DR LEE FAWCETT

MISS AMY CHADWICK

HOW TO USE R WITHIN YOUR DEGREE

NEWCASTLE UNIVERSITY

Contents

1 Background

Objectives

This chapter will give a description of R and an introduction to the course.

1.1 *History*

R is an implementation of S, where S is a programming language that first appeared in 1976 and was created by John Chambers alongside colleagues at Bell Laboratories. There are some important differences between the programmes, however the majority of code written for S runs unaltered under R.

Ross Ihaka and Robert Gentlemen created R at the University of Auckland, New Zealand, thus naming the programme partly after their first names and partly as a play on the name S. A special debt is owed to John Chambers due to his contributions and encouragement in R's early days, and later becoming a member of the R Development Core Team which currently develops R.

1.2 *What is R?*

R is a programming language and software environment for statistical computing and graphics. For example, R is widely used by *Google*, *IBM*, *Shell*, *Thomas Cook*, *Facebook* and other familiar names.

The R language is mostly used by statisticians and data miners for developing statistical software and analysing data, although it can also be used as a general matrix calculation toolbox with comparable benchmark results to many other software packages. It has been shown in recent polls, surveys and studies that R's popularity has increased substantially over recent years.

R uses a command line interface, though several graphical user interfaces are available. The system provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering and others) and graphical techniques.

R is highly extensible through the use of user-submitted libraries and includes various facilities for data manipulation, calculation and graphical display, such as:

- an effective data handling and storage facility;
- a suite of operators for calculations on arrays, in particular matrices;
- a large, coherent, integrated collection of intermediate tools for data analysis;
- graphical facilities for data analysis and display either on-screen or on hardcopy;
- quality graphs that can include mathematical symbols and formulae;
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

You will use R throughout your degree at Newcastle in various assignments and project work. We will be using RStudio, which is an R IDE (Integrated development environment).

1.3 Installing R

R and RStudio are installed on all University machines and are available as Free Software under the terms of the Free Software Foundation's General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS, so you can freely install R and RStudio on your own computer. See

`http://www.ncl.ac.uk/maths/students/teaching/installingr/`

for more details.

1.4 Previous computing knowledge

This course is intended to teach you the basics of programming. No previous programming knowledge is assumed. It is crucial that you understand all the material in this course and have completed all the relevant exercises/Numbas tests before September.

1.5 Movie data set

The internet movie database (IMBD), `http://imdb.com/`, is a website devoted to collecting movie data supplied by studios and fans. It claims to be the biggest movie database on the web and is run by amazon. See

`http://imdb.com/help/show_leaf?about`

and

`http://imdb.com/help/show_leaf?infosource`, for more information about `http://imdb.com/`, including information about the data collection process. IMDB makes their raw data available at

`http://uk.imdb.com/interfaces/`.

We will use a selection of movies from the IMDB as our main dataset throughout this course to demonstrate some of R's data manipulation and graphical capabilities. Movies were selected for inclusion if they had a known length, had been rated by at least one IMDB user and had an mpaa (motion picture association of America) rating. The dataset contains the following fields:

- **Title.** Title of the movie.
- **Year.** Year of release.
- **Budget.** Total budget in US dollars. If the budget isn't known, then it is stored as '-1'.
- **Length.** Length in minutes.
- **Rating.** Average IMDB user rating.
- **Votes.** Number of IMDB users who rated this movie.
- **r1-10.** Percentage(to nearest 5%) of users who rated this movie a 1, ..., 10
- **mpaa.** MPAA rating. A movie is rated based on who it is suitable to be viewed by, e.g. 'PG-13' if a movie should have parental guidance for children under the age of 13.
- **Action, Animation, Comedy, Drama, Documentary, Romance, Short.** Binary variables indicating if movie was classified as belonging to that genre. A movie can belong to more one genre. See for example the film *Ablaze* in Table 1.1.

There are a total of 24 variables and 4847 films. The first few rows are given in Table 1.1. This, however, is only a subset of the data, the actual data set contains information on over 50,000 movies.

This dataset lends itself very well to learning how to use R due to its size and the many different types of variables it contains. We will be making use of this dataset throughout these notes. The dataset is available in plain text format at:

`www.mas.ncl.ac.uk/∼nlf8/basicR/movies.txt`

				Voting statistics					Movie Genre							
Title	Year	Length	Budget	Rating	Votes	r1	...	r10	mpaa	Action	Animation	Comedy	Drama	Documentary	Romance	Short
A.k.a. Cassius	1970	85	-1	5.7	43	4.5	...	14.5	PG	0	0	0	0	1	0	0
AKA	2002	123	-1	6.0	335	24.5	...	1.5	R	0	0	0	1	0	0	0
Alien Vs. Pred	2004	102	45000000	5.4	14651	4.5	...	4.5	PG-13	1	0	0	0	0	0	0
Abandon	2002	99	25000000	4.7	2364	4.5	...	4.5	PG-13	0	0	0	1	0	0	0
Abendland	1999	146	-1	5.0	46	14.5	...	24.5	R	0	0	0	0	0	0	0
Aberration	1997	93	-1	4.8	149	14.5	...	4.5	R	0	0	0	0	0	0	0
Abilene	1999	104	-1	4.9	42	0.0	...	24.5	PG	0	0	0	1	0	0	0
Ablaze	2001	97	-1	3.6	98	24.5	...	14.5	R	1	0	0	1	0	0	0
Adominable Dr	1971	94	-1	6.7	1547	4.5	...	14.5	PG-13	0	0	0	0	0	0	0
About Adam	2000	105	-1	6.4	1303	4.5	...	4.5	R	0	0	1	0	0	1	0

Table 1.1: The first ten rows of the movie data set. **Credit:** This data set was initially constructed by Hadley Wickham at `http://had.co.nz/`

Conclusion

You should now:

- be able to give a brief description of the statistical software R;
- know the background of R;
- understand why R is used;
- know how to install R on your own computer;
- recognise the movie data set that we will be using throughout the course.

Interactive learning

You should work through this booklet ”interactively”, working through **all** the R code yourselves at the same time as it is presented, no matter how easy it looks. You should look out for coloured boxes too:

- red indicates you should complete a CBA (Computer Based Assessment that can be found on the webpage);
- blue indicates you should complete a Practical (short worksheet of questions designed to practice what you have read and challenge you - also found on the webpage with solutions.)

There is also an online book available for a more thorough description of things, available from our webpage: `https://cran.r-project.org/doc/manuals/R-intro.pdf`

2 Introduction to R

Objectives

In this chapter you will learn about the basics of R. To make the most of these notes you need to try all the R code as you read. To do this please install R on your own machine or use a university computer, see Section 1.3 for more information.

If you have any difficulty in doing this you can contact either:

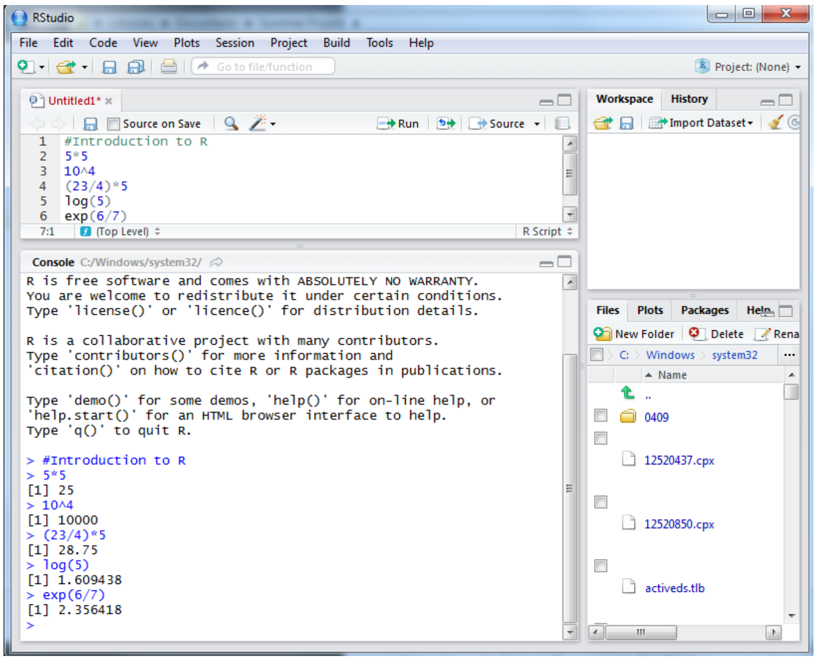
- Christian Lawson-Perfect at *christian.perfect@newcastle.ac.uk*;
- Christopher Graham at *Christopher.Graham@newcastle.ac.uk*.

