Modelling behavioural data using R

Daniel Nettle

25 September 2017

Session 1: Introducing R

This session is mostly an introduction to R and the RStudio environment. We will go as far as loading a dataset in, and performing a very simple analysis or two. The discussion of statistical models, why we do them, and what they mean is going to be deferred to session 2. There is quite a lot of slightly nerdy background about how the R language works. You may not see the value of this right now, but we are trying to secure some useful concepts that may pay for themselves in future.

Getting started and oriented

Installation

We are going to interact with R through a programme called R Studio. R Studio is a nice piece of software that provides a user interface in which you can send commands to R, edit your scripts, preview graphics, and see what is going on in your environment. R Studio is useful, but it is not necessary in order to work with R; there are a number of other interfaces for interacting with R, some graphical, some text based. And almost all the code you write will be usable by R regardless of whether you are going through R Studio or not.

You first need to download and install R from https://cran.r-project.org/, choosing your operating system. Then and only then download and install R Studio (the free R Studio Desktop Open Source Licence).

The R Studio screen

Open R Studio. You should have a divided screen with the **console** window on the left, and on the right, a small window that can tab between **environment** and **history**, above another that can tab between various things such as **files**, **plots** and **help**.

A word of warning

We are about to start sending commands to R. In my experience, when you are starting out, you are going to get error messages about half the time and find it frustrating. This will almost always be because you have mis-typed. Working with R is not like working with a package like SPSS (where you choose what you want from a menu), or indeed like Word (where typos are auto-corrected). For R to do what you want, every comma has to be exactly in the right place; every opened bracket has to be closed with a bracket of the same kind; quotation marks may be needed, and if they are, they need to be closed at the right point; variable names need to be specified correctly; and everything is case-sensitive. You will get much better at not making typos, surprisingly quickly.

First steps

Speaking to R

The **console** is the actual place through which you can directly 'speak' to R, by typing commands, one at a time, which R will execute when you press enter. R is listening whenever you have a > cursor. So go ahead to the console and type:

```
3 + 5
```

```
## [1] 8
```

Then press return. R's answers here are going to be denoted with ##. You will see that R's answer is [1] component long, and it can do simple addition. Hurrah! So R is a very sophisticated calculator. Now as well as adding numbers, R can hold **objects** in its head. So for example, type:

```
x = 3
```

The first thing you see is that in the **environment** tab to your right, an object x appears, with the current value of 3. That means R current has 'in its mental workspace', object x. And once it is there, you can do anything to it you could do to a number. So for example try a few calculations like.

```
x + 5
x * 4
sqrt(x)
x^2
```

The last two are the square root and the square of x. You should get the numbers you expect! And if you want R to report the value of an object in its environment, then you just type the object's name:

```
x
```

```
## [1] 3
```

All numeric objects in R are actually vectors. This means that they are not single numbers, but ordered series of numbers (our x so far is a vector of length 1, so this has not been evident). You make several numbers into a vector using the function **c**, which stands for combine. So:

```
x = c(2, 3, 5, 7)
```

Now you have a numeric vector in your environment of length 4. Now repeat the calculations you did before, and note what the output is.

```
x + 5
x * 4
sqrt(x)
x^2
```

Make sense? (Maths nerds hote that if you multiply the vector x by something here, for example itself, then you do not get vector multiplication, by element-wise scalar multiplication.). You can also query or change any particular element of a vector using the square bracket notation []. For example:

```
x[3]
```

```
## [1] 5
```

or

```
x[3]=100
x
```

```
## [1]   2   3 100   7
```

Now that x is a vector of more than 1 number, we can also apply some other functions to it. Try:

```
mean(x)
sd(x)
max(x)
median(x)
```

```
min(x)
length(x)
```

Happy with the answers to all of these?

Assigning and equalling

Consider:

```
y = 10
```

It is important to understand that when we say this in R, we are not using '=' in quite the way it is used in a mathematical equation. The line 'y = 10' says 'assign the value on the right hand side (10) to the object on the left hand side (y)'. In R '<-' also means 'assign'. You will see '<-' a lot in other people's code; for our purposes, '<-' and '=' are synonymous. Because '=' means assign in R, then expressions like the following make perfect sense:

```
y = y + 1
```

Try it. Set y to 10, then assign y + 1 to y. The value of y should increase to 11. This is perfectly sensible programming (assign the object y with a value one greater than its current value), but very bad mathematics (y is not equal to y+1. It is equal to y!).

