STAT2008/6038 Regression Modelling Worksheet 2

1 Introduction

Before attempting this worksheet you should have completed *Worksheet 1*

1.1 Getting into R

As described in *Worksheet 1*, you should log in to the PC using your University ID and password and begin R by Start → All Programs → R → R 2.13.0
Note that if you get into difficulties with any R command and wish to cancel it after you have pressed the return key, hit the *Esc* key on the keyboard to interrupt R. Also, clicking the red "STOP" button in the top GUI panel will do the same.

1.2 Arithmetic Operators in R

Now that you are in R, the first thing you can do is to use R as a basic calculator. R will perform all the usual arithmetic operations on numbers:

+	addition
−	subtraction
∗	multiplication
/	division
^	exponentiation (i.e. raising to power)

To make sure you know how to apply these operators (and to check that R will give you the expected results), try doing a few simple calculations such as adding, subtracting, multiplying and so on with some chosen pair of numbers. For example try finding 3 cubed (i.e. 3 to the power 3). R also has the following comparison and logical operators:

==	is equal to
! =	is not equal to
<	is less than

<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
&&	and
||	or
!	not

These operators are primarily used in identifying subsets of data objects and in writing branching statements (such as *if ... then ...* statements, etc.) which you will need when writing your own functions (which we will discuss in more detail in Worksheet 3). Try typing in the following expressions and interpreting the results:

$3 < 5$

$3 > 5$

$3 > 5 \,||\, 3 < 5$

There are also a number of simple R univariate functions which will work using "ordinary" numbers as arguments:

`abs()`	the absolute value of the argument
`floor()`	the integer part of the argument
`sqrt()`	the square root of the argument

`exp(), log(), log10()`	exponential and log functions
`sin(), cos(), tan()`	trigonometric functions
`asin(), acos(), atan()`	inverse trigonometric functions
`sinh(), cosh(), tanh()`	hyperbolic functions
`asinh(), acosh(), atanh()`	inverse hyperbolic functions

Try applying these functions, e.g. test the square root function by typing: `sqrt(4)`

1.3 Storing Objects in R

So far, all we have been doing in R is evaluating expressions. We can also create and store the results of these evaluations as permanent objects by using the assignment function, $<-$. An example of how this function is used is:

`c` $<- 5$

The assignment function tells *R* to first evaluate the right-hand side (which in this case isn't much of an evaluation) and assign its value to the object named on the left side. Note that *R* doesn't seem to do anything in response to the above command. What it has done is create a new permanent object called `c` which has the value 5. To see the contents of any object such as `c`, simply type the name of the object. Arithmetic operations can be performed on this object as we did for numbers. For example, try typing the following commands:

```
cˆ3
c*3
```

R will not automatically save any objects you create for use in future sessions. However, they will be available to use in the current session. If you want to continue your current work in a future session, the easiest way is to save the codes you have written either in the form of a text file or a *R* script file and run them before continuing in all future sessions. In a current *R* session, all the objects created can be viewed by either of the following commands:

```
objects()
ls()
```

The object called `c` which we created earlier in this session is probably the only object currently in your area. This object will remain in your area until you remove it or you end the session. (To remove an object we use the `rm()` command.Your workspace, remember, consists of all the data objects you've created or loaded during your R session. When R asks if you want to save your workspace image before quitting, and you say "no", all of the new objects you've created goes away. If you say "yes", then a file called ".RData" is written to your working directory. The next time you start R in this directory, that workspace and all its data objects will be restored.

There are a number of other areas which are accessible from your account, such as packages, functions and datasets of *R* (see *An Introduction to R* available at http://cran.r-project.org/doc/manuals/R-intro.html for more information).

Earlier we saw that the `objects()` command showed that the only object in this session is the object `c`. *R* stores both both data and functions as objects. This is why it is essential to type the brackets when you wish to use a function.

We removed the object c previously and will need to use it again below, so first type:

```
c <− 5
```

To create a new function for finding the square of a number type in the following:

```
square <- function(x) { x^2}
```

There will be more on writing your own R functions in Worksheet 3. Typing `ls()` should now show the two objects you have created this session, i.e. `c` and `square`. To see what an object contains, just type the name of that object. So, typing `square` without any brackets will show you the contents of this new function, whilst typing `square(c)` will calculate the value of `c` squared. Try typing `square()`, without any number or argument, and see what happens. The error message occurs because typing a function name followed by parentheses indicates to R that you wish to calculate something using the function (as opposed to seeing its contents), and the `square()` function requires an input value in order to perform its calculations. New R objects can be called anything you like, which can lead to problems if you use the names of common R functions to store data items. `c()` and `t()` are the names of two very frequently used R functions and using these names for your own objects can lead to problems as we will see below. Stored R objects can be deleted using the `rm()` function. It is generally advisable to use names with more than one character to name new objects in order to avoid overwriting any pre-defined function.

