Statistical programming in R

Lars Snipen

Preface

This text is the result of teaching the course *STIN300 Statistical programming in R* at the Norwegian University of Life Sciences over a few years. It has evolved by gradual interaction between the teachers and the students. It is still evolving, and this is the 2015 version.

Contents

Chapter 1

Getting started

1.1 Installing R

The latest version of R is available for free from http://www.r-project.org/. The Comprehensive R Archive Network (CRAN, http://cran.r-project.org/) has several mirror sites all over the world, and you choose one to download from. The Norwegian mirror is at the University of Bergen (UiB).

The R software consists of a *base* distribution plus a large number of *packages* that you can add if you like. You start by downloading and installing the base distribution. Choose the proper platform (Windows, Mac of Unix) for your computer. We will come back to the packages later.

Our own resource for R is found at http://repository.umb.no/R/. Here you can find documents that may be helpful for installing R. Please note that we will not use R Commander in this course, you only need to install R.

1.2 The R software

Having installed R, you *can* start using it right away. However, in these days it is customary to use some Integrate Development Environment (IDE), and in this course we will use the RStudio software. Before we move on to RStudio we should just briefly look at how R can be used alone.

1.2.1 The windows in R

If you start R the first window you see is the *Console* window. This is the main window, and where we communicate with the R software. This is an example of a command line communication. Most people are used to graphical user interfaces, but this is different. There are no push-buttons here. In order to communicate with R we have to write commands in the Console window, and R (usually) responds by outputting some text in the same window. The Console window has a menu. Under the `Edit` menu you find
`GUI preferences...`, where you can to some extent change the appearance of your Console window.

Since we need to write programs, we will very soon need an editor where we can write, edit and save text. In principle you can use any text editor (e.g. Notepad, Wordpad etc) as long as you can save your text as simple text files. However, R also has a built-in editor. If you choose the `File` menu on the Console window, and choose `New script` you should get the Editor window opened. In this window you can now type any text, and save it on file in the same way you do with any simple text editor.

Sometimes R does not respond to our commands by outputting text, but rather by displaying some graphical output. This will appear in the Graphics window. Move to the Console window and type the command

```
> plot(c(1,2,3,4),pch=16,col="red")
```

on the command line, and return. This should produce a Graphics window with a simple plot.

Let us now move on to the RStudio software that we will use throughout this course.

1.3 The RStudio software

Instead of running R directly, as in the previous section, it has been more common these days to make use of some of the freely available IDEs made for R. One such IDE is RStudio. You download and install RStudio from http://www.rstudio.com/.

With RStudio you have all windows in one customizable environment, and you also have many facilities available through menus and toolbars. We will not spend much energy on all the possibilities in RStudio in this course, but some of the basic features will become apparent as we proceed. The previously mentioned Console, Editor and Graphics windows are available in RStudio, along with some other windows as well. An example of an RStudio workplace is shown in Figure 1.1.

RStudio divides the workplace into 4 panes (2 by 2), and you can to some degree decide which windows should appear in each pane. On the menu you choose `Tools` and `Options...` and a window pops up where you can edit the pane layout (among other things). The two most important windows are the Console window (Console) and the Editor window (Source). These will typically be in the upper two panels, but feel free to customize this if you like.

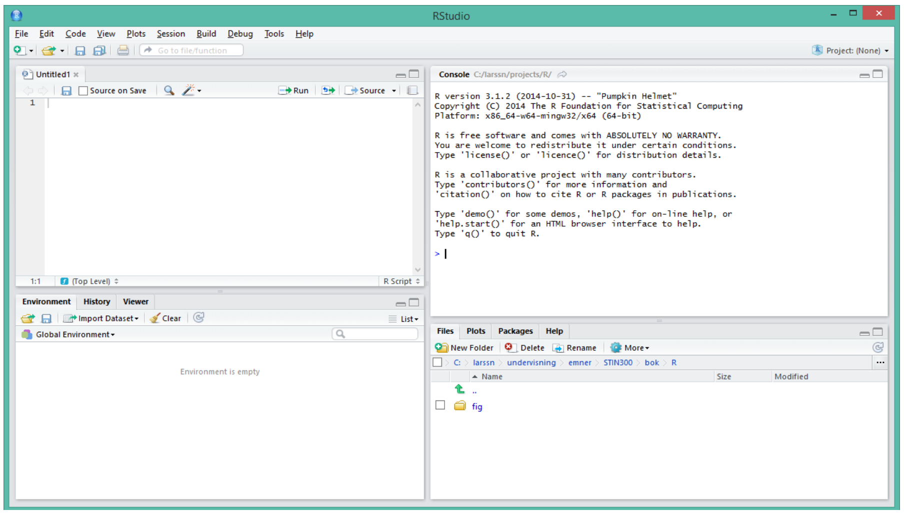
1.4 Help facilities

1.4.1 The Help window

In RStudio you open the Help-window through the menu, choose `Help` and then `R Help`. In Figure 1.2 the main Help-window is displayed in RStudio. If you follow the link named An Introduction to R and learn everything you find there, you can skip the rest of this course!

1.4.2 Help on commands

A command in R is called a function. The simplest and most common way of obtaining help for a given function is to simply write `?` and the function name


RStudio
File Edit Code View Plots Session Build Debug Tools Help
Project: (None)
Untitled1
Source on Save
Run
Source
1:1
(Top Level)
R Script
Console C:/larssn/projects/R/
Environment History Viewer
Import Dataset
Clear
List
Global Environment
Environment is empty
Files Plots Packages Help
New Folder Delete Rename More
C: larssn undervisning emner STIN300 bok R
Name Size Modified
fig

```
R version 3.1.2 (2014-10-31) -- "Pumpkin Helmet"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Figure 1.1: An example of the RStudio workplace, containing several customizable windows as well as a rich menu and toolbars.

in the Console window. As an example, let us look for help for the function called `lm` in R: Write

```
> ?lm
```

in the Console window, and then return. The Help-file for this function will be displayed. The command `help(lm)` is just an alternative to the question mark, and produces identical results.

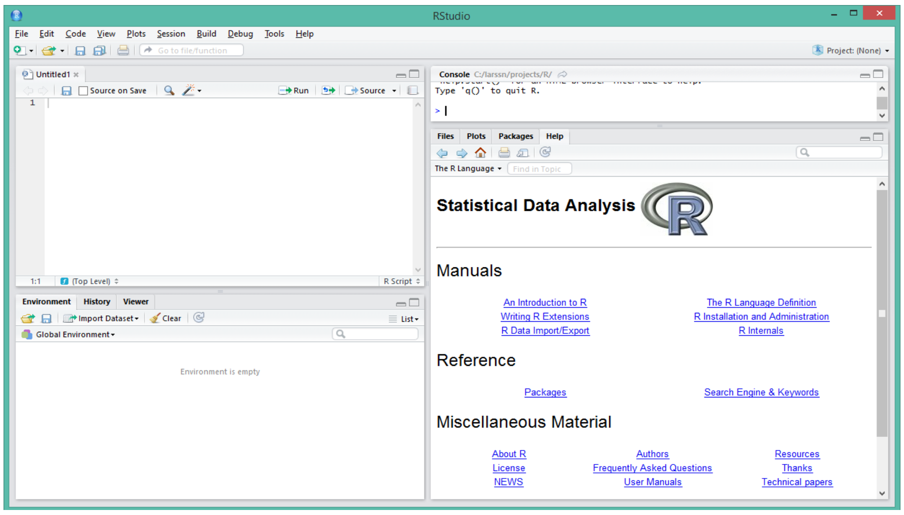
NOTE: When we start some code by `>`, as above, it indicates we write this directly in the Console window. You do not actually type in the `>`. We will see later that we often write code in files instead, and then we have no `>` at the start of the lines.

1.4.3 Searching for commands

The problem with this approach is that you need to know the name of the function you are seeking help for. When you are new to R, you will in general know very few function names. In these cases you could use `help.search`:

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

which will give you a list of available functions where the term `"linear"` is mentioned. If you scroll down this list you should find somewhere the entry stats::lm. This means that in the package called `stats` there is a function called `lm` and that the help file for this function uses the term `linear`. But, as you can see, the list of hits is rather long, and the term `linear` is found in many Help-files.



```
Type 'q()' to quit R.
>
```

Figure 1.2: The main Help window invoked in RStudio.

1.4.4 Partly matching command names

The function `apropos` can be used to get a list of available function that partly match a name. If you do not remember exactly what a function name was, you can run

```
> apropos("part")
```

and get a listing of all available functions containing the term `part` in its name.

1.4.5 Web search

The Help facilities we have seen so far will only give you help on functions and packages that you have installed on your system. To make a search on the internet, you can use the function:

```
> RSiteSearch("linear")
```

to get an overview of (official) available packages and functions on the internet. Note: You need to have an internet connection to make this work. Note also that this will only look up the official R sites, there is most likely much more to be found on the internet if you make a straightforward Google search (e.g. ”linear in R”).

1.4.6 Remarks on quotes

Before we conclude this section, please note how we used the quotes `""`. The name of a function, like `lm`, was never enclosed in quotes. But, the text `"linear"` and `"part"` must have the quotes in order to be seen by R as texts. We will look more closely at texts later, but already now you should take notice of the difference between a command (function names are commands) that we want R to interpret, and a text that should not be given any interpretation.

1.5 Demos

There are several demos in the base distribution of R. Type

```
> demo()
```

in the Console window, and return, to get a list of available demos. To run a specific demo, e.g. `graphics`, type

```
> demo(graphics)
```

in the Console window.

1.6 Important functions

One important aspect of learning R is to get familiar with the commands available. Most of these commands are functions (we will talk more about functions later) and have a Help-file you should look into. I will end every chapter with a section named Important functions, to sum up some of the important commands we have seen in that chapter. In this first chapter we have not really seen much, but this will soon change!

1.7 Exercises

1.7.1 Install and customize

Install R and RStudio on your laptop. Customize the appearance of RStudio. If you want to have the same pane layout as the teacher, take a look at Figures 1.1 and 1.2 for guidance.

Open the Help on RStudio (`Help` and then `RStudio Docs`). This will open in a separate web-browser. We will not spend time on learning RStudio as such, but you may find it helpful to take some time exploring this.

1.7.2 Demos

Run some of the demos available. At least try out `demo(graphics)` and `demo( persp)`.

1.7.3 Help facilities

In R we have a function named `log`. Find its Help file, and read about this function. R Help files follow the same basic format, with some variation. In this case you will also see there are other functions documented in the same file. If several functions belong together in some way, this is done to reduce the number of Help files. If you search Help for `exp` you will be guided to the same Help file.

Some questions:

1. What does `log` do?
2. There are also some other functions listed in the same file. What does `log1p` do? Why?
3. Read the Usage, Arguments and Details sections, and compute the natural logarithm of 10 in the Console window.
4. Compute the logarithm with base 3 of 10.
5. Compute the logarithm of 0 and of -1.
6. At the end of the file are always some examples. Copy these into the Console window line by line, observe the output and try to understand what happens.

Chapter 2

Basic operators and data types

2.1 Arithmetic operators

R is a computing environment, and we have all basic arithmetic operators available. Here are addition, subtraction, multiplication and division:

```
2+2
2-2
2*2
2/2
```

Some other frequently used operators are exponentiation, square-root and logarithm:

```
2^2
sqrt(2)
log(2)
log2(2)
log10(2)
```

Notice that `log()` means natural logarithm, usually called ln in mathematical notation. The base-2 and base-10 logarithms have their own functions (see exercises in previous chapter).

If you type the lines above in your console window, ending each line with a return, the result of each operation is printed in the console window directly. However, the results are never stored anywhere.

2.2 Variables and assignment

In order to start programming we need *variables*. The fundamental property of any computer is the ability to store something in a *memory*. A variable can be seen as a location in the computer memory where we can store something.

This 'something' depends on the data type of the variable. In R, as in all programming languages, we have several data types available.

The most common data type in R is called a *numeric*. A variable of type numeric can store numbers, all kinds of numbers. We can construct a numeric variable by just giving it a name and a numerical value. Try this in the Console window:

```
> a <- 2
```

This line of code should be read as follows: Create the variable with name `a` and assign to it the value `2`. Notice how we never really specified that `a` should be a numeric data type, this is understood implicitly by R since we immediately give it a numerical value. This is different from most programming languages, where an explicit declaration of data type is required for all variables.

Notice also the assignment operator `<-`, which is quite unique to R (the S-language). It is a left-arrow, which is a nice symbol, since it means that whatever is on its right hand side should now be stored in the memory location indicated by whatever is on its left hand side. Information flows along the arrow from right to left. You can even write

```
> 2 -> a
```

and the assignment is identical! Again, the value 2 is assigned to the variable a, information flows along the arrow direction. In R you are also allowed to use `=` as the assignment operator, i.e. we could have written `a=2` above. This is similar to some other languages, but notice that it says nothing about flow direction, and you *must* always assign from right to left (e.g. you cannot write `2=a`). We will stick to the `<-` for assignments in this text.

The arithmetic operators can be used on variables of type numeric:

```
> a <- 2
> b <- 3
> c <- a*b
```

which results in three variables (`a`, `b` and `c`), all of type numeric, and with values 2, 3 and 6. To see the value of a variable, just type its name in the console window, and upon return R will print its value.

2.3 Other data types

2.3.1 Integer

Not all numbers are numeric. In some cases R will use the type *integer* instead of numeric. An integer occupies less memory than a numeric. If you *know* that a variable will never contain decimal numbers, you can save some memory by just using the integer data type. You can specify that a number should be stored as an integer by adding an `L` after the number:

```
> a <- 2L
```

We will look more into this when we come to vectors in Chapter 4.

2.3.2 Text

In addition to numbers, we often need texts when dealing with real problems. A variable that can store a text has the data type *character* in R. We can create a character variable in the same way as a numeric:

```
> name <- "Lars"
```

Notice the quotes, a text must always be enclosed by quotes in R. The arithmetic operators are in general meaningless for texts. Text-handling is something we will come back to in later chapters.

2.3.3 Logicals

A *logical* data type is neither a number nor a text, but a variable that can take either the value `TRUE` or `FALSE`. Here is an example:

```
> male <- TRUE
```

where the variable `male` now is a logical. Notice that there are no quotes around `TRUE` since this is *not* a text. Logicals are often used as the outcome of some test, which we will see more of soon.

2.3.4 Factors

Since R has been developed by statisticians a separate data type has been created to handle categorical data. Categorical data are discrete data without an ordering. Here is an example:

```
> nationality <- factor("norwegian",levels=c("swedish","
   norwegian","danish"))
```

This example shows several things. First, the variable `nationality` is created, and a factor is assigned to it. The factor value is specified as a text (`"norwegian
"`) but then changed to a factor by giving it as input to the command `factor
()`. This command also takes another input, namely a specification of which category levels this factor is allowed to have. Hence, the text we use as first input to `factor` must be one of those listed in the second input. Factors can be created from numbers as well, replacing the texts in the example above with numbers.

2.4 Type conversion

It is to some degree possible to convert between data types. Sometimes we makes specific use of this in order to create solutions. Other times we create errors by implicitly making conversions where we did not mean to. Since there are no declarations in R, we are more free to convert between data types than in most programming languages. This freedom can be both a benefit and a problem. The problems occur when programs run without errors, but produces meaningless results. Thus, we need to understand how data types are converted in R to make proper use of this facility. The explicit conversion functions for the basic data types are `as.character()`, `as.numeric()`, `as.logical()` and `as.factor()` (there are many more conversion functions).

2.4.1 Converting to character

Converting to character is usually non-problematic. Any number, logical or factor can be converted to a character (text). Converting a number to a character is done as follows:

```
> a <- 12
> a.as.txt <- as.character(a)
```

resulting in `a` being a numeric variable with value `12` and `a.as.txt` being a character variable with value `"12"`. Converting from character to numeric is only possible if the character contains *only* numbers. If not, something strange occurs:

```
> a.as.txt <- "12a"
> a <- as.numeric(a.as.txt)
[1] NA
Warning message:
NAs introduced by coercion
```

This results in the variable `a` having the value `NA`, which means Not Available. This is the R way of saying that a value is missing. It was simply not possible to convert the text `"12a"` to a number, but the variable `a` is still created and assigned `NA` as value. We will talk more about `NA` later in this chapter.

2.4.2 Logicals and numerics

A logical can always be converted to a numeric. The rule is that `TRUE` is converted to `1` and `FALSE` to `0`. We can also convert any numeric to a logical. The value `0` is converted to `FALSE` and *any other value* to `TRUE`. Notice this lack of symmetry in the conversions!

2.4.3 Factor to numeric

The conversion from factor to numeric is sometimes very useful. This is always possible irrespective of how the factor levels are indicated. If we use the example from above we can convert `nationality` to a numeric:

```
> nat <- as.numeric(nationality)
```

Now the variable `nat` is a numeric with value `2`. The reason for this value is that `nationality` has the value `norwegian`, which is the second level of this factor. Levels are numbered by their occurrence in the `levels` specification. Notice that even if the factor was specified by the use of characters (texts), the factor itself only store levels, and these levels can always be converted to numeric. Hence, we can always convert texts to numbers by first converting to factor and then to numeric.

2.4.4 Revealing a variables data type

When we have created many variables, it is sometimes useful to inspect some of them and see which data type they have. The function `class()` gives you this information:

```
> class(a.as.txt)
[1] "character"
> class(nat)
[1] "numeric"
> class(male)
[1] "logical"
```

2.5 Non-types

It is possible to create a variable without any data type in R:

```
> a <- NULL
```

The `NULL` type is a 'neutral' or unspecified data type. We create the variable, give it a name, but does not yet specify what type of content it can take.

We briefly encountered the `NA` value above, indicating missing information. This value is special in the sense that all data types can be assigned `NA`. Any variable, regardless of data type, can take the value `NA` without causing any problems. This makes sense, since any type of data can be missing.

2.6 Variable names

We are free to choose our own names on variables, within certain limits. Try to use names that describe the variables, and avoid short names like `a`, `b` and similar used in the examples here. In addition to the letters of the English alphabet (always avoid the æ, ø and å!) a variable name can also contain integers and some additional characters like `_` (underscore) or `.` (dot). The dot has no special meaning in R, like it has in many other programming languages, and is frequently used as a 'separator' in case the variable name is a concatenation of two or more words. For example, names like `car.registration.number` can be seen in R programs, while in Java it would by convention be `carRegistrationNumber`. The latter is of course also possible in R, and replacing the `.` with `_` is a third option. You decide.

2.7 More operators

In addition to calculations we can also make comparisons. We have in R the standard comparisons

```
a < b
a > b
a <= b
a >= b
a == b
a != b
```

The first four are 'larger/smaller than' and 'larger/smaller or equal to'. The fifth operator is 'equals'. Notice the double equal symbol. The last is 'not equal'. The outcome of all these comparisons are the logical values `TRUE` or `FALSE`. We can store the outcome of such tests just as any other value:

```
> a.is.greater.than.b <- a>b
```

and the variable `a.is.greater.than.b` now has the value `TRUE` or `FALSE` depending the values of `a` and `b`.

In R we have a clear distinction between assignment, which is done by the `<-` and comparison by `==`. This distinction is important, and not always clear to people new to programming. Typing `a<-b` means we copy the value of `b` into `a`. Typing `a==b` just compare if the value of `a` equals the value of `b`, no assignment or copying is made.

Variables of the data type logical are usually the result of some comparison.

2.8 Warning: re-assignments

Since we have no declarations in R, a variables data type can change during program execution. This can be of some benefit, but will perhaps more often cause problems. Here is an example:

```
> a <- 2
> class(a)
[1] "numeric"
> a <- "test"
> class(a)
[1] "character"
```

First we create the variable `a` and assign to it the value `2`. This makes `a` a numeric. But, then we assign a text to this variable, and R does not complain, but immediately changes the type of `a` to character! There is no warning or error, and it is easy to see how such re-assignments can cause problems in larger programs.

You are hereby warned: Always keep track of your variables, R will not help you!

2.9 Important functions

Command	Remark
`factor`	Creates a variable of type `factor`
`class`	Returns the type of a variable
`as.character, as.numeric` etc.	Conversion functions
`sqrt, log, exp` etc.	Basic mathematical functions

2.10 Exercises

2.10.1 Volume of cylinder

We will compute the volume of a cylinder. Create two numeric variables named `height` and `radius`, and assign values to them. Let them have values 10 and 1, respectively. Then, let the variable `volume` be the volume, and use `height` and `radius` to compute the corresponding value for `volume`. HINT: The number π (3.1415...) is always available in R as `pi`. Give new values to `height` and/or `radius`, and recompute the volume. HINT: Use the up-arrow on your keyboard (multiple times) to repeat previous commands in the Console window.

2.10.2 Data type conversion

Create four variables of different data types: `name` containing your name (character), `age` containing your age (numeric), `male` containing your gender (logical) and `student` indicating if you are `bachelor`, `master`, `phd` or `other` (factor). Try to convert each data type to the other three data types. Make a 4×4 table (use pen and paper!) with one row/column for each of the four basic data types mentioned here. Let rows indicate 'from' and columns indicate 'to' and write in each cell if a conversion 'from' a certain data type 'to' a certain data type is always (A), sometimes (S) or never (N) possible.

2.10.3 Special values

Compute `a <- 1/0` and `b <- log(0)`. What are the values of `a` and `b`? Compute `c <- a*0`. What is the value of `c`? Read the Help-files for `Inf` and `NaN`.

2.10.4 Operator priorities

The priority of arithmetic operators are in R just as in any other programming language. Compute `2*3+4/5-6` in R, but see if you can find the solution manually (calculator) first. Next, add parentheses to make it `-14`.

2.10.5 Implicit conversions

R will implicitly convert between data types, when possible. Create the variables `aa<-2`, `bb<-TRUE` and `cc<-factor("A")`. Try to compute `aa+bb`. What happens? Try `aa+bb+cc`. What happens now? How about `aa+bb+as.numeric(cc)`?

Chapter 3

Scripting

3.1 R sessions

So far we have typed commands directly in the Console window. This is not the way we usually work in an R session.

3.1.1 Scripts

Instead we create a file (or files), and put all our commands in this file, line by line as we did in the console window. Such files are called *scripts*. Then, we save the file under a name using the extension `.R` as a convention. In order to execute the commands, we can either copy them from the file and paste them into the Console window, or more commonly, use the `source()` function. This means we spend 99% of the time in the Editor window, and only visit the Console window each time we want to execute our script.

In RStudio we have a separate editor for writing scripts. From the menu you choose `File - New`, and then `R Script` to create a new script-file. It will show in the Source pane. Here you can type in your commands and save the file as in any simple editor. In the header of the editor window you find some shortcut buttons to run the code. The Souce-button will run the entire script, and corresponds exactly to the `source()` function mentioned above. The Run-button is used to run only parts of the script. Either the single line of code where your cursor i located, or you first select the code-lines using your mouse, and the press Run to execute only the selected lines of code.

3.1.2 Working directory

Any R session runs in a *directory* or *folder* on your computer. The default choice, set during installation, is rarely a good choice in the long run. Instead you create a folder at a proper location on your computer, and run R in this folder. We will refer to this as the Working Directory, since this is the term used by RStudio. In RStudio you can easily change the default Working Directory. From the menu, choose `Tools - Options...` and a window pops up where you can customize your RStudio in various ways. In the field named `Default working directory` you can specify in which folder you would like RStudio to start each time. In the rest of this text we will use `RHOME` to symbolize the Working Directory, i.e. replace `RHOME` with the full path to your Working Directory.

It is quite customary to create a folder for each project where you are going to make R-scripts and other R-files that we will see later. In RStudio you can define such projects by using the `Project` on the menu. Try to create a new project, and you will see that the first thing you must decide is the directory (folder) where it resides. Get used to organizing everything in directories. It is one of the most common errors for R-beginners to store their files in one directory and then run R in another, and of course nothing works!

In RStudio there are several ways to change the Working Directory during a session. If you have a script in the editor, and this has been saved to some directory, you can make this the Working Directory by the menu `Session`, then `Set Working Directory` and then `To Source File Location`. Another option is to use the Files-window (look for the Files tab in the lower to panes). In this window you can maneuver through your file-tree. In the header of this window there is a pop-up menu named `More`. From this you can choose `Set As Working Directory` and the folder in the Files-window is now be your working folder.

Your Working Directory is always listed in the header of the Console-window. Note that in R we use the slash `/` and not the backslash `\` whenever we specify a file-path, even in Windows.

3.2 The first script

Let us make a small script. Open the editor window (`File - New - R script`) and type in the lines

```
v1 <- 2
v2 <- 3
w <- v1*v2
d <- v1^v2
```

and save the file under the name `script_3_1.R`. NOTE: Lines of code in this text that does *not* start with a `>` is supposed to be written in a file (script), not directly in the Console window.

Click the `Source` button. If your script is error free it will produce no output, and nothing really seems to happen. Well, something did happen. The four lines of code were executed from the top down. First the variable `v1` was created, and given the value `2`. Next, the variable `v2` was created, and so on. The four variables still exist in the R memory, even if they are not displayed. If you want to list all existing variables in the Console window, use the command `ls()`. If you type `ls()` in the Console-window it should look something like this now:

```
> ls()
[1] "v1" "v2" "w" "d"
```

which means there are 4 variables existing, having the names listed. If you type the name of any of them in the Console window, R will output its value. These variables will now exist in the R memory until you either delete them or end you R session.

3.3 The Global Environment

As previously mentioned, a computers basic property is to store data in a memory. When you run an R session, the software R has been allocated a certain amount of the computers internal memory. R divides this into several parts, but the most important one is called the Global Environment. This is the basic R memory. In RStudio you will find a window named Environment, look among the tabs in the panes where you do not have the Source and Console. This Environment window will, by default, list the content of the Global Environment. You should see the variables named `v1`, `v2`, `w` and `d` listed here now. In the upper right corner of the Environment window you can switch between a List and a Grid view of the content, providing slightly different information.

Notice that the Global Environment is not the only Environment you can have. Packages are also environments, but we will come back to this later, and for now we will only focus on the Global Environment.

You can delete a variable from the workspace by the function `rm()`:

```
> rm(v1)
> rm(list=ls(all=TRUE))
```

where the first line deletes variable `v1` only and the second deletes everything listed by `ls()`. The latter command has a shortcut in RStudio. In the header of the Workspace-window there is a broom (a Harry Potter flying device) button, and if you push it the entire workspace is cleared.

When you end R (use the command `q()` or the `File - Quit RStudio` menu) you are always asked if the 'workspace image' should be saved (workspace means Global Environment here). In most cases we do not save this. The reason is that we try to build programs such that everything is created by the running of scripts (and functions), and the next time we start R we can re-create the results by just running the same scripts again. This is the whole idea of scripting. We store the recipes, and re-compute the results whenever needed. There are exceptions, if the results will take very long time to re-compute you should of course store them!

When we write R-programs (scripts) it is often a good practice to create all variables within the program, not relying on that certain variables already exist in the Global Environment. It is actually a good habit to clear the Global Environment before you re-run a script, as existing variables may cause your program to behave erroneously. Make a habit of always clearing the Global Environment before you `Source` your scripts.

3.4 Comments

As soon as we start to build larger programs, we will need to make some *comments* in our script files. A comment is a note we leave in our program to make it easier to read, both for others and ourselves. It is amazing how much you have forgotten if you open a script you made a month ago! Comments are not read by R, they do not affect the program execution at all, they are only there for our eyes. In R you use the symbol `#` to indicate the start of a comment, and the rest of the that line will be ignored by R when the script is run. Sometimes we use this to *comment out* some lines in our script, usually in order to test the remaining code to search for errors. Here is an example in the script we just made:

```
a <- 2
b <- 3
#c <- a*b
d <- a^b  # d should be 8
```

If you run this script line 3 is skipped because it starts with a `#`, and the variable `c` is never created. The comment in the last line does not affect anything because it is after the code of that line.

The RStudio editor recognizes an R-comment, and will display it in a different font and color, making it easy to spot comments in larger programs.

3.5 Important functions

Command	Remark
`cat`	See exercises below

3.6 Exercises

3.6.1 Directories

Make a directory on your computer for this course, and make RStudio use this as startup-directory. You may find it convenient to create subdirectories under this later, e.g. separate subdirectories for exercises, data, the compulsory project etc.

3.6.2 Cylinder volumes again

Use the cylinder volume exercise from the previous chapter, and make a script where you assign values to the height and radius, and then computes the volume. Output the values of height and radius as well as the corresponding volume on the Console window by the use of the command `cat`. Read about this function in the help-files (type `?cat` in the console). HINT: In `cat` you typically splice together all elements that should make up your output, like `cat( "Hello", " world", "\n")`, which will be concatenated into a single string before output. The symbol `"\n"` means line-break. Remember that (almost) anything can be converted to a string (and then concatenated)! This means the `cat` function can take many arguments of different data types, and they are all converted to text and 'glued' into a single string. Extend the script by reading height and radius from the console by using the function `scan`.

Chapter 4

Basic data structures

4.1 What is a data structure?

A simple answer is a variable with the capability to store several values in some structured way. All programming languages have data structures, and it is not until we have some of these under our control that we can really start programming.

4.2 Vectors

4.2.1 Creating a vector

The basic data structure in R is the vector. It is a linear structure of elements, and corresponds very well to what we think of as vectors in mathematics and statistics. However, a vector in R can be of any data type, i.e. not restricted only to numbers but also be filled with either texts, logicals or factors. Note the 'either-or' here. In a vector we can have *either* numbers, *or* texts *or*...etc. We cannot mix different data types inside the same vector.

A vector can be created by the function `c()` (short for concatenate). Let us make a new script-file, and fill in the following lines of code:

```
# Creating vectors using the c() function
month <- c("January","February","March")
days <- c(31,28,31,30)
```

where the first vector is of data type character and has 3 elements, and the second is of type numeric and has 4 elements. Note that each element is separated by a comma. Save this file under the name `script_vectors.R`", and run it (use the source-button i RStudio or type `source("script_vectors.R")` in the console window). Verify that the two vectors are created, they should appear in the Workspace-window of RStudio.

Vectors are *indexed* which means every element has a 'position' in the vector and that we can refer directly to an element by its position-number, the index. If we write `month[2]` we refer to element number 2 in the vector `month`. The brackets `[]` are *only* used for indexing in R. We will look at this in detail below.

4.2.2 Extending vectors

We can use the `c()` function to add new elements to existing vectors. Add the following line in the script from above:

```
month <- c(month,"April")  # extending a vector
```

and the vector `month` now has 4 elements. The new element is added at the end. Had we written `c("April",month)` it would have been put first. In general, the `c()` function can be used to splice together several vectors.

4.2.3 Systematic vectors

Quite often we need vectors with a systematic ordering of elements, like this:

```
> idx <- 1:4
```

which fills `idx` with the integer values `1,2,3,4`. To create more general sequential vectors, use the function `seq()`. Then you can specify the starting value, the ending value and the step-length, and none of them need be integers (see `?seq`).

Another systematic vector can be created by `rep()`. Here is an example of two frequent uses of this command:

```
> rep(idx,times=3)
 [1] 1 2 3 4 1 2 3 4 1 2 3 4
> rep(idx,each=3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

NOTE: Whenever we start a line of code with a `>` as above, it just indicates this is done directly in the Console window, not in a script-file! The output directly below the statement is what you will see in the Console window.

4.2.4 Properties of vectors

In object oriented programming languages (e.g. Java, python C++ etc.) all variables are objects with attributes or properties. They typically carry a set of functions or methods that you can invoke in each object. In R this is different. R is basically a *functional* language instead of a object oriented, even if object orientation is possible to some degree also in R. More of this later.

The properties of a data structure like a vector is in R decided by the functions available. A typical example is the function `length`, taking a vector as input and returning the number of elements in that vector:

```
> length(month)
 [1] 4
```

The data type of a vector is given by `class`:

```
> class(idx)
 [1] "integer"
```

and here we see an example of the data type *integer*. We mentioned already in Chapter 2 that numbers can be either numeric or integer. When we create a vector as a systematic sequence, like we did above, R will make it an integer. If we made it 'manually' it would be a numeric:

```
> idx <- c(1,2,3,4)
> class(idx)
 [1] "numeric"
```

The reason for this distinction may be more apparent after the next subsection.

Vectors can have names! This means every element in a vector can have its separate name. This is something we make use of from time to time when handling data. The function `names` is used both to extract the names from a vector and to assign (new) names to a vector. We can give names to the numeric vector , add this to the script (and re-run):

```
names(days) <- month  # adding names to a variable
```

which means each element of `days` now also has a name in addition to the value. Type `days` in the console to see its values and names. Notice that the names are themselves a vector, a character vector, that is stored along with the 'real' vector. Remember, the value of, say, `days[2]` is still `28`. Its name is just a label.

4.2.5 All scalars are vectors

We should be aware that all scalar (single element) variables we create in R are in fact vectors of length 1. We should make no distinction between scalars and vectors, but view them all as vectors. The functions `length`, `class` and `names` work just as fine on a single variable, and if you have created a scalar variable `x` you can refer to it both as just `x` or `x[1]`.

4.2.6 Operators on vectors

A vector of numeric (or integer) can be added, subtracted, multiplied or divided by a single number, e.g.:

```
> days+2
 January February    March    April
      33       30       33       32
```

Notice that the new vector produced, with the increased values, 'inherits' the names from the vector `days`. This is a general rule in R, if the input is named, the output inherits the names, if possible.

The same arithmetic operator can also be used between vectors, as long as they have the same number of elements, e.g.

```
> days - days
 January February    March    April
       0        0        0        0
```

and the operators are used elementwise. This means the result is a new vector, having the same number of elements as the two input vectors, and where every element is the result of using the operator on the corresponding pair of input elements.

The same applies to comparisons, see chapter 2 for an overview. We can compare two vectors as long as they are of the same length, and the result is a new vector where each element is the result of comparing the corresponding input elements. Comparing a vector to a single value is also straightforward:

```
> days>30
 January February    March    April
    TRUE    FALSE     TRUE    FALSE
```

i.e. every element of the vector is compared to the value, and the result of this comparison is returned in the corresponding element of the logical vector produced. Notice again how the element names are inherited from `days`.

4.3 Vector manipulation

One important aspect of R programming is the ability to write *vectorized code*. We will return to this important concept later, but fundamental to this is the understanding of how we can manipulate vectors. This is one of the topics most people find difficult when learning R programming.

4.3.1 Indexing

We can retrieve an element by referring to its position using the brackets `[]`:

```
> a.month <- month[3]
```

which means we create a variable called `a.month` and assign to it the value of element number 3 in `month`. We can also assign new values to an element in a similar way:

```
> days[2] <- 29
```

We can retrieve a selection of elements from a vector

```
> months <- month[2:3]
```

resulting in `months` now being a vector of length 2, having the same values as elements 2 and 3 of `month`. We should stop for a second at this example, because it is important to realize exactly what we are doing here. In fact, the example can be illuminated by replacing the statement above by

```
idx <- 2:3
months <- month[idx]
```

Add these two lines to the script, clear the workspace, and re-run.

We first create a vector called `idx` containing the values `2,3`. Next, we use this vector to *index* a subset of `species`, and create `months` to store a copy of this subset. Let us refer to `idx` here as the *index vector*. The index vector should typically only contain integers, since it will be used to refer to the elements in another vector. It should also contain integers in the range from 1 to `length(species)`, i.e. no negative values, no zeros and no values above 4 in this case. Apart from this, there are no restrictions. How about if we decided to make

```
> idx <- c(1,2,3,3,2,3,2,2,1)
```

Could we still use `idx` to extract elements from `month`? The answer is yes:

```
> month[idx]
[1] "January" "February" "March" "March" "February"
    "March" "February" "February" "January"
```

Notice that the index vector can be both shorter or longer than the vector we retrieve from, as long as it only contains integers within the proper range. Notice also that it is the length of the index vector that determines the length of the resulting vector. If `idx` has 100 elements, the result here will also have 100 elements, regardless of how many elements `month` had.

It should now be apparent that any index vector should be of data type integer, since it should only contain integers. In the last example above, we created `idx` in a 'manual' way, and if we look at its data type, it is a numeric, not an integer. However, as soon as we use it as an index vector in `month[idx]` it is converted to integer. Let us illustrate this by creating this strange index vector:

```
> idx <- c(1,2.5,3,2)
> month[idx]
[1] "January" "February" "March" "February"
```

Notice how our index vector `idx` now contains a non-integer, which is silly really, there is no element number 2.5. However, as soon as we use it as an index vector it is converted to integer, and this is what happens:

```
> as.integer(idx)
[1] 1 2 3 2
```

The second element is no longer 2.5, but converted to just 2. Converting a decimal number to an integer is *not* done by rounding, everything after the decimal point is simply discarded, i.e. even 2.9999 is converted to 2.

4.3.2 Logical vectors

A very common use of logicals in R programming is to create index vectors. If we use a vector of logicals instead of integers for indexing, we retrieve those elements with value `TRUE`. A small example illustrates this. We can retrieve element 2 and 3 from the `month` vector in the following way:

```
> lgic <- c(TRUE,FALSE,TRUE,FALSE)
```

and then use this as we used the index vector, `month[lgic]`. Notice that in this case the logical vector *must* have the same number of elements as the vector we are indexing, in this case 4. If the logical vector is longer than the vector we index, R will fill in `NA` in the extra positions of the resulting vector. If the logical vector is shorter than the vector we index, R will 'circulate' the logical vector. We will look into this in more detail below.

Logical vectors are typically created by performing some kind of comparison. Let us consider the numeric vector `weight` from above. Add these lines to the script:

```
is.long <- days>30 # is.long will be TRUE for elements
                   # in days larger than 30
long.month <- month[is.long]
```

save, clear workspace and re-run. The variable `long.month` should now be a vector of length 2, containing the texts `"January"` and `"March"`.