Sometimes you are going to need to use equals in the mathematical sense, and so here there is special R lingo for doing this: double equals, also known as checking equals.

```
y = 10
y == y + 1
```

```
## [1] FALSE
```

Quite rightly, R tells you, since you asked, y is not currently equal to y + 1, it is equal to y. Now try this:

```
x = c(2, 3, 5, 7)
x == 3
```

What do you get? A series of TRUEs and FALSEs, element wise. When you use equals in R, always ask yourself, do I mean 'R: Set this thing equal to this!' (single equals); or do I mean 'R: check whether this thing is equal to this' (double equals).

If you are feeling frisky can define another object, z, that holds the values of whether each element of x is equal to 3 or not

```
z = (x == 3)
```

Now look in your environment window. You now have another object, z. But note that whereas the object x is denoted 'num' (it's a numeric vector), z is denoted 'logi', because it is a different type of vector called a logical vector (it consists of TRUEs and FALSEs). So it's now time to say a bit more about objects and functions in R.

Objects and functions

The things you care about in R will mostly be either **functions** or **objects**. Functions are commands followed by () brackets, and they basically say to R, do the operation defined by the function to the objects listed in the brackets (these are called the arguments to the function). For example:

```
mean(x)
```

```
## [1] 4.25
```

If you don't say otherwise, R will return the outcome of doing the function to the console. But you might want the outcome stored in an object instead, as in:

```
overall.mean=mean(x)
```

See now the object overall.mean appears in your environment to the right. This could be useful, for example if you want to know which of your data points in x lie below the mean.

```
x = c(2, 3, 5, 7)
overall.mean=mean(x)
which(x<overall.mean)
```

```
## [1] 1 2
```

The first and second elements of x are below the mean.

Just occasionally, there are functions with no obligatory arguments. The function call still requires its () brackets; that's how R knows it's a function you want, not an object. The brackets are just empty in these cases. For example, the citation() function has no obligatory arguments and returns the citation for your version of R to put into your publications.

```
citation()
```

```
##
## To cite R in publications use:
##
##   R Core Team (2017). R: A language and environment for
##   statistical computing. R Foundation for Statistical Computing,
##   Vienna, Austria. URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {R: A Language and Environment for Statistical Computing},
##     author = {{R Core Team}},
##     organization = {R Foundation for Statistical Computing},
##     address = {Vienna, Austria},
##     year = {2017},
##     url = {https://www.R-project.org/},
##   }
##
## We have invested a lot of time and effort in creating R, please
## cite it when using it for data analysis. See also
## 'citation("pkgname")' for citing R packages.
```

Objects are the things you perform functions on. The important thing to grasp is that R, every object has a **class**. The class determines the kind of thing it is, and hence how any particular function can be applied to it. To find out the class of an object in your environment, use the class() function.

```
class(x)
```

```
## [1] "numeric"
```

```
class(z)
```

```
## [1] "logical"
```

So x is a numeric vector (it consists of data values), and z is a logical one (it consists of TRUEs and FALSEs). R Studio also helpfully shows this information all the time in your environment window to your right.

There are also other kinds of objects. For example you might have some categorical data (note the quotation marks):

```
gender=c("female", "female", "male", "female")
class(gender)
```

```
## [1] "character"
```

You see that gender is an object of the class character vector. And an object's class may determine whether a given function can be properly applied to it. What happens if you now try:

```
mean(gender)
```

```
## Warning in mean.default(gender): argument is not numeric or logical:
## returning NA

## [1] NA
```

Quite right. The vector has no mean because it is not numeric.

There are objects of many classes in R. Your data variables are objects, usually of classes numeric, logical, character or factor (we will meet factors anon; you can also force R to convert data from one class to another if needed, see Appendix I).

Your whole data set is also an object (usually of class data.frame). And - this is the thing that takes some getting used to - your statistical tests are also objects (with their own classes). Your plots may be objects too. This is useful because once they are defined in your environment, you can apply all kinds of functions to them like printing them, plotting them, combining them, comparing them, and so forth. So the thing you have to get used to in R is: everything is an object, with a class proper to the kind of object that it is.

