *model selection* problem. The candidate models for selection need not be completely different models. They could be models obtained with the same data mining technique but using different parameter settings. For instance, the neural network we have obtained used at least two parameters: The number of hidden nodes; and the weight decay. Different settings lead to different neural networks. Deciding which are the best settings is also a model selection problem.

To address the model selection problem we need to answer two questions:

- Which statistics should be used to evaluate and compare the candidates?
- What experimental setup should be used to ensure the reliability of our estimates?

In Section **??** we have already partially answered these questions. We have described several statistics that can be used to evaluate financial times series models. Moreover, we have decided upon an experimental setup to evaluate our models. This setup consisted of splitting our data in two time windows, one for obtaining the models and the other for testing them. As mentioned before, model selection should not be carried out using the test data as this would bias our choices. Instead, we will divide our training period in two time windows: One for obtaining the candidate model variants; and the other for selecting the "best" alternative. Once this selection is done, we can obtain this "best" model using all training period data so as not to "loose" the data of the selection period.

In summary, we will divide our 20 years of training data in two parts. The first part will consist of the initial 12 years (1970-1981) and will be used to obtain the model variants. The remaining 8 years (1982-1989) will be used to select the "best" model according to this selection period. Finally, this "best" model will be obtained again using the full 20 years, leading to the result of our data mining process.

Let us start by obtaining these time slices,

```
> first12y <- nrow(ibm.data[ibm.data$Date < '1982-01-01',])
> train <- ibm.train[1:first12y,]
> select <- ibm.train[(first12y+1):nrow(ibm.train),]
```

We will first illustrate this model selection strategy by using it to find the best settings for a particular data mining technique. Namely, we will consider the problem of selecting the "best" neural network setup.

Tuning the neural network

The neural network we have previously obtained used certain parameter settings. Most data mining tools have some kind of parameters that you may tune to achieve better results. Some tools are easier to tune than others. There are even tools that perform this tuning by themselves.[16]

We will try several setups for the `size` and `decay` parameters of our neural nets. The following code tries 12 different combinations of size and decay, storing the mean squared error and hit rate results on a data frame,

```
> res <- expand.grid(Size=c(5,10,15,20),
+                    Decay=c(0.01,0.05,0.1),
+                    MSE=0,
+                    Hit.Rate=0)
> for(i in 1:12) {
```

[16]As is the case for regression trees in R, which use an internal cross validation process to "guess" the best tree size.

```
+   nn <- nnet(r1.f1 ~ .,data=train[,-ncol(train)],linout=T,
+              size=res[i,'Size'],decay=res[i,'Decay'],maxit=1000)
+   nn.preds <- predict(nn,select)
+   res[i,'MSE'] <- mean((nn.preds-select[,1])^2)
+   res[i,'Hit.Rate'] <- hit.rate(nn.preds,select[,1])
+ }
```

This experiment may take a while to run on your computer since training of neural networks is usually computationally intensive.

A few comments on the code given above. The function `expand.grid()` can be used to generate all combinations of different factors or vector values.[17] For each of the parameter variants we obtain the respective neural net and evaluate it on the model selection time period. We could have used the function `annualized.timeseries.eval()` to carry out this evaluation. However, due to the number of variants that we are trying this would over clutter the output. As such, we have decided to use two statistics representing the most important features evaluated by that function, namely the regression accuracy and the sign of the market accuracy. For the former we have used the mean squared error. We have not used the Theil coefficient because we are comparing the 12 alternatives and thus the comparison to a baseline predictor (as provided by this coefficient) is not so interesting.

In the end you may inspect the results by typing,

```
> res
   Size Decay          MSE  Hit.Rate
1     5  0.01 0.0002167111 0.5005149
2    10  0.01 0.0002167079 0.5025747
3    15  0.01 0.0002167094 0.5025747
4    20  0.01 0.0002167162 0.5036045
5     5  0.05 0.0002171317 0.5087539
6    10  0.05 0.0002171317 0.5087539
7    15  0.05 0.0002171318 0.5087539
8    20  0.05 0.0002171317 0.5087539
9     5  0.10 0.0002171317 0.5087539
10   10  0.10 0.0002171317 0.5087539
11   15  0.10 0.0002171317 0.5087539
12   20  0.10 0.0002171316 0.5087539
```

The results in terms of regression accuracy are quite similar. Still, we can observe that for the same decay the best results are usually obtained with a larger size. Due to the fact that larger values of the decay lead to an increase in the hit rate we may be inclined to select a network with 20 hidden units and a decay of 0.1 as the best setting for our data.

Selecting the best regression model

We will now illustrate this same selection process to find the "best" model from the three alternatives that we have considered. For the neural network we will use the settings that resulted from our tuning process. For the other models we will use the same parameter settings as before to simplify our experiments.[18]

The following code obtains the three models using the first 12 years data,

```
> nn <- nnet(r1.f1 ~ .,data=train[,-ncol(train)],linout=T,size=20,decay=0.1,maxit=1000)
> nn.preds <- predict(nn,select)
> pp <- ppr(r1.f1 ~ ., data=train[,-ncol(train)],nterms=5)
```

[17]You may want to try a few examples with this function to understand its behavior.
[18]Still, a similar parameter tuning process could have been carried out for these other models.

```
> pp.preds <- predict(pp,select)
> m <- mars(train[,2:20],train[,1])
> m.preds <- predict(m,select[,2:20])
> naive.returns <- c(train[first12y,1],select[1:(nrow(select)-1),1])
```