The function `which` will produce an index vector from a vector of logicals. Instead of using `is.long` directly for indexing, we could have done as follows

```
> idx <- which(is.long)
> idx
January   March
      1       3
```

Notice how `which` will return a vector with the index of all elements in `is.long` being `TRUE`. We can now use `idx` as an index vector to produce the same result as before (`month[idx]` instead of `month[is.small]`). It is in many ways easier to read the code if we use `which`. The line `which(is.long)` is almost meaningful in our own language: Which (month) is long? The answer is month number 1 and 3.

4.3.3 Vector circulation

In R vectors are often circulated when they are too short. This mean that R extends a vector implicitly, without telling us. We should be aware of this, or else it may give us some nasty surprises.

From the example above, let us specify

```
> lgic <- c(FALSE,TRUE)
```

and then use it to index `month`. This sounds difficult, since `month` has 4 elements and `lgic` only 2. But, in this case R will re-use the elements in `lgic` enough times to extend it to the proper length. Let us try:

```
> month[lgic]
[1] "February" "April"
```

and we see that element 2 and 4 is extracted. Implicitly R creates the vector `c(lgic,lgic)`, having the proper 4 elements, and use this as the logic vector.

The same applies when we use operators on vectors. Let us create two simple numeric vector, and compute their difference

```
> a <- 1:5
> b <- 1:3
> a-b
[1] 0 0 0 3 3
Warning message:
In a - b : longer object length is not a multiple of shorter
    object length
```

Notice how subtracting `b` from `a` is possible, even if it gives a warning. The vector `b` is too short really, having 3 elements, so R extends it to 5 elements as follows: `b.new<-c(b[1],b[2],b[3],b[1],b[2])` and then performs `a-b.new`.

In fact, this is also what happens when we subtract a scalar from a vector. The scalar is 'circulated' enough times to make up a vector of the proper length, and then the operator is used element by element.

4.4 Matrices

4.4.1 Creating a matrix

The other basic data structure in R is a *matrix*. It is a two-dimensional data structure, and we may think of a matrix as a vector of vectors. We create a matrix from a vector like this:

```
> avec <- 1:10
> amat <- matrix(avec,nrow=5,ncol=2,byrow=TRUE)
> amat
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
```

Since our vector `a` has 10 elements we need not specify both `nrow` and `ncol`, if we just state that `nrow=5` R will understand that `ncol` must be 2 (or vice versa). The argument `byrow` should be a logical indicating if we should fill inn the matrix row-by-row or column-by-column:

```
> amat <- matrix(avec,nrow=5,byrow=FALSE)
> amat
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
```

```
[5,]    5   10
```

We can bind together matrices using the functions `rbind` and `cbind`. If we have two matrices, `A` and `B`, with the same number of columns, say a 10×4 and a 5×4, we can use `rbind(A,B)` to produce a third matrix of dimensions 15×4. The function `rbind` binds the rows of the input matrices, i.e. put them on top of each other, and they need to have the same number of columns. Opposite, `cbind` will bind the columns, put the matrices next to each other, and they need to have the same number of rows. Here is an example:

```
> A <- matrix("A",nrow=3,ncol=2)
> B <- matrix("B",nrow=3,ncol=4)
> cbind(A,B)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,] "A"  "A"  "B"  "B"  "B"  "B"
[2,] "A"  "A"  "B"  "B"  "B"  "B"
[3,] "A"  "A"  "B"  "B"  "B"  "B"
> rbind(A,B)
Error in rbind(A, B) :
  number of columns of matrices must match (see arg 2)
```

4.4.2 Properties of matrices

Instead of a `length` a matrix has dimensions, extracted by `dim`

```
> dim(amat)
[1] 5 2
```

It always returns a vector with 2 elements, the number of rows and the number of columns.

Instead of just `names` as we had for a vector, a matrix has `colnames` and `rownames`. Again, such names are just labels we may put on columns/rows if we like.

The `class` function works just as before. Notice that all elements in a matrix must be of the same data type, just as for vectors. All basic data types can be stored in matrices, not just numbers, but in real life matrices usually contain numbers.

4.4.3 Matrix manipulation

We refer to elements in a matrix in exactly the same way as we did for vectors, but we need two indices for every value, row-number and column-number:

```
> amat[1,2]
[1] 6
> amat[2:3,1]
[1] 2 3
> amat[3,]
[1] 3 8
```

The latter is an example of how we can refer to an entire row. We specify the row number, but leave the column number unspecified. This means 'all columns' and we get the entire row. The same applies to columns, try `amat[,1]` and you should get the entire column number 1.

It is fruitful to think of a matrix as a vector of vectors, and realize that indexing is similar, only in two dimensions instead of one. It is in fact possible to refer to an element in a matrix by a single index only, just as if it was a vector. If we retrieve element number 7 in our matrix `amat`, R will count elements in the first column (5 elements) and then proceed in the second column until the index has been reached. Since `amat` has 5 rows and 2 columns we reach 7 at element `[2,2]`. This illustrates how 'vector-like' a matrix really is.

4.4.4 Operators on matrices

Again things are very similar to vectors. The rule is that operators work element by element. We can add/subtract/multiply/divide by scalars and other matrices of similar dimensions. Basic mathematical functions like `sqrt` or `log` will also work elementwise.

Matrices can be transposed, i.e. flipping row and columns, and the function for this is `t`:

```
> t(amat)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

4.4.5 What is a matrix really?

The data structure matrix can be seen as a table, but this is not really a proper description. We will see tables in later chapters, and there is a distinction, e.g. a matrix cannot contain a mixture of data types (a table can).

A matrix can also be seen as a mathematical object, and in this case it must be filled with numbers to make sense. We have functions for computing inner products, determinants and eigenvalues etc. in R, and in such cases a matrix is clearly a mathematical object.

In this text we will most often see matrices as mathematical objects, and rarely use them as 'containers' for storing data. Vectors we meet all the time, matrices only occasionally.

4.5 Important functions

Command	Remark
`c`	Constructs a vector
`seq`	Constructs a sequential vector
`rep`	Constructs a vector by repeating
`length`	Number of elements in a vector
`names`	Adds/extracts names to/from a vector
`which`	Very important! Read its Help file, we will use this function frequently
`matrix`	Constructs a matrix
`dim`	Dimensions of a matrix
`t`	Transpose of a matrix
`cbind, rbind`	Binding matrices
`plot, points`	Basic plotting functions
`mean, median, min, max, sd, var` etc.	Basic statistical functions
`sin, cos` etc.	Trigonometric functions
`rnorm, runif` etc.	Generates random numbers from some known distribution
`hist`	Creates histograms
`sample`	Random sampling from some vector
`paste`	Pasting multiple texts into one text

4.6 Exercises

4.6.1 Sequential vectors

Create a vector called `x` that takes the values from 0.0 to 10.0 with steps of 0.1 (use the `seq` function). Then compute the sine (sinus) of `x` using the function `sin` (see `?sin`) and store this in `y`. Plot `y` against `x` using the basic plotting function `plot`. Try `plot(x,y,type="l",col="blue")`. Read about `plot`.

Extend the script by also reversing the order of the elements of `y`. Use an index vector to do this. Make a new plot with `y` in reversed order.

Extend the script by making new variables `x3` and `y3` that contains every third element of `x` and `y`, respectively. Then plot above, but add the line `points (x3,y3,pch=16,col="red")`. Read about `points`.

4.6.2 Vector computations

There are many functions in R that takes a vector as input and gives a single variable as result. In statistics we often make use of the following functions: `mean, median, sd, min, max, sum`. Read the help files of these functions. Use them on the vector `y` from above, and verify that they give reasonable results.

4.6.3 More vector computations

Many functions in R take a vector as input and returns a new vector of the same length as output. The function `sin` from above is an example. Other mathematical functions we use often are `sqrt, abs, log`. Read the help files for these functions. Use these functions on the `y` vector from the first exercise.

4.6.4 Normal distributed random numbers

R have many functions to generate random numbers. The function `rnorm` produces independent normal distributed values. The command `rnorm(10,mean=0, sd=1)` will produce a vector of 10 values sampled from the normal distribution with mean value 0 and standard deviation 1. Make a script that samples 100 such values and put them into a vector `v`. Use the function `hist` to plot a histogram and verify that they have an approximate normal distribution. To indicate that they are independent, plot each value against its following value. Use the `plot` function, and plot `v[1:99]` against `v[2:100]`.

There are many similar functions in R, producing random numbers from various known distributions. Try to generate numbers from `rt, rf, runif, rchisq` and make the same plots.

4.6.5 Random sampling

We also have a function `sample` that we can use to sample at random from a vector. Perhaps the simplest random variable is the binomial. The binomial variable is the number of ’successes’ on n independent trials, where each trial has two possible outcomes, ’success’ or ’failure’. We can think of it as flipping a coin and count the number of heads (or tails). Make a vector `coin` with two elements, the texts `"Head"` and `"Tail"`. Then, make a vector `coin.flips` that samples at random 100 times from the `coin` vector. Hint: Use `replace=TRUE` as an option in `sample`.

Count the number of `"Head"` in `coin.flip`. The procedure for doing this can be sketched as follows:

1. Create a logical vector `is.head` with one element for each element in `coin .flip`, and such that `is.head` is `TRUE` where `coin.flip` is `"Head"`.
2. Convert `is.head` to numeric. Remember how logicals are converted to numeric, put the result in some vector `x`.
3. Sum the elements of `x`.

Implement this, and make certain you understand what goes on in each step. The actual code to do this can be reduced to one very short line, but start by doing it step by step to understand what goes on.

4.6.6 Matrix manipulations

Create a 10×20 matrix (10 rows and 20 columns) with only 1 in the first row, 2 in the second,..., 10 in the last row. Use, for instance, commands like `rep` and `rbind` or `matrix` to achieve this. Check help files if necessary. Use the command `image` to get a graphical overview of the matrix. Then, create another matrix of the same dimensions, but with 1 in the first column,2 in the second,...,20 in the last column.

4.6.7 Special matrices

Create a 5×5 identity matrix, i.e. a matrix with 1 on the main diagonal and 0 elsewhere. You may use the function `diag`. This function works in several ways, read about it in the help files. For instance, check what happens for:

```
> A <- diag(5)
> diag(A)
> diag(diag(A))
```

Chapter 5

More data structures

5.1 The data.frame

In the previous chapter we introduced matrices as a two-dimensional data structure, but we also mentioned that a matrix should in general *not* be seen as a data table. The proper data structure for this is called a data.frame in R.

5.1.1 Creating a data.frame

We can construct data.frames in many ways. Here is a start of another script called `script_tables.R`:

```
# Creating a data.frame
monthdata <- data.frame(Name=c("Jan","Feb","Mar","Apr"),
                        Days=c(31,28,31,30))
```

Type `> monthdata` in the console and return, or click the names `monthdata` in the Workspace window to open a display of the data.frame in RStudio. It looks like a matrix, but notice how we can have texts in the first column and number in the second. This was not possible in a matrix. In a data.frame all elements in the same column must be of the same data type, but different columns can have different data types. This column-orientation reflects the data table convention used in all statistical literature: The columns of the data table are the variables, and rows are the samples. Note that it is not required to have a line-break between the column-specifications as above, it is just added for convenience here.

We gave names to the columns directly in the above example. Names on the columns are not required, but is of course convenient, and we should make efforts to always have informative names on the columns of a data.frame. The function `colnames` that we used for matrices works just as fine on data.frames, in case you would like to add/change names on the columns. The same applies to `rownames`, but quite often the rows are just numbered and any description of the samples are included as separate columns inside the data.frame.

We can also add new rows or columns to an existing data.frame, here we add a row:

```
# Adding a new row
monthdata <- rbind(monthdata,data.frame(Name="May",Days=31))
```

Notice how we create a new data.frame on the fly here, having a single row, but the same columns as the old `monthdata`, and then use `rbind` to 'glue' it below the existing `monthdata` rows, producing a new, extended, data.frame.

New columns can be added in a similar way, but perhaps more common is this approach: We want to add a column indicating the season to our existing `fishdata`, and do it simply by

```
# Adding a new column
monthdata$Season <- c("Winter","Winter","Spring","Spring","
    Spring")
```

Here we create a character vector using the `c` function again, and assign this to the column named `Season` in `monthdata`. Since this column does not exist, it is created on the fly as an extra column. This is a short and simple way of adding a column and giving it a name at the same time.

5.1.2 Manipulating data.frames

We use the same indexing for data.frames as we did for matrices. We can index single elements, subsets or entire rows/columns. Remember that a data.frame is column oriented. Each column is a vector, but a row of a data.frame can be a mixture of various data types. If we retrieve a column, say `monthdata[,2]`, this will be a vector. If we retrieve a row, say `monthdata[2,]`, this will be a single-row data.frame, not a vector. Remember this distinction between rows and columns in data.frames.

Since columns very often have names, we can also refer to a column by its name:

```
> w <- monthdata$Days
```

where the statement to the right of the assignment is an alternative to `monthdata[,2]`. Use the `class` function to verify that `w` is now a numeric vector.

Here we see the important operator `$` for the first time. This operator means we refer to the variable called `Days` which is found inside the variable `monthdata`. This correponds to the dot-notation in many other programming languages. We will see more use of `$` later.

From time to time we would like to convert a data.frame to a matrix. Remember that a matrix can only contain a single data type. Since most data.frames will contain different data types, we usually only convert a part of the data.frame to a matrix. Here is an example:

```
> monthmat <- as.matrix(monthdata[,c(1,3)])
> monthmat
  Name    Season
1 "Jan"   "Winter"
```

```
2 "Feb"   "Winter"
3 "Mar"   "Spring"
4 "Apr"   "Spring"
5 "May"   "Spring"
```

Here we extracted the text columns 1 and 3 of `monthdata`, and converted them to a matrix (verify that `monthmat` is a matrix). Notice how the column names were inherited. The statement `monthdata[,c(1,3)]` produces a new data.frame consisting of the two specified columns:

```
> class(monthdata[,c(1,3)])
[1] "data.frame"
```

Notice that when we retrieve a single column, the result is a vector (see the example with `w` from above), but if we retrieve two or more columns it is still a data.frame. Can you imagine why?

We can also convert a matrix to a data.frame, which is always straightforward:

```
> atable <- as.data.frame(monthmat)
> class(atable)
[1] "data.frame"
```

Why convert between data.frame and matrix? A data.frame is a *container* where you store your data, and typically we extract the numeric parts of the data.frame and convert these to a matrix to do some *computations*. Some computations can be done directly on the data.frame, but more extensive modeling and analysis often requires the matrix format.

The functions `cbind` and `rbind` from the previous chapter can also be used on a data.frame and works as for a matrix. In the coming chapters we will see some other functions we can use to manipulate data.frames.

5.1.3 Text as factors

If we look at the data type of the first column of `monthdata`, we find it is no longer a text like we entered:

```
> class(monthdata[,1])
[1] "factor"
```

Since R has been made by statisticians, they have decided that text entered in a data table should probably be used as a factor in the end, and the default behavior of the function `data.frame` is to convert all texts to factors. Personally, I rate this as a design failure. Text should be text, and we should convert it to a factor when we need to. However, it is possible to override the default behavior. We can either add the argument `stringsAsFactors=FALSE` to the `data.frame` command during creation, or we can state

```
options(stringsAsFactors=FALSE)
```

in the console or at the top of our script, and the default conversion to factors is turned off for the entire R session. Note the capital letters in `stringsAsFactors`!

Put the above statement at the top of the script, and re-run.

5.1.4 Properties of data.frames

We can use the `dim` function on a data.frame just as we did on a matrix, and the result is a vector of two numbers indicating the number of rows and the number of columns of the data.frame. We can also use the `length` function on a data.frame, it will return the number of columns, same as the second value returned by `dim`.

5.1.5 Operators on data.frames

A data.frame is meant to be a container, and we rarely use it as input in computations. You cannot in general *not* use the arithmetic operators, e.g. adding a number, to a data.frame. The reason is of course that it may contain non-numeric data.

You can transpose a data.frame! This is silly really, as it is not a mathematical object. Try to transpose the `monthdata` data.frame and observe what happens.

5.1.6 The `table` command

A data.frame can be seen as a table of data. There is also a function named `table` in R, but this has nothing to do with data.frames! Instead, `table` will take a vector of discrete elements (e.g. integers or texts) and count the number of occurrences of each element. We will see the use of this function in some exercises, but remember it has nothing to do with data.frames.

5.2 Lists

A list is the most general data structure we will look into here. Briefly, a list is a linear structure, not unlike a vector, but in each element we can store any other data structure.

5.2.1 Creating a list

A list is created like this:

```
> a.list <- list(2,              # a single number
                 "Hello",        # a text
                 c(1,1,1,2,3),# a vector
                 list("STIN300",5,TRUE)# another list
                 )
```

which is very similar to how we created vectors, except that we use `list` instead of `c` before the comma-separated listing of elements.

Here we created a list with 4 elements. The first element contains the numeric `2`, the second the character `"Hello"`, the third a numeric vector and the fourth element contains another list. Thus, any list element can contain any data structure irrespective of the other list elements. This makes lists extremely flexible containers of data. If we have several variables, of different types, that in one way or another belong together, we can bundle them all into the same list.

The list elements can be given names, just like the columns of a data.frame (actually, a data.frame is just a special list). Let us specify names:

```
> a.list <- list(Value=2,          # a single number
                 Word="Hello",     # a text
                 Vector=c(1,1,1,2,3),# a vector
                 List2=list("STIN300",5,TRUE)# another list
                 )
```

where the 4 list elements now have the names `Value`,`Word`,`Vector` and `List2`. Just as for data.frames we should always try to use informative names on list elements. We are free to choose names as we like, the convention of having a capital first letter is just a personal habit I have inherited from Java programming.

5.2.2 Manipulating lists

A list can be indexed similar to vectors, but there are some complications. If we state

```
> a.list[2]
[[1]]
[1] "Hello"
```

we do *not* refer to the content of element 2 in the list. Instead, we refer to element 2 itself, which is a list of length 1, and inside this single-element list we find the text `"Hello"`. In order to refer to the *content* of element 2 we must write

```
> a.list[[2]]
[1] "Hello"
```

i.e. we must use the double brackets. The distinction between a list element (which is itself a single-element list) and its content is important to be aware of.

A list cannot take an index vector in the same elegant way as vectors. We can refer to a sub-list of the three first elements by `a.list[1:3]`, but we cannot write

```
> a.list[[1:3]]
Error in alist[[1:3]] : recursive indexing failed at level 2
```

If we want to retrieve the contents of all list elements, we simply have to loop through the list and retrieve them one by one. This means lists are less suited for doing heavy computations, their use is as containers of data.

If the list elements have names we refer to the content of a list element directly by using its name:

```
> a.list$Word
[1] "Hello"
```

The nice thing about using the name is that we do not need to keep track of the numbering of the elements. As long as we know there is an element named `Word`, we can retrieve it no matter where in the list it occurs.

5.2.3 Properties of lists

A list has a length, and the `length()` function works properly for a list.

5.3 Arrays

An array is simply an extension of a matrix to more than 2 dimensions. We can see it as a vector of vectors of vectors... We can create a 3-dimensional array like this:

```
> a.box <- array(1:30,dim=c(2,3,5))
```

If you type `a.box` in the console window you will see it is displayed as 5 matrices, each having dimensions 2×3.

Arrays are very similar to matrices, just having more indices. We refer to the elements just like for matrices, e.g. `a.box[2,2,4]` refers to the element in row 2, column 2 of matrix 4 in the array. We don't use arrays very often, but some functions may produce them as output.

5.4 Important functions

Command	Remark
`data.frame`	Constructs a data.frame
`colnames`, `rownames`	Adds or extracts column/row names for a matrix or data.frame
`list`	Constructs a list
`unlist`	Extracts the content of a list and store it in a vector, if possible (dangerous)
`array`	Constructs an array
`cor`	Sample correlation
`colMeans, rowMeans, colSums, rowSums`	Self explanatory...
`options`	Setting global options
`attach`	Making a data.frame an environment
`table`	Tabulates data

5.5 Exercises

5.5.1 Lists

Make a list containing the month-data from the text. Each list element should contain the data for one month. Use only the `Name` and `Days` data. The list should have length 5. Add the following single-element list to this list: `list(Name="June",Days=30)`. Verify that the extended list has 6 elements.

Make another list where the first element has a vector containing the number 3 three times, the second element has the number 2 two times and the last element has the number 1 once. Use the function `unlist` on this list. What happens? Try `unlist` on the month-data list as well.

5.5.2 Weather data

Load the content of the file `daily.weather.RData` into R. This is done by the statement `load("daily.weather.RData")`. This loads the data.frame `daily.weather`. It contains daily measurements of some weather variables from Ås in the period 1988 to 2013. When referring to columns in this data.frame you either specify it using the dollar-notation (see text above), or you can 'attach' the data.frame and then refer to the columns directly. Here we try out the latter: Attach the data.frame by `attach(daily.weather)`. Now you can refer to Air.temp or any other column as if they were vectors in the Global Environment (in fact, `daily.weather` will show up as an environment in RStudio).

Compute the difference between Air.temp.max and Air.temp.min for each day, and store this in a new vector. Make a histogram of these values (use `hist`). What was the largest and smallest daily temperature differences? Make a plot of this daily difference against Air.temp. Is there a relation between the two?

Compute the mean Humidity (use `mean`). Here we see a very common problem in large data sets: Some values are missing, indicated by `NA` (Not Available). When a vector contains one or more `NA`, the mean of this vector is also `NA`. Read the help file for `mean` and find out how you can tell this function to ignore `NA`'s, and compute the mean from the remaining values. Similar behavior is found for many other R functions as well.

The last column of this data.frame is the only one that does not contain numeric data. This is wind direction specified by a text. The function `table` is very convenient for categorical data like this. Run `table(Wind.dir)` and figure out how `table` works.

Next, we will extract a subset of these data. Columns 4 to 15 are numeric measurements, extract these data for all January days, and convert this into a matrix. The columns are straightforward since they can be specified directly as the index vector `4:15`. The rows are more difficult since every year has 31 January days. A very convenient function in this respect is `which` that we saw in thee previous chapter. Here we sketch the procedure:

1. Create a logical vector of the same length as Month and with `TRUE` on all January days.
2. Use this vector as input to `which`
3. The output from `which` is the index vector to specify the rows.

Compute the mean of every column of the matrix, using `colMeans` (read its Help file).

5.5.3 Climate data

Load the file `Tmonthly.RData` into R. This is the longest series of monthly temperature measurements in Norway, taken here at Ås since 1874. It is stored as a data.frame with the year in the first column and the monthly temperatures in the remaining 12 columns. Extract the 12 columns of temperature and convert it to a matrix. Next, put these data into a vector, and make certain the data are in chronological order. HINT: A matrix `M` is converted to a vector with `as.vector(M)`, but this function will concatenate the columns of `M`. We want to concatenate the rows, and this is achieved by transposing `M` first. Plot the monthly temperatures, and use the option `type="l"` in the `plot` function to get a curve. Can you visually see any indication of global warming?

Chapter 6

Input and output

6.1 Input and output of formatted text

By formatted text we mean data with a 'rectangle-like' look. Data are arranged in columns, and there are the same number of data in every column. There is a single separator symbol between each column (e.g. a comma, semicolon, tab, space, etc.). These are files typically produced by saving data in a spreadsheet into a text file, e.g. exporting as .csv file from Excel.

6.1.1 Reading formatted text

The command `read.table` (or its close relatives `read.csv`, `read.delim` etc.) is used to import the content of a formatted text file into the R memory. Download the file `bears.txt` from the course website (Fronter) and store it in your `RHOME` folder. Create a new script and enter the following lines:

```r
# Reading data from formatted file. Data columns are
# separated by a single space and each column has a
# name (header).
beardata <- read.table("bears.txt",sep=" ",header=T,
                       stringsAsFactors=F)
```

Save the script under the name `script_bears.R` (or choose your own file name). Clear the memory and run the script. In the Workspace window of RStudio you should now see the `beardata` variable, which is a data.frame containing 24 observations (samples) of 11 variables. If you type `beardata` in the console you get the data set listed. Alternatively, you can click on the `beardata` variable in the Workspace window, and RStudio will display the data in a nicer way in the editor.

The first and basic input to `read.table` is the name of the file you want to read. If this file is in a different folder than your R session you need to specify the entire path of that file, e.g. if the file was in a subfolder called `data` we would have to enter `"data/bears.txt"` as the first input. Notice that the file separator is `/` in R (just like in UNIX), not `\`, even if you run R under Windows.

Additional inputs to `read.table` are optional, and see `?read.table` for a full description. The argument `sep` indicates how the columns are separated in the file. Here it is just a single blank, but other separators may be used. If the columns are tab-separated we must specify `sep="\t"` which is the symbol for a tab. The logical `header` indicates if the first row should be seen as headers, i.e. column names.

If you have specified `options(stringsAsFactors=F)` before you read the data, you don't need this argument here. The `read.table` function also have other ways of overriding the default conversion of texts to factors. The arguments `as.is` can be used to specify which columns *not* to convert to factor, while `colClasses` can be used to specify the data type of every column. Read about them in the help files.

6.1.2 Formatted output

Once you have a data.frame in R you can output this using the `write.table` function. This will produce a text file similar to the file `bears.txt` that we read above:

```
> write.table(beardata,file="bears_copy.txt",sep="\t",
              row.names=F)
```

We need to first specify the data.frame to output, then the file to put it into. After that we have a number of options we may use if we like. Here we specify that columns should be tab-separated (`sep="\t"`) and that no row numbers should be output. The `write.table` is in many ways quite similar (inverse) to `read.table`, see help files for details.

6.2 Internal format

Once you have created some variables in an R session, you may want to save this for later. Often it will be cumbersome to save everything as formatted text files. Instead, you may use the `save` function. If we want to save the variables `a`, `b` and `txt` we write

```
> save(a,b,txt,file="dataset1.RData")
```

Notice that we first list all variables we want to save, and then finally `file=` and the name of the file to create. It is a convention that these type of files should have the `.RData` extension.

Files saved from R like this are binary files and can only be read by R. If you want to send data to other people in such files, you must make certain they have R and know how to read such files.

Reading these files is very easy:

```
load("dataset1.RData")
```

will restore the variables `a`, `b` and `txt` in the memory of R.

6.3 Unformatted text files

Sometimes we need to read data without a format that `read.table` can handle. In this case we need to revert to a lower level approach to reading files.

6.3.1 Connections

A connection we can think of as a *pipe* that R sets up between its memory and some data source, typically a file or a URL on the internet. There are several functions for creating such connections, see `?connections`. Once a connection has been set up, we open it, read or write data, end finally close it:

```
> conn <- file("influenza.gbk",open="rt")
> # here we can do some reading...
> close(conn)
```

In the first line we create a connection using the function `file`. This function opens a connection to a file (surprise, surprise...). The first argument is therefore the file name, in this case `"influenza.gbk"`. The second argument opens the connection. This argument takes a text describing the *mode* of the connection. The string `"rt"` means read text, i.e. this pipe can only be used for reading (not writing) and it will only read text. There are several other functions besides `file` that creates and opens connections, but we will only focus on `file` and `url` here.

6.3.2 Reading line by line

Once we have established a connection ready for reading text, we can use the function `readLines` for reading text line by line from the source. Having the file `influenza.gbk` in the `RHOME` folder, we can make the following short script for reading it:

```
conn <- file("influenza.gbk",open="rt")
lines <- readLines(conn)
close(conn)
```

This should result in the entire file being read, line by line, into the variable `lines`. This variable is now a character vector, with one element for each line of text. Retrieving the relevant information from these text lines is a problem we will look at later, when we talk more specifically about handling texts.

Instead of reading from a file, we can also read directly from the internet. The following code retrieves the document from the Nucleotide database at NCBI for the same file that we read above:

```
co <- url("http://www.ncbi.nlm.nih.gov/nuccore/CY134463.1",
          open="rt")
lines <- readLines(co)
close(co)
```

In this case the variable `lines` contains all codes needed to format the web-page in addition to the text we saw in the file.

6.3.3 Writing line by line

The `writeLines` function writes a character vector through a connection, i.e. it is the inverse of `readLines`, and its use is very similar. See `?writeLines` for details.

6.3.4 Reading huge files

Sometimes we may have to read files so large that we do not have enough memory to read the entire file into one variable. In modern science data files become larger and larger, and files of many gigabytes are not uncommon. In `readLines` we can read only a certain number of lines at a time, and in this way we can split a large file into several pieces. As long as the connection remains open, `readLines` will remember where it ended reading, and once we call `readLines` again it will continue to read the following lines. In this way we can read a chunk of text, retrieve whatever we are looking for, and then continue reading the next chunk, and so on without reading the entire file into memory. The same principle applies to `writeLines`, i.e. we can output to files chunk by chunk if we like.

6.4 Important functions

Command	Remark
`read.table, write.table`	Reads/writes formatted text
`save, load`	Saves/loads data in R format
`file, url` etc.	Open connections for reading or writing
`close`	Closes a connection
`readLines, writeLines`	Reads/writes line-by-line from/to text files
`summary`	Gives a summary of a data.frame (etc.)
`sum`	Sums the elements in a vector or matrix

6.5 Exercises

6.5.1 Formatted text

Download the file `bears.txt` from Fronter and save it. Make a script that reads this file into R. Note that if you save the file in a directory different from your working directory (quite common) you need to specify the path in the file name, e.g. `read.table("data/bears.txt")` if the file is in the subdirectory `data`.

Use the function `summary` to get an overview of this data set. Read the Help-file for `summary`. When we are looking at a new data set, we usually want to make some plotting before we proceed with more specific analyses. We will look into plotting in more detail later, but we will take a 'sneek-peak' at some possibilities right away.

The first 8 columns of the data.frame you get by reading `bears.txt` are numerical observations from a set of 24 bears. We can quickly make pairwise plots of each of these variables against every other variable. Do this by `plot( beardata[,1:8])` (here I have assumed `beardata` is the variable name). Which columns correlate?

Use `boxplot(beardata[,1:8])` to make a plot showing much of the information from `summary` above graphically. Read about boxplots.

The column `Gender` indicates which bears are male and female. We can use this to look at the distribution of `Weight` between genders. Make a boxplot of `Weight` for males and females separately by `boxplot(Weight~Gender,data=beardata)`. Here we see the use of the operator `~`. It is related to what we call *formulas* in R, and we will see this again later. NOTE: Here we first specified the column names only, and then stated which `data.frame` to look for these columns. An alternative statement would be `boxplot(beardata$Weight~beardata$Gender)`. Here we use the `$` operator to specify directly which data.frame we are using.

Let us perform a two-sample t-test for the difference between male and female weights of bears. We use the function `t.test`. First, make an index vector describing which rows of the data.frame correspond to male bears. Use this to extract male weights, and store these in the vector `mw`. Then, do the same for female bears, and store their weights in `fw`. Finally, perform the t-test by `t.test(mw,fw)`. What does this test tell you?

6.5.2 Extracting data from text files

Download the file `influenza.gbk` from Fronter and save it. This a plain text file with sequence data in a format called the GenBank format. You can open the file in any standard text editor and have a look. Obviously, this file cannot be read by `read.table`.

Make a script that reads this file into R line by line. After reading the file the script should output how many lines were read.

This file contains information about a number of influenza genes, one 'chunk' of text for each separated by a line containing just `//` (double slash). Make the script output how many genes there are information about. HINT: Use a comparison to produce a vector of logicals (`TRUE` or `FALSE`), remember how logicals can be converted to numerics, and use the function `sum` (see exercise 4.6.5).

If you take a look at the file, you will notice that in the left margin you find some standard 'tags', i.e. keywords of the GenBank format. One of them is SOURCE (in upper case). We would like to retrieve all the SOURCE-lines. This can be achieved by the use of `grep`. This function searches a text for a *regular expression*. We will come back to this later, but for now we can just think of it as a text search. Use `grep` to find all the SOURCE-lines in the file, retrieve them and store these in a vector.

Chapter 7

Control structures

7.1 Loops

Fundamental to all programming is the ability to make programs repeat some tasks under varying conditions, known as looping. R has these basic facilities as well. The short words written in blue below are known as *keywords* in the language, and we are *not* allowed to use these words as variable names. They usually appear in blue font in RStudio, hence the coloring here.

7.1.1 The `for` loop

This is the type of loop we use when we *a priori* know how many repetitions we need. We typically use a `for` loop when we want to visit all elements in some data structure. Let us consider a list called `alist`, and retrieve the content of the list elements one by one:

```
for(i in 1:length(alist)){
  x <- alist[[i]]
  #...do something with x...
}
```

Here we specify that the variable `i` should take on the values `1,2,...length(alist)`, and that for each of these values the code between `{` and `}` should be repeated. The keywords here are `for` and `in`.

The 'skeleton' of a `for` loop looks like this: `for(var in vec){}`. The variable named `var` here is the *looping variable.* This variable is created by the `for` statement, and takes on new values for each iteration. We often use very short names for the counting variable, e.g. `i`, `j` etc., but any name could be used. The values it will take are specified in the vector `vec`. The length of the vector `vec` determines the number of iterations the loop will make. Any vector can be used here, but most often we construct a vector on-the-fly just as we did in the example above (remember `1:length(alist)` will create a systematic vector).

Notice how already before we start the loop, we know there will be need for exactly `length(alist)` iterations. The `for` loop is the most common type of loops in R programs.

7.1.2 The `while` loop

Sometimes we want a loop to run until some condition has been met, and we do not know exactly how many iterations it will take to achieve this. In such cases we use a `while` loop. Here is an example:

```
x <- 1
while(is.finite(x)){
  x <- x*2
  cat( "x=", x, "\n", sep="")
}
```

Here we give the value `1` to the variable `x`. Then, we start a loop where we inside the loop let the new value of `x` be the old value times 2. We also make a print of `x` in the console window (the `cat` function). The `while` loop has in the first line a *test*. After the keyword `while` we have some parentheses, and inside them is something that produces a logical, `TRUE` or `FALSE`. As long as we produce a `TRUE` the loop goes on. Once we see a `FALSE` it halts. The function `is.finite` takes a numeric as input and outputs a logical `TRUE` if the input is finite, or `FALSE` if it is infinite. Run these lines of code, it gives you an impression of what R think is 'infinite'...

Notice that in the test of the `while` above we have to involve something that changes inside the loop. Here the testing involves `x`, and `x` changes value at each iteration of the loop. If we wrote `y <- x*2` instead of `x <- x*2` inside the loop, it would go on forever because `x` no longer changes value! This is a common error when people make use of `while` loops. Such never-ending loops can be terminated by Session-Interrupt R in the RStudio file menu.

7.1.3 Avoiding loops

Looping is unavoidable in all except the simplest of programs. However, making explicit loops using either `for` or `while` loops is something we should try to avoid in R. The reason is speed. Loops run extremely slow in R compared to in lower level programming languages. Thus, if you want your programs to be efficient, you should learn how to compute without using the explicit loops. This is achieved by either using vectorized code or by the use of built-in functions where the looping has been compiled to run much faster. Here are some examples of what we talk about.

Vectorized code

Create two numeric vectors, `a` and `b`, each having 100 000 elements. Then compute `c` as the sum of `a` and `b`. We have learnt that as long as vectors have identical length, we can compute this as

```
c <- a+b
```

Compare this to the looping solution:

```
c <- numeric(length(a)) # creates numeric vector of same
    length as a
```

```
for(i in 1:length(c)){
  c[i] <- a[i]+b[i]
}
```

On my computer the latter took 390 times longer to compute! The concept of vectorized code simply means we use operations directly on vectors (or other data structures) as far as possible, and try to avoid making loops traversing through the data structure. This is often possible, but not always. If you want to do something to the contents of a list you have to loop, there is no way to 'vectorize' a list.

Using built-in looping functions

Assume we want to compute the standard deviation for every column of a big numeric matrix `A`. We can of course make a `for` loop, and consider one column at the time, using `A[,i]` as input to the function `sd`, where `i` is the looping variable:

```
sd_columns <- numeric(dim(A)[2])
for(i in 1:dim(A)[2]){
  sd_column[i] <- sd(A[,i])
}
```

The alternative approach is to use the function `apply`. The statement

```
sd_columns <- apply(A,2,sd)
```

will do exactly the same looping, but faster since the 'hard labour' of the looping is done by some compiled code inside this function. As you can see, the code is also much shorter.

However, sometimes we deliberately use explicit loops also in R programs. Programs are usually easier to read for non-experts when they contain loops. The gain in speed is sometimes so small that we are willing to sacrifice this for more readable code. It should also be said that for smaller problems, where loops run only over a smallish number of iterations, the speed consideration can be ignored altogether. The avoiding of loops is most important when we build functions, because these may themselves be used inside other loops, and the speed issue becomes a more real problem.

7.2 Conditionals

As soon as we have loops, we will need some conditionals. Conditionals means we can let our program behave differently depending on what has happened previously in the program.

7.2.1 The `if` statement

The basic conditional is the `if` statement. Here is an example:

```
for(i in 1:10){
  cat("i=", i, sep="")
  if(i>5){
    cat(" which is larger than 5!")
  }
  cat("\n")
}
```

The loop runs with `i` taking the values, `1,2,...,10`. Inside the loop we have an `if` statement. There is a test `if(i>5)` where the content of the parentheses must produce a logical. If this logical has the value `TRUE`, the content of the statement (between `{` and `}`) will be executed. If it is `FALSE` the code between `{` and `}` will be skipped. Note that all code outside the `if` statement is always executed, independent of the test. Only the code between the curly braces are affected.

Most `if` statements are seen inside loops, where the execution of the loop depends on some status that changes as the loop goes on. But, we can put `if` statements anywhere we like in our programs.

7.2.2 The `else` branch

In the basic `if` statement above we either executed the conditional code or skipped it. This means whatever code comes after the `}` of the `if` statement will always be executed regardless of the test. We can extend our conditional statement to a `if`-`else` statement if we want to execute *either* one chunk of code *or* another chunk of code