Writing scripts

So far we have been working directly into the console, writing one command at a time for R to execute. But when you do data analysis, what you really want to do is to write a whole series of commands, edit and perfect them, then send them to R for execution as a block. We do that by writing what is called a **script**. A script is just a text file with a series of lines of code that will get sent to R when you are ready to execute it.

Within R Studio, you open a new script using the 'File > New File > R Script' menu commands at the top. Try it.

There's your blank script. The first thing you might want to do is save it somewhere with an approriate name (save icon on the top of the script window, or 'File > Save' via the menus. Or Control+s.). The file extension should be .r. Done that? Ok. Lets write something in our script. Try writing out:

```
# This is my first script
a = 2
print(a)
a = a *3
print(a)
print(rep("I love R", times = a))
```

Now, nothing happens. That's because we have written our script, not sent it to R for processing. So go ahead now and hit the button at the top right of your script window marked 'Source'. See what you get? So now all the lines are executed by R, in the order they are written in.

By the way, if you want to run just part of your script rather than the whole thing, put the cursor on the line you want to run, and hit the 'Run' button. You can also use the mouse to select several lines and run them. Control plus enter when the cursor is in the script window also runs the part of the script where the cursor currently is.

The thing about having a script is that you can edit it. So now go back and edit the line 'a = 2' so it says 'a = 50'. Now source the script again.

Great. So what did we learn about what the function 'rep()' does? And what is the significance of its 'times' argument?

One more thing on our first script: the first line. What did that line do? Absolutely nothing, because it began with a #. Lines that begin # are ignored when the script is sourced. They are called comments. They are there to remind yourself or explain to others what the script, or a particular part of the script, is doing. You can also make non-executed section headings in your scripts using three comment symbols at the beginning of the heading, and three at the end, as in:

```
### Calculating the final value of a ###
a = 2
print(a)
a = a *3
print(a)
### Printing out 'I love R' a times###
print(rep("I love R", times = a))
```

These headings can then be jumped between using a little menu at the bottom of the script window - very useful for longer scripts.

The working directory

We are getting on to using a more interesting script, but first, a bit about **working directories**. The working directory is the place on your computer R is going to look first for files (like scripts and data files) to read in, and also the default place it is going to save to. It's worth setting the working directory right before you begin so you know where everything needs to be. You find your current working directory with the function:

```
getwd()
```

```
## [1] "D:/Dropbox/Fun with R/Session 1"
```

And you set the working directory with the setwd() function:

```
setwd("C:/Users/Daniel/Dropbox/R")
```

Or whatever directory you want. It is tedious to type out a long path name. R Studio instead allows you to set the working directory via a menu by browsing your computer: 'Session > Set Working Directory > Choose Directory'.

You can set the default working directory on start up to be anything you want, though it is slightly fiddly. In fact, you may want to set several of the start-up defaults for your R installation. Information on how to do this can be found in Appendix II.

A first data set

Ok, we really are going to do something more interesting now. We are going to read in some data and do a simple analysis or two. The data come from a study of the weights and heights of US military personnel (I have randomly sampled 50 individuals from the originally vast dataset).

First, have you got the file 'weight.data.csv' saved somewhere? Yes. Good. So use the Session menu to set the working directory to that directory. Now, open a new script and save it with a sensible name.

I have prepared a data file for us to analyse, in .csv format. In this format, the data values are separated by commas and the first row tells you the variable names. It is easy to save to .csv format from Excel, SPSS, or any other spreadsheet or stats package you are likely to be using. If you are entering data manually, do it in Excel, and save as .csv. R can handle many other data formats, but .csv files are simple and small.