After obtaining the models we can compare their performance on the 8 years left for model selection,

```
> annualized.timeseries.eval(nn.preds,naive.returns,select)
        N Theil HitRate PosRate NegRate Perc.Up Perc.Down
avg  2022 0.684   0.509       1       0       1         0
1982  253 0.687   0.506       1       0       1         0
1983  253 0.666   0.551       1       0       1         0
1984  253 0.695   0.492       1       0       1         0
1985  252 0.760   0.543       1       0       1         0
1986  253 0.696   0.492       1       0       1         0
1987  253 0.671   0.528       1       0       1         0
1988  253 0.670   0.490       1       0       1         0
1989  252 0.699   0.467       1       0       1         0
> annualized.timeseries.eval(pp.preds,naive.returns,select)
        N Theil HitRate PosRate NegRate Perc.Up Perc.Down
avg  2022 0.713   0.504   0.490   0.519   0.488     0.512
1982  253 0.709   0.506   0.492   0.522   0.498     0.502
1983  253 0.687   0.519   0.500   0.541   0.490     0.510
1984  253 0.727   0.463   0.450   0.476   0.482     0.518
1985  252 0.780   0.490   0.444   0.545   0.452     0.548
1986  253 0.722   0.561   0.521   0.600   0.455     0.545
1987  253 0.703   0.524   0.562   0.483   0.545     0.455
1988  253 0.700   0.485   0.449   0.520   0.474     0.526
1989  252 0.748   0.484   0.500   0.469   0.508     0.492
> annualized.timeseries.eval(m.preds,naive.returns,select)
        N Theil HitRate PosRate NegRate Perc.Up Perc.Down
avg  2022 0.745   0.516   0.495   0.539   0.476     0.524
1982  253 0.687   0.545   0.517   0.574   0.482     0.518
1983  253 0.664   0.572   0.545   0.606   0.462     0.538
1984  253 0.699   0.504   0.533   0.476   0.522     0.478
1985  252 0.767   0.465   0.436   0.500   0.456     0.544
1986  253 0.709   0.492   0.455   0.528   0.462     0.538
1987  253 0.828   0.549   0.538   0.560   0.486     0.514
1988  253 0.705   0.519   0.483   0.553   0.470     0.530
1989  252 0.707   0.488   0.447   0.523   0.464     0.536
```

The neural network model is clearly the best in term of regression predictive accuracy. However, this regression accuracy is not transposed into a good score in terms of predicting the sign of the market change for the next day. Actually, the neural network always predicts positive returns thus leading to 100% accuracy on the positive movements of the market. Still, from the perspective of predicting the sign of the market changes the best performance is achieved by the MARS model. However, this is not the end in terms of model evaluation. As we will see in the next section if we translate the prediction of these models into trading actions we can get further information on the "value" of these predictions.

Further readings on time series analysis

A good general reference on time series analysis is the book by Chatfield (1989). Sauer et al. (1991) is a standard reference on time delay embeded. This technique is used for instance in many traditional time series modeling methods like the ARMA models (Box and Jenkins, 1976) and it is theoretically justified by the Takens theorem (Takens, 1981).

3.6 From predictions into trading actions

As mentioned at the beginning of this chapter our overall goal in this case study is to build a trading system based on the predictions of our data mining models. Thus our final evaluation criterion will be the trading results obtained by this system.

The first step to obtain such a system is to transform the predictions of our models into trading actions. There are three possible actions: buy, sell, or hold (do nothing). We will use the predictions of our models as the sole information for deciding which action to take. Given the type of data we are using to obtain our models it does not make sense to use an intra-day trading strategy. Thus, our decisions will be made at the end of each market session, and our eventual orders will be posted after this session but before the next day opening.

Ideally we would be using several predictive models with different forecasting scenarios (1 day ahead, 5 days ahead, etc.). This could give us a mid-term prediction of the evolution of the market and thus allow for more informed decisions concerning the best action at the end of each day. In our case study we will simplify this approach, and will use only our 1 day ahead predictions for generating the signals. Still, some decisions have to be made given a set of predictions of 1 day ahead returns. Namely, we have to decide when to buy or sell. The following R function is a simple approach that uses some thresholds on the predicted market variations to generate a vector of trading signals,

```
> buy.signals <- function(pred.ret,buy=0.05,sell=-0.05) {
+  sig <- ifelse(pred.ret < sell,'sell',
+                ifelse(pred.ret > buy,'buy','hold'))
+  factor(sig,levels=c('sell','hold','buy'))
+ }
```

With this simple function we can generate a set of trading signals from the predictions of any of the models obtained in Section ??.

3.6.1 Evaluating trading signals

Having a set of trading signals we can compare them to the effective market movements to check the performance of our actions when compared to the reality. The following is a small example of this comparison with the MARS predictions for the selection period,

```
> mars.actions <- buy.signals(m.preds,buy=0.01,sell=-0.01)
> market.moves <- buy.signals(select[,1],buy=0.01,sell=-0.01)
> table(mars.actions,market.moves)
            market.moves
mars.actions sell hold buy
        sell    7    3   7
        hold  377 1200 415
        buy     3    8   2
```