1.4 Types of Data Objects in R

The simplest type of stored object in R is a single number, i.e. a scalar. We have already seen an example of a stored scalar; namely, the object we created called `c`. We can change the value of this stored object by simply reassigning it, i.e.:

```
c <- 4
```

The next simplest stored data object is a vector of numbers (a scalar can be regarded as simply a vector with only one component). A number of functions can be used to create special vectors. The functions `seq` and `rep` are useful to create vectors containing sequences of numbers or repeated pattern of numbers. Experiment with the functions `seq()` and `rep()`. Examine the help files on these functions - use `help(seq)` and `help(rep)`. One of the most versatile methods of creating vectors is to use the `c()` function, which can be used to combine smaller vectors and/or single numbers into larger vectors. The `c()` function also provides a very simple way of entering small amounts of data into R. Use the `c()` function to enter the following vectors:

```
x <- c(1, 0, 0)
y <- c(1, 2, 0)
z <- c(1, 2, 5)
```

A vector is a one-dimensional array of numbers. *R* can also handle two-dimensional arrays (matrices) and even more general arrays of higher dimensions (simply called arrays in *R*).
Vectors can be combined to form a matrix (and matrices can be combined to form multi-dimensional arrays) in a number of ways. Two of the easiest are the `cbind()` function and the `rbind()` function. Compare the results of the following commands:

```
cbind(x, y, z)
rbind(x, y, z)
```

Store the results of the second command as a new object, as follows:

```
mymat <- rbind(x, y, z)
```

Note that the `cbind()` version can be extracted from this new object by using the `t()` function which creates the transpose of a matrix:

```
t(mymat)
```

Other functions which can be applied to a vector, say `x`, include:

`length(x)`	number of elements in the argument `x`
`sort(x)`	sorts the elements of `x` into ascending order
`order(x)`	gives the ranks of the elements of `x`
`max(x), min(x)`	maximum or minimum of elements of `x`
`median(x)`	median value of the elements of `x`
`sum(x)`	sum of elements in `x`
`prod(x)`	product of elements in `x`
`mean(x)`	mean of all elements in `x`
`mean(x,trim=0.25)`	25% trimmed mean of all elements in `x`
`var(x)`	sample variance-covariance matrix of `x`
`cor(x)`	sample correlation matrix of `x`

Functions which can be applied to a matrix, say `amat`, include:

`ncol(amat)`	number of columns of `amat`
`nrow(amat)`	number of rows of `amat`
`t(amat)`	transpose of `amat`
`solve(amat)`	inverse of `amat`
`apply(), sweep()`	row- or column-wise function application

The `apply()` and `sweep()` functions are particularly useful. The `apply()` function is used to apply the same function to each column or row of a matrix. For example, to calculate the mean of each column of `mymat`, type:

```
apply(mymat, 2, mean)
```

and to calculate the mean of each row of `mymat`, type:

```
apply(mymat, 1, mean)
```

In the above examples, the `mean()` function returns a single value so the result of the `apply()` function is a vector. When the applied function returns a vector, some care is needed. For example, to sort the rows of the matrix `mymat`, use:

```
t(apply(mymat, 1, sort))
```

The transpose is needed here because the first sorted row becomes the first column and so on. However, to sort columns, we can use:

```
apply(mymat, 2, sort)
```

The `sweep()` function subtracts values off from each row or column of a matrix. Thus to get a matrix with mean-centred columns, type:

```
sweep(mymat, 2, apply(mymat, 2, mean))
```

Apart from using functions, we can also perform arithmetic with vectors and matrices. The operators +, − and * will work with vectors and matrices in a component-by-component fashion. Try typing the following commands to see what happens:

`y + z`
`y − z`
`y * z`

The matrix multiplication operator is %*% and it works in the expected fashion. As an example, type the following command, but try to work out what the result should be, before you press the return button:

`mymat` %*% `solve(mymat)`

Other operations with vectors and matrices are also possible, e.g. the outer product of two vectors. Refer to the help files on the `outer()` function or try the %o% operator on the vectors `y` and `z`.

We can also refer to and manipulate parts of data objects separately. For instance, suppose that we had made a mistake in entering the third element of the vector `z`, above, and it should have been a 3, not a 5. We can correct the vector `z` as follows:

```
z[3] <- 3
```

However, we also used the vector `z` when we created the matrix `mymat`, and we can also correct that mistake using:

```
mymat[3, 3] <- 3
```

In this discussion we have only looked at numeric data objects, however R can also store data in other modes, e.g. complex numbers, logical values and characters. For more information examine the help files on attributes and mode. There are two more complicated ways of storing data in R: lists, which allow a number of different types of objects to be grouped together; and dataframes which combine some of the features of both matrices and lists.

