Regression Modelling

(STAT2008/STAT2014/STAT6014)

Tutorial 1 – Introductory to **R**

In this course, we will be using the statistical computing software **R** to perform reasonably complex statistical analysis. Many of you will have already encountered **R** or possibly some other statistical computing software in your earlier prerequisite statistics courses. There are detailed instructions on how to access and/or download **R** on the Wattle site for this course, along with links to a range of introductory **R** material.

If you have never used **R** before, then you have some catching up to do. Start working through some of the introductory **R** material available on Wattle and if you need help getting started, you are welcome to come to our consultation. A good indication of whether or not you have managed to catch up is whether or not you can complete the compulsory (non-optional) exercises in this worksheet by the end of week 2. At that stage, if you are still behind and need additional help, then please make a time to see one of the tutors or me during our consultation times, which are advertised on Wattle.

In the school responsible for this course (RSFAS – the Research School of Finance, Actuarial Studies and Statistics), tutorials start in week 2. This worksheet is designed to be covered in the first tutorial (in week 2) by students who have previously used **R**, but is almost certainly too long to be covered in a single hour, so you may have to put in some additional work in order to complete the compulsory (non-optional) exercises by the end of week 2.

It is also a good idea to attempt the rest of this worksheet prior to your first tutorial and come to the tutorial with some questions, so that your tutor can “fill in the gaps”. The same advice is also good for the later tutorials – the more work you put in before the tutorial, the more you will get out of the tutorial.

Exercise One

On the ANU InfoCommons computers (the machines in the computing labs and libraries), there are a few interfaces available for **R** (the basic **R** GUI or “graphical user interface” for different processors and **RStudio**) or you can download a variety of other interfaces for your own computer. It doesn’t really matter which interface you use to access **R**, as they only really differ in the availability of menu options and associated tools; the underlying **R** commands are typically the same in all of the different versions.

If you want everything to look like the examples that I demonstrate in lectures, then it is best to get used to using **RStudio**. To get started with the following simple exercises, locate the shortcut, app or executable (.exe) file for **RStudio** (or your chosen interface) and open it. Then try the following:

(a) In relatively simple computing terms, **R** is a command language, which means you issue commands by typing them at the command prompt (**>**) in the **R** Console (the window in the lower left hand corner in **RStudio**). Try using **R** as a simple calculator by typing the following commands (press the Enter key after typing each line):

```
2 + 2
2 * 2
sqrt(4)
abs(4)
abs(-4)
```

```
log(4)
log(4, base=2)
4 > 2
4 < 2
4 < 2 || 4 > 2
qnorm(0.025)
qnorm(0.975)
```

Note that if you make a typing mistake, the up and down arrows on your keyboard can be used to recall any previous commands, so that you can correct your typing and try again.

(b) If you are unsure about what any command does, you can consult the help file for that command. The help files appear in the Help part of the output window, which is the window in the lower right hand corner in R studio.

For example, if you type **help(log)** in the Console or type **log** in the Help window search box, the relevant help file will appear in the Help window. What is the default base for logarithms produced using the **log()** function in **R**?

Note that you will quickly find that **R** is indeed "freeware", which has been freely contributed by a number of different contributors (mainly statisticians), and that the range of detail and the use of simple language differs considerably for different help files – some are very helpful and some are almost incomprehensible – correct interpretation takes practice and time. Incorrectly typed commands will produce error and/or warning messages; which, like the help files, can range from the obvious to the incomprehensible and only time and practice will equip you to deal with all the errors that can (and will) occur.

(c) **R** is also an object-oriented language, which means that whilst some commands simply produce output in the console or the output window, other commands will manipulate stored objects and produce no visible output (they will just give you back a blank cursor **>** in the Console, waiting for your next input). Other more complex commands will do both (produce output and change stored objects). In **RStudio**, newly created objects will appear in the Environment window, which is the window in the upper right hand corner.

Before you start to create stored objects, it is a good idea to do a bit of file management. Type **getwd()** (short for "get working directory") to find out where **R** is going to store any objects that you create – if you are working on a InfoCommons machine this will typically be the top level of your own personal H: drive (notice how all the \ characters in the file path have become **/** in **R**'s version).

