Modelling and visualizing data using R: A practical introduction

Daniel Nettle

December 2018

Contents

Session 1: Introducing R **3**
- About this course 3
- Getting started and oriented 3
- First steps 4
- Functions and objects 6
- Writing scripts 8
- A first data set 9
- Contributed packages 15
- Version control 16
- Getting help 16

Session 2: Introducing the general linear model **17**
- Cutting through the forest of statistical tests 17
- The small world and the large world 18
- The general linear model with a bit more mathematical formality 24
- Interpreting the meaning of parameter estimates in a multivariate model 27
- The bit about p-values 31

Session 3: Plotting and interactions **33**
- A first ggplot2 figure 33
- Saving your figure 42
- Modifying the appearance of your figure 43
- A second figure and introduction to interactions 43
- Justifying the interaction terms you include 48
- What to report if you have an interaction 48

Session 4: Mixed and generalized models **50**
- Generalized linear models 50
- An example with a dichotomous outcome 51
- Other generalized families 55
- Mixed models: For when there is structure in your data set 55
- An example linear mixed model 58
- A note on data structure 60
- Generalized and mixed: The generalized linear mixed model 61

Session 5: ANOVA and factorial experiments **62**
- A dataset, and experimental terminology 62
- First steps with ANOVA 62
- Multi-way ANOVA: Exploiting your factorial design 65
- ANCOVA: Adding a covariate to your ANOVA 68
- ANOVA or not? 69
- ANOVA tables for within-subjects or repeated-measures situations 69

Session 6: Model selection, model averaging, and meta-analysis **72**
Model selection . . . 72
Estimating parameters using model averaging . . . 76
Taking it further with model selection and model averaging . . . 79
Meta-analysis . . . 79
Making a forest plot . . . 84
Difference of significance is not significance of difference . . . 86
Using meta-analysis to review the literature . . . 87

Session 7: R programming, writing simulations, and scripting tips **88**
Writing your own functions . . . 88
Simulating the power of an experiment . . . 89
Good practices for scripting . . . 96
Taking R further . . . 98

Appendix **99**
Converting classes and making dataframes . . . 99
Some tips for working with factors . . . 99
Default working directories and library paths . . . 100
Subsetting data frames . . . 101
Handling missing data . . . 101
Plotting a bar graph with error bars . . . 102
Reshaping data between wide and long formats . . . 104
Merging two or more dataframes . . . 105

Session 1: Introducing R

About this course

This course provides a basic, gentle introduction data analysis and visualization using R. It is designed to get you up to speed if you have had some kind of statistics training before, but not used R, or if you have done a small amount of R but not become confident. The emphasis is on getting on with analysing your data in a defensible way with a minimum of pain. Working through this course would provide a good pre-habilitation for taking on a higher-level R-based stats course such as Richard McElreath's *Statistical Rethinking* (CRC Press, 2015). The treatment of statistical mathematics here is brief to say the very least. Please forgive me this: the idea is to get people going as quickly as possible. Once they are on top of the material presented her, they can look up more rigorous treatments of the mathematics for themselves. There are plenty of good statistics books out there, many of them using R.

There are seven sessions (plus an appendix which is just useful bits and pieces there was no room for anywhere else). The idea is that you could work through each of the seven first sessions in a single sitting. So if you start on a Monday morning, you can be an R whizz by the end of the week (though, unlike The Creation, there is no rest on the seventh day). Please contact me if there are bits that do not work.

Getting started and oriented

Installation

We are going to interact with R through a programme called RStudio. RStudio 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 RStudio 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 RStudio from https://www.rstudio.com (the free RStudio Desktop Open Source Licence).

You then need to set up a folder for this course on your computer somewhere you will be able to find it easily. Save into it the eight data files (.csv files) and two script files (.r files) that come with this course. You can download these now from: https://www.danielnettle.org.uk/r-modelling/.

The RStudio screen

I assume you have installed R and RStudio. Open RStudio. 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. But be warned: at first you will see a lot of red error messages, and R being R, these will not make it very obvious exactly what you have typed wrong.

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 current 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. Repeat the calculations you did before, and note what the output is.

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

Make sense? (Maths nerds note that if you multiply the vector **x** by something here, for example itself, then you do not get vector multiplication, but 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
```

Note that in R, element numbering begins at 1, not 0. Thus **x[3]** denotes the third element of the vector **x**. Now try changing one element:

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

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

We can also apply some statistical functions to our vector. This is where you begin to see how R could be used for data analysis. 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 ask the following:

```
y = 10
y == 10
```

```
## [1] TRUE
```

R tells you that it is TRUE: **y** *is* equal to 10. If you had asked **y==9**, you would have got FALSE as the answer. 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 want, you 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).

Functions and objects

The things you care about in R will mostly be either **functions** or **objects**. Functions are the doing words, the verbs, and objects are the denoting words, the nouns. It's now time to say a bit more about functions and objects and how we use them.

Functions

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 objects are called the arguments of the function). For example:

```
mean(x)
```

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

This means do the function arithmetic mean to the argument vector **x**.

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. What if we wanted to know which elements of **x** were equal to the mean? Here we need that double equals again:

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

```
## integer(0)
```

There are no individual elements of **x** that are exactly equal to the mean. R tells us this in a characteristically opaque way!

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 (2018). 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 = {2018},
##     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

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. (I assume you still have objects **x** and **z** in your environment from earlier.)

```
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). RStudio 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'. 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 its class is 'character'.

There are objects of many classes in R. Your data variables are objects, usually of classes numeric, logical, character, factor or matrix. You can force R to convert an object from one class to another if needed (see Appendix, 'Converting classes and making dataframes').

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 graphs 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, combining them, comparing them, extracting values from 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 them.

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 appropriate 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. Let's 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, but 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? Now all the lines in the script 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(s) you want to run, and hit the 'Run' button. 'Control' plus 'Enter' ('Command' plus 'Enter' for Mac) when the cursor is in the script window also runs the part of the script where the cursor currently is, or which is currently selected.

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 more than 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] "C:/Users/Daniel/Dropbox/Fun with R/Version 2.0"
```

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

```
setwd("C:/Users/Daniel/Another directory")
```

Or whatever directory you want. It is tedious to type out a long path name. RStudio 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. You may want to set other start-up defaults for your R installation too. Information on how to do this can be found in Appendix, 'Default working directories and library paths'.

A first data set

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 original 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 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
```

```
##  [1] Male   Male   Female Male   Female Female Male   Female Female Female
## [11] Female Male   Female Female Female Female Male   Female Male   Female
## [21] Male   Female Male   Female Male   Female Male   Female Male   Female
## [31] Male   Female Female Female Female Male   Female Female Female Male
## [41] Female Male   Female Male   Male   Male   Male   Male   Female Male
## Levels: Female Male
```

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, ‘Some tips for working with factors’ for more advice on this. (For interest: factors are actually encoded as integers by R, with each integer corresponding to a level label.)

Note that you can also interrogate a data frame using row and column positions:

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

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

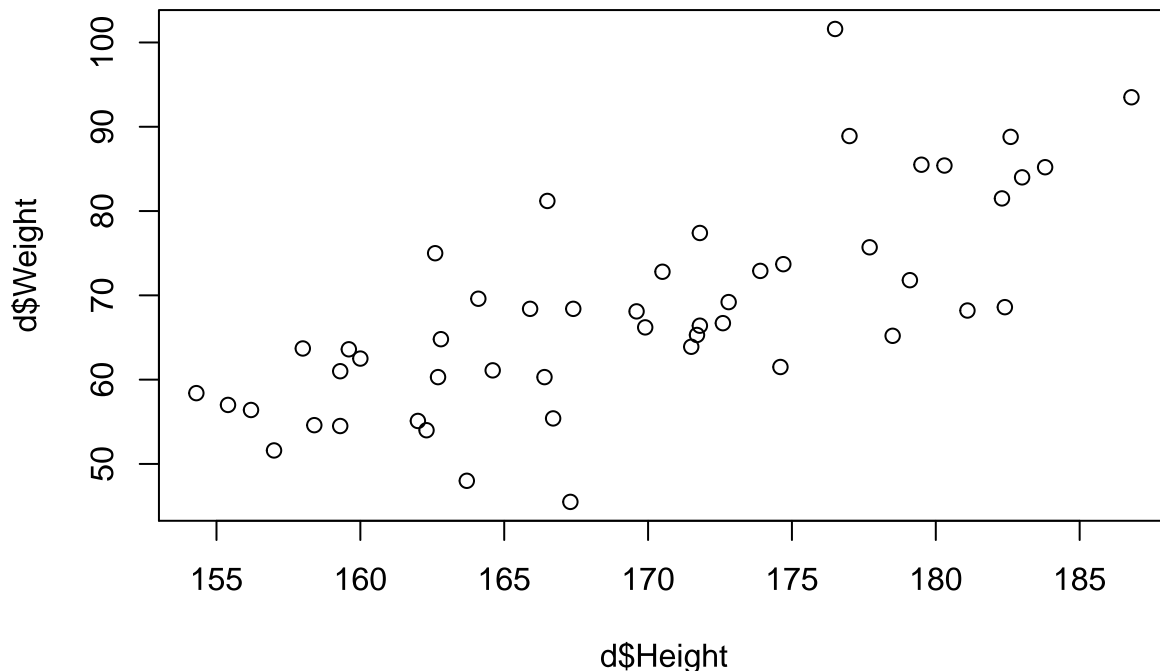
The seventh variable of the third case of the data set is ‘Female’.

If you have missing data in your .csv file, some issues can arise. For more on R's handling of missing data, see Appendix, 'Handling missing data'.

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 session 3.

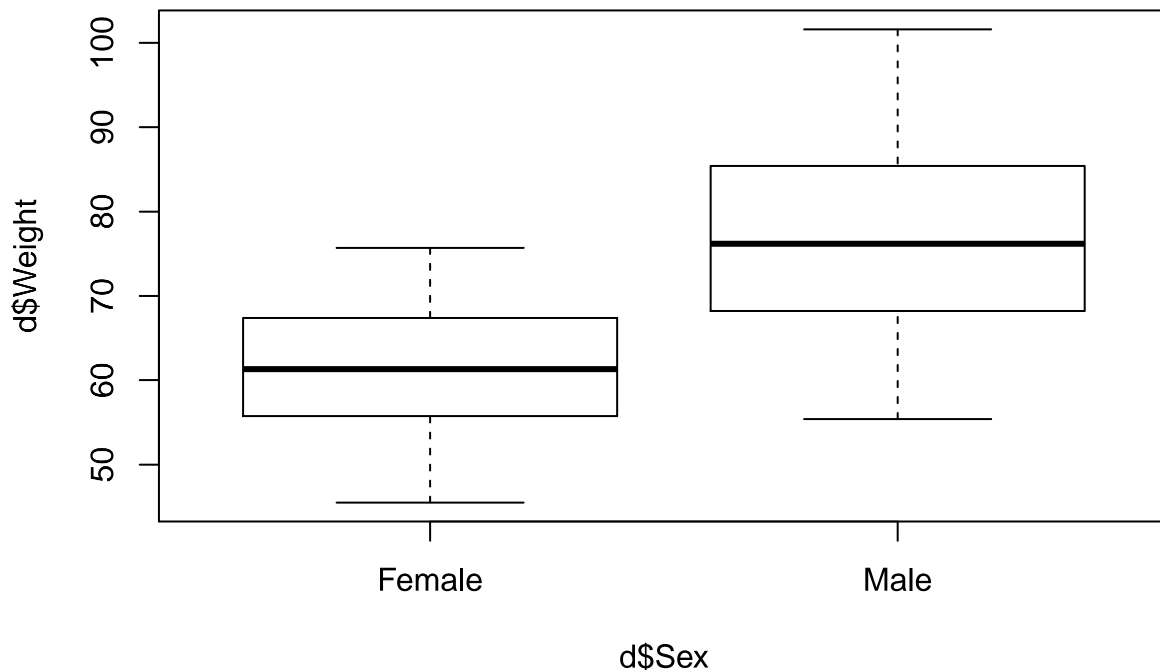
In the **plot()** command you just wrote, we encountered for the first time something we will often meet in R: the formula. An R formula is something on the left hand side, then a tilde symbol, then something on the right hand side, as in:

```
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. You can also make them bigger and copy them to the clipboard with the buttons shown there.

A first statistical analysis

Shall we do a test of whether we can infer that men are generally heavier than women? Let me introduce the **lm()** function. The **lm()** function will be your workhorse for much 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:

```r
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:

```r
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** (you could have called it anything you want) and assigns to it a general linear model of **Weight** by **Sex**. 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 now, 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. It is easy to do this. Let's create a new variable in data frame d called 'Sex2':

```r
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
```

Or how about:

```
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
```

Once you become adept at pulling out parts of a model, you can compare them to the corresponding parts of other models, write 'if…then' statements concerning them, plot them, and so on.

Piece of housekeeping. It is tedious and repetitive to have to write 'd$…' on each side of your formula. The **data** argument to the **lm()** function allows you to specify that all variables in the formula come from the data frame **d**, and hence avoid having to specify 'd$…' 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")
```

Various files will download and progress messages will be displayed. The files that make up the package will be saved. On how to control where in your computer they get saved to (it's not the working directory), see Appendix, 'Default working directories and library paths'.

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 the following:

```
library(psych)
```

Note that the package name is in inverted commas in **install.packages("psych")** but not in **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:

```
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 package 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 in 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.

Session 2: Introducing the general linear model

This session introduces our most basic workhorse of statistical modelling, the general linear model. Along the way, we need to consider a few issues like why it is we do statistical tests on our data, and what their output means. We are mostly going to work practically, with occasional asides about the philosophy and mathematics of what we are actually doing. We will also practice analyses on the weights dataset that we met briefly in session 1, so you will need the file 'weight.data.csv' saved in an appropriate directory.

First, change your working directory to where the weights data file is (**setwd()** function or 'Session' menu, remember). Then make a data frame called **d** (or anything else you like) with the weight data:

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

Cutting through the forest of statistical tests

If you have done an introductory statistics course before, one thing you probably took away from it is that there is a large number of different statistical tests you can do to understand the patterns in your data. You have to choose one of these according to the type of situation you have:

- One continuous outcome and two groups to compare, that's the t-test
- One continuous outcome, more than two groups to compare, that's ANOVA
- One continuous outcome, one continuous predictor, that's univariate regression
- One continuous outcome, more than one continuous predictors, that's multiple regression
- One continuous outcome, several groups to compare, one continuous covariate, that's ANCOVA
- and so on.

The philosophy I want to introduce here is rather different. In all of the cases described above, and others, you need to do the same thing: fit a general linear model. All of those different tests you learned about turn out to be just special cases of one, more general, procedure: the general linear model. That's because, in every case, you are trying to understand how the mean of the *outcome* or *response variable* changes as one or more *predictor variable(s)* varies. There are some differences in how you might report the results of your general linear model, according to the design of your study, but if you think you are doing a t-test, or an ANOVA, or multiple regression, then really you are just fitting a general linear model. This means that in all these cases, you can do what you need to do with one main function in R, the function **lm()**. We don't need to learn separate commands for t-test, ANOVA, ANCOVA and so on.

The general linear model can't do quite everything. It is not suitable where the outcome is a yes/no rather than a continuous measurement, or for certain types of count data. But it turns out that the general linear model in its turn is just a special case of something even broader, the generalized linear model. With the generalized linear model you can accommodate yes/no and count outcomes,and more besides, as well as the cases you are more familiar with. We will stick with the general linear model for now, but once you have mastered it, going generalized is easy (see session 4). (Confusingly, both the general linear model and generalized linear model are sometimes abbreviated GLM. The developers of the generalized linear model were said to have commented that they had not named it very wisely.)

Hold on, why do I need a model, all I want to know is whether my difference is significant!

When you ask people why they want to do a statistical test on some association or difference in their data, they will usually respond 'to find out whether it is significant!'. But they can't always shed much light on what this 'significance' means, except that something called the p-value tells you whether they have it or not - and a small p-value is somehow good, whilst a large one is somehow bad!

We need to step back a little. We need to understand why we do statistical tests (and by the way, all statistical testing involves fitting a model; a statistical test is one outcome of applying a dataset to a model). And that will help us get less focussed on p-values: these are something you can often get out of the process of modelling, but they are not the most important thing. They are not, or should not be, the only thing we do data modelling for. Indeed, these days I often write papers that contain no p-values at all, and one of the journals in my field has explicitly banned calling results 'significant' or not on the basis of a p-value!

The small world and the large world

Let's start from the simplest beginning. Often in science, you have some kind of dataset. For example, you might be interested in sex differences in height and weight in the US population, so you weigh and measure 50 men and women. You have measurements from a *small world* of those particular 50 people. If you wanted to merely describe your small world, then all you need is the tools of *descriptive statistics*. You can calculate the weights and heights of the men, and the women, in your sample, as well as measures of dispersion like the standard deviation, and you can report these. You can ask whether the men in your sample are heavier than the women, and, as long as you are only interested in the small world of these 50 people, you can know your answer exactly and with perfect confidence.

However, in reality, you are probably not interested just in those 50 people you have measured. The chances are, you are really interested in some *large world*, for example, the world of male and female Americans in general. In this large world, you want to know, are males generally heavier than females? The tricky situation you are in is that you want to be able to say something about the sex difference in weight *in the large world*, but all you have is the weights of the 50 people *in the small world*. You are exactly in the situation of an opinion pollster who wants to know how the whole electorate (a large world) will vote, but has only actually asked a few hundred people (a small world). The pollster knows exactly what the few hundred people said; what they can't be sure of is how that will generalize to the whole electorate.

What you need to do when you want to say something about a large world on the basis of data from a small world is to use *inferential statistics*, because you are going to need to make inferences about the (not directly observed) properties of the large world, on the basis of the small world you have directly looked at. And as well as *what* those inferences are, you are going to need to say something about *how precise* they are. Again this is just like the pollster who says, since I have only surveyed a few hundred people, there has to be a margin of error of plus or minus three percent on my projections for the whole electorate.

Inferential statistics is about parameter estimation

In this section, I want to introduce the idea that what we are trying to do when we do inferential statistics is estimate parameters in the large world, on the basis of the information we have from the small world of our sample. This links closely to the idea of setting up a model of the large world. A model is a schematic representation of how we think the large world works.

Let's work through the example of the weight data we started looking at in session 1. I am interested in whether men are heavier than women, in the large world of Americans in general. Let's set up a model of the large world in which I am interested. Let's say that in the large world, women have an average weight, which we will call β_0. So the average weight of women is given by:

$$E(w_f) = \beta_0$$

Now, men have an average weight which might or might not (depending on how my question ends up being answered) be different from women's average weight. We can capture this idea by saying that the average man's weight is the average woman's weight, plus another amount, which we will denote β_1. We are not prejudging the answer to the research question here. β_1 could turn out to be zero, if there was no difference between male and female weights. If men are actually lighter than women, β_1 would be negative. Whatever the answer to the question, we can nonetheless model men's average weight as:

$$E(w_m) = \beta_0 + \beta_1$$

Putting together the expressions for men's and women's weights, we can say that the average weight of a person will be:

$$E(w) = \beta_0 + \beta_1 * Male$$

Here $Male$ denotes a dichotomous variable that has the value 1 if they are male, and zero otherwise.

So now we have begun to sketch out a model of sex differences in weight in the large world. Our problem is: we don't actually know what the values of the parameters β_0 and β_1 are. We haven't measured all American males and females, and we are not going to be able to do so any time soon. So our task is: to estimate the large-world β_0 and β_1 using just the data we have from the small world of our sample.

Now it turns out, using some maths we won't go into, that if the only information I have to go on is the weights of the men and women in the small world of my sample, then the best possible estimate I can make of β_0 in the large world is just the average weight of the women in my sample; and the best possible estimate of β_1 that I can make is just the difference between the average male and average female weights in my sample. (What I say here is correct if you have no other information about the parameter valuess. If you have other, prior information about the parameter values, then you need Bayesian parameter estimation to optimally combine this prior information with the information in your sample. Bayesian parameter estimation is beyond our scope here.)

So when we fit a general linear model of weight by sex to our data, our estimates of the large-world parameters β_0 and β_1 will correspond to the descriptive statistics for male and female weight in the small world of the sample. To verify this, get the descriptive statistics of weight by sex for your sample (I assume here you installed the contributed package 'psych' in session 1; in not, you need **install.packages("psych")**):

```
library(psych)
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
```

You can see that the mean female weight is 61.65 kgs, and the mean male weight just over 15kgs more than this at 76.92kgs. Now fit a linear model to the data and summarise it:

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

```
##
## Call:
## lm(formula = Weight ~ Sex, data = d)
##
## 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 ***
## 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
```

You can see that we have a column called 'Estimate'. There is an estimate for something called the intercept, and for something called SexMale. These are our best estimates of β_0 and β_1 respectively, and you can see that they are numerically identical to the mean female weight, and the difference between the mean male and female weights, respectively. So that is all straightforward.

Bringing in uncertainty

We are not quite done with estimating parameters. Although we have made our best possible estimates of what the large-world parameters β_0 and β_1 are, we have to acknowledge that there is some margin of uncertainty around them. The uncertainty arises because we only measured the people from one particular small world: if we had sampled a different 50 individuals, the estimates could have been rather different. When we fit an **lm()**, R gives us, again through some maths I won't go into, a standard error for each parameter estimate, as well as the estimate itself. The standard error gives an idea of the *precision* of the estimate. A way to think about the standard error of the parameter estimate is: if I were to take a second, different sample of 50 people from the large world, how far away might my next estimate of that parameter typically be? In our case, the standard error of 2.716kgs tell us that if we were to sample again from the population, we might typically get an estimate of the male-female weight difference that differed by 2 or 3 kgs either way from the 15.268 we got this time. That's quite reassuring: it says that the male-female weight difference, if we measured it again, might come out at 13 kgs, or 17 kgs, but probably not 0kgs or 40kgs.

An alternative and related measure of precision to the standard error of the parameter estimate is the 95% confidence interval. This is an estimate of that interval within which, were you to sample 100 times from the big world, 95 of your samples would fall. To get your 95% confidence intervals from R, try (assuming here that you have already fitted a model called **m**):

```
confint(m)
```

```
##                 2.5 %   97.5 %
## (Intercept) 58.027060 65.27294
## SexMale      9.806394 20.72997
```

What this tells us is that, on the basis of the small-world evidence we have, if we were to take samples like this one from the broader population 100 times, about 95 times out of 100, the average female weight would be between 58 and 65 kgs, and the difference between the average male and weights would fall between about 10kgs and about 21kgs. If you are wondering, those intervals are calculated straightforwardly from the standard errors of the parameter estimates, using the fact that in a normal distribution, 95% of values lie within 1.96 standard deviations either side of the mean.

Note: the 95% is purely conventional. It could be 90%, or 75%, or 99% or whatever. Richard McElreath suggests drawing attention to the arbitrariness by presenting 89% confidence intervals. When someone asks you why you presented 89% confidence intervals rather than 95%, you should say, with a meaningful look, 'Well, 89 is a prime number'. You get 89% confidence intervals with:

```
confint(m, level = 0.89)
```

```
##                5.5 %   94.5 %
## (Intercept) 58.71592 64.58408
## SexMale     10.84488 19.69148
```

The outcomes that matter most from fitting a statistical model are: the parameter estimates (what do I believe, on the basis of the small world of my data, is the most likely value of the thing in the large world that I am interested in?), and the precision of parameter estimates (by how much could I be wrong about the values in the large world?).

The precision of your parameter estimates is determined by two things:

1. The size of your sample. Other things being equal, as your sample gets bigger, your precision gets better, and the standard error of your parameter estimates gets smaller. In the limit when your sample consists of the whole population, then you have perfect precision, and your standard error is zero.
2. The variability in the population. If all men in your population were exactly the same weight as one another, and all women were exactly the same weight as one another, then you would only need to weigh one woman and one man in order to know the sex difference in weight with perfect precision. But of course this is not true: since there is variation *within* the women and *within* the men, any small sample does not capture the true sex difference with perfect precision.

Because parameter estimates and their precision are the most important outcomes of your inferential statistics, they should be the most prominent things that you report. For example, when reporting the present analysis, you might say "the parameter estimate for being male rather than female was 15.27 kg (s.e. 2.72)", or equivalently that "the parameter estimate for being male rather than female was 15.27 kg (95% confidence interval 9.81 - 20.73)". Either of these statements captures the key information.

What about if my predictors are continuous?

So far we have worked entirely with an example where the predictor variable, sex, is dichotomous (either Male or Female). What if the predictor is a continuous measure?

In fact the logic is exactly the same. For example, let's look at the same dataset and consider how someone's height predicts their weight. We set up the model of the large world in exactly the same way. A person's expected weight is going to be:

$$E(w) = \beta_0 + \beta_1 * Height$$

And we can run a general linear model in very similar fashion to before (let's call it **h** this time):

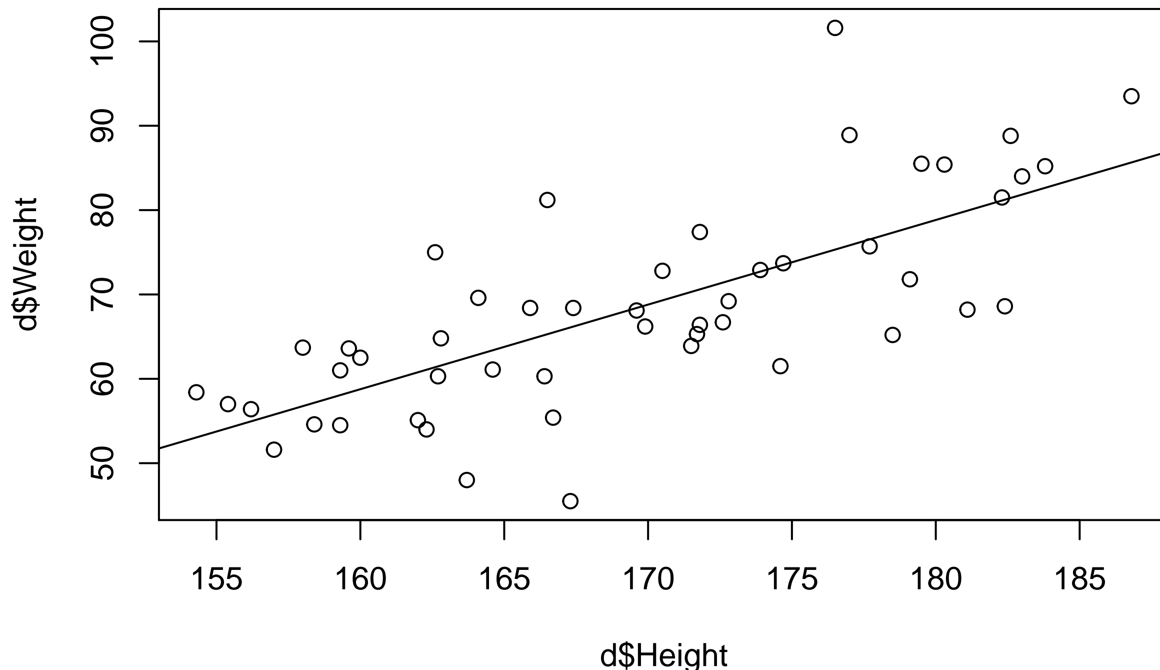
```
h=lm(Weight~Height, data=d)
summary(h)
```

```
##
## Call:
## lm(formula = Weight ~ Height, data = d)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -20.5910  -4.8430   0.1875   4.8572  26.2808
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -101.722     23.256  -4.374 6.54e-05 ***
## Height         1.003      0.137   7.323 2.35e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8.438 on 48 degrees of freedom
## Multiple R-squared:  0.5277, Adjusted R-squared:  0.5179
## F-statistic: 53.63 on 1 and 48 DF,  p-value: 2.352e-09
```

The only slight difference from the case we already encountered is in how β_0 and β_1 should be interpreted. β_1 here is 'the increase in weight for every unit increase in height'. In our dataset, height is measured in cms, so the parameter estimate tells us that weight goes up almost exactly 1kg for every additional cm of height. Of course, this parameter could in principle have had a zero value, if taller people were no heavier than shorter ones, or even a negative value, if taller people were lighter than shorter ones. However, its observed value, as you might expect, is in fact positive.

In the previous example of sex, the interpretation of β_0 was the average height of a woman (i.e. of a person for whom the sex Male had the value of zero.) In the present case, β_0 is negative! So what on earth could it mean? One way we can get an idea is to plot the weights against the heights with a straight line fitted through (don't worry too much about the code for plotting here, since we will do plotting in more detail in session 3):

```
plot(d$Weight~d$Height)
abline(lm(Weight~Height, data=d))
```



If we followed that straight line right down to a height of zero cms, it would strike the vertical axis at a weight of around -100 kgs. So that's what β_0 represents: the predicted outcome at the extrapolation point where the predictor variable (height) is equal to zero. That's why it is called the intercept. In this particular case, the predictor having a value of zero is not biologically possible (you could not be 0 cms tall), but this is not something that R can be expected to know. It just means you need to be sensible in how you interpret the value of this parameter.

You can make the intercept parameter more meaningful by centering your predictors. To centre a predictor means to set its mean to zero (so that someone with a height 1 cm below the mean height gets -1, and someone who height is 3 cm taller than the mean height gets +3, and the person of average height gets 0). You can do this with R as follows:

```r
d$Height.centred=d$Height - mean(d$Height)
h2=lm(Weight~Height.centred, data=d)
summary(h2)
```

```
##
## Call:
## lm(formula = Weight ~ Height.centred, data = d)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -20.5910  -4.8430   0.1875   4.8572  26.2808
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)       68.368      1.193  57.295  < 2e-16 ***
## Height.centred     1.003      0.137   7.323 2.35e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8.438 on 48 degrees of freedom
## Multiple R-squared:  0.5277, Adjusted R-squared:  0.5179
## F-statistic: 53.63 on 1 and 48 DF,  p-value: 2.352e-09
```

The first line subtracts the mean height from everyone's height, thus centering values on zero. The output now makes a lot more sense. Since centered height is zero at the mean height of the sample, β_0 now represents the mean weight (about 68 kgs), and β_1, as before, tells you that for every cm taller, weights tend to be about 1kg heavier.

If you have more than one predictor, the logic is exactly the same

Our examples so far have both been **univariate models**. That is, we have modelled how the response variable (Weight) changes as one predictor variable (Sex in the first case, Height in the second) varies. Often, though, you want to see how your response variables changes as multiple predictors vary. This is called **multivariate** analysis. The basic mechanics are just the same, except that your formula now has several predictors on the right-hand side, separated by addition signs.