```
for(i in 1:10){
  cat("i=", i, sep="")
  if(i>5){
    cat(" which is larger than 5!")
  } else {
    cat(" which is not larger than 5!")
  }
  cat("\n")
}
```

The only difference is that this statement has two sets of braces. The code inside the first set is executed if the test is `TRUE`, and the other if it is `FALSE`.

It is possible to extend the `else` branch with another `if`, like this:

```
if(test1){
  # code here
} else if(test2){
  # code here
} else {
  # code here
}
```

and this can be extended in as many cases as you need. Often a branching like this can be better handled by a `switch` statement, see below. Note the space between `else` and the following `if`, there is no keyword called `elseif` in R!

7.2.3 The `switch` statement

In an `if` statement we consider binary situations, where a test comes out as either `TRUE` or `FALSE`. Sometimes we want to execute different code depending on a range of possible outcomes. In these cases we may use the `switch` statement:

```
for(i in seq(from=0.1,to=5,by=0.1)){
  cat("i=", i, sep="")
  switch( EXPR=ceiling(i),
  cat("...the loop has just started...\n"),
  cat("...we are on our way...\n"),
  cat("...halfway there...\n"),
  cat("...we are approaching the end...\n"),
  cat("...almost done...\n") )
}
cat("...done!\n")
```

The first argument of the `switch` (the `EXPR=ceiling(i)`) is an expression that evaluates to an integer 1,2,...5. The function `ceiling` will round up to the smallest integer larger than `i`. After this expression we have 5 lines of code separated by a comma. The integer that comes out as `EXPR` decides which of these 5 code-lines will be executed.

The first expression of the `switch` must either result in an integer or a text. If it is a text, the comma-separated statement that follows in the `switch` must have a *label* that indicates which text belongs to which statement. Here is an example:

```
names <- c("Solve","Hilde","Lars","Guro","Trygve")
for(n in names){
  age <- switch(EXPR=n,
                Lars   =46,
                Solve  =40,
                Trygve =27,
                        "hidden")
  cat( age, "\n" )
}
```

The loop has a text `n` as the looping variable. This takes on the different values in `names`, hence the loop runs 5 iterations. Inside the loop we assign values to the variable `age`. The `EXPR` of the `switch` is a text, one of the names. The three following code-lines are each identified by a label (e.g. `Lars`), and `switch` will try to match the text in `n` to one of these labels. If a specific match is found, the corresponding line of code is executed. Note that these code-lines do nothing but assign a value to the variable `age`. If no match is found, the last line of code, with no label, is the one to be executed. This is the default choice when no specific match is found.

7.2.4 Short conditional assignment

Sometimes we want to assign to a variable one out of two possible values depending on some condition. This could be solved by the `if-else` statements as follows:

```
if(test){
  var1 <- x
} else {
  var1 <- y
}
```

Thus, after this statement the variable `var1` has either value `x` or `y`. Since this type of conditional assignments occurs quite frequently in programs, there is a short-version for writing this kind of statements in R:

```
var1 <- ifelse(test,x,y)
```

It means `var1` is assigned the value `x` if the `test` comes out as `TRUE` and `y` if it comes out as `FALSE`. Notice that the function name is `ifelse` without any space inside.

7.3 Logical operators

We see that logicals play a central role in looping and testing, and this is a proper place to look closer at logical variables.

Logicals typically occur from some kind of comparison, and we have seen the standard comparison operators previously: `>`, `<`, `>=`, `<=`, `==` and `!=`. Let us for now assume that we only compare numerical quantities, we will come back to the comparison of texts later. We have seen how these operators can be used to compare vectors, i.e. a vector can be compared against another vector, producing a logical vector. Let us consider a simple example:

```
a <- 11:15
b <- rep(13,times=5)
c <- rep(12,times=5)
lgic1 <- a>b
lgic2 <- a>c
```

The comparison in the last two lines are straightforward since both `a`, `b` and `c` have 5 elements, and the result is in each case a logical vector of 5 elements. Make a script and run this example.

The *logical operators* are operators used on logical variables. The three operators we will look into here are `&`, `|` and `!`. The first operator is called AND, and combines two logical variables into a new logical variable. If we consider the vectors from the example above we can write

```
lgic3 <- lgic1&lgic2
```

and the vector `lgic3` will now have 5 elements where any element is `TRUE` if and only if the corresponding elements in `lgic1` and `lgic2` are both `TRUE`. In all other cases it will have the value `FALSE`. You can combine as many logical variables as you like this way (with `&` between each) and the result will be `TRUE` only for elements where all input variables are `TRUE`.

The second operator is OR, and

```
lgic3 <- lgic1|lgic2
```

will now result in `lgic3` being `TRUE` if *either* `lgic1` *or* `lgic2` is `TRUE`. Again you can combine many variables in this way.

The last operator is NOT. Unlike the other two, this is a unary operator (the others are binary), meaning it involves only one variable, and `!lgic1` means that we flip `TRUE` and `FALSE`, i.e. any elements originally `TRUE` become `FALSE` and vice versa.

We can combine the operators into making more complex logical expressions. What does `lgic3` look like now:

```
lgic3 <- (lgic1|!lgic2)&lgic2
```

As usual, expressions inside parentheses are evaluated first. Sort it out in your head before you run the code and see the answer.

Logical operators are very common, and you should make certain you understand how they operate.

7.4 Stopping loops

7.4.1 The keyword `break`

The keyword `break` can be used to stop a loop if some condition occurs that makes the continued looping senseless. Then the only reasonable solution may be to stop the looping, perhaps issue some warning, and proceed with the rest of the program. This can be done by the `break` statement:

```
# some code up here...
for(i in 1:N){
  # more code here...
  if( test ){
    break
  }
  # more code here...
}
# some code down here...
```

Here we see that inside the loop is an `if` statement, and if the `test` becomes `TRUE` we perform a `break`. This results in the loop terminating, and the program continues with the code after the loop.

We do not use the `break` very often. It is in most cases possible to use a `while` loop in those cases where we can think of using a `break`.

7.4.2 Errors

If we want to issue a warning to the console window, we can use the function `warning` taking the text to display as input. Notice that a warning is just a warning: It does not stop the program from continuing.

If a critical situation occurs, and we have to terminate the program, we use the function `stop`. This function also takes a text as input, the text we feel is proper to display to explain why the program stops. We could use `stop` instead of `break` in the example above. This would result in not only stopping the loop, but stopping the entire program from running.

For those familiar with other programming languages, the *exception handling* might be a familiar concept. This is a way to deal with errors in a better way than just give up and terminate the program. In R we can also handle exceptions, but this is beyond the scope of this text.

7.5 Important functions

Command	Remark
`is.finite`	Is used to look for `inf`
`numeric`, `character`, etc.	Constructs vectors of various data types
`apply`	Applies a supplied function to the rows/columns of a matrix
`stop`	Stops execution
`warning`	Produces a warning, but execution is not stopped

7.6 Exercises

Simulation is much used in statistics, and it can be a nice way to illustrate and analyze certain problems. There is a range of methods called Monte Carlo methods in statistics, and they all rely on simulation of some kind. In order to simulate we need 1) a way to sample random numbers, and 2) looping to repeat this many many times. We will have a first look at stochastic simulation in the following exercises.

7.6.1 The central limit theorem

The central limit theorem is one of the fundamental mathematical results in statistics. We will illustrate its result by simulation in R. The theorem says that the mean value of independent random variables will tend to be normal distributed, irrespective of how the random variables themselves are distributed. We illustrate this by sampling uniform data, and show that their mean is approximately normal distributed. Make a script where you first sample 100 random values from the uniform distribution between 0 and 1 (use `runif`. Make a histogram of them to verify they are indeed uniform between 0 and 1. Then, create a numeric vector named `umeans` of length 1000 (you can fill it with 0's). Then make a loop running 1000 times, and inside the loop you sample 100 uniform values like before, and then compute the mean of these. This mean value you store in the `umeans` vector, one by one. After the loop you make a histogram of `umeans`. Does it look like a normal distribution?

7.6.2 Yatzy in a single roll

We will now look at the game of Yatzy, where we roll 5 dice. The rolling of 5 dice can be done in R by `sample(x=1:6,size=5,replace=TRUE)`. This means we sample from the set of integers `1,2,...,6`. We sample from this set 5 times, and we sample with replacement, i.e. any previously sampled value can be sampled again (two dice can show the same number). Now make a loop run 100 000 times and in each iteration roll the 5 dice. If all 5 dice are the same, we have Yatzy. What seems to be the probability of having Yatzy in a single roll? What is the exact probability of this (use brain, pen and paper)?

7.6.3 Expected number of rolls to get Yatzy

How many rolls do we need to get Yatzy if you can hold (set aside) some dice after each roll? Try to make an R script that simulates this situation and estimate the expected number of rolls. Note: It does not matter if you get fives 1s or five 6s, as long as all dice are equal you have Yatzy.

After each roll you need to decide which dice to hold and which to keep rolling. Example: First roll gives `1 2 2 3 5`. I would then hold the two 2s and try to get three more of these. Second roll gives `3 4 6`. Still my two 2s from first round is the best bet. Third roll gives `1 2 4`. Another 2, and I hold this along with my previous two. I now have only two dice left to roll. Fourth roll gives `4 5`. Nothing improves. Fifth roll gives `2 6`. I hold yet another 2, and have only one die left to roll. Sixth roll gives `1`. Seventh roll gives `4`, ... and so on until finally the last die is also 2. The number of rolls we needed to get here is stored, this is the number of rolls needed to get Yatzy. Now, this is repeated many times (at least 1000) and the average number of rolls is our estimate of the expected number.

In addition to the mean number of rolls needed you should also plot the histogram to see how it distributes. How many rolls do you need to be 90% certain of getting Yatzy?

7.6.4 Likelihood ratio test

How can we reveal a dishonest die? A fair die has all six outcomes equally likely, but a manipulated die can produce different results. Here is the theory we need:

In a fair die the six outcomes have probabilities $\rho_1 = \rho_2 = \cdots = \rho_6 = 1/6$. We roll the actual die N times, and count the number of times we observe 1,2,...,6. We denote these $y_1, y_2, ..., y_6$. This is a multinomial vector, which is the extension of the binomial (exercise 4.6.5) when we have more than 2 possible outcomes. We can estimate the probabilities as $\hat{\rho}_i = y_i/N$. If we have made many rolls (large N) we expect all these estimates to be around 1/6, but due to the randomness this will never be exactly the case. How can we detect non-random deviations?

To compare our observations to the fair die we compute the likelihood ratio:

$$\lambda = \sum_{i=1}^{6} y_i \log\left(\frac{\rho_i}{\hat{\rho}_i}\right)$$

Notice that if all y_i are equal then $\hat{\rho}_i = 1/6 = \rho_i$ and $\lambda = 0$. The more λ deviates from 0 the more likely it is that the die is dishonest. But how much can it deviate from 0 and still be a fair die?

According to the theory -2λ should have an approximate χ^2 distribution (chi-square) with $6-1$ degrees of freedom. We will investigate this by simulation. In R, roll a fair die 30 times, compute the vector `y` (use `table`) and compute λ. Repeat this 1000 times, and store the λ values from each round. Make a histogram of -2λ and compare to the χ^2 density (use `dchisq`). (Tip: Use the option `freq=FALSE` in `hist`).

We can now make probabilistic statements around the deviation that λ has from 0. We want a fixed value for λ that indicates a dishonest die, and accept that a fair die will have a 5% probability of also crossing this limit. Which value of λ is this? Hint: Use `qchisq`.

Chapter 8

Building functions

8.1 About functions

8.1.1 What is a function?

Before we start looking at how we build functions, we should reflect a little on what a function really is.

A function is a machine. You put something into it, it does something to this input, and eventually gives some output. In mathematics we have learnt about mathematical functions. The same picture applies. If the function is called f you feed it with some input, say x, and it results in some output. We could write this as $y = f(x)$ indicating that the input is called x, the function itself is called f and the output is stored in something we call y. In the mathematical case the input and output are numbers. In a more general function there are no such restrictions on neither input nor output.

A function is *doing a job*. Data structures, like vectors, data.frames or lists, are *things*. They store data and are the *nouns* of the computer language. A function is a *verb*, does not store anything, it provides action only.

We have already used many functions in R. The function `sd` is an example. It is a machine that computes standard deviation. If we have a vector of numbers called `vec` we can type `sd(vec)`. This means we invoke the action built into the function called `sd`. We give the vector `vec` as input to it. Inside the function many things starts to happen, but we do not see this. In fact, we have no idea what is actually going on inside this machine, and we do not really want to know it either. We are just happy with getting an output, and this output should correspond, without error, to what is described in the documentation of this function.

The name of a function should reflect what it does, sometimes what it outputs. There is no naming convention for functions in R, but try to think of a function as a verb, some action. The name should reflect this action.

8.1.2 Documentation of functions

Documentation is important for functions. First, we need a description of what the function does. We cannot use the function unless we have some idea of what it does. Would you buy a machine and install it in your house without having any idea of what it does? Next, we need to know what type of input it takes. We may categorize input into two levels: Essential input and options. The essential input must always be provided, without this the function will not work properly, or at all. The options are inputs we may give, but they are not absolutely necessary. They can be used to change the behavior of this machine. Finally, we need to know what is the output of the function. An example is the function `plot` that we have used many times already. If we write `plot(x,type="l",col="blue")` the first input, `x`, is essential. This is the data to plot, without this the function has no meaning. The last two inputs, `type="l",col="blue"` are options. The function would work without these inputs, but we may provide them to change the function behavior. In the help files for R functions we find a system:

Description

First we have a short **Description** of what the function does. Sometimes this is too short to really give us the whole picture, but it gives you an impression of what this machine does.

Usage

Next, we find **Usage**. This is what we refer to as the function *call*, i.e. how do you call upon this function to do a job. It gives you a skeleton for how to use this function, what to type in order to invoke it. Often only some of the possible inputs are shown in this call, and ... (three dots) indicates there are other possibilities.

Arguments

Next we usually find **Arguments**. This lists all possible inputs. Arguments is just another word for input here.

Value

Next we may find **Value**, which is a description of the output from the function.

Details

There may also be a section called **Details**. Here you find more detailed description of how the function works.

Other sections

There may be other sections as well, depending on what the maker of the function found necessary. At the end of the help files you always find some example code.

8.1.3 Why functions?

Once people started to write programs of some size, it soon became clear that in order to make this efficient we have to split programs into *modules*, different parts that can be built and maintained on their own almost independent of the other modules. All programming languages have some variant of functions. In computer programs we typically do the same jobs over and over again, under slightly different conditions. Think of plotting as an example. We may need to make many plots, but with different data, and with different views on how the plot should look like. Still, sufficiently parts of this job is the same each time to put it all inside a machine, give it a name, and deploy it as a function in the R-space.

When we decide to build a function it is for these reasons:

- Re-use of code. Once we have written a function we can use and re-use this in all other programs without re-writing it.

- Hiding code. When we use a function we do not need to look inside it. The function itself is often stored away on a separate file. This means programs get much shorter.

- Saving memory. All variables and data structures we make use of inside a function only exists for the very short time that the function is running, and once it is finished they are automatically deleted. Only the output from the function remains available to us.

It is natural that in a programming course like this we focus on *how to* build a function. At the end of the day, it is much more difficult to learn *when to* write a function, i.e. what type of jobs should be granted a separate function. We need a minimum of experience before we can discuss the latter topic.

8.2 The R function syntax

Let us immediately look at an example. In the exercises of the previous chapter we did some simulation of the Yatzy game. We now want to write a function that simulates the rolling of dice. It takes as input the number of dice to roll, and returns the obtained values sorted in ascending order. We want to name the function `roll.dice`:

```r
roll.dice <- function(n.dice){
  d <- sample(1:6,n.dice,replace=T)
  d.sorted <- sort(d)
  return(d.sorted)
}
```

We start out by specifying the function name in the first line. This is always followed by the assignment operator `<-` and the keyword `function`. Then we have the ordinary parentheses in which we specify the names of the arguments. In this example we have only one argument. This first line is the *signature* of the function. After this we have the curly braces indicating the start and end of the function *body*. Inside the body we can write whatever code we like, but the function should always end with the keyword `return` and the name of the variable (in parenthesis) to use as output from this function.

8.2.1 The output: `return`

A function in R can only return one single variable, hence we can only name one variable in the parentheses after `return`. If you want to return many different variables as output, you have to put them into a list, and then return the list. Notice that it is what we specify in the `return` statement that is used as output. Any code you (for some silly reason) enter *after* the `return` statement will never be executed, the function terminates its job as soon as it reaches the `return`.

8.2.2 The input: Arguments

We mentioned above that some arguments are essential while others can be options. It is common to have the essential arguments first. The options are then listed, usually the most frequently used options first. The distinction between essential arguments and options is not a formal one, the difference is simply that options always have a *default value*. This must be specified in the signature line. Default values will be used if no input to that argument is given. In the example above we can have:

```
roll.dice <- function(n.dice=1){
  d <- sample(1:6,n.dice,replace=T)
  d.sorted <- sort(d)
  return(d.sorted)
}
```

which means we have no essential input, only one option. If we call this function without specifying `n.dice` it will take the value `1`, but if we provide an input value this will override the default value. We can call it without any input, like this: `roll.dice()`. You will see there are many functions in R that takes no input. This means they can do their job without any input. Try for instance the function `Sys.time()` in the console window.

8.3 Using a function

Open a new file in the RStudio editor, and type in the `roll.dice` function from above, just as if it was a script. Then, save the file under the name `yatzyfun.R` in your R-folder.

In order to use the function we have to `source` it just like we did with a script. However, there is a huge difference between a script and a function: When you source a script, the program is executed line by line. When you source a function nothing is executed, the function is just created in the workspace memory of R. Try to source the file `yatzyfun.R` and you will see that the function appears in the workspace (unless you have some errors in the code).

A function is executed when you *call* it. This means we type its name and give it some input. We call functions in our scripts, and in other functions, or even directly from the console command line. If you have successfully sourced the `roll.dice` function you can write

```
> roll.dice()
```

in the console window, and the function should return, at random, a single integer from 1 to 6.

Notice that no function can be called until it has been sourced into memory. Also, if you make changes to your function you must remember to save the file and then re-source it into memory. It is only the last version that has been sourced that is used. This is a very common mistake for beginners, editing their functions, but forgetting to save and re-source them before trying them out again.

When we call a function we have to provide it with values for the essential arguments. In our `roll.dice` function there are no essential arguments, and we could get away by calling it without any input. If we want to roll five dice we have to specify

```
> roll.dice(5)
```

indicating that the argument `n.dice` should take the value `5`. Sometimes we also write

```
> roll.dice(n.dice=5)
```

which is identical to the previous call. The difference is that in the latter case we specified that it is the input argument `n.dice` that should take the value `5`. For this function this was not necessary, since there is only one argument. But, most functions have multiple arguments, and specifying the name of the argument is a good idea to make certain there is no confusion about which argument should take which value. We have seen this when we used the `plot` function. We wrote something like `plot(vec,type="l",col="red")`, where the second and third argument is specified by their name.

If you do not specify the names of the arguments during call, R expects the values you give as input to come in the exact same order as the arguments were defined in the function. This makes it cumbersome for functions with many input arguments, one mistake leads to error. However, if you name each input argument, their order makes no matter at all! Naming the arguments also allows you to leave many optional arguments unspecified. Try the following uses of `plot`

```
> plot(x=1:5,y=c(1,2,3,2,1),pch=16,cex=3,col="red" )
> plot(x=1:5,y=c(1,2,3,2,1),col="red",cex=3,pch=16 )
```

Notice they both produce exactly the same plot, even if some of the arguments are in a different order in the second call. As long as we name them there is no confusion. The conclusion is that it is a good idea to always name arguments in function calls.

8.4 Organizing functions

You may create functions inside your scripts, but it is a good habit to not mix functions into scripts, but keep them on separate files. You can have many functions in the same file, but some prefer to have one function per file, and even name the file similar to the function. This is your choice.

I have a habit of naming all my scripts along this pattern; `script_blablabla.R`. Files containing functions lack the prefix `script_`. In this way I can quickly see which files in my folder are containing functions and which are scripts. I prefer to have several, related, functions in the same file, simply to have less files.

Typically, a script makes use of several functions. If I make a script about Yatzy-simulation I would start the script-file like this:

```
source("yatzyfun.R")
```

making certain that the file `yatzyfun.R`, containing my Yatzy-related functions, is sourced as the very first step of each new run of the script. Once this line of code has been executed, all functions in the file `yatzyfun.R` have been created in the workspace, ready to be called.

You may ask; how about all the ready-made function we have used, they are not seen in the workspace? How can we call these functions? The answer is that the ultimate organization of functions in R is by *packages*. Once you install a package, all functions in that package is made available to you without further sourcing. In the current version of RStudio (version 0.98.501) there is an Environment window. The workspace is the same as the Global Environment, which is shown by default in this window, and the functions you build and source are found here. But, any installed package also has an environment, and by selecting one of them (click the Global Environment pop-up) you get listed all functions (and datasets etc.) found in that package. We will talk more about packages later, but keep in mind a package is (usually little more than) a collection of functions.

8.5 Local variables

If you have been programming in other languages before, this is a familiar concept. If you are new to programming, this is usually one of the more difficult topics, and you should study this carefully to make certain you follow this.

All variables we create in a function, including the arguments, are what we call local variables. This means they are created and exist only as long as it takes for the function to complete its job. Once it has finished they are deleted from the memory. Make a call to the function `roll.dice` we created above. After it has finished there are no variables named `n.dice`, `d` or `d.sorted` in the workspace.

Whenever we call a function we communicate with this machine through its input and output. Make the following small script:

```
source("yatzyfun.R")
number.of.dice <- 7
roll7 <- roll.dice(number.of.dice)
```

and source it. Upon completion you will find that the variables `number.of.dice` and `roll7` exist in the workspace, but not the local variables inside the `roll.dice` function. This is what happens:

First we create `number.of.dice` and give it the value `7`. Next, we call the function `roll.dice` with `number.of.dice` as argument. This means the *value* of our variable `number.of.dice` is copied into the function and assigned to the argument `n.dice`. Thus, when the function starts to execute its body code lines, the variable `n.dice` exists and has the same value as `number.of.dice`. Note, these are two different variables, we just copied the value from one to the other.

Inside the function the job is completed, and the variable `d.sorted` has some values. Then, as the function terminates the values of `d.sorted` is copied to the workspace-variable `roll7`. We specify by the line
`roll7 <- roll.dice(number.of.dice)` that the variable `roll7` should be created, and assigned the value that is output from the call `roll.dice(number.of.dice)`.

It is imperative that you understand the copying going on here. The *value* of `number.of.dice` is copied into the function, and the *value* of `d.sorted` is copied out of the function.

Why is it designed like this? Why could not the variables inside the function be visible in the workspace, just like the variables we create in our scripts? We will not dwell on this issue here, but a short answer is to save space and to make functions easier to make use of. Remember a function is a machine, like a car. It is much more convenient to use a car if all the details are hidden away, and the interface we have to relate to is simple. All machines are designed like this, even virtual machines.

8.6 Important functions

Command	Remark
`sort`	Sorting data in a vector
`order`	The first part of the job done by `sort`

8.7 Exercises

Let us expand on the file `yatzyfun.R` and create some more functions relating to the Yatzy game. In a regular game of Yatzy you always roll the five dice up to three times in each round. The first part of the game consists of six rounds where you collect as many as possible of the values 1,2,...,6. In the first round you must collect the value 1, i.e. after each roll you hold all dice having the value 1, and you roll the remaining dice. After three rolls you count the number of dice with value 1, which is your score in this round. In the second round it is similar but you must now collect the value 2. The score you get after three rolls is the number of 2's times 2. Similar applies to the values 3, 4, 5, and 6. After these six rounds you sum up the scores so far, and if you have more than 62 points you get 50 extra bonus points.

8.7.1 Yatzy - bonus

Make a function called `yatzy.part.one` that takes as input the target value to collect (1 or 2 or... or 6). Then, the function rolls the dice (maximum) three times, and holds dice after each roll to get as many as possible of the target value. The function should return the score you achieve on the round. You should make use of the function `roll.dice` from the text.

Make a script that uses the functions to simulate the first six rounds of Yatzy thousands of times, and compute the score to see if the total after six rounds qualify for bonus. What is your estimate of the probability of getting the bonus? How does the score-distribution look like after this first part, before you include the bonus points?

8.7.2 Yatzy - straights

The second part of Yatzy is more strategic (the first part above is pure luck). You have to make some more or less smart choices after each roll. However, there are two exception. In two of the part-two rounds you seek what is called *small* and *large straight*. A small straight means the five dice show the values 1, 2, 3, 4 and 5. In a large straight they show 2, 3, 4, 5, and 6. If you obtain a straight you get the sum of the five values as score (15 and 20, respectively). If you fail you get 0.

Make a function called `yatzy.straight` taking a logical as input, deciding if you seek small or large straight. Then the function rolls the dice (maximum) three times and return the score, i.e. 15 (or 20) if you obtain a small (or large) straight, 0 otherwise. Again, simulate thousands of rounds to estimate a) the probability of obtaining small or large straight, and b) the expected number of points on a round where we seek small or large straight.

8.7.3 Yatzy - pair

Finally, a taste of strategic choice. One of the part-two rounds seeks *one pair*. This means we need (at least) two dice to show the same value, and the score we get is the sum of these two values. It is usually not difficult to get a pair on three rolls, but we should now aim to get as high values as possible. A pair of 6's gives you 12 points, while a pair of 5's give you 10, and so on. If you have made two of your three rolls, and you have a pair of 3's, should you hold these two dice and roll the remaining three to try to get something better? Or, should you abandon this pair altogether, and roll all five dice, increasing the chance of getting more points, but also risking to fail?

Make a function called `yatzy.one.pair` that takes as input a logical named `optimist`. If `optimist` is `TRUE` you always go for the pair of 6's no matter how it looks like after roll 1 and 2. If it is `FALSE` you are always satisfied with any pair, only rolling the three remaining dice to improve the existing score. You only set aside a pair, with no pair seen you roll all dice again, i.e. you either roll five or three dice in this scenario. Note: If you have no pair after three rolls your score is 0. Which strategy is the best, the 'Solan' (optimist) or the 'Ludvig' (pessimist)?

Can you make the ultimate `yatzy.one.pair` function, i.e. the function that beats both the strategies from above? This is a very small taste of artificial intelligence...

Chapter 9

Plotting

In this chapter we will look closer at how to visually display data and results, which is an important part of statistics and data analysis in general. We will introduce certain aspects of R graphics along the way. There is no way we can dig into the depths of R graphics, there are simply way too many possibilities. We will restrict this to the most common ways of plotting.

9.1 Scatterplots

A scatterplot is simply a plot of the values in one vector against those in another vector. The vectors must be of the same length. The scatterplot shows a two-dimensional coordinate system, and a *marker symbol* is displayed at each coordinate pair. The commands we use to make scatterplots in R are the functions `plot` and `points` that we have already seen. Let us make an example with some random values from the normal distribution that shows some possibilities:

```
many.x <- rnorm(1000)*2
many.y <- rnorm(1000)*2
medium.x <- rnorm(30)
medium.y <- rnorm(30)
few.x <- rnorm(5)*0.75
few.y <- rnorm(5)*0.75
plot(x=many.x,y=many.y,pch=16,cex=0.5,col="gray70",
     xlab="The x-values",ylab="The y-values",
     main="A scatterplot")
points(x=medium.x,y=medium.y,pch=15,col="sandybrown")
points(x=few.x,y=few.y,pch=17,cex=2,col="steelblue")
```

A result of running this code is shown in Figure 9.1. Notice how we use both `plot` and `points`. The reason is that each time you call `plot` any previous plot is erased before the new is displayed. Thus, if you want to add several plots to the same window, you use `plot` the first time, and then add more by using `points`. You cannot use `points` unless there is already an existing plot.

There are many options we can give to the `plot` and `points` functions. The option `pch` takes an integer that sets the marker type. In the example above the gray markers are round, the blue are squares and the red are triangles (pch

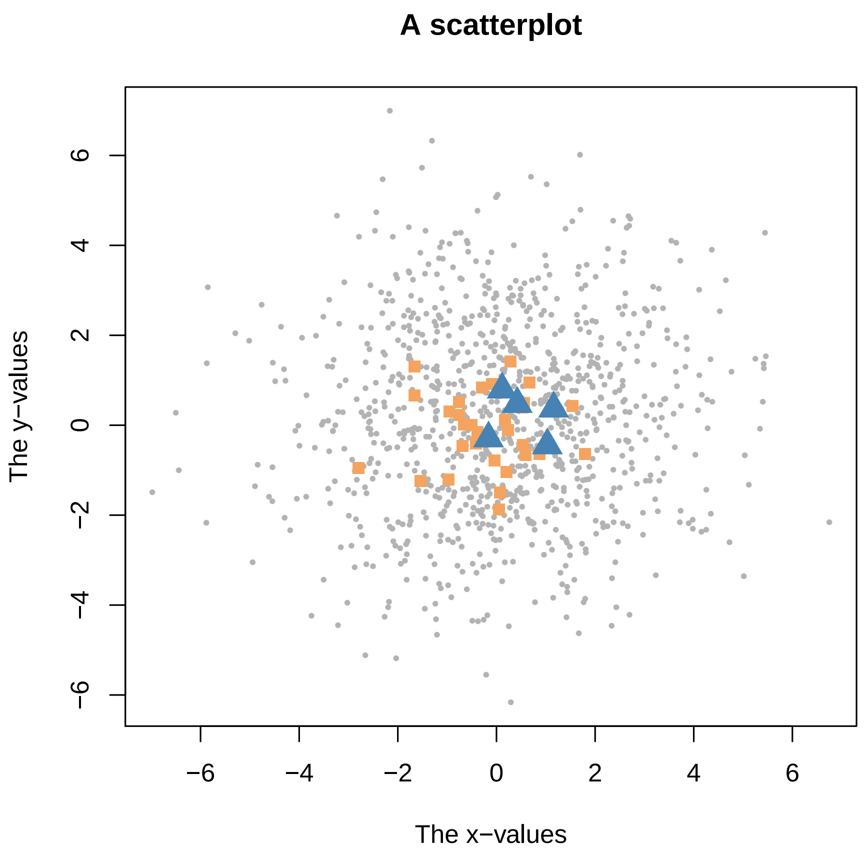


Figure 9.1: A scatterplot with three different markers and colors for three groups of data.

values 16, 15 and 17 respectively). See `?points` for the details on which marker corresponds to which integer. The option `cex` scales the size of the marker. By default it is 1.0, i.e. no scaling. The gray markers have been scaled down (`cex=0.3`) and the red up (`cex=2.0`). The color is set by `col`, type `colors()` in the console window to get a listing of all available named colors.

When we make the first `plot` there are some options we can set here that cannot be changed by later additional use of `points`. The range of the axes are two. The options `xlim` and `ylim` specify the outer limits of the two axes. If these are not specified they are chosen to fit the data given to `plot`. This is fine, but if you want to add more plots you must make certain the axes limits are wide enough to include later data as well. Before you start plotting multiple plots you must always check for the smallest and largest x- and y-values that you are going to see, and specify the `xlim` and `ylim` accordingly. The function `range` can be handy for checking this. Other options you set in the first `plot` are the texts to put on the axes, i.e. `xlab` and `ylab`, and the title `main`.

9.2 Lineplots

If the values on the x-axis (horizontal axis) are sampled from a continuous variable, like time-points, it is more common to use lines than markers. It also requires the x-data to be ordered. Here is an example of some plots of this type:

```
time <- 1:10
time <- 1:10
y1 <- rnorm(10,mean=time)
y2 <- y1+1
y3 <- y1+2
y4 <- y1+3
plot(x=time,y=y1,type="l",lwd=1.0,lty="solid",
     col="saddlebrown",xlim=range(time),
     ylim=range(c(y1,y2,y3,y4)),xlab="Time (days)",
     ylab="y values",main="A lineplot")
points(x=time,y=y2,type="l",lwd=2.0,lty="dashed",
       col="slategray3")
points(x=time,y=y3,type="l",lwd=1.5,lty="dotted",
       col="purple4")
points(x=time,y=y4,type="b",lwd=5,lty="longdash",
       col="seagreen")
```

The result is shown in Figure 9.2. The functions we use are the same as for the scatterplot, but some of the options are different. We specify `type="l"` for *line* or `type="b"` for *both* (both marker and line) as in the last code line. Instead of `cex` we use `lwd` to specify line width, but the scaling principle is the same. The option `lty` is used to specify the line type.

There are many more options you can give to `plot` and `points`, and a good way to explore them is to explore the function `par`. This is used to set graphics parameters, see `?par`. Options to `par` can also be used as options to `plot` and `points`.

9.3 Histograms

We have already seen histograms. A histogram takes a single vector of values as input, and split the range of values into a set of intervals. Then counts the number of occurrences in each interval. The function we used is called `hist`. Here is an example:

```
v1 <- rnorm(1000)
hist(x=v1,breaks=30,col="lemonchiffon2",
     border="lemonchiffon4",main="A histogram")
```

The result is shown in Figure 9.3. The option `breaks` indicate how many intervals we should divide the range into, i.e. how many bars we get in the plot. It is just a suggestion, the `histogram` function makes the final decision. The default is `breaks=10`. The more data you give as input, the more intervals you should use. The `col` option specifies the color of the interior of the bars, while `border` sets the colors of the bar borders. If you set these to the same color the bars will 'grow' into each other. You give texts to the title and the axes as for `plot`.

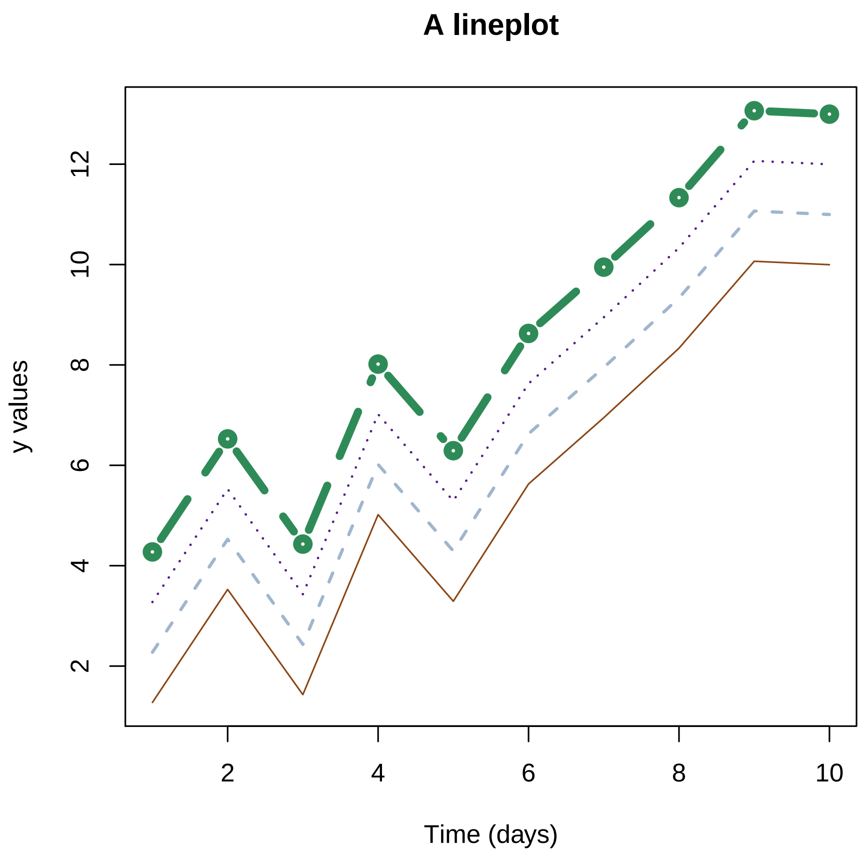


Figure 9.2: A plot with lines (curves) instead of markers.

9.4 Barplot

A barplot is somewhat similar to the lineplot in Figure 9.2, but we use bars if the x-axis is discrete (not continuous). It is different from a histogram in the sense that a histogram bar represent an interval, but a barplot bar represents a single point on the x-axis. For this reason, the histogram bars have no 'air' between them, while barplot bars always should have some space between each bar. Let us look at an example:

