

Just Enough R

Contents

Introduction	4
I Getting started	5
1 Working with R	5
Installation	5
Workflow	6
First commands	8
Naming things	9
Vectors and lists	10
Working with vectors	12
Functions to learn now	16
Packages	17
II Data	19
2 The <code>dataframe</code>	19
Working with dataframes	22
Selecting columns	25
Selecting rows	26
‘Operators’	26
Sorting	29
Pipes	30
Modifying and creating new columns	32
3 ‘Real’ data	33
Importing data	33
Saving and exporting	35
Dealing with multiple files	36
Joining different datasets	39
Types of variable	41
Missing values	44
Tidying data	53
Reshaping	53
4 Summaries	57
A generalised approach	57
5 Graphics	61
Benefits of visualising data	62
Which tool to use?	62
Layered graphics with <code>ggplot</code>	62
‘Quick and dirty’ (utility) plots	89
Tricks with <code>ggplot</code>	100
Exporting for print	108

III Models	108
6 Commonly used statistics	108
6.1 Non-parametric statistics	109
Crosstabulations and χ^2	109
Correlations	111
t-tests	118
7 Regression	123
Factors and variable codings	127
Model specification	128
8 Anova	141
Anova ‘Cookbook’	144
Checking assumptions	154
Followup tests	156
9 Generalized linear models	157
10 Multilevel models	162
Fitting multilevel models in R	163
Extending traditional RM Anova	164
Variance partition coefficients and intraclass correlations	171
3 level models with ‘partially crossed’ random effects	172
Contrasts and followup tests using <code>lmer</code>	175
Troubleshooting	177
Bayesian multilevel models	178
11 Mediation and covariance modelling	178
Mediation	178
Testing the indirect effect	182
11.1 Mediation using Path models	184
Covariance modelling	187
Path models	187
Confirmatory factor analysis (CFA)	191
CFA model fit	198
Modification indices	198
Model modification and improvement	199
Structural equation modelling (SEM)	200
‘Identification’ in CFA and SEM	207
Missing data	208
Goodness of fit statistics in CFA	208
12 Bayesian model fitting	210
13 Power analysis	216
IV Patterns	219
14 Learning key patterns	219
15 Unpicking interactions	220
16 Making predictions	230
Predicted means and margins using <code>lm()</code>	236

Predictions with continuous covariates	241
Visualising interactions	241
17 Models are data	241
‘Processing’ results	247
Printing tables	247
APA formatting for free	248
Simplifying and re-using	250
“Table 1”	252
18 Dealing with quirks of R	257
Rownames are evil	257
Working with character strings	258
Colours	260
19 Getting help	279
V Explanations	279
20 Confidence and Intervals	280
21 Multiple comparisons	282
22 Non-independence	296
23 Fixed and random effects	297
24 Scaling predictor variables	297
25 Non-scale outcomes	300
25.1 Link functions	300
26 Building and choosing models	306
References	309

Introduction



R makes it easy to work with and learn from data.

It also happens to be a programming language, but if you're reading this, that might not be of interest. That's OK — the goal here is not to teach programming¹. The goal is to teach you *just enough R* to be confident to explore your data.

This book uses R like any other statistics software: To work-with and visualise data, run statistical analyses, and share our results with others. To do that you don't need more than the *absolute basics* of the R language itself. The first chapters walk you through what you need to know to be productive.

0.0.0.1 Things to know Before you start

This guide is fairly opinionated, but for good reason.

¹This is a lie, but hopefully it won't be obvious until it's too late.

There are lots of ways to use R, and this has been a barrier for beginners. In particular base-R functions can be oddly-named, or lack a regular or predictable interface. For this reason we:

- Recommend (strongly) that you install and use ‘packages’ that extend some of R’s basic functionality. These packages are powerful tools in their own right, but also hide some of the complexities of R in a clear and consistent way. It might seem restrictive but, to begin with, learning *only* these packages will help you form a more consistent mental model and make rapid progress. You can learn the crusty old bits (which still have their uses) later on.
- Assume you are using the RStudio editor and working in an RMarkdown document (see the next section). If you don’t have access to RStudio yet, see the installation guide.

0.0.0.2 License

These documents are licensed under the CC BY-SA licence.

Part I

Getting started

1 Working with R

There are many ways of working with R. This guide focusses on a fairly specific setup and workflow, and assumes you will use the RStudio editor, use R markdown documents to document and share your analyses, and install a number of recent packages, including the ‘tidyverse’, which give working with R a shallower learning curve, and let you get powerful things done quickly.

Installation

1.0.0.1 Installing on your own machine.

1. Download RStudio 1.01 or later Use whatever version is most recent and expect to upgrade every 6 months or so, as new versions become available.
2. Install the packages listed below
3. Optionally, if you want to ‘knit’ your work into a pdf format, you should also install LaTeX. For most people this isn’t necessary, and is something you can skip for the moment, but it can be helpful when sharing finished analyses with colleagues. On windows use this installer. Make sure to do a ‘full install’, not just a basic install. On a Mac install homebrew and type `brew cask install mactex`.

1.0.0.2 Package dependencies

If you are just getting started on a windows machine, these instructions for students at Plymouth University make it easy to install R and most of the packages necessary to complete the examples in this book.

Further details of a recommended installation are given here. These scripts will install all needed packages on a recent Linux or Mac system.

For some of the sections on Bayesian estimation you will also need to install `rstan` and `rstanarm`. Details are also here, but this can wait till later.

Workflow

One big adjustment to make when moving away from tools like SPSS is to find a ‘way of working’ that suits you. We have often developed ways of working, saving, and communicating our work, and become comfortable with them. In part, these habits and routines may be attempts to work around limitations of these tools. But nevertheless, habits are easier to replace than break, so here’s an alternative model to adopt:

1. Work in RStudio, and use RMarkdown documents (see next sections).
2. Save your raw data in .csv format. Never edit data by hand unless absolutely necessary.
3. Use R to process your data and RMarkdown to document the process.

RMarkdown

Conventional statistics software like SPSS lacks a simple way to document and share your analyses, and make repeating or editing your work later very hard.

RMarkdown is a format for documenting and sharing statistical analyses.

This it might seem an odd place to start: we haven’t got anything to share yet! But using RMarkdown in RStudio provides a really nice way to work with data interactively and share our results, so we start as we mean to go on.

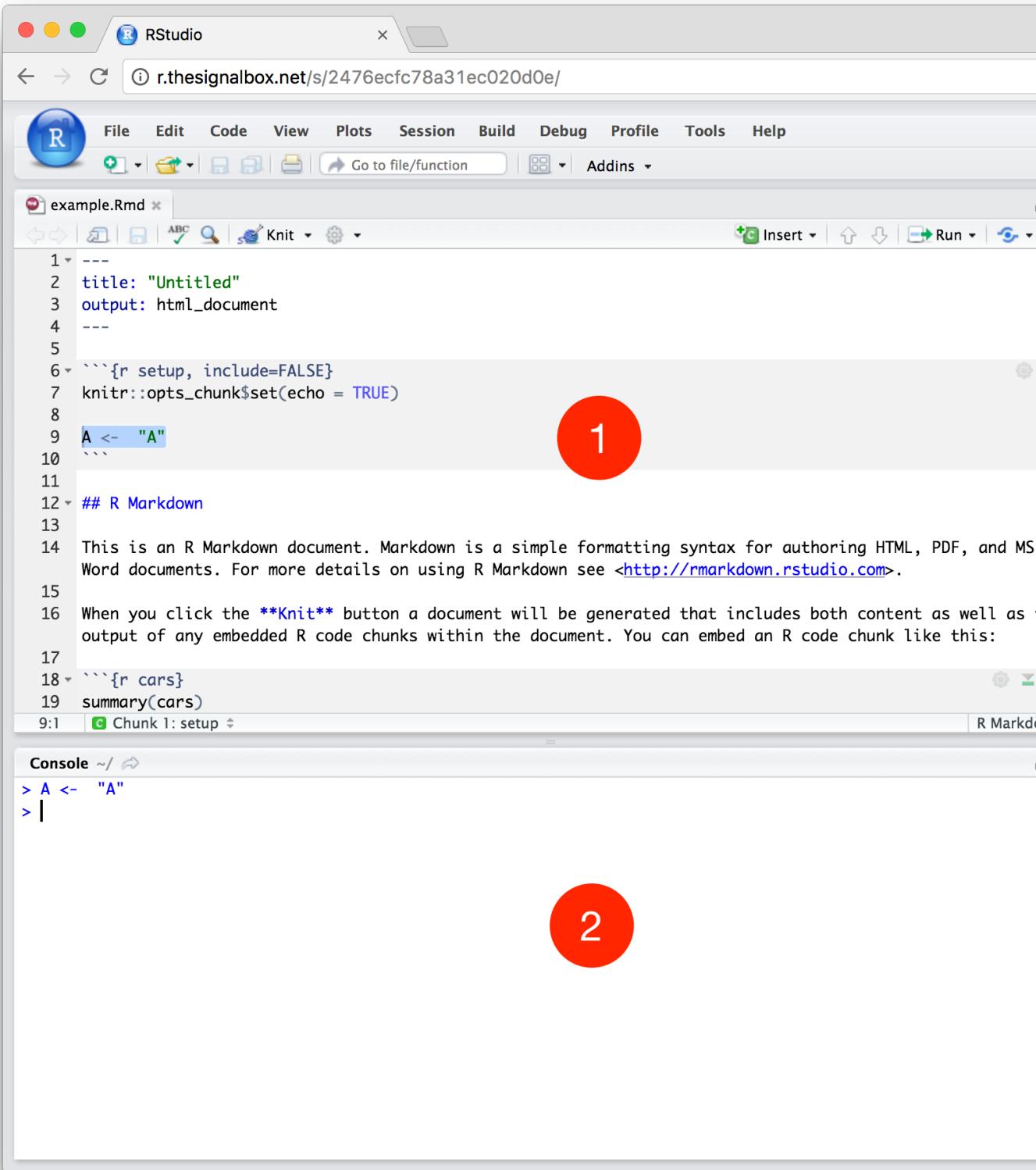
You are currently reading the output of an ‘RMarkdown’ document. An RMarkdown document mixes R code with Markdown:

- R is a computer language designed for working with data.
- Markdown is a simple text-based format which can include prose, hypertext links, images, and code (see <http://commonmark.org/help/>).

Like computer code, RMarkdown can be ‘run’ or ‘executed’. But in the language of RStudio, you ‘knit’ your RMarkdown to produce a finished document. This combines analyses, graphs, and explanatory text in a single pdf, html, or Word document which can be shared.

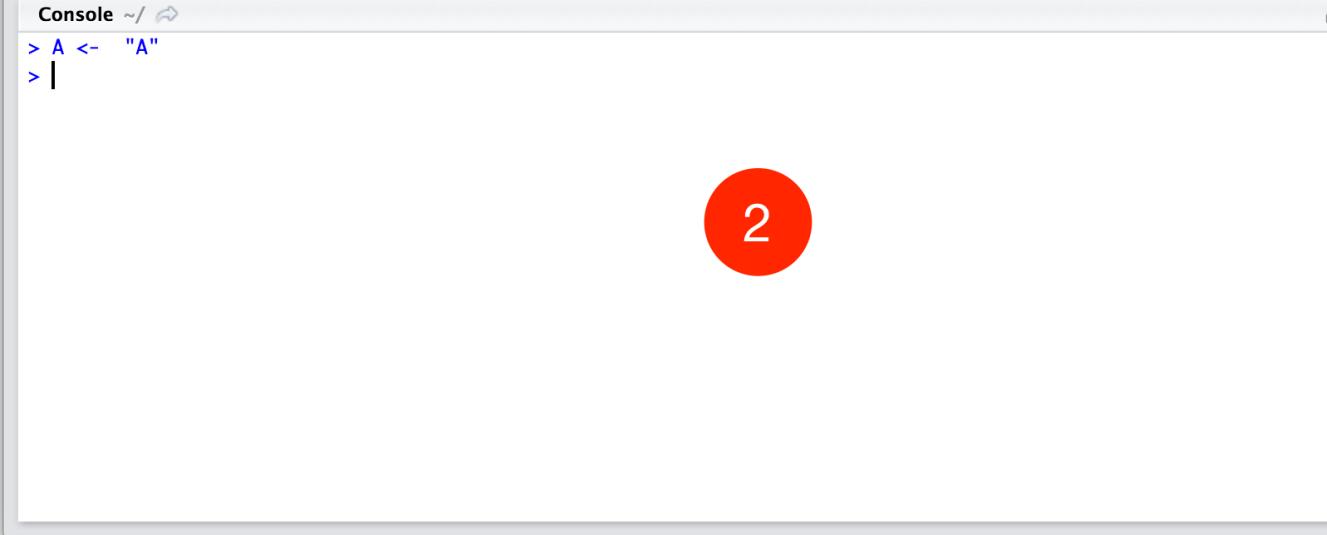
RStudio

RStudio is a special text editor that has been customised to make working with R easy. It can be installed on your own computer, or you can login to a shared RStudio server (for example, one run by your university) from a web browser. Either way the interface is largely the same and contains 4 main panels:



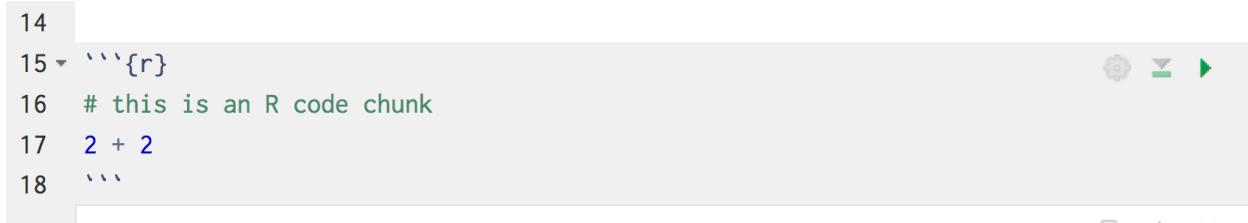
The screenshot shows the RStudio interface with an R Markdown file named "example.Rmd". A red circle with the number "1" is overlaid on the code editor area.

```
1 ---  
2 title: "Untitled"  
3 output: html_document  
4 ---  
5  
6 ```{r setup, include=FALSE}  
7 knitr::opts_chunk$set(echo = TRUE)  
8  
9 A <- "A"  
10 ````  
11  
12 ## R Markdown  
13  
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.  
15  
16 When you click the **Knit** button a document will be generated that includes both content as well as  
output of any embedded R code chunks within the document. You can embed an R code chunk like this:  
17  
18 ```{r cars}  
19 summary(cars)  
9:1 | C Chunk 1: setup
```



The screenshot shows the RStudio console window with the command "A <- "A"" entered and its output. A red circle with the number "2" is overlaid on the console output area.

```
Console ~/  
> A <- "A"  
> |
```



```
14
15 ~~~{r}
16 # this is an R code chunk
17 2 + 2
18 ~~~
```

Figure 1: A code chunk in the RMarkdown editor

The figure above shows the main RStudio interface, comprising:

1. The main R-script or RMarkdown editor window. This is where you write commands, which can then be executed (to run the current line type ctrl-Enter or cmd-Enter on a Mac).
2. The R console, into which you can type R commands directly, and see the output of commands run in the script editor.
3. The ‘environment’ panel, which lists all the variables you have defined and currently available to use.
4. The files and help panel. Within this panel the ‘files’ tab enables you to open files stored on the server, in the current project, or elsewhere on your hard drive.

You can see a short video demonstrating the RStudio interface here:

The video:

- Shows you how to type commands into the Console and view the results.
- Run a plotting function, and see the result.
- Create RMarkdown file, and ‘Knit’ it to produce a document containing the results of your code and explanatory text.

Once you have watched the video:

- Open RStudio and create a new RMarkdown document.
- Edit some of the text, and press the Knit button to see the results.
- Edit some of the R blocks and see what happens.

Creating code chunks

To include R code within RMarkdown we write 3 backticks (~~~), followed by {r}. We then include our R code, and close the block with 3 more backticks (how to find the backtick on your keyboard).

When a document including this chunk is run or ‘knitted’, the final result will include the the line 2+2 followed by the number 4 on the next line. We can use RMarkdown to ‘show our workings’: our analysis can be interleaved with narrative text to explain or interpret the calculations.

1.0.0.2.1 More about RMarkdown

A more in depth explanation of RMarkdown is here: <https://rmarkdown.rstudio.com>, and a detailed user guide here: <https://rmarkdown.rstudio.com/lesson-1.html>

First commands

You can type R commands directly into the ‘console’ (see here) and see the result there, but you should make a habit of working in an RMarkdown file. This keeps a record of everything you try, and makes it easy to edit/amend things which didn’t quite work.

Create a new Rmarkdown document from the ‘file’ menu in RStudio.

1.0.0.3

To run code in the RStudio interface put your cursor on a line within an R Block (or select the code you want to run), and press **Ctrl-Enter**. The result will appear below the code block.

The command in the R block below prints (i.e. shows on screen) the first few rows of a dataset that is built-in to R as an example, called `mtcars`.

Place your cursor somewhere in the line the command is on and run it by typing **Ctrl-Enter**, shown in this brief video:

Create an R block in RMarkdown, then run some simple commands.

```
head(mtcars)
  mpg cyl disp hp drat wt qsec vs am gear carb
Mazda RX4     21.0   6 160 110 3.90 2.620 16.46 0 1 4 4
Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02 0 1 4 4
Datsun 710    22.8   4 108 93 3.85 2.320 18.61 1 1 4 1
Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44 1 0 3 1
Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02 0 0 3 2
Valiant      18.1   6 225 105 2.76 3.460 20.22 1 0 3 1
```

If you are reading this from within RStudio, running `head(mtcars)` makes an interactive table in your document, which you can use this to browse the `mtcars` dataset.

If you are still reading the compiled html or pdf document you will see a static table containing the same data.

Hopefully at this point it’s obvious that RStudio and RMarkdown give you:

- A nice place to work your data interactively
- A way to ‘show your workings’ and save this for later
- A way to share your analysis

Naming things

We can assign labels to the results of calculations and other parts of our analyses to keep track of them.

To assign labels we use the `<-` symbol. The `<-` symbol points from the value we want to store, to the name we want to use. For example:

```
the_magic_number <- 3
```

This assigns the value 3 to the variable `the_magic_number`.

This block wouldn’t display anything because assigning a variable doesn’t create any output.

To both assign a variable *and* display it we would type:

```
the_magic_number <- 3
the_magic_number
[1] 3
```

Or we can use a shortcut: if we wrap the line in parentheses this both makes the assignment and prints the result to the console:

```
(i_am_a_new_variable <- 22)
[1] 22
```

We can also do calculations as we assign variables:

```
one_score <- 20
(four_score_years_and_ten <- one_score * 4 + 10)
[1] 90
```

We can give *anything* a label by assigning it to a variable.

It doesn't have to be a number; we can also assign letters, words, graphics, the results of a statistical model, or *lists* of any of these things.

This will come in handy later.

Vectors and lists

When working with data, we often have lists or sequences of 'things'. For example: a list of measurements we have made.

- When all the things are of the same type, R calls this a *vector*².
- When there is a mix of different things R calls this a *list*.

Vectors

We can create a vector of numbers and display it like this:

```
# this creates a vector of heights, in cm
heights <- c(203, 148, 156, 158, 167,
            162, 172, 164, 172, 187,
            134, 182, 175)
```

The `c()` command is shorthand for *combine*, so the example above combines the individual elements (numbers) into a new vector.

We can create a vector of alphanumeric names just as easily:

```
names <- c("Ben", "Joe", "Sue", "Rosa")
```

And we can check the values stored in these variables by printing them. You can either type `print(heights)`, or just write the name of the variable alone, which will print it by default. E.g.:

```
heights
[1] 203 148 156 158 167 162 172 164 172 187 134 182 175
```

1.0.0.4

Try creating your own vector of numbers in a new code block below³ using the `c(...)` command. Then change the name of the variable you assign it to.

Accessing elements

Once we have created a vector, we often want to access the individual elements again. We do this based on their *position*.

Let's say we have created a vector:

²It's actually a matrix if has 2 dimensions, like a table, or an array if it has more than 2 dimensions.

³i.e. edit the RMarkdown document

```
my.vector <- c(10, 20, 30, 40)
```

We can display the whole vector by just typing its name, as we saw above. But if we want to show only the *first* element of this vector, we type:

```
my.vector[1]  
[1] 10
```

Here, the square brackets specify a *subset* of the vector we want - in this case, just the first element.

Selecting more than one element

A neat feature of subsetting is that we can grab more than one element at a time.

To do this, we need to tell R the *positions* of the elements we want, and so we provide a *vector of the positions of the elements we want*.

It might seem obvious, but the first element has position 1, the second has position 2, and so on. So, if we wanted to extract the 4th and 5th elements from the vector of heights we saw above we would type:

```
elements.to.grab <- c(4, 5)  
heights[elements.to.grab]  
[1] 158 167
```

We can also make a subset of the original vector and assign it to a *new* variable:

```
first.two.elements <- heights[c(1, 2)]  
first.two.elements  
[1] 203 148
```

Making and slicing with sequences

One common task in R is to create sequences of numbers, letters or dates.

The simplest way of doing this is to define a range, with the colon:

```
onetoten <- 1:10  
onetoten  
[1] 1 2 3 4 5 6 7 8 9 10
```

This creates a vector which can be sliced like any other:

```
onetoten[8]  
[1] 8
```

One common use of sequences is to slice other vectors:

```
onetoten[1:3]  
[1] 1 2 3
```

Or the first 10 values in the `heights` vector we defined above:

```
heights[1:10]  
[1] 203 148 156 158 167 162 172 164 172 187
```

This works backwards, and with negative numbers too:

```
5:-5  
[1] 5 4 3 2 1 0 -1 -2 -3 -4 -5
```

When your sequence doesn't contain only whole numbers, or non-consecutive numbers, you can use the `seq` function:

```
seq(1,10,by=2)
[1] 1 3 5 7 9
seq(0, 1, by=.2)
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

Conditional slicing

One neat feature of R is that you can create a sequence of TRUE or FALSE values, by asking whether each value in a sequence matches a particular condition. For example:

```
1:10 > 5
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Re-using the heights vector from above, we can then use this to select values that are above the average:

```
heights > mean(heights)
[1] TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
[12] TRUE  TRUE
```

And we can use the vector of TRUE and FALSE values to select from the actual scores:

```
heights[heights > mean(heights)]
[1] 203 172 172 187 182 175
```

Working with vectors

Many of R's most useful functions process *vectors of numbers* in some way. For example (as we've already seen) if we want to calculate the average of our vector of heights we just type:

```
mean(heights)
[1] 167.6923
```

R contains *lots* of built in functions which we can use to summarise a vector of numbers. For example:

```
median(heights)
[1] 167
sd(heights)
[1] 17.59443
min(heights)
[1] 134
max(heights)
[1] 203
range(heights)
[1] 134 203
IQR(heights)
[1] 17
length(heights)
[1] 13
```

All of these functions accept a vector as input, do some proccesing, and then return a *single number* which gets displayed by RStudio.

But not all functions return a single number in the way that `mean` did above. Some return a new vector, or some other type of object instead. For example, the `quantile` function returns the values at the 0, 25th,

50th, 75th and 100th percentiles (by default).

```
height.quantiles <- quantile(heights)
height.quantiles
  0% 25% 50% 75% 100%
 134 158 167 175 203
```

If a function returns a vector, we can use it just like any other vector:

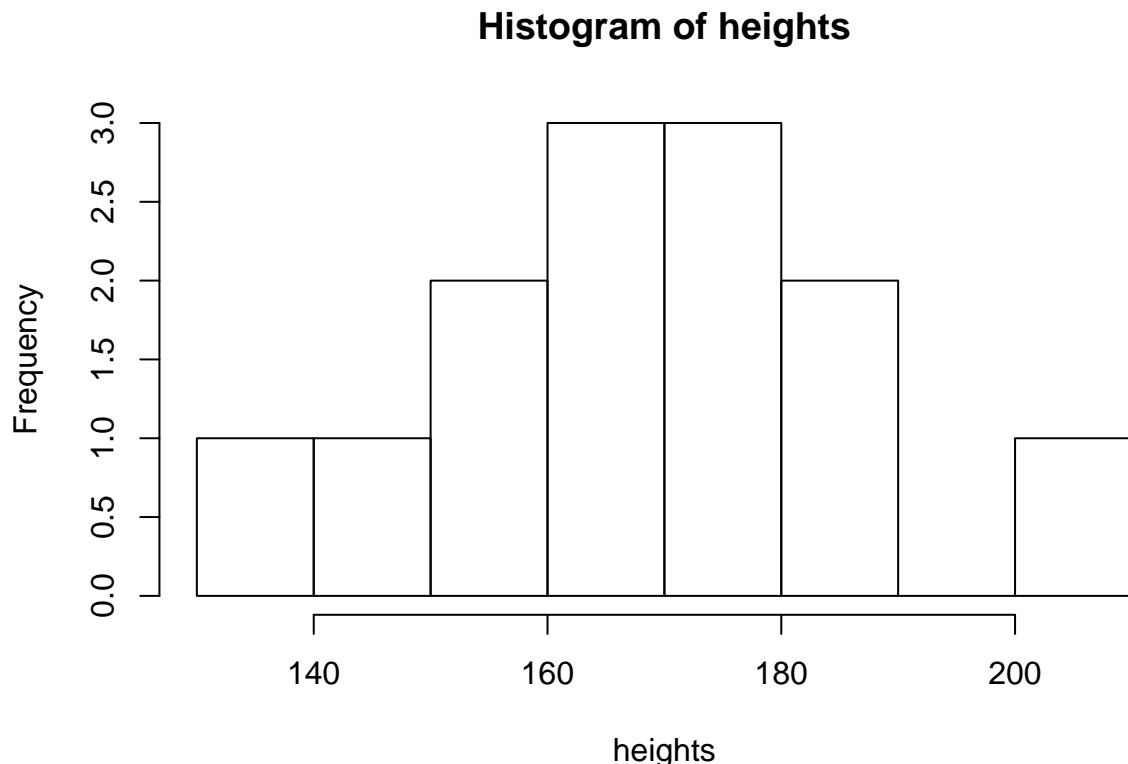
```
height.quantiles <- quantile(heights)

# grab the third element, which is the median
height.quantiles[3]
50%
167

# assign the first element to a variable
min.height <- height.quantiles[1]
min.height
0%
134
```

But other functions process a vector without returning any numbers. For example, the `hist` function returns a histogram:

```
hist(heights)
```



We'll cover lots more plotting and visualisation later on.

Making new vectors

So far we've seen R functions which process a vector of numbers and produce a single number, a new vector of a different length (like `quantile` or `fivenum`), or some other object (like `hist` which makes a plot). However many other functions accept a single input, do something to it, and return a single processed value.

For example, the square root function, `sqrt`, accepts a single value and returns a single value: running `sqrt(10)` will return 3.1623.

In R, if a function accepts a single value as input and returns a single value as output (like `sqrt(10)`), then you can usually give a vector as input too. Some people find this surprising⁴, but R assumes that if you're processing a vector of numbers, you want the function applied to each of them in the same way.

This turns out to be very useful. For example, let's say we want the square root of each of the elements of our height data:

```
# these are the raw values
heights
[1] 203 148 156 158 167 162 172 164 172 187 134 182 175

# takes the sqrt of each value and returns a vector of all the square roots
sqrt(heights)
[1] 14.24781 12.16553 12.49000 12.56981 12.92285 12.72792 13.11488
[8] 12.80625 13.11488 13.67479 11.57584 13.49074 13.22876
```

This also works with simple arithmetic So, if we wanted to convert all the heights from cm to meters we could just type:

```
heights / 100
[1] 2.03 1.48 1.56 1.58 1.67 1.62 1.72 1.64 1.72 1.87 1.34 1.82 1.75
```

This trick also works with other functions like `paste`, which combines the inputs you send it to produce an alphanumeric string:

```
paste("Once", "upon", "a", "time")
[1] "Once upon a time"
```

If we send a vector to `paste` it assumes we want a vector of results, with each element in the vector pasted next to each other:

```
bottles <- c(100, 99, 98, "...")
paste(bottles, "green bottles hanging on the wall")
[1] "100 green bottles hanging on the wall"
[2] "99 green bottles hanging on the wall"
[3] "98 green bottles hanging on the wall"
[4] "... green bottles hanging on the wall"
```

In other programming languages we might have had to write a 'loop' to create each line of the song, but R lets us write short statements to summarise *what* needs to be done; we don't need to worry worrying about *how* it gets done.

1.0.0.5

The `paste0` function does much the same, but leaves no spaces in the combined strings, which can be useful:

```
paste0("N=", 1:10)
[1] "N=1"  "N=2"  "N=3"  "N=4"  "N=5"  "N=6"  "N=7"  "N=8"  "N=9"  "N=10"
```

⁴Mostly people who already know other programming languages like C. It's not that surprising if you read the R code as you would English.

Making up data (new vectors)

Sometimes you'll need to create vectors containing regular sequences or randomly selected numbers.

To create regular sequences a convenient shortcut is the ‘colon’ operator. For example, if we type `1:10` then we get a vector of numbers from 1 to 10:

```
1:10  
[1] 1 2 3 4 5 6 7 8 9 10
```

The `seq` function allows you to create more specific sequences:

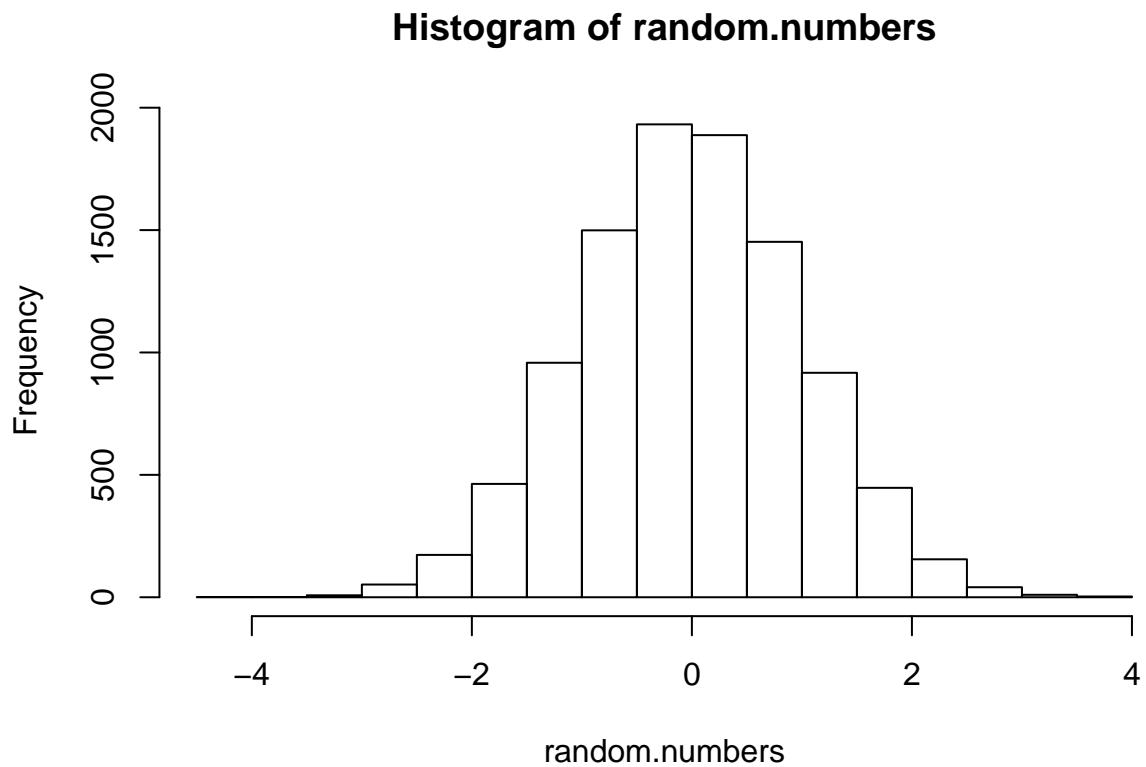
```
# make a sequence, specifying the interval between them  
seq(from=0.1, to=2, by=.1)  
[1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7  
[18] 1.8 1.9 2.0
```

We can also use random number-generating functions built into R to create vectors:

```
# 10 uniformly distributed random numbers between 0 and 1  
runif(10)  
[1] 0.68178208 0.90411456 0.50615136 0.98745549 0.45402399 0.03190653  
[7] 0.91578234 0.99891484 0.61407334 0.11438643  
  
# 1,000 uniformly distributed random numbers between 1 and 100  
my.numbers <- runif(1000, 1, 10)  
  
# 10 random-normal numbers with mean 10 and SD=1  
rnorm(10, mean=10)  
[1] 10.748900 10.159444 9.888892 11.338383 7.736228 9.424838 11.330858  
[8] 8.327885 9.348344 8.853372  
  
# 10 random-normal numbers with mean 10 and SD=5  
rnorm(10, 10, 5)  
[1] 11.7342244 7.9680336 7.9065691 9.5719237 8.1691346 10.3218285  
[7] 12.1666092 5.5941250 17.7576063 -0.7157989
```

We can then use these numbers in our code, for example plotting them:

```
random.numbers <- rnorm(10000)  
hist(random.numbers)
```



Functions to learn now

There are *thousands* of functions built into R. Below are just a few examples which are likely to be useful as you work with your data:

Repetition

```
# repeat something N times
rep("Apple pie", 10)
[1] "Apple pie" "Apple pie" "Apple pie" "Apple pie" "Apple pie"
[6] "Apple pie" "Apple pie" "Apple pie" "Apple pie" "Apple pie"

# repeat a short vector, combining into a single longer vector
rep(c("Custard", "Gravy"), 5)
[1] "Custard" "Gravy"    "Custard" "Gravy"    "Custard" "Gravy"    "Custard"
[8] "Gravy"    "Custard" "Gravy"
```

Sequences

```
# make a sequence
(countdown <- 100:1)
[1] 100  99  98  97  96  95  94  93  92  91  90  89  88  87  86  85  84
[18] 83   82   81   80   79   78   77   76   75   74   73   72   71   70   69   68   67
[35] 66   65   64   63   62   61   60   59   58   57   56   55   54   53   52   51   50
[52] 49   48   47   46   45   44   43   42   41   40   39   38   37   36   35   34   33
[69] 32   31   30   29   28   27   26   25   24   23   22   21   20   19   18   17   16
[86] 15   14   13   12   11   10   9    8    7    6    5    4    3    2    1
```

Make sequences with steps of a particular size:

```
(tenths <- seq(from=0, to=1, by=.1))
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

(twelfths <- seq(from=0, to=10, length.out=12))
[1] 0.0000000 0.9090909 1.8181818 2.7272727 3.6363636 4.5454545
[7] 5.4545455 6.3636364 7.2727273 8.1818182 9.0909091 10.0000000
```

Ranking

```
# generate some random data (here, ages in years)
ages <- round(rnorm(10, mean=40, sd=10))

# get the rank order of elements (i.e. what their positions would be if the vector was sorted)
ages
[1] 54 48 40 53 39 27 41 21 35 42
rank(ages, ties.method="first")
[1] 10 8 5 9 4 2 6 1 3 7
```

Unique values

```
# return the unique values in a vector
unique(rep(1:10, 100))
[1] 1 2 3 4 5 6 7 8 9 10
```

Lengths

```
# return the unique values in a vector
length(seq(1,100, 2))
[1] 50
```

Try and experiment with each of these functions. Check the output against what you expected to happen, and make sure you understand what they do.

Packages

R has been around for ages. It remains popular because it's *easy for people to add to it*.

You can run almost any statistical model and produce many different plots in R because users write 'packages' which extend the base language. For now we assume someone has helped you install all the packages you need⁵.

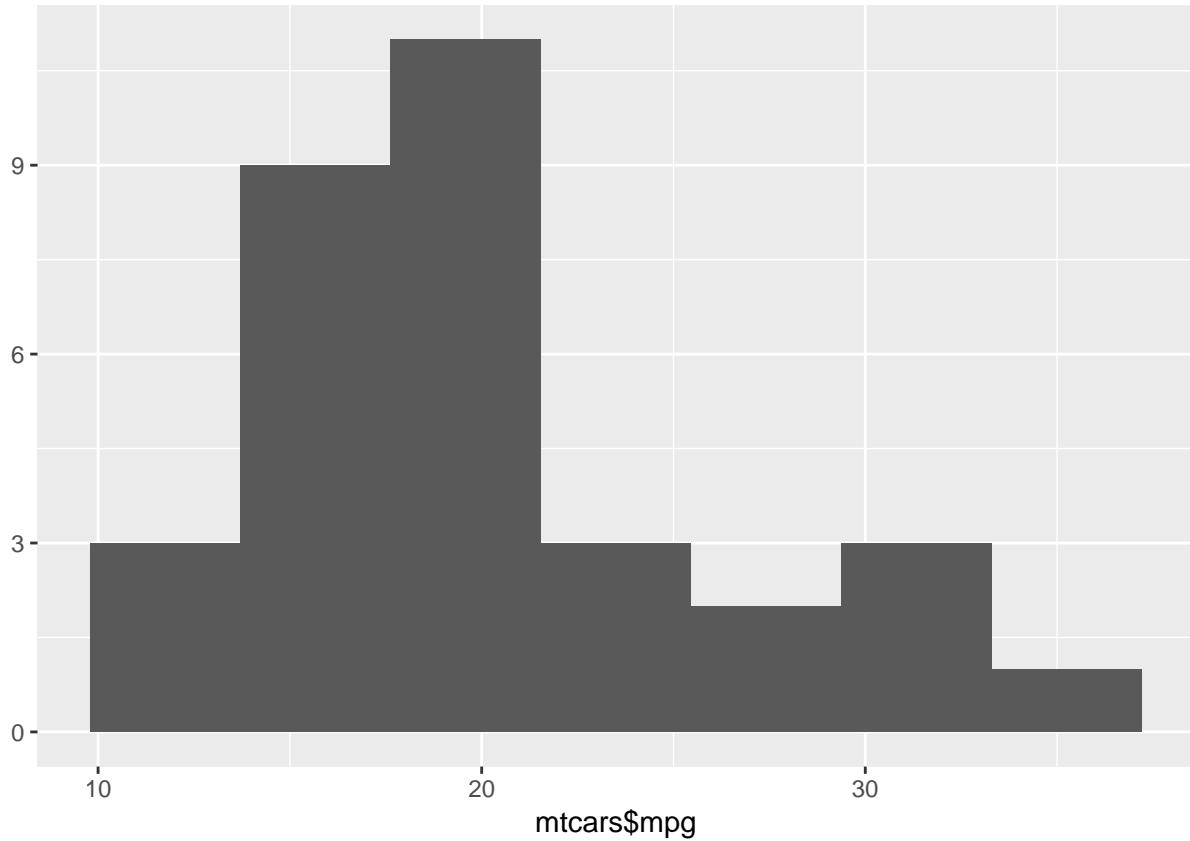
To access features in packages, you normally load the package with the `library()` function. Running `library(<packagename>)` loads all the new functions within it, and it is then possible to call them from your code. For example, typing:

```
library(ggplot2)
```

Will load the `ggplot2` package. You can then call the `qplot` function it provides:

```
qplot(mtcars$mpg, bins=7)
```

⁵See the installation guide if this isn't the case

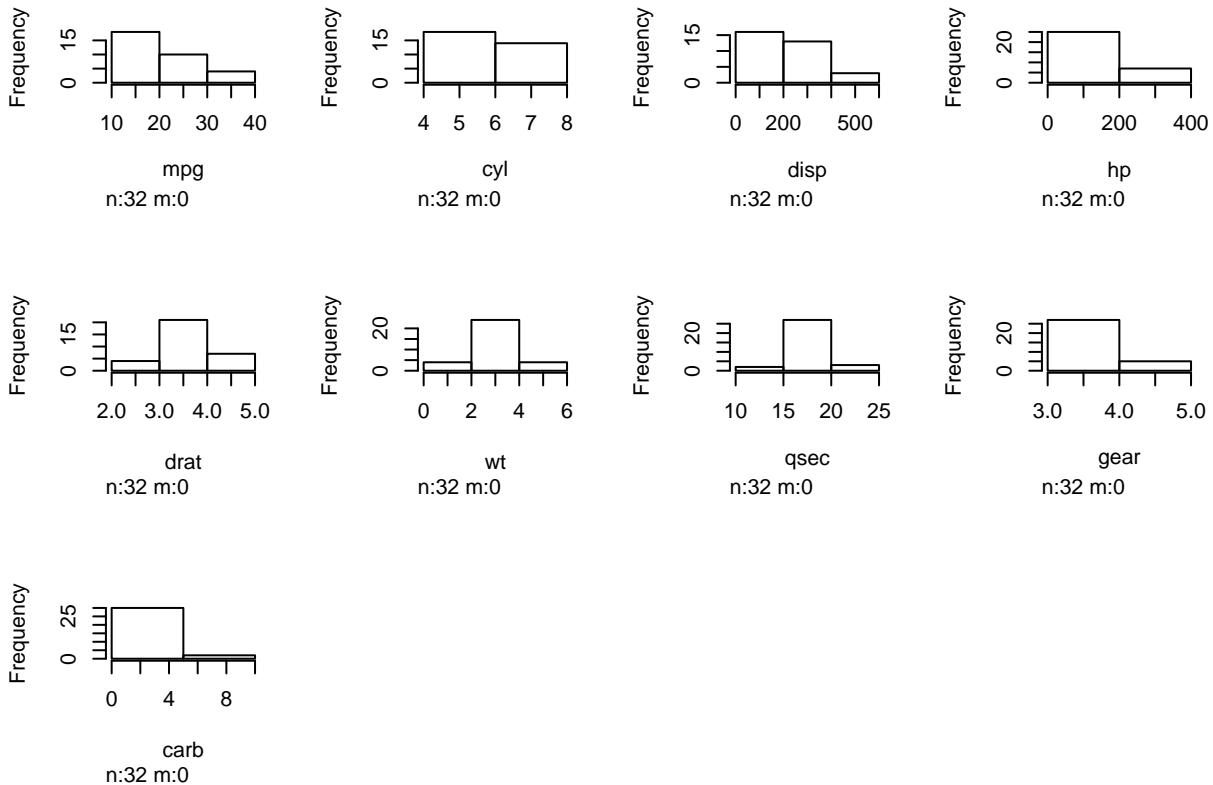


It's good style to load packages at the top of an R script, or in the first chunk of an RMarkdown document. This makes it easy for others to see what packages they need to install, and helps avoid certain sorts of errors in your code.

1.0.0.6

You don't strictly *need* to load packages to use their features. If a package is installed on your system you can also call a function it provides directly. In the example below we call the `hist.data.frame` from the `Hmisc` package, and obtain histograms of all the variables in the `mtcars` dataset:

```
Hmisc::hist.data.frame(mtcars)
```



The rule is to type `package::function(parameters)`, where `::` separates the package and function names. Parameters are just the inputs to the function.

There are two reasons not to load a package before using it:

1. Laziness: it can save typing if you just want to use one function from a package, and only once.
2. Explicitness: It's an unfortunate truth that some function names are repeated in different packages. This can be confusing if they work differently or do completely different things. If you don't know which package the version you are using comes from. Using `package_name:function_name` can help make things explicit.

Try using the `hist.data.frame` function in the `Hmisc` package on the `mtcars` data.

- First using the `::` syntax
- Then load the `Hmisc` package, and repeat without using the `::`.

Part II

Data

2 The `dataframe`

A `dataframe` is a container for our data.

It's much like a spreadsheet, but with some constraints applied. ‘Constraints’ might sound bad, but they're actually helpful: they make dataframes more structured and predictable to work with. The main constraints are that:

- Each column is a vector, and so can only store one type of data.
- Every column has to be the same length (although missing values are allowed).
- Each column must have a name.

A **tibble** is an updated version of a dataframe with a whimsical name, which is part of the **tidyverse**. It's almost exactly the same a dataframe, but with some rough edges smoothed off — it's safe and preferred to use **tibble** in place of **data.frame**.

You can make a simple tibble or dataframe like this:

```
data.frame(myvariable = 1:10)
myvariable
1      1
2      2
3      3
4      4
5      5
6      6
7      7
8      8
9      9
10     10
```

Using a tibble is much the same, but allows some extra tricks like creating one variable from another:

```
tibble(
  height_m = rnorm(10, 1.5, .2),
  weight_kg = rnorm(10, 65, 10),
  bmi = weight_kg / height_m ^ 2,
  overweight = bmi > 25
)
# A tibble: 10 x 4
  height_m weight_kg   bmi overweight
    <dbl>     <dbl> <dbl>   <lgl>
1     1.18     55.8  40.2  TRUE
2     1.30     57.5  34.0  TRUE
3     1.65     74.7  27.6  TRUE
4     1.51     48.0  20.9 FALSE
5     1.61     61.2  23.7 FALSE
6     1.20     65.8  45.9  TRUE
7     1.49     52.5  23.7 FALSE
8     1.81     63.7  19.5 FALSE
9     1.55     66.0  27.6  TRUE
10    1.45     57.3  27.4  TRUE
```

2.0.0.1 Using ‘built in’ data

The quickest way to see a dataframe in action is to use one that is built in to R (this page lists all the built-in datasets). For example:

```
head(airquality)
Ozone Solar.R Wind Temp Month Day
```

1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6

Or

```
head(mtcars)
#> #> #> #> #> #>
#> #> #> #> #> #>
#> #> #> #> #> #>
#> #> #> #> #> #>
#> #> #> #> #> #>
#> #> #> #> #> #>
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

In both these examples the datasets are already loaded and available to be used with the `head()` function.

To find a list of all the built in datasets you can type `help(datasets)` into the console, or see <https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html>.

Familiarise yourself with some of the other included datasets, e.g. `datasets::attitude`. Watch out that not all the included datasets are *dataframes*: Some are just vectors of observations (e.g. the `airmiles` data) and some are ‘time-series’, (e.g. the `co2` data)

2.0.0.2 Looking at dataframes

As we’ve already seen, using `print(df)` within an RMarkdown document creates a nice interactive table you can use to look at your data.

However you won’t want to print your whole data file when you Knit your RMarkdown document. The `head` function can be useful if you just want to show a few rows:

```
head(mtcars)
#> #> #> #> #> #>
#> #> #> #> #> #>
#> #> #> #> #> #>
#> #> #> #> #> #>
#> #> #> #> #> #>
#> #> #> #> #> #>
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

Or we can use `glimpse()` function from the `dplyr::` package (see the section on loading and using packages) for a different view of the first few rows of the `mtcars` data. This flips the dataframe so the variables are listed in the first column of the output:

```
glimpse(mtcars)
Observations: 32
Variables: 11
$ mpg <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19....
$ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 4, 4, ...
$ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 1...
$ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, ...
$ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.9...
$ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3...
```

```
$ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 2...
$ vs   <dbl> 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, ...
$ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ...
$ gear <dbl> 4, 4, 4, 3, 3, 3, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, ...
$ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, ...
```

You can use the `pander()` function (from the `pander::` package) to format tables nicely, for when you Knit a document to HTML, Word or PDF. For example:

```
library(pander)
pander(head(airquality), caption="Tables always need a caption.")
```

Table 1: Tables always need a caption.

Ozone	Solar.R	Wind	Temp	Month	Day
41	190	7.4	67	5	1
36	118	8	72	5	2
12	149	12.6	74	5	3
18	313	11.5	62	5	4
NA	NA	14.3	56	5	5
28	NA	14.9	66	5	6

See the section on sharing and publishing for more ways to format and present tables.

Other useful functions for looking at and exploring datasets include:

- `summary(df)`
- `psych::describe(df)`
- `skimr::skim(df)`

Experiment with a few of the functions for viewing/summarising dataframes.

There are also some helpful plotting functions which accept a whole dataframe as their input:

```
boxplot(airquality)
```

```
psych::cor.plot(airquality)
```

These plots might not be worth including in a final write-up, but are very useful when exploring your data.

Working with dataframes

Introducing the tidyverse

This guide deliberately ignores many common patterns for working with dataframes.

There are plenty of other guides for working in these older ways, but for beginners, these techniques can be confusing. The approach shown here is based only on functions in the `tidyverse`. Although simple — and easy to read — the approach is extremely flexible and covers almost all of the cases you will encounter when working with psychological data.

Specifically, we make extensive use of two tidyverse packages:

- `dplyr`: to select, filter and summarise data
- `ggplot2`: to make plots

To load the tidyverse first write:

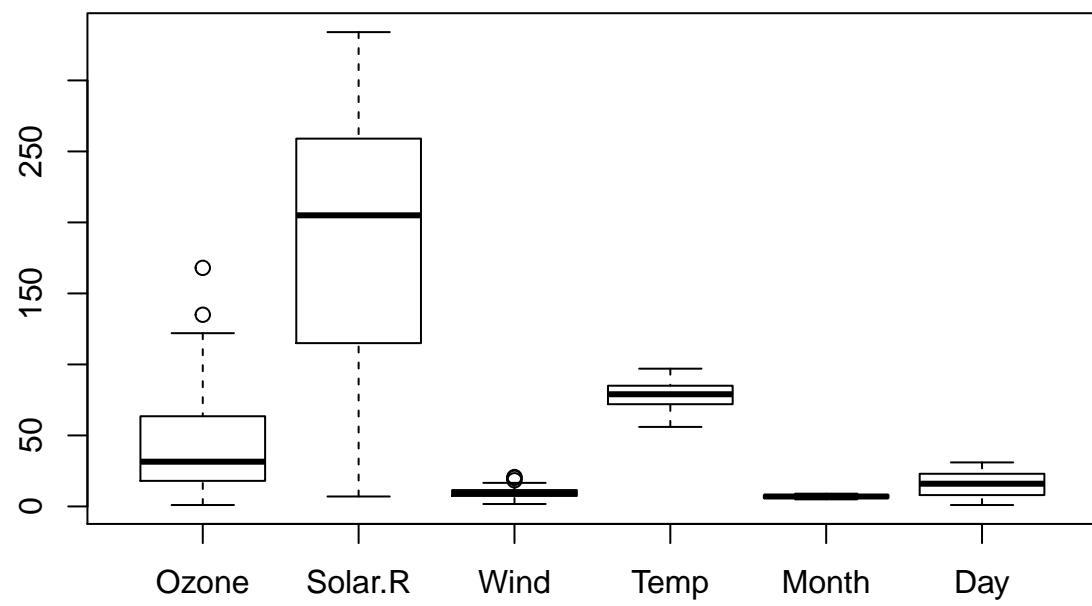


Figure 2: Box plot of all variables in a dataset.

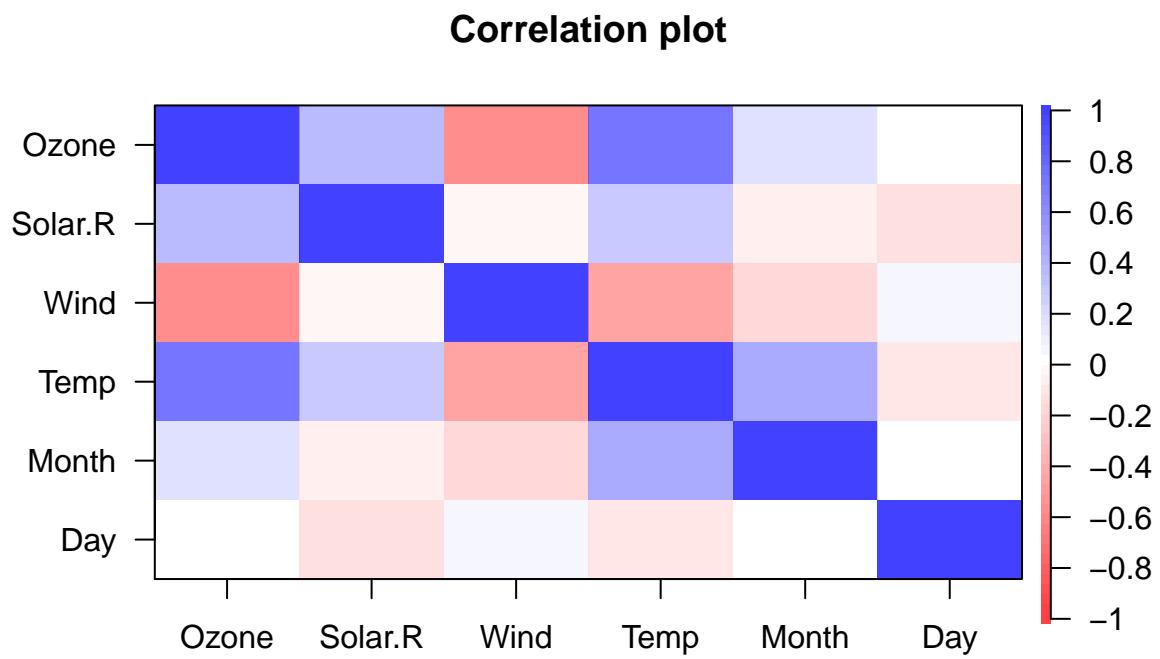


Figure 3: Correlation heatmap of all variables in a dataset. Colours indicate size of the correlation between pairs of variables.

```
library(tidyverse)
```

This can either be typed into the console or (better) included at the top of an markdown file.

Selecting columns

To pick out single or multiple columns use the `select()` function.

The `select()` function expects a dataframe as it's first input ('argument', in R language), followed by the names of the columns you want to extract with a comma between each name.

It returns a *new* dataframe with just those columns, in the order you specified:

```
head(  
  select(mtcars, cyl, hp)  
)  
      cyl  hp  
Mazda RX4     6 110  
Mazda RX4 Wag 6 110  
Datsun 710    4  93  
Hornet 4 Drive 6 110  
Hornet Sportabout 8 175  
Valiant       6 105
```

2.0.0.3 Saving a subset of the data

Because `dplyr` functions return a *new* dataframe, we can assign the results to a variable:

```
justcylandweight <- select(mtcars, cyl, wt)  
summary(justcylandweight)  
      cyl          wt  
Min. :4.000  Min. :1.513  
1st Qu.:4.000  1st Qu.:2.581  
Median :6.000  Median :3.325  
Mean   :6.188  Mean   :3.217  
3rd Qu.:8.000  3rd Qu.:3.610  
Max.  :8.000  Max.  :5.424
```

2.0.0.4 Excluding columns

If you want to keep most of the columns — perhaps you just want to get rid of one and keep the rest — put a minus (-) sign in front of the name of the column to drop. This then selects everything *except* the column you named:

```
# Note we are just dropping the Ozone column  
head(select(airquality, -Ozone))  
  Solar.R Wind Temp Month Day  
1     190  7.4   67    5   1  
2     118  8.0   72    5   2  
3     149 12.6   74    5   3  
4     313 11.5   62    5   4  
5      NA 14.3   56    5   5  
6      NA 14.9   66    5   6
```

2.0.0.5 Matching specific columns

You can use a patterns to match a subset of the columns you want. For example, here we select all the columns where the name contains the letter d:

```
head(select(mtcars, contains("d")))
      disp drat
Mazda RX4     160 3.90
Mazda RX4 Wag 160 3.90
Datsun 710    108 3.85
Hornet 4 Drive 258 3.08
Hornet Sportabout 360 3.15
Valiant       225 2.76
```

And you can combine these techniques to make more complex selections:

```
head(select(mtcars, contains("d"), -drat))
      disp
Mazda RX4     160
Mazda RX4 Wag 160
Datsun 710    108
Hornet 4 Drive 258
Hornet Sportabout 360
Valiant       225
```

2.0.0.6 Other methods of selection

As a quick reference, you can use the following ‘verbs’ to select columns in different ways:

- `starts_with()`
- `ends_with()`
- `contains()`
- `everything()`

See the help files for more information (type `??dplyr::select` into the console).

Selecting rows

To select rows from a dataframe use the `filter()` function (again from `dplyr`).

If we only wanted to rows for 6-cylindered cars, we could write:

```
filter(mtcars, cyl==6)
  mpg cyl disp hp drat    wt  qsec vs am gear carb
1 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
2 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
3 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
4 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
5 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
6 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
7 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
```

‘Operators’

When selecting rows in the example above we used two equals signs `==` to select rows where `cyl` was exactly 6.

As you might guess, there are other ‘operators’ we can use to create filters.

Rather than describe them, the examples below demonstrate what each of them do.

Equality and matching

As above, to compare a single value we use ==

```
2 == 2  
[1] TRUE
```

And in a filter:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
5	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
6	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
7	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6

You might have noted above that we write `==` rather than just `=` to define the criteria. This is because most programming languages, including R, use two `=` symbols to distinguish: *comparison* from *assignment*.

Presence/absence

To test if a value is in a vector of suitable matches we can use: `%in%`:

```
5 %in% 1:10  
[1] TRUE
```

Or for an example which is not true:

```
100 %in% 1:10  
[1] FALSE
```

Perhaps less obviously, we can test whether each value in a vector is *in* a second vector.

This returns a vector of TRUE/FALSE values as long as the first list:

```
c(1, 2) %in% c(2, 3, 4)  
[1] FALSE TRUE
```

This is very useful in a dataframe filter:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
4	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
6	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2

Here we selected all rows where `cyl` matched either 4 or 6. That is, where the value of `cyl` was ‘in’ the vector `c(4,6)`.

Greater/less than

The < and > symbols work as you'd expect:

```
head(filter(mtcars, cyl > 4))  
head(filter(mtcars, cyl < 5))
```

You can also use `>=` and `<=`:

```
filter(mtcars, cyl >= 6)  
filter(mtcars, cyl <= 4)
```

Negation (opposite of)

The `!` is very useful to tell R to reverse an expression; that is, take the opposite of the value. In the simplest example:

[1] FALSE

This is helpful because we can reverse the meaning of other expressions:

```
is.na(NA)
[1] TRUE
!is.na(NA)
[1] FALSE
```

And we can use in dplyr filters.

Here we select rows where Ozone is missing (NA):

```
filter(airquality, is.na(Ozone))
```

And here we use ! to reverse the expression and select rows which are not missing:

```
filter(airquality, !is.na(Ozone))
```

Try running these commands for yourself and experiment with changing the operators to make select different combinations of rows

Other logical operators

There are operators for ‘and’/‘or’ which can combine other filters.

Using `&` (and) with two conditions makes the filter more restrictive:

In contrast, the pipe symbol, `|`, means ‘or’, so we match more rows:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
2	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
3	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
4	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
5	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
6	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
7	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
8	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8

Finally, you can set the order in which operators are applied by using parentheses. This means these expressions are subtly different:

```
# first
filter(mtcars, (hp > 200 & wt > 4) | cyl==8)
  mpg cyl disp hp drat    wt  qsec vs am gear carb
1 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
2 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
3 16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
4 17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
5 15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
6 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
7 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
8 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
9 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
10 15.2  8 304.0 150 3.15 3.435 17.30  0  0    3    2
11 13.3  8 350.0 245 3.73 3.840 15.41  0  0    3    4
12 19.2  8 400.0 175 3.08 3.845 17.05  0  0    3    2
13 15.8  8 351.0 264 4.22 3.170 14.50  0  1    5    4
14 15.0  8 301.0 335 3.54 3.570 14.60  0  1    5    8

# second reordered evaluation
filter(mtcars, hp > 200 & (wt > 4 | cyl==8))
  mpg cyl disp hp drat    wt  qsec vs am gear carb
1 14.3   8 360 245 3.21 3.570 15.84  0  0    3    4
2 10.4   8 472 205 2.93 5.250 17.98  0  0    3    4
3 10.4   8 460 215 3.00 5.424 17.82  0  0    3    4
4 14.7   8 440 230 3.23 5.345 17.42  0  0    3    4
5 13.3   8 350 245 3.73 3.840 15.41  0  0    3    4
6 15.8   8 351 264 4.22 3.170 14.50  0  1    5    4
7 15.0   8 301 335 3.54 3.570 14.60  0  1    5    8
```

Try writing in plain English the meaning of the two filter expressions above

Sorting

Sort dataframes using `arrange()` from `dplyr`:

```
airquality %>%
  arrange(Ozone) %>%
  head
  Ozone Solar.R Wind Temp Month Day
1 1 8 9.7 59 5 21
2 4 25 9.7 61 5 23
3 6 78 18.4 57 5 18
4 7 NA 6.9 74 5 11
5 7 48 14.3 80 7 15
6 7 49 10.3 69 9 24
```

By default sorting is ascending, but you can use a minus sign to reverse this:

```
airquality %>%
  arrange(-Ozone) %>%
  head
  Ozone Solar.R Wind Temp Month Day
```

1	168	238	3.4	81	8	25
2	135	269	4.1	84	7	1
3	122	255	4.0	89	8	7
4	118	225	2.3	94	8	29
5	115	223	5.7	79	5	30
6	110	207	8.0	90	8	9

You can sort on multiple columns too, but the order of the variables makes a difference. This:

```
airquality %>%
  select(Month, Ozone) %>%
  arrange(Month, -Ozone) %>%
  head
Month Ozone
1     5   115
2     5    45
3     5    41
4     5    37
5     5    36
6     5    34
```

Is different to this:

```
airquality %>%
  select(Month, Ozone) %>%
  arrange(-Ozone, Month) %>%
  head
Month Ozone
1     8   168
2     7   135
3     8   122
4     8   118
5     5   115
6     8   110
```

Pipes

We often want to combine `select` and `filter` (and other functions) to return a subset of our original data.

One way to achieve this is to ‘nest’ function calls.

Taking the `mtcars` data, we can select the weights of cars with a poor `mpg`:

```
gas.guzzlers <- select(filter(mtcars, mpg < 15), wt)
summary(gas.guzzlers)
wt
Min. :3.570
1st Qu.:3.840
Median :5.250
Mean   :4.686
3rd Qu.:5.345
Max.   :5.424
```

This is OK, but can be confusing to read. The more deeply nested we go, the easier it is to make a mistake.

2.0.0.7 tidyverse provides an alternative to nested function calls, called the ‘pipe’



Figure 4: Think of your data ‘flowing’ down the screen.

Imagine your dataframe as a big bucket, containing data.

From this bucket, you can ‘pour’ your data down the screen, and it passes through a series of tubes and filters.

At the bottom of your screen you have a smaller bucket, containing only the data you want.

The ‘pipe’ operator, `%>%` makes our data ‘flow’ in this way:

```
big.bucket.of.data <- mtcars

big.bucket.of.data %>%
  filter(mpg <15) %>%
  select(wt) %>%
  summary
  wt
Min. :3.570
1st Qu.:3.840
Median :5.250
```

```
Mean    : 4.686
3rd Qu.: 5.345
Max.    : 5.424
```

The `%>%` symbol makes the data flow onto the next step. Each function which follows the pipe takes the incoming data as its first input.

Pipes do the same thing as nesting functions, but the code stays more readable.

It's especially nice because the order in which the functions happen is the same as the order in which we read the code (the opposite is true for nested functions).

We can save intermediate 'buckets' for use later on:

```
smaller.bucket <- big.bucket.of.data %>%
  filter(mpg < 15) %>%
  select(wt)
```

This is an incredibly useful pattern for processing and working with data.

We can 'pour' data through a series of filters and other operations, saving intermediate states where necessary.

You can insert the `%>%` symbol in RStudio by typing `cmd-shift-M`, which saves a lot of typing.

Modifying and creating new columns

We often want to compute new columns from data we already have.

Imagine we had heights stored in cm, and weights stored in kg for 100 participants in a study on weight loss:

```
set.seed(1234)

weightloss <- tibble(
  height_cm = rnorm(100, 150, 20),
  weight_kg = rnorm(100, 65, 10)
)

weightloss %>% head
# A tibble: 6 x 2
  height_cm weight_kg
  <dbl>     <dbl>
1 126.      69.1
2 156.      60.3
3 172.      65.7
4 103.      60.0
5 159.      56.7
6 160.      66.7
```

If we want to compute each participants' Body Mass Index, we first need to convert their height into meters. We do this with `mutate`:

```
weightloss %>%
  mutate(height_meters = height_cm / 100) %>%
  head
# A tibble: 6 x 3
  height_cm weight_kg height_meters
  <dbl>     <dbl>        <dbl>
1 126.      69.1        1.26
2 156.      60.3        1.56
```

3	172.	65.7	1.72
4	103.	60.0	1.03
5	159.	56.7	1.59
6	160.	66.7	1.60

We then want to calculate BMI:

```
weightloss %>%
  mutate(height_meters = height_cm / 100,
        bmi = weight_kg / height_meters ^ 2) %>%
head
# A tibble: 6 x 4
  height_cm weight_kg height_meters   bmi
    <dbl>     <dbl>      <dbl> <dbl>
1     126.      69.1      1.26  43.7
2     156.      60.3      1.56  24.9
3     172.      65.7      1.72  22.3
4     103.      60.0      1.03  56.4
5     159.      56.7      1.59  22.6
6     160.      66.7      1.60  26.0
```

You could skip the intermediate step of converting to meters and write: `bmi = weight_kg / (height_cm/100) ^ 2`. But it's often best to be explicit and simplify each operation.

3 ‘Real’ data

Note: If you already lucky enough to have nicely formatted data, ready for use in R, then you could skip this section and revisit it later, save for the section on factors and other variable types.

Most tutorials and textbooks use neatly formatted example datasets to illustrate particular techniques. However in the real-world our data can be:

- In the wrong format
- Spread across multiple files
- Badly coded, or with errors
- Incomplete, with values missing for many different reasons

This chapter will give you techniques to address each of these problems.

Importing data

If you have data outside of R, *the simplest way to import it is to first save it as a comma or tab-separated text file*, normally with the file extension `.csv` or `.txt`⁶.

Let's say we have file called `angry_moods.csv` in the same directory as our `.Rmd` file. We can read this data using the `read_csv()` function from the `readr` package⁷:

```
angry.moods <- readr::read_csv('data/angry_moods.csv')
head(angry.moods)
# A tibble: 6 x 7
  Gender Sports Anger.Out Anger.In Control.Out Control.In Anger.Expression
```

⁶This is easy to achieve in Excel and most other stats packages using the `Save As...` menu item

⁷There are also standard functions built into R, such as `read.csv()` or `read.table()` for importing data. These are fine if you can't install the `readr` package for some reason, but they are quite old and the default behaviour is sometimes counterintuitive. I recommend using the `readr` equivalents: `read_csv()` or `read_tsv()`.

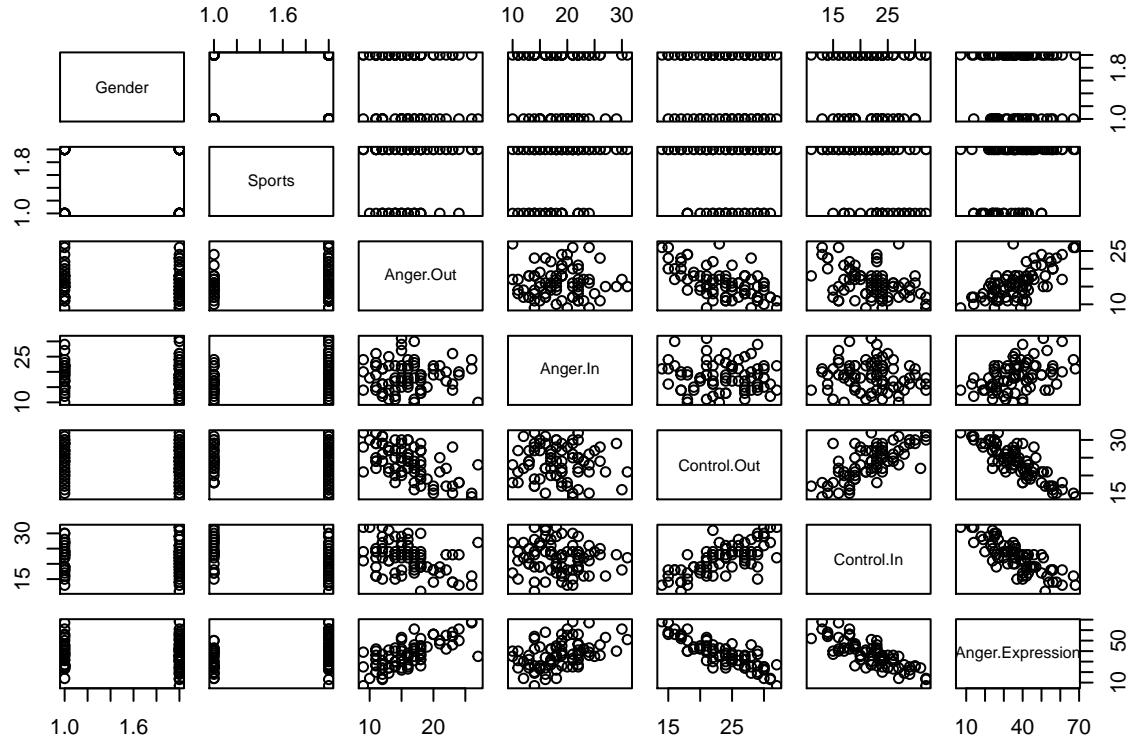
	<dbl>	<dbl>		<dbl>	<dbl>		<dbl>		<dbl>
1	2	1		18	13		23		20
2	2	1		14	17		25		24
3	2	1		13	14		28		28
4	2	1		17	24		23		23
5	1	1		16	17		26		28
6	1	1		16	22		25		23
									38

As you can see, when loading the `.csv` file the `read_csv()` makes some assumptions about the *type* of data the file contains. In this case, all the columns contain integer values. It's worth checking this message to make sure that stray cells in the file you are importing don't cause problems when importing. Excel won't complain about this sort of thing, but R is more strict and won't mix text and numbers in the same column.

A common error is for stray notes or text values in a spreadsheet to cause a column which should be numeric to be converted to the `character` type.

Once it's loaded, you can use this new dataset like any other:

```
pairs(angry.moods)
```



Importing data over the web

One neat feature of the `readr` package is that you can import data from the web, using a URL rather than a filename on your local computer. This can be really helpful when sharing data and code with colleagues. For example, we can load the `angry_moods.csv` file from a URL:

```
angry.moods.from.url <- readr::read_csv(  
  "https://raw.githubusercontent.com/benwhalley/just-enough-r/master/angry_moods.csv")  
  
head(angry.moods.from.url)
```

Importing from SPSS and other packages

This is often more trouble than it's worth. If using Excel for example, it's best just to save your data a csv file first and import that.

But if you really must use other formats see <https://www.datacamp.com/community/tutorials/r-data-import-tutorial>.

Saving and exporting

Before you start saving data in csv or any other format, ask yourself: “Do I need to save *this* dataset, or should I simply save my raw data and code?”

Oftentimes it's best to keep only your raw datafiles, with the R code you used to process them. This keeps your disk tidier, and avoids confusion with multiple versions of files.

If it takes a long time to process your data though you might want to save interim steps. And if you share your data (which you should) you might also want to save simplified or anonymised versions of it, in widely-accessible formats.

Use CSV files

Comma-separated-values files are a plain text format which are ideal for storing and sharing your data. They are:

- Understood by almost every piece of software, ever
- Will be readable in future
- Perfect for storing 2D data (like dataframes)
- Readable by humans (just open them in Notepad)

Commercial formats like Excel, SPSS (.sav) and Stata (.dta) don't have these properties.

Although CSV has some disadvantages, they are all easily overcome if you save the steps of your data processing and analysis in your R code, see below.

Saving a dataframe to .csv is as simple as:

```
readr::write_csv(mtcars, 'mtcars.csv')
```

If you run this within an RMarkdown document, this will create the new csv file in the same directory as your .Rmd file.

You can also use the `write.csv()` function in base R, but this version from `readr` is faster and has more sensible defaults (e.g. it doesn't write rownames, but does save column names in the first row)

3.0.0.1 Save processes, not just outcomes

Many students (and academics) make errors in their analyses because they process data by hand (e.g. editing files in Excel) or use GUI tools to run analyses.

In both cases these errors are hard to identify or rectify because only the outputs of the analysis can be saved, and *no record has been made of how these outputs were produced*.

In contrast, if you do your data processing and analysis in R/RMarkdown you benefit from a concrete, repeatable series of steps which can be checked/verified by others. This can also save lots of time if you need to process additional data later on (e.g. if you run more participants).

Some principles to follow when working:

- Save your raw data in the simplest possible format, in CSV
- Always include column names in the file
- Use descriptive names, but with a regular structure.
- Never include spaces or special characters in the column names. Use underscores (_) if you want to make things more readable.
- Make names <20 characters in length if possible

3.0.0.2 Saving interim steps

If you are saving data to use again later in R, the best format is RDS. Saving files to RDS is covered in a later section (click to see).

If you are saving interim steps but think you might possibly want to access it from other programmes in future use csv though.

To save something using RDS:

```
# create a huge df of random numbers...
massive.df <- data_frame(nums = rnorm(1:1e6))
saveRDS(massive.df, file="massive.RDS")
```

Then later on you can load it like this:

```
restored.massive.df <- readRDS('massive.RDS')
```

If you do this in RMarkdown, by default the RDS files will be saved in the same directory as your .Rmd file.

Archiving, publication and sharing

If you want to share data with someone else, or open it in a different software package, using '.csv' format is strongly recommended unless some other format is common in your field.

When archiving data, or sharing with others, you must document what each column measures, and any processing steps used to create the file. RMarkdown is a good way of doing this because it can combine the processing with narrative explaining what is being done, and why.

Dealing with multiple files

Often you will have multiple data files - for example, those produced by experimental software.

This is one of the few times when you might have to do something resembling 'real programming', but it's still fairly straightforward.

In the repeated measures Anova example later on in this guide we encounter some data from an experiment where reaction times were recorded in 25 trials (`Trial`) before and after (`Time`) one of 4 experimental manipulations (`Condition = {1,2,3,4}`). There were 48 participants in total:

Let's say that we have saved all the files are in a single directory, and these are numbered sequentially: `person01.csv`, `person02.csv` and so on.

Using the `list.files()` function we can list the contents of a directory on the hard drive:

```
list.files('data/multiple-file-example/')

[1] "person1.csv"  "person10.csv" "person11.csv" "person12.csv"
[5] "person13.csv" "person14.csv" "person15.csv" "person16.csv"
[9] "person17.csv" "person18.csv" "person19.csv" "person2.csv"
[13] "person20.csv" "person21.csv" "person22.csv" "person23.csv"
[17] "person24.csv" "person25.csv" "person26.csv" "person27.csv"
[21] "person28.csv" "person29.csv" "person3.csv"  "person30.csv"
[25] "person31.csv" "person32.csv" "person33.csv" "person34.csv"
[29] "person35.csv" "person36.csv" "person37.csv" "person38.csv"
[33] "person39.csv" "person4.csv"  "person40.csv" "person41.csv"
[37] "person42.csv" "person43.csv" "person44.csv" "person45.csv"
[41] "person46.csv" "person47.csv" "person48.csv" "person5.csv"
[45] "person6.csv"  "person7.csv"  "person8.csv"  "person9.csv"
```

The `list.files()` function creates a vector of the names of all the files in the directory.

At this point, there are many, many ways of importing the contents of these files, but below we use a technique which is concise, reliable, and less error-prone than many others. It also continues to use the `dplyr` library.

This approach has 3 steps:

1. Put all the names of the .csv files into a dataframe.
2. For each row in the dataframe, run a function which imports the file as a dataframe.
3. Combine all these dataframes together.

Putting the filenames into a dataframe

Because `list.files` produces a vector, we can make them a column in a new dataframe:

```
raw.files <- data_frame(filename = list.files('data/multiple-file-example/'))
```

And we can make a new column with the complete path (i.e. including the directory holding the files), using the `paste0` which combines strings of text. We wouldn't have to do this if the raw files were in the same directory as our RMarkdown file, but that would get messy.

```
raw.file.paths <- raw.files %>%
  mutate(filepath = paste0("data/multiple-file-example/", filename))

raw.file.paths %>%
  head(3)
# A tibble: 3 x 2
  filename      filepath
  <chr>        <chr>
1 person1.csv  data/multiple-file-example/person1.csv
2 person10.csv  data/multiple-file-example/person10.csv
3 person11.csv  data/multiple-file-example/person11.csv
```

Using `do()`

We can then use the `do()` function in `dplyr::` to import the data for each file and combine the results in a single dataframe.

The `do()` function allows us to run any R function for each group or row in a dataframe.

The means that our original dataframe is broken up into chunks (either groups of rows, if we use `group_by()`, or individual rows if we use `rowwise()`) and each chunk is fed to the function we specify. This function must do its work and return a new dataframe, and these are then combined into a single larger dataframe.

So in this example, we break our dataframe of filenames up into individual rows using `rowwise` and then specify the `read_csv` function which takes the name of a csv file, and returns the content as a dataframe (see the importing data section).

For example:

```
raw.data <- raw.file.paths %>%
  # 'do' the function for each row in turn
  rowwise() %>%
  do(., read_csv(file=.$filepath))
```

We can check these data look OK by sampling 10 rows at random:

```
raw.data %>%
  sample_n(10) %>%
  pander()
```

Condition	trial	time	person	RT
3	17	2	35	360.2
1	3	2	1	220.4
1	15	2	7	191
4	1	2	37	361
3	19	1	28	246.8
3	16	1	30	199.6
3	15	2	30	206.2
4	5	1	39	295.5
2	22	2	15	298.7
1	24	1	5	237.2

3.0.0.2.1 Using custom functions with `do()`

In this example, each of the raw data files included the participant number (the `person` variable). However, this isn't always the case.

This isn't a problem though, if we create our own helper function to import the data. Writing small functions in R is very easy, and the example below wraps the `read.csv()` function and adds a new column, `filename` to the imported data frame which would enable us to keep track of where each row in the final combined dataset came from.

This is the helper function:

```
read.csv.and.add.filename <- function(filepath){
  read_csv(filepath) %>%
    mutate(filename=filepath)
}
```

In English, you should read this as:

“Create a new R function called `read.csv.and.add.filename` which expects to be passed a path to a csv file as an input. This function reads the csv file at the path (converting it to a dataframe), and adds a new column containing the original file path it read from. It then returns this dataframe.”

We can use our helper function with `do()` in place of the bare `read_csv` function we used before:

```

raw.data.with.paths <- raw.file.paths %>%
  rowwise() %>%
  do(., read.csv.and.add.filename(.filepath))

raw.data.with.paths %>%
  sample_n(10) %>%
  pander()

```

Table 3: Table continues below

Condition	trial	time	person	RT
2	25	2	17	90.31
2	5	1	22	341.3
1	2	1	8	239.3
4	13	1	42	197.1
2	17	1	16	169.5
2	6	2	15	166.4
3	22	1	30	214
4	3	1	48	238.7
4	6	2	48	273.5
2	6	1	15	232.1

filepath
data/multiple-file-example/person17.csv
data/multiple-file-example/person22.csv
data/multiple-file-example/person8.csv
data/multiple-file-example/person42.csv
data/multiple-file-example/person16.csv
data/multiple-file-example/person15.csv
data/multiple-file-example/person30.csv
data/multiple-file-example/person48.csv
data/multiple-file-example/person48.csv
data/multiple-file-example/person15.csv

At this point you might need to use the `extract()` or `separate()` functions to post-process the filename and re-create the `person` variable from this (although in this case that's already been done for us).

Joining different datasets

Many analyses combine data from different sources.

Even within a single study, you may find that you have different datafiles (e.g. spreadsheets) for different sorts of information. For example you might have

1. Multiple csv files output by experimental software.
2. A spreadsheet of demographic characteristics of participants (e.g. their age, gender, handedness)
3. Questionnaire data, collected before the reaction time task.

Your main analysis might want to model individual RTs using predictors including age, handedness or some personality variable. Thus, you probably want data which look something like this:

```
df %>%
  pander()
```

person	trial	condition	female	left.handed	age	extraversion	RT
1	1	A	TRUE	FALSE	19	50	251.7
1	2	A	TRUE	FALSE	19	50	311.1
1	3	A	TRUE	FALSE	19	50	343.4
1	...	A	TRUE	FALSE	19	50	206.2
2	1	B	FALSE	FALSE	24	34	317.2
2	2	B	FALSE	FALSE	24	34	320.2
2	3	B	FALSE	FALSE	24	34	277
2	...	B	FALSE	FALSE	24	34	278.1

3.0.0.3 The raw data

In this example, we could imagine our raw data files looking like this:

```
rts %>%
  pander()
```

person	trial	RT
1	1	251.7
1	2	311.1
1	3	343.4
1	...	206.2
2	1	317.2
2	2	320.2
2	3	277
2	...	278.1

```
demographics %>%
  pander()
```

person	age	left.handed	female
1	19	FALSE	TRUE
2	24	FALSE	FALSE
3	33	TRUE	FALSE

And:

```
personality %>%
  pander()
```

person	extraversion
1	50
2	34
3	47

3.0.0.4 Joining the parts

To create the combined data file we want, we have to *join* the different files together.

As you might have noticed, though, the RT's file has many observations per participant, whereas the demographics and personality data has one row per person.

What we need is to smush this wide format data into the RTs file, such that values get repeated for every row of each participants' data.

To do this, the simplest method is to use the `dplyr::left_join()` function:

```
left_join(rts, demographics, by="person") %>%  
  pander
```

person	trial	RT	age	left.handed	female
1	1	251.7	19	FALSE	TRUE
1	2	311.1	19	FALSE	TRUE
1	3	343.4	19	FALSE	TRUE
1	...	206.2	19	FALSE	TRUE
2	1	317.2	24	FALSE	FALSE
2	2	320.2	24	FALSE	FALSE
2	3	277	24	FALSE	FALSE
2	...	278.1	24	FALSE	FALSE

In this example I explicitly set `id="person"` to let R know which variable to use to match the rows of data, although you don't *have* to, and `dplyr` can normally guess. You *do* have to use the same variable name in both files though (so, `person` and `participant` couldn't be matched, for example).

3.0.0.4.1 Other types of joins

As the `dplyr` manual states: left joins return for two dataframes, x and y will return all rows from x, and all columns from both x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

Left joins are probably the most useful, but there are other types which can be useful. To see all of them look at the help files (`help('join', 'dplyr')`).

To show one common example, if we wanted to check whether we were missing RT data for one of our participants, we could use an `anti_join`:

```
anti_join(demographics, rts, by="person") %>%  
  pander
```

person	age	left.handed	female
3	33	TRUE	FALSE

This table lists the data for *all the people in the demographics file, for whom we don't have RT data*. This can be a useful way of checking you haven't mislaid a raw datafile (e.g. forgot to copy it from the lab machine!).

Types of variable

When working with data in Excel or other packages like SPSS you've probably become aware that different types of data get treated differently.

For example, in Excel you can't set up a formula like `=SUM(...)` on cells which include letters (rather than just numbers). It doesn't make sensible.

However, Excel and many other programmes will sometimes make guesses about what to do if you combine different types of data.

For example, in Excel, if you add 28 to 1 Feb 2017 the result is 1 March 2017. This is sometimes what you want, but can often lead to unexpected results and errors in data analyses.

R is much more strict about not mixing types of data. Vectors (or columns in dataframes) can only contain one type of thing. In general, there are probably 4 types of data you will encounter in your data analysis:

- Numeric variables
- Character variables
- Factors
- Dates

The file `data/lakers.RDS` contains a dataset adapted from the `lubridate:::lakers` dataset (this is a dataset built into an add-on package for R).

This dataset contains four variables to illustrate the common variable types (a subset of the original dataset which provides scores and other information from each Los Angeles Lakers basketball game in the 2008-2009 season). We have the `date`, `opponent`, `team`, and `points` variables.

```
lakers <- readRDS("data/lakers.RDS")
lakers %>%
  glimpse()
Observations: 34,624
Variables: 4
$ date      <date> 2008-10-28, 2008-10-28, 2008-10-28, 2008-10-28, 2008...
$ opponent   <chr> "POR", "POR", "POR", "POR", "POR", "POR", "POR", "POR...
$ team       <fctr> OFF, LAL, LAL, LAL, LAL, LAL, POR, LAL, LAL, POR, LAL...
$ points     <int> 0, 0, 0, 0, 0, 2, 0, 1, 0, 2, 2, 0, 0, 2, 2, 0, 0, 2, ...
```

One thing to note here is that the `glimpse()` command tells us the *type* of each variable. So we have

- `points`: type `int`, short for integer (i.e. whole numbers).
- `date`: type `date`
- `opponent`: type `chr`, short for ‘character’, or alphanumeric data
- `team`: type `fctr`, short for factor and

Differences in *quantity*: numeric variables

We've already seen numeric variables in the section on vectors and lists. These behave pretty much as you'd expect, and we won't expand on them here.

3.0.0.5

There are different types of numeric variable. Integers (whole numbers) are stored as type `int` but other types, like `dbl`, can store numbers with a decimal place. For most purposes (in doing analyses of psychological data) the differences won't matter.

Differences in *quality or kind*

In many cases variables will be used to identify values which are *qualitatively different*. For example, different groups or measurement occasions in an experimental study, or perhaps different genders or countries in survey data.

In practice, these qualitative differences get stored in a range of different variable types, including:

- Numeric variables (e.g. `time = 1`, or `time = 2...`)
- Character variables (e.g. `time = "time 1"`, `time = "time 2"...`)
- Boolean or logical variables (e.g. `time1 == TRUE` or `time1 == FALSE`)
- ‘Factors’

Storing categories as numeric variables can produce confusing results when running regression models.

For this reason, it’s normally best to store your categorical variables as descriptive strings of letters and numbers (e.g. “Treatment”, “Control”) and avoid simple numbers (e.g. 1, 2, 3). Or as a factor.

Factors for categorical data

Factors are R’s answer to the problem of storing categorical data. Factors assign one number for each unique value in a variable, and allow you to attach a label to it.

This means the categories are stored as numbers ‘under the hood’, but you can also work with factors as though they were strings of letters and numbers, and they display nicely when making tables and graphs.