```r
m2=lm(Weight~Sex + Height.centred, data=d)
summary(m2)
```

```
##
## Call:
## lm(formula = Weight ~ Sex + Height.centred, data = d)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -18.1549  -4.4810   0.8562   4.6440  24.1804
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)      65.3700     1.7446  37.470  < 2e-16 ***
## SexMale           6.8136     2.9926   2.277   0.0274 *
## Height.centred    0.7556     0.1705   4.431 5.59e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 8.092 on 47 degrees of freedom
## Multiple R-squared:  0.5746, Adjusted R-squared:  0.5565
## F-statistic: 31.75 on 2 and 47 DF,  p-value: 1.888e-09
```

There are some subtleties of intepretation required in what these parameters mean. You will notice, for example, that the parameter estimate for Sex in **m2** is not the same as the parameter estimate for Sex in **m**; it is smaller. We will come back to this later. The basic point here is that you can add more predictors into the right-hand side and the corresponding parameters will be simultaneously estimated. Now, though, having got this far, we need to be a little bit more mathematically formal.

The general linear model with a bit more mathematical formality

It's time to review a bit formally what we are assuming when we fit a general linear model. R is doing some calculations behind the scenes, and these are based on assumptions. Formally, the general linear model assumes that, in the large world, the value of the response variable for the ith individual is related to the value of all the predictors for that individual, times the respective parameters describing how a one-unit change in that predictor affects the response variable, plus some random error. That is, for the case where there are three predictors:

$$y_i = \beta_0 + \beta_1 * x1_i + \beta_2 * x2_i + \beta_3 * x3_i + \epsilon_i$$

The $\beta_0 + \beta_1 * x1_i + \beta_2 * x2_i + \beta_3 * x3_i$ part is called the 'linear predictor'. Note by the way that the term 'linear' here does not mean a general linear model can only capture linear relationships between predictors and response variables. Not at all. Your predictor variables could be squared, cubed, raised to the power ten, or logarithmically transformed. The 'linear' in linear predictor merely means that the parameters β_0, β_1, once weighted by the predictor variables, combine in a linear way: they are added up, rather than for example being multiplied by one another. You can however capture some kinds of non-additive combinations using interaction terms (see session 3).

The other part of the model on the right-hand side is the error term, ϵ. What does the error term represent? Well, we acknowledge that the model does not fit the data perfectly. Real individuals will be a bit higher or lower than the corresponding value of the linear predictor, even if the model is right on average. The ϵ thus have an average of zero, but will be more or less than zero for particular individuals. The variation in the values of ϵ is called the 'residual variation' or 'error variation'. The better your predictor variables explain the variation in the response variable, the smaller this residual variation is going to be.

You can work out the value of ϵ for any individual, as the difference between the response value they have and the response value the model says they should have. For example, try:

```
m$residuals
```

```
##           1           2           3           4           5           6
##   4.5818182  11.8818182 -16.1500000   8.4818182  11.1500000  14.0500000
##           7           8           9          10          11          12
## -11.7181818  -0.6500000  -1.3500000   3.1500000  -4.6500000  24.6818182
##          13          14          15          16          17          18
##  -0.5500000  -6.5500000  -5.2500000  -7.0500000  -7.7181818 -13.6500000
##          19          20          21          22          23          24
##   8.5818182   4.5500000  11.9818182   4.7500000   8.2818182   6.7500000
##          25          26          27          28          29          30
##  -5.1181818  -3.2500000   4.2818182   5.0500000  -1.9181818  -1.3500000
##          31          32          33          34          35          36
##   7.0818182  -7.1500000  12.0500000  -0.1500000   7.9500000 -11.6181818
##          37          38          39          40          41          42
##  -7.6500000   0.8500000   6.7500000 -13.0181818   1.9500000 -21.5181818
##          43          44          45          46          47          48
```

```
##   6.4500000 -13.2181818  -4.0181818  16.5818182  -8.3181818   0.4818182
##          49          50
## -10.0500000  -8.7181818
```

This tells us that the first person in the dataset is about 4.58 kgs heavier than we would have predicted for their sex, the 3rd person is about 16.15 kgs lighter than we would have predicted, and so on.

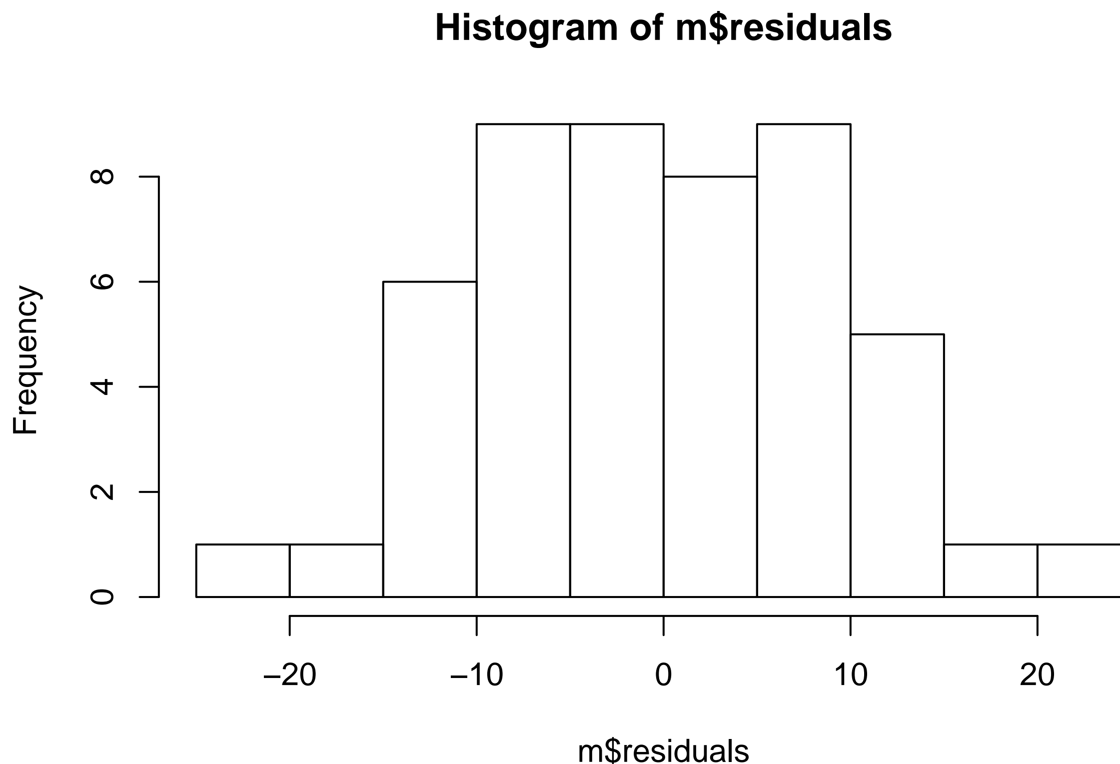
Some of the maths behind the general linear model is based on the assumption that in the large world, the errors ϵ are normally distributed (that means they have a bell-shaped distribution). Note that this is not the same as your data variables being normally distributed: your variables can have all kinds of distributions. The maths just assumes that the difference between the actual response and the best possible linear combination of all the predictors has a normal distribution. Formally, then, we say:

$$\epsilon \sim N(0, \sigma^2)$$

This just means the ϵ follow a normal distribution with a mean of 0 (since overall they are just as likely to be negative as positive), but with a constant variance σ^2. We don't know the value of σ^2, since it is a parameter of the large world, but we can estimate it from our data.

We can try to assess whether the assumption of residual normality might be reasonable by looking at the distibution of our observed residuals, using the histogram function **hist()**, as follows:

```
hist(m$residuals)
```



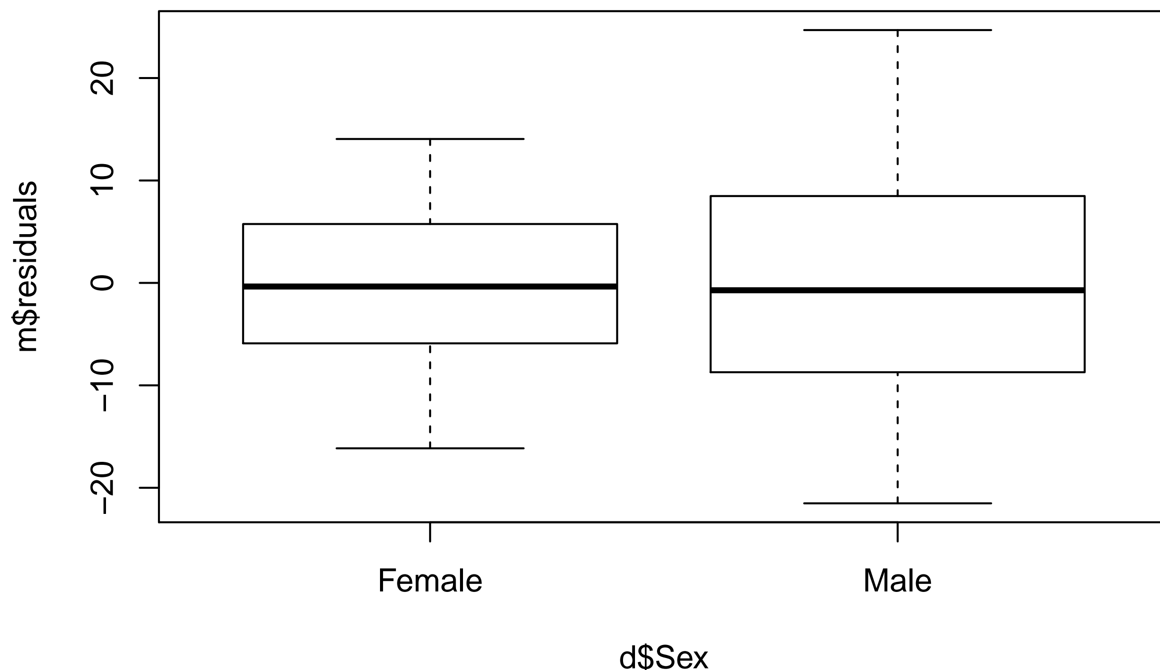
I'll settle for that. You wouldn't expect it to be a perfect bell curve, since we have only sampled 50 points, but it's pretty symmetrical and the right kind of shape. I would only worry if I saw some hugely asymmetrical distribution here. So the assumption of residual normality seems appropriate for this case, and we can have some confidence in our parameter estimates. By the way, some people are keen on doing formal 'tests' of normality on their data, and claiming the assumptions to have been violated if these tests give a p-value less than 0.05. I don't recommend this, for three reasons:

1. it is not actually the data that need to be normally distributed, but the errors;
2. there are respectable points of view that even the errors don't need to be exactly normally distributed, as long as their distribution is symmetric and there is no major violation of homoscedasticity (see below);
3. the p-value of a test for normality is largely a function of your sample size - if your sample is big enough, the test will always become significant - and doesn't tell you much about either the nature or the magnitude of any departure from normality.

The other key assumption of the general linear model is that the variance of the ϵ is the same regardless of the values of the predictor variables. So, for example, in model **m** we are assuming that male weights differ from the average male weight about as much as female weights differ from the average female weight. This is called the assumption of constant variance or homoscedasticity; where it is violated you have heteroscedasticity. Where the assumption of constant variance is violated, your parameter estimates are still alright, but your estimates of their standard error are not, so you are in danger of reaching incorrect conclusions about how precise your knowledge of the large world is.

How can we check whether the assumption of constant variance is reasonable? For the case of model **m**, it is very simple, we just look at the variance of the residuals by sex.

```
plot(m$residuals~d$Sex)
```

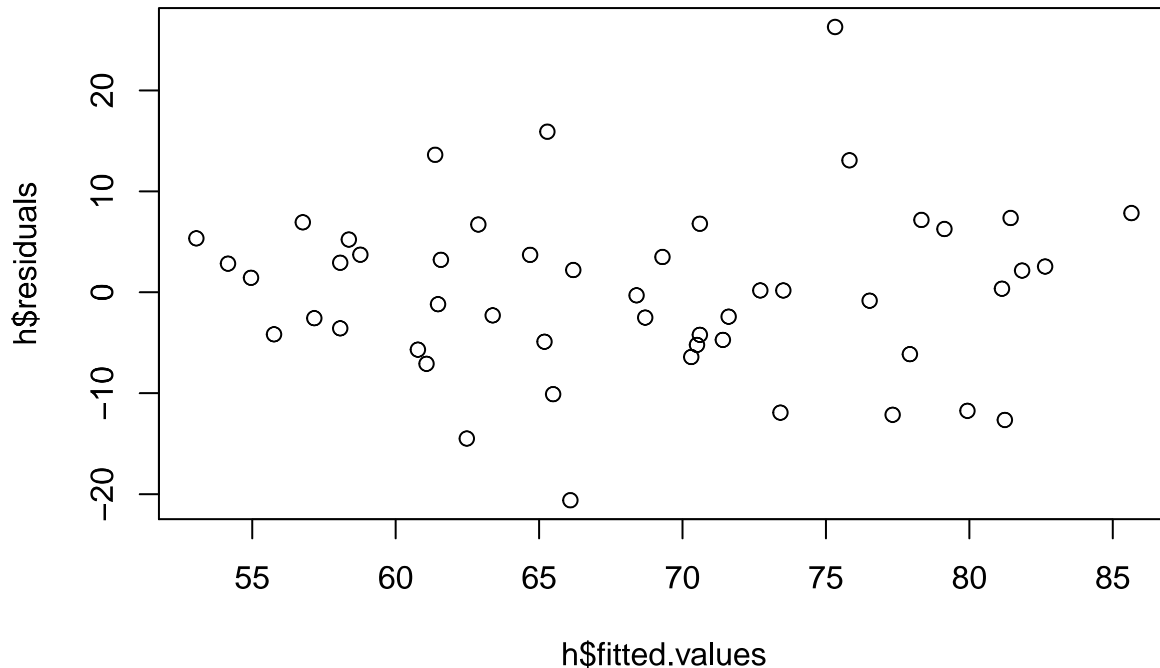


There's at least a hint there that the variability in male weight about its mean might be larger than that the variability in female weight about its mean, which would violate the assumption of constant variance. It's not too dramatic, so we will not be unduly concerned, but we will return shortly to what we might do in such a situation if we were worried.

When your model is more complicated than just one dichotomous predictor, then the way to check the assumption of constant variance is to plot the residuals against the fitted values. (The fitted values are the

values of the linear predictor for each case. The residuals are the differences between the fitted values and the actual value of the response.) So for model **h** for example, this plot looks like this:

```
plot(h$residuals~h$fitted.values)
```



What you are hoping to see here is the absence of any overall pattern. Ideally, the residuals should be equally variable about zero at low fitted values and at high fitted values. If they look like they are narrowly bunched at one end and spreading out towards the other (a 'shotgun' or 'beam of light' pattern), then you might have an issue of heteroscedasticity.

What should I do if the assumptions are violated?

If your residual plots show the assumptions are badly violated, do not fret. I would explore transforming your data; you can usually improve problems of asymmetrically distributed residuals or heteroscedasticity by choosing an appropriate transformation of the response variable. Taking the logarithm (R function **log()** gives you natural logarithms) will tend to correct the situation where some positive residuals are very large, and/or the residual variation tends to get larger as the fitted values get larger. Other transformations include the reciprocal and the square root: keep trying until you improve the distribution of the residuals. If your data are counts with a lot of zeros, or take on only a small number of different values, then you may need a generalized linear model, which we will meet in session 4.

Interpreting the meaning of parameter estimates in a multivariate model

Let us return to model **m2**, where we predicted **Weight** from both **Height** and **Sex** simultaneously.

```r
m2=lm(Weight~Sex + Height.centred, data=d)
```

We are going to compare this to the output of model **m**, where Sex alone was the predictor.

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

Look at the parameter estimates from **m** and **m2** respectively:

```r
summary(m)
```

```
##
## Call:
## lm(formula = Weight ~ Sex, data = d)
##
## 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 ***
## 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
```

```r
summary(m2)
```

```
##
## Call:
## lm(formula = Weight ~ Sex + Height.centred, data = d)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -18.1549  -4.4810   0.8562   4.6440  24.1804
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)     65.3700     1.7446  37.470  < 2e-16 ***
## SexMale          6.8136     2.9926   2.277   0.0274 *
## Height.centred   0.7556     0.1705   4.431 5.59e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8.092 on 47 degrees of freedom
## Multiple R-squared:  0.5746, Adjusted R-squared:  0.5565
## F-statistic: 31.75 on 2 and 47 DF,  p-value: 1.888e-09
```

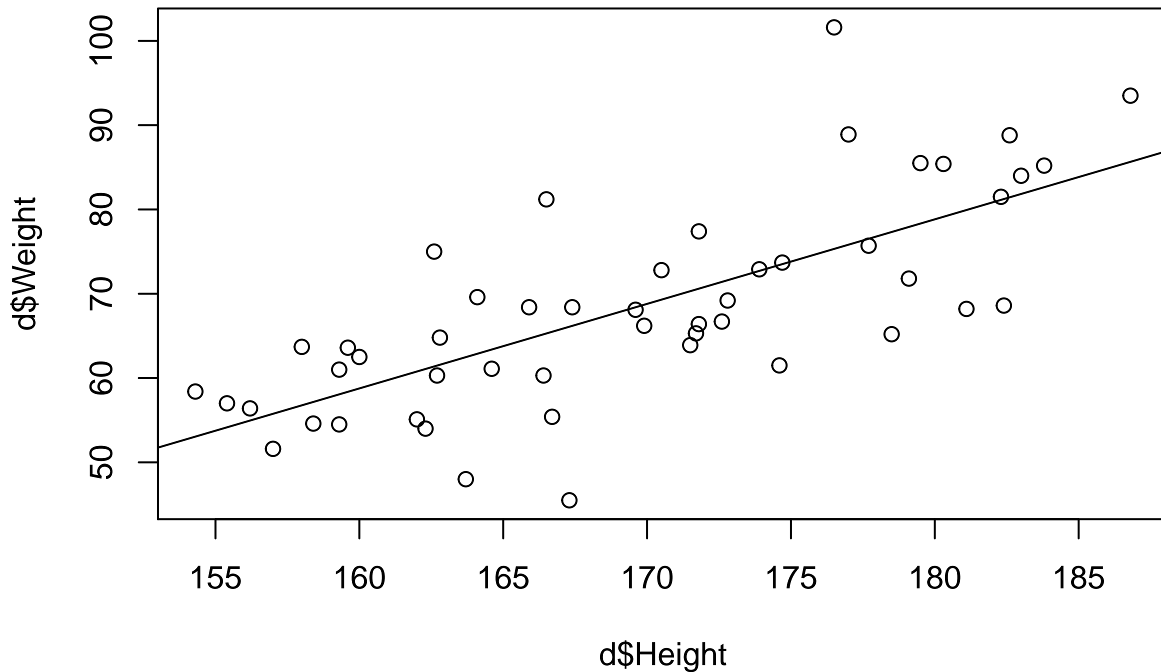
Model **m** seems to say that our best estimate of the weight difference between a male and a female is 15 kgs, whilst model **m2** seems to say that it is less than 7 kgs. What is going on? Is the estimate from **m2** somehow better because we have taken more other factors into account?

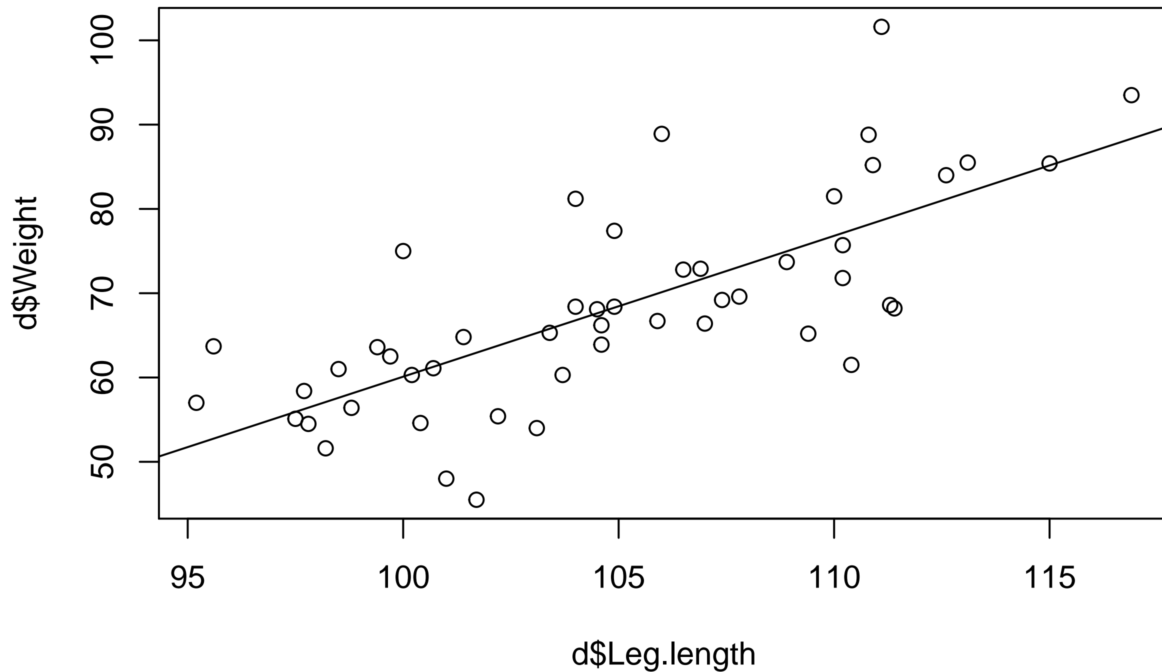
It is crucial to understand that when you add more predictors to your model, you change the meaning of the existing predictors' parameter estimates. Parameters in a general linear model are what is known as partial;

that is, they represent the net influence of that particular predictor on the response, when all the other predictors in the model are held constant. In model **m**, β_1 thus represents 'the weight difference between a male and a female about whom we have no other information'. In model 2, β_1 represents 'the weight difference between a male and a female whose heights have been mathematically equalised'. Thus, 15 kg is the answer to the question 'How much more does a man weigh than a woman on average?', whereas 7 kg is the answer to the question 'How much more does a man weigh than a woman on average when the man and the woman in question are the same height?'. I hope you can see that these are just different questions; it is not that one model is better than the other, just that you use them for different purposes.

I have insisted on this point because people have strange intuitions that your result is 'better', or 'more accurate' if you control for more things by putting extra predictors into your model. It is not; you have simply asked a different question. It should be your research question that dictates the structure of your model, not a general desire to 'control for more things'. In fact, controlling for more predictors can get you into real trouble if you have not thought about what information those predictors contain.

For example, our data set contains a variable called **Leg.length**. Let's look at plots of how **Weight** relates to **Height**, and how **Weight** relates to **Leg.length**.





It looks like **Weight** increases with greater **Height** (as we already know), and Weight increases with greater **Leg.length** (no surprise). So let's put them both into a model. Surely that model is going to be *really* good at predicting weight, with two predictors in it, both of which seem to be linked to the response.

```
l= lm(Weight ~ Height + Leg.length, data=d)
summary(l)
```

```
##
## Call:
## lm(formula = Weight ~ Height + Leg.length, data = d)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -18.632  -4.534   0.065   3.795  24.074
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -109.9894    23.5525  -4.670 2.54e-05 ***
## Height         0.4310     0.3955   1.090    0.281
## Leg.length     1.0030     0.6518   1.539    0.131
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8.32 on 47 degrees of freedom
## Multiple R-squared:  0.5504, Adjusted R-squared:  0.5312
## F-statistic: 28.76 on 2 and 47 DF,  p-value: 6.954e-09
```

```r
confint(l)
```

```
##                   2.5 %     97.5 %
## (Intercept) -157.3709584 -62.607794
## Height        -0.3646205   1.226701
## Leg.length    -0.3082818   2.314339
```

Here's the strange thing. In model **l**, *neither* **Height** nor **Leg.length** seems to be important in predicting Weight (both have confidence intervals that include zero). Yet we know that *both* of them are associated with Weight, from the plots we have just seen.

To understand the issue, recall that parameters in a general linear model are partial: they tell you how much that predictor explains variation in the response after controlling for the other predictors in the model. **Height** and **Leg.length** both explain variation in weight. It's just that they explain *the same* variation in weight, because tall people also have long legs. So when we put them both into the model, we are asking: how much better can you predict someone's weight by knowing their height, given that you already knew their leg length, and how much better can you predict someone's weight by knowing their leg length, given than you already knew their height. And the answer is: no better, because the same information is in leg length and height. The partial information of both is minimal, but the partial information is not the same as the total information. (You can see this, by the way, in the **summary(l)**. The R-squared of the model, which is the proportion of the variation in the outcome that the overall model explains, it pretty decent at 0.53. It's just that neither of the predictors has much partial predictive power.)

For this reason, you should be very careful about what predictors you put into your model: have the ones specified as important by your research question; don't have multiple ones that are capturing the same information; keep it as simple as sensible; and interpret your parameter estimates appropriately.

The bit about p-values

We are going to finish with a little homily about p-values. Everyone is obsessed with getting p-values out of their model, and that's often the only inferential statistic that people actually report. This is bad. The statistic known as the p-value has its applications, but it is never sufficient information in terms of the answer to your research question, and it is often completely irrelevant.

The p-value is the probability that you would have obtained a parameter estimate as large as you did if the large-world parameter value was actually zero. So let's look at model **m**:

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

```
## 
## Call:
## lm(formula = Weight ~ Sex, data = d)
## 
## 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 ***
## 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
```

So we see the probability of obtaining, in the small world of our 50 people, a difference in average male and female weights of 15 kgs if in fact, in the large world of the whole population, the difference between the average male and female weights was zero, is very small indeed ('9.47e-07' means $9.47 \times 10^{-}7$, i.e. 0.000000947). So people would conventionally go on to say: we can therefore reject the hypothesis that male and female Americans do not differ in weight.

That's all fine. However, it is important to be aware that the p-value is a function of three things:

1. The magnitude of the actual difference in average weights between men and women;
2. The variability in weight within men, and within women;
3. The size of your sample.

From the p-value being small, you cannot therefore reconstruct whether the real difference is a major one, or whether there is just rather little variability within each sex, or whether the sample size is just large (if you sample 10,000 individuals, even the most trivial differences will give you a tiny p-value). So at the very least, readers are going to want to know the magnitude of your parameter estimate, alongside any p-value. Just giving them the p-value tells them almost nothing about what they should believe about the phenomenon of interest.

It is actually worse than this, because often, our research question is not about whether some parameter is zero or not. Say I am a bicycle manufacturer, and I want to know whether I can use the same components for male bikes and female bikes. I will get away with it as long as male weights are within, say, about 20kgs of female ones. So I commission a researcher who comes back and says: 'I looked at it thoroughly, and I can tell you that males are heavier than females ($p < 0.0001$)'. This is entirely useless to me. I never thought the weight difference between males and females might be zero. I want to know *how big it is.*

The most important inferential statistics are your parameter estimate itself (how much heavier are men?); its precision (if I were to sample again, how much might this estimate change?); and some sense of the variability in the response (how much variation is there within men, compared to the male-female difference?). Never report a bare p-value without other information, and never report one at all unless it is relevant to the question whether the parameter in question might be zero or not.

Since I find students are very p-focussed, we will end with some examples. In each case, have a think about whether reporting a p-value actually answers the research question.

- You are asked to investigate whether cancer treatment A, which costs $10,000 a month, is a worthwhile investment compared to treatment B, which costs $50 a month. You report back: The average survival of patients receiving treatment A was longer than those receiving treatment B ($p < 0.01$). Does this answer the question?
- There are two theories about the growth patterns of trees. One theory says that the breadth of the canopy should scale with the height of the tree to the power of 2/3. The other theory says that the breadth of the canopy should scale with the height of the tree to the power of 3/4. You measure breadth and height of 100 trees and report: 'There was a positive association between canopy breadth and height ($p < 0.05$)'. Does this address the research question?
- Your theory about gender roles says that parental values are the major influence on how traditional a person's conception of gender is. You survey 5000 people and their parents, and report: 'Parental values predicted gender traditionalism ($p < 0.0001$)'. Is this a test of your theory?

Session 3: Plotting and interactions

In this session, we will be exploring the basics of R's capability for producing lovely production-quality graphs. In tandem, we will also explore the topic of interaction terms in general linear models, when you should fit them, and what they mean.

A new dataset

We are going to use a new data set for this session. It is a dataset about telomeres in young starlings. Telomeres are the DNA caps on the ends of chromosomes that shorten with age. They shorten particularly fast in early life, and may shorten faster in some individuals than others. In this dataset, we blood sampled 89 starling chicks and measured the change in telomere length between hatching and two weeks old (**Telo.change**). What we want to do is understand whether this varies by the type of brood (small, i.e. 5 or fewer chicks; or large, i.e. 6 chicks), and by the chick's growth rate, which we estimate by their weight on day 7 of life.

You will need the file 'star.data.csv' saved in an appropriate directory. Set the working directory to the correct place using the 'Session > Set Working Directory' menu in RStudio, then:

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

There is also an R script to go with this session ('session3.r'). I suggest you have it open as you work through. Rather than having to type out all the code for each example, you can just select the appropriate bit of the script and run it. I will identify the various chunks of code as we go along.

Installing ggplot2

The R base installation has graphics capabilities (using functions like **plot()** and **barplot()**), but if you want a very beautiful, flexible language for making your figures, it's best to use a contributed package called 'ggplot2'. So let's start by installing it (you will need to be connnected to the internet):

```
install.packages("ggplot2")
```

Once you have done this (or if you already installed it previously), then we just need to load it in to the current session.

```
library(ggplot2)
```

Now we are ready to make our first figures.

A first ggplot2 figure

First let's look at what variables we have in our data frame **d**.