```
parties <- c("R","SV","Ap","Sp","V","Krf","H","Frp","Other")
poll <- c(1.4,4.9,28.4,4.7,4.5,5.3,32.9,15.6,2.2)
barplot(height=poll,names.arg=parties,col="brown",
    xlab="Political parties",ylab="Percentage votes",
    main="A barplot")
```

The result is shown in Figure 9.4. The option `names.arg` should be a vector of texts to display below each bar, thus it must be of the same length as the

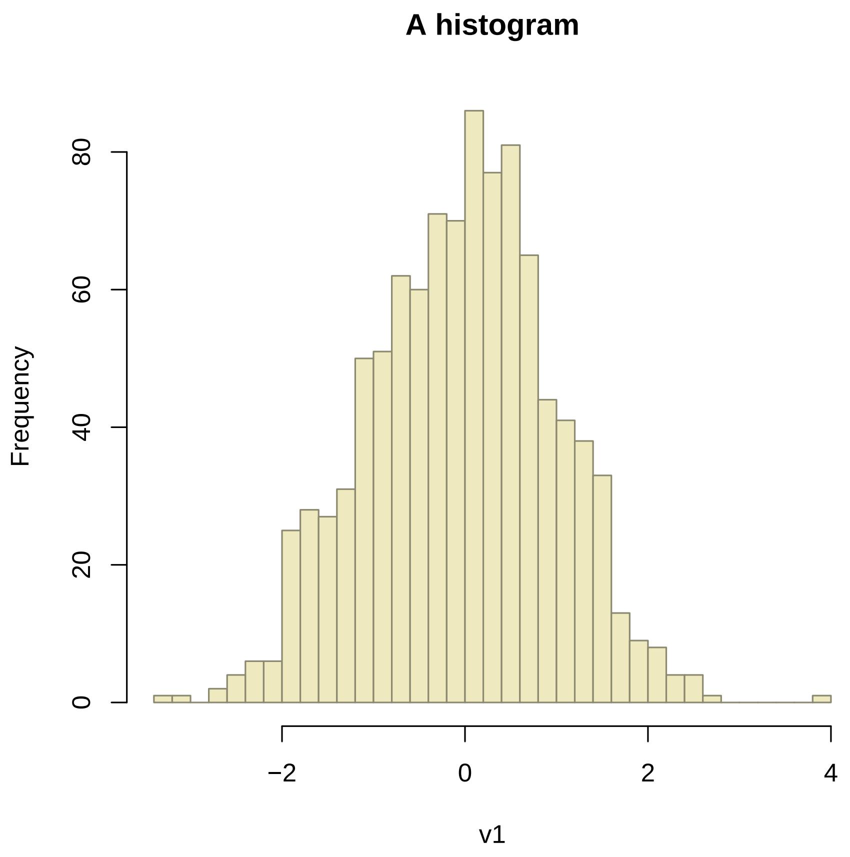


Figure 9.3: A histogram with colored bars and borders.

essential argument `height`.

Sometimes we want to put two (or more) sets of bars in the same plot. Let us roll a fair die 1000 times, and compare the outcomes to a die where the probability of 1 and 6 is larger then in a fair die:

```
d1 <- sample(x=1:6,size=1000,replace=T)
d2 <- sample(x=1:6,size=1000,replace=T,prob=c(3,1,1,1,1,3))
c1 <- table(d1)
c2 <- table(d2)
countmat <- matrix(c(c1,c2),nrow=2,byrow=T)
die.names <- paste("Die=",names(c1),sep="")
barplot(height=countmat,names.arg=die.names,
        col=c("tan2","tan4"),beside=TRUE,horiz=TRUE,las=2,
        xlab="Outcomes",main="Fair and unfair die")
```

In Figure 9.5 we show the result, where the brighter bars show the distribution of the fair die, and the darker bars of the unfair die, having larger probabilities of getting a 1 or a 6. We used the option `beside=TRUE` to put the bars beside

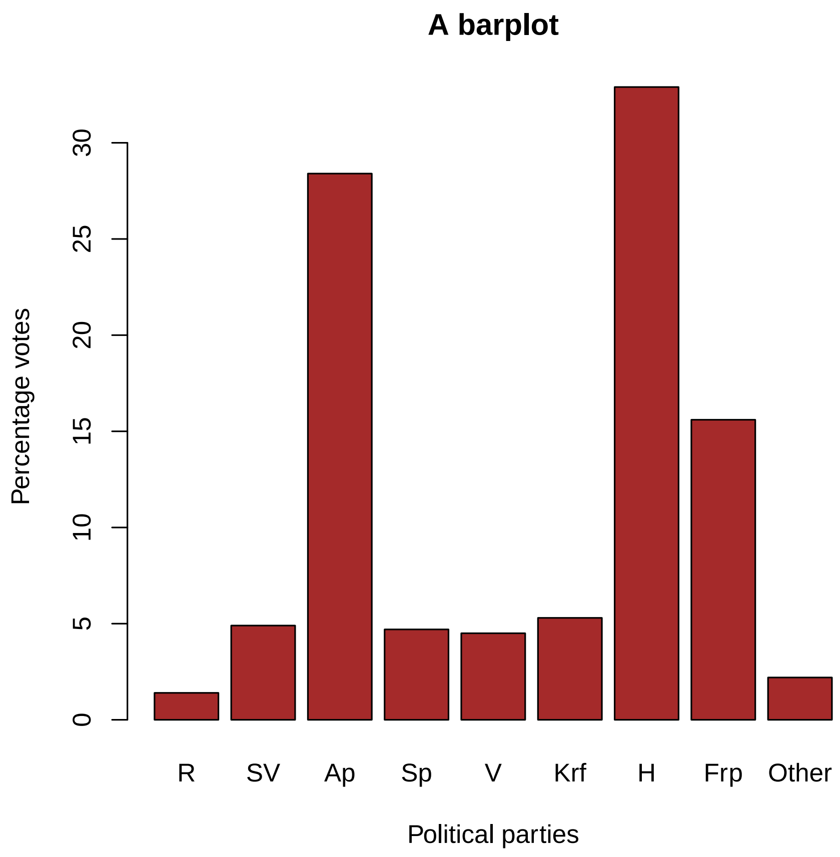


Figure 9.4: A barplot is used when the x-axis is discrete.

each other. Try to run this code with `beside=FALSE` to see the effect. We also used the option `horiz=TRUE` to put the bars horizontally.

9.5 Pie charts

An alternative to the barplot is the pie chart. This will only give you the relative distribution between the outcomes, not the absolute count in each group. Here is the pie chart version of the votes from above, where we also arrange the parties in the 'political' order from left to right:

```
parties <- c("R","SV","Ap","Sp","V","Krf","H","Frp","Other")
poll <- c(1.4,4.9,28.4,4.7,4.5,5.3,32.9,15.6,2.2)
p.col <- c("darkred","red2","red","green","green2",
           "yellow","blue","blue2","black")
pie(x=poll,labels=parties,col=p.col,main="A pie chart")
```

See in Figure 9.6 how it looks like. Pie charts can be used for giving an overview

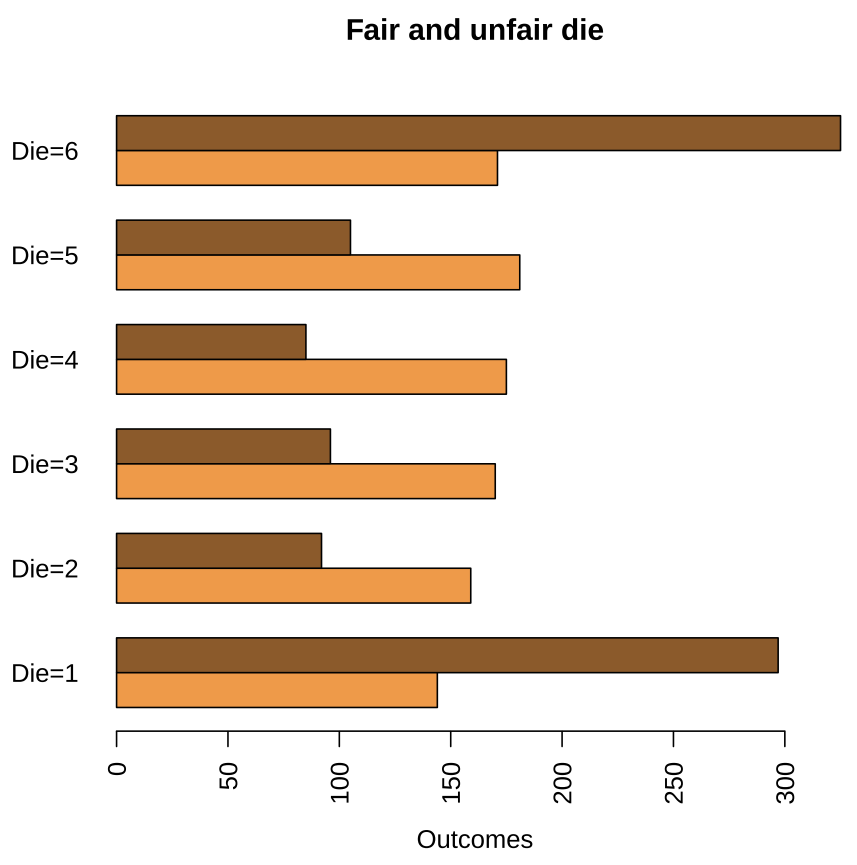


Figure 9.5: A barplot with two sets of count data as bars beside (above!) each other.

of discrete distributions. They are more popular in business than in academic sciences, probably because they make it difficult to read exact (academia is about telling the truth, business is about telling a good lie!). Look at the help-file for `pie`. In the very first sentence under **Note** it says: *Pie charts are a very bad way of displaying information*!

9.6 Boxplots

A boxplot, or box-and-whisker plot, is a way to visualize a distribution. We can have several distributions in the same boxplot. Let us sample data from three different distributions: The standard normal distribution, the student-t distribution with 3 degrees of freedom and the uniform distribution on the interval `(0,1)`. Then we make boxplots of them.

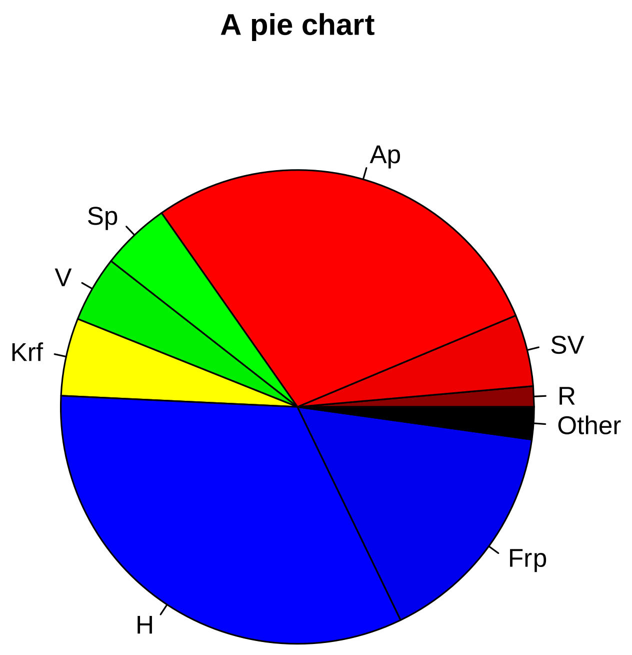


Figure 9.6: A pie chart as an alternative to a barplot.

```
dist.samples <- list(Normal=rnorm(25),Student=rt(30,df=3),
                     Uniform=runif(20))
boxplot(dist.samples,main="A boxplot",
        col=c("bisque1","azure1","wheat1"))
```

In Figure 9.7 you can see the boxplot produced. For each group of data we have a box. The horizontal line in the middle is the median observation. The width of the box and the notches reaching out from the box describe the spread of the data. Their exact meaning can be adjusted by options, see `?boxplot` for details. Finally, extreme observations are plotted as single markers. We see from the current boxplot that the uniform distribution has the smallest spread, and is also centered above zero. The normal distribution is centered close to zero, and the same applies to the Student-t distribution. The latter has 1 extreme observation.

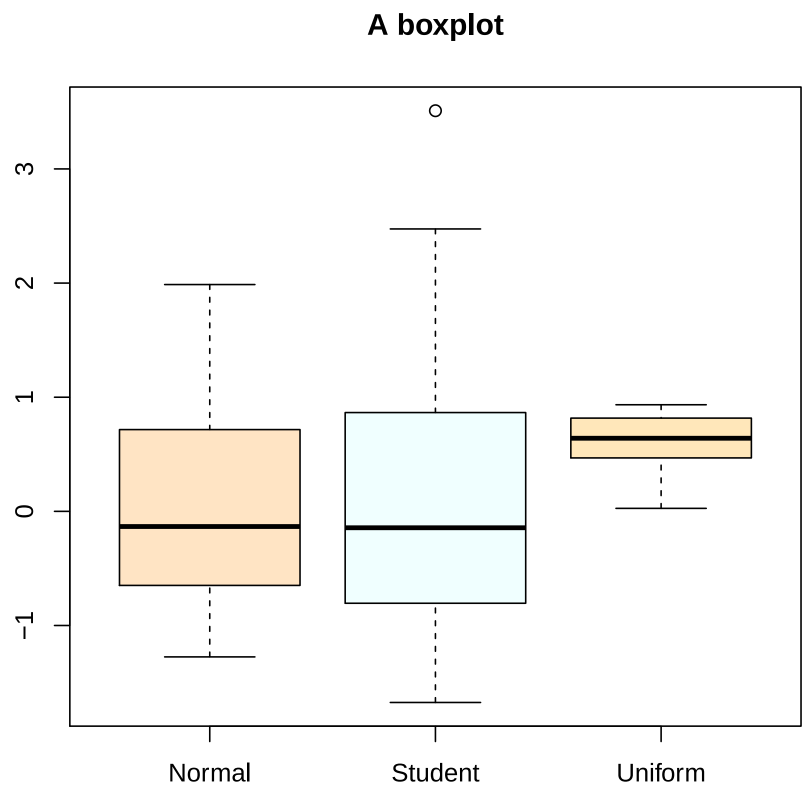


Figure 9.7: A boxplot.

9.7 Surfaces

We can also plot surfaces in 3D. A surface is a matrix of heights (the z-coordinate) where each row and column corresponds to a location in two directions (x-coordinate and y-coordinate). Here is an example:

```
vx <- 1:20
vy <- (1:30)/5
vz <- matrix(0,nrow=20,ncol=30)
for(i in 1:20){
  for(j in 1:30){
    vz[i,j] <- max(0,log10(vx[i])+sin(vy[j]))
  }
}
persp(x=vx,y=vy,z=vz,theta=-60,phi=30,main="A surface",
      col="snow2")
```

The surface can be seen in Figure 9.8. The essential arguments to `persp` are the

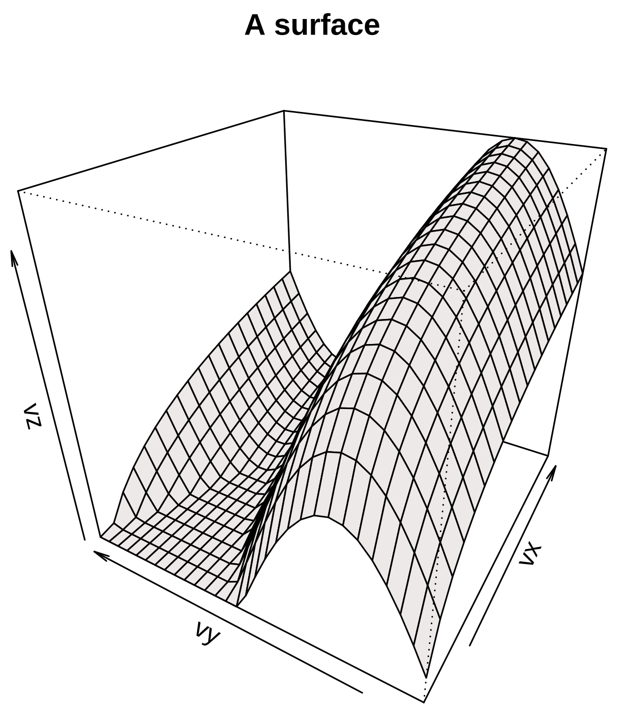


Figure 9.8: A 3D surface.

data (x-, y- and z-coordinates, where the latter is a matrix). The two additional options specified here control the angle from where we see the surface. Try different values for `theta` and `phi` and see how you can see the surface from various perspectives, or replace the line `persp(x=vx,y=vy,z=vz,theta=-60,phi =30)` with

```
for(i in 1:100){
  th <- i-150
  ph <- (100-i)/2
  persp(x=vx,y=vy,z=vz,theta=th,phi=ph)
  Sys.sleep(0.1)
}
```

and run the 'movie'.

9.8 Contourplots

Even if surfaces may look pretty, they are difficult to read, and a better way of displaying them is often by a contourplot. The function `filled.contour` can be used for this purpose, and here we use exactly the same data as in the surface example to make such a plot:

```
vx <- 1:20
vy <- (1:30)/5
vz <- matrix(0,nrow=20,ncol=30)
for(i in 1:20){
  for(j in 1:30){
    vz[i,j] <- max(0,log10(vx[i])+sin(vy[j]))
  }
}
filled.contour(x=vx,y=vy,z=vz,main="A contourplot")
```

The plot it produces is shown in Figure 9.9. A contourplot sees the surface from 'above' and divides it into levels of different colors, just like a map. Until now we have just used named colors in our plots. This is no longer the case for contourplots. It is time we take a closer look at colors.

9.9 Color functions

In a contourplot we need a range of colors. A color *palette* is a function that gives you a vector of colors. The palette function typically takes an integer as input and outputs a vector of that length. There are several built-in palette functions in R. Here is one example called `rainbow`:

```
cols <- rainbow(80)
barplot(rep(1,time=80),col=cols,border=cols,
        main="Eighty colors produced by rainbow()")
```

See the colors in the barplot in Figure 9.10. If you inspect the vector `cols` you will see it contains texts, and these texts are actually hexadecimal numbers that R converts to colors. In the example above we used these color-codes instead of the named colors as input to `barplot`. In all cases we have used named colors we could have used these color-codes instead. Notice that it was our choice to look at 80 colors in the example above, you could just as well have chosen another integer.

The function `filled.contour` from above takes as one of its options a color palette. The default is a palette called `cm.colors`, but let us try the palette called `terrain.colors`:

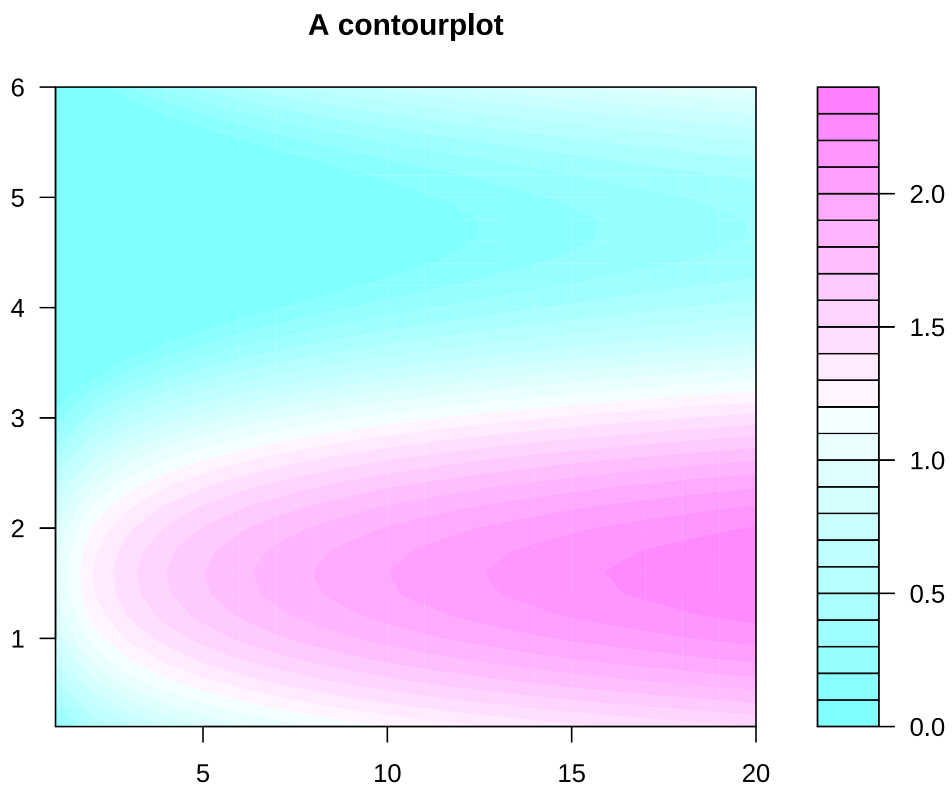


Figure 9.9: A contourplot is like a map, and usually a more informative display than a 3D surface.

```
vx <- 1:20
vy <- (1:30)/5
vz <- matrix(0,nrow=20,ncol=30)
for(i in 1:20){
  for(j in 1:30){
    vz[i,j] <- max(0,log10(vx[i])+sin(vy[j]))
  }
}
filled.contour(x=vx,y=vy,z=vz,color.palette=terrain.colors,
    main="A contourplot with terrain.colors")
```

Compare Figure 9.9 to Figure 9.11.

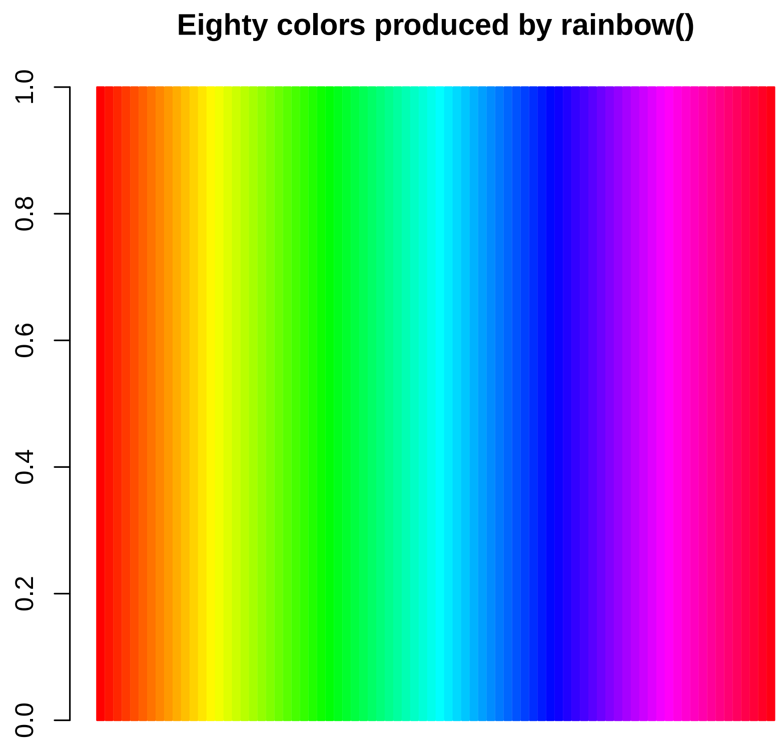


Figure 9.10: The colors produced by a built-in palette. Each vertical bar has a different color.

9.9.1 Making your own palette

A palette is a function, and we have seen how we can make our own functions. However, we do not need to type in a lot of code to produce a palette function, there are actually built-in functions in R that produces another function!

The function `colorRampPalette` can be used to create a new palette function. You give it as input a vector of colors, and the created palette function will interpolate between these colors to produce the final colors. This means the ordering of the colors is very important for the final result. Let us make an example where we try to make a palette function that gives us colors similar to what we find in maps. In maps we usually see dark blue colors for the deepest oceans, turning gradually into lighter blue and then green when we enter the sea surface. In-land the colors gradually change to yellow, then orange and red-brown as we approach high mountains. Here is an effort to create a palette for this coloring:

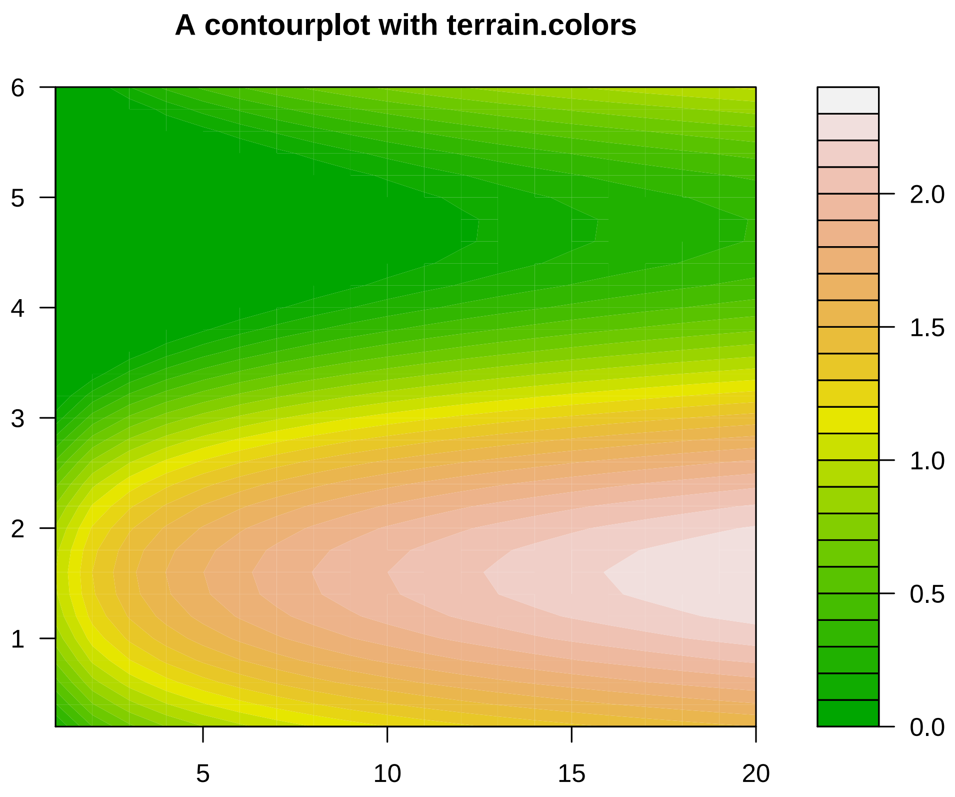


Figure 9.11: A contourplot can take a palette function as input, and display the colors produced by the palette.

```
base.cols <- c("blue4","blue2","blue","green","yellow",
               "orange","red4","brown")
my.colors <- colorRampPalette(base.cols)
filled.contour(x=vx,y=vy,z=vz,color.palette=my.colors,
               main="A contourplot with our own colors")
```

Here we assume the data for making the contourplot is still available in the workspace, if not, run the previous examples first. The result is shown in Figure 9.12.

9.10 Multiple panels

We can easily produce several plots in the same window. The simplest approach is to use the `par` function. One of its option is `mfrow`, and this should be given a vector of two elements. It divides the plot-window into rows and columns, and

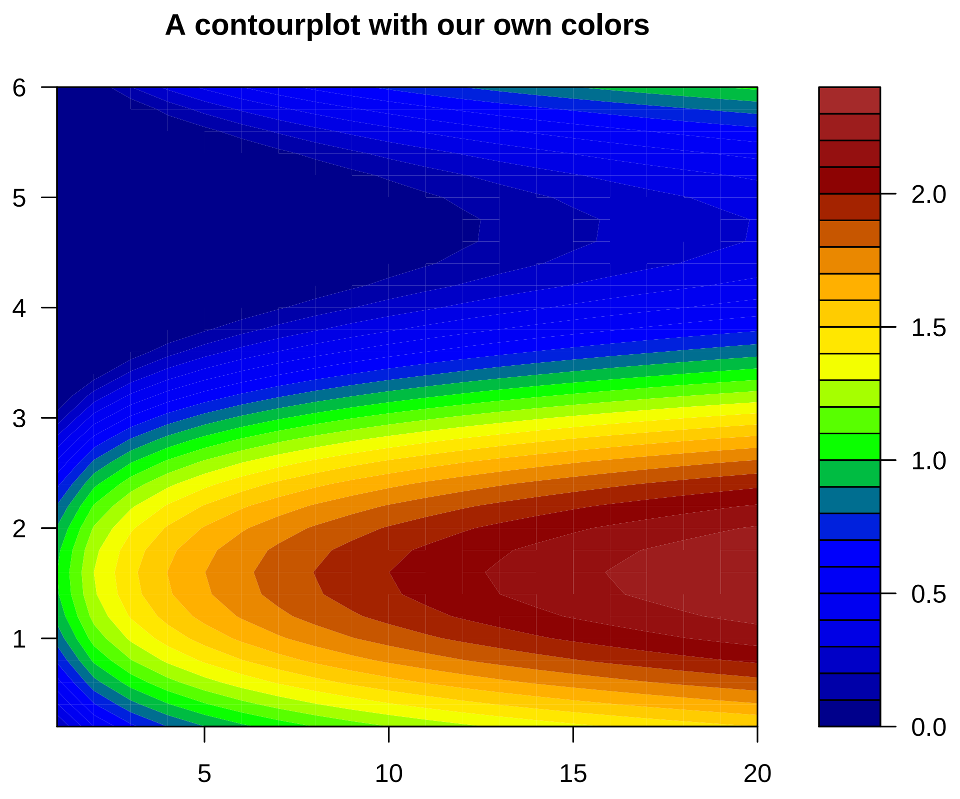


Figure 9.12: A contourplot where we have used our own palette, see text for details.

the first value of `mfrow` is the number of row, the second the number of columns. If we want to make two plots beside each other, it means we divide the window into one row and two columns, hence we set `mfrow=c(1,2)`. Here is an example where we divide into 2 rows and 3 columns:

```
par(mfrow=c(2,3))
plot(1:10,rnorm(10),pch=15,col="red4")
plot(rnorm(1000),rnorm(1000),pch=16,cex=0.5,col="steelblue")
plot(1:10,sin(1:10),type="l")
barplot(c(4,5,2,3,6,4,3))
hist(rt(1000,df=10),breaks=30,col="tan")
pie(1:5)
```

See Figure 9.13 for the result. Notice that for each plot command the next position in the window is used. The alternative option `mfcol` will do the same job, but new plots are added column-wise instead of row-wise.

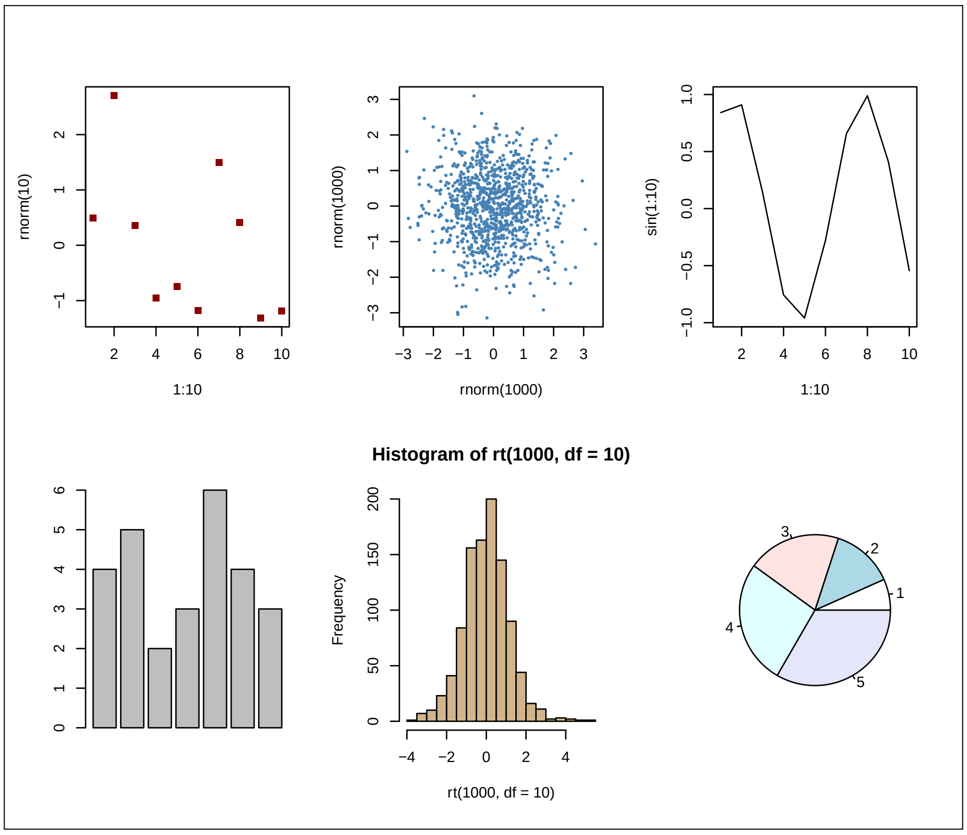


Figure 9.13: Plotting several plots in the same window, in this case the plots are arranged into two rows and three columns.

Another approach is to use the `layout` function. This allows us to divide the graphics window into different regions where the plots should appear. Again we divide into rows and columns, but a plot can occupy several 'cells'. This is indicated by a matrix, where all cells having the same value makes up a region. Let us divide the window into 2 rows and 2 columns, but let both cells in the first row be a region, while the cells of the second row are separate regions. This means the first plot we add takes up the top half of the window, the next two plots share the bottom half:

```
lmat <- matrix(c(1,1,2,3),nrow=2,byrow=T)
layout(lmat)
barplot(rnorm(20),col="pink")
plot(rnorm(1000),rnorm(1000),pch=16,cex=0.7)
hist(rchisq(1000,df=10),col="thistle",border="thistle",
    main="")
```

Have a look at the matrix `lmat` in the console, and compare it to the result in Figure 9.14. Notice how the regions are specified in `lmat` and how the plots appear along the same pattern.

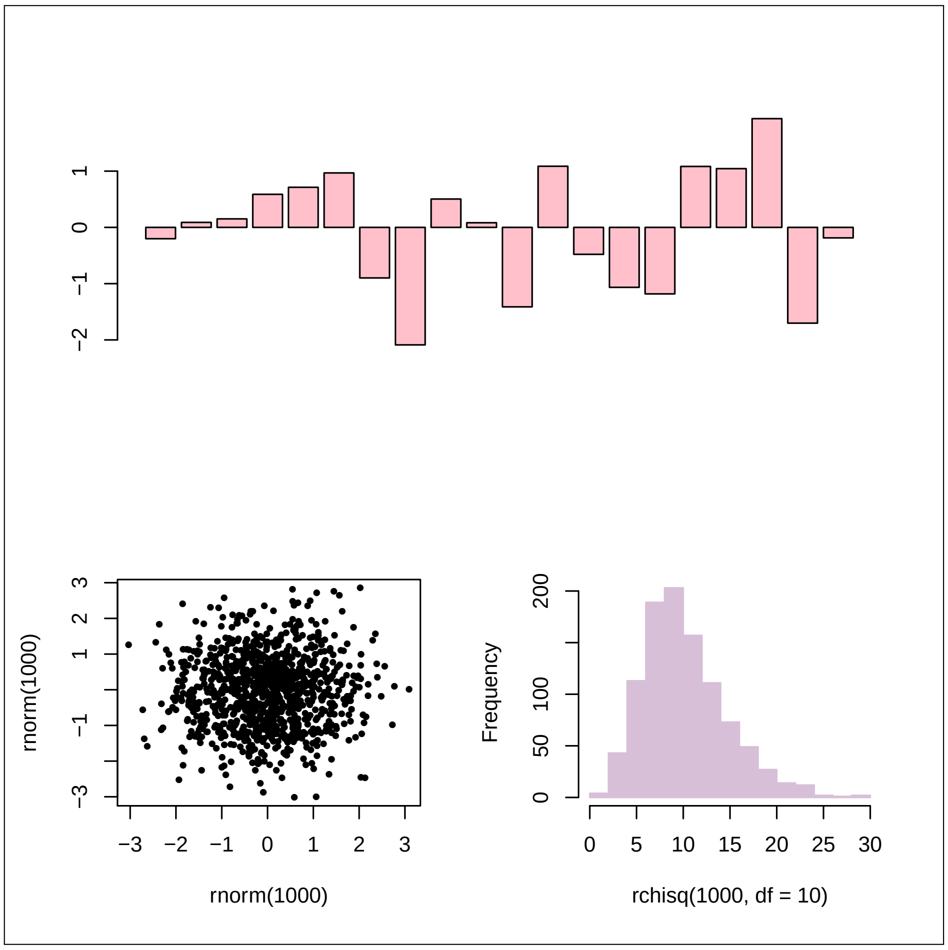


Figure 9.14: Here we have created regions of different size for the plots, with one wide region at the top and two smaller below.

9.11 Manipulating axes

At times we would like to put some specific text on the axes, or add extra axes, to a plot. All this is possible, but it takes some lines of code to do it. Before we look at the code, let us define some phrases we will meet. An *axis* is the 'ruler' type of line we find at the outer sides of a plot, e.g. in Figure 9.1, where we have a horizontal axis below and a vertical axis to the left of the markers. The markings on the axis are called *tick marks*, and their corresponding texts are called *tick labels*. The *axis label* is the text we may or may not give to the axis, describing what it measures. Also, the fours *sides* of a plot are numbered: Side 1 is the horizontal side below, side 2 is the left hand side, side 3 is on the top and side 4 is the right hand side.

There are many built-in options we can give to plotting commands to manipulate the axes and their texts, see `?par` for all details on this. Here are some:

`cex.axis`	Scaling of the tick labels.
`col.axis`	Coloring of tick labels.
`font.axis`	Font of the tick labels.
`las`	Orientation of tick labels.
`tck`	Length of tick marks. See also `tcl`.
`xaxp`	How to place the tick marks. See also `xaxs`.
`xaxt`	Can be used to suppress the plotting of an axis. Important!

A common approach to really manipulating axes is to use the option `xaxt="n"` to suppress the plotting of an axis, and then add a new axis using the function called `axis`. This function allows you to add a new axis, place it where you like, and completely specify the tick marks, the tick labels, colors and anything else you would like to change. Let us have a look at an example.

```
x.vec <- rnorm(30)
y.vec <- rnorm(30)
plot(x=x.vec,y=y.vec,pch=16,cex=0.75,col="black",
     xlim=c(-3,3),xaxt="n",xlab="",
     ylim=c(-3,3),yaxt="n",ylab="")
```

This produces a plot completely void of axes. Let us now add a horizontal axis at the top of the plot.