If you are not happy with simply dumping everything there (and you probably shouldn't be), then go and create a new directory where you do want to store material for this course (with ANU InfoCommons machines, it is better to use a new directory on your networked H: drive, or on a USB stick, but not the C: drive of the machine you are currently working on, as this will be not be accessible on any other computers). In the **R** Console type **setwd("**file path"**)**, to set the new working directory, where file path is the path to your new directory (note that you will have to change all the \ characters in the path name to **/**).

(d) The first time you open **RStudio** there may be a blank script window in the upper right corner. Script windows are used to store **R** commands. If there is not one there, then use the menu option File > New File > R Script to open a new script file.
Now, outside **RStudio**, download a copy of the "Hello World" text file ("helloworld.txt"), from Wattle, to your current working directory. Open this file in **RStudio** (using the menu option File > Open File). This should open a new window of stored **R** commands with just one line – copy and paste this line to your blank script file (note how the formatting changes in **RStudio**, which now realises it is dealing with **R** code, not just plain text).

Now highlight the line of code in the script file and run the code (use the menu options or the Ctrl + Enter keys on your keyboard). This will create your first stored **R** object, a function with no arguments. This object should appear in appear in your Environment window. To see what it does, type **hello()** in the Console.

(e) Now copy some or all of the commands we used earlier in this exercise from the Console to the **R** script file (you can simply cut and paste the commands from the Console to the script file, but remember to edit out any output and any cursor **>** characters). Add in some comments (lines of text preceded by the **#** symbol, which **R** will ignore and not try to execute), so that you can later remember what you did (and why you did it).

Good readable computer code includes comments and also uses object and variable names that are long enough to actually mean something. Remember that **R** stores everything (commands, functions, data) as objects and you will start encountering cryptic messages if you start naming objects with very short names like **t** for "temporary". In base **R**, **t()** is also a function which transposes matrices and if you create a temporary object called **t** and some other command tries to access the **t()** function to transpose a matrix, you will get an error message when **R** tries to use your temporary object instead of the **t()** function. You can always use the **rm()** command to remove unwanted temporary objects e.g. **rm(t)**.

Use the menu option File > Save As to save your **R** script file in your working directory in a file called "Tutorial1_ex1.R".

To end this exercise, exit **R** by typing the command **q()** in the console (or use the menu option File > Quit RStudio). You will be asked if you want to save your workspace image and you should respond yes (**y**), otherwise you will lose any stored objects you have created.
Once you have left **R**, have a look at the contents of your working directory. You should see an .RData file which contains any stored objects you created and a .Rhistory file which contains all the commands you executed – these will be available next time you access **R** from this directory. If you are using a Windows PC, you may need to switch on the options to "Show hidden files" and to "Display file extensions" in order to see these files with their full file names.

Exercise Two

One of the hardest parts of mastering any statistical computing software is importing and managing data. If you want more details, there is a good manual on "R Data Import/Export" on the **R** Project website (https://cran.r-project.org/manuals.html). In this exercise, we are going to import some data and use **R** to manipulate and analyse the data.

Before you re-start **RStudio**, download the file "worksheet2_women.csv" from Wattle to the same area you used to store files for Exercise 1. Open this file to examine the contents – if you right-click on the file and choose "Open with" Notepad (or Wordpad), you will see that it contains two columns of numbers, with the two columns separated by commas (.csv stands for "comma separated variables"). If instead you simply double left-click on the file, it will probably open in **MS Excel**, which will automatically convert the file to two separate columns, based on where the commas where located.

Now re-start **RStudio**, by right-clicking on the file "Tutorial1_ex1.R", that you saved at the end of Exercise 1. Choose "Open with" **RStudio**. If you are working on a machine where you can reset the file associations (which unfortunately does not include the ANU InfoCommons machines), you can also tick the options to change the file associations, so that .R files become associated with **RStudio** and next time you will be able to simply double left-click on any .R file to open **RStudio**. When **RStudio** opens, you should see the workspace you saved last time with the **hello()** function as a stored object in the Environment window. Now try the following:

(a) Open a new **R** script file. Include a comment at the top about the code being for "Exercise 2 of Tutorial 1". We will use this script file to type some commands, which we can later save in a file called "Tutorial1_ex2.R" (you can use File > Save As to do that now, but remember to also File > Save later). Now we will type some commands in the script window, and then execute them so that you can see the results in the **R** Console. Try typing and executing the following commands: **c**, **t**, **vector**, **matrix**, **number**, **constant**, **scalar**.
The first few give some cryptic output (**R** is showing you the contents of these stored objects, which are typically functions). You should also try typing **help(**command**)** to see what these functions do.