```
head(d)
```

```
##    X Brood.type Number.chicks Sex    Farm Telo.change Weight
## 1  1      Small             5   m Whittle   0.2449029   52.5
## 2  2      Small             5   f Whittle   0.3810148   51.0
## 3  4      Small             5   f Whittle   0.2560100   47.5
## 4  5      Small             5   f Whittle   0.1794750   41.7
## 5  8      Large             6   m Whittle   0.3661215   48.3
## 6 11      Large             6   f Whittle  -0.6420882   35.1
```

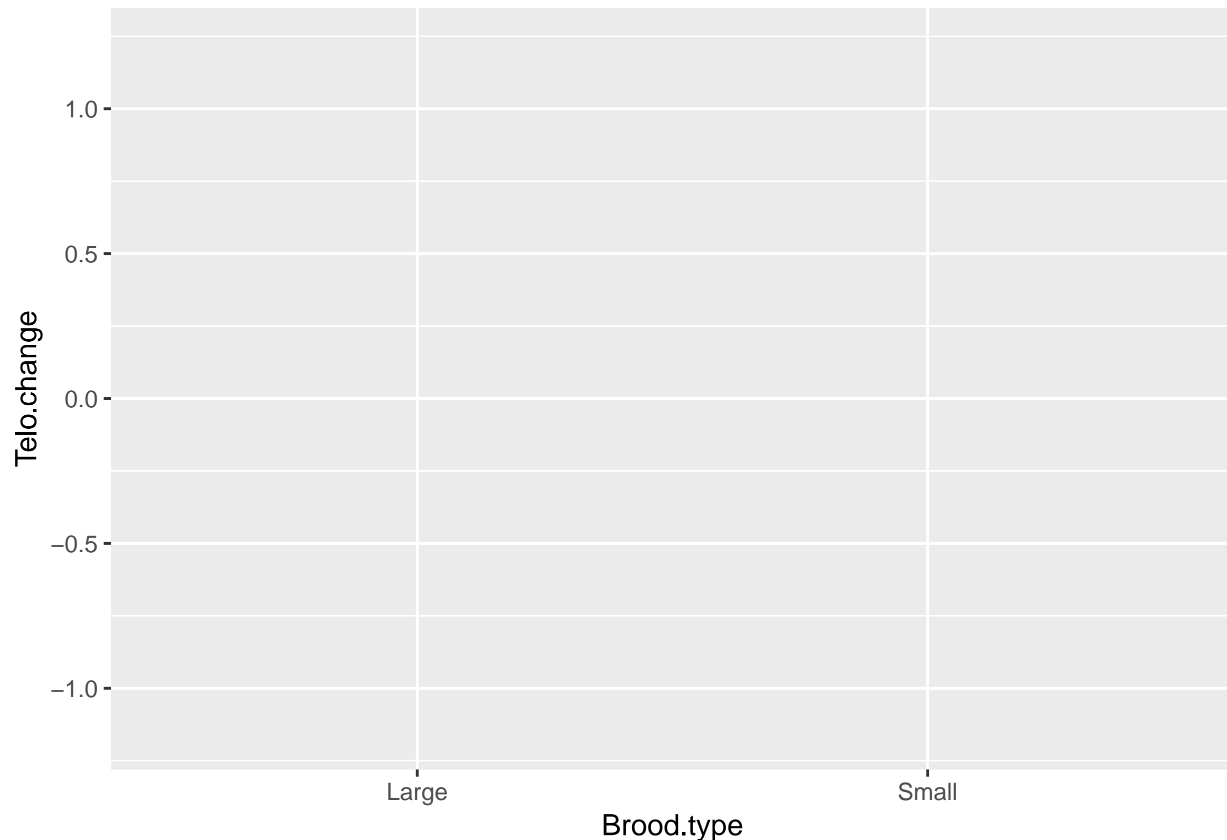
The essence of ggplot2 is that you build up your figure from a particular data frame, in layers. At the heart of your figure lie something called the *aesthetics*, which are not what you might think they are. The aesthetics are mappings between the data frame you are working from, and the figure you are building up. So let's say

we want to plot, from data frame **d**, the variable telomere change (**Telo.change**) against whether the brood is Small or Large. Our aesthetics are then going to be that the variable **Brood.type** in the data frame maps onto the x-axis of the figure, and the variable **Telo.change** maps onto the y-axis of the figure. So let's define our figure:

```
fig1 = ggplot(d, aes(x=Brood.type, y=Telo.change))
```

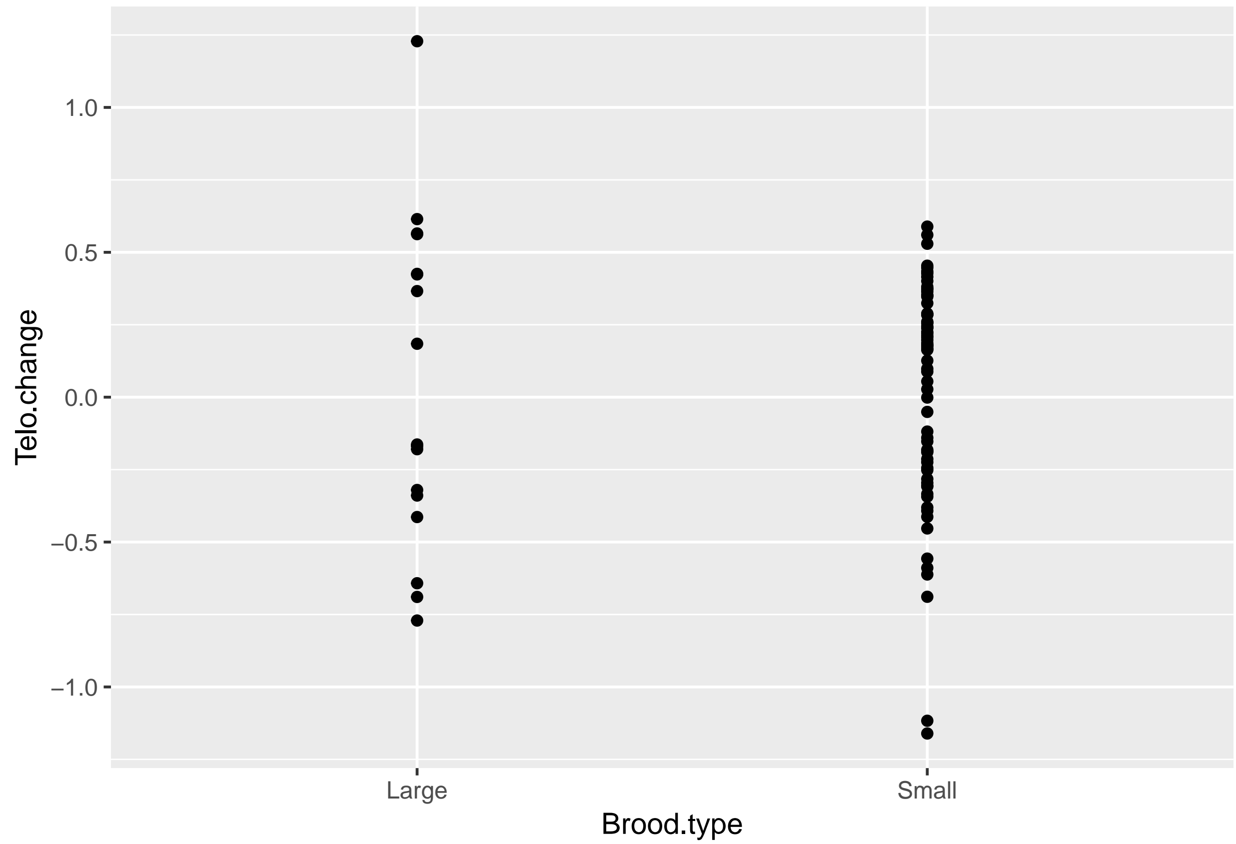
This says that **fig1** (an object) is going to be a ggplot of dataframe **d** with aesthetics **x=Brood.type** and **y=Telo.change**. Now let's look at a preview of **fig1** (it should appear in your plot preview window down on the right):

```
fig1
```



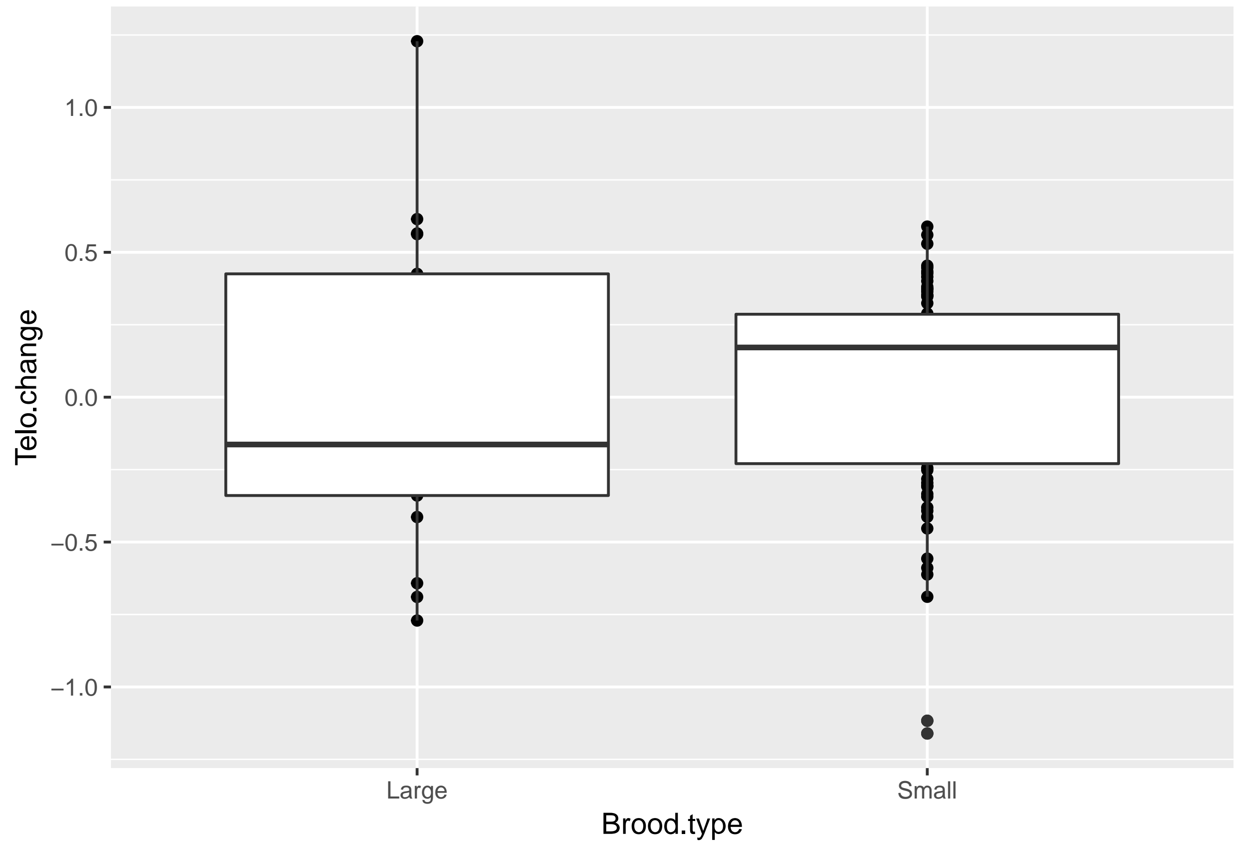
There's nothing there! That's because we have defined our canvas, but not yet put anything onto it. The way you make a ggplot interesting is by adding geometric objects, or **geoms**, to your canvas. So try instead adding the simplest geom, which is points:

```
fig1 = fig1 + geom_point()
fig1
```



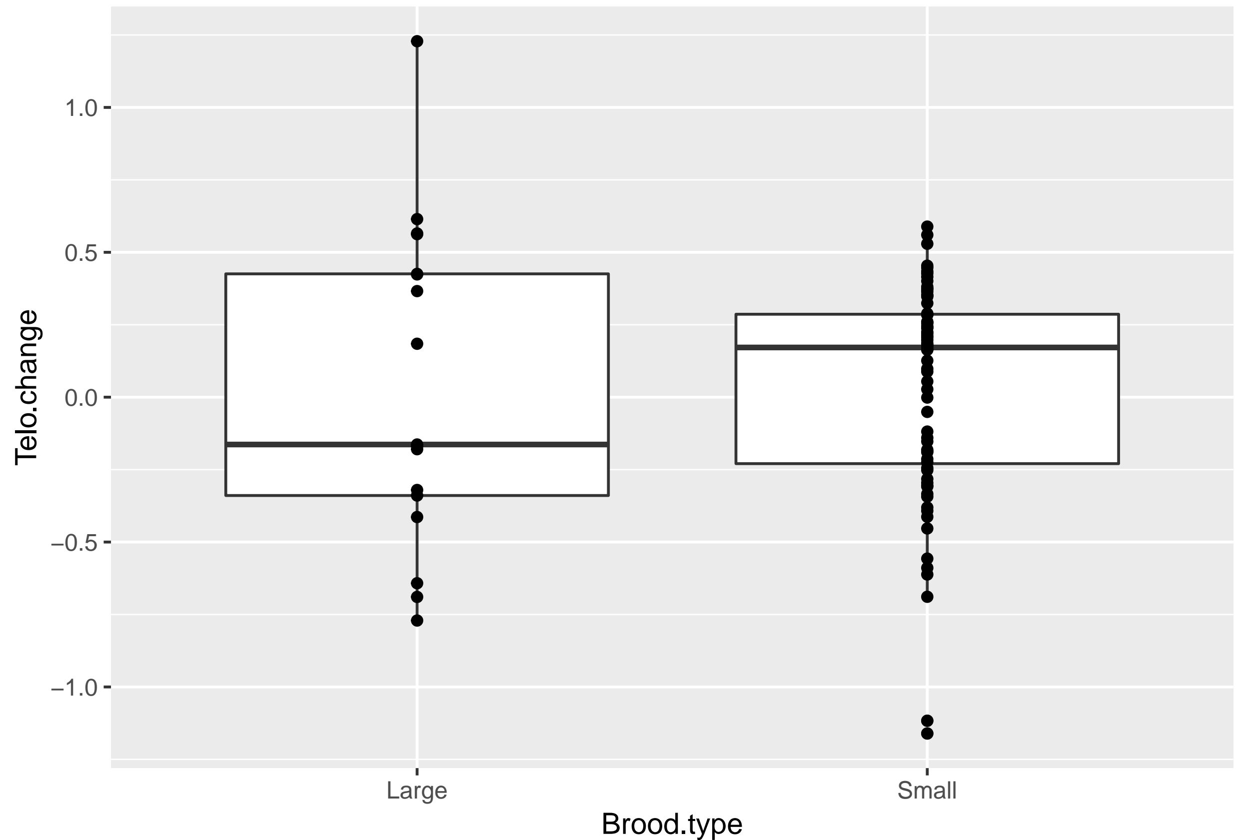
That's more like it. But maybe you want some better representation of the central tendency on there, so how about a boxplot?

```
fig1 = fig1 + geom_boxplot()
fig1
```



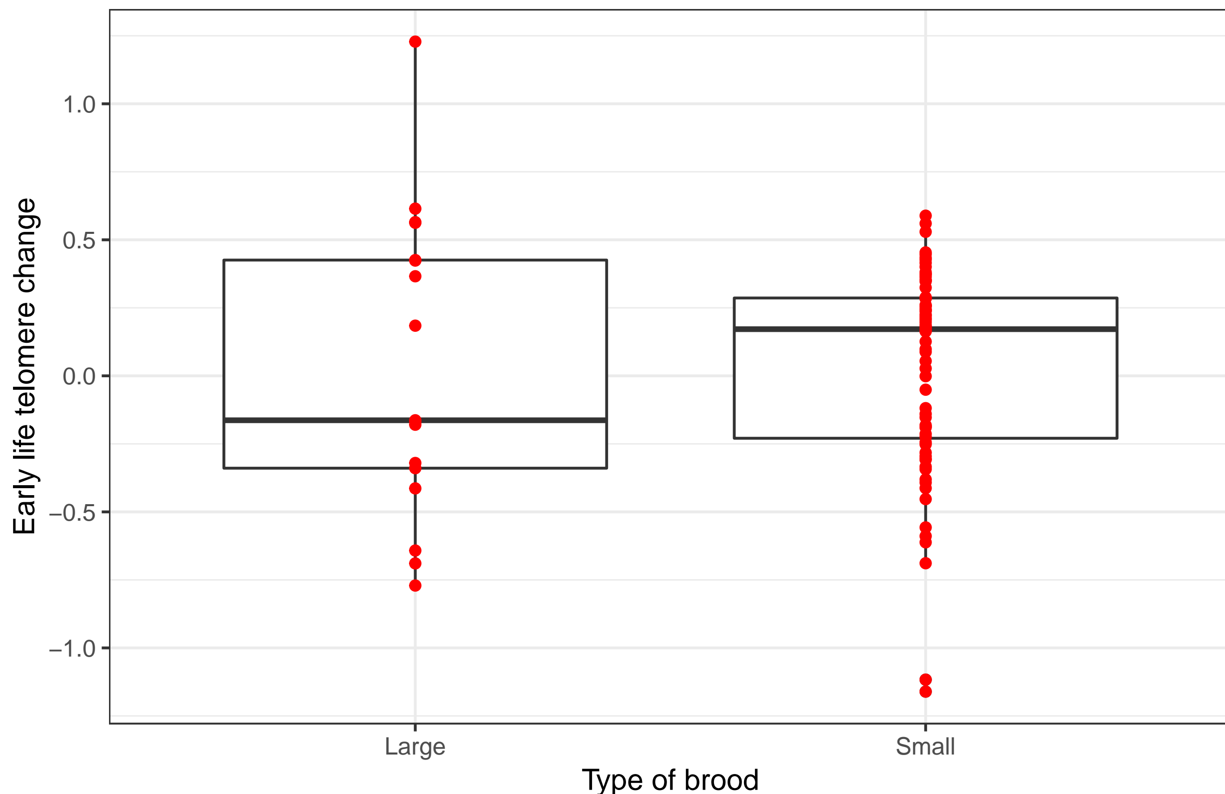
Getting there. But really it would be better to have our points over the top of our boxes, rather than the other way around. The order in which you add layers to the object affects their final display. So let's build **fig1** again from the bottom up (note that the position of the plus signs before the carriage return matters here):

```
fig1 = ggplot(d, aes(x=Brood.type, y=Telo.change)) +
  geom_boxplot() +
  geom_point()

fig1
```



Now as well as adding geoms, you can add **theme** elements (which change the appearance of the plot), and instructions about things like axis labels. I am now going to improve the appearance of **fig1**, getting rid of the nasty grey background, giving the axes better labels, and adding a title. This chunk of code is chunk 1 in the script file 'session3.r'.

```
fig1 = ggplot(d, aes(x=Brood.type, y=Telo.change)) +
  geom_boxplot() +
  geom_point(colour="red") +
  theme_bw() +
  xlab("Type of brood") +
  ylab("Early life telomere change") +
  ggtitle("Telomere change by brood type")

fig1
```



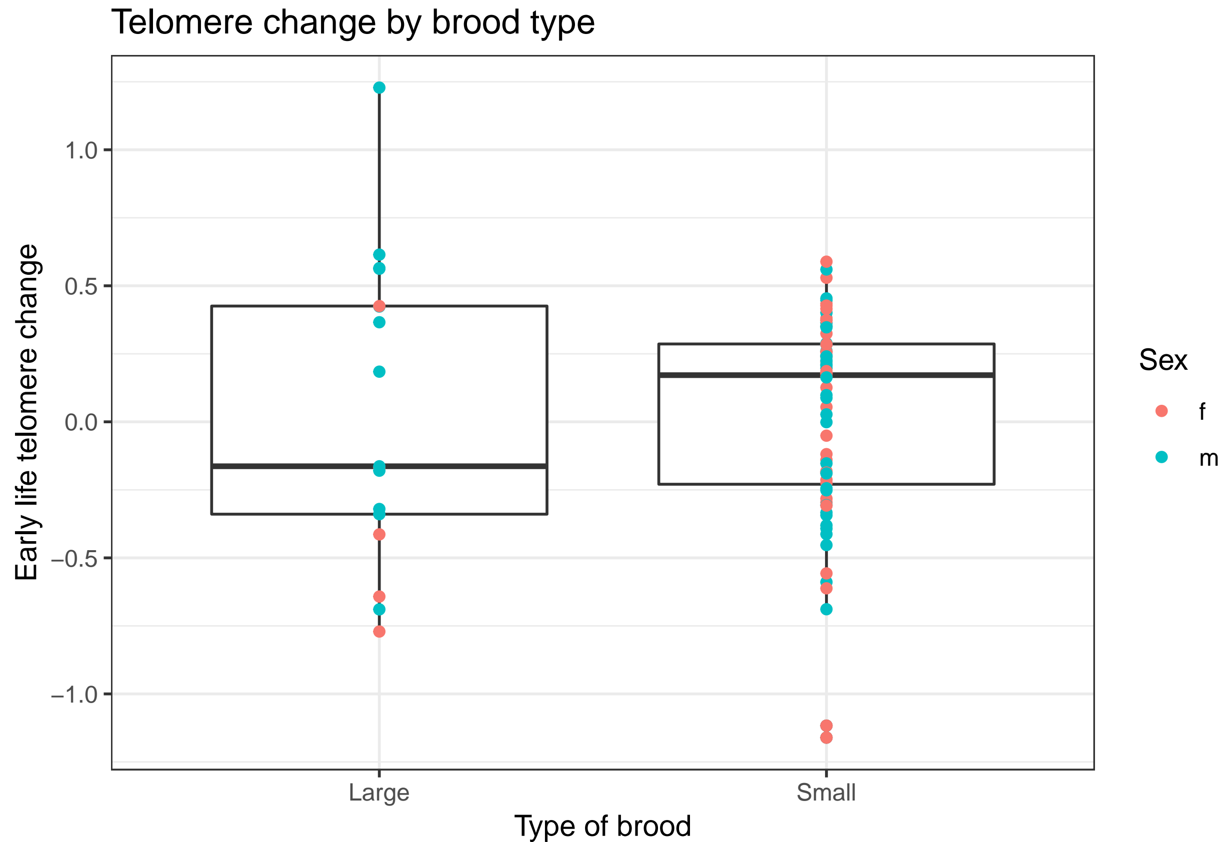
I always apply **theme_bw()** because I do not like the grey canvas that 'ggplot2' gives you by default. To avoid having to include this line in the definition of every figure, you can specify it once and for all at the top of your script (but after **library(ggplot2)**), as follows:

```
theme_set(theme_bw())
```

This line means that you don't need to add **theme_bw()** to each individual plot. However, in my code here, I will continue to do so, in case you are not using the code in order.

Each geom inherits the aesthetics from the first line that defined the ggplot. You might, however, want to have different aesthetics for different geoms on the same plot. For example, you might want the points to be coloured by sex of bird. This is an aesthetic, because it involves mapping an aspect of the graph (point colour) to a column in the data frame (**Sex**). So we need to put an extra aesthetic into the **geom_point()**. Try the following (this is chunk 2 in the script 'session3.r', so you can just run it from there):

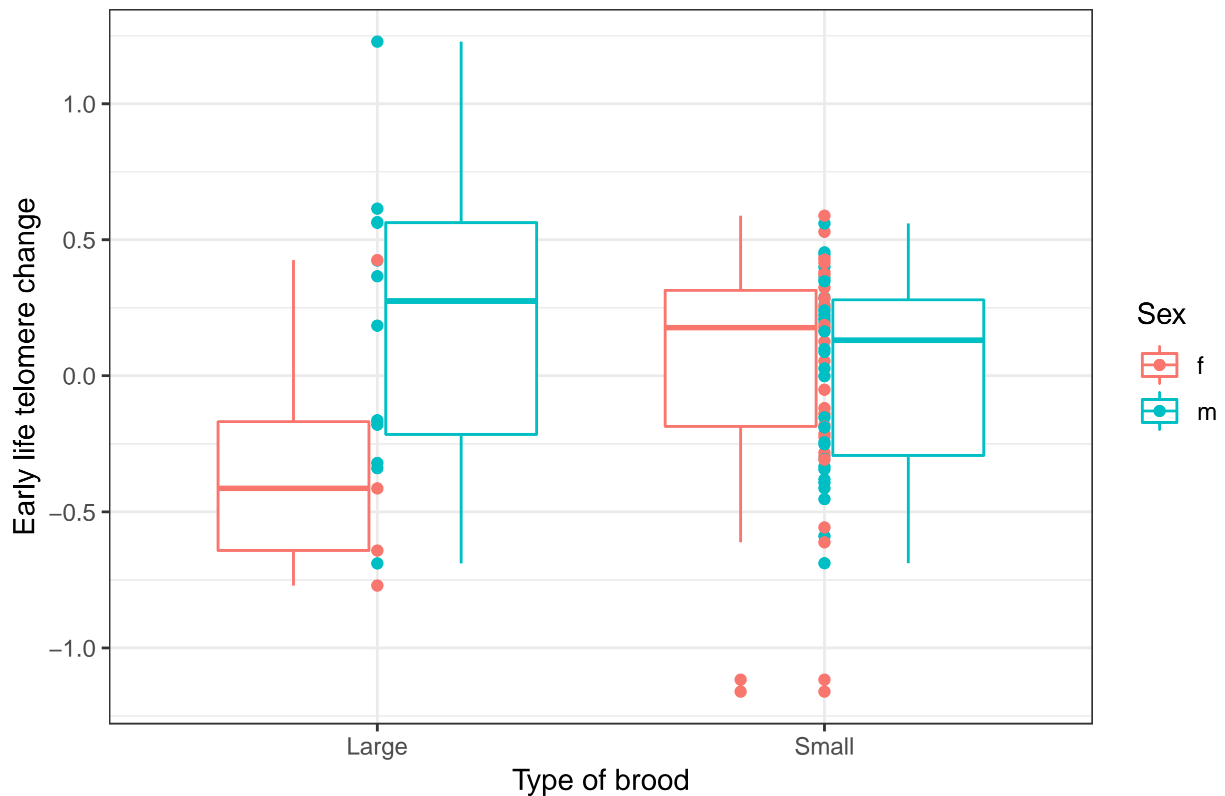
```
fig1 = ggplot(d, aes(x=Brood.type, y=Telo.change)) +
  geom_boxplot() +
  geom_point(aes(colour=Sex)) +
  theme_bw() +
  xlab("Type of brood") +
  ylab("Early life telomere change") +
  ggtitle("Telomere change by brood type")

fig1
```



Note a subtle point: When the colour of our points was just ‘red’, colour was not an aesethetic in the call to **geom_point()**, just a regular argument. That’s because all points were red, and so there was no mapping of point colour to anything in our data frame **d**. Now we want to colour by something in the data frame, colour has become an aesthetic: hence the **geom_point(aes(colour=Sex))**, rather than just **geom_point(colour=Sex)**.

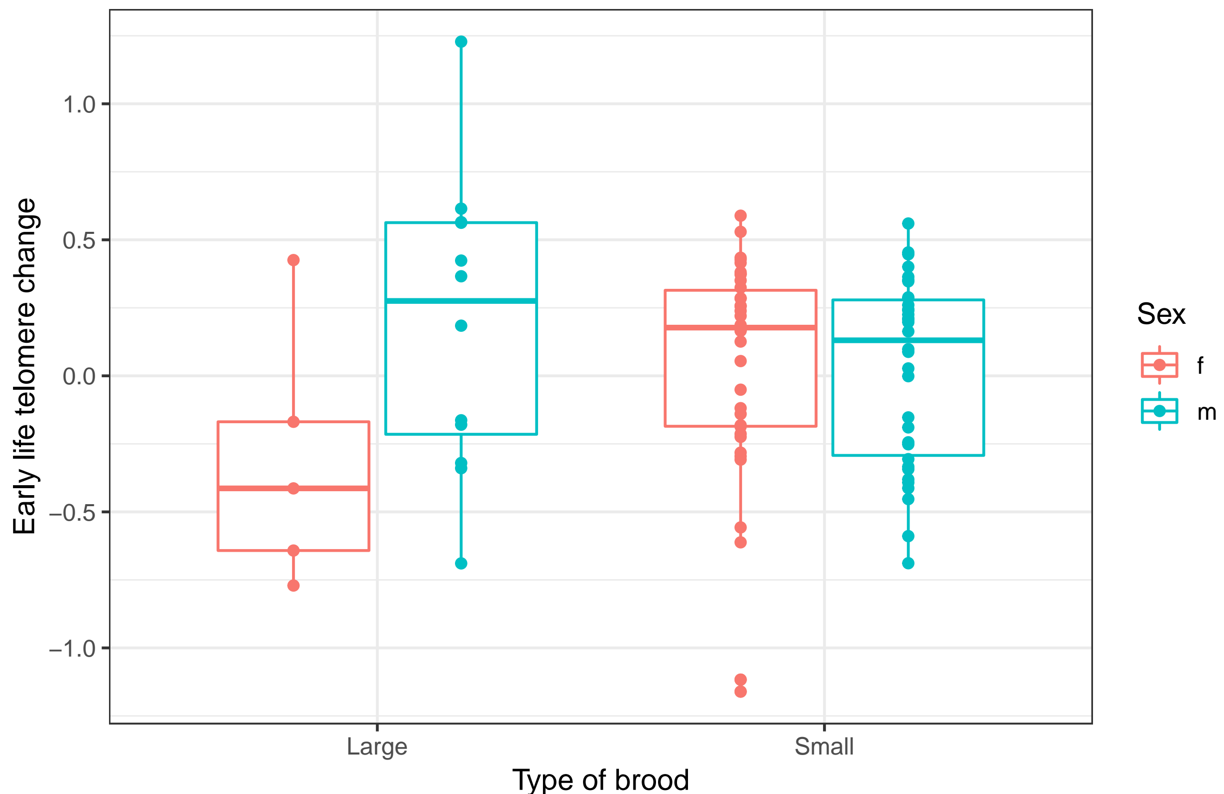
Note also that if we set the aesthetic **colour=Sex** in the global call (the first line) rather than in the call to **geom_point()**, then both the points and the boxplot inherit it (this is chunk 3 in the script).

```
fig1 = ggplot(d, aes(x=Brood.type, y=Telo.change, colour=Sex)) +
  geom_boxplot() +
  geom_point() +
  theme_bw() +
  xlab("Type of brood") +
  ylab("Early life telomere change") +
  ggtitle("Telomere change by brood type")

fig1
```



Now there is just one thing: really you would want the red points over the red box, and the blue points over the blue box. We can do this by specifying position in the call to the **geom_point()** (chunk 4 of script).

```
fig1 = ggplot(d, aes(x=Brood.type, y=Telo.change, colour=Sex)) +
  geom_boxplot() +
  geom_point(position=position_dodge(width=0.75)) +
  theme_bw() +
  xlab("Type of brood") +
  ylab("Early life telomere change") +
  ggtitle("Telomere change by brood type")

fig1
```

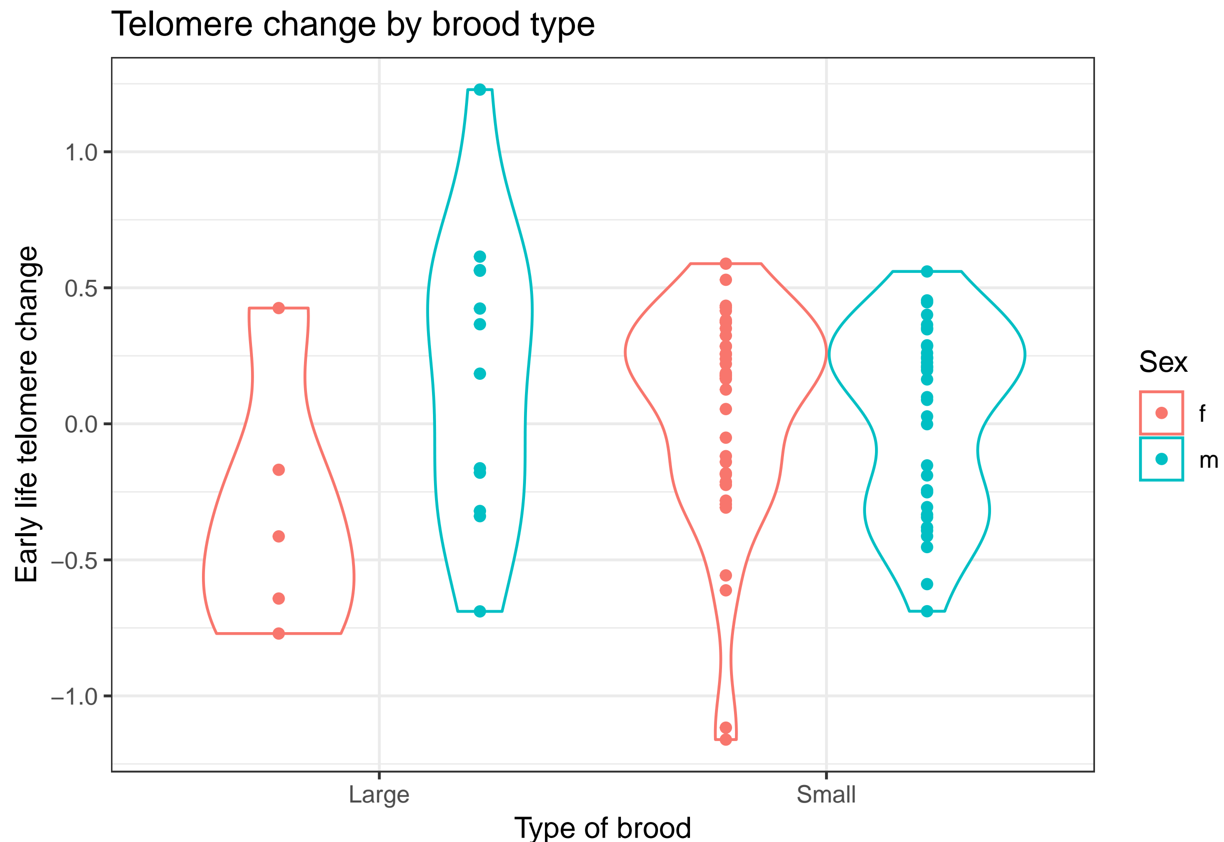


How did I know that I needed to use **position_dodge()** with **width=0.75**? It's just one of those things you pick up with experience.

In this example, to summarise the distribution of data per group, we have used boxplots (which show medians and inter-quartile ranges). If instead you want to make a bar plot with means and error bars, you can do this too, but it requires some intermediate steps: see Appendix, 'Plotting a bar graph with error bars'.

There is also something called a violin plot, which I am rather fond of as it shows the whole distribution of the data in each group. It's only robust in large datasets though: don't draw a violin if you only have a few data points per group.

```
fig1 = ggplot(d, aes(x=Brood.type, y=Telo.change, colour=Sex)) +
  geom_violin() +
  geom_point(position=position_dodge(width=0.9)) +
  theme_bw() +
  xlab("Type of brood") +
  ylab("Early life telomere change") +
  ggtitle("Telomere change by brood type")

fig1
```



Saving your figure

What you see in the plot window at bottom right is just a small preview. The proportions are not as you will finally want them. To get a better idea of the final product, hit **Zoom** in the plot window, and then resize until the proportions of text and graph are as you want. You can also use the Export button in the plot window to copy your plot to the clipboard so you can insert it into a Word or Powerpoint document.

When it comes time to make your publication version, you need to print the figure to a graphical device. R has several of these: the most likely ones you will use are 'pdf' (as the name suggests, makes PDFs), 'png' (makes .png pictures), and 'jpeg' (makes .jpeg). PDF is vector-based and so should not appear blocky at any resolution. On the other hand, you can't put PDFs into Powerpoint and some journals don't accept figures in this format.

To save your figure, you open the device, print the figure to it, and close it again to finalise the output file using the command **dev.off()**. Here's the code you need for saving our figure 1 in PDF, when I want a figure 5 inches wide and 5 inches high (the PDF device only uses inches, I don't know why). (This is chunk 5 in the script.)

```
pdf("figure1.pdf", width = 5, height =5)
fig1
dev.off()
```

The PDF will be saved in your current working directory - find it and check it is ok. Note that the size of axis and title text will automatically scale according to the size of the image. Allowing a square of about 4 to 6 inches for each figure usually produces something that looks about right, but you will need to experiment until you get the right look for your figure. Sometimes journals will mandate a particular size of image.

For .png and .jpeg, the language is slightly different, because you have to specify the resolution in pixels, and you can use cm instead of inches for the size of the image if you want (this is chunk 6 in the script).