1.5 Entering Data in R

Data is most easily entered in R by importing it from an external file. This external file are most commonly either text (`.txt`) or comma separated files (`.csv`). For the purpose of this course we will focus on importing data from `csv` files.
The command to import data from a csv file in R is `read.csv()`. For example, consider the `worksheet2_women.csv` dataset available on wattle. Save the file to your working directory. In order to import the dataset in R we use the following commands:

```
women<-read.csv("worksheet2_women.csv",header=F)
```

The contents of this dataset can now be accessed by typing the name of the object: `women`. The `str()` function provides a compact display of the internal structure of the dataset and is an alternative to the `summary()` function. Let us display the structure of the women dataset:

```
str(women)
```

Note that the columns do not have names associated with them. The first column refers to the *height* and the second refers to the *weight* of the sampled women. In order to name the columns appropriately, we use the following commands:

```
colnames(women)
```

```
names(women)<-c("Height","Weight")
```

We can extract the heights and weights as separate vectors from the matrix by using appropriate "square bracket" notation:

```
women[,1]
women[,2]
```

Here, the initial blank preceding the comma indicates that the entire column is being referenced rather than any specifically designated row as was the case when we editted the matrix `mymat` previously. Similarly, we could indicate an entire row of a matrix by using a trailing blank following the comma. Another, more commonly used method to recall components of a data frame (all data sets imported using `csv` files are dataframes by default), is to treat the columns as objects themselves. This is done by using the `attach()` function:

```
attach(women)
```

Having attached the dataframe, we can then refer to the named components as if they were separate R objects, i.e. simply type:

```
Weight
Height
```

Note that object names in R are case-sensitive. If we typed `height` instead of `Height`, the R console would give us an error message!

1.6 Graphics in R

The graphics capability of R is one of its powerful features. A full description of the possibilities is way beyond the scope of the present worksheet so we will concentrate on some simple, commonly used graphical procedures. For information on how to set graphical parameters for these procedures, use `help(par)` and also see the help files for each of the graphical functions listed below. When a graphics command is issued, R automatically opens a graphics window. One very rudimentary graphic which doesn't need a graphics device is a stem-and-leaf plot. To see stem-and-leaf plots for the women's data, type:

```
stem(Height)
stem(Weight)
```

noindent To plot histograms of the women's data type:

```
hist(Height)
hist(Weight)
```

To obtain side-by-side boxplots the women's heights and weights, type:

```
boxplot(Height,Weight)
```

To get a scatterplot of `Weight` versus `Height`, type:

```
plot(Height,Weight)
```

To join the points on the plot, type:

```
plot(Height,Weight,type="l")
```

Other more elaborate plots will probably be introduced later in the course. Note that `plot()` is the first example you have encountered of a generic function, one which examines the object it is given to work on and tries to come up with an appropriate treatment. You will probably encounter more of these generic functions (such as the `summary()` function) later in the course. Control over the plots is usually achieved by specifying specific graphical parameters as additional arguments to the `plot()` function. The most commonly used of these parameters include the labelling arguments:

```
main="...", sub="...", ylab="...", xlab="..."
```

and the plotting control arguments:

`type=` `"l"` to plot lines; `"p"` for points; `"b"` for both

`pch=` A numeric value specifies a specific plotting symbol:

0=square, 1=octagon, 2=triangle,3=cross,

4=X, 5=diamond, 6=inverted triangle, etc.

A character in double-quotes specifies a user-chosen plotting symbol

(e.g. `"o"` will plot a lower-case "o" as the plotting symbol)

`lty=` 1 for a solid line, 2 or higher for varying degrees of dashed or dotted lines

It is also possible to add to a plot once it has been drawn. For example, to add the least-squares regression line to a scatterplot of `Weight` versus `Height`, type:

```
plot(Height,Weight)
abline(lsfit(Height,Weight)$coef)
```

The `abline()` function can be used to include lines of the form $y = a + bx$ on a plot, including horizontal and vertical lines. The argument to `abline` should be a vector whose first component is a, the intercept of the line, and whose second component is b, the slope. The `lsfit()` function requires two main inputs (and also allows various optional input parameters) the first being either a vector or matrix of predictor variable values and the second being a vector of response values.

The `lsfit()` returns a named list containing various relevant information about a least-squares regression of the response variable on the predictor variable(s). As noted above, we can access elements of a list using the $ operator, and in this instance we have chosen the `coef` element of the list which is a vector containing the intercept and slope of the least-squares regression line. Note that the `lm()` function can also be used to perform least-squares regressions, so we could replace the second line above with:

```
abline(lm(Weight ~ Height))
```

Also available are a number of presentation plots (`barplot()`, `pie()`), `pairs()`, `plclust()`, `stars()`), time series plots (`monthplot()`, `ts.plot()`), three dimensional plots (`contour()`, `persp()`) and a whole lot more!.

1.7 Preparing for Worksheet 3

Worksheet 3 assumes that you have covered the materials in the first two worksheets in detail. Do not attempt Worksheet 3 until you are happy that you understand the first two worksheets.