For example:

```
1:10
[1] 1 2 3 4 5 6 7 8 9 10

group.factor <- factor(1:10)
group.factor
[1] 1 2 3 4 5 6 7 8 9 10
Levels: 1 2 3 4 5 6 7 8 9 10

group.labelled <- factor(1:10, labels = paste("Group", 1:10))
group.labelled
[1] Group 1 Group 2 Group 3 Group 4 Group 5 Group 6 Group 7
[8] Group 8 Group 9 Group 10
10 Levels: Group 1 Group 2 Group 3 Group 4 Group 5 Group 6 ... Group 10
```

We can see this ‘underlying’ number which represents each category by using `as.numeric`:

```
# note, there is no guarantee that "Group 1" == 1 (although it is here)
as.numeric(group.labelled)
[1] 1 2 3 4 5 6 7 8 9 10
```

For simple analyses it’s often best to store everything as the `character` type (letters and numbers), but factors can still be useful for making tables or graphs where the list of categories is known and needs to be in a particular order. For more about factors, and lots of useful functions for working with them, see the `forcats::` package: <https://github.com/tidyverse/forcats>

Dates

Internally, R stores dates as the number of days since January 1, 1970. This means that we can work with dates just like other numbers, and it makes sense to have the `min()`, or `max()` of a series of dates:

```
# the first few dates in the sequence
head(lakers$date)
[1] "2008-10-28" "2008-10-28" "2008-10-28" "2008-10-28" "2008-10-28"
[6] "2008-10-28"
```

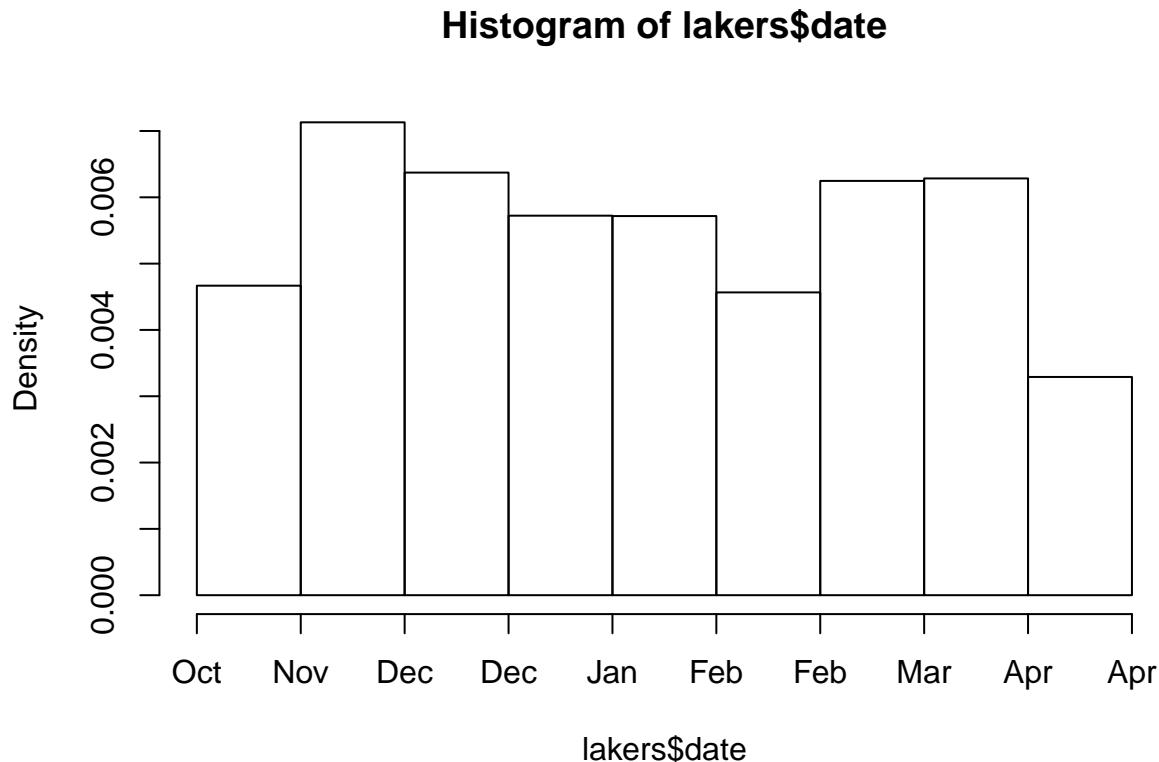
```
# first and last dates
min(lakers$date)
[1] "2008-10-28"
max(lakers$date)
[1] "2009-04-14"
```

Because dates are numbers we can also do arithmetic with them, and R will give us a difference (in this case, in days):

```
max(lakers$date) - min(lakers$date)
Time difference of 168 days
```

However, R does treat dates slightly differently from other numbers, and will format plot axes appropriately, which is helpful (see more on this in the graphics section):

```
hist(lakers$date, breaks=7)
```



Missing values

Missing values aren't a data type as such, but are an important concept in R; the way different functions handle missing values can be both helpful and frustrating in equal measure.

Missing values in a vector are denoted by the letters `NA`, but notice that these letters are unquoted. That is to say `NA` is not the same as `"NA"`!

To check for missing values in a vector (or dataframe column) we use the `is.na()` function:

```

nums.with.missing <- c(1, 2, NA)
nums.with.missing
[1] 1 2 NA

is.na(nums.with.missing)
[1] FALSE FALSE TRUE

```

Here the `is.na()` function has tested whether each item in our vector called `nums.with.missing` is missing. It returns a new vector with the results of each test: either TRUE or FALSE.

We can also use the negation operator, the `!` symbol to reverse the meaning of `is.na`. So we can read `!is.na(nums)` as “test whether the values in `nums` are NOT missing”:

```

# test if missing
is.na(nums.with.missing)
[1] FALSE FALSE TRUE

# test if NOT missing (note the exclamation mark in front of the function)
!is.na(nums.with.missing)
[1] TRUE TRUE FALSE

```

We can use the `is.na()` function as part of dplyr filters:

```

airquality %>%
  filter(is.na(Solar.R)) %>%
  head(3) %>%
  pander

```

Ozone	Solar.R	Wind	Temp	Month	Day
NA	NA	14.3	56	5	5
28	NA	14.9	66	5	6
7	NA	6.9	74	5	11

Or to select only cases without missing values for a particular variable:

```

airquality %>%
  filter(!is.na(Solar.R)) %>%
  head(3) %>%
  pander

```

Ozone	Solar.R	Wind	Temp	Month	Day
41	190	7.4	67	5	1
36	118	8	72	5	2
12	149	12.6	74	5	3

3.0.0.6 Complete cases

Sometimes we want to select only rows which have no missing values — i.e. *complete cases*.

The `complete.cases` function accepts a dataframe (or matrix) and tests whether each *row* is complete. It returns a vector with a TRUE/FALSE result for each row:

```

complete.cases(airquality) %>%
  head

```

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

This can also be useful in dplyr filters. Here we show all the rows which are *not* complete (note the exclamation mark):

```
airquality %>%  
  filter(!complete.cases(airquality))
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	NA	NA	14.3	56	5	5
2	28	NA	14.9	66	5	6
3	NA	194	8.6	69	5	10
4	7	NA	6.9	74	5	11
5	NA	66	16.6	57	5	25
6	NA	266	14.9	58	5	26
7	NA	NA	8.0	57	5	27
8	NA	286	8.6	78	6	1
9	NA	287	9.7	74	6	2
10	NA	242	16.1	67	6	3
11	NA	186	9.2	84	6	4
12	NA	220	8.6	85	6	5
13	NA	264	14.3	79	6	6
14	NA	273	6.9	87	6	8
15	NA	259	10.9	93	6	11
16	NA	250	9.2	92	6	12
17	NA	332	13.8	80	6	14
18	NA	322	11.5	79	6	15
19	NA	150	6.3	77	6	21
20	NA	59	1.7	76	6	22
21	NA	91	4.6	76	6	23
22	NA	250	6.3	76	6	24
23	NA	135	8.0	75	6	25
24	NA	127	8.0	78	6	26
25	NA	47	10.3	73	6	27
26	NA	98	11.5	80	6	28
27	NA	31	14.9	77	6	29
28	NA	138	8.0	83	6	30
29	NA	101	10.9	84	7	4
30	NA	139	8.6	82	7	11
31	NA	291	14.9	91	7	14
32	NA	258	9.7	81	7	22
33	NA	295	11.5	82	7	23
34	78	NA	6.9	86	8	4
35	35	NA	7.4	85	8	5
36	66	NA	4.6	87	8	6
37	NA	222	8.6	92	8	10
38	NA	137	11.5	86	8	11
39	NA	64	11.5	79	8	15
40	NA	255	12.6	75	8	23
41	NA	153	5.7	88	8	27
42	NA	145	13.2	77	9	27

3.0.0.7

Sometimes it's convenient to use the . (period) to represent the output from the previous pipe command.

For example, we could rewrite the previous example as:

```
airquality %>%
  filter(!complete.cases(.)) # note the . (period) here in place of `airmiles`
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	NA	NA	14.3	56	5	5
2	28	NA	14.9	66	5	6
3	NA	194	8.6	69	5	10
4	7	NA	6.9	74	5	11
5	NA	66	16.6	57	5	25
6	NA	266	14.9	58	5	26
7	NA	NA	8.0	57	5	27
8	NA	286	8.6	78	6	1
9	NA	287	9.7	74	6	2
10	NA	242	16.1	67	6	3
11	NA	186	9.2	84	6	4
12	NA	220	8.6	85	6	5
13	NA	264	14.3	79	6	6
14	NA	273	6.9	87	6	8
15	NA	259	10.9	93	6	11
16	NA	250	9.2	92	6	12
17	NA	332	13.8	80	6	14
18	NA	322	11.5	79	6	15
19	NA	150	6.3	77	6	21
20	NA	59	1.7	76	6	22
21	NA	91	4.6	76	6	23
22	NA	250	6.3	76	6	24
23	NA	135	8.0	75	6	25
24	NA	127	8.0	78	6	26
25	NA	47	10.3	73	6	27
26	NA	98	11.5	80	6	28
27	NA	31	14.9	77	6	29
28	NA	138	8.0	83	6	30
29	NA	101	10.9	84	7	4
30	NA	139	8.6	82	7	11
31	NA	291	14.9	91	7	14
32	NA	258	9.7	81	7	22
33	NA	295	11.5	82	7	23
34	78	NA	6.9	86	8	4
35	35	NA	7.4	85	8	5
36	66	NA	4.6	87	8	6
37	NA	222	8.6	92	8	10
38	NA	137	11.5	86	8	11
39	NA	64	11.5	79	8	15
40	NA	255	12.6	75	8	23
41	NA	153	5.7	88	8	27
42	NA	145	13.2	77	9	27

This is nice because we can apply the `complete.cases` function to the output of the previous pipe. For example, if we wanted to select complete cases for a subset of the variables we could write:

```
airquality %>%
  select(Ozone, Solar.R) %>%
  filter(!complete.cases(.))
```

```

1   NA    NA
2   28    NA
3   NA    194
4   7     NA
5   NA    66
6   NA    266
7   NA    NA
8   NA    286
9   NA    287
10  NA    242
11  NA    186
12  NA    220
13  NA    264
14  NA    273
15  NA    259
16  NA    250
17  NA    332
18  NA    322
19  NA    150
20  NA    59
21  NA    91
22  NA    250
23  NA    135
24  NA    127
25  NA    47
26  NA    98
27  NA    31
28  NA    138
29  NA    101
30  NA    139
31  NA    291
32  NA    258
33  NA    295
34  78    NA
35  35    NA
36  66    NA
37  NA    222
38  NA    137
39  NA    64
40  NA    255
41  NA    153
42  NA    145

```

Or alternatively:

```

rows.to.keep <- !complete.cases(select(airquality, Ozone, Solar.R))
airquality %>%
  filter(rows.to.keep) %>%
  head(3) %>%
  pander

```

Ozone	Solar.R	Wind	Temp	Month	Day
NA	NA	14.3	56	5	5
28	NA	14.9	66	5	6

Ozone	Solar.R	Wind	Temp	Month	Day
NA	194	8.6	69	5	10

3.0.0.8 Missing data and R functions

It's normally good practice to pre-process your data and select the rows you want to analyse *before* passing dataframes to R functions.

The reason for this is that different functions behave differently with missing data.

For example:

```
mean(airquality$Solar.R)
[1] NA
```

Here the default for `mean()` is to return NA if any of the values are missing. We can explicitly tell R to ignore missing values by setting `na.rm=TRUE`

```
mean(airquality$Solar.R, na.rm=TRUE)
[1] 185.9315
```

In contrast some other functions, for example the `lm()` which runs a linear regression will ignore missing values by default. If we run `summary` on the call to `lm` then we can see the line near the bottom of the output which reads: “(7 observations deleted due to missingness)”

```
lm(Solar.R ~ Temp, data=airquality) %>%
  summary

Call:
lm(formula = Solar.R ~ Temp, data = airquality)

Residuals:
    Min      1Q      Median      3Q      Max 
-169.697 -59.315   6.224   67.685  186.083 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -24.431     61.508  -0.397 0.691809    
Temp         2.693      0.782    3.444 0.000752 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 

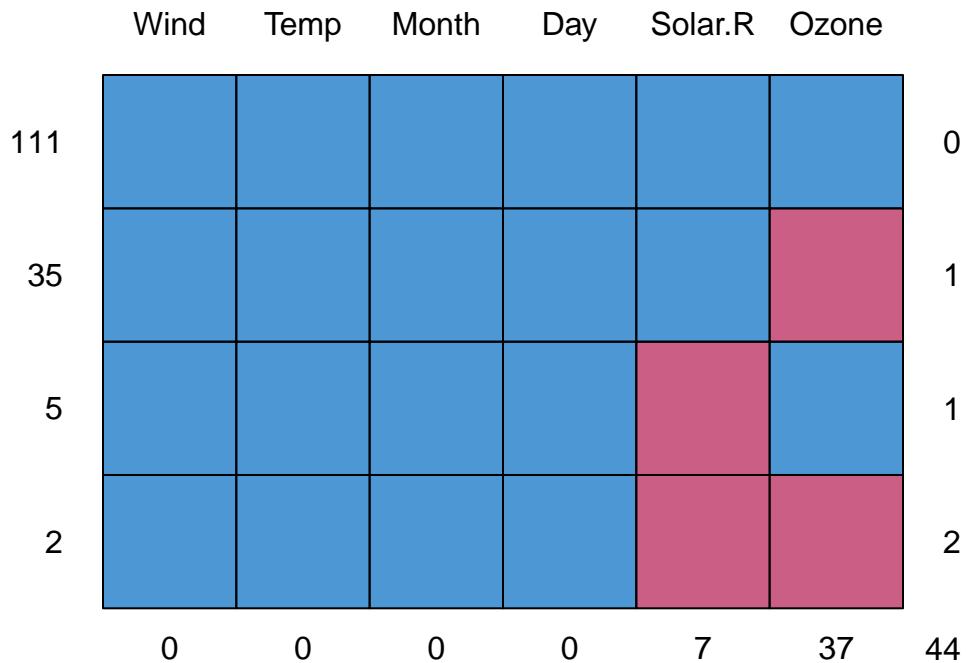
Residual standard error: 86.86 on 144 degrees of freedom
(7 observations deleted due to missingness)
Multiple R-squared:  0.07609, Adjusted R-squared:  0.06967 
F-statistic: 11.86 on 1 and 144 DF,  p-value: 0.0007518
```

Normally R will do the ‘sensible thing’ when there are missing values, but it’s always worth checking whether you do have any missing data, and addressing this explicitly in your code

3.0.0.9 Patterns of missingness

The `mice` package has some nice functions to describe patterns of missingness in the data. These can be useful both at the exploratory stage, when you are checking and validating your data, but can also be used to create tables of missingness for publication:

```
mice::md.pattern(airquality)
```



	Wind	Temp	Month	Day	Solar.R	Ozone
111	1	1	1	1	1	1
35	1	1	1	1	1	0
5	1	1	1	1	0	1
2	1	1	1	1	0	0
	0	0	0	0	7	37
						44

In this table, `md.pattern` lists the number of cases with particular patterns of missing data. - Each row describes a missing data ‘pattern’ - The first column indicates the number of cases - The central columns indicate whether a particular variable is missing for the pattern (0=missing) - The last column counts the number of values missing for the pattern - The final row counts the number of missing values for each variable.

3.0.0.9.1 Visualising missingness

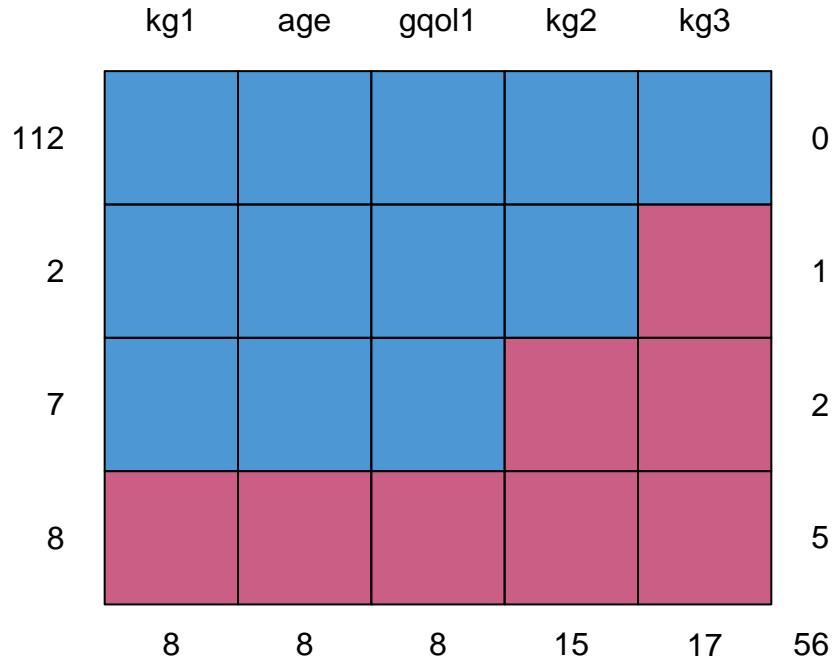
Graphics can also be useful to explore patterns in missingness.

`rct.data` contains data from an RCT of functional imagery training (FIT) for weight loss, which measured outcome (weight in kg) at baseline and two followups (`kg1`, `kg2`, `kg3`). The trial also measured global quality of life (`gqol1`).

As is common, there were some missing data at the followup:

```
fit.data <- readRDS("data/fit-weight.RDS") %>%
  select(kg1, kg2, kg3, age, gqol1)
```

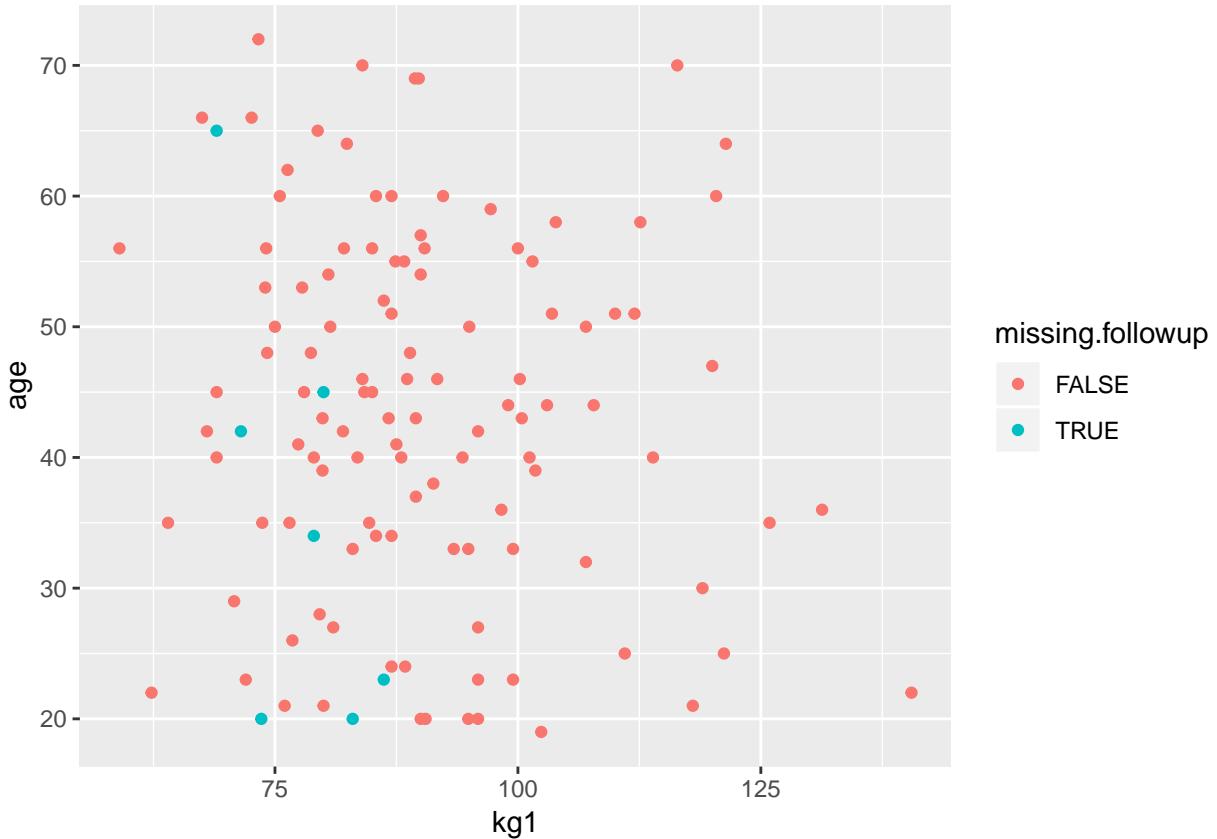
```
mice::md.pattern(fit.data)
```



```
kg1 age gqol1 kg2 kg3
112 1 1 1 1 1 0
2 1 1 1 1 0 1
7 1 1 1 0 0 2
8 0 0 0 0 0 5
8 8 8 15 17 56
```

We might be interested to explore patterns in which observations were missing. Here we use colour to identify missing observations as a function of the data recorded at baseline:

```
fit.data %>%
  mutate(missing.followup = is.na(kg2)) %>%
  ggplot(aes(kg1, age, color=missing.followup)) +
  geom_point()
```



There's a clear trend here for lighter patients (at baseline) to have more missing data at followup. There's also a suggestion that younger patients are more likely to have been lost to followup.

If needed, we could perform inferential tests for these differences:

```
t.test(kg1 ~ is.na(kg2), data=fit.data)

Welch Two Sample t-test

data: kg1 by is.na(kg2)
t = 4.7153, df = 11.132, p-value = 0.000614
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 7.005116 19.236238
sample estimates:
mean in group FALSE mean in group TRUE
 90.59211          77.47143

t.test(age ~ is.na(kg2), data=fit.data)

Welch Two Sample t-test

data: age by is.na(kg2)
t = 1.2418, df = 6.5246, p-value = 0.2571
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -7.39455 23.25169
sample estimates:
```

```
mean in group FALSE mean in group TRUE
43.50000          35.57143
```

However, given the small number of missing values and the post-hoc nature of these analyses these tests are rather underpowered and we might prefer to report and comment on the plot alone.

For some nice missing data visualisation techniques, including those for repeated measures data, see Zhang [2015].

Tidying data

‘Tidying’ data means converting it into the format that is most useful for data analyses, and so we have already covered many of the key techniques: selecting and filtering data, reshaping and summarising.

However the ideas behind ‘tidying’ draw together other related concepts which link together the way we enter, store and process data: for example the idea of ‘relational data’ and techniques to join together related datasets.

3.0.0.10 A philosophy of tidy data

The chapter on tidying in ‘R for data science’ is well worth reading for its thoughtful explanation of why we want tidy data, and the core techniques to clean up untidy data: <http://r4ds.had.co.nz/tidy-data.html>

Reshaping

This section will probably require more attention than any other in the guide, but will likely be one of the most useful things you learn in R.

As previously discussed, most things work best in R if you have data in *long format*. This means we prefer data that look like this:

person	time	outcome
1	Time 1	21.78
2	Time 1	20.06
3	Time 1	20.43
1	Time 2	18.7
2	Time 2	23.95
3	Time 2	19.86
1	Time 3	20.06
2	Time 3	17.42
3	Time 3	22.02
1	Time 4	19.2
2	Time 4	17.01
3	Time 4	19.82

And NOT like this:

person	Time 1	Time 2	Time 3	Time 4
1	21.78	18.7	20.06	19.2
2	20.06	23.95	17.42	17.01
3	20.43	19.86	22.02	19.82

In long format data:

- each row of the dataframe corresponds to a single measurement occasion
- each column corresponds to a variable which is measured

Fortunately it's fairly easy to move between the two formats, provided your variables are named in a consistent way.

3.0.0.11 Wide to long format

This is the most common requirement. Often you will have several columns which actually measure the same thing, and you will need to convert these two columns - a 'key', and a value.

For example, let's say we measure patients on 10 days:

```
sleep.wide %>%
  head(4) %>%
  pander(caption="Data for the first 4 subjects")
```

Table 16: Data for the first 4 subjects

Subject	Day.0	Day.1	Day.2	Day.3	Day.4	Day.5	Day.6	Day.7	Day.8	Day.9
1	249.6	258.7	250.8	321.4	356.9	414.7	382.2	290.1	430.6	466.4
2	222.7	205.3	203	204.7	207.7	216	213.6	217.7	224.3	237.3
3	199.1	194.3	234.3	232.8	229.3	220.5	235.4	255.8	261	247.5
4	321.5	300.4	283.9	285.1	285.8	297.6	280.2	318.3	305.3	354

We want to convert RT measurements on each Day to a single variable, and create a new variable to keep track of what Day the measurement was taken:

The `melt()` function in the `reshape2::` package does this for us:

```
library(reshape2)
sleep.long <- sleep.wide %>%
  melt(id.var="Subject") %>%
  arrange(Subject, variable)

sleep.long %>%
  head(12) %>%
  pander
```

Subject	variable	value
1	Day.0	249.6
1	Day.1	258.7
1	Day.2	250.8
1	Day.3	321.4
1	Day.4	356.9
1	Day.5	414.7
1	Day.6	382.2
1	Day.7	290.1
1	Day.8	430.6
1	Day.9	466.4
2	Day.0	222.7
2	Day.1	205.3

Here melt has created two new variable: `variable`, which keeps track of what was measured, and `value` which contains the score. This is the format we need when plotting graphs and running regression and Anova models.

3.0.0.12 Long to wide format

To continue the example from above, these are long form data we just made:

```
sleep.long %>%
  head(3) %>%
  pander(caption="First 3 rows in the long format dataset")
```

Table 18: First 3 rows in the long format dataset

Subject	variable	value
1	Day.0	249.6
1	Day.1	258.7
1	Day.2	250.8

We can convert these back to the original wide format using `dcast`, again in the `reshape2` package. The name of the `dcast` function indicates we can ‘cast’ a dataframe (the d prefix). So here, casting means the opposite of ‘melting’.

Using `dcast` is a little more fiddly than `melt` because we have to say *how* we want the data spread wide. In this example we could either have:

- Columns for each day, with rows for each subject
- Columns for each subject, with rows for each day

Although it’s obvious to *us* which format we want, we have to be explicit for R to get it right.

We do this using a formula, which we’ll see again in the regression section.

Each formula has two sides, left and right, separated by the tilde (~) symbol. On the left hand side we say which variable we want to keep in rows. On the right hand side we say which variables to convert to columns. So, for example:

```
# rows per subject, columns per day
sleep.long %>%
  dcast(Subject~variable) %>%
  head(3)
  Subject   Day.0    Day.1    Day.2    Day.3    Day.4    Day.5    Day.6
  1       1 249.5600 258.7047 250.8006 321.4398 356.8519 414.6901 382.2038
  2       2 222.7339 205.2658 202.9778 204.7070 207.7161 215.9618 213.6303
  3       3 199.0539 194.3322 234.3200 232.8416 229.3074 220.4579 235.4208
  Day.7    Day.8    Day.9
  1 290.1486 430.5853 466.3535
  2 217.7272 224.2957 237.3142
  3 255.7511 261.0125 247.5153
```

To compare, we can convert so each Subject has a column by reversing the formula:

```
# note we select only the first 7 Subjects to
# keep the table to a manageable size
sleep.long %>%
  filter(Subject < 8) %>%
  dcast(variable~Subject)
```

```

variable    1      2      3      4      5      6      7
1 Day.0 249.5600 222.7339 199.0539 321.5426 287.6079 234.8606 283.8424
2 Day.1 258.7047 205.2658 194.3322 300.4002 285.0000 242.8118 289.5550
3 Day.2 250.8006 202.9778 234.3200 283.8565 301.8206 272.9613 276.7693
4 Day.3 321.4398 204.7070 232.8416 285.1330 320.1153 309.7688 299.8097
5 Day.4 356.8519 207.7161 229.3074 285.7973 316.2773 317.4629 297.1710
6 Day.5 414.6901 215.9618 220.4579 297.5855 293.3187 309.9976 338.1665
7 Day.6 382.2038 213.6303 235.4208 280.2396 290.0750 454.1619 332.0265
8 Day.7 290.1486 217.7272 255.7511 318.2613 334.8177 346.8311 348.8399
9 Day.8 430.5853 224.2957 261.0125 305.3495 293.7469 330.3003 333.3600
10 Day.9 466.3535 237.3142 247.5153 354.0487 371.5811 253.8644 362.0428

```

One neat trick when casting is to use `paste` to give your columns nicer names. So for example:

```

sleep.long %>%
  filter(Subject < 4) %>%
  dcast(variable~paste0("Person.", Subject))
  variable Person.1 Person.2 Person.3
1 Day.0 249.5600 222.7339 199.0539
2 Day.1 258.7047 205.2658 194.3322
3 Day.2 250.8006 202.9778 234.3200
4 Day.3 321.4398 204.7070 232.8416
5 Day.4 356.8519 207.7161 229.3074
6 Day.5 414.6901 215.9618 220.4579
7 Day.6 382.2038 213.6303 235.4208
8 Day.7 290.1486 217.7272 255.7511
9 Day.8 430.5853 224.2957 261.0125
10 Day.9 466.3535 237.3142 247.5153

```

Notice we used `paste0` rather than `paste` to avoid spaces in variable names, which is allowed but can be a pain. See more on working with character strings in a later section.

3.0.0.12.1

For a more detailed explanation and various other methods for reshaping data, see: <http://r4ds.had.co.nz/tidy-data.html>

Which package should you use to reshape data?

There are three main options:

- `tidyverse`, which comes as part of the `tidyverse`, using `gather` and `spread()`
- `reshape2`: using `melt()` and `dcast()`
- `data.table`: also using functions called `melt()` and `dcast()` (but which are slightly different from those in `reshape2`)

This post walks through some of the differences: <https://www.r-bloggers.com/how-to-reshape-data-in-r-tidyr-vs-reshape2/> but the short answer is whichever you find simplest and easiest to remember (for me that's `melt` and `dcast`).

‘

Aggregating and reshaping at the same time

One common trick when reshaping is to convert a datafile which has multiple rows and columns per person to one with only a single row per person. That is, we aggregate by using a summary (perhaps the mean) and

reshape at the same time.

Although useful this isn't covered in this section, because it is combining two techniques:

- Reshaping (i.e. from long to wide or back)
- Aggregating or summarising (converting multiple rows to one)

In the next section we cover summarising data, and introduce the 'split-apply-combine' method for summarising.

Once you have a good grasp of this, you could check out the 'fancy reshaping' section which does provide examples of aggregating and reshaping simultaneously.

– title: 'Summarising data'

4 Summaries

Before you begin this section, make sure you have fully understood the section on datasets and dataframes, and in particular that you are happy using the `%>%` symbol to describe a flow of data.

Although R contains many functions like `table`, or `xtabs`, `describe` or `summarise` which you might see used elsewhere to tabulate or summarise data, for beginners these base-R functions can be confusing; their names and input names are not always consistent, and they don't always work together nicely.

Instead we recommend using `dplyr` and other parts of the tidyverse because they provide a general set of tools to make any kind of table or summary.

They also encourage more coherent thinking about *what summary is really needed*, rather than accepting or fighting with default options.

A generalised approach

4.0.0.1 The 'split, apply, combine' model

The `dplyr::` package, and especially the `summarise()` function provides a generalised way to create dataframes of frequencies and other summary statistics, grouped and sorted however we like.

Each of the dplyr 'verbs' acts on a dataframe in some way, and returns a dataframe as its result. This is convenient because we can chain together the different verbs to describe exactly the table we want.

For example, let's say we want the mean of some of our variables across the whole dataframe:

```
angry.moods %>%
  summarise(
    mean.anger.out=mean(Anger.Out),
    sd.anger.out=sd(Anger.Out)
  )
# A tibble: 1 x 2
  mean.anger.out sd.anger.out
            <dbl>        <dbl>
1           16.1        4.22
```

Here the `summarise` function accepts the `angry.moods` dataframe as an input, and has returned a dataframe containing the statistics we need. In this instance the result dataframe only has one row.

What if we want the numbers for men and women separately?

The key is to think about what we want to achieve, and work out how to describe it. However, in general, we will often want to follow this pattern:

- *Split* our data (into men and women, or some other categorisation)
- *Apply* some operation (function) to each group individually (e.g. calculate the mean)
- *Combine* it into a single table again

It's helpful to think of this *split* → *apply* → *combine* pattern whenever we are processing data because it *makes explicit what we want to do*.

4.0.0.2 Split: breaking the data into groups

The first task is to organise our dataframe into the relevant groups. To do this we use `group_by()`:

```
angry.moods %>%
  group_by(Gender) %>%
  glimpse
Observations: 78
Variables: 7
Groups: Gender [2]
$ Gender      <dbl> 2, 2, 2, 2, 1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 2, ...
$ Sports       <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, ...
$ Anger.Out    <dbl> 18, 14, 13, 17, 16, 16, 12, 13, 16, 12, 12, 1...
$ Anger.In     <dbl> 13, 17, 14, 24, 17, 22, 12, 16, 16, 16, 13, 2...
$ Control.Out  <dbl> 23, 25, 28, 23, 26, 25, 31, 22, 22, 29, 24, 2...
$ Control.In   <dbl> 20, 24, 28, 23, 28, 23, 27, 31, 24, 29, 25, 2...
$ Anger.Expression <dbl> 36, 30, 19, 43, 27, 38, 14, 24, 34, 18, 24, 4...
```

Weirdly, this doesn't seem to have done anything. The data aren't sorted by `Gender`, and there is no visible sign of the grouping, but stick with it... the grouping is there and the effect will be clearer in a moment.

4.0.0.3 Apply and combine

Continuing the example above, once we have grouped our data we can then *apply* a function to it — for example, we can summarise each group by taking the mean of the `Anger.Out` variable:

```
angry.moods %>%
  group_by(Gender) %>%
  summarise(
    mean.anger.out=mean(Anger.Out)
  )
# A tibble: 2 x 2
  Gender mean.anger.out
  <dbl>          <dbl>
1     1            16.6
2     2            15.8
```

The **combine** step happens automatically for us: `dplyr` has combined the summaries of each gender into a single dataframe for us.

In summary, we:

- *split* the data by `Gender`, using `group_by()`
- *apply* the `summarise()` function
- *combine* the results into a new data frame (happens automatically)

4.0.0.4 A ‘real’ example

Imagine we have raw data from a study which had measured depression with the PHQ-9 scale.

Each **patient** was measured on numerous occasions (the **month** of observation is recorded), and were split into treatment **group** (0=control, 1=treatment). The **phq9** variable is calculated as the sum of all their questionnaire responses.

```
phq9.df %>% glimpse
Observations: 2,429
Variables: 4
$ patient <dbl> 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, ...
$ group    <dbl> 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, ...
$ month    <dbl> 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 18, 0, 1, ...
$ phq9     <dbl> 2.56, 2.33, 1.89, 2.00, 2.44, 2.33, 2.44, 2.11, 2.22, ...
```

If this were our data we might want to:

- Calculate the average PHQ-9 score at each month, and in each group
- Show these means by group for months 0, 7 and 12

We can do this using **group_by** and **summarise**:

```
phq9.df %>%
  group_by(group) %>%
  summarise(average_phq9 = mean(phq9))
# A tibble: 2 x 2
  group  average_phq9
  <dbl>      <dbl>
1     0        1.87
2     1        1.60
```

You can load the PHQ9 data above by typing:

```
# remeber to load the tidyverse package first
phq9 <- read_csv('data/phq-summary.csv')
Parsed with column specification:
cols(
  patient = col_double(),
  group = col_double(),
  month = col_double(),
  phq9 = col_double()
)
```

Try to edit the code above to:

- Create summary table with the mean at each month
- The mean at each month, in each group
- The mean and SD by month and group

Fancy reshaping

As noted above, it’s common to combine the process of reshaping and aggregating or summarising in the same step.

For example here we have multiple rows per person, 3 trial at time 1, and 3 more trials at time 2:

```
expt.data %>%
  arrange(person, time, trial) %>%
```

```
head %>%
pander
```

Condition	trial	time	person	RT
1	1	1	1	219.8
1	2	1	1	194.4
1	3	1	1	394.1
1	1	2	1	272
1	2	2	1	180.1
1	3	2	1	277

We can reshape and aggregate this in a single step using `dcast`. Here we request the mean for each person at each time, with observations for each time split across two columns:

```
library(reshape2)
expt.data %>%
  dcast(person~paste0('time',time),
    fun.aggregate=mean) %>%
head %>%
pander
Using RT as value column: use value.var to override.
```

person	time1	time2
1	269.4	243
2	259.2	219.8
3	211.3	323.1
4	255.5	249.2
5	264.1	307
6	274.5	291

Here `dcast` has correctly guessed that `RT` is the value we want to aggregate (you can specify explicitly with the `value.var` parameter).

`dcast` knows to aggregate using the mean because we set this with the `agg.function` parameter; this just stands for ‘aggregation function’.

We don’t have to request the mean though: any function will do. Here we request the SD instead:

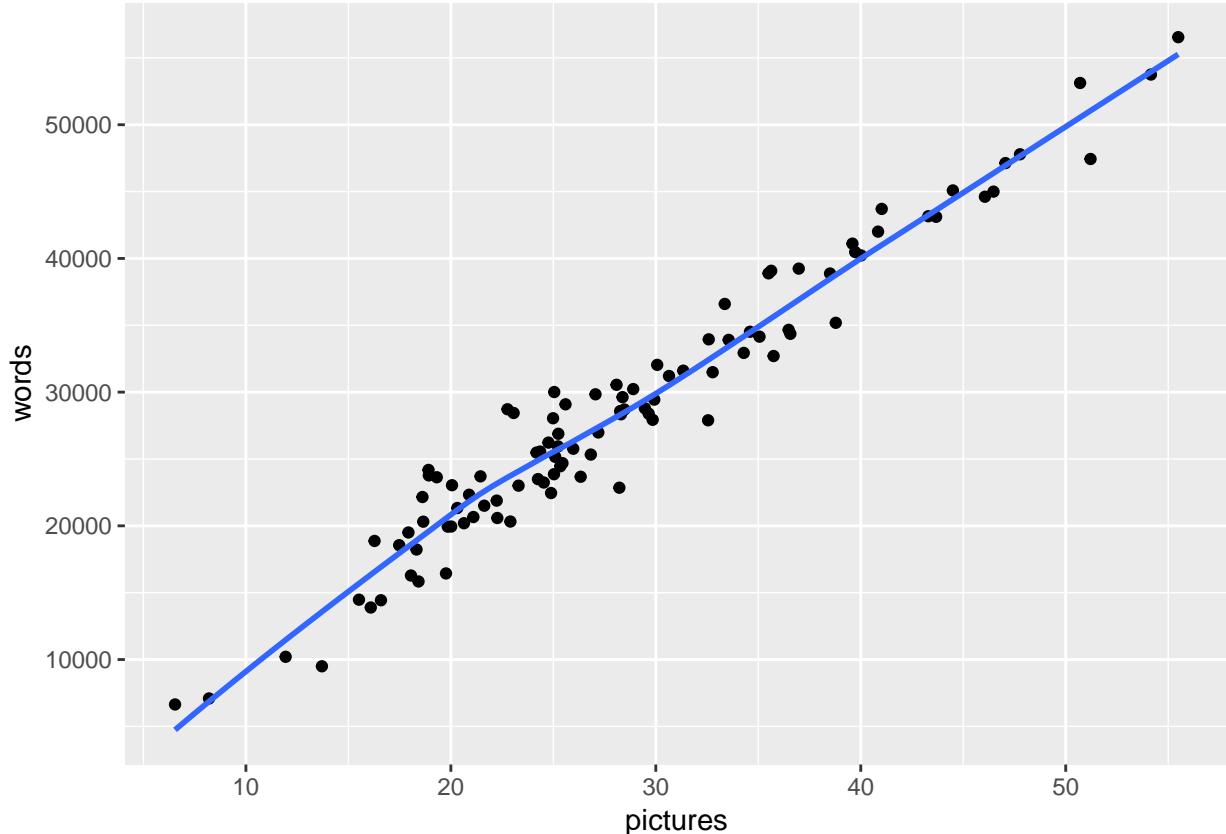
```
expt.data %>%
  dcast(person~time,
    fun.aggregate=sd) %>%
head %>%
pander
Using RT as value column: use value.var to override.
```

person	1	2
1	108.7	54.56
2	53.07	79.45
3	38.84	28.52
4	43.3	52.74
5	45.72	45.02

person	1	2
6	38.74	94.47

5 Graphics

Warning: `data_frame()` is deprecated, use `tibble()`.
 This warning is displayed once per session.



Plotting and data visualisation is probably the best thing about R.

The base system alone provides lots of useful plotting functions, but the `ggplot2` package is exceptional in the consistent and powerful approach it takes to visualising data. This chapter focusses mostly on `ggplot`, but does include some pointers to other useful plotting functions.

It's also worth pointing out here that the O'Reilly R Graphics cookbook is available as a pdf download and is a much more comprehensive source than this book.

The examples below are more selective and show plots likely to be of particular use in reporting your studies.

The emphasis on showing you how to make *good* plots that help you explore data and communicate your findings, rather than simply reproduce the output of SPSS or Excel.

Benefits of visualising data

Scientists attempt to understand natural processes, predict events in the observable world, and communicate this understanding to others. In each case, graphics are a powerful tool in their armoury.

David McCandless recently spoke at a TedX event about his work on data journalism, and makes a persuasive case for paying more attention to visualisations:

Which tool to use?

When plotting data it pays to decide if you need:

1. A quick way to visualise something specific – for example to check some feature of your data before you continue your analysis.
2. A plot that is specifically designed to communicate your data effectively, and where you care about the details of the final output.

For the first case — for example to visualise a distribution of a single variable, or to check diagnostics from a linear model — there are lots of useful built-in functions in base-R and other packages.

For the second case — for example where you want to visualise the main outcomes of your study, or draw attention to specific aspects of your data — there is `ggplot2`.

We case 2 first, because using `ggplot` highlights many important aspects of plotting in general.

Layered graphics with `ggplot`

If you've never given much thought to data visualisation before, you might be surprised at the sheer variety of graphs types available.

One way to cut through the multitude of options is to determine what the purpose of your plot is. Although not a complete list, it's likely your plot will show at least one of:

- Relationships
- Distributions
- Comparison
- Composition

The `ggplot` library makes it easy to produce high quality graphics which serve these ends, and to layer them to produce information-dense plots which are really effective forms of communication.

A thought on ‘chart chooser’ guides

There are various simple chart selection guides available online, of which these are quite nice examples:

- Chart selection guide (pdf)
- ‘Show me the numbers’ chart guide (pdf)

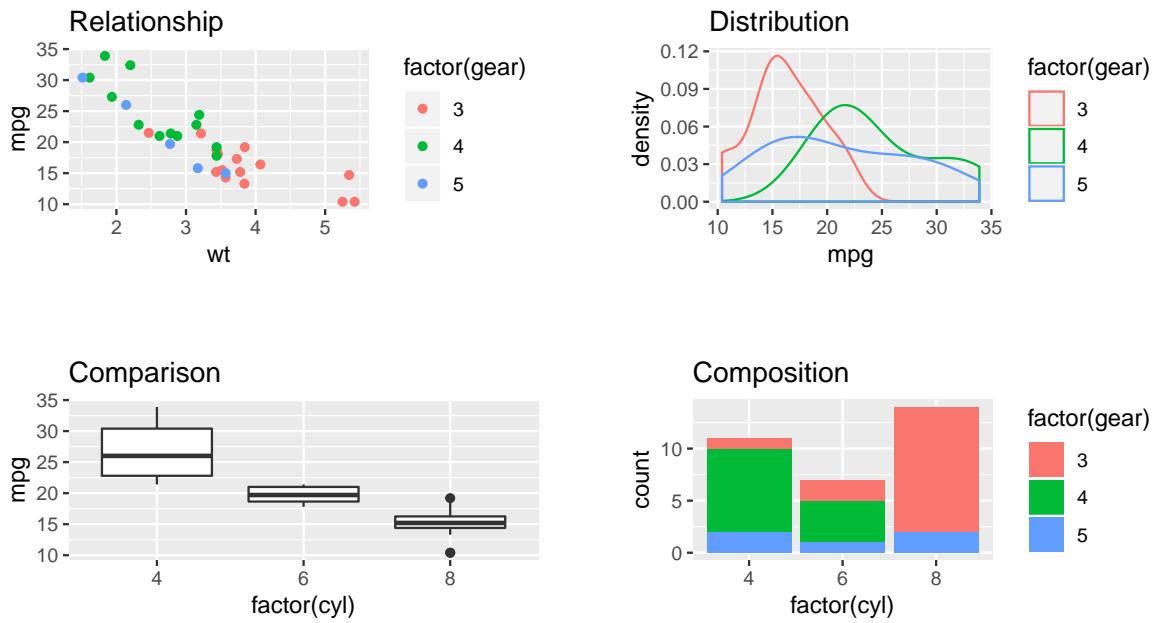
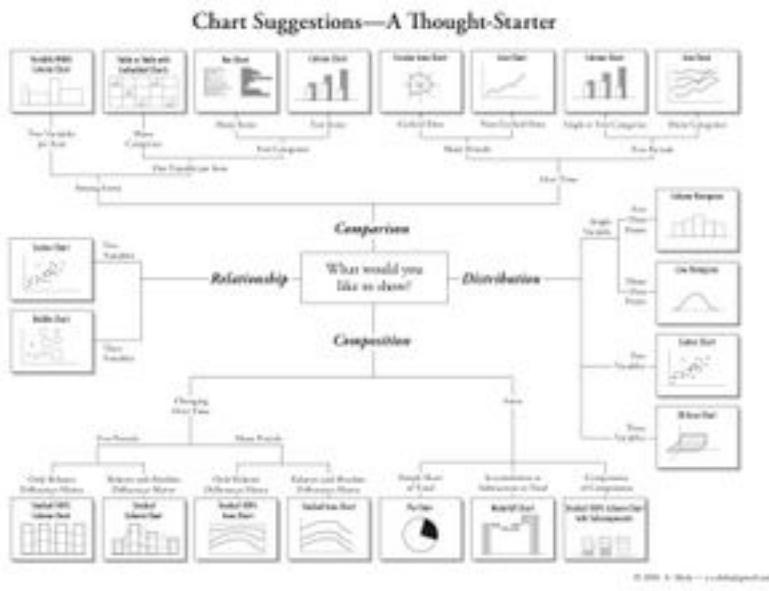


Figure 5: Examples of charts showing comparisons, relationships and distribution and composition. The comparison, distribution and composition plots show 2 variables, but the relationship plot includes 3, increasing the density of the information displayed.



However, guides which attempt to be comprehensive and show you a full range of plot types are perhaps not as useful as those which reflect our knowledge of which plots are the most effective forms of communication.

For example, almost all guides to plotting, and especially R textbooks, will show you how to plot a simple bar graph. But bar graphs have numerous disadvantages over other plots which can show the same information.

Specifically, they:

- are low in information density (and so inefficient in use of space)
- make comparisons between multiple data series very difficult (for example in interaction plots), and
- perhaps most importantly, even when they include error bars, readers consistently misinterpret the quantitative information in bar graphs (specifically, when bar graphs are used to display estimates which contain error, readers assume points above the bar are less likely than points within the bar, even though this is typically not the case).

You should be guided in choosing plots not by mechanical rules based on the number or type of variables you want to display. Instead, you should be guided by the evidence from basic studies of human perception, and applied data on how different types of information displays are really used by readers.

This guide is restricted to examples likely to be useful and effective for experimental and applied psychologists.

Thinking like ggplot

When using `ggplot` it helps to think of five separate steps to making a plot (2 are optional, but commonly used):

1. Choose the data you want to plot.
2. Map variables to axes or other features of the plot (e.g. sizes or colours).
3. (Optionally) use `ggplot` functions to summarise your data before the plot is drawn (e.g. to calculate means and standard errors for point-range plots).
4. Add visual display layers.
5. (Optionally) Split the plot up across multiple panels using groupings in the data.

You can then customise the plot labels and title, and tweak other presentation parameters, although this often isn't necessary unless sending a graphic for publication. You can also export graphics in multiple high quality formats.

The simplest way to demonstrate these steps is with an example, and we begin with a plot showing the relationship between variables:

‘Relationships’

Problem to be solved: *You want to check/show whether variables are related in a linear fashion, e.g. before running linear regression*

5.0.0.0.1 Step 1: Select data to plot

Step 1 is to select our data. As is typical in R, `ggplot` works best with long-format data. In the examples below we will use the `mtcars` dataset for convenience, so our first line of code is to use the `dplyr` pipe symbol (operator) to send the `mtcars` dataset to the next line of code:

```
mtcars %>%  
  ...
```

5.0.0.0.2 Step 2: Map variables to axes, colours, and other features

Step 2 is to map the variables we want to axes or other features of the plot (e.g. the colours of points, or the linetypes used in line plots).

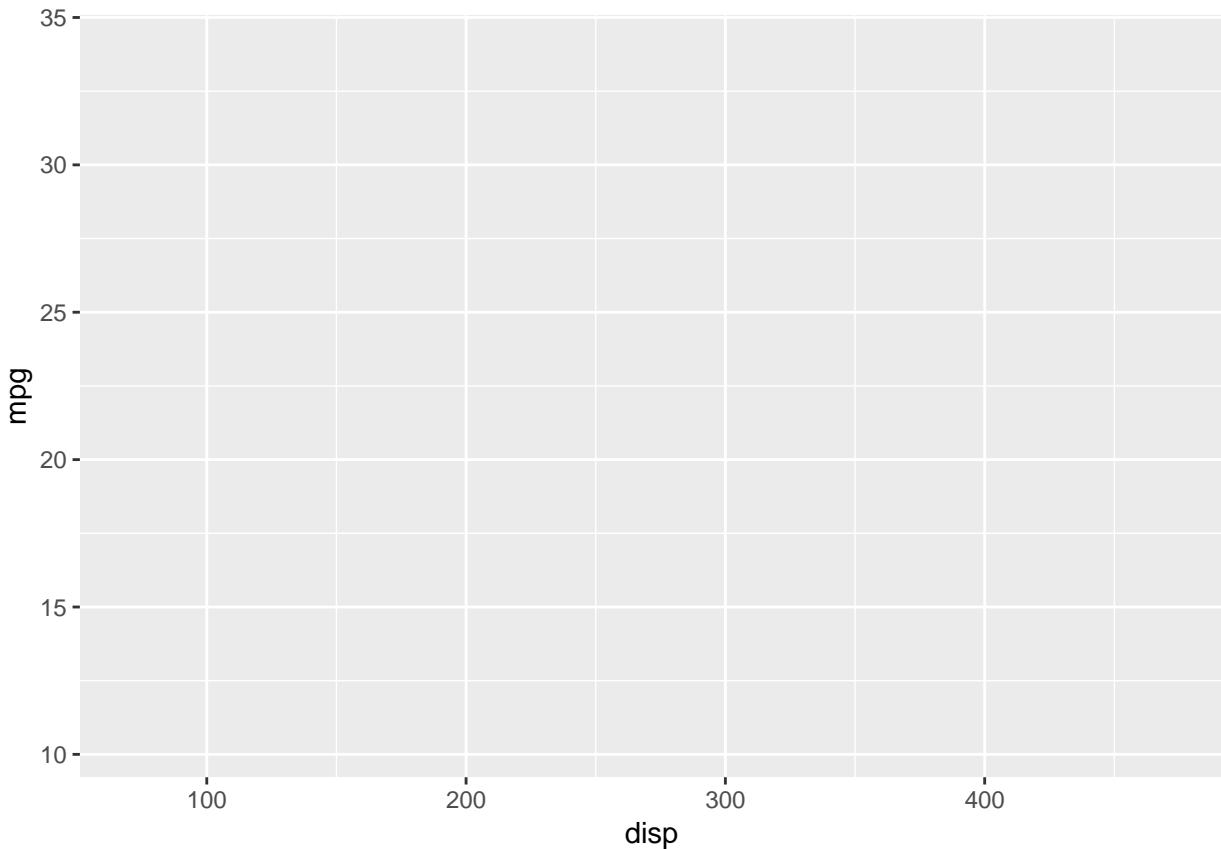
To specify these mappings we use the `aes()` function, which is slightly cryptic, but short for ‘aesthetics mapping’. Depending on the plot type you will specify different aesthetics, and they can also have different effects depending on the plot type, but you will commonly specify:

- `x` the variable to use as the x axis
- `y` the variable to use as the y axis
- `colour`: the variable to use to colour points or lines

Here we tell `ggplot` to use `disp` (engine size) on the x axis, and `mpg` on the y axis. We also tell it to colour the points differently depending on the value of `hp` (engine horsepower).

At this point `ggplot` will create and label the axes and plot area, but doesn’t yet display any of our data. For this we need to add visual display layers (in the next step).

```
mtcars %>%  
  ggplot(aes(x = disp, y = mpg, colour=hp))
```



Other aesthetics

There are many other aesthetics which can be specified. Some of the most useful are:

- `ymin` and `ymax`: for upper and lower bounds, e.g. on error bars
- `group`: which tells `ggplot` to group observations by some variable and, for example, plot a different line per-group)
- `fill`: like `colour` but for areas/shapes

- **alpha** and **size**: control the size and opacity of visual features (useful for de-emphasising some features of a plot to make others stand out)

See the ggplot documentation for more details: <http://ggplot2.tidyverse.org/reference/#section-aesthetics>

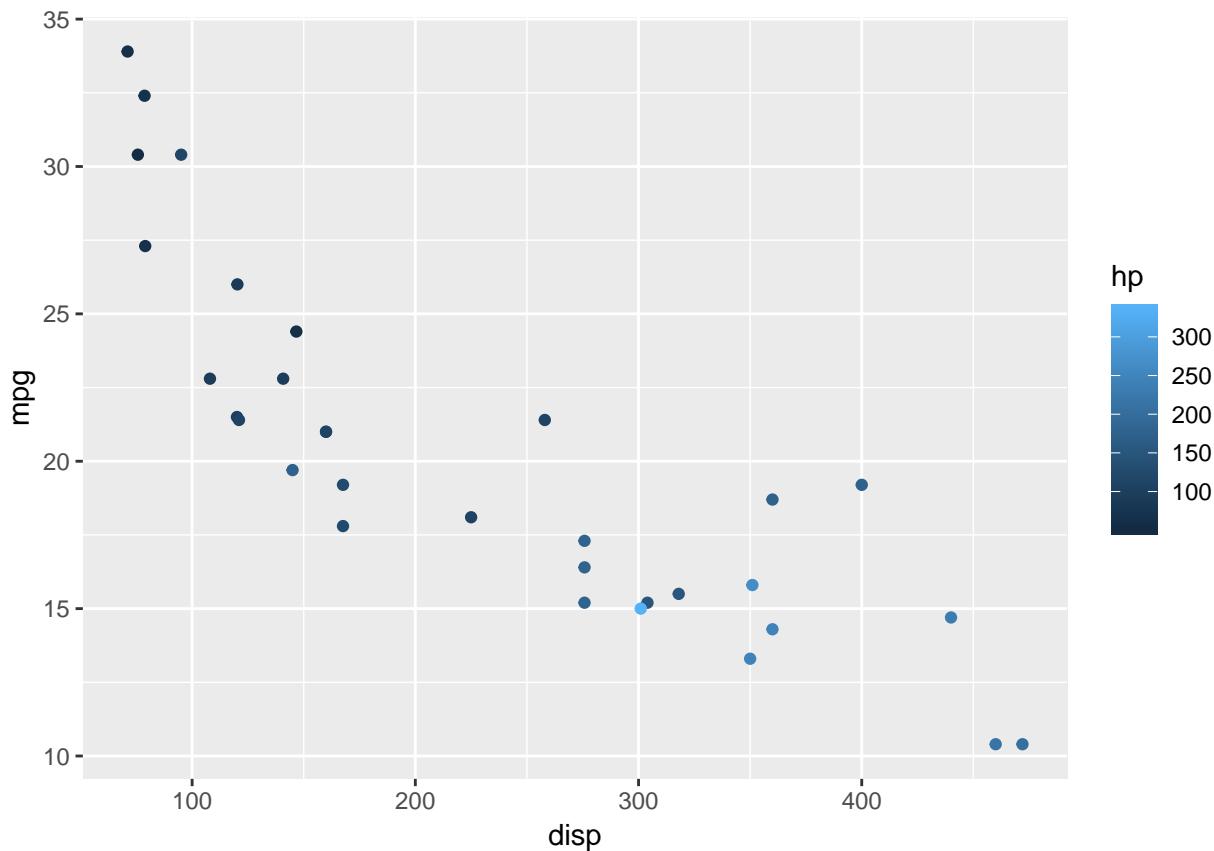
5.0.0.0.3 Step 3

We skip step 3 for this example (asking ggplot to automatically make summaries of our data before plotting), but will cover it below - see the `stat_summary()` function.

5.0.0.0.4 Step 4: Display data

To display data, we have to add a visual layer to the plot. For example, let's say we want to make a scatter plot, and so draw points for each row of data:

```
mtcars %>%
  ggplot(aes(x = disp, y = mpg, colour=hp)) +
  geom_point()
```



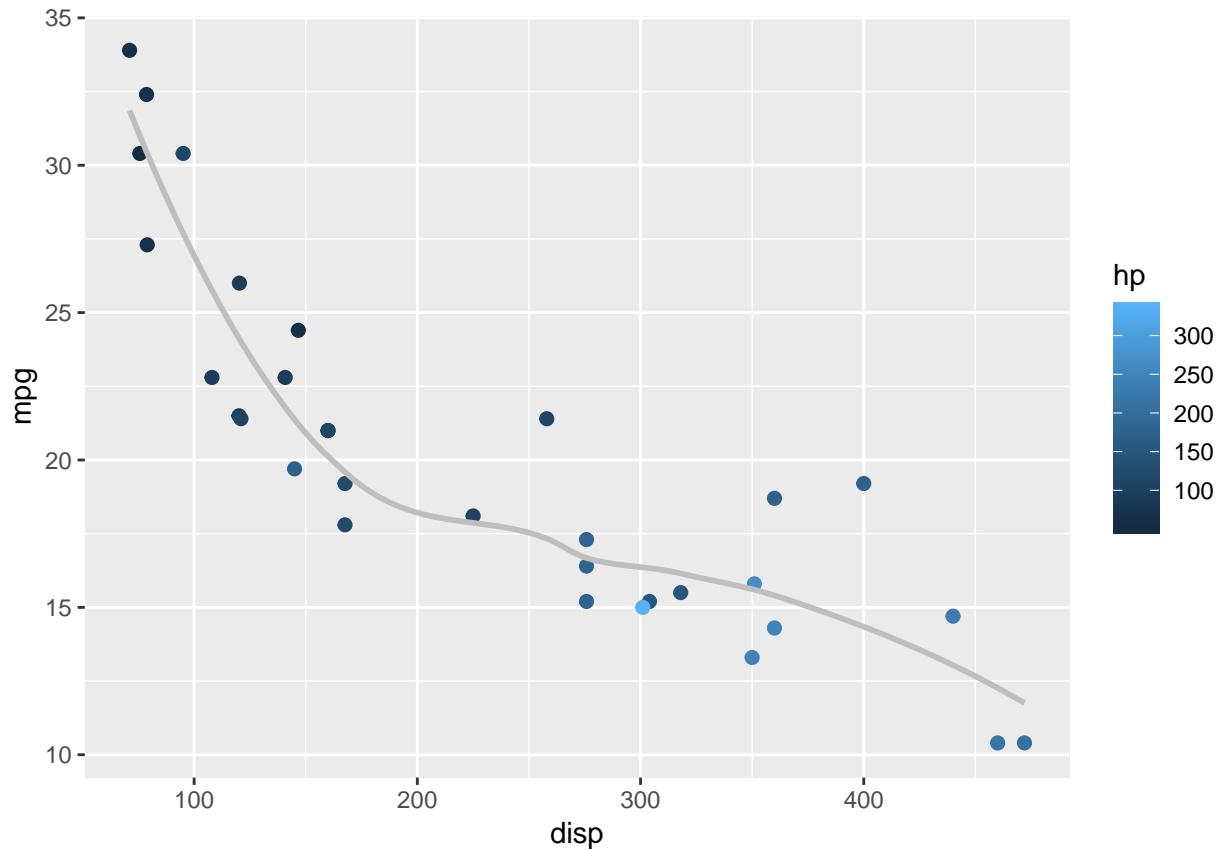
And we have a pretty slick graph: `ggplot` has now added points for each pair of `disp` and `mpg` values, and coloured them according to the value of `hp` (see choosing colours below).

Use the `airquality` dataset and create your own scatterplot and try to colour the points using the `Month` variable. Should `Month` be used as a factor or a numeric variable when colouring the points?

What's even neater about `ggplot` though is how easy it is to *layer* different visualisations of the same data. These visual layers are called `geom`'s and the functions which add them are all prefixed with `geom_`, so

`geom_point()` for scatter plots, or `geom_line()` for line plots, or `geom_smooth()` for a smoothed line plot. We can add this to the scatter plot like so:

```
mtcars %>%
  ggplot(aes(x = disp, y = mpg, colour=hp)) +
  geom_point(size=2) +
  geom_smooth(se=F, colour="grey")
```



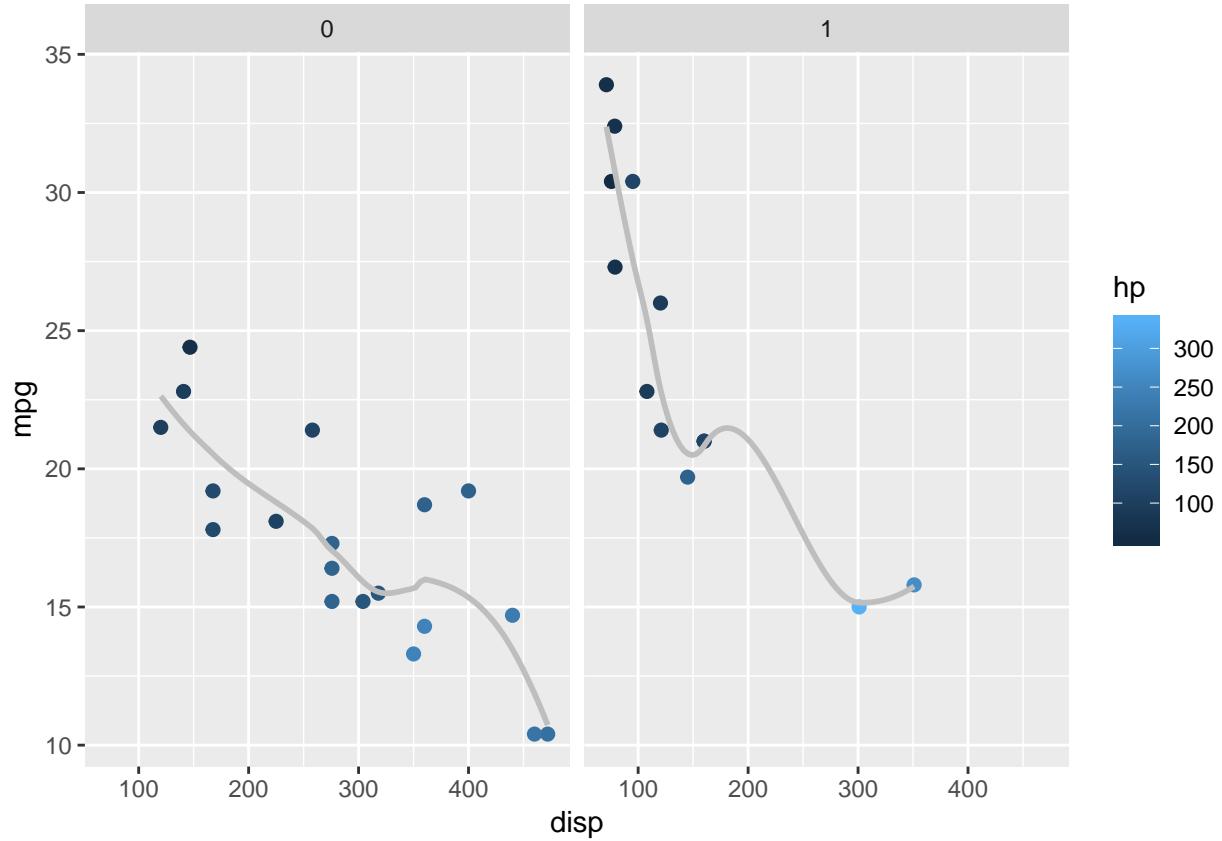
In the example above, I have also customised the smoothed line, making it grey to avoid over-intrusion into our perception of the points. Often less is more when plotting graphs: not everything can be emphasised at once, and *it's important to make decisions about what should be given visual priority*.

5.0.0.0.5 Step 5: ‘Splitting up’ or repeating the plot.

Very often, you will have drawn plot and think things like *I wonder what that would look like if I drew it for men and women separately?*. In `ggplot` this is called facetting, and is easy to achieve, provided your data are in a long format.

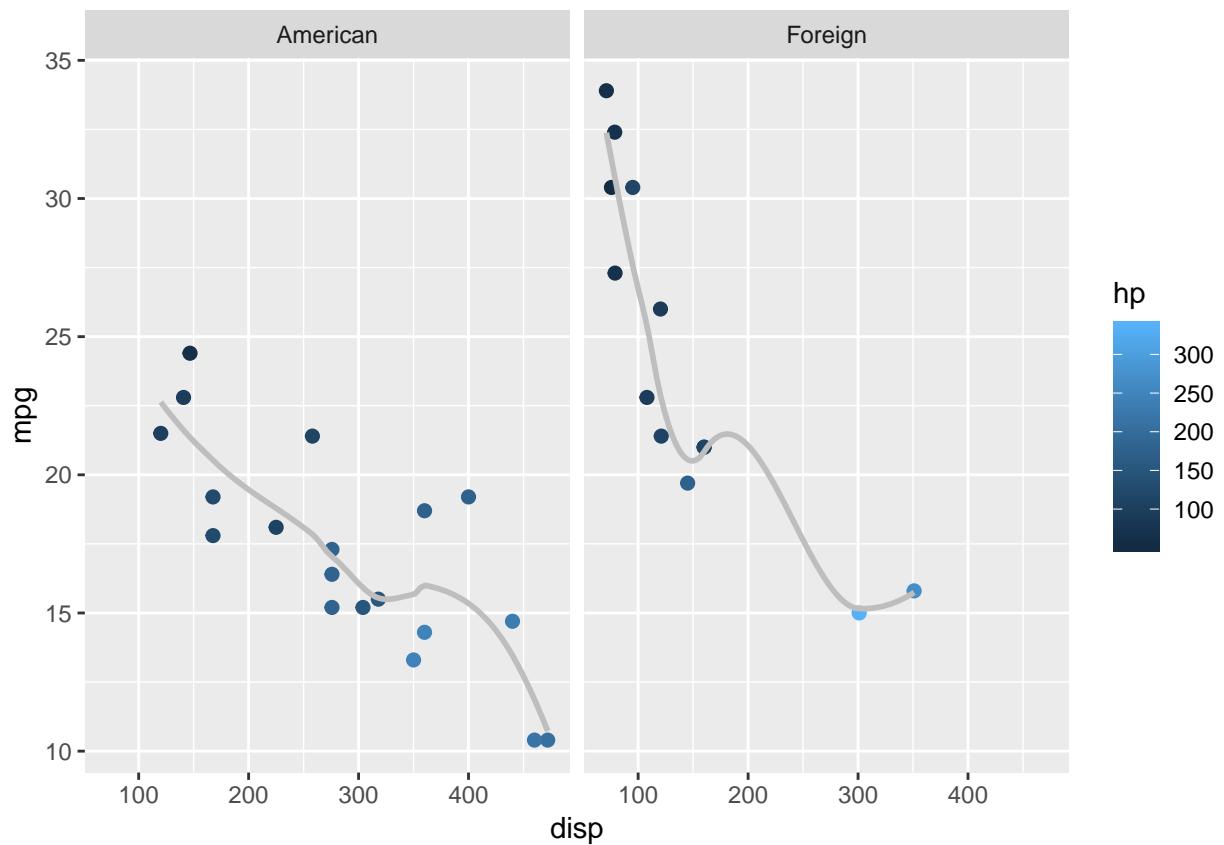
Using the same `mtcars` example, let’s say we wanted separate panels for American vs. Foreign cars (information held in the `am` variable). We simply add the `facet_wrap()`, and specify the "`am`" variable:

```
mtcars %>%
  ggplot(aes(x = disp, y = mpg, colour=hp)) +
  geom_point(size=2) +
  geom_smooth(se=F, colour="grey") +
  facet_wrap("am")
```



One trick is to make sure factors are labelled nicely, because these labels appear on the final plot. Here the `mutate()` call relabels the factor which makes the plot easier to read:

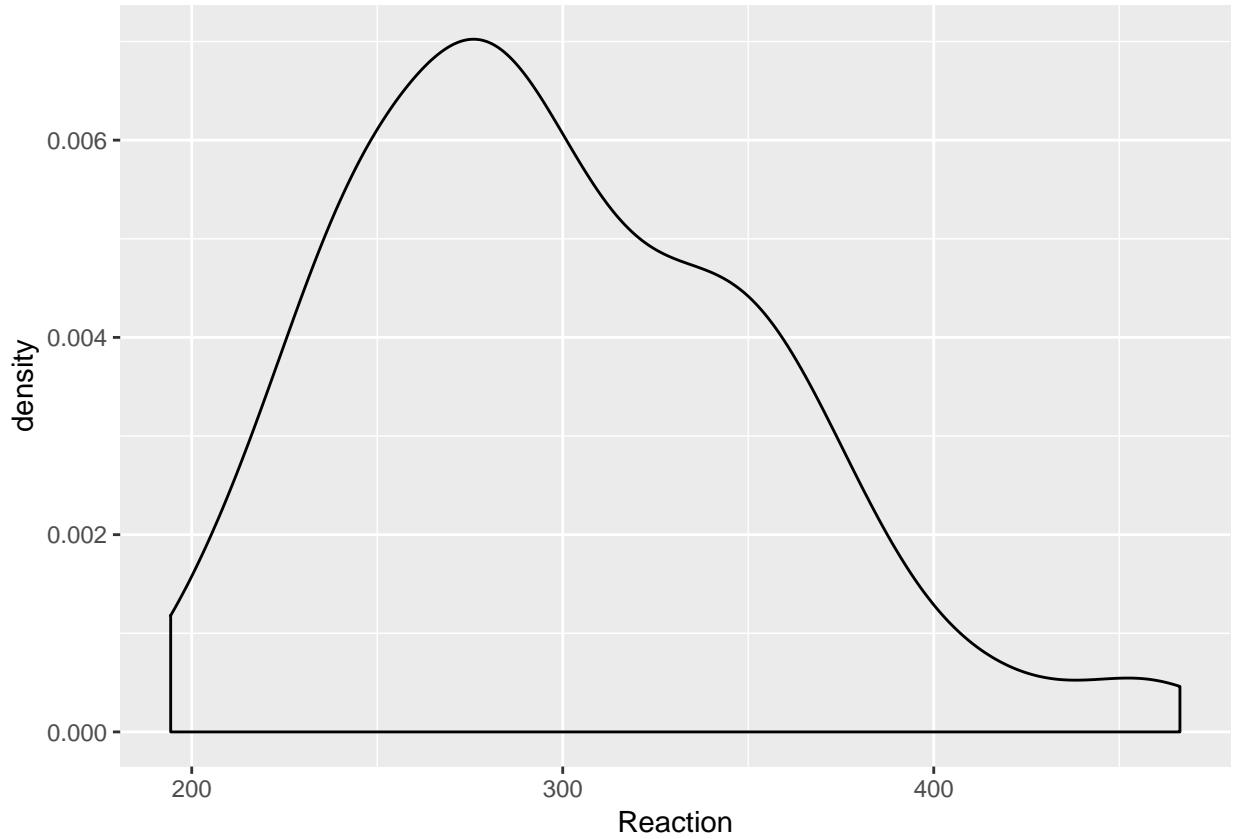
```
mtcars %>%
  mutate(american = factor(am, labels=c("American", "Foreign"))) %>%
  ggplot(aes(x = disp, y = mpg, colour=hp)) +
  geom_point(size=2) +
  geom_smooth(se=F, colour="grey") +
  facet_wrap("american")
```



See the ggplot documentation on facetting for more details.

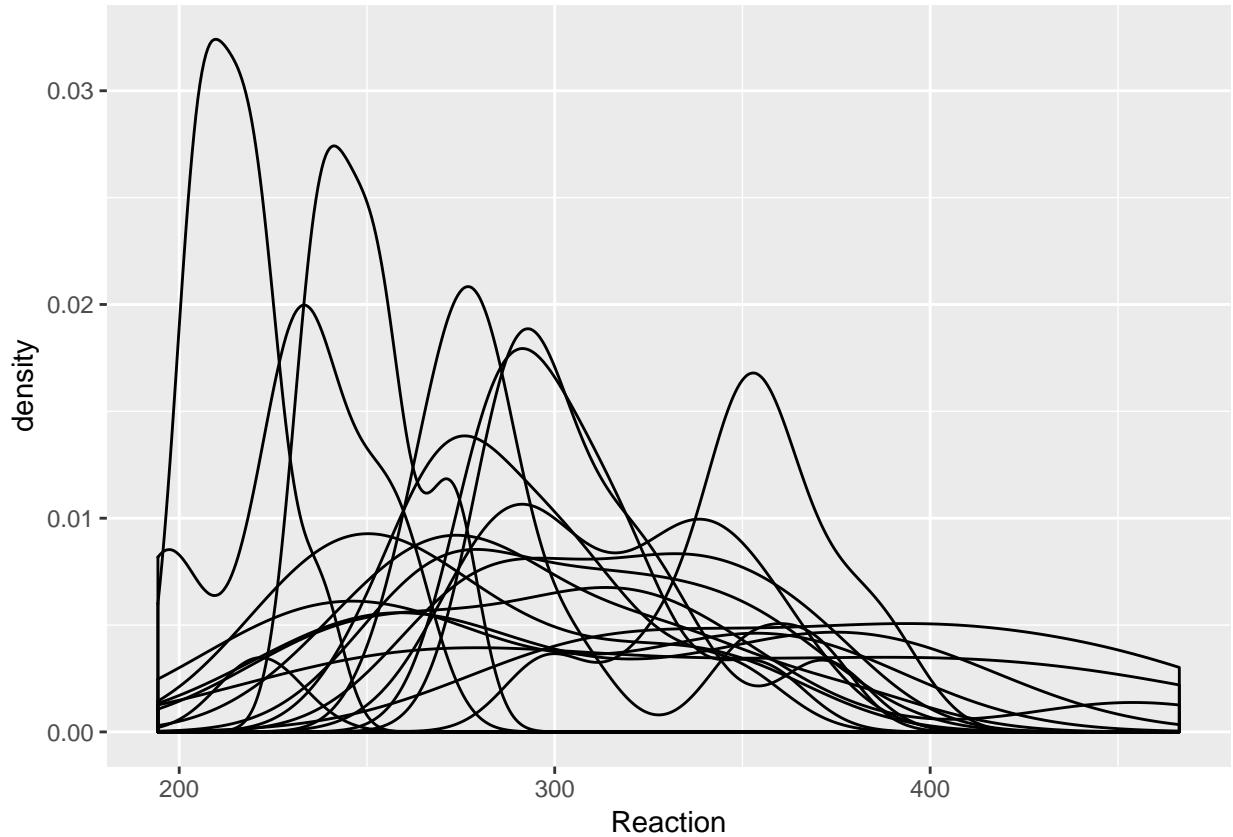
'Distributions'

```
lme4:::sleepstudy %>%
  ggplot(aes(Reaction)) + geom_density()
```



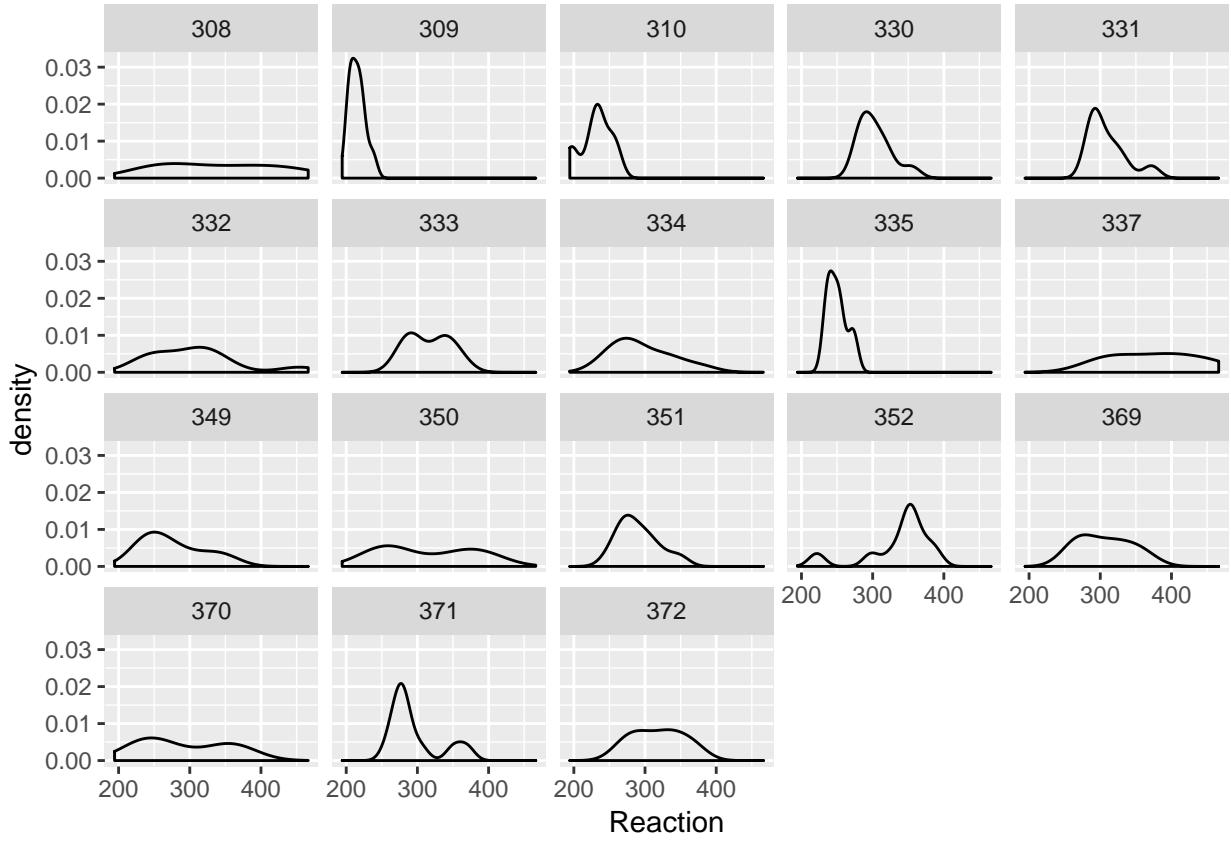
Imagine we wanted to compare distributions for individuals. Simply overlaying the lines is confusing:

```
lme4:::sleepstudy %>%  
  ggplot(aes(Reaction, group=Subject)) + geom_density()
```



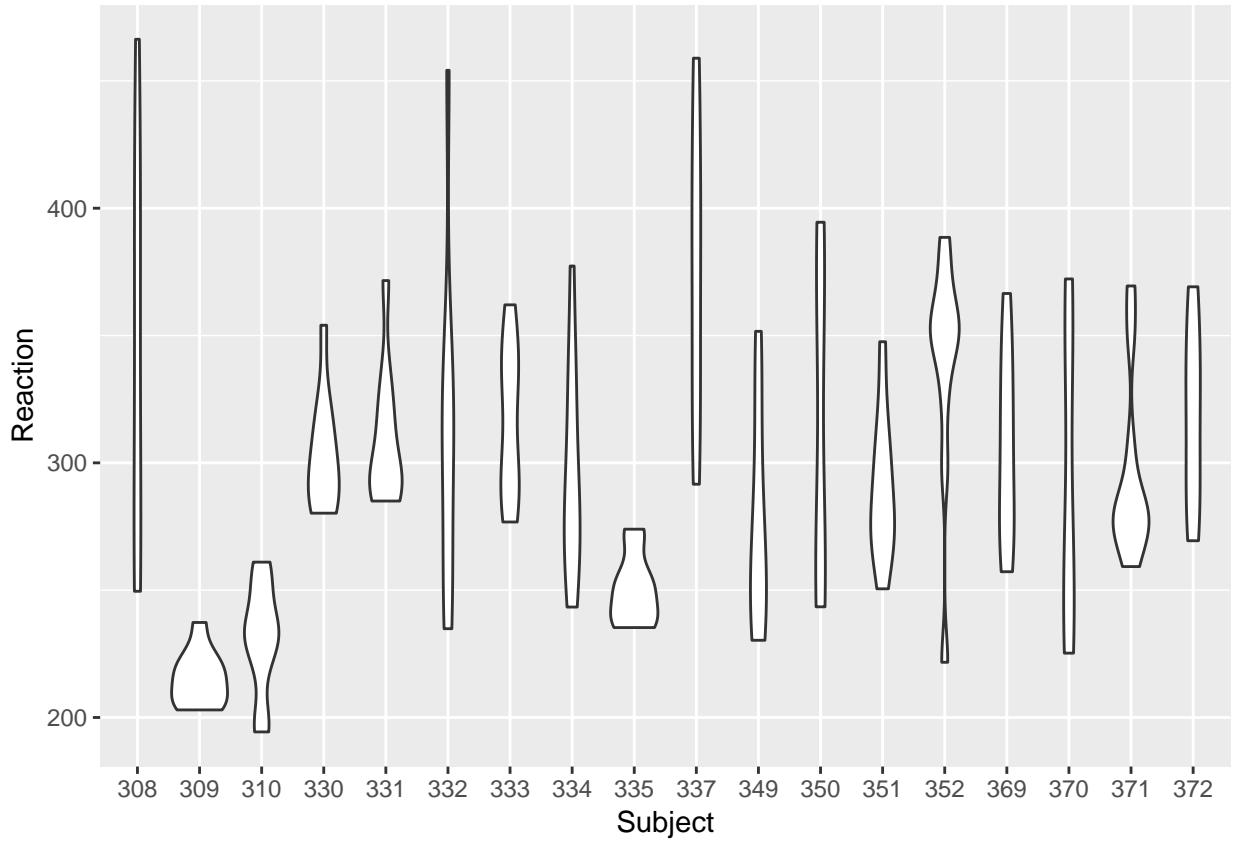
Facetting produces a nicer result:

```
lme4:::sleepstudy %>%  
  ggplot(aes(Reaction)) + geom_density() + facet_wrap("Subject")
```



But we could present the same information more compactly, and with better facility to compare between subjects, if we use a bottleplot:

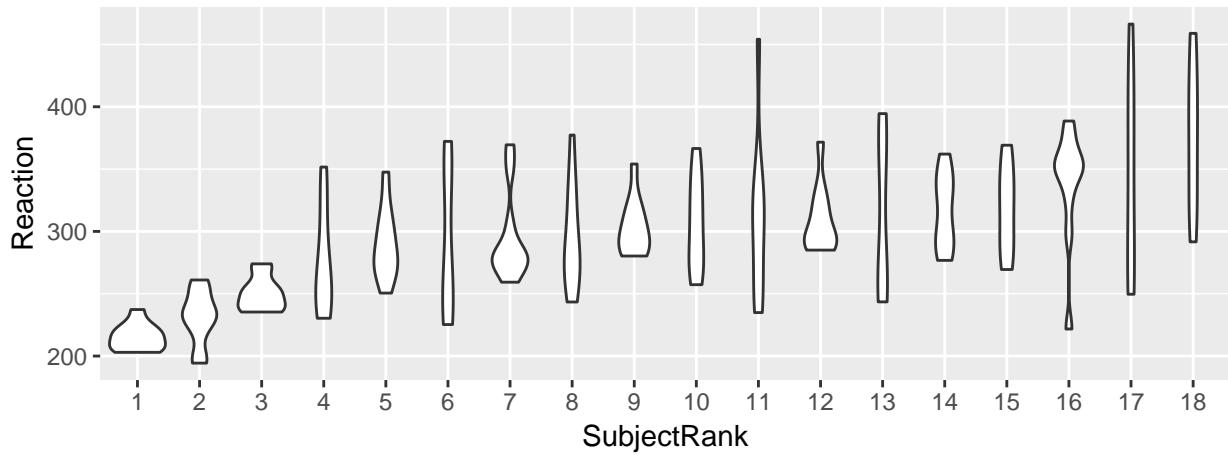
```
lme4:::sleepstudy %>%
  ggplot(aes(Subject, Reaction)) +
  geom_violin()
```



We might want to plot our Subjects in order of their mean RT:

```
mean.ranked.sleep <- lme4::sleepstudy %>%
  group_by(Subject) %>%
  # calculate mean RT
  mutate(RTm = mean(Reaction)) %>%
  # sort by mean RT
  arrange(RTm, Days) %>%
  ungroup() %>%
  # create a rank score but convert to factor right away
  mutate(SubjectRank = factor(dense_rank(RTm)))

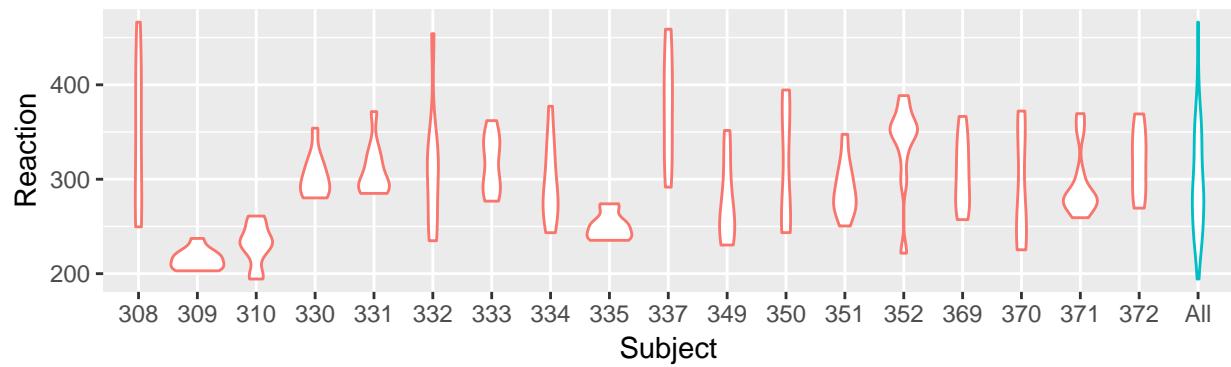
mean.ranked.sleep %>%
  ggplot(aes(SubjectRank, Reaction)) +
  geom_violin() +
  theme(aspect.ratio = .33) # change the aspect ratio to make long and wide
```



Or we might want to compare individuals against the combined distribution:

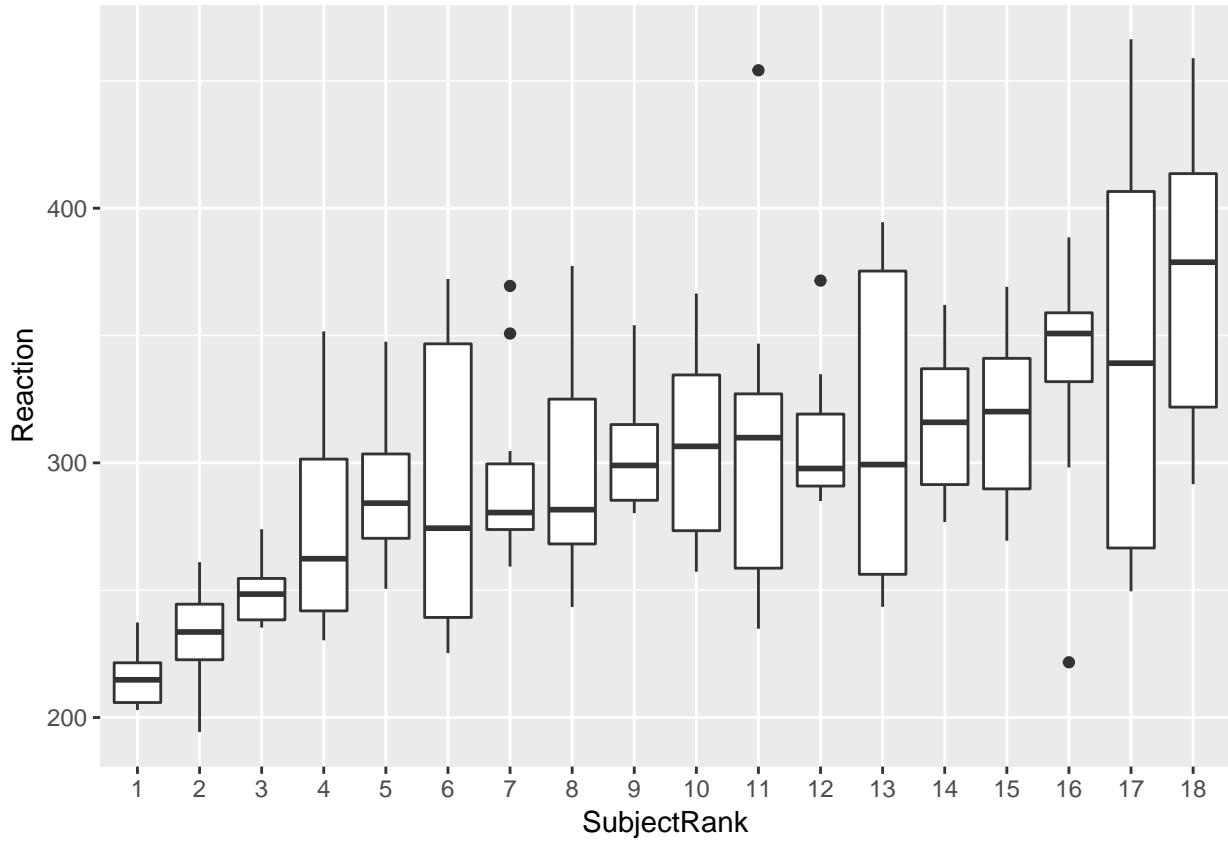
```
# duplicate all the data, assigning one-replication to a single subject, "All"
sleep.repeat <- bind_rows(lme4::sleepstudy,
                           lme4::sleepstudy %>% mutate(Subject="All"))
Warning in bind_rows_(x, .id): binding factor and character vector,
coercing into character vector
Warning in bind_rows_(x, .id): binding character and factor vector,
coercing into character vector

sleep.repeat %>%
  mutate(all = Subject=="All") %>%
  ggplot(aes(Subject, Reaction, color=all)) +
  geom_violin() +
  guides(colour=FALSE) + # turn off the legend because we don't really need it
  theme(aspect.ratio = .25) # change the aspect ratio to make long and wide
```



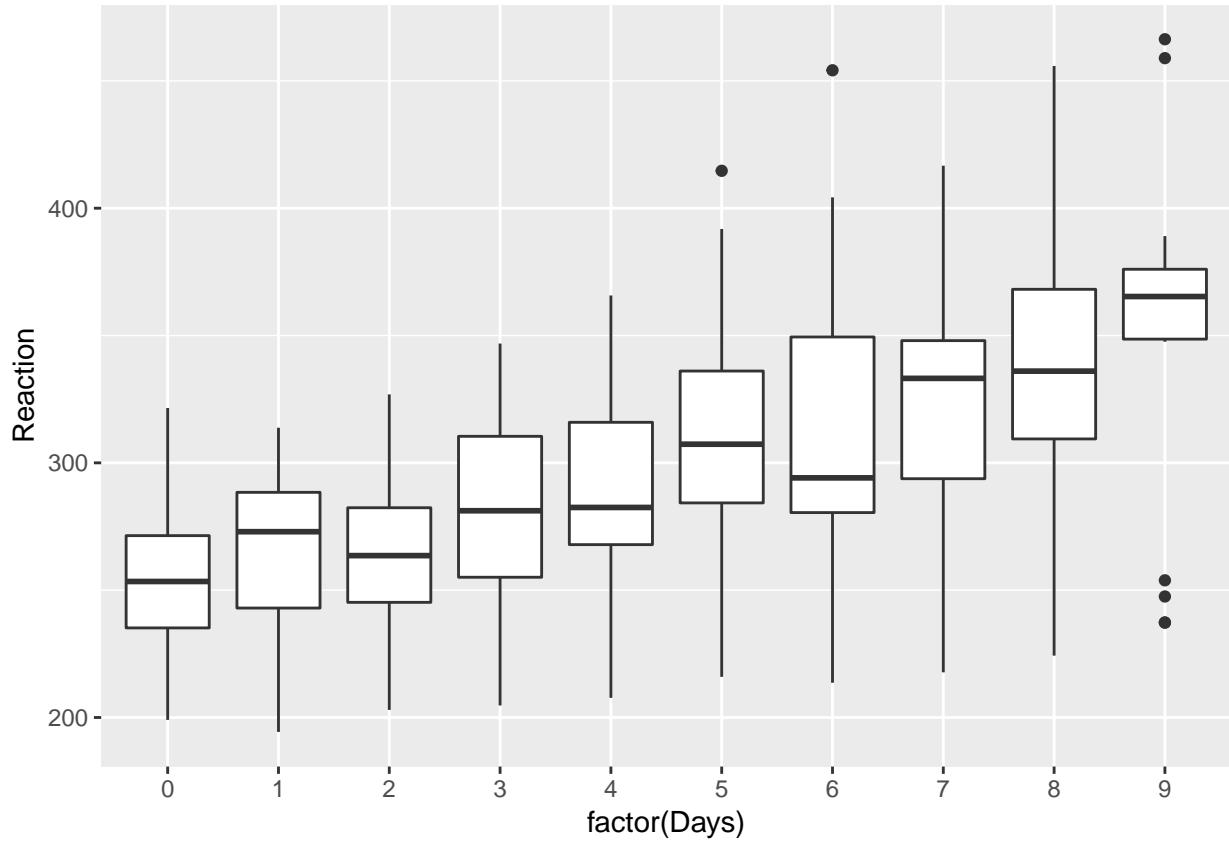
Boxplots can also work well to show distributions, and have the advantage of showing the median explicitly:

```
mean.ranked.sleep %>%
  ggplot(aes(SubjectRank, Reaction)) +
  geom_boxplot()
```



If we plot the same data by-day, we can clearly see the effect of sleep deprivation, and the increase in variability between subjects as time goes on: the lack of sleeps seems to be affecting some subjects more than others

```
lme4:::sleepstudy %>%
  ggplot(aes(factor(Days), Reaction)) +
  geom_boxplot()
```



'Comparisons'

Imagine we have rainfall and temperature data for various regions, across months:

```
DAAG::bomregions %>%
  psych::describe(fast=T) %>%
  pander
```

	vars	n	mean	sd	min	max	range	se
Year	1	109	1954	31.61	1900	2008	108	3.028
eastAVt	2	99	20.43	0.4585	19.35	21.61	2.255	0.04608
seAVt	3	99	14.59	0.4547	13.62	15.94	2.32	0.0457
southAVt	4	99	18.48	0.4584	17.43	19.53	2.1	0.04607
swAVt	5	99	16.18	0.4734	15.08	17.05	1.97	0.04758
westAVt	6	99	22.33	0.4548	21.22	23.39	2.165	0.04571
northAVt	7	99	24.6	0.5042	23.57	25.94	2.365	0.05067
mdbAVt	8	99	17.59	0.4958	16.36	18.79	2.425	0.04983
auAVt	9	99	21.71	0.4429	20.67	22.87	2.195	0.04452
eastRain	10	109	601.6	123.8	315.3	1030	715	11.85
seRain	11	109	598.1	104.6	354.9	900.6	545.7	10.02
southRain	12	109	381.7	68.63	236	618.2	382.1	6.574
swRain	13	109	657.8	103	420.5	988.8	568.4	9.861
westRain	14	109	352.2	84.55	173.5	646.5	473	8.098
northRain	15	109	520.7	110.2	312.8	946.9	634	10.55
mdbRain	16	109	476	110.9	255.8	821	565.2	10.62

	vars	n	mean	sd	min	max	range	se
auRain	17	109	457.1	82.55	317.2	785.3	468.1	7.906
SOI	18	109	-0.002676	6.845	-20.01	20.79	40.8	0.6556
co2mlo	19	50	345.6	21.01	316	385.4	69.47	2.972
co2law	20	79	310.4	9.586	295.8	333.7	37.9	1.078
CO2	21	109	324.3	24.59	296.3	385.4	89.19	2.355
sunspot	22	109	60.08	47.69	1.4	190.2	188.8	4.568

5.0.0.0.6

Because these data are in wide format (multiple columns have contain the same type of data) we need to first convert to long format. This process has become known as tidying.