2.1 Accessing R

Assuming you are on a University machine, RStudio can be found via

Start > All Programs > Statistical Software > R > RStudio.

Once opened, select File > New > R Script, or Hold Ctrl+Shift+N. This opens an R Script document in the top left hand corner. Although you don't necessarily need an R Script, it makes life easier as you can edit and re-run code from the script. Any R code written directly onto the 'Console' (bottom left-hand corner) section cannot be rewritten.



```
#Introduction to R
5*5
10^4
(23/4)*5
log(5)
exp(6/7)
```

```
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.

> #Introduction to R
> 5*5
[1] 25
> 10^4
[1] 10000
> (23/4)*5
[1] 28.75
> log(5)
[1] 1.609438
> exp(6/7)
[1] 2.356418
>
```

Figure 2.1: Print screen of R: the top left and bottom left hand corners are the R Script and Console which you will mainly be using; the top right hand corner represents the workspace/history section; and the bottom right hand corner shows your files/plots/the help section.

2.2 A simple R session

At its most basic, R can be used as a calculator. You can type calculations into your R script document, highlight them and either click 'Run' (in the top right corner of your RScript) or Ctrl+Enter. You will then see them run in blue through the console below, with the corresponding answers in black. This is show in Figure 2.1.

Table 2.1 includes some basic maths symbols. Remember the use of brackets can be important when completing some longer calculations, in the same way as they are used on your calculators.

Symbol	**Meaning**
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`^`	To the power of
`exp()`	Exponential
`log()`	Logarithm (base e)
`%%`	Modulus

Table 2.1: A table showing basic math symbols that you will use within R

The modulus operator returns the remainder when the first number is divided by the second. This seemingly simple operator is actually a very powerful tool for mathematicians, especially those who specialize in Number Theory. It can also be very useful for data manipulation in R and is probably most often seen written as ”mod”, e.g. 4 mod 3.

The `#` symbol is used for commenting. We use comments to describe what a piece of code is doing. That way, when we look at the code in a few months/years we have a record of what our code does and why it was written. This can be especially useful when it comes to your assignments to label questions or write your own notes.

R essentially ”ignores” any line that starts with `#`.

NOW THAT we have explained the use of RScript, the following notes shall be written as though directly using the console in R. When using the console, type your calculation and click 'Enter' to run it. You will see the same thing happens as when using the RScript, except now it isn't as easy to go back and edit what you have done.

You can edit in the console window - if you click the up-arrow, it takes you back to your last line of code and you can edit and re-run.

Remember whilst following these notes you should be trying to run each example code yourself.

We recommend always using RScript unless fully confident.

The > symbol is the command prompt in the R console, the point at which you can type your code. You don't need the > symbol if you are typing in your RScript. When the code is executed R will generate the > symbol for you.

Here are some simple R calculation examples:

```
> #This is a comment and is ignored by R
> 5*5

[1] 25

> 10.2/6

[1] 1.7
```

and more 'advanced' operations:

```
> #Powers
> 2^3

[1] 8

> #Exponential
> exp(1.5)

[1] 4.481689

> #Logs
> log(10)

[1] 2.302585

> #Modulus
> 4 %% 3

[1] 1
```

The `[1]` tells you how many items you have in your output up to, and including, the first element in that line.

2.3 Movie data set

The movie data set considered at the end of Chapter 1 will be used throughout these notes. Now that you are in RStudio you will need to install the movies database so that you can view it. To do this you will first need to download the associated R package for this course:

```
> install.packages("mas1343",repos="http://R-Forge.R-project.org",type="source")

The downloaded source packages are in
        'C:\Users\B2017960\AppData\Local\Temp\Rtmpqierxe\downloaded_packages'

> #To load the package use
> library(mas1343)
```

Packages are collections of R functions, data, and compiled code in a well-defined format. The directory where packages are stored is called the library. R comes with a standard set of packages. Others are available for download and installation. Once installed, they have to be loaded into the session to be used.

We are using a package here to allow easy access to datasets and commands that have been specifically set-up for use in this course. The package is called "mas1343" after an old stage 1 R course that first year single honours Mathematics/Statistics students were previously taught.

To load the movies dataset, type:

The example only uses budget and length but you can use the `data()` function to extract any column/row you want.

```
> data(movies)
> #This extracts the movies dataset from the package
> data(Budget)
> #This extracts the budgets column from the full movies dataset.
> data(Length)
> #This extracts the length column from the full movies dataset.
```

You can view the first couple of movies by typing:

See Table 1.1 in Chapter 1.

```
> head(movies)
```

At the start of each new R session you will have to type the `library(mas1343)` and `data(movies)` commands - but you **won't** have to use `install.packages` each time.

2.4 *Assignment operations*

In R, we assign values to an object using the equals sign. For example:

```
> x = 5
> x = x + 1
> x

[1] 6
```

Notice that when we type `x = 5`, R doesn't display or print any output to the screen. Don't confuse this with R not doing anything, if there is no error message then R has always done something! If we want to see what value has been assigned to the variable we can type `x` and click enter. Equivalently, we can surround the expression with brackets. For example:

```
> (x = 2*x)

[1] 12
```

You can also use the `<-` operator for assignment. This is, for almost all situations, identical to the `=` operator.

This is also where the 'Workspace' section in the top right hand corner becomes useful. R remembers all variables you have assigned, until you overwrite them. So by looking in your 'Workspace' you can quickly see what variables you have in R at the moment and what they are assigned to.

You can also view available variables with the `ls()` command:

```
> #You will get a different list of variables
> ls()

[1] "x"
```

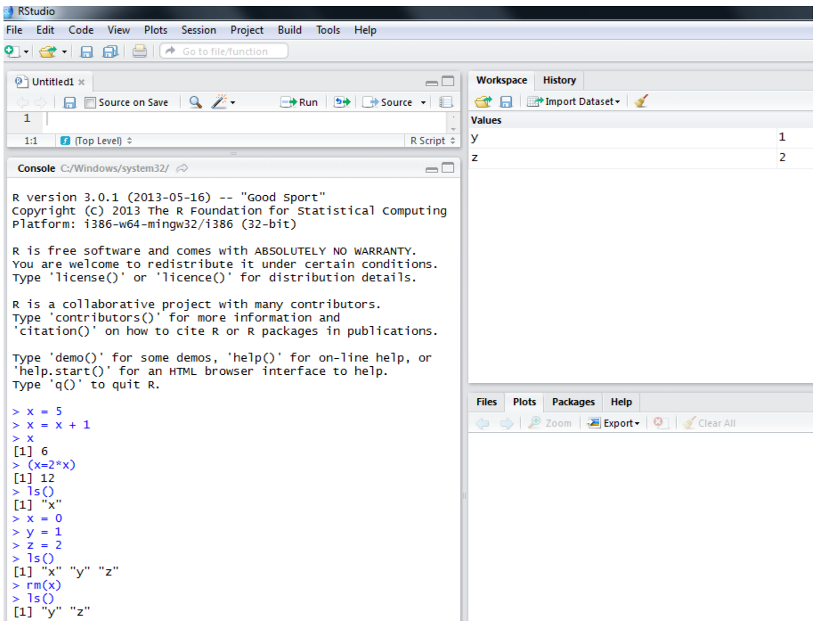
To delete a variable in R, we use the `rm` function. For example:

```
> x = 0; y = 1; z = 2;
> ls()

[1] "x" "y" "z"

> #Remove x
> rm(x)
> ls()

[1] "y" "z"
```

See Figure 2.2 which shows a print screen of the use of these functions in the ‘Console’ and how they affect the ‘Workspace’. We can remove everything in the ‘Workspace’ using `rm(list=ls())`:

```
> #Remove everything in your workspace
> rm(list=ls())
```

We recommend running `rm(list=ls())` at the beginning of each new R session. This stops you relying on previously stored variables and makes your code more portable. You can also remove everything in the ‘Workspace’ by clicking the brush symbol next to ‘Import Dataset’, see the right hand corner of Figure 2.2.