```
axis(side=3,at=c(-3,-2,-1,0,1,2,3),
    labels=c("This","is","a","silly","set","of","labels"),
    col="red")
```

Notice how we specified `side=3` to place it at the top, and how we can completely specify where to put the tick marks (`at`) and how to label them (`labels`). We also added color to the axis (`col`) as well as the tick labels (`col.axis`). Let us add another axis to the right

```
axis(side=4,at=c(-2.5,0,1),labels=c("A","B","C"),
     col="green",col.axis="blue",las=2)
```

By not specifying anything but the `side` we get an axis just like we usually have:

```
axis(side=1)
axis(side=2)
```

Try to run these lines of code, and observe the results. Study the help file `?axis`.

When you start to manipulate axes you will soon also need to manipulate the *margins* surrounding the plot to make more or less space for tick labels and axis labels. See the `mar` option to the `par` function for how to manipulate margins. Again the four sides of a plot is numbered clockwise starting at the bottom.

9.12 Adding text

You can add a text at any specified location in an existing plot. Here is how:

```
plot(x=0,y=0,pch=16,col="blue",cex=2)
text(x=0,y=0,labels="The origin",pos=3)
```

Notice the option `pos` that directs where to put the text relative to the point specified. Here we used `pos=3` to put the text above the point. You can place many texts at many positions by using vectors as input to `x`, `y` and `labels`. See `?text` for more details.

A *legend* is a box with some explanatory text added to a plot. The function `legend` allows you to do this in a simple way. Here is an example:

```
t <- seq(from=0,to=10,by=0.1)
s <- sin(t)
c <- cos(t)
plot(t,s,type="l",col="red",lwd=2)
points(t,c,type="l",col="blue",lwd=1)
legend(2.7,0.95,legend=c("Cosine","Sine"),
       col=c("blue","red"),lwd=c(1,2),
       text.width=1)
```

The result is shown in Figure 9.15. Notice how we specify the upper left corner of the box to be located at `(2.7,0.95)`. This must usually be fine-tuned by trial and error to avoid it from obscuring the lines or markers of the plot.

9.13 Graphics devices

We have so far been plotting in the plot-window of RStudio. If you would like to have your plot in a separate window this can be achieved by the function `X11`. This creates a separate graphics window, and all subsequent plotting command will now create plots in this window. You can specify several options to `X11`, e.g. the `width` and `height` to modify the shape of the window, see `?X11` for details.

Other devices can also be invoked. The functions `pdf` and `postscript` will send all subsequent plot to a file (PDF-file or postscript-file). The functions takes as input the file name, and again `width` and `height` can be specified. Functions like these are typically nice to use when you are preparing plots to include in reports. You first direct the plotting to a `X11` window for inspection. After you have edited your script to make it look exactly the way you want, you replace `X11` by `pdf` or something else to produce the graphics file. NB! When you use devices other than `X11`, always remember to end the plotting with the code line `dev.off()` to close the connection to the graphics device. Files will be incomplete until the device has been switched off.

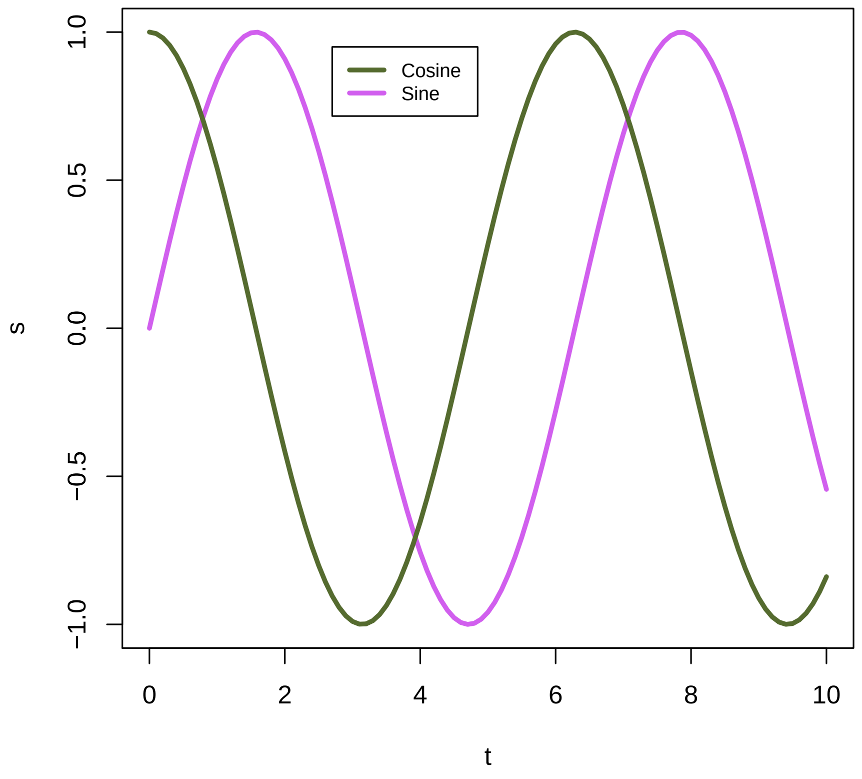


Figure 9.15: A plot with a legend.

9.14 Important functions

Command	Remark
`par`	Can be used to specify a long list of graphics parameters before plotting
`plot,points`	Basic plotting (scatterplot/lineplot)
`hist`	Histograms
`barplot`	Barplots, for discrete data
`pie`	Piechart, (poor) alternative to `barplot`
`boxplot`	Alternative to histogram
`persp`	Surfaces
`filled.contour`	Contourplot
`colorRampPalette`	Creates a color function
`layout`	Divides a plot window into panels
`axis`	Adds an axis to a plot
`text`	Adds text to a plot
`legend`	Adds a legend to a plot
`X11,pdf,postscript`	Alternative graphical devices
`dev.off`	Used after plotting to file (e.g. `pdf`)

9.15 Exercises

9.15.1 Plotting function

Make a function that creates a scatterplot of `y` versus `x`, but in addition to this also put histograms showing the distribution of `y` and `x` in the margins. See Figure 9.16 for an example.

9.15.2 Weather data

We will have a look at the weather data in the file `daily.weather.RData` again.

Make a script that creates a scatterplot of `Air.temp` (horizontal axis) against `Air.temp.min` (vertical axis). Use round markers with a dark blue color. Then plot `Air.temp` against `Air.temp.max` in the same plot (use `points`). Use lighter blue (cyan) triangles. Add texts to the axes. Remember to scale the axes to make certain all dots fit inside the plot.

The variable `Wind.dir` indicates the wind direction, and is a categorical variable (8 levels). Make a barplot showing how many days we have observed the different wind directions in this period. HINT: Use `table` to count the number of occurrences. Again, put proper texts on the axes. Which wind directions are common here at Ås? Change the barplot to make horizontal bars. Also, arrange the directions clockwise from North (N,NE,E,SE,S,SW,W,NW). HINT: Convert the text variable to a factor where you specify the 8 levels in this order, then use it as input to `table`. Add separate colors to each bar.

Make a boxplot showing how `Precipitation` varies by `Month`. Make the boxplot horizontal, and use the three-letter text for each month (e.g. Jan, Feb,...) instead of numbers as tick labels. HINT: Use the option `las` to orient labels properly. What does this boxplot tell you about `Precipitation`?

If you inspect the data.frame you will see there are many days of missing observations (`NA`). If we ignore the three first columns of `daily.weather` (dates),

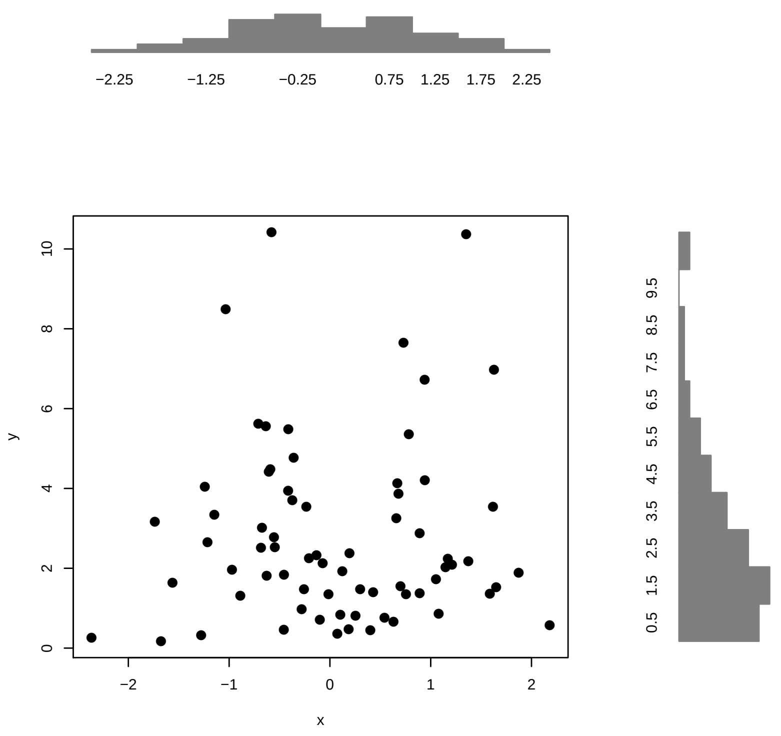


Figure 9.16: A scatter plot with histograms in the margin.

there are 13 columns of weather observations. Let us make a matrix called `Missing`, having 13 columns and the same number of rows as `daily.weather`, and name the columns as in `daily.weather`. This matrix should contain only the values 0 or 1. Fill in a 1 in row `i` and column `j` if the corresponding cell of `daily.weather` has a missing observation, 0 otherwise. Compute the column sums (use `colSums`) and make a barplot of this. Compute the row sums and make a lineplot of this. Make both plots as two panels in the same plot window.

Chapter 10

Handling texts

R is a tool developed for statistics and data analysis, and computations and 'number-crunching' is an essential part of this. However, in many applications we also need to handle texts. Texts are often part of data sets, as categorical indicators or as numeric data 'hidden' in other formats. In the field of bioinformatics the data themselves are texts, either DNA or protein sequences, usually represented as long texts. Thus, in order to be able to analyze data in general we should have some knowledge of how to handle texts in an effective way.

10.1 Some basic text functions

Texts can be compared using the comparison operators, just as numbers. Notice that R is case-sensitive, i.e. `"This"=="this"` is `FALSE` since the first letter differ. The ordering of texts depends on your computers *locale*, i.e. a computer set up with Norwegian language may behave different from one with American, etc. For letters in the English language (a to z and A to Z) there are usually no surprises, they are sorted in alphabetical order, and lowercase before uppercase. Symbols outside these letters are more uncertain, check your computer before assuming anything. A shorter text is sorted before a longer text, if it is a subtext:

```
> "Thi"<"This"
[1] TRUE
> "his"<"This"
[1] TRUE
```

The number of characters in a text is counted by the function `nchar`:

```
> nchar("This is a test")
[1] 14
```

This function can also take as input a vector of texts, and will return the length of each in a vector of integers:

```
> words <- c("This","is","a","test")
> nchar(words)
```

```
[1] 4 2 1 4
```

Notice the difference between `length(words)` and `nchar(words)`. The first give you the length of the vector (in this case 4), while the second gives you the number of characters in each element of the vector.

The functions `tolower` and `toupper` will convert to lowercase and uppercase text, respectively. Again they can take a vector as input, producing a vector as output:

```
> toupper(c("This","is","a","test"))
[1] "THIS" "IS"   "A"    "TEST"
```

10.2 Merging texts

We use the function `paste` to merge texts. This function can behave differently depending on the input and the options `sep` and `collapse`. Here is an example:

```
> name1 <- c("Lars","Hilde","Thore")
> name2 <- c("Snipen","Vinje","Egeland")
> paste(name1,name2)
[1] "Lars Snipen"   "Hilde Vinje"   "Thore Egeland"
```

Notice how we gave 2 vectors of texts as input, and got a vector of texts as output. The first elements are merged with a single space between them, and similar for all other elements. The `sep` option specifies the separating symbol (a single space is the default choice):

```
> paste(name1,name2,sep="_")
[1] "Lars_Snipen"   "Hilde_Vinje"   "Thore_Egeland"
```

If you want no symbol at all, just use `sep=""` (empty text).

The `paste` function also takes non-texts as input, and convert them to text before merging them. It will also circulate vectors, as we have seen before, i.e. if one of the vectors is shorter than the other, its elements are re-used to make it long enough. We can use this to our benefit, as shown in this example:

```
> paste("Observation",1:4,sep=" ")
"Observation 1" "Observation 2" "Observation 3"
"Observation 4"
```

Here our first input is a single text (vector with 1 element). The second input is a numeric vector with 4 elements. The first vector is then re-used 4 times. The second vector is converted to text, and finally they are merged, using a single space as separator.

If we want to merge all elements in a vector into a single text, we use the `collapse` option:

```
> paste(name1,collapse="_")
[1] "Lars_Hilde_Thore"
```

If you want no separator between the elements, again use the empty text:

```
> paste(c("A","A","C","G","T","G","T","C","G","G"),
        collapse="")
[1] "AACGTGTCGG"
```

10.2.1 Special characters

There are some special characters that we often make use of when constructing texts. The two most popular are `"\t"` and `"\n"`. The `"\t"` is the tab (multi-space) and `"\n"` is the newline (linebreak). These characters will affect the printing of a text, as shown here:

```
> txt1 <- paste(c("This","is","\t","a","test"),collapse="")
> txt2 <- paste(c("This","is","\n","a","test"),collapse="")
> cat(txt1)
Thisis  atest
> cat(txt2)
Thisis
atest
```

10.3 Splitting texts

The function `strsplit` will split a text into many subtexts. The option `split` specifies at which symbol we should split. Here is an example:

```
> strsplit("This is a test",split=" ")
[[1]]
[1] "This" "is"   "a"    "test"
```

Notice from the output here that it is a list of length 1, having a vector of 4 elements as its content. This function will always output a list. The reason is that it can split several texts, and since they may split into a different number of subtexts, the result must be returned as a list. Here is a short example:

```
> strsplit(c("This is a test","and another test"),split=" ")
[[1]]
[1] "This" "is"   "a"    "test"

[[2]]
[1] "and"     "another" "test"
```

Both input texts are split, resulting in 4 subtexts in the first case and 3 subtexts in the second.

Notice also that the splitting symbol (here we used a space `" "`) has been removed from the output, only the texts *between* the splitting symbols are returned.

If we give one single text as input to `strsplit`, the 'list-wrapping' is pointless and we should eliminate it to simplify all further handling of the output. To do this we use the `unlist` function. This takes as input a list, and returns the content of the list after all 'list-wrapping' has been peeled off:

```
> unlist(strsplit("This is a test",split=" "))
[1] "This" "is"   "a"    "test"
```

The `unlisting` of lists should only be done when you are absolutely certain of what you are doing. It is in general safe to use when the list has 1 element only. If the list has more than one element it can produce strange and undesired results.

In many cases the result of `strsplit` is a list of several elements, and we should in general not use the `unlist` function. We will discuss later in this chapter how to deal with lists in an effective way, since many other text-manipulating functions also produce such lists.

10.4 Extracting subtexts

We can extract a subtext from a longer text by specifying the start and stop position along the text:

```
> substring("This is a test",5,10)
[1] " is a "
```

Here we extract character number 5,6,7,8,9 and 10 from the input text. Both `substr` and `substring` will extract subtexts this way, but the latter is slightly more general, and the one we usually use. Here is an example of how we can extract different parts of a text in a quick way:

```
> dna <- "ATGTTCTGATCT"
> starts <- 1:(nchar(dna)-2)
> stops <- 3:(nchar(dna))
> substring(dna,starts,stops)
[1] "ATG" "TGT" "GTT" "TTC" "TCT" "CTG" "TGA" "GAT" "ATC" "
   TCT"
```

The first subtext is from position 1 to 3, the second is from 2 to 4, the third is from 3 to 5, etc. The function `substring` will circulate the first argument, since this is a vector of length 1, while argument two and three are vectors of many elements. It is possible to provide `substring` with many input texts, and then specify equally many starts and stops, extracting different parts of every input text.

10.5 Regular expressions

A regular expression is a syntax to describe patterns in texts. You will find regular expressions in many programming languages, and the R syntax is adopted from Perl. We will first look at some functions using regular expressions, and then see how to build such expressions.

10.5.1 Functions using regular expressions

The simplest function using regular expression is `grep`, which is similar to the one found in UNIX-like operating systems. This function will search for a given expression in a vector of texts, and output the index of the vector where it finds the expression. Here is a short example:

```
> grep(pattern="is",x=c("This","is","a","test"))
[1] 1 2
```

The first argument is the pattern, in this case simply the text `"is"`. The second argument is the vector of texts in which we want to to search. The pattern `"is"` is found in element 1 and 2 of this vector (in `"This"` and in `"is"`), and the output is accordingly.

The `grep` function does not tell us where in the texts we found the pattern, just in which elements of the vector it is found. The function `regexpr` is an extension to `grep` in this respect:

```
> regexpr(pattern="is",text=c("This","is","a","test"))
[1]  3  1 -1 -1
attr(,"match.length")
[1]  2  2 -1 -1
attr(,"useBytes")
[1] TRUE
```

The basic output (first line) is a vector of four elements since the second argument to `regexpr` also has four elements. It contains the values 3, 1, -1 and -1. This indicates that in the first text vector element the pattern is found starting at position 3, in the second text vector element it is found starting position 1 and in the last two text vector elements it is not found (-1).

After the basic output we see `attr(,"match.length")` and then another vector of four elements. This is an example of an *attribute* to a variable. All R variables can have attributes, i.e. some extra information tagged to them in addition to the actual content. We have seen how variables can have names, and this is something similar. In this case the output has two attributes, one called `"match.length"` and one called `"useBytes"`. The first indicates how long the pattern match is in those cases where we have a match. Since our pattern is the text `"is"` the `"match.length"` must always be 2 (or -1 if there are no hits). See `?regexpr` for more on `"useBytes"`. Note that attributes like these are just extra information attached to the variable, the main content are still the four integers displayed in the first line.

The function `regexpr` also has a limitation, it will only locate the *first* occurrence of the pattern in each text. In order to have *all* occurrences of the pattern in every text we use `gregexpr`. Consider the following example:

```
> DNA <- c("ATTTCTGTACTG","CCTGTAACTGTC","CATGAATCAA")
> gregexpr(pattern="CT",text=DNA)
[[1]]
[1]  5 10
attr(,"match.length")
[1] 2 2
attr(,"useBytes")
[1] TRUE

[[2]]
[1] 2 8
attr(,"match.length")
[1] 2 2
attr(,"useBytes")
[1] TRUE

[[3]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"useBytes")
[1] TRUE
```

We first notice that the output is a list (since we have the double-brackets [[1]]). The list has the same number of elements as we had in the vector of the second input argument (three). Each list element contains a result similar to what we saw for `regexpr`, but now it is one result for each hit in the corresponding input text. Like element two, showing that in the second input text `"CCTGTAACTGTC"` we find the pattern `"CT"` at position 2 and 8.

A frequent use of regular expressions is to replace a pattern (subtext) with some text. The function `sub` will replace the first occurrence of the pattern, while `gsub` replaces all occurrences, and it is this latter we use in most cases. We can illustrate using the previous example:

```
> DNA <- c("ATTTCTGTACTG","CCTGTAACTGTC","CATGAATCAA")
> gsub(pattern="CT",replacement="X",x=DNA)
[1] "ATTTXGTAXG" "CXGTAAXGTC" "CATGAATCAA"
```

The text we use as `replacement` can be both shorter or longer than the pattern. In fact, using `replacement=""` (the empty text) will just remove the pattern from the texts. Replacing a pattern with nothing (removing) is perhaps the most frequent use of `gsub`.

10.5.2 Building patterns

We will only give a very short introduction to regular expressions, as this is a large topic and not very specific to R. We have so far just used a text as a pattern, and this is the strictest possible pattern where only exact matching is allowed. The idea is that we can allow for some degrees of freedom to allow partial matching.

Here are some of the frequently used elements of the regular expression syntax in R:

`"AG[AG]GA"`	The brackets mean *either or*, i.e. either `"A"` or `"G"`. This pattern will match both `"AGAGA"` and `"AGGGA"`.
`"AG[^TC]GA"`	The 'hat' inside brackets mean *not*, i.e. neither `"T"` nor `"C"`. This will match the same as above, plus subtexts like `"AGXGA"` or `"AG@GA"`.
`"AG.GA"`	The dot is the *wildcard* symbol in R, and means *any symbol*. Be careful with `"."`, it could make the pattern too unspecific! (matches everywhere)
`"The[a-z]"`	A range of symbols, here all lower-case English letters. Other frequently used ranges are `"A-Z"` and `"0-9"`.
`"^This"`	A 'hat' starting the expression means matching at the start of the text only.
`"TA[AG]$"`	A 'dollar' ending the pattern means matching at the end of the text only.
`"NC[0-9]+"`	A 'plus' means the previous symbol or group of symbols can be matched multiple times. This is typically used to include an unspecific number in a pattern, i.e. both `"NC1"` and `"NC001526"` will match here.

There are of course many more possibilities, and `?regex` will show you the help file on this. The last example above also shows why we can have matches of different lengths, hence the need for the `"match.lengths"` attribute in `regexpr` and `gregexpr`.

10.6 Traversing lists

We have seen that both `strsplit` and `gregexpr` give lists as output. Both these functions are frequently applied to long vectors of texts, and the results will be long lists. In order to extract the content from these lists we need to *traverse* these lists in some way, i.e. loop through them and do something to the content of each list element. We have previously mentioned how we should try to avoid explicit looping in R, since this can be slow. Here are some examples of how to traverse lists by the use of the function-family `lapply`, `sapply` and `vapply`.

These functions typically take as input a list and a function to be applied to each list-element. This function is something we can make ourselves. Here is an example where we first use `gregexpr` on a (long) vector of texts, and then use `sapply` to extract the number of hits in each text.

First we make the function to apply:

```
count.hits <- function(x){
  n.hits <- sum(x>0)
  return(n.hits)
}
```

We saw from the example on `gregexpr` above that each list element it returns has a vector indicating with positive numbers each hit. If there are no hits, the vector contains only the value `-1`. Thus, the simple `sum(x>0)` will give us the number of hits. Notice that we expect the input to this function (`x`) to be the content of a list element produced by `gregexpr`.

After this function has been `source`d into the R workspace, we can use it in `sapply`:

```
> DNA <- c("ATTTCTGTACTG","CCTGTAACTGTC","CATGAATCAA")
> lst <- gregexpr(pattern="CT",text=DNA)
> sapply(lst,count.hits)
[1] 2 2 0
```

The pattern `"CT"` is found twice in input text one and two, and zero times in input text three. The function `sapply` loops through the list, sending the content of each list element as input to the provided function `count.hits`. This function returns exactly one integer for each list element, and then `sapply` outputs these results as a vector of integers.

Notice that the output from `sapply` is a vector, not a list. This makes it possible to extract something (depending on the function you apply) from each list element, and put the result into a vector. This of course requires that one quantity or text is extracted from every list element, regardless of what it contains.

10.6.1 In-line functions

In real R programs the code is often shortened compared to the version we have shown here. In fact, the creation of the function to apply can be done *in line* directly as you call `sapply`, like this:

```
> sapply(gregexpr(pattern="CT",text=DNA),
         function(x){sum(x>0)})
[1] 2 2 0
```

Notice how the in-line function has no name, it exists only during the call to `sapply`. This way of writing R code makes programs less readable for newcomers. The detailed version above is easier to understand, but this latter approach is something you will meet in real life.

10.7 Example: Reading many files

Often we want to read data from many different files. This can be cumbersome and time-consuming to do manually. Instead we make a script that reads the files, one by one.

10.7.1 Systematic filenames

Sometimes the filenames follow some system, and then we can easily re-construct the filenames in R. Let us assume we want to read the files `Data10.txt`, `Data20.txt`, `Data30.txt`,...,`Data110.txt`, `Data120.txt`. Each file is a formatted text-file, and we can read them all by the same call to `read.table`, we only need to change the filename each time.

We then make a `for`-loop going from 10 to 120 by step 10, and use the `paste`-function to create the names:

```
for(k in seq(from=10,to=120,by=10)){
  fname <- paste("Data",k,".txt",sep="")
  dta <- read.table(fname) #additional options to read.table
       may be needed
  #extract what you need from dta before next iteration
  #because then dta will be over-written...
}
```

Actually, it is often a good idea to first read one file, and then create the data structures you need to store whatever you want from the file. Then you proceed with the loop (starting at the second file) and read the files, adding their content to the existing data structure.

10.7.2 Listing folder content

Sometimes the files have no systematic name. Let us say we have the data files in the folder `C:\Download\Data`. In this folder we have some formatted data-files, and the only common about their names are they all end with the extension `.txt`. The folder also contains many other files, with different extensions.

The function `dir` in R will list all files and subfolders in a given folder. If you in the Console window just type

```
> dir()
```

you will get listed all files and folders in the current working directory. By specifying a folder, e.g. `dir("C:/Download/Data")` the function returns a text-vector containing the names of all files and folders in `C:\Download\Data`. NOTE! In R we use the slash (`/`) but in Windows we use the backslash (`\`) when specifying a path. Here are some lines that indicates how to read the data-files in `C:\Download\Data`:

```
all.files <- dir("C:/Download/Data")
idx <- grep("txt$",all.files) #use regular expression to
                              #locate the txt-files
data.files <- all.files[idx]
for(fname in data.files){
  dta <- read.table(fname)
  #extract what you need from dta before next iteration
  #because then dta will be over-written...
}
```

Notice that we used `grep` to select only the filenames ending with `.txt`. In some cases this is not enough, there may be many `.txt`-files, and we only want to read some of them. Then we need to make another `grep` where we look for some pattern found only in the names of those files we seek.

10.8 Important functions

Command	Remark
`nchar`	Number of characters in a text (vector)
`toupper,tolower`	Converting text to uppercase/lowercase
`paste`	Merging texts
`strsplit`	Splitting texts
`unlist`	Removes the list-property of a list, if possible
`substring,substr`	Retrieves a sub-text from a text
`grep`	Search with a regular expression in a text (vector)
`regexpr`	Finds position of first occurrence of a regular expression
`gregexpr`	Finds position of all occurrences of a regular expression
`gsub`	Replacing a regular expression with text
`sapply`	Applies a function to elements in a list
`dir`	Lists folder content

10.9 Exercises

10.9.1 Poetry

Can R understand poetry? Well, at least we can make R-programs recognize poetry to some degree. We will pursue this exercise when we come to modelling in later chapters.

In the file `poem_unknown.txt` you find a poem written by some unknown author. According to some expert in English literature the author of this poem is very likely to be either Shakespeare, Blake or Eliot. To investigate this, we need to convert this poem into a 'numerical fingerprint'.

Make a script that reads the file `poem_unknown.txt` line by line using the `readLine` function from chapter 6. Also, load the file called `symbols.RData`. This creates the vector `symbols` in your workspace. This vector has 30 elements, all single character texts.

Make a function called `poem2num` that takes as input a vector of symbols, like `symbols` and a poem (text vector). The function should return the number of occurrences of each of the symbols in the poem. Remember to convert all text in the poem to lower-case. Use the poem and the symbols from above to test the function. It should return a vector of 30 integers. This is our (simplistic) version of the poems numerical fingerprint.

10.9.2 Dates

In the weather data we have seen each date is registered by its Day, Month and Year as three integers. In the raw data files this was not the case. Most years the date is given like `"01.01.2001"` (day-month-year). Typically, the format has changed over the years, and in some cases the year is only written with the last two digits, e.g. `"01.01.93"` for 1. January 1993. In some years the format is `"01011998"` (no dots). It is quite typical that rawdata are messy like this, and some kind of programming is needed to get it all into identical formats.

Make a function that takes as input texts like all the three mentioned above, and extracts the date as three integers, Day, Month and Year, and return these in a vector of length 3. The year must be the 4-digit integer (1993, not just

93). It means the function should recognize these three formats, and behave accordingly.

Chapter 11

Packages

11.1 What is a package?

We can think of a package in R as a bundle of R-functions, data files, documentation files and possibly some external (non-R) programs that we can install on our computer to extend the basic capabilities of R. The idea behind this is to make it easy to write extensions to R, and share these within a user community. When you install R you do not install a huge monolithic program. Instead, you install a basic tool-set that can be extended and specialized in various ways. As you become more familiar with R you may want some more tools. Often you will find that others have made these tools already, or at least something quite close to what you need. If these are provided in an R package, you can easily import them, and use them or modify them according to your need.

We will see below how easy it is to import code from others through packages. We will also have a brief look at how we could create our own package, for sharing with others.

11.2 Default packages

When you install R you also install the `base` package. This contains the most basic functions in R, functions you simply cannot do without. In a default installation you also get a number of other packages. These packages have been considered by the *R core team* to be so essential that any installation of R should have them. If you start RStudio, and locate the Package-vane in one of your panes, you will get a list of all installed packages on your system.

If you click on the name of some package, you should be taken to the documentation for that package. This documentation may vary slightly from package to package, there are some degrees of freedom in how much and how detailed a package must be documented. First, there should be a DESCRIPTION file, a short text file containing some very basic description. You may for some packages also find a link to what is called a *vignette* as well as a *users guide*. Both these (if supplied) are longer descriptions or user guides usually worth reading. Next, you should expect to see a list of all functions and all data sets in the package. By clicking these you are taken to the Help-files for each function/data set. These are the same Help-files you see if you type `?` and the function name in the Console window.

11.3 Where to look for packages?

The official repository for R-packages is the Comprehensive R Archive Network (CRAN, see `http://cran.r-project.org/`). By default R is set up to look for packages at CRAN. CRAN has many mirror-sites around the world with the same content. Packages found at CRAN have passed a quality-check and are regarded as safe with respect to virus and other problems.

Since R has grown to be so popular, there are now many other sites on the internet providing huge amounts of R-packages. The R-forge (`https://r-forge.r-project.org/`) is a website where you can store and share your R-packages. If many people work together to build the same package, sites like this are used for *version control*, i.e. all developers work on a copy of the original package, and new versions are uploaded and downloaded from this central site. Packages developed at R-forge may be open for anyone to download and install. These packages do not have the same quality-control as the CRAN-packages.

Another popular site for obtaining R-packages related to computational biology is the Bioconductor (`http://www.bioconductor.org/`). This site provides its own installation-function (called `biocLite()`) that you use to install packages from their site. We will have a look at this below.

R is already used in many scientific fields, and is still expanding, which means I have no overview of all these communities. There are most likely many R-sites you will find useful that I have never even heard of!

11.4 Installing packages

The basic way of installing a package in R is to use the function `install.packages()`. This can be used directly in the Console window, but as long as we use RStudio we might just as well invoke it from the menu: From the menu select `Tools` and then `Install Packages...`. A small window should pop up. In this window you first select from where you install the package. By default there should be two choices: Either directly from CRAN or from some archive-file that you have downloaded.

11.4.1 Installing from CRAN

Let us install the package called `pls` from the CRAN. Needless to say, you must have an internet connection to do this! Select the CRAN in the Install from field. You may also need to select which mirror site to use (select the Norwegian site when in Norway).

Start typing the name of the package in the next field. RStudio will show you all available packages fitting with the name you type.

Next, there is a pop-up menu for selecting the library (Install into Library). A library is, in this context, just a directory on your system where R will install packages. When you install R, the default directory is selected, and in many cases this is fine. See `?Startup` for more Help on this. Let us use the default choice for now.

Finally, there is a check-box where you can decide to install dependencies. Many packages need other packages to work properly, and if these are not already in your system they will also be installed if you tick this box. Usually this is a good idea.

Click the Install button, and the package, along with any dependencies, are installed. This may take some time since we need to first download from CRAN and then unpack and also possibly compile some code. If the installation was successful, the package `pls` should now appear on your list of packages in the Package-vane.

11.4.2 Installing from file

On R-forge there is a project called *The forensim package* (see under Projects-Project Tree-Bioinformatics-Statistics). We happen to know the people behind this project, and we feel safe downloading this package. From the project site we click on the R Packages menu, and download the zip-archive (or tar.gz archive). We should now have the file `forensim_4.3.zip` (or `forensim_4.3.tar.gz`) in our Download-folder.

Again we use the `Tools` and `Install Packages...` in RStudio, but instead of CRAN we select to install from a Package Archive File. Browse to locate the archive in your Download-folder, and install the package. This package depends on another package called `tkrplot`, and unless you have this in your system the installation will fail. This package can be installed from CRAN. Install `tkrplot` in the same way as you did for the `pls` package above, and then repeat the installation of the `forensim` package.

11.4.3 Installing from the Bioconductor

As mentioned above, the Bioconductor has its own installation script. Let us show how we use it. We want the package called `bioDist` from the Bioconductore. Then we type the following directly in the console window:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("bioDist")
```

and the package is installed!

11.5 Loading packages

You only install a package once on your computer. However, the package is not available for use in an R session yet. You always need to load the package before you can use it, and this must be done once in every R session. You typically load a package by the statement

```
library(pls)
```

where the keyword `library` is used to load the `pls` package. You can also use `require` in the same way.

If I make a script where I know that I will use some functions from a certain package, I always make certain the required `library` statements are entered at the start of the script. This guarantees that the packages I need have been loaded before the functions are being used later in the script. You do not need to load the package more than once, but there are no problems executing the `library` statement repeatedly, it is just ignored by the system if the packages has already been loaded.

In the Package-vane in RStudio all currently loaded packages have their check-box ticked.

11.6 Building packages

In this course we will only have a very brief look at how to create an R-package. The reason is of course that most package developers need more experience in R programming. On the other hand, it is nice to have seen this once at an early stage in order to understand a little bit more about packages. Also, in RStudio it is very simple to get started creating packages.

Before you start building packages in Windows, you need to install the *Rtools* that you find on the same web-page as R itself, see `http://cran.r-project.org/`

11.6.1 Without RStudio

There are some built-in functions in R that can help you creating a package. More specifically, the function `package.skeleton` will be helpful. There is also a huge documentation about how to create R packages, see the link Writing R Extensions in the main R Help window.

11.6.2 Using RStudio

In this brief example we will use the facilities in the current version of RStudio (ver 0.98.501). Any package you create must have its own directory somewhere on your computer. The name of the package is the name of this directory. In such a package-directory we must put certain files and subdirectories that R require to turn this into a package. We will use RStudio to create this structure for us.

From the `File` menu, select `New Project...`. A small window should pop up. Select to create the project in a New Directory. In the next window, select R Package. In the final window you need to give the package a name, and also select where in your file-tree the package should reside. Press the Create Project button, and the directory is created, along with all the required files and subdirectories.

If you have a look at the Files-vane in RStudio, you should now see the package-directory content. There will typically be two files called `DESCRIPTION` and `NAMESPACE`. There are also typically two subdirectories called `R` and `man`. There will also be some other files, at least one with the extension `.proj` which is something RStudio adds to any Project-folder. If you want to open an existing Project (package or something else) in RStudio, this is the file you look for.

The R subdirectory

This is where you put all R-programs of the package. All functions in the package must be defined in files with the `.R` extension and put into this directory. Nothing else should be added to this subdirectory.

The man subdirectory

This is where you put all the Help-files. A huge part of any package development is to create the Help-files required for every function and every data set in the package. The Help-files must all follow a certain format. In RStudio we can create a skeleton for a Help-file from the `File` menu. Select `New File`, but instead of choosing `R Script` as before, we scroll down to the `Rd File` option. A small window pops up asking for the Topic. This is usually the name of the function that we want to document. Type in the name, and a file is created containing the skeleton for an R documentation file. You will recognize most of the sections from any Help-file in R. Filling in files like these is a significant job in every R package development. In the `man` directory there will typically be one such file for each function, but in some cases (as we have seen) very related functions may be documented within the same file.

Other subdirectories

We may add other subdirectories to our package-directory, but there are some rules we must obey. If the package contains data these are usually saved in `.RData` files (using the `save()` function, see chapter 6). Such files should be placed in a subdirectory called `data`. Such data sets can be accessed by the function `data()` once the package has been loaded. If we have loaded a package containing the data set named `daily.weather.RData`, we can load it by

```
> data(daily.weather)
```

We can also have subdirectories for other types of data, for external (non-R) programs etc, but you will have to read about them from sources outside this text.

11.6.3 How to create the package archive?

Once we have written all the R-programs, written all Help-files, created data files and other thing belonging to our package, we must bundle this all together to form a package archive that we share with the rest of the world.

If you have the package-project open in RStudio there should now be a Build-vane that give you the tools required. The Build & Reload button will 'build' the package and load it in your currently running R session. This is nice for quickly checking that things work as they should during development.

Once you are ready to deploy your code, you can choose the More pop-up and select Build Source Package. This should result in an archive file in the parent-directory of your package-directory.

11.7 Important functions

Command	Remark
`install.packages`	Installs packages
`library`,`require`	Loads a package
`package.skeleton`	Creates a folder with package structure
`data`	Loads data available in a (loaded) package

11.8 Exercises

11.8.1 Imputation of data

Larger data sets will often have some missing data, for various reasons. If we want to apply some multivariate statistical methods to our data, these will in general not tolerate missing data. Think of the data set as a data.frame or matrix. If the data in cell `[i,j]` is missing we must either discard the entire column `j`, the entire row `i` or *impute* the value in this cell. The latter means finding some realistic and non-controversial value to use in this cell.

Load the matrix in the file `weather.matrix.RData`. This is a small subset of the weather data we have seen before. It is a matrix (not data.frame!) with 5 columns and 22 rows. Notice that column 3 (Humidity) has two missing data (`NA`).

Imputation by the mean

It is not uncommon to impute missing data by the mean value of the variable. Do this by computing the mean of the Humidity-column (ignore the NA's). Both missing cells will then get this value. This procedure will work fine if the variable Humidity has the exact same expected value for all days (rows), and the only contribution to variation is a completely random day-to-day fluctuation. If this is the case, the imputation by the mean makes sense.