The code below selects the columns related to rainfall and temperature and then ‘melts’ the data to long format: one row per observation. We then `extract` the region and the type of measurement from the `variable` column which is created by using a regular expression:

```
weather.data.long <- DAAG::bomregions %>%
  select(Year, ends_with('Rain'), ends_with('AVt')) %>%
  reshape2::melt(id.var="Year") %>%
  extract(variable,
    into=c("Region", "variable"),
    regex="(east|north|se|south|west|sw|au)(\\w+)" ) %>%
  filter(!is.na(variable))

weather.data.long %>% head
  Year Region variable   value
1 1900   east     Rain 429.98
2 1901   east     Rain 500.12
3 1902   east     Rain 315.33
4 1903   east     Rain 694.09
5 1904   east     Rain 564.86
6 1905   east     Rain 443.11
```

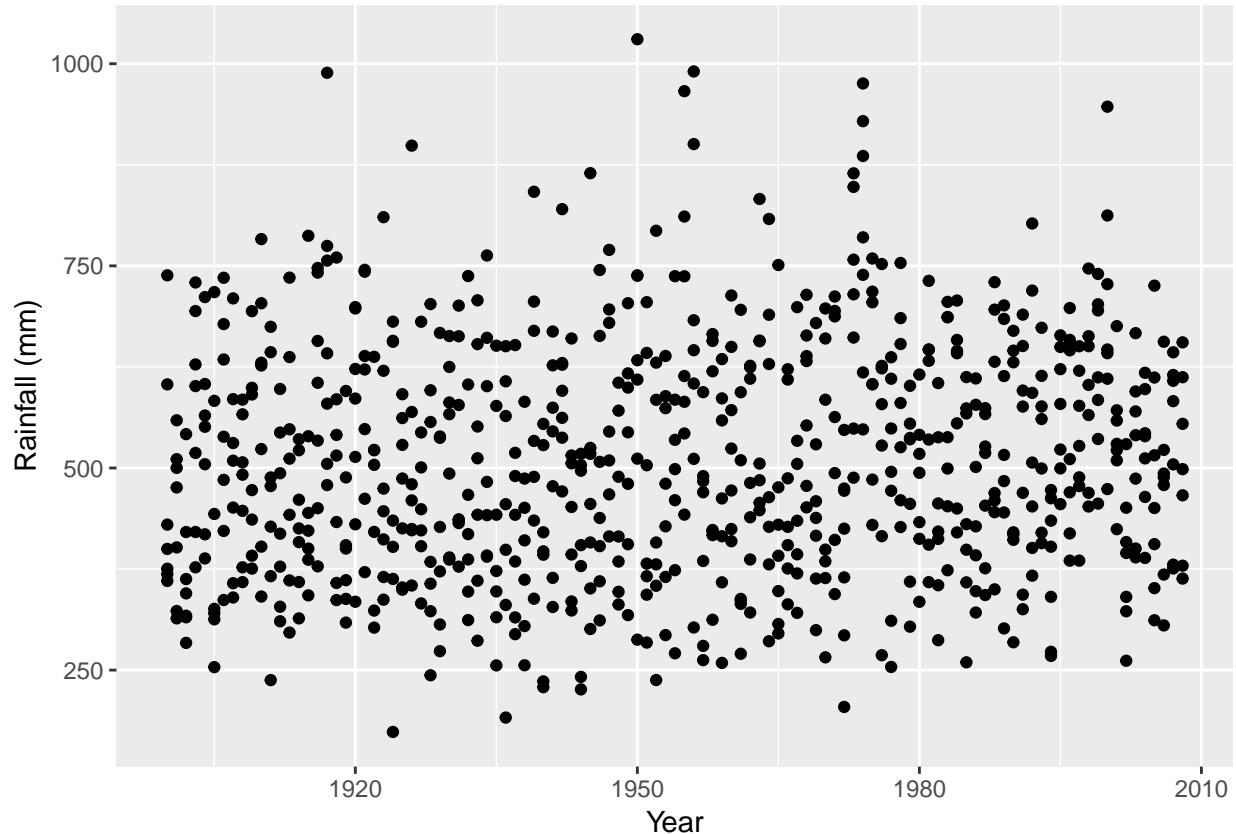
5.0.0.0.7

Try stepping through the code above line by line and see what is produced at each step.

With our data in long form we can use `ggplot` to plot our data over time. In the plot below, we use `filter()` to select only the rainfall measurements:

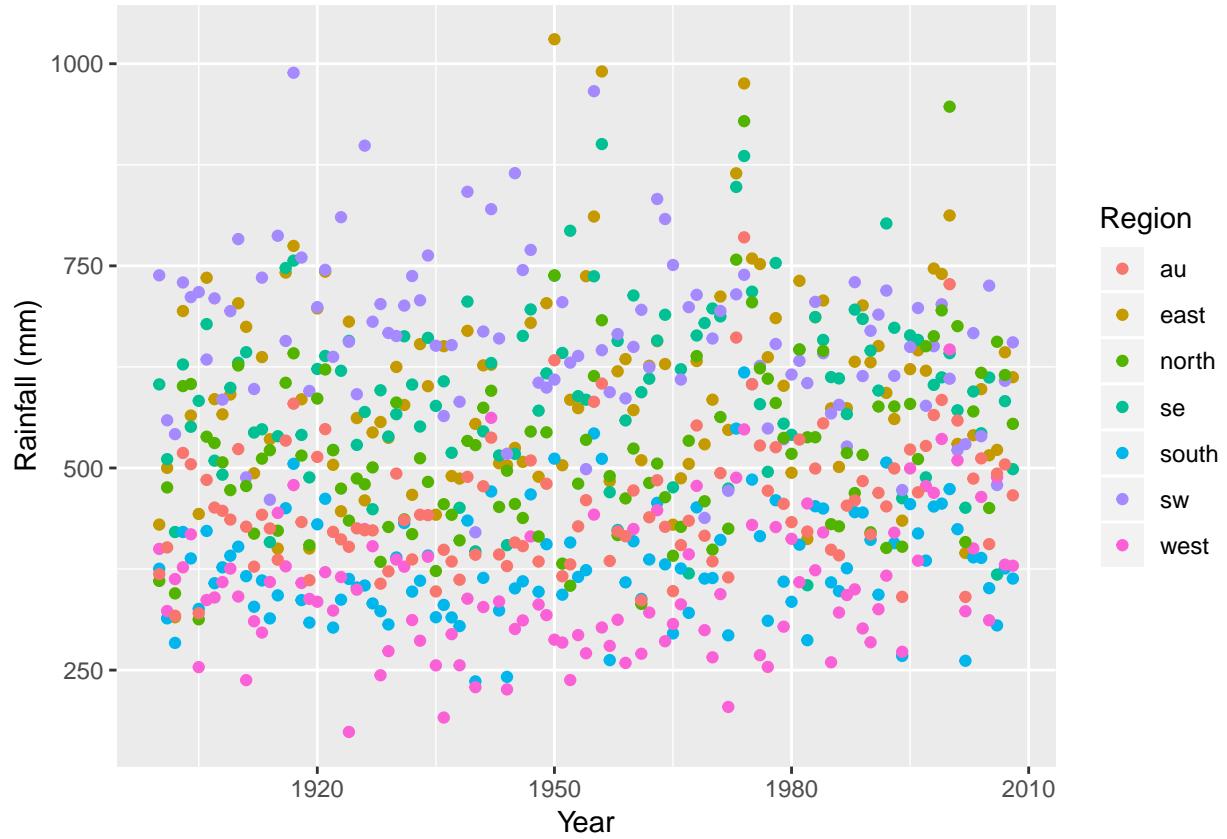
```
rain <-
  weather.data.long %>%
  filter(variable=='Rain')

rain %>%
  ggplot(aes(Year, value)) +
  geom_point() +
  ylab('Rainfall (mm)')
```



There are many ways to extract structure from these data, and make comparisons over time. One is to use colour:

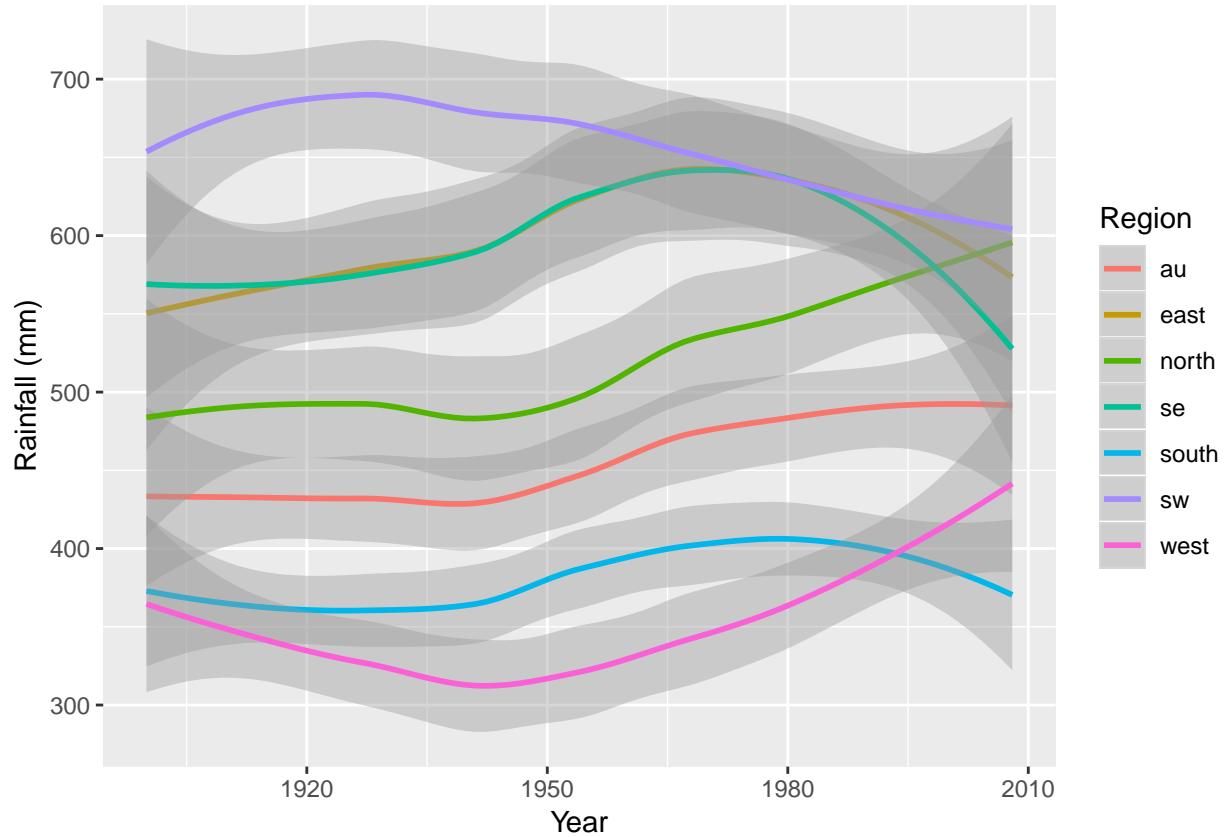
```
rain %>%
  ggplot(aes(Year, value, color=Region)) +
  geom_point() +
  ylab('Rainfall (mm)')
```



It's now easy to see the crude differences between the regions (west appears the driest, sw the wettest), but it's still hard to compare between regions, or over time.

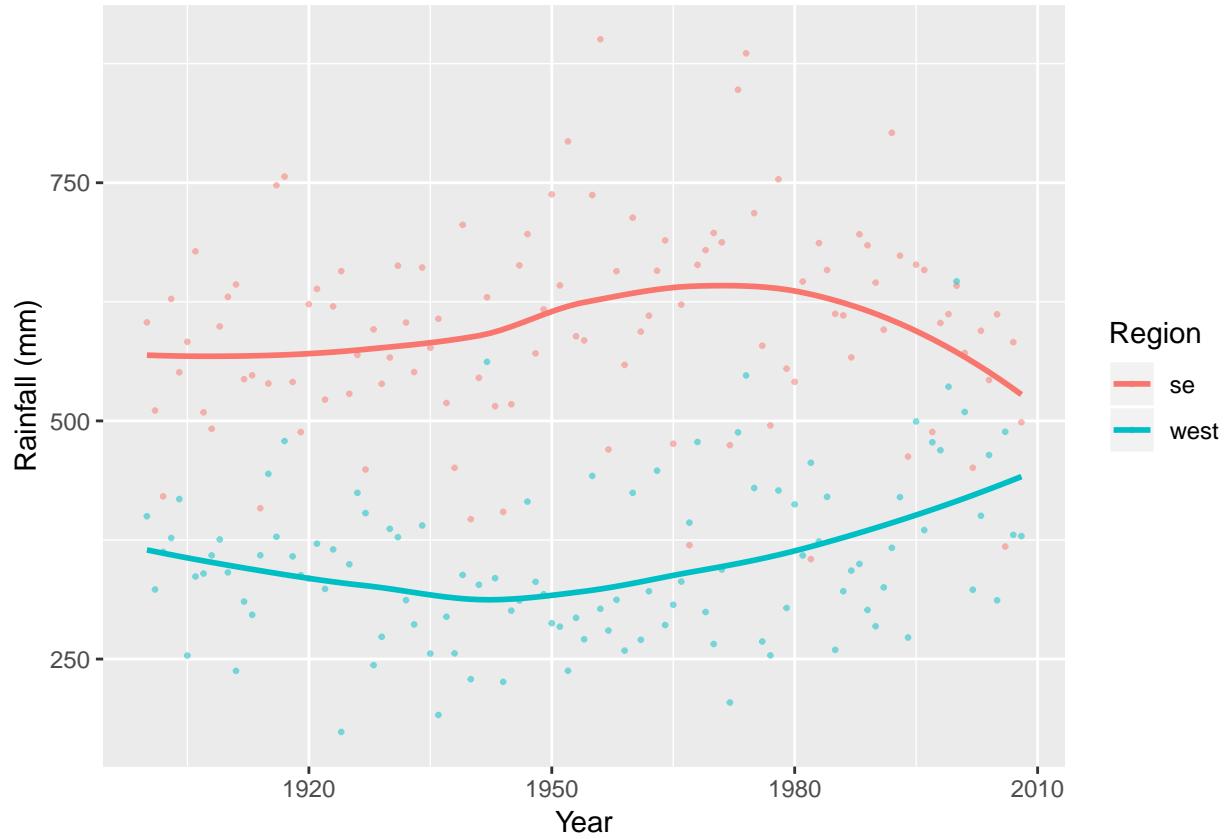
For this we can use various forms of summaries, for example a smoothed line plot (the shaded area is the standard error), which clearly shows the relative changes in rainfall in each region over time:

```
rain %>%
  ggplot(aes(Year, value, color=Region)) +
  geom_smooth() +
  ylab('Rainfall (mm)')
```



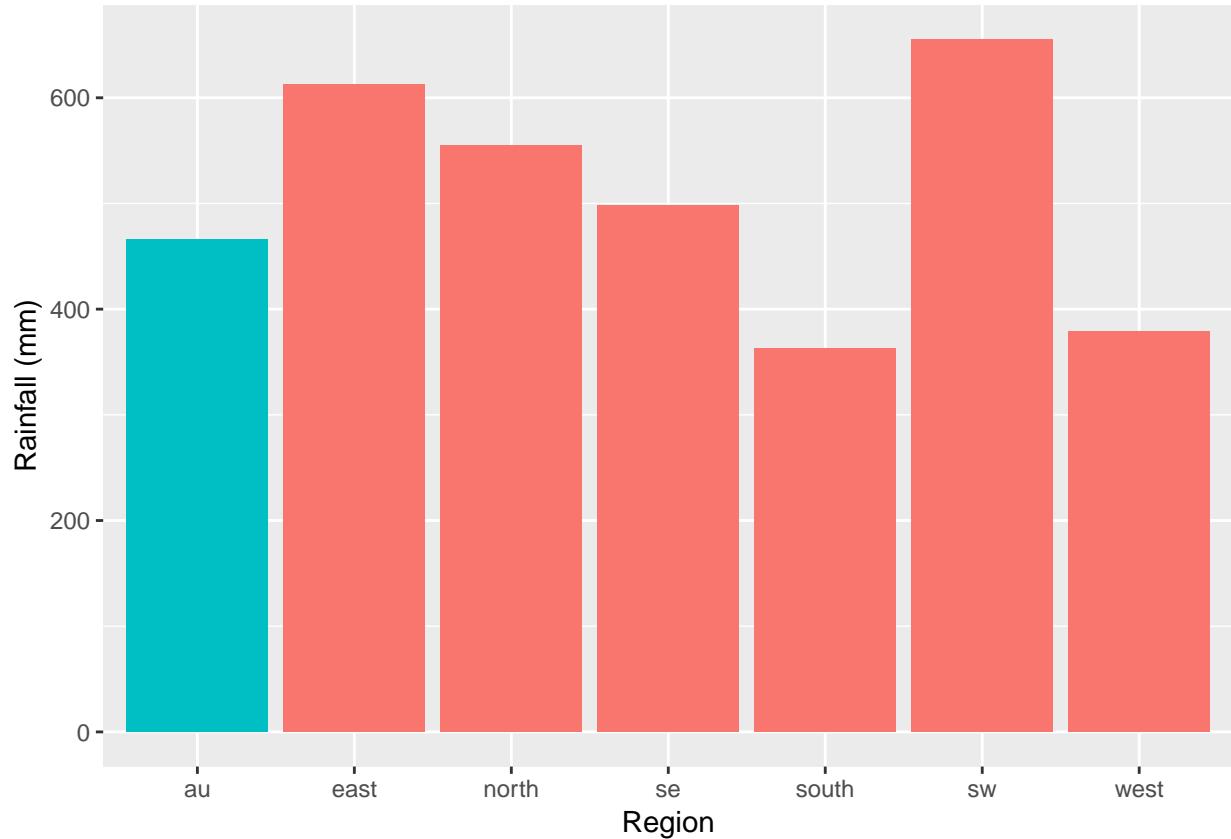
It can sometimes be desirable to include points to preserve the relationship between the summary and the raw data:

```
rain %>%
  filter(Region %in% c('west', 'se')) %>%
  ggplot(aes(Year, value, color=Region)) +
  geom_point(alpha=.5, size=.5) +
  geom_smooth(se=F) +
  ylab('Rainfall (mm)')
```



If we weren't interested in the time series and just wanted to focus on the most recent year, we might take a different approach. Here a bar plot is used to compare between regions, with the national average (au) highlighted in blue:

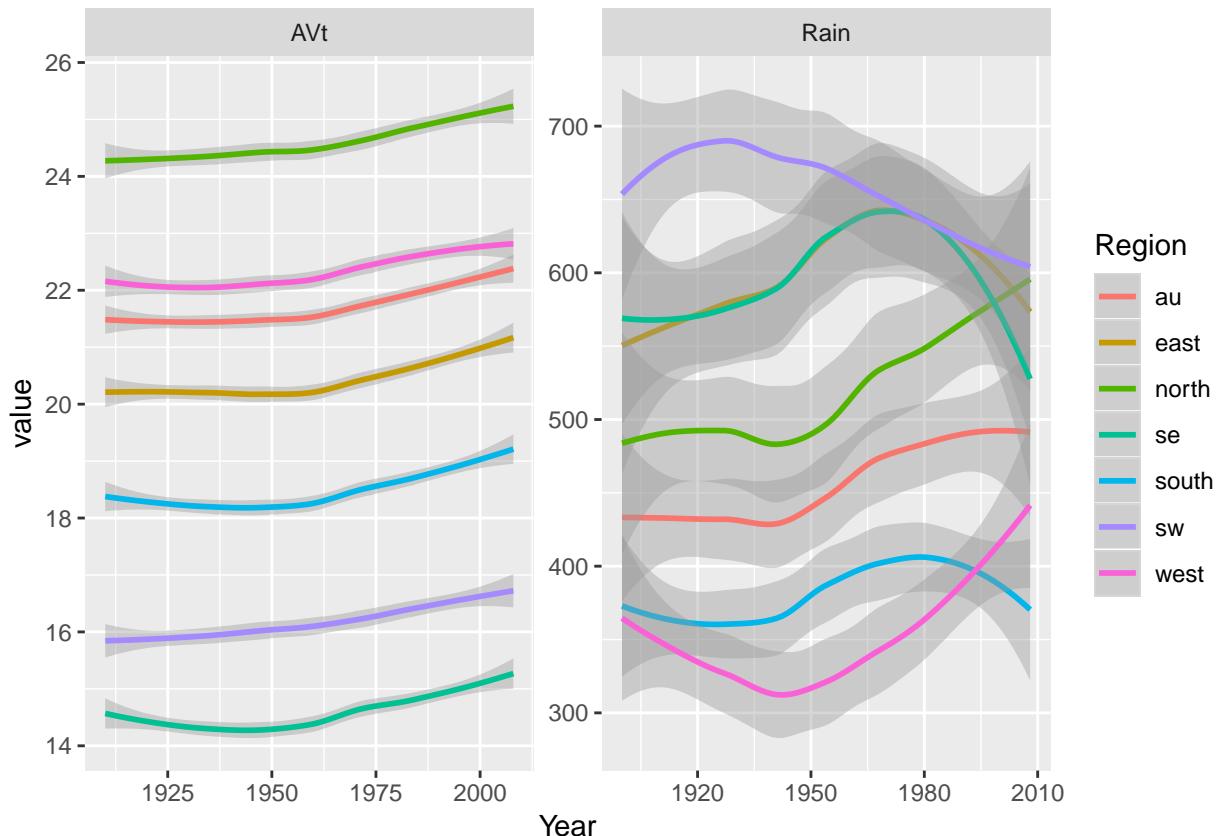
```
rain %>%
  filter(Year == max(Year)) %>%
  ggplot(aes(Region, value, fill=Region=="au")) +
  stat_summary(geom="bar") +
  ylab('Rainfall (mm)') +
  guides(fill=F)
```



Finally, we shouldn't forget that this dataset included both rainfall and temperature data, and we can display these in different ways, depending on what our research question was.

In this case we can use facetting to display temperature and rainfall data side by side. Because AVt and Rain are on such different scales we need to allow the y axis to vary between variables:

```
weather.data.long %>%
  ggplot(aes(Year, value, color=Region)) +
  geom_smooth() +
  facet_wrap(~variable, scales="free")
Warning: Removed 70 rows containing non-finite values (stat_smooth).
```



'Composition'

5.0.0.1 Waffle plots or 'pictograms'

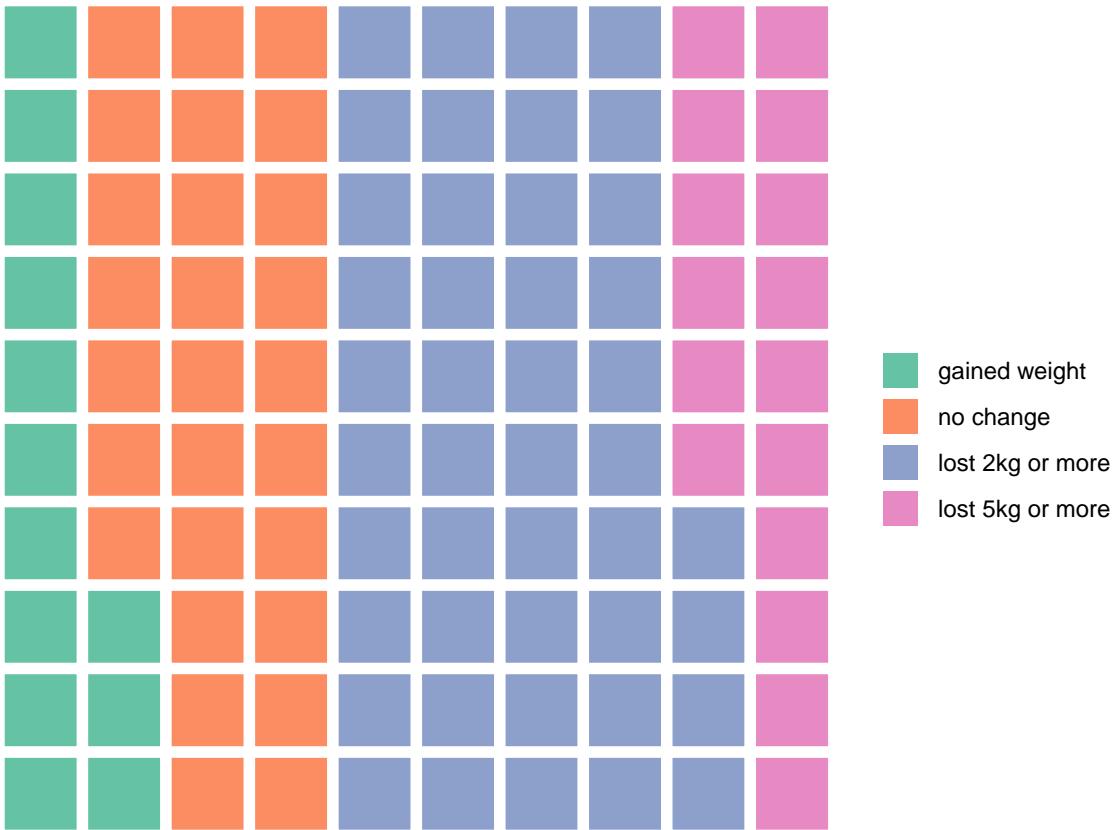
Waffle plots are neat way of showing the relative frequency of different categories.

In applied settings it's well known that when considering the risks or benefits of interventions clinicians, patients and researchers benefit from statements made using 'natural frequencies' [Gigerenzer and Edwards, 2003], and pictographs or 'waffle plots' have been shown to provide patients and their families with a better understanding of the risks of treatments [Tait et al., 2010].

Waffle plots can be implemented in R via the `waffle:::` package:

```
outcomes <- c("gained weight"=13,
              "no change"=27,
              "lost 2kg or more" = 44,
              "lost 5kg or more" = 16)

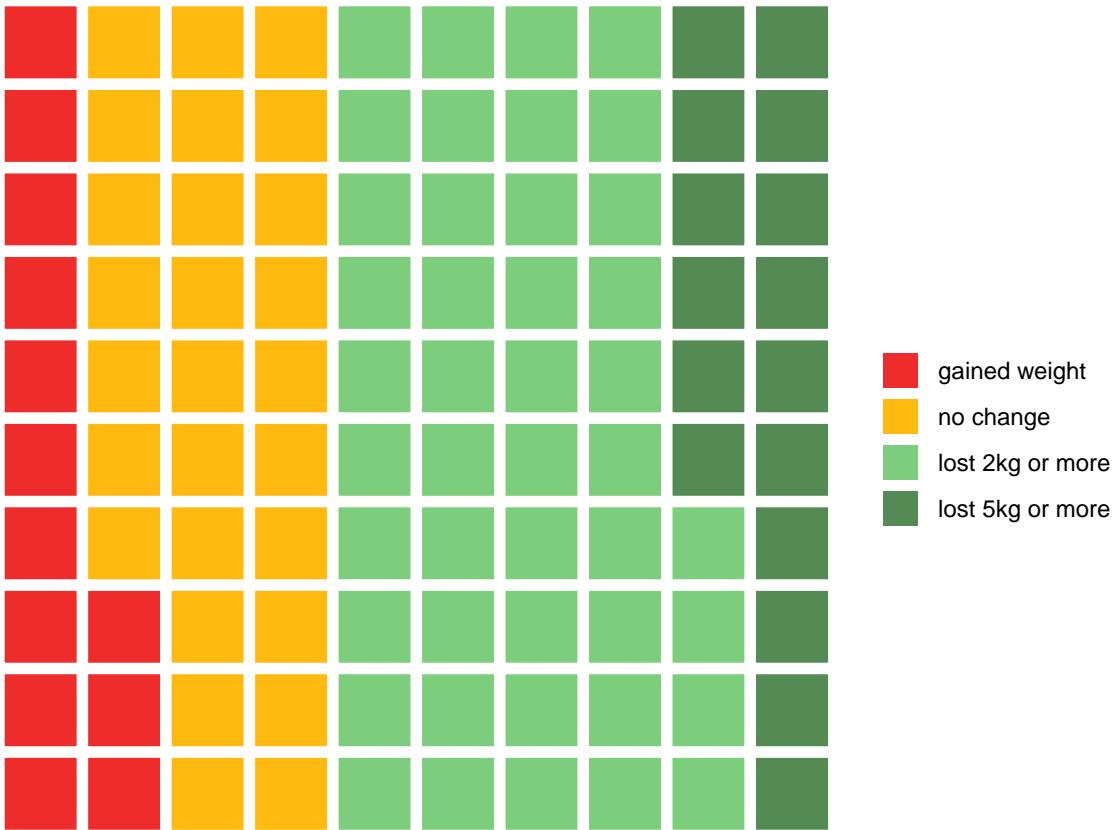
waffle:::waffle(outcomes)
```



Calculating these summary figures is left as an exercise to for the reader, but see the section on summarising data with dplyr.

This is one of those occasions where the default ggplot colours could probably be improved, and we can do this by passing the names of the colours we would like to use:

```
weight.loss.colours <- c('firebrick2', 'darkgoldenrod1', 'palegreen3', 'palegreen4')
waffle::waffle(outcomes, colors = weight.loss.colours)
```



Selecting colours by hand isn't always the best way though: the `colourbrewer` library provides some nice shortcuts for using palettes from the excellent ColourBrewer website.

5.0.0.2 Stacked bars

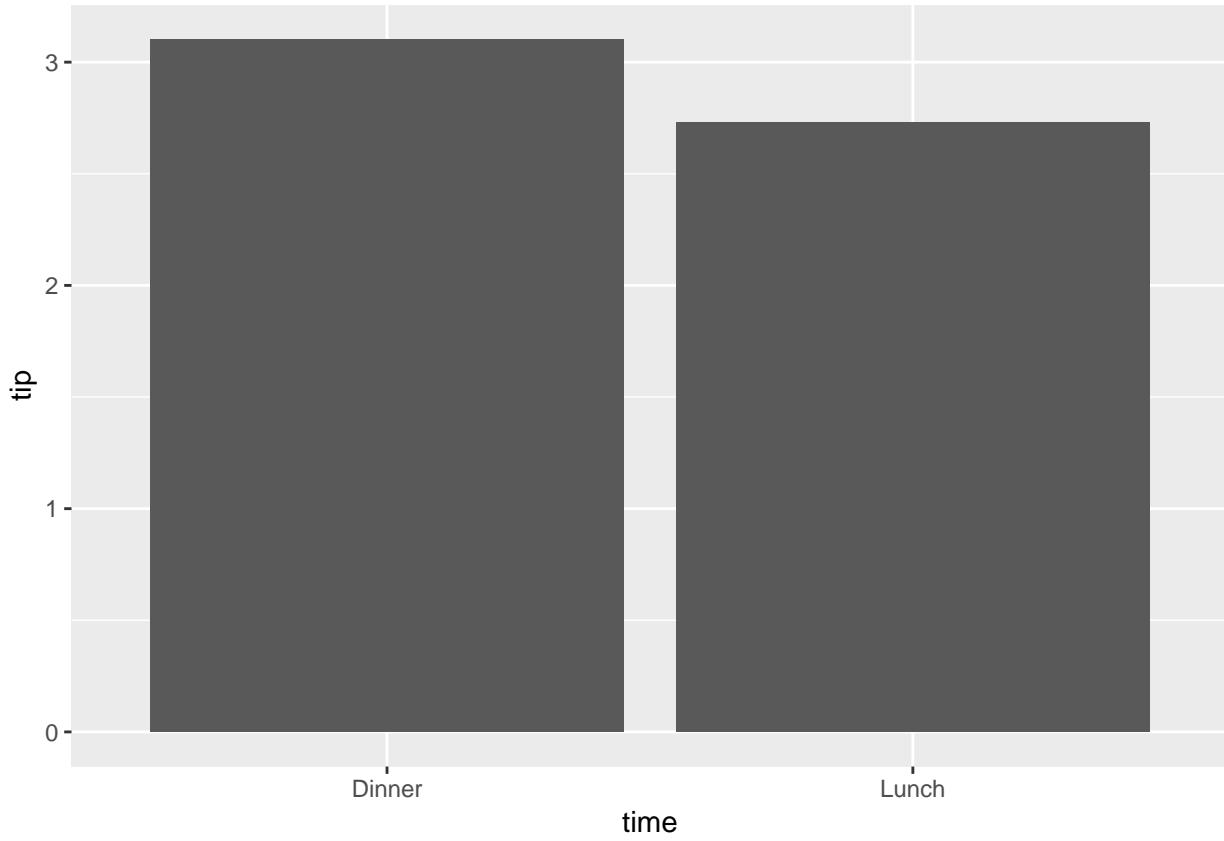
The `reshape2` package includes data on tipping habits in restaurants for male and female bill-payers, and where the party was for various sizes:

```
reshape2::tips %>%
  head %>%
  pander
```

total_bill	tip	sex	smoker	day	time	size
16.99	1.01	Female	No	Sun	Dinner	2
10.34	1.66	Male	No	Sun	Dinner	3
21.01	3.5	Male	No	Sun	Dinner	3
23.68	3.31	Male	No	Sun	Dinner	2
24.59	3.61	Female	No	Sun	Dinner	4
25.29	4.71	Male	No	Sun	Dinner	4

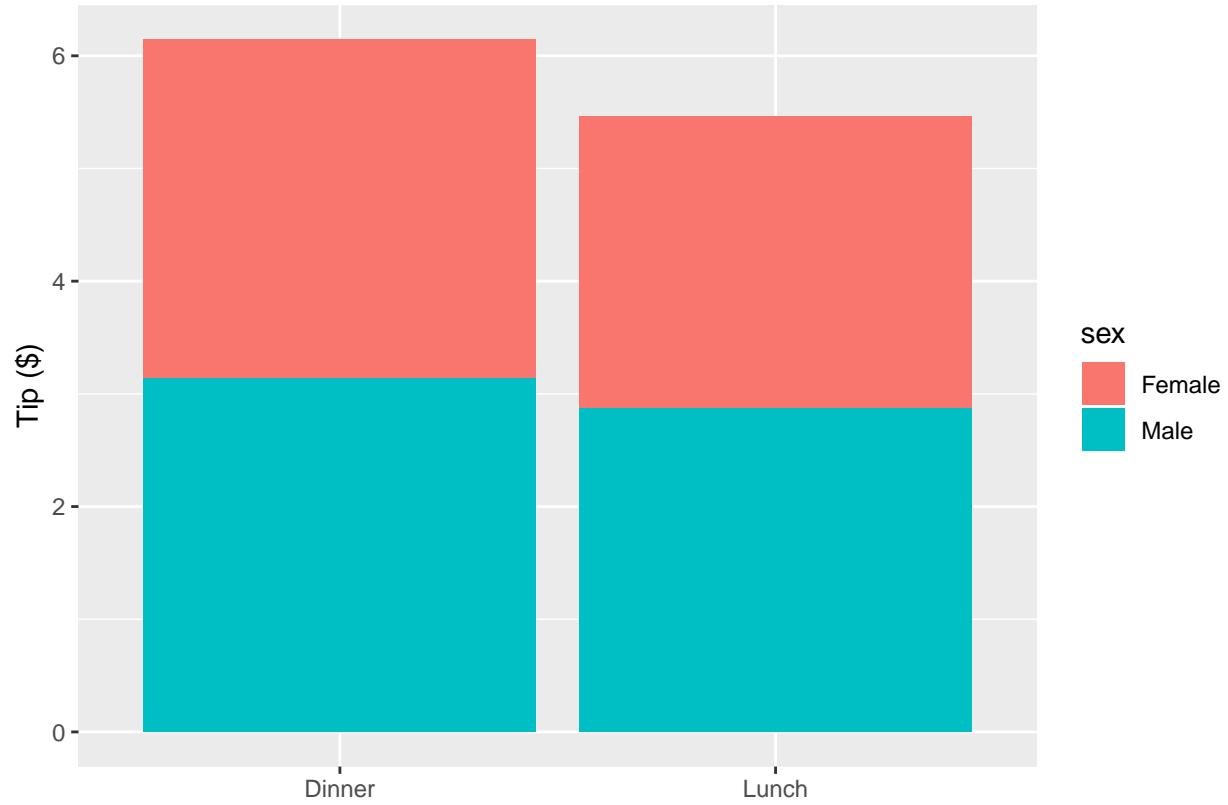
To begin, we might like to plot tips by time of day:

```
reshape2::tips %>%
  ggplot(aes(time, tip)) +
  stat_summary(geom="bar")
```



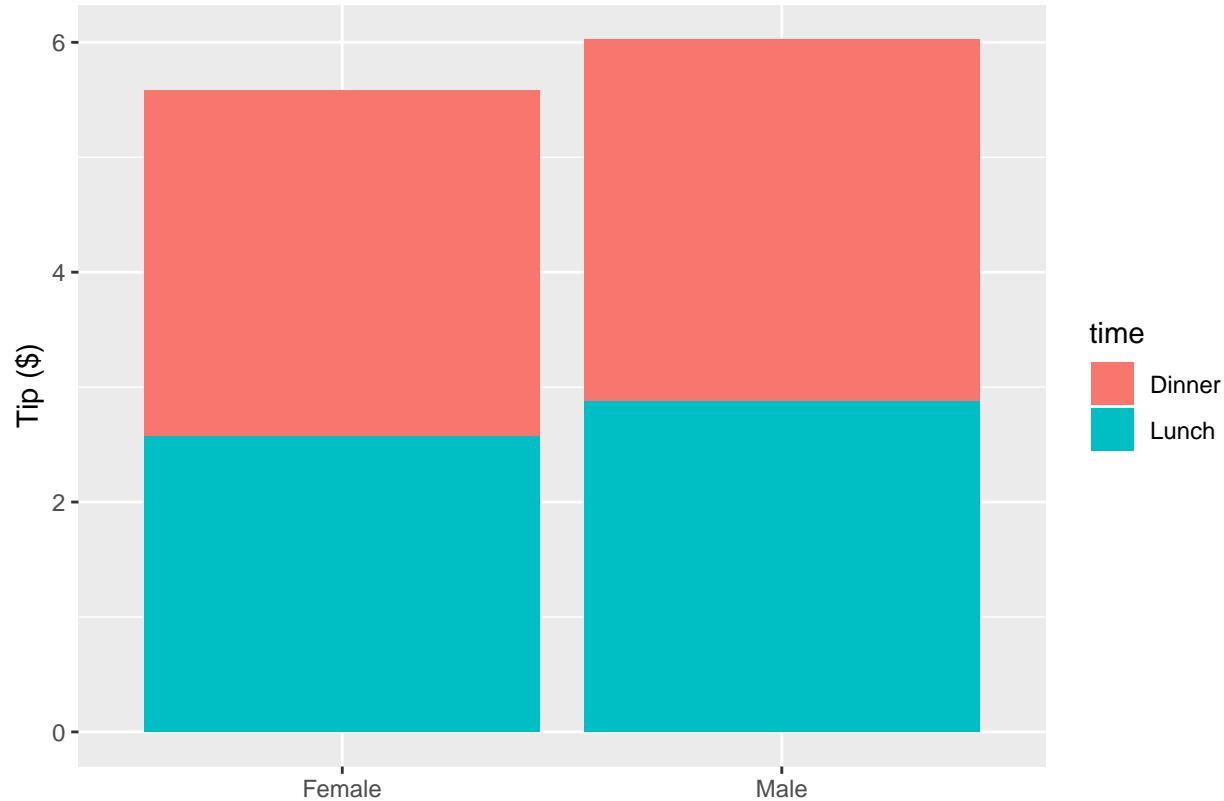
And to facilitate the comparison between men and women we could colour portions of the bars using `position_stack()`:

```
reshape2::tips %>%
  ggplot(aes(time, tip, fill=sex)) +
  stat_summary(geom="bar", position=position_stack()) +
  xlab("") + ylab("Tip ($)")
```



Or to reverse the comparisons:

```
reshape2::tips %>%
  ggplot(aes(sex, tip, fill=time)) +
  stat_summary(geom="bar", position=position_stack()) +
  xlab("") + ylab("Tip ($)")
```



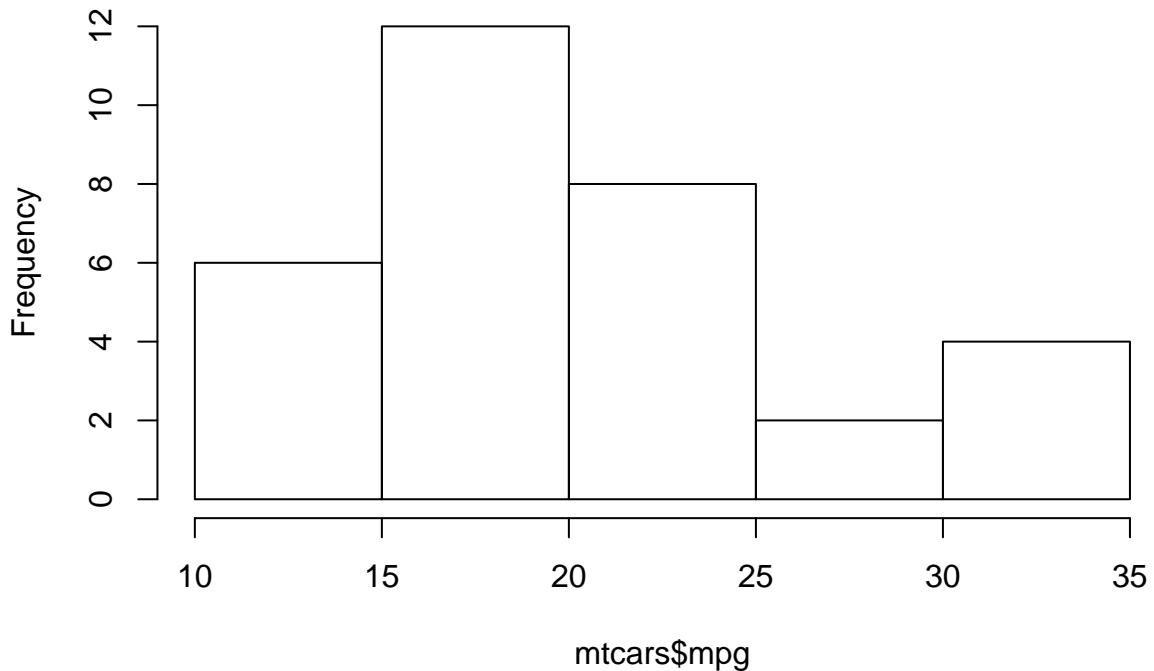
‘Quick and dirty’ (utility) plots

When exploring a dataset, often useful to use built in functions or helpers from other libraries. These help you quickly visualise relationships, but aren’t always *exactly* what you need and can be hard to customise.

5.0.1 Distributions

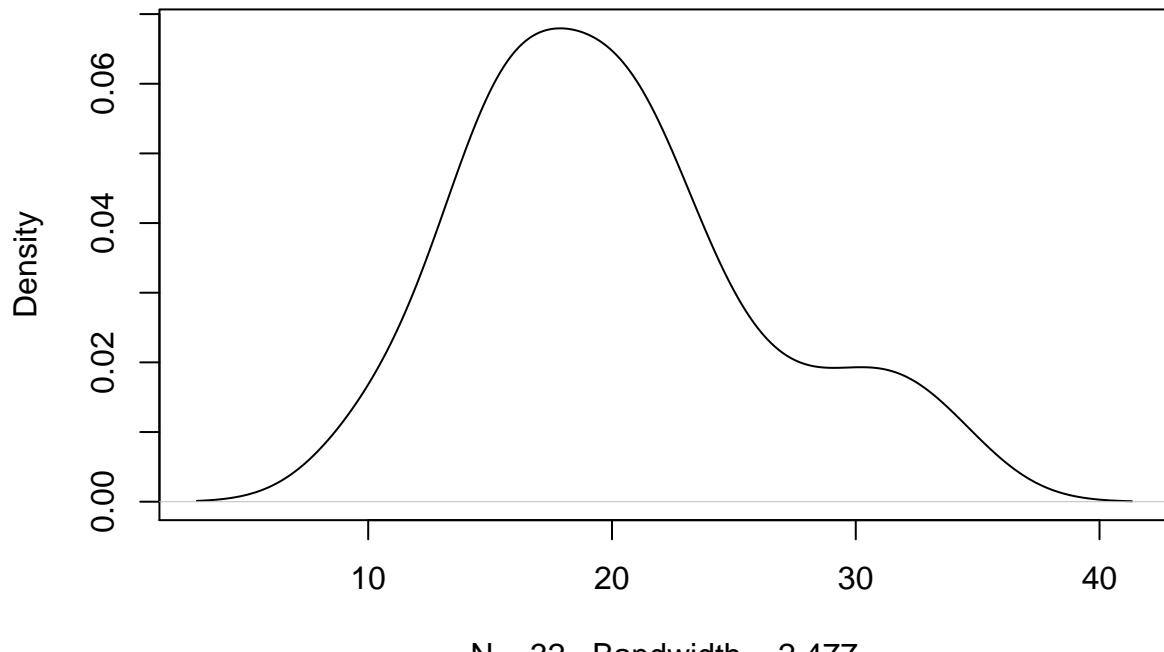
```
hist(mtcars$mpg)
```

Histogram of mtcars\$mpg

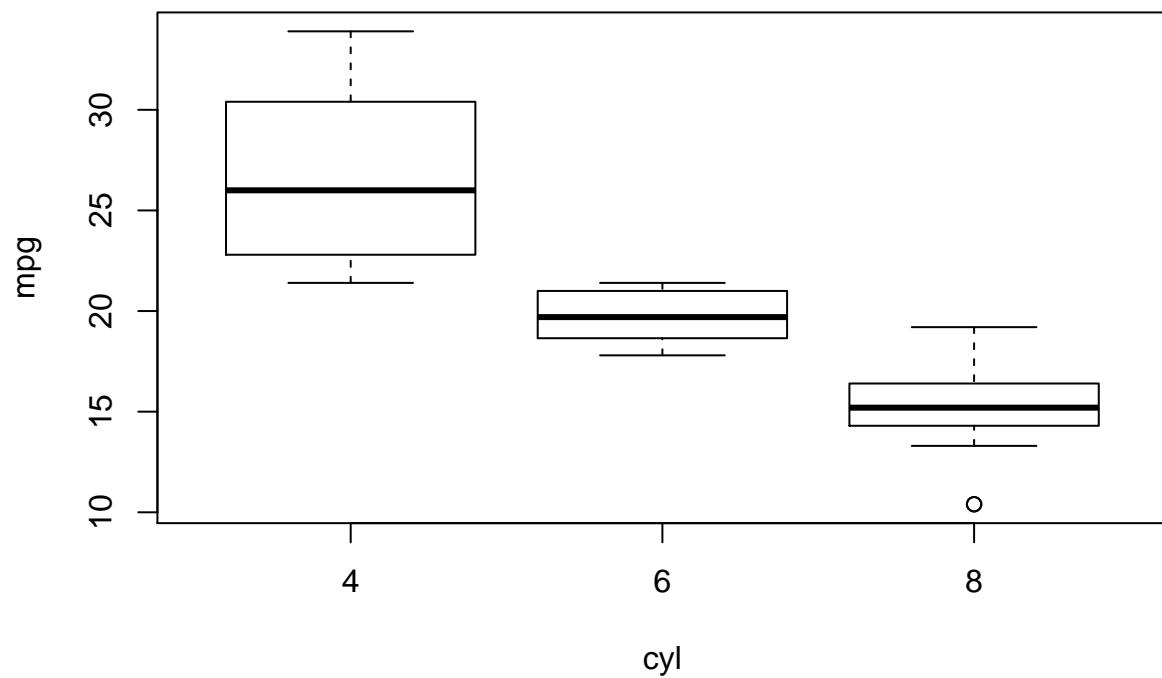


```
plot(density(mtcars$mpg))
```

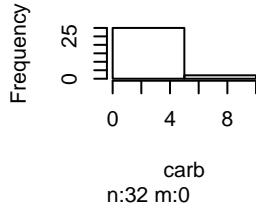
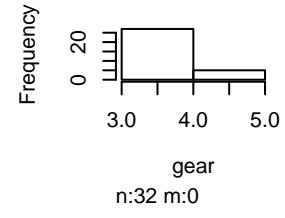
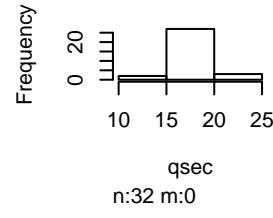
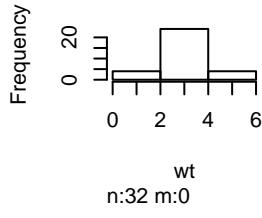
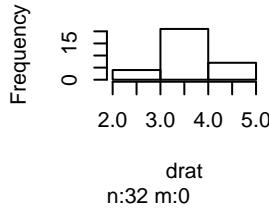
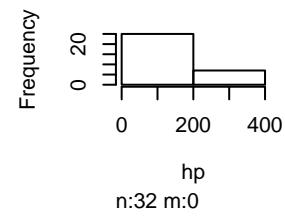
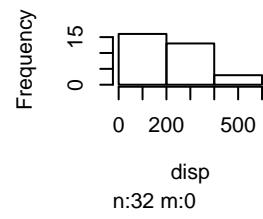
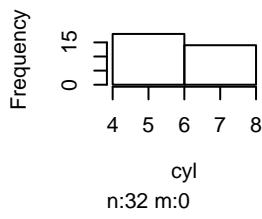
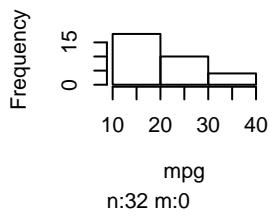
density.default(x = mtcars\$mpg)



```
boxplot(mpg~cyl, data=mtcars)
```

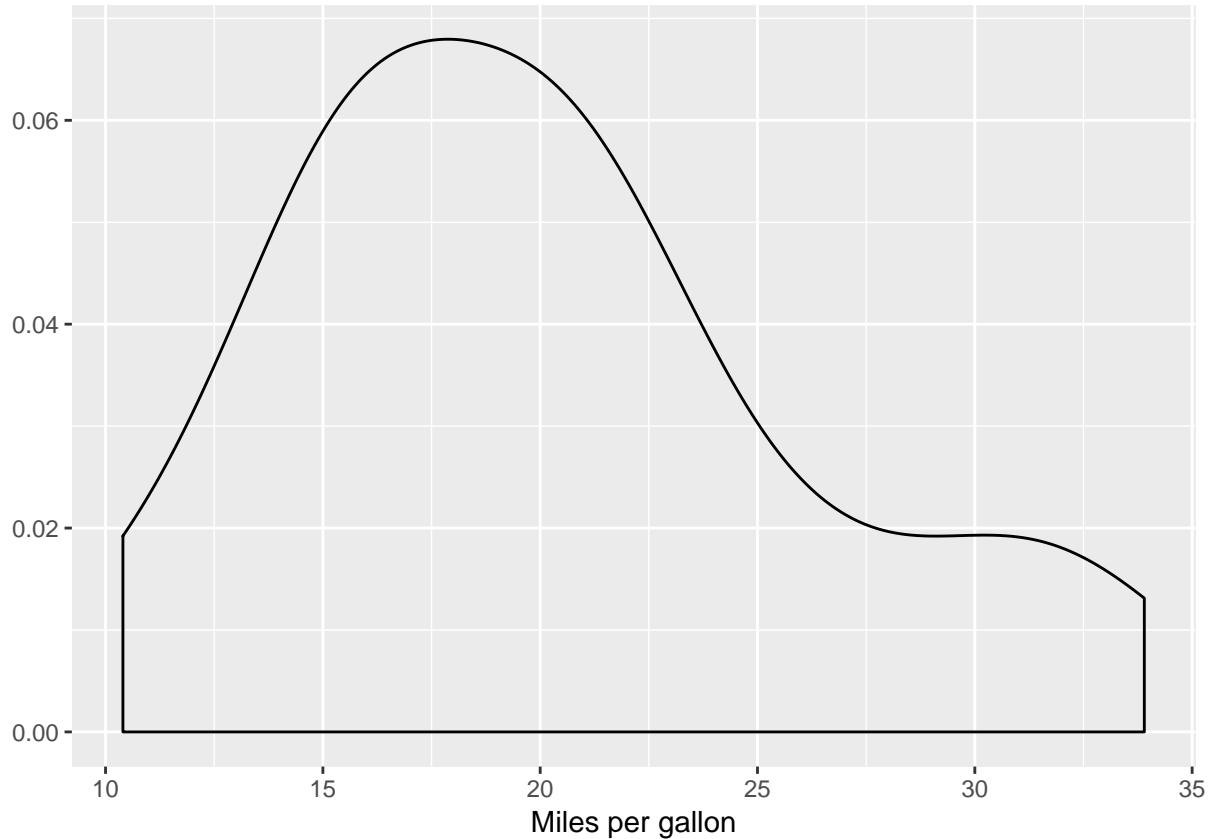


```
Hmisc::hist.data.frame(mtcars)
```

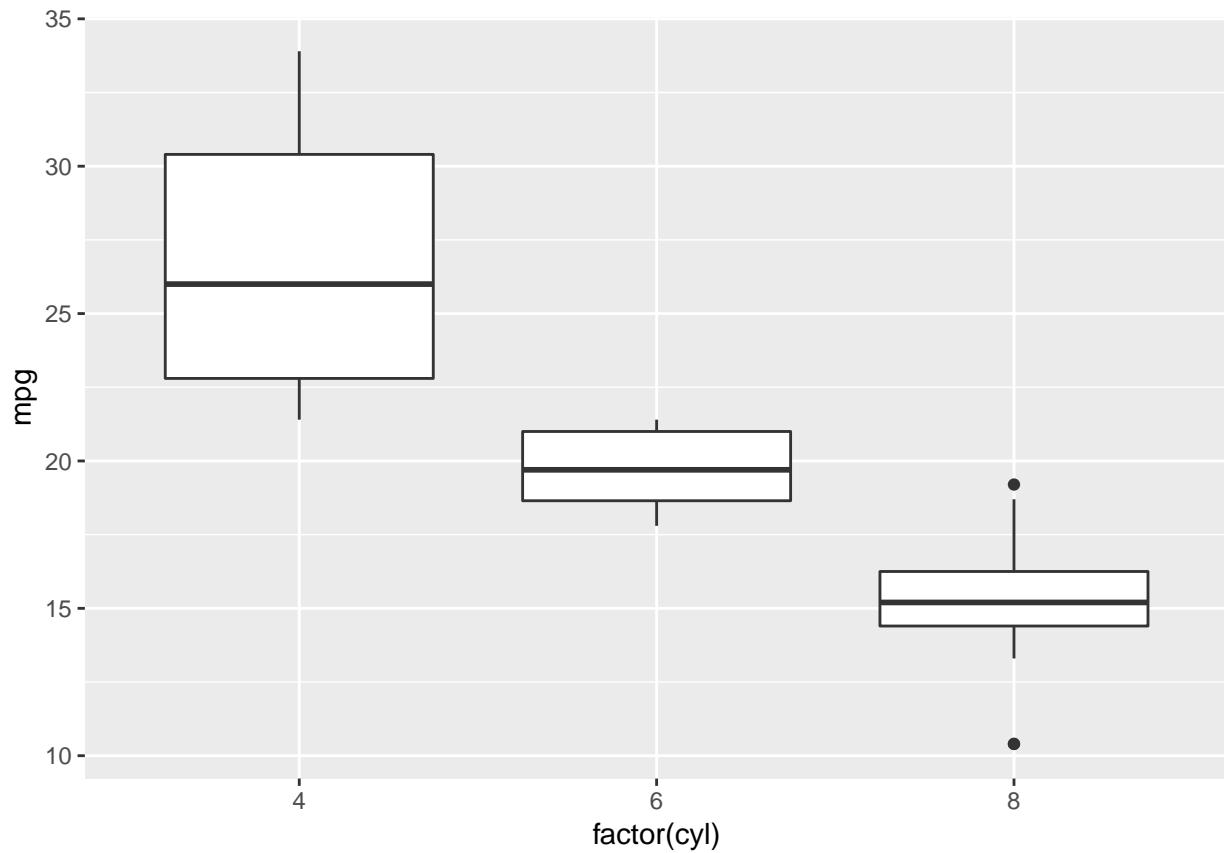


Even for simple plots, ggplot has some useful helper functions though:

```
qplot(mpg, data=mtcars, geom="density") + xlab("Miles per gallon")
```

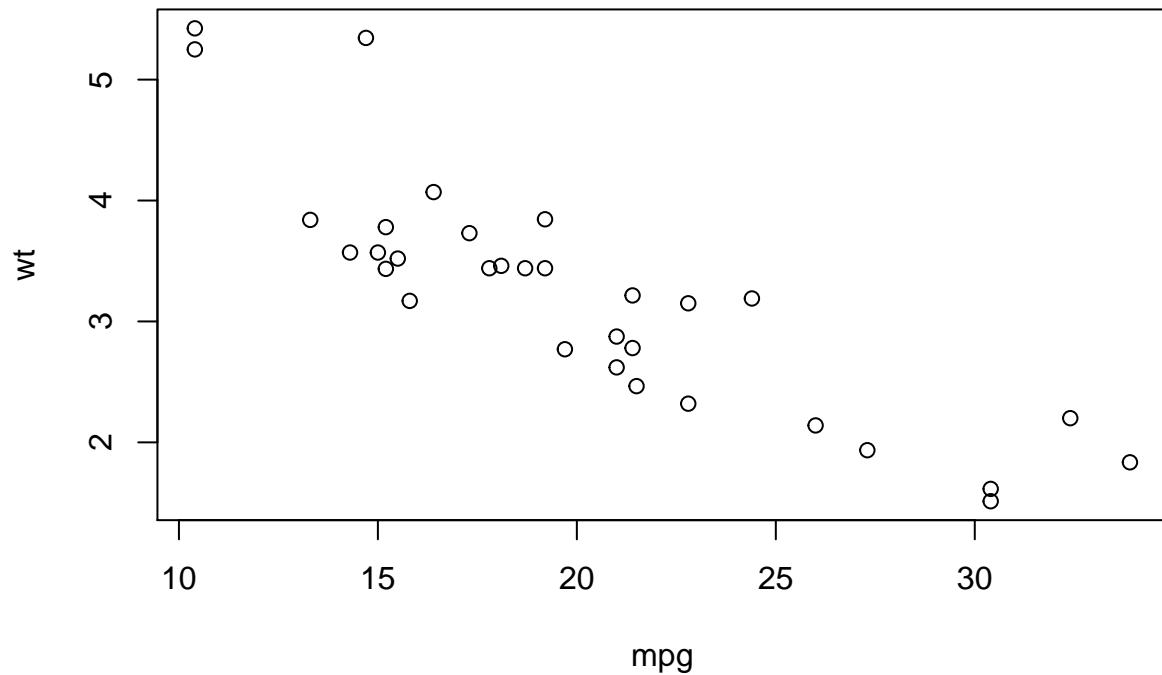


```
qplot(x=factor(cyl), y=mpg, data=mtcars, geom="boxplot")
```

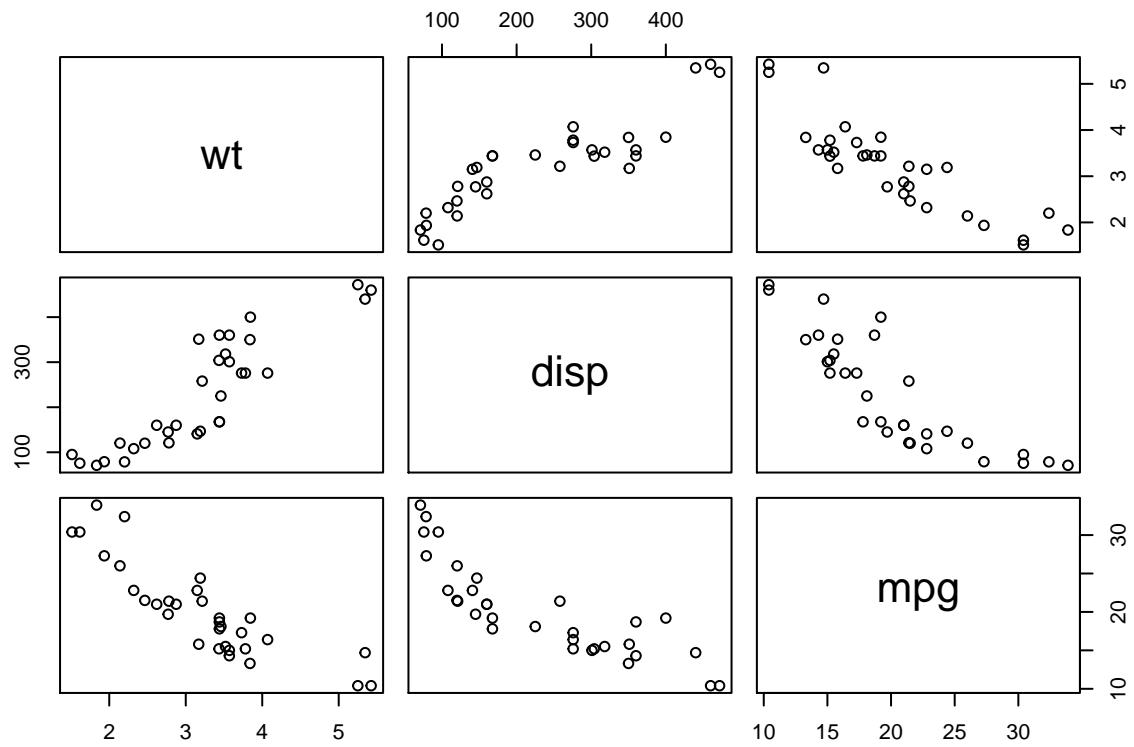


5.0.2 Relationships

```
with(mtcars, plot(mpg, wt))
```

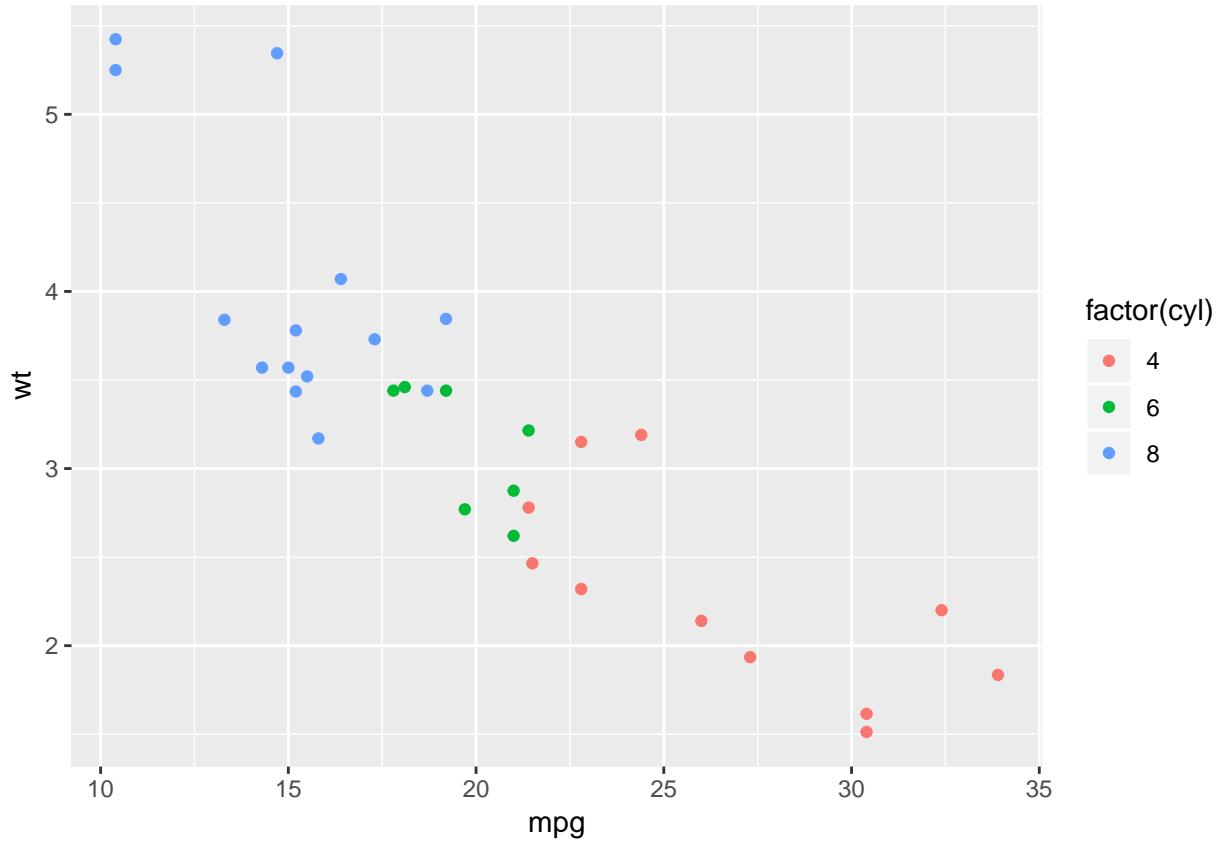


```
pairs(select(mtcars, wt, disp, mpg))
```



Again, for quick plots ggplot also has useful shortcut functions:

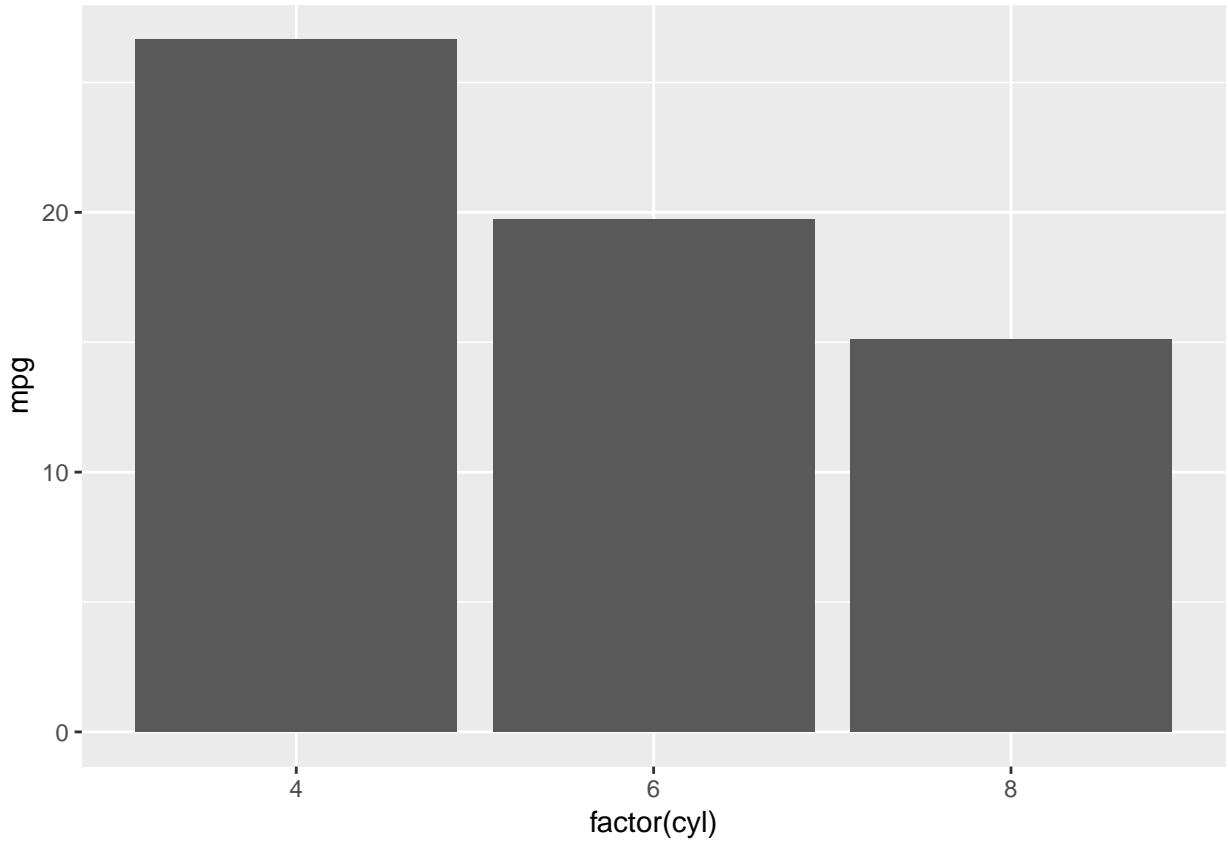
```
qplot(mpg, wt, color=factor(cyl), data = mtcars)
```



5.0.3 Quantities

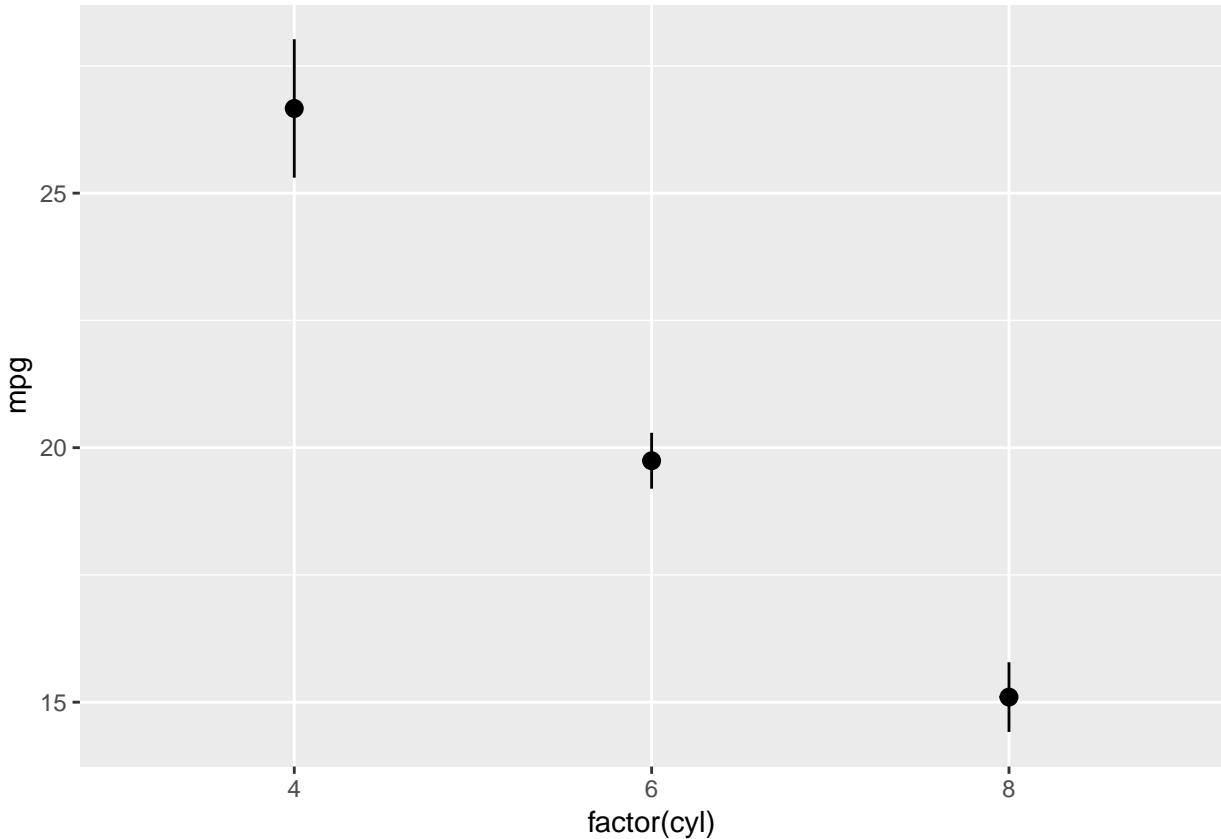
I don't think the base R plots are that convenient here. `ggplot2::` and the `stat_summary()` function makes life much simpler:

```
ggplot(mtcars, aes(factor(cyl), mpg)) +  
  stat_summary(geom="bar")
```



And if you are plotting quantities, as discussed above, showing a range is sensible (a boxplot would also fill both definitions):

```
ggplot(mtcars, aes(factor(cyl), mpg)) +  
  stat_summary(geom="pointrange")
```



Tricks with ggplot

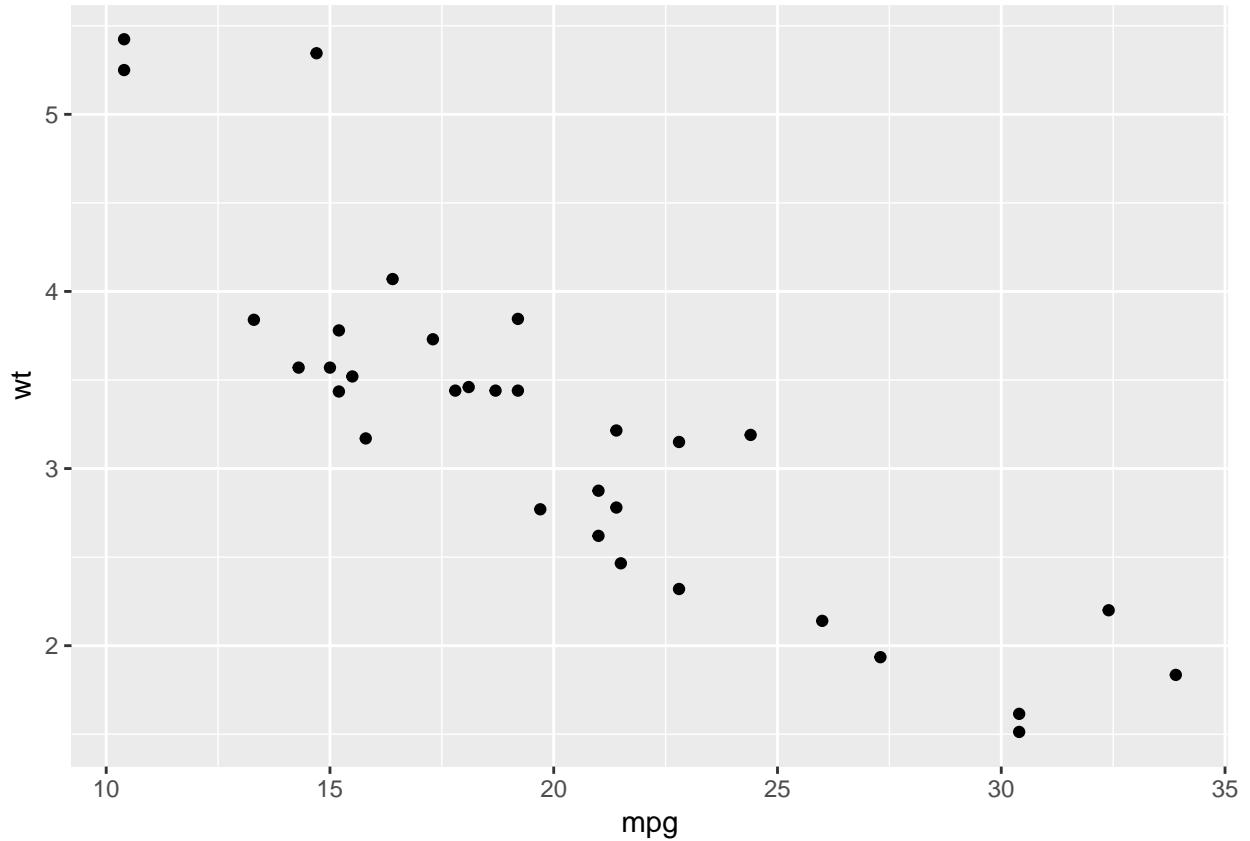
More ways to facet a plot

Facets are ways to repeat a plot for each level of another variable. `ggplot` has two ways of defining and displaying facets:

- As a list of plots, using `facet_wrap`.
- As a grid or matrix of plots, using `facet_grid()`.