Values	
y	1
z	2

```
R version 3.0.1 (2013-05-16) -- "Good Sport"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-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.

> x = 5
> x = x + 1
> x
[1] 6
> (x=2*x)
[1] 12
> ls()
[1] "x"
> x = 0
> y = 1
> z = 2
> ls()
[1] "x" "y" "z"
> rm(x)
> ls()
[1] "y" "z"
```

Figure 2.2: Print screen of R using the `ls()` and `rm(x)` commands. You can see what is happening in the ‘Console’ in the bottom left hand corner. Notice how the ‘Workspace’ in the top right hand corner shows the variables.

Computer Based Assessments

You should now take the opportunity to complete the Computer Based Assessment (CBA 1) which will cover the material you have read so far. We would advise not continuing with the material until you are confident with all the questions covered in CBA 1.

2.5 Data types

R has a variety of data types:

Notice how in the `doubles` section R displays the value as an integer, without the decimal point, if it has no values after the decimal point.

```
> #Logicals
> #Just return TRUE or FALSE nothing else
> (v = TRUE)

[1] TRUE

> #characters
> #Usually (but not always) non-numeric values
> #always entered surrounded by quotation marks
> (w = "fred")

[1] "fred"

> #doubles
> #A numeric value, any real number
> (x = 5.0)

[1] 5
```

and also some "special" data types:

```
> (y = 5/0) #infinity

[1] Inf

> (z = y-y) #Not a number

[1] NaN
```

Another important data type in R is `NA` ("Not Available.") This is used to represent missing values. A list of data types is given in Table 2.2.

Type	Example 1	Example 2	Example 3	Example 4
Doubles	2	3.1242	-45.6	4e-10
Logicals/Boolean	TRUE	FALSE		
Characters/String	"FRED"	"x"	"Male"	"TRUE"
Infinity	Inf	5/0		

Table 2.2: Summary of data types in R.

2.6 Vectors

Vectors are the most basic of all data structures, but are used in almost all R code. An R vector contains n values of the same type, where n can be zero. For example:

The `c` command stands for "concatenate", and is a built-in function used to create a vector. Therefore, it is wise to avoid assigning any values to `c` - doing so can over-write such built-in functions!

```
> c(0, 1, 2, 3, 4, 5)

[1] 0 1 2 3 4 5

> (my_first_vec = c(0, 1, 2, 3, 4, 5))

[1] 0 1 2 3 4 5

> (my_second_vec = c("Male", "Female", "Male"))

[1] "Male"   "Female" "Male"
```

In the above code, we create a vector of **doubles**. Notice how R then shows you your vector. We then assigned the vector to the variable `my_first_vec`. Now whenever we type `my_first_vec` it will represent our vector of **doubles**. Notice how this now appears in your workspace. We can create vectors of any data type. For example, `my_second_vec` is a vector of **characters**.

In R, when we type:

```
> x = 5; y = "Fred"
```

we have actually created vectors of doubles and characters of length one (when $n = 1$). There are special functions in R to determine the variable type:

```
> x = 5
> is.double(x)

[1] TRUE

> is.character(x)

[1] FALSE

> is.vector(x)

[1] TRUE
```

To determine the length of a vector in R, we use the `length` function:

```
> length(my_first_vec)

[1] 6

> length(my_second_vec)

[1] 3
```

To create sequences of numbers we use the `seq` command. For example:

```
> (x1 = seq(1, 6))

[1] 1 2 3 4 5 6

> (x2 = seq(-4, 4, by=2))

[1] -4 -2  0  2  4
```

Table 2.3 summarises these basic R functions, as well as giving a few extra ones which you might find useful.

Command description	Example	Result
Length	`length(x)`	4
Reverse order	`rev(x)`	3,5,5,1
Sort	`sort(x)`	1,3,5,5
Sum	`sum(x)`	14
Extract unique elements	`unique(x)`	1,5,3
Indices of particular elements	`which(x==5)`	2,3

Table 2.3: Useful vector functions. In the examples, `x = c(1,5,5,3)`. Check the associated R help for further information.

2.6.1 Vector operations

When our data is in vector structure, we can apply standard operations to the entire vector. For example:

```
> (x = seq(-4, 4))

[1] -4 -3 -2 -1  0  1  2  3  4

> x*x;

[1] 16  9  4  1  0  1  4  9 16

> x - 5

[1] -9 -8 -7 -6 -5 -4 -3 -2 -1

> x + x

[1] -8 -6 -4 -2  0  2  4  6  8
```

So we can treat our vector in the same way we treated numbers in section 2.2 and complete basic calculations with it.

2.6.2 Extracting elements from vectors

R has a number of useful methods that we use to extract subsets of our data. For example to pick out particular elements:

```
> my_first_vec[2]

[1] 1

> my_second_vec[2:3]

[1] "Female" "Male"

> my_first_vec[4:2]

[1] 3 2 1
```

We see that the first line of R code picks out the second element of the vector; the second line of R code picks out the second and third; whereas the third line of R code picks out the fourth to second elements in the order specified.

We can also use other arguments. For example to remove the last entry in the vector, we use the `length` function that we saw earlier:

```
> l = length(my_first_vec)
> #Notice the brackets!
> my_first_vec[1:(l-1)]

[1] 0 1 2 3 4
```

We determine the length of the vector using the `length` function and select particular elements using the `[·]` operator.

Computer Based Assessments

You should now take the opportunity to complete the Computer Based Assessments (CBA 2a and CBA 2b) which will cover the material you have read so far. We would advise not continuing with the material until you are confident with all the questions covered.

2.7 Logical vectors

R supports the logical elements: `TRUE` and `FALSE`. Boolean algebra tells us how to evaluate the truth of compound statements. Table 2.4 gives a summary of R operations. For example,

Reading `!A` as **NOT** A, we see that because A has been assigned as TRUE, `!A` (**NOT** A) must be FALSE.
Read `A & B` as A **AND** B.
Read `A | B` as A **OR** B.

```
> A = TRUE; B = FALSE
> !A

[1] FALSE

> !B

[1] TRUE

> A & B

[1] FALSE

> A | B

[1] TRUE
```

Boolean	A	B	$\bar{A}$	$\bar{B}$	$A \cap B$	$A \cup B$
R	A	B	!A	!B	A & B	A | B
	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE

Table 2.4: Truth table for Boolean operations.

2.7.1 Using logicals for sub-setting vectors

We can construct vectors of logical operators and use them to take subsets of vectors. For example:

```
> (logic1 = c(TRUE, FALSE, TRUE, FALSE))

[1]  TRUE FALSE  TRUE FALSE

> (vec1 = seq(1, 4))

[1] 1 2 3 4

> vec1[logic1]

[1] 1 3
```

We can see in this example that our `logic1` vector is sub setting only the numbers in the sequence that relate to the `TRUE` elements. So we include 1 and 3. We remove 2 and 4 because they correspond to the `FALSE` elements in the `logic1` vector.