In this small example we have decided to generate a buy signal whenever our model predicts a 1-day ahead returns larger than 1%, while sell signals where generated for a decrease of 1%. We have used the function `table()` to obtain a contingency table that shows the results of our actions when compared to the effective market movements. We can see that from the 17 (=7+3+7) sell actions generated by our model, only 7 corresponded to an effective down move of the market.[19] On 3 occasions the market closed at the same price, while on 7 occasions the market even closed at a higher price, meaning that if we have posted the sell order we would be loosing money! Obviously these numbers only give us half of the picture, because it is also relevant to observe the amount of money corresponding to each correct (or incorrect) guess. This means that even if our hit rate regarding market movements is not very good we could be earning money. The following two functions provide us further information on the results of our trading signals,

```
> signals.eval <- function(pred.sig,true.sig,true.ret) {
+   t <- table(pred.sig,true.sig)
+   n.buy <- sum(t['buy',])
+   n.sell <- sum(t['sell',])
+   n.sign <- n.buy+n.sell
+   hit.buy <- round(100*t['buy','buy']/n.buy,2)
+   hit.sell <- round(100*t['sell','sell']/n.sell,2)
+   hit.rate <- round(100*(t['sell','sell']+t['buy','buy'])/n.sign,2)
+   ret.buy <- round(100*mean(as.vector(true.ret[which(pred.sig=='buy')])),4)
+   ret.sell <- round(100*mean(as.vector(true.ret[which(pred.sig=='sell')])),4)
+   data.frame(n.sess=sum(t),acc=hit.rate,acc.buy=hit.buy,acc.sell=hit.sell,
+              n.buy=n.buy,n.sell=n.sell,ret.buy=ret.buy,ret.sell=ret.sell)
+ }
> annualized.signals.results <- function(pred.sig,test) {
+   true.signals <- buy.signals(test[,1],buy=0,sell=0)
+   res <- signals.eval(pred.sig,true.signals,test[,1])
+   years <- unique(substr(test[,'Date'],1,4))
+   for(y in years) {
+     idx <- which(substr(test[,'Date'],1,4)==y)
+     res <- rbind(res,signals.eval(pred.sig[idx],true.signals[idx],test[idx,1]))
+   }
+   row.names(res) <- c('avg',years)
+   res
+ }
> annualized.signals.results(mars.actions,select)
     n.sess   acc acc.buy acc.sell n.buy n.sell ret.buy ret.sell
avg    2022 46.67   46.15    47.06    13     17 -0.0926   0.9472
1982    253  0.00    0.00      NaN     1      0 -1.0062      NaN
1983    253   NaN     NaN      NaN     0      0     NaN      NaN
1984    253   NaN     NaN      NaN     0      0     NaN      NaN
1985    252   NaN     NaN      NaN     0      0     NaN      NaN
1986    253   NaN     NaN      NaN     0      0     NaN      NaN
1987    253 40.00   42.86    37.50     7      8 -0.2367   2.2839
1988    253 50.00   33.33    60.00     3      5  0.0254   0.4773
1989    252 66.67  100.00    50.00     2      4  0.6917  -1.1390
```

The function `signals.eval()` calculates the hit rates of our trading actions, the number of different types of actions, and also the average percentage return for our buy and sell signals. The function `annualized.signals.results()` provides average an annualized results for these statistics. When applying this function to our MARS-based actions we observe a very poor performance. The accuracy of the signals is rather disappointing and the average returns are even worse (negative values for the buy actions and positive for the sell signals!). Moreover, there are very few signals generated each year. The only acceptable performance was achieved in 1989, in spite of the low number of signals.

[19] Actually the market could have been falling more often but at least not below 1%.

If we apply this evaluation function to the signals generated with the same thresholds from the projection pursuit and neural network predictions, we get the following results,

```
> annualized.signals.results(buy.signals(pp.preds,buy=0.01,sell=-0.01),select)
     n.sess    acc acc.buy acc.sell n.buy n.sell ret.buy ret.sell
avg    2022  45.61   44.12    47.83    34     23 -0.4024  -0.2334
1982    253  42.86   50.00     0.00     6      1 -0.4786   1.4364
1983    253 100.00     NaN   100.00     0      1     NaN  -1.8998
1984    253 100.00     NaN   100.00     0      1     NaN  -1.8222
1985    252    NaN     NaN      NaN     0      0     NaN      NaN
1986    253   0.00    0.00     0.00     4      2 -0.7692   0.1847
1987    253  48.00   47.37    50.00    19      6 -0.7321  -0.1576
1988    253  70.00   75.00    66.67     4      6  1.7642  -0.7143
1989    252  28.57    0.00    33.33     1      6 -0.8811   0.2966
> annualized.signals.results(buy.signals(nn.preds,buy=0.01,sell=-0.01),select)
     n.sess acc acc.buy acc.sell n.buy n.sell ret.buy ret.sell
avg    2022 NaN     NaN      NaN     0      0     NaN      NaN
1982    253 NaN     NaN      NaN     0      0     NaN      NaN
1983    253 NaN     NaN      NaN     0      0     NaN      NaN
1984    253 NaN     NaN      NaN     0      0     NaN      NaN
1985    252 NaN     NaN      NaN     0      0     NaN      NaN
1986    253 NaN     NaN      NaN     0      0     NaN      NaN
1987    253 NaN     NaN      NaN     0      0     NaN      NaN
1988    253 NaN     NaN      NaN     0      0     NaN      NaN
1989    252 NaN     NaN      NaN     0      0     NaN      NaN
```

Once again, the results are quite bad. Neural networks do not generate any signal at all. These models are making predictions which never go above (or below) our trading thresholds. This should not be a surprise because these situations are quite rare on the training data (*c.f.* Figure 3.4, page 99).

Before we try to improve our results let us see how to use these signals to perform real trading.

3.6.2 A simulated trader

In order to assess the value of a set of trading signals, we need to use these signals for real trading. Many possible trading strategies exist. In this section we will select a particular strategy. No assertions are made regarding the question whether this is the best way to trade in a market. Still, this will serve to evaluate the signals of our models in a more realistic setup.