```
png("figure1.png", res=300, width = 12, height =12, units="cm")
fig1
dev.off()
```

Check that it has saved ok. The code for jpeg is very similar.

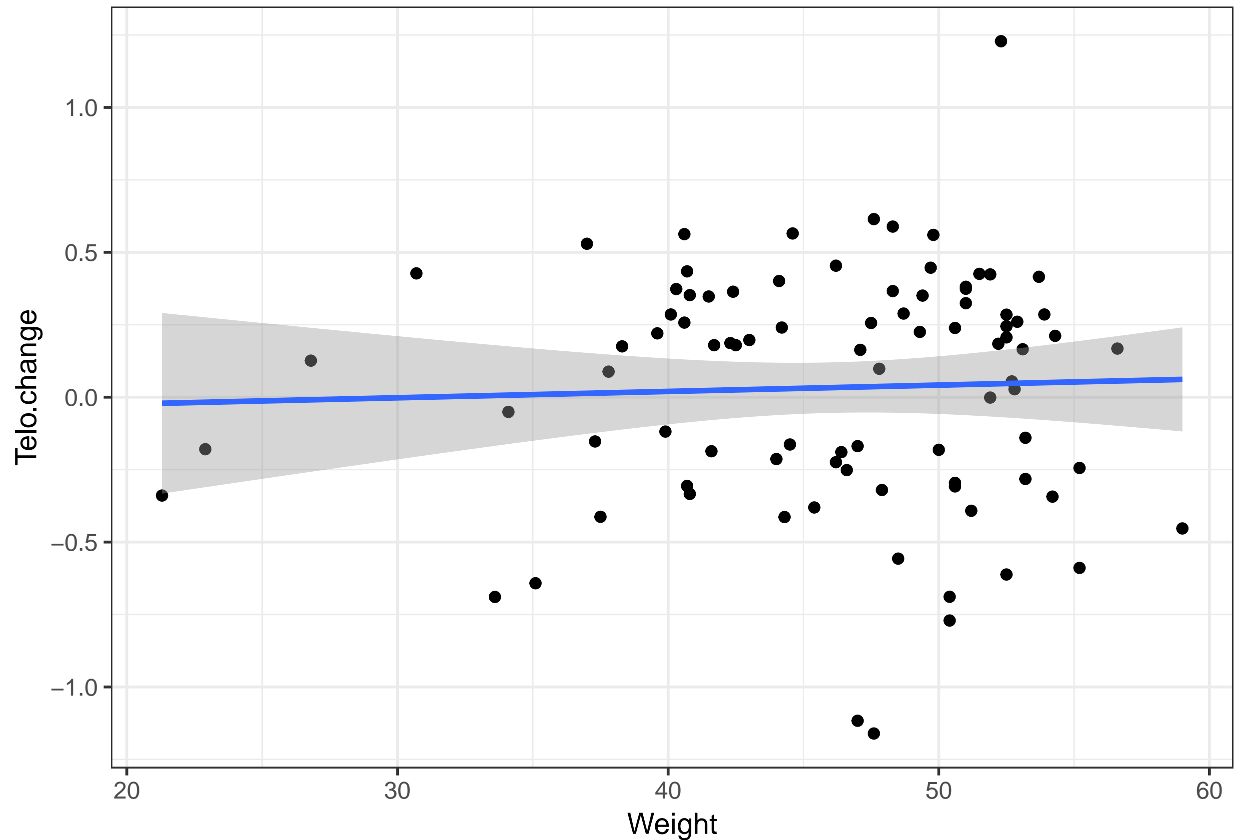
Modifying the appearance of your figure

You can change the appearance of everything on your ggplot figure - the background, the text, the legend, the title - and you can add all kinds of additional markers and labels. You often do this through **theme()** statements or other kinds of modifications. Rather than me trying to go through all the possibilities, just get into the habit of Googling 'how do I XXXX ggplot?'. There is masses of example code on the web. There is also a very useful book, Winston Chang's *R Graphics Cookbook*, PDFs of which can be found online. This tells you how to modify almost every aspect of your ggplot.

A second figure and introduction to interactions

Now we are going to make a second figure where we look at how telomere change is related to weight. Let's start with a simple scatterplot plus a regression line (this is chunk 7 in your script).

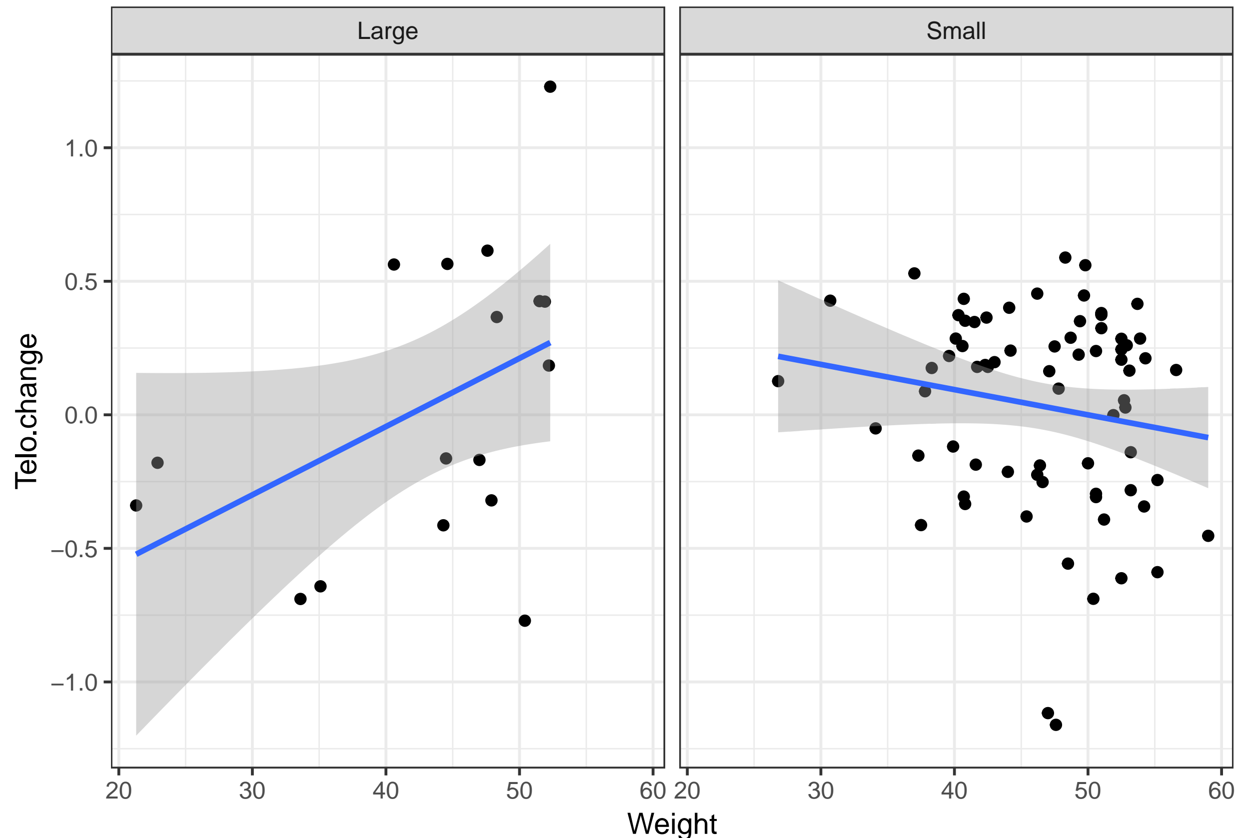
```
fig2=ggplot(d, aes(x=Weight, y=Telo.change)) +
      theme_bw() +
      geom_point() +
      geom_smooth(method="lm")

fig2
```



The **geom_smooth()** gives you the regression line, and the **method="lm"** specifies that it is to be a linear fit, rather than for example a loess (moving average) fit. If you want to turn off the standard error shading around the line, you can do it by specifying **geom_smooth(method="lm", se=FALSE)**.

Now, I want to investigate whether the relationship between **Weight** and **Telo.change** is the same in both large and small broods. Drawing on what we did with figure 1, we could set a new aesthetic of **colour=Brood.type**, and hence get the points for the large and brood sizes distinguished visually. But I want to introduce another trick for showing sub-groups of your data separately: facetting. We can put the birds from each type of brood onto separate panels or 'facets'. If you have already run the code setting up figure 2, try the following:

```
fig2 = fig2 + facet_wrap(~Brood.type)
fig2
```



Interesting. It looks the relationship between **Weight** and **Telo.change** goes in opposite directions in large and small broods. In large broods, the heavier you are, the less your telomeres shorten over development. In small broods, the lighter you are, the less your telomeres shorten, the opposite relationship. We won't go into the biology here, but it seems that maybe in large broods, there is a kind of winner-take-all situation, where the individuals that win out in competition can do both more growing and more telomere repair, whereas in small broods, there is more of a specialization relationship, with some individuals doing faster growth, and others doing more telomere repair.

In situations like this, we are not going to capture what is going on with a simple linear model of the type we considered in session 2. Why not? Well let's set up the model. First let us centre the **Weight** variable. Either of the following two lines will do the job (they both achieve the same thing):

```
d$Weight.centred=d$Weight-mean(d$Weight)
d$Weight.centred=scale(d$Weight, center=TRUE, scale=FALSE)
```

Now let's run a simple linear model (chunk 8 in script):

```
d$Weight.centred=d$Weight-mean(d$Weight)
m=lm(Telo.change ~ Brood.type + Weight.centred, data=d)
summary(m)
```

```
##
## Call:
## lm(formula = Telo.change ~ Brood.type + Weight.centred, data = d)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.1941 -0.2955  0.1310  0.3129  1.1671
```

```
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)      0.046624   0.102326   0.456    0.650
## Brood.typeSmall -0.016884   0.114137  -0.148    0.883
## Weight.centred   0.002354   0.006230   0.378    0.706
##
## Residual standard error: 0.416 on 86 degrees of freedom
## Multiple R-squared:  0.001731,   Adjusted R-squared:  -0.02148
## F-statistic: 0.07457 on 2 and 86 DF,  p-value: 0.9282
```

The parameter estimates for both **Brood.type** and **Weight.centred** are basically zero, indicating no predictive power. But let's go back to the formula of the model:

$$Telo.change_i = \beta_0 + \beta_1 * Brood.type_i + \beta_2 * Weight_i + \epsilon_i$$

What we are saying here is that to get to the **Telo.change** of an individual from a small brood from that of an individual from a large brood, we have to add β_1; and to get to the **Telo.change** of an individual of centred **Weight** 0 to that of an individual with centred **Weight** of 1, you have to add β_2. And therefore we are saying that to get from the **Telo.change** of an individual from a large brood with centred **Weight** of 0 to the **Telo.change** of an individual from a small brood with centred **Weight** of 1, we add $\beta_1 + \beta_2$. That is, the effects of being from a different brood type and those of being a different weight simply add together. For this reason, we call this an *additive model*. But our figure 2 suggests that the additive model is wrong. Instead, it suggests that the effect of adding a gram of weight on your **Telo.change** will be completely different for an individual in a large brood as for an individual in a small brood. The line connecting **Telo.change** to weight needs to be able to have a *different slope* in each of the two types of brood.

We achieve this with what is called an *interactive model*. Here, we specify:

$$Telo.change_i = \beta_0 + \beta_1 * Brood.type_i + \beta_2 * Weight_i + \beta_3 * Brood.type_i * Weight_i + \epsilon_i$$

Now the effect on telo.change of adding a gram of weight will be β_2 for individuals from large broods, but $\beta_2 + \beta_3$ for individuals from small broods (why? because when **Brood.type** is "Large", R treats it as a zero, so **Brood.type*Weight** is always zero for individuals from large broods). The interactive model allows us to ask: is the slope of the association between **Weight** and **Telo.change** the same in both types of brood, in which case β_3 will be estimated close to zero; or is it different, in which case β_3 will have a non-zero estimate?

We specify the interactive model as follows in R (chunk 9):

```
mi=lm(Telo.change ~ Brood.type + Weight.centred + Brood.type:Weight.centred, data=d)
```

Now let's look at the summary and the confidence intervals:

```
summary(mi)
```

```
##
## Call:
## lm(formula = Telo.change ~ Brood.type + Weight.centred + Brood.type:Weight.centred,
##     data = d)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.1831 -0.2725  0.1141  0.2766  0.9581
##
## Coefficients:
##                               Estimate Std. Error t value Pr(>|t|)
## (Intercept)                    0.10998    0.10126   1.086  0.28049
## Brood.typeSmall               -0.07263    0.11184  -0.649  0.51782
```

```
## Weight.centred                  0.02556    0.01034   2.472  0.01543 *
## Brood.typeSmall:Weight.centred -0.03501    0.01270  -2.757  0.00715 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4009 on 85 degrees of freedom
## Multiple R-squared:  0.08365,    Adjusted R-squared:  0.05131
## F-statistic: 2.586 on 3 and 85 DF,  p-value: 0.05838
```

```
confint(mi)
```

```
##                                       2.5 %       97.5 %
## (Intercept)                    -0.091345457  0.311299304
## Brood.typeSmall                -0.294995216  0.149734092
## Weight.centred                  0.005003046  0.046126538
## Brood.typeSmall:Weight.centred -0.060266360 -0.009758499
```

The parameter estimates for **Brood.typeSmall** and **Weight.centred** are the estimates of β_1 and β_2: these are known as the main effects of brood type and weight respectively. The final parameter estimate is known as the interaction term and estimates β_3. It does indeed appear to be different from zero: the slope has a more negative value in the small broods than the large, just as figure 2 led us to expect.

By the way, if you have interactive models, it is a very good idea to centre all your continuous predictors, as we have done here. This is because, if you don't, the meaning of the main effects is hard to interpret. The main effect of **Brood.type** in model **mi** represents the difference between the **Telo.change** of small and large broods when the other variable (**Weight.centred**) is equal to zero. If we had not centred **Weight**, that would be kind of meaningless: a bird with weight zero is biologically impossible, and hence extrapolating from the model what the difference between small and large broods would be at this point is hard to make sense of. By centring, we have put the zero point at the average of the weight distribution, so the main effect of **Brood.type** now represents the difference in **Telo.change** between small and large broods when the chicks are of average weight. That's a much more interpretable quantity.

There is another way of specifying model **mi**, which is as follows:

```
mi=lm(Telo.change ~ Brood.type*Weight.centred, data=d)
summary(mi)
```

```
##
## Call:
## lm(formula = Telo.change ~ Brood.type * Weight.centred, data = d)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.1831 -0.2725  0.1141  0.2766  0.9581
##
## Coefficients:
##                                Estimate Std. Error t value Pr(>|t|)
## (Intercept)                     0.10998    0.10126   1.086  0.28049
## Brood.typeSmall                -0.07263    0.11184  -0.649  0.51782
## Weight.centred                  0.02556    0.01034   2.472  0.01543 *
## Brood.typeSmall:Weight.centred -0.03501    0.01270  -2.757  0.00715 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4009 on 85 degrees of freedom
## Multiple R-squared:  0.08365,    Adjusted R-squared:  0.05131
## F-statistic: 2.586 on 3 and 85 DF,  p-value: 0.05838
```

As you can see, this produces the same model as before. The **Brood.type*Weight.centred** notation is a shorthand: it is interpreted as 'fit the main effects of **Brood.type** and **Weight.centred**, and also their interaction'. Note that interactive models can rapidly become complex. For example, if you have three predictors, **x1**, **x2** and **x3**, you can have up to four interaction terms, namely **x1:x2**, **x1:x3**, **x2:x3**, and **x1:x2:x3**. These are interpreted respectively as: the effect of a unit change in **x2** on the slope of the outcome on **x1**; the effect of a unit change in **x3** on the slope of the outcome on **x1**; the effect of a unit change in **x3** on the slope of the outcome on **x2**; and the effect of a unit change in **x3** on the effect of a unit change in **x2** on the slope of the outcome on **x1**. If you use the notation **y~x1*x2*x3**, all four of these interaction terms will be added to the model. In my view you should usually avoid adding all possible interaction terms promiscuously. Use the **x1:x2** notation to manually add just those interactions you have some theoretical or empirical reason to suspect might be important.

Justifying the interaction terms you include

An interactive model will always fit the data as least as well as a simpler additive one (because the additive one is just the special case of the interactive one where the parameter for the interaction term is zero). But interactive models are always more complex than the corresponding additive ones, because they contain more parameters to be estimated. And more complex models are less *parsimonious*. Parsimony is the principle that, other things being equal, a less complex model should be preferred over a more complex one.

Often your research hypothesis will tell you which interactions must be included in the model, but sometimes it does not: it leaves open the possibility that your predictors might have interactive effects, without specifically predicting that they will do so. In such cases, one way to justify your choice of which interaction terms to include and which to exclude is by comparing AIC values of different models. AIC stands for Akaike Information Criterion. Comparing the AIC of two models is a way of establishing which one gives the best predictive power to complexity ratio. Another, and more precise, way of saying this is that a model with a lower AIC will usually do a better job of predicting the outcome in future samples from the same large world than a model with a higher AIC will.

Let's look at the AIC values of our two models, the additive model **m** and the interactive model **mi**.

```
AIC(m)
```

```
## [1] 101.415
```

```
AIC(mi)
```

```
## [1] 95.79448
```

With AIC, a smaller value is better, and if one model's AIC is at least two units smaller than another model's, it is clearly preferable (if the gap is less than two units, they are said to be nearly the same in terms of explanatory value). Here, the difference is more than 5 units, so we say **mi** is clearly the better model in this case. And that makes sense given the pattern we saw in figure 2. By the way, the absolute value of AIC is not really meaningful, since it depends on the dataset: you can't say that 95 is a particularly large or small AIC. It's the *difference* in AIC between two models of the same data (sometimes notated as 'ΔAIC' or 'delta AIC') that matters. You cannot compare the AICs of models based on different data. And just to do your head in, AIC values can sometimes be negative (in which case, more negative is better).

What to report if you have an interaction

If there is an important interaction going on, you will find that the parameter estimate for the interaction term of your model has a 95% confidence interval that does not cross zero, and that the AIC of the interactive model is lower than that of a corresponding model from which the interaction term is omitted, just as we found in the present case. In the present case, I would report the parameter estimates and their 95% confidence intervals for the main effects of **Brood.type** and **Weight.centred**, and the interaction effect. In

addition, I would calculate the slope of **Telo.change** on **Weight.centred** for the two types of brood (these are sometimes known as the 'simple slopes'). This is easy: they are, respectively, β_2 and $\beta_2 + \beta_3$. You can get them out of the model summary. For large broods:

```
# Beta 2
summary(mi)$coefficients[3, 1]
```

```
## [1] 0.02556479
```

And for small broods:

```
# Beta 2 plus Beta 3
summary(mi)$coefficients[3, 1] + summary(mi)$coefficients[4, 1]
```

```
## [1] -0.009447637
```

So, a positive slope in large broods, and a slightly negative one in large ones, just as figure 2 suggests.

If your model contains a three-way interaction, that's trickier. Evolving as a mobile hunter-gatherer on the great savannahs of Africa has left *Homo sapiens* poorly equipped for being able to understand three-way interactions. If you find one, I would recommend splitting your dataset in half on the basis of one of the three variables involved. You should then look for two-way interactions between the remaining two variables in each of those two datasets separately.

Session 4: Mixed and generalized models

This session extends the standard general linear model that we have met so far (function **lm()**) in two different ways. First, we look at a case where the outcome variable is binary rather than a continuous variable. In such a case, we cannot assume that the response is the sum of a linear predictor and an approximately normally-distributed error term, and we need a slightly different modelling approach. We switch from a general linear model to a generalized linear model.

Second, we look at a case where there is some non-independence or structure in the dataset, that we need to account for in our model. This calls for what is called a linear mixed model.

At the very end, I will point out that you can readily combine these two extensions in the same model: if your data requires it, you fit have a model that is both generalized and mixed. Such a model is, appropriately enough, called a generalized linear mixed model!

Generalized linear models

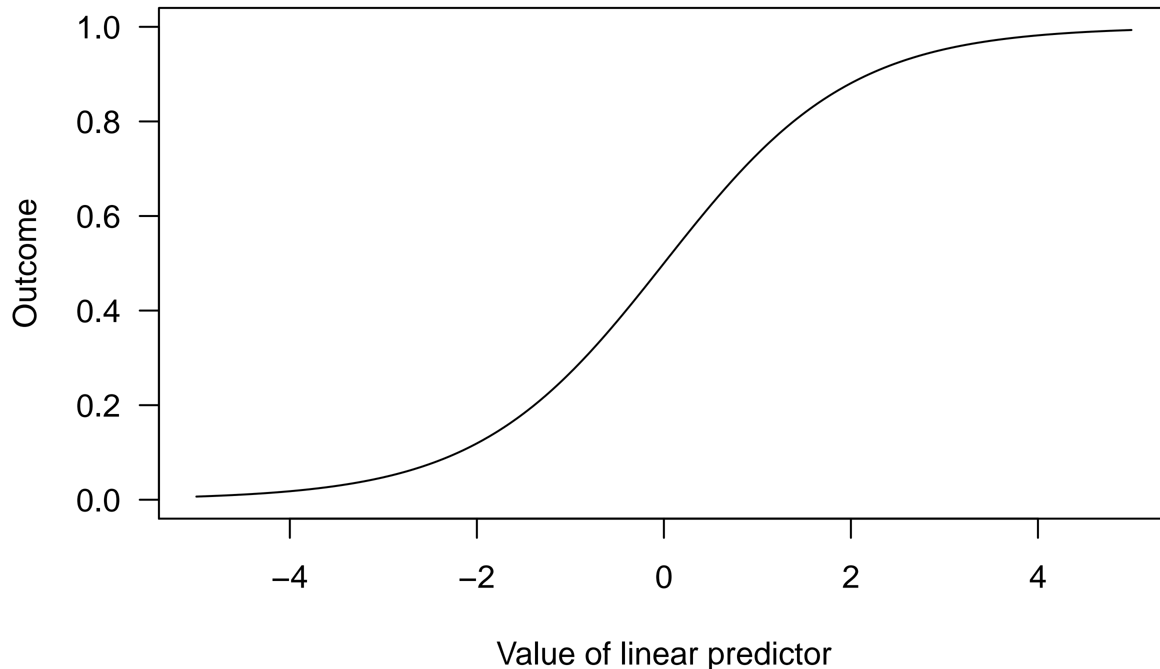
In the cases we have met so far, we had a model of the kind:

$$y_i = \beta_0 + \beta_1 * x1_i + \beta_2 * x2_i + \beta_3 * x3_i + \epsilon_i$$

where the y variable can vary continuously, and ϵ represents an error term that is normally distributed with a mean of zero. But there are some cases where this assumption is not sensible. For example, if our y variable can only be equal to either 0 or 1 (that is, the outcome either happens or it does not), then predicting it from the value of a continuously varying linear predictor$\beta_0 + \beta_1 * x1 + \beta_2 * x2...$ plus a normally distributed error ϵ doesn't make sense. We really want the linear predictor plus the error to go from 0 at one extreme of its range, to 1 at the other extreme, and never stray outside of the interval [0,1].

Generalized linear models solve this problem with two innovations. First, instead of relating the value of the linear predictor directly to the expected value of the outcome, they relate *a function* of the linear predictor, called the link function, to the expected value of the outcome. Then they make different assumptions about the probability distribution (also often described as the error distribution) of the outcome. Instead of a normal distribution, the error distribution can for example be binomial or Poisson. There are many different families of generalized linear model, with different combinations of link function and error distribution. The linear model we have met so far is in fact the special case of the generalized linear model where the link function is just the identity function, and the error distribution is normal or Gaussian.

Today we will work with a generalized linear model for when your response variable is binary. The link function we need is called the *logistic* function. Here is an example of what a logistic function looks like:



As we see, the logistic function is bounded at zero when the linear predictor is very small, and one when the linear predictor becomes large. This naturally captures the idea that at one end of the range of the predictors, the outcome is never going to happen, and at the other, it always is. The parameters of the linear predictor are going to influence the location and slope of the part of the function that rises from zero to one. The error distribution we are going to assume is *binomial*. The resulting model is called 'binary logistic regression' or 'generalized linear model with binomial error distribution and logistic link function'.

An example with a dichotomous outcome

Let's load in a new dataset for working with our generalized linear mixed model. The data come from a recent study of the developmental predictors of gender identity (Atkinson, B. M., Smulders, T. V. & Wallenberg, J. C. 2017 An endocrine basis for tomboy identity: The second-to-fourth digit ratio (2D:4D) in 'tomboys'. *Psychoneuroendocrinology* 79, 9-12). The authors made their data and R script freely available with their paper so I was able to download it when I read the paper. You might also be interested to know that this study was a Newcastle University MRes project, by Beth Atkinson. The research question is whether women who are labelled 'tomboys' as children have been exposed to higher levels of testosterone in utero than girls who are not so labelled. It happens that a simple proxy measure of the levels of testosterone you are exposed to in utero is the ratio of the lengths of your 2nd and 4th fingers (henceforth the digit ratio; lower digit ratio indicates more testosterone in utero). So that makes the prediction that girls who are labelled as tomboys will have a lower digit ratio than those who are not. Because digit ratio does not change much with age, and because people can remember whether they were called a tomboy or not, you can study this question in adult women by measuring their hands as they are now, and asking them to think back to their childhoods about whether they were called tomboy. The study measured the digit ratio (of the right hand, and it has been log transformed in the data we are using) in a sample of 44 women who said whether they were called tomboys or not. I have simplified the dataset for our purposes.

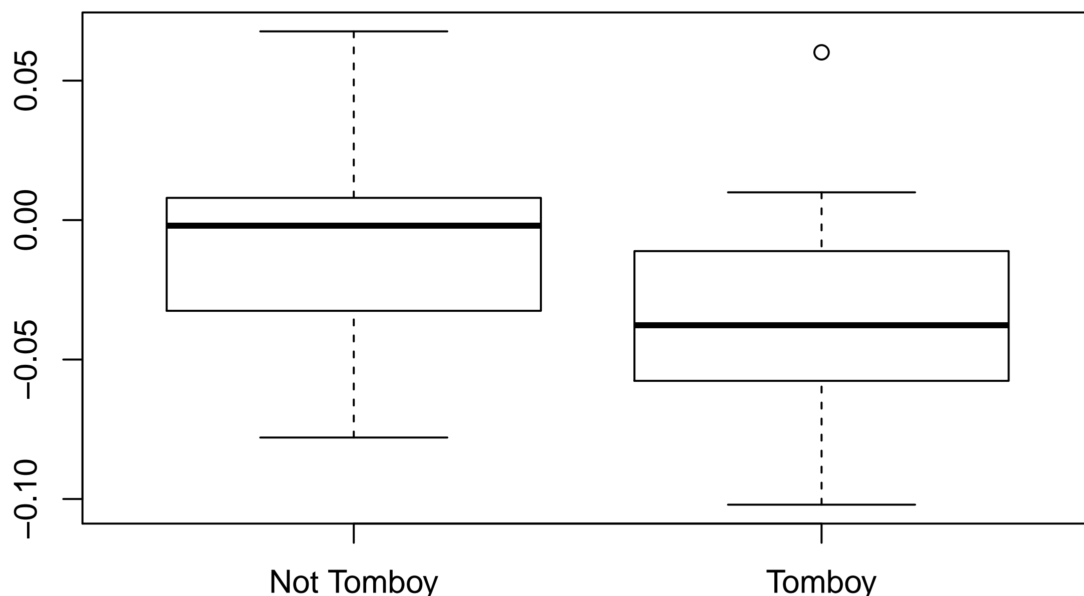
Let's load in the data (you need to have saved the file 'tomboy.data.csv' into the directory you are working from).

```
d=read.csv("tomboy.data.csv")
head(d)
```

```
##   X  digit.ratio     tomboy age
## 1 1  0.062974799 Not Tomboy  43
## 2 2 -0.013085240 Not Tomboy  20
## 3 4 -0.002002003 Not Tomboy  45
## 4 5 -0.033556784     Tomboy  35
## 5 6  0.001998003     Tomboy  24
## 6 7  0.000000000     Tomboy  33
```

So we have the logged digit ratio, and whether the women were called tomboy or not, plus the age at time of the study. Let's do a quick plot to see if the tomboys and non-tomboys might differ in digit ratio.

```
boxplot(d$digit.ratio~d$tomboy)
```



It certainly looks as if the tomboy participants might have lower digit ratios. Now let's set up a model to predict the outcome (being a tomboy or not) by digit ratio, and also including age as an additional predictor. We do this with the **glm()** function, which is like **lm()**, but generalized to other link functions and error distributions.

```
m = glm(tomboy~digit.ratio + age, data=d, family=binomial)
```

As you can see, the code is much the same as for **lm()**, but with the additional specification that the family of generalized model is binomial in this case, because the outcome is dichotomous. You don't need to spell out what the link function is, because R assumes you need a logistic function in the binomial family of model. If you wanted to be explicit, the following would achieve exactly the same as the above:

```r
m = glm(tomboy~digit.ratio + age, data=d, family=binomial(link="logit"))
```

Now let's look at a summary of the model, and the confidence intervals:

```r
summary(m)
```

```
##
## Call:
## glm(formula = tomboy ~ digit.ratio + age, family = binomial(link = "logit"),
##     data = d)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.5494  -0.9781  -0.5947   1.1255   2.0131
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -7.060e-01  7.827e-01  -0.902   0.3670
## digit.ratio -1.973e+01  9.261e+00  -2.131   0.0331 *
## age          4.041e-04  2.175e-02   0.019   0.9852
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 60.176  on 43  degrees of freedom
## Residual deviance: 54.715  on 41  degrees of freedom
## AIC: 60.715
##
## Number of Fisher Scoring iterations: 4
```

So it looks like the parameter estimate relating digit ratio to being a tomboy is significantly different from zero (with a central estimate of about -19.73), whereas that for age is around zero. But what does the parameter estimate of -19.73 actually mean? In the case of a standard general linear model, the parameter estimate tells you how much the mean of the response goes up for every increase of one unit in the predictor, but clearly here that does not quite apply.

The answer is that the parameter estimate tells you how much the *log odds* of the outcome changes with every increase of one unit in the digit ratio. So let's first translate that from log odds to odds by taking the anti-log (remember the base of logarithms here is natural).

```r
exp(-19.73)
```

```
## [1] 2.700038e-09
```

The number 2.7×10^{-9} is very small indeed. It tells us that for every increase of one unit in the digit ratio, the odds of being called a tomboy are reduced by more than 99.99999%! That seems like a huge effect, until you realise that the observed range of variation in the digit ratio is actually very small.

```r
sd(d$digit.ratio)
```

```
## [1] 0.04029197
```

So although the model says a one-unit change in digit ratio would indeed reduce the odds of being called a tomboy by over 99.99%, a change of one unit would represent 25 standard deviations of this particular measure. So a more sensible measure of effect size here would be to standardize the digit ratio so that it has a standard deviation of 1, then use that in the model, as follows (the R function **scale()** rescales a continuous variable so that it has a mean of 0 and a standard deviation of 1).