Examples of both are shown below, using the following plot as a starting point:

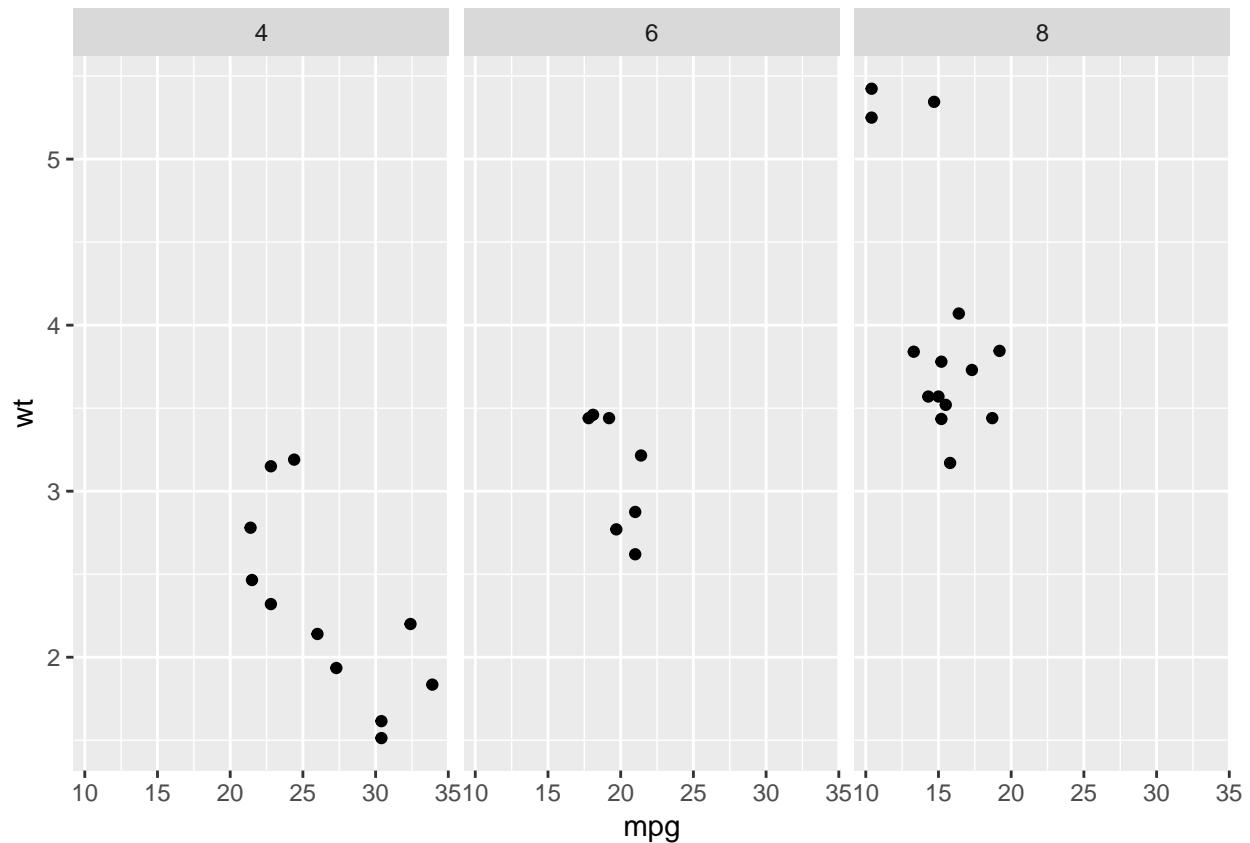
```
base.plot <- ggplot(mtcars, aes(mpg, wt)) + geom_point()  
base.plot
```



```
facet_wrap
```

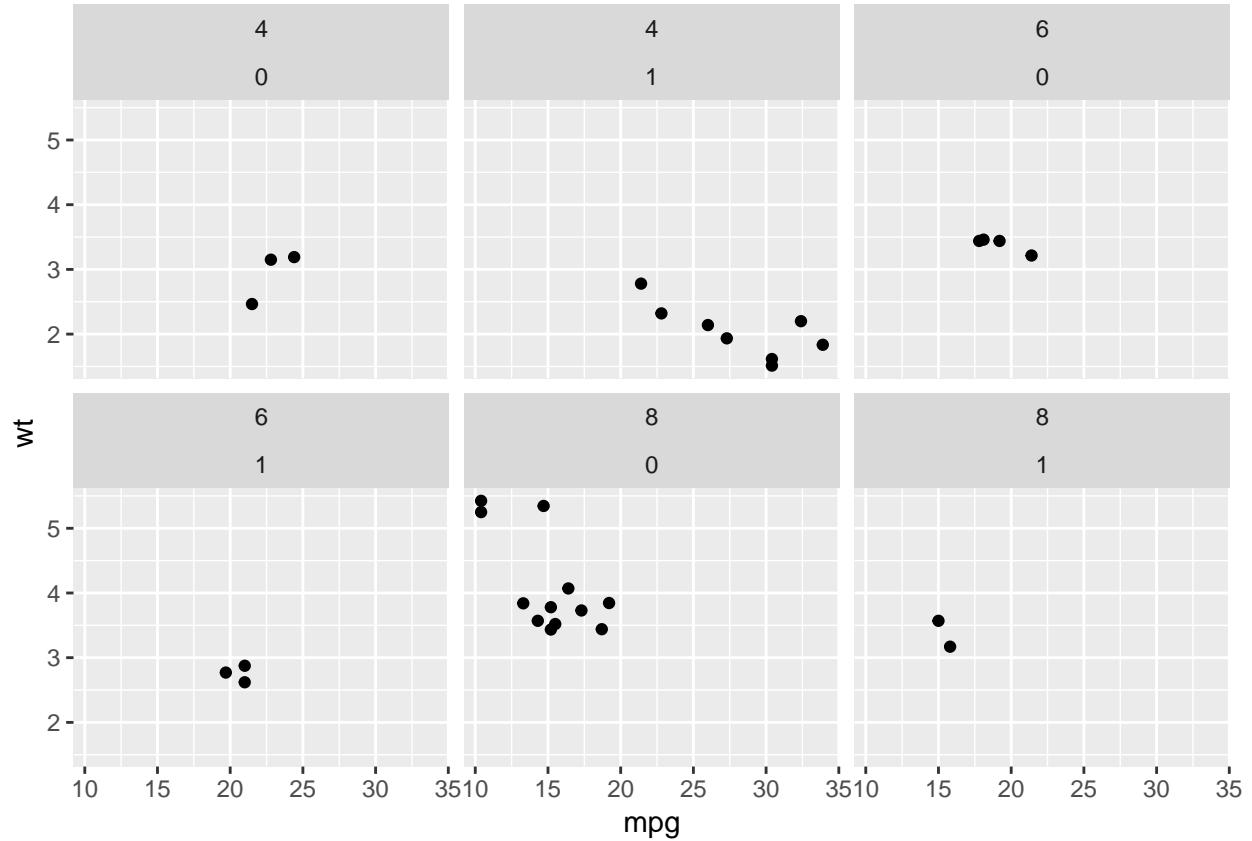
If we want one facet we just type the tilde (~) symbol and then the name of the variable. This is like typing the right hand side of a formula for a regression model:

```
base.plot + facet_wrap(~cyl)
```



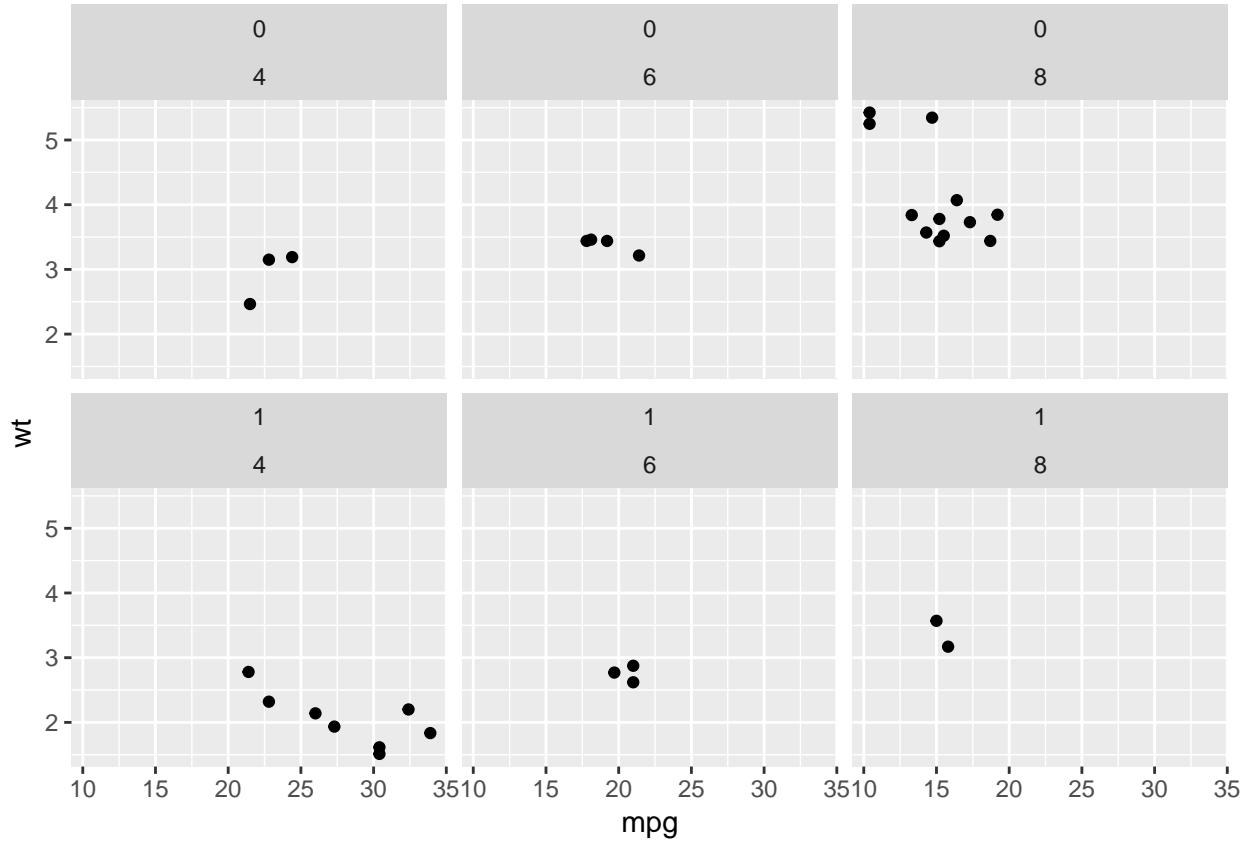
If we want two facets we extend the formula, using the + sign:

```
base.plot + facet_wrap(~cyl+am)
```



Note, the order of variables in the formula makes a difference:

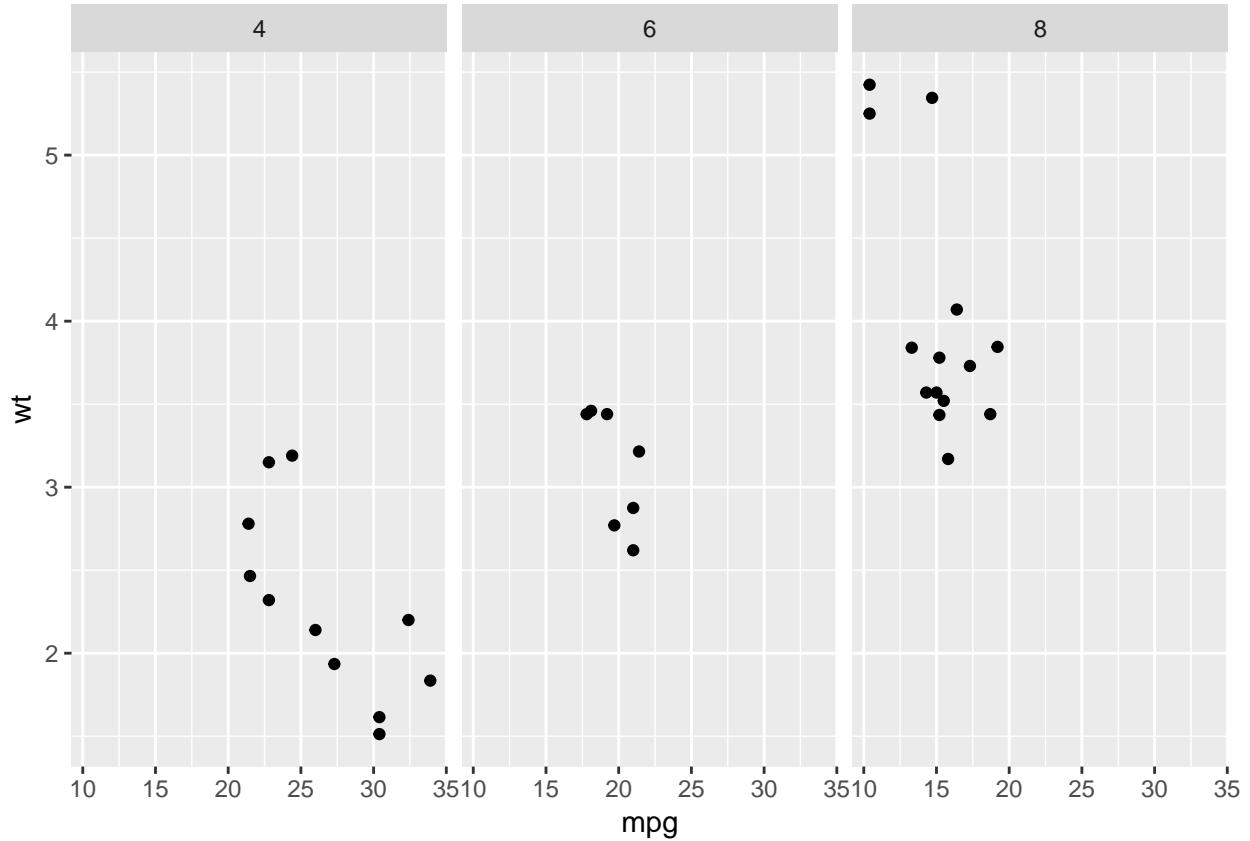
```
base.plot + facet_wrap(~am+cyl)
```



`facet_grid`

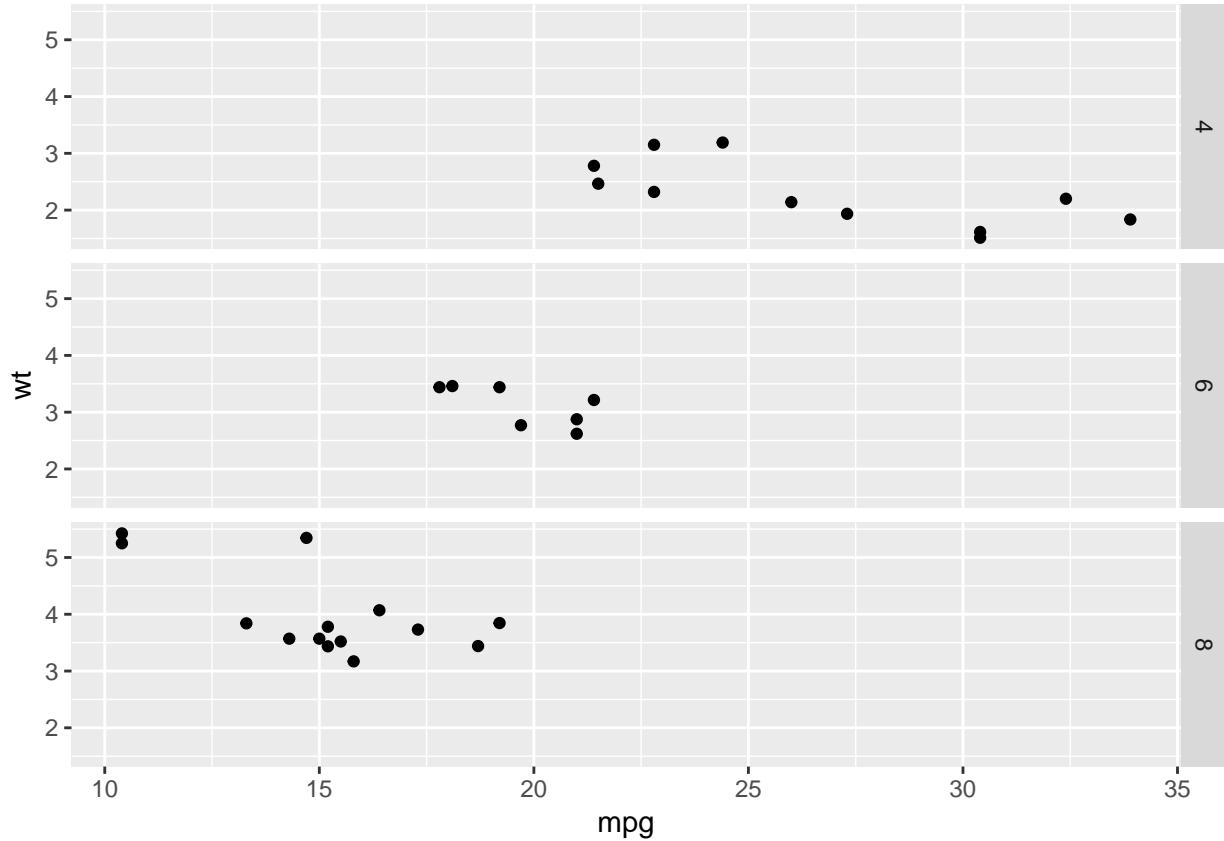
With one variable `facet_grid` produces similar output. Note the `.` (period) on the left hand side of the formula now to make explicit we only have one variable, and we want it on the x axis:

```
base.plot + facet_grid(.~cyl)
```



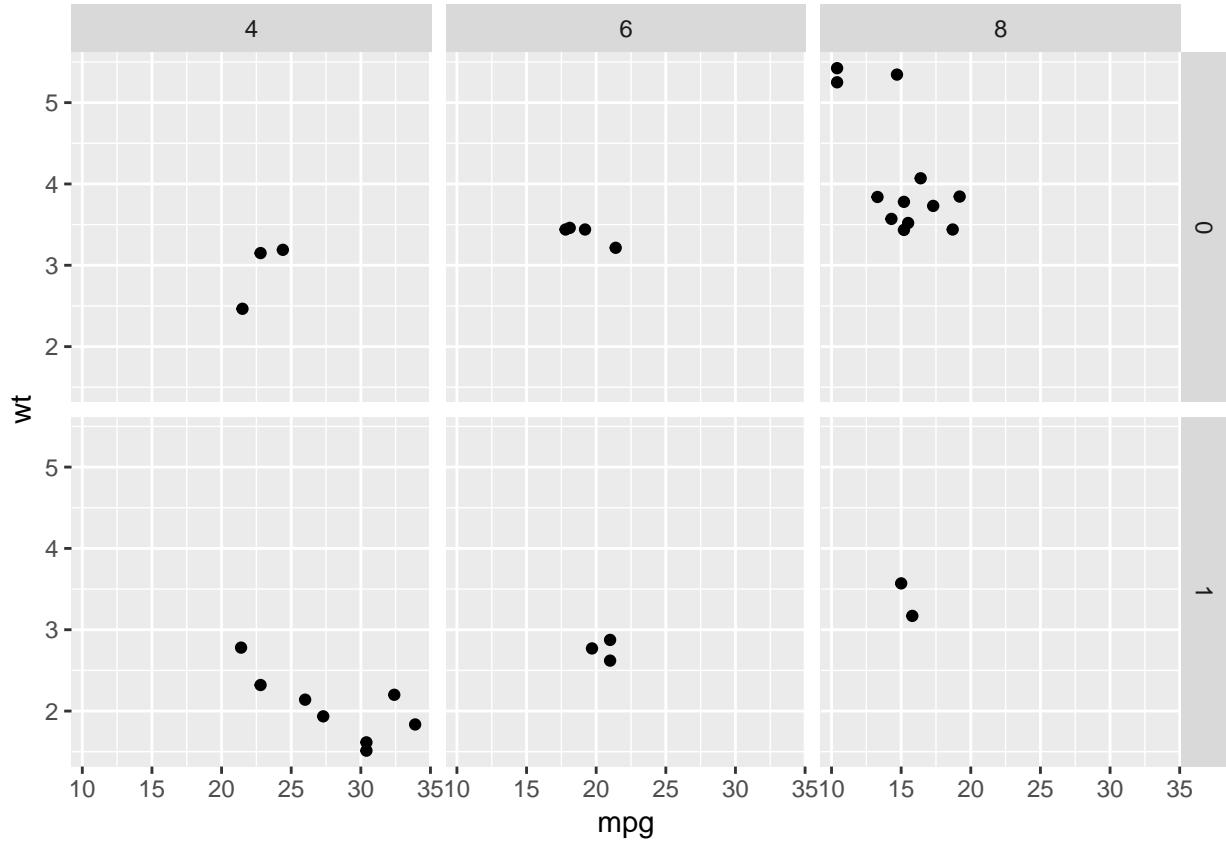
We can flip the facets around by putting the `cyl` variable on the left hand side of the ~:

```
base.plot + facet_grid(cyl~.)
```



And `facet_grid` can also create facets for two or more variables:

```
base.plot + facet_grid(am~cyl)
```



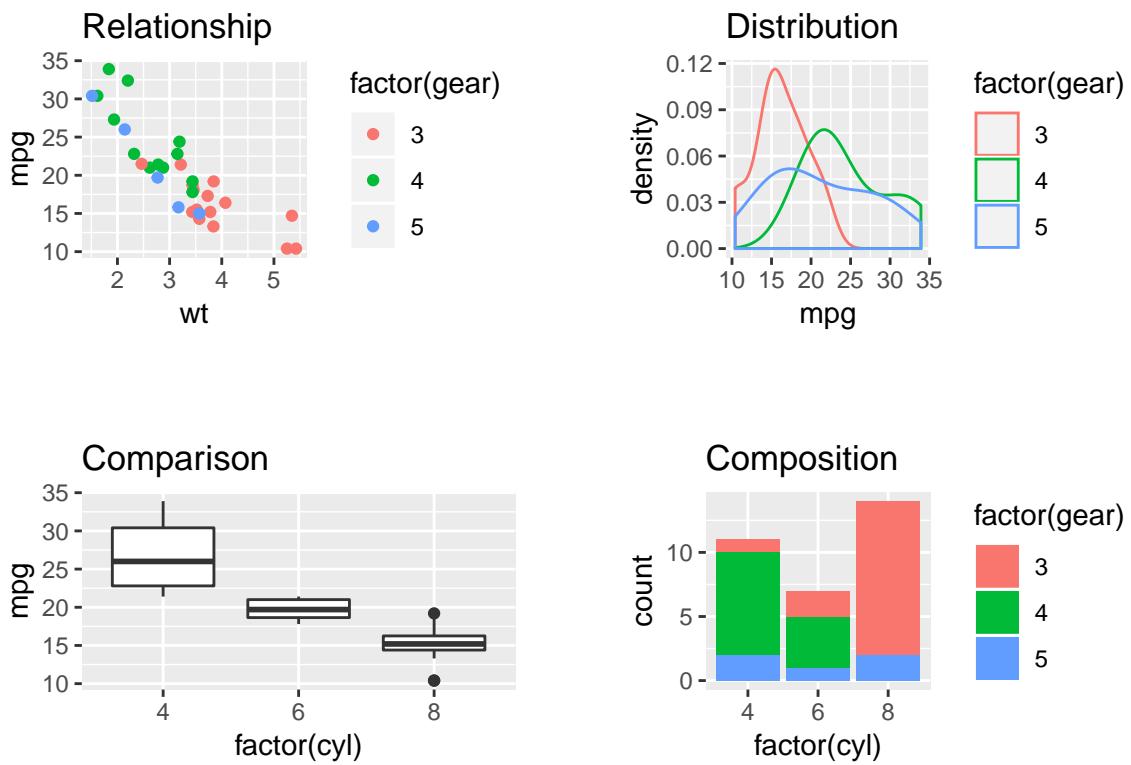
Here the labelling and the arrangement of plots is perhaps nicer because it is clearer that plots for cyl are arranged left to right, and for am they are top to bottom.

Combining separate plots in a grid

Note that combining separate plots in a grid is different from facetting, and it may be you want that instead.

If you really want to combine several plots, the `gridExtra` and `cowplot` packages can be helpful. This is the code from the example in the graphics section, which may be a useful starting point:

```
comparison <- ggplot(mtcars, aes(factor(cyl), mpg)) + geom_boxplot() + ggtitle("Comparison")
relationships <- ggplot(mtcars, aes(wt, mpg, color=factor(gear))) + geom_point() + ggtitle("Relationship")
distributions <- ggplot(mtcars, aes(mpg, color=factor(gear))) + geom_density() + ggtitle("Distribution")
composition <- ggplot(mtcars, aes(factor(cyl), fill = factor(gear))) + geom_bar() + ggtitle("Composition")
mm <- theme(plot.margin=unit(rep(1.5,4), "line"))
gridExtra::grid.arrange(relationships+mm, distributions+mm, comparison+mm, composition+mm, ncol=2)
```



Exporting for print

To export ggplot graphics you can use the `ggsave()` function:

```
ggplot(mtcars, aes(wt, mpg)) + geom_point()
ggsave(filename = "myplot.pdf")
```

See the ggplot docs on exporting or page 323 of the R Graphics Cookbook for lots more detail.

Part III

Models

6 Commonly used statistics

R has simple functions for common inferential statistics like χ^2 , t-tests, correlations and many more. This section is by no means exhaustive, but covers statistics for crosstabulations, differences in means, and linear correlation.

6.1 Non-parametric statistics

This guide is very light on non-parametric statistics in R.

For more on this topic this page on the statmethods site is a useful guide.

The `coin::` package implements many resampling tests, which can also be useful when assumptions of parametric tests are not valid. See this intro to resampling statistics.

Crosstabulations and χ^2

We saw in a previous section how to create a frequency table of one or more variables. Using that previous example, assume we already have a crosstabulation of `age` and `prefers`

```
lego.table
  prefers
age      duplo  lego
  4 years    38   20
  6 years    12   30
```

We can easily run the inferential χ^2 (sometimes spelled “chi”, but pronounced “kai”-squared) test on this table:

```
lego.test <- chisq.test(lego.table)
lego.test

  Pearson's Chi-squared test with Yates' continuity correction

data: lego.table
X-squared = 11.864, df = 1, p-value = 0.0005724
```

Note that we can access each number in this output individually because the `chisq.test` function returns a list. We do this by using the `$` syntax:

```
# access the chi2 value alone
lego.test$statistic
X-squared
  11.86371
```

Even nicer, you can use an R package to write up your results for you in APA format!

```
libraryapa)
apa(lego.test, print_n=T)
[1] "$\\chi^2(1, n = 100) = 11.86, *p* < .001"
```

See more on automatically displaying statistics in APA format

Three-way tables

You can also use `table()` or `xtabs()` to get 3-way tables of frequencies (`xtabs` is probably better for this than `table`).

For example, using the `mtcars` dataset we create a 3-way table, and then convert the result to a dataframe. This means we can print the table nicely in RMarkdown using the `pander.table()` function, or process it further (e.g. by sorting or reshaping it).

```

xtabs(~am+gear+cyl, mtcars) %>%
  as_data_frame() %>%
  pander()
Warning: `as_data_frame()` is deprecated, use `as_tibble()` (but mind the new semantics).
This warning is displayed once per session.

```

am	gear	cyl	n
0	3	4	1
1	3	4	0
0	4	4	2
1	4	4	6
0	5	4	0
1	5	4	2
0	3	6	2
1	3	6	0
0	4	6	2
1	4	6	2
0	5	6	0
1	5	6	1
0	3	8	12
1	3	8	0
0	4	8	0
1	4	8	0
0	5	8	0
1	5	8	2

Often, you will want to present a table in a wider format than this, to aid comparisons between categories. For example, we might want our table to make it easy to compare between US and non-US cars for each different number of cylinders:

```

xtabs(~am+gear+cyl, mtcars) %>%
  as_data_frame() %>%
  reshape2::dcast(am+gear~paste(cyl, "Cylinders")) %>%
  pander()

```

Using n as value column: use value.var to override.

am	gear	4 Cylinders	6 Cylinders	8 Cylinders
0	3	1	2	12
0	4	2	2	0
0	5	0	0	0
1	3	0	0	0
1	4	6	2	0
1	5	2	1	2

Or our primary question might be related to the effect of `am`, in which case we might prefer to include separate columns for US and non-US cars:

```

xtabs(~am+gear+cyl, mtcars) %>%
  as_data_frame() %>%
  reshape2::dcast(gear+cyl~paste0("US=", am)) %>%
  pander()

```

```
Using n as value column: use value.var to override.
```

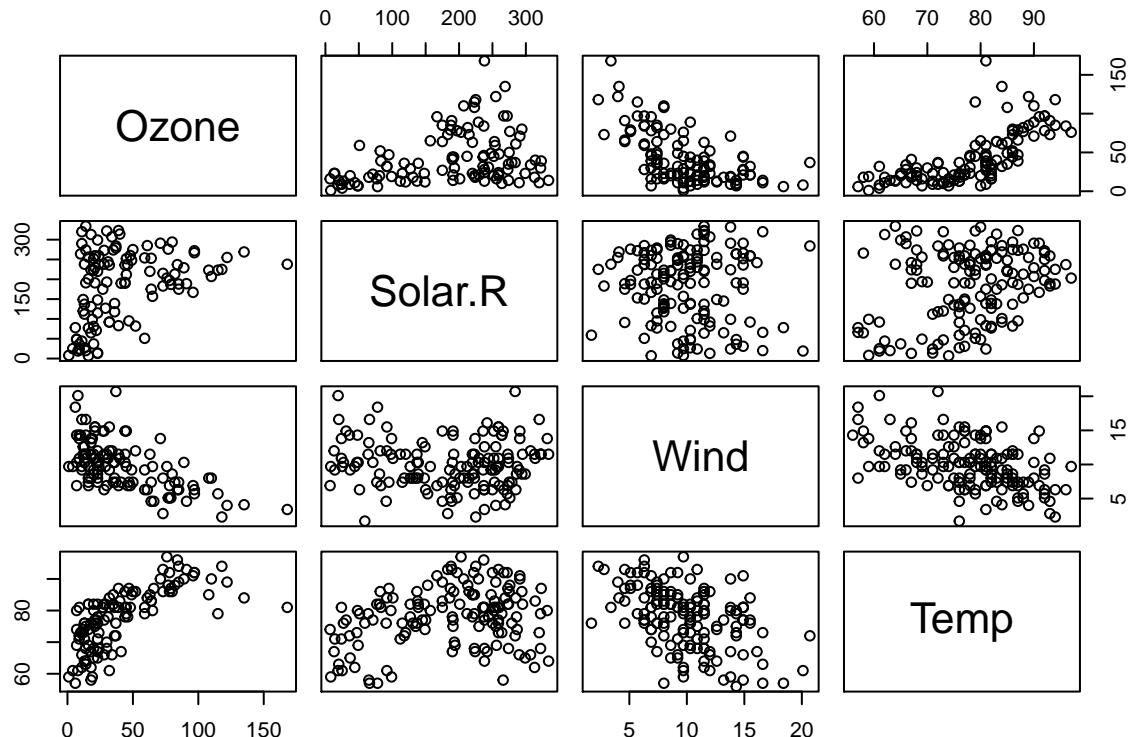
gear	cyl	US=0	US=1
3	4	1	0
3	6	2	0
3	8	12	0
4	4	2	6
4	6	2	2
4	8	0	0
5	4	0	2
5	6	0	1
5	8	0	2

Correlations

The base R `cor()` function provides a simple way to get Pearson correlations, but to get a correlation matrix as you might expect from SPSS or Stata it's best to use the `corr.test()` function in the `psych` package.

Before you start though, plotting the correlations might be the best way of getting to grips with the patterns of relationship in your data. A pairs plot is a nice way of doing this:

```
airquality %>%
  select(-Month, -Day) %>%
  pairs
```



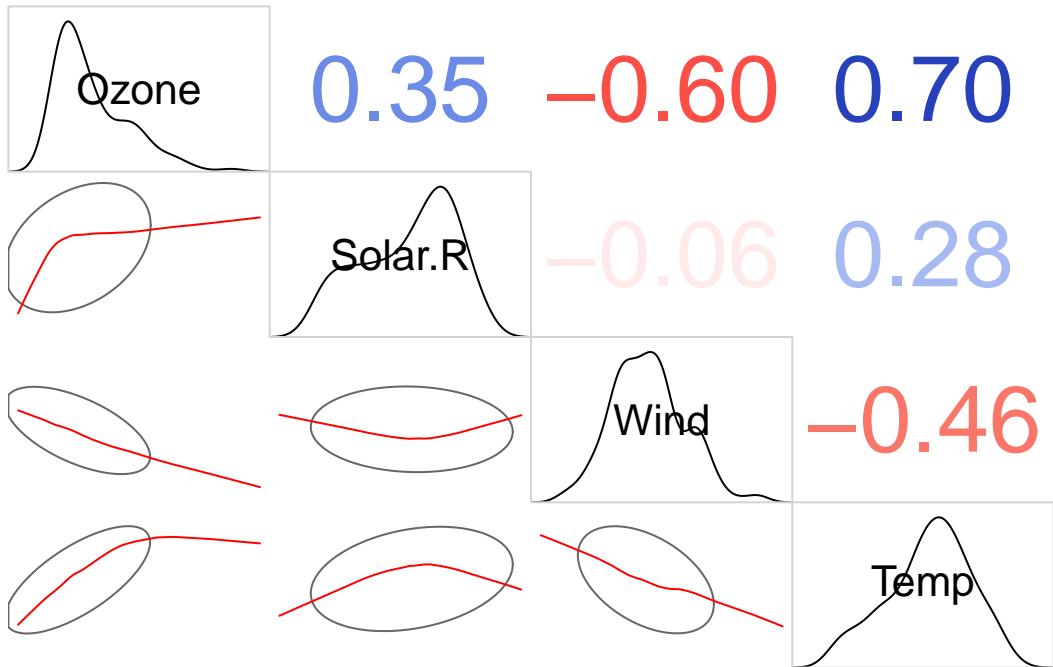


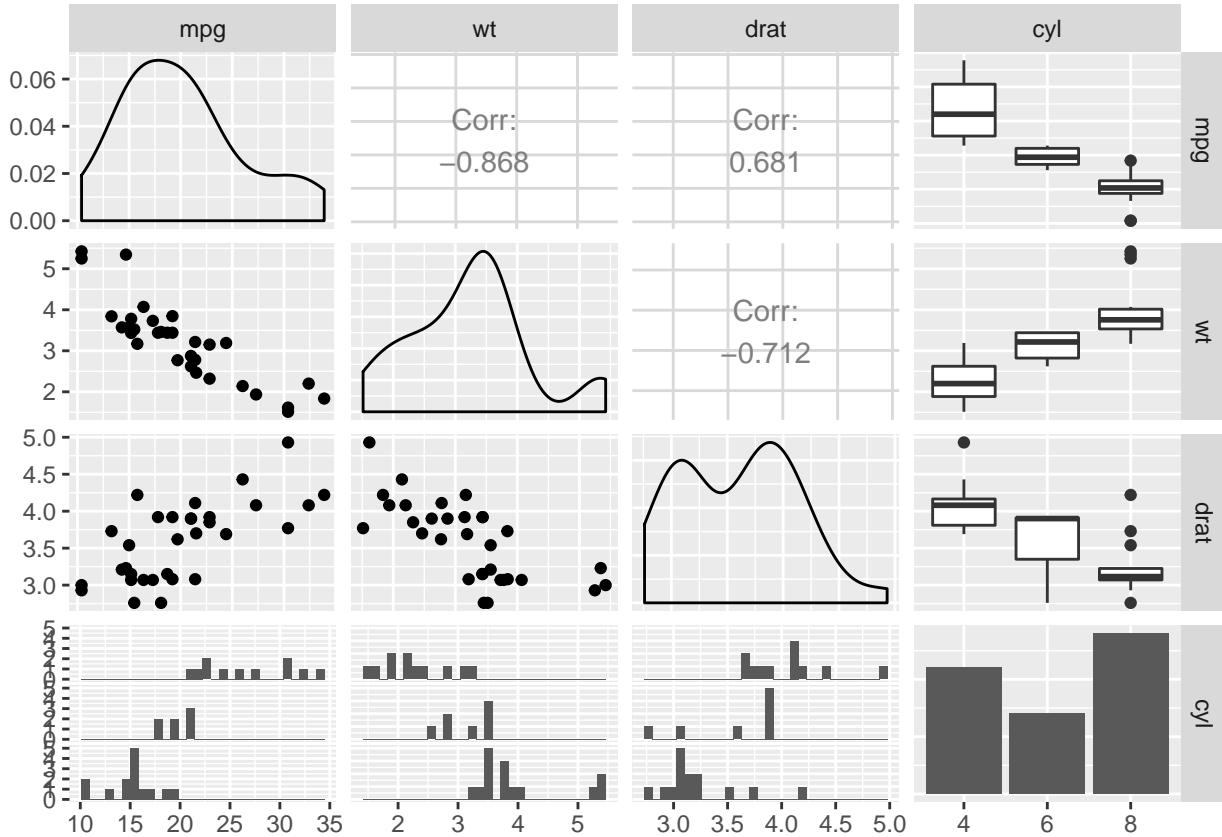
Figure 6: A corrgram, showing pearson correlations (above the diagonal), variable distributions (on the diagonal) and ellipses and smoothed lines of best fit (below the diagonal). Long, narrow ellipses denote large correlations; circular ellipses indicate small correlations.

If we were satisfied the relationships were (reasonably) linear, we could also visualise correlations themselves with a ‘corrgram’, using the `corrgram` library:

```
library("corrgram")
Registered S3 method overwritten by 'seriation'
  method      from
  reorder.hclust gclus
airquality %>%
  select(-Month, -Day) %>%
  corrgram(lower.panel=corrgram::panel.ellipse,
            upper.panel=panel.cor,
            diag.panel=panel.density)
```

The `ggpairs` function from the `GGally` package is also a nice way of plotting relationships between a combination of categorical and continuous data - it packs a lot of information into a limited space:

```
mtcars %>%  
  mutate(cyl = factor(cyl)) %>%  
  select(mpg, wt, drat, cyl) %>%  
  GGally::ggpairs()
```



Creating a correlation matrix

The `psych::corr.test()` function is a quick way to obtain a pairwise correlation matrix for an entire dataset, along with p values and confidence intervals which the base R `cor()` function will not provide:

```
mycorrelations <- psych::corr.test(airquality)
mycorrelations
Call:psych::corr.test(x = airquality)
Correlation matrix
      Ozone Solar.R Wind Temp Month Day
Ozone   1.00   0.35 -0.60  0.70  0.16 -0.01
Solar.R 0.35   1.00 -0.06  0.28 -0.08 -0.15
Wind    -0.60  -0.06  1.00 -0.46 -0.18  0.03
Temp    0.70   0.28 -0.46  1.00  0.42 -0.13
Month   0.16   -0.08 -0.18  0.42  1.00 -0.01
Day     -0.01  -0.15  0.03 -0.13 -0.01  1.00
Sample Size
      Ozone Solar.R Wind Temp Month Day
Ozone   116    111   116   116   116  116
Solar.R 111    146   146   146   146  146
Wind    116    146   153   153   153  153
Temp    116    146   153   153   153  153
Month   116    146   153   153   153  153
Day     116    146   153   153   153  153
Probability values (Entries above the diagonal are adjusted for multiple tests.)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
Ozone	0.00	0.00	0.00	0.00	0.56	1.00
Solar.R	0.00	0.00	1.00	0.01	1.00	0.56
Wind	0.00	0.50	0.00	0.00	0.25	1.00
Temp	0.00	0.00	0.00	0.00	0.00	0.65
Month	0.08	0.37	0.03	0.00	0.00	1.00
Day	0.89	0.07	0.74	0.11	0.92	0.00

To see confidence intervals of the correlations, print with the short=FALSE option

One thing to be aware of is that by default `corr.test()` produces p values that are adjusted for multiple comparisons in the top right hand triangle (i.e. above the diagonal). If you want the uncorrected values use the values below the diagonal (or pass `adjust=FALSE` when calling the function).

Working with correlation matrices

It's important to realise that, as with all R objects, we can work with correlation matrices to continue our data analyses.

For example, as part of exploring your data, you might want to know whether correlations you observe in one sample are similar to those from another sample, when using the same questions. For example, let's say we ran a survey measuring variables from the theory of planned behaviour first in students, and later in older adults:

We could run correlations for each sample separately:

```
corr.students <- cor(students)
corr.public <- cor(public)
```

And we could 'eyeball' both of these correlation matrices and try and spot patterns or differences between them, but this is quite hard:

```
corr.students %>%
  pander()
```

	behaviour	intention	control	social.norm	attitude
behaviour	1.000	0.55	0.648	0.072	0.132
intention	0.551	1.00	0.394	0.298	0.438
control	0.648	0.39	1.000	0.017	0.014
social.norm	0.072	0.30	0.017	1.000	-0.011
attitude	0.132	0.44	0.014	-0.011	1.000

```
corr.public %>%
  pander()
```

	behaviour	intention	social.norm	attitude	control
behaviour	1.00	0.54	0.287	0.143	0.23
intention	0.54	1.00	0.361	0.290	0.37
social.norm	0.29	0.36	1.000	0.019	-0.08
attitude	0.14	0.29	0.019	1.000	0.04
control	0.23	0.37	-0.080	0.040	1.00

But we could also simply *subtract* one matrix from the other to show the difference directly:

```
(corr.students - corr.public) %>%
  pander()
```

	behaviour	intention	control	social.norm	attitude
behaviour	0.000	0.013	0.361	-0.071	-0.093
intention	0.013	0.000	0.033	0.008	0.067
control	0.361	0.033	0.000	-0.002	0.093
social.norm	-0.071	0.008	-0.002	0.000	-0.050
attitude	-0.093	0.067	0.093	-0.050	0.000

Now it's much more obvious that the behaviour/control correlation differs between the samples (it's higher in the students).

The point here is not that this is an analysis you are likely to actually report — although you might find it useful when exploring the data and interpreting your findings.

But rather this shows that a correlation matrix, in common with the results of all the statistical tests we run, are themselves *just data points*. We can do whatever we like with our results — storing them in data frames to display later, or process as we need.

In reality, if you wanted to test the difference in correlations (slopes) in two groups for one outcome variable you probably want to use multiple regression, and if you wanted to test a complex model like the theory of planned behaviour, you might consider CFA and/or SEM).

Tables for publication

6.1.0.1 Using apaTables

If you want to produce nice correlation tables for publication the `apaTables` package might be useful. This block saves an APA formatted correlation table to an external Word document like this.

Note though, that the APA table format does encourage ‘star gazing’ to some degree. Try to avoid interpreting correlation tables solely based on the significance (or not) of the *r* values. The `pairs` or `corrgram` plots shown above are a much better summary of the data, and are can be just as compact.

```
library(apaTables)
apa.cor.table(airquality, filename="Table1_APA.doc", show.conf.interval=F)
The ability to suppress reporting of reporting confidence intervals has been deprecated in this version
The function argument show.conf.interval will be removed in a later version.
```

Means, standard deviations, and correlations with confidence intervals

Variable	M	SD	1	2	3
1. Ozone	42.13	32.99			
2. Solar.R	185.93	90.06	.35**		
			[.17, .50]		
3. Wind	9.96	3.52	-.60**	-.06	
			[-.71, -.47]	[-.22, .11]	

4.	Temp	77.88	9.47	.70**	.28**	-.46**
				[.59, .78]	[.12, .42]	[-.57, -.32]
5.	Month	6.99	1.42	.16	-.08	-.18*
				[-.02, .34]	[-.23, .09]	[-.33, -.02]

6.	Day	15.80	8.86	-.01	-.15	.03
				[-.20, .17]	[-.31, .01]	[-.13, .19]

4 5

.42**
[.28, .54]

-.13 -.01
[-.28, .03] [-.17, .15]

Note. M and SD are used to represent mean and standard deviation, respectively.

Values in square brackets indicate the 95% confidence interval.

The confidence interval is a plausible range of population correlations that could have caused the sample correlation (Cumming, 2014).

* indicates $p < .05$. ** indicates $p < .01$.

6.1.0.2 By hand

If you're not bothered about strict APA format, you might still want to extract r and p values as dataframes which can then be saved to a csv and opened in Excel, or converted to a table some other way.

You can do this by storing the `corr.test` output in a variable, and then accessing the `$r` and `$p` values within it.

First, we create the `corr.test` object:

```
mycorrelations <- psych::corr.test(airquality)
```

Then extract the r values as a table:

```
mycorrelations$r %>%
  pander()
```

	Ozone	Solar.R	Wind	Temp	Month	Day
Ozone	1.000	0.348	-0.602	0.70	0.165	-0.013
Solar.R	0.348	1.000	-0.057	0.28	-0.075	-0.150

	Ozone	Solar.R	Wind	Temp	Month	Day
Wind	-0.602	-0.057	1.000	-0.46	-0.178	0.027
Temp	0.698	0.276	-0.458	1.00	0.421	-0.131
Month	0.165	-0.075	-0.178	0.42	1.000	-0.008
Day	-0.013	-0.150	0.027	-0.13	-0.008	1.000

And we can also extract p values:

```
mycorrelations$p %>%
  pander()
```

	Ozone	Solar.R	Wind	Temp	Month	Day
Ozone	0.000	0.002	0.000	0.000	0.56	1.00
Solar.R	0.000	0.000	1.000	0.008	1.00	0.56
Wind	0.000	0.496	0.000	0.000	0.25	1.00
Temp	0.000	0.001	0.000	0.000	0.00	0.65
Month	0.078	0.366	0.027	0.000	0.00	1.00
Day	0.888	0.070	0.739	0.108	0.92	0.00

Saving as a .csv is the same as for other dataframes:

```
write.csv(mycorrelations$r, file="airquality-r-values.csv")
```

And can also access the CI for each pairwise correlation as a table:

```
mycorrelations$ci %>%
  head() %>%
  pander(caption="First 6 rows of the table of CI's for the correlation matrix.")
```

Table 32: First 6 rows of the table of CI's for the correlation matrix.

	lower	r	upper	p
Ozone-Slr.R	0.173	0.348	0.50	0.000
Ozone-Wind	-0.706	-0.602	-0.47	0.000
Ozone-Temp	0.591	0.698	0.78	0.000
Ozone-Month	-0.018	0.165	0.34	0.078
Ozone-Day	-0.195	-0.013	0.17	0.888
Slr.R-Wind	-0.217	-0.057	0.11	0.496

Other methods for correlation

By default `corr.test` produces Pearson correlations, but You can pass the `method` argument `psych::corr.test()`:

```
psych::corr.test(airquality, method="spearman")
psych::corr.test(airquality, method="kendall")
```

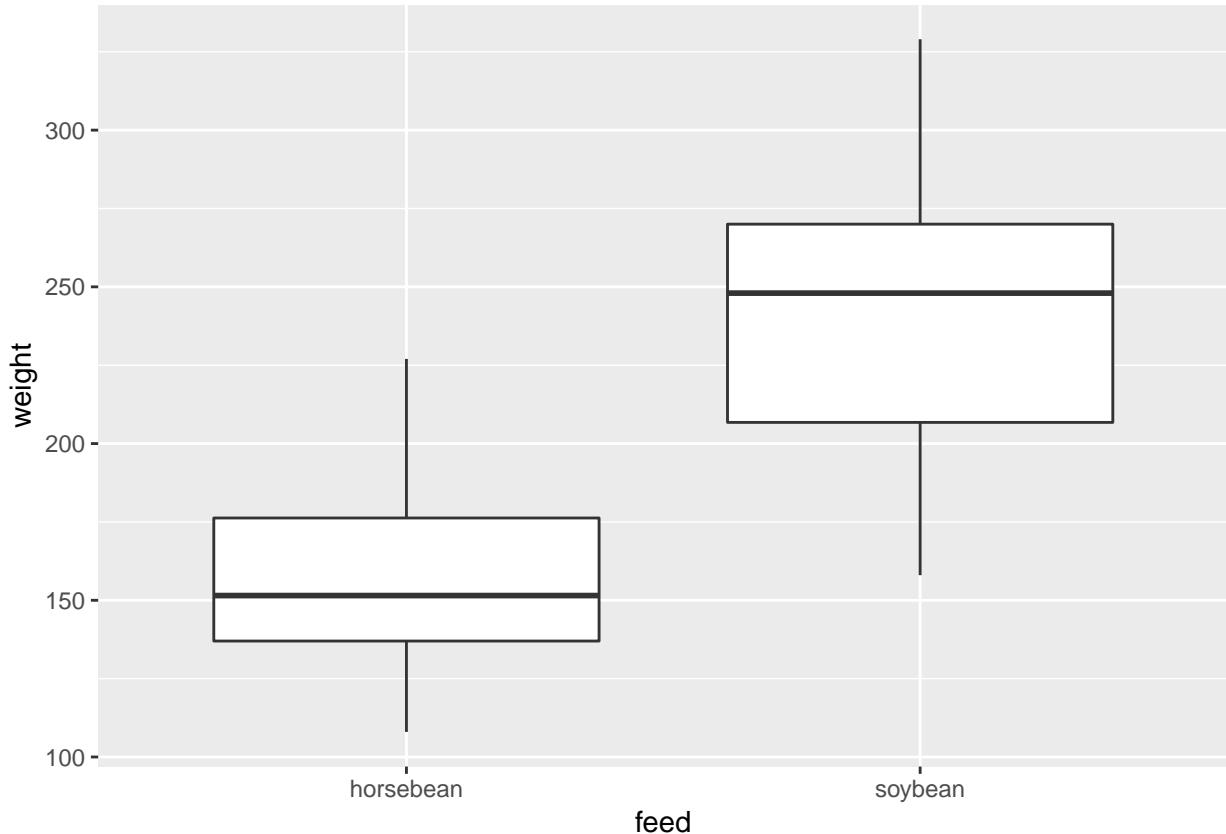


Figure 7: The box in a boxplot indicates the IQR; the whisker indicates the min/max values or 1.5 times the IQR, whichever is the smaller. If there are outliers beyond 1.5 times the IQR then they are shown as points.

t-tests

Visualising your data first

Before you run any tests it's worth plotting your data.

Assuming you have a continuous outcome and categorical (binary) predictor (here we use a subset of the built in `chickwts` data), a boxplot can work well:

```
chicks.eating.beans <- chickwts %>%
  filter(feed %in% c("horsebean", "soybean"))

chicks.eating.beans %>%
  ggplot(aes(feed, weight)) +
  geom_boxplot()
```

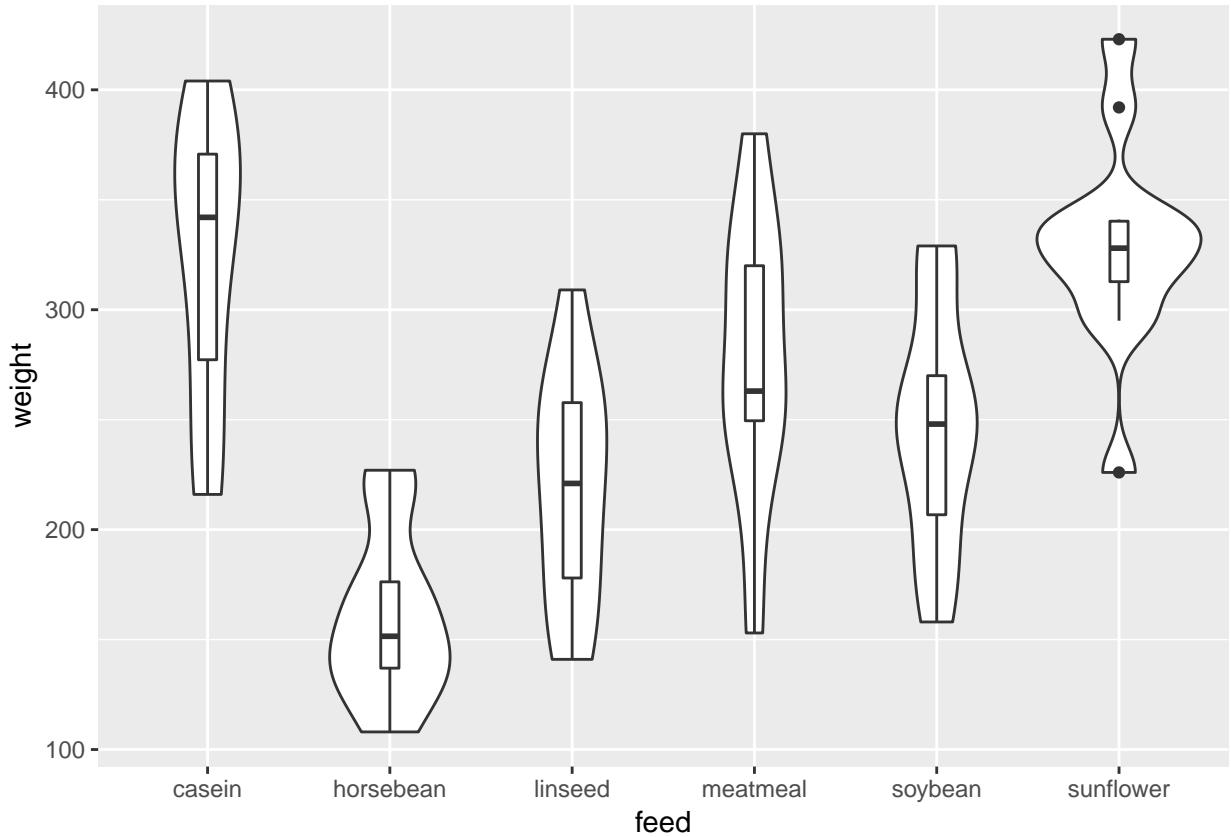
Or a violin or bottle plot, which shows the distributions within each group and makes it relatively easy to check some of the main assumptions of the test:

```
chicks.eating.beans %>%
  ggplot(aes(feed, weight)) +
  geom_violin()
```



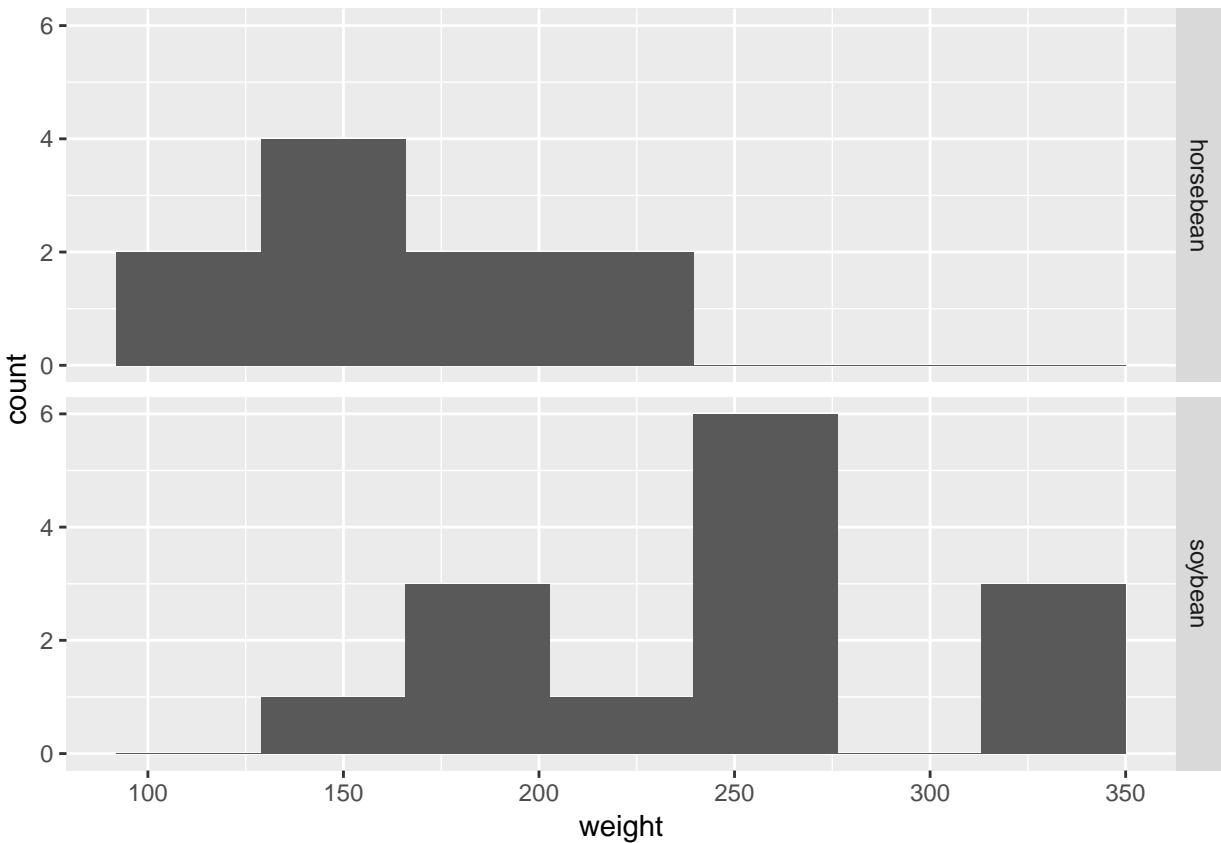
Layering boxes and bottles can work well too because it combines information about the distribution with key statistics like the median and IQR, and also because it scales reasonably well to multiple categories:

```
chickwts %>%
  ggplot(aes(feed, weight)) +
  geom_violin() +
  geom_boxplot(width=.1)
```



And density plots are just smoothed histograms (which you might prefer if you're a fan of 80's computer games):

```
chicks.eating.beans %>%
  ggplot(aes(weight)) +
  geom_histogram(bins=7) +
  facet_grid(feed ~ .)
```



->

Running a t-test

Assuming you really do still want to run a null hypothesis test on one or two means, the `t.test()` function performs most common variants, illustrated below.

6.1.0.2.1 2 independent groups

Assuming your data are in long format:

```
t.test(weight ~ feed, data=chicks.eating.beans)

Welch Two Sample t-test

data: weight by feed
t = -4.5543, df = 21.995, p-value = 0.0001559
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-125.49476 -46.96238
sample estimates:
mean in group horsebean   mean in group soybean
          160.2000            246.4286
```

Or equivalently, if your data are untidy and each group has it's own column (e.g. chicks eating soybeans in one column and those eating horsebeans in another):

```

with(untidy.chicks, t.test(horsebean, soybean))

Welch Two Sample t-test

data: horsebean and soybean
t = -4.5543, df = 21.995, p-value = 0.0001559
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-125.49476 -46.96238
sample estimates:
mean of x mean of y
160.2000 246.4286

```

6.1.0.2.2 Equal or unequal variances?

By default R assumes your groups have unequal variances and applies an appropriate correction (you will notice the output labelled ‘Welch Two Sample t-test’).

You can turn this correction off (for example, if you’re trying to replicate an analysis done using the default settings in SPSS) but you probably do want to assume unequal variances [see Ruxton, 2006].

6.1.0.2.3 Paired samples

If you have repeated measures on a sample you need a paired samples test.

```

# simulate paired samples in pre-post design
set.seed(1234)
baseline <- rnorm(50, 2.5, 1)
followup = baseline + rnorm(50, .5, 1)

# run paired samples test
t.test(baseline, followup, paired=TRUE)

Paired t-test

data: baseline and followup
t = -4.36, df = 49, p-value = 6.661e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-0.9342988 -0.3447602
sample estimates:
mean of the differences
-0.6395295

```

Note that we could also ‘melt’ the data into long format and use the paired=TRUE argument with a formula:

```

long.form.data <- data_frame(baseline=baseline, follow=followup) %>%
  reshape2::melt()
Warning: `data_frame()` is deprecated, use `tibble()` .
This warning is displayed once per session.
No id variables; using all as measure variables

with(long.form.data, t.test(value~variable, paired=TRUE))

Paired t-test

```

```

data: value by variable
t = -4.36, df = 49, p-value = 6.661e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-0.9342988 -0.3447602
sample estimates:
mean of the differences
-0.6395295

```

6.1.0.2.4 One-sample test

Sometimes you might want to compare a sample mean with a specific value:

```

# test if mean of `outcome` variable is different from 2
set.seed(1234)
test.scores <- rnorm(50, 2.5, 1)
t.test(test.scores, mu=2)

```

One Sample t-test

```

data: test.scores
t = 0.37508, df = 49, p-value = 0.7092
alternative hypothesis: true mean is not equal to 2
95 percent confidence interval:
1.795420 2.298474
sample estimates:
mean of x
2.046947

```

title: ‘Regression in R’

7 Regression

7.0.0.1

This section assumes most readers will have done an introductory statistics course and had some practice running multiple regression and or Anova in SPSS or a similar package.

Describing statistical models using formulae

R requires that you are explicit about the statistical model you want to run but provides a neat, concise way of describing models, called a **formula**. For multiple regression and simple Anova, the formulas we write map closely onto the underlying *linear model*. The formula syntax provides shortcuts to quickly describe all the models you are likely to need.

Formulas have two parts: the left hand side and the right hand side, which are separated by the tilde symbol: ~. Here, the tilde just means ‘is predicted by’.

For example, this formula specifies a regression model where `height` is the *outcome*, and `age` and `gender` are the *predictor* variables.⁸

```
height ~ age + gender
```

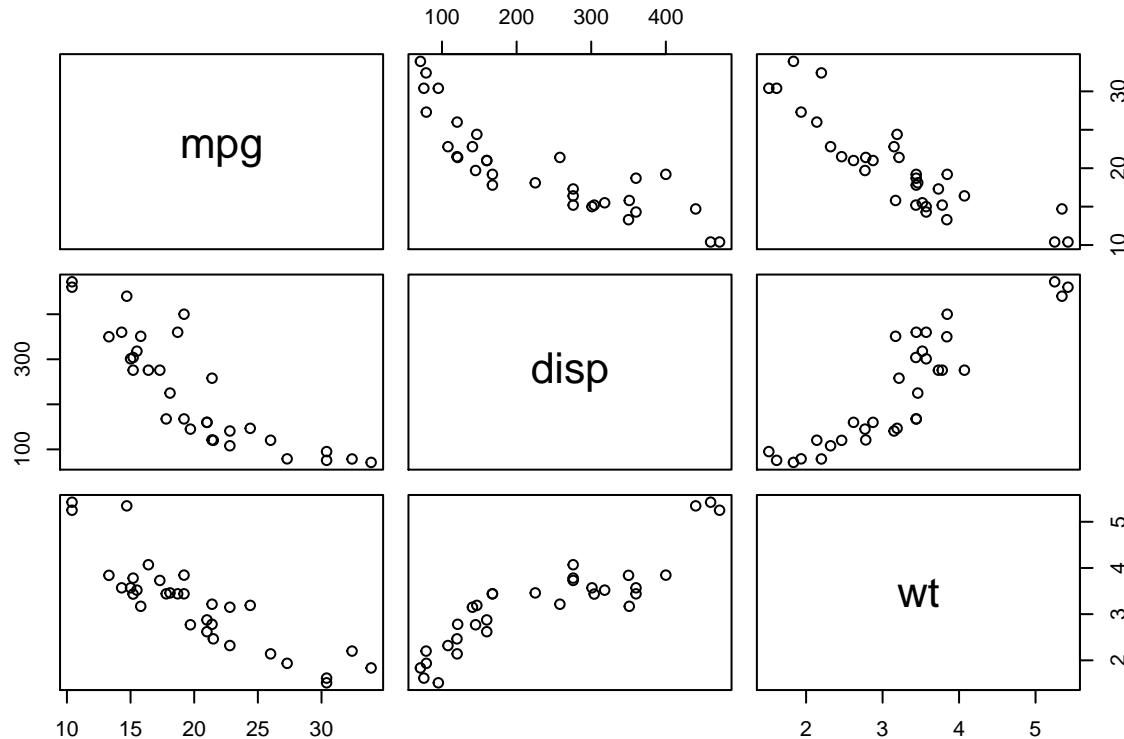
There are lots more useful tricks to learn when writing formulas, which are covered below. But in the interests of instant gratification let's work through a simple example first:

Running a linear model

Linear models (including Anova and multiple regression) are run using the `lm(...)` function, short for 'linear model'. We will use the `mtcars` dataset, which is built into R, for our first example.

First, we have a quick look at the data. The pairs plot suggests that `mpg` might be related to a number of the other variables including `disp` (engine size) and `wt` (car weight):

```
mtcars %>%
  select(mpg, disp, wt) %>%
  pairs
```



Before running any model, we should ask ourselves: "what question we are trying to answer?"

In this instance, we can see that both weight (`wt`) and engine size (`disp`) are related to `mpg`, but they are also correlated with one another. We might want to know, then, "are weight and engine size independent predictors of `mpg`?" That is, if we know a car's weight, do we gain additional information about its `mpg` by measuring engine size?

⁸I avoid the terms dependent/independent variables because they are confusing to many students, and because they are misleading when discussing non-experimental data.

To answer this, we could use multiple regression, including both `wt` and `disp` as predictors of `mpg`. The formula for this model would be `mpg ~ wt + disp`. The command below runs the model:

```
lm(mpg ~ wt + disp, data=mtcars)

Call:
lm(formula = mpg ~ wt + disp, data = mtcars)

Coefficients:
(Intercept)          wt          disp
 34.96055     -3.35083    -0.01772
```

For readers used to wading through reams of SPSS output R might seem concise to the point of rudeness. By default, the `lm` commands displays very little, only repeating the formula and listing the coefficients for each predictor in the model.

So what next? Unlike SPSS, we must be explicit and tell R exactly what we want. The most convenient way to do this is to first store the results of the `lm()` function:

```
m.1 <- lm(mpg ~ wt + disp, data=mtcars)
```

This stores the results of the `lm()` function in a variable named `m.1`. As an aside, this is a pretty terrible variable name — try to give descriptive names to your variables because this will prevent errors and make your code easier to read.

We can then use other functions to get more information about the model. For example:

```
summary(m.1)

Call:
lm(formula = mpg ~ wt + disp, data = mtcars)

Residuals:
    Min      1Q  Median      3Q      Max 
-3.4087 -2.3243 -0.7683  1.7721  6.3484 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 34.96055   2.16454  16.151 4.91e-16 ***
wt          -3.35082   1.16413  -2.878  0.00743 **  
disp        -0.01773   0.00919  -1.929  0.06362 .    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.917 on 29 degrees of freedom
Multiple R-squared:  0.7809,    Adjusted R-squared:  0.7658 
F-statistic: 51.69 on 2 and 29 DF,  p-value: 2.744e-10
```

Although still compact, the `summary` function provides some familiar output, including the estimate, *SE*, and *p* value for each parameter.

Take a moment to find the following statistics in the output above:

- The coefficients and *p* values for each predictor
- The R^2 for the overall model. What % of variance in `mpg` is explained?

Answer the original question: ‘accounting for weight (`wt`), does engine size (`disp`) tell us anything extra about a car’s `mpg`?’

More on formulas

Above we briefly introduced R's formula syntax. Formulas for linear models have the following structure:

```
left_hand_side ~ right_hand_side
```

For linear models *the left side is our outcome*, which must be a continuous variable. For categorical or binary outcomes you need to use `glm()` function, rather than `lm()`. See the section on generalised linear models for more details.

The right hand side of the formula lists our predictors. In the example above we used the `+` symbol to separate the predictors `wt` and `disp`. This told R to simply add each predictor to the model. However, many times we want to specify relationships *between* our predictors, as well as between predictors and outcomes.

For example, we might have an experiment with 2 categorical predictors, each with 2 levels — that is, a 2x2 between-subjects design.

Below, we define and run a linear model with both `vs` and `am` as predictors, along with the interaction of `vs:am`. We save this model as `m.2`, and use the `summary` command to print the coefficients.

```
m.2 <- lm(mpg ~ vs + am + vs:am, data=mtcars)
summary(m.2)

Call:
lm(formula = mpg ~ vs + am + vs:am, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max 
-6.971 -1.973  0.300  2.036  6.250 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  15.050     1.002  15.017 6.34e-15 ***
vs           5.693     1.651   3.448  0.0018 **  
am           4.700     1.736   2.708  0.0114 *   
vs:am        2.929     2.541   1.153  0.2589    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.472 on 28 degrees of freedom
Multiple R-squared:  0.7003,    Adjusted R-squared:  0.6682 
F-statistic: 21.81 on 3 and 28 DF,  p-value: 1.735e-07
```

We'd normally want to see the Anova table for this model, including the F-tests:

```
car:::Anova(m.2)
Anova Table (Type II tests)

Response: mpg
          Sum Sq Df F value    Pr(>F)    
vs       367.41  1 30.4836 6.687e-06 ***
am       276.03  1 22.9021 4.984e-05 ***
vs:am    16.01   1  1.3283    0.2589    
Residuals 337.48 28

---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

But before you do too much with Anova in R read this section.

7.0.0.2 Other formula shortcuts

In addition to the `+` symbol, we can use other shortcuts to create linear models.

As seen above, the colon `(:)` operator indicates the interaction between two terms. So `a:b` is equivalent to creating a new variable in the data frame where `a` is multiplied by `b`.

The `*` symbol indicates the expansion of other terms in the model. So, `a*b` is the equivalent of `a + b + a:b`.

Finally, it's good to know that other functions can be used within R formulas to save work. For example, if you wanted to transform your dependent variable then `log(y) ~ x` will do what you might expect, and saves creating temporary variables in your dataset.

The formula syntax is very powerful, and the above only shows the basics, but you can read the `formulae` help pages in RStudio for more details.

7.0.0.2.1

Run the following models using the `mtcars` dataset:

- With `mpg` as the outcome, and with `cyl` and `hp` as predictors
- As above, but adding the interaction of `cyl` and `hp`.
- Repeat the model above, but write the formula a different way (make the formula either more or less explicit, but retaining the same predictors in the model).

Factors and variable codings

7.0.0.3

If you store categorical data as numbers (e.g. groups 1, 2, 3 ...) it's important to make sure your predictors are entered correctly into your models.

In general, R works in a ‘regressiony’ way and will assume variables in a formula are linear predictors. So, a `group` variable coded 1...4 will be entered as a single parameter where 4 is considered twice as large as 2, etc.

See below for example. In the first model `cyl` is entered as a ‘linear slope’; in the second each value of `cyl` (4,5, or 6) is treated as a separate category. The predictions from each model could be very different:

```
linear.model <- lm(mpg ~ cyl, data=mtcars)
categorical.model <- lm(mpg ~ factor(cyl), data=mtcars)
```

In the case of different experimental groups what you would normally want is for `group` to be coded and entered as a number of categorical parameters in your model. The most common way of doing this is to use ‘dummy coding’, and this is what R will implement by default for character or factor variables.

To make sure your categorical variables are entered into your model as categories (and not a slope) you can either:

- Convert the variable to a character or factor type in the dataframe or
- Specify that the variable is a factor when you run the model

For example, here we specify `cyl` is a factor within the model formula:

```
lm(mpg ~ factor(cyl), data=mtcars)
```

Call:

```
lm(formula = mpg ~ factor(cyl), data = mtcars)
```

```
Coefficients:
(Intercept) factor(cyl)6  factor(cyl)8
  26.664      -6.921      -11.564
```

Whereas here we convert to a factor in the original dataset:

```
mtcars$cyl.factor <- factor(mtcars$cyl)
lm(mpg ~ cyl.factor, data=mtcars)

Call:
lm(formula = mpg ~ cyl.factor, data = mtcars)

Coefficients:
(Intercept) cyl.factor6 cyl.factor8
  26.664      -6.921      -11.564
```

Neither option is universally better, but if you have variables which are *definitely* factors (i.e. should never be used as slopes) it's probably better to convert them in the original dataframe, before you start modelling

Model specification

It's helpful to think about regression and other statistical models as if they were machines that do work for us — perhaps looms in a cloth factory. We feed the machines raw materials, and they busy themselves producing the finished cloth. The nature of the finished cloth is dependent on two factors: the raw material we feed it, and the setup and configuration of the machine itself.

In regression (and Anova) the same is true: Our finished results are the parameter estimates the model weaves from our raw data. The pattern we see depends on the configuration of the machine, and it's important to realise the same data can provide very different outputs depending on the setup of the machine.

7.0.0.4 Equivalent models

In some cases the ‘setup’ of the machine produces changes which, although they appear very different, are in fact equivalent in some sense. Let's say our weaving machine produces a lovely set of rugs, shown in the figure below:

Now imagine that we flip all the standard settings on the weaving machine. We feed the same raw materials to the loom, but the results *look* very different:

The second set of rugs are inversions of the first, *but the patterns remain the same*. The same sort of thing happens when we recode variables before entering them in our regression models. For example:

```
coef(lm(mpg ~ wt, data=mtcars))
(Intercept)          wt
 37.285126     -5.344472
```

We can run a completely equivalent model if we ‘flip’ the weight (`wt`) coefficient by multiplying by `-1`:

```
mtcars$wt.reversed <- -1 * mtcars$wt
coef(lm(mpg ~ wt.reversed, data=mtcars))
(Intercept) wt.reversed
 37.285126     5.344472
```

These models are equivalent in all the important ways: the test statistics, p values are all the same: only the sign of the coefficient for weight has changed.



Figure 8: Rugs made in configuration 1



Figure 9: Rugs made after configuration is changed

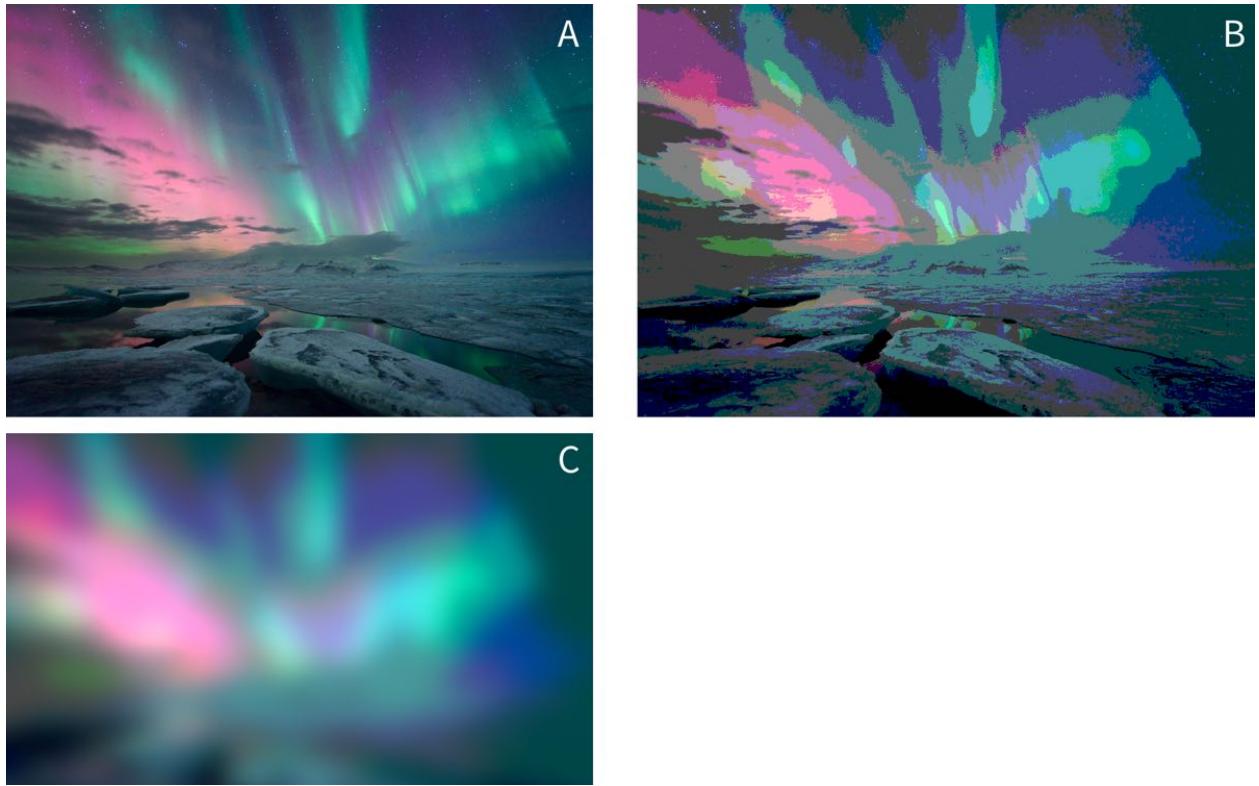


Figure 10: Image credit: <https://www.flickr.com/photos/undercrimson/13155461724>

The same kind of thing happens when we choose a different *coding scheme* for categorical variables (see section below): although the parameter estimates change when the coding format changes, the *underlying model is equivalent because it would make the same predictions for new data*.

7.0.0.5 Non-equivalent models

In the case above we saw models which were equivalent in the sense that they produced identical predictions for new observations.

Now we need to stretch our rug analogy a little, but imagine the output of our machine is now an image of the northern lights, as in image A below, but that by changing some settings of the machine, we might instead produce image B:

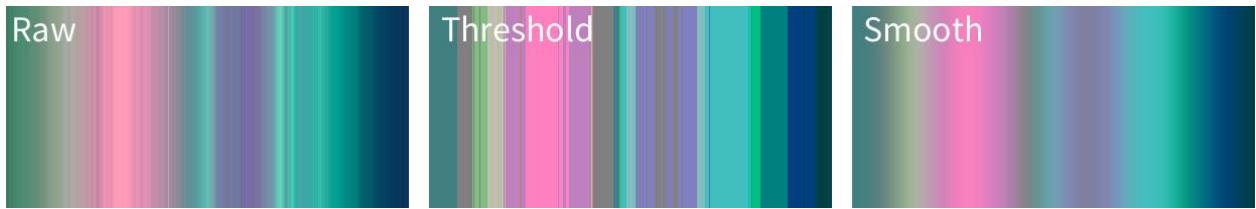
You might reasonably ask why we would prefer image B or C to image A? The answer is that, when running statistical models, we must remember that they are always *simplifications* of reality that are (hopefully) *useful* to us.

For example, our goal might be to use images of the northern lights to track the position of the aurora in the sky. If so, we might find that thresholding the picture in this way makes it easier to see where the centre of mass of the light is. By smoothing out many of the ‘ripples’ and surface level patterning in the light, the overall shape becomes clearer. And in fact this is one of the techniques computer vision systems do use to pre-process image inputs.

Likewise, we face similar problems when analysing psychological data. For example, when we measure an outcome repeatedly over a period of seconds, days or months we are probably interested in the overall *shape* of the change, rather than in the surface-level patterning of these changes.

Let’s stretch the images analogy again and simplify the image above by taking a single pixel high ‘slice’

through the centre of each image (A, B and C). In the figure below I've stretched these slices vertically so that you can see the banding in the colour, but each of these images is just a single pixel high:



Let's simplify this even further, and convert these images to greyscale:



We might think of these 3 images as follows:

- ‘Raw’ represents our raw data (image A above), and the greys represent the value of our ‘outcome’ (intensity of light). The x-axis in this case is position in the sky, but could just as well be time or some other continuous variable.
- ‘Threshold’ represents some kind of categorical model for these data (akin to image B above), in which we ‘chunk’ up the x-axis and make predictions for each chunk.
- ‘Smooth’ (akin to image C above) represents some kind of linear model with terms which represent the gradual changes in the outcome across the range of the x-axis (like slopes or polynomial terms in regression).

Because a digital image is just a list of `intensity` values for each pixel we can read the image into R and plot the raw values like any other:

```
intensity.vec <- as.vector(png::readPNG('media/aurora-1-px-raw.png'))

aurora.image <- data_frame(
  Intensity = intensity.vec) %>%
  mutate(x = row_number())
Warning: `data_frame()` is deprecated, use `tibble()` .
This warning is displayed once per session.

aurora.image %>%
  ggplot(aes(x, Intensity)) +
  geom_point(size=.5)
```

We can fit linear models to these data, just like any other. So we might start by predicting intensity with a simple slope:

```
aurora.linear <- lm(Intensity ~ x, data=aurora.image)
```

And we could make predictions from this model and plot them against the original:

```
aurora.image %>%
  mutate(linear.prediction = predict(aurora.linear)) %>%
  reshape2::melt(id.var='x') %>%
  ggplot(aes(x, value, group=variable, color=variable)) +
  geom_point(size=.5)
```

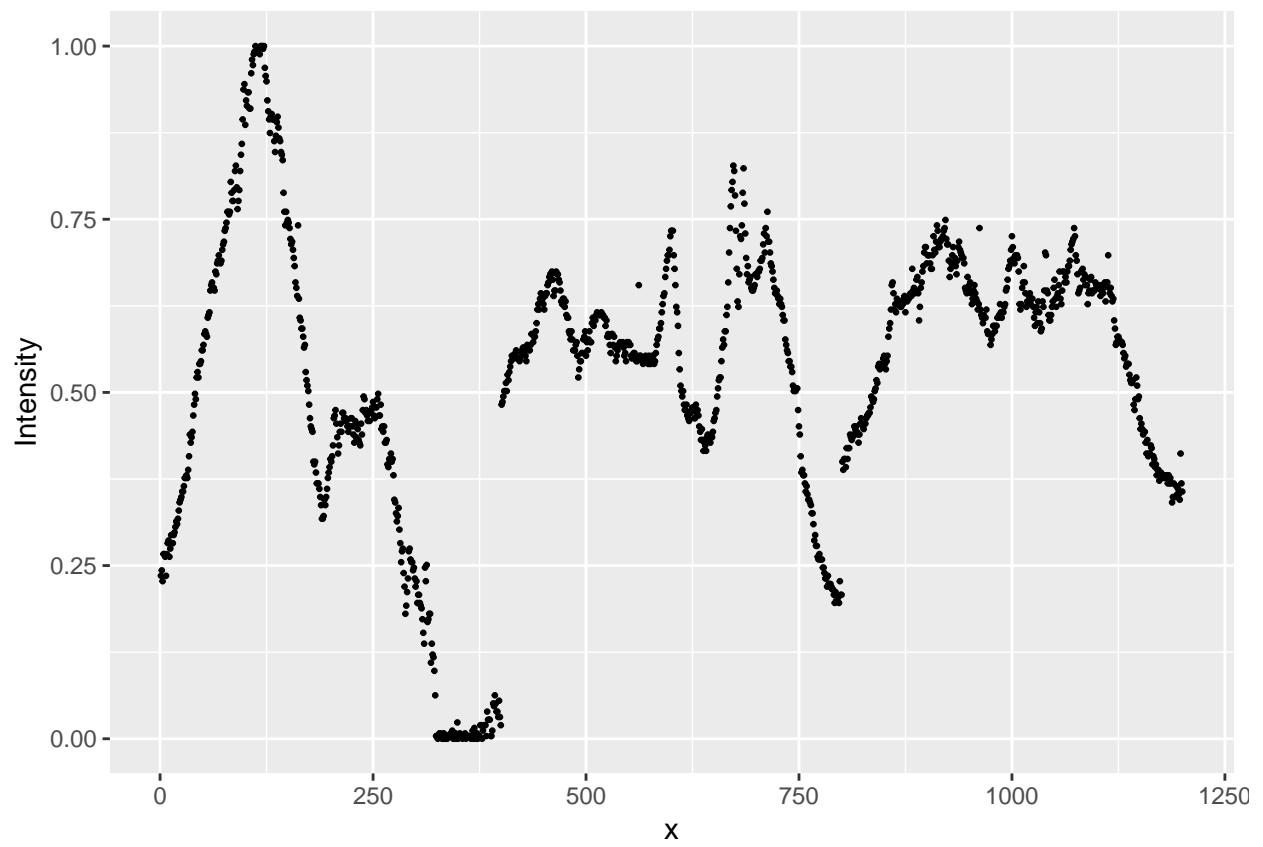
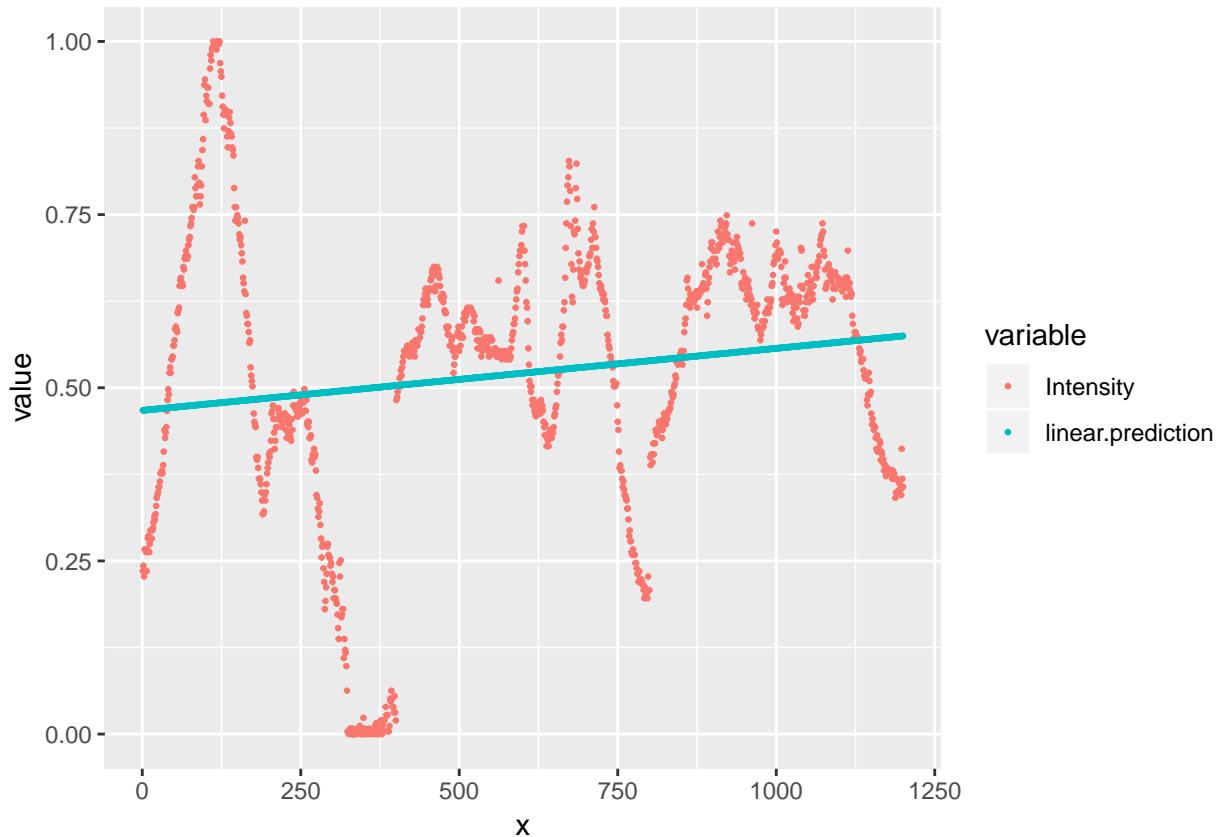


Figure 11: Plot of the intensity of light in the single pixel slice from the Aurora image. Intensity of 1 corresponds to white, and 0 to black in the original image.

```
Warning: attributes are not identical across measure variables; they will  
be dropped
```



As we can see, our predictions are pretty terrible, because the linear model only allows for a simple slope over the range of x .