Our trading strategy

Our trading strategy will follow a certain number of rules and will have a set of constraints. A first constraint is the capital available at the start of the trading period. In our experiments we will start with a capital of 10000 monetary units. An important issue when designing a trading policy is the timing of the orders submitted to the market. We will post our buy orders at the end of each session every time a buy signal is generated by our model. The amount of capital that will be invested is a percentage (we will use as default 20%) of our equity (defined as the available money plus the value of our current stock portfolio). We will assume that this buy order will always be carried out when the market opens (we assume there are always stocks to sell). This means that our order will be fulfilled at the next day's open price. Immediately after posting this buy order we will post a sell order of the stocks just bought. This order will be posted with a target sell price, which will be set as a percentage (we will use 3% as default) above the todays' closing price of the stock. This can be seen as our target profit. The sell order will also be accompanied by a due date (defaulting to 10 days). If the stock reaches the target sell price till the due date the stocks are sold at that price. Otherwise, the stocks are sold at the closing price of the due date session. Finally, we will set a transaction cost of 5 Euros for each order. Notice that this strategy is using only the buy signals of our models, because as soon as we buy we automatically post an associated sell order.

We will now implement this trading strategy. Our goal is to build a function that given the market evolution and our predicted signals, will automatically simulate the trading according to the rules described above, returning as a result a series of statistics that evaluate the trading performance of our signals during a certain period. The function we provide (`trader.eval()`) to implement this artificial trader is a bit long, so we only include it at the book Web site for space reasons.

The simulated trader function produces a list containing several results of the actions taken during the trading period using our signals. Before describing the details of these statistics let us see how to use the function,

```
> market <- ibm[ibm$Date > '1981-12-31' & ibm$Date < '1990-01-01',]
> t <- trader.eval(market,mars.actions)
> names(t)
 [1] "trading"         "N.trades"        "N.profit"        "Perc.profitable"
 [5] "N.obj"           "Max.L"           "Max.P"           "PL"
 [9] "Max.DrawDown"    "Avg.profit"      "Avg.loss"        "Avg.PL"
[13] "Sharpe.Ratio"
```

We started by selecting the quotes data for the trading period. We then used the `trader.eval()` function to trade during this period using the signals of the model that were obtained for that period. We have used all default parameter values of our trading policy. The final instruction shows us the names of the components of the list that the trading function returned.

The `trading` component is a data frame with as many rows as there are sessions in the testing period. For each row (session) we store the date, the closing price of the stocks at the end of the session, the money available at the end of the day, the number of stocks currently at our portfolio and the equity (money+value of the stocks). We can use this data frame to inspect more closely (for instance through graphs) the performance of our trader. The following code is an example of such a graphical inspection, shown in Figure 3.6

```
> plot(ts(t$trading[,c(2,4,5)]),main='MARS Trading Results')
```

Notice how we have used the function `ts()` to transform the results into a time series object, thus taking advantage of the special plotting methods within R.

The remaining components summarize the more relevant features of the trading performance. Namely, we get information on the number of executed trades[20], the number of profitable trades[21] and the percentage of profitable trades. The element `N.obj` indicates how many of the trades achieved our target percentage profit (the parameter `exp.prof` of the `trader.eval()` function). The other trades are completed because of the due date constraint, meaning

[20]The full buy-sell cycle.

[21]We take into account the trading costs in calculating profits.

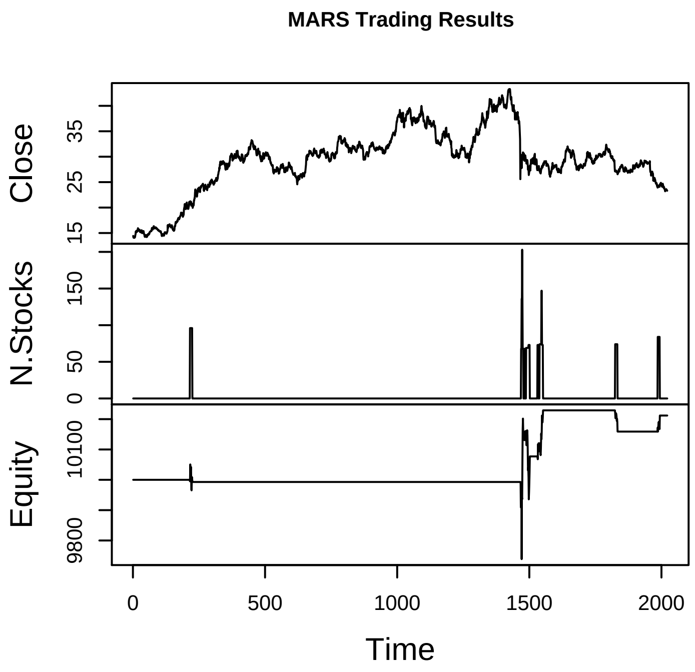

Figure 3.6: Some trading results of the actions entailed by the Neural Network predictions.

that the stocks were sold at the closing price of the session at the due date. The elements Max.L and Max.P show the maximum loss and profit of a trade, respectively, while the element PL shows the overall result at the end of the trading period (that is the gain or loss from the initial capital). The Max.DrawDown shows the maximum successive decrease in the value of the Equity that was experienced during the trading period. The Avg.profit element shows the average gain in Euros of each profitable trade, while Avg.loss shows the average loss of the non-profitable trades. The Avg.PL element shows the average result of the trades carried out. Finally, Sharpe.Ratio is an indicator frequently used in financial markets whose main goal is to provide an idea of the profit and associated risk of a trading strategy. It is calculated as the average profit of the trader over time divided by the standard deviation of this profit. The value should be as high as possible,for a system with a combined high profit and low risk.