First thing to type into your script:

```
d = read.csv("weight.data.csv")
```

The read.csv() function, astoundingly, reads in a dataset from a .csv file. Execute the command and an object called d appears in our environment (we didn't have to call it d, by the way; you could call it anything you like). Let's find out what class d is.

```
class(d)
```

```
## [1] "data.frame"
```

Yes, d is a special class of object called a data.frame that consists of several data vectors lined up against one another. To see what d looks like, the head() function is useful.

```
head(d)
```

```
##   X SubjectID Weight Height Leg.length Age    Sex
## 1 1      4885   81.5  182.3      110.0  32   Male
## 2 2     15016   88.8  182.6      110.8  30   Male
## 3 3     25895   45.5  167.3      101.7  20 Female
## 4 4     11885   85.4  180.3      115.0  19   Male
## 5 5     19382   72.8  170.5      106.5  26 Female
## 6 6     11841   75.7  177.7      110.2  21 Female
```

This gives you the first six rows to look at. Another useful function is the structure function str().

```
str(d)
```

```
## 'data.frame':    50 obs. of  7 variables:
##  $ X         : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ SubjectID : int  4885 15016 25895 11885 19382 11841 13454 25913 13127 8035 ...
##  $ Weight    : num  81.5 88.8 45.5 85.4 72.8 75.7 65.2 61 60.3 64.8 ...
##  $ Height    : num  182 183 167 180 170 ...
##  $ Leg.length: num  110 111 102 115 106 ...
##  $ Age       : int  32 30 20 19 26 21 19 19 31 29 ...
##  $ Sex       : Factor w/ 2 levels "Female","Male": 2 2 1 2 1 1 2 1 1 1 ...
```

This tells you lots of useful information. The object d is made up of 50 observations of 7 variables. And it tells you that d has 7 sub-objects called things like Weight, Height and Sex. (You can also get this information in the Environment window, by clicking on the little expansion arrow next to the object d.) Another useful function is colnames(), which returns the column names of a data frame.

```
colnames(d)
```

```
## [1] "X"          "SubjectID"  "Weight"     "Height"     "Leg.length"
## [6] "Age"        "Sex"
```

You can access the sub-objects of an object (usually) using the $ notation. So, for example:

```
d$Sex
```

This returns the value of the sub-object (i.e. column) called d$Sex. Note that you can also interrogate a data frame using row and column positions:

```
d[3, 7]
```

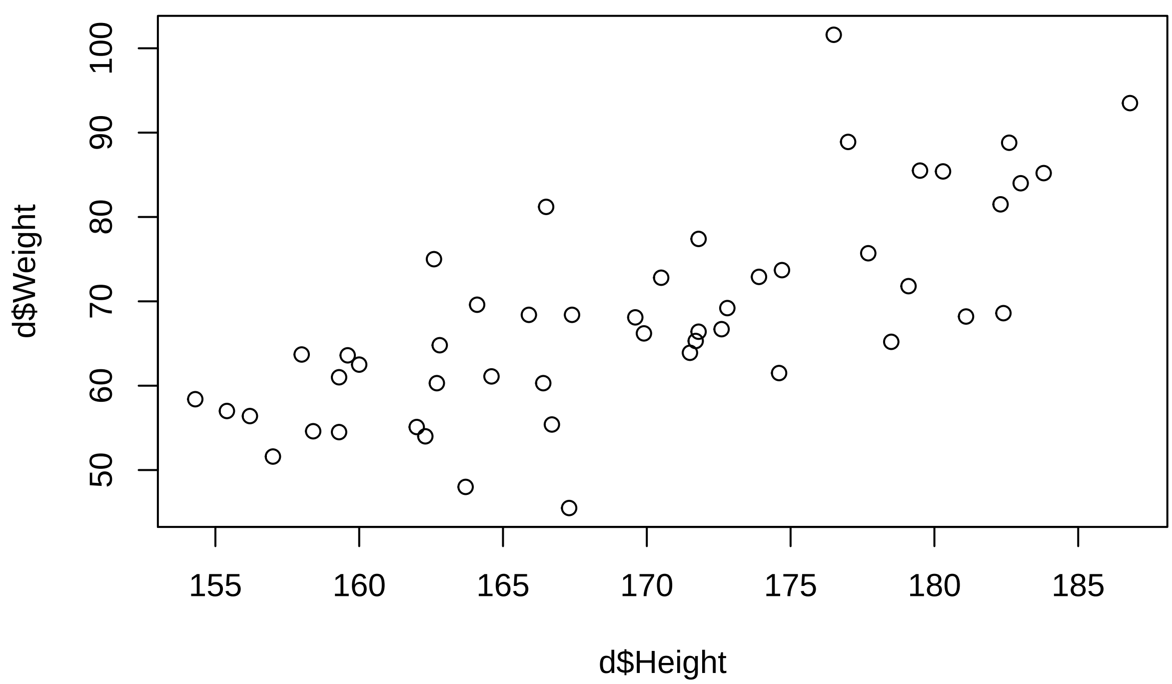
```
## [1] Female
## Levels: Female Male
```