The last three, though they are all common mathematical terms, are not standard **R** commands and give an error message, saying that the object was not found. This means they are safe to use as names for stored objects that you create (though even more meaningful names would be a better choice). A good test to see if a name is safe to use, is to type it in the Console and see if you get this error message. Another test is that if you do type some of the standard function names in the script window, **RStudio** will helpfully show some auto-complete options and/or try to add brackets at the end (so that you can then start typing the arguments for that function). Note that object names in R are case-sensitive, so **vector** and **Vector** would refer to different objects.

(b) There are already a large number of stored objects available with standard **R** and **RStudio** – these are located somewhere in your search path. Type **search()** to see which packages are already included in your search path. The **ls()** or list command will list the contents of the packages in your search path. By default, **ls()** lists the contents of position 1, **ls(pos=1)**, which is where objects you create are stored, but you can ask for a listing of any package; for example, **ls(pos="package:datasets")** will show you the sample datasets that come with standard **R**. Type **help(search)**, **help(ls)** and **help(**dataset name**)** for further details.
Note that items higher up the search path (in a package with a lower position number) will be found first and will effectively mask objects of the same name that are located further down the search path.

(c) We will now create and manipulate some stored data objects. Type the commands shown below. Each command consists of an assignment, which evaluates some expression on the right-hand side and then uses the assignment operator (**<–**) to assign the result to a new object on the left-hand side. These assignment commands create or over-write a stored object and do not produce any output in the Console. After each command, type the name of the new object, so that you can see the contents of the object, and also use the **class()** and **str()** commands to get some information about the type of object that has been have created. You should also use the **help()** command to examine any new functions that are used in the expressions:

```
constant1 <- 1
vector1 <- rep(constant1, 7)
constant2 <- length(vector1)
vector2 <- 1:constant2
vector3 <- c(0, 2^(1/3), sqrt(2), 2, 2*2, 2^3, 2^4)
tm3 <- mean(vector3, trim=0.25)
vector4 <- seq(28, 1, -3.5)
vector4 <- vector4[1:7]
matrix1 <-cbind(vector1, vector2, vector3, vector4)
constant3 <- ncol(matrix1)
matrix2 <- rbind(vector1, vector2, vector3, vector4)
matrix3 <- matrix1 - t(matrix2)
matrix4 <- matrix1 %*% matrix2
vector4d <- dim(matrix4)
matrix5 <- matrix2 %*% matrix1
vector5d <- dim(matrix5)
vector6 <- apply(matrix1, 2, mean)
matrix6 <- sweep(matrix1, 2, apply(matrix1, 2, mean))
matrix6[3:5, 3] <- 0
```

In summary, by applying a range of functions and operators, **R** can be used to create a number of different data structures and can work with vectors and matrices as easily as it does ordinary arithmetic. You should also try applying other simple commands such as **median**, **sum**, **prod**, **order** and **sort** to some of the objects you have just created and also examine the help files for these commands.

There are now a large number of stored objects in your Global Environment, you can use the **ls()** or **objects()** commands to list them or you can see them listed in the Environment window. If you create another object of the same name as an existing object, you will over-write the old object and if you want to remove an object, you can use the **rm()** command. Type **rm(hello)** command now to remove the **hello()** function (as it isn't really very useful).

(d) Functions, which are also stored objects, are lines of stored **R** code. We will now create and test two versions of a simple **R** function:

```
square <- function(x) {x*x}
class(square)
square(2)

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

The second version looks a little more complicated, but represents better coding practice. The last line in the second version is needed as the assignment statement (to the temporary variable **y**) produces no output and the function would produce no visible output without it. Note that if you execute all of the above code, the second version will over-write the first and you will end up with only one version of this function (the second version).

(e) Now to have a look at some realistic data. Data is most easily entered in **R** by importing from an external file, typically text (.txt) files or comma separated variables (.csv) files. Most of the data files on Wattle will be .csv files. To enter such a file, download the file to your working directory (as you did with the "worksheet2_women.csv" file earlier) and use the command **read.csv(**file name**)**. For example:

```
help(read.csv)

ws2.women <- read.csv("worksheet2_women.csv", header=F)
ws2.women
str(ws2.women)
attributes(ws2.women)
```

Note that **read.csv()** shares a help file with the **read.table()** command, as **read.csv()** is a special case of the more general command for reading in text files. When reading in the file, we have to specify **header=F**, as this particular file does not have variable names as column headings (most of the files in this course will include names). You could also have stored the file somewhere other than your working directory, but then you would have had to include the full path name in the file name (replacing the \ characters with **/**).

The object that has been created is a new class of **R** object, a data frame. Data frames are ideal objects for storing data for analysis, as they are organised with each row representing an individual case or observation and each column being a different named variable. As the input file didn't have headings, **R** has assigned **V1** and **V2** as variable names. **V1** and **V2** are actually the heights (in cm) and the weights (in kg) of a sample of 10 women. I used the name **ws2.women** as there is already a similar standard R dataset of the same name (type **women** and **help(women)** to see the details). We can fix the variable names in our data frame (**ws2.women**) and make these variables more accessible for analysis by doing the following:

```
names(ws2.women) <- c("Height", "Weight")
ws2.women
attach(ws2.women)
search()
ls(pos=2)
```

The **attach()** command has varied your search path by temporarily including the **ws2.women** data frame at position 2. This means you can access the variables **Height** and **Weight** by simply typing those names, for the remainder of this **R** session. The data frame will be saved when you save the workspace, but you will need to attach it again for use in later sessions.

(f) Now that we have looked at basic data structures and objects, we are finally ready to have a look at the main **R** functions we will be using to do regression modelling in the remainder of this course. As we have attached the **ws2.women** data frame in part (e), we can first do a little EDA (exploratory data analysis) on the variables:

```
summary(ws2.women)
hist(Height)
hist(Weight)
boxplot(Height, Weight)
plot(ws2.women)
```

The functions **summary()** and **plot()** are generic functions. Generic functions work on a variety of different objects by first working out what type of object it is dealing with and then calling an appropriate method to deal with that object. You can adjust various aspects of the graphics produced by **plot()**, **hist()**, **barplot()** and other graphics functions in **R** by specifying different graphical parameters, see **help(par)** and the help files for the different functions for details.

All research is designed to answer questions. The researchers who collected the data were almost certainly trying to some research question, but as we have been given almost no information about the data, it is difficult to say what exactly that research question was. They could have been simply interested in examining the relationship (or correlation or association) between height and weights in whatever population this sample was taken from, or they might have been interested in trying to predict a women's weight based on her height (heights are arguably slightly easier to measure than weights). As we will see later in the course, we can often use a simple linear regression model to address aspects of such questions. We can fit such a model in **R** using the following code:

```
women.lm <- lm(Weight ~ Height)
attributes(women.lm)
women.lm$coef
```

The new object **women.lm** is a linear model (lm) object, which is stored in yet another new kind of R object, a list object, which is a list (collection) of named objects and you see the various objects in the list by typing **women.lm$**name. For example, the estimated regression coefficients are stored in **women.lm$coefficients**, but you access this object using an abbreviation such as **women.lm$coef**, which is long enough not to be confused with one of the other names in the list. We can also apply various generic functions to get other output about this lm object:

```
plot(women.lm)
anova(women.lm)
summary(women.lm)
```

We will be discussing data frames, list objects, linear models and the above linear model output in a lot more detail later in this course (and may even revisit this example again), but for now let's draw a plot of the data with better labels (than the earlier plot) and with the fitted simple linear regression model added to the plot, using the **abline()** function:

```
plot(Height, Weight, xlab="Height (in cm)", ylab="Weight (in kg)")
title("Sample of 10 women")
abline(women.lm$coef)
```

Do you think this is good (well-fitting) model for these data? Is it a useful model for addressing the research question (whatever that was)?

Exercise Three (optional extra – additional examples for students who want more practice using **R**)

The following examples have been chosen to give you some idea of the range of tasks you can do with **R**. They are for you to do in your own time, however, the course tutors and I will be happy to answer questions about them. You should complete the first two (compulsory) exercises before you attempt these additional questions and you may not be ready to successfully complete all of these additional questions until after you have reviewed a number of the lecture examples, and have become a little more accustomed to using **R**.

(a) Produce a plot of the function:

$$f(x) = e^{-|x|}$$

for values of *x* ranging from –3 to 3. This function is sometimes called the *double exponential* function. Make sure to appropriately label your plot and both axes.

(b) The file "worksheet4_mussels.csv", which is available on Wattle, contains data from a study of the age and growth characteristics of selected mussel (shellfish) species in two distinct locations in Southwestern Virginia, USA. The file contains three columns: the first indicates the **location** (region 1 or 2) where the data was collected; the second contains the **age** (in years) of the selected mussels; and the third column gives the **weight** (in grams) of the selected mussels.

On the same set of axes, plot the **weight** versus **age** relationship for each of the two locations, connecting the data points within each **location** with lines. Make sure to use distinct symbols and line-types for the data from the two different locations and label the plot appropriately. [Hint: look at the help file on the **lines()** command for instructions on how to overlay plots.]

Use **lm()** to fit a least-squares linear regressions to the data from each of the two locations separately. On the same set of axes, plot the weight versus age relationship for each of the two locations, and superimpose the two regression lines. Again, make sure to use different symbols and line-types for the two different sets of points and regression lines. Read the help file for the legend() command and add an appropriate legend to your plot.

Describe any apparent differences in how the mussels grow in the two locations.

(c) Use the **rnorm()** function to generate 1000 normal variates from the normal distribution with mean 3 and variance 16, and put the result into a vector called **normals** and construct a histogram of these values. What are the smallest and largest values in **normals**?
The histogram command can be forced to use certain break points rather than using the default breakpoints. For example, the command:

```
hist(x,breaks=seq(floor(min(x)),floor(max(x)+1),1))
```

will create a histogram of data values in **x**, but it will use bins of width 1 spanning the range between the smallest and largest integers bracketing the range of **x** values respectively. By adjusting the bin-width (that is, the third and final argument in the **seq()** function above), you can achieve a smoother or rougher looking histogram than that produced by the default bin-width. Experiment with various bin-widths by varying the line given above and see what bin-width yields the "smoothest-looking" histograms. What do you notice about the histograms as the bin-width decreases? What about as bin-width increases?

When we talk about "smoothness" of a histogram (as above), we are referring to how close the histogram is to a smooth curve. What curve is the histogram approximating? Since the variable **normals** ostensibly comes from a N(3, 16) distribution, it makes sense to compare it with that normal distribution. To see what this distribution looks like, take a vector that spans the range of the values in **normals**, for example, the vector **grid**:

```
grid <- seq(floor(min(normals)),floor(max(normals)+1),1)
```

is such a vector. Use the function **dnorm()** to find the values of a N(3, 16) density corresponding to the values in **grid**, and store these in a vector called **really.normal**. Now, use the **lines()** command to superimpose the plot of **grid** versus **really.normal** on the histogram with bin-width 1 that you constructed above. What do you notice? If you said "not much", you are right, since the normal density we plotted has a completely different scale to the histogram. This is because the histogram uses raw frequencies rather than relative frequencies. To account for this, we will want to "blow up" the normal density we plotted by a factor of 1000 (the total number of simulated values) multiplied by the bin width (in this case 1). So now superimpose on your histogram the plot of **grid** versus 1000 × bin-width × **really.normal**. You may like to compare a plotted density against each of the other histograms you created above to see which bin-width corresponds most closely to the true curve. Is the default in **R** the right choice of bin-width to accurately portray the data?

Like other computer packages, **R** actually generates "pseudo-random" numbers based on some seed, such as the time of day. Given this, are the normal variates generated by **R** credibly normal? A better plot than the histogram for "testing" normality is a normal q-q plot, which tries to match the ordered data values with what we would expect to be the ordered values from a normal distribution with the appropriate mean and variance (the latter values are called normal scores). Use the **qqnorm()** command to construct a normal q-q plot of the simulated data in the vector normal. Do you think the simulated data might reasonably have come from a normal distribution?

Finally, let's look at a small sample from a non-normal distribution. Generate a vector **expo** of 15 variates from an exponential distribution with mean 1. What is the mean of your sample? What is the variance? Plot a normal q-q plot for the data in **expo**. What do you notice? How do these data differ from a normal distribution?

(d) Recall the difference between a sample median and a sample mean – the median is less sensitive to outliers, but the mean is usually a better summary measure for normally-distributed data. To check this out, create a sample of size 50 from a standard normal distribution, and find the mean and median for the data. Now add 100 to the first observation in your sample, to make it an outlier. How are the median and the mean affected by the outlier?

Now do some simulations to compare the performance of the median and the mean for the standard normal and the Cauchy distributions in repeated sampling. Sampling from the Cauchy distribution typically produces samples with apparent outliers.

In the following code, we first create some space to store the results and then we repeatedly simulate samples of size 50 from both the normal and Cauchy distributions and calculate the mean and median of each sample. We use a **for**() loop to do this 100 times to fill the space we created:

```
means.normal <- rep(0, 100)
medians.normal <- rep(0, 100)
means.cauchy <- rep(0, 100)
medians.cauchy <- rep(0, 100)
for(i in 1:100) {
  x <- rnorm(50)
  y <- rcauchy(50)
  means.normal[i] <- mean(x)
  medians.normal[i] <- median(x)
  means.cauchy[i] <- mean(y)
  medians.cauchy[i] <- median(y)}
```

Investigate the results using histograms, means, medians and standard deviations.

(e) Functions in **R** have the form: **function(**argument1, argument2, …**)**{lines of code**}**

As we saw in exercise 2, the last line of code in the function will be the value returned by the function, though some functions may just manipulate other stored objects and not return any visible output.

We are now going to write a function which will fit a simple linear regression model and then fit the same simple linear regression model excluding one or more of the observations (which may be possible outliers); so we can examine how the coefficients of the fitted regression model change, with and without those observations. To introduce some more **R** programming constructs, we will use conditional branching with **if** and **else** statements to check whether we have been given a valid list of observations to exclude:

```
beta.change <- function (y, x, exclude){
  excl <- unique(exclude) # check there are no repeated values in exclude
  if(min(excl)<1) {print("invalid exclusion - index too small")}
  else if(max(excl)>length(y)) {print("invalid exclusion - index too large")}
  else {
    beta <- lm(y ~ x)$coef
    beta.excl <- lm(y[-excl]~x[-excl])$coef
    beta - beta.excl
    }
}
```

Run the above code to create this function and try it out on the data **ws2.women** from exercise 2 (hopefully you still are using the same workspace). Can you find any data points which have a large effect on the values of the regression coefficients when you exclude them?

(f) Now it is your turn to write a function. Suppose we want to implement a method called the "jackknife" which can be used to examine the bias of estimation procedures in many situations. We don't want to go into the theory behind the jackknife here, but we would like to write an **R** function to implement a jackknife bias estimate for the slope coefficient from a simple linear regression.

We start with a dataset of size n, with data points $(x_1; y_1)$, $(x_2; y_2)$, … $(x_n; y_n)$, contained in two vectors x and y. The basic idea of the jackknife is to form n new datasets, each one the same as the original dataset except that the ith new dataset is missing the ith point from the original dataset. The new datasets are therefore each of size n–1. For example, the first new dataset is $(x_2; y_2)$, $(x_3; y_3)$, … $(x_n; y_n)$ (i.e. the original dataset missing the first data point), the second is $(x_1; y_1)$, $(x_3; y_3)$, … $(x_n; y_n)$, and so on.

Now, for each of these reduced datasets, calculate the slope of the least-squares linear regression and construct a vector containing the n differences of each of these slopes from the overall slope of the least-squares linear regression on the entire original dataset. In other words, create a vector of length n, whose ith component is the difference between the original slope and the slope of the least-squares linear regression on the reduced dataset. [Hint: recall the function **beta.change()** from part (e).]

The jackknife estimate of the bias of the regression slope is then just n–1 times the average of this vector of differences.

Write an **R** function that takes two columns of data, a response variable (y) and a predictor (x), and returns the jackknife estimate of the bias for a regression slope.

Suppose you wanted to generalize your function to calculate the jackknife bias estimate for a general user-chosen estimator based on some dataset, $\aleph$. If the user-chosen estimator was defined to be a function of the data, say $\hat{\theta}(\aleph)$ which had been coded as the **R** function **theta()**, how you might modify your function to accommodate this generalization?