As before, it is interesting to have a look at the annualized results,

```
> annualized.trading.results <- function(market,signals,...) {
+
+   res <- data.frame(trader.eval(market,signals,...)[-1])
+
+   years <- unique(substr(market[,'Date'],1,4))
+   for(y in years) {
+     idx <- which(substr(market[,'Date'],1,4)==y)
+     res <- rbind(res,data.frame(trader.eval(market[idx,],signals[idx],...)[-1]))
+   }
+   row.names(res) <- c('avg',years)
+   round(res,3)
+ }
```

If we apply this function to the trading actions of MARS we get the following results,

```
> annualized.trading.results(market,mars.actions)
     N.trades N.profit Perc.profitable N.obj Max.L Max.P      PL Max.DrawDown
avg        13        9          69.231     9 6.857 5.797 211.818      312.075
1982        1        0           0.000     0 0.359 0.000  -7.120       85.440
1983        0        0             NaN     0 0.000 0.000   0.000        0.000
1984        0        0             NaN     0 0.000 0.000   0.000        0.000
1985        0        0             NaN     0 0.000 0.000   0.000        0.000
1986        0        0             NaN     0 0.000 0.000   0.000        0.000
1987        7        5          71.429     5 6.857 5.797  85.087      266.203
1988        3        3         100.000     3 0.000 2.875 149.341       38.580
1989        2        1          50.000     1 3.462 2.625 -17.029       69.150
     Avg.profit Avg.loss Avg.PL Sharpe.Ratio
avg       2.853    3.725  0.829        0.011
1982        NaN    0.359 -0.359       -0.004
1983        NaN      NaN    NaN          NaN
1984        NaN      NaN    NaN          NaN
1985        NaN      NaN    NaN          NaN
1986        NaN      NaN    NaN          NaN
1987      3.100    5.543  0.631        0.014
1988      2.510      NaN  2.510        0.087
1989      2.625    3.462 -0.419       -0.015
```

This function may take a while to run. You may notice that in some years there was no trading at all. Moreover, in some years there is a loss (1982 and 1989). The overall results are not impressive. This function uses the trading simulator to obtain its results. We can give it other trading parameters to check the effect on the results,

```
> annualized.trading.results(market,mars.actions,bet=0.1,exp.prof=0.02,hold.time=15)
     N.trades N.profit Perc.profitable N.obj Max.L Max.P      PL Max.DrawDown
avg        13        9          69.231    10 5.685 4.258  34.672      197.031
1982        1        1         100.000     1 0.000 2.028  20.115        5.000
1983        0        0             NaN     0 0.000 0.000   0.000        0.000
1984        0        0             NaN     0 0.000 0.000   0.000        0.000
1985        0        0             NaN     0 0.000 0.000   0.000        0.000
1986        0        0             NaN     0 0.000 0.000   0.000        0.000
1987        7        4          57.143     5 5.685 4.258 -22.738      197.031
1988        3        3         100.000     3 0.000 1.353  29.594       21.560
1989        2        1          50.000     1 0.321 1.108   7.701       34.160
     Avg.profit Avg.loss Avg.PL Sharpe.Ratio
avg       1.553    2.624  0.268        0.004
1982      2.028      NaN  2.028        0.045
1983        NaN      NaN    NaN          NaN
1984        NaN      NaN    NaN          NaN
1985        NaN      NaN    NaN          NaN
1986        NaN      NaN    NaN          NaN
1987      1.959    3.392 -0.334       -0.006
1988      1.001      NaN  1.001        0.051
1989      1.108    0.321  0.393        0.016
```

As we can see things got even worse with these settings. Still, these poor overall results just reinforce our previous concerns on the quality of the predictions of our models. In the following sections we will try to improve this performance.

3.7 Going back to data selection

Knowledge discovery is a cyclic process. The results of the different knowledge discovery steps are usually fed back to try to improve the overall performance. Motivated by the disappointing trading performance of the experiments of the previous section, we will try to iterate our knowledge discovery process to improve predictive performance.

The models we have obtained for predicting the next day returns used only the lagged values of this ratio. It is common knowledge (e.g. Almeida and Torgo, 2001; Deboeck, 1994)) that this is clearly insufficient for accurate predictions in such a non-linear domain as stock quotes. We will thus try to improve our performance by enriching our data set through the use of other information apart from the previous returns.

3.7.1 Enriching the set of predictor variables

In this section we will add input variables that can carry more information than the returns on the previous days. One possibility is to try to capture some features of the dynamic behavior of the times series. Alternatively, one can also try to include some financial information as input to the models. A common choice is the use of technical indicators. The idea behind *Technical Analysis* is to study the evolution of prices over the previous days and use this information to generate signals. Most of the analysis is usually carried out through charting.

There is a large number of technical indicators[22] that can be used to enrich our set of variables. In this section we will use a few as illustrative examples.

Technical indicators

Moving averages

Moving averages are among the simplest technical indicators. A moving average is an indicator that shows the average value of a security's price over a period of time. The most popular method of using this indicator is to compare the moving average of the security's price with the security's price itself. A *buy* signal is generated when the security's price rises above its moving average and a sell signal is generated when the security's price falls below this moving average.

A moving average can be easily obtained with the following function,

```
> ma <- function(x,lag) {
+   require('ts')
+   c(rep(NA,lag),apply(embed(x,lag+1),1,mean))
+ }
```

As the function needs the `embed()` function which is made available by the package `ts`, we need to ensure that this package is loaded when the user calls our function. This is accomplished through the function `require()` which loads it if it is not loaded yet.