Imputation by KNN

However, the assumption that *the only contribution to variation is a completely random day-to-day fluctuation* seems too severe. It is reasonable to assume that Humidity varies *systematically* and not at random, by how the other weather variables behave, at least to some degree. For instance, on a rainy day (much Precipitation) it is reasonable to assume that the Humidity should be quite high compared to a clear day. Notice that other weather variables have been observed on those days where Humidity is missing, and we should make use of this information.

The K-nearest-neighbour (KNN) imputation method uses the values for the other variables to find the most likely missing values, i.e. based on the values for Air.temp, Soil.temp.10, Precipitation and Wind, find other days in the data set with similar values, and use the Humidity for those days to compute the imputation value (e.g. the mean of these Humidity-values).

In the Bioconductor package `impute` you find a function for doing this. Install the package, load it and read the Help-file for the function `impute.knn`. This has been purpose-made for gene expression data, but any matrix can be used actually. Use `impute.knn` to impute the two missing data in `weather.matrix` . HINT: By default `impute.knn` uses `k=10` neighbours, try to use `k=3` instead (since our data set is small).

11.8.2 Making a package

We will create a small R-package around the weather data set we have seen in previous exercises. The package will contain this data set and some related functions.

First, create a new project with an R-package, as described above. Add a new directory named `data` to the package-directory, and put `daily.weather.RData` (from previous exercises) in this subdirectory. In the `R` subdirectory we will have a function named `plotTrends`. Create the file with this function:

```
plotTrends <- function(dwd){
  month <- c("Jan","Feb","Mar","Apr","May","Jun","Jul",
              "Aug","Sep","Oct","Nov","Dec" )
  par(mfrow=c(3,4))
  for(m in 1:12){
    idx <- which(dwd$Month==m)
    y <- tapply(dwd$Air.temp[idx],dwd$Year[idx],mean,
                 na.rm=T)
    x <- as.integer(names(y))
    plot(x,y,pch=16,cex=1,col="cyan",xlab="",
         ylab="Temperature",main=month[m])
    lfit <- loess(y~x)
    y.hat <- predict(lfit)
    points(x,y.hat,type="l",lwd=2,col="blue")
  }
}
```

Create Help-files for the function `plotTrends` above, as well as the data set in `daily.weather.RData`. Save these in the `man` subdirectory.

Build and load the package, first locally in RStudio and then build the package-archive.

Chapter 12

Data modeling basics

By data modelling we mean the process of finding trends, relations or patterns in data. This is the setting for many data analysis problems. Often we want to *predict* the value or outcome of some variable given some observations of some other, hopefully related, quantities. In many cases the predictions are themselves a goal, but just as often it is the relation between variables we are interested in, and the ability to make good predictions (small errors) is a criterion for finding the 'real' relations. In this chapter we will consider some modelling situations, and see how we can approach these using simple R programming.

In this chapter we introduce some simple regression and classification methods. The focus is on how to use R for performing such analyses, and we will not dig into the theory behind these methods. We actually assume you have some basic knowledge of this from previously. The free book *An Introduction to Statistical Learning* (AISL) is a very good text for those who want to repeat some of the basic theory behind regression and classification methods. It also gives a number of motivating examples, and has separate sections introducing the same basic R functions as below. We recommend that you take a look at the first 4-5 chapters of AISL together with this chapter.

12.1 Data sets

The data sets we will consider in this chapter are all of the type that could be stored in a data.frame. More specifically, when we talk of *data variables* we can think of columns in a data.frame. The variables we will consider are of two data types: Either numeric (or integer) or factor. When it comes to actual computations, the variables of interest are often retrieved from the data.frame and put into a matrix, since matrix is the desired input format to many functions.

12.1.1 Explanatory and response variables

It is customary to distinguish between *explanatory* variables and the *response* variable. In the prediction problem, the explanatory variables are those we have observed while the response variable is the one we want to predict, based on the observed explanatory variables. We may have one or more explanatory variables. In this course we will only consider problems with a single response variable.

By convention, the explanatory variables are stored in a matrix named X in statistical literature. If we have all data in a data.frame named D, and we want to use columns 1 and 3 as explanatory variables, we can think of X as

```
X <- as.matrix(D[,c(1,3)])
```

It is also a similar convention to name the response variable y. We can think of y as a column-vector, i.e. if we want to use column 2 in D as response it means

```
y <- D[,2]
```

It should be noticed that as long as we use the basic functions in R introduced below, we never explicitly create neither `X` nor `y` since the function are built to extract them directly from the data.frame (`D`). But, we should at least imagine both X and y could have been created like above, it is part of the statistical language. In the book AISL you will find the same symbols.

Notice that the distinction between explanatory and response variables is purely operational. In one problem a certain variable may be the response, but in another the same variable can be among the explanatory variables. The data.frame D exists as a passive data source in the background, and we extract and use the variables we need depending on the problem.

12.1.2 Objects and variables

For all variables we have sampled a set of values, and if we also include the occasional missing value, all variables have the same number of samples. A *data object* refers to a row in the matrix X, i.e. data object i is $X[i,] = (X[i,1],...,X[i,p])$. Sometimes data objects also include the corresponding value of y, i.e. data object i is $(y[i], X[i,])$. It will (hopefully) be clear from the context whether the y-values are included or not.

Where does the word 'object' come from in this context? Traditionally, a data table is arranged such that columns (variables) correspond to *properties* and the rows correspond to some observed *object*. We have previously seen (and will see below) the bear data set. Each column in the data.frame are properties of bears (weight, length, gender, etc.), and each row is a new bear. In the weather data set we have seen each column is a property of a day (temperatures, wind, radiation etc.) and in each row we list the observation for each day. The bears or the days are the 'objects' we observe.

By convention, the number of data objects (rows in X) is named n, while the number of explanatory variables (columns in X) is named p.

12.1.3 Training and test data

In all modelling situations we can also divide the data set into what we call the *training set* and the *test set*. The training set are the data objects we use to fit our models, i.e. we use these data to learn from, and train our method to behave as good as possible. We often denote this data set (y_{train}, X_{train}). In the training set the values of the response y_{train} is always known. When we talk about the number of objects n we usually refer to the number of objects in the training set only.

The test set consists of the data objects (y_{test}, X_{test}), which are different observations of the same variables as in the training set. In principle the response y_{test} is missing/unobserved in this case, and we want to find it. We have the observed data in X_{test}, and based on the model we trained on the training data, we want to combine this with X_{test} to predict the values of y_{test}.

Both data sets are sampled from the same population, i.e. any relation that holds between X_{train} and y_{train} should also hold between X_{test} and y_{test}, and vice versa. Think of them as two subsets of the same data.frame.

In some cases the values of y_{test} are also known. We then pretend they are not, and predict them as if they were indeed missing. Then we can compare this prediction to the true values of y_{test}. This is a valuable exercise for evaluating the ability of a model to make good predictions, as we will see below.

The distinction between training data and test data is again purely operational, i.e. data objects used in the training set can in another exercise be used in the test set, and vice versa. In some cases the training data and test data are identical, i.e. we use the same data for both purposes. This is possible, but then we have to remember that the 'predictions' of y_{test} are no longer actual predictions, since the same data were used for training as well.

12.2 Regression

Let us repeat the regression idea for a very simple data set. In Chapter 6 we met a data set on bears, in the file called `bears.txt`. This data.frame contains some measurements made on 24 different bears. More specifically, there is a variable named `Weight`, which is the body weight of each bear. Weighing a bear can be quite cumbersome (and dangerous?), and it would be very nice if we could *predict* the weight of a bear from its body length. Measuring the length of a bear is, after all, much easier. Hence, we want to predict `Weight` based on some observed value of `Length`. We consider only these two variables from the data set, and the response variable y is the `Weight` and the only explanatory variable in X (which means $p = 1$) is `Length`. Notice that both variables are continuous, i.e. they may take any numerical value within a reasonable range of values.

We first use all $n = 24$ data objects as training data, and we will split into training and test data later. First, we read the data into R:

```
beardata <- read.table(file="bears.txt",header=TRUE)
```

12.2.1 Simple linear regression

In general, if we have some response variable y_{train} and a single explanatory variable in X_{train} the linear regression model assumes

$$y_{train} = \beta[1] + X_{train} \cdot \beta[2] + e \tag{12.1}$$

where $\beta[1]$ and $\beta[2]$ are unknown coefficients and e is some noise or error term. We usually refer to $\beta[1]$ as the intercept and $\beta[2]$ as the slope. The simple formula $\beta[1] + X_{train}\beta[2]$ describes a straight line, i.e. our model says that the relation between X_{train} and y_{train} is a linear relation. This means that if we had no noise (if e was absent) we could have plotted y_{train} against X_{train} and found all points on the same straight line. For real data there are always some deviations from the straight line, and this we assume is due to the random term e. The basic step of the data modelling is to find proper values for $\beta[1]$ and $\beta[2]$, i.e. to *estimate* these unknown coefficients from data. This is the training step.

In our case we now use the column `Weight` as y_{train} and `Length` as X_{train}.

12.2.2 Fitting a linear model using `lm`

We will not go into any details on the theory of estimation, these are topics in other statistics courses. Fitting the above linear model to data can be done by the function `lm` in R.

```
fitted.mod <- lm(Weight~Length,data=beardata)
```

The construction `Weight~Length` is a notation we see sometimes in R. It is what we call a *formula*. It simply means that the variable to the left of the `~` is the response variable to be related to the variables to the right of the `~`. It is another way of writing the formula from (12.1). Notice that we did not write `beardata$Weight` and `beardata$Length` here (we could have). The `lm` function has an option `data` where we can specify the name of the data.frame we refer to, and if we do this we can use the column names in the formula. Read about formulas in R (`?formula`), they are quite common in many R applications.

In Figure 12.1 we have displayed the data as well as the estimated regression line, i.e. the straight line relation corresponding to the estimates above. From Figure 12.1 we would say the fitted model is not very good. The slope is positive, which is fine, a longer bear should also mean a heavier bear, but the intercept is nonsense. Notice that a bear of 100 centimeters is predicted to weigh almost nothing, and even negative predictions of weights for very short bears! There are potentials for improvements here!

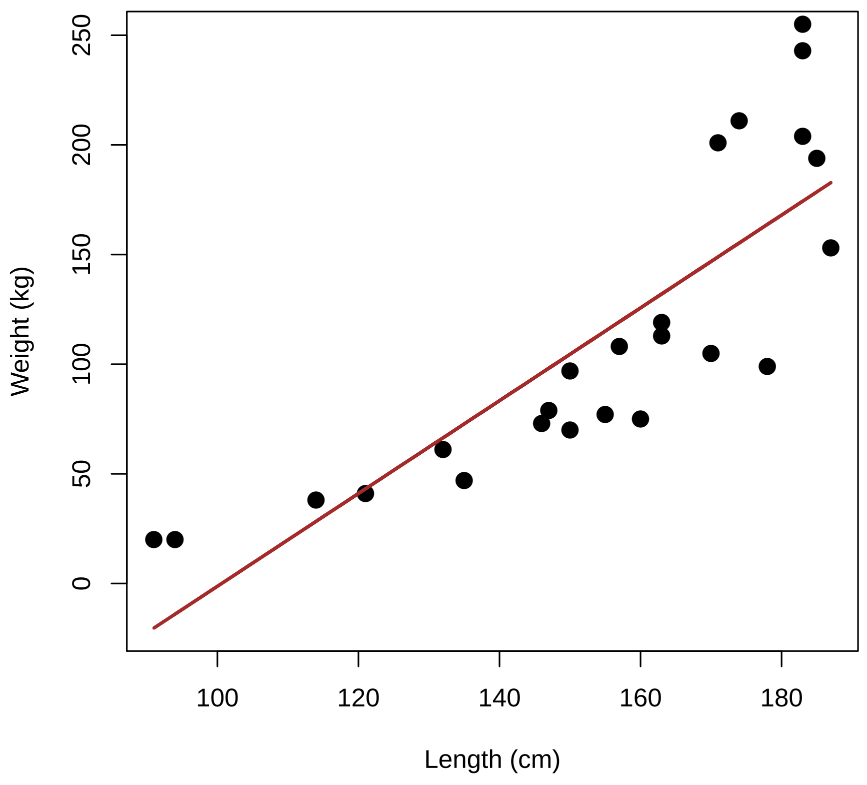


Figure 12.1: A scatter plot of Weight versus Length from the bears data set. Each marker corresponds to an object, having one Weight-value and one `Length`-value. The straight line is the fitted linear regression line, see the text for the details.

12.2.3 The lm-object

The function `lm` returns something we can call an lm-object. The function `class` tells us this:

```
> class(fitted.mod)
[1] "lm"
```

We can think of this as an ordinary list, but with some added properties. For those familiar with object oriented programming, we would say that the lm-object has inherited list. The `summary` function can take an lm-object as input, and produce some helpful print-out:

```
> summary(fitted.mod)

Call:
lm(formula = Weight ~ Length, data = beardata)

Residuals:
   Min     1Q Median     3Q    Max
-64.76 -26.78  -9.42  30.74  80.67

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -212.8522    47.2818  -4.502 0.000177 ***
Length         2.1158     0.3027   6.989 5.15e-07 ***
---
Signif. codes:  0   ***   0.001   **   0.01   *   0.05
        .   0.1        1

Residual standard error: 40.1 on 22 degrees of freedom
Multiple R-squared:  0.6895,  Adjusted R-squared:  0.6754
F-statistic: 48.85 on 1 and 22 DF,  p-value: 5.147e-07
```

In the Coefficients section of this print-out we find some t-test results for each of the coefficients of the model. These tests are frequently made to see if the explanatory variables have a significant impact on the response variable. In this particular case we would be interested in the null-hypothesis $H_0 : \beta[2] = 0$ versus $H_1 : \beta[2] \neq 0$. If H_0 is true it means `Length` has no linear relation to `Weight`. The print-out row starting with `Length` displays the result for this test. Clearly the p-value (`Pr(>|t|)`) is small ($5.15 \cdot 10^{-7}$), and there is a significant relation.

Another useful function for lm-objects is `anova`:

```
> anova(fitted.mod)
Analysis of Variance Table

Response: Weight
          Df Sum Sq Mean Sq F value    Pr(>F)
Length     1  78560   78560   48.85 5.147e-07 ***
Residuals 22  35380    1608
---
Signif. codes:  0   ***   0.001   **   0.01   *   0.05
        .   0.1        1
```

In this case we perform an F-test on the overall fit of the model. Read about this test in AISL.

We can also inspect the lm-object as if it was a straightforward list. First we take a look inside:

```
> names(fitted.mod)
 [1] "coefficients" "residuals" "effects" "rank"
 [5] "fitted.values" "assign" "qr" "df.residual"
 [9] "xlevels" "call" "terms" "model"
```

There is one variable named `coefficients`, and to retrieve these estimates we type

```
> fitted.mod$coefficients
(Intercept)      Length
-212.852240    2.115778
```

We can also retrieve the fitted values, and use these to plot the fitted line. Here is the code that produced the plot in Figure 12.1:

```
attach(beardata) # Read in ?attach to repeat...
plot(Length,Weight,pch=16,cex=1.5,xlab="Length (cm)",
     ylab="Weight (kg)",ylim=c(-20,250))
points(Length,fitted.mod$fitted.values,type="l",lwd=2,
       col="brown")
# Alternative to two last lines above:
#abline(fitted.mod,col="red",lwd=1.5)
```

The residuals are the differences between observed `Weight` and predicted `Weight` (the straight line). We often plot the residuals versus the explanatory variables, to see if there are any systematic deviations. Remember, the assumption of the linear model is that *all* deviations from the straight line are due to random noise. We can plot the residuals by:

```
residuals <- fitted.mod$residuals
plot(Length,residuals,pch=16,xlab="Length (cm)",
     ylab="Residual (kg)")
points(range(Length),c(0,0),type="l",lty=2,col="gray")
```

As seen in Figure 12.2 there is a tendency that the residuals are negative for medium `Length` values, and positive at each end. This indicates the assumptions of the linear model are too simple, and in this case it is the linear relation between that `Length` and `Weight` that is too simple.

12.2.4 Making predictions

One important motivation for statistical modeling is to make predictions. Assume we have observed a new bear, and found that its length is 120 cm. This is a test set data object $X_{test} = 120$. What is this bears weight? Our prediction of this, based on the existing model, would be

$$\hat{y}_{test} = \hat{\beta}[1] + \hat{\beta}[2] \cdot X_{test} = -212.9 + 2.116 \cdot 120 = 41.02 \tag{12.2}$$

where the ˆ indicates estimated values. In R we make predictions based on an lm-object by the function `predict`:

```
new.bear <- data.frame(Length=120)
Weight.predicted <- predict(object=fitted.mod,
                            newdata=new.bear)
```

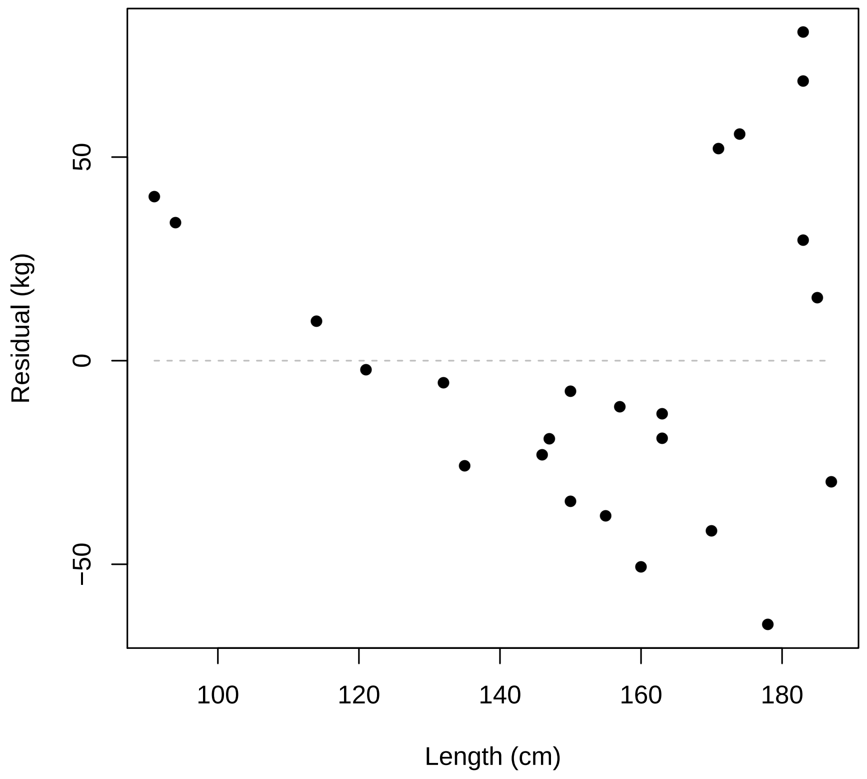


Figure 12.2: A scatter plot of the residuals versus Length.

Notice that `predict` takes as input the fitted model and a data.frame with the test set data objects. This data.frame must contain the same column names as we used in the model, in our case `Length`. You can have data for as many new data objects (bears) you like, and `predict` will compute the predicted y-value (`Weight`) for each.

12.2.5 Multiple linear regression

As we have already concluded, predicting bear weight from bear length in a simple linear model does not give very good predictions. We need to extend the model in some way, because the residual plot tells us there are systematic variations in weight that our model ignores. One obvious way of extending the simple linear model is to include more than one explanatory variable. If we have multiple explanatory variable we refer to this as multiple linear regression.

One extension is to add higher order terms to the model, e.g. a second order term of the single explanatory variable we have used so far. The general model would then look like

$$y_{train} = \beta[1] + X_{train}[,1] \cdot \beta[2] + X_{train}[,2] \cdot \beta[3] + e \tag{12.3}$$

where the matrix X_{train} now has two columns. The second column is just the first column squared. Using the bear data again, let us create a new data.frame containing only the variables of interest:

```
beardata2 <- beardata[,1:2] #retrieving Weight and Length
beardata2$Length.Sq <- (beardata2$Length)^2 #adding Length^2
```

The first 5 rows of this data.frame now looks like this:

```
> head(beardata2)
  Weight Length Length.Sq
1     38    114     12996
2     41    121     14641
3     47    135     18225
4    201    171     29241
5    243    183     33489
6    204    183     33489
```

We can fit a new model including the second order term like this

```
fitted.mod2 <- lm(Weight~Length+Length.Sq,data=beardata2)
```

If you run a `summary` on this object, you will see the second order term is highly significant. Notice also that the first order coefficient $\beta[2]$ has changed its sign! It is in general difficult to interpret coefficients as soon as you add second (or higher) order terms.

In Figure 12.3 we have plotted the fitted values as we did before, and it seems clear this model gives a better description of how weight is related to length. It is no longer a straight line, due to the second order term, and here are the lines of code we used to create this plot:

```
attach(beardata2)
plot(Length,Weight,pch=16,cex=1.5,xlab="Length (cm)",
     ylab="Weight (kg)",ylim=c(-20,250))
idx <- order(Length)
points(Length[idx],fitted.mod2$fitted.values[idx],type="l",
       lwd=2,col="brown")
```

Notice the use of the function `order`. This gives us an index vector that when applied to `Length` will arrange its elements into ascending order. When making a line-plot, the values along the x-axis must be in ascendig order, otherwise the line will cross back and forth, making it look like a spiders web! Try to make a plot of the residuals of this model versus `Length`.

Instead of using a higher order term, we could of course have used some of the other variables in the original data set in our model. The variable `Chest.G` (chest girth) measures how 'fat' the bear is. Together with `Length` this says

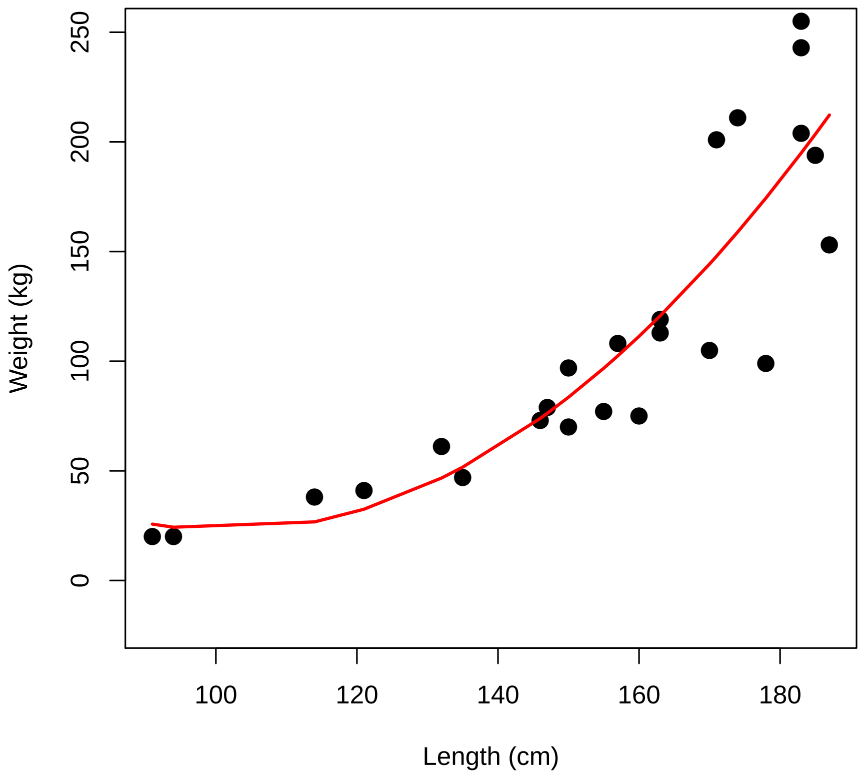


Figure 12.3: The fit of the model that includes a second order term of `Length`.

something about the size of the bear. Think of a cylinder, its volume is given by

$$V = \pi \cdot r^2 \cdot h \tag{12.4}$$

where r is the radius and h is the height (length) of the cylinder. Imagine a bear standing up, and visualize a cylinder where this bear just fits inside. Then `Length` is the height h and `Chest.G` is $\pi \cdot 2r$. Thus, the term `Length*Chest.G` should come close to describing the volume of this cylinder, and the bear's volume is some fraction of this. If we can (almost) compute the bear's volume, the weights is just a matter of scaling. Thus we try the model

```
volume.mod <- lm(Weight~Length+Chest.G+Length*Chest.G,
                 data=beardata)
```

Run the `anova` on `volume.mod` and compare it to similar outcome for `fitted.mod` and `fitted.mod2` from above. In Figure 12.4 we have plotted how the 'volume model' fits the data. Here is the code that produced this plot:

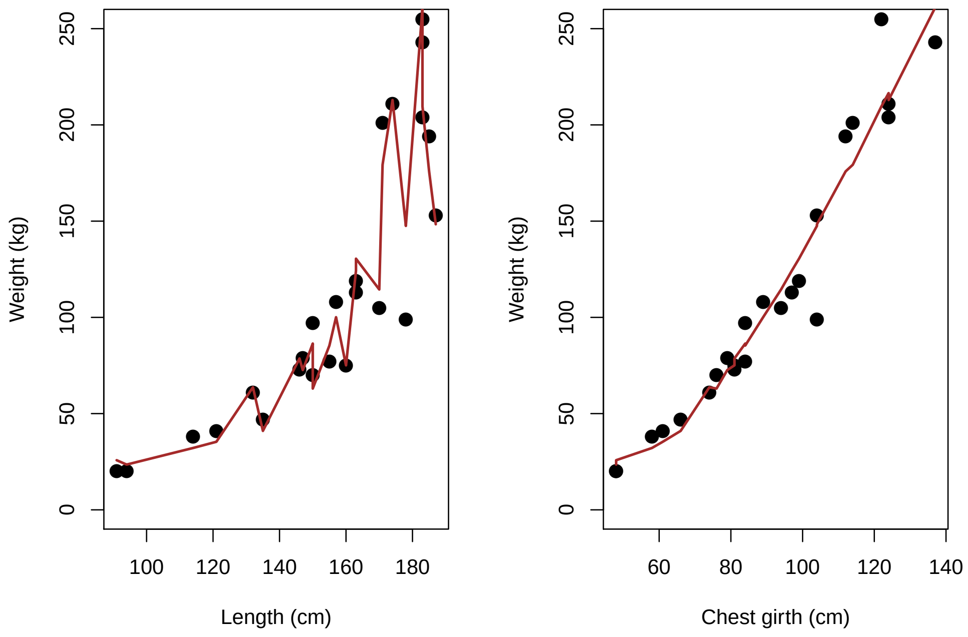


Figure 12.4: The fit of the model that includes both `Length` and `Chest.G`, referred to as the 'volume model' in the text. Here we have displayed the fit in two panels, once against `Length` and once against `Chest.G`.

```
attach(beardata)
par(mfrow=c(1,2))
plot(Length,Weight,pch=16,cex=1.5,xlab="Length (cm)",
     ylab="Weight (kg)",ylim=c(0,250))
W.predicted <- volume.mod$fitted.values
idx <- order(Length)
points(Length[idx],W.predicted[idx],type="l",lwd=2,
       col="brown")
plot(Chest.G,Weight,pch=16,cex=1.5,ylim=c(0,250),
     xlab="Chest girth (cm)",ylab="Weight (kg)")
idx <- order(Chest.G)
points(Chest.G[idx],W.predicted[idx],type="l",lwd=2,
       col="brown")
```

12.2.6 Regression with factors

So far we have only used numerical variables in our model. A regression model can also have factors among its explanatory variables (not in the response!). In the bears data set there is one factor variable, the `Gender`. As long as we use the `lm()` function, factors can be included in the model in the same way as a numerical variable. Let us illustrate this, by using a `Length` and `Gender` to predict `Weight`:

```
gender.mod1 <- lm(Weight~Length+Gender,data=beardata)
```

If we run `summary` on this fitted model we get

```
> summary(gender.mod1)

Call:
lm(formula = Weight ~ Length + Gender, data = beardata)

Residuals:
    Min      1Q  Median      3Q     Max
-53.861 -26.893   2.048  14.629  70.982

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -206.000     43.496  -4.736 0.000112 ***
Length         1.935      0.289   6.695 1.26e-06 ***
GenderM       35.882     15.854   2.263 0.034333 *
---
Signif. codes:  0   ***   0.001   **   0.01   *   0.05
       .   0.1        1

Residual standard error: 36.8 on 21 degrees of freedom
Multiple R-squared:  0.7504,  Adjusted R-squared:  0.7266
F-statistic: 31.56 on 2 and 21 DF,  p-value: 4.693e-07
```

Notice that `GenderM` is listed as a 'variable'. The `lm` function will consider level 1, `F`, of the factor as the 'default' level. The estimated intercept and effect of `Length` applies to bears of type `F`. The estimated value for `GenderM` is *added effect* of level `M`, i.e. if a male and a female bear has the exact same `Length`, the male bear is predicted to weigh 38.882 kilograms more. The following code produces the plot in Figure 12.5.

```
attach(beardata)
plot(Length,Weight,pch=16,cex=1.5,xlab="Length (cm)",
     ylab="Weight (kg)",ylim=c(-20,250))
Weight.predicted <- gender.mod1$fitted.values
is.female <- (Gender=="F")
points(Length[is.female],Weight.predicted[is.female],
       type="l",col="magenta",lwd=2)
points(Length[!is.female],Weight.predicted[!is.female],
       type="l",col="cyan",lwd=2)
```

We can of course include interaction terms as well, i.e. higher order terms where we mix numerical and factor variables. The following model

```
gender.mod2 <- lm(Weight~Length+Gender+Length*Gender,
                  data=beardata)
```

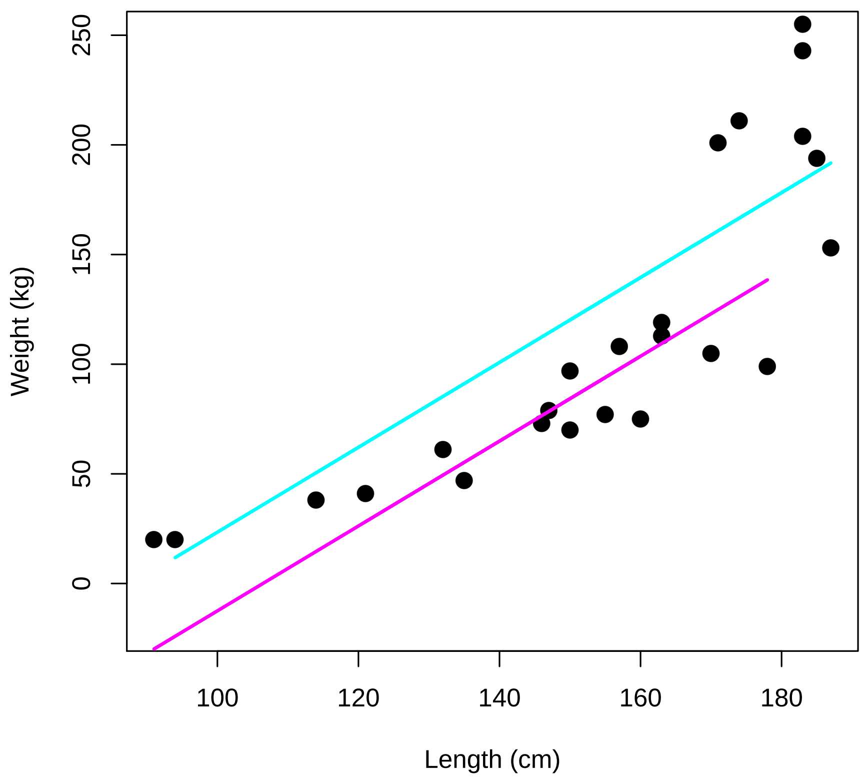


Figure 12.5: The fit of the model where `Gender` has an additive effect. The magenta line applies to female bears, the cyan line to male bears.

includes both the additive and the interaction effect of `Gender`, and the fitted model is shown in Figure 12.6. This actually corresponds (almost) to fitting two separate models here, one for male and one for female bears.

It is quite common to also have models with *only* factors as explanatory variables. This is typical for designed experiments, where the explanatory variables have been systematically varied over some levels. If you take some course in analysis-of-variance (ANOVA) of designed experiments, this will be the topic. The `lm` function can still be used.

12.2.7 Final remarks on regression

There are several other functions beside `lm` that can be used in regression or ANOVA type of data analysis. One commonly used is `glm`, which is short for Generalized Linear Models. If you want to use logistic regression or log-linear models this is the one to consider. A package called `lme4` contains useful extensions of `lm` to handle more complicated error terms (variance components,

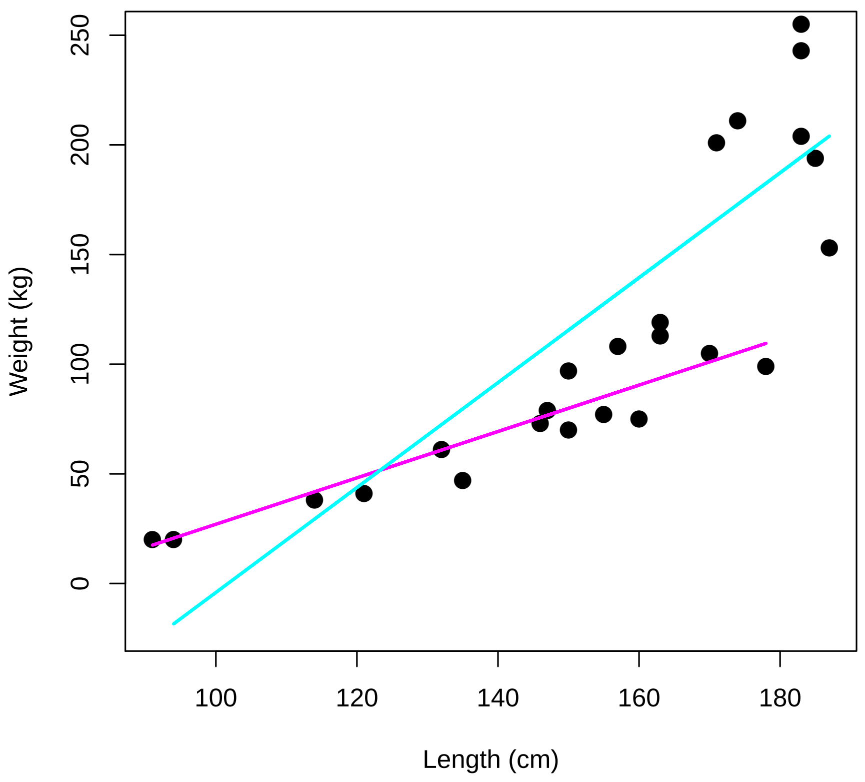


Figure 12.6: The fit of the model where `Gender` has both an additive and interaction effect. The code for making this plot is the same as for Figure 12.5, just replace the fitted models.

longitudinal data, pedigree information, etc.). In the next chapter we will return to the regression problem when we take a look at localized models.

12.3 Classification

A classification problems differs from a regression problem by the response y being a factor instead of numerical. In principle this factor can have many levels, but in many cases we have just two. Classification problems are common, perhaps even more common than regression problems.

One example of a classification problem is to diagnose a patient. A patient belongs to either the class "Healthy" or "Disease A" (or "Disease B" or "Disease C" or...etc). A number of measurements/observations are taken from the patient. These make up the data object for this patient (X_{test}, containing a single row). Based on these we want to classify the patient into one of the pre-defined classes. To accomplish this, we need a model (or Gregory House) that tells us how to classify (predict) given the current X_{test}-values. The model must be fitted to a training set, X_{train} and y_{train}, which is a set of patients where we *know* the true class and where we have measured the same explanatory variables. Thus, it is almost identical to a regression problem, but computations must be done different due to the factor response.

12.3.1 Linear discriminant analysis (LDA)

Just as linear regression is a basic approach to regression problems, the LDA is a basic approach to classification problems. The LDA idea is simple. Consider a training data set with y_{train} and X_{train} data for many objects from each class. Denote the classes A, B,... etc, i.e. the response vector y_{train} is a factor whose levels are these classes. We then assume the X_{train} data are normal distributed around the mean for all objects belonging to the same class, i.e. we have a *centroid* for each class, $\mu_A, \mu_B, \ldots$. We also assume the spread, i.e. the standard deviation, is identical for all classes. The centroid is computed as the mean X_{train}-value(s) for each class, and the standard deviation is computed from all classes. The last piece we need to classify is the *prior probability* for each class.

The prior is the probability of belonging to a class without considering any X-data. It says something about the size of the classes, the larger the class the more likely any random object belongs to it. If we think of the diagnosing problem, and class A corresponds to some rare disease and B is "everything else", we know that when classifying any random person the class A is much smaller than B, i.e. any random person is more likely to belong to class B, and the prior for class A should be small compared to that of B. On the other hand, if we wanted to classify persons who have already gone through some screening for this rare disease, this may no longer be true. These are no longer "any random person", in fact we could easily have a population where A is more common than B. The priors we use should reflect the *population* we are interested in. In most cases we either estimate the priors by the proportions of the classes in the training data set, or we simply set them equal to each other ("flat priors", or all outcomes equally likely).

Once we have computed the centroids, the spread and the priors from the training data, we can compute the *posterior probability* of any class given the data object X_{test}. This is a probability of any given data object to belong to a certain class, i.e. we get one posterior probability for each of the possible outcomes (classes, factor levels), and these probabilities sum to 1.0 for any given data object. It is common to assign y_{test} to the class with the largest posterior probability. Sometimes, if two or more classes are almost equally likely, we refuse to distinguish between them and assign the object to some 'doubt-class' indicating this is impossible to classify based on the current data.