The seventh variable of the third case of the data set is 'Female'. Note that it also helpfully tells you that the Sex variables has two levels, Female and Male. This is because in reading in the data, R has recognised that this variable might be a categorical variable with a limited set of defined options, and thus given it the class 'factor', which is specially designed for variables of this type. This will probably be fine, but the class 'factor' does cause some headaches occasionally, such as when you want to add in more data points and some of these belong to a category that was not in the original set. See Appendix I for converting factors to other classes if need be. (For interest: factors are actually encoded as integers by R, with each integer corresponding to a level label.)

Some simple plots

Now we have some data, let's do some simple exploratory plots. Add to your script the following line and execute it:

```
plot(d$Weight~d$Height)
```



You should have got a draft plot in the bottom right window, showing that as people's height in cms goes up, their weight in kg tends to do so too. Hurrah! You can confirm this if you like by calculating the correlation coefficient between height and weight.

```
cor.test(d$Weight, d$Height)
```

```
##
##  Pearson's product-moment correlation
##
## data:  d$Weight and d$Height
## t = 7.3235, df = 48, p-value = 2.352e-09
## alternative hypothesis: true correlation is not equal to 0
```

```
## 95 percent confidence interval:
##  0.5616595 0.8357877
## sample estimates:
##       cor
## 0.7264383
```

There are lots of ways of making your plot nicer, labelling the axes and so forth, but we won't go into these today as we will be doing plotting in a later session.

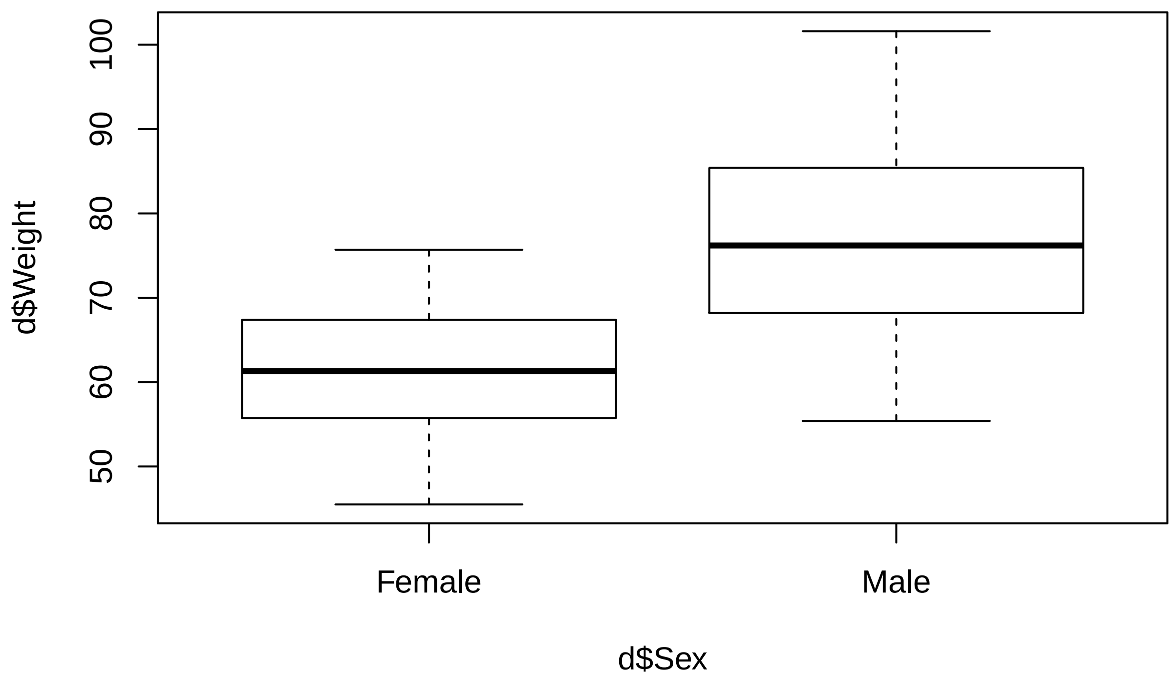
In the plot command you just wrote, we encountered for the first time something we will often meet in R: the formula. An R formula is something on the left hand side, then a tilde symbol, then something on the right hand side, as in:

```
d$Weight~d$Height
```

You will use formulas of this kind throughout your R career, both for plots and for statistical tests. The left hand side is what you want as the outcome variable / vertical axis; the right hand side is what you want as the predictor variable(s) / horizontal axis.