```r
m2 = glm(tomboy~scale(digit.ratio) + age, data=d, family=binomial)
summary(m2)
```

```
##
## Call:
## glm(formula = tomboy ~ scale(digit.ratio) + age, family = binomial,
##     data = d)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.5494  -0.9781  -0.5947   1.1255   2.0131
##
## Coefficients:
##                      Estimate Std. Error z value Pr(>|z|)
## (Intercept)        -0.3323841  0.7512225  -0.442   0.6582
## scale(digit.ratio) -0.7950358  0.3731353  -2.131   0.0331 *
## age                 0.0004041  0.0217513   0.019   0.9852
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 60.176  on 43  degrees of freedom
## Residual deviance: 54.715  on 41  degrees of freedom
## AIC: 60.715
##
## Number of Fisher Scoring iterations: 4
```

So now we get a parameter estimate of -0.795, and taking the antilog gives us:

```r
exp(-0.795)
```

```
## [1] 0.4515812
```

So every standard deviation increase in the digit ratio roughly halves your odds of being called a tomboy.

The odd thing about odds

Note that we expressed our effect size in the above example in terms of impact of the predictor on the *odds* of the outcome. Odds are defined as the probability of an event divided by one minus the probability of the event. The odds of something happening is thus not the same as the probability of it happening. If the probability of something happening is 0.2, then the odds of it happening is:

$$\frac{0.2}{(1-0.2)} = 0.25$$

If you double the probability of the event happening to 0.4, the odds are now:

$$\frac{0.4}{(1-0.4)} = 0.67$$

.

So a doubling of the probability is not the same as a doubling of the odds, or vice versa.

You might be curious to know why in statistical modelling we tend to express effects and associations in terms of changes in the odds of events, rather than changes in their probabilities. It's because odds have certain useful mathematical properties. You can always double the odds of an event, even if that event is

already near certain to happen. You can't always double the probability of an event. For example, if the probability of an event is already 0.9, then doubling its probability has no meaning, but doubling its odds is perfectly sensible (its odds are currently 9, and you could double them to 18, which would equate to a probability of the event happening of about 0.95). This property of odds means that a predictor can affect the odds of an event uniformly regardless of that event's baseline probability.

Other generalized families

Our example has used a generalized linear model from the binomial family for a case where the response was dichotomous. There are other families of generalized linear models that you might need. For example, if your data are counts of events, especially counts that are quite small and sometimes zero, then you might want a generalized model of the Poisson family. This is because your response is bounded at zero (you can't have an event happening fewer than 0 times), and hence asymmetrically distributed. Poisson models are fitted using the **glm()** function in just the same way as the binomial family. The default link function for the Poisson family is the logarithmic function.

```
p = glm(response ~ predictor1 + predictor2, data=my.data.frame, family=poisson)
```

You can readily choose another family of generalized models suitable for your data, as long as you have understood the general principles we have gone through in this session.

Mixed models: For when there is structure in your data set

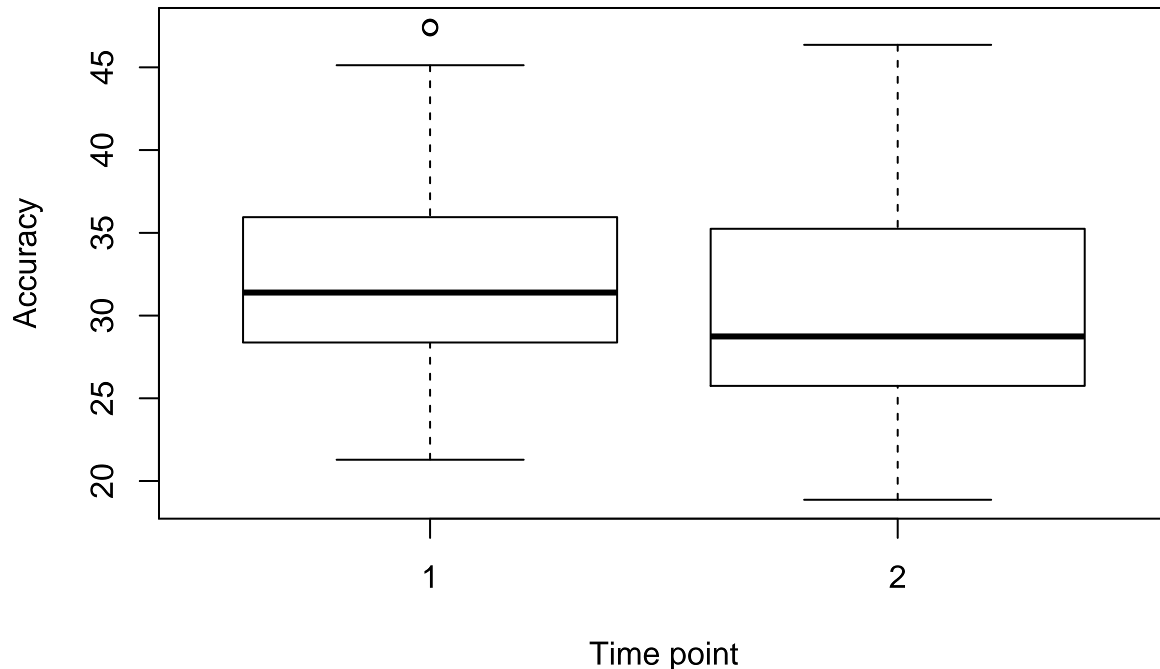
In this section, we are going to need a data file called 'sleep.data.csv', and two contributed packages: 'lme4' and 'afex'. The package lme4 fits linear mixed models. The package 'afex' provides an easy wrapper for specifying your linear mixed models and getting p-values and other statistics out of them. If you have not previously installed these packages, you will need to run **install.packages("lme4")** and **install.packages("afex")** before going any further. Once you have done this:

```
d=read.csv("sleep.data.csv")
head(d)
```

```
##   PersonID Time Accuracy
## 1        1    1 41.26861
## 2        2    1 30.23323
## 3        3    1 34.81972
## 4        4    1 34.18728
## 5        5    1 36.49972
## 6        6    1 30.45886
```

This dataset contains accuracy scores for people carrying out a computer task (**Accuracy**), taken at two time points (**Time**): time point 1 was after a normal night's sleep, and time point 2 was after a night of sleep deprivation. There are 20 observations at each time point. The research question is whether sleep deprivation makes people's accuracy on the task worse. If so, we would predict that accuracy will be lower at time 2 than time 1. So let's plot a boxplot of accuracy by time point.

```
boxplot(d$Accuracy~d$Time, xlab="Time point", ylab="Accuracy")
```



Well, maybe the median is a bit lower at time point 2, but there's not much difference compared to the spread of accuracies *within* a time point. So it does not seem like sleep deprivation makes a huge difference. This appears to be borne out when we fit a standard general linear model. Note that below I specify **factor(Time)** rather than just **Time**. Because R found just numbers in the column called **Time**, it did not realise it was actually a factor with discrete levels (1 and 2). So we have to tell R to treat it as such in models.

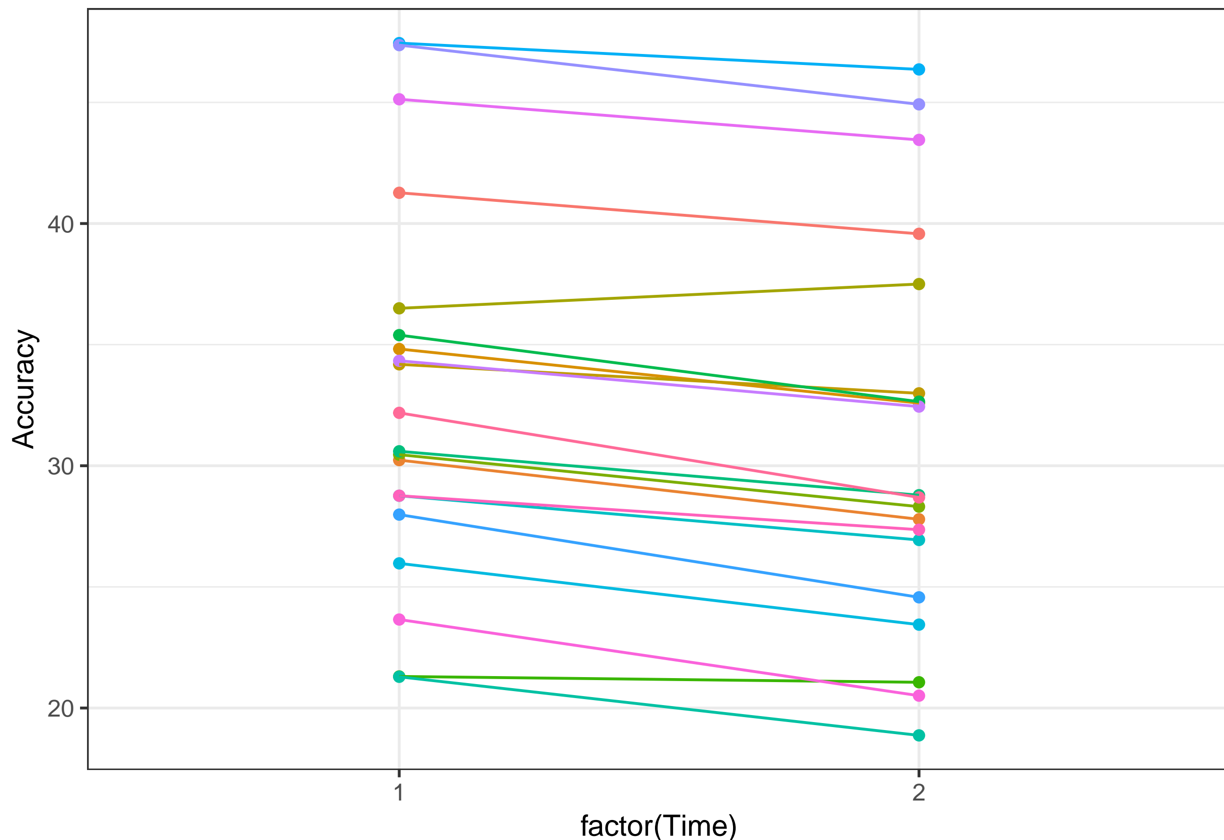
```
m = lm(Accuracy~factor(Time), data=d)
summary(m)
```

```
##
## Call:
## lm(formula = Accuracy ~ factor(Time), data = d)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -12.069  -4.314  -2.201   2.787  15.421
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)     32.882      1.766  18.618   <2e-16 ***
## factor(Time)2   -1.942      2.498  -0.777    0.442
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.899 on 38 degrees of freedom
## Multiple R-squared:  0.01565,    Adjusted R-squared:  -0.01025
## F-statistic: 0.6043 on 1 and 38 DF,  p-value: 0.4418
```

The parameter estimate for **Time2** is negative (-1.942), but it is small compared to its standard error. We would conclude that it is not significantly different from zero, and hence that sleep deprivation is not making much difference to accuracy.

There is however something I have not told you. The second 20 observations, the ones at time 2, are *the very same people as were tested as time 1*. A variable called **PersonID** tells you which person an observation belongs to, and so you can match up the same individual's time 1 and time 2 accuracy. When we do this, we get a rather different picture. Look at the figure below, which links every individual's time 2 accuracy to their time 1 accuracy.

```
library(ggplot2)
f=ggplot(d, aes(x=factor(Time), y=Accuracy, group=factor(PersonID), colour=factor(PersonID))) +
  geom_point() +
  geom_line() +
  guides(colour=FALSE) +
  theme_bw()
f
```



We can see that although there is a lot of variation between individuals at both 1 and time 2, when we link the observations of the same individual up, we see that most individuals get worse at time 2 (only one or two individuals buck the trend). So the true effect of sleep deprivation might well not be zero; it is just being masked by the large baseline differences between individuals on this task.

In cases like this, there is structure in the data that our model needs to take account of. That is, we don't have twenty observations taken after a good night's sleep, and twenty independent observations after sleep deprivation. Really, we we have 20 pairs of non-independent data points (one pair per person). So the question is not so much 'How much lower is the average accuracy at time 2 than at time 1?', but rather 'By how much is the average individual worse time 2 than time 1?'.

You could solve this problem by making a difference score: for every individual, subtract their time 2 accuracy from their time 1 accuracy, and see if this difference tended to be systematically different from zero (this is the approach behind the paired t-test, with which you may be familiar). That approach would be fine for the current case, but it would not generalise to more complex cases, for example where you have more than two measurements per individual, or different numbers of measurements for different individuals. The approach we will use here, the linear mixed model, works for the present situation, and also for more complex ones.

An example linear mixed model

In a mixed model, you have your response variable and your predictors as before. You also have some way of identifying which observations are grouped with which others, for example because they come from the same individual. So in the present case, the variable **PersonID** tells us which observations belong to the same person. When we set up the model, we tell it that each group of observations has its own intercept or average level. For example, some people are just much more accurate than others overall. We call these intercepts 'random intercepts' (which though widespread is a potentially misleading term, because they are not randomly generated) or 'varying intercepts'. Let's fit the model.

```
library(afex)
```

```
mx=mixed(Accuracy~factor(Time) + (1|PersonID), data=d)
```

```
## Numerical variables NOT centered on 0: Time
## If in interactions, interpretation of lower order (e.g., main) effects difficult.
## Fitting one lmer() model. [DONE]
## Calculating p-values. [DONE]
```

Within the call to the **mixed()** function, you will recognise the same formula you would use with an ordinary **lm()**, but with an extra term **+ (1|PersonID)**. We can spell this out in English as 'plus a varying intercept for each distinct PersonID'. The red warning message you may get is just reminding you that there are non-centred numeric variables in the formula and, as we saw in the previous section, this makes parameter estimates for main effects hard to interpret if there are interactions in the model. In this particular case, this warning does not concern us because (a) there are no interactions in the model; and (b) we are treating the **Time** variable as a factor anyway.

Now let's look at the summary of our mixed model **mx**.

```
summary(mx)
```

```
## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
## lmerModLmerTest]
## Formula: Accuracy ~ factor(Time) + (1 | PersonID)
##    Data: data
##
## REML criterion at convergence: 194.3
##
## Scaled residuals:
##      Min       1Q   Median       3Q      Max
## -1.94105 -0.38524  0.00887  0.34055  2.00219
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  PersonID (Intercept) 61.8320  7.8633
##  Residual              0.5569  0.7463
## Number of obs: 40, groups:  PersonID, 20
##
```

```
## Fixed effects:
##                Estimate Std. Error     df t value Pr(>|t|)
## (Intercept)     32.882      1.766 19.170  18.618 9.79e-14 ***
## factor(Time)2   -1.942      0.236 19.000  -8.228 1.10e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##             (Intr)
## factor(Tm)2 -0.067
```

There is a lot of useful information here. The output tells us we have 40 observations from 20 **PersonID** groups, i.e. 20 people. And notice that we now have two standard deviations, where in a normal **lm()**, we just have the one residual standard error. We have a standard deviation of 7.86 at the level of the **PersonID**, and a residual standard error of 0.75. What this is telling us is that after we have fitted our model to the data, there is a lot of variation (7.86) between individuals (i.e. some individuals are more accurate and some less, regardless of time point), and a much smaller amount of variation (0.75) within individuals (i.e. you can't perfectly predict accuracy at the second time point just from knowing accuracy at the first time point, plus the average effect of time point). The between-individual standard deviation is much larger than the within-individual residual standard deviation, which confirms what we saw when we plotted the data.

Now we come to the estimate of the fixed effects. The parameter estimate for **factor(Time)** is -1.942. Notice that this is exactly the same as when we fitted a simple **lm()**. The difference is that the standard error of this parameter estimate is now much smaller (0.236 instead of 2.498 in the **lm()**). Why? Well now we are modelling the fact that each individual has a characteristic overall accuracy, and looking for deviations from that, so we are estimating the experimental effect we seek with much greater precision.

The confidence interval for the effect of time point now does not include zero. Getting this confidence interval is a slight faff (due to the class of object that the **mixed()** function returns), but here is a line of code that will do the job:

```
confint(mixed(Accuracy~factor(Time) + (1|PersonID), data=d, return="merMod"))
```

```
##                    2.5 %    97.5 %
## .sig01          5.7772187 10.839373
## .sigma          0.5492341  1.027570
## (Intercept)    29.3408561 36.423544
## factor(Time)2 -2.4150356 -1.468297
```

So people really do seem to be systematically worse at time 2, by about 1.94 points (95% confidence interval 1.47 to 2.42).

In linear mixed models, there are two ways of fitting the model, known as maximum likelihood (ML) and restricted maximum likelihood (REML). The parameter estimates should be fairly similar either way. There are also several ways of estimating p-values for linear mixed models. In the summary of **mx**, the p-values are generated using something called Satterthwaite's method. An alternative way of getting p-values is the likelihood ratio test (LRT). You have to use the LRT if you want a generalized linear mixed model, but it's an option in the normal case too. If you switch to LRT, the model fitting is automatically changed from REML to ML. Here's how you do it:

```
mx=mixed(Accuracy~factor(Time) + (1|PersonID), data=d, method="LRT")
```

```
## Numerical variables NOT centered on 0: Time
## If in interactions, interpretation of lower order (e.g., main) effects difficult.

## REML argument to lmer() set to FALSE for method = 'PB' or 'LRT'

## Fitting 2 (g)lmer() models:
## [..]
```

```
mx
```

```
## Mixed Model Anova Table (Type 3 tests, LRT-method)
##
## Model: Accuracy ~ factor(Time) + (1 | PersonID)
## Data: d
## Df full model: 4
##          Effect df     Chisq p.value
## 1 factor(Time)  1 30.36 ***  <.0001
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '+' 0.1 ' ' 1
```

The likelihood ratio test gives a value (here 30.36) that is compared to a χ^2 distribution with 1 degree of freedom.

I would report the results of what we have done as follows:

We fitted a linear mixed model with a varying intercept of person, and a fixed effect of time point, using restricted maximum likelihood estimation. Participants were less accurate at time 2 than time 1 (β_{time2} = -1.94, 95% CI -2.42 to -1.47; t-test using Satterthwaite's method, t_{19} = -8.23, p < 0.01) (You might wish to add: The between-person standard deviation was 7.86, and the residual standard deviation was 0.75.)

A note on data structure

You will note that the dataframe we have been using is in so called 'long and thin' or 'narrow' format. This means that the repeated observations from the same individual are on different rows, and the ones that go together are identified by the values of the variable **PersonID**. The number of rows in the dataframe is thus the number of observations, not the number of people. You might be more used to entering data in 'short' or 'wide' format, whereby each person is a row, and repeated observations are successive columns. A wide version of the present dataset would look like this.

```
##    PersonID Accuracy.1 Accuracy.2
## 1         1   41.26861   39.57513
## 2         2   30.23323   27.79155
## 3         3   34.81972   32.58701
## 4         4   34.18728   32.99018
## 5         5   36.49972   37.50081
## 6         6   30.45886   28.31217
## 7         7   21.30288   21.06290
## 8         8   35.39293   32.64569
## 9         9   30.59776   28.78532
## 10       10   21.28879   18.87109
## 11       11   28.76250   26.93711
## 12       12   25.97058   23.44192
## 13       13   47.44780   46.36153
## 14       14   27.98423   24.56982
## 15       15   47.36467   44.92028
## 16       16   34.33363   32.44295
## 17       17   45.12842   43.45231
## 18       18   23.65355   20.50936
## 19       19   28.76422   27.36066
## 20       20   32.18460   28.69289
```

Long format is almost always better for working with R, whether you are plotting graphs or running statistical models. If you have data sets that are already entered in wide format, don't worry. You can convert between

the formats with the function **reshape()** or with the 'tidyr' package: see Appendix, 'Reshaping data between wide and long formats'.

Going further with linear mixed models

We have looked at the very simplest case, where the data are grouped into pairs, but linear mixed models are extremely powerful and flexible. You could apply the same kinds of formula we have used here for cases with: more than two observations per group; more than one fixed predictor; and different numbers of observations per group. And you can easily build mixed models with: more than one source of non-independence, such as observations from the same or different pupils at the same or different schools; or more complex patterns of non-independence other than just different groups having different averages, for example if you wanted to build in that different individuals had different characteristic tendencies to get better or worse over time. This latter situation is what is called a 'random slopes model', and it follows the same general logic.

Generalized and mixed: The generalized linear mixed model

In this session, we have dealt with a generalized linear model, and we have dealt with a mixed model. The question is bound to have occurred to you: what if my error structure is not normal AND there is structure in my data? Well, in such a situation, you need a model that is both generalized and mixed: A generalized linear mixed model. You can fit such a model still using the 'afex' function **mixed()**. The syntax is something like the following:

```
m=mixed(response ~ predictors + (1|ID.variable), data=my.data.frame, family=binomial, method="LRT")
```

The only difference is the addition of **family=binomial**. Note that the **method="LRT"** is obligatory in the generalized linear mixed case whereas it was only an option in the linear mixed case.

Session 5: ANOVA and factorial experiments

If your work is experimental in nature, you may be used to an approach to statistical analysis called 'analysis of variance' or 'ANOVA'. You might well be familiar with the 'ANOVA table', where all your predictors have an F-ratio and two numbers representing degrees of freedom. Back in session 2, I told you that ANOVA was just one specific case of the general linear model. Well that's true: in doing ANOVA, you are asking whether the mean of your outcome variable is predicted by a linear sum of your predictors, as you always do in a general linear model. But the significance testing in ANOVA is done in a slightly different way to the standard output of an **lm()**. So you should think of the ANOVA table as a different way of asking about significance (and the variance explained by your predictors) within the general linear model. ANOVA finds its greatest utility in the analysis of designed experiments.

In this session, we explore ANOVA in R, primarily using the 'car' package. We will concentrate on when you might choose to use an ANOVA approach, and how it differs from just using **summary(lm( ))**.

A dataset, and experimental terminology

We need a dataset for this session, so we are going to load in 'blood.data.csv'.

```
d=read.csv("blood.data.csv")
head(d)
```

```
##   X Participant Group Diet   Drug     Blood Hemoglobin
## 1 1           1     1 Iron Drug A  8.270519   24.92110
## 2 2           2     1 Iron Drug A 10.456802   24.72211
## 3 3           3     1 Iron Drug A 10.174261   25.53018
## 4 4           4     1 Iron Drug A 10.432760   26.75365
## 5 5           5     1 Iron Drug A 10.945257   32.26370
## 6 6           6     1 Iron Drug A  8.654645   23.31615
```

This is an (invented) dataset from an experiment where 120 participants had their hemoglobin (variable **Hemoglobin**) levels measured after they had been taking one of three drugs (drug A, drug B or drug C); and following one of two diets (iron supplementation, or no iron supplementation). The research question is whether the drugs and diets affected hemoglobin levels.

The terminology of designed experiments can be a bit confusing, so let's get this straight. In this experiment, we have 6 experimental groups each consisting of 20 participants: drug A iron; drug A no iron; drug B iron; drug B no iron; drug C iron; and drug C no iron. However, we only have two experimental **treatments**: drug, and diet. The first experimental treatment, drug, has three **levels**, namely drug A, drug B and drug C. The second experimental treatment has two levels, namely iron and no iron. Each level of one treatment is applied at every level of the other: that is, there were groups taking all three drugs in combination with the iron supplementation diet, and also groups taking all three drugs in the no iron supplementation diet. A design like this, where every possible combination of the levels of the different treatments is observed, is known as a **full factorial design**. In a full factorial design, the number of experimental groups is the product of the numbers of levels of the treatments: here we have 2×3 levels, so we have $2 \times 3 = 6$ experimental groups.

First steps with ANOVA

Let's first imagine we were only interested in the effects of the diet treatment, and were going to ignore the drugs. We could fit a **lm()** in the normal way, as follows.

```
m1 = lm(Hemoglobin~Diet, data=d)
summary(m1)
```

```
##
## Call:
## lm(formula = Hemoglobin ~ Diet, data = d)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.9567 -1.1482 -0.2142  1.0874  6.2828
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  25.9809     0.2600  99.926  < 2e-16 ***
## DietNo iron   2.4283     0.3677   6.604 1.21e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.014 on 118 degrees of freedom
## Multiple R-squared:  0.2699, Adjusted R-squared:  0.2637
## F-statistic: 43.61 on 1 and 118 DF,  p-value: 1.205e-09
```

This seems to show a significant effect of the diet treatment on the mean hemoglobin. Now let us get an ANOVA table from the same model. We could do this with R's in-built function **anova()** (try **anova(m1)**), but for a number of reasons, I am going to prefer the **Anova()** function (note capital A) from the package 'car'. Install the 'car' package now if you have not already done so: **install.packages("car")**.

```
library(car)
Anova(m1, type="III")
```

```
## Anova Table (Type III tests)
##
## Response: Hemoglobin
##             Sum Sq  Df  F value    Pr(>F)
## (Intercept)  40500   1 9985.236 < 2.2e-16 ***
## Diet           177   1   43.613 1.205e-09 ***
## Residuals      479 118
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The line **type="III"** line in the code specifies 'type III significance tests'. This is not the only option (the default is type II for the **Anova()** function, and type I for R's in-built **anova()** function). We won't go into the difference between the different types here: it only makes a difference under certain conditions. For example, it never makes any difference when there is only one predictor variable. However, type III is quite typical in commercial statistical packages like SPSS, and so we will use it throughout.

From the ANOVA table for model **m1**, we would say: 'there was a significant effect of the diet treatment on mean hemoglobin ($F(1,118) = 43.61$, $p < 0.01$)'. So the outcome is much the same for the ANOVA and the **summary(lm())**. In fact, the outcome is *exactly* the same. Take the value of the t-test for the **Diet** parameter in the **summary(m1)** (6.604) and square it:

```
6.604^2
```

```
## [1] 43.61282
```

Hey presto, you get 43.61, which is the F-ratio from the ANOVA table. So the value of the t-test in the **summary(lm())** is just the square root of the F-ratio from the ANOVA.

Now let's do the same exercise but looking at the Drug treatment (and ignoring the Diet treatment for now). Using **summary(lm())**:

```r
m2=lm(Hemoglobin ~ Drug, data=d)
summary(m2)
```

```
##
## Call:
## lm(formula = Hemoglobin ~ Drug, data = d)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.8775 -1.4595  0.0813  1.6429  6.1121
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  26.5730     0.3636  73.078   <2e-16 ***
## DrugDrug B    1.3436     0.5142   2.613   0.0102 *
## DrugDrug C    0.5225     0.5142   1.016   0.3117
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.3 on 117 degrees of freedom
## Multiple R-squared:  0.05599,    Adjusted R-squared:  0.03985
## F-statistic:  3.47 on 2 and 117 DF,  p-value: 0.03437
```

Now we get *two* parameter estimates for **Drug**. That's because there are *three* levels of **Drug**, and the way we work out the drug effects on the mean hemoglobin is to estimate one parameter that is the difference between drug A and drug B, and a second parameter that is the difference between drug A and drug C. (There is no need for a separate parameter that compares B to C, since this information is fully contained in the first two parameters.) The significance tests associated with these parameter estimates therefore test whether A is different from B (it appears to be) and whether A is different from C (it appears not to be). You could use **relevel()** (see session 1) if you wanted to directly test whether B is different from C.

Now let's look at the same analysis through the lens of ANOVA.

```r
Anova(m2, type="III")
```

```
## Anova Table (Type III tests)
##
## Response: Hemoglobin
##              Sum Sq  Df   F value  Pr(>F)
## (Intercept) 28245.0   1 5340.3778 < 2e-16 ***
## Drug           36.7   2    3.4696 0.03437 *
## Residuals     618.8 117
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

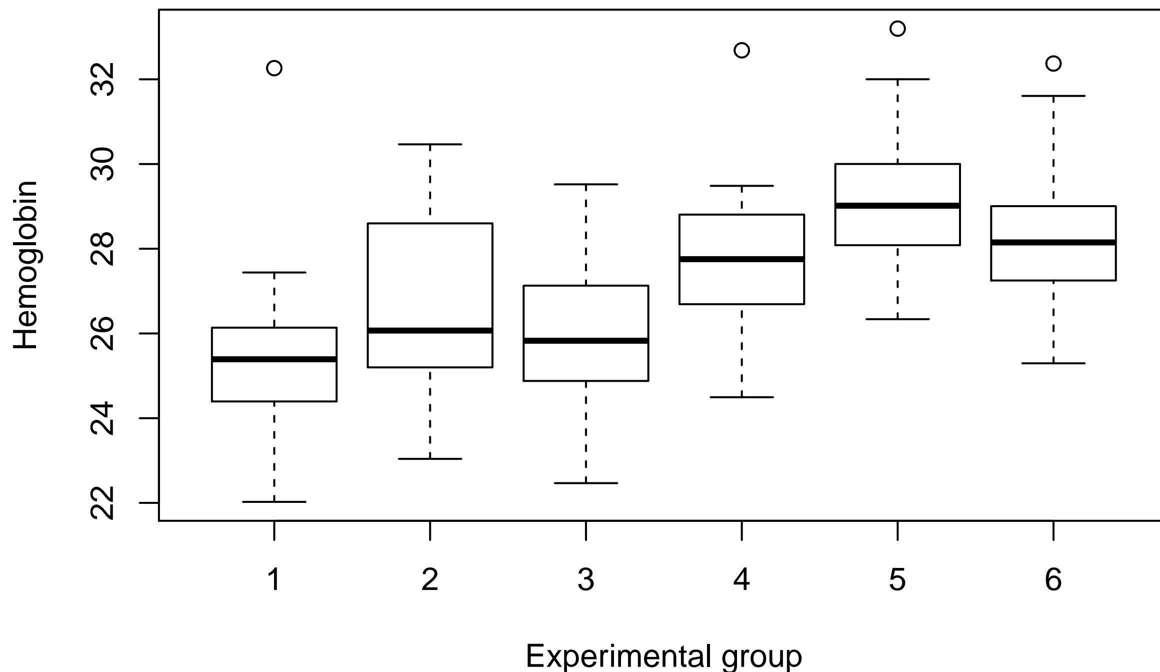
Now we get a *single* F-ratio for **Drug**, regardless of how many levels of **Drug** there are: $F(2, 117) = 3.47$, $p < 0.05$. What this tests is the null hypothesis that there is no difference in outcome between *any* of the drugs. To put this another way, the significance of this test at the 5% level suggests that at least one of the drugs produces a different mean hemoglobin from at least one of the others (to say which would require further investigation).

To sum up so far, where a treatment has more than two levels, the usual **lm()** approach to significance testing tests separately whether each drug in turn is different from the reference drug; whereas the ANOVA approach tests whether there is any difference between any of the drugs. These are not quite the same thing, though obviously related.

Multi-way ANOVA: Exploiting your factorial design

Recall that in our experiment there were six experimental groups (these are identified in the variable **Group**). Let's look at a boxplot of the hemoglobin levels in each group.

```
boxplot(d$Hemoglobin~d$Group, xlab="Experimental group", ylab="Hemoglobin")
```



It certainly looks like there is higher hemoglobin in some groups (e.g. 5) than others (e.g. 1). A lot of people would follow up by fitting an ANOVA like the following one:

```
Anova(lm(Hemoglobin ~ Group, data=d), type="III")
```

```
## Anova Table (Type III tests)
##
## Response: Hemoglobin
##              Sum Sq  Df  F value    Pr(>F)
## (Intercept) 14193.9   1 3406.326 < 2.2e-16 ***
## Group         163.8   1   39.312 6.138e-09 ***
## Residuals     491.7 118
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Yes, there is a significant difference between groups. But on deeper reflection, this is not very informative. Recall that the groups are constituted by *combinations* of drugs and diets. So the finding that some groups are different from some others does not tell us whether it is the drugs doing something, the diets doing something, or both.

In fact, the simple ANOVA using **Group** as the predictor fails to exploit the full power of the experimental design. In factorial designs, your power to estimate the effect of either one of your treatments is greater than

the size of your experimental groups. Think about testing the effect of diet. Although your experimental groups consist of only 20 people, you actually have a total of 60 people on the iron supplementation and 60 who are not. So for testing the effect of diet, you can collapse across the three levels of drug, and produce a powerful 60 against 60 comparison. By the same token, for testing the effects of drug, you have 40 people on each drug, which you can exploit by collapsing across the two diets. This means that, as long as the effects of drug and diet are independent from on another, it is as if you getting two experiments for the price of one: a 60 versus 60 test of diet, and a 40 vs. 40 vs. 40 test of drugs. The hypotheses you need to be testing are not 'the groups will differ' but 'collapsing across drugs, the outcomes for one diet will be different from the other' and 'collapsing across diets, the outcomes for the drugs will be different from one another'.