If we want to obtain a moving average of the last 20 days for the closing prices of our IBM stock, we can type the following command:

```
> ma20.close <- ma(ibm$Close,lag=20)
```

We can get a better fell for the use of moving averages as technical indicators by plotting a few of them with different time lags. An example of such plot is given in Figure 3.3, which shows the closing prices of IBM together with two different moving averages for the first 1000 stock sessions. The code to generate the figure is the following,

```
> plot(ibm$Close[1:1000],main='',type='l',ylab='Value',xlab='Day')
> lines(ma20.close[1:1000],col='red',lty=2)
> lines(ma(ibm$Close,lag=100)[1:1000],col='blue',lty=3)
> legend(1.18, 22.28, c("Close", "MA(20)", "MA(100)"),
+        col = c('black','red','blue'),lty = c(1,2,3))
```

As it can be seen from Figure 3.3 the larger the moving average window the smoother the resulting curve.

As mentioned above we can compare the moving average line and the closing prices to obtain trading signals. Instead of looking at the graph, we can build a function to obtain the signals,

```
> ma.indicator <- function(x,ma.lag=20) {
+   d <- diff(sign(x-c(rep(NA,ma.lag-1),apply(embed(x,ma.lag),1,mean))))
+   factor(c(rep(0,ma.lag),d[!is.na(d)]),
+          levels=c(-2,0,2),labels=c('sell','hold','buy'))
+ }
```

[22]Find a very exhaustive list at http://www.equis.com/free/taaz/.

The function `sign()` is used to obtain the signals of subtracting the closing prices from the moving average we want to use. We then apply the function `diff()` to check for changes of the signals in successive time steps. These correspond to the places were the closing prices cross the moving average line. Finally, these differences are used to build a factor with the trading signals. When the difference between the signs is -2 this means that we changed from a closing price above the average (a 1 sign) into a closing price below the average (a -1 sign).[23] This is a situation were the interpretation of the moving average technical indicator says we should sell the stock. When the difference is 2 we have the case where the closing price goes from a value below the average to a value above it, which leads to a buy signal. In the begining of the time series, where we do not have sufficient data to calculate the moving average of the required length, we generate hold actions. The function produces as many trading signals as there are values in the vector of closing prices.

Using this function we can obtain the trading actions suggested by some moving average indicator. The following code shows an example of using a 20-days moving average for trading on the first 1000 sessions, and presents the dates when the first 10 buy signals are generated by this indicator,

```
> trading.actions <- data.frame(Date=ibm$Date[1:1000],
+                               Signal=ma.indicator(ibm$Close[1:1000]))
> trading.actions[which(trading.actions$Signal=='buy'),][1:10,]
          Date Signal
29  1970-02-11    buy
58  1970-03-25    buy
69  1970-04-10    buy
104 1970-05-29    buy
116 1970-06-16    buy
138 1970-07-17    buy
147 1970-07-30    buy
162 1970-08-20    buy
207 1970-10-23    buy
210 1970-10-28    buy
```

We can evaluate this kind of trading signals with our `annualized.trading.results()` function as done before for the signals generated by a MARS model. This is illustrated with the code shown below, for the signals generated by a 30-days moving average during the model selection period,

```
> ma30.actions <- data.frame(Date=ibm[ibm$Date < '1990-01-01','Date'],
+      Signal=ma.indicator(ibm[ibm$Date < '1990-01-01','Close'],ma.lag=30))
> ma30.actions <- ma30.actions[ma30.actions$Date > '1981-12-31','Signal']
> annualized.trading.results(market,ma30.actions)
     N.trades N.profit Perc.profitable N.obj Max.L Max.P        PL Max.DrawDown
avg        96       38          39.583     2 9.195 8.608 -1529.285      1820.19
1982       10        5          50.000     2 7.194 8.608   207.665       243.28
1983       15        5          33.333     0 4.585 3.563  -300.700       385.66
1984       13        5          38.462     0 4.246 3.222  -375.140       391.56
1985       12        2          16.667     0 9.168 3.520  -399.420       428.33
1986       10        4          40.000     0 6.218 1.895  -223.870       295.71
1987       11        6          54.545     0 4.015 5.043    23.170       179.33
1988       11        5          45.455     0 5.054 1.777  -140.190       226.79
```

[23] A better grasp of this reasoning can be obtained by trying the `sign()` function with a small vector of numbers.

```
1989       14        7          50.000     0 6.827 4.608  -358.330       481.30
     Avg.profit Avg.loss Avg.PL Sharpe.Ratio
avg       2.003    2.738 -0.861       -0.059
1982      5.261    3.112  1.074        0.047
1983      1.594    2.332 -1.023       -0.087
1984      1.389    3.272 -1.479       -0.152
1985      1.932    2.433 -1.706       -0.124
1986      1.475    2.880 -1.138       -0.078
1987      2.180    2.360  0.117        0.007
1988      0.770    1.823 -0.644       -0.038
1989      1.363    3.962 -1.299       -0.111
```

We have used the data before the start of the model selection period (recall it was from 1981-12-31 till 1990-01-01), so that we can start calculating the moving average from the start of that period.[24]

As expected this very simple indicator does not produce very good overall trading results. Still, we should recall that our objective when introducing technical indicators was to use them as input variables for our regression models as a means to improve their predictive accuracy. We have three alternative ways of doing this: We can use the value of the moving average as an input variable; we can use the signals generated by the indicator as an input variable; or we can use both the value and the signals as input variables. Any of these alternatives is potentially useful. The same observation applies to the use of any technical indicator.