12.3.2 The `lda` function

Just as `lm` is the 'workhorse' for linear regression, the `lda` has a similar role for classification. This function is part of the MASS package, which is always installed along with the base R installation. Let us consider a small example using the bear data set again.

In the bear data set the variable `Gender` is a factor, and we will use it as our response. Let us try to classify bears into male or female based on data for `Weight` and `Chest.G` (we assume the beardata have been read into the workspace, see previous section):

```
library(MASS) # the MASS package is loaded
fitted.mod <- lda(Gender~Weight+Chest.G,data=beardata)
```

Just as `lm` returns an lm-object, the `lda` function returns an lda-object (try `class(fitted.mod)` to verify). We can inspect this object as if it was a list, let us have a look at the content:

```
> names(fitted.mod)
 [1] "prior" "counts" "means" "scaling" "lev" "svd" "N"
 [8] "call" "terms" "xlevels"
```

The `prior` is the vector of prior probabilities, and these are estimated by the proportion of each class in the training data:

```
> fitted.mod$prior
        F         M
0.4166667 0.5833333
```

Since male bears are slightly more common than female in this (small) data set, this will make our model classify more bears as males in the future. If we want both genders to have the exact same prior probability we can specify this when we fit the model. The `lda()` has an option for priors, and if we add `prior=c(0.5,0.5)` to the call we achieve this.

The class centroids are also interesting to inspect:

```
> fitted.mod$means
    Weight   Chest.G
F   74.6000 79.80000
M  139.7857 97.28571
```

As expected, male bears are characterized by being heavier and with larger chest girth.

12.3.3 Making predictions

If we have a new data object $X_{test} = (108, 88)$, i.e. this bears weight is 108 kg and its chest girth is 88 cm. Is this a male or a female? Again we can use the `predict` function just as we did for the regression problem:

```
new.bear <- data.frame(Weight=108,Chest.G=88)
class.lst <- predict(fitted.mod,newdata=new.bear)
```

The output is a list, and we inspect it:

```
> class.lst
$class
[1] M
Levels: F M

$posterior
          F         M
1 0.3908931 0.6091069

$x
         LD1
1 -0.0249895
```

First we see that the list contains a variable named `class`, and this is the prediction for this bear, in this case it is classified as `M`. Then, we have the posterior probabilities for this bear. It is approximately 60% – 40% that this is a male, indicating a substantial uncertainty attached to this prediction.

In Figure 12.7 we have plotted the training data in the left panel and the predicted class for all bears, including the new bear, in the right panel. Here is the code that produced this plot:

```
attach(beardata)
par(mfrow=c(1,2))
is.male <- (Gender=="M")
xlims <- range(Weight)
ylims <- range(Chest.G)
plot(Weight[is.male],Chest.G[is.male],pch=15,cex=1.5,
     col="cyan",xlab="Weight (kg)",ylab="Chest girth (cm)",
     main="Training data")
points(Weight[!is.male],Chest.G[!is.male],pch=15,cex=1.5,
       col="magenta")
legend(x=20,y=140,legend=c("Male","Female"),cex=0.8,pch=15,
       pt.cex=1.5,col=c("cyan","magenta"))
fitted.lst <- predict(fitted.mod)  #using training as test
    data
is.male <- (fitted.lst$class=="M") #predicted class for each
     bear
plot(Weight[is.male],Chest.G[is.male],pch=0,cex=1.5,
     col="cyan3",xlab="Weight (kg)",ylab="Chest girth (cm)",
     main="Predicted class")
points(Weight[!is.male],Chest.G[!is.male],pch=0,cex=1.5,
       col="magenta")
points(new.bear$Weight,new.bear$Chest.G,pch=17,cex=1.5,
       col="cyan3")
legend(x=100,y=70,legend=c("Predicted male",
                           "Predicted female",
                           "New bear"),
       cex=0.8,pch=c(0,0,17),pt.cex=1.5,
       col=c("cyan3","magenta","cyan3"))
```

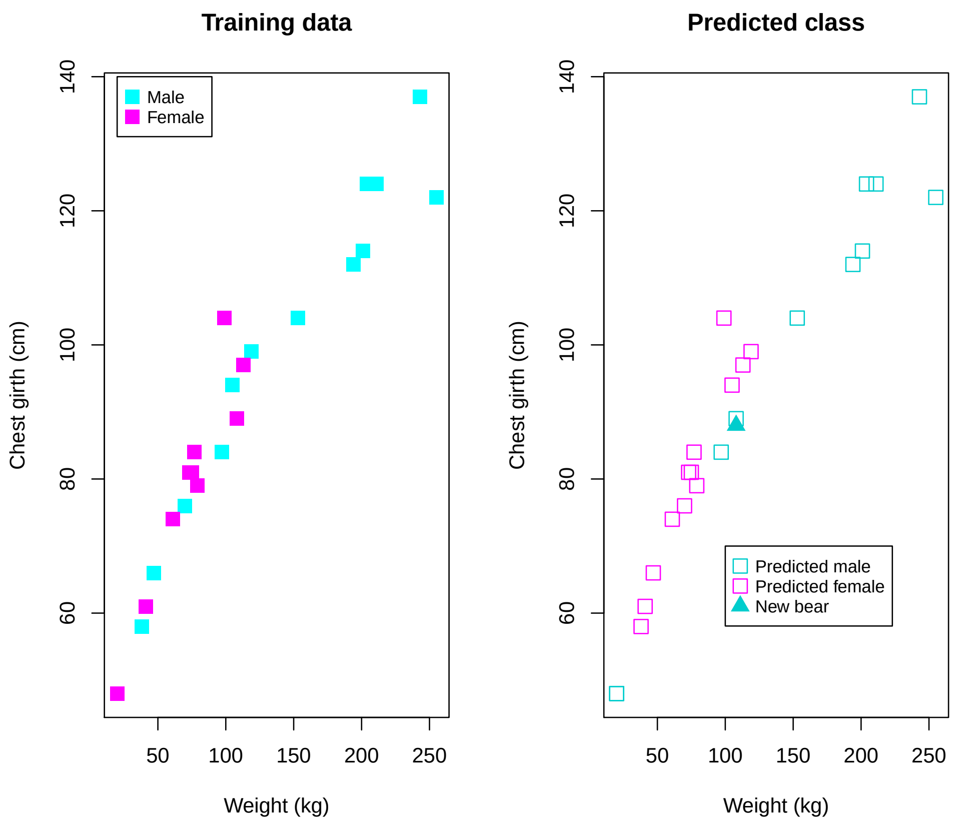


Figure 12.7: Classifying bears as male or female based on their weight and chest girth, using LDA. The left panel are the training data, with the correct separation of males and females. The right panel shows how the LDA-model would classify them. The blue square in the right panel is a new bear, which is classified as male by this model.

12.3.4 Sensitivity and specificity

As shown in the right panel of Figure 12.7 we have made predictions of the gender of every bear in the data set, i.e. we have used the same data as training set and then as test set. In the cases where we know the correct value for y_{test} the only reason we still predict it is to see how well our fitted model behaves. We can now compare the predictions to the correct classification.

The *accuracy* of a fitted model is simply the fraction of correct classifications it produces. Let us compute this for the bear example:

```
correct.gender <- beardata$Gender
fitted.lst <- predict(fitted.mod)  # using training data as
    test data
predicted.gender <- fitted.lst$class
accuracy <- sum(correct.gender==predicted.gender)/
            length(correct.gender)
```

and we find the accuracy is around 0.71.

We realize there are some errors, and there are two ways we can make errors here: Either classify a male bear as female or a female bear as male. To get an overview of all outcomes we can use the `table` function that we have seen before. We give this function two input vectors, the correct and predicted classes, and it will output what we refer to as a *confusion table*:

```
> table(correct.gender,predicted.gender)
              predicted.gender
correct.gender F M
             F 8 2
             M 5 9
```

This table has 2 rows and 2 columns (since we have 2 classes). Here the rows correspond to the correct classification and the columns to the predicted. We see that in 8 cases a female bear is predicted as female and in 9 cases a male bear is classified as male (the diagonal elements). These are the 17 correct classifications behind the accuracy we computed above. There are 7 errors, and we see that 5 of these are male bears being predicted as female, while 2 female bears have been incorrectly assigned as males.

We can now compute the *sensitivity* and the *specificity* of the fitted model. In this example we may define sensitivity as the ability to detect a female bear, and specificity as the ability to detect a male, but we could just as well reverse it. Here both classes are equally interesting, but in many situations one outcome is of special interest (think of the diagnosing example) and then we define sensitivity as the ability to detect this class.

If we consider the confusion table above, sensitivity is simply the number of correctly classified females divided by the total number of females, i.e. $8/(8+2) = 0.8$. The specificity is similar for the males, $9/(9+5) = 0.64$. Here is the R code that computes it:

```
ct <- table(correct.gender,predicted.gender)
sensitivity <- ct[1,1]/sum(ct[1,])
specificity <- ct[2,2]/sum(ct[2,])
```

Notice that we sum the rows since we gave the correct classes as the first input to `table()`. Had we reversed the order of the inputs, the correct classes would have been listed in the columns, and we should have summed over columns instead.

Despite the elevated prior probability for males, this fitted model seems to have a lower ability to detect male bears compared to female bears, as seen from the numbers above. We must stress that since we here have used the same data first as training data and then as test data, the actual errors is most likely larger than we see here. Quantities like accuracy, sensitivity and specificity should be computed from real test data. The results we get here are 'best case scenarios', in a real situation they will most likely be lower.

12.3.5 Final remarks on classification

A close relative of the LDA method is the QDA, and the function `qda` is found in the MASS package as well. You can also use regression methods for classification, typically when you have 2 classes only. Logistic regression, using the `glm` function, is an example of this. There are *many* classification methods implemented in various R-packages, we will not even try to mention them all here. In the next chapter we will return to classification problems and the K-nearest-neighbor type of methods.

12.4 Important functions

Command	Remark
`lm`	Fitting a linear model
`summary`	Generic function that can be used on (almost) all R data structures
`anova`	Analysis of variance for lm-objects
`predict`	Predicting using an lm-model or lda-model (works for many other models as well)
`lda`	Fitting linear discriminant model
`range`	Retrieves minimum and maximum value
`table`	With two input vectors it produces a confusion table
`tapply`	Applies a function to groups of elements in a vector, grouped by a factor

12.5 Exercises

12.5.1 LDA on English poems

This is the continuation of the poetry exercise from chapter 10. It gives you a flavour of the power of pattern recognition and statistical learning...

In order to reveal the identity behind the unknown poem we need to train a classification model on some training data. This means we need some poems that we *know* are written by Blake, Eliot and Shakespeare, and we must compute the numerical fingerprint of each poem.

Load the file `training_poems.RData`. This contains a list of 28 poems, and a vector indicating which author has written each poem. Also load the file `symbols.RData` from the poetry exercise in chapter 10. Make a vector `y.train`, a factor based on the authors. Make a matrix `X.train` with one row for each poem and one column for each symbol. Make a loop counting the number of symbols in each poem, putting these into the rows of `X.train`. Use the symbol-counting function from the poetry exercise in chapter 10.

Fit an LDA-model based on `y.train` and `X.train`. Make a plot of the resulting lda-object. What does this tell you?

Finally, predict which author wrote the unknown poem. Look at the posterior probabilities. What is your conclusion?

12.5.2 Imputing in the longest series of monthly temperatures in Norway

In this exercise we will impute some missing values in Norways longest series of monthly average air temperatures. This temperature series has been observed here at NMBU starting at January 1874, and is a valuable input in todays climate research. We saw this in an exercise in chapter 5, and you may have noticed there were some missing data for some months in recent years. We will now predict the missing temperatures from daily temperatures taken from another data file. In this exercise we will repeat some topics from this and previous chapters, but also look at some new and useful functions.

First, load the data in the file `Tmonthly.RData`. Put the temperatures into one long vector, just as in the chapter 5 exercise. Let us call this vector `Temp.longest`. Create a vector of texts indicating the time for each observation. Each text must be a combination of month and year like this: `"1_1874"` corresponds to January in 1874, `"2_1874"` corresponds to February in 1874, and so on, up to `"12_2013"`. HINTS: Use the `rep` function to create a vector of years and a vector of months, and then use `paste` to merge them. Let us call this text vector `Time.longest`. Both `Temp.longest` and `Time.longest` should be vectors of 1680 elements.

Next, load the file `daily.weather.RData` that we have seen before. There are some days missing in the Air.temp variable, but still enough data to compute a monthly average for every single month in this time span, which we will do next.

We could have used loops here, but will instead use the function `tapply`. This function takes three basic arguments: A vector of numerical data, in our case the daily temperatures, a vector of factors indicating some grouping of the data in the first argument, in our case a text vector grouping all days of the same month and finally a function that should be applied to the data in every group, in our case the function `mean`. Since we may have some `NA`'s in our data, we must also provide the `na.rm=TRUE` option that we have seen before.

To use `tapply` we first need the grouping vector, i.e. a vector with the same number of elements as as there are daily temperatures. We use the same principle as above for `Time.longest`, i.e. the first 31 elements of this vector should all be "1_1988", the next 29 elements should be "2_1988" and so on. Then use this in tapply to compute monthly average temperatures from the daily temperatures. Call this vector Temp.daily. It should now have names corresponding to the factor levels, i.e. the months. The statement

```
Time.daily <- names(Temp.daily)
```

should produce a vector Time.daily with texts similar to those in Time.longest. Both Temp.daily and Time.daily should be vectors of 312 elements.

Next, we need to identify which elements in Time.longest are also found in Time.daily. We use the function match for this. It takes two vectors as input, let's call them x and y. It finds the position of the first match of each element of x in y. A code like this should do the job:

```
idx.overlap <- match(Time.daily,Time.longest)
Temp.longest.overlap <- Temp.longest[idx.overlap]
```

Try out the match function and make certain you understand how it works, this is a very handy function! Try this small example in the Console window: match (c(2,4,1),c(2,2,4,3,6,4,3,2,1)). Make a plot of Temp.longest.overlap versus Temp.daily, these should be temperatures from the exact same months.

If you make the plot, you will see the temperatures are not identical, i.e. we cannot just use the Temp.daily temperatures directly to fill in the holes in Temp.longest.overlap. Instead we fit a linear model, predicting Temp.longest .overlap from Temp.daily. Create a data.frame called train.set containing the non-missing data in Temp.longest.overlap and the corresponding values in Temp.daily. Then, fit a linear model with Temp.longest.overlap as response and Temp.daily as explanatory variable. Next, make another data.frame called test.set similar to train.set except that you use only the missing data in Temp .longest.overlap and the corresponding values of Temp.daily. Finally, predict the missing values in test.set using the fitted linear model.

Before we are truly finished, we need to put the imputed data back into their correct positions in Temp.longest. Can you do this? (requires very good insight in the use of index vectors...)

REMARK: Notice how small part of this exercise is the actual statistical modelling, and how much more labour is related to the handling of data. This is quite typical of real-life problems!

Chapter 13

Local models

13.1 Large data sets

In the previous chapter we introduced data modelling problems and the response variable y and the matrix of explanatory variables X. In this chapter we will focus on the *prediction* part of data modelling. We seek to establish some relation (fitted model) between the response and the explanatory variables in such a way that if we observe a new set of explanatory variables (new rows of X) we can use the fitted model to predict the corresponding outcome of y.

The number of data objects in a data set, i.e. the number of rows in X, is by convention named n. The number of variables (columns) in the X matrix is named p. In many branches of modern science we often meet large data sets. This could mean we have a large number of objects (large n) and/or it could mean we have a large number of variables (large p). The cases where we have $n > p$, and even $n >> p$ ($>>$ means 'much larger than'), usually calls for a different approach to data analysis than if we have $n < p$. In this chapter we will consider problems where we have $n > p$, i.e. the matrix X is 'tall and slim'.

These type of data sets arise in many situations. In some cases it is some automated measurement system that samples some quantities over and over again, producing long 'series' of data. Observations of weather and climate are examples of such data. A data object contains typically the measurements from the same time (e.g. same day). Another example is image data. Each pixel corresponds to a data object, with data for location (row and column in the image) and some color intensities. A third example is modern biology, where a large number of persons/animals/plants/bacteria have been sequenced to produce some genetic markers for each individual. From these markers we can obtain some numbers, corresponding to a data object for that individual, and we can do this for a large number of individuals.

Due to ever-improving technologies, we often find data sets containing thousands or even millions of data objects. These are situations where 'push-button' solutions in standard statistical software packages often do not work very well, if at all. The only solution is to do some programming.

13.2 Local models

A typical approach to a data set with many objects is to consider *local models.* The general model

$$y = f(X) + e$$

says that the response variable is related to the explanatory variables through some function f. It also has some contribution from an error term e. The function f could be any function, and this makes the model very general. In the case of simple linear regression in the previous chapter, we assumed

$$f(X) = \beta[1] + \beta[2]X$$

However, relations are rarely exactly linear, and sometimes very far from it, and the more data we have the more apparent this becomes.

Instead of searching for the 'true' function f that produced the data, we often approximate this function by splitting the data set into smaller regions, and then fit a local model within that region. These local models can be simple, often linear models, and even if the function f is far from linear, it can usually be approximated well by splicing together many local models. In Figure 13.1 we illustrate the idea.

13.3 Training- and test-set

In the previous section we saw that any data set can be split into two parts, the training-set and the test-set. We may think of the test-set as the 'new' data that we have not yet observed, but just as often both the training-set and the test-set are just two (randomly chosen) subsets of the same total data set. If we have the full data set called (y_{all}, X_{all}) we can split this into a training-set (y_{train}, X_{train}) and a test-set (y_{test}, X_{test}).

The whole idea of this splitting is to fit our model *only* to the training data, and then use the test-set to test how well the fitted model can predict. This means we use the X_{test} together with the fitted model to predict y_{test}, pretending y_{test} is unknown. Finally, when the predictions are done we can compare them to the actual observations of y_{test} and see how close to the 'truth' we came.

Above we mentioned that we use n to symbolize the number of data objects (rows) in a data set. More specifically, n denotes the number of data objects in the training set. Both the training and test-set will of course have p variables.

Notice that for very small data sets this splitting into two subsets may be difficult, since we would then have critically small training- and/or test-sets. However, for data sets with many data objects this is rarely a problem. Even if we set aside a substantial number of data objects in a test set, we still have plenty of data objects left in the training set.

The distinction between training- and test-data are not always clear in the literature. We will make some efforts to pay attention to this distinction, because it is of vital importance for understanding the methods we are looking at in this chapter. In the next chapter we will focus on this splitting of the data even more.

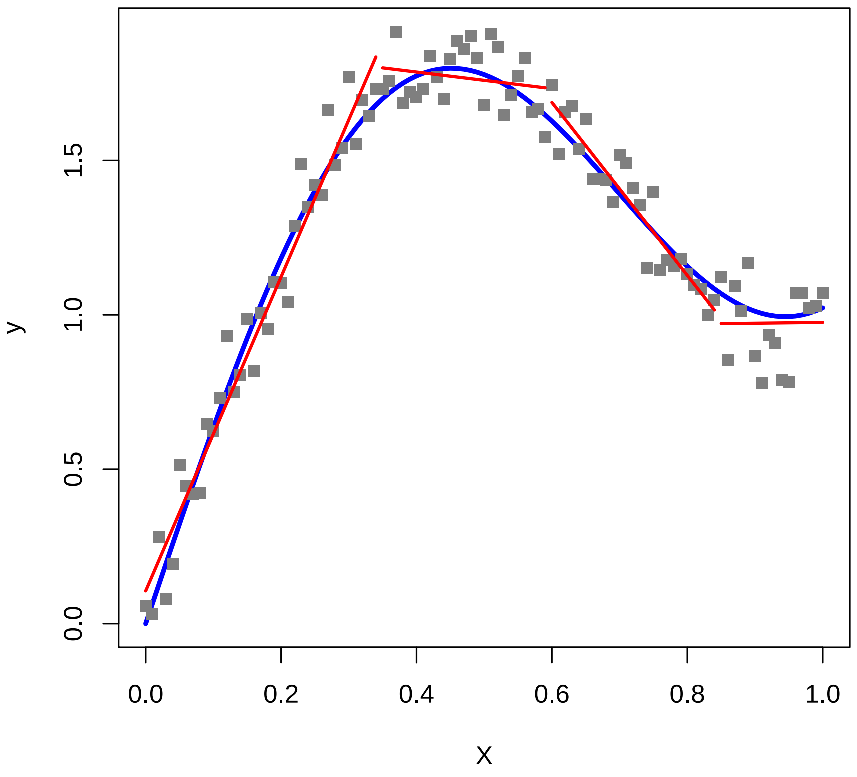


Figure 13.1: An example of the local model idea. The thick blue curve is the true function relating X and y. Note that this function is in general unknown to us! It is clearly not linear. The gray dots are the training data sampled, and due to the error term (noise) they fluctuate around the blue curve, but the non-linear shape of the relation is visible in the data as well. Instead of trying to find the (perhaps complex) function behind that blue curve, we approximate it by splitting the data set into regions, and then fit a simple linear model in each region. The red lines are the fitted local models in each region. Here we chose to split into 4 regions, but this will depend on how much the data fluctuate and the size of the data set.

13.4 Local regression

13.4.1 Moving average

The simplest form of a local regression method is what the book AISL denote *K-nearest-neighbour* regression and that we will refer to as the *moving average*. The moving average idea means that when we are to predict the outcome $y_{test}[i]$ we use the corresponding data object $X_{test}[i,]$ to find the *neighbourhood* of this data object among the training data objects in X_{train}. Once we have located which data objects of X_{train} are the neighbours, we predict $y_{test}[i]$ by computing the mean of the neighbours' y_{train}-values.

The algorithm can be described as follows:

1. For a given data object $X_{test}[i,]$ compute the distances to all X_{train} data objects. This will be a vector with n distances, one for each data object in X_{train}. Call this d.

2. Find which data objects produces the K smallest distances in d. Let I_i be the index vector of these objects.

3. Use the index vector I_i to retrieve the corresponding values $y_{train}[I_i]$, and predict by $y_{test}[i] = \bar{y}_{train}[I_i]$.

The term 'moving average' comes from the fact that this model predicts the response y_{test} at a certain point in the space spanned by the explanatory variables as an average of the responses of the neighbouring data objects. As we 'move' through this space (consider new test-set data), we compute new averages.

Let us consider a small example to make the ideas more transparent. In the previous chapter we used the bears data set for illustration. Let us again (as in section 12.2) try to predict `Weight` from `Length` in this data set, i.e. the response variable is `Weight` and the only explanatory variable ($p = 1$) is `Length`. The full data set has 24 data objects. We split the data set into a training-set consisting of the first 23 data objects, and a test-set with the last data object. Here is some code that sets the scene:

```r
beardata <- read.table(file="bears.txt",header=TRUE)
y.train <- beardata[1:23,1]            # column 1 is Weight
X.train <- as.matrix(beardata[1:23,2]) # column 2 is Length
y.test <- beardata[24,1]               # column 1 is Weight
X.test <- as.matrix(beardata[24,2])    # column 2 is Length
```

Let us predict the `Weight` of the bear in the test-set to illustrate the procedure.

In the algorithm above we start by computing the distances from `X.test[i,]` to all data objects in `X.train`. In this case the test-set has only one object, but for the sake of generality we still use the index `i`, and set it to `i<-1`.

What is a distance? We will talk more about distances below, but for now we say that distance is simply the absolute difference in `Length` (a distance cannot be negative). We compute all distances in one single line of code:

```r
i <- 1
d <- abs(X.test[i,]-X.train) # abs gives absolute value
```

This results in the vector `d` with 23 elements, one for each data object in `X.train`. If we look at this distance vector

```
> as.vector(d)
 [1] 41 34 20 16 28 28  2 23 32 19  8  8 30 61  5 15 23  5
     8 64  5 28  9
```

we notice that data object 7 has distance 2 to `X.test[i,]`, meaning this bear has a `Length` 2 cm longer or shorter than our bear of interest. We also notice that data object 15 and 18 have small distances to our `X.test[i,]`. Next, we must find which distances are the smallest. Notice, we are not interested in the actual distances. We are only interested in *which* data objects have the smallest distances (not how small they are).

To find this index vector (called I_i above) we use the function `order` in R. This will take as input a vector, and return as output the index vector that, when used in conjunction with the input vector, will re-arrange the elements in ascending order. This may sound complicated, let us illustrate:

```
> idx <- order(d)
> idx
 [1]  7 15 18 21 11 12 19 23 16  4 10  3  8 17  5  6 22 13
     9  2  1 14 20
> d[idx]
 [1]  2  5  5  5  8  8  8  9 15 16 19 20 23 23 28 28 28 30
     32 34 41 61 64
```

We give `d` as input, and the output is stored in `idx`. Notice that `idx` specifies that element 7 in `d` should be put first, then element 15, then element 18, then 21, and so on, to obtain a sorted version of `d`. We also use the index vector to produce this sorting, by `d[idx]`, and indeed we see the elements are now shuffled in a way that sorts them in ascending order. You should really make certain you understand how `order` works, this is one of the most convenient functions in R!

In the moving average method there is one parameter, the number of neighbours we should consider. This is the K in the KNN name. In this example we will use $K = 3$, but will return to this choice later. The prediction is made by adding the following lines of code:

```
idx <- order(d)
K <- 3
y.test.hat <- mean(y.train[idx[1:K]])
```

Notice how we use the `K` first elements of `idx` to directly specify which elements in `y.train` to compute the mean from. The whole procedure is illustrated in Figure 13.2.

Notice that the moving average method does not fit an overall model to the training data first, and then use some `predict` function, as we saw for `lm` and `lda` in the previous chapter. Instead, we use the X_{test} data object and make a look-up in the training data for similar data objects, and fit a very simple model to these. This model is only valid at this particular point, and as soon as we move to some other X_{test} values, we must repeat the procedure.

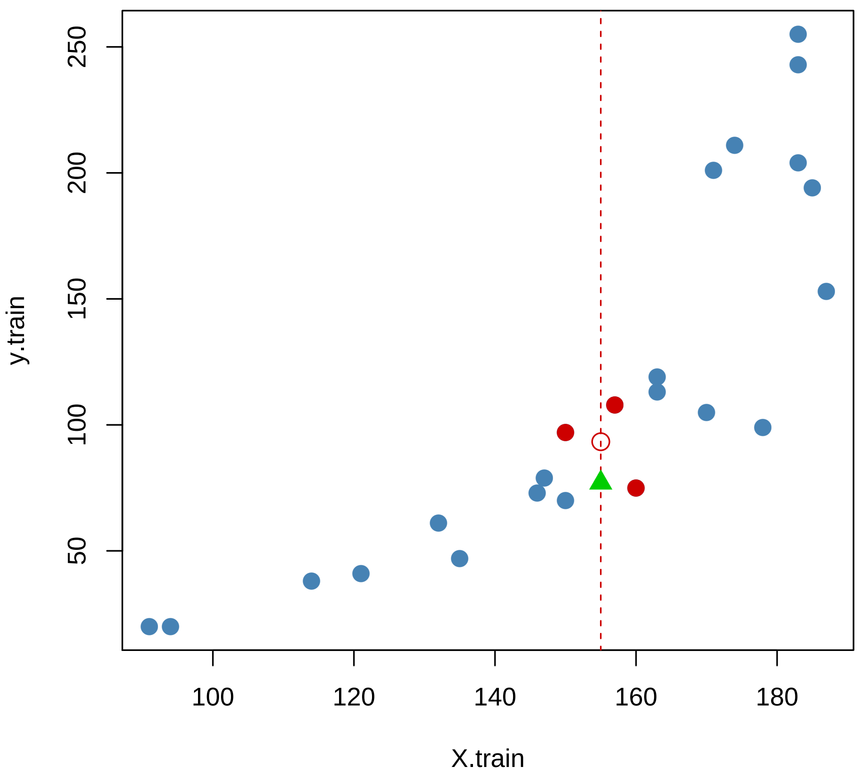


Figure 13.2: The filled dots (blue and red) are the training data. The red vertical broken line marks the length of the test-set bear. Based on its length, we find the $K = 3$ bears in the training data with most similar lengths, these are the red filled dots. The green triangle is the predicted weight, computed as the mean of the weights behind the red dots. The open red dot is the actual observed weight for the test-set bear.

13.4.2 Local linear regression

The local model we fitted in the moving average method is extremely simple. It is simply assuming the relation between response and explanatory variables is constant in the region it considers. This is OK if the region is very small. However, sometimes there are no, or at least very few, neighbours very close to our X_{test}. In such cases we may benefit from making slightly more complex models even in a local regression.

This is the idea behind the *lowess* method (Locally Weighted Scatterplot Smoother), where we fit a linear regression model based on the neighbour data objects, and use this to predict the y_{test} value. Apart from this, the rest is identical to the moving average. Here is a small function that can be used for predicting a single y_{test} value based on training data with a single explanatory variable:

```r
locLinReg <- function(X.test,y.train,X.train,K=1){
  # X.test must be a single data object
  # y.train must be a vector of n elements
  # X.train must be a matrix of n rows and 1 column
  d <- abs(X.test-X.train)
  idx <- order(d)
  dfr <- data.frame(y=y.train[idx[1:K]],X=X.train[idx[1:K]])
  lm.fit <- lm(y~X,data=dfr)
  y.test.hat <- predict(lm.fit,newdata=data.frame(X=X.test))
  return(y.test.hat)
}
```

If we `source` this function, we can use it to predict the `Weight` of all bears in the data set by the following script:

```r
y.hat <- rep(0,24)
for( i in 1:length(y.hat) ){
  y.hat[i] <- locLinReg(beardata$Length[i],
                        beardata$Weight[-i],
                        beardata$Length[-i],K=6)
}
```

Here we used $K = 6$, and the results are shown in Figure 13.3.

13.4.3 The `loess` function

In R we have an implementation of a local regression in the function called `loess()`. It can predict the response based on one or two explanatory variables, and the input is specified quite similar to `lm()`, using a formula. Instead of the parameter K, there is an option called `span` which is a number between 0 and 1, specifying how large fraction of the training set should be used as the neighbourhood. Thus, if the training-set has $n = 100$ data objects, and we specify `span=0.25`, it corresponds to using $K = 25$ neighbours.

Here is an example of how we can use `loess()` to estimate how bears weights are related to their lengths:

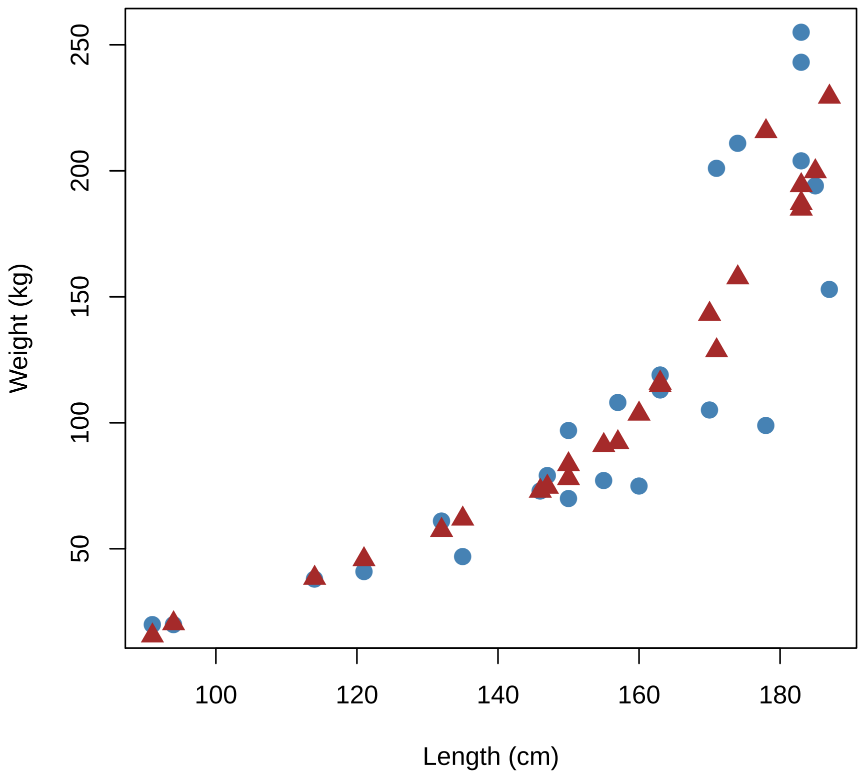


Figure 13.3: The blue dots are the observed and the brown triangles the predicted weights of bears using the local linear regression function from the text.

```
loess.fit <- loess(Weight~Length,data=beardata)
y.predicted <- predict(loess.fit,newdata=data.frame(Length
    =91:187))
```

The result can be seen in Figure 13.4.

13.5 Local classification

Local methods are just as popular for classification as for regression, and again they are more typically used when we are faced with large data sets where we have many data objects.

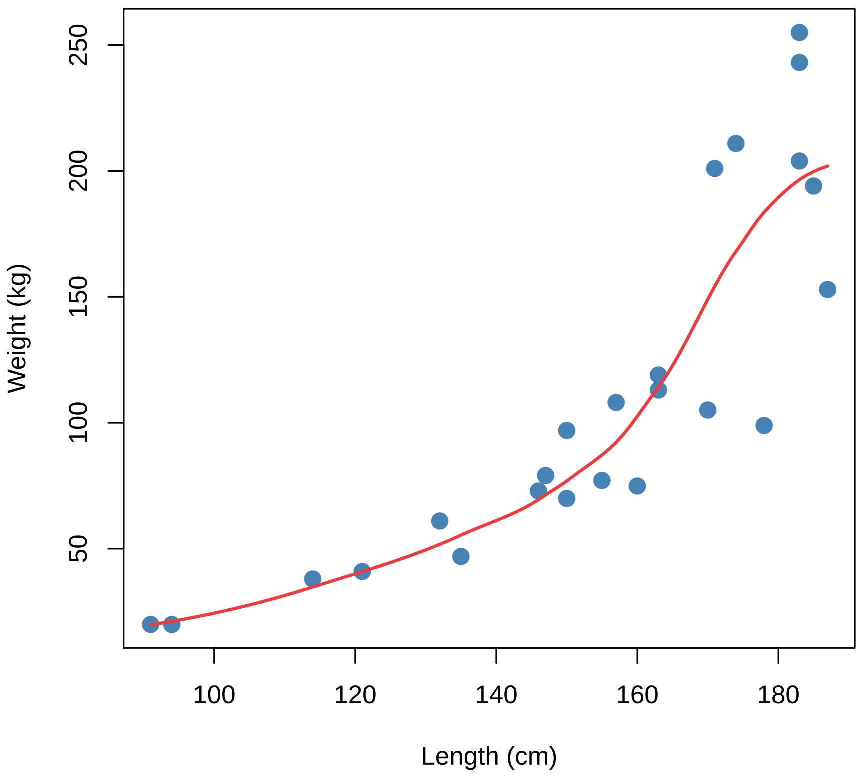


Figure 13.4: The curve shows the predicted weights of bears with lengths from 91 to 187 cm using the `loess` function as described in the text.

13.5.1 K-Nearest-Neighbour classification

The KNN classification method is widely used, and for good reasons. It is both intuitive, simple and performs very well in many cases. The idea is parallel to the moving average and local linear regression:

1. For a data object in the test-set ($X_{test}[i,]$), compute the distance to all data objects in the training-set (X_{train}).

2. Find the K training-set objects having the smallest distance to the test-set data object. We call these the nearest neighbours, and the index vector I_i tells us where we find them among the training-set objects.

3. Classify y_{test} to the most common class among the nearest neighbours, i.e. a simple majority vote.

Thus, the only difference is that instead of computing a mean value (which is senseless since the response is no longer numerical) we count how often we see the various classes (factor levels) among the nearest neighbours, and choose the most common one.

In the `class` package in R you find a standard implementation of this method in the `knn` function. There are, however, good reasons for us to make our own version of the KNN method. First, it is the best and perhaps only way of really understanding the method. In fact, unless you can build it, you haven't really understood it (who said this?). Second, the `knn` function computes unweighted euclidean distances only, which is something we may want to expand, as we will see below. In the exercises we will return to the KNN classification.

13.6 Distances

Common to all the local methods we have looked at here is the need for computing distances. The nearest neighbours are simply defined by how we compute the distances, and this step becomes the most important in all these methods. In the bears example we used a single explanatory variable. In general, we will more often use several, and the matrices X_{train} and X_{test} can have two or more columns. In such cases we need to give some thought to how we compute a distance between two data objects.

13.6.1 Minkowski distances

Let $X[i,]$ and $X[j,]$ be two data objects. We assume both rows have $p > 1$ elements, i.e. we are using multiple explanatory variables. A *Minkowski* distance between the two objects is defined as

$$d[i,j] = \left(\sum_{k=1}^{p} |X[i,k] - X[j,k]|^q \right)^{1/q} \tag{13.1}$$

where the exponent $q > 0$. This is a general formula, and by choosing different values for q we get slightly different ways of computing a distance.

If we use $q = 2$ the Minkowski formula give us the standard euclidean distance

$$d[i,j] = \sqrt{\sum_{k=1}^{p} |X[i,k] - X[j,k]|^2} \tag{13.2}$$

which is what we call 'distance' in daily life. Can you see how this follows from Pythagoras theorem? Imagine we have two explanatory variables, i.e. $p = 2$. Any data object can then be plotted as a point in the plane spanned by these two variables. The euclidean distance between two points (data objects $X[i,]$ and $X[j,]$) is the length of the straight line between them.