2.7.2 Relational Operators

When programming it is often necessary to test relations for equality and inequality. To do this in R we use the relation operators. First let's define some variables:

```
> x = 5; y = 7
```

To test for equality we use `==`:

```
> x == 5 #Does x = 5?
```

```
[1] TRUE
```

```
> x == y #Does x = y
```

```
[1] FALSE
```

So in the first line of R code we are asking R if the assigned x **is** equal to 5, and R returns a `TRUE/FALSE` answer. Similarly, to test for inequality we use `!=`:

```
> x != 5 #Or !(x==5)
```

```
[1] FALSE
```

```
> y != x
```

```
[1] TRUE
```

There are also commands for greater/less than:

```
> y > 6 #Is y > 6?
```

```
[1] TRUE
```

```
> x >= 5 #Is x greater than or equal to 5?
```

```
[1] TRUE
```

```
> x <= y #Is x less than or equal to y?
```

```
[1] TRUE
```

Table 2.5 gives a summary of the commands.

Operator	Tests for	Example	Result
==	Equality	`x == 5`	`TRUE`
!=	Inequality	`x != 5`	`FALSE`
<	Less than	`x < 5`	`FALSE`
<=	Less or equal	`x <= 5`	`TRUE`
>	Greater	`x > 5`	`FALSE`
>=	Greater or equal	`x >= 5`	`TRUE`

Table 2.5: Summary of R relational operators. The example is for `x = 5`.

We can also apply these techniques to vectors. For example:

```
> #Generates a sequence
> (vec2 = seq(0, 10, by=2.5))

[1]  0.0  2.5  5.0  7.5 10.0

> #Asks if the numbers are greater than 3
> vec2 > 3

[1] FALSE FALSE  TRUE  TRUE  TRUE

> #Asks if the numbers are less than 9
> vec2 < 9

[1]  TRUE  TRUE  TRUE  TRUE FALSE

> #Asks if the numbers are between 3 and 9 (exclusively)
> (vec2 > 3) & (vec2 < 9)

[1] FALSE FALSE  TRUE  TRUE FALSE
```

We can see R has considered each number in the vector in turn and returned logicals determining whether or not it is greater than three/less than nine/between the two.

We can also combine logical operators:

```
> vec2 > 3

[1] FALSE FALSE  TRUE  TRUE  TRUE

> !(vec2 > 3)

[1]  TRUE  TRUE FALSE FALSE FALSE
```

2.7.3 Vector Partitions

We can construct vectors of logical operators and use them to take subsets of vectors. For example:

```
> (logic1 = c(TRUE, FALSE, TRUE, FALSE))

[1]  TRUE FALSE  TRUE FALSE

> (vec1 = seq(1, 4))

[1] 1 2 3 4

> vec1[logic1]

[1] 1 3

> vec2

[1]  0.0  2.5  5.0  7.5 10.0

> vec2 > 3 & vec2 < 9

[1] FALSE FALSE  TRUE  TRUE FALSE

> # So...
> vec2[vec2 > 3 & vec2 < 9]

[1] 5.0 7.5
```

So you can see in the final line that R has applied whether or not the numbers are between three and nine, to the vector, to then return the actual numbers that are.

Using relational operators allows us to extract subsets of data very easily. Consider the movie budgets:

Remember from Section 2.3 that to load the movie budgets, use the following commands:
`library(mas1343);`
`data(Budget);`
`data(Length)`.

```
> length(Budget)

[1] 4847
```

To select movies where the budget is known, we use the following command:

```
> non_zero_b = Budget[Budget != -1]
> length(non_zero_b)

[1] 1785
```

Remember, for unknown budgets the data set has stored '-1', so the R code above is selecting known budgets from the budget vector by selecting all those that aren't '-1'. The last line shows the length of this non zero budget vector, and you can see a substantial amount have been removed.

In the same way, we can select movies where the movie length is greater than 60 mins but shorter than 90 mins.

```
> m_l = Length[Length > 60 & Length < 90]
```

2.8 Data frames

A data frame is a special kind of object. We use data frames for storing and managing data sets that have a rectangular structure. Typically the rows correspond to *cases* and the columns to *variables*. The crucial difference between a data frame and a `matrix` is that all values in a matrix must be of the same type. The next code segment constructs a simple data frame.

First, we construct three vectors:

```
> age = c(24, 26, 25, 21)
> sex = c("Male", "Female", "Male", "Female")
> respond = c(TRUE, FALSE, FALSE, FALSE)
```

Then we combine them using the `data.frame` function:

```
> (df1 = data.frame(age=age, gender=sex, respond=respond))

  age gender respond
1  24   Male    TRUE
2  26 Female   FALSE
3  25   Male   FALSE
4  21 Female   FALSE
```

The data frame we have named `df1` has three columns and four rows. Therefore this represents, say, four individuals. Each of the four individuals has their information stored with their ages, genders and response in each of the corresponding columns.

The use of a data frame is useful in this case because a matrix would not be able to store both **doubles** (age) and **characters** (gender). Once we put our data into a data frame, then data manipulation is easier. To calculate the dimensions of a data frame we use `dim`:

```
> #Dimensions of the data frame
> dim(df1)#This returns a vector

[1] 4 3
```

To extract the first column we use square brackets, say if we wanted all the ‘ages’:

```
> #Extract the first column
> df1[ ,1] #Another vector

[1] 24 26 25 21
```

Similarly, we can get the first row, say if we wanted all of one individual's information:

```
> df1[1, ] #Extract the first row

  age gender respond
1  24   Male    TRUE
```

The column names are also easily manipulated:

```
> colnames(df1) #A vector of characters

[1] "age"     "gender"  "respond"

> #We can easily change the column names
> (colnames(df1) = c("Age", "Sex", "Respond"))

[1] "Age"     "Sex"     "Respond"
```

When we download the `movies` data set, we automatically create a data frame:

See section 1.5 and section 2.3

```
> #If you have started a new session since last time remember to type:
> library(mas1343)
> data(movies)
> #Ask R if we have a dataframe
> is.data.frame(movies)

[1] TRUE

> #View it's dimensions
> dim(movies)

[1] 4847   24

> colnames(movies)[1:4]

[1] "Title"  "Year"   "Length" "Budget"

> #Extract the 4061st row and columns 1 to 4
> movies[4061, 1:4]

                                         Title Year Length   Budget
4061 Star Wars: Episode I - The Phantom Menace 1999    133 1.15e+08
```

We can see above that we have selected the 4061st movie and the first four columns of its information.

2.8.1 Subsets of data frames

We can also retrieve subsets from the data frame. For example, if we wanted only female responses, then:

```
> (female_only = df1$Sex=="Female")
```

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

```
> (df2 = df1[female_only, ])
```

```
  Age    Sex Respond
2  26 Female   FALSE
4  21 Female   FALSE
```

The $ symbol is used to ask R about a part of a data frame, so the first line asks whether responses in the Sex column are ”Female”, thus returning TRUE/FALSE statements. We then define our new data frame as being the responses from df1 that had TRUE responses.

We can only use the $ symbol if the dataframe has column headings

Note the text is before the colon in the square brackets, as we are dealing with the columns.

In the same way we can do this to retrieve subsets of people 25 and over:

```
> #Greater than or equal to 25
> over_25 = df1$Age>=25
> (df3 = df1[over_25, ])
```

```
  Age    Sex Respond
2  26 Female   FALSE
3  25   Male   FALSE
```

2.8.2 Example: movie data

We can select movies where the budget is greater than $100,000:

```
> # movies$Budget > 100000 is a logical vector
> # Select rows
> m1 = movies[movies$Budget > 100000,]
> dim(m1)
```

```
[1] 1738   24
```

or movies that cost more than $100,000 but are not R rated:

```
> m2 = movies[movies$Budget > 100000
+   & movies$mpaa != "R",]
> dim(m2)
```

```
[1] 727  24
```

In the above code we have shown the dimensions of the new data frame each time, and you can see these decrease as the conditions we want increase.