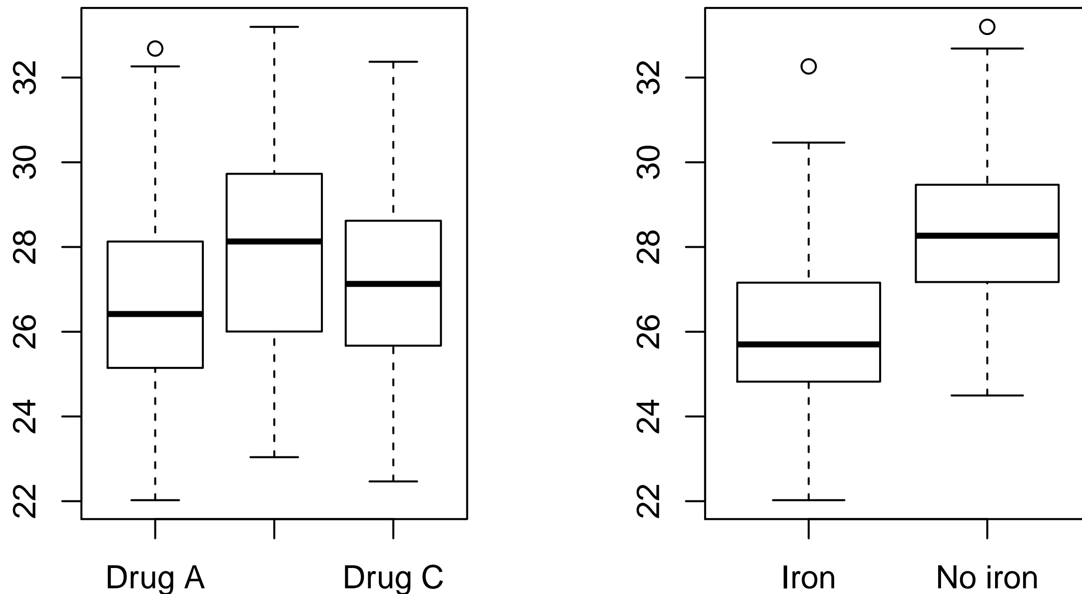
You do this by presenting a multi-way (in the present case, two-way) ANOVA table. On the right hand side of the model equation are all the things you varied independently in setting up your experimental design; and on the left hand side is the outcome:

```
m3 = lm(Hemoglobin ~ Drug + Diet, data=d)
Anova(m3, type="III")
```

```
## Anova Table (Type III tests)
##
## Response: Hemoglobin
##              Sum Sq  Df  F value    Pr(>F)
## (Intercept) 19292.1   1 5064.127 < 2.2e-16 ***
## Drug           36.7   2    4.817   0.00978 **
## Diet          176.9   1   46.435 4.478e-10 ***
## Residuals     441.9 116
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This ANOVA suggests that both drug ($F(2, 116) = 4.82$, $p < 0.01$) and diet ($F(1, 116) = 46.43$, $p = 0.01$) have separate effects on hemoglobin. We can confirm that this is the case by making boxplots again, but this time making one plot for the three levels of drug (collapsing across diets) and one plot for the two levels of diet (collapsing across drugs), as follows:

```
par(mfrow=c(1, 2))# This line sets up two plots on one row
boxplot(d$Hemoglobin~d$Drug)
boxplot(d$Hemoglobin~d$Diet)
```



Looks like the drug B and, separately, the no iron diet, produce higher hemoglobin.

The model underlying the ANOVA we have just done was *additive*. That is, it assumed that the diet has the same effect on mean hemoglobin regardless of which drug the person was taking. This might not be so: the diet might have a different effect in the presence of one drug than in the presence of another one. To test for this possibility, we need an *interactive* model (look back to session 3 if you want a refresher on additive and interactive models).

In session 3, I emphasised the value of centring your predictors whenever you have a model that includes interaction terms. This makes the main effects and the intercept more informative in such a model. But in session 3, the predictor we centred was a continuous one. Here, both our predictors are factors, so the idea of centring does not seem so relevant. Actually, it is.

The three-level predictor drug is represented as two variables in the **lm()** (these are known as dummy variables). Drug A has the values 0 and 0 for these two variables; drug B has 1 and 0; and drug C has 0 and 1. This means that in a model with an interaction between drug and diet, the main effect of diet will represent the effect of diet when both drug variables are 0, i.e. when the person is on drug A. But the hypothesis you want to test concerns the effect of diet *on average across all three drugs*, which is not the same thing. So really you want the mean of both drug variables to be zero, so that the main effect of diet represents the effect of diet when the person is on the 'average' drug.

To achieve what we want here, we need to adopt 'contrast coding'. In contrast coding, the variables representing the factors are cented on zero. For example, a contrast coding scheme for our diet variable here would be -1 for no iron and +1 for iron. A contrast coding scheme for drug again involves two variables: the first takes the value 1 for drug A, 0 for drug B, and -1 for drug C; and the second takes the value 0 for drug A, 1 for drug B, and -1 for drug C. Both variables have means of zero, and each drug is uniquely identified.

Let us set contrast coding for our predictor variables.

```
contrasts(d$Diet)=contr.sum(2)
contrasts(d$Drug)=contr.sum(3)
```

Note that for a model with no interaction terms, it would not make any difference for the ANOVA table whether we set contrast coding or not. But it does make a difference for our interactive model, which we specify as follows:

```
m4 = lm(Hemoglobin~Drug + Diet + Drug:Diet, data=d)
Anova(m4, type="III")
```

```
## Anova Table (Type III tests)
##
## Response: Hemoglobin
##             Sum Sq  Df    F value    Pr(>F)
## (Intercept)  88748   1 22897.7427 < 2.2e-16 ***
## Drug            37   2     4.7346   0.01059 *
## Diet           177   1    45.6406 6.309e-10 ***
## Drug:Diet        0   2     0.0081   0.99198
## Residuals      442 114
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Looks like there was no need for the interaction term. The interaction is basically zero, and far from significant. The significant main effects of drug and diet persist.

ANCOVA: Adding a covariate to your ANOVA

Quite often, even in a designed experiment, there is some variable that was not part of the experimental design but nonetheless explains some of the variation in the outcome. For example, in the present data, there was a little bit of variation in the volume of blood that the researchers managed to extract from different participants (variable **Blood** in the dataset). Though this is not a designed experimental treatment, it could explain some of the variation in the hemoglobin measurements.

People often get the idea that in such a case, it might be good to calculate a ratio in order to 'correct for' this non-designed variation. For example, they might divide the **Hemoglobin** value by the **Blood** variable and produce a 'Hemoglobin ratio' or 'amount of hemoglobin per unit of blood'.

Calculation of such ratios is a very bad idea, and you should never do it. There are several reasons why, but the most important one is this: if you do, your experimental effects will be impossible to interpret. Let's say we calculate such a ratio, and find a significant effect of the diet treatment on the ratio. We now have idea whether the iron diet changed the ratio by making the amount of hemoglobin less (i.e. decreasing the numerator of the ratio), or the amount of blood in the sample more (i.e. increasing the denominator of the ratio). We just won't know whether our treatment has affected the thing we care about (the amount of hemoglobin) or an incidental thing we do not care about (blood volume).

So what is the alternative? Always keep the actual outcome you care about as the sole term on the left hand side of your model equation. If there is a nuisance variable, you might well be able to ignore it. After all, should not be systematically related to any of your treatments, and so it is just a source of residual error, evenly spread across levels of your treatments. But if you do feel you need to control for it (for example, because it explains most of the variation in the outcome due to some simple scaling relationship), then add it to the *right hand side* of your equation instead, as a covariate. A covariate is basically 'an additional thing to statistically control for', and is usually a continuous numeric variable. ANOVA with one or more additional covariates is often referred to as ANCOVA.

To fit an ANCOVA to our present data, just set up the appropriate **lm()** and then get the ANOVA table:

```r
m5 = lm(Hemoglobin~Drug + Diet + Drug:Diet + Blood, data=d)
Anova(m5, type="III")
```

```
## Anova Table (Type III tests)
##
## Response: Hemoglobin
##             Sum Sq  Df  F value    Pr(>F)
## (Intercept) 357.45   1 103.2371 < 2.2e-16 ***
## Drug         45.39   2   6.5543 0.0020270 **
## Diet        184.90   1  53.4038 4.138e-11 ***
## Blood        50.60   1  14.6137 0.0002164 ***
## Drug:Diet     1.21   2   0.1745 0.8400953
## Residuals   391.25 113
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

What we see is that **Blood** does indeed significantly predict **Hemoglobin**, but even after controlling for this, there are significant effects of **Drug** and **Diet**. In this case, our conclusion is unchanged by adding a covariate.

ANOVA or not?

When should generate an ANOVA table, and when should you simply report the parameter estimates and their confidence intervals from your **summary(lm())**? This is a good question, and it is partly a matter of tradition and taste. I have tried to show you that the ANOVA (or ANCOVA) table is not a different thing from a general linear model, but a calculation that arises from it. I usually prefer to work from the **summary(lm())** as this naturally causes you to focus on the parameter estimates (and their precision), rather than p-values. Parameter estimation is ultimately what statistics is about, and F-ratios only tell you that your groups are different, not the magnitude or even direction of that difference. Many is the time a student has come to me with an ANOVA table and said 'look, I found a significant effect!', but when I asked, 'In what direction?', they could only say 'I don't know, the F-ratio doesn't tell me'.

The ANOVA table comes into its own in experiments where more than one treatment variable has more than two levels, especially where we wish to consider interactions. For example, if you have two predictor variables each with three levels, then the **summary(lm())** will contain two parameter estimates for each of the main effects, plus a further four parameter estimates for the interactions. It is hard to quickly see from this whether the two treatment variables overall made a difference to the outcome. I would be tempted in such a case to generate the ANOVA table, which would give me one F-ratio for each main effect, and a further one for the interaction, to give me the initial picture of what seems to have happened. However, this ANOVA table would not tell me, in the event of significant effects, which levels of each treatment variable were different from one another (or which levels of one treatment variable interacted with which levels of the other). I would need to follow up, using appropriate graphs and the **summary(lm())**, to shed light on this.

ANOVA tables for within-subjects or repeated-measures situations

The ANOVA example used in this session was a simple between-subjects design; that is, the people in each experimental group were different. Often though, we perform within-subjects or repeated-measures designs (the same people do a test more than once under different conditions); or mixed designs (where some things are varied within subjects and some between). We have already seen one example of a within-subjects design in this course: the sleep experiment in session 4. There we used a linear mixed model with a varying intercept of participant to capture the repeated-measures structure. You can use a linear mixed model in any situation where you have a within-subjects or mixed experimental design.

Just as you can generate an ANOVA table corresponding to an **lm()** model, you can generate an ANOVA table corresponding to an 'afex' **mixed()** model. Below I reproduce the code for loading the sleep data from session 4 and fitting a mixed model to it.

```
s=read.csv("sleep.data.csv")
library(afex)
mx=mixed(Accuracy~factor(Time) + (1|PersonID), data=s)
```

```
## Numerical variables NOT centered on 0: Time
## If in interactions, interpretation of lower order (e.g., main) effects difficult.
```

```
## Fitting one lmer() model. [DONE]
## Calculating p-values. [DONE]
```

```
summary(mx)
```

```
## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
## lmerModLmerTest]
## Formula: Accuracy ~ factor(Time) + (1 | PersonID)
##    Data: data
##
## REML criterion at convergence: 194.3
##
## Scaled residuals:
##      Min       1Q   Median       3Q      Max
## -1.94105 -0.38524  0.00887  0.34055  2.00219
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  PersonID (Intercept) 61.8320  7.8633
##  Residual              0.5569  0.7463
## Number of obs: 40, groups:  PersonID, 20
##
## Fixed effects:
##               Estimate Std. Error     df t value Pr(>|t|)
## (Intercept)     32.882      1.766 19.170  18.618 9.79e-14 ***
## factor(Time)2   -1.942      0.236 19.000  -8.228 1.10e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##             (Intr)
## factor(Tm)2 -0.067
```

The 'afex' package will happily generate you the corresponding ANOVA table, as follows:

```
mx$anova_table
```

```
## Mixed Model Anova Table (Type 3 tests, KR-method)
##
## Model: Accuracy ~ factor(Time) + (1 | PersonID)
## Data: s
##              num Df den Df      F    Pr(>F)
## factor(Time)      1     19 67.693 1.102e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The 'afex' package uses type-III sums of squares by default. Note once again that for this simple case, the

F-ratio in the ANOVA table is just the square of the t-value in the **summary()** of the mixed model.

Session 6: Model selection, model averaging, and meta-analysis

This session has two parts. The first part is on model selection and model averaging, and uses a contributed package called 'MuMIn'. The second part is on meta-analysis, and uses a contributed package called 'metafor'. It might be a good idea at this stage to install these packages: Type **install.packages("MuMIn")** and **install.packages("metafor")** at the command line when connected to the internet. Note the exact capitalization in the package names or you will get an error.

The data files we are going to use today are called 'GPA.csv' (for the first part of the session); and 'stress.2012.csv' and 'stress.2013.csv' (for the second part of the session). Make sure you have all three of these saved in your working directory before going any further.

Model selection

Model selection refers to a family of procedures for using the data to tell you which of several candidate models are favoured by the evidence. Central to model selection is the Akaike Information Criterion (AIC or AICc; the MuMIn package uses the AICc variant, but the logic of the two is the same), which we met briefly in session 3. This is a number that is sensitive to how much of the variation in the data the predictors in your model explain (other things equal, more variation explained produces a lower AIC); but also the complexity of your model (other things equal, a more complex model has a higher AIC than a simpler one). The AICs of any two models of exactly the same data are directly comparable (though comparing the AICs of models of different data sets is meaningless). Of several possible models of your data, the one with the lowest AIC is the one likely to best predict the patterns in new datasets from the same large world, and hence is the model that is most explanatory of whatever is going on. Model selection incorporates a trend in modern statistical practices in looking for and reporting continuous degrees of support for hypotheses, instead of Yes/No, 'either it is significant or it is not', kinds of answers. Very often, multiple different models come out with about the same AIC. Instead of choosing one model, you can then report *all* of the models, along with an indication of their relative AIC values.

Let's load in the data set we need for this part of the session ('GPA.csv'; it's based on a data set that comes with 'MuMIn', but I have modified is slightly for our purposes). The data set concerns predicting an outcome (students' college GPA scores) from up to four predictors (their SAT maths score, their SAT verbal score, their high school maths grade, and their high school English grade). You will see that I have already standardized all of the predictors and the outcome variable so that they have a mean of zero and a standard deviation of 1.

```
d=read.csv("GPA.csv")
head(d)
```

```
##   X         GPA  SAT.maths SAT.verbal   HS.maths HS.English
## 1 1 -1.00194690 -1.1468084 -1.7437494 -0.6496626 -0.7839121
## 2 2  0.23641444  1.2418744 -0.3185877  1.1093516  1.1577937
## 3 3 -0.64812938 -0.9241856  0.7521687  0.1477572 -0.9078507
## 4 4  0.01125783 -0.6534281 -0.2356418  0.8513629 -1.6514827
## 5 5  0.62239719  0.7725614  0.6390606  0.6168277  0.9718857
## 6 6 -1.51659057 -1.6522224 -1.0273983 -1.6112570 -1.7960779
```

Let's imagine that we are analysing the GPA data, and that there are two theoretical questions we want to answer. The first is 'what type of skills matter?'. There are two camps in the literature. Camp A says that all that is really important in predicting which students will do well at college is just their mathematical skills: students with good mathematical skills get high GPAs. Camp B says that maths skills are not the whole picture. This camp says that verbal skills contribute too, so that other things being equal, a person with better verbal skills will get a higher GPA.

The second question is 'which is the better measure of skills, SAT or high school grades?'. Again there are two camps. Camp (i) says SAT scores are better measures of skills, since they are from standardized tests.

Camp (ii) says high school grades are better measures of skills, since they come from teachers who know the students well and have worked with them over the long term.

Because we have two theoretical questions, each of which has two possible answers, we actually have four possible outcomes in the data that would be of interest:

1. SAT scores are better than high school grades at predicting GPA, and you only need the maths component of the SAT. Camps A and (i) are right.
2. SAT scores are better than high school grades at predicting GPA, and you need both the maths and verbal components of the SAT. Camps B and (i) are right.
3. High school grades are better than SAT scores at predicting GPA, and you only need the maths grade. Camps A and (ii) are right.
4. High school grades are better than SAT scores at predicting GPA, and you need both the maths and English grades. Camps B and (ii) are right.

The traditional way we would answer this is to test the predictions of the various theories against the null hypothesis that there is no association with GPA, So, for example, to test whether SAT scores predict GPA. you would run a model predicting GPA from **SAT.maths** and **SAT.verbal**.

```
m = lm(GPA~SAT.maths + SAT.verbal, data=d)
summary(m)
```

```
##
## Call:
## lm(formula = GPA ~ SAT.maths + SAT.verbal, data = d)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.68872 -0.25184 -0.03858  0.36492  0.77813
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 2.045e-10  1.028e-01   0.000   1.0000
## SAT.maths   6.965e-01  1.185e-01   5.879 1.82e-05 ***
## SAT.verbal  3.357e-01  1.185e-01   2.834   0.0115 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4596 on 17 degrees of freedom
## Multiple R-squared:  0.811,  Adjusted R-squared:  0.7888
## F-statistic: 36.47 on 2 and 17 DF,  p-value: 7.079e-07
```

Significant effects for both predictors! So it seems like the answer to our question is that camps B and (i) are correct.

Now what about possibility 4? We might run a model with high school maths and high school English, like this one:

```
m = lm(GPA~HS.maths + HS.English, data=d)
summary(m)
```

```
##
## Call:
## lm(formula = GPA ~ HS.maths + HS.English, data = d)
##
## Residuals:
```

```
##      Min       1Q   Median       3Q      Max
## -0.9201 -0.3611 -0.1143  0.3285  1.4627
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) 2.309e-10  1.356e-01   0.000  1.00000
## HS.maths    5.721e-01  1.446e-01   3.956  0.00102 **
## HS.English  4.508e-01  1.446e-01   3.117  0.00627 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6065 on 17 degrees of freedom
## Multiple R-squared:  0.6708, Adjusted R-squared:  0.6321
## F-statistic: 17.32 on 2 and 17 DF,  p-value: 7.908e-05
```

Significant effects of both predictors again! So it looks like our data support camps B and (i) (from model **m1**), and also B and (ii) (from model **m2**). So although we have apparently adjudicated in the question of A versus B, we seem to have supported *both* (i) and (ii), when they really ought to be mutually exclusive possibilities.

We might think the solution is another model with **SAT.maths** and **SAT.verbal** and **HS.maths** and **HS.English** *all* entered as predictors together. We might think this will adjudicate between (i) and (ii). Careful though - it does not exactly do so. In such a model, the parameter estimate for **SAT.maths** is the predictive power of **SAT.maths** for GPA after **SAT.verbal**, **HS.maths** and **HS.English** have all been controlled for, whereas the parameter estimate for **HS.maths** is the predictive power of **HS.maths** when **SAT.maths**, **SAT.verbal** and **HS.English** have all been controlled for. If **SAT.maths** and **HS.maths** are highly correlated to one another (which they are), it is quite possible that *neither* will have an effect significantly different from zero when you test for the effect of the one after the other has been controlled for (see session 2). But that's not the question posed in the debate between (i) and (ii). The question is, considered singly, which is the better predictor, a SAT score or a high school grade? Also, the model with all four variables entered simultaenously is quite complex (4 predictor variables plus the intercept), and our data set is small (20 cases). As we increase the number of parameters to estimate, the precision with which we can estimate each one declines, so if we keep adding more and more variables in, we will soon enough end up concluding that no parameter is significantly different from zero.

Taking a model selection approach

We were all taught that we test scientific predictions by setting up a 'straw man' null hypothesis, in this case that each of our variables has no value in predicting GPA, and then seeing if that null hypothesis is implausible in the light of the data. This is quite a strange way of going about things though. In the current case, we have four possible predictions, none of which is really a null hypothesis. So showing that the null hypothesis is *wrong* does not help much with saying which of the interesting theories is more likely to be *right*. It would be much better to be able to assess the strength of support for each of the predictions *against the other three predictions*, not just the support for each in turn against a null hypothesis. First, we are going to set up our four models of theoretical interest. These models are the direct operationalisations of possibilities 1-4 listed above.

```
m1 = lm(GPA~SAT.maths, data=d)
m2 = lm(GPA~SAT.maths + SAT.verbal, data=d)
m3 = lm(GPA~HS.maths, data=d)
m4 = lm(GPA~HS.maths + HS.English, data=d)
```

Now, I am also going to add a fifth model, which is the null hypothesis. This is important, because if the model selection tells us that **m1-m4** are about equally good in explaining GPA, then it would help to further know if that is because they are all really good at predicting GPA, or because they are all equally useless.

Comparing them to a null hypothesis allows us to assess this. Note that this is still rather different from conventional null hypothesis significance testing, because the various models will be tested against one another *as well as* against the null hypothesis. Here's how we make the null hypothesis model:

```
m5 = lm(GPA~1, data=d)
```

The `~1` on the right-hand side of the formula means 'just an intercept'.

Now we have our models set up, we activate the MuMIn package, and pass the list of models we want to its **model.sel()** function. Before we get the table, we have to set R's options concerning what to do in the event of there being any missing data. You only have to do this once per session; it is to stop **model.sel()** comparing models fitted to different subsets of the data, which would be invalid. By the way, there is more on handling missing data in R in Appendix, 'Handling missing data'.

```
library(MuMIn)
options(na.action="na.fail")
model.sel(list(m1, m2, m3, m4, m5))
```

```
## Model selection table
##       (Int) SAT.mth SAT.vrb HS.mth HS.Eng df  logLik AICc delta weight
## 2 2.045e-10  0.6965  0.3357                4 -11.206 33.1  0.00  0.904
## 1 2.274e-10  0.8495                        3 -15.075 37.6  4.57  0.092
## 4 2.309e-10                 0.5721 0.4508  4 -16.754 44.2 11.10  0.004
## 3 2.042e-10                 0.6947         3 -21.276 50.1 16.97  0.000
## 5 1.000e-10                                2 -27.866 60.4 27.36  0.000
## Models ranked by AICc(x)
```

This table ranks the five models by AICc. The model that comes out best is **m2**. The **delta** column tells you how far behind each other model comes, and the weight column can be interpreted as the degree of support from the data for each model being the best of this set. So, basically, the support splits 90% for possibility B (i), 9% for A (i), and negligible support for both of the (ii)s. There is no support for the null **m5** either: it was much the worst of the models under consideration. This means *all* of the non-null models would have had at least one predictor significantly different from zero if we had proceeded in the conventional null hypothesis significance testing manner.

The moral of this story is that both **m2** and **m4** had two significant predictors, and so it would have been easy to conclude that the data suggest that high school grades and SAT scores are equally good predictors of GPA. This is not right: although the data show that both high school grades and SAT scores are better than no information, they clearly show that the SAT scores do better than the high school grades in a direct comparison. And the direct comparison is what the research question demanded.

Dredging: Automated model generation

Writing out all the models to compare can be tedious, especially if there are a large number to be considered. You can partly automate this using a 'MuMIn function' called **dredge()**. This allows you to specify the most complex model you are prepared to consider (which is called the *global model*): what gets considered is all possible simpler sub-models of the global model.

I'll now do the analysis of the GPA dataset again, using this dredging approach, with the global model defined as the model containing all four predictors and no interactions. I am not saying that this is the right analysis in this particular case by the way; it's just an illustration of the method.

Our global model will, for the sake of the example, be the additive model in which all four of the predictors are present. Let's call it **g**.

```
g = lm(GPA ~ SAT.maths + SAT.verbal + HS.maths + HS.English, data=d)
```

The command **dredge(g)** now generates all of the possible sub-models of **g**, and then we pass the resulting list into the function **model.sel()** to get the model selection table.

```
model.sel(dredge(g))
```

```
## Fixed term is "(Intercept)"

## Global model call: lm(formula = GPA ~ SAT.maths + SAT.verbal + HS.maths + HS.English,
##     data = d)
## ---
## Model selection table
##         (Int)   HS.Eng HS.mth SAT.mth SAT.vrb df  logLik AICc delta weight
## 15 2.246e-10          0.2467  0.5840  0.2799  5  -8.871 32.0  0.00  0.454
## 13 2.045e-10                  0.6965  0.3357  4 -11.206 33.1  1.05  0.268
## 7  2.486e-10          0.3197  0.6708          4 -12.165 35.0  2.97  0.103
## 16 2.264e-10 0.068180 0.2598  0.5373  0.2671  6  -8.708 35.9  3.85  0.066
## 14 2.046e-10 0.008677         0.6913  0.3345  5 -11.204 36.7  4.67  0.044
## 8  2.499e-10 0.136700 0.3393  0.5692          5 -11.662 37.6  5.58  0.028
## 5  2.274e-10                  0.8495          3 -15.075 37.6  5.62  0.027
## 6  2.275e-10 0.077020         0.7985          4 -14.955 40.6  8.55  0.006
## 12 2.053e-10 0.352900 0.4669          0.3034  5 -14.523 43.3 11.30  0.002
## 4  2.309e-10 0.450800 0.5721                  4 -16.754 44.2 12.15  0.001
## 11 1.760e-10          0.5066          0.4333  4 -17.787 46.2 14.21  0.000
## 10 1.404e-10 0.404400                 0.4847  4 -19.612 49.9 17.86  0.000
## 3  2.042e-10          0.6947                  3 -21.276 50.1 18.02  0.000
## 9  1.000e-10                          0.6533  3 -22.301 52.1 20.07  0.000
## 2  1.606e-10 0.606500                         3 -23.280 54.1 22.03  0.000
## 1  1.000e-10                                  2 -27.866 60.4 28.41  0.000
## Models ranked by AICc(x)
```

In this larger set, the model with the lowest AICc, at 32.0, is a model with **HS.maths**, **SAT.maths** and **SAT.verbal** in it, but not **HS.English**. Close behind with an AICc of 33.1 is a model that also omits **HS.maths**. Note that (1) the most favoured model here was not even in the hand-picked set before; that's because it does not correspond straightforwardly to any of our theoretical predictions, and hence we did not include it before; and (2) the AICc weights for models that *were* in our set before are not the same now. This makes sense because an AICc weight is defined for a particular model in relation to a particular comparison set. Change the comparison set, and you change the weight. In this instance, the first model (**HS.maths** + **SAT.maths** + **SAT.verbal**) gets about 45% of the support; the next (**SAT.maths** + **SAT.verbal**) gets 27%; the third (**HS.maths** + **SAT.maths**) gets 10%; and nothing else gets more than 1%.

Estimating parameters using model averaging

Now we want to get the definitive parameter estimates to report in our write-up for the associations between our predictors and GPA. The exact answer we get will depend on which model we opt for: for example, 0.58 for **SAT.maths** if we go for the first model; 0.70 if we go for the second; and 0.67 if we go for the third. However, rather than having to choose one model or the other, we can just average the parameter estimates from all of the models together. Of course, it does not make sense to give a bad model of the data the same weight as a good model in coming up with this average. So instead we can average the parameter estimates weighted by the final column of the model selection table, so the top one gets 45% of the weight, the next 27%, the next 10%, and the others a negligible amount. We do this with the function **model.avg()**.

```
summary(model.avg(dredge(g)))
```

```
## Fixed term is "(Intercept)"

##
```

```
## Call:
## model.avg(object = dredge(g))
##
## Component model call:
## lm(formula = GPA ~ <16 unique rhs>, data = d)
##
## Component models:
##        df logLik  AICc delta weight
## 234     5  -8.87 32.03  0.00   0.45
## 34      4 -11.21 33.08  1.05   0.27
## 23      4 -12.17 35.00  2.97   0.10
## 1234    6  -8.71 35.88  3.85   0.07
## 134     5 -11.20 36.69  4.67   0.04
## 123     5 -11.66 37.61  5.58   0.03
## 3       3 -15.07 37.65  5.62   0.03
## 13      4 -14.95 40.58  8.55   0.01
## 124     5 -14.52 43.33 11.30   0.00
## 12      4 -16.75 44.17 12.15   0.00
## 24      4 -17.79 46.24 14.21   0.00
## 14      4 -19.61 49.89 17.86   0.00
## 2       3 -21.28 50.05 18.02   0.00
## 4       3 -22.30 52.10 20.07   0.00
## 1       3 -23.28 54.06 22.03   0.00
## (Null)  2 -27.87 60.44 28.41   0.00
##
## Term codes:
## HS.English   HS.maths  SAT.maths SAT.verbal
##          1          2          3          4
##
## Model-averaged coefficients:
## (full average)
##              Estimate Std. Error Adjusted SE z value Pr(>|z|)
## (Intercept) 2.216e-10  1.003e-01   1.082e-01   0.000    1.000
## HS.maths    1.731e-01  1.634e-01   1.685e-01   1.027    0.304
## SAT.maths   6.311e-01  1.493e-01   1.582e-01   3.990 6.61e-05 ***
## SAT.verbal  2.503e-01  1.555e-01   1.613e-01   1.551    0.121
## HS.English  1.025e-02  6.529e-02   6.928e-02   0.148    0.882
##
## (conditional average)
##              Estimate Std. Error Adjusted SE z value Pr(>|z|)
## (Intercept) 2.216e-10  1.003e-01   1.082e-01   0.000   1.0000
## HS.maths    2.647e-01  1.287e-01   1.384e-01   1.912   0.0559 .
## SAT.maths   6.331e-01  1.452e-01   1.544e-01   4.101 4.12e-05 ***
## SAT.verbal  2.999e-01  1.187e-01   1.278e-01   2.347   0.0189 *
## HS.English  6.969e-02  1.576e-01   1.688e-01   0.413   0.6796
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Relative variable importance:
##                      SAT.maths SAT.verbal HS.maths HS.English
## Importance:          1.00      0.83       0.65     0.15
## N containing models:    8         8          8        8
```

The numbers to look for for the parameter estimates are the 'full average' values. For **SAT.maths**, what the

table tells us is that across the whole set of models weighted by AICc, the parameter estimate for **SAT.maths** is 0.63, with a standard error of 0.15. I find the final 'relative variable importance' table helpful too. It tells us for each predictor the cumulative weight given to models containing that predictor. As you can see, SAT.maths gets a cumulative weight of basically 1, but **HS.English** a very low weight (0.15). This reiterates our conclusion that the SAT variables are more important than the high school ones, and maths more important than English/verbal.

The 'conditional average' table gives you another set of parameter estimates, based on averaging only those models in which the predictor in question does appear. This can give different estimates in some circumstances to the full average. I favour the 'full average' parameter estimate, since the other one inflates the estimate for predictors that are not in fact very important. In the present case, you will see that the parameter estimate for **SAT.maths** is almost the same in the 'full' and 'conditional'. That is because **SAT.maths** is present in all the models that get any appreciable amount of AICc weight anyway, so it does not make much difference. By contrast, **HS.English**, which is absent from all the best models, gets a much higher 'conditional' than 'full' estimate.

Best-models subsets

In practice, people would very rarely average together a huge set of models as we have just done. It can be cumbersome, because in complex cases there are literally hundreds of models to consider. People usually make a 'best models subset' before averaging. By convention, this consists of all of the models within two AICc units of the best model (in terms of our table, the **delta** value has to be less than 2). Here's how you get the 'best models subset' model-averaged parameter estimates.

```
summary(model.avg(dredge(g), delta<2))
```

```
## Fixed term is "(Intercept)"
## Fixed term is "(Intercept)"
```

```
##
## Call:
## model.avg(object = dredge(g), subset = delta < 2)
##
## Component model call:
## lm(formula = GPA ~ <2 unique rhs>, data = d)
##
## Component models:
##     df logLik  AICc delta weight
## 123  5  -8.87 32.03  0.00   0.63
## 23   4 -11.21 33.08  1.05   0.37
##
## Term codes:
##   HS.maths  SAT.maths SAT.verbal
##          1          2          3
##
## Model-averaged coefficients:
## (full average)
##              Estimate Std. Error Adjusted SE z value Pr(>|z|)
## (Intercept) 2.171e-10  9.751e-02   1.053e-01   0.000   1.0000
## HS.maths    1.550e-01  1.526e-01   1.576e-01   0.984   0.3253
## SAT.maths   6.258e-01  1.322e-01   1.410e-01   4.438  9.1e-06 ***
## SAT.verbal  3.006e-01  1.176e-01   1.265e-01   2.377   0.0175 *
##
## (conditional average)
##              Estimate Std. Error Adjusted SE z value Pr(>|z|)
```

```
## (Intercept) 2.171e-10  9.751e-02   1.053e-01   0.000   1.0000
## HS.maths    2.467e-01  1.202e-01   1.301e-01   1.897   0.0579 .
## SAT.maths   6.258e-01  1.322e-01   1.410e-01   4.438  9.1e-06 ***
## SAT.verbal  3.006e-01  1.176e-01   1.265e-01   2.377   0.0175 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Relative variable importance:
##                      SAT.maths SAT.verbal HS.maths
## Importance:          1.00      1.00       0.63
## N containing models:    2         2          1
```

So now there are just the best two models in the averaging. The parameter estimate for **SAT.maths** is virtually identical to last time, since the models we excluded under the subsetting had such low weights anyway. You will note however that **HS.English** has totally disappeared; it did not appear in any of the models in the best-models subset, and hence we would conclude that it is unimportant.