What happens if you execute:

```
plot(d$Weight~d$Sex)
```



Because R can see that the class of d$Sex is factor, not numeric, it is smart enough to see that it needs to choose a different type of plot than for weight against height, so you get a boxplot instead of a scatter plot. And we see that the men seem to be generally heavier than the women. By the way, in your plot window, you can go back to earlier plots and forward to later ones using the left and right arrows at the top of the window, and you can also make them bigger and copy them to the clipboard.

A first statistical analysis

Shall we do a test of whether we can infer that men are heavier than women? You know you want to. Let me introduce the lm() function. The lm() function will be your workhorse for statistical analysis. It fits a general linear model to the data with the formula that you give it when you call the function. So, for example, try executing:

```
lm(d$Weight ~ d$Sex)
```

```
##
## Call:
## lm(formula = d$Weight ~ d$Sex)
##
## Coefficients:
## (Intercept)    d$SexMale
##       61.65        15.27
```

Well, some coefficients come up, but it does not seem to be very useful. That's because to make lm() useful, it's best to save the resulting model to an object. So now try:

```
m=lm(d$Weight ~ d$Sex)
summary(m)
```

```
##
## Call:
## lm(formula = d$Weight ~ d$Sex)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -21.518  -7.125  -0.350   6.750  24.682
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   61.650      1.802  34.214  < 2e-16 ***
## d$SexMale     15.268      2.716   5.621 9.47e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.535 on 48 degrees of freedom
## Multiple R-squared:  0.3969, Adjusted R-squared:  0.3844
## F-statistic: 31.59 on 1 and 48 DF,  p-value: 9.47e-07
```

The first line creates a new object m and assigns it a general linear model of Weight by Sex from data frame d. You might be interested to check the class of object m with class(m). Then the second line summarises object m, which gives you all kinds of useful information. And m is still there in your environment for when you need to refer back to it.

Let's look at the summary of m in a bit more detail. The model shows two parameter estimates of interest: a 61.650 described as the intercept, and a 15.268 described as SexMale. What does this mean? Well, we will talk about this a lot more in session 2, but for here suffice it to say that our best estimate of the weight of a person who is not male in this population is 61.650 kgs, and our best estimate for the weight of a male person is 15.268 kgs more than that, i.e. 76.918 kgs. And the very small p-value suggests that if we sampled again from the population, we have a very small chance of obtaining a sample where the females were as heavy as the males, or heavier. Looks like there is probably a robust weight difference between males and females in this population, a difference of about 15 kgs on average.

As it stands, female is the default sex, and so the intercept gives the weight of a female, and then male is expressed as a deviation from that. You might for some reason want it the other way around. Fair enough. Let's create a new variable in data frame d called 'Sex2':

```
d$Sex2=relevel(d$Sex, ref="Male")
```

Now run model m2 with the Sex2 variable.

```
m2=lm(d$Weight ~ d$Sex2)
summary(m2)
```

```
##
## Call:
## lm(formula = d$Weight ~ d$Sex2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -21.518  -7.125  -0.350   6.750  24.682
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    76.918      2.033  37.838  < 2e-16 ***
## d$Sex2Female  -15.268      2.716  -5.621 9.47e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.535 on 48 degrees of freedom
## Multiple R-squared:  0.3969, Adjusted R-squared:  0.3844
## F-statistic: 31.59 on 1 and 48 DF,  p-value: 9.47e-07
```

Compare summary(m) with summary(m2). It comes to the same thing of course, but in one, the default weight is nearly 77 kgs, and females are that minus 15 and a bit kgs, whereas in the other, the default weight was 61 and a bit kgs, and males were this plus 15 and a bit kgs.