Exponential moving average

Another type of moving average is the exponential moving average. The idea of this variant is to give more weight to the recent values of the series. Formally, an exponential moving average is recursively defined as,

$$\begin{aligned} \text{ema}_\beta(X_t) &= \beta X_t + (1-\beta) \times \text{ema}_\beta(X_{t-1}) \\ \text{ema}_\beta(X_1) &= X_1 \end{aligned} \tag{3.10}$$

Exponential moving averages can be easily implemented in R with the following code,

```
> ema <- function (x, beta = 0.1, init = x[1])
+ {
+   require('ts')
+   filter(beta*x, filter=1-beta, method="recursive", init=init)
+ }
```

We have used the function `filter()` from package `ts`, which allows the application of linear filters to time series.

Sometimes people prefer to use the number of days to set the time lag of a moving average, instead of using this β factor. We can calculate the value of β corresponding to a given number of days as follows,

$$\beta = \frac{2}{\text{n.days} + 1} \tag{3.11}$$

Inversely, we can calculate the number of days corresponding to a given value of β with,

[24]Actually, as we have used a 30-days moving average we would only need the prices of the previous 30 days.

$$\text{n.days} = \frac{2}{\beta} - 1 \tag{3.12}$$

This means that the default value of β for our `ema()` function corresponds to an exponential average of 21 days ($= 2/0.1 + 1$).

Figure 3.7 shows the difference between a 100-days moving average and an exponential average over the same time window, which was obtained with the following code,

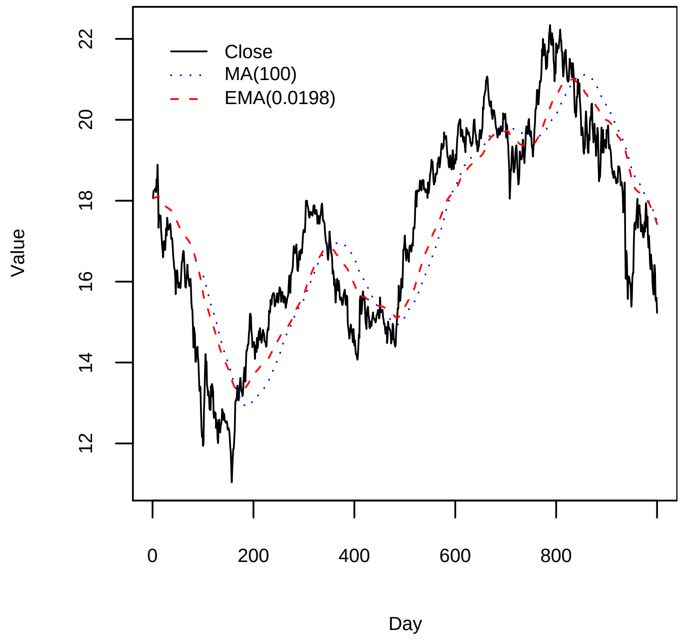

Figure 3.7: 100-days Moving vs. Exponential averages.

```
> plot(ibm$Close[1:1000],main='',type='l',ylab='Value',xlab='Day')
> lines(ma(ibm$Close[1:1000],lag=100),col='blue',lty=3)
> lines(ema(ibm$Close[1:1000],beta=0.0198),col='red',lty=2)
> legend(1.18, 22.28, c("Close", "MA(100)", "EMA(0.0198)"),
+         col = c('black','blue','red'),lty = c(1,3,2),bty='n')
```

As we can see, there is no big difference between these two moving averages, although we can say that the exponential average is more "reactive" to recent movements of the closing prices.

The MACD indicator

Another technical indicator which is frequently used is the Moving Average Convergence/Divergence (MACD). The MACD was designed to capture trend effects on a time series. The MACD shows the relationship between two moving averages of prices. This indicator is usually calculated as the difference between a 26-days and 12-days exponential moving average.[25] It can be obtained with the following function,

[25] The idea is to compare long term and short term averages.

```
> macd <- function(x,long=26,short=12) {
+   ema(x,lambda=1/(long+1))-ema(x,lambda=1/(short+1))
+ }
```

There are several ways to interpret the MACD values with the goal of transforming them into trading signals. One way is to use the crossovers of the indicator against a signal line. The latter is usually a 9-days exponential moving average. The resulting trading rule is the following: Sell when the MACD falls below its signal line; and buy when the MACD rises above its signal line. This can be implemented by the code:

```
> macd.indicator <- function(x,long=26,short=12,signal=9) {
+   v <- macd(x,long,short)
+   d <- diff(sign(v-ema(v,lambda=1/(signal+1))))
+   factor(c(0,0,d[-1]),levels=c(-2,0,2),labels=c('sell','hold','buy'))
+ }
```

Other variables

Technical indicators provide some financially-oriented information for model construction. However, we can also use variables that try to capture other features of the dynamics of the time series.

For instance, we can use not only 1-day returns but also information on larger lagged returns, like 5-, 10- and 20-days returns. We can also calculate the variability of the closing prices over the last sessions.

Another variable which is sometimes useful is a difference of differences. For instance, we can calculate the differences of the returns (which are by themselves differences).

Finally, we can also calculate the linear trend of the prices in the recent sessions.

The following R code implements some of these variables,

```
> d5.returns <- h.returns(ibm[,'Close'],h=5)
> var.20d <- moving.function(ibm[,'Close'],20,sd)
> dif.returns <- diff(h.returns(ibm[,'Close'],h=1))
```

Bibliography

Adriaans, P. and Zantinge, D. (1996). *Data Mining.* Addison-Wesley.