Reporting a model selection and model averaging

Here's how I would report what we have done in this (dredging) section:

To investigate which factors predicted GPA score, we performed model selection based on AICc on all possible additive models containing any or all of the set of predictors: SAT maths score; SAT verbal score; high school maths score; and high school English score (16 models in total). All variables were standardized prior to analysis. We retained all models within 2 AICc units of the lowest value in a best-models subset. The best-models subset contained a model with SAT maths score, SAT verbal score, and high school maths score as predictors (AICc weight 0.63); and a model with just SAT maths score and SAT verbal score as predictors (AICc weight 0.37). The cumulative AICc weights for models containing SAT maths score and SAT verbal score were therefore 1.00; and the cumulative AICc weight for models containing high school maths score was 0.37. High school English score did not appear in any of the best models. Model-averaged parameter estimates were as follows: SAT maths score 0.63 (SE 0.13); SAT verbal score 0.30 (SE 0.12); high school maths score 0.16 (SE 0.15).

To my mind this provides a lot more information, and is a lot more robust, than just reporting one model of the data, along with p-values.

Taking it further with model selection and model averaging

Model selection and model averaging are increasingly widely used, especially in fields like ecology and social science, where many predictors are potentially available, and there are competing theories about which ones should be important. They are less widely used in fields with designed experiments. With designed experiments, you often know a priori what the right set of predictors is, namely exactly those independent variables whose value you manipulated experimentally (see session 5). But even here, model selection and model averaging can be useful under certain circumstances.

A good reference for learning more about model selection and model averaging is: Symonds, M. R. E., & Moussalli, A. (2010). A brief guide to model selection, multimodel inference and model averaging in behavioural ecology using Akaike's information criterion. *Behavioral Ecology and Sociobiology*, 65, 13-21.

Meta-analysis

Quite often, multiple studies are done that test the same hypothesis. It would be nice to have a formal way of synthesizing their results, in particular a more quantitative way than just saying 'this study finds

a significant effect, that study does not'. That is, we want to be able to answer questions like: given the combined evidence from *all* the studies, what is our best estimate for the association between A and B? And: How similar are the findings across the studies?

The combined analysis of multiple data sets is called *meta-analysis*, and it is very easy to perform in R. We are going to use a contributed package called 'metafor' (install now if you have not done so already).

Stressed starlings

The data we are going to consider come from two recent starling studies of ours (for the original paper, see Andrews, C. et al. (2017). A marker of biological age explains individual variation in the strength of the adult stress response. *Royal Society Open Science*, 4, 171208; I have simplified the data set for present purposes).

In two successive cohorts of starlings (one in 2012, one in 2013), we measured a marker of biological aging. (This was the extent to which the birds' telomeres had shortened over their early lives, or **telomere.change**; note that a higher value of telomere.change means better preserved telomeres and hence younger, not older, biological age). In each study, we then measured their stress responses. The stress-response variable we will consider here is **peakcort**, the highest level of the stress hormone corticosterone in their blood after exposure to a stressor.

Let's load in the 2012 data.

```
d12 = read.csv("stress.2012.csv")
head(d12)
```

```
##   X BirdID telomere.change agedays peakcort
## 1 1   DBDB       0.1771173     293 72.04825
## 2 2   DBOR      -0.2196555     214  8.64343
## 3 3   DBPB       0.4852822     359 23.61482
## 4 4   DBPI      -0.1755098     214 37.57925
## 5 5   DBPU              NA     359 66.37224
## 6 6   DBYE      -0.4277878     429 20.93456
```

Now the 2013 data:

```
d13 = read.csv("stress.2013.csv")
head(d13)
```

```
##   X BirdID telomere.change peakcort
## 1 1   BL17      0.09641425 18.68966
## 2 2   BL18     -0.10270581 16.19778
## 3 3   BL22     -0.05959383 16.85029
## 4 4   BL23      0.02670418 10.36971
## 5 5   BL25      0.25951293 10.80436
## 6 6   BL26     -0.38095153 21.96393
```

Now let's ask whether **telomere.change** predicts **peakcort**. We have missing values in these datasets, so we are first going to need to reset our default action for missing values (you may recall we changed this for doing model selection; see Appendix for more information on handling missing data).

```
options(na.action="na.omit")
```

Now, the 2012 study.

```
m12 = lm(peakcort ~ telomere.change, data=d12)
summary(m12)
```

```
##
## Call:
```

```
## lm(formula = peakcort ~ telomere.change, data = d12)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -51.089 -19.374  -1.546  20.084  66.690
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)       41.914      6.472   6.476 4.31e-06 ***
## telomere.change   28.847     13.123   2.198   0.0413 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 28.94 on 18 degrees of freedom
##   (12 observations deleted due to missingness)
## Multiple R-squared:  0.2116, Adjusted R-squared:  0.1678
## F-statistic: 4.832 on 1 and 18 DF,  p-value: 0.04126
```

So it looks like in the 2012 data, as **telomere.change** goes up, so too does **peakcort**; and the parameter estimate is just about significantly different from zero at the $p < 0.05$ level.

Now the 2013 data:

```
m13 = lm(peakcort ~ telomere.change, data=d13)
summary(m13)
```

```
##
## Call:
## lm(formula = peakcort ~ telomere.change, data = d13)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.4630 -3.5115  0.2994  1.9247 13.7916
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)       16.264      0.924  17.602   <2e-16 ***
## telomere.change    3.563      3.544   1.005    0.323
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.094 on 29 degrees of freedom
##   (5 observations deleted due to missingness)
## Multiple R-squared:  0.03369,    Adjusted R-squared:  0.0003652
## F-statistic: 1.011 on 1 and 29 DF,  p-value: 0.323
```

In the 2013 data, the parameter estimate for **telomere.change** is not significantly different from zero, though it is the same side of zero (positive) as it was for the 2012 data. So now the question we want to ask is, on the basis of these two data sets *combined*, what is our best parameter estimate for association between **telomere.change** and peakcort (and what is the precision of that estimate). We are going to establish this by meta-analysing the two datasets. An alternative approach would be simply to merge the two datasets into one. In this case, we did not want to do this, because although the study question was the same in the two studies, a lot of the details of the methods were too different to treat this as one big dataset.

The first thing we need to establish is that **peakcort** and **telomere.change** were measured on comparable scales in the two studies. If they were not, the meta-analysis will not make much sense (by comparison, what sense would it make if you averaged together two estimates for the height of oak trees, one of which was in

feet and the other in metres?). In fact, our lab procedures were different in the two years, and the average **peakcort** values are very different across the two datasets. So to make them comparable, we will standardize all the variables in our models of each dataset individually. This means that the parameter estimates are on the standard deviation scale in both cases. This means we are are asking: for each standard deviation that **telomere.change** goes up, how many standard deviations does **peakcort** go up. So let's recompute the models:

```
m12 = lm(scale(peakcort) ~ scale(telomere.change), data=d12)
summary(m12)
```

```
##
## Call:
## lm(formula = scale(peakcort) ~ scale(telomere.change), data = d12)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.74078 -0.66013 -0.05269  0.68433  2.27236
##
## Coefficients:
##                        Estimate Std. Error t value Pr(>|t|)
## (Intercept)             0.02653    0.22051   0.120   0.9056
## scale(telomere.change)  0.48662    0.22138   2.198   0.0413 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.986 on 18 degrees of freedom
##   (12 observations deleted due to missingness)
## Multiple R-squared:  0.2116, Adjusted R-squared:  0.1678
## F-statistic: 4.832 on 1 and 18 DF,  p-value: 0.04126
```

```
m13 = lm(scale(peakcort) ~ scale(telomere.change), data=d13)
summary(m13)
```

```
##
## Call:
## lm(formula = scale(peakcort) ~ scale(telomere.change), data = d13)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.26841 -0.68915  0.05877  0.37773  2.70669
##
## Coefficients:
##                        Estimate Std. Error t value Pr(>|t|)
## (Intercept)            -0.06302    0.17971  -0.351    0.728
## scale(telomere.change)  0.18070    0.17972   1.005    0.323
##
## Residual standard error: 0.9997 on 29 degrees of freedom
##   (5 observations deleted due to missingness)
## Multiple R-squared:  0.03369,    Adjusted R-squared:  0.0003652
## F-statistic: 1.011 on 1 and 29 DF,  p-value: 0.323
```

Note that the parameter estimates in the two datasets are rather more similar to one another now we have standardized.

Performing the meta-analysis

To proceed with the meta-analysis, we pass to a 'metafor' function called **rma()** the output of our models of each dataset separately, **m12** and **m13**. In particular, we need from each model the parameter estimate for **telomere.change** and its standard error. We can read these off the summaries we have already seen, but there is no need to type them again by hand. Instead, we can find them within the stored model objects.

To understand how, you need to know that **summary(m12)$coefficients** returns a 2 by 4 array with, on the first row, the Estimate, Std. Error, t-value and p-value for the intercept; and, on the second row, the same outputs for **scale(telomere.change)**.

```
summary(m12)$coefficients
```

```
##                          Estimate Std. Error   t value   Pr(>|t|)
## (Intercept)            0.02652672  0.2205133 0.1202953 0.90558180
## scale(telomere.change) 0.48662334  0.2213764 2.1981716 0.04125862
```

So if we want to pull out the parameter estimate for telomere.change, it is going to be the first element of the second row of this array.

```
summary(m12)$coefficients[2, 1]
```

```
## [1] 0.4866233
```

And the second element of the second row will be the standard error.

```
summary(m12)$coefficients[2, 2]
```

```
## [1] 0.2213764
```

The same logic can be used for model **m13**.

For the meta-analysis, we want a vector of parameter estimates from our different datasets. Given what we have just done, that's easy to specify (the function **c()** means combine, remember):

```
estimates=c(summary(m12)$coefficients[2, 1], summary(m13)$coefficients[2, 1])
```

We also want a vector of standard errors. Again, that's easy.

```
serrors=c(summary(m12)$coefficients[2, 2], summary(m13)$coefficients[2, 2])
```

Now, we activate the package 'metafor', and pass our vectors of estimates and standard errors to its **rma()** function. By the way, this is what is called a 'random effects' meta-analysis. This means it assumes that there is methodological variability between studies, and hence that even with infinite sample sizes of all the studies, they would not all produce a numerically identical result. There are alternative assumptions available within 'metafor' (e.g. the fixed-effects meta-analysis), but random-effects is widely used and sensible for many purposes.

```
library(metafor)
ma = rma(yi = estimates, sei= serrors)
```

Now let us look at the summary of the meta-analytic model:

```
summary(ma)
```

```
##
## Random-Effects Model (k = 2; tau^2 estimator: REML)
##
##   logLik  deviance       AIC       BIC      AICc
##   0.1121   -0.2241    3.7759   -0.2241   15.7759
##
## tau^2 (estimated amount of total heterogeneity): 0.0061 (SE = 0.0662)
```

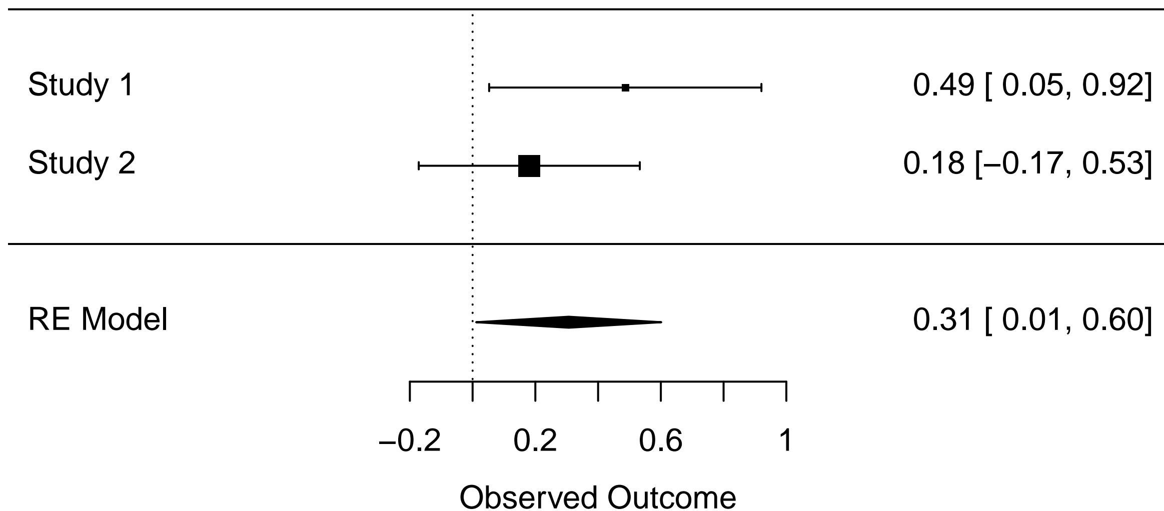
```
## tau (square root of estimated tau^2 value):      0.0784
## I^2 (total heterogeneity / total variability):   13.12%
## H^2 (total variability / sampling variability):  1.15
##
## Test for Heterogeneity:
## Q(df = 1) = 1.1510, p-val = 0.2833
##
## Model Results:
##
## estimate      se    zval    pval   ci.lb   ci.ub
##   0.3064  0.1505  2.0355  0.0418  0.0114  0.6013  *
##
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

What this tells us is that in the two datasets combined, the best estimate for the association between **telomere change** and **peakcort** is 0.31, with a 95% confidence interval of 0.01 to 0.60. And this is just significantly different from zero at the $p < 0.05$ level. There are also many other useful statistics we will not go into here. The thing to note is that this is a better-powered estimate than those provided by either dataset individually.

Making a forest plot

A useful plot you can get from a meta-analysis is called a forest plot. Let's get one for our meta-analytic model.

```
forest(ma)
```

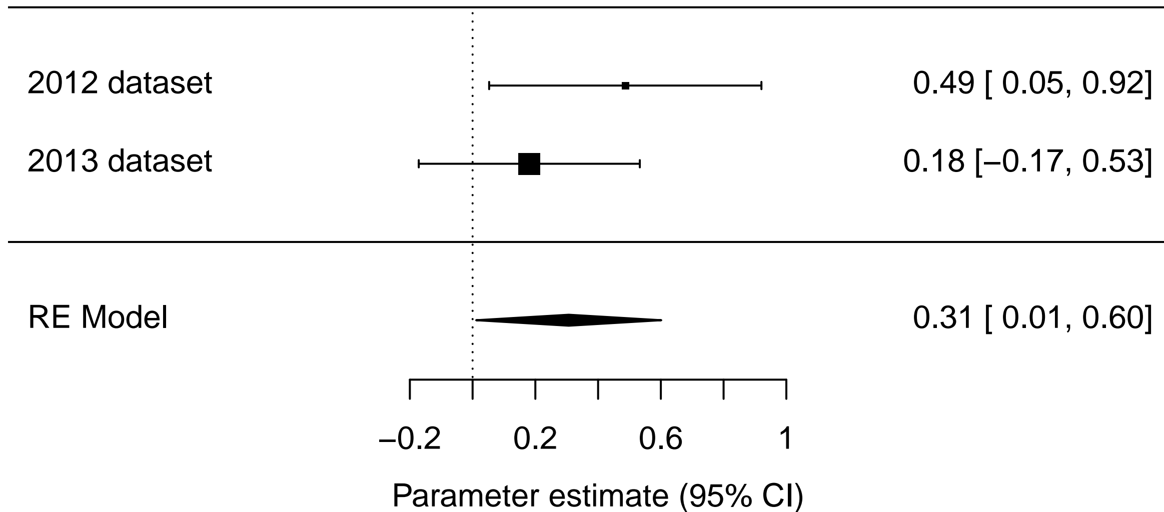


The first row shows the 2012 dataset. The central square shows the parameter estimate, and the horizontal line shows the 95% confidence interval. This does not overlap zero which corresponds to the fact that the association was significant at the $p < 0.05$ level in the 2012 data set. The second row shows the same information for the 2013 dataset; here, as you should expect, the 95% confidence interval overlaps zero.

The bottom row "RE Model" shows, as a diamond, the estimate and confidence interval for the two studies combined. The centre of the diamond is higher than 2013 but lower than 2012; somewhere in the middle, just as you would expect from an average. And the confidence interval for the meta-analytic model is narrower than those for either of the individual studies: of course, since the meta-analysis uses twice as much data as either individual estimate. And you can see that the diamond does not quite overlap zero. By the way, the two studies do not contribute equally to the averaged parameter estimate. Instead, they are weighted by their standard error. For this reason, 2013 contributes a bit more heavily than 2012. You can tell this by the size of the squares in the first and second row of the forest plots.

We can improve our forest plot a better labelling, as follows:

```
forest(ma, slab=c("2012 dataset", "2013 dataset"), xlab="Parameter estimate (95% CI)",
       main="Associations between telomere change and peak cort")
```



Difference of significance is not significance of difference

This analysis has revealed a very important general point. If we thought that whether the p-value was less than 0.05 or not was the best way of summarising the result of an experiment, then we would say something like the following: 'The results of the two studies were inconsistent. One study suggested that **peakcort** was associated with **telomere.change**, but the other study did not replicate this. Perhaps the first study was a fluke, or perhaps there were methodological differences between the two studies that explain their differing results'.

If instead we look at the forest plot as the best summary of the results of the two studies, we would come to a very different view. The first study says that every s.d. increase in **telomere.change** produces somewhere between a 0.05 and a 0.92 s.d. change in **peakcort**. And the second study says every s.d. increase in **telomere.change** produces somewhere between a -0.17 and a 0.53 s.d. change in **peakcort**. There's a huge overlap between these inferences. For example, if the 'true' parameter value were, say, 0.3 or 0.4, that would be completely compatible with *both* individual experiments. So the evidence from two experiments is not really mixed: there are plenty of scenarios simultaneously compatible with both findings, without having to invoke flukes or methodological differences. And our best possible belief about the world uses both datasets to inform it, as we have done in the meta-analytical model.

For this reason, statisticians have a mantra: 'Difference of significance is not significance of difference'. Just because two analyses produced p-values that fell on different sides of an arbitrary cut-off line like p = 0.05 does not mean that the inferences from two models were substantively discrepant from one another.

Using meta-analysis to review the literature

Meta-analysis is great to use when you yourself have done multiple studies on the same question. Its more familiar use though is in reviews of the literature. Let's say that there are ten published studies out the on the question that interests you. An old-fashioned way of reviewing this evidence would be count up which of these studies reported a 'significant' finding, and which did not, and come to some kind of conclusion. But for all the reasons discussed above, this would be a poor way of going about it. It would be much better to pool all their findings in a meta-analysis. Even if *none* of the individual studies produced a parameter estimate significantly different from zero, the meta-analytic parameter estimate might be significantly different from zero, because you estimate it with much greater power than any of its individual constituents.

To do a meta-analytic review, you need to pull out estimates of effect size from each of the studies, and pass them, and some indication of their precision, to the **rma()** function of 'metafor'. This is easy in principle, though when you do it, you quickly realise that most published papers do not report enough detail to extract what you need, and also do not make their raw data easily available. This is frustrating and very bad for the efficiency of science.

The effect sizes combined in a meta-analysis do not necessarily have to be standardized parameter estimates from general linear models. They could be correlation coefficients, or odds ratios, depending on the kind of studies you are reviewing - see the 'metafor' documentation for more detail. You may even be in the situation of combining effect sizes from studies reporting different kinds of analysis from one another. This can be done, because there are approximate formulae for converting, for example, odds ratios into correlation coefficients, or vice versa. The 'metafor' package has an excellent function, **escalc()**, for turning almost any kind of statistic (raw frequencies, odds ratios, correlations, t statistics. . . .) into almost any effect size measure; see its documentation for details.

If you want to take meta-analysis further, the 'bible' is: Borenstein, M., L.V. Hedges, J.P.T. Higgins, and H. R. Rothstein (2011 and other editions). *Introduction to Meta-Analysis*. Chichester: Wiley.

Session 7: R programming, writing simulations, and scripting tips

A great thing about R is that it is not just for data analysis. It is also good for writing simulations, or other kinds of computation and data processing. The only things it is not good at are driving external devices, and symbolic algebraic operations of the kind you might do in Mathematica or Maple.

In this session, we will learn how to write our own functions, and then use this to simulate the statistical power of a study we plan to do. Simulating the power of your study is a good habit to get into: indeed, I often simulate the whole dataset ahead of time to see what the data might look out if my various predictions are, or are not, met. We will conclude with some tips on optimal R scripting for easy collaboration and your own sanity.

There is an R script 'session7.r' to accompany this session. It provides the chunks of code you need for a lot of the following examples, to save you typing.

Writing your own functions

We will start by look at how to write our own functions. R has built-in functions like **mean()**, **plot()** and so on, but you can also write your own for particular tasks you want to achieve. When writing a function, you need to answer three questions:

- What do I want my function to be called?
- What inputs do I want my function to expect?
- What computations do I want my function to perform on those inputs?

We'll start with a simple case: I want it to be called **nettle()**. I want the user to give it two numbers. I want it to determine which of the two numbers is greater. So here's the code to achieve this. Type it out and send it to the console (alternatively, it is chunk 1 in the script 'session7.r').

```
nettle=function(a, b) {
  if (a>b){answer="greater"}
  if (b>a){answer = "less"}
  if (a==b){answer = "the same"}
  return(answer)
}
```

The first line says: **nettle()** is going to be a function that takes two arguments. These are called **a** and **b** in the function definition but this is arbitrary: it could be **x** and **y** or anything else. Also, it won't matter if you have another object called **a** or **b** in your script elsewhere: the names **a** and **b** here are used purely internally to the workings of the function. After the function definition there is a pair of {} braces. These in effect say 'when I call **nettle()**, look for two numbers **a** and **b** and do to them everything contained within the {}'.

The next three lines are *if* statements. Each of these has the structure 'if (this condition is true) {do the thing in the braces}'. So the first of these lines says: if **a** is the larger number, set the variable called **answer** to the value 'greater', and so on. The final line within the function definition tells the function to return the object called **answer** to the general environment. Functions must usually end by returning something to the general environment (otherwise, what is their point?).

Now, let's call our function.

```
nettle(a=10, b=9)
```

```
## [1] "greater"
```

We can also set an object to the value returned by **nettle()**.

```
g=nettle(a=4, b=9)
g
```

```
## [1] "less"
```

It's not strictly necessary to specify which number is **a** and which is **b**. R will interpret it from the order you give them in (unless you explicitly specify them in a different order).

```
nettle(4, 5)
```

```
## [1] "less"
```

However, you must always provide two numbers, or you get an error. Try:

```
nettle(7)
```

Adding a default value

Say we are often in the situation of wanting to know whether a number is bigger than 10 or not. We can use the **nettle()** function to do this, but we are going to have to type '10' for the second number every time we call it. It would be nice if we could say, in the function definition, 'assume the second number is 10 unless I tell you otherwise'. Here's how we do this (chunk 2 in the script).

```
nettle=function(a, b=10) {
    if (a>b){answer="greater"}
    if (b>a){answer = "less"}
    if (a==b){answer = "the same"}
    return(answer)
  }
```

The **b=10** in the specification of the function means that if **b** is not provided, it will be assumed to be 10. So now:

```
nettle(4)
```

```
## [1] "less"
```

But:

```
nettle(4, 2)
```

```
## [1] "greater"
```

Simulating the power of an experiment

The function **nettle()** was a trivial example. Now we are going to make some functions that do more useful things. Here's the scenario. I am going to perform an experiment with two conditions. I want to know how many subjects I need to run in each experimental group to achieve any given level of statistical power, under different assumptions about what size of effect on the mean of the outcome variable the experimental treatment might cause.

We need to build this up in stages. First, the size of the experimental effect can be expressed in terms of fractions of a standard deviation difference it makes (this is what the effect size measure Cohen's d represents). So my experimental treatment might make, say, 0.5 standard deviations difference to the mean response. This means that if we express the outcome variable in the control condition as standardized, that is, having a mean of 0 and a standard deviation of 1, then the outcome variable in the experimental condition will have a

mean of 0.5 and a standard deviation of 1. So, representing my effect size as f, we can say that:

$$y_{control} \sim N(0, 1)$$

and

$$y_{experimental} \sim N(0 + f, 1)$$

So, we are going to need to simulate two sets of Normally distributed random numbers, the first with mean 0 and s.d. 1, the second with mean **f** and s.d. 1. We do this with the R function **rnorm()**. This function takes three arguments, corresponding respectively to how many numbers you want, their mean, and their standard deviation.

```
rnorm(n=5, mean=0, sd=1)
```

```
## [1] -0.07431768 -0.26054849  0.14343862 -0.13250314 -0.67011004
```

The other useful function we are going to use is **rep()**, which just repeats its first argument as many times as you specify.

```
rep("control", times=5)
```

```
## [1] "control" "control" "control" "control" "control"
```

Putting these together, we can simulate a dataset with 100 subjects in each experimental group, then test for a significant effect of the treatment on the outcome, as follows (chunk 3 of script).

```
y=c(rnorm(100, mean=0, sd=1), rnorm(100, mean=0.5, sd=1))
condition=c(rep("control", times=100), rep("experimental", times=100))
m=lm(y~condition)
summary(m)
```

```
##
## Call:
## lm(formula = y ~ condition)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.2623 -0.6742  0.0038  0.6658  3.5063
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)           -0.04354    0.10553  -0.413   0.6804
## conditionexperimental  0.37221    0.14924   2.494   0.0135 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.055 on 198 degrees of freedom
## Multiple R-squared:  0.03046,    Adjusted R-squared:  0.02556
## F-statistic:  6.22 on 1 and 198 DF,  p-value: 0.01345
```

The first line makes the **y** variable up out of 100 control values with a mean of 0 and an s.d. of 1, plus 100 experimental values with a mean of 0.5 and an s.d. of 1. The second line makes up the predictor variable (**condition**) out of 'control' repeated one hundred times followed by 'experimental' repeated 100 times. The third line fits a linear model with **y** as the outcome and **condition** as the predictor. The final line asks for the summary of the model, which shows a highly significant effect of condition.

You can repeat this chunk of code as many times as you like, altering either the effect size (currently 0.5), or the number of subjects per group (currently 100), to assess the impact of these.

Rather than doing this manually though, it would be much better to have a function that we could call that would do it with one line of code. Here's a function I have written to do this job (chunk 4 in the script).

```
p.please=function(n, f){
  y=c(rnorm(n=n, mean = 0, sd = 1), rnorm(n=n, mean = f, sd = 1))
  condition=c(rep("control", times=n), rep("experimental", times=n))
  m=lm(y~condition)
  return(summary(m)$coefficients[2, 4])
}
```

Look over the function definition and see if everything is clear. The user can specify the number of subjects per group **n**, and the effect size **f**. You might be puzzled by the **n=n** and **f=f** in this line. What this is saying is: "take the values of **n** and **f** given in the call to **p.please()** and pass them in turn into the function **rnorm()**". The last line may also be puzzling to you. What you have to appreciate is that the **summary()** of a general linear model is an object with a sub-object called **$coefficients**, and that sub-object has four columns and as many rows as there are predictors (including the intercept). Here's how it looks:

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

```
##                          Estimate Std. Error    t value   Pr(>|t|)
## (Intercept)           -0.04353797  0.1055312 -0.4125601 0.68037532
## conditionexperimental  0.37220762  0.1492437  2.4939587 0.01345157
```

So the fourth column of this object contains the p-values, and the second row contains the values for condition. If we want the p-value for the effect of condition, it's therefore the second row and fourth column of the coefficients of the summary that we want.

```
summary(m)$coefficients[2, 4]
```

```
## [1] 0.01345157
```

So that's why the last line in the definition of the **p.please()** function is the way it is: it causes the function to return the p-value for condition from the model testing for an effect of condition. Let's try it:

```
p.please(n=30, f=0.5)
```

```
## [1] 9.347419e-06
```

So when you have an effect size of 0.5 standard deviations and you run 30 subjects per group, you get (in this instance) the p-value shown.

Multiple runs of the simulation

Just doing the analysis once and getting the p-value is not very illuminating, because the answer differs a quite lot from run to run. Instead, we want to know what the typical *range* of outcomes is for different levels of **n** and **f**. So a good thing to do would be to run **p.please()** one hundred times and look at the distribution of p-values produced.

We can do this by writing another function that calls **p.please()** repeatedly (say, 100 times) and logs the output into a vector. To do this, I first need to introduce the idea of a *for loop*. In a for loop, the same operation is repeated whilst an index moves through a set of possible values. For example, look at:

```
for (i in 1:10){print(i)}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

```
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Here, the index **i** moves through the set of possible values 1, 2, 3,..10, and at each possible value of **i**, the operation in the braces is performed (here, **print(i)**).

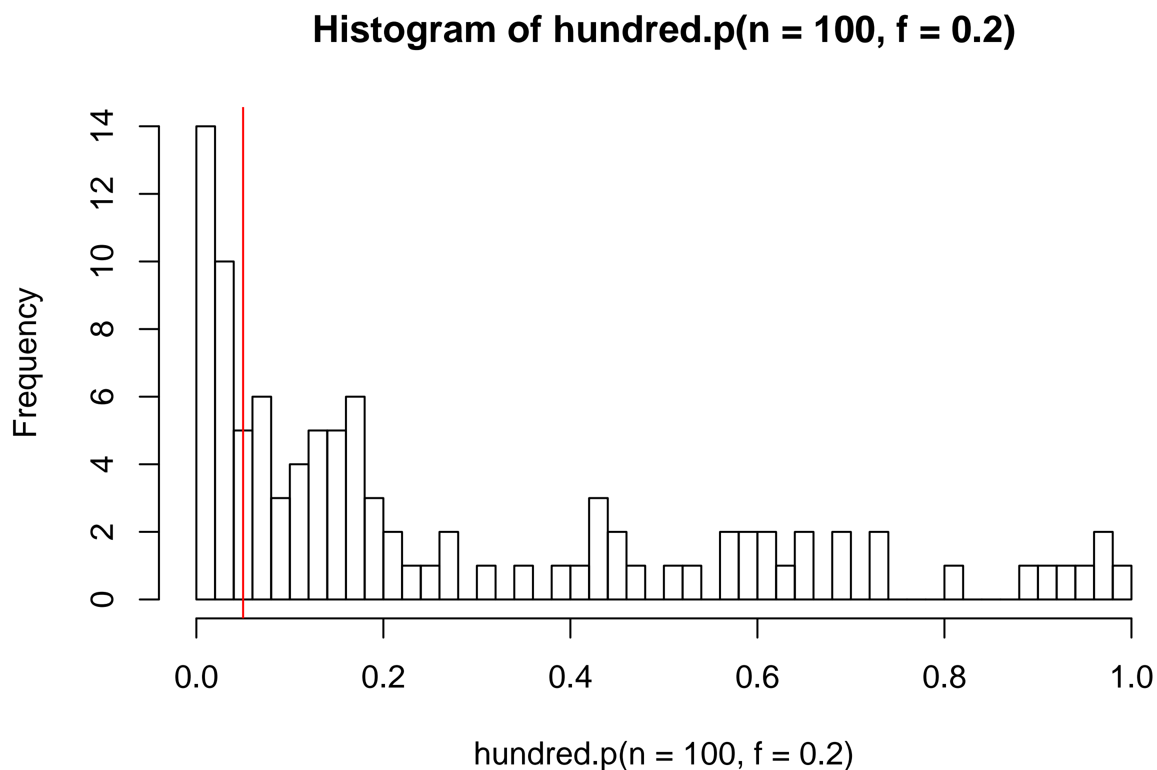
With this in mind, let me introduce a function to return the p-values from 100 repetitions of **p.please()** at given values of **n** and **f**. This is chunk 5 in the script.