Now, imagine in this plane you can only travel in straight lines either horizontal or vertical (no diagonals). This is typically the case if you walk in a city centre like Manhattan. If you are going from one point to another, you have to follow the streets and avenues, and there is no way you can follow diagonal straight lines (unless you can walk through concrete walls). Thus, the distance between two points is no longer euclidean! Instead, if we use $q = 1$ in the

Minkowski formula we get the Manhattan distance

$$d[i,j] = \sum_{k=1}^{p} |X[i,k] - X[j,k]| \tag{13.3}$$

which is the distance between two points $X[i,]$ and $X[j,]$ trapped inside a grid. Notice how we just sum the absolute difference in each coordinate.

We can also use $q = 3$ or larger. If we allow q to approach infinity we get

$$d[i,j] = \lim_{q\to\infty} \left(\sum_{k=1}^{p} |X[i,k] - X[j,k]|^q \right)^{1/q} = \max_k |X[i,k] - X[j,k]| \tag{13.4}$$

which is usually referred to as the maximum-distance. Notice how the distance between the two points is now just the largest coordinate difference.

Let us consider a small example to illustrate that the choice of distance metric has impact on a nearest neighbour method. We have two explanatory variables, and let

$$X_{test} = \begin{pmatrix} 0.0 & 0.0 \end{pmatrix} \quad \text{and} \quad X_{train} = \begin{pmatrix} 1.0 & 1.1 \\ 0.4 & 1.2 \\ 0.0 & 1.5 \end{pmatrix}$$

First, we compute the euclidean distance from X_{test} to all three data objects in X_{train} using the formula above:

$$d = \begin{pmatrix} \sqrt{(0.0-1.0)^2 + (0.0-1.1)^2} \\ \sqrt{(0.0-0.4)^2 + (0.0-1.2)^2} \\ \sqrt{(0.0-0.0)^2 + (0.0-1.5)^2} \end{pmatrix} = \begin{pmatrix} 1.49 \\ 1.26 \\ 1.50 \end{pmatrix}$$

and we see that the second data object in X_{train} is the nearest neighbour of X_{test}. If we compute the Manhattan distance from the same data we get

$$d = \begin{pmatrix} |0.0-1.0| + |0.0-1.1| \\ |0.0-0.4| + |0.0-1.2| \\ |0.0-0.0| + |0.0-1.5| \end{pmatrix} = \begin{pmatrix} 2.1 \\ 1.6 \\ 1.5 \end{pmatrix}$$

and this time the third data object is the nearest neighbour. Finally, if we use the maximum-distance

$$d = \begin{pmatrix} \max_{1,2}(|0.0-1.0|, |0.0-1.1|) \\ \max_{1,2}(|0.0-0.4|, |0.0-1.2|) \\ \max_{1,2}(|0.0-0.0|, |0.0-1.5|) \end{pmatrix} = \begin{pmatrix} 1.1 \\ 1.2 \\ 1.5 \end{pmatrix}$$

which means the first data object produces the smallest distance from X_{test}. This shows that the choice of distance metric may have an effect, and that we should be conscious of what we use.

There is a function in R for computing distances between the objects, called `dist`. It takes as input a matrix and computes distances between all pairs of data objects of this matrix. However, keep in mind that we do not really seek the distances *between* the data objects of the training-set (or test-set). Instead, we want to have the distance from a test data object, $X_{test}[i,]$, to all training data objects in X_{train}. Since we are now experts in R programming, we can just as well make this function ourselves!

13.6.2 Scaling of variables

Just as important as the choice of distance formula is the scaling of the variables. Before we compute distances and look for neighborhoods, we should make certain all variables have comparable values, i.e. they are on the same scale.

Previously we have seen some data for body weight and body length of bears. These could very well be used as the explanatory variables (i.e. $p = 2$) in some modelling exercise. If length is measured in meters and weight in milligrams, the numbers in the weight column will be huge compared to those in the length column. Plugging this into the distance-formulas will result in the neighborhood of any bear are the bears with (almost) similar weight, regardless of height. And vice versa, if we instead measure length in millimeters and weight in tons, any neighborhood is completely decided by length. Clearly, we do not want measurement units to affect our methods like this.

The common way to *standardize* the explanatory variables is to *center* and *scale* the data in each column of the X_{train} matrix.

- Centering means we compute the mean value of the column, and subtract this from every element in the column.

- Scaling means we compute the standard deviation of a column, and divide every element in the column by this number.

The function `scale` in R can be used to transform your X_{train} matrix in this way prior to the computing of distances.

Remember that if you do something to the columns of X_{train} you must do exactly the same to the corresponding columns of X_{test}. Any centering and scaling is computed from the X_{train} matrix only. Each column of this matrix has a mean and a standard deviation. Then, after scaling X_{train} you must use the exact same means and standard deviations to do the same on X_{test}. Never re-compute the mean and standard deviation based on the X_{test} matrix! If you do this, the training-set and the test-set are on (slightly) different scales, and any model fitted to the training-set will perform poorer on the test-set.

13.7 Important functions

Command	Remark
`loess`	Local regression
`predict`	Can also be used for `loess` objects
`knn`	K-nearest-neighbour in the `class` package
`dist`	Computes several types of distances
`scale`	Scales matrices

13.8 Exercises

13.8.1 Classification of bacteria

We have a data set where thousands of bacteria have been assigned to three different *Phyla.* All bacteria are divided into a hierarchy starting at Phylum (division), then Class, ..., down to Species. Thus, Phylum is the top level within the kingdom of bacteria. For each bacterium we have also measured the size of the genome (DNA) in megabases, and the percent of GC in the genome. DNA consists of the four bases A,C,G and T, and GC is simply the percentage of G and C among all bases.

Is it possible to recognize the Phylum of a bacterium just by measuring the genome size and the GC-percentage?

In order to answer this we need to do some pattern-recognition (which is another term for classification).

Part A - Overview the data

Read the data from the file `three_phyla.txt`. Plot Size versus GC as a scatter-plot, and color the markers by Phylum. Are the classes well separated? Based on this data set, what is the prior probability of each class?

Part B - Splitting and scaling data

Split the data set into a training-set and a test-set. Sample at random 1000 rows of the data.frame, and put these into a data.frame named `phyla.tst`. Put the remaining rows into `phyla.trn`. Use the `sample` function for random sampling.

The two explanatory variables are on very different scales in this example, one being in megabases and the other in percent. To avoid any effects of this, we should scale the explanatory variables. Use the function `scale` to center and scale the explanatory variables of the training-set.

The test-data must also be scaled with the exact same values, i.e. you cannot just re-compute the scaling values based on the test-set, you MUST use the same values that were computed for the training-set. These values are available from the previous step. Let `X.scaled` be the variable where we stored the result of the scaling in the previous step (e.g. the output from `scale(as.matrix(phyla.trn[,2:3]))`). This variable has two *attributes*. Attributes are extra information we may give to any variable in R. In this case there is one attribute named `"scaled:center"` and one named `"scaled:scale"`. These contain the values we need. We can retrieve them by the `attr()` function, like this:

```
cntrs <- attr(X.scaled,"scaled:center")
sdvs <- attr(X.scaled,"scaled:scale")
```

There will be two values in both cases, since we have two explanatory variables. Now, we subtract the `cntrs` values from the corresponding columns of `phyla.tst` and then divide each column og `phyla.tst` by the corresponding value in `sdvs`.

You should make a plot as in part A above, of the training-set using open circles as markers, and the test-set with crosses as markers, just to verify that they are found more or less in the same part of the scaled Size-GC-plane. Notice how the numbers on the axes are now comparable in size.

Part C - LDA

Fit an LDA-model to the training-set, and predict the classes of the test-set (see chapter 12). Compute the accuracy of this fitted model.

Make a plot to compare the predicted and the true classes. Make two panels, and plot the predicted classes as in part A above in the left panel and the true classes in a similar way in the right panel.

Part D - KNN

Next, we classify each test-set object by KNN. You can use the function `knn` in the `class` package for this. Again, compute the accuracy, and make plots like for LDA above. How does it compare to LDA?

Part E - Make your own KNN function

In order to really understand the KNN classification method, you should build your own KNN function. The input arguments can be more or less the same as for the `knn` in part D above. That function uses euclidean distances only, perhaps you can add an option for other distance metrics? HINT: Make first a function `my.dist` that takes as input a single test-set object and a training-set matrix, and computes the distance from the test-set object to all training-set objects. This function should then be used by the KNN function!

Chapter 14

Cross-validation

14.1 The bias-variance trade-off

In chapters 12 and 13 we fitted models to (training) data. When fitting any model to data, there are two sources of error. These are usually referred to as the *bias* and the *variance* of the fitted model.

The bias is the error we get from using a simplification of the real world. When we assume some model, no matter how simple or complex, there is always some degree of simplification in it. If not, it is not a model! The concept of a model means simplification. Thus, the bias is what we may also call the 'modelling error'. In the local methods of the previous chapter we used a simple model on a small neighbourhood, but even in this small neighbourhood the simple model is a simplification. However, the smaller the neighbourhood, the smaller this error will be. So, why don't we make our neighbourhoods extremely (infinitely) small?

The reason is the variance source of the error, the 'estimation error'. We know that if we have many data points, this error becomes smaller. Each data point carries some information, and the more data points we use, the more information we have. In this perspective, we should actually broaden the neighbourhoods of the local model, since larger neighbourhoods means more data are found inside them.

This is the classical trade-off between bias and variance. We cannot have it both ways, and somewhere in between there is usually a balance where we find the smallest sum of errors (bias+variance). Going to the extremes in either direction will result in one of the sources blowing sky-high and we have a useless fitted model. This is illustrated in Figure 14.1.

14.1.1 Model selection

In the local models we have a parameter, K, governing the size of the neighbourhood, and by tuning this up or down we can choose between

- Small K, which means smaller bias, but larger variance.
- Large K, which means larger bias, but smaller variance.

Searching, and finding, the *optimal* value for K is often referred to as *model selection.*

It should be noted that the bias-variance trade-off applies to *all* models, not only the local models of the previous chapter. In multiple linear regression or LDA we include several explanatory variables to predict the response. As we saw briefly in chapter 12, these explanatory variables can be either distinct variables or just higher-order terms of some other variable, or interactions between two or more variables (with higher order terms...etc). Anyway, we can *choose* to include or exclude an explanatory variable in such a model. The inclusion/exclusion of variables in this setting corresponds exactly the choosing the value of K above:

- Include many variables, which means smaller bias, but larger variance.
- Include few variables, which means larger bias, but smaller variance.

Again there is always some optimal level of inclusion/exclusion, and finding the *best* set of explanatory variables to include in the model is again called model selection.

Notice that with the local methods, the model as such is fixed but the data set used for fitting varies by the size of the neighbourhood. If we use a moving average, it has one single parameter, the mean inside the given neighbourhood. This parameter must be estimated from the data in the neighbourhood, and the more data we have, the better it is estimated. For LDA and multiple regression, the data set used for fitting is fixed, but the number of parameters (e.g. the β's in the regression model) increase/decrease depending on how many variables we include/exclude .

14.1.2 Why model selection?

The obvious answer to this is to get best possible predictions. If a fitted model is to be used for predicting or classifying new obervations (e.g. weather forecast, stock market changes, diagnosing patients, etc.) it is of vital importance that it has been fine-tuned with respect to the bias-variance trade-off. A common mistake is to focus too much on reducing the bias, since this is where we enter our 'knowledge' about the phenomenon. The result of this will then be *over-fitting*, i.e. the fitted model is fitted too closely to the training-set data, and when applied to the test-set it will fail severely.

However, model selection is also interesting from another perspective. In science we are often *not* interested in prediction tools. Instead, we want to understand how the response variable is related (or not related) to the explanatory variables. The model selection exercise can tell us which relations are the 'real' ones and which are not. Here is an example from modern biology:

We want to diagnose newborn babies with respect to a severe illness related to digestion. The response is a factor with the levels A (healthy) or B (the baby has the disease). The explanatory variables are counts of various types of bacteria taken from the baby's gut. If we have a training-set of some babies with and some without this disease, and their gut bacteria samples, we can make a model where we try to classify A and B with respect to the bacteria-data. Some bacteria may be relevant and some may not, and performing a systematic model selection may tell us exactly which bacteria provide us with the diagnosing information. These are the bacteria we should look more closely at in order to understand the disease.

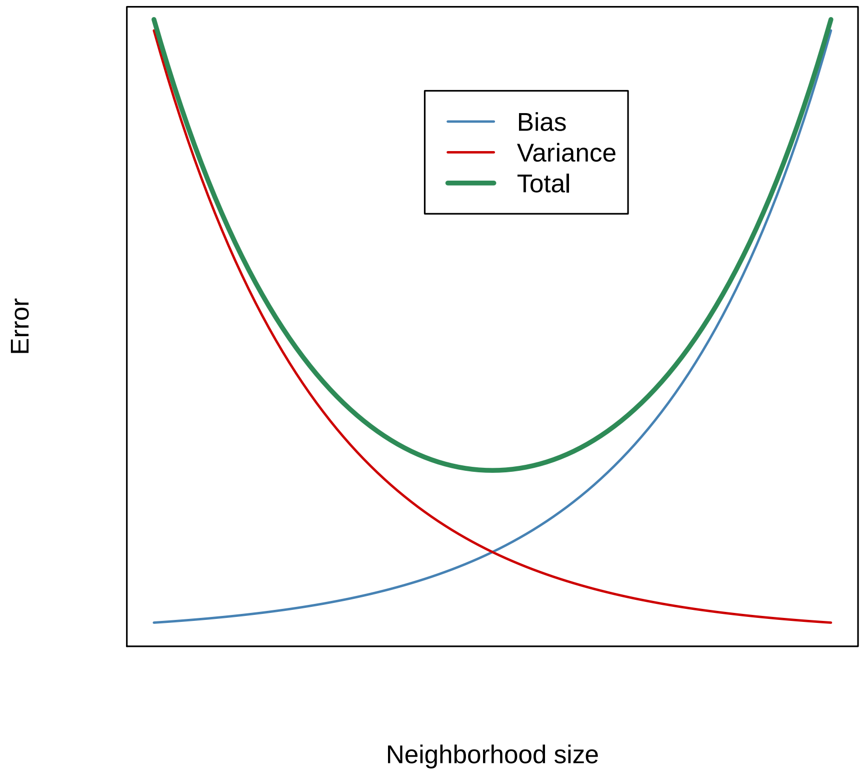


Figure 14.1: A schematic overview of the trade-off between bias and variance. The horizontal axis indicates the choice of neighborhood size K in the local models. Using a small K will produce a small bias (blue curve), but a large variance (red curve). The sum of them (green curve) will be quite large. At the other end of the K-axis we get opposite effects, and again the sum is quite large. Somewhere in between we often find a balance where both bias and variance are fairly small, and their sum is minimized. Note: The shape of the total error (green curve) is in reality never as smooth as shown here, but this illustrates the principle.

14.1.3 Prediction error

We cannot observe neither the bias nor the variance of a fitted model, simply because we do not know the true underlying function (if we did the whole modelling would be meaningless). But, we can estimate the sum of them.

As we have seen before, we split the full data set into a training-set and a test-set. We fit our model to (y_{train}, X_{train}) only. Pretending y_{test} is unknown, we predict its values from X_{test} using the fitted model, and then compare these predictions to y_{test}. From this we can compute the *prediction error*. A commonly used formula for regression problems is to compute the Mean Squared Error of Prediction (MSEP):

$$MSEP = \frac{1}{m}\sum_{i=1}^{m}(y_{test}[i] - \hat{y}_{test}[i])^2 \tag{14.1}$$

where m is the number of data objects in the test-set, and $\hat{y}_{test}[i]$ is the predicted value of $y_{test}[i]$.

For classification problems the MSEP is replaced by the Classification Error Rate (CER):

$$CER = \frac{1}{m}\sum_{i=1}^{m} I(y_{test}[i] \neq \hat{y}_{test}[i]) \tag{14.2}$$

where the function $I()$ takes the value 1 if its input is `TRUE`, and 0 otherwise. It just counts the number of cases where we have mis-classified. In chapter 12 we mentioned the accuracy of a classification, and the CER is 1 minus the accuracy.

Both the $MSEP$ and the CER are quantities we can compute, and they can both be seen as a sum of bais+variance. Notice there are some important aspects here:

1. The training- and the test-sets are strictly separated. When we fit the model to (y_{train}, X_{train}) we do not involve any information about (y_{test}, X_{test}). The fitted model is based *only* on (y_{train}, X_{train}).

2. When predicting and computing $\hat{y}_{test}[i]$, we use the fitted model and X_{test}. No information about y_{test} is used, we pretend it is unknown.

3. Both training- and test-set data must come from the same *population*, i.e. all objects in the test set could have been part of the training set, and vice versa, and we think of it as random which objects are in which subsets.

14.2 The cross-validation algorithm

Cross-validation is used for computing the $MSEP$ or CER from the previous section. We split the data set into n training-set data objects and m test-set data objects, i.e. the full data set has $L = n + m$ data objects. However, this splitting can obviously be done in many ways. All the data objects in the training-set *could* have been in the test-set instead, and vice versa. Thus, our computed $MSEP$ or CER are random variables affected by this choice. The cross-validation idea is to repeat the splitting many times in such a way that *all* data objects have been part of the test-set once.

14.2.1 The leave-one-out cross-validation

This is the most extreme version of cross-validation, but is frequently used. It means we use one single data object as test-set ($m = 1$). We then use the rest of the data objects ($n = L - 1$) as training-set, and use this to predict the test-set response, computing $MSEP$ or CER as shown above. Then, this test-set data object is put back into the training-set and another data object takes the role as test-set. Everything is re-computed, and another error is computed, added to the first. We then loop through the entire data set this way, producing new errors that we add up to the total error.

We may sketch the algorithm like this:

```
# Assume we have a full data set (y,X)
# The full data set has L data objects
err <- rep(0,L)
for(j in 1:L){
    # Split into (y.train,X.train) and (y.test,X.test)
    idx.test <- j
    y.test <- y[idx.test]     # elements idx.test in y
    X.test <- X[idx.test,]    # rows idx.test in X
    y.train <- y[-idx.test]   # elements NOT idx.test in y
    X.train <- X[-idx.test,]  # rows NOT idx.test in X

    # Fitting the model...
    # Predicting y.test, we denote this y.test.hat

    # Computing error...
    err[j] <- (y.test-y.test.hat)^2
} # next split...
MSEP <- sum(err)/L
```

As you can see the fitting and prediction has been left as comments here, and must be filled in, depending on the method we use. The cross-validation itself will work for any method plugged in.

Notice that if the full data set has $L = 1000000$ data objects, the loop above will run a million times. The model is (re-)fitted inside the loop, and if this takes some time, the looping may take a long time. If the fitting takes 1 second, then 1 million iterations will take a week to complete!

Also, if the data set is huge, the training-set is almost identical in each iteration, and the fitted models will be almost identical. In a huge data set almost nothing changes by leaving out a single data object. Leaving out larger subsets of the data is required to get some real variation between the fitted models.

14.2.2 C-fold cross-validation

In general we split the full data set into C (almost) equally large subsets, and perform cross-validation by looping over these subsets, using each in turn as test-set. This means the loop from above will only run C times. A typical value is $C = 10$. We use the term *segment* for these subsets, i.e. we have C cross-validation segments.

How do we decide which data objects belong to which segment? We could sample at random, but this can produce problems if the data set is small(ish) or is skewed with respect to the response. Imagine you have a classification problem, and one of the classes (factor levels) have few data objects. Then, if all these end up in the same segment, you will never be able to test the ability to recognize this class! If this segment is the test-set, then the class is lacking in the training-set, and no method in the world will be able to recognize it. If it is part of the training-set, it is lacking in the test-set, and you will never be asked to recognize it. Thus, all classes should be divided into as many different segments as possible. Here is a trick to assure this in R:

```
# We want C segments
idx <- order(y)
y <- y[idx]
X <- X[idx,]
segment <- rep(1:C,length.out=L)
```

The vector `segment` will now have one element for each data object. Each element is an integer from 1 to C and those objects with the same integer belong to the same segment. Using this `segment` vector, two consecutive data objects will never belong to the same segment, and since we first sorted them this will guarantee maximum spread over the segments.

The C-fold cross-validation algorithm can then be sketched as

```
err <- rep(0,L)
for(j in 1:C){
    # Split into (y.train,X.train) and (y.test,X.test)
    idx.test <- which(segment==j)
    y.test <- y[idx.test]     # elements idx.test in y
    X.test <- X[idx.test,]    # rows idx.test in X
    y.train <- y[-idx.test]   # elements NOT idx.test in y
    X.train <- X[-idx.test,]  # rows NOT idx.test in X

    # Fitting the model...
    # Predicting y.test, we denote this y.test.hat

    # Computing error...
    err[idx.test] <- (y.test-y.test.hat)^2
} # next split...
MSEP <- sum(err)/L
```

The only difference to the leave-one-out code is the way we find `idx.test` and that we compute the error for several data objects in each iteration.

14.3 Example: Temperature and radiation

14.3.1 The data

From the daily weather data set we want to see how maximum air temperature depends on global radiation (incoming solar energy). In the file `daily.maxtemp.rad.RData` we have a data.frame with the columns `Air.temp.max` and `Radiation`.

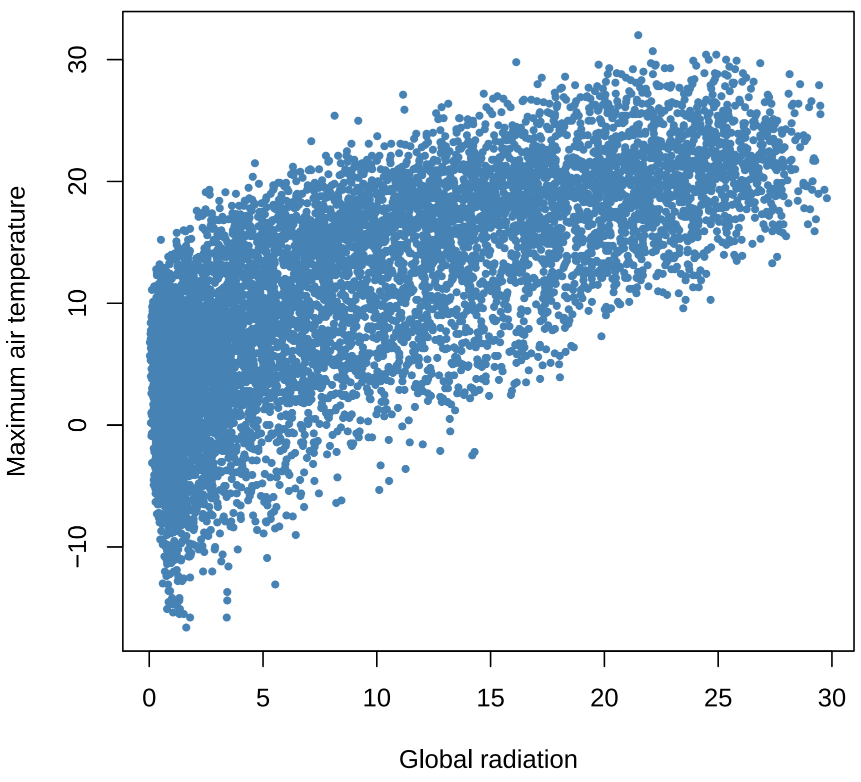


Figure 14.2: Each of the 9108 dots represent a daily weather observation of maximum air temperature and global radiation.

In Figure 14.2 we have plotted the response (maximum air temperature) versus the single explanatory variable (radiation).

14.3.2 Fitting a linear model

Most people would say that this looks pretty much like a straight line relation, and fit a simple linear model:

$$y = \beta[1] + \beta[2]x + e$$

where y is the temperature and x is the radiation. The result looks like this:

```
> lm.fit <- lm(Air.temp.max~Radiation,
               data=daily.maxtemp.rad)
> summary(lm.fit)

Call:
lm(formula = Air.temp.max ~ Radiation, data = daily.maxtemp.
    rad)

Residuals:
     Min       1Q   Median       3Q      Max
-21.3020  -3.8528   0.1647   4.1500  16.0160

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.709310   0.090538   29.93   <2e-16 ***
Radiation   0.818983   0.007475  109.56   <2e-16 ***
---
Signif. codes:  0   ***   0.001   **   0.01   *   0.05
       .   0.1        1

Residual standard error: 5.762 on 9106 degrees of freedom
Multiple R-squared:  0.5686,  Adjusted R-squared:  0.5686
F-statistic: 1.2e+04 on 1 and 9106 DF,  p-value: < 2.2e-16
```

We can see the slope is estimated to 0.82, and in Figure 14.3 we have added the fitted line to the scatterplot of the data.

14.3.3 The local model alternative

This data set has many data objects ($n = 9108$) and only a single explanatory variable ($p = 1$). This is a typical situation where a local model would often give us a better description of the relation than a simple linear model. Notice from Figures 14.2 and 14.3 how dense the data are, and almost any point here will have plenty of near neighbours. A nearest-neighbour approach is worth trying.

We will consider a simple moving average model. We have made a function for moving average or knn-regression, and it looks like this:

```
knn.regression <- function(X.test,y.train,X.train,K=3){
  # Simple local regression method for one single
     explanatory variable,
  # i.e. both X.test and X.train must be vectors
  n <- length(X.train)
  m <- length(X.test)
  y.test.hat <- rep(NA,m)
  for(i in 1:m){
    d <- abs(X.test[i]-X.train)
    idx <- order(d)
    y.test.hat[i] <- mean(y.train[idx[1:K]])
  }
  return(y.test.hat)
}
```

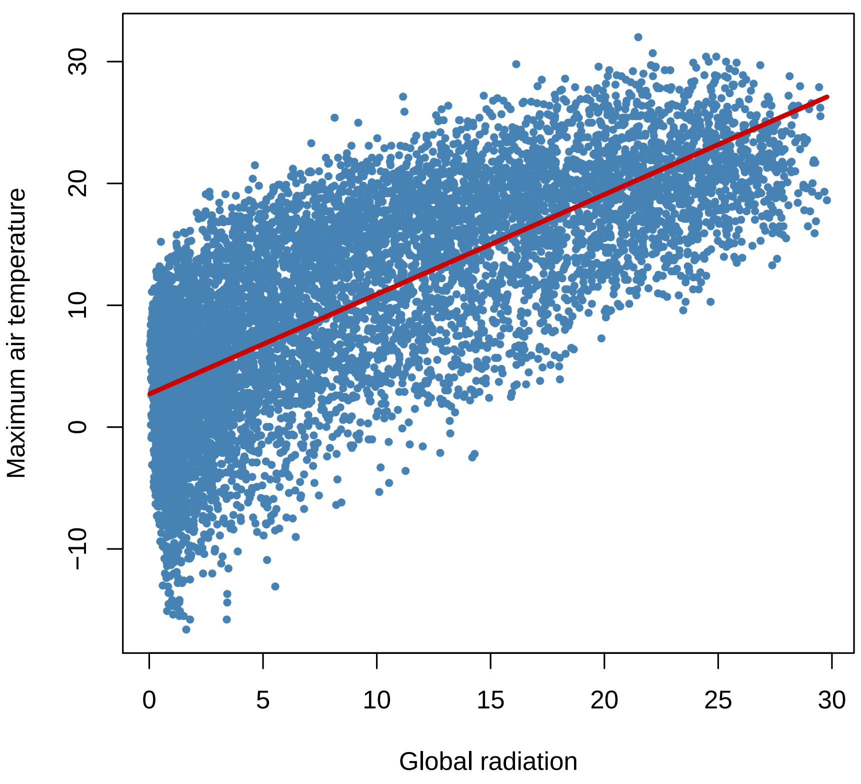


Figure 14.3: The line shows the fitted linear model.

Notice that this simple version of knn-regression *only* take as input a single explanatory variable, i.e. both `X.test` and `X.train` must be vectors (cannot have more than one column). For a given choice of `K` (number of neighbours) we choose any radiation value within the span of this data set, and compute the corresponding predicted maximum air temperature by this function. The problem is how to choose a proper `K`. We will first use cross-validation to give us some hint on how to choose a proper `K`.

First we need to decide upon some values of `K` that we want to try out. We choose

```
K.values <- c(50,100,300,500,1000,2000,3000)
nK <- length(K.values)
```

Since computations will take some time, we start out by a smallish number of `K` values, and may increase this later. For each value in `K.values` we want to compute the *MSEP* from equation (14.1), as a measure of prediction error. Thus we need a vector

```
MSEP <- rep(0,nK)
```

to store the computed prediction error for each choice of `K`.

Before we start the cross-validation looping, we have to divide the full data set into segments. We choose to use 10-fold cross-validation, i.e. we split the data set into $C = 10$ segments:

```
L <- dim(daily.maxtemp.rad)[1]
C <- 10
idx <- order(daily.maxtemp.rad$Air.temp.max)
daily.maxtemp.rad <- daily.maxtemp.rad[idx,]
daily.maxtemp.rad$Segment <- rep(1:C,length.out=L)
```

which means the data.frame `daily.maxtemp.rad` has been sorted by the `Air.temp.max` and then got a new column named `Segment`. All data objects having the same value of `Segment` belong to the same segment.

We can then start the looping:

```
attach(daily.maxtemp.rad)
# The first loop is over the K-values
for(i in 1:nK){
  cat("For K=",K.values[i], sep="")
  # Then we loop over the cross-validation segments
  err <- rep(0,L)
  for(j in 1:C){
    # Split into (y.test,X.test) and (y.train,X.train)
    idx.test <- which(Segment==j)
    y.test <- Air.temp.max[idx.test]
    X.test <- Radiation[idx.test]
    y.train <- Air.temp.max[-idx.test]
    X.train <- Radiation[-idx.test]

    # Predicting y.test, pretending its unknown
    y.test.hat <- knn.regression(X.test,y.train,X.train,
                                 K=K.values[i])

    # Computing error...
    err[idx.test] <- (y.test-y.test.hat)^2
    cat(".")
  } # next segment...
  MSEP[i] <- sum(err)/L
  cat("the MSEP is",MSEP[i],"\n")
} # next K-value
```

Notice the double looping, since we first have to consider each element in `K.values`, and then for each of them consider all possible splitting into test- and training-sets. This makes the computations slow, and this is also why we add some `cat` statements here and there. It is good to see some output during computations, just to verify that things are proceeding as they should. Here is the output we get:

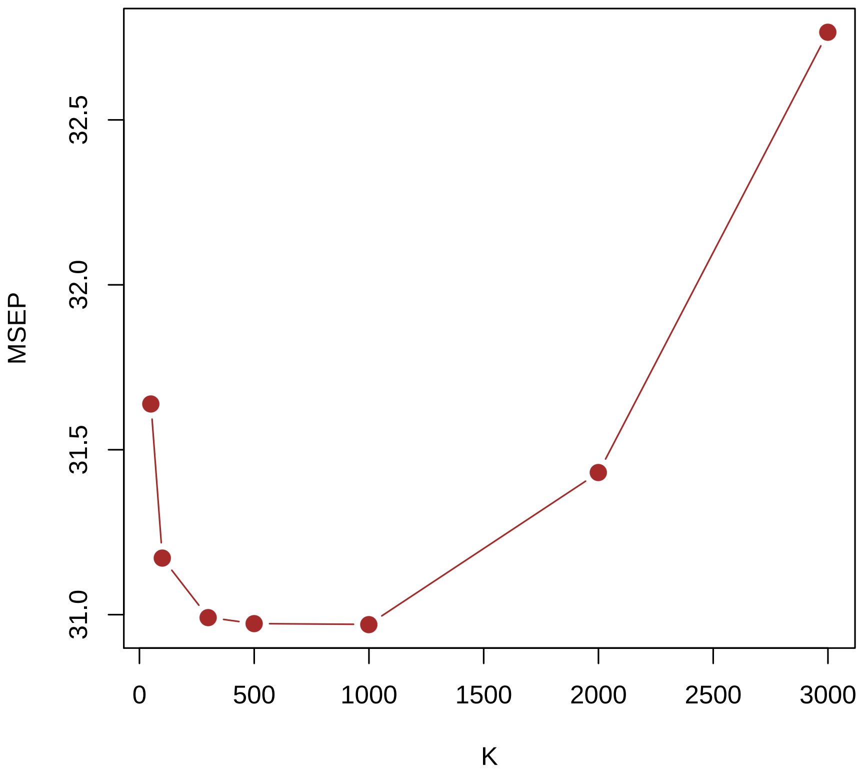


Figure 14.4: The brown dots show the $MSEP$ values computed for the corresponding choice of K. The shape is typical in the sense that both a too small and too large choice of K produce larger errors than some 'optimal' value. In this case it seems like the optimal choice of K is somewhere between 500 and 1000.

```
For K=50..........the MSEP is 31.63935
For K=100..........the MSEP is 31.17152
For K=300..........the MSEP is 30.99186
For K=500..........the MSEP is 30.97284
For K=1000..........the MSEP is 30.97058
For K=2000..........the MSEP is 31.4305
For K=3000..........the MSEP is 32.76581
```

In Figure 14.4 we have plotted how the prediction error $MSEP$ varies over the different choices of K. It looks like a K close to 1000 is a good choice for this particular data set, giving the best balance between a too small K (inflating the variance) and a too large K (inflating the bias).

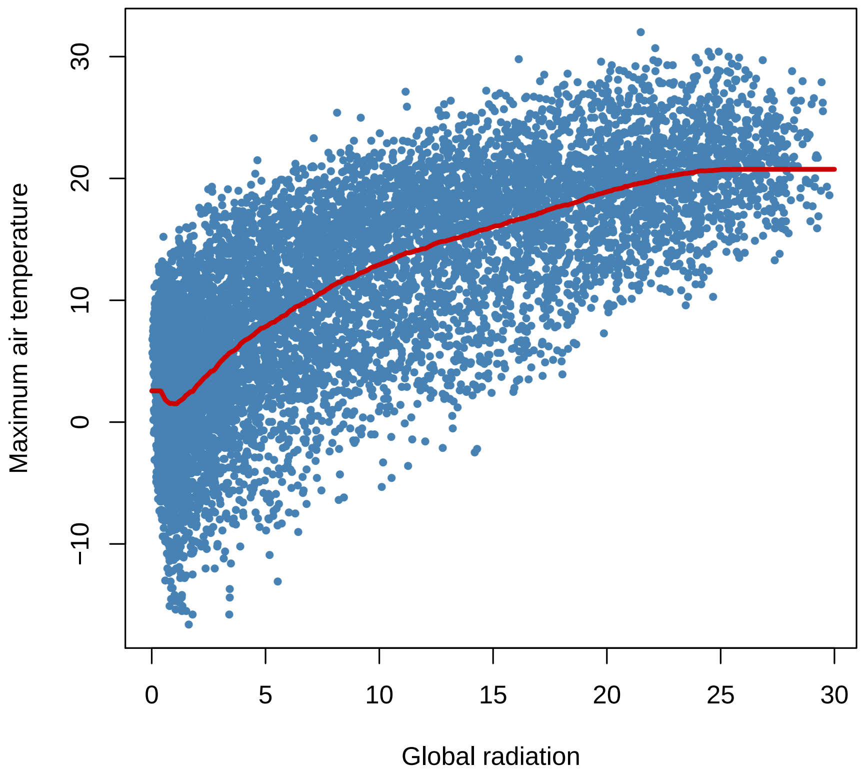


Figure 14.5: The curve shows the relation between radiation and maximum air temperature as described by the local knn-regression method.

14.3.4 Fitting the final local model

Based on the model selection we choose $K = 1000$. We can now predict maximum air temperature for 'all' values of radiation within the span of the data set, using the `knn.regression` function from above. The result is shown in Figure 14.5. Notice the differences to the linear model. First, we see that at very large radiations, there is no longer any impact on temperature, the curve is flat. This is, however, an artifact of the method, since any moving average will tend to show 'flatness' as we move towards the outer range of the explanatory variables. More important, perhaps, is that at very low radiations the relation between radiation and temperature is actually reversed! Notice also that in this region there are *a lot* of data. In fact, around15% of all the data have radiations between 0 and 1. What does this mean?

These are daily measurements, and the low-radiation days are all in mid-winter. At this time of the year an overcast day gives extremely small radiation (very close to 0), but temperatures are usually not very low. A day with clear skies will have slightly larger radiation (close to 1), but at the same time also very cold weather with lower maximum temperature. Thus, the negative effect of radiation on maximum temperature is actually present at one out of six days during a year here at NMBU! This is not something we can 'see' by plotting since the data are so dense, and the linear model, that 'looks' fine to our eye, will completely obscure this effect.

As a conclusion, we can say that a fine-tuned (by model selection) local model may detect relations that we otherwise could have missed.

14.4 Exercises

14.4.1 Classification of bacteria again

Expand last weeks exercise on classification of bacteria, and implement a cross-validation to estimate the CER (classification error rate). Use the code from the radiation-maxtemp example above. Try different choices for K to see (approximately) which choice of K is optimal. HINT: Try small values for K.

14.4.2 Faster computations

Cross-validation can be time-consuming, and we should make efforts to speed up the code as much as possible. In the procedure sketched in this chapter we have two loops. The outer loop is over the different values of K, the inner loop is the cross-validation. This can be improved.

In the inner loop we use a KNN-type of method, and the slow part of such methods is the computation of all the distances. If we are going to predict the response for data object i, we must compute the distances from this to all data objects of the training set. Notice that this is first done once for the first choice of K, then it is re-computed again for every new choice of K that we try out! These are exactly the same distances, and it is silly to re-compute them.

Instead we eliminate the looping over the K-values, i.e. the outer loop is the cross-validation. Then, for each cross-validation segment, we try *all* K-values. This means the function we use (`knn.classify`) should be modified to take as input a vector of K-values instead of a single value. Then, after the distances have been computed, the classification is done for each choice of K, and all results are returned as output. Try to make this modification to the code from the exercise above.