Using the `&` symbol allows us to select movies that fit both conditions. Similarly, we can select movies that fit either condition or both using `|`, for example, movies that are either PG or PG-13:

```
> m3 = movies[
+   movies$mpaa == "PG" | movies$mpaa == "PG-13",]
> dim(m3)
```

```
[1] 1515   24
```

Conclusion

You should now be able to perform some of the basic commands in R including:

- simple and more complicated calculations;
- assignment operations;
- identifying different data types;
- use of vectors and logical vectors, their operations, extracting elements, relational operators and vector partitions;
- simple manipulation of data frames and their subsets.

Practicals

You should now complete Practical 1 as this covers material up to this point, including more complex questions on vectors and data frames.

3 Data summaries

Objectives

This chapter will be split into two main sections, the first will give a brief overview of how R can be used to find summary statistics using numerical methods. Before you begin this chapter it may be useful for you to remind yourself of how to complete the following calculations by hand or on a scientific calculator:

- sample mean;
- sample median;
- sample mode;
- range;
- sample variance and standard deviation;
- quartiles and the interquartile range.

This chapter will then cover graphical presentation of data. Graphical displays of data can be very useful for showing the main features of a data set. The appropriate form of graph depends on the nature of the variables being displayed and what aspects are to be shown. However it should always be borne in mind that the object is to provide a clear and truthful representation of the data, not to distort and not to impress with unnecessary "fancy" features. There are loads of different types of plots R can do, we are just going to focus on the most commonly-used.

3.1 Numerical Summaries

3.1.1 Sample mean

One of the most important and widely used measures of location is the (arithmetic) mean:

$$\bar{x} = \frac{x_1 + x_2 + \ldots + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i \, .$$

So if our data set was $\{0, 3, 2, 0\}$, then $n = 4$. Hence,

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{0+3+2+0}{4} = 1.25 \, .$$

3.1.2 *Sample median*

The sample median is the middle observation when the data are ranked in ascending order. Denote the ranked observations as $x_{(1)}, x_{(2)}, \ldots, x_{(n)}$. The sample median is defined as:

Remember that $x_{(n+1)/2}$ is the $(n+1)/2^{\text{th}}$ ordered observation.

$$\text{Sample median} = \begin{cases} x_{(n+1)/2}, & n \text{ odd;} \\ \frac{1}{2}x_{(n/2)} + \frac{1}{2}x_{(n/2+1)}, & n \text{ even .} \end{cases}$$

The median is more robust than the sample mean, but has less useful mathematical properties.

FOR OUR simple data set $\{0, 3, 2, 0\}$, to calculate the median we re-order it to: $\{0, 0, 2, 3\}$; then take the average of the middle two observations, to get 1.

3.1.3 *Sample mode*

The mode is the value which occurs with the greatest frequency. It only makes sense to calculate or use it with discrete data. In R we use the `table` function to calculate the mode.

Warning: in R the function `mode` doesn't give you the sample mode. Use `table` instead.

3.1.4 *Range*

The range is easy to calculate. It is simply the largest minus the smallest.

When you get a new data set, calculating the range is useful when checking for obvious data-inputting errors.

$$\text{Range } = x_{(n)} - x_{(1)} \ .$$

So for our data set of $\{0, 3, 2, 0\}$, the range is $3 - 0 = 3$. It is very useful for data checking purposes, but in general it's not very robust.

Obviously, the range can be distorted by outliers or extreme observations.

3.1.5 *Sample variance and standard deviation*

The sample variance, s^2, is defined as

In statistics, the mean and variance are used most often. This is mainly because they have nice mathematical properties, unlike the median, say.

$$\begin{aligned} s^2 &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \\ &= \frac{1}{(n-1)}\left\{\left(\sum_{i=1}^{n} x_i^2\right) - n\bar{x}^2\right\} . \end{aligned}$$

The second formula is easier for calculations. So for our data set, $\{0, 3, 2, 0\}$, we have

The divisor is $n-1$ rather than n in order to correct for the bias which occurs because we are measuring deviations from the sample mean rather than the "true" mean of the population we are sampling from.

$$\sum_{i=1}^{4} x_i^2 = 0^2 + 3^2 + 2^2 + 0^2 = 13.$$

So:

$$s^2 = \frac{1}{n-1}\left\{\left(\sum_{i=1}^{n} x_i^2\right) - n\bar{x}^2\right\} = \frac{1}{3}\left(13 - 4 \times 1.25^2\right) = 2.25.$$

The sample standard deviation, s, is the square root of the sample variance, i.e. for our toy example $s = \sqrt{2.25} = 1.5$.

The standard deviation is preferred as a summary measure as it is in the units of the original data. However, it is often easier from a theoretical perspective to work with variances.

3.1.6 Using R for basic numerical summaries

We can easily use R to calculate summary statistics for any vector. R has lots of built-in ”intrinsic” functions to do basic statistics, some of which will be demonstrated here. Try running the following code yourself in R whilst calculating the summary statistics yourself by hand to check you get the same result.

```
> #creates a vector
> x = c(1,2,3,3,4,5)
> #calculates the mean
> mean(x)

[1] 3

> #calculates the median
> median(x)

[1] 3

> #calculates the standard deviation
> sd(x)

[1] 1.414214

> #calculates the variance
> var(x)

[1] 2
```

3.1.7 Numerical summaries for the movie data set

For the movie data, we can easily use R to calculate the summary statistics.

Section 1.5 and Section 2.3. If you have started a new session since last time remember to type: `library(mas1343);` `data(movies);` `data(Budget).`

To calculate the mean and median, we use the `mean` and `median` functions:

```
> mean(Budget)

[1] 10286893

> median(Budget)

[1] -1
```

Notice that the budget mean and median are substantially different ... why?
We can repeat these calculations to get more sensible results by removing all the movies with unknown budgets.

R uses notation for standard form for very large/small numbers. You can see this in the result for the median.

```
> #Start by creating a vector of the budgets removing `-1'
> non_zero_b=Budget[Budget != -1]
> mean(non_zero_b)

[1] 27933093
```

```
> median(non_zero_b)

[1] 1.6e+07
```

We can also calculate measures of spread:

```
> range(non_zero_b)

[1] 6e+03 2e+08

> var(non_zero_b)

[1] 9.487355e+14

> sd(non_zero_b)

[1] 30801550
```

To get the quartiles from R we use the `quantile` command, i.e.

Note the default in R gives you a slightly different quartile range, i.e. if you don't enter the `type=6` argument. As $n \to \infty$, the different quartile functions converge. The "type=6" argument gives the quartiles that are used in most Statistics modules

```
> quantile(movies$Rating, type=6)

  0%  25%  50%  75% 100%
 1.0  4.6  5.7  6.6  9.1

> summary(movies$Rating, type=6)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   4.600   5.700   5.523   6.600   9.100

> #Can see the IQR for movie ratings is 6.6-4.6 = 2.
```

Command	Comment	Example
`mean`	Calculates the mean of a vector	`mean(x)`
`median`	Calculates the median of a vector	`median(x)`
`sd`	Calculates the standard deviation of a vector	`sd(x)`
`var`	Calculates the variance of a vector	`var(x)`
`quantile`	The vector quartiles. Make sure you use `type=6`	`quantile(x, type=6)`
`range`	Calculates the vector range	`range(x)`
`summary`	Calculates the quartiles **However,** it doesn't use `type=6` quartiles. **BUT** you can use `summary(x, type=6)`	`summary(x)`

Table 3.1: Summary of R commands so far this chapter.

Computer Based Assessments

You should now take the opportunity to complete the Computer Based Assessment (CBA 3) which will cover the material you have read so far. We would advise not continuing with the material until you are confident with all the questions covered.

3.2 Graphical summaries

3.2.1 Qualitative data: bar charts

The most useful way to display qualitative data is usually with a bar chart. The length of each bar is proportional to the frequency of the corresponding value of the variable in the sample of data. Note that the widths of the bars should be equal to avoid giving a false impression, as should the width *between* bars.