To improve the model, we can go in one of two ways:

1. Fit slopes and curves for x
2. Break x up into chunks

7.0.0.6 Chunks

```
# Create a new chunked x variable (a factor)  
x.in.chunks <- cut(aurora.image$x, breaks=8)  
  
# Run a model with these chunks as a factor  
aurora.chunks <- lm(Intensity ~ x.in.chunks, data=aurora.image)  
  
# Plot the predictions again  
aurora.image %>%  
  mutate(  
    linear.prediction = predict(aurora.linear),  
    chunked.prediction = predict(aurora.chunks)  
  ) %>%  
  reshape2::melt(id.var='x') %>%  
  ggplot(aes(x, value, group=variable, color=variable)) +
```

```
geom_point(size=.5)
Warning: attributes are not identical across measure variables; they will
be dropped
```



That's somewhat better, although we can still see that the extremes of our observed data are not well predicted by either the linear model (the flat line) or the chunked model.

Try cutting the `x` variable into more chunks. What are the pros and cons of doing this? How many chunks would you need to reproduce the original data faithfully?

7.0.0.7 Slopes and curves

An alternative strategy at this point is to try and fit smooth curves through the data. One way of doing this (explained in greater detail in the section on polynomials) is to fit multiple parameters to represent the initial slope, and then changes in slope, across the values of `x`. In general, we need to fit one parameter for each change in ‘direction’ we want our curve to take.

For example, we can fit a curve with 3 changes of direction by fitting the ‘third degree’ polynomial:

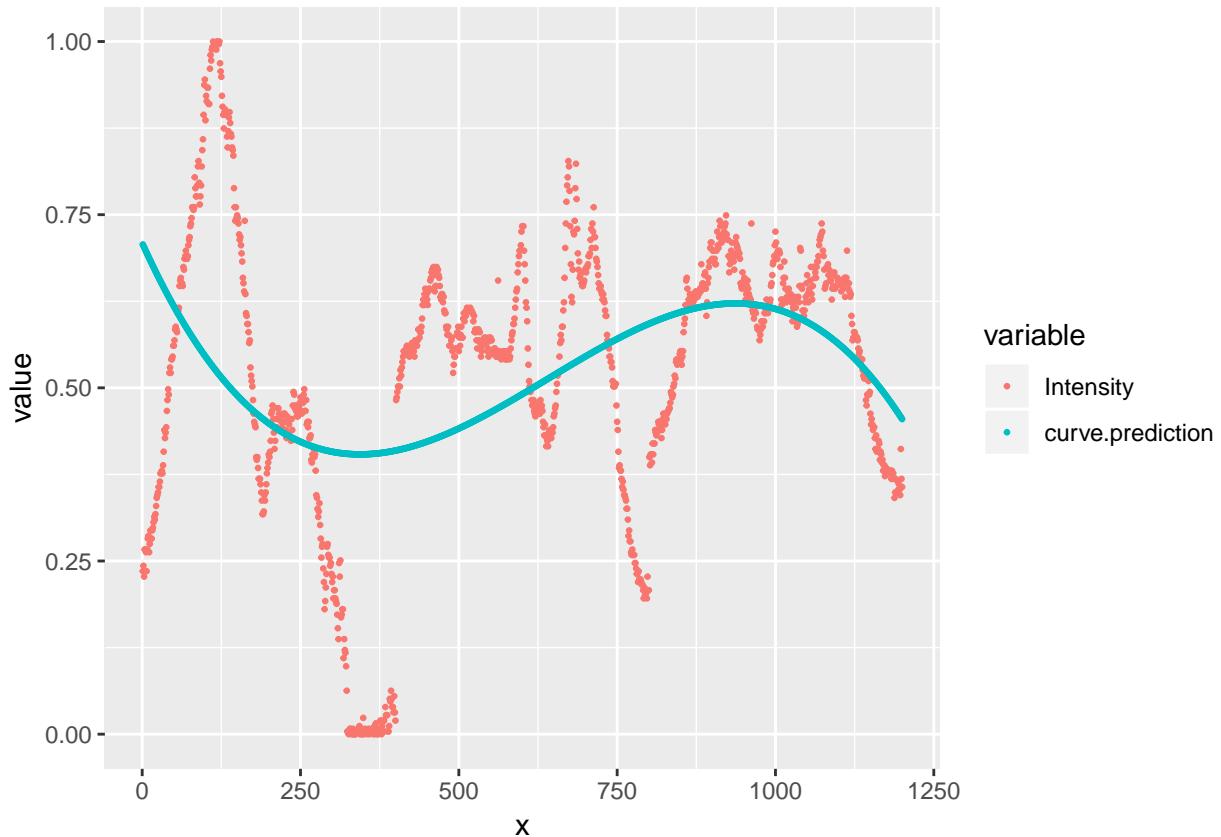
```
aurora.curve <- lm(Intensity ~ poly(x, 3), data=aurora.image)

aurora.image %>%
  mutate(
    curve.prediction = predict(aurora.curve)
  ) %>%
  reshape2::melt(id.var='x') %>%
  ggplot(aes(x, value, group=variable, color=variable)) +
```

```

geom_point(size=.5)
Warning: attributes are not identical across measure variables; they will
be dropped

```



Or we could increase the number of parameters in our curve to allow a tighter fit with the raw data and plot all the models together:

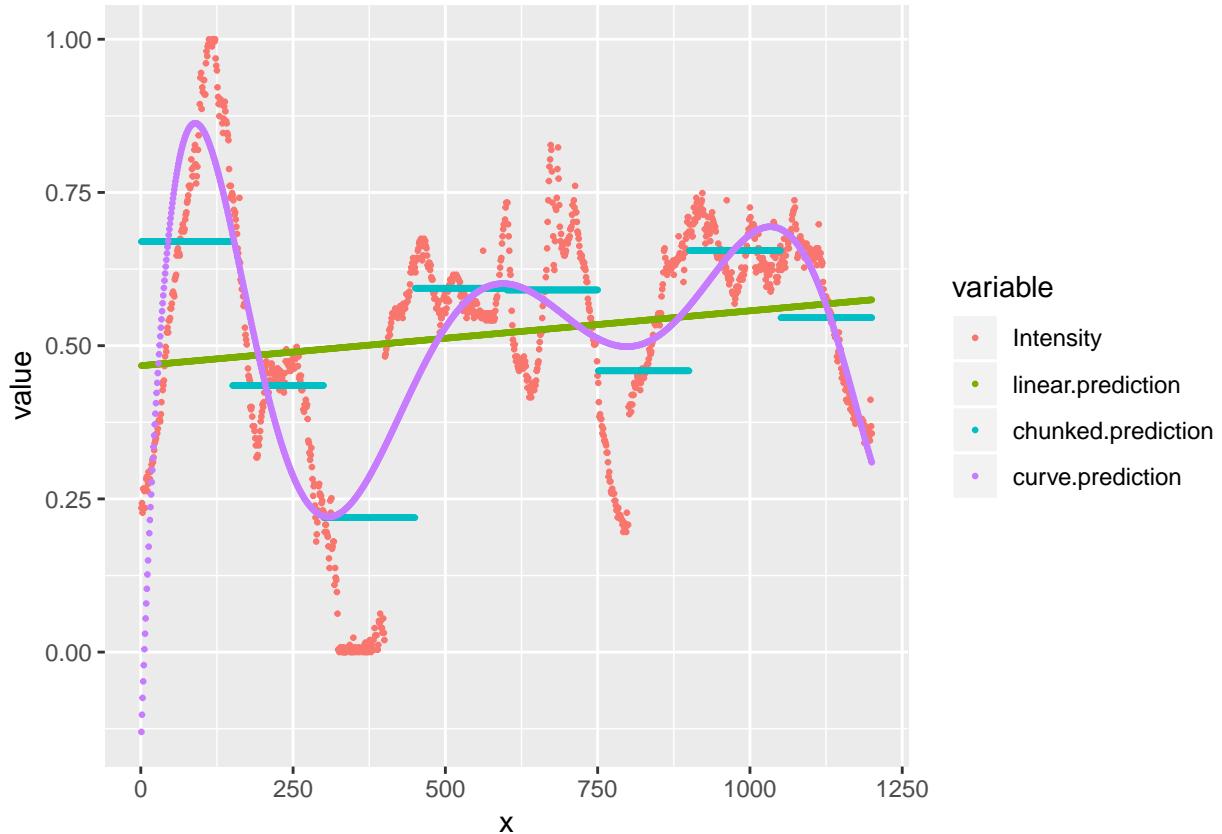
```

aurora.curve <- lm(Intensity ~ poly(x, 7), data=aurora.image)

all.predictions <- aurora.image %>%
  mutate(
    linear.prediction = predict(aurora.linear),
    chunked.prediction = predict(aurora.chunks),
    curve.prediction = predict(aurora.curve)
  ) %>%
  reshape2::melt(id.var='x')
Warning: attributes are not identical across measure variables; they will
be dropped

all.predictions %>%
  ggplot(aes(x, value, group=variable, color=variable)) +
  geom_point(size=.5)

```



We can see that this curved model is a better approximation to the raw data than our ‘chunked’ model in some places (e.g. $x = 100$), but worse in others (e.g. $x = 625$). Overall though, the R^2 is much higher for the curves model here:

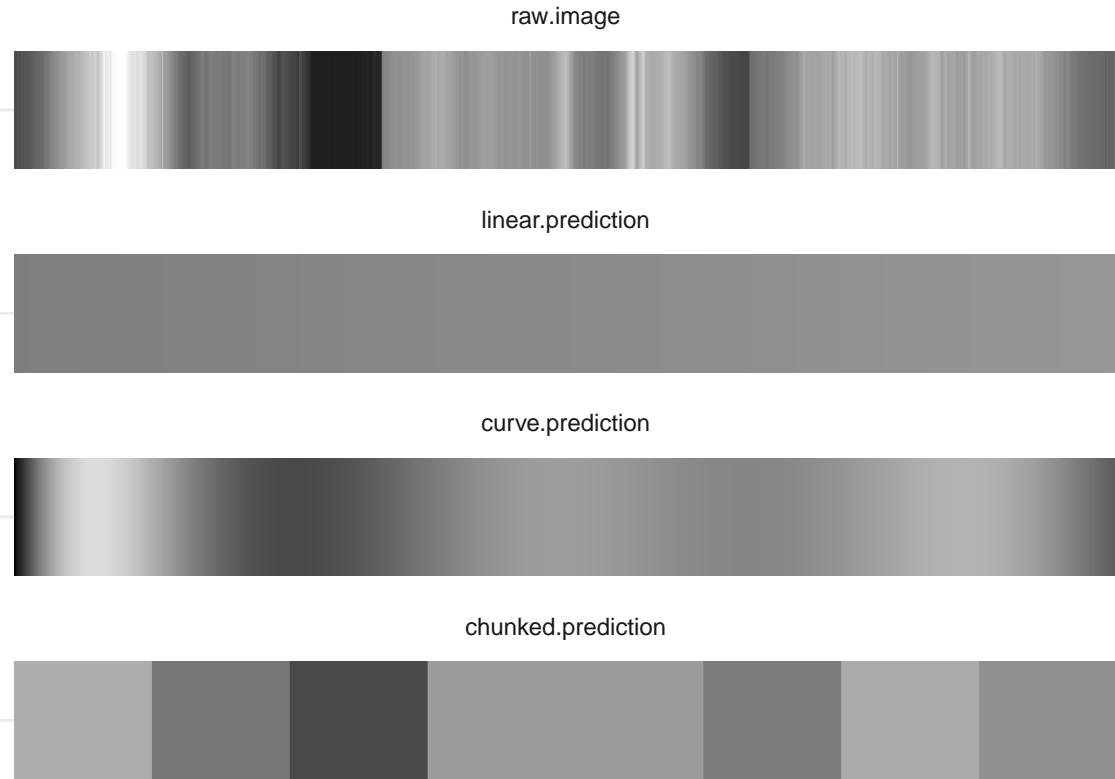
```
summary(aurora.chunks)$r.squared
[1] 0.4463338
summary(aurora.curve)$r.squared
[1] 0.6162421
```

And this is the case even though our model contains only 8 parameters, and so is just as parsimonious as the chunked model above.

```
# count the number of parameters in the chunked and curved models
length(coef(aurora.chunks))
[1] 8
length(coef(aurora.curve))
[1] 8
```

Try to plot a curve that fits even more closely to the data. There are 1200 pixels in our original image. How many parameters would you need for the model to fit the image exactly? What happens in practice if you try and fit this model?

For fun, we can even plot our data back in image form and see which is closest to matching the original:



There is no ‘right answer’ here: each model has pros and cons. You need to think about what the purpose of your model is, how you want to simplify your data, and then set up your models appropriately.

Effect/dummy coding and contrasts

TODO: Explain this:

```
options(contrasts = c("contr.treatment", "contr.poly"))
lm(mpg~factor(cyl), data=mtcars)

Call:
lm(formula = mpg ~ factor(cyl), data = mtcars)

Coefficients:
(Intercept)  factor(cyl)6  factor(cyl)8
      26.664       -6.921       -11.564

options(contrasts = c("contr.sum", "contr.poly"))
lm(mpg~factor(cyl), data=mtcars)

Call:
lm(formula = mpg ~ factor(cyl), data = mtcars)

Coefficients:
(Intercept)  factor(cyl)1  factor(cyl)2
      20.5022       6.1615      -0.7593
```

Centering (is often helpful)

When interpreting regression coefficients, and especially when interactions are present in a model, it's often overlooked that the regression parameters are the effect on the outcome of a 1-unit change in the predictor, *when all the other predictors are zero*. This applies to the intercept too: it is the predicted value of the outcome when *all* of the predictors are zero.

This is unhelpful because it makes the intercept mostly meaningless, and the other coefficients harder to interpret.

It's often a good idea to *center* your predictors so that you can interpret the intercept of the model as the average of your sample.

Scaling inputs

Interpreting regression coefficients requires that we think in the *units* of the predictor.

For example, if we include ‘age in years’ in our model, then this `yob` coefficient gives us the change in the outcome for each additional year.

However, we’re often not interested in the effect of a single year. If we are dealing with the effect of age in the general population, we’re unlikely to care about the effect of 1 year, and it might be more useful and natural to think about 10-year differences in age. In contrast, if our research is on adolescence, then the changes of the course of a year might be too crude, and we may want to think about changes over months instead.

It’s important to realise there is no general or ‘correct’ solution to this problem. Regression models don’t care about the scale of our variables (within limits), but we do need to make choice about how we scale inputs. These choices should aim to

- make regression coefficients easily interpretable and
- make results comparable across studies

These two goals will not always be 100% aligned, and there will be tradeoffs needed as you select your strategy, which will normally be one of:

1. Putting coefficients on a ‘natural’ scale or relate to meaningful quantities
2. Standardising coefficients.

7.0.0.7.1 Using a ‘natural’ scale

This will often mean just leaving your predictors ‘as-is’, but it might also mean dividing your predictor by some number to put it into more convenient units. For example, dividing age in years by 10 would mean that you can interpret the coefficient as the change over a decade, which might be easier to think about.

7.0.0.7.2 Standardising

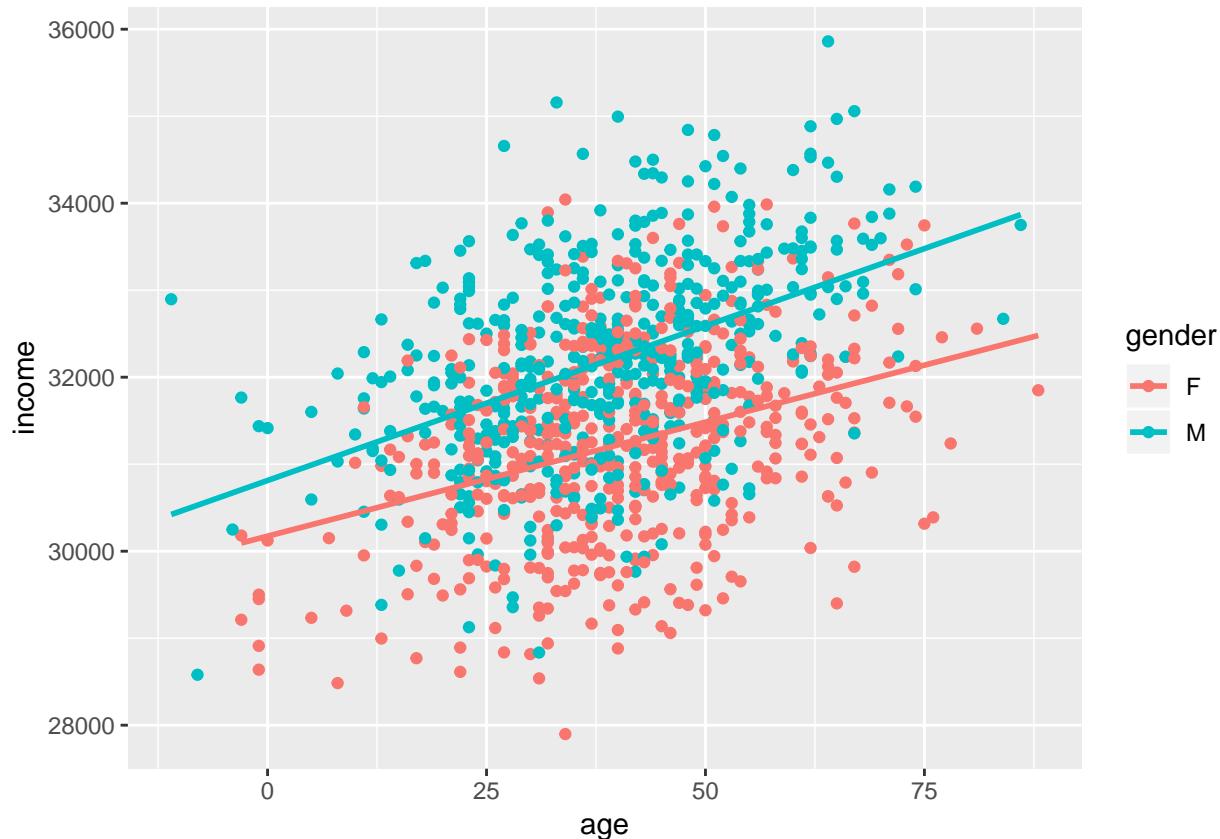
Gelman [Gelman, 2008] recommends standardising coefficients by centering and dividing by two standard deviations. This can be useful because binary variables like sex (male/female) will then be on a *similar* scale to numeric inputs.

However, be cautious when standardising. You will sometimes see people interpret standardised coefficients in terms of ‘relative importance’ of the predictors. For example, they might say that if $\beta^1 = .2$ and $\beta^2 = .4$ then β^2 is twice as important as β^1 . Although this is appealing, it’s not always valid.

The main problem is that you don't always know whether you have a full range of values of predictors in your sample. For example, imagine a case where the regression coefficient for age was linear, and = .5 in a sample from the general population.

We can plot these data to show the effect of age, and gender:

```
ggplot(incomes, aes(age, income, group=gender, color=gender)) + geom_point() + geom_smooth(se=F, method=
```



Older people earn more than younger people, and men earn slightly more than women (in this simulated dataset), but this gender gap doesn't change with age.

We can model this and print the effects of age and gender:

```
m1 <- lm(income~age+gender, data=incomes)
coef(m1)
(Intercept)      age      gender1
30493.82430   30.76665  -504.52116
```

And we can standardize these effects using the `standardize` function:

```
coef(arms::standardize(m1))
(Intercept)      z.age      c.gender
31702.1240    921.0568   1009.0423
```

Based on these standardised coefficients we might say that age and gender are of roughly equal importance in predicting income.

If we re-fit the model on a subset of the data, for example only individuals under 40, the regression coefficients won't change much because the effect was constant across the range of ages:

```

younger.incomes <- incomes %>%
  filter(age<40)

m2 <- lm(income~age+gender, data=younger.incomes)
coef(m2)
(Intercept)      age      gender1
30528.65050   29.51746  -429.80520

```

But, the standardised coefficients *do* change, because we have restricted the range of ages in the sample:

```

coef(arms::standardize(m2))
(Intercept)      z.age      c.gender
31352.7078     539.1859    859.6104

```

The standardised effect of `age` is now roughly half that of `gender`.

The take home message here is that standardisation can be useful to put predictors on a similar scale, but it's not a panacea and can't be interpreted as a simple measure of 'importance'. You still need to think!

Alternatives to rescaling

A nice alternative to scaling the inputs of your regression is to set aside the raw coefficients and instead make predictions for values of the predictors that are of theoretical or practical interest. The section on predictions and marginal effects has lots more detail on this.

What next

It is strongly recommended that you read the section on Anova before doing anything else.

As noted above, R has a number of important differences in its default settings, as compared with packages like Stata or SPSS. These can make important differences to the way you interpret the output of linear models, especially Anova-type models with categorical predictors.

8 Anova

Be sure to read the section on linear models in R *before* you read this section, and specifically the parts on specifying models with formulae.

This section attempts to cover in a high level way how to specify anova models in R and some of the issues in interpreting the model output. If you need to revise the basic idea of an Anova, the Howell textbook [Howell, 2016]. For a very quick reminder, this interactive/animated explanation of Anova is helpful.

If you just want the 'answers' — i.e. the syntax to specify common Anova models – you could skip to the next section: Anova cookbook

There are 4 rules for doing Anova in R and not wanting to cry:

1. Keep your data in 'long' format.
2. Know the differences between character, factor and numeric variables
3. Do not use the `aov()` or `anova()` functions to get an Anova table unless you know what you are doing.
4. Learn about the types of sums of squares and always remember to specify `type=3`, unless you know better.

Rules for using Anova in R

8.0.0.1 Rule 1: Use long format data

In R, data are almost always most useful a long format where:

- each row of the dataframe corresponds to a single measurement occasion
- each column corresponds to a variable which is measured

For example, in R we will have data like this:

```
df %>%
  head %>%
  pander
```

person	time	predictor	outcome
1	1	1	11
1	2	1	11
1	3	1	6
2	1	4	10
2	2	4	5
2	3	4	15

Whereas in SPSS we might have the same data structured like this:

```
df.wide %>%
  head %>%
  pander
```

person	predictor	Time 1	Time 2	Time 3
1	1	11	11	6
2	4	10	5	15
3	2	8	6	8
4	2	11	13	10
5	3	10	10	7
6	5	7	10	12

R always uses long form data when running an Anova, but one downside is that it therefore has no automatic to know which rows belong to which person (assuming individual people are the unit of error in your model). This means that for repeated measures designs you need to make explicit which measures are repeated when specifying the model (see the section on repeated designs below).

8.0.0.2 Rule 2: Know your variables

See the section on dataframes and on the different column types and be sure you can distinguish:

- Numeric variables
- Factors
- Character strings.

In Anova:

- Outcomes will be numeric variables
- Predictors will be factors or (preferably) character strings

If you want to run Ancova models, you can also add numeric predictors.

8.0.0.3 Rule 3: Don't use `aov()` or `anova()`

This is the most important rule of all.

The `aov` and `anova` functions have been around in R a long time. For various historical reasons the defaults for these functions won't do what you expect if you are used to SPSS, Stata, SAS, and most other stats packages. These differences are important and will be confusing and give you misleading results unless you understand them.

The recommendation here is:

- If you have a factorial experiment define your model using `lm()` and then use `car::Anova()` to calculate F tests.
- If you have repeated measures, your data are perfectly balanced, and you have no missing values then use `afex::car_aov()`.
- If you think you want a repeated measures Anova but your data are not balanced, or you have missing data, use linear mixed models instead via the `lme4::` package.

8.0.0.4 Rule 4: Use type 3 sums of squares (and learn why)

You may be aware, but there are at least 3 different ways of calculating the sums of squares for each factor and interaction in an Anova. In short,

- SPSS and most other packages use type 3 sums of squares.
- `aov` and `anova` use type 1.
- By default, `car::Anova` and `ez::ezANOVA` use type 2, but can use type 3 if you ask.

This means you must:

- Make sure you use type 3 sums of squares unless you have a reason not to.
- Always pass `type=3` as an argument when running an Anova.

8.0.0.4.1

A longer explanation of *why* you probably want type 3 sums of squares is given in this online discussion on stats.stackexchange.com and practical implications are shown in this worked example.

An even longer answer, including a much deeper exploration of the philosophical questions involved is given by Venables [1998].

Recommendations for doing Anova

1. Make sure to Plot your raw data *first*
2. Where you have interactions, be cautious in interpreting the main effects in your model, and always plot the model predictions.
3. If you find yourself aggregating (averaging) data before running your model, think about using a mixed or multilevel model instead.
4. If you are using repeated measures Anova, check if you should be using a mixed model instead. If you have an unbalanced design or any missing data, you probably should use a mixed model.

Anova ‘Cookbook’

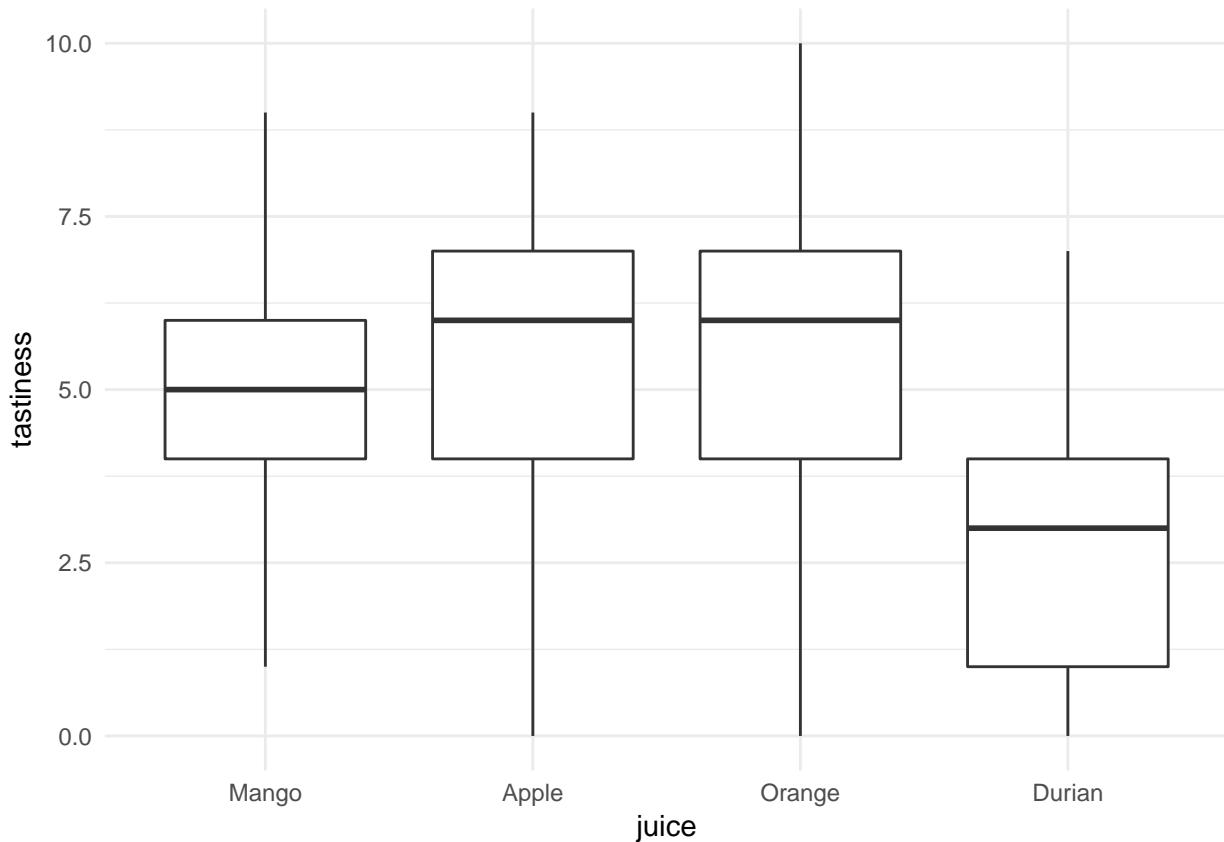
This section is intended as a shortcut to running Anova for a variety of common types of model. If you want to understand more about what you are doing, read the section on principles of Anova in R first, or consult an introductory text on Anova which covers Anova [e.g. Howell, 2012].

Between-subjects Anova

8.0.0.5 Oneway Anova (> 2 groups)

If your design has more than 2 groups then you should use oneway Anova.

Let's say we asked people to taste 1 of 4 fruit juices, and rate how tasty it was on a scale from 0 to 10:



We can run a oneway Anova with type 3 sums of squares using the `Anova` function from the `car::` package:

```
juice.lm <- lm(tastiness ~ juice, data=tasty.juice)
juice.anova <- car:::Anova(juice.lm, type=3)
juice.anova
Anova Table (Type III tests)

Response: tastiness
          Sum Sq Df  F value    Pr(>F)
(Intercept) 615.04  1 114.4793 < 2.2e-16 ***
juice       128.83  3    7.9932 8.231e-05 ***
Residuals   515.76 96
```

```
--  
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

And we could compute the contrasts for each fruit against the others (the grand mean):

```
juice.lsm <- emmeans::emmeans(juice.lm, pairwise~juice, adjust="fdr")  
juice.contrasts <- emmeans::contrast(juice.lsm, "eff")  
juice.contrasts$contrasts  


| contrast               | estimate | SE    | df | t.ratio | p.value |
|------------------------|----------|-------|----|---------|---------|
| Mango - Apple effect   | -1.90    | 0.637 | 96 | -2.982  | 0.0060  |
| Mango - Orange effect  | -1.54    | 0.512 | 96 | -3.005  | 0.0060  |
| Mango - Durian effect  | 1.02     | 0.346 | 96 | 2.952   | 0.0060  |
| Apple - Orange effect  | -0.78    | 0.637 | 96 | -1.224  | 0.2239  |
| Apple - Durian effect  | 1.78     | 0.512 | 96 | 3.473   | 0.0046  |
| Orange - Durian effect | 1.42     | 0.637 | 96 | 2.229   | 0.0338  |



P value adjustment: fdr method for 6 tests


```

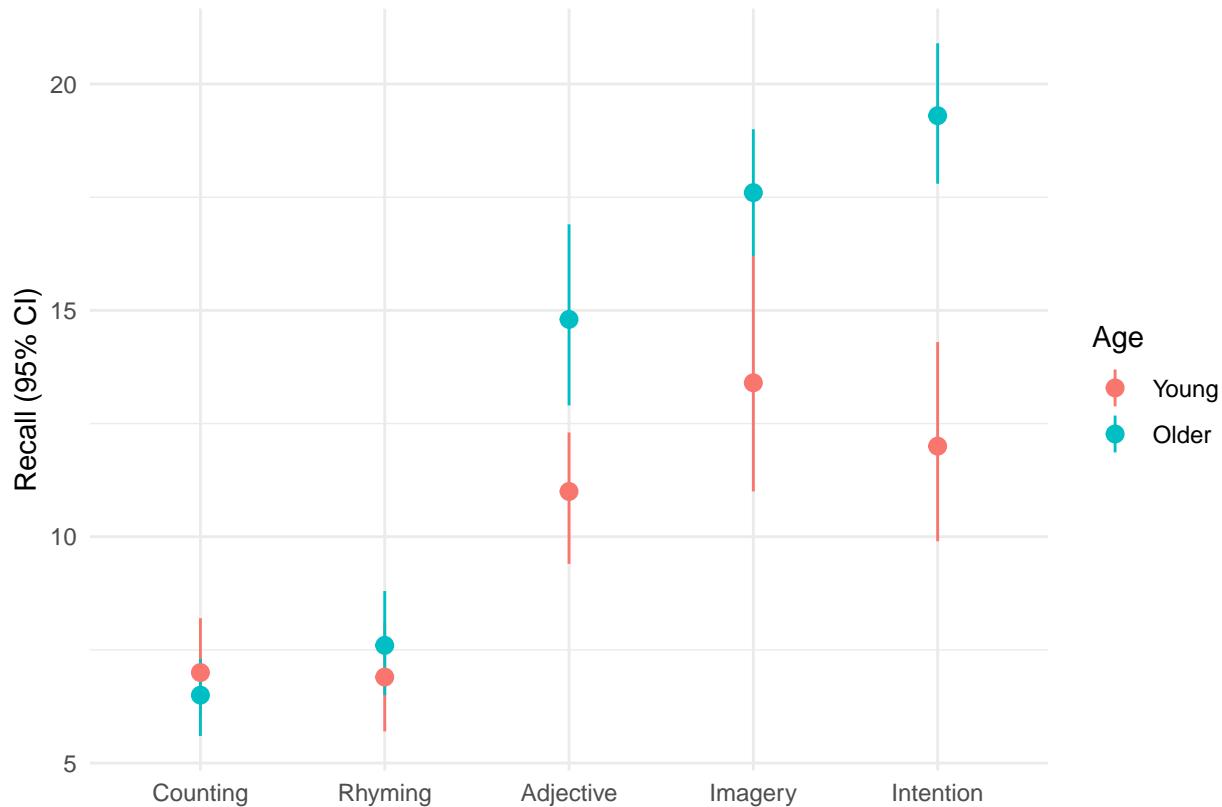
We found a significant main effect of juice, `r apastats::describe.Anova(juice.anova, 2)`. Followup tests (adjusted for false discovery rate) indicated that only Durian differed from the other juices, and was rated a significantly less tasty Mango, Apple, and Orange juice.

8.0.0.6 Factorial Anova

We are using a dataset from Howell [Howell, 2012, chapter 13]: an experiment which recorded `Recall` among young v.s. older adults (`Age`) for each of 5 conditions.

These data would commonly be plotted something like this:

```
eysenck <- readRDS("data/eysenck.Rdata")  
eysenck %>%  
  ggplot(aes(Condition, Recall, group=Age, color=Age)) +  
    stat_summary(geom="pointrange", fun.data = mean_cl_boot) +  
    ylab("Recall (95% CI)") +  
    xlab("")
```



Visual inspection of the data (see Figure X) suggested that older adults recalled more words than younger adults, and that this difference was greatest for the intention, imagery, and adjective conditions. Recall performance was worst in the counting and rhyming conditions.

Or alternatively if we wanted to provide a better summary of the distribution of the raw data we could use a boxplot:

```
eysenck %>%
  ggplot(aes(Age, Recall)) +
  geom_boxplot(width=.33) +
  facet_grid(~Condition) +
  ylab("Recall (95% CI)") +
  xlab("")
```

We can run a linear model including the effect of Age and Condition and the interaction of these variables, and calculate the Anova:

```
eysenck.model <- lm(Recall ~ Age * Condition, data=eysenck)
car:::Anova(eysenck.model, type=3)
```

Anova Table (Type III tests)

```
Response: Recall
          Sum Sq Df F value    Pr(>F)
(Intercept) 490.00  1 61.0550 9.85e-12 ***
Age         1.25  1  0.1558 0.6940313
Condition   351.52  4 10.9500 2.80e-07 ***
Age:Condition 190.30  4  5.9279 0.0002793 ***
Residuals   722.30 90
```

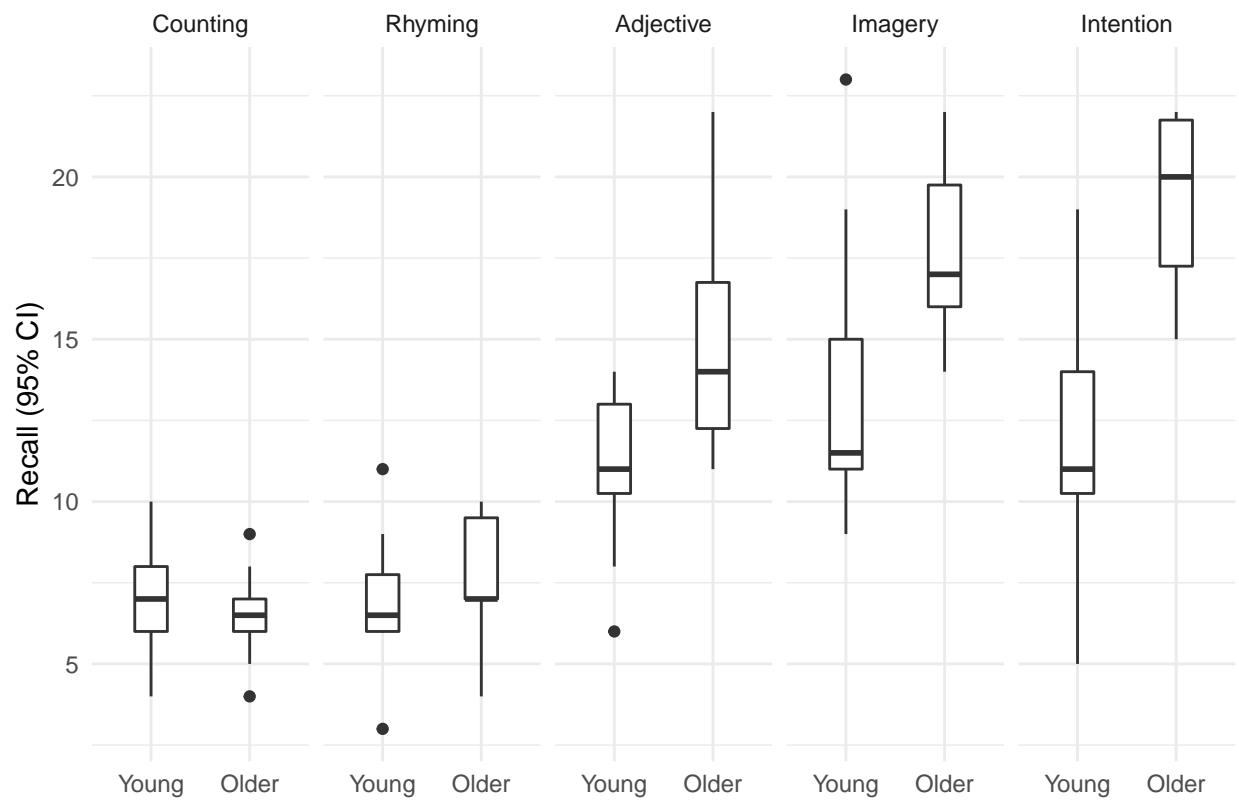


Figure 12: Boxplot for recall in older and young adults, by condition.

```
--  
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Repeated measures or ‘split plot’ designs

It might be controversial to say so, but the tools to run traditional repeat measures Anova in R are a bit of a pain to use. Although there are numerous packages simplify the process a little, their syntax can be obtuse or confusing. To make matters worse, various textbooks, online guides and the R help files themselves show many ways to achieve the same ends, and it can be difficult to follow the differences between the underlying models that are run.

At this point, given the many other advantages of linear mixed models over traditional repeated measures Anova, and given that many researchers abuse traditional Anova in practice (e.g. using it for unbalanced data, or where some data are missing), the recommendation here is to simply give up and learn how to run linear mixed models. These can (very closely) replicate traditional Anova approaches, but also:

- Handle missing data or unbalanced designs gracefully and efficiently.
- Be expanded to include multiple levels of nesting. For example, allowing pupils to be nested within classes, within schools. Alternatively multiple measurements of individual patients might be clustered by hospital or therapist.
- Allow time to be treated as a continuous variable. For example, time can be modelled as a slope or some kind of curve, rather than a fixed set of observation-points. This can be more parsimonious, and more flexible when dealing with real-world data (e.g. from clinical trials).

It would be best at this point to jump straight to the main section multilevel or mixed-effects models, but to give one brief example of mixed models in use:

8.0.0.7

The `sleepstudy` dataset in the `lme4` package provides reaction time data recorded from participants over a period of 10 days, during which time they were deprived of sleep.

```
lme4::sleepstudy %>%  
  head(12) %>%  
  pander
```

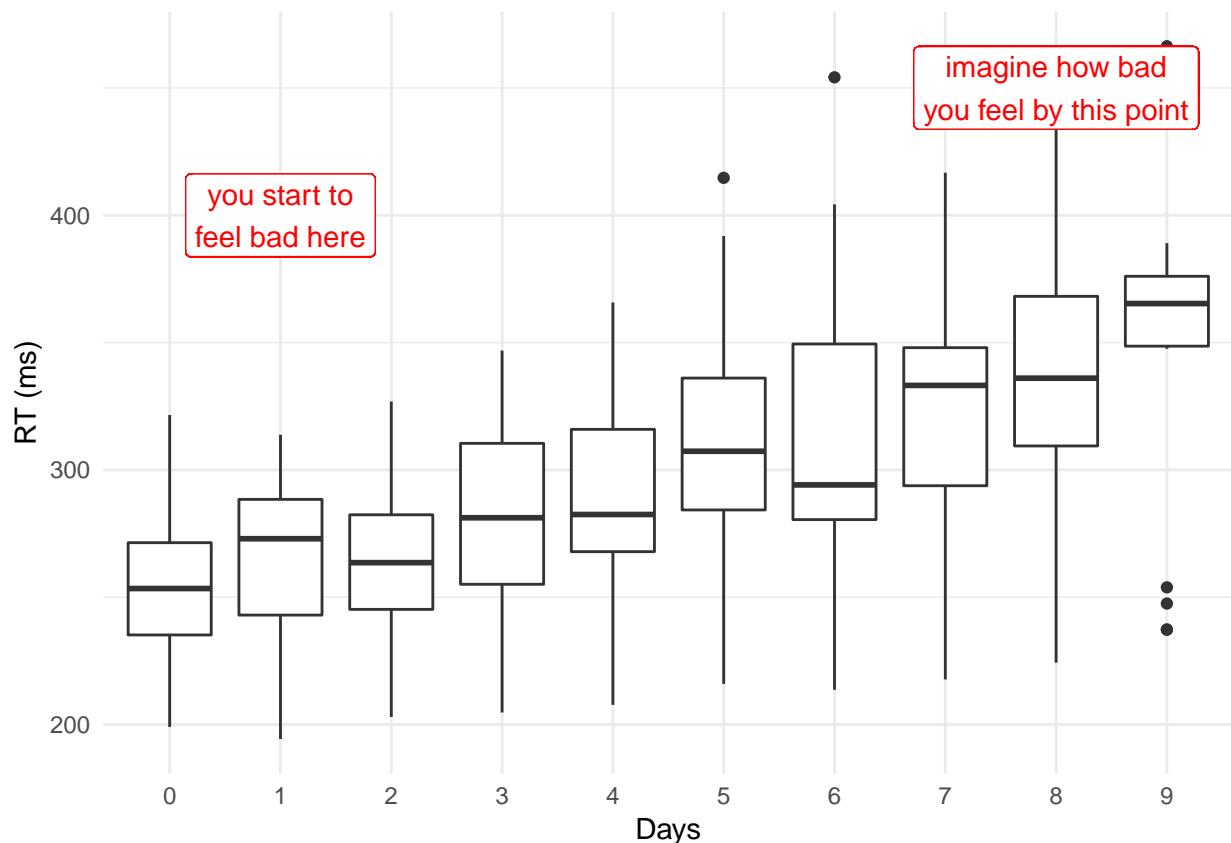
Reaction	Days	Subject
249.6	0	308
258.7	1	308
250.8	2	308
321.4	3	308
356.9	4	308
414.7	5	308
382.2	6	308
290.1	7	308
430.6	8	308
466.4	9	308
222.7	0	309
205.3	1	309

We can plot these data to show the increase in RT as sleep deprivation continues:

```

lme4::sleepstudy %>%
  ggplot(aes(factor(Days), Reaction)) +
  geom_boxplot() +
  xlab("Days") + ylab("RT (ms)") +
  geom_label(aes(y=400, x=2, label="you start to\nfeel bad here"), color="red") +
  geom_label(aes(y=450, x=9, label="imagine how bad\nyou feel by this point"), color="red")

```



If we want to test whether there are significant differences in RTs between Days, we could fit something very similar to a traditional repeat measures Anova using the `lme4::lmer()` function, and obtain an Anova table for the model using the special `anova()` function which is added by the `lmerTest` package:

```

sleep.model <- lmer(Reaction ~ factor(Days) + (1 | Subject), data=lme4::sleepstudy)
anova(sleep.model)
Type III Analysis of Variance Table with Satterthwaite's method
  Sum Sq Mean Sq NumDF DenDF F value    Pr(>F)
factor(Days) 166235   18471      9     153 18.703 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

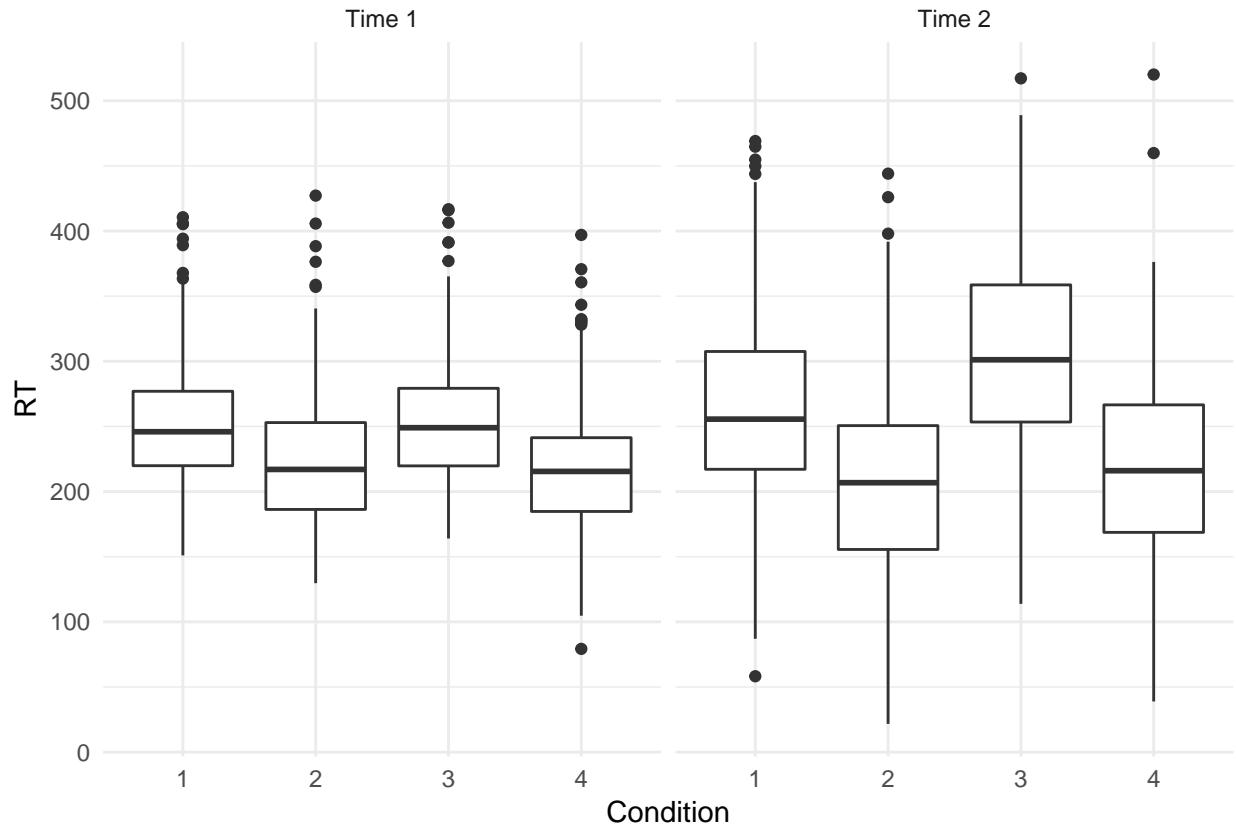
Traditional repeated measures Anova

If you really need to fit the traditional repeated measures Anova (e.g. your supervisor/reviewer has asked you to) then you should use either the `afex::` or `ez::` packages.

Let's say we have an experiment where we record reaction 25 times (`Trial`) before and after (`Time = {1, 2}`) one of 4 experimental manipulations (`Condition = {1,2,3,4}`). You have 12 participants in each condition

and no missing data:

```
expt.data %>%
  ggplot(aes(Condition, RT)) +
  geom_boxplot() +
  facet_wrap(~paste("Time", time))
```



We want to use our repeated measurements before and after the experimental interventions to increase the precision of our estimate of the between-condition differences.

Our first step is to aggregate RTs for the multiple trials, taking the mean across all trials at a particular time:

```
expt.data.agg <- expt.data %>%
  group_by(Condition, person, time) %>%
  summarise(RT=mean(RT))

head(expt.data.agg)
# A tibble: 6 x 4
# Groups:   Condition, person [3]
  Condition person time     RT
  <fct>    <fct>  <fct> <dbl>
1 1         1      1     270.
2 1         1      2     246.
3 1         2      1     257.
4 1         2      2     247.
5 1         3      1     239.
6 1         3      2     249.
```

Because our data are still in long form (we have two rows per person), we have to explicitly tell R that `time` is a within subject factor. Using the `afex::` package we would write:

```
expt.afex <- afex::aov_car(RT ~ Condition * time + Error(person/time),
                           data=expt.data.agg)
expt.afex$anova_table %>%
  pander(caption="`afex::aov_car` output.")
```

Table 36: `afex::aov_car` output.

	num Df	den Df	MSE	F	ges	Pr(>F)
Condition	3	44	160.6	142.1	0.8289	1.193e-22
time	1	44	160.5	20.85	0.1915	3.987e-05
Condition:time	3	44	160.5	29.42	0.5006	1.358e-10

Using the `ez::` package we would write:

```
expt.ez <- ez::ezANOVA(data=expt.data.agg,
                        dv = RT,
                        wid = person,
                        within = time,
                        between = Condition)

expt.ez$ANOVA %>%
  pander(caption="`ez::ezANOVA` output.")
```

Table 37: `ez::ezANOVA` output.

	Effect	DFn	DFd	F	p	p<.05	ges
2	Condition	3	44	142.1	1.193e-22	*	0.8289
3	time	1	44	20.85	3.987e-05	*	0.1915
4	Condition:time	3	44	29.42	1.358e-10	*	0.5006

These are the same models: any differences in the output are simply due to rounding. You should use whichever of `ez::` and `afex::` you find easiest to understand

The `ges` column is the generalised eta squared effect-size measure, which is preferable to the partial eta-squared reported by SPSS [Bakeman, 2005].

8.0.0.8 But what about [insert favourite R package for Anova]?

Lots of people like `ez::ezANOVA` and other similar packages. My problem with `ezANOVA` is that it doesn't use formulae to define the model and for this reason encourages students to think of Anova as something magical and separate from linear models and regression in general.

This guide is called 'just enough R', so I've mostly chosen to show only `car::Anova` because I find this the most coherent method to explain. Using formulae to specify the model reinforces a technique which is useful in many other contexts. I've make an exception for repeated because many people find specifying the error structure explicitly confusing and hard to get right, and so `ez::` may be the best option in these cases.

Comparison with a multilevel model

For reference, a broadly equivalent (although not identical) multilevel model would be:

```
expt.mlm <- lmer(RT ~ Condition * time + (1|person),
  data=expt.data.agg)

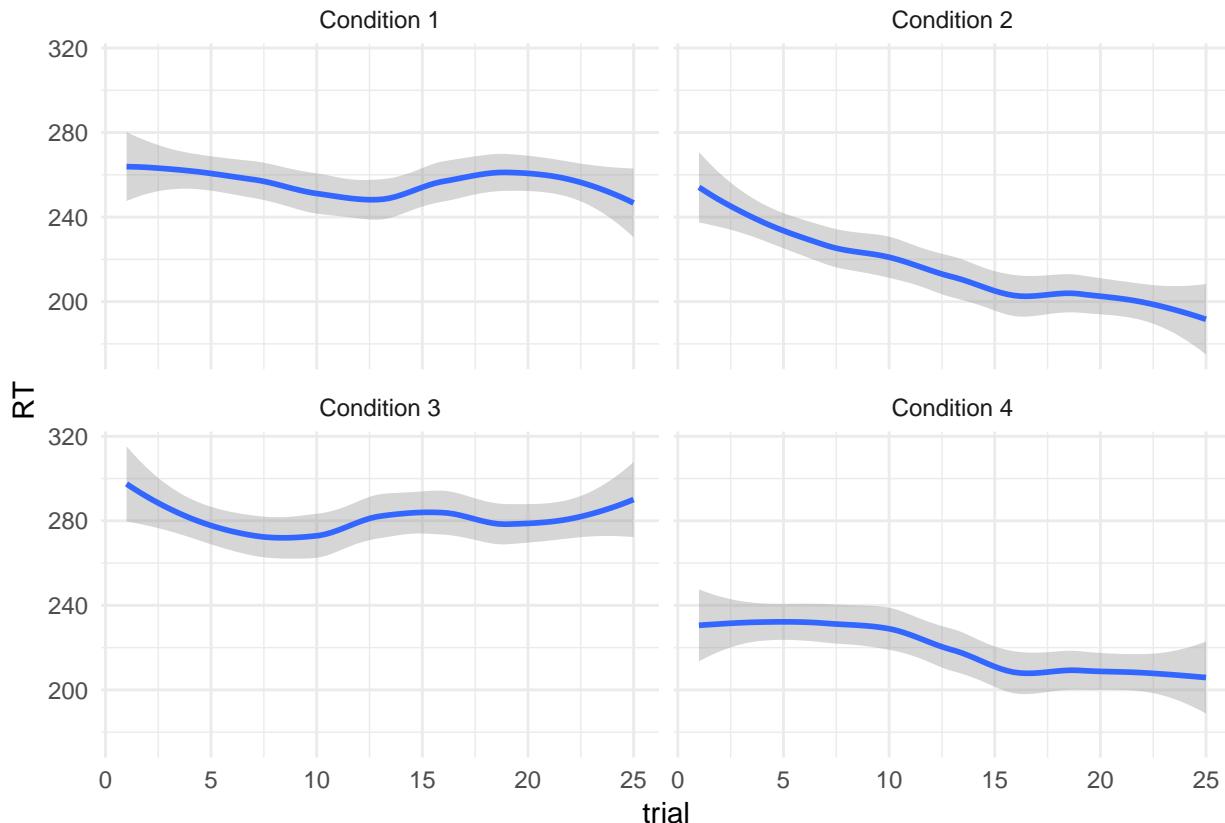
anova(expt.mlm) %>%
  pander()
```

Table 38: Type III Analysis of Variance Table with Satterthwaite's method

	Sum Sq	Mean Sq	NumDF	DenDF	F value	Pr(>F)
Condition	68424	22808	3	41.22	142.1	9.189e-22
time	3346	3346	1	41.21	20.84	4.443e-05
Condition:time	14167	4722	3	41.21	29.41	2.486e-10

Although with a linear mixed model it would also be possible to analyse the trial-by-trial data. Let's hypothesise, for example, that subjects in Conditions 2 and 4 experienced a 'practice effect', such that their RTs reduced over multiple trials. If we plot the data, we can see this suspicion may be supported (how convenient!):

```
ggplot(expt.data,
  aes(trial, RT)) +
  geom_smooth() +
  facet_wrap(~paste("Condition", Condition))
```



If we wanted to replicate the aggregated RM Anova models shown above we could write:

```
options(contrasts = c("contr.sum", "contr.poly"))
expt.mlm2 <- lmer(RT ~ Condition * time + (time|person), data=expt.data)
anova(expt.mlm2)
Type III Analysis of Variance Table with Satterthwaite's method
      Sum Sq Mean Sq NumDF DenDF F value    Pr(>F)
Condition     1331102   443701      3   67.844 118.473 < 2.2e-16 ***
time          65082    65082      1   67.921  17.378 8.875e-05 ***
Condition:time 275524   91841      3   67.921  24.523 7.285e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

But we can now add a continuous predictor for trial:

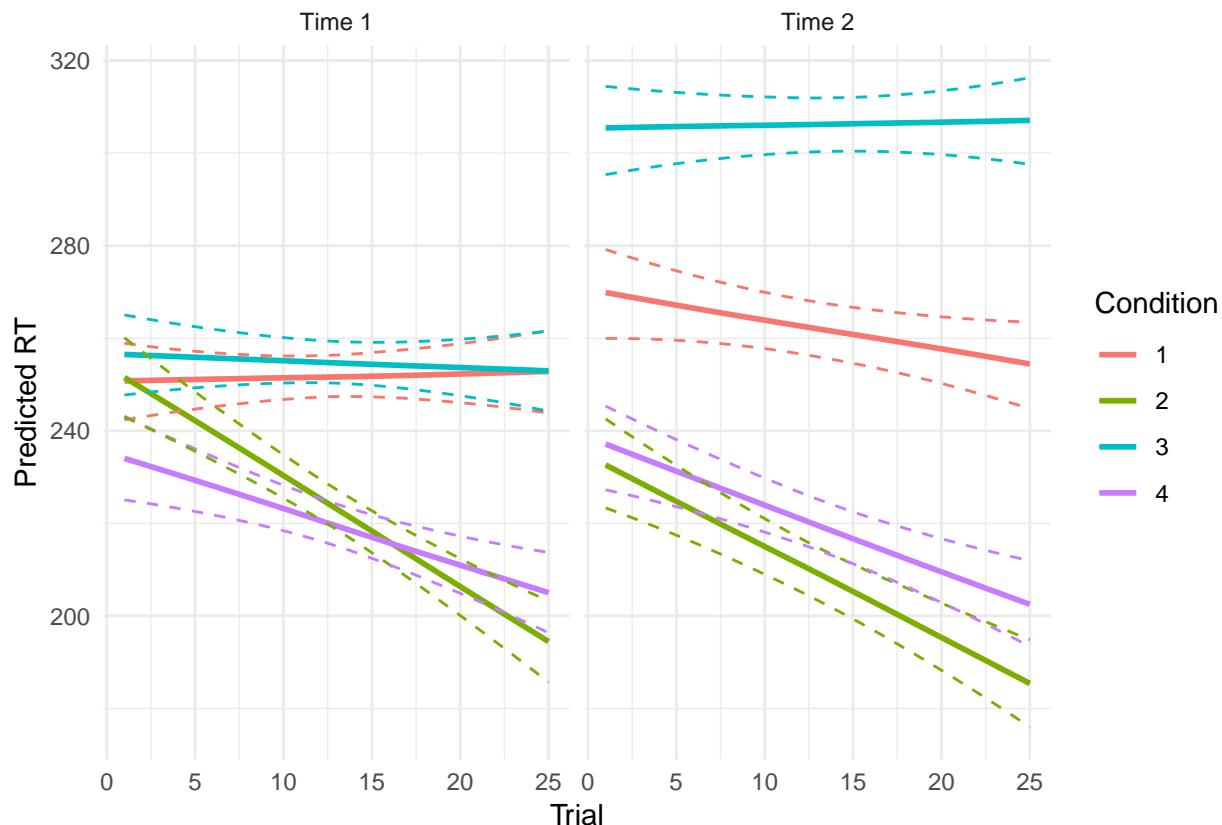
```
expt.mlm.bytrial <- lmer(RT ~ Condition * time * trial +
                           (time|person),
                           data=expt.data)

anova(expt.mlm.bytrial)
Type III Analysis of Variance Table with Satterthwaite's method
      Sum Sq Mean Sq NumDF DenDF F value    Pr(>F)
Condition     150397   50132      3   658.88 13.6651 1.174e-08 ***
time          21561    21561      1   659.67  5.8772  0.01561 *
trial         112076   112076      1 2340.00 30.5499 3.615e-08 ***
Condition:time 81051    27017      3   659.67  7.3643 7.362e-05 ***
Condition:trial 90818    30273      3 2340.00  8.2517 1.845e-05 ***
time:trial       178      178      1 2340.00  0.0485  0.82576
Condition:time:trial 5940    1980      3 2340.00  0.5397  0.65509
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The significant **Condition:trial** term indicates that there was a difference in the practice effects between the experimental conditions.

8.0.0.9

We found a significant interaction between condition and the linear term for trial number, $F(3, 2340.18) = 10.83, p < .001$. We explored this effect by plotting model-estimated reaction times for each group for trials 1 through 25 (see Figure X): participants in condition 2 and 4 experienced a greater reduction in RTs across trial, suggesting a larger practice effect for these conditions.



8.0.0.10

See the multilevel models section for more details, including analyses which allow the effects of interventions to vary between participants (i.e., relaxing the assumption that an intervention will be equally effective for all participants).

8.0.0.11 RM Anova v.s. multilevel models

- The RM Anova is perhaps more familiar, and may be conventional in your field which can make peer review easier (although in other fields mixed models are now expected where the design warrants it).
- RM Anova requires complete data: any participant with any missing data will be dropped from the analysis. This is problematic where data are expensive to collect, and where data are unlikely to be missing at random, for example in a clinical trial. In these cases RM Anova may be less efficient and more biased than an equivalent multilevel model.
- There is no simple way of calculating effect size measures like η^2 from the `lmer` model. This may or may not be a bad thing. Baguley [2009], for example, recommends reporting simple (rather than standardised) effect size measures, and is easily done by making predictions from the model.

Checking assumptions

The text below continues on from this example of factorial Anova.

If we want to check that the assumptions of our Anova models are met, these tables and plots would be a reasonable place to start. First running Levene's test:

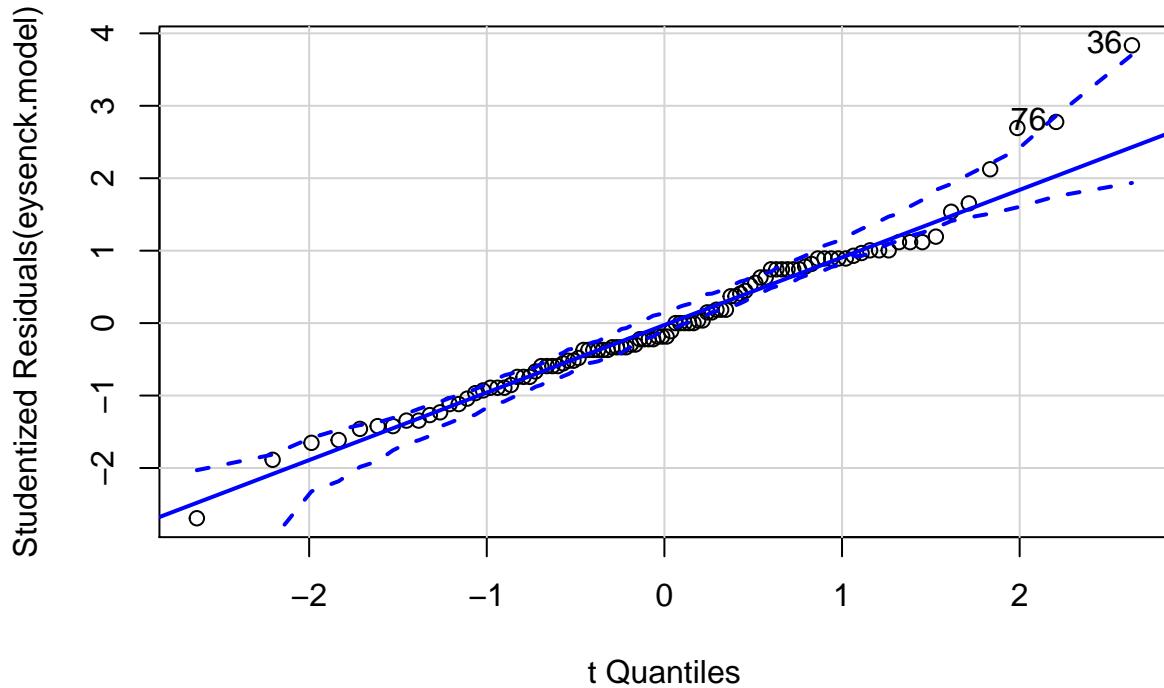


Figure 13: QQ plot to assess normality of model residuals

```
car::leveneTest(eysenck.model) %>%
  pander()
```

Table 39: Levene's Test for Homogeneity of Variance (center = median)

	Df	F value	Pr(>F)
group	9	1.031	0.4217
	90	NA	NA

Then a QQ-plot of the model residuals to assess normality:

```
car::qqPlot(eysenck.model)
```

[1] 36 76

And finally a residual-vs-fitted plot:

```
data_frame(
  fitted = predict(eysenck.model),
  residual = residuals(eysenck.model)) %>%
  # and then plot points and a smoothed line
  ggplot(aes(fitted, residual)) +
  geom_point() +
```

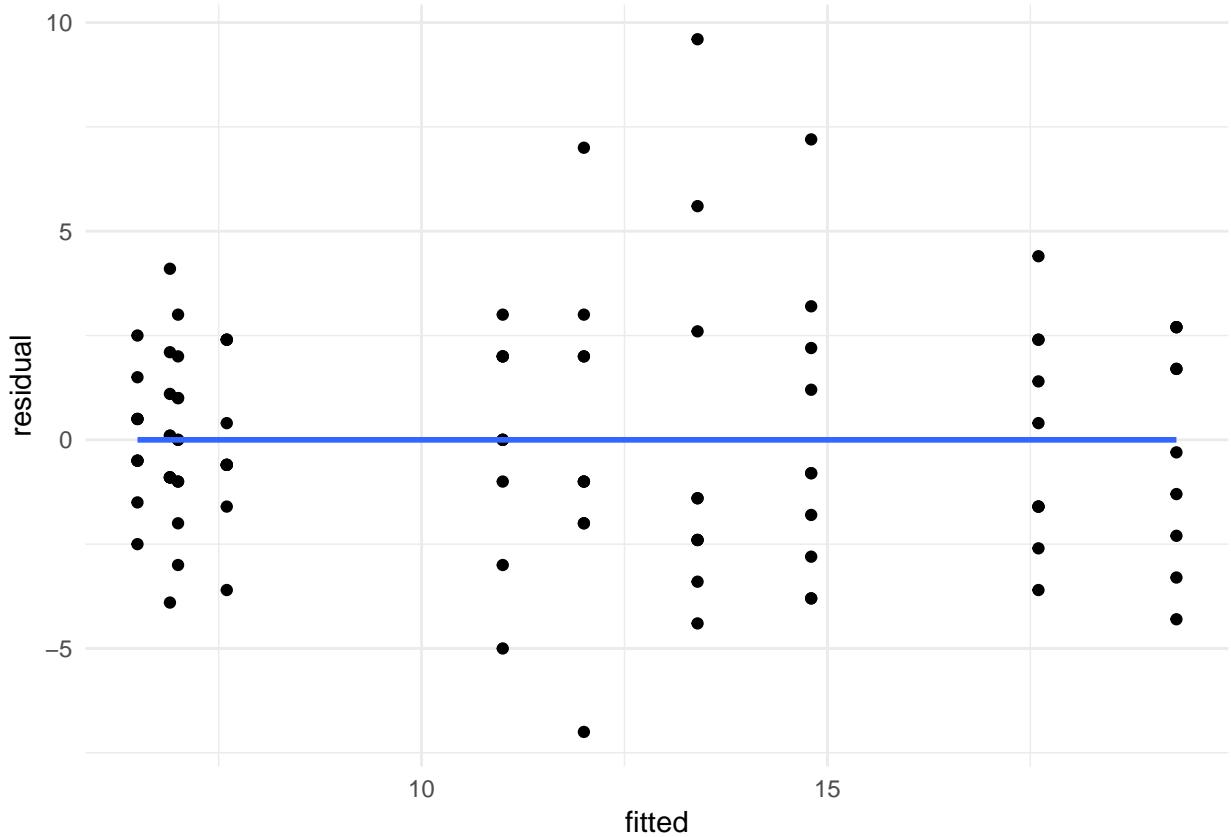


Figure 14: Residual vs fitted (spread vs. level) plot to check homogeneity of variance.

```
geom_smooth(se=F)
```

For more on assumptions checks after linear models or Anova see: <http://www.statmethods.net/stats/anovaAssumptions.html>

Followup tests

The text below continues on from this example of factorial Anova.

If we want to look at post-hoc pairwise tests we can use the the `emmeans()` function from the `emmeans::` package. By default Tukey correction is applied for multiple comparisons, which is a reasonable default:

```
em <- emmeans::emmeans(eysenck.model, pairwise~Age:Condition)
em$contrasts
  contrast           estimate    SE df t.ratio p.value
  Young,Counting - Older,Counting   0.5 1.27 90  0.395 1.0000
  Young,Counting - Young,Rhyming   0.1 1.27 90  0.079 1.0000
  Young,Counting - Older,Rhyming  -0.6 1.27 90 -0.474 1.0000
  Young,Counting - Young,Adjective -4.0 1.27 90 -3.157 0.0633
  Young,Counting - Older,Adjective -7.8 1.27 90 -6.157 <.0001
  Young,Counting - Young,Imagery   -6.4 1.27 90 -5.052 0.0001
  Young,Counting - Older,Imagery  -10.6 1.27 90 -8.367 <.0001
  Young,Counting - Young,Intention -5.0 1.27 90 -3.947 0.0058
```

Young,Counting	-	Older,Intention	-12.3	1.27	90	-9.709	<.0001
Older,Counting	-	Young,Rhyming	-0.4	1.27	90	-0.316	1.0000
Older,Counting	-	Older,Rhyming	-1.1	1.27	90	-0.868	0.9970
Older,Counting	-	Young,Adjective	-4.5	1.27	90	-3.552	0.0205
Older,Counting	-	Older,Adjective	-8.3	1.27	90	-6.551	<.0001
Older,Counting	-	Young,Imagery	-6.9	1.27	90	-5.446	<.0001
Older,Counting	-	Older,Imagery	-11.1	1.27	90	-8.761	<.0001
Older,Counting	-	Young,Intention	-5.5	1.27	90	-4.341	0.0015
Older,Counting	-	Older,Intention	-12.8	1.27	90	-10.103	<.0001
Young,Rhyming	-	Older,Rhyming	-0.7	1.27	90	-0.553	0.9999
Young,Rhyming	-	Young,Adjective	-4.1	1.27	90	-3.236	0.0511
Young,Rhyming	-	Older,Adjective	-7.9	1.27	90	-6.236	<.0001
Young,Rhyming	-	Young,Imagery	-6.5	1.27	90	-5.131	0.0001
Young,Rhyming	-	Older,Imagery	-10.7	1.27	90	-8.446	<.0001
Young,Rhyming	-	Young,Intention	-5.1	1.27	90	-4.025	0.0044
Young,Rhyming	-	Older,Intention	-12.4	1.27	90	-9.787	<.0001
Older,Rhyming	-	Young,Adjective	-3.4	1.27	90	-2.684	0.1963
Older,Rhyming	-	Older,Adjective	-7.2	1.27	90	-5.683	<.0001
Older,Rhyming	-	Young,Imagery	-5.8	1.27	90	-4.578	0.0006
Older,Rhyming	-	Older,Imagery	-10.0	1.27	90	-7.893	<.0001
Older,Rhyming	-	Young,Intention	-4.4	1.27	90	-3.473	0.0260
Older,Rhyming	-	Older,Intention	-11.7	1.27	90	-9.235	<.0001
Young,Adjective	-	Older,Adjective	-3.8	1.27	90	-2.999	0.0950
Young,Adjective	-	Young,Imagery	-2.4	1.27	90	-1.894	0.6728
Young,Adjective	-	Older,Imagery	-6.6	1.27	90	-5.209	0.0001
Young,Adjective	-	Young,Intention	-1.0	1.27	90	-0.789	0.9986
Young,Adjective	-	Older,Intention	-8.3	1.27	90	-6.551	<.0001
Older,Adjective	-	Young,Imagery	1.4	1.27	90	1.105	0.9830
Older,Adjective	-	Older,Imagery	-2.8	1.27	90	-2.210	0.4578
Older,Adjective	-	Young,Intention	2.8	1.27	90	2.210	0.4578
Older,Adjective	-	Older,Intention	-4.5	1.27	90	-3.552	0.0205
Young,Imagery	-	Older,Imagery	-4.2	1.27	90	-3.315	0.0411
Young,Imagery	-	Young,Intention	1.4	1.27	90	1.105	0.9830
Young,Imagery	-	Older,Intention	-5.9	1.27	90	-4.657	0.0005
Older,Imagery	-	Young,Intention	5.6	1.27	90	4.420	0.0011
Older,Imagery	-	Older,Intention	-1.7	1.27	90	-1.342	0.9409
Young,Intention	-	Older,Intention	-7.3	1.27	90	-5.762	<.0001

P value adjustment: tukey method for comparing a family of 10 estimates

Both cell means and pairwise contrasts are shown here. There is much more detail on computing pairwise comparisons and other types of contrasts in the section on multiple comparisons, including ways to extract and present your comparisons in APA format.

9 Generalized linear models

Linear regression is suitable for outcomes which are continuous numerical scores. In practice this requirement is often relaxed slightly, for example for data which are slightly skewed, or where scores are somewhat censored (e.g. questionnaire scores which have a minimum or maximum).

However, for some types of outcomes standard linear models are unsuitable. Examples here include binary (zero or one) or count data (i.e. positive integers representing frequencies), or proportions (e.g. proportion

of product failures per batch). This section is primarily concerned with binary outcomes, but many of the same principles apply to these other types of outcome.

Logistic regression

In R we fit logistic regression with the `glm()` function which is built into R, or if we have a multilevel model with a binary outcome we use `glmer()` from the `lme4::` package.

Fitting the model is very similar to linear regression, except we need to specify the `family="binomial"` parameter to let R know what type of data we are using.

Here we use the `titanic` dataset (you can download this from Kaggle, although you need to sign up for an account).

Before we start fitting models, it's best to plot the data to give us a feel for what is happening.

Figure 1 reveals that, across all fare categories, women were more likely to survive the disaster than men. Ticket class also appears to be related to outcome: those with third class tickets were less likely to survive than those with first or second class tickets. However, differences in survival rates for men and women differed across ticket classes: women with third class tickets appear to have been less advantaged (compared to men) than women with first or second class tickets.

```
titanic <- read.csv('data/titanic.csv')
titanic %>%
  ggplot(aes(factor(Pclass), Survived,
             group=Sex, color=Sex)) +
  stat_summary() +
  stat_summary(geom="line") +
  xlab("Ticket class")
```

Given the plot above, it seems reasonable to predict survival from `Sex` and `Pclass`, and also to include the interaction between these variables.

To run a logistic regression we specify the model as we would with `lm()`, but instead use `glm()` and specify the `family` parameter:

```
m <- glm(Survived ~ Sex * factor(Pclass),
          data=titanic, family = binomial(link="logit"))
```

9.0.0.1

Because it can become repetitive to write out the `family` parameter in full each time, I usually write a 'helper function' called `logistic()` which simply calls `glm` with the right settings. For example:

```
# define a helper function for logistic regression the ...
# means 'all arguments', so this function passes all it's
# arguments on to the glm function, but sets the family correctly
logistic <- function(...) {
  glm(..., family = binomial(link="logit"))
}
```

Which you can use like so:

```
logistic(Survived ~ Sex * factor(Pclass), data=titanic)

Call:  glm(formula = ..1, family = binomial(link = "logit"), data = ..2)

Coefficients:
```

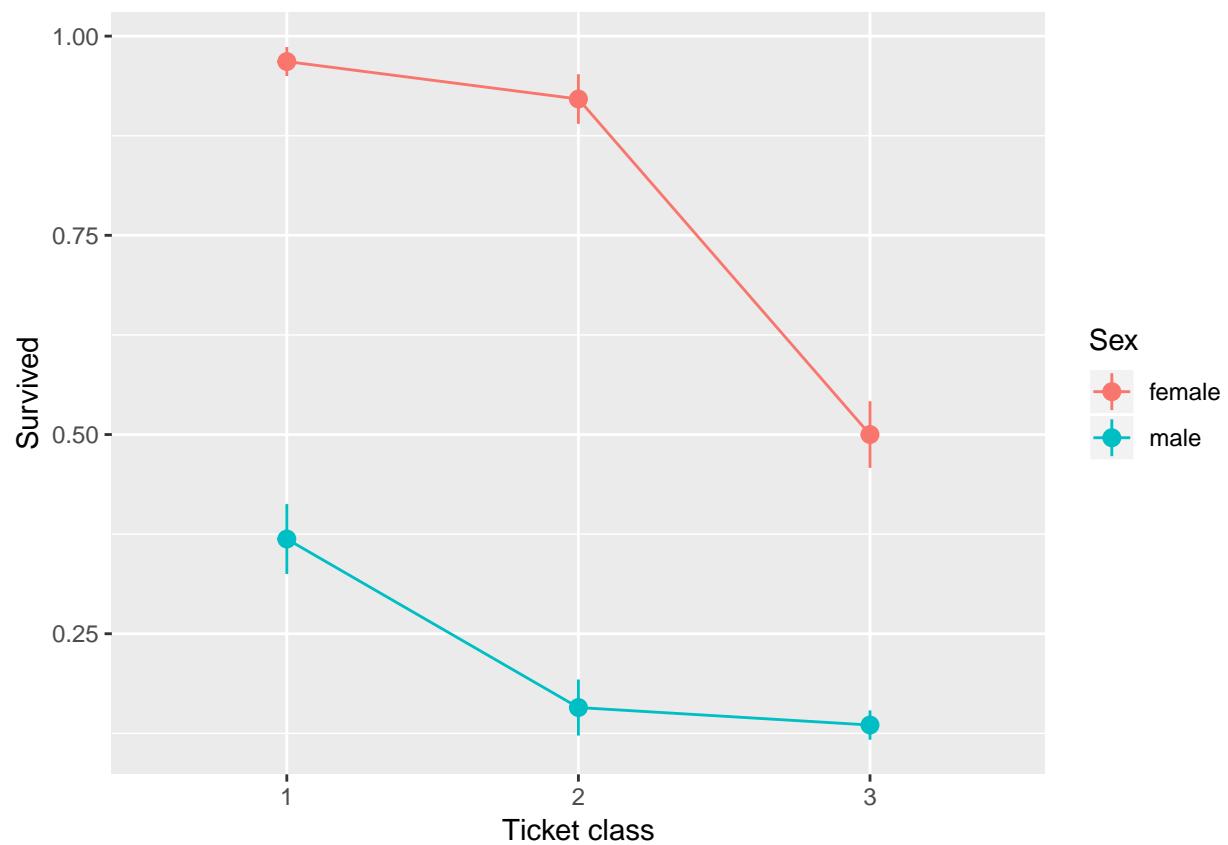


Figure 15: Survival probabilities by Sex and ticket class.

```

(Intercept)                      Sexmale          factor(Pclass)2
            3.4122           -3.9494           -0.9555
factor(Pclass)3  Sexmale:factor(Pclass)2  Sexmale:factor(Pclass)3
            -3.4122           -0.1850           2.0958

Degrees of Freedom: 890 Total (i.e. Null); 885 Residual
Null Deviance: 1187
Residual Deviance: 798.1    AIC: 810.1

```

9.0.0.2 Tests of parameters

As with `lm()` models, we can use the `summary()` function to get p values for parameters in `glm` objects:

```
titanic.model <- logistic(Survived ~ Sex * factor(Pclass), data=titanic)
summary(titanic.model)
```

Call:

```
glm(formula = ..1, family = binomial(link = "logit"), data = ..2)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-2.6248	-0.5853	-0.5395	0.4056	1.9996

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	3.4122	0.5868	5.815	6.06e-09 ***
Sexmale	-3.9494	0.6161	-6.411	1.45e-10 ***
factor(Pclass)2	-0.9555	0.7248	-1.318	0.18737
factor(Pclass)3	-3.4122	0.6100	-5.594	2.22e-08 ***
Sexmale:factor(Pclass)2	-0.1850	0.7939	-0.233	0.81575
Sexmale:factor(Pclass)3	2.0958	0.6572	3.189	0.00143 **

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

```
Null deviance: 1186.7 on 890 degrees of freedom
Residual deviance: 798.1 on 885 degrees of freedom
AIC: 810.1
```

Number of Fisher Scoring iterations: 6

You might have spotted in this table that `summary` reports z tests rather than t tests for parameters in the `glm` model. These can be interpreted as you would the t-test in a linear model, however.