Almeida, P. and Torgo, L. (2001). The use of domain knowledge in feature construction for financial time series prediction. In Brazdil, P. and Jorge, A., editors, *Proceedings of the Portuguese AI Conference (EPIA'01)*, LNAI-2258. Springer.

Bontempi, G., Birattari, M., and Bersini, H. (1999). Lazy learners at work: the lazy learning toolbox. In *Proceedings of the 7th European Congress on Intelligent Tecnhiques & Soft Computing (EUFIT'99).*

Box, G. and Jenkins, G. (1976). *Time series analysis, forecasting and control.* Holden-Day.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24:123–140.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and Regression Trees.* Statistics/Probability Series. Wadsworth & Brooks/Cole Advanced Books & Software.

Chan, R. (1999). Protecting rivers & streams by monitoring chemical concentrations and algae communities. In *Proceedings of the 7th European Congress on Intelligent Tecnhiques & Soft Computing (EUFIT'99).*

Chatfield, C. (1983). *Statistics for technology.* Chapman and Hall, 3rd edition.

Chatfield, C. (1989). *The Analysis of Time Series - an introduction.* Chapman and Hall, 4rd edition.

Cleveland, W. (1993). *Visualizing Data.* Hobart Press.

Cleveland, W. (1995). *The Elements of Graphing Data.* Hobart Press.

Dalgaard, P. (2002). *Introductory Statistics wit R.* Springer.

Deboeck, G., editor (1994). *Trading on the edge.* John Wiley & Sons.

Devogelaere, D., Rijckaert, M., and Embrechts, M. J. (1999). 3rd international competition: Protecting rivers and streams by monitoring chemical concentrations and algae communities solved with the use of gadc. In *Proceedings of the 7th European Congress on Intelligent Tecnhiques & Soft Computing (EUFIT'99).*

Dietterich, T. G. (2000). Ensemble methods in machine learning. *Lecture Notes in Computer Science*, 1857:1–15.

Drapper, N. and Smith, H. (1981). *Applied Regression Analysis*. John Wiley & Sons, 2nd edition.

DuBois, P. (2000). *MySQL*. New Riders.

Freund, Y. and Shapire, R. (1996). Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning*. Morgan Kaufmann.

Friedman, J. (1981). Projection pursuit regression. *Journal of the American Statistical Association*, 76(376):817–823.

Friedman, J. (1991). Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–141.

Gershenfeld, N. and Weigend, A. (1994). The future of time series: learning and understanding. In Weigend, A. and Gershenfeld, N., editors, *Time series prediction: forecasting the future and understanding the past*, pages 1–70. Addison-Wesley Publishing Company.

Hastie, T. and Tibshirani, R. (1990). *Generalized Additive Models*. Chapman & Hall.

Hellstrom, T. and Holmstrom, K. (1998). Predicting the stock market.

Hong, S. (1997). Use of contextual information for feature ranking and discretization. *IEEE Transactions on Knowledge and Data Engineering*.

Ihaka, R. and Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314.

McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133.

Minsky, M. and Papert, S. (1969). *Perceptrons: an introduction to computational geometry*. MIT Press.

Myers, R. (1990). *Classical and modern regression with applications*. Duxbury Press, 2nd edition.

Pyle, D. (1999). *Data preparation for data mining*. Morgan Kaufmann.

Quinlan, R. (1993). *C4.5: programs for machine learning*. Morgan Kaufmann Publishers.

Rogers, R. and Vemuri, V. (1994). *Artificial neural networks forecasting time series*. IEEE Computer Society Press.

Rojas, R. (1996). *Neural Networks*. Springer-Verlag.

Ronsenblatt, F. (1958). The perceptron: a probabilistic models for information storage and organization in the brain. *Psychological Review*, 65:386–408.

Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning internal representations by error propagation. In et. al., D. R., editor, *Parallel distributed processing*, volume 1. MIT Press.

Sauer, T., Yorke, J., and Casdagli, M. (1991). Embedology. *Journal of Statistical Physics*, 65:579–616.

Shapire, R. (1990). The strength of weak learnability. *Machine Learning*, 5:197–227.

Takens, F. (1981). Detecting strange attractors in turbulance. In Rand, D. and Young, L., editors, *Lecture notes in mathematics*, volume 898, pages 366–381. Springer.

Theil, H. (1966). *Applied Economic Forecasting*. North-Holland Publishing Co.

Torgo, L. (1999a). *Inductive Learning of Tree-based Regression Models*. PhD thesis, Faculty of Sciences, University of Porto.

Torgo, L. (1999b). Predicting the density of algae communities using local regression trees. In *Proceedings of the 7th European Congress on Intelligent Tecnhiques & Soft Computing (EUFIT'99)*.

Torgo, L. (2000). Partial linear trees. In Langley, P., editor, *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pages 1007–1014. Morgan Kaufmann Publishers.

Weiss, S. and Indurkhya, N. (1999). *Predictive data mining*. Morgan Kaufmann.

Werbos, P. (1974). *Beyond regression - new tools for prediction and analysis in the behavioral sciences*. PhD thesis, Harvard University.

Werbos, P. (1996). *The roots of backpropagation - from observed derivatives to neural networks and political forecasting*. John Wiley & Sons.

Wilson, D. and Martinez, T. (1997). Improved heterogeneous distance functions. *JAIR*, 6:1–34.

Zirilli, J. (1997). *Financial prediction using neural networks*. Thomson Computer Press.

Index