The good thing about having our model saved as an object is that we can pull out sub-objects from it. For example, try:

```
summary(m)$r.squared
```

```
## [1] 0.3969212
```

```
summary(m)$coefficients
```

```
##             Estimate Std. Error   t value     Pr(>|t|)
## (Intercept) 61.65000   1.801889 34.214101 2.272530e-35
## d$SexMale   15.26818   2.716449  5.620639 9.469594e-07
```

And once you pull out these aspects of the model, you can compare them to otherr things, write if…then statements concerning them, plot them, and so on.

Piece of housekeeping. It is tedious and repetitive to have to write d dollar on each side of your formula. The data argument to the lm() function allows you to specify that all variables in the formula come from the data frame d, and hence avoid having to specify d dollar for each variable. As in:

```
m=lm(Weight ~ Sex, data=d)
summary(m)
```

So what we have done in this section is fitted a general linear model for a continuous outcome variable (Weight) and a categorical predictor (Sex). If you have done some statistics before, you may well ask, why did you not just do a t-test? Isn't that the simple test you use when you have a continuous outcome and a single dichotomous predictor. Well yes, that is true. But really the t-test is just a sub-case of a general linear model. We will deal with this more in session 2. But if you want to convince yourself that this is the case, try:

```
t.test(d$Weight~d$Sex, var.equal=TRUE)
```

```
##
##  Two Sample t-test
##
## data:  d$Weight by d$Sex
## t = -5.6206, df = 48, p-value = 9.47e-07
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -20.729969  -9.806394
## sample estimates:
## mean in group Female   mean in group Male
##             61.65000             76.91818
```

Now compare the output you get with that you get from summary(m). You should see that the results, though presented in a slightly different format, are numerically identical. So the answer to the question: why did you fit a general linear model rather than doing a t-test? is: mathematically they are the same thing. (That is, when we are talking about the standard t-test that assumes equal variances in the two groups, which we specified using var.equal=TRUE. There is a slightly different t-test that does not assume equal variances in the two groups, which you can call using var.equal=FALSE in the t.test() function. The results in the present case are pretty similar.)

Contributed packages

There is one more small thing we want to do today. R is not just one language, but a **base** package (which we have already been using), and an archipelago of modules called **contributed packages**. These are sets of functions for doing specific things that R does not do conveniently by default. The contributed packages are written by many different people, and each has its own lingo, though of course they are all broadly compatible with the rest of R. In this section, we will download and use one contributed package as an example. The package is called 'Psych', and contains a lot of useful functions for easily doing the kinds of analyses psychologists typically require (amongst other things it's good for factor analysis, for example).

To install a package, we need to be connected to the internet, as it is a download. Then type and execute:

```
install.packages("psych", dependencies = TRUE)
```

The dependencies = TRUE argument is not strictly necessary, but since people author packages that in turn require other packages, it makes sure you get eveything you need to use the package 'psych'. Various files will download and progress messages will be displayed. The files that make up the package(s) will be saved - for more on controlling where in your computer they get saved to (it's not the working directory), see Appendix II.

You only need to do install.packages() once on a given computer to get a particular package, though new versions of packages are always being released, so you might wish to reinstall from time to time. But **every** session you want to use the package, you need to call it using the library() function (require() does almost the same thing). So if in an analysis you will use the package psych, then somewhere in that script/session you need to execute:

```
library(psych)
```

Now psych is ready and waiting for you to enjoy its smorgasbord of functions. A great function provided by psych is the describe() function, which gives you all your descriptives in an economical way. Try:

```r
psych::describe(d)
```

The 'psych::' before the function name tells R we are talking about a function from the psych package rather than anywhere else. But as long as you have no other packages active that also contain a function called describe(), then it is fine to omit it. So you get the same outcome with:

```r
describe(d)
```

You can also describe just one variable rather than the whole data frame. Try:

```r
describe(d$Weight)
```

Finally, I particularly like the related function describeBy(), very suitable when your data contains a discrete number of conditions or sexes or types of individual.

```r
describeBy(d$Weight, d$Sex)
```

```
##
##  Descriptive statistics by group
## group: Female
##    vars  n  mean  sd median trimmed mad  min  max range  skew kurtosis
## X1    1 28 61.65 7.6   61.3    61.8 8.6 45.5 75.7  30.2 -0.15    -0.74
##      se
## X1 1.44
## --------------------------------------------------------
## group: Male
##    vars  n  mean    sd median trimmed   mad  min   max range skew kurtosis
## X1    1 22 76.92 11.55   76.2   76.56 13.49 55.4 101.6  46.2 0.17    -0.88
##      se
## X1 2.46
```

There are many other contributed packages you will use, but this section has given you an example of how easy it is to do so.