9.0.0.2.1 Tests of categorical predictors

Where there are categorical predictors we can also reuse the `car::Anova` function to get the equivalent of the F test from a linear model (with type 3 sums of squares; remember not to use the built in `anova` function unless you want type 1 sums of squares):

```
car::Anova(titanic.model, type=3)
Analysis of Deviance Table (Type III tests)
```

```

Response: Survived
          LR Chisq Df Pr(>Chisq)
Sex             97.547  1 < 2.2e-16 ***
factor(Pclass)   90.355  2 < 2.2e-16 ***
Sex:factor(Pclass) 28.791  2 5.598e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Note that the Anova table for a `glm` model provides χ^2 tests in place of F tests. Although they are calculated differently, you can interpret these χ^2 tests and p values as you would for F tests in a regular Anova.

9.0.0.3 Predictions after `glm`

As with linear models, we can make predictions from `glm` models for our current or new data.

One twist here though is that we have to choose whether to make predictions in units of the response (i.e. probability of survival), or of the transformed response (logit) that is actually the ‘outcome’ in a `glm` (see the explainer on transformations and links functions).

You will almost always want predictions in the units of your response, which means you need to add `type="response"` to the `predict()` function call. Here we predict the chance of survival for a new female passenger with a first class ticket:

```

new.passenger = expand.grid(Pclass=1, Sex=c("female"))
predict.glm(titanic.model, newdata=new.passenger, type="response")
    1
0.9680851

```

And we could plot probabilities for each gender and class with a standard error for this prediction if desired:

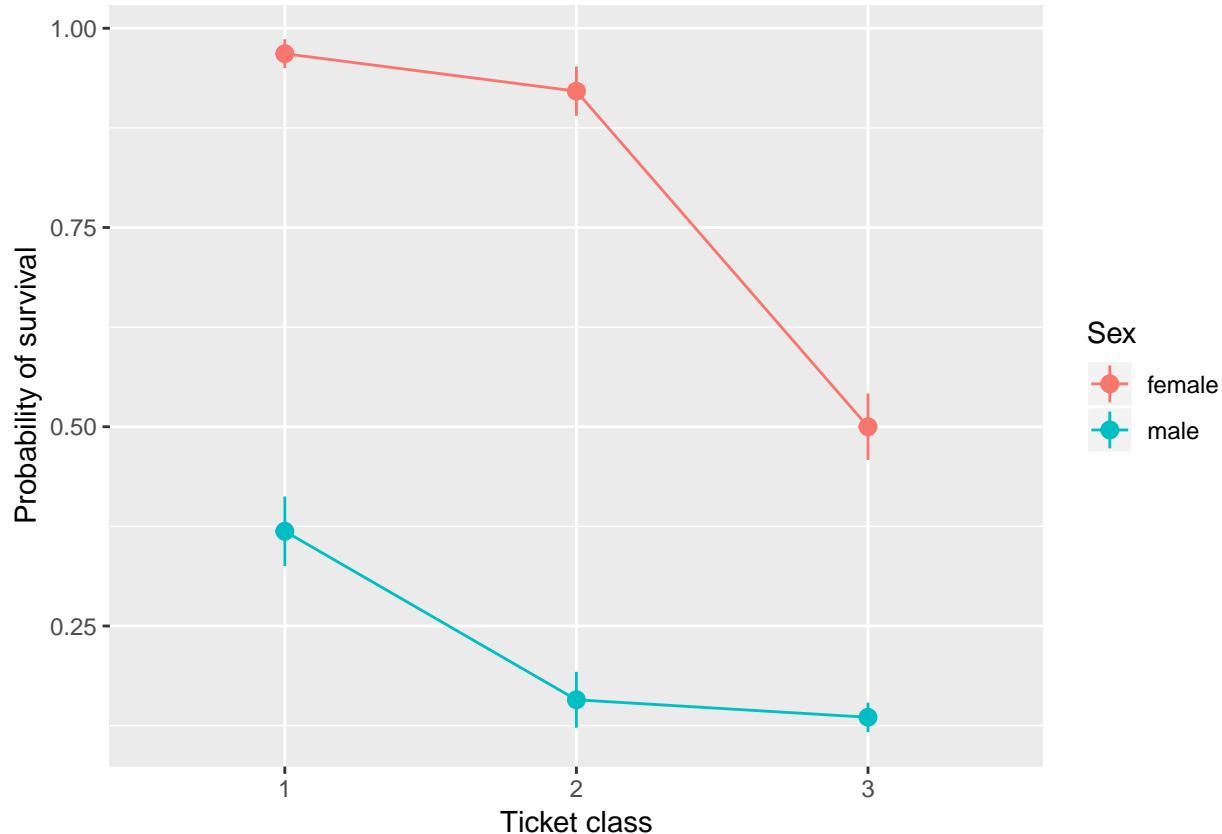
```

new.passengers = expand.grid(Pclass=1:3, Sex=c("female", "male"))

# this creates two vectors: $fit, which contains
# predicted probabilities and $se.fit
preds <- predict.glm(titanic.model,
                      newdata=new.passengers,
                      type="response",
                      se.fit=T)

new.passengers %>%
  mutate(fit = preds$fit,
        lower=fit - preds$se.fit,
        upper=fit + preds$se.fit) %>%
  ggplot(aes(factor(Pclass), fit,
             ymin=lower, ymax=upper,
             group=Sex, color=Sex)) +
  geom_pointrange() +
  geom_line() +
  xlab("Ticket class") +
  ylab("Probability of survival")

```



9.0.0.4 Evaluating logistic regression models

`glm` models don't provide an R^2 statistic, but it is possible to evaluate how well the model fits the data in other ways.

Although there are various pseudo- R^2 statistics available for `glm`; see <https://www.r-bloggers.com/evaluating-logistic-regression-models/>

One common technique, however, is to build a model using a 'training' dataset (sometimes a subset of your data) and evaluate how well this model predicts new observations in a 'test' dataset. See <http://r4ds.had.co.nz/model-assess.html> for an introduction.

10 Multilevel models

Psychological data often contains natural *groupings*. In intervention research, multiple patients may be treated by individual therapists, or children taught within classes, which are further nested within schools; in experimental research participants may respond on multiple occasions to a variety of stimuli.

Although disparate in nature, these groupings share a common characteristic: they induce *dependency* between the observations we make. That is, our data points are *not independently sampled* from one another.

When data are clustered in this way then multilevel, sometimes called linear mixed models, serve two purposes:

1. They overcome limitations of conventional models which assume that data *are* independently sampled (read a more detailed explanation of why handling non-independence properly matters)

2. They allow us to answer substantive questions about *sources of variation* in our data.

Repeated measures Anova and beyond

RM Anova is another technique which relaxes the assumption of independent sampling, and is widely used in psychology: it is common that participants make repeated responses which can be categorised by various experimental variables (e.g. time, condition).

However RM Anova is just a special case of a much wider family of models: linear mixed models, but one which makes a number of restrictions which can be inconvenient, inefficient, or unreasonable.

Substantive questions about variation

Additionally, rather than simply ‘managing’ the non-independence of observations — treating it is a kind of nuisance to be eliminated — mixed models can allow researchers to focus on the sources of variation in their data directly.

It can be of substantive interest to estimate how much variation in the outcome is due to different levels of the nested structure. For example, in a clinical trial researchers might want to know how much influence therapists have on their clients’ outcome: if patients are ‘nested’ within therapists then multilevel models can estimate the variation between therapists (the ‘therapist effect’) and variation ‘within’ therapists (i.e. variation between clients).

Fitting multilevel models in R

Use `lmer` and `glmer`

Although there are multiple R packages which can fit mixed-effects regression models, the `lmer` and `glmer` functions within the `lme4` package are the most frequently used, for good reason, and the examples below all use these two functions.

p values in multilevel models

For various philosophical and statistical reasons the author of `lme4`, Doug Bates, has always refused to display *p* values in the output from `lmer` (his reasoning is explained here).

That notwithstanding, many people have wanted to use the various methods to calculate *p* values for parameters in mixed models, and calculate F tests for effects and interactions. Various methods have been developed over the years which address at least some of Bates’ concerns, and these techniques have been implemented in R in the `lmerTest::` package. In particular, `lmerTest` implements an `anova` function for `lmer` models, which is very helpful.

10.0.0.1

Don’t worry! All you need to do is to load the `lmerTest` package rather than `lme4`. This loads updated versions of `lmer`, `glmer`, and extra functions for things like calculating *F* tests and the Anova table.

10.0.0.1 The `lmer` formula syntax

Specifying `lmer` models is very similar to the syntax for `lm`. The ‘fixed’ part of the model is exactly the same, with additional parts used to specify random intercepts, random slopes, and control the covariances of these random effects (there’s more on this in the troubleshooting section).

Random intercepts

The simplest model which allows a ‘random intercept’ for each level in the grouping looks like this:

```
lmer(outcome ~ predictors + (1 | grouping), data=df)
```

Here the outcome and predictors are specified in a formula, just as we did when using `lm()`. The only difference is that we now add a ‘random part’ to the model, in this case: `(1|grouping)`.

The `1` refers to an intercept, and so in English this part of the formula means ‘add a random intercept for each level of grouping’.

Random slopes

If we want to add a random slope to the model, we could adjust the random part like so:

```
lmer(outcome ~ predictor + (predictor | grouping), data=df)
```

This implicitly adds a random intercept too, so in English this formula says something like: let `outcome` be predicted by `predictor`; let variation in outcome to vary between levels of `grouping`, and also allow the effect of `predictor` to vary between levels of `grouping`.

The `lmer` syntax for the random part is very powerful, and allows complex combinations of random intercepts and slopes and control over how these random effects are allowed to correlate with one another. For a detailed guide to fitting two and three level models, with various covariance structures, see: <http://rpsychologist.com/r-guide-longitudinal-lme-lmer>

10.0.0.2 Are my effects fixed or random?

If you’re not sure which part of your model should be ‘fixed’ and which parts should be ‘random’ theres a more detailed explanation in this section.

Extending traditional RM Anova

As noted in the Anova cookbook section, repeated measures anova can be approximated using linear mixed models.

For example, repring the `sleepstudy` example, we can approximate a repeated measures Anova in which multiple measurements of `Reaction` time are taken on multiple `Days` for each `Subject`.

As we saw before, the traditional RM Anova model is:

```
sleep.ranova <- afex::aov_car(Reaction ~ Days + Error(Subject/(Days)), data=lme4::sleepstudy)
Registered S3 methods overwritten by 'car':
  method                 from
  influence.merMod      lme4
  cooks.distance.influence.merMod lme4
  dfbeta.influence.merMod    lme4
  dfbetas.influence.merMod   lme4
sleep.ranova
Anova Table (Type 3 tests)

Response: Reaction
  Effect        df      MSE          F ges p.value
1  Days 3.32, 56.46 2676.18 18.70 *** .29 <.0001
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '+' 0.1 ' ' 1

Sphericity correction method: GG
```

The equivalent lmer model is:

```

library(lmerTest)
sleep.lmer <- lmer(Reaction ~ factor(Days) + (1|Subject), data=lme4::sleepstudy)
anova(sleep.lmer)
Type III Analysis of Variance Table with Satterthwaite's method
  Sum Sq Mean Sq NumDF DenDF F value    Pr(>F)
factor(Days) 166235   18471      9     153  18.703 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

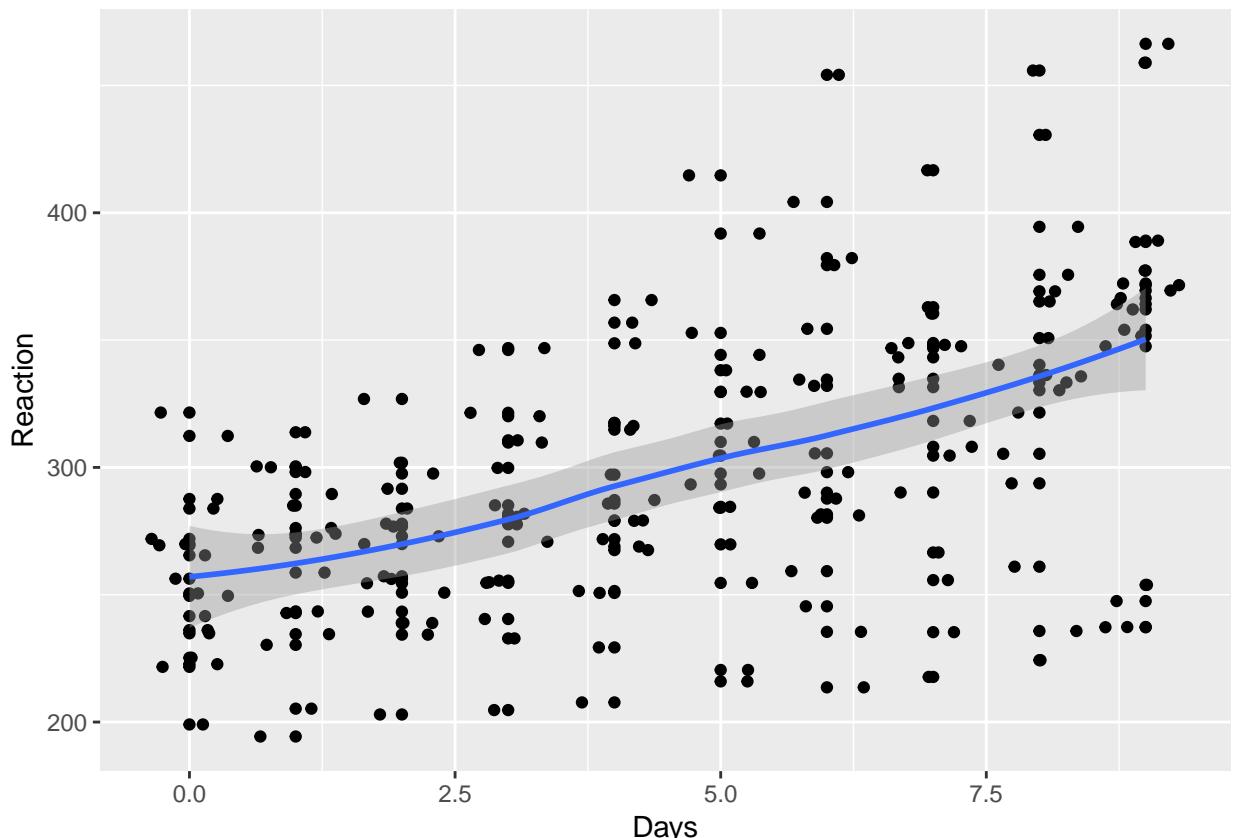
The following sections demonstrate just some of the extensions to RM Anova which are possible with multilevel models,

Fit a simple slope for Days

```

lme4::sleepstudy %>%
  ggplot(aes(Days, Reaction)) +
  geom_point() + geom_jitter() +
  geom_smooth()
`geom_smooth()` using method = 'loess' and formula 'y ~ x'

```



```

slope.model <- lmer(Reaction ~ Days + (1|Subject), data=lme4::sleepstudy)
anova(slope.model)
Type III Analysis of Variance Table with Satterthwaite's method
  Sum Sq Mean Sq NumDF DenDF F value    Pr(>F)
Days 162703 162703      1     161  169.4 < 2.2e-16 ***

```

```

---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
slope.model.summary <- summary(slope.model)
slope.model.summary$coefficients
  Estimate Std. Error      df t value    Pr(>|t|)
(Intercept) 251.40510  9.7467163 22.8102 25.79383 2.241351e-18
Days        10.46729  0.8042214 161.0000 13.01543 6.412601e-27

```

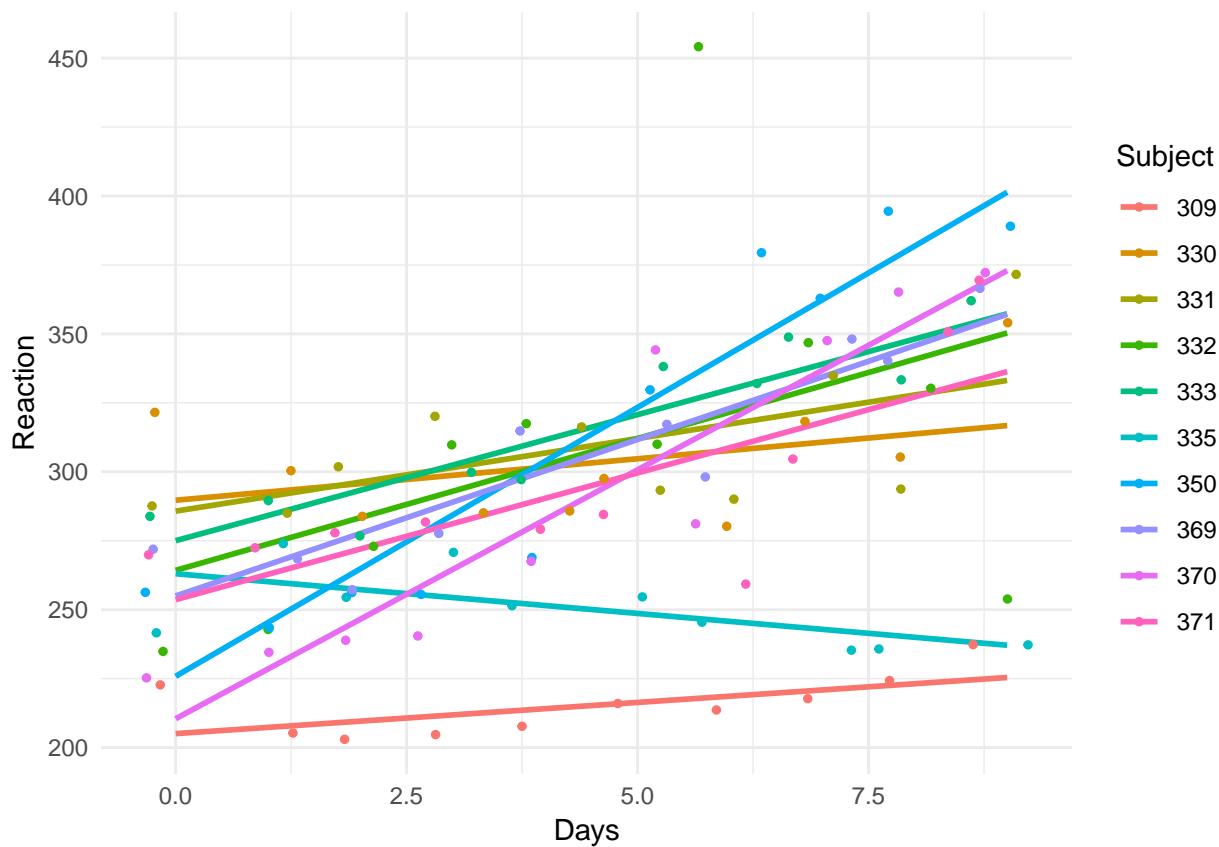
Allow the effect of sleep deprivation to vary for different participants

If we plot the data, it looks like sleep deprivation hits some participants worse than others:

```

set.seed(1234)
lme4:::sleepstudy %>%
  filter(Subject %in% sample(levels(Subject), 10)) %>%
  ggplot(aes(Days, Reaction, group=Subject, color=Subject)) +
  geom_smooth(method="lm", se=F) +
  geom_jitter(size=1) +
  theme_minimal()

```



If we wanted to test whether there was significant variation in the effects of sleep deprivation between subjects, by adding a random slope to the model.

The random slope allows the effect of Days to vary between subjects. So we can think of an overall slope (i.e. RT goes up over the days), from which individuals deviate by some amount (e.g. a resiliant person will have a negative deviation or residual from the overall slope).

Adding the random slope doesn't change the F test for `Days` that much:

```
random.slope.model <- lmer(Reaction ~ Days + (Days|Subject), data=lme4::sleepstudy)
anova(random.slope.model)
Type III Analysis of Variance Table with Satterthwaite's method
  Sum Sq Mean Sq NumDF DenDF F value    Pr(>F)
Days 30024 30024     1 16.995 45.843 3.273e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Nor the overall slope coefficient:

```
random.slope.model.summary <- summary(random.slope.model)
slope.model.summary$coefficients
  Estimate Std. Error      df t value    Pr(>|t|)
(Intercept) 251.40510 9.7467163 22.8102 25.79383 2.241351e-18
Days         10.46729 0.8042214 161.0000 13.01543 6.412601e-27
```

But we can use the `lmerTest::ranova()` function to show that there is statistically significant variation in slopes between individuals, using the likelihood ratio test:

```
lmerTest::ranova(random.slope.model)
ANOVA-like table for random-effects: Single term deletions

Model:
Reaction ~ Days + (Days | Subject)
  npar logLik   AIC   LRT Df Pr(>Chisq)
<none>          6 -871.81 1755.6
Days in (Days | Subject) 4 -893.23 1794.5 42.837 2  4.99e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

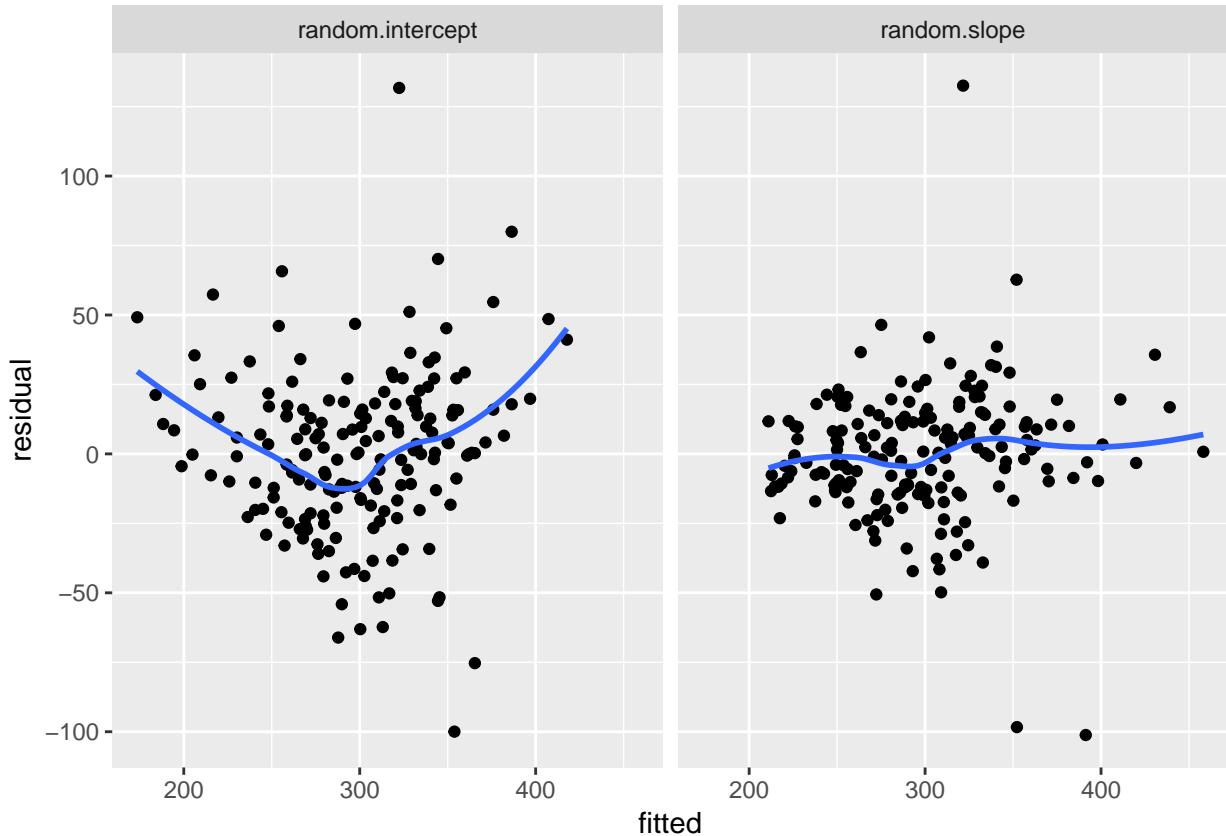
Because the random slope for `Days` is statistically significant, we know it improves the model. One way to see that improvement is to plot residuals (unexplained error for each datapoint) against predicted values. To extract residual and fitted values we use the `residuals()` and `predict()` functions. These are then combined in a `data_frame`, to enable us to use `ggplot` for the subsequent figures.

```
# create data frames containing residuals and fitted
# values for each model we ran above
a <- data_frame(
  model = "random.slope",
  fitted = predict(random.slope.model),
  residual = residuals(random.slope.model))
Warning: `data_frame()` is deprecated, use `tibble()``.
This warning is displayed once per session.
b <- data_frame(
  model = "random.intercept",
  fitted = predict(slope.model),
  residual = residuals(slope.model))

# join the two data frames together
residual.fitted.data <- bind_rows(a,b)
```

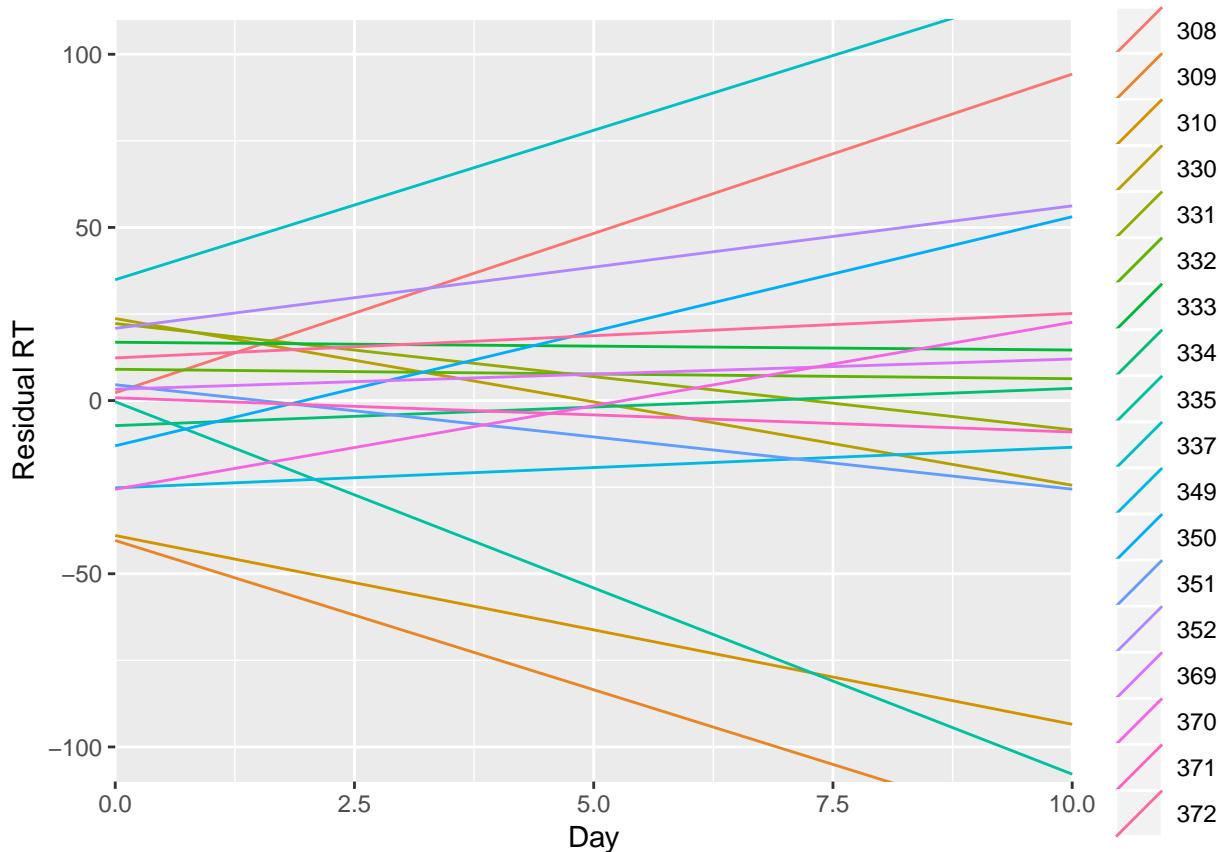
We can see that the residuals from the random slope model are much more evenly distributed across the range of fitted values, which suggests that the assumption of homogeneity of variance is met in the random slope model:

```
# plots residuals against fitted values for each model
residual.fitted.data %>%
  ggplot(aes(fitted, residual)) +
  geom_point() +
  geom_smooth(se=F) +
  facet_wrap(~model)
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



We can plot both of the random effects from this model (intercept and slope) to see how much the model expects individuals to deviate from the overall (mean) slope.

```
# extract the random effects from the model (intercept and slope)
ranef(random.slope.model)$Subject %>%
  # implicitly convert them to a dataframe and add a column with the subject number
  rownames_to_column(var="Subject") %>%
  # plot the intercept and slope values with geom_abline()
  ggplot(aes()) +
  geom_abline(aes(intercept=(Intercept), slope=Days, color=Subject)) +
  # add axis label
  xlab("Day") + ylab("Residual RT") +
  # set the scale of the plot to something sensible
  scale_x_continuous(limits=c(0,10), expand=c(0,0)) +
  scale_y_continuous(limits=c(-100, 100))
```



Inspecting this plot, there doesn't seem to be any strong correlation between the RT value at which an individual starts (their intercept residual) and the slope describing how they change over the days compared with the average slope (their slope residual).

That is, we can't say that knowing whether a person has fast or slow RTs at the start of the study gives us a clue about what will happen to them after they are sleep deprived: some people start slow and get faster; other start fast but suffer and get slower.

However we can explicitly check this correlation (between individuals' intercept and slope residuals) using the `VarCorr()` function:

```
VarCorr(random.slope.model)
Groups      Name      Std.Dev.   Corr
Subject (Intercept) 24.7366
          Days       5.9229  0.066
Residual           25.5918
```

The correlation between the random intercept and slopes is only 0.066, and so very low. We might, therefore, want to try fitting a model without this correlation. `lmer` includes the correlation by default, so we need to change the model formula to make it clear we don't want it:

```
uncorrelated.reffs.model <- lmer(
  Reaction ~ Days + (1 | Subject) + (0 + Days|Subject),
  data=lme4::sleepstudy)

VarCorr(uncorrelated.reffs.model)
Groups      Name      Std.Dev.
Subject (Intercept) 25.0499
```

Subject.1 Days	5.9887
Residual	25.5652

The variance components don't change much when we constrain the *covariance* of intercepts and slopes to be zero, and we can explicitly compare these two models using the `anova()` function, which is somewhat confusingly named because in this instance it is performing a likelihood ratio test to compare the two models:

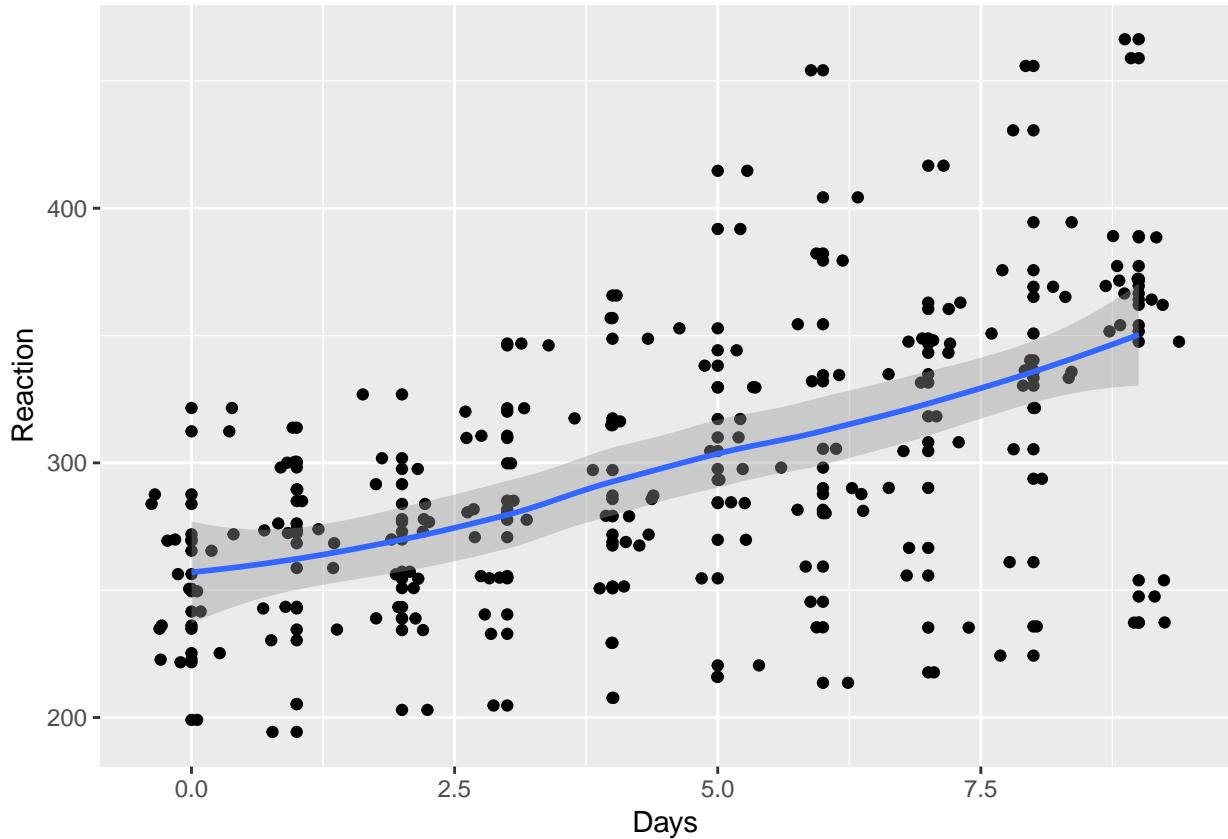
```
anova(random.slope.model, uncorrelated.reffs.model)
refitting model(s) with ML (instead of REML)
Data: lme4::sleepstudy
Models:
uncorrelated.reffs.model: Reaction ~ Days + (1 | Subject) + (0 + Days | Subject)
random.slope.model: Reaction ~ Days + (Days | Subject)
      Df    AIC    BIC logLik deviance Chisq Chi Df
uncorrelated.reffs.model 5 1762.0 1778.0 -876.00  1752.0
random.slope.model       6 1763.9 1783.1 -875.97  1751.9 0.0639     1
Pr(>Chisq)
uncorrelated.reffs.model
random.slope.model      0.8004
```

Model fit is not significantly worse with the constrained model, so for parsimony's sake we prefer it to the more complex model.

Fitting a curve for the effect of Days

In theory, we could also fit additional parameters for the effect of Days, although a combined smoothed line plot/scatterplot indicates that a linear function fits the data reasonably well.

```
lme4::sleepstudy %>%
  ggplot(aes(Days, Reaction)) +
  geom_point() + geom_jitter() +
  geom_smooth()
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



If we insisted on testing a curved (quadratic) function of Days, we could:

```
quad.model <- lmer(Reaction ~ Days + I(Days^2) + (1|Subject), data=lme4::sleepstudy)
quad.model.summary <- summary(quad.model)
quad.model.summary$coefficients
  Estimate Std. Error      df   t value    Pr(>|t|)
(Intercept) 255.4493728 10.4656347 30.04058 24.408398 2.299848e-21
Days         7.4340850  2.9707976 160.00001  2.502387 1.334036e-02
I(Days^2)    0.3370223  0.3177733 160.00001  1.060575 2.904815e-01
```

Here, the p value for $I(Days^2)$ is not significant, suggesting (as does the plot) that a simple slope model is sufficient.

Variance partition coefficients and intraclass correlations

The purpose of multilevel models is to partition variance in the outcome between the different groupings in the data.

For example, if we make multiple observations on individual participants we partition outcome variance between individuals, and the residual variance.

We might then want to know what *proportion* of the total variance is attributable to variation within-groups, or how much is found between-groups. This statistic is termed the variance partition coefficient VPC, or intraclass correlation.

We calculate the VPC with some simple arithmetic on the variance estimates from the lmer model. We can extract the variance estimates from the VarCorr function:

```

random.intercepts.model <- lmer(Reaction ~ Days + (1|Subject), data=lme4::sleepstudy)
VarCorr(random.intercepts.model)
Groups   Name      Std.Dev.
Subject (Intercept) 37.124
Residual           30.991

```

And we can test the variance parameter using the `rand()` function:

```

rand(random.intercepts.model)
ANOVA-like table for random-effects: Single term deletions

Model:
Reaction ~ Days + (1 | Subject)
      npar logLik    AIC    LRT Df Pr(>Chisq)
<none>        4 -893.23 1794.5
(1 | Subject)  3 -946.83 1899.7 107.2  1 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Helpfully, if we convert the result of `VarCorr` to a dataframe, we are provided with the columns `vcov` which stands for `variance or covariance`, as well as the `sdcor` (standard deviation or correlation) which is provided in the printed summary:

```

VarCorr(random.intercepts.model) %>%
  as_data_frame()
Warning: `as_data_frame()` is deprecated, use `as_tibble()` (but mind the new semantics).
This warning is displayed once per session.
# A tibble: 2 x 5
  grp     var1     var2    vcov sdcor
  <chr>   <chr>   <chr> <dbl> <dbl>
1 Subject (Intercept) <NA>  1378.  37.1
2 Residual <NA>     <NA>   960.  31.0

```

The variance partition coefficient is simply the variance at a given level of the model, divided by the total variance (the sum of the variance parameters). So we can write:

```

VarCorr(random.intercepts.model) %>%
  as_data_frame() %>%
  mutate(icc=vcov/sum(vcov)) %>%
  select(grp, icc)
# A tibble: 2 x 2
  grp       icc
  <chr>   <dbl>
1 Subject  0.589
2 Residual 0.411

```

Intraclass correlations were computed from the mixed effects mode. 59% of the variation in outcome was attributable to differences between subjects, $\chi^2(1) = 107$, $p < .001$.

[It's not straightforward to put an confidence interval around the VPC estimate from an lmer model. If this is important to you, you should explore re-fitting the same model in a Bayesian framework]

3 level models with ‘partially crossed’ random effects

The `lme4::InstEval` dataset records University lecture evaluations by students at ETH Zurich. The variables include:

- **s** a factor with levels 1:2972 denoting individual students.
- **d** a factor with 1128 levels from 1:2160, denoting individual professors or lecturers.
- **studage** an ordered factor with levels $2 < 4 < 6 < 8$, denoting student's "age" measured in the semester number the student has been enrolled.
- **lectage** an ordered factor with 6 levels, $1 < 2 < \dots < 6$, measuring how many semesters back the lecture rated had taken place.
- **service** a binary factor with levels 0 and 1; a lecture is a "service", if held for a different department than the lecturer's main one.
- **dept** a factor with 14 levels from 1:15, using a random code for the department of the lecture.
- **y** a numeric vector of ratings of lectures by the students, using the discrete scale 1:5, with meanings of 'poor' to 'very good'.

For convenience, in this example we take a subsample of the (fairly large) dataset:

```
set.seed(1234)
lectures <- sample_n(lme4::InstEval, 10000)
```

We run a model without any predictors, but respecting the clustering in the data, in the example below. This model is a three-level random intercepts model, which splits the variance between lecturers, students, and the residual variance. Because, in some cases, some of the same students provide data on a particular lecturer these data are 'partially crossed' (the alternative would be to sample different students for each lecturer).

```
lectures.model <- lmer(y~(1|d)+(1|s), data=lectures)
summary(lectures.model)
Linear mixed model fit by REML. t-tests use Satterthwaite's method [
lmerModLmerTest]
Formula: y ~ (1 | d) + (1 | s)
Data: lectures

REML criterion at convergence: 33063.8

Scaled residuals:
    Min      1Q  Median      3Q     Max 
-2.62960 -0.74245  0.03516  0.76510  2.62158

Random effects:
 Groups   Name        Variance Std.Dev. 
 s        (Intercept) 0.1119   0.3345  
 d        (Intercept) 0.2717   0.5213  
 Residual           1.3656   1.1686  
Number of obs: 10000, groups: s, 2707; d, 1065

Fixed effects:
            Estimate Std. Error       df t value Pr(>|t|)    
(Intercept) 3.219e+00 2.381e-02 1.023e+03 135.2 <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As before, we can extract only the variance components from the model, and look at the ICC:

```
VarCorr(lectures.model) %>% as_data_frame() %>%
  mutate(icc=vcov/sum(vcov)) %>%
```

```

  select(grp, vcov, icc)
# A tibble: 3 x 3
  grp      vcov     icc
  <chr>    <dbl>   <dbl>
1 s        0.112  0.0640
2 d        0.272  0.155
3 Residual 1.37   0.781

```

And we can add predictors to the model to see if they help explain student ratings:

```

lectures.model.2 <- lmer(y~service*dept+(1|d)+(1|s), data=lectures)
anova(lectures.model.2)
Type III Analysis of Variance Table with Satterthwaite's method
  Sum Sq Mean Sq NumDF DenDF F value    Pr(>F)
service      3.001  3.0006     1 7457.1  2.2026 0.137820
dept         21.306  1.6389    13 1157.3  1.2031 0.270909
service:dept 41.546  3.1958    13 6638.0  2.3459 0.004052 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Here we can see the `service` variable does predict evaluations, and we can use the model to estimate the mean and SE for service == 1 or service == 0 (see also the sections on multiple comparisons, followup contrasts, and doing followup contrasts with lmer models for more options here):

```

service.means <- emmeans::emmeans(lectures.model.2, 'service')
Note: D.f. calculations have been disabled because the number of observations exceeds 3000.
To enable adjustments, set emm_options(pbkrttest.limit = 10000) or larger,
but be warned that this may result in large computation time and memory use.
Note: D.f. calculations have been disabled because the number of observations exceeds 3000.
To enable adjustments, set emm_options(lmerTest.limit = 10000) or larger,
but be warned that this may result in large computation time and memory use.
NOTE: Results may be misleading due to involvement in interactions
service.means %>%
  broom::tidy() %>%
  select(service, estimate, std.error) %>%
  pander

```

	service	estimate	std.error
	0	3.257	0.02911
	1	3.196	0.03914

Or change the proportions of variance components at each level (they don't, much, in this instance):

```

VarCorr(lectures.model.2) %>% as_data_frame() %>%
  mutate(icc=vcov/sum(vcov)) %>%
  select(grp, vcov, icc)
# A tibble: 3 x 3
  grp      vcov     icc
  <chr>    <dbl>   <dbl>
1 s        0.112  0.0643
2 d        0.262  0.151
3 Residual 1.36   0.785

```

Contrasts and followup tests using `lmer`

Many of the contrasts possible after `lm` and Anova models are also possible using `lmer` for multilevel models.

Let's say we repeat one of the models used in a previous section, looking at the effect of `Days` of sleep deprivation on reaction times:

```
m <- lmer(Reaction~factor(Days)+(1|Subject), data=lme4::sleepstudy)
anova(m)
Type III Analysis of Variance Table with Satterthwaite's method
      Sum Sq Mean Sq NumDF DenDF F value    Pr(>F)
factor(Days) 166235   18471      9     153  18.703 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

10.0.0.2.1

We can see a significant effect of `Days` in the Anova table, and want to compute followup tests.

To first estimate cell means and create an `emmeans` object, you can use the `emmeans()` function in the `emmeans::` package:

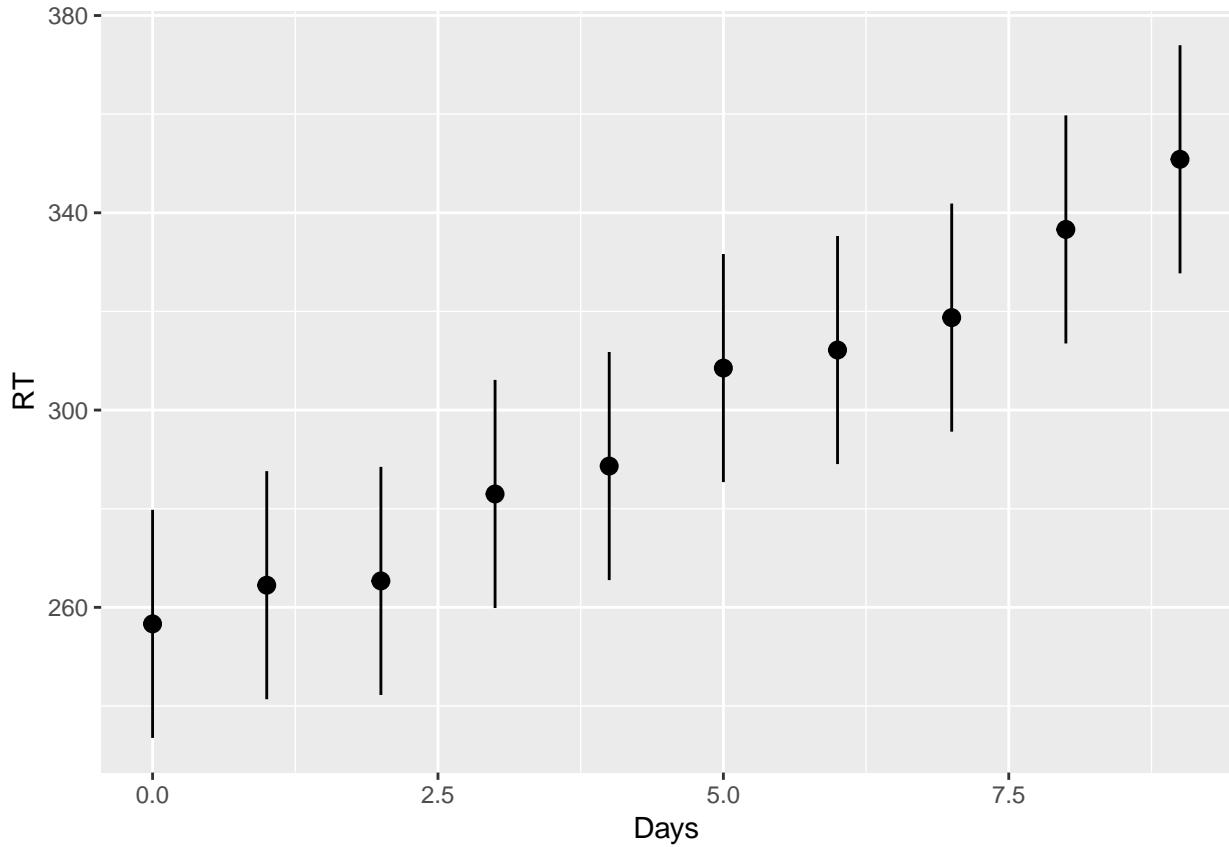
```
m.emm <- emmeans(m, "Days")
m.emm
Days emmean    SE df lower.CL upper.CL
 0    257 11.5 42     234     280
 1    264 11.5 42     241     288
 2    265 11.5 42     242     288
 3    283 11.5 42     260     306
 4    289 11.5 42     266     312
 5    309 11.5 42     285     332
 6    312 11.5 42     289     335
 7    319 11.5 42     296     342
 8    337 11.5 42     314     360
 9    351 11.5 42     328     374
```

```
Degrees-of-freedom method: kenward-roger
Confidence level used: 0.95
```

It might be nice to extract these estimates and plot them:

```
m.emm.df <-
  m.emm %>%
  broom::tidy()

m.emm.df %>%
  ggplot(aes(Days, estimate, ymin=conf.low, ymax=conf.high)) +
  geom_pointrange() +
  ylab("RT")
```



If we wanted to compare each day against every other day (i.e. all the pairwise comparisons) we can use `contrast()`:

```
# results not shown to save space
contrast(m.emm, 'tukey') %>%
  broom::tidy() %>%
  head(6)
```

Or we might want to see if there was a significant change between any specific day and baseline:

```
# results not shown to save space
contrast(m.emm, 'trt.vs.ctrl') %>%
  broom::tidy() %>%
  head %>%
  pander
```

Perhaps more interesting in this example is to check the polynomial contrasts, to see if there was a linear or quadratic change in RT over days:

```
# results not shown to save space
contrast(m.emm, 'poly') %>%
  broom::tidy() %>%
  head(3) %>%
  pander(caption="The first three polynomial contrasts. Note you'd have to have quite a fancy theory to")
```

Troubleshooting

Convergence problems and simplifying the random effects structure

10.0.0.3

It's common, when variances and covariances are close to zero, that `lmer` has trouble fitting your model. The solution is to simplify complex models, removing or constraining some random effects.

For example, in an experiment where you have multiple `stimuli` and different experimental conditions, with many repeated `trials`, you might end up with data like this:

```
df %>%
  head()
# A tibble: 6 x 5
  trial condition block subject    RT
  <int>     <int> <int>   <int> <dbl>
1     1         1     1       1  299.
2     2         1     1       1  300.
3     3         1     1       1  300.
4     4         1     1       1  300.
5     5         1     1       1  301.
6     6         1     1       1  301.
```

Which you could model with `lmer` like this:

```
m1 <- lmer(RT ~ block * trial * condition + (block+condition|subject), data=df)
boundary (singular) fit: see ?isSingular
```

You can list the random effects from the model using the `VarCorr` function:

```
VarCorr(m1)
Groups   Name      Std.Dev.   Corr
subject (Intercept) 1.12152879
          block      0.00052841  1.000
          condition  0.00673767 -1.000 -1.000
Residual           1.00551111
```

As `VarCorr` shows, this model estimates:

- random intercepts for `subject`,
- random slopes for `trial` and `condition`, and
- three covariances between these random effects.

If these covariances are very close to zero though, as is often the case, this can cause convergence issues, especially if insufficient data are available.

If this occurs, you might want to simplify the model. For example, to remove all the covariances between random effects you might rewrite the model this way:

```
m2 <- lmer(RT ~ block * trial * condition +
  (1|subject) +
  (0+block|subject) +
  (0+condition|subject), data=df)
VarCorr(m2)
```

To remove only covariances with the intercept:

```
m3 <- lmer(RT ~ block * trial * condition +
  (1|subject) +
```

```
(0+block+condition|subject), data=df)
```

```
VarCorr(m3)
```

In general, the recommendation is to try and fit a full random effects structure, and simplify it by removing the least theoretically plausible parameters. See:

- This tutorial on mixed models in linguistics: http://www.bodowinter.com/tutorial/bw_LME_tutorial2.pdf
- Barr et al. [2013], which recommends you ‘keep it maximal’, meaning that you should keep all random effects terms, including covariances, where this is possible.

See this page for lots more examples of more complex mixed models

Bayesian multilevel models

Complex models with many random effects it can be challenging to fit using standard software [see `eager2017mixed` and Gelman et al., 2014]. Many authors have noted that a Bayesian approach to model fitting can be advantageous for multilevel models.

A brief example of fitting multilevel models via MCMC is given in this section: Bayes via MCMC

```
library(tidyverse)
Registered S3 methods overwritten by 'ggplot2':
  method      from
  [.quosures    rlang
  c.quosures    rlang
  print.quosures rlang
Registered S3 method overwritten by 'rvest':
  method      from
  read_xml.response xml2
-- Attaching packages ----- tidyverse 1.2.1 --
v ggplot2 3.1.1     v purrr   0.3.2
v tibble   2.1.1     v dplyr    0.8.1
v tidyr    0.8.3     v stringr  1.4.0
v readr    1.3.1     v forcats  0.4.0
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
library(broom)
library(pander)
source('diagram.R')
```

11 Mediation and covariance modelling

Mediation

Mediation is a complex topic, and the key message to take on — before starting to analyse your data — is that mediation analyses make many strong assumptions about the data. These assumptions can often be pretty unreasonable, when spelled out, so be cautious in the interpretation of your data.

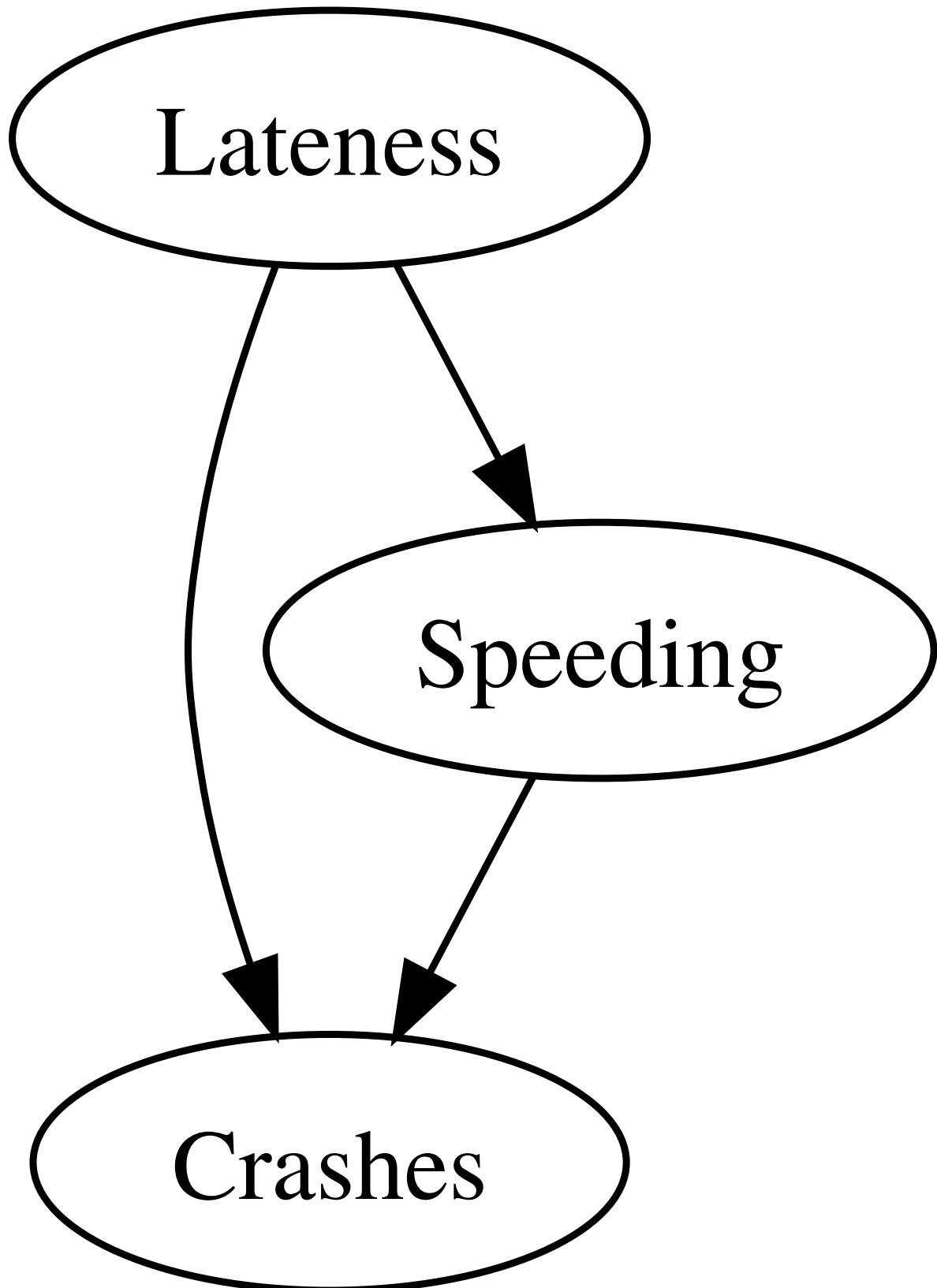
Put differently, mediation is a correlational technique aiming to provide a causal interpretation of data; caveat emptor.

Mediation with multiple regression

One common (if outdated) way to analyse mediation is via the 3 steps described by Baron and Kenny [1986] (also see Zhao et al.).

Let's say we have a hypothesised situation such as this:

```
knit_gv("  
Lateness -> Crashes  
Lateness -> Speeding  
Speeding -> Crashes  
")
```



Baron and Kenny propose 3 steps to establishing mediation. These steps correspond to three separate regression models:

Mediation Steps

11.0.0.1 Step 1 (check distal variable predicts mediator)

That is, show Lateness predicts Crashes

11.0.0.2 Step 2 (check distal variable predict mediator)

That is, show Lateness predicts Speeding

11.0.0.3 Step 3 (check for mediation)

That is, show Speeding predicts Crashes, controlling for Lateness

An additional step, which allows us to test whether the effect is *completely* mediated, also uses the final regression model:

11.0.0.4 Step 4 (check for total mediation)

That is, check if Lateness still predicts crashes, controlling for Lateness

Mediation example after Baron and Kenny

Using simulated data, we can work through the steps.

```
smash %>% glimpse
Observations: 200
Variables: 4
$ person <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16...
$ lateness <int> 11, 12, 9, 8, 11, 4, 11, 7, 8, 11, 9, 10, 9, 11, 13, ...
$ speed <dbl> 48.88524, 43.07030, 33.72812, 44.17897, 51.22378, 40....
$ crashes <int> 15, 9, 12, 20, 25, 13, 22, 14, 22, 18, 11, 16, 19, 15...
```

Step 1: does lateness predict crashes?

```
step1 <- lm(crashes ~ lateness, data=smash)
tidy(step1) %>% pander()
```

term	estimate	std.error	statistic	p.value
(Intercept)	12.15	1.204	10.09	1.365e-19
lateness	0.4448	0.1125	3.953	0.0001074

Step 2: Does lateness predict speed?

```
step2 <- lm(speed ~ lateness, data=smash)
tidy(step2, conf.int = T) %>% pander()
```

term	estimate	std.error	statistic	p.value	conf.low	conf.high
(Intercept)	33.42	2.275	14.69	1.563e-33	28.93	37.9
lateness	0.515	0.2126	2.422	0.01633	0.09573	0.9343

The coefficient for `lateness` is statistically significant, so we would say yes.

Step 3: Does speed predict crashes, controlling for lateness?

```
step3 <- lm(crashes ~ lateness+speed, data=smash)
tidy(step3) %>% pander()
```

term	estimate	std.error	statistic	p.value
(Intercept)	2.542	1.465	1.735	0.08427
lateness	0.2967	0.09611	3.088	0.002309
speed	0.2875	0.03166	9.083	1.122e-16

The coefficient for speed is statistically significant, so we can say mediation does occur.

Step 4: In the same model, does lateness predict crashes, controlling for speed? That is to say, is the mediation via speed *total*?

Here, the coefficient is still statistically significant. According to the Baron and Kenny steps, this would indicate the mediation is *partial*, although the fact the p value falls one side or another of .05 is not necessarily the best way to express this (see below for ways to calculate the proportion of the effect which is mediated).

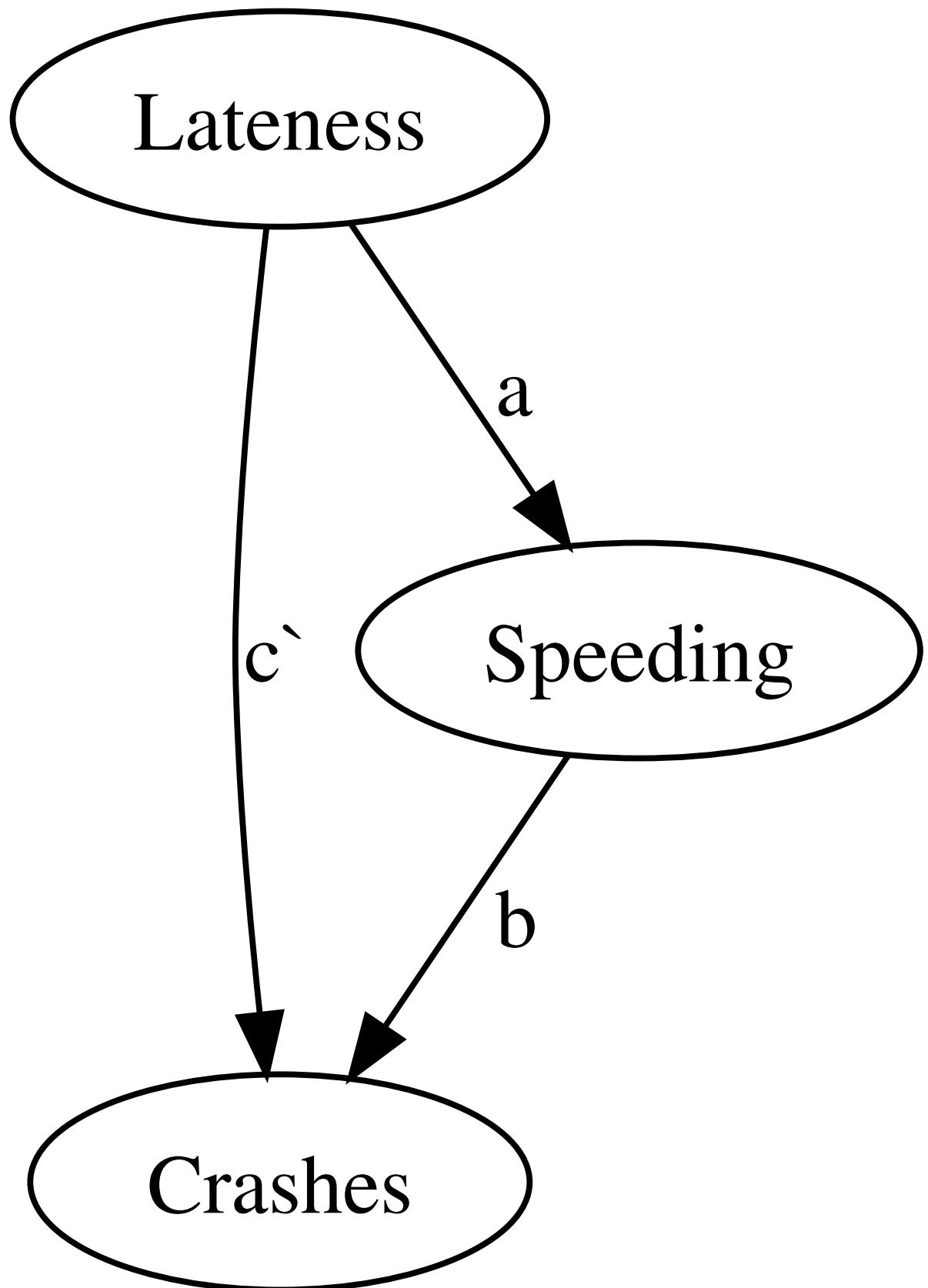
We should also be concerned here with the degree to which predictor and mediator are measured with error — if they are noisy measures, then the proportion of the effect which appears to be mediated will be reduced artificially (see the SEM chapter for more on this).

Testing the indirect effect

Baron and Kenny also introduced conventions for labelling some of the coefficients from the regressions described above.

Specifically, the described **a** as the path from the predictor to the mediator, **b** as the path from the mediator to the outcome, and **c'** (**c prime**) as the path from predictor to outcome, controlling for the mediator. As shown here:

```
knit_gv("
Lateness -> Crashes[label='c`']
Lateness -> Speeding[label=a]
Speeding -> Crashes[label=b]
")
```



Subsequent authors wished to provide a test for whether the path through **a** and **b** — the indirect effect — was statistically significant. Preacher and Hayes published SPSS macros for computing this indirect effect and providing a non-parametric (bootstrapped) test of this term. The same approach is now implemented in a number of R packages.

The `mediation::mediate` function accepts the 2nd and 3rd regression models from the ‘Baron and Kenny’ steps, along with arguments which identify which variables are the predictor and the mediator. From this, the function calculates the indirect effect, and the proportion of the total effect mediated. This is accompanied by a bootstrapped standard-error, and associated p value.

For example, using the models we ran above, we can say:

```
set.seed(1234)
crashes.mediation <- mediation::mediate(step2, step3, treat = "lateness", mediator = "speed")
summary(crashes.mediation)

Causal Mediation Analysis

Quasi-Bayesian Confidence Intervals

      Estimate 95% CI Lower 95% CI Upper p-value
ACME       0.1477    0.0237     0.28   0.018 *
ADE        0.2998    0.1166     0.50  <2e-16 ***
Total Effect 0.4475    0.2293     0.66  <2e-16 ***
Prop. Mediated 0.3295    0.0741     0.60   0.018 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Sample Size Used: 200

Simulations: 1000
```

From this output, we can see that the indirect effect is statistically significant, and that around half of the total effect is mediated via speed. Because lateness in itself is not a plausible cause of a crash, this suggest that other factors (perhaps distraction, inattention) might be important in mediating this residual direct effect.

11.1 Mediation using Path models

An even more flexible approach to mediation can be taken using path models, a type of structural equation model which are covered in more detail in the next section.

Using the `lavaan` package, path/SEM models can specify multiple variables to be outcomes, and fit these models simultaneously. For example, we can fit both step 2 and step 3 in a single model, as in the example below:

```
library(lavaan)
This is lavaan 0.6-3
lavaan is BETA software! Please report any bugs.

smash.model <- '
  crashes ~ speed + lateness
  speed ~ lateness
'
```

```

smash.model.fit <- sem(smash.model, data=smash)
summary(smash.model.fit)
lavaan 0.6-3 ended normally after 19 iterations

Optimization method                           NLMINB
Number of free parameters                   5
Number of observations                      200
Estimator                                    ML
Model Fit Test Statistic                  0.000
Degrees of freedom                         0
Minimum Function Value                    0.000000000000000

Parameter Estimates:

Information                                Expected
Information saturated (h1) model           Structured
Standard Errors                            Standard

Regressions:
            Estimate Std.Err z-value P(>|z|)
crashes ~
  speed          0.288   0.031   9.152   0.000
  lateness       0.297   0.095   3.111   0.002
speed ~
  lateness       0.515   0.212   2.434   0.015

Variances:
            Estimate Std.Err z-value P(>|z|)
.crashes      18.190   1.819  10.000   0.000
.speed        92.135   9.214  10.000   0.000

```

The summary output gives us coefficients which correspond to the regression coefficients in the step 2 and step 3 models — but this time, from a single model.

We can also use `lavaan` to compute the indirect effects by labelling the relevant parameters, using the `*` and `:=` operators. See the `lavaan` syntax guide for mediation for more detail.

Note that the `*` operator does not have the same meaning as in formulas for linear models in R — in `lavaan`, it means ‘apply a constraint’.

```

smash.model <- '
  crashes ~ B*speed + C*lateness
  speed ~ A*lateness

  # computed parameters, see http://lavaan.ugent.be/tutorial/mediation.html
  indirect := A*B
  total := C + (A*B)
  proportion := indirect/total
'

smash.model.fit <- sem(smash.model, data=smash)
summary(smash.model.fit)
lavaan 0.6-3 ended normally after 19 iterations

```

```

Optimization method          NLINMB
Number of free parameters    5
Number of observations       200
Estimator                   ML
Model Fit Test Statistic   0.000
Degrees of freedom          0
Minimum Function Value     0.0000000000000000

Parameter Estimates:

Information                         Expected
Information saturated (h1) model      Structured
Standard Errors                      Standard

Regressions:
            Estimate Std.Err z-value P(>|z|)
crashes ~
  speed      (B)  0.288   0.031   9.152  0.000
  lateness   (C)  0.297   0.095   3.111  0.002
speed ~
  lateness   (A)  0.515   0.212   2.434  0.015

Variances:
            Estimate Std.Err z-value P(>|z|)
.crashes      18.190   1.819  10.000  0.000
.speed        92.135   9.214  10.000  0.000

Defined Parameters:
            Estimate Std.Err z-value P(>|z|)
indirect      0.148   0.063   2.353  0.019
total         0.445   0.112   3.973  0.000
proportion    0.333   0.121   2.756  0.006

```

We can again get a bootstrap interval for the indirect effect, and print a table of just these computed effects like so:

```

set.seed(1234)
smash.model.fit <- sem(smash.model, data=smash, test="bootstrap", bootstrap=100)

parameterEstimates(smash.model.fit) %>%
  filter(op == ":=") %>%
  select(label, est, contains("ci")) %>%
  pander::pander()

```

label	est	ci.lower	ci.upper
indirect	0.1481	0.02472	0.2715
total	0.4448	0.2254	0.6643
proportion	0.3329	0.09614	0.5697

Comparing these results with the `mediation::mediate()` output, we get similar results. In both cases, it's

possible to increase the number of bootstrap resamples if needed to increase the precision of the interval (the default is 1000, but 5000 might be a good target for publication).

Covariance modelling

The CFA examples here were adapted from a guide originally produced by Jon May

This section covers path analysis (path models), confirmatory factor analysis (CFA) and structural equation modelling (SEM). You are encouraged to work through the path models and CFA sections, and especially the material on assessing model fit, before tackling SEM.

Before you start this either section make sure you have the `lavaan` package installed (see installing packages).

```
install.packages(lavaan)
```

And load the package to make all the functions available with minimal typing:

```
library(lavaan)
```

Path models

Path models are an extension of linear regression, but where multiple observed variables can be considered as ‘outcomes’.

Because the terminology of outcomes v.s. predictors breaks down when variables can be both outcomes and predictors at the same time, it’s normal to distinguish instead between:

- *Exogenous* variables: those which are not predicted by any other
- *Endogenous* variables: variables which do have predictors, and may or may not predict other variables

Defining a model

To define a path model, `lavaan` requires that you specify the relationships between variables in a text format. A full guide to this lavaan model syntax is available on the project website.

For path models the format is very simple, and resembles a series of linear models, written over several lines, but in text rather than as a model formula:

```
# define the model over multiple lines for clarity
mediation.model <- "
  y ~ x + m
  m ~ x
"
```

In this case the `~` symbol just means ‘regressed on’ or ‘is predicted by’. The model in the example above defines that our outcome `y` is predicted by both `x` and `m`, and that `x` also predicts `m`. You might recognise this as a mediation model.

Make sure you include the closing quote symbol, and also be careful when running the code which defines the model. RStudio can sometimes get confused and only run some of the lines, leading to errors. The simplest solution is to select the entire block explicitly and run that.

To fit the model we pass the model specification and the data to the `sem()` function:

```
mediation.fit <- sem(mediation.model, data=mediation.df)
```

As we did for linear regression models, we have saved the model fit object into a variable, here named `mediation.fit`.

To display the model results we can use `summary()`. The key section of the output to check is the table listed ‘Regressions’, which lists the regression parameters for the predictors for each of the endogenous variables.

```
summary(mediation.fit)
lavaan 0.6-3 ended normally after 12 iterations

Optimization method                           NLMINB
Number of free parameters                      5
Number of observations                         200
Estimator                                     ML
Model Fit Test Statistic                     0.000
Degrees of freedom                            0
Minimum Function Value                      0.000000000000000

Parameter Estimates:

Information                                         Expected
Information saturated (h1) model                Structured
Standard Errors                                    Standard

Regressions:
              Estimate Std.Err z-value P(>|z|)
y ~
  x          0.166   0.075  2.198  0.028
  m          0.190   0.070  2.721  0.007
m ~
  x          0.530   0.067  7.958  0.000

Variances:
             Estimate Std.Err z-value P(>|z|)
.y          0.967   0.097 10.000  0.000
.m          0.993   0.099 10.000  0.000
```

From this table we can see that both `x` and `m` are significant predictors of `y`, and that `x` also predicts `m`. This implies that mediation is taking place, but see the mediation chapter for details of testing indirect effects in `lavaan`.

11.1.0.1 Where’s the intercept?

Path analysis is part of the set of techniques often termed ‘covariance modelling’. As the name implies the primary focus here is the relationships between variables, and less so the mean-structure of the variables. In fact, by default the software first creates the covariance matrix of all the variables in the model, and the fit is based only on these values, plus the sample sizes (in early SEM software you typically had to provide the covariance matrix directly, rather than working with the raw data).

Nonetheless, because path analysis is an extension of regression techniques it is possible to request that intercepts are included in the model, and means estimated, by adding `meanstructure=TRUE` to the `sem()` function (see the `lavaan` manual for details).

In the output below we now also see a table labelled ‘Intercepts’ which gives the mean values of each variable *when its predictors are zero* (just like in linear regression):

```

mediation.fit.means <- sem(mediation.model,
                            meanstructure=T,
                            data=mediation.df)

summary(mediation.fit.means)
lavaan 0.6-3 ended normally after 16 iterations

Optimization method                           NLMINB
Number of free parameters                      7
Number of observations                         200
Estimator                                     ML
Model Fit Test Statistic                     0.000
Degrees of freedom                           0
Minimum Function Value                      0.0000000000000000

Parameter Estimates:

Information                                         Expected
Information saturated (h1) model                 Structured
Standard Errors                                    Standard

Regressions:
            Estimate Std.Err z-value P(>|z|)
y ~
  x          0.166   0.075   2.198   0.028
  m          0.190   0.070   2.721   0.007
m ~
  x          0.530   0.067   7.958   0.000

Intercepts:
            Estimate Std.Err z-value P(>|z|)
.y          10.629   0.362  29.323   0.000
.m          5.097   0.070  72.298   0.000

Variances:
            Estimate Std.Err z-value P(>|z|)
.y          0.967   0.097  10.000   0.000
.m          0.993   0.099  10.000   0.000

```

11.1.0.2 Tables of model coefficients

If you want to present results from these models in table format, the `parameterEstimates()` function is useful to extract the relevant numbers as a dataframe. We can then manipulate and present this table as we would any other dataframe.

In the example below we extract the parameter estimates, select only the regression parameters (~) and remove some of the columns to make the final output easier to read:

```

parameterEstimates(mediation.fit.means) %>%
  as_tibble() %>%
  filter(op=="~") %>%
  mutate(Term=paste(lhs, op, rhs)) %>%

```

```

rename(estimate=est,
      p=pvalue) %>%
select(Term, estimate, z, p) %>%
pander::pander(caption="Regression parameters from `mediation.fit`")

```

Table 45: Regression parameters from `mediation.fit`

Term	estimate	z	p
y ~ x	0.1657	2.198	0.02797
y ~ m	0.1899	2.721	0.006515
m ~ x	0.5298	7.958	1.776e-15

11.1.0.3 Diagrams

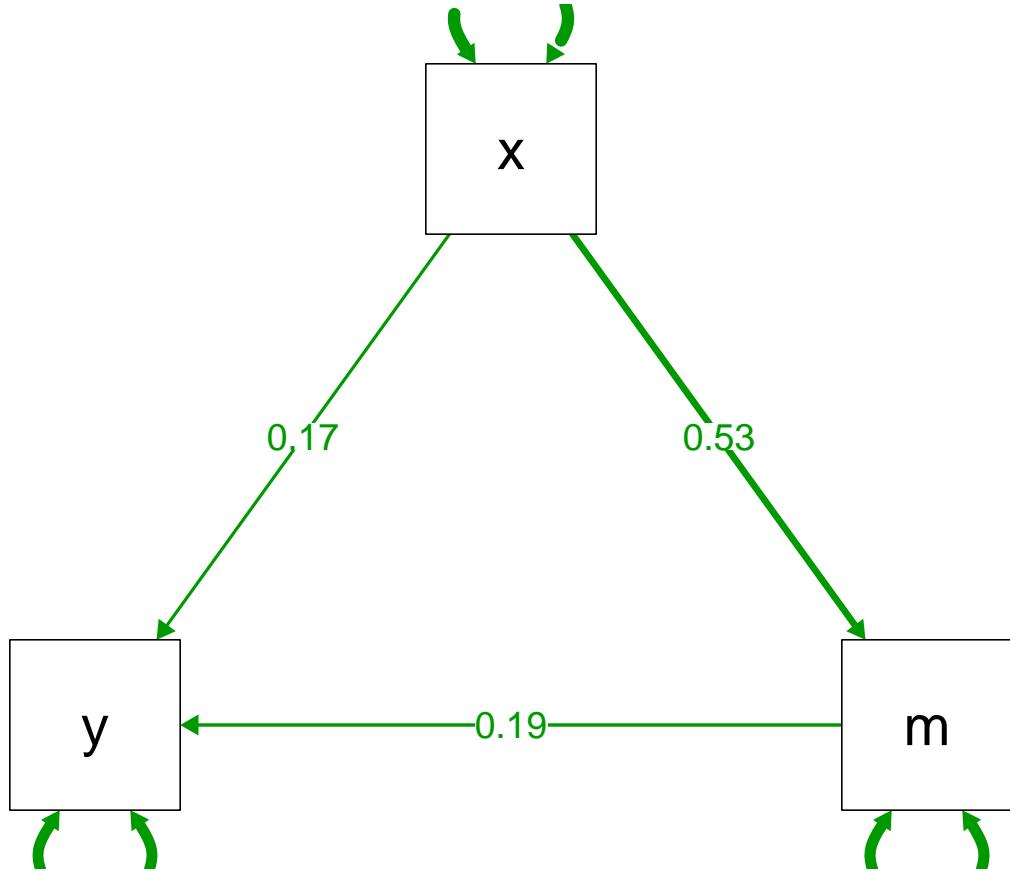
Because describing path, CFA and SEM models in words can be tedious and difficult for readers to follow it is conventional to include a diagram of (at least) your final model, and perhaps also initial or alternative models.

The `semPlot::` package makes this relatively easy: passing a fitted `lavaan` model to the `semPaths()` function produces a line drawing, and gives the option to overlap raw or standardised coefficients over this drawing:

```

# unfortunately semPaths plots very small by default, so we set
# some extra parameters to increase the size to make it readable
semPlot::semPaths(mediation.fit, "par",
                   sizeMan = 15, sizeInt = 15, sizeLat = 15,
                   edge.label.cex=1.5,
                   fade=FALSE)

```



Confirmatory factor analysis (CFA)

In psychology we make observations, but we're often interested in *hypothetical constructs*, e.g. Anxiety, working memory. We can't measure these directly, but we assume that our observations are related to these constructs in some way.

Regression and related techniques (e.g. Anova) require us to assume that our outcome variables are good indices of these underlying constructs, and that our predictor variables are measured without any error.

When outcomes are straightforward observed variables like plant yield or weight reduction, and where predictors are experimentally manipulated, then these assumptions are reasonable. However in many applied fields these are not reasonable assumptions to make: For example, to assume that depression or working memory are indexed in a straightforward way by responses to a depression questionnaire or performance on a laboratory task is naive. Likewise, we should not assume that a construct like working memory is measured without error when we use it to predict some other outcome (e.g. exam success).

Confirmatory factor analysis (CFA), structural equation models (SEM) and related techniques are designed to help researchers deal with these *imperfections in our observations*, and can help to explore the correspondence between our measures and the underlying constructs of interest.

Latent variables

CFA and SEM introduce the concept of a *latent variable* which is either the cause of, or formed by, the observations we make. Latent variables aren't quite the same thing as hypothetical constructs, but they are similar many in some ways. The original distinction between hypothetical constructs and intervening variables is quite interesting in this context, see MacCorquodale and Meehl (1948).

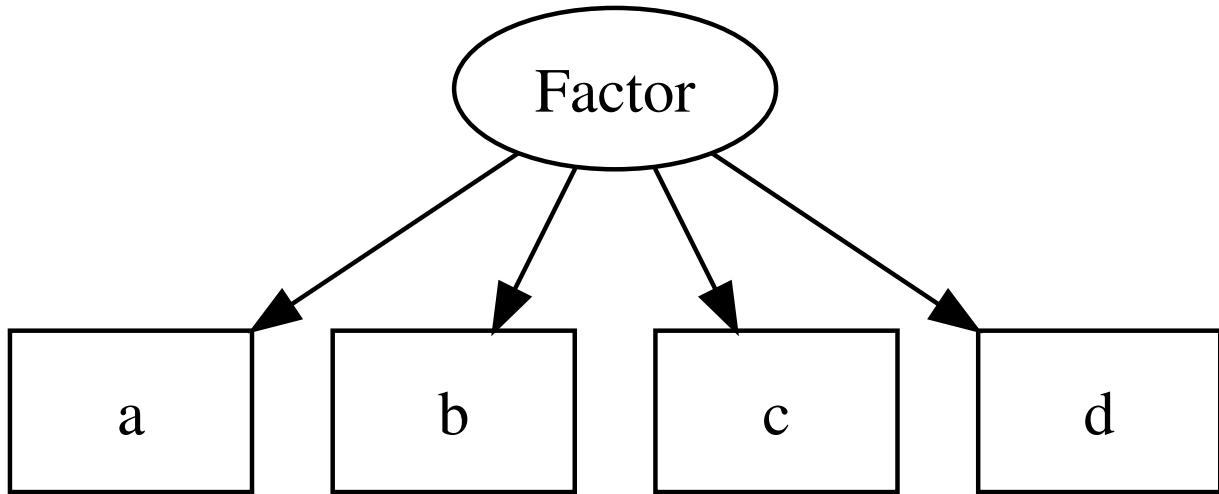


Figure 16: Example of a CFA model, including one latent variable or factor, and 4 observed variables.

To achieve this, CFA requires that researchers to make predictions about the patterns of correlations they will observe in their observations, based on the process they think is generating the data. CFA provides a mechanism to test and compare different hypotheses about these patterns, which correspond to different models of the underlying process which generates the data.

It is conventional within CFA and SEM to extend the graphical models used to describe path models (see above). In these diagrams, square edged boxes represent observed variables, and rounded or oval boxes represent latent variables, sometimes called factors:

```

knit_gv('
Factor -> a
Factor -> b
Factor -> c
Factor -> d
a[shape=rectangle]
b[shape=rectangle]
c[shape=rectangle]
d[shape=rectangle]
')
  
```

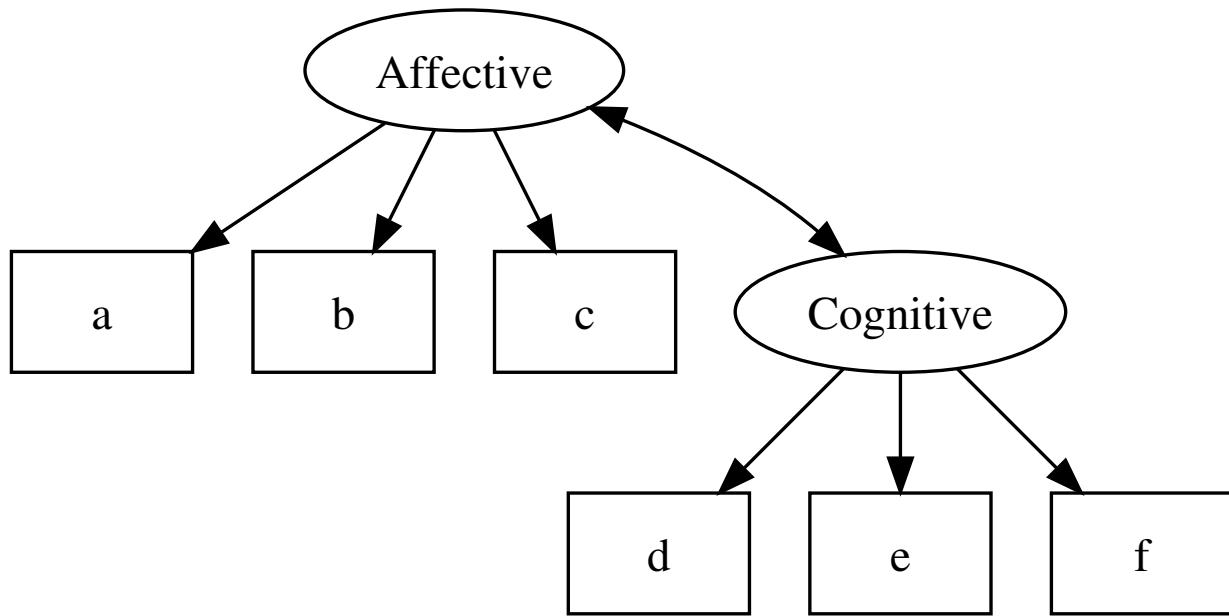
CFA models can also include multiple latent variables, and estimate the covariance between them:

```

knit_gv('
    Affective -> a
Affective -> b
Affective -> c
Cognitive -> d
Cognitive -> e
Cognitive -> f
Affective -> Cognitive:nw [dir=both]

a [shape=box]
b [shape=box]
c [shape=box]
d [shape=box]
e [shape=box]
')
  
```

```
f [shape=box]
')
)
```



SEM models extend this by allowing regression paths between latent variables and observed or other latent variables:

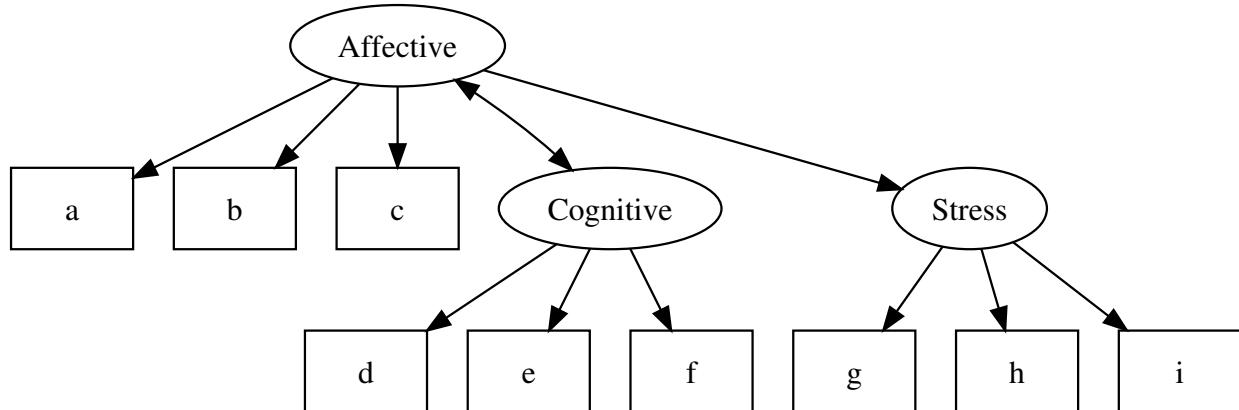
```
knit_gv(
    Affective -> a
    Affective -> b
    Affective -> c
    Cognitive -> d
    Cognitive -> e
    Cognitive -> f
    Affective -> Cognitive:nw [dir=both]

a [shape=box]
b [shape=box]
c [shape=box]
d [shape=box]
e [shape=box]
f [shape=box]

Stress -> g
Stress -> h
Stress -> i
g [shape=box]
h [shape=box]
i [shape=box]

Affective -> Stress

')
```



For now though, we will focus on building a CFA model. Later we'll show how a *well fitting* measurement model can be used to test hypotheses related to the structural relations between latent variables.

Defining a CFA model

First, open some data and check that all looks well. This is a classic CFA example — see the help file for more info.

```

hz <- lavaan::HolzingerSwineford1939
hz %>% glimpse()
Observations: 301
Variables: 15
$ id      <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, ...
$ sex     <int> 1, 2, 2, 1, 2, 2, 1, 2, 2, 2, 1, 1, 2, 2, 1, 2, 2, 1, 2...
$ ageyr   <int> 13, 13, 13, 13, 12, 14, 12, 12, 13, 12, 12, 12, 12, 12, ...
$ agemo   <int> 1, 7, 1, 2, 2, 1, 1, 2, 0, 5, 2, 11, 7, 8, 6, 1, 11, 5...
$ school  <fct> Pasteur, Pasteur, Pasteur, Pasteur, Pasteur, Pasteur, P...
$ grade   <int> 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7...
$ x1      <dbl> 3.333333, 5.333333, 4.500000, 5.333333, 4.833333, 5.333...
$ x2      <dbl> 7.75, 5.25, 5.25, 7.75, 4.75, 5.00, 6.00, 6.25, 5.75, 5...
$ x3      <dbl> 0.375, 2.125, 1.875, 3.000, 0.875, 2.250, 1.000, 1.875, ...
$ x4      <dbl> 2.333333, 1.666667, 1.000000, 2.666667, 2.666667, 1.000...
$ x5      <dbl> 5.75, 3.00, 1.75, 4.50, 4.00, 3.00, 6.00, 4.25, 5.75, 5...
$ x6      <dbl> 1.2857143, 1.2857143, 0.4285714, 2.4285714, 2.5714286, ...
$ x7      <dbl> 3.391304, 3.782609, 3.260870, 3.000000, 3.695652, 4.347...
$ x8      <dbl> 5.75, 6.25, 3.90, 5.30, 6.30, 6.65, 6.20, 5.15, 4.65, 4...
$ x9      <dbl> 6.361111, 7.916667, 4.416667, 4.861111, 5.916667, 7.500...

```

As noted above, to define models in `lavaan` you must specify the relationships between variables in a text format. A full guide to this `lavaan` model syntax is available on the project website.

For CFA models, like path models, the format is fairly simple, and resembles a series of linear models, written over several lines.

In the model below there are three latent variables, `visual`, `writing` and `maths`. The latent variable names are followed by `=~` which means ‘is manifested by’, and then the observed variables, our measures for the latent variable, are listed, separated by the `+` symbol.

```

hz.model <- '
visual =~ x1 + x2 + x3
writing =~ x4 + x5 + x6
maths =~ x7 + x8 + x9'

```

Note that we have saved our model specification/syntax in a variable named `hz.model`.