```
hundred.p=function(n, f){
  output=NULL
  for (i in 1:100) {output[i]=p.please(n=n, f=f)}
  return(output)
}
```

The first line inside the braces sets up an empty vector called **output**. Then we move through all the values of index **i** from 1 to 100, and at the ith position of **output**, we place another run of the p-value from function **p-please()**.

Let's see what we can do with **hundred.p()**. The following will give us a histogram of the p-values when we do the experiment 100 times with **n=100** and **f=0.2**.

```
hist(hundred.p(n=100, f=0.2), breaks=50)
abline(v=0.05, col="red")
```

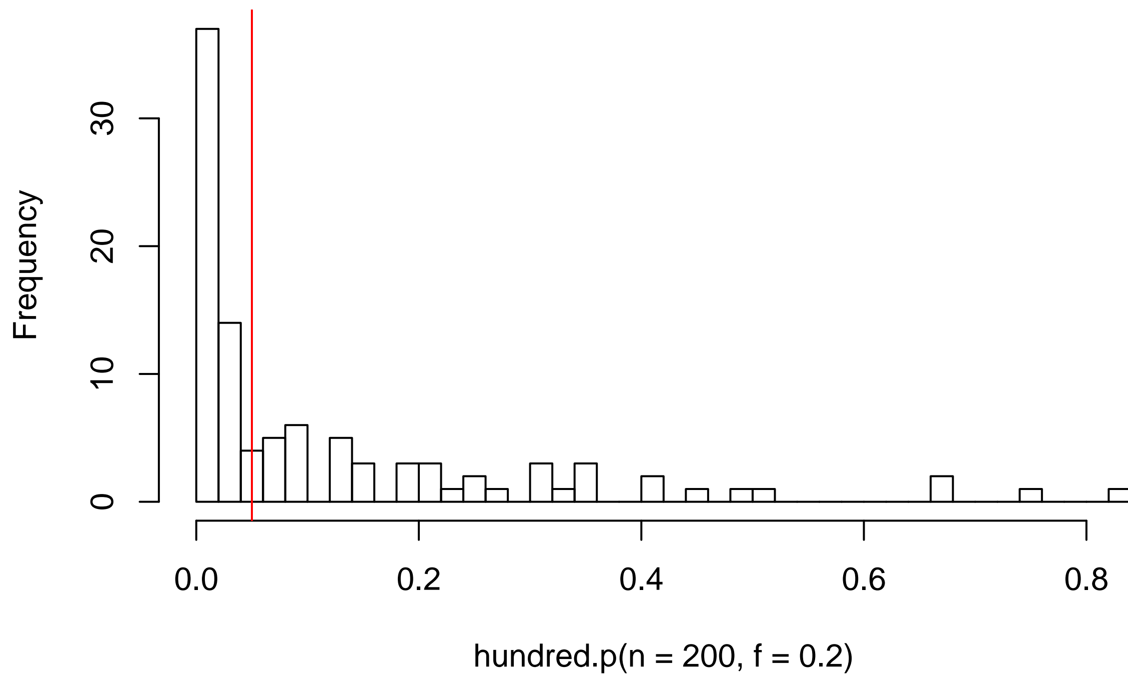


The line **abline(v=0.05, col="red")** is to give us a vertical red line at p=0.05, the conventional cutoff for statistical significance. What the histogram shows us is that if we run 100 subjects in each group, with a true

experimental effect of 0.2 standard deviations, then we may quite often see a p-value less than 0.05. On the other hand, just by the luck of the draw, we might see p-values considerably greater than 0.05, even up to almost 1.0. So even if a true effect exists with a magnitude of 0.2 standard deviations, we are often not going to see it with a 'significant' p-value in samples of this size.

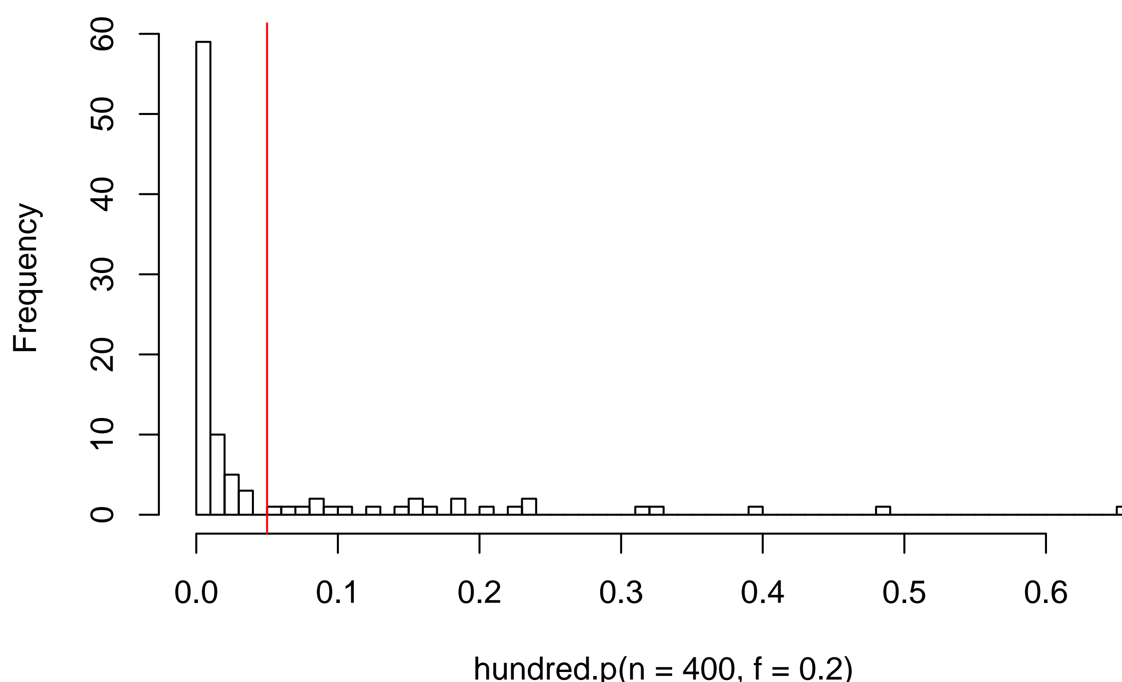
So how much would it help us to double the sample size to 200 subjects per group?

```
hist(hundred.p(n=200, f=0.2), breaks=50)
abline(v=0.05, col="red")
```



That looks quite a lot better. And what about doubling again, to 400 subjects per group?

```
hist(hundred.p(n=400, f=0.2), breaks=50)
abline(v=0.05, col="red")
```



Better still, though very occasionally you can be unlucky even at this sample size.

Calculating the power

What we would really like is a numeric answer to the question: given that there is a real experimental effect of magnitude **f**, what proportion of the times I run an experiment with **n** subjects per group would I expect to find a p-value that meets the conventional criterion of statistical significance? This number is known as the *power* of the experiment.

We can estimate the power using the functions we have built so far.

```
l = hundred.p(200, 0.2)
power=mean(l<0.05)
power
```

```
## [1] 0.5
```

The second line needs some explaining. Given that the vector **l** consists of 100 p-values, then **(l<0.05)** is a vector of 100 TRUEs and FALSEs (TRUE where the p-value is less than 0.05, FALSE otherwise). When forced to treat TRUE and FALSE numerically, R considers TRUE to be equal to 1, an FALSE to be equal to 0. So the mean of **(l<0.05)** is identical to the proportion of TRUEs it contains. This is why **power=mean(l<0.05)** does what it does in the code above. And the answer tell us that the power (the chance of correctly getting a significant result given that the experimental effect really exists) is about 0.5 under these conditions.

If we double the number of subjects per group, the power improves a lot:

```
l = hundred.p(400, 0.2)
power=mean(l<0.05)
power
```

```
## [1] 0.71
```

And improves still more when we double again:

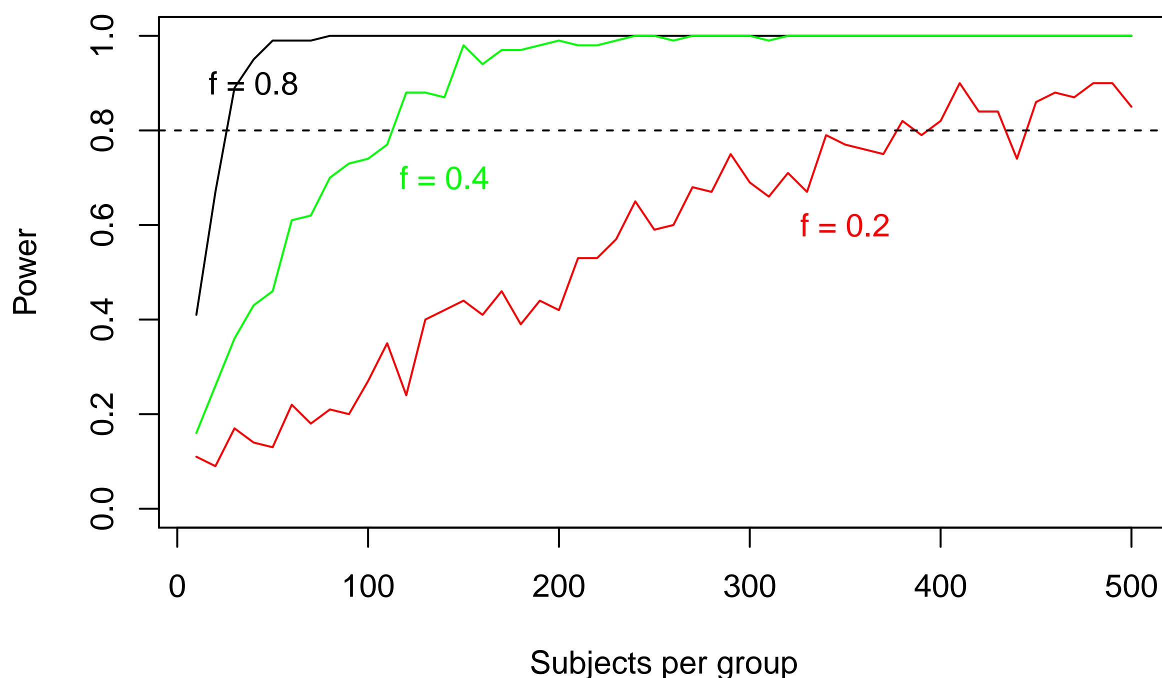
```
l = hundred.p(800, 0.2)
power=mean(l<0.05)
power
```

```
## [1] 0.95
```

Plotting power

As a final example of what you can do with these functions, here's some code that plots how power increases with increase **n** for different true effect sizes (0.2, 0.4, 0.8). I will just reproduce the code (chunk 6 in the script), rather than going through line by line. You can probably work out what it all does. I have included comments for clarity. It takes a while to run!

```
# Chunk 6
# First define the possible numbers of subjects to consider (10-500)
subjects=seq(10, 500, by=10)
# Set up some empty vectors
power.1=NULL
power.2=NULL
power.3=NULL
# Now set a counter at zero to keep track of where we are
counter=0
# Set up a for loop to cycle through all the sample sizes to be considered
for (i in subjects){
  counter=counter+1 #Augment the counter
  # Now calculate the power at that sample size for three different values of f
  power.1[counter] = mean(hundred.p(n=i, f=0.2)<0.05)
  power.2[counter] = mean(hundred.p(n=i, f=0.4)<0.05)
  power.3[counter] = mean(hundred.p(n=i, f=0.8)<0.05)
  }

# Now we have generated the data we need, plot power ~ subjects for f=0.5
plot(power.3~subjects,
     ylim=c(0, 1), xlab="Subjects per group", ylab="Power", type="l")
# Add lines for the other effect sizes
lines(power.2~subjects, col="green")
lines(power.1~subjects, col="red")
# Add dashed horizontal line at power of 0.8
abline(h=0.8, lty=2)
# And some labels
text(40, 0.9, "f = 0.8")
text(140, 0.7, "f = 0.4", col="green")
text(350, 0.6, "f = 0.2", col="red")
```



We can see from this plot that if our experimental effect is to increase the response by 0.8 standard deviations, 50 subjects per group is masses to detect it reliably. If our experimental effect is 0.4 standard deviations, we need to be looking at more than 100 per group, and if our experimental effect is going to be 0.2 standard deviations, well, maybe we should study a different experimental effect!

Concluding with our simulation

I hope you can see the potential value of simulating in R. With just an hour of work, you can understand aspects of your empirical problem, like what your p-value means or what your experiment is capable of telling you, much more deeply.

This particular simulation was pointless in some ways, because there are online statistical power calculators widely available, and also because there are exact analytical expressions available for the power of an experiment with such a simple design as this one. Nonetheless, if you learn how to simulate, then you can generate results for your own bespoke situations, which may be more complex. This can give you a really good feel for what kinds of sample sizes and designs could work for your problem.

You don't have to use simulations just for questions of statistical power. R can be used for simulations of all kinds, and I use it frequently in theoretical work. Even when you have analytical mathematical results, it's good to simulate them too, to make sure your maths really does pan out. It is easy and quick to do this using R.

Good practices for scripting

We conclude the session with some tips for optimal script writing in R. In principle, the great thing about R is that you can send your script and data over to a collaborator, and they can run your script, add things,

modify things, and ping it back to you. In practice, this does not always work as well as it should. The main reason is that other people's scripts can be hard to understand and may not run in someone else's hands. Even your own scripts from last week can be quite non-transparent to you this week!

Scripts rapidly become very long, and the different parts of the script are often dependent on other parts having been previously run. Also, it may not be obvious what a particular variable name is or what a particular line is doing if you do not already know. To avoid some of these problems, here are a few tips for writing R scripts that are easy for yourself and others to work with.

Good scripts are modular

In a typical analysis, there are several sections (e.g. loading the data, finding the descriptive stats, fitting a model for outcome 1, plotting figure 1, fitting a model for outcome 2, plotting figure 2, and so on.). On a particular day, you (or someone) might just want to work on plotting figure 2. Therefore, they don't want to have to run all the code of the earlier sections to make this section work. In other words, you want your script to be *modular*: that is, it should be possible to run one section without depending on all the other sections.

I always divide my scripts into sections. You make a section header in RStudio by three ### at the beginning and more than three #### at the end of the comment line you want to be the header of the section. The user can then see a little menu of the headers at the bottom of their script window, and jump straight to any particular one.

The way I achieve modularity is to start with a section called the **###Head#####**. In this section, I read in all the datasets that will be required, attach all the contributed packages I am going to use, and make any extra data operations that are going to be widely used (for example, if you are making a new variable that is a composite of two others or something, then do it in the **Head** if it will be used in several places in your analysis). The point of the **Head** is that once you have run it, you can jump to any other section you like and the code there will work, without necessarily doing any of the other intermediate sections.

For example, it would not be modular if at the beginning of the script you loaded some data in, then at line 157 where you are working on figure 1, you calculate the log of one of your variables **d$log.RT**, then at line 323 when you are working on figure 2, you refer to **d$log.RT** again. That means that if I work directly on figure 2, I will get an 'Error: No such variable' message when I run the code that does something with **d$log.RT**, unless I have previously also made figure 1. Instead, calculate **d$log.RT**, if you need to calculate it at all, in the **Head** section.

Economise on intermediate objects

When you first start coding in R, you find yourself creating masses of intermediate objects. For example, you have a data frame **d**. But at one point in your script, you only want to look at the females. So you make a second object **df=subset(d, gender==“female”)**. Then one you have done that you need to calculate a variable which is the log of the reaction time so you run **df$log.RT=log(df$RT)**. Only then do you fit a model, as in:

```
m=lm(log.RT~age, data=df)
summary(m)
```

Here you have created several intermediate objects, **m**, **log.RT** and **df** that may not in fact be necessary. With practice you can just embed one function inside another, as in:

```
summary(lm(log(RT)~age, data=subset(df, gender=="female"))
```

This code achieves the same as calculating **log.RT**, **df** and **m**, but does so in one line with a minimum of intermediate objects. It also has the advantage of not relying on the user having previously run the intermediate lines creating **log.RT**, **df** and **m**.

Make your variable names a good compromise between intelligibility and economy

You have to type your variable names a lot whilst working with R. It is probably therefore a bad idea to call your data frame something like **starling.2017.field.weight.data**, or your variables something like **corrected.qpcr.telomere.change.early.life**. On the other hand, some reasonable transparency will help the reader, so I tend to use short descriptive labels such as **weight**, **height** and so on. And it's good practice to be consistent, at least within one dataset, about whether your variable names have capital letters or not. It drives me up the wall when in a particular dataset the variable **Weight** has a capital and **height** for some reason does not. It's much easier in terms of not getting error messages if either everything has a capital, or nothing does.

Comment your scripts

A final point is that a good script is well commented. As well as section headings, put intelligible comment lines prior to each few lines of code saying what those lines do. You can also add comments to particular lines of code if it is not obvious what they are doing, or there is something the user needs to remember.

Taking R further

The main things to take away about using R are:

- It's fun
- You get fewer errors as you get more used to it
- If it exists, it probably can be and has been done in R.
- Someone else has had the problem you are having, and probably posted the solution somewhere online like StackOverFlow.
- You should feel empowered to just try stuff.
- R users are a friendly and sharing bunch who will answer your questions and have probably put some help on the web somewhere
- It's good to be open with your code and data: we all benefit and learn together.

Appendix

This appendix consists of some tricks that did not come up naturally elsewhere, but you might need. There's no need to work through from beginning to end; this session is just here for reference.

Converting classes and making dataframes

Sometimes you are going to need to change the class of an R object in order to 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 continuous 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** currently of class **factor**, and wish to treat it as a continuous numeric variable, then **as.numeric(v)** will not do what you want. The conversion you want is achieved by:

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

Just one of those strange things you pick up - it's do with how factors are represented underlyingly as integers.

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, e.g. **x1**, **x2** and **x3**, it's:

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

To name the columns:

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

Some tips for working with factors

Factors are a class of vectors with a fixed number of discrete levels. These levels are represented underlyingly as integers. If you have a text variable in a .csv file, it will probably be automatically classed as a factor on import. There are a few problems you encounter when working with factors.

The first is when you want to alter a case within a factor, and find you can't. For example, let's quickly load in the starling data from session 3.

```
d = read.csv("star.data.csv")
head(d)
```

```
##    X Brood.type Number.chicks Sex    Farm Telo.change Weight
## 1  1      Small             5   m Whittle   0.2449029   52.5
## 2  2      Small             5   f Whittle   0.3810148   51.0
## 3  4      Small             5   f Whittle   0.2560100   47.5
## 4  5      Small             5   f Whittle   0.1794750   41.7
## 5  8      Large             6   m Whittle   0.3661215   48.3
## 6 11      Large             6   f Whittle  -0.6420882   35.1
```

Now let's say we had found, looking back at our field records, that the first case was actually from a brood of 2 chicks, and we want to consider this 'Very Small' to distinguish it from the regular 'Small' broods. So, let's change the first value of **d$Brood.type** to 'Very Small'.

```r
d$Brood.type[1] = "Very Small"
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = structure(c(NA, 2L, 2L, 2L, :
## invalid factor level, NA generated
```

Oops. What went wrong? Well, a factor has a *predefined* set of discrete levels. If you add in a datapoint that does not correspond to one of these levels, you will just get an **NA**. So to achieve the operation we want, we must first convert the variable to class 'character' (no predefined levels), as follows:

```r
d$Brood.type = as.character(d$Brood.type)
```

Now we can go ahead and make our first case 'Very Small', and then make the variable back into a factor again.

```r
d$Brood.type[1]="Very Small"
d$Brood.type=factor(d$Brood.type)
```

Let's look at the first few cases of the factor to convince ourselves it has worked:

```r
d$Brood.type[1:10]
```

```
##  [1] Very Small Small      Small      Small      Large      Large
##  [7] Small      Small      Small      Small
## Levels: Large Small Very Small
```

The other thing you may wish to do is to change the order of the factor levels. A reason for doing this might be, for example, to change the order they appear in a figure. At the moment, as you can see from the output above, the order of levels is 'Large', 'Small', 'Very Small'.

To change this:

```r
d$Brood.type = factor(d$Brood.type, levels=c("Very Small", "Small", "Large"))
```

The levels will now display in the order 'Very Small', 'Small', 'Large'.

Default working directories and library paths

Contributed packages you download will be sent to a library directory that got set up in the installation of R (not the same thing as the current working directory). These packages contain many files and can be very large, and so you may want to control where they go (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:

```r
.libPaths()
```

```
## [1] "C:/Users/Daniel/Dropbox/R"
## [2] "C:/Program Files/R/R-3.5.0/library"
```

To set this to something else, specify where, as in the following:

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

I run a command like this at start up every time. I use a Dropbox folder as my library directory. All of my three computers point at the same Dropbox folder, which synchronises via the internet. 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. I'm that sad.

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")
```

If you want, you can preset R so that it automatically looks somewhere non-standard for packages, and also uses a working directory of your choice. You do this with by writing an **.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 an uplifting message. Here it is:

```
First <- function(){
  .libPaths("C:/Users/Daniel/Dropbox/R") #Set library directory
  setwd("C:/Users/Daniel/Dropbox/R") #Set working directory
  library(ggplot2) # Much-used packages
  library(dplyr)
  library(psych)
  library(tidyr)
  library(metafor)
  cat("\014") # Clear screen
  cat("\nWelcome Daniel on ", date(), "\n") #Welcome message
  cat("\nWhen we cut mere stones, we should be envisioning cathedrals.\n")
}
```

Subsetting data frames

You will often have a dataframe containing your data, but want to perform an analysis on only part of it (certain columns, and/or certain rows). The **subset()** command allows you to do this. For example, say I have a dataframe **d**, then the subset where **gender** is **female** is just:

```
subset(d, gender=="female")
```

You can set this subset as a new separate object, or pass it directly to a **lm()** or other function.

You can also address parts of the dataframe using the [ ] notation to identify particular rows and columns. For example:

```
d[1:10, 1:5]
```

This represents the first ten rows of the first five columns of dataframe **d**. A blank here is interpreted as 'everything', so that the following means 'the first ten rows of all columns' of **d**.

```
d[1:10, ]
```

Handling missing data

In R, missing data are represented by the code **NA**. If there are blanks in a .csv file you read in, R will convert them to **NA**. When you have **NA**s in a vector, some difficulties can arise. Let's define a vector with some missing data.

```
x=c(2, 4, 7, 1, NA, NA, 6, 3)
```

If we now ask for the mean of **x**, we get an unhelpful response:

```
mean(x)
```

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

The mean is **NA** because at least one of the values is **NA**. If we want to remove the **NA**s prior to calculation, we need to specify this.

```
mean(x, na.rm=TRUE)
```

```
## [1] 3.833333
```

With the **lm()** function, you may need to specify to omit the **NA**s:

```
m=lm(y ~ x, data=d, na.action=na.omit)
```

Note that if you are using package 'MuMIn' to do model selection, you will need to preset the action for **NA**s as follows.

```
options(na.action="na.fail")
```

This is to prevent models with different numbers of data points getting compared by AIC (see session 6). If you have some **NA**s in your data, you will first need to subset to just the complete cases and then run the model selection on this subset (see below).

Finally, note that the answer to whether an **NA** is equal to some value or not is not **FALSE**, but **NA**.

```
x==3
```

```
## [1] FALSE FALSE FALSE FALSE    NA    NA FALSE  TRUE
```

This can cause you problems when you have some missing data and you try to select cases on the basis of whether they have a certain value or not:- as well as **TRUE**s and **FALSE**s in the answer, there will be some **NA**s.

To find out where you have **NA**s, you need the **is.na()** function.

```
is.na(x)
```

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

Its negation (the non-missing cases) is **!is.na()** (! means logical NOT in R).

```
!is.na(x)
```

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

Thus, the non-missing cases of **x** can be found with:

```
x[!is.na(x)]
```

```
## [1] 2 4 7 1 6 3
```

Note though that this has a different length from the original vector **x**, and hence the values are not in their original positions.

You can select complete cases for performing an analysis using multiple **!is.na()** conditions. This will look something like the following:

```
complete.data=subset(d, !is.na(y.variable) & !is.na(x1.variable) & !is.na(x2.variable))
```

Plotting a bar graph with error bars

A common type of graph used in the behavioural sciences is a bar plot showing the group means of the outcome variable, plus some error bars. Surprisingly, this is not completely straightforward 'off the peg' in R, perhaps because statisticians tend to see box plots as more informative.

Here's a quick guide on how to make a bar graph with error bars. I am going to manage to use three different contributed packages in doing so! The packages are 'psych', 'dplyr' and 'ggplot'. Install them now if needed. We have not used 'dplyr' elsewhere, but it is a handy tool for summarising and transforming dataframes.

First, let's load in the data from sessions 1 and 2.

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

Now the packages.

```
library(psych)
library(dplyr)
library(ggplot2)
```

Now, to make our graph, first we need a dataframe that summarises out from **d** the means and standard errors we require (**summarize()** and **group_by()** are functions in 'dplyr', and **describe()** is from 'psych'):

```
fig.data=summarize(group_by(d, Sex), Mean.Height=mean(Height),
se.Height=describe(Height)$se)
```

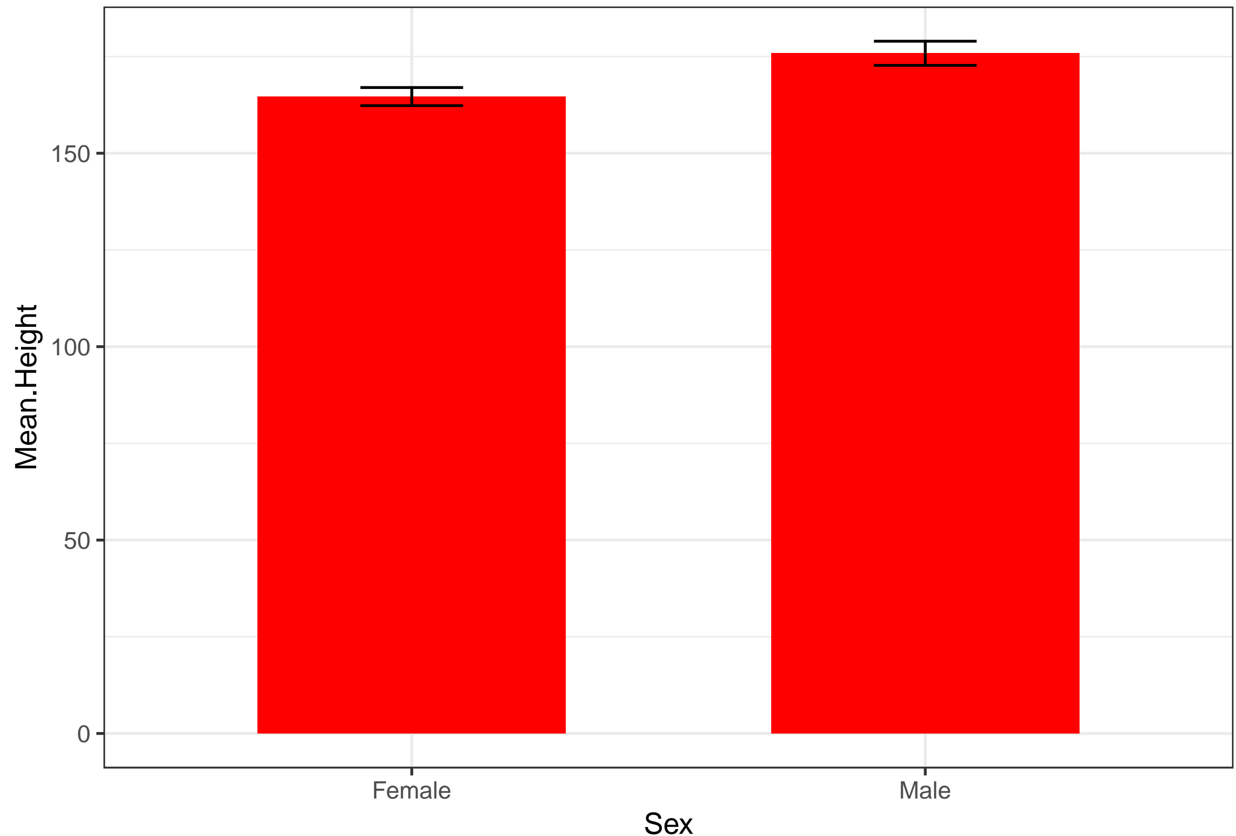
Now look at the resulting dataframe.

```
fig.data
```

```
## # A tibble: 2 x 3
##   Sex    Mean.Height se.Height
##   <fct>        <dbl>     <dbl>
## 1 Female        165.      1.19
## 2 Male          176.      1.59
```

Now we need to make a ggplot from the dataframe **fig.data** rather than **d**, as follows:

```
fig1 = ggplot(fig.data, aes(x=Sex, y=Mean.Height)) +
  geom_bar(stat="identity", fill="red", width=0.6) +
  geom_errorbar(aes(ymax=Mean.Height+1.96*se.Height,
                    ymin=Mean.Height-1.96*se.Height), width=0.2) +
  theme_bw()
fig1
```



The 1.96 standard errors is to give a 95% confidence interval. You can adjust the appearance of the figure to taste, as per session 3.

Reshaping data between wide and long formats

Sometimes you will need to reshape data from long to wide format or vice versa. Let's load in the sleep data from session 4.

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

The current format of **d** is long and thin: the two measures from each participant are on separate rows, so the number of rows is the number of observations rather than the number of people. To reshape **d** into wide format, we use the **reshape()** function:

```
d.wide = reshape(d, idvar = "PersonID", timevar = "Time", direction = "wide")
```

Have a look and confirm that it has worked:

```
head(d.wide)
```

```
##   PersonID Accuracy.1 Accuracy.2
## 1        1   41.26861   39.57513
## 2        2   30.23323   27.79155
## 3        3   34.81972   32.58701
## 4        4   34.18728   32.99018
## 5        5   36.49972   37.50081
## 6        6   30.45886   28.31217
```

And now if we want to go from wide to long:

```r
d.long.again=reshape(d.wide, idvar = "PersonID", v.names="Accuracy",
                     direction = "long", varying=c("Accuracy.1", "Accuracy.2"),
                     timevar = "Time")
```

Have a look:

```r
head(d.long.again)
```

```
##     PersonID Time Accuracy
## 1.1        1    1 41.26861
## 2.1        2    1 30.23323
## 3.1        3    1 34.81972
## 4.1        4    1 34.18728
## 5.1        5    1 36.49972
## 6.1        6    1 30.45886
```

It's quite easy to go back and forward. Wide format is useful for certain purposes, such as making correlation matrices and calculating difference scores, and long format is useful for almost eveything else. For complex forms of reshaping, the package 'tidyr' is very useful, with its functions **spread()** (spread one column across multiple columns) and **gather()** (gather multiple columns into one). You may find you need to do multiple rounds of selecting subsets of variables, spreading or gathering them, then merging them back into the original dataset (see below on how to merge).

Merging two or more dataframes

You can merge two dataframes using the **merge()** function as long as they both contain one common identifier variable that labels the cases. The syntax will be something like:

```r
merged.data=merge(d1, d2, by="Name.of.ID.varable")
```

Note that you will get all the variables from both dataframes in the merged dataframe. This means that if there are any variables other than the identifier that are common to both, these will be duplicated and given the names **variable.x** and **variable.y**, according to whether they came from the first or second dataframe.

If the identifier variable has a different name in the two dataframes, that is not a problem, as long as the *values* are the same in the two dataframes (otherwise there is no way of matching corresponding cases).

```r
merged.data=merge(d1, d2, by.x="ID", by.y="id")
```

By default, the only cases included in the merged dataframe are those which appeared in *both* source dataframes. This might not be what you want. For example, the second dataframe might contain some extra variables on some of your cases but not all, and yet you want all cases to appear in the merged dataframe. If you want to specify that all rows from the first dataframe should be kept in the merged dataframe, regardless of whether they appear in the second dataframe or not, the code would be:

```r
merged.data=merge(d1, d2, by.x="ID", by.y="id", all.x=TRUE)
```

The **merged.data** in this instance would have all the rows of **d1**, and the cases that had no counterpart in **d2** would have **NA**s for all the variables that originated in **d2**.

Note that by 'merging' here, we mean adding additional variables on the same cases to a dataframe. Something else you might want to do is to add additional cases which have the same variables to a dataframe. You do this with the **rbind()** function.

```r
combined.data=rbind(monday.data, tuesday.data)
```

Here, **monday.data** and **tuesday.data** would have to have the same number of columns and exactly the same column names.