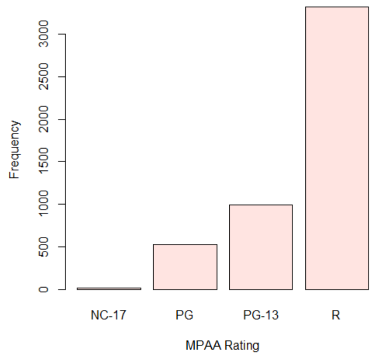


Figure 3.1: Barchart of the MPAA ratings for 4847 films

To create Figure 3.1 in R we use the `table` command...

```
> table(movies$mpaa)

NC-17    PG PG-13     R
   16   526   989  3316
```

... inside the `barplot` function:

```
> barplot(table(movies$mpaa), xlab="MPAA Rating",
+         ylab="Frequency", border = "black",
+         col="mistyrose")
```

In the above code we have set the colours of the bars and borders, you can do these whichever colours you like.

Type `colours()` into R to get a list of colours.

You can change the x-axis label and the y-axis label by using the codes `xlab=""` and `ylab=""` respectively, including the label you want between the speech marks.

It is important that you always include the speech marks. You can also include an appropriate title using the code `main=""`, with the title in between the speech marks.

Remember to load the data first!

AND ALWAYS LABEL YOUR AXES!

3.2.2 Histograms

To represent the distribution of a sample of values from a continuous variable we can use a histogram. The range of values of the variable is divided into intervals, known as *classes* or *bins*, and the frequencies in classes are represented by columns. As the variable is continuous, there are no gaps between neighbouring columns, unlike a bar chart. Note also that, strictly speaking, it is the *area* of the column which is proportional to the frequency, not the height. The reason for this is that columns need not be of the same width. Most computer packages, and reports/publications using histograms, tend to use columns of the same width. However, this default can be overridden in R if you really want to do this.

Unless, of course, a particular class has zero frequency

When dealing with densities (relative frequency), we can easily work out the height using this formula:

$$\text{Height} = \frac{\text{frequency}}{n \times \text{ Bin-width}} .$$

When the y-axis is labelled with density or relative frequencies, the area under the histogram is one. Bin widths should be chosen so that you get a good idea of the distribution of the data, without being swamped by random variation.

There are various different ways of ”defining” the histogram, according to number of bars or the width of the bars. The default is usually good enough, although this can be changed. For more information see the online book:

`https://cran.r-project.org/doc/manuals/R-intro.pdf`.

To generate Figure 3.2 in R we use the following commands:

```
> hist(movies$Budget, col="grey",
+       main="Mean film budget",
+       freq=FALSE, xlab="Budget ($)")
```

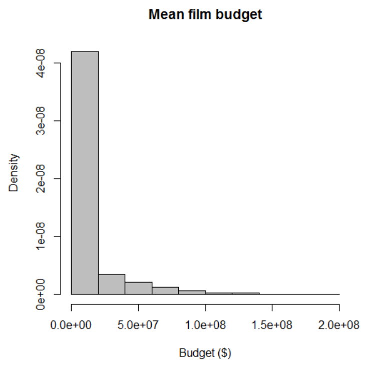


Figure 3.2: Histogram of movie budgets.

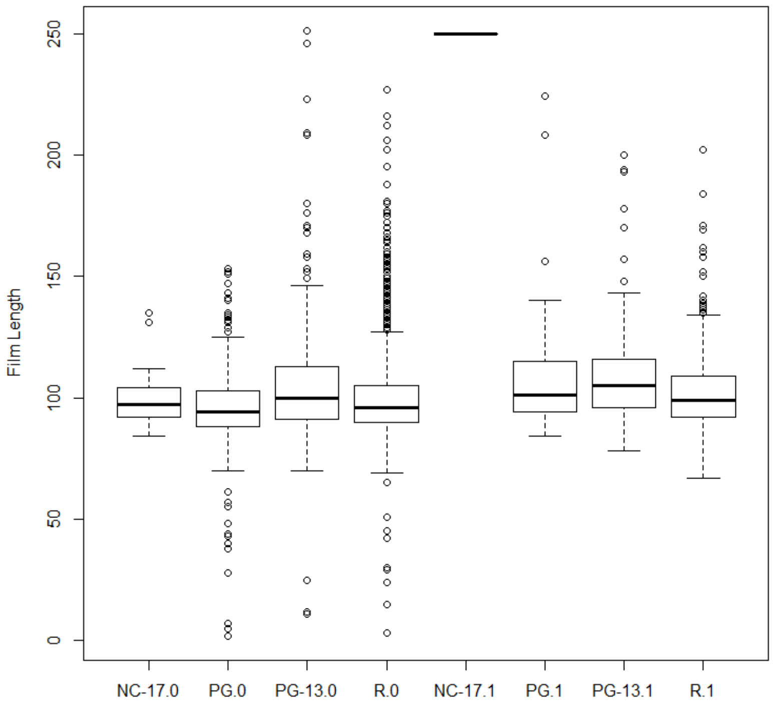


Figure 3.3: Box and whisker plots movie length split according to mpaa rating and whether the film was a romance.

3.2.3 Box and whisker plots

A box and whisker plot, often referred to simply as a boxplot, is another way to represent continuous data. This kind of plot is particularly useful for comparing two or more groups, by placing the boxplots side-by-side. Figure 3.3 and figure 3.4 shows boxplots of film length for different categories of film.

The central bar in the "box" is the sample *median*. The top and bottom of the box represent the upper and lower sample *quartiles*, respectively. Just as the median represents the 50% point of the data, the lower and upper quartiles represent the 25% and 75% points respectively.

The lower whisker is drawn from the lower end of the box to the smallest value that is no smaller than $1.5IQR$ below the lower quartile. Similarly, the upper whisker is drawn from the middle of the upper end of the box to the largest value that is no larger than $1.5IQR$ above the upper quantile. Points outside the whiskers are classified as outliers.

To do this in R we use the following commands:

```
> #Figures 3.4
> par(mfrow=c(2, 1))
> boxplot(movies$Length, ylab="Film length",
+         col="bisque")
> boxplot(movies$Length ~ movies$mpaa, ylab="Film length",
+          col="bisque")
> #Figure 3.3
> boxplot(movies$Length ~ movies$mpaa + movies$Romance,
+          ylab="Film length")
```

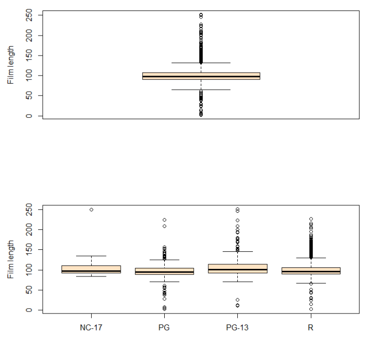


Figure 3.4: Box and whisker plots of (a) film length (b) film length split according to the mpaa rating.

The first line is used to plot more than one plot at the same time. So the first number represents how many rows of plots and the second number how many columns. Thus creating Figure 3.4 as one image involving two plots. This is useful when you are wanting to compare several plots.

Practicals

You should now complete Practical 2 which involves plotting and representing data from The TV show "The Big Bang Theory."

Conclusion

You should now easily be able to use R to calculate summary statistics and be able to create graphical presentations of data in R using:

- bar charts;
- histograms;
- box and whisker plots (boxplots).

Command	Comment	Example
`table`	Contingency table	`table(x)`
`barplot`	Generate a bar chart	`barplot(table(x))`
`hist`	Histogram	`hist`
`plot`	Scatter plot. See Practical 2.	`plot(x, y)`
`points`	Add points to a plot. See Practical 2.	`points(x,y)`
`lines`	Add lines to a plot. See Practical 2.	`lines(x,y)`
`boxplot`	Box and whiskers plot	`boxplot(x)`

Table 3.2: Summary of R commands for the remainder of this chapter.

4
Control Statements and Functions

Objectives

This section will discuss writing your own functions and using control statements.

4.1 Functions

A very powerful aspect of R is that it is relatively easy to write your own functions. Functions can take inputs (or *arguments*) and return a *single* value. However, this single value could be a list, vector or matrix, for example, containing many values. Let's look at some simple functions.