The other special symbols in the `lavaan` syntax which can be used for CFA models are:

- `a ~~ b`, which represents a *covariance*.
- `a ~~ a`, which is a *variance* (you can think of this as the covariance of a variable with itself)

To run the analysis we again pass the model specification and the data to the `cfa()` function:

```
hz.fit <- cfa(hz.model, data=hz)
summary(hz.fit, standardized=TRUE)
lavaan 0.6-3 ended normally after 35 iterations
```

Optimization method	NLMINB
Number of free parameters	21
Number of observations	301
Estimator	ML
Model Fit Test Statistic	85.306
Degrees of freedom	24
P-value (Chi-square)	0.000

Parameter Estimates:

Information	Expected
Information saturated (h1) model	Structured
Standard Errors	Standard

Latent Variables:

	Estimate	Std.Err	z-value	P(> z)	Std.lv	Std.all
visual =~						
x1	1.000				0.900	0.772
x2	0.554	0.100	5.554	0.000	0.498	0.424
x3	0.729	0.109	6.685	0.000	0.656	0.581
writing =~						
x4	1.000				0.990	0.852
x5	1.113	0.065	17.014	0.000	1.102	0.855
x6	0.926	0.055	16.703	0.000	0.917	0.838
maths =~						
x7	1.000				0.619	0.570
x8	1.180	0.165	7.152	0.000	0.731	0.723
x9	1.082	0.151	7.155	0.000	0.670	0.665

Covariances:

	Estimate	Std.Err	z-value	P(> z)	Std.lv	Std.all
visual ~~						
writing	0.408	0.074	5.552	0.000	0.459	0.459
maths	0.262	0.056	4.660	0.000	0.471	0.471
writing ~~						
maths	0.173	0.049	3.518	0.000	0.283	0.283

Variances:

	Estimate	Std.Err	z-value	P(> z)	Std.lv	Std.all
.x1	0.549	0.114	4.833	0.000	0.549	0.404

.x2	1.134	0.102	11.146	0.000	1.134	0.821
.x3	0.844	0.091	9.317	0.000	0.844	0.662
.x4	0.371	0.048	7.779	0.000	0.371	0.275
.x5	0.446	0.058	7.642	0.000	0.446	0.269
.x6	0.356	0.043	8.277	0.000	0.356	0.298
.x7	0.799	0.081	9.823	0.000	0.799	0.676
.x8	0.488	0.074	6.573	0.000	0.488	0.477
.x9	0.566	0.071	8.003	0.000	0.566	0.558
visual	0.809	0.145	5.564	0.000	1.000	1.000
writing	0.979	0.112	8.737	0.000	1.000	1.000
maths	0.384	0.086	4.451	0.000	1.000	1.000

11.1.0.4 lavaan CFA Model output

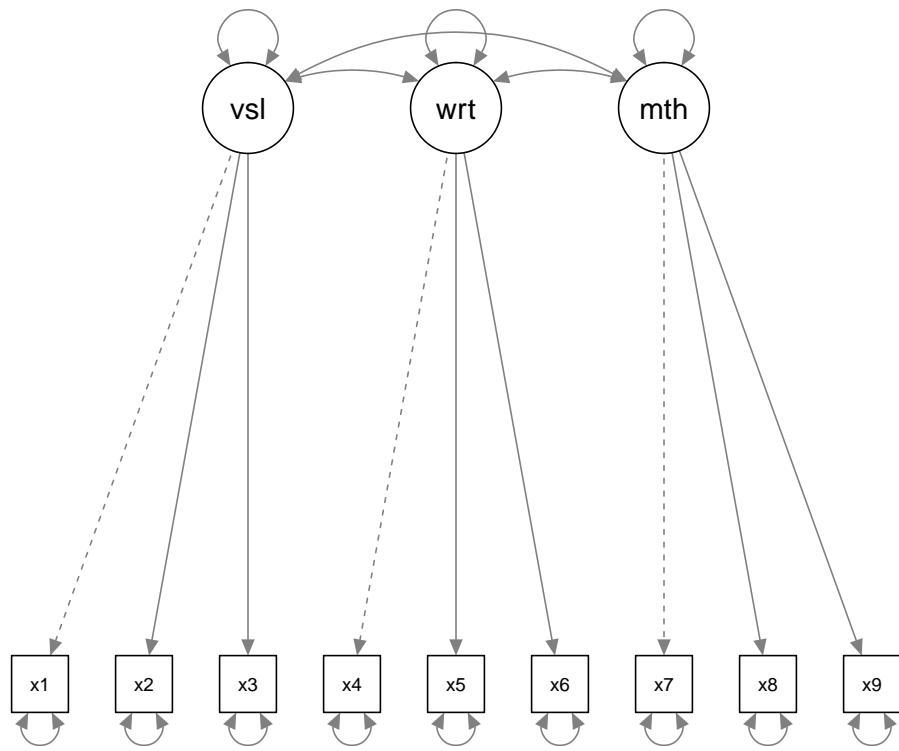
The output has three parts:

1. Parameter estimates. The values in the first column are the standardised weights from the observed variables to the latent factors.
2. Factor covariances. The values in the first column are the covariances between the latent factors.
3. Error variances. The values in the first column are the estimates of each observed variable's error variance.

11.1.0.5 Plotting models

As before, we can use the `semPaths()` function to visualise the model. This is an important step because it helps explain the model to others, and also gives you an opportunity to check you have specified your model correctly.

```
semPlot::semPaths(hz.fit)
```

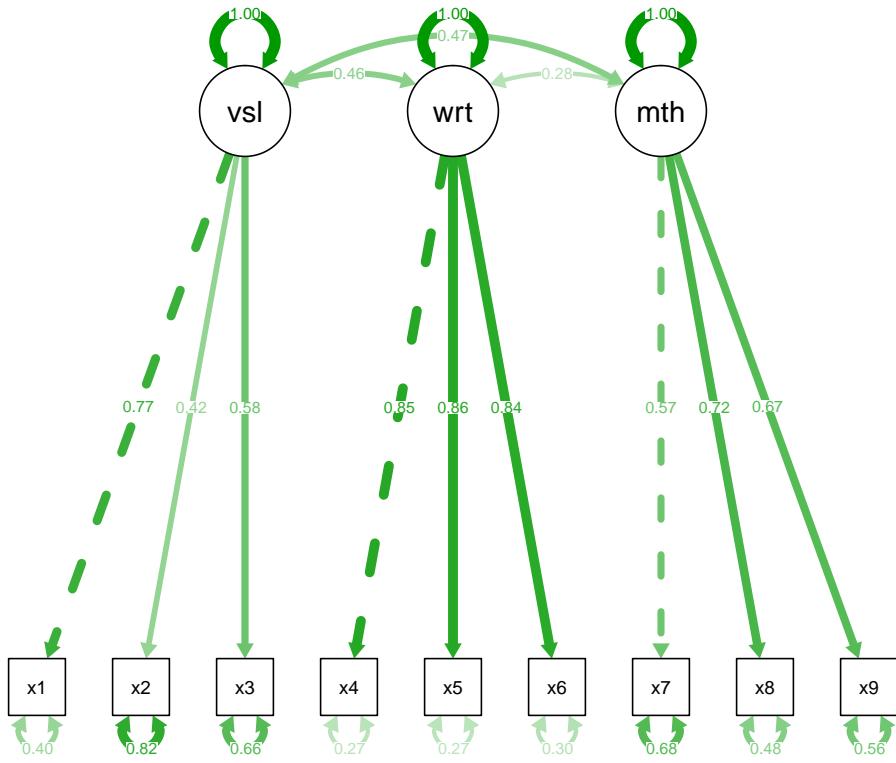


And for ‘final’ models we might want to overplot model parameter estimates (in this case, standardised):

```

# std refers to standardised estimates. "par" would plot
# the unstandardised estimates
semPlot::semPaths(hz.fit, "std")

```



CFA model fit

To examine the model fit we use `fitmeasures()` and pass a list of the names of the fit indices we would like calculated:

```
library(lavaan)
fitmeasures(hz.fit, c('cfi', 'rmsea', 'rmsea.ci.upper', 'bic'))
      cfi        rmsea rmsea.ci.upper      bic
    0.931     0.092       0.114    7595.339
```

This looks OK, but the fit indices indicate the model could be improved. In particular the RMSEA figure is above 0.05. See the notes on goodness of fit statistics for more detail.

Modification indices

Modification indices help us answer ‘what if?’ questions about whether freeing parameter constraints or adding paths to our models would help improve it. The modification index is the χ^2 value, with 1 degree of freedom, by which model fit would improve if a particular path was added or constraint freed. Values bigger than 3.84 indicate that the model would be ‘improved’, and the p value for the added parameter would be $< .05$, and values larger 10.83 than indicate the parameter would have a p value $< .001$. *This does not mean that all parameters which appear in the MI table should be added*, but it can be an aid to improving the model, in combination with domain or theoretical knowledge. The rule of thumb is to add parameters only when they ‘make sense’ substantively. See the notes on model improvements for more guidance.

To examine the modification indices we type:

```
modificationindices(hz.fit)
```

But because this function produces a very long table of output, it can be helpful to sort and filter the rows to show only those model modifications which might be of interest to us.

The command below converts the output of `modificationindices()` to a dataframe. It then:

- Sorts the rows by the `mi` column, which represents the change in model χ^2 we see if the path was included (see sorting)
- Filters the results to show only those with χ^2 change > 10
- Selects only the `lhs`, `op`, `rhs`, `mi`, and `epc` columns.

```
modificationindices(hz.fit) %>%
  as_data_frame() %>%
  arrange(-mi) %>%
  filter(mi > 11) %>%
  select(lhs, op, rhs, mi, epc) %>%
  pander(caption="Largest MI values for hz.fit")
```

Table 46: Largest MI values for hz.fit

lhs	op	rhs	mi	epc
visual	=~	x9	36.41	0.577
x7	~~	x8	34.15	0.5364
visual	=~	x7	18.63	-0.4219
x8	~~	x9	14.95	-0.4231

The `lhs` (left hand side, or outcome), `rhs` (right hand side, or predictor) and `op` (operation) columns specify what modification should be made.

Paths linking latent variables to the observed variables which index them have `=~` in the ‘op’ column.

Error covariances for observed variables have `~~` as the op. These symbols match the symbols used to describe a path in the lavaan model syntax.

If we add the largest MI path to our model it will look like this:

```
# same model, but with x9 now loading on visual
hz.model.2 <- "
visual =~ x1 + x2 + x3 + x9
writing =~ x4 + x5 + x6
maths =~ x7 + x8 + x9"

hz.fit.2 <- cfa(hz.model.2, data=hz)
fitmeasures(hz.fit.2, c('cfi', 'rmsea', 'rmsea.ci.upper', 'bic'))
      cfi        rmsea rmsea.ci.upper          bic
      0.967      0.065      0.089      7568.123
```

RMSEA has improved somewhat, but we’d probably want to investigate this model further, and make additional improvements to it (although see the notes on model improvements)

Model modification and improvement

Modification indices are a way of improving your model by identifying parameters which, if included, would improve model fit (or constraints removed). However, remember that:

- Use of modification indices should be informed by theory
- MI may suggest paths which don't make substantive sense

It's very important to avoid adding paths in a completely data-driven way because this is almost certain to lead to over-fitting.

It's also important to work one step at a time, because the table of modification indices may change as you add additional paths. For example, the second largest MI value may change once you add the path with the largest MI to the model.

The basic steps to follow are:

1. Run a simple, theoretically-derived model
2. Notice it fits badly
3. Add any additional paths which make theoretical sense
4. Check GOF; If it still fits badly then,
5. Run MI and identify the largest value
6. If this parameter makes theoretical sense, relax the constraint
7. Re-run the model and return to step 4

Structural equation modelling (SEM)

Combining Path models and CFA to create structural equation models (SEM) allows researchers to combine allow for measurement imperfection whilst also (attempting to) infer information about causation.

SEM involves adding paths to CFA models which are, like predictors in standard regression models, are assumed to be causal in nature; i.e. rather than variables x and y simply covarying with one another, we are prepared to make the assumption that x causes y .

It's worth pointing out though, right from the offset, that *causal relationships drawn from SEM models always dependent on assumptions we are prepared to make when setting up our model*. There is nothing magical in the technique that makes allows us to infer causality from non-experimental data (although note SEM can be used for some experimental analyses).

It is only be our substantive knowledge of the domain that makes any kind of causal inference reasonable, and when using SEM the onus is always *on us* to check our assumptions, provide sensitivity analyses which test alternative causal models, and interpret observational data cautiously.

[Note, there are techniques which use SEM as a means to make stronger kinds of causal statements, for example instrumental variable analysis, but even here, inferring causality still requires that we make strong assumptions about the process which generated our data.]

Nonetheless, with these caveats in mind, SEM can be a useful technique to quantify relationships between observed variables where we have measurement error, and especially where we have a theoretical model linking these observations.

11.1.0.6 Steps to running an SEM

1. Identify and test the fit of a *measurement model*. This is a CFA model which includes all of your observed variables, arranged in relation to the latent variables you think generated the data, and where covariances between all these latent variables are included. This step may include many rounds of model fitting and modification.
2. Ensure your measurement model fits the data adequately before continuing. Test alternative or simplified measurement models and report where these perform well (e.g. are close in fit to your desired model). SEM models that are based on a poorly fitting measurement model will produce parameter estimates that are imprecise, unstable or both, and you should not proceed unless an adequately fitting measurement model is found (see this nice discussion, which includes relevant references).

3. Convert your measurement model by removing covariances between latent variables where necessary and including new structural paths. Test model fit, and interpret the paths of interest. Avoid making changes to the measurement part of the model at this stage. Where the model is complex consider adjusting p values to allow for multiple comparisons (if using NHST).
4. Test alternative models (e.g. with paths removed or reversed). Report where alternatives also fit the data.
5. In writing up, provide sufficient detail for other researchers to replicate your analyses, and to follow the logic of the amendments you make. Ideally share your raw data, but at a minimum share the covariance matrix. Report GOF statistics, and follow published reporting guidelines for SEM [Schreiber et al.]. Always include a diagram of your final model (at the very least).

11.1.0.7 A worked example: Building from a measurement model to SEM

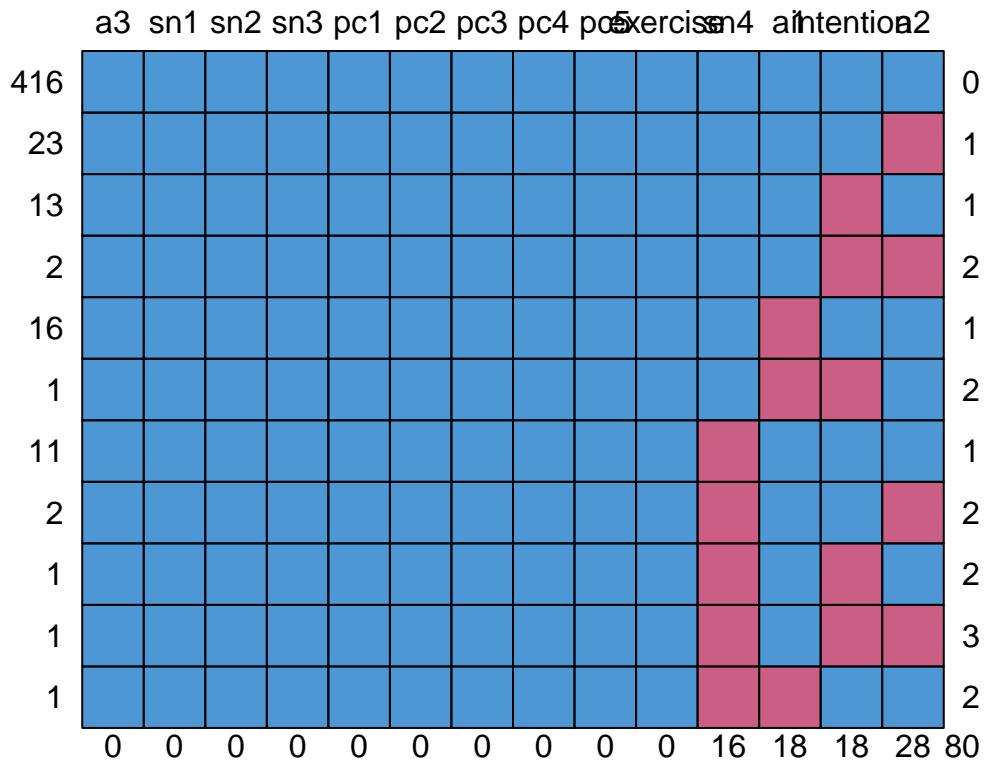
Imagine we have some data from a study that aimed to test the theory of planned behaviour. Researcher measured exercise and intentions, along with multiple measures of attitudes, social norms and perceived behavioural control.

```
tpb.df %>% psych::describe(fast=T)
```

vars	n	mean	sd	min	max	range	se
1	469	-0.00539	1.61	-5.99	4.78	10.8	0.0745
2	459	0.117	1.22	-3.17	4.18	7.35	0.057
3	487	-0.001	1.08	-3.22	3.13	6.35	0.0487
4	487	0.078	1.53	-5.04	5.43	10.5	0.0695
5	487	0.0376	1.22	-3.76	3.61	7.37	0.0551
6	487	0.00863	1.11	-3.21	3.11	6.33	0.0501
7	471	0.0261	1.1	-2.94	3.14	6.08	0.0506
8	487	-0.00329	1.35	-4.13	4.03	8.16	0.061
9	487	-0.0865	1.31	-4.86	4.08	8.94	0.0593
10	487	-0.00563	1.19	-3.12	3.34	6.46	0.0537
11	487	-0.0863	1.06	-3.23	3.34	6.57	0.0481
12	487	-0.0663	1.16	-4.11	3.5	7.61	0.0525
13	469	10.1	2.55	1.83	16.7	14.9	0.118
14	487	80.2	18.8	15	138	123	0.851

There were some missing data, but nothing to suggest a systematic pattern. For the moment we continue with standard methods:

```
mice::md.pattern(tpb.df)
```



	a3	sn1	sn2	sn3	pc1	pc2	pc3	pc4	pc5	exercise	sn4	a1	intention	a2
416	1	1	1	1	1	1	1	1	1	1	1	1	1	0
23	1	1	1	1	1	1	1	1	1	1	1	1	0	1
13	1	1	1	1	1	1	1	1	1	1	1	1	0	1
2	1	1	1	1	1	1	1	1	1	1	1	1	0	2
16	1	1	1	1	1	1	1	1	1	1	1	0	1	1
1	1	1	1	1	1	1	1	1	1	1	1	0	0	2
11	1	1	1	1	1	1	1	1	1	1	0	1	1	1
2	1	1	1	1	1	1	1	1	1	1	0	1	0	2
1	1	1	1	1	1	1	1	1	1	1	0	1	0	2
1	1	1	1	1	1	1	1	1	1	1	0	0	0	3
1	1	1	1	1	1	1	1	1	1	1	0	0	1	2
0	0	0	0	0	0	0	0	0	0	0	16	18	18	80

We start by fitting a measurement model. The model syntax includes lines with

- `=~` separating left and right hand side (to define the latent variables)
- `~~` to specify latent covariances

We are not including `exercise` and `intention` yet because these are observed variables only (we don't have multiple measurements for them) and so they don't need to be in the measurement model:

```
mes.mod <- '
  # the "measurement" part, defining the latent variables
  AT =~ a1 + a2 + a3 + sn1
  SN =~ sn1 + sn2 + sn3 + sn4'
```

```

PBC =~ pc1 + pc2 + pc3 + pc4 + pc5

# note that lavaan automatically includes latent covariances
# but we can add here anyway to be explicit
AT ~~ SN
SN ~~ PBC
AT ~~ PBC

```

We can fit this model to the data like so:

```

mes.mod.fit <- cfa(mes.mod, data=tpb.df)
summary(mes.mod.fit)
lavaan 0.6-3 ended normally after 52 iterations

```

Optimization method	NLMINB
Number of free parameters	28
	Used Total
Number of observations	429 487
Estimator	ML
Model Fit Test Statistic	50.290
Degrees of freedom	50
P-value (Chi-square)	0.462

Parameter Estimates:

Information	Expected
Information saturated (h1) model	Structured
Standard Errors	Standard

Latent Variables:

	Estimate	Std.Err	z-value	P(> z)
AT =~				
a1	1.000			
a2	0.471	0.056	8.466	0.000
a3	0.239	0.042	5.654	0.000
sn1	-0.145	0.223	-0.651	0.515
SN =~				
sn1	1.000			
sn2	0.529	0.153	3.456	0.001
sn3	0.294	0.089	3.300	0.001
sn4	0.335	0.099	3.383	0.001
PBC =~				
pc1	1.000			
pc2	0.860	0.098	8.813	0.000
pc3	0.647	0.081	7.978	0.000
pc4	0.425	0.069	6.133	0.000
pc5	0.643	0.081	7.964	0.000

Covariances:

	Estimate	Std.Err	z-value	P(> z)
AT ~~				

SN	1.488	0.460	3.233	0.001
SN ~~				
PBC	0.045	0.085	0.525	0.600
AT ~~				
PBC	0.010	0.086	0.116	0.908
Variances:				
	Estimate	Std.Err	z-value	P(> z)
.a1	0.586	0.195	2.997	0.003
.a2	1.068	0.086	12.494	0.000
.a3	1.018	0.072	14.222	0.000
.sn1	0.870	0.239	3.640	0.000
.sn2	0.940	0.090	10.404	0.000
.sn3	1.071	0.077	13.887	0.000
.sn4	1.028	0.076	13.544	0.000
.pc1	0.871	0.104	8.401	0.000
.pc2	1.084	0.100	10.842	0.000
.pc3	1.018	0.082	12.431	0.000
.pc4	0.992	0.072	13.690	0.000
.pc5	1.014	0.081	12.447	0.000
AT	2.035	0.259	7.859	0.000
SN	1.873	0.882	2.124	0.034
PBC	0.888	0.135	6.600	0.000

And we can assess model fit using `fitmeasures`. Here we select a subset of the possible fit indices to keep the output manageable.

```
useful.fit.measures <- c('chisq', 'rmsea', 'cfi', 'aic')
fitmeasures(mes.mod.fit, useful.fit.measures)
  chisq      rmsea       cfi       aic
  50.290    0.004     1.000  16045.276
```

This model looks pretty good (see the guide to fit indices), but still check modification indices to identify improvements. If they made theoretical sense we might choose to add paths:

```
modificationindices(mes.mod.fit) %>%
  as_data_frame() %>%
  filter(mi>4) %>%
  arrange(-mi) %>%
  pander(caption="Modification indices for the measurement model")
```

Table 47: Modification indices for the measurement model

lhs	op	rhs	mi	epc	sepc.lv	sepc.all	sepc.nox
AT	==	sn2	20.39	0.5284	0.7536	0.6226	0.6226
sn1	~~	sn2	15.04	-0.4689	-0.4689	-0.5184	-0.5184
AT	==	sn4	12.49	-0.3035	-0.4329	-0.3891	-0.3891
a1	~~	sn2	8.476	0.2457	0.2457	0.331	0.331
sn1	~~	sn4	6.54	0.216	0.216	0.2285	0.2285
a2	~~	pc5	4.098	-0.1118	-0.1118	-0.1074	-0.1074

However, in this case unless we had substantive reasons to add the paths, it would probably be reasonable to continue with the original model.

11.1.0.7.1 The measurement model fits, so proceed to SEM

Our SEM model adapts the CFA (measurement model), including additional observed variables (e.g. intention and exercise) and any relevant structural paths:

```
sem.mod <- '
  # this section identical to measurement model
  AT =~ a1 + a2 + a3 + sn1
  SN =~ sn1 + sn2 + sn3 + sn4
  PBC =~ pc1 + pc2 + pc3 + pc4 + pc5

  # additional structural paths
  intention ~ AT + SN + PBC
  exercise ~ intention
'
```

We can fit it as before, but now using the `sem()` function rather than the `cfa()` function:

```
sem.mod.fit <- sem(sem.mod, data=tpb.df)
```

The first thing we do is check the model fit:

```
fitmeasures(sem.mod.fit, useful.fit.measures)
  chisq      rmsea       cfi       aic
  171.065    0.058     0.929  20638.124
```

RMSEA is slightly higher than we like, so we can check the modification indices:

```
sem.mi <- modificationindices(sem.mod.fit) %>%
  as_data_frame() %>%
  arrange(-mi)

sem.mi %>%
  head(6) %>%
  pander(caption="Top 6 modification indices for the SEM model")
```

Table 48: Top 6 modification indices for the SEM model

lhs	op	rhs	mi	epc	sepc.lv	sepc.all	sepc.nox
exercise	~	PBC	90.14	7.601	7.049	0.3695	0.3695
PBC	~	exercise	89.49	0.04938	0.05325	1.016	1.016
intention	~	exercise	47.03	-0.1233	-0.1233	-0.9106	-0.9106
intention	~~	exercise	47.03	-16.18	-16.18	-0.6779	-0.6779
AT	=~	sn2	16.14	0.5093	0.7043	0.5855	0.5855
pc1	~~	exercise	15.08	2.405	2.405	0.2205	0.2205

Interestingly, this model suggests two additional paths involving `exercise` and the PBC latent:

```
sem.mi %>%
  filter(lhs %in% c('exercise', 'PBC') & rhs %in% c('exercise', 'PBC')) %>%
  pander()
```

lhs	op	rhs	mi	epc	sepc.lv	sepc.all	sepc.nox
exercise	~	PBC	90.14	7.601	7.049	0.3695	0.3695
PBC	~	exercise	89.49	0.04938	0.05325	1.016	1.016

Of these suggested paths, the largest MI is for the one which says PBC is predicted by exercise. However, the model would also be improved by allowing PBC to predict exercise. Which should we add?

The answer will depend on both previous theory and knowledge of the data.

If it were the case that exercise was measured at a later time point than PBC. In this case the decision is reasonably clear, because the temporal sequencing of observations would determine the most likely path. These data were collected contemporaneously, however, and so we can't use our *design* to differentiate the causal possibilities.

Another consideration would be that, by adding a path from exercise to PBC we would make the model non-recursive, and likely non-identified.

A theorist might also argue that because previous studies, and the theory of planned behaviour itself, predict that PBC may exert a direct influence on behaviour, we should add the path with the smaller MI (so allow PBC to predict exercise).

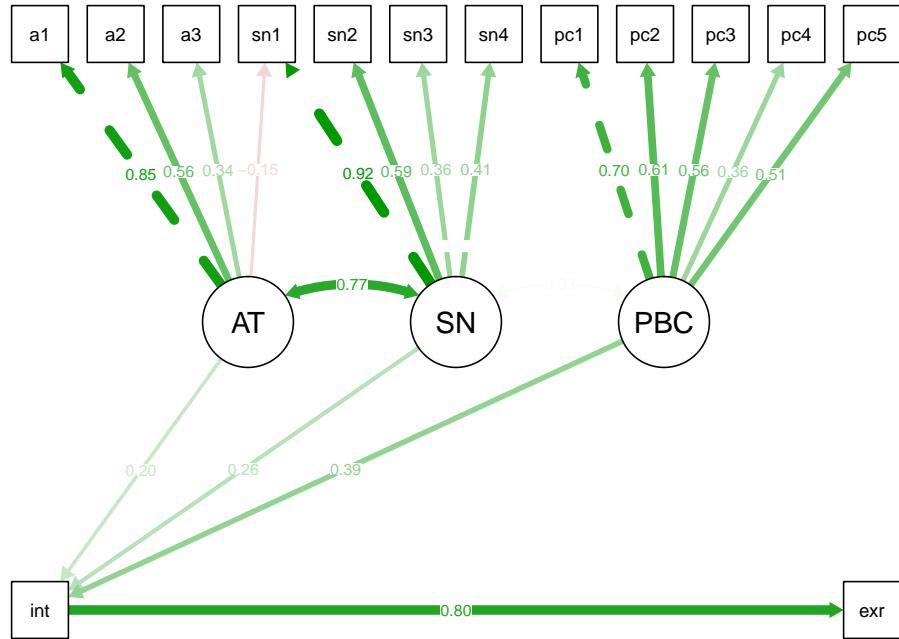
In this case, the best course of action would probably be to report the theoretically implied model, but also test alternative models in which causal relations between the variables are reversed or otherwise altered (along with measures of fit and key parameter estimates). The discussion of your paper would then make the case for your preferred account, but make clear that the data were (most likely) unable to provide a persuasive case either way, and that alternative explanations cannot be ruled out.

11.1.0.7.2 Interpreting and presenting key parameters

One of the best ways to present estimates from your final model is in a diagram, because this is intuitive and provides a simple way for readers to comprehend the paths implied by your model.

We can automatically generate a plot from a fitted model using `semPaths()`. Here, the `what='std'` is requesting standardised parameter estimates be shown. Adding `residuals=F` hides variances of observed and latent variables, which are not of interest here. The line thicknesses are scaled to represent the size parameter itself:

```
semPlot::semPaths(sem.mod.fit, what='std', residuals=F)
```



For more information on reporting SEM however, see Schreiber et al. [2006].

'Identification' in CFA and SEM

Identification refers to the idea that a model is ‘estimable’, or more specifically whether there is a single best solution for the parameters specified in the model. An analogy would be the ‘line of best fit’ in regression - if we could draw two lines that fit the data equally well then our method doesn’t enable us to choose between these possibilities, and is essentially meaningless (or uninterpretable, anyway).

This is a complex topic, but David Kenny has an excellent page here which covers identification in lots of detail: <http://davidakenny.net/cm/identify.htm>. Some of the key ideas to takeaway are:

11.1.0.8

- Feedback loops and other non-recursive models are likely to cause problems without special attention.
- Latent variables need a scale. To do this either fix their variance, or fix a factor loading to 1.
- You need ‘enough data’. Normally this will be at least 3 measured variables per latent. Sometimes 2 is enough, provided the errors of these variables are uncorrelated, but you may struggle to fit models because of ‘empirical under-identification’⁹
- If a model is non-identified, it may either i) fail to run or, worse, ii) produce spurious results.

⁹Note, indicators themselves should be correlated with one another in a bivariate correlation matrix. It’s only the errors which should be uncorrelated.

11.1.0.8.1 Rule B

For structural models, ‘Rule B’ also applies when deciding when a model is identified: No more than one of the following statements should be true about variables or latents in your model:

- X directly causes Y
- Y directly causes X
- X and Y have a correlated disturbance
- X and Y are correlated exogenous variables

But see http://davidakenny.net/cm/identify_formal.htm#RuleB for a proper explanation.

Missing data

If you have missing data you can use the `missing = "ML"` argument to ask lavaan to estimate the ‘full information maximum likelihood’ (see <http://lavaan.ugent.be/tutorial/est.html>).

```
# fit ML model including mean structure to make comparable with FIML fit below
# (means are always included with FIML model fits)
sem.mod.fit <- sem(sem.mod, data=tpb.df, meanstructure=TRUE)

# fit again including missing data also
sem.mod.fit.fiml <- sem(sem.mod, data=tpb.df, missing="ML")
```

It doesn’t look like the parameter estimates change much. To compare them explicitly we can extract the relevant coefficients from each (they don’t look all that different):

```
bind_cols(parameterestimates(sem.mod.fit) %>%
  select(lhs, op, rhs, est, pvalue) %>%
  rename(ml=est, ml.p = pvalue),
parameterestimates(sem.mod.fit.fiml) %>%
  transmute(fiml=est, fiml.p = pvalue)) %>%
# select only the regression paths
filter(op=="~") %>%
as_huxtable() %>%
set_caption("Comparison of ML and MLM parameter estimates.") %>%
print_md()

-----
```

intention	~ AT	0.379	0.0513	0.306	0.0608
intention	~ SN	0.472	0.0234	0.479	0.00381
intention	~ PBC	1.09	3.51e-12	1.05	2.01e-13
exercise	~ intention	5.91	0	5.9	0

Table: Comparison of ML and MLM parameter estimates.

Goodness of fit statistics in CFA

It’s worth noting that many ‘goodness of fit’ statistics are misnamed and are in fact indexing ‘badness of fit’. This applies to RMSEA, χ^2 , BIC, AIC and others.

However all of these indices are trying to solve similar problems in subtly different ways. The problem is that we would like a model which:

- Fits the data we have *and also*
- Predicts new data

You might think that these goals would be aligned and that a model which fits the data we have would also be good at predicting new data, but this isn't the case. In fact, if we overfit our current data we won't be able to predict new observations very accurately.

11.1.0.9 How fit indices work

There is a tradeoff involved to avoid over-fitting the data, and most fit indices attempt to:

- Quantify how well the model fits the current data but
- Penalise models which use many parameters (i.e. those in danger of overfitting)

Each formula for a goodness of fit statistic represents a different tradeoff between these goals.

11.1.0.10

Model fit statistics are useful but can be misleading and misused. See David Kenny's page on model fit for more details: <http://davidakenny.net/cm/fit.htm>

11.1.0.11

Below are some of the most useful and commonly reported GOF statistics for CFA and SEM models:

11.1.0.12 Root Mean Square Error of Approximation (RMSEA)

MacCallum, Browne and Sugawara (1996) have used 0.01, 0.05, and 0.08 to indicate excellent, good, and mediocre fit, respectively.

$\text{RMSEA} < .05$ often used as a cutoff for a reasonably fitting model, although others suggest .1.

RMSEA is also used to calculate the 'probability of a close fit' or `pclose` statistic — this is the probability that the RMSEA is under 0.05.

11.1.0.13 Comparative fit index (CFI)

CFI (and the related TLI) assesses the relative improvement in fit of your model compared with the baseline model.

CFI ranges between 0 and 1.

The conventional (rule of thumb) threshold for a good fitting model is for CFI to be $> .9$

11.1.0.14 Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC)

The AIC and BIC are measures of comparative fit, so can be used when models are non-nested (and therefore otherwise not easily comparable).

AIC is particularly attractive because it corresponds to a measure of *predictive* accuracy. That is, selecting the model with the smallest AIC is one way of asking: "which model is most likely to accurately predict new data?"

11.1.0.15 Factors which can influence fit statistics

All of the following can influence or bias fit statistics:

- Number of variables (although note RMSEA tends to reduce with more parameters included, but other fit statistics will increase).
- Model complexity (different statistics reward parsimony to different degrees).
- Sample size (varies by statistic: some increase and others decrease with sample size).
- Non-normality of outcome data will (tend to) worsen fit.

11.1.0.16 Which statistics should you report?

When reporting absolute model fit, RMSEA and CFI are the most widely reported, and are probably sufficient.

However, you should almost never just report a single model, and so:

- When comparing nested models you should report the χ^2 `lrtest`.
- When comparing non-nested models you should also report differences in BIC and AIC.

11.1.0.17 Further reading

This set of slides on model fit provides all fo the formulae and an explanation for many different fit indices:
http://www.psych.umass.edu/uploads/people/79/Fit_Indices.pdf

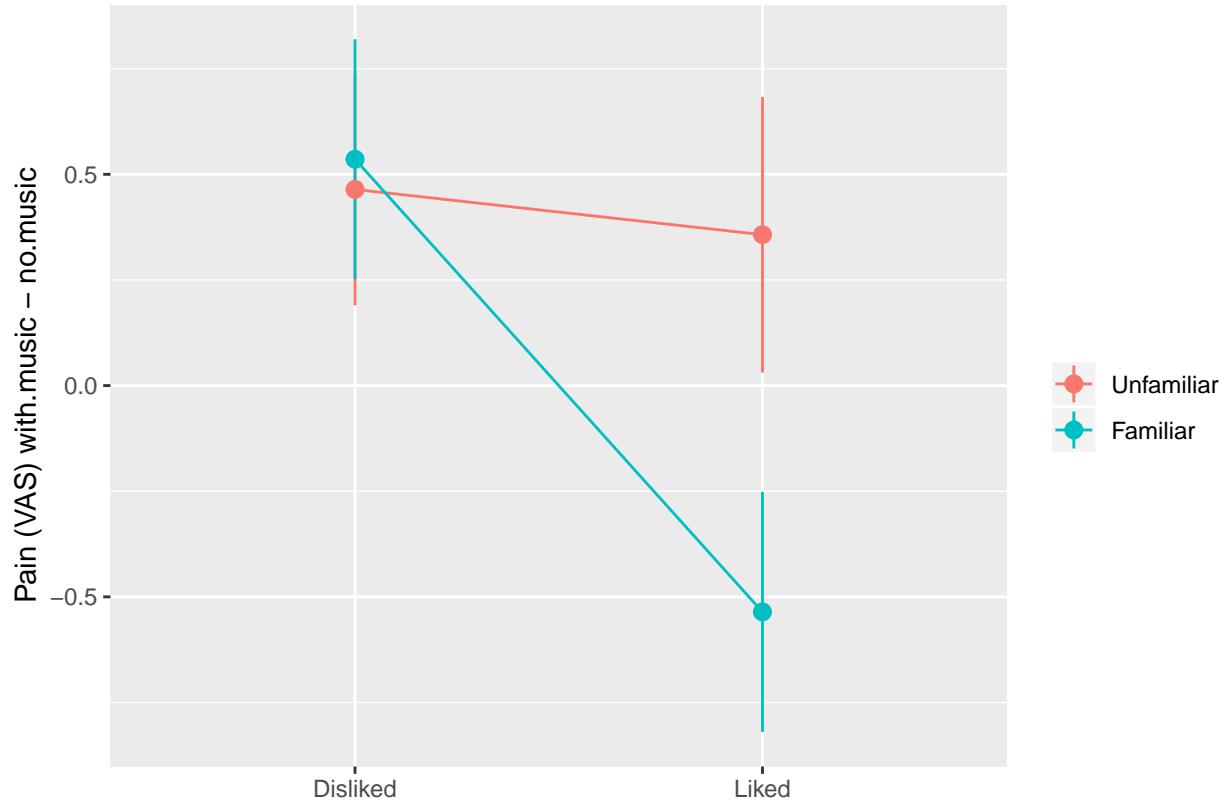
12 Bayesian model fitting

Baysian fitting of linear models via MCMC methods

This is a minimal guide to fitting and interpreting regression and multilevel models via MCMC. For *much* more detail, and a much more comprehensive introduction to modern Bayesian analysis see Jon Kruschke's *Doing Bayesian Data Analysis*.

Let's revisit our previous example which investigated the effect of familiar and liked music on pain perception:

```
painmusic <- readRDS('data/painmusic.RDS')
painmusic %>%
  ggplot(aes(liked, with.music - no.music,
             group=familiar, color=familiar)) +
  stat_summary(geom="pointrange", fun.data=mean_se) +
  stat_summary(geom="line", fun.data=mean_se) +
  ylab("Pain (VAS) with.music - no.music") +
  scale_color_discrete(name="") +
  xlab("")
```



```
# set sum contrasts
options(contrasts = c("contr.sum", "contr.poly"))
pain.model <- lm(with.music ~
                  no.music + familiar * liked,
                  data=painmusic)
summary(pain.model)

Call:
lm(formula = with.music ~ no.music + familiar * liked, data = painmusic)

Residuals:
    Min      1Q  Median      3Q     Max 
-3.5397 -1.0123 -0.0048  0.9673  4.8882 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 1.55899   0.40126  3.885 0.000177 ***
no.music    0.73588   0.07345 10.019 < 2e-16 ***
familiar1   0.20536   0.13895  1.478 0.142354    
liked1       0.30879   0.13900  2.222 0.028423 *  
familiar1:liked1 -0.18447   0.13983 -1.319 0.189909    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.47 on 107 degrees of freedom
Multiple R-squared:  0.5043,    Adjusted R-squared:  0.4858
```

```
F-statistic: 27.22 on 4 and 107 DF, p-value: 1.378e-15
```

Do the same thing again, but with MCMC using Stan:

```
library(rstanarm)
options(contrasts = c("contr.sum", "contr.poly"))
pain.model.mcmc <- stan_lm(with.music ~ no.music + familiar * liked,
                            data=painmusic, prior=NULL)

summary(pain.model.mcmc)
```

Model Info:

```
function: stan_lm
family: gaussian [identity]
formula: with.music ~ no.music + familiar * liked
algorithm: sampling
priors: see help('prior_summary')
sample: 4000 (posterior sample size)
observations: 112
predictors: 5
```

Estimates:

	mean	sd	2.5%	25%	50%	75%	97.5%
(Intercept)	1.7	0.4	0.9	1.4	1.7	1.9	2.5
no.music	0.7	0.1	0.6	0.7	0.7	0.8	0.8
familiar1	0.2	0.1	-0.1	0.1	0.2	0.3	0.5
liked1	0.3	0.1	0.0	0.2	0.3	0.4	0.6
familiar1:liked1	-0.2	0.1	-0.4	-0.3	-0.2	-0.1	0.1
sigma	1.5	0.1	1.3	1.4	1.5	1.5	1.7
log-fit_ratio	0.0	0.1	-0.1	0.0	0.0	0.0	0.1
R2	0.5	0.1	0.4	0.4	0.5	0.5	0.6
mean_PPD	5.3	0.2	4.9	5.2	5.3	5.5	5.7
log-posterior	-205.9	2.2	-211.0	-207.2	-205.6	-204.4	-202.6

Diagnostics:

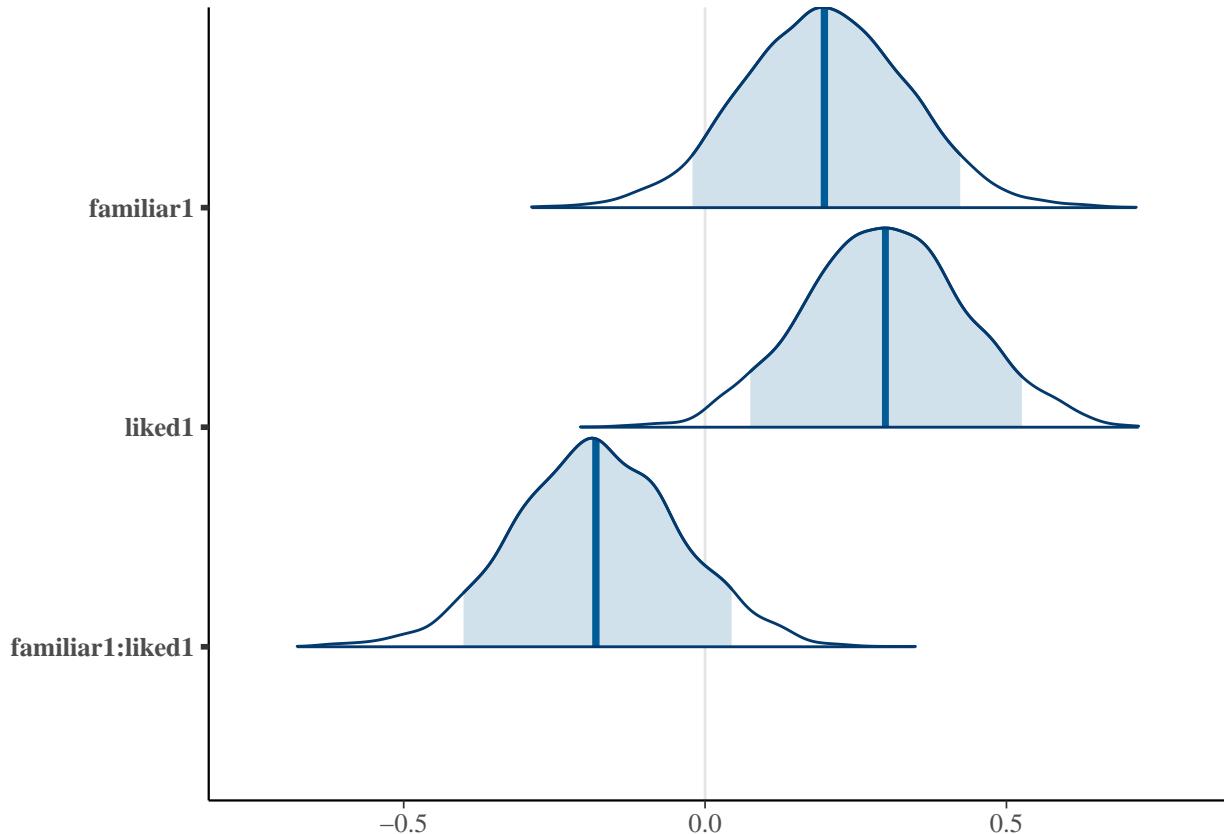
	mcse	Rhat	n_eff
(Intercept)	0.0	1.0	1278
no.music	0.0	1.0	1281
familiar1	0.0	1.0	3969
liked1	0.0	1.0	3616
familiar1:liked1	0.0	1.0	2862
sigma	0.0	1.0	3343
log-fit_ratio	0.0	1.0	2015
R2	0.0	1.0	1396
mean_PPD	0.0	1.0	4037
log-posterior	0.1	1.0	996

For each parameter, mcse is Monte Carlo standard error, n_eff is a crude measure of effective sample size.

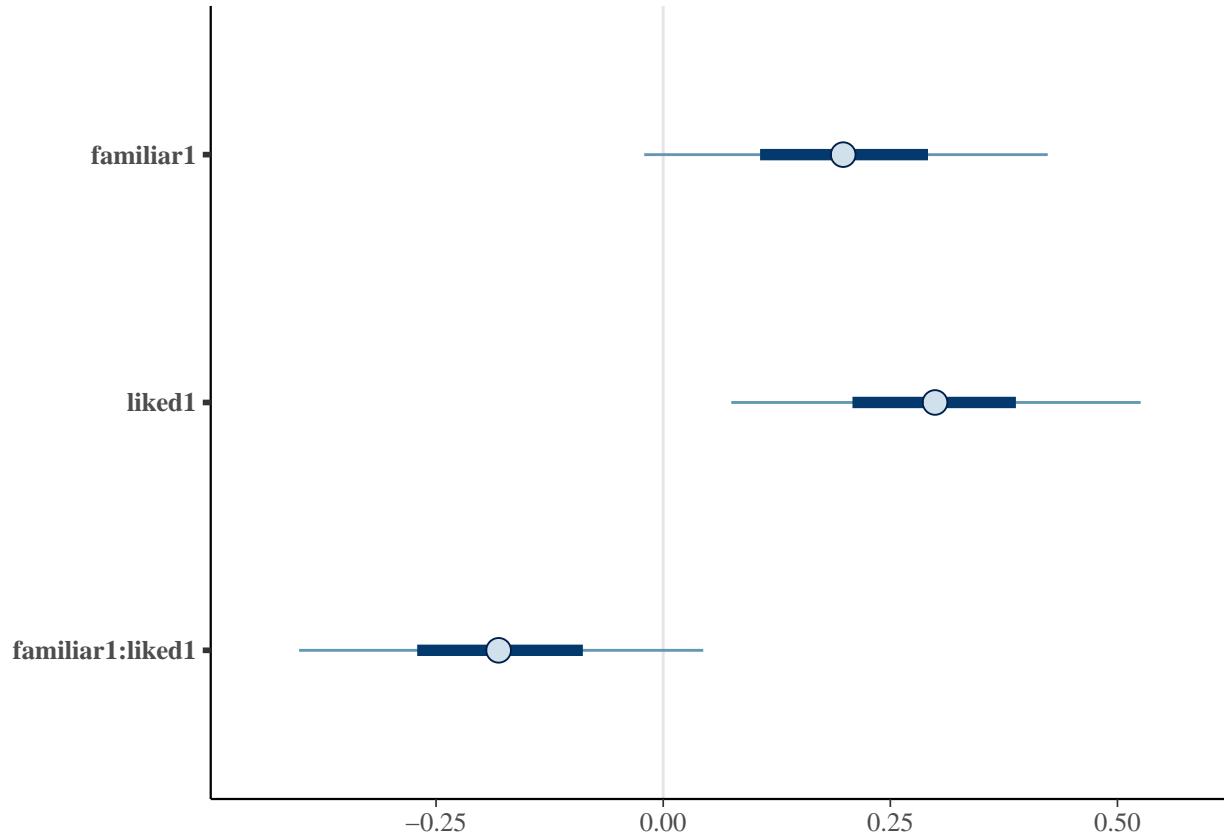
Posterior probabilities for parameters

```
library(bayesplot)

mcmc_areas(as.matrix(pain.model.mcmc), regex_pars = 'familiar|liked', prob = .9)
```



```
mcmc_intervals(as.matrix(pain.model.mcmc), regex_pars = 'familiar|liked', prob_outer = .9)
```



Credible intervals

Credible intervals are distinct from confidence intervals

TODO EXPAND

```
params.of.interest <-
  pain.model.mcmc %>%
  as_tibble %>%
  reshape2::melt() %>%
  filter(stringr::str_detect(variable, "famil|liked")) %>%
  group_by(variable)

params.of.interest %>%
  tidybayes::mean_hdi() %>%
  pander::pandoc.table(caption="Estimates and 95% credible intervals for the parameters of interest")
```

variable	value	.lower	.upper	.width	.point	.interval
familiar1	0.1992	-0.0688	0.464	0.95	mean	hdi
liked1	0.2989	0.02668	0.5567	0.95	mean	hdi
familiar1:liked1	-0.1794	-0.4393	0.09208	0.95	mean	hdi

Table: Estimates and 95% credible intervals for the parameters of interest

Bayesian ‘p values’ for parameters

We can do simple arithmetic with the posterior draws to calculate the probability a parameter is greater than (or less than) zero:

```
params.of.interest %>%
  summarise(estimate=mean(value),
            `p (x<0)` = mean(value < 0),
            `p (x>0)` = mean(value > 0))
# A tibble: 3 x 4
  variable      estimate `p (x<0)` `p (x>0)`
  <fct>          <dbl>     <dbl>     <dbl>
1 familiar1      0.199    0.0682    0.932
2 liked1         0.299    0.00975   0.990
3 familiar1:liked1 -0.179   0.903     0.0972
```

Or if you’d like the Bayes Factor (evidence ratio) for one hypotheses vs another, for example comparing the hypotheses that a parameter is $>$ vs. ≤ 0 , then you can use the `hypothesis` function in the `brms` package:

```
pain.model.mcmc.df <-
  pain.model.mcmc %>%
  as_tibble

brms:::hypothesis(pain.model.mcmc.df,
                  c("familiar1 > 0",
                    "liked1 > 0",
                    "familiar1:liked1 < 0"))
Hypothesis Tests for class :
  Hypothesis Estimate Est.Error CI.Lower CI.Upper Evid.Ratio
1 (familiar1) > 0     0.20      0.14    -0.02      Inf     13.65
2 (liked1) > 0       0.30      0.13     0.07      Inf    101.56
3 (familiar1:liked1) < 0 -0.18      0.14     -Inf      0.04      9.28
  Post.Prob Star
1      0.93
2      0.99 *
3      0.90
---
`*`: The expected value under the hypothesis lies outside the 95%-CI.
Posterior probabilities of point hypotheses assume equal prior probabilities.
```

Here although we only have a ‘significant’ p value for one of the parameters, we can also see there is “very strong” evidence that familiarity also influences pain, and “strong” evidence for the interaction of familiarity and liking, according to conventional rules of thumb when interpreting Bayes Factors.

TODO - add a fuller explanation of why multiple comparisons are not an issue for Bayesian analysis [Gelman et al., 2012], because p values do not have the same interpretation in terms of long run frequencies of replication; they are a representation of the weight of the evidence in favour of a hypothesis.

TODO: Also reference Zoltan Dienes Bayes paper.

13 Power analysis

For most inferential statistics

If you want to do power analysis for a standard statistical test, e.g. t-tests, chi² or Anova, the `pwr::` package is what you need. This guide has a good walkthrough.

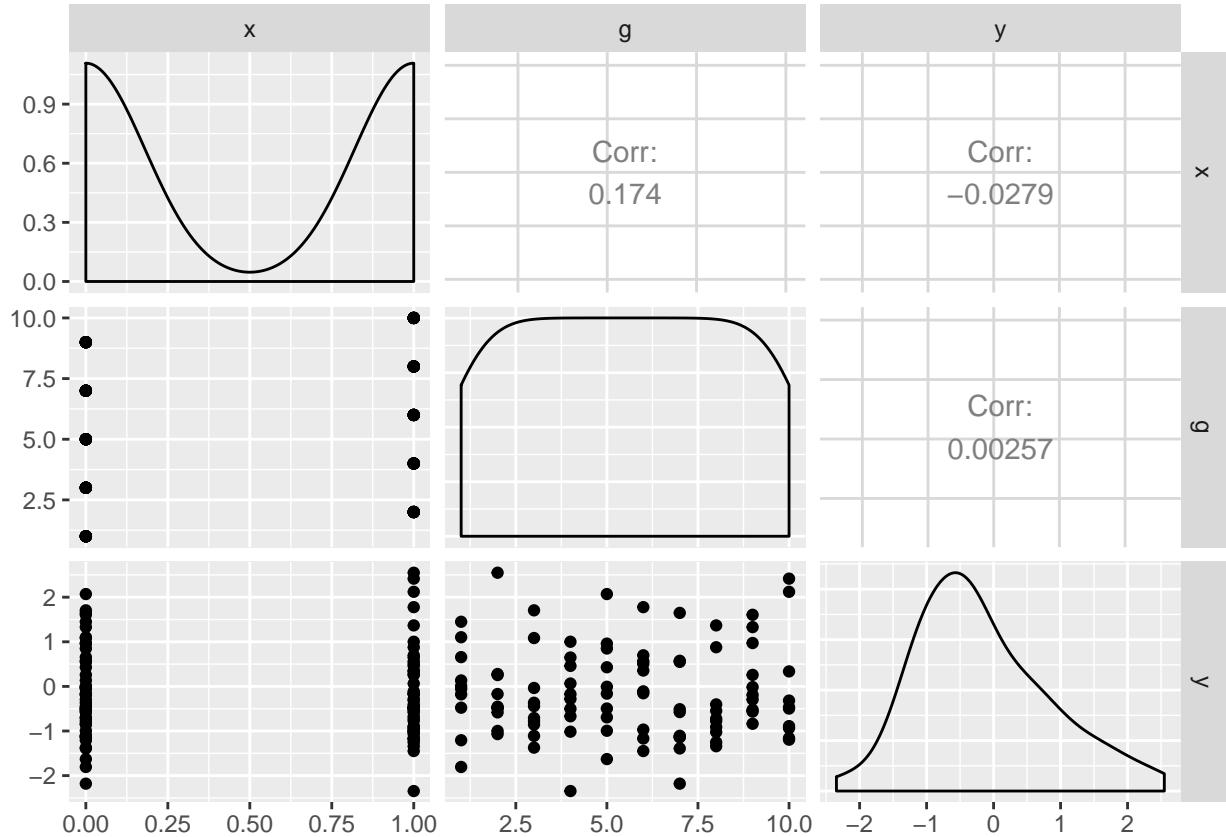
For multilevel or generalised linear models

If you'd like to run power analyses for linear mixed models (multilevel models) then you need the `simr::` package. It has some neat features for calculating power by simulating data and results from a model you specify.

To take a simple example, let's fabricate some data where outcomes `y` are nested within groups `g`, and where there is a covariate of interest `x`. That is, we are interested in estimating our power to detect an effect of `x`.

Note though that in this simulated data, we create outcomes (`y`) with no relationship to `x`, and no clustering by `g` at this stage (i.e. `y`, `x` and `g` are uncorrelated).

```
set.seed(1234)
simulated.df <- data_frame(
  x = rep(0:1, 50),
  g = rep(1:10, 10),
  y = rnorm(100)
)
Warning: `data_frame()` is deprecated, use `tibble()` .
This warning is displayed once per session.
GGally::ggpairs(simulated.df)
Registered S3 method overwritten by 'GGally':
  method from
  +.gg   ggplot2
```



To use `simr`: we first run our ‘model of interest’. In this case it’s a random-intercepts multilevel model. We can verify `x` is unrelated to outcome with `lmerTest::anova`:

```
library(lmerTest)

Attaching package: 'lmerTest'
The following object is masked from 'package:lme4':
    lmer

The following object is masked from 'package:stats':
    step

model.of.interest <- lmer(y ~ x + (1|g), data=simulated.df)
boundary (singular) fit: see ?isSingular
anova(model.of.interest)
Type III Analysis of Variance Table with Satterthwaite's method
  Sum Sq Mean Sq NumDF DenDF F value Pr(>F)
x 0.077811 0.077811     1     98  0.0764 0.7828
```

The next step is to modify our saved model of interest. We tweak the fitted parameters to represent our predicted values for effect sizes, variances, and covariances.

First, let’s change the ‘true’ effect of `x` to be 0.2:

```
fixef(model.of.interest)[‘x’] <- .2
```

We now use the `powerSim` function to use our tweaked model to:

- create a dataset using the parameters of our model (i.e. make random draws of `y` which relate to `g` and

- `x` as specified in the model summary).
- re-run `lmer` on this simulated data.
 - repeat this hundreds or thousands of times
 - count how many times (i.e., for what proportion) we get a significant p value

```
power.sim
Power for predictor 'x', (95% confidence interval):
  11.00% ( 5.62, 18.83)

Test: unknown test
Effect size for x is 0.20

Based on 100 simulations, (1 warning, 0 errors)
alpha = 0.05, nrow = 100

Time elapsed: 0 h 0 m 12 s
```

Our observed power (proportion of times we get a significant p value) is very low here, so we might want increase our hypothesised effect of `x`, for example to see what power we have to detect an effect of $x = .8$:

```
fixef(model.of.interest)['x'] <- .8

power.sim
Power for predictor 'x', (95% confidence interval):
  95.00% (88.72, 98.36)

Test: unknown test
Effect size for x is 0.80

Based on 100 simulations, (0 warnings, 0 errors)
alpha = 0.05, nrow = 100

Time elapsed: 0 h 0 m 11 s
```

We might also want to set one of the variance parameters of our model to represent clustering within-`g`. First we can use `VarCorr()` to check the variance parameters of the model we just ran:

```
VarCorr(model.of.interest)
Groups   Name      Std.Dev.
g        (Intercept) 0.0000
Residual           1.0091
```

And we could simulate increasing the variance parameter for `g` to 0.5:

```
VarCorr(model.of.interest)['g'] <- .5

power.sim
Power for predictor 'x', (95% confidence interval):
  33.00% (23.92, 43.12)

Test: unknown test
Effect size for x is 0.80

Based on 100 simulations, (0 warnings, 0 errors)
alpha = 0.05, nrow = 100

Time elapsed: 0 h 0 m 11 s
```

Because the amount of clustering in our data has increased our statistical power has gone down. This is because, when clustering is present, each new observation (row) in the dataset provides less new *information* to estimate our treatment effect. Note that in this example we increased the variance associated with g by quite a lot: setting the variance of g to 0.5 equates to an ICC for g of .33 (because $.5 / (.5 + 1) = .33$; see the section on calculating ICCs and VPCs)

For more details of `simr` see: <http://onlinelibrary.wiley.com/doi/10.1111/2041-210X.12504/full>

Note that for real applications you would want to set `nsim` to something reasonably large. Certainly at least 1000 simulations, and perhaps up to ~5000.

Part IV

Patterns

14 Learning key patterns



Statistics courses teach statistics, and software training or programming courses can teach you the fundamentals of a particular package — for example R.

However, a large part of learning how to analyse data is actually not about particular statistical techniques, nor the details of implementing them in a particular package.

Instead, the most useful things to take away from a course can be how to think about different types of problem, and general strategies for tackling them. One label applied to these approaches and strategies are ‘patterns’.

We’ve already come across one pattern in the section on summarising data: the split, apply, combine method.

This section outlines some other patterns or ways of working that may be helpful.

15 Unpicking interactions

Objectives of this section:

- Clarify/recap what an interaction is
- Appreciate the importance of visualising interactions
- Compare different methods of plotting interactions in raw data
- Visualise interactions based on statistical model predictions
- Deal with cases where predictors are both categorical and continuous (or a mix)

What is an interaction?

For an interaction to occur we must measure:

- An *outcome*: severity of injury in a car crash, for example.
- At least 2 *predictors* of that outcome: e.g. age and gender.

Let's think of a scenario where we've measured severity of injury after road accidents, along with the age and gender of the drivers involved. Let's assume¹⁰:

- Women are likely to be more seriously injured than men in a crash (a +10 point increase in severity)
- Drivers over 60 are more likely to be injured than younger drivers (+10 point severity vs <60 years)

For an interaction to occur we have to show that, for example:

- If you are old and also female then you will be more severely injured
- This increase in severity of injury is more than we would expect simply by adding the effects for being female (+10 points) and for being over 60 (+10 points). That is, if an interaction occurs the risk of being older and female is > a 20 point increase in severity.

Think of some other example of interactions from your own work.

Interactions capture the idea that the *effect* of one predictor changes depending on the value of another predictor.

Visualising interactions from raw data

In the previous section we established that interactions capture the idea that the *effect* of one predictor changes depending on the value of another predictor.

We can see this illustrated in the traditional bar plot below. In the left panel we see a dummy dataset in which there is no interaction; in the right panel are data which do show evidence of an interaction:

Warning: `as_data_frame()` is deprecated, use `as_tibble()` (but mind the new semantics). This warning is displayed once per session.

However this bar plot might be better if it were re-drawn as a point and line plot:

The reason the point and line plot improves on the bars for a number of reasons:

- Readers tend to misinterpret bar plots by assuming that values ‘above’ the bar are less likely than values contained ‘within’ the bar, when this is not the case [Newman and Scholl, 2012].

¹⁰This example is loosely based on figures reported by Kockelman and Kweon [2002]

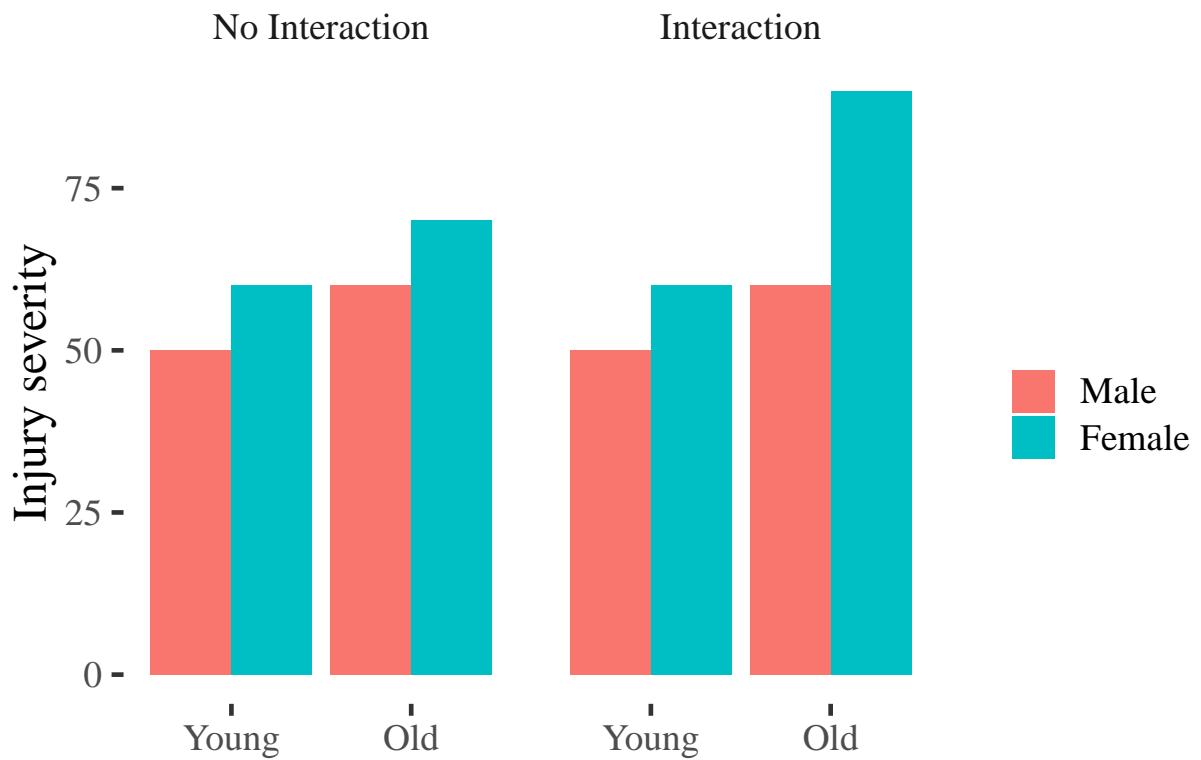


Figure 17: Bar plot of injury severity by age and gender.

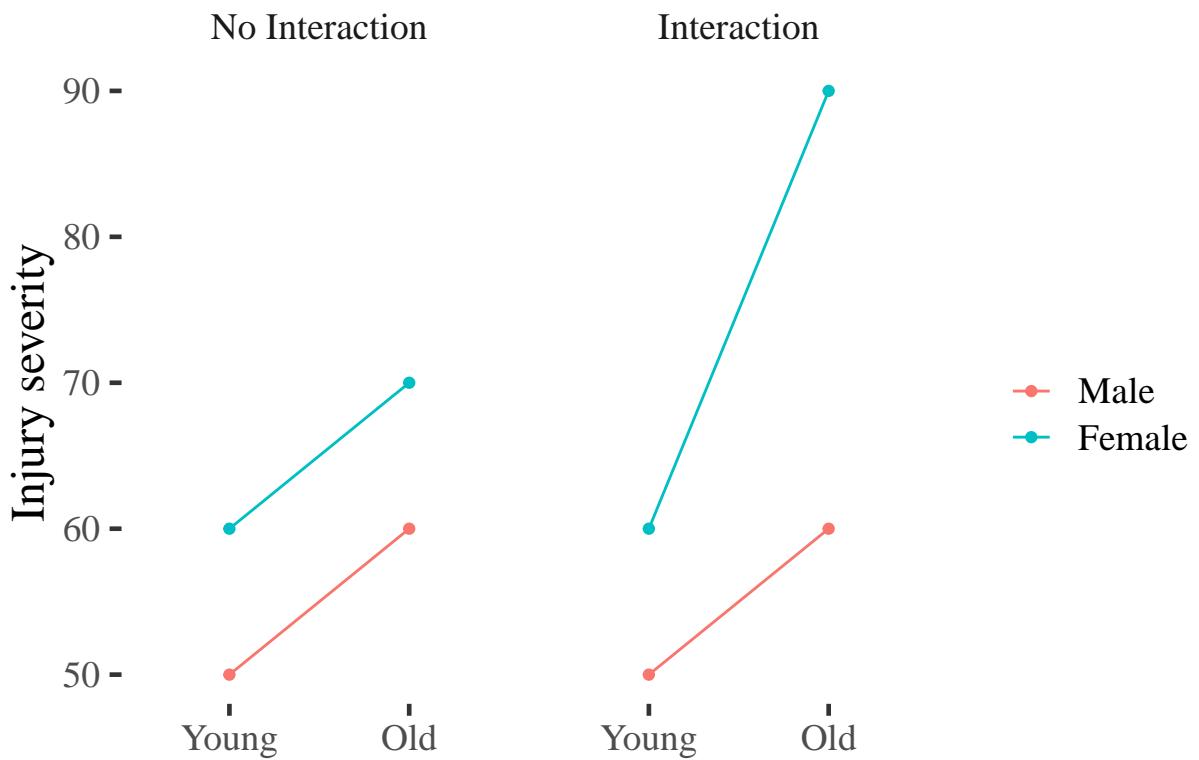


Figure 18: Point and line plot of injury severity by age and gender.

- The main effects are easy to distinguish in the line plot: just ask yourself if the lines are horizontal or not, and whether they are separated vertically. In contrast, reading the interaction from the bar graph requires that we average pairs of bars (sometimes not adjacent to one another) and compare them - a much more difficult mental operation.
- The interaction is easy to spot: Ask yourself if the lines are parallel. If they *are* parallel then the *difference* between men and women is constant for individuals of different ages.

A painful example

Before setting out to *test* for an interaction using some kind of statistical model, it's always a good idea to first visualise the relationships between outcomes and predictors.

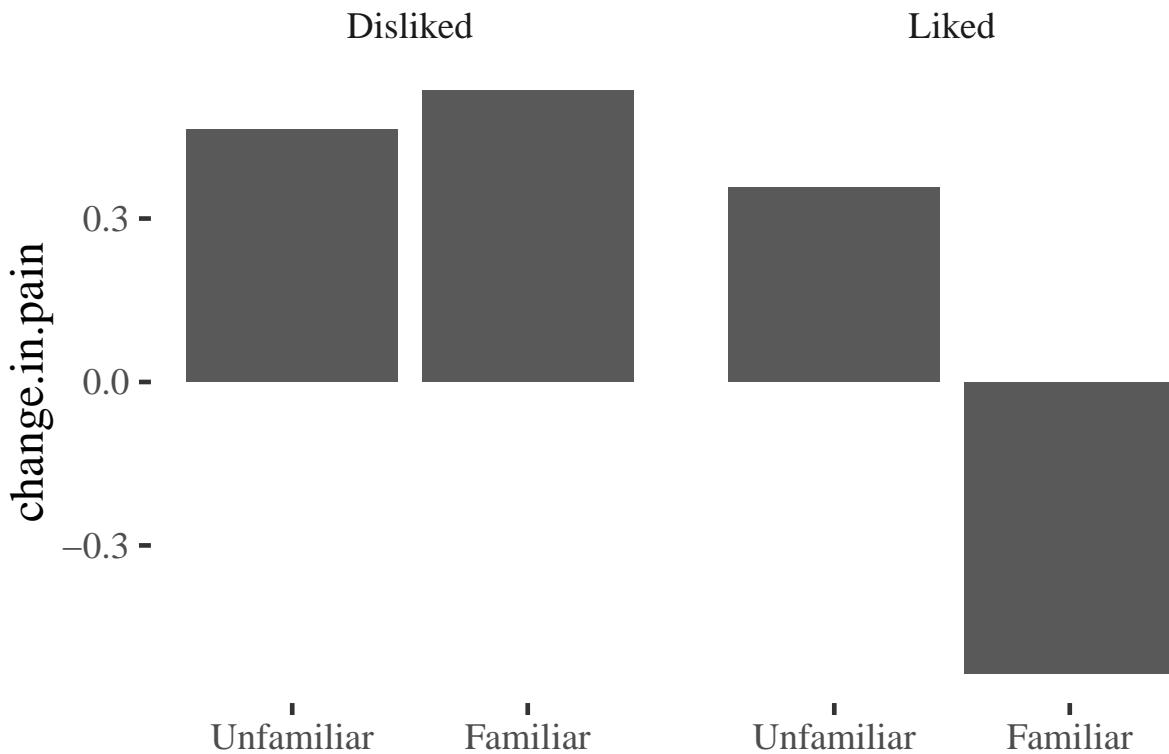
A student dissertation project investigated the analgesic quality of music during an experimental pain stimulus. Music was selected to be either *liked* (or disliked) by participants and was either *familiar* or unfamiliar to them. Pain was rated without music (`no.music`) and with music (`with.music`) using a 10cm visual analog scale anchored with the labels “no pain” and “worst pain ever”.

```
painmusic <- readRDS('data/painmusic.RDS')
painmusic %>% glimpse
Observations: 112
Variables: 4
$ liked      <fct> Disliked, Disliked, Liked, Disliked, Liked, ...
$ familiar   <fct> Familiar, Unfamiliar, Familiar, Familiar, Familiar, ...
$ no.music   <dbl> 4, 4, 6, 5, 3, 2, 6, 6, 7, 2, 7, 3, 5, 7, 6, 3, 7, ...
$ with.music <dbl> 7, 8, 7, 3, 3, 1, 6, 8, 9, 8, 7, 5, 7, 8, 4, 5, 4, ...
```

Before running inferential tests, it would be helpful to see if the data are congruent with the study prediction that *liked* and *familiar* music would be more effective at reducing pain than disliked or unfamiliar music

We can do this in many different ways. The most common (but not the best) choice would be a simple bar plot, which we can create using the `stat_summary()` function from `ggplot2`.

```
painmusic %>%
  mutate(change.in.pain = with.music - no.music) %>%
  ggplot(aes(x = familiar, y=change.in.pain)) +
  facet_wrap(~liked) +
  stat_summary(geom="bar") + xlab("")
```

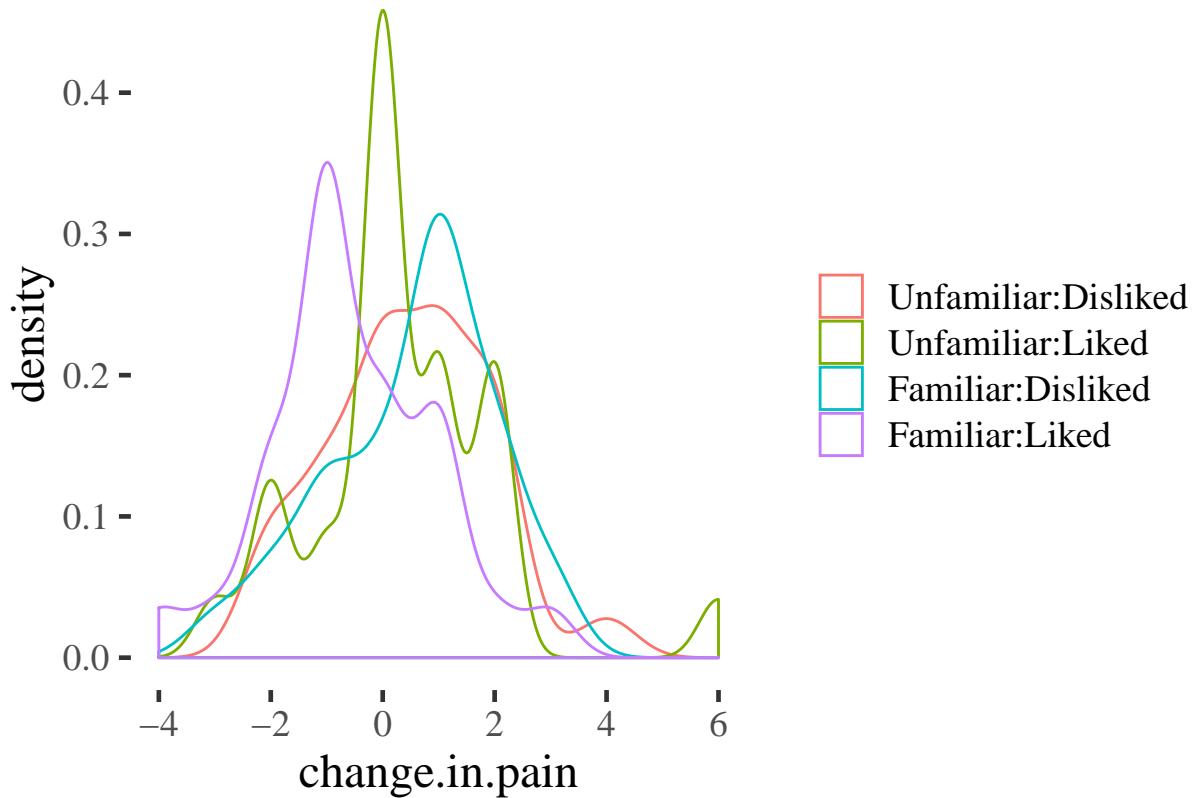


This gives a pretty clear indication that something is going on, but we have no idea about the *distribution* of the underlying data, and so how much confidence we should place in the finding. We are also hiding distributional information that could be useful to check that assumptions of models we run later are also met (for example of equal variances between groups).

If we want to preserve more information about the underlying distribution we can use density plots, boxplots, or pointrange plots, among others.

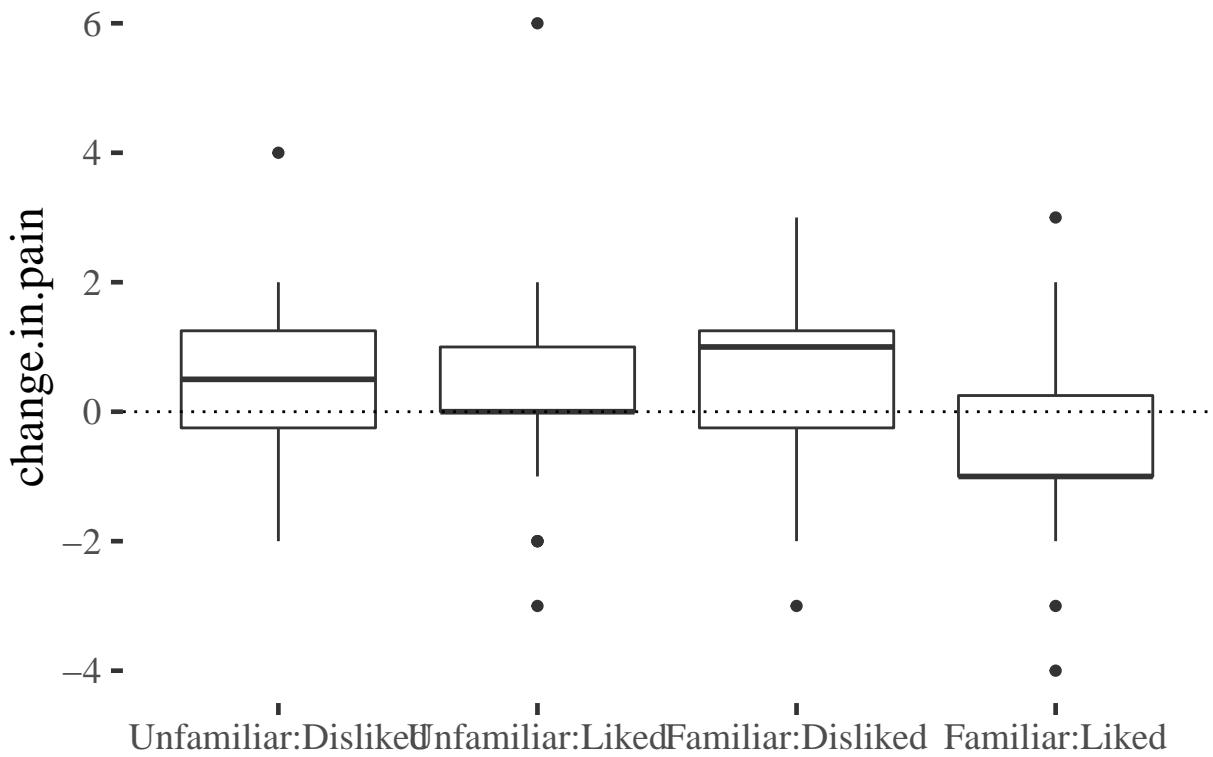
Here we use a grouped density plot. The `interaction()` function is used to automatically create a variable with the 4 possible groupings we can make when combining the `liked` and `familiar` variables:

```
painmusic %>%
  mutate(change.in.pain = with.music - no.music) %>%
  ggplot(aes(x = change.in.pain,
             color = interaction(familiar:liked))) +
  geom_density() +
  scale_color_discrete(name="")
```



And here we use a boxplot to achieve similar ends:

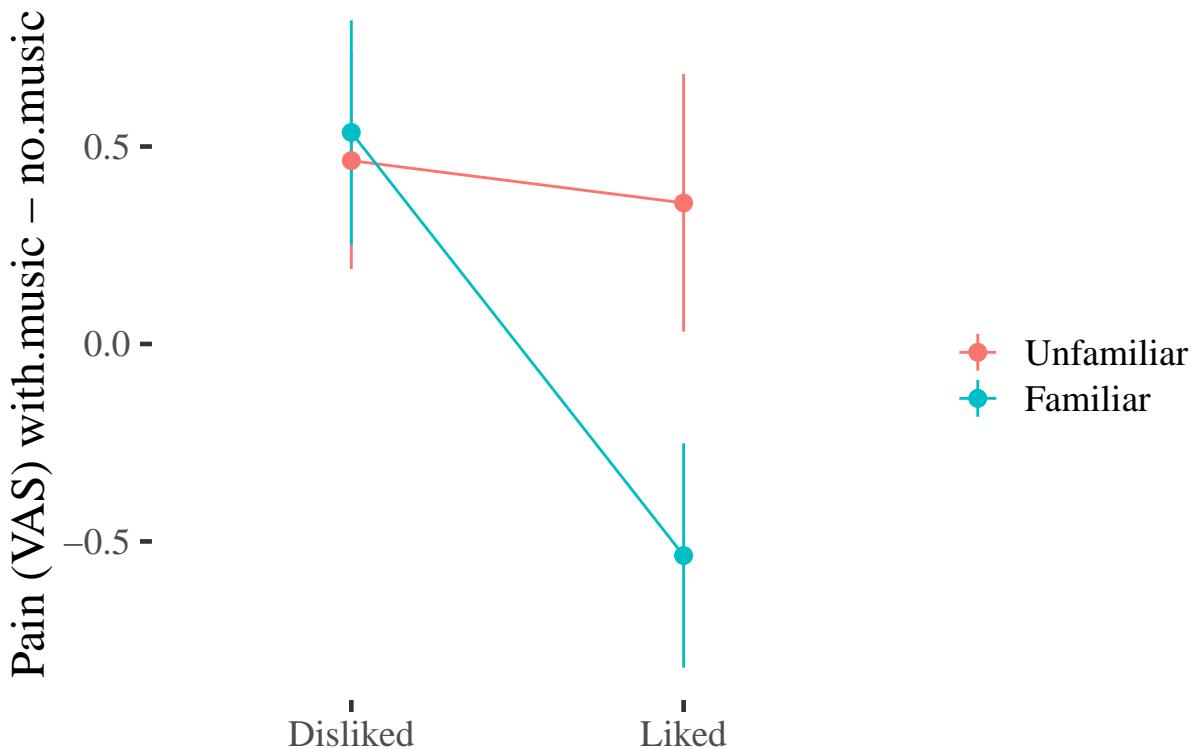
```
painmusic %>%
  mutate(change.in.pain = with.music - no.music) %>%
  ggplot(aes(x = interaction(familiar:liked), y = change.in.pain)) +
  geom_boxplot() +
  geom_hline(yintercept = 0, linetype="dotted") +
  xlab("")
```



The advantage of these last two plots is that they preserve quite a bit of information about the variable of interest. However, they don't make it easy to read the main effects and interaction as we saw for the point-line plot above.

We can combine some benefits of both plots by adding an error bar to the point-line plot:

```
painmusic %>%
  ggplot(aes(liked, with.music - no.music,
             group=familiar, color=familiar)) +
  stat_summary(geom="pointrange", fun.data=mean_se) +
  stat_summary(geom="line", fun.data=mean_se) +
  ylab("Pain (VAS) with.music - no.music") +
  scale_color_discrete(name="") +
  xlab("")
```



This plot doesn't include all of the information about the distribution of effects that the density or boxplots do (for example, we can't see any asymmetry in the distributions any more), but we still get some information about the variability of the effect of the experimental conditions on pain by plotting the SE of the mean over the top of each point¹¹

At this point, especially if your current data include only categorical predictors, you might want to move on to the section on making predictions from models and visualising these.

Continuous predictors

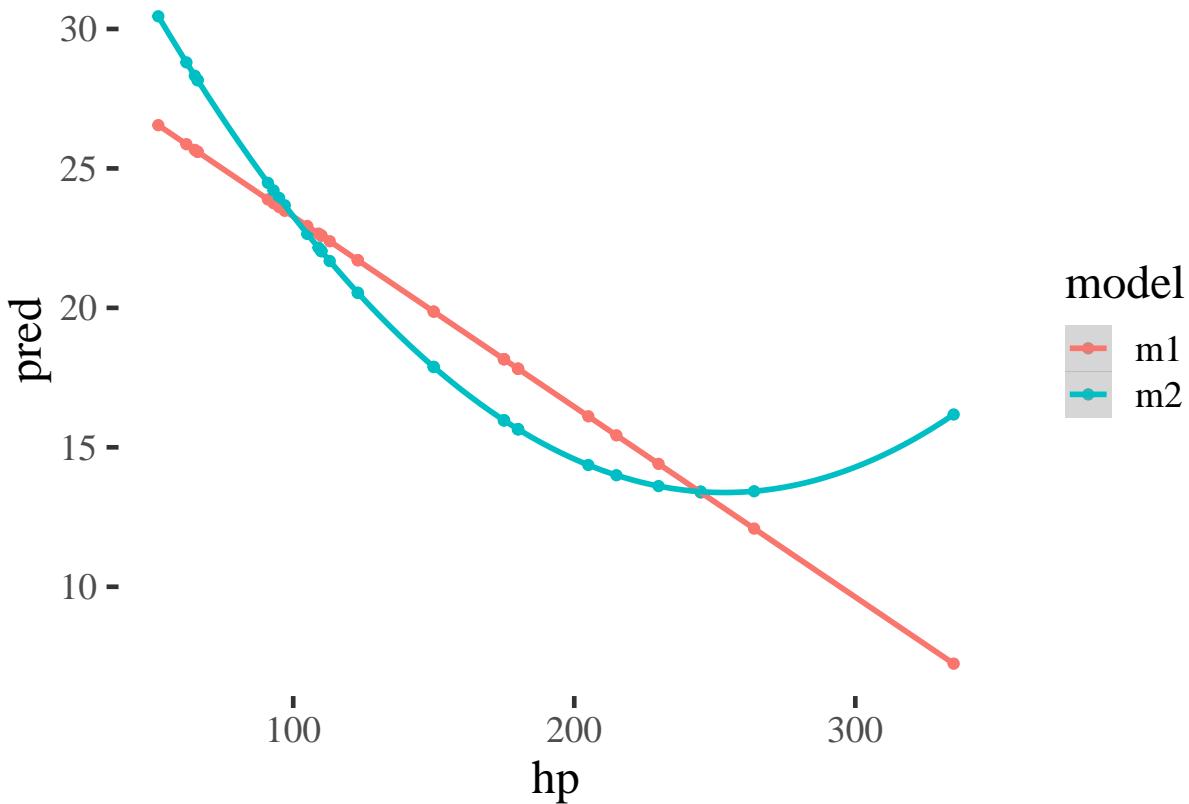
The `modelr` package contains useful functions which enable you to make predictions from models, and visualise them easily.