Version control

New versions of R and of contributed packages are being issued constantly. This is annoying, but all kinds of little things do get fixed and improved. You can have problems when a contributed package was written on a more recent version of R than you are currently using, and also problems when old versions of one package make another one fall over. I recommend reinstalling the latest version of R every couple of months, and also reinstalling contributed packages you depend on from time to time. Nasty error messages–including those which do not tell you that version incompatibility is the problem– can often be eliminated this way. Not always of course!

Getting help

There is loads of good R help on the web. To understand more about a function, you can search via the Help tab n the bottom right window. Or just Google your problem or the name of the function/package you are trying to master. There is an amazing amount of freely-available information out there, including example code that you can copy. Also, if you get a particularly nasty error message (and R's error messages are generally unhelpful and obscure), type it into Google and some poor soul will have had the same experience.

Appendix I: Converting classes

Sometimes you are going to need to change the class of an R object in order to make do sensible things with it. For example, if you have a vector of 1s, 2s and 3s that represents which of three conditions in your experiment the participant had, then R will treat this as an ordinary numeric variable, which is not right. The general approach to coercing one class into another is that it involves as.something(). For example:

```
x=c(1, 1, 1, 2, 2, 2, 3, 3, 3)
x2=as.factor(x)
```

Now use x2 in your analysis and it will be treated correctly as a factor. An oddity: if you have a vector of integers v of class factor, and wish to treat it as a continuous numeric variable, then as.numeric(v) will almost certainly not do what you want. It will assign the lowest number in v to 1, the next lowest to 2, and so on. The conversion you want is acheived by:

```
v=as.numeric(as.character(v))
```

Just one of those strange things you pick up.

You can also convert other classes in similar ways, e.g. a data frame into a matrix via as.matrix(). And to make a data frame out of some vectors, it's just something like:

```
my.dataframe=data.frame(x1, x2, x3)
```

And if you wish name the columns:

```
colnames(my.dataframe)=c("Condition", "Latency", "Accuracy")
```

Appendix II: Library paths and default working directories

Contributed packages you download will be sent to a directory that got set up in the installation of R (not your current working directory). These packages contain many files and can be very large, and so you may want to control where they do (which by default will probably be in Documents somewhere, depending on the setup of your computer). To find out where your contributed packages are going, it is the first path returned by:

```
.libPaths()
```

```
## [1] "D:/Dropbox/R"
## [2] "C:/Program Files/R/R-3.3.3/library"
```

To set this to something else, just use:

```
.libPaths("C:/Users/Daniel/Dropbox/R")
```

I run a command like this at start up every time, and I use a Dropbox folder as my target directory. And all of my three computers point at the same folder. So any package I have ever installed on one of my computers is thereby automatically available on the other two (in the same version), without my having to do anything active about it.

At the point of using a package, you can also specify a non-standard place for R to look for it, like this:

```
library(psych, lib.loc="H:/My Documents")
```

Something you may wish to do is to preset R so that it automatically looks in all the places you want, for both packages and working directory. You do this with a .Rprofile file. This is a little text file saved in a place where R will look for it (e.g. Documents), that tells it what to do first when setting up. In my .Rprofile file (which must just be called .Rprofile with no extension), I set the default working directory, point to a

Dropbox folder for my contributed packages, load in some packages I use particularly often, and display a soothing message. Here it is:

```
First <- function(){
  .libPaths("C:/Users/Daniel/Dropbox/R")
  setwd("C:/Users/Daniel/Dropbox/R")
  library(ggplot2)
  library(dplyr)
  library(psych)
  library(tidyr)
  library(metafor)
  cat("\014")
  cat("\nWelcome Daniel on ", date(), "\n")
  cat("\nWhen we cut mere stones, we should be envisioning cathedrals.\n")
}
```

If you get hooked I am sure you can come up with one of your own.