4.1.1 Basic functions

This function takes in a single argument x and returns x^2:

```
> Fun1 = function(x) {
+   return (x*x)
+ }
```

The key elements in the function call are:

- The word `function`;
- The brackets () which enclose the **argument** list.
- A sequence of statements in curly braces { }.
- A `return` statement.

The above code transforms the argument (in this case x) by following the statements in the curly braces and R gives back the argument in the return statement.

Once we have defined our function, calling it what we like, we call it in the following manner:

To ”call” a function means you ”run” the function or perhaps ”invoke” the function.

```
> Fun1(5)

[1] 25

> y = Fun1(10)
> y

[1] 100

> #We can also pass a vector
> z = c(1, 2, 3, 4)
> Fun1(z)

[1]  1  4  9 16
```

You can see in the above code that this also works for vectors.
Of course, the old saying ‘garbage in, garbage out’ is true:

```
  > Fun1()
  Error in Fun1() : argument "x" is missing, with no default
  > Fun1("5")
  Error in x * x : non-numeric argument to binary operator
```

The error messages give you an idea of what went wrong. Other variations to this simple function are:

```
> #Default argument
> Fun2 = function(x=1) {
+   return (x*x)
+ }
> Fun2()

[1] 1

> Fun2(4)

[1] 16

> #We can have multiple arguments
> Fun3 = function(x, y) {
+   return (x*y)
+ }
> Fun3(3, 4)

[1] 12
```

4.1.2 A more useful function

Here the function below takes in a vector, plots a histogram and returns a vector containing the mean and standard deviation:

```
> Investigate = function(values) {
+   hist(values)
+   m_std = c(mean(values), sd(values))
+   return(m_std)
+ }
```

Once we have created our function, we can put it to good use:

Obviously, a histogram would also be created – it's just not shown here. See Section 3.2.2 for examples of histograms.

```
> #Call the function (plot not shown)
> Investigate(movies$Rating)

[1] 5.522715 1.451864
```

4.1.3 Variable scope

When we call a function, R first looks for *local* variables (inside function), then *global* variables (outside funtion). For example, `Fun4` uses a global variable:

R scoping rules are actually a bit more complicated than described below. R uses something called *lexical* scope, but this doesn't affect us.

```
> blob = 5
> Fun4 = function() {
+   return(blob)
+ }
> #Function uses the global blob
> Fun4()

[1] 5
```

However, in `Fun5`, we use a local variable:

```
> #Here we use a local variable
> Fun5 = function() {
+   blob = 6
+   return(blob)
+ }
> Fun5()

[1] 6

> #Local doesn't affect global!
> blob

[1] 5
```

As a general rule, functions should only use **local** variables. This makes your code more portable and less likely to have bugs.

4.2 Conditionals

Conditional statements are features of a programming language which perform different computations or actions depending on whether a condition evaluates to `TRUE` or `FALSE`. They are used in almost all computer programs.

4.2.1 If statements

The basic structure of an `if` statement is:

```
> if(expr) {
+   #do something
+ }
```

where `expr` is evaluated to be either `TRUE` or `FALSE`. The following example illustrates `if` statements in R:

```
> x = 5; y = 5
> if(x<5) {
+   y = 0
+ }
> y

[1] 5
```

In this code chunk, `x < 5` evaluates to be `FALSE` so the following brackets are not evaluated. We test for greater than in a similar manner:

```
> x = 5; y = 5
> if(x > 0) {
+   y = 0
+ }
> y

[1] 0
```

Here `x > 0` evaluates to be `TRUE` so, `y` is set equal to 0. If we wanted to test for equality with zero, then we would use `==`.

We can also use `if` statements in functions, for example to check that our data is negative we can construct the following function:

```
> IsNegative = function(value) {
+   I = FALSE
+   if(value < 0) {
+     I = TRUE
+   }
+   return(I)
+ }
> IsNegative(1)

[1] FALSE
```

```
> IsNegative(-5.6)

[1] TRUE
```

A more sophisticated function could be:

```
> IsGreaterThan = function(value1, value2) {
+   is_greater_than = FALSE
+   if(value1 > value2) {
+     is_greater_than = TRUE
+   }
+   return(is_greater_than)
+ }
```

Which we can then call:

```
> IsGreaterThan(-5, -6)

[1] TRUE

> IsGreaterThan(10, 10)

[1] FALSE
```

4.3 Control statements

At times we would like to perform some operation on a vector or a data frame. Often R has built-in functions that will do this for you, e.g. `mean`, `sd`,... Other times we have to write our own functions. For example, suppose we want to calculate $\sum_{i=1}^{10} i^2$.
In R we can use a `for` loop:

```
> x = 0
> for(i in 1:10) {
+   x = x + i^2
+ }
> x

[1] 385
```

So in the above code R is 'looping' from $i = 1$ to $i = 10$, and after each loop you get a value of x that is then added onto the next, consequentially summing all the values of i.

Another example is $\sum_{j=-5}^{-1} e^j / j^2$. This can be done in R in the following way:

```
> total = 0
> for(j in -5:-1) {
+   total = total + exp(j)/j^2
+ }
> total

[1] 0.4086594
```

A more tricky example: Calculate $\sum e^k/k^2$, for $k = 3, 6, 9, \ldots, 21$:

```
> total = 0
> for(i in 1:7) {
+   k = i*3
+   total = total + exp(k)/k^2
+ }
> total
```

```
[1] 3208939
```

Exercise: Using the inbuilt R function `sum`, calculate the above summations without using `for` loops.

4.4 *Putting it all together*

Rather than have to constantly write R code to solve the summations in §4.3 we can create a function to solve the general form:

$$\sum_{i=i_s}^{i_e} \frac{e^i}{i^2} \quad \text{for } i = i_s,\ i_s + j,\ i_s + 2j,\ \ldots, i_e\ .$$

So in R we have:

```
> Summation1 = function(i_s, i_e, j) {
+   total = 0
+   for(i in 1:(i_e/j)) {
+     k = i*j
+     total = total + exp(k)/k^2
+   }
+   return(total)
+ }
> #Summation1(8, 14, 2)
> Summation1(3, 21, 3)
```

```
[1] 3208939
```

4.5 *The apply family*

R has been designed with manipulating data in mind. Due to this, there are two important functions that are unique to R.

Probably not unique, but not common in other programming languages.

4.5.1 *The `apply` function*

We use the `apply` function when we want to *apply* the same function to every row or column of a data frame. For example, suppose we have a data frame with three columns:

```
> (df4 = data.frame(c1 = 1:4, c2 = 4:7, c3 = 2:5))
```

```
  c1 c2 c3
1  1  4  2
2  2  5  3
3  3  6  4
4  4  7  5
```

The `apply` function takes (at least) three arguments. The first argument is the data frame, the second the number `1` or `2` indicating row or column and the third a function to apply to each row or column. So:

```
> apply(df4, 1, mean)
```

```
[1] 2.333333 3.333333 4.333333 5.333333
```

calculates the mean value of every row, while:

```
> apply(df4, 2, sd) #sd of the columns
```

```
      c1       c2       c3 
1.290994 1.290994 1.290994 
```

calculates the standard deviation of every column.
Suppose one of the columns was non-numeric:

```
> (df5 = data.frame(c1 = 1:3, c2 = 4:6,
+    c3 = LETTERS[1:3]))
```

```
  c1 c2 c3
1  1  4  A
2  2  5  B
3  3  6  C
```

then taking the mean doesn't really make sense:

```
> apply(df5, 1, mean)
```

```
[1] NA NA NA
```

Instead, we remove the column, then calculate the mean:

```
> apply(df5[ ,1:2], 1, mean)
```

```
[1] 2.5 3.5 4.5
```

4.5.2 The `tapply` function

The function `tapply` is very useful, but at first glance can be tricky to understand. It's best described using an example:

```
> #The first two arguments are vectors
> tapply(movies$Length, movies$mpaa, mean)
```

```
    NC-17        PG     PG-13         R 
110.18750  97.23384 104.97877 100.18818 
```

In the above code, we have calculated the average movie length conditional on its MPAA rating. So the average length of a PG movie is 97 minutes and the average NC-17 movie length is 110mins.

With `tapply` we can do we very interesting things. For example, in the next piece of code, we plot the average movie length conditional on it's rating:

```
> tapply(movies$Length, movies$Rating, mean)[1:6]

   1  1.2  1.3  1.4  1.5  1.6
85.5 93.0 87.5 85.0 67.0 86.0

> rating_by_len = tapply(movies$Length, movies$Rating,
+    mean)
> plot(names(rating_by_len), rating_by_len)
```

Imagine trying to produce Figure 4.1 in Excel!

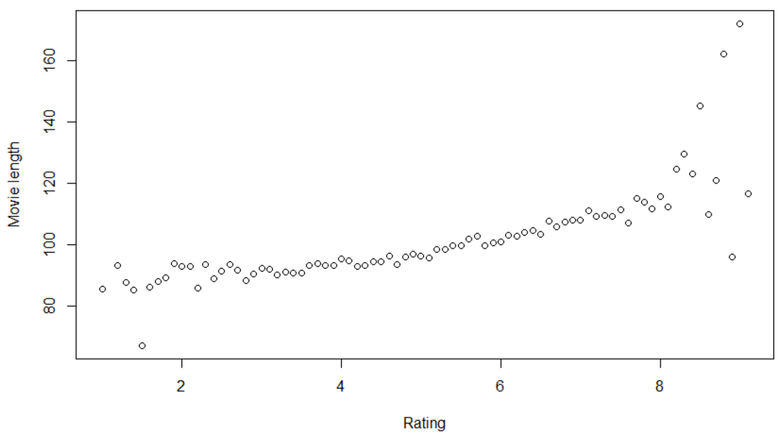


Figure 4.1: Plot of mean movie length conditional on its rating.

4.6 Help

R has a very good help system. If you need information about a particular function – say `plot` – then typing `?plot` in a R terminal will bring up the associated help page.

The internet is another very good source of R help. Unfortunately, using Google isn't particularly useful since the letter "R" appears on most web pages! However, you can use

`http://www.rseek.org/`

Using this search engine limits searches to R web-pages.

Practicals

You should now complete Practical 3 which involves questions on Functions and Loops.

Conclusion: Summary of R commands

You should now know the meaning of, and be able to use, all the functions in Table 5.1.

Command	Comment
`for`	A for loop. See §4.3
`if` or `else`	A conditional statement. See §4.2.
`function`	An R function constructor. See §4.1
`apply` or `tapply`	See §4.5

Table 4.1: Summary of R commands in this chapter.

5
Random Number Generation

Objectives

This chapter will focus on random number generation in R.

5.1 Randomness: quantifying uncertainty

The concepts of uncertainty and randomness have intrigued humanity for a long time. The world around us is not *deterministic* and we are faced continually with chance occurrences. Uncertainty is inherent in nature: for example, the behaviour of fundamental physical particles, genes and chromosomes in biology, and individuals in society under stress or strain. The methodology for exploring uncertainty involves the use of *random numbers*.

5.2 Pseudo–random numbers

Suppose we need to obtain a list of random digits `0,1,2,...,9`. we use Pseudo-random numbers generators (RNG), that is, algorithms for generating sequences of numbers that approximate the properties of true random numbers.

5.3 Random numbers in R

5.3.1 The `runif` function

R has a number of commands that generate and manipulate random numbers. One function that we will make use of here is:

The default random number generator used by R is the Mersenne-Twister.

```
> runif(n, min=0, max=1)
```

The function is a blend with ‘r’ for random and ‘unif’ for uniform.

This essentially generates random numbers from a continuous Uniform distribution - something you will see in your Stage 2 probability/statistics modules. This is what most calculators do when we press the random number button!

This function will generate `n` random numbers between the values of `min` and `max`. If the arguments `min` or `max` are omitted, then the default values are 0 and 1 respectively. For example:

```
> runif(1)
```

```
[1] 0.1309786
```

```
> runif(1)

[1] 0.862699

> runif(5)

[1] 0.3791398 0.2896961 0.8604805 0.5334447 0.3611861

> runif(1, 6, 7)

[1] 6.021599
```

Notice that calling `runif` returns different values. If we wish to rerun a computer experiment, we may need repeatability . . .
In this case we use the command `set.seed`, e.g.

The function `set.seed` sets the seeds of all possible underlying random number generators. This function is actually an interface to `.Random.seed`. Don't worry about this.

```
> set.seed(12345)
> runif(1)

[1] 0.7209039

> runif(1)

[1] 0.8757732

> set.seed(12345)
> runif(1)

[1] 0.7209039
```

You can see that the `set.seed` function above allows R to remember the experiment so that it can be rerun.

If we want to generate integers, say 0, 1, . . ., 9, then we could simply take the first value after the decimal place.

5.3.2 *The `sample` function*

Another important R function that we will use is the `sample` function:

```
> sample(x, size, replace = FALSE, prob = NULL)
```

This takes the following arguments:

- `x`: a list of values, usually a vector.
- `size`: non-negative integer giving the number of items to choose.
- `replace`: Should sampling be with replacement? Default: `FALSE`.
- `prob`: A vector of probability weights. Default: All values equally likely.

See `?sample` for help.

Example usage of `sample`

Suppose we wish to sample five numbers from $\{1, 2, 3, 4, 5, 6\}$, then:

```
> set.seed(1)
> x = c(1, 2, 3, 4, 5, 6)
> sample(x, 5)

[1] 2 6 3 4 1
```

We can also sample with replacement:

```
> sample(x, 5, replace=TRUE)

[1] 6 6 4 4 1
```

This means that values **may** appear more than once.

5.3.3 Simulating the Capital One Cup draw

We are in the semi-finals of the Capital One cup, and need to organise the draw for the final stage. The remaining teams are:

Manchester Utd, Manchester City, Sunderland, West Ham.

Here's how we do this in R.

```
> set.seed(3)
> teams = c("Man Utd", "Man City", "Sunderland", "West Ham")
> sample(teams, 4)

[1] "Man Utd"    "Sunderland" "West Ham"   "Man City"
```

So, we have 'Man Utd vs Sunderland' and 'Man City vs West Ham' [1]. However, if we think Sunderland are likely to get beaten by Manchester United, we can rig the sampling mechanism:

[1] As actually happened in 2014.

```
> prob_weights = c(0.4, 0.4, 0.05, 0.2)
> sample(teams, 4, prob=prob_weights)

[1] "Man Utd"    "Man City"   "West Ham"   "Sunderland"
```

That's better! The two Manchester teams had higher probability weights therefore they were more likely to get chosen first and be against each other.

Conclusion: Summary of R commands

You should be able to understand and use all the commands in Table 5.1.

Command	Comment	Example
`sample`	Sample discrete numbers	`sample(c(1,2,3))`
`runif`	Generate a random number between 0 & 1	`runif(1)`
`set.seed`	Set the seed of the random number generator	`set.seed(10)`

Table 5.1: Summary of R commands in this chapter.

YOU SHOULD NOW FEEL PREPARED for most of the R you will see in your Stage 2 statistics courses. How confident you are depends on how well you have worked through this booklet and the tasks on the webpage. Remember the more you practice the better you will be and there is plenty of material you can attempt! If you do have any problems then do not hesitate to email any of the following lecturers:

- Lee Fawcett at *lee.fawcett@newcastle.ac.uk*;
- Christian Lawson-Perfect at *christian.perfect@newcastle.ac.uk*;
- Christopher Graham at *Christopher.Graham@newcastle.ac.uk.*

Computer Based Assessments

You should now take the opportunity to complete the Computer Based Assessment (CBA 4) which will cover the material you have read so far. We would advise not continuing with the material until you are confident with all the questions covered.