In this example we run two models, with and without a polynomial effect for `hp`. The predictions from both models are then plotted against one another.

```
library(modelr)
m1 <- lm(mpg ~ hp, data = mtcars)
m2 <- lm(mpg ~ poly(hp, 2), data = mtcars)

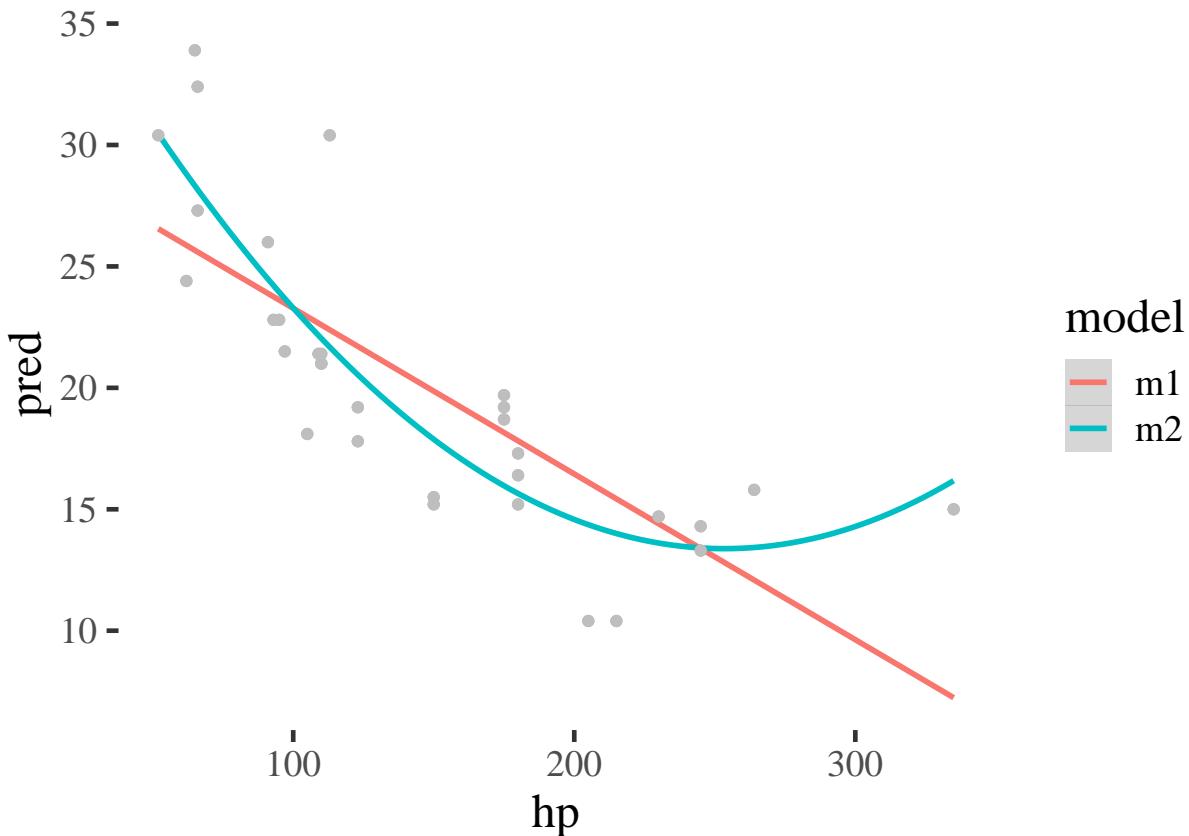
mtcars %>% gather_predictions(m1, m2) %>%
  ggplot(aes(hp, pred, color=model)) +
  geom_point() +
  geom_smooth()
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

¹¹We could equally well plot the 95% confidence interval for the mean, or the interquartile range)



We could also plot this over the top of the original data to give an example of how the models fit the data.

```
mtcars %>% gather_predictions(m1, m2) %>%
  ggplot(aes(hp, pred, color=model)) +
  geom_smooth() +
  geom_point(aes(y=mpg), color="grey")
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



The `gather_predictions` function can also be used to plot interactions.

```
m3 <- lm(mpg~wt*hp, data=mtcars)
summary(m3)

Call:
lm(formula = mpg ~ wt * hp, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max 
-3.0632 -1.6491 -0.7362  1.4211  4.5513 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 49.80842   3.60516 13.816 5.01e-14 ***
wt          -8.21662   1.26971 -6.471 5.20e-07 ***
hp         -0.12010   0.02470 -4.863 4.04e-05 ***
wt:hp        0.02785   0.00742  3.753 0.000811 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

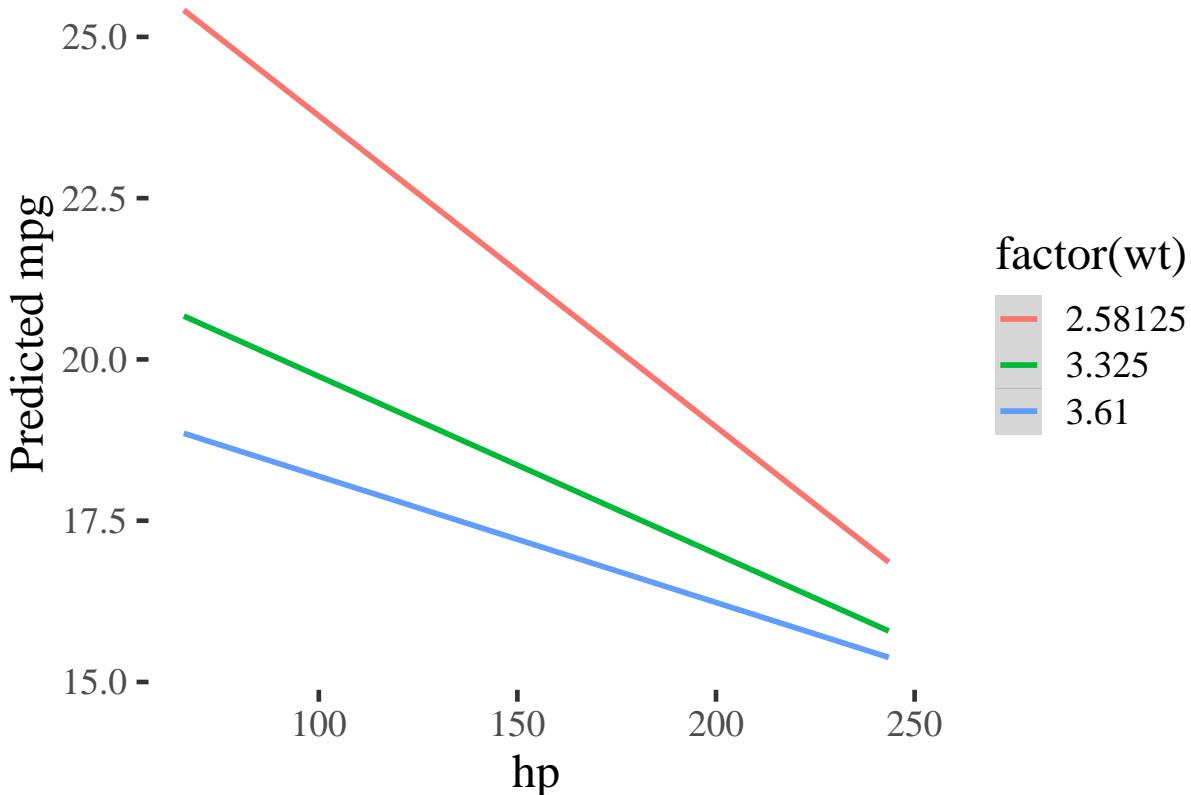
Residual standard error: 2.153 on 28 degrees of freedom
Multiple R-squared:  0.8848,    Adjusted R-squared:  0.8724 
F-statistic: 71.66 on 3 and 28 DF,  p-value: 2.981e-13
```

By making a new grid of data, using `expand.grid()`, at values of interest to us, we can plot the interaction

and see that the effect of `wt` is diminished as `hp` increases.

```
grid <- expand.grid(wt = quantile(mtcars$wt, probs=c(.25,.5,.75)),
                     hp = quantile(mtcars$hp, probs=c(.1, .25,.5,.75, .9)))

grid %>%
  gather_predictions(m3) %>%
  ggplot(aes(hp, pred, color=factor(wt))) +
  geom_smooth(method="lm") +
  ylab("Predicted mpg")
```



16 Making predictions

Objectives of this section:

- Distinguish predicted means (predictions) from predicted effects ('margins')
- Calculate both predictions and marginal effects for a `lm()`
- Plot predictions and margins
- Think about how to plot effects in meaningful ways

Predictions vs margins

Before we start, let's consider what we're trying to achieve in making predictions from our models. We need to make a distinction between:

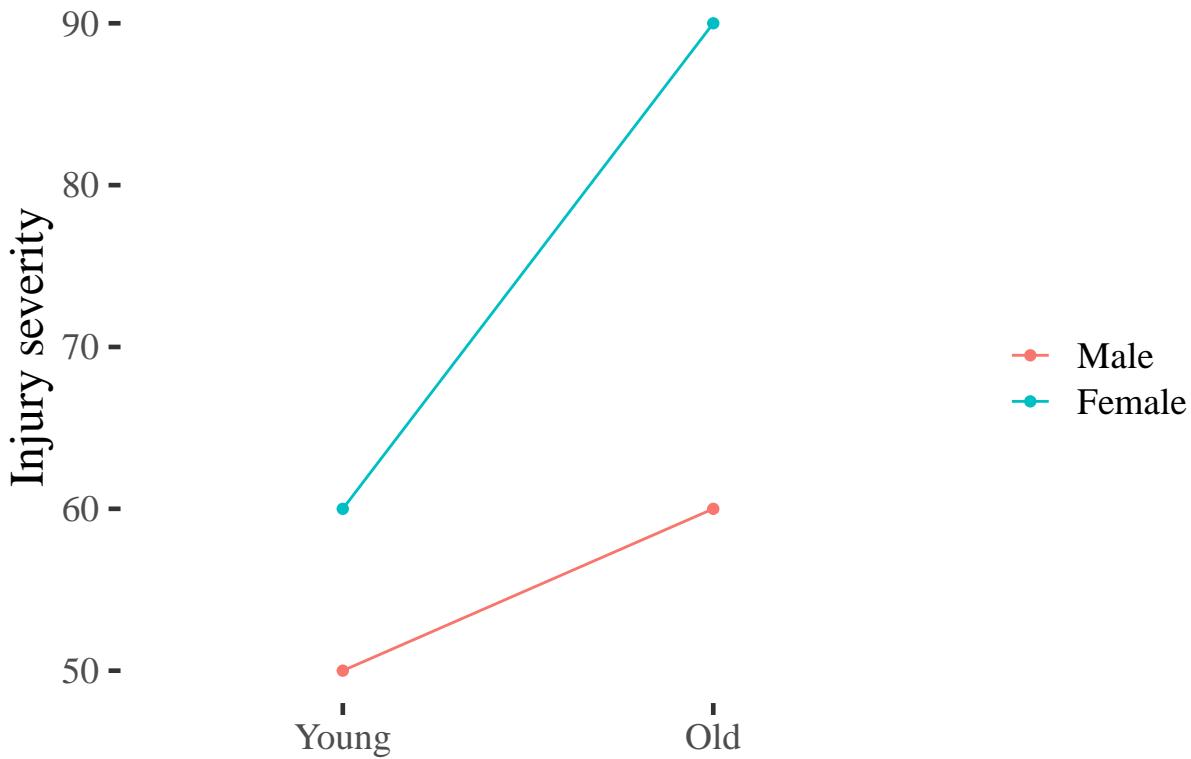


Figure 19: Point and line plot of injury severity by age and gender.

- Predicted means
- Predicted effects or *marginal effects*

Consider the example used in a previous section where we measured `injury.severity` after road accidents, plus two predictor variables: `gender` and `age`.

Predicted means

‘Predicted means’ (or predictions) refers to our best estimate for each category of person we’re interested in. For example, if `age` were categorical (i.e. young vs. older people) then might have 4 predictions to calculate from our model:

Age	Gender	mean
Young	Male	?
Old	Male	?
Young	Female	?
Old	Female	?

And as before, we might plot these data:

This plot uses the raw data, but these points could equally have been estimated from a statistical model which adjusted for other predictors.

Effects (margins)

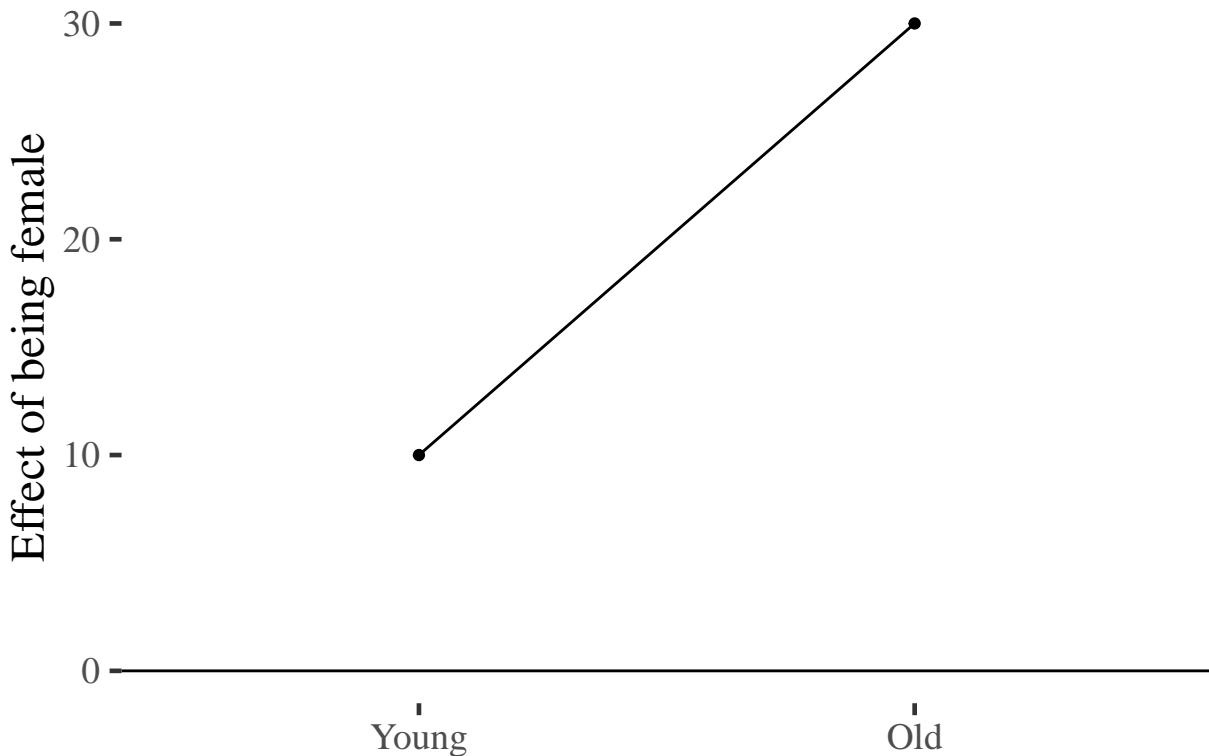
Terms like: *predicted effects*, *margins* or *marginal effects* refer, instead, to the effect of one predictor.

There may be more than one marginal effect because *the effect of one predictor can change across the range of another predictor*.

Extending the example above, if we take the difference between men and women for each category of age, we can plot these differences. The steps we need to go through are:

- Reshape the data to be wide, including a separate column for injury scores for men and women
- Subtract the score for men from that of women, to calculate the effect of being female
- Plot this difference score

```
margins.plot <- inter.df %>%
  # reshape the data to a wider format
  reshape2::dcast(older~female) %>%
  # calculate the difference between men and women for each age
  mutate(effect.of.female = Female - Male) %>%
  # plot the difference
  ggplot(aes(older, effect.of.female, group=1)) +
  geom_point() +
  geom_line() +
  ylab("Effect of being female") + xlab("") +
  geom_hline(yintercept = 0)
margins.plot
```



As before, these differences use the raw data, but *could* have been calculated from a statistical model. In

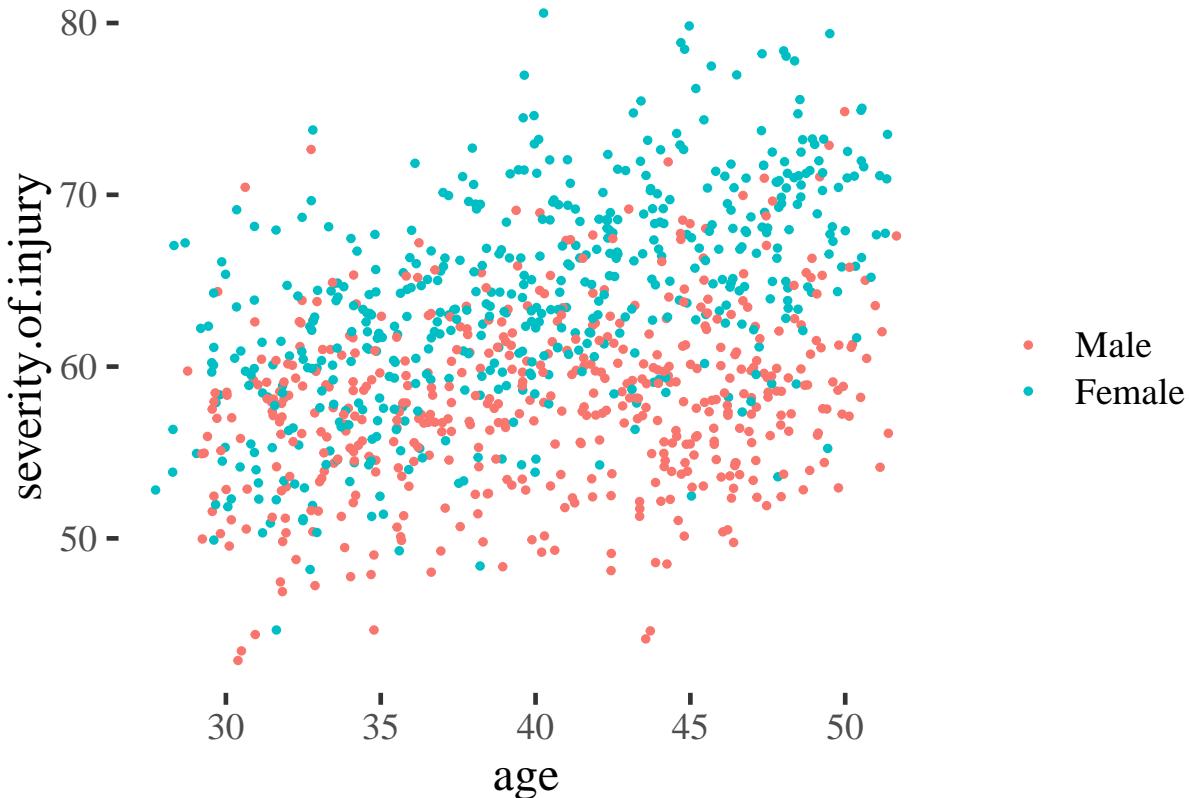
the section below we do this, making predictions for means and marginal effects from a `lm()`.

Continuous predictors

In the examples above, our data were all categorical, which mean that it was straightforward to identify categories of people for whom we might want to make a prediction (i.e. young men, young women, older men, older women).

However, `age` is typically measured as a continuous variable, and we would want to use a grouped scatter plot to see this:

```
injuries %>%  
  ggplot(aes(age, severity.of.injury, group=gender, color=gender)) +  
    geom_point(size=1) +  
    scale_color_discrete(name="")
```



But to make predictions from this continuous data we need to fit a line through the points (i.e. run a model). We can do this graphically by calling `geom_smooth()` which attempts to fit a smooth line through the data we observe:

```
injuries %>%  
  ggplot(aes(age, severity.of.injury, group=gender, color=gender)) +  
    geom_point(alpha=.2, size=1) +  
    geom_smooth(se=F)+  
    scale_color_discrete(name="")
```

And if we are confident that the relationships between predictor and outcome are sufficiently *linear*, then we can ask ggplot to fit a straight line using linear regression:

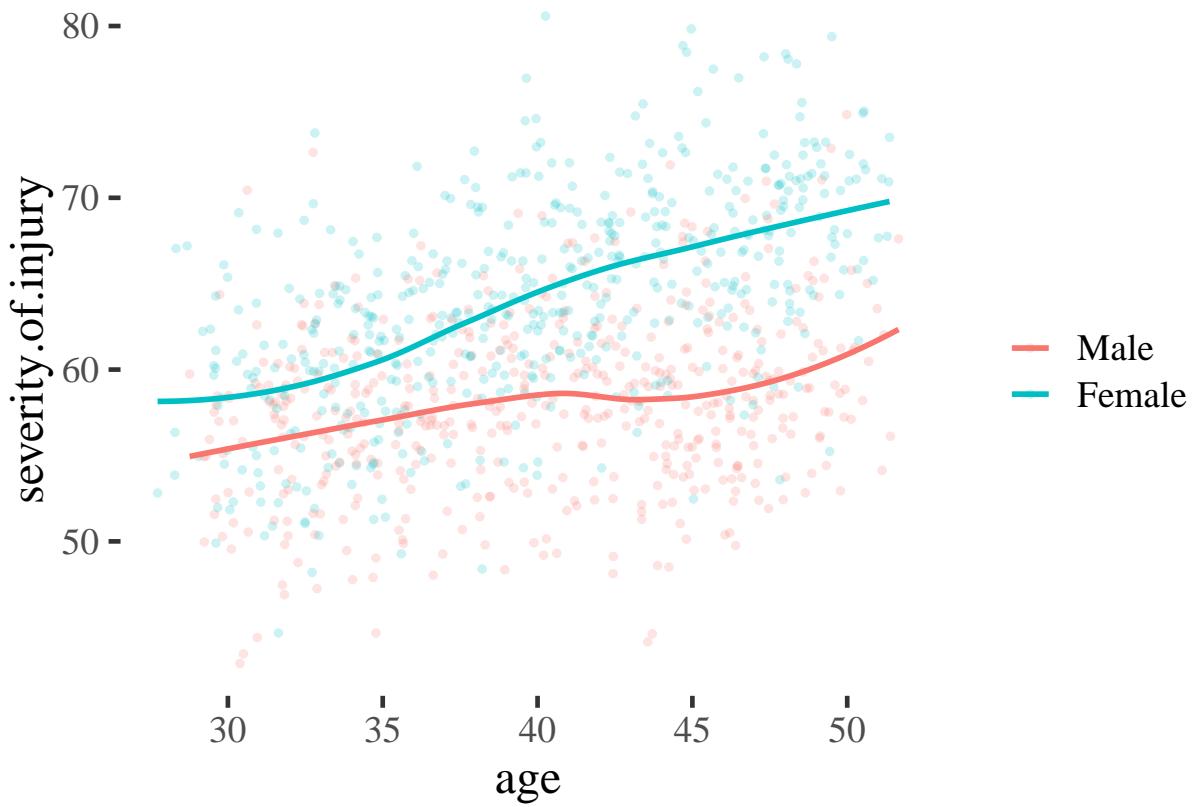


Figure 20: Scatter plot overlaid with smooth best-fit lines

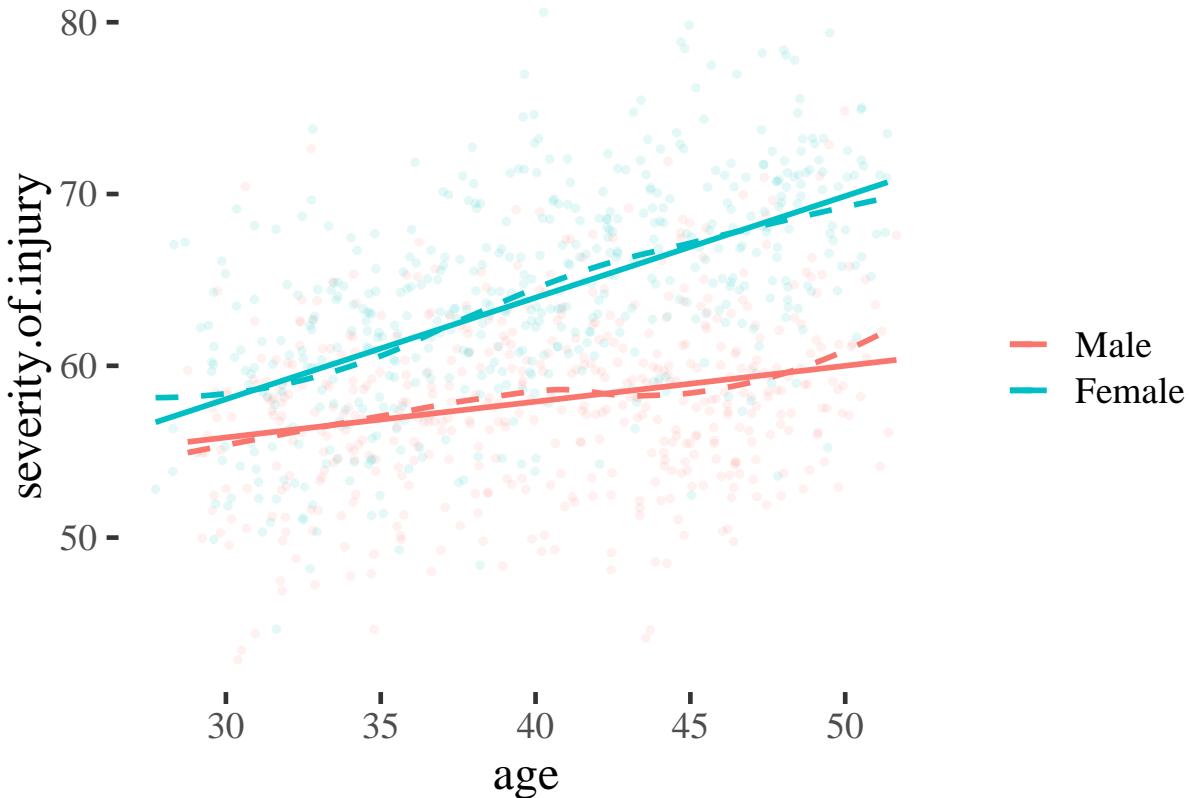


Figure 21: Scatter plot overlaid with smoothed lines (dotted) and linear predictions (coloured)

```

injuries %>%
  ggplot(aes(age, severity.of.injury, group=gender, color=gender)) +
  geom_point(alpha = .1, size = 1) +
  geom_smooth(se = F, linetype="dashed") +
  geom_smooth(method = "lm", se = F) +
  scale_color_discrete(name="")

```

What these plots illustrate is the steps a researcher might take *before* fitting a regression model. The straight lines in the final plot represent our best guess for a person of a given age and gender, assuming a linear regression.

We can read from these lines to make a point prediction for men and women of a specific age, and use the information about our uncertainty in the prediction, captured by the model, to estimate the likely error.

To make our findings simpler to communicate, we might want to make estimates at specific ages and plot these. These ages could be:

- Values with biological or cultural meaning: for example 18 (new driver) v.s. 65 (retirement age)
- Statistical convention (e.g. median, 25th, and 75th centile, or mean +/- 1 SD)

We'll see examples of both below.

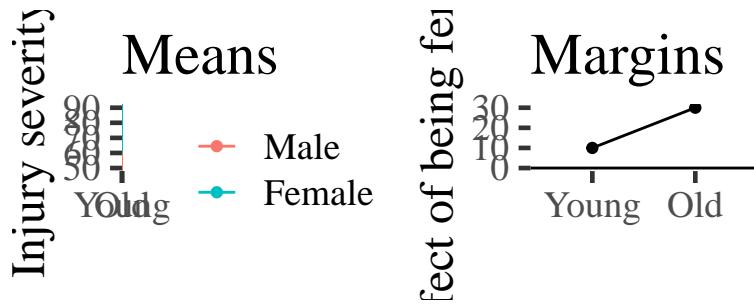


Figure 22: Example of predicted means vs. margins. Note, the margin plotted in the second panel is the difference between the coloured lines in the first. A horizontal line is added at zero in panel 2 by convention.

Predicted means and margins using `lm()`

The section above details two types of predictions: predictions for means, and predictions for margins (effects). We can use the figure below as a way of visualising the difference:

```
gridExtra::grid.arrange(means.plot+ggtitle("Means"), margins.plot+ggtitle("Margins"), ncol=2)
```

Running the model

Lets say we want to run a linear model predicts injury severity from gender and a categorical measurement of age (young v.s. old).

Our model formula would be: `severity.of.injury ~ age.category * gender`. Here we fit it an request the Anova table which enables us to test the main effects and interaction¹²:

```
injurymodel <- lm(severity.of.injury ~ age.category * gender, data=injuries)
anova(injurymodel)
Analysis of Variance Table

Response: severity.of.injury
              Df  Sum Sq Mean Sq F value    Pr(>F)
age.category      1  3920.1  3920.1 144.647 < 2.2e-16 ***
gender            1   9380.6  9380.6 346.134 < 2.2e-16 ***
age.category:gender  1   1148.7  1148.7  42.384 1.187e-10 ***
Residuals       996 26992.7     27.1
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Having saved the regression model in the variable `injurymodel` we can use this to make predictions for means and estimate marginal effects:

Making predictions for means

When making predictions, they key question to bear in mind is ‘predictions for what?’ That is, what values of the predictor variables are we going to use to estimate the outcome?

It goes like this:

1. Create a new dataframe which contains the values of the predictors we want to make predictions at
2. Make the predictions using the `predict()` function.

¹²Because this is simulated data, the main effects and interactions all have tiny p values.

3. Convert the output of `predict()` to a dataframe and plot the numbers.

16.0.0.1 Step 1: Make a new dataframe

```
prediction.data <- data_frame(  
  age.category = c("young", "older", "young", "older"),  
  gender = c("Male", "Male", "Female", "Female")  
)  
  
prediction.data  
# A tibble: 4 x 2  
  age.category gender  
  <chr>        <chr>  
1 young        Male  
2 older        Male  
3 young        Female  
4 older        Female
```

16.0.0.2 Step 2: Make the predictions

The R `predict()` function has two useful arguments:

- `newdata`, which we set to our new data frame containing the predictor values of interest
- `interval` which we here set to confidence¹³

```
injury.predictions <- predict(injurymodel, newdata=prediction.data, interval="confidence")  
  
injury.predictions  
  fit      lwr      upr  
1 56.91198 56.24128 57.58267  
2 58.90426 58.27192 59.53660  
3 60.84242 60.20394 61.48090  
4 67.12600 66.48119 67.77081
```

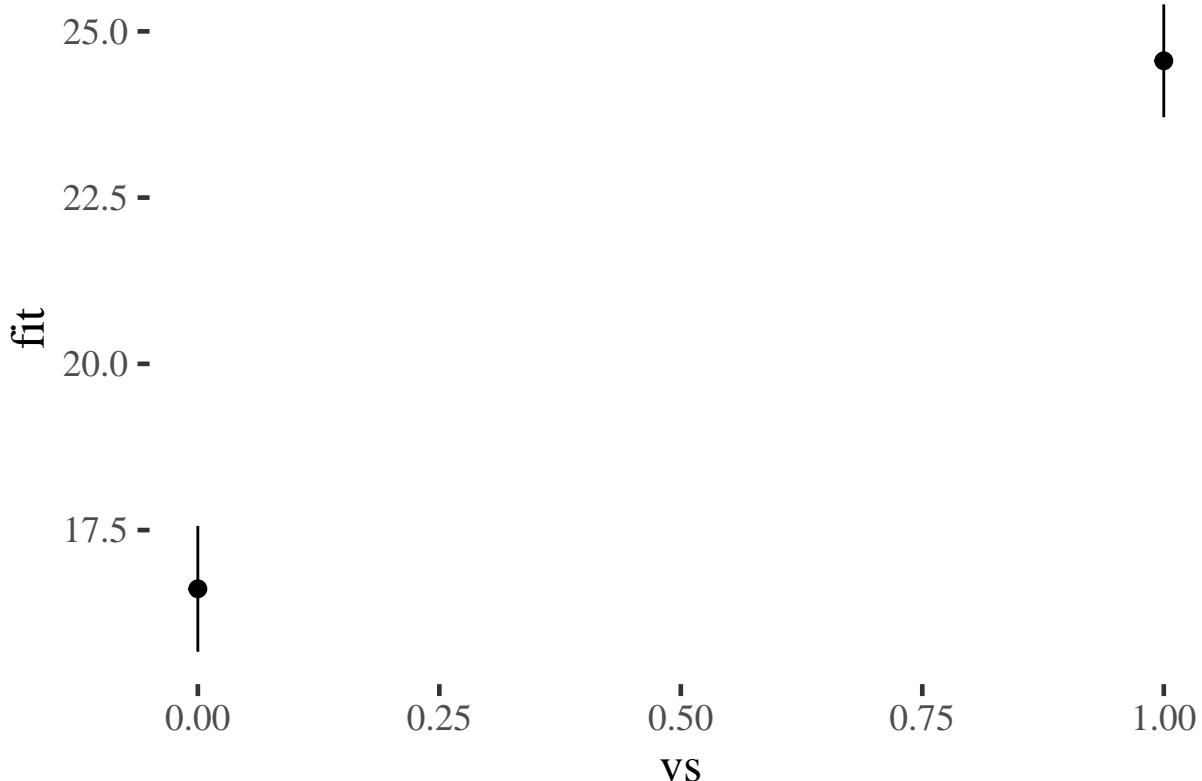
Making predictions for margins (*effects of predictors*)

```
library('tidyverse')  
m <- lm(mpg~vs+wt, data=mtcars)  
m.predictions <- predict(m, interval='confidence')  
  
mtcars.plus.predictions <- bind_cols(  
  mtcars,  
  m.predictions %>% as_data_frame()  
)  
  
prediction.frame <- expand.grid(vs=0:1, wt=2) %>%  
  as_data_frame()  
  
prediction.frame.plus.predictions <- bind_cols(  
  prediction.frame,  
  predict(m, newdata=prediction.frame, interval='confidence') %>% as_data_frame()
```

¹³This gives us the confidence interval for the prediction, which is the range within which we would expect the true value to fall, 95% of the time, if we replicated the study. We could ask instead for the `prediction` interval, which would be the range within which 95% of new observations with the same predictor values would fall. For more on this see the section on confidence v.s. prediction intervals

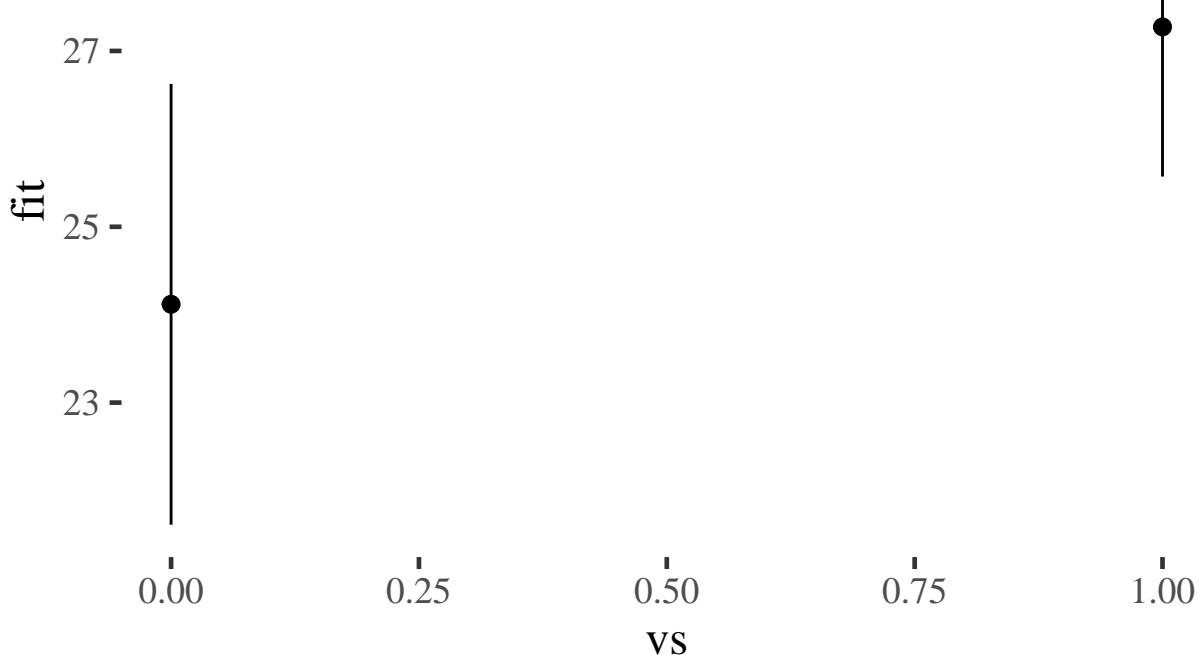
```
)
```

```
mtcars.plus.predictions %>%
  ggplot(aes(vs, fit, ymin=lwr, ymax=upr)) +
  stat_summary(geom="pointrange")
```



```
prediction.frame.plus.predictions %>% ggplot(aes(vs, fit, ymin=lwr, ymax=upr)) + geom_pointrange()
```

29 -



```
prediction.frame.plus.predictions
# A tibble: 2 x 5
  vs     wt   fit   lwr   upr
  <int> <dbl> <dbl> <dbl> <dbl>
1     0     2 24.1  21.6  26.6
2     1     2 27.3  25.6  29.0
mtcars.plus.predictions %>% group_by(vs) %>%
  summarise_each(funs(mean), fit, lwr, upr)
# A tibble: 2 x 4
  vs     fit   lwr   upr
  <dbl> <dbl> <dbl> <dbl>
1     0    16.6 14.9 18.3
2     1    24.6 22.8 26.3
```

Marginal effects

What is the effect of being black or female on the chance of you getting diabetes?

Two ways of computing, depending on which of these two you hate least:

- Calculate the effect of being black for someone who is 50% female (marginal effect at the means, MEM)
- Calculate the effect first pretending someone is black, then pretending they are white, and taking the difference between these estimate (average marginal effect, AME)

```
library(margins)
margins(m, at = list(wt = 1:2))
```

```

at(wt)      vs      wt
 1 3.154 -4.443
 2 3.154 -4.443

m2 <- lm(mpg~vs*wt, data=mtcars)
summary(m2)

Call:
lm(formula = mpg ~ vs * wt, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max 
-3.9950 -1.7881 -0.3423  1.2935  5.2061 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 29.5314    2.6221   11.263 6.55e-12 ***
vs           11.7667    3.7638    3.126  0.0041 **  
wt          -3.5013    0.6915   -5.063 2.33e-05 ***
vs:wt        -2.9097    1.2157   -2.393  0.0236 *   
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

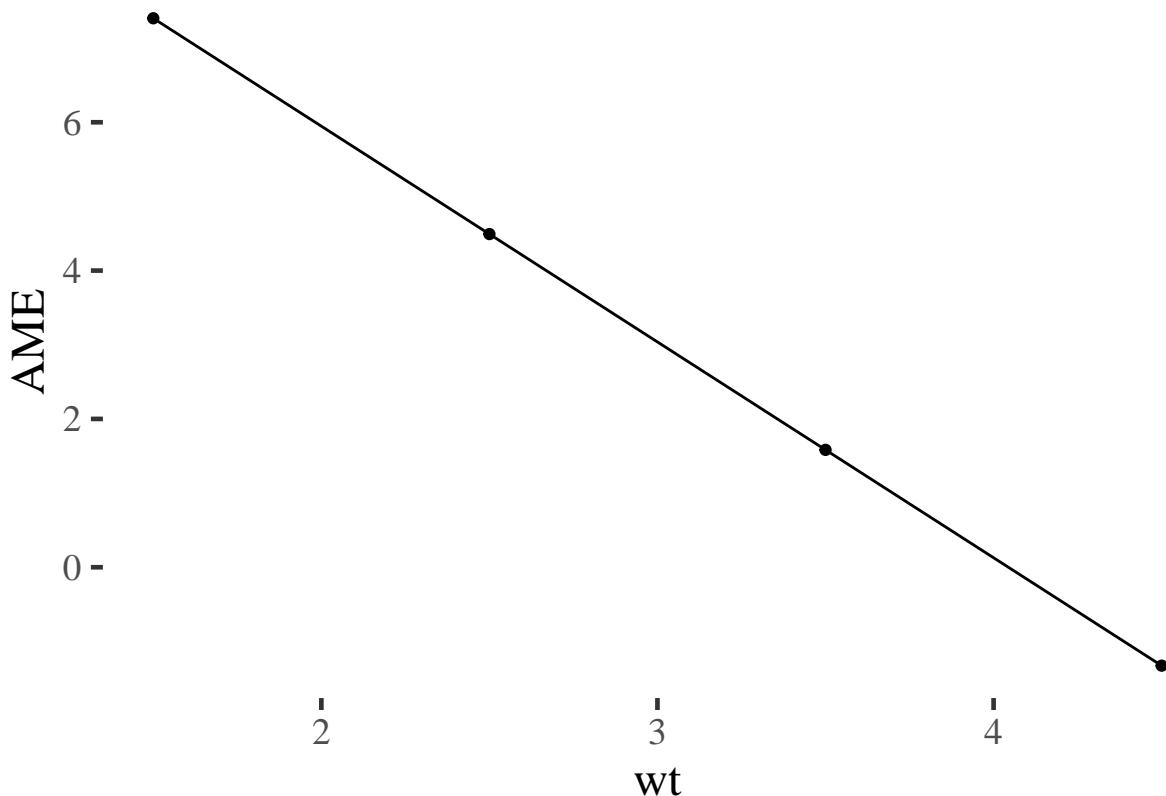
Residual standard error: 2.578 on 28 degrees of freedom
Multiple R-squared:  0.8348,    Adjusted R-squared:  0.8171 
F-statistic: 47.16 on 3 and 28 DF,  p-value: 4.497e-11

m2.margins <- margins(m2, at = list(wt = 1.5:4.5))

summary(m2.margins)
factor      wt      AME      SE      z      p    lower    upper
vs 1.5000  7.4021  2.0902  3.5414  0.0004  3.3055 11.4988
vs 2.5000  4.4924  1.2376  3.6300  0.0003  2.0668  6.9180
vs 3.5000  1.5827  1.2846  1.2320  0.2179 -0.9351  4.1006
vs 4.5000 -1.3270  2.1736 -0.6105  0.5415 -5.5872  2.9333
wt 1.5000 -4.7743  0.5854 -8.1561  0.0000 -5.9216 -3.6270
wt 2.5000 -4.7743  0.5854 -8.1557  0.0000 -5.9217 -3.6270
wt 3.5000 -4.7743  0.5854 -8.1561  0.0000 -5.9216 -3.6270
wt 4.5000 -4.7743  0.5854 -8.1561  0.0000 -5.9216 -3.6270

summary(m2.margins) %>% as_data_frame() %>%
  filter(factor=="vs") %>%
  ggplot(aes(wt, AME)) +
  geom_point() + geom_line()

```



Predictions with continuous covariates

- Run 2 x Continuous Anova
- Predict at different levels of X

Visualising interactions

Steps this page will work through:

- Running the the model (first will be a 2x2 between Anova)
- Using `predict()`.
- Creating predictions at specific values
- Binding predictions and the original data together.
- Using GGplot to layer points, lines and error bars.

17 Models are data

You might remember the episode of the Simpsons where Homer designs a car for ‘the average man’. It doesn’t end well. Traditional statistics packages are a bit like Homer’s car. They try to work for everyone, but in the process become bloated and difficult to use.

This is particularly true of the *output* of software like SPSS, which by default produces multiple pages of ‘results’ for even relatively simple statistical models. However, the problem is not just that SPSS is incredibly

verbose.

The real issue is that SPSS views the results of a model as the *end* of a process, rather than the beginning. The model SPSS has is something like:

1. Collect data
2. Choose analysis from GUI
3. Select relevant figures from pages of output and publish.

This is a problem because in real life it just doesn't work that way. In reality you will want to do things like:

- Run the same model for different outcomes
- Re-run similar models as part of a sensitivity analysis
- Compare different models and produce summaries of results from multiple models

All of this requires an *iterative process*, in which you may want to compare and visualise the results of multiple models. In a traditional GUI, this quickly becomes overwhelming.

However, if we treat modelling as a process which *both consumes and produces data*, R provides many helpful tools.

This is an important insight: in R, the results of analyses are not the end point — instead *model results are themselves data*, to be processed, visualised, and compared.

Storing models in variables

This may seem obvious (and we have seen many examples in the sections above), but because R variables can contain anything, we can use them to store the results of our models.

This is important, because it means we can keep track of different versions of the models we run, and compare them.

Extracting results from models

One of the nice things about R is that the `summary()` function will almost always provide a concise output of whatever model you send it, showing the key features of an model you have run.

However, this text output isn't suitable for publication, and can even be too verbose for communicating with colleagues. Often, when communicating with others, you want to focus in on the important details from analyses and to do this you need to extract results from your models.

Thankfully, there is almost always a method to extract results to a `dataframe`. For example, if we run a linear model:

```
model.fit <- lm(mpg ~ wt + disp, data=mtcars)
summary(model.fit)

Call:
lm(formula = mpg ~ wt + disp, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max 
-3.4087 -2.3243 -0.7683  1.7721  6.3484 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 34.96055   2.16454  16.151 4.91e-16 ***
wt          -3.35082   1.16413  -2.878  0.00743 **
```

```

disp      -0.01773   0.00919  -1.929  0.06362 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.917 on 29 degrees of freedom
Multiple R-squared:  0.7809,    Adjusted R-squared:  0.7658
F-statistic: 51.69 on 2 and 29 DF,  p-value: 2.744e-10

```

We can extract the parameter table from this model by saving the `summary()` of it, and then using the `$` operator to access the `coefficients` table (actually a matrix), which is stored within the summary object.

```

model.fit.summary <- summary(model.fit)
model.fit.summary$coefficients
  Estimate Std. Error t value Pr(>|t|)
(Intercept) 34.96055404 2.164539504 16.151497 4.910746e-16
wt          -3.35082533 1.164128079 -2.878399 7.430725e-03
disp        -0.01772474 0.009190429 -1.928609 6.361981e-02

```

'Poking around' with `$` and `@`

It's a useful trick to learn how to 'poke around' inside R objects using the `$` and `@` operators (if you want the gory details see this guide).

In the video below, I use RStudio's autocomplete feature to find results buried within a `lm` object:

For example, we could write the following to extract a table of coefficients, test statistics and *p* values from an `lm()` object (this is shown in the video):

```

model.fit.summary <- summary(model.fit)
model.fit.summary$coefficients %>%
  as_data_frame()
Warning: `as_data_frame()` is deprecated, use `as_tibble()` (but mind the new semantics).
This warning is displayed once per session.
# A tibble: 3 x 4
  Estimate `Std. Error` `t value` `Pr(>|t|)`
  <dbl>     <dbl>     <dbl>     <dbl>
1 35.0      2.16      16.2     4.91e-16
2 -3.35     1.16      -2.88    7.43e- 3
3 -0.0177    0.00919   -1.93    6.36e- 2

```

Save time: use a `broom`

The `broom::` library is worth learning because it makes it really easy to turn model results into dataframes, which is almost always what we want when working with data.

It takes a slightly different approach than simply poking around with `$` and `@`, because it providing general methods to 'clean up' the output of many older R functions.

For example, the `lm()` or `car::Anova` functions display results in the console, but don't make it easy to extract results as a dataframe. `broom::` provides a consistent way of extracting the key numbers from most R objects.

Let's say we have a regression model:

```
(model.1 <- lm(mpg ~ factor(cyl) + wt + disp, data=mtcars))
```

```

Call:
lm(formula = mpg ~ factor(cyl) + wt + disp, data = mtcars)

Coefficients:
(Intercept)  factor(cyl)6  factor(cyl)8          wt          disp
 34.041673     -4.305559     -6.322786    -3.306751     0.001715

```

We can extract model fit statistics — that is, attributes of the model as a whole — with `glance()`. This produces a dataframe:

```

glance(model.1)
# A tibble: 1 x 11
  r.squared adj.r.squared sigma statistic p.value    df logLik    AIC    BIC
      <dbl>        <dbl>   <dbl>    <dbl>     <dbl>  <int>  <dbl>  <dbl>  <dbl>
1     0.838       0.813   2.60    34.8 2.73e-10     5  -73.3  159.  167.
# ... with 2 more variables: deviance <dbl>, df.residual <int>

```

If we want to extract information about the model coefficients we can use `tidy`:

```

tidy(model.1, conf.int = T) %>%
  pandoc

```

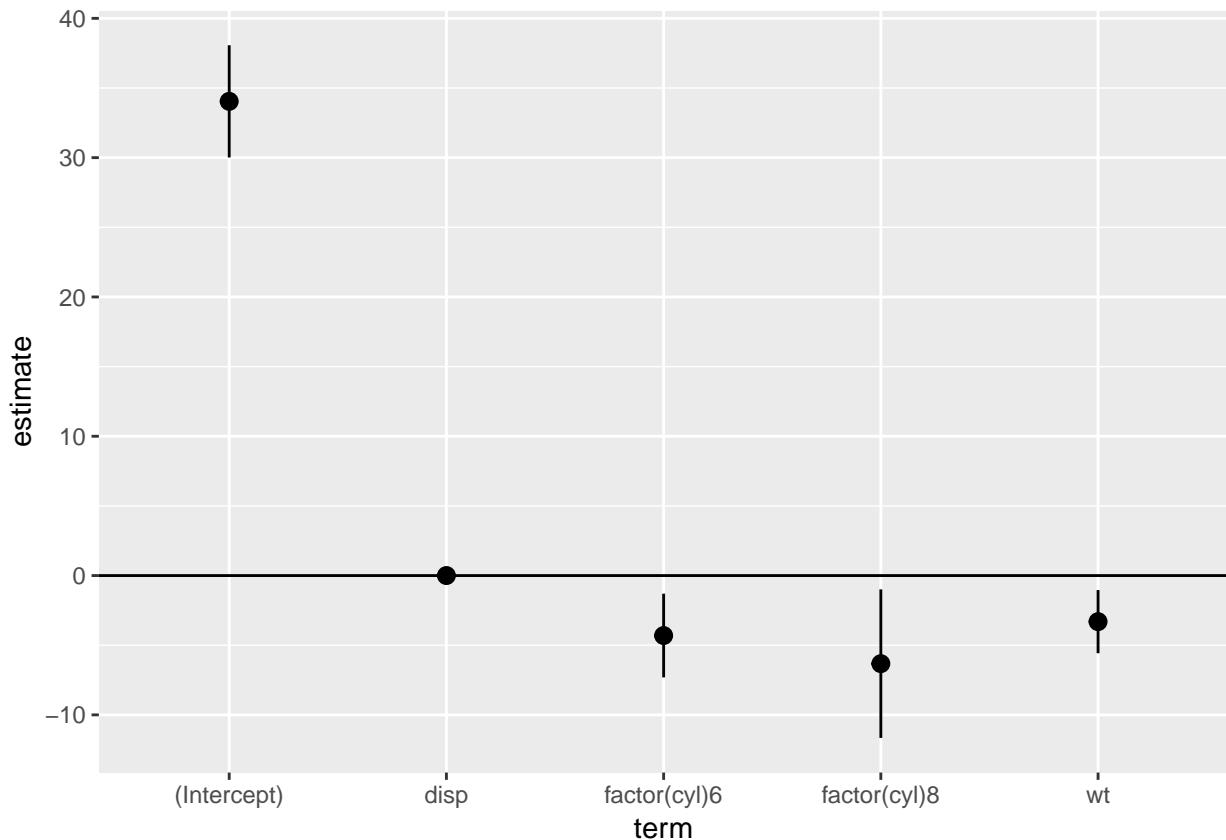
term	estimate	std.error	statistic	p.value	conf.low	conf.high
(Intercept)	34.04	1.963	17.34	3.662e-16	30.01	38.07
factor(cyl)6	-4.306	1.465	-2.939	0.006662	-7.311	-1.3
factor(cyl)8	-6.323	2.598	-2.433	0.02186	-11.65	-0.9913
wt	-3.307	1.105	-2.992	0.005855	-5.574	-1.039
disp	0.001715	0.01348	0.1272	0.8997	-0.02595	0.02938

Which can then be plotted easily (adding the `conf.int=T` includes 95% confidence intervals for each parameter, which we can pass to `ggplot`):

```

tidy(model.1, conf.int = T) %>%
  ggplot(aes(term, estimate, ymin=conf.low, ymax=conf.high)) +
  geom_pointrange() +
  geom_hline(yintercept = 0)

```



Finally, we can use the `augment` function to get information on individual rows in the modelled data: namely the fitted and residual values, plus common diagnostic metrics like Cooks distances:

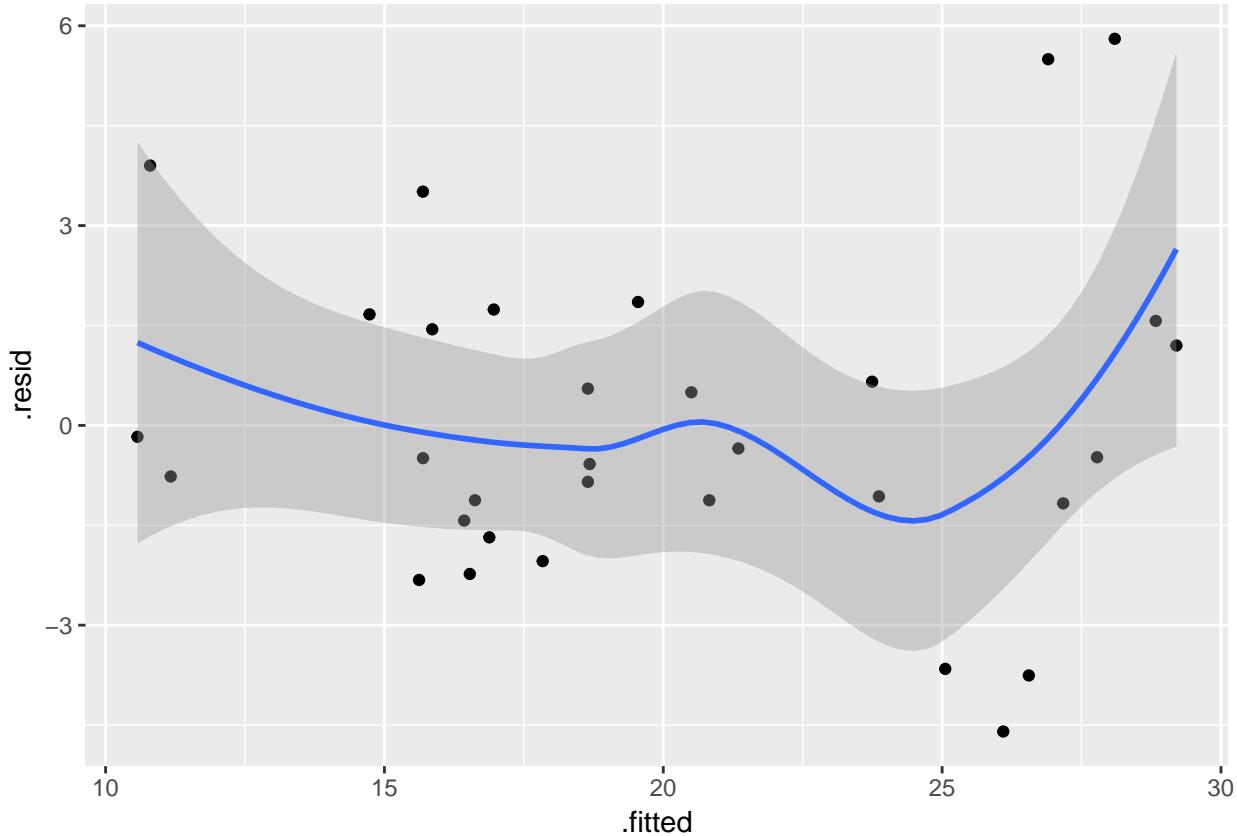
```
augment(model.1) %>%
  head() %>%
  pander(split.tables=Inf)
```

.rownames	mpg	factor.cyl.	wt	disp	.fitted	.se.fit	.resid	.hat	.sigma	.cooksdi	.std.resid
Mazda RX4	21	6	2.62	160	21.35	1.058	-	0.1653	2.652	0.00084230	0.1458
Mazda RX4 Wag	21	6	2.875	160	20.5	1.009	0.4964	0.1501	2.651	0.001512	0.2069
Datsun 710	22.8	4	2.32	108	26.56	0.7854	-	0.09103	2.538	0.04586	-1.513
Hornet 4 Drive	21.4	6	3.215	258	19.55	1.355	1.853	0.2711	2.618	0.05169	0.8336
Hornet Sportabout	18.7	8	3.44	360	16.96	0.9784	1.739	0.1413	2.627	0.0171	0.7209
Valiant	18.1	6	3.46	225	18.68	1.059	-	0.1654	2.65	0.002363	-0.2442
							0.5806				

Again these can be plotted:

```
augment(model.1) %>%
  ggplot(aes(x=.fitted, y=.resid)) +
  geom_point()
```

```
geom_smooth()
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Because `broom` always returns a dataframe with a consistent set of column names we can also combine model results into tables for comparison. In this plot we see what happens to the regression coefficients in model 1 when we add `disp`, `carb` and `drat` in model 2. We plot the coefficients side by side for ease of comparison, and can see that the estimates for `cyl1` and `wt` both shrink slightly with the addition of these variables:

```
# run a new model with more predictors
(model.2 <- lm(mpg ~ factor(cyl) + wt + disp + carb + drat, data=mtcars))

Call:
lm(formula = mpg ~ factor(cyl) + wt + disp + carb + drat, data = mtcars)

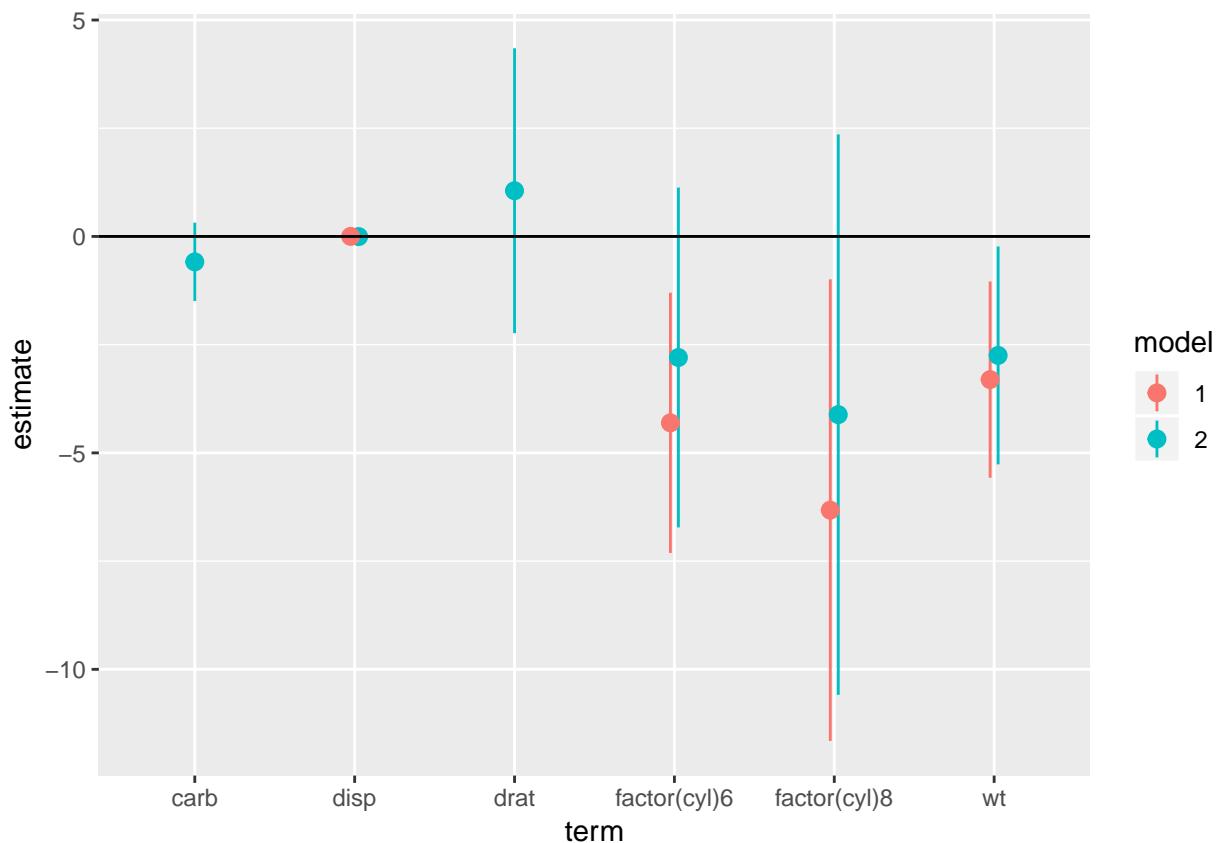
Coefficients:
(Intercept)  factor(cyl)6  factor(cyl)8          wt          disp
29.849209    -2.796142     -4.116561    -2.748229    -0.002826
           carb          drat
-0.587422    1.056532

# make a single dataframe from both models
# addin a new `model` column with mutate to
# identify which coefficient came from which model
combined.results <- bind_rows(
  tidy(model.1, conf.int = T) %>% mutate(model="1"),
  tidy(model.2, conf.int = T) %>% mutate(model="2"))
```

```

combined.results %>%
  # remove the intercept to make plot scale more sane
  filter(term != "(Intercept)") %>%
  ggplot(aes(term, estimate, ymin=conf.low, ymax=conf.high, color=model)) +
  geom_pointrange(position=position_dodge(width=.1)) +
  geom_hline(yintercept = 0)

```



'Processing' results

XXX TODO e.g.:

- Calculate VPC/ICC from an lmer models using `model %>% summary %>% as_data_frame()$varcor`

Printing tables

XXX TODO

- Pander and pandoc
- Dealing with rounding and string formatting issues
- Missing values/unequal length columns
- Point out that arbitrarily complex tables often not worth the candle, longer easier than wider etc.

```

44
45 And if you are using RMarkdown, you can even drop this into your test without copying and pasting. Just type the following and the te
46 is automatically inserted inline in your text:
47 Age (4 vs 6 years) was significantly associated with preference for duplo v.s. lego, `r apastats::describe.chi(lego.table, addN=T)`
48

```

Figure 23: Example of inline call to R functions within the text. This is shown as an image, because it would otherwise be hidden in this output (because the function is evaluated when we knit the document)

APA formatting for free

A neat trick to avoid fat finger errors is to use functions to automatically display results in APA format. Unfortunately, there isn't a single package which works with all types of model, but it's not too hard switch between them.

Chi²

For basic stats the `apa::` package is simple to use. Below we use the `apa::chisq_ap()` function to properly format the results of our chi² test ([see the full chi² example]`#crosstabs`):

```

lego.test <- chisq.test(lego.table)
lego.test

Pearson's Chi-squared test with Yates' continuity correction

data: lego.table
X-squared = 11.864, df = 1, p-value = 0.0005724

```

And we can format in APA like so:

```

apa::apa(lego.test, print_n=T)
[1] "$\\chi^2(1, n = 100) = 11.86, *p* < .001"

```

or using `apastats::` we also get Cramer's V, a measure of effect size:

```

apastats::describe.chi(lego.table, addN=T)
[1] "$\\chi^2(1, _N_ = 100) = 11.86, _p_ < .001, _V_ = .34"

```

17.0.0.1 Inserting results into your text

If you are using RMarkdown, you can drop formatted results into your text without copying and pasting. Just type the following and the chi² test result is automatically inserted inline in your text:

Age (4 vs 6 years) was significantly associated with preference for duplo v.s. lego, $\chi^2(1, N = 100) = 11.86$, $p < .001$, $V = .34$

T-test

```

# run the t test
cars.test <- t.test(wt~am, data=mtcars, var.equal=T)
cars.test

```

Two Sample t-test

```
data: wt by am
t = 5.2576, df = 30, p-value = 1.125e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
0.8304317 1.8853577
sample estimates:
mean in group 0 mean in group 1
3.768895      2.411000
```

And then we can format as APA

```
apa::apa(cars.test)
[1] "*t*(30) = 5.26, *p* < .001, *d* = 1.86"
```

American cars were significantly heavier than foreign cars, mean difference=1358lbs; $t(30) = 5.26, p < .001$, $d = 1.86$

Anova

```
mpg.anova <- car::Anova(lm(mpg~am*cyl, data=mtcars))
Registered S3 methods overwritten by 'car':
  method                  from
  influence.merMod        lme4
  cooks.distance.influence.merMod lme4
  dfbeta.influence.merMod   lme4
  dfbetas.influence.merMod  lme4

# extract and format main effect
apastats::describe.Anova(mpg.anova, term="am")
[1] "_F_(1, 28) = 4.28, _p_ = .048"

# and the interaction
apastats::describe.Anova(mpg.anova, term="am:cyl")
[1] "_F_(1, 28) = 3.41, _p_ = .076"
```

There was no interaction between location of manufacture and number of cylinders, $F(1, 28) = 3.41, p = .076$, but there was a main effect of location of manufacture, $F(1, 28) = 3.41, p = .076$, such that US-made cars had significantly higher fuel consumption than European or Japanese brands (see [Figure X or Table X])

Multilevel models

If you have loaded the `lmerTest` package `apastats` can output either coefficients for single parameters, or F tests:

```
sleep.model <- lmer(Reaction~factor(Days)+(1|Subject), data=lme4::sleepstudy)

#a single coefficient (this is a contrast from the reference category)
apastats::describe.glm(sleep.model, term="factor(Days)1")
Loading required namespace: Hmisc
[1] "_t_ = 0.75, _p_ = .455"
```

```
# or describe the F test for the overall effect of Days
apastats::describe.lmtaov(anova(sleep.model), term='factor(Days)')
[1] "_F_(9, 153.0) = 18.70, _p_ < .001"
```

There were significant differences in reaction times across the 10 days of the study, $F(9, 153.0) = 18.70, p < .001$ such that reaction latencies tended to increase in duration (see [Figure X]).

Simplifying and re-using

Complex programming is beyond the scope of this guide, but if you sometimes find yourself repeating the same piece of code over and over then it might be worth writing a simple function.

The examples shown in this book are mostly very simple, and are typically not repetitive. However in running your own analyses you may find that you start to repeat chunks of code in a number of places (even across many files).

Not only is this sort of copying-and-pasting tedious, it can be hard to read and maintain. For example, if you are repeating the same analysis for multiple variables and discover an error in your calculations you would have to fix this in multiple places in your code. You can also introduce errors when copying and pasting code in this way, and these can be hard to spot.

The sorts of tasks which can end up being repetitive include:

- Running models
- Extracting results from models you run
- Producing tables
- Specifying output settings for graphs

In these cases it might be worth writing your own function to facilitate this repetition, or use some other forms of code re-use (e.g. see the example for `ggplot` graphics below).

Writing helper functions

For example, in the section on logistic regression we wrote a helper function called `logistic()` which was simply a shortcut for `glm()` but with the correct ‘family’ argument pre-specified.

For more information on writing your own R functions I recommend Hadley Wickham’s ‘R for Data Scientists’, chapter 19 and, if necessary, ‘Advanced R’: <http://adv-r.had.co.nz>.

Or you prefer a book, try Grolemund [2014].

Re-using code with `ggplot`

Another common case where we might want to re-use code is when we produce a number of similar plots, and where we might want to re-use many of the same settings.

Sometimes repeating plots is best achieved by creating a single long-form dataframe and facetting the plot. However this may not always be possible or desireable. Lets say we have two plots like these:

```
plot.mpg <- mtcars %>%
  ggplot(aes(factor(cyl), mpg)) +
  geom_boxplot()

plot.wt <- mtcars %>%
```

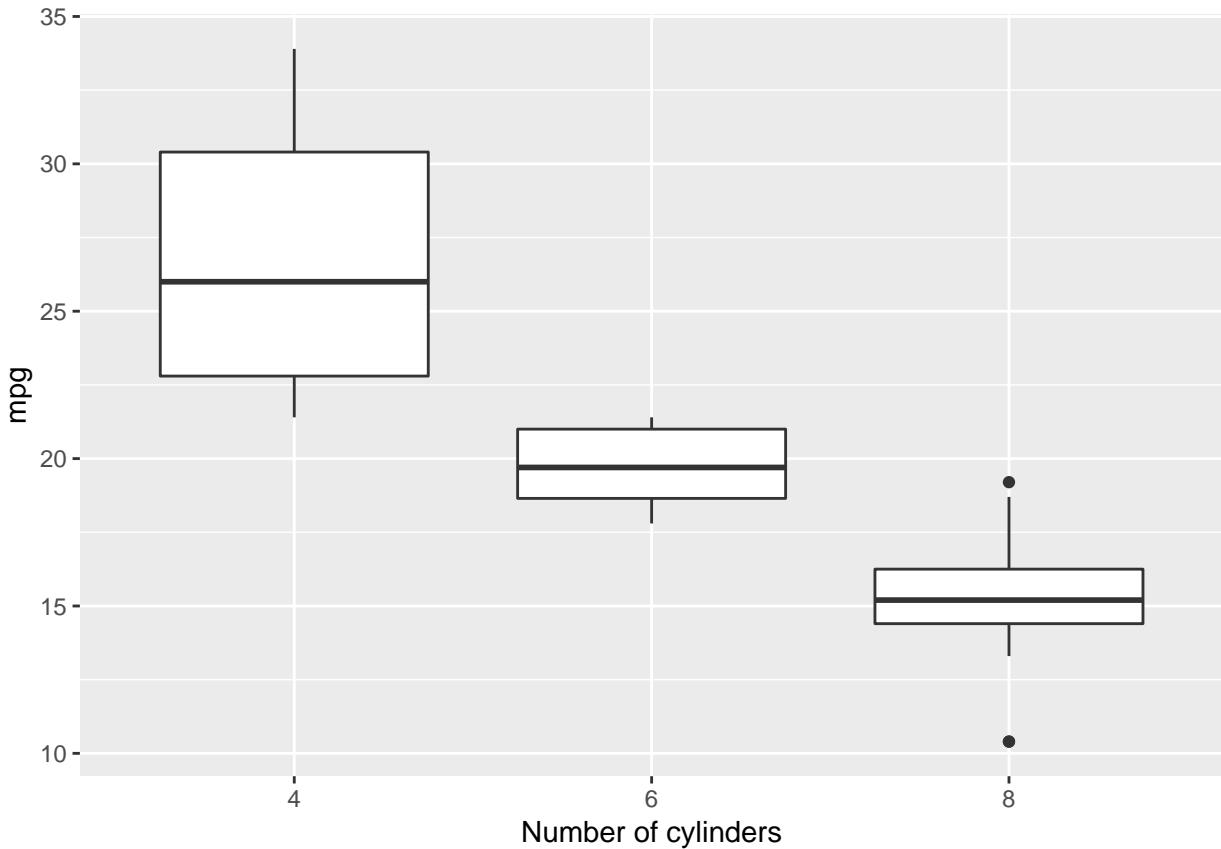
```
ggplot(aes(factor(cyl), wt)) +  
  geom_boxplot()
```

And we want to add consistent labels and other settings to them. We can simply type:

```
myplotsettings <- xlab("Number of cylinders")
```

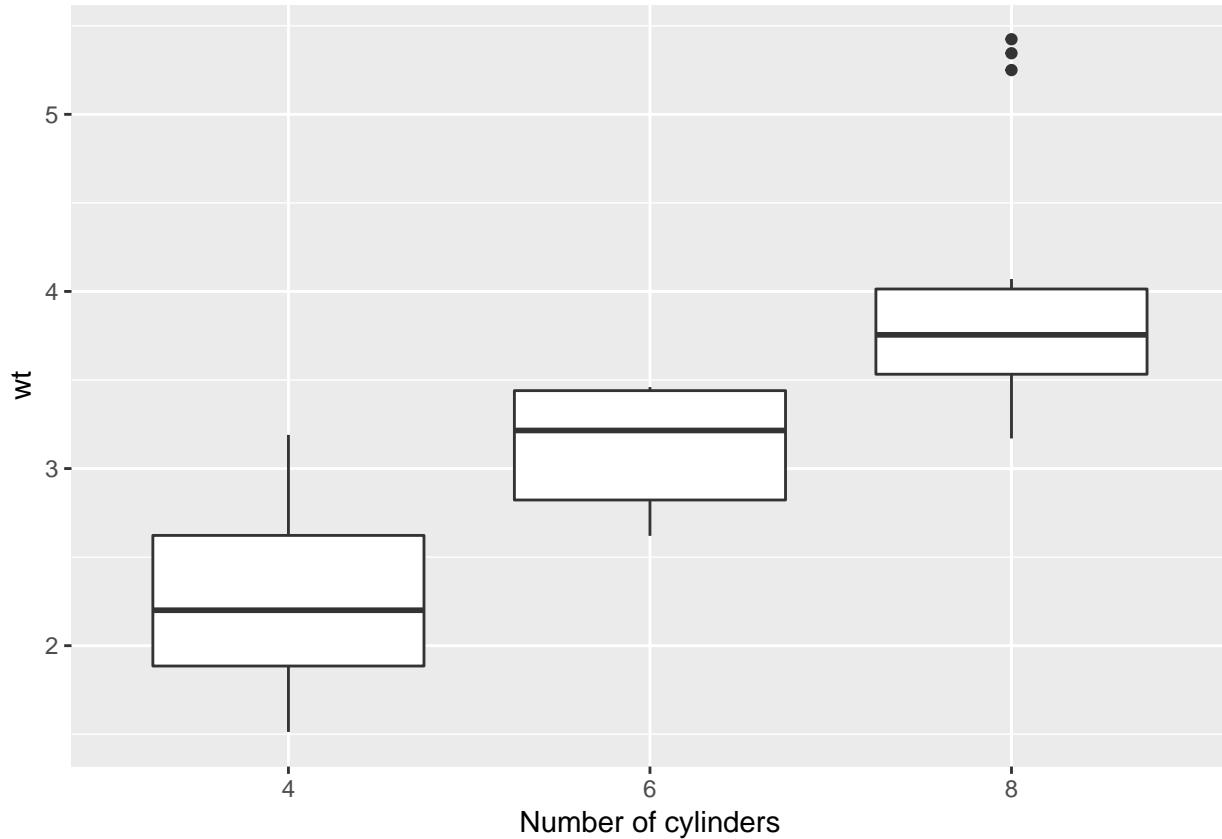
And then we simply add (+) these settings to each plot:

```
plot.mpg + myplotsettings
```



And:

```
plot.wt + myplotsettings
```



This reduces typing and makes easier to produce a consistent set of plots.

“Table 1”

Table 1 in reports of clinical trials and many psychological studies reports characteristics of the sample. Typically, you will want to present information collected at baseline, split by experimental groups, including:

- Means, standard deviations or other descriptive statistics for continuous variables
- Frequencies of particular responses for categorical variables
- Some kind of inferential test for a zero-difference between the groups; this could be a t-test, an F-statistic where there are more than 2 groups, or a chi-squared test for categorical variables.

Producing this table is a pain because it requires collating multiple statistics, calculated from different functions. Many researchers resort to performing all the analyses required for each part of the table, and then copying-and-pasting results into Word.

It can be automated though! This example combines and extends many of the techniques we have learned using the split-apply-combine method.

To begin, let's simulate some data from a fairly standard 2-arm clinical trial or psychological experiment:

Check our data:

```
boring.study %>% glimpse
Observations: 280
Variables: 8
$ person    <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 1...
$ time      <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
```

```

$ condition <fct> Control, Control, Control, Control, Control...
$ yob       <dbl> 1969, 1978, 1969, 1989, 1975, 1975, 1989, 1983, 1978...
$ WM        <dbl> 103, 103, 94, 96, 109, 98, 102, 93, 78, 102, 98, 91, ...
$ education <chr> "Graduate", "Secondary", "Postgraduate", "Primary", ...
$ ethnicity <chr> "Asian / Asian British", "White British", "White Bri...
$ Attitude   <dbl> 12, 5, 8, 6, 4, 9, 14, 12, 10, 9, 12, 14, 8, 11, 8, ...

```

Start by making a long-form table for the categorical variables:

```

boring.study.categorical.melted <-  

  table1.categorical.Ns <- boring.study %>%  

  select(condition, education, ethnicity) %>%  

  melt(id.var='condition')

```

Then calculate the N's for each response/variable in each group:

```

(table1.categorical.Ns <-  

  boring.study.categorical.melted %>%  

  group_by(condition, variable, value) %>%  

  summarise(N=n()) %>%  

  dcast(variable+value~condition, value.var="N"))  

  variable  

  value Control  

1 education             Graduate      39  

2 education            Postgraduate  25  

3 education            Primary      21  

4 education            Secondary     27  

5 education           <NA>         28  

6 ethnicity           Asian / Asian British 32  

7 ethnicity Black / African / Caribbean / Black British 42  

8 ethnicity Mixed / multiple ethnic groups 33  

9 ethnicity White British    33  

  Intervention  

1          33  

2          25  

3          24  

4          24  

5          34  

6          37  

7          35  

8          30  

9          38

```

Then make a second table containing Chi2 test statistics for each variable:

```

(table1.categorical.tests <-  

  boring.study.categorical.melted %>%  

  group_by(variable) %>%  

  do(., chisq.test(.value, .condition) %>% tidy) %>%  

  # this purely to facilitate matching rows up below  

  mutate(firstrowforvar=T)  

# A tibble: 2 x 6  

# Groups:   variable [2]  

  variable statistic p.value parameter method firstrowforvar  

  <fct>      <dbl>    <dbl>    <int> <chr>      <lgl>  

1 education    0.712    0.870      3 Pearson's Chi-squared TRUE  

2 ethnicity   1.49     0.684      3 Pearson's Chi-squared TRUE

```

Combine these together:

```
(table1.categorical.both <- table1.categorical.Ns %>%
  group_by(variable) %>%
  # we join on firstrowforvar to make sure we don't duplicate the tests
  mutate(firstrowforvar=row_number()==1) %>%
  left_join(., table1.categorical.tests, by=c("variable", "firstrowforvar")) %>%
  # this is gross, but we don't want to repeat the variable names in our table
  ungroup() %>%
  mutate(variable = ifelse(firstrowforvar==T, as.character(variable), NA)) %>%
  select(variable, value, Control, Intervention, statistic, parameter, p.value))
# A tibble: 9 x 7
  variable   value   Control Intervention statistic parameter p.value
  <chr>     <chr>     <int>      <int>    <dbl>    <int>    <dbl>
1 education Graduate      39        33     0.712      3    0.870
2 <NA>       Postgraduate  25        25      NA      NA    NA
3 <NA>       Primary       21        24      NA      NA    NA
4 <NA>       Secondary     27        24      NA      NA    NA
5 <NA>       <NA>          28        34      NA      NA    NA
6 ethnicity Asian / Asian~ 32        37     1.49      3    0.684
7 <NA>       Black / Afric~ 42        35      NA      NA    NA
8 <NA>       Mixed / multi~ 33        30      NA      NA    NA
9 <NA>       White British 33        38      NA      NA    NA
```

Now we deal with the continuous variables. First we make a ‘long’ version of the continuous data

```
continuous_variables <- c("yob", "WM")
boring.continuous.melted <-
  boring.study %>%
  select(condition, continuous_variables) %>%
  melt() %>%
  group_by(variable)
Using condition as id variables
```

```
boring.continuous.melted %>% head
# A tibble: 6 x 3
# Groups:   variable [1]
  condition variable value
  <fct>     <fct>   <dbl>
1 Control    yob     1969
2 Control    yob     1978
3 Control    yob     1969
4 Control    yob     1989
5 Control    yob     1975
6 Control    yob     1975
```

Then calculate separate tables of t-tests and means/SD’s:

```
(table.continuous_variables.tests <-
  boring.continuous.melted %>%
  # note that we pass the result of t-test to tidy, which returns a dataframe
  do(., t.test(.value~.condition) %>% tidy) %>%
  select(variable, statistic, parameter, p.value))
# A tibble: 2 x 4
# Groups:   variable [2]
  variable statistic parameter p.value
  <chr>     <dbl>    <dbl>    <dbl>
```

```

<fct>      <dbl>      <dbl>      <dbl>
1 yob        0.941       278.   0.348
2 WM         -0.695      278.   0.488

(table.continuous_variables.descriptives <-
  boring.continuous.melted %>%
  group_by(variable, condition) %>%
  # this is not needed here because we have no missing values, but if there
  # were missing value in this dataset then mean/sd functions would fail below,
  # so best to remove rows without a response:
  filter(!is.na(value)) %>%
  # note, we might also want the median/IQR
  summarise(Mean=mean(value), SD=sd(value)) %>%
  group_by(variable, condition) %>%
  # we format the mean and SD into a single column using sprintf.
  # we don't have to do this, but it makes reshaping simpler and we probably want
  # to round the numbers at some point, and so may as well do this now.
  transmute(MSD = sprintf("%.2f (%.2f)", Mean, SD)) %>%
  dcast(variable~condition))

Using MSD as value column: use value.var to override.

variable          Control     Intervention
1    yob  1979.51 (5.21) 1978.92 (5.21)
2      WM  100.08 (9.18)  100.86 (9.56)

```

And combine them:

```

(table.continuous_variables.both <-
  left_join(table.continuous_variables.descriptives,
            table.continuous_variables.tests))
Joining, by = "variable"
variable          Control     Intervention   statistic parameter    p.value
1    yob  1979.51 (5.21) 1978.92 (5.21)  0.9407657 277.9999 0.3476417
2      WM  100.08 (9.18)  100.86 (9.56) -0.6948592 277.5507 0.4877249

```

Finally put the whole thing together:

```

(table1 <- table1.categorical.both %>%
  # make these variables into character format to be consistent with
  # the Mean (SD) column for continuus variables
  mutate_each(funs(format), Control, Intervention) %>%
  # note the '.' as the first argument, which is the input from the pipe
  bind_rows(.,
            table.continuous_variables.both) %>%
  # prettify a few things
  rename(df = parameter,
         p=p.value,
         `Control N/Mean (SD)` = Control,
         Variable=variable,
         Response=value,
         `t/ 2` = statistic))
Warning: funs() is soft deprecated as of dplyr 0.8.0
please use list() instead

# Before:
funs(name = f(.))

```

```

# After:
list(name = ~ f(.))
This warning is displayed once per session.
Warning in bind_rows_(x, .id): binding character and factor vector,
coercing into character vector
# A tibble: 11 x 7
  Variable Response `Control N/Mean` Intervention `t/ 2`    df      p
  <chr>   <chr>        <chr>           <chr>       <dbl> <dbl>    <dbl>
1 education Graduate     39             33          0.712    3    0.870
2 <NA>     Postgraduate  25             25          NA       NA    NA
3 <NA>     Primary       21             24          NA       NA    NA
4 <NA>     Secondary     27             24          NA       NA    NA
5 <NA>     <NA>          28             34          NA       NA    NA
6 ethnicity Asian / Asi~ 32             37          1.49     3    0.684
7 <NA>     Black / Afr~  42             35          NA       NA    NA
8 <NA>     Mixed / mul~  33             30          NA       NA    NA
9 <NA>     White Briti~ 33             38          NA       NA    NA
10 yob      <NA>          1979.51 (5.21) 1978.92 (5.~  0.941   278.   0.348
11 WM      <NA>          100.08 (9.18)   100.86 (9.5~ -0.695   278.   0.488

```

And we can print to markdown format for outputting. This is best done in a separate chunk to avoid warnings/messages appearing in the final document.

```

table1 %>%
  # split.tables argument needed to avoid the table wrapping
  pander(split.tables=Inf,
         missing="-",
         justify=c("left", "left", rep("center", 5)),
         caption='Table presenting baseline differences between conditions. Categorical variables tested with Pearson 2, continuous variables with two-sample t-test')

```

Table 53: Table presenting baseline differences between conditions. Categorical variables tested with Pearson 2, continuous variables with two-sample t-test.

Variable	Response	Control N/Mean		t / 2	df	p
		(SD)	Intervention			
education	Graduate	39	33	0.7119	3	0.8704
-	Postgraduate	25	25	-	-	-
-	Primary	21	24	-	-	-
-	Secondary	27	24	-	-	-
-	-	28	34	-	-	-
ethnicity	Asian / Asian British	32	37	1.494	3	0.6837
-	Black / African / Caribbean / Black British	42	35	-	-	-
-	Mixed / multiple ethnic groups	33	30	-	-	-
-	White British	33	38	-	-	-
yob	-	1979.51 (5.21)	1978.92 (5.21)	0.9408	278	0.3476
WM	-	100.08 (9.18)	100.86 (9.56)	-0.6949	277.6	0.4877

Some exercises to work on/extensions to this code you might need:

- Add a new continuous variable to the simulated dataset and include it in the final table
- Create a third experimental group and amend the code to i) include 3 columns for the N/Mean and ii) report the F-test from a one-way Anova as the test statistic.
- Add the within-group percentage for each response to a categorical variable.

18 Dealing with quirks of R

Rownames are evil

Historically ‘row names’ were used on R to label individual rows in a dataframe. It turned out that this is generally a bad idea, because sorting and some summary functions would get very confused and mix up row names and the data itself.

It’s now considered best practice to avoid row names for this reason. Consequently, the functions in the `dplyr` library remove row names when processing dataframes. For example here we see the row names in the `mtcars` data:

```
mtcars %>%
  head(3)
#>
#> # A tibble: 3 x 11
#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 21.0     6    160   110  3.90  2.620 16.46     0    1     4     4
#> 2 21.0     6    160   110  3.90  2.875 17.02     0    1     4     4
#> 3 22.8     4    108    93  3.85  2.320 18.61     1    1     4     1
```

But here we don’t because `arrange` has stripped them:

```
mtcars %>%
  arrange(mpg) %>%
  head(3)
#>
#> # A tibble: 3 x 11
#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 10.4     8    472   205  2.93  5.250 17.98     0    0     3     4
#> 2 10.4     8    460   215  3.00  5.424 17.82     0    0     3     4
#> 3 13.3     8    350   245  3.73  3.840 15.41     0    0     3     4
```

Converting the results of `psych::describe()` also returns rownames, which can get lots if we sort the data.

We see row names here:

```
psych::describe(mtcars) %>%
  as_data_frame() %>%
  head(3)
#> Warning: `as_data_frame()` is deprecated, use `as_tibble()` (but mind the new semantics).
#> This warning is displayed once per session.
#> # A tibble: 3 x 13
#>   vars      n    mean     sd median trimmed    mad    min    max range skew
#>   <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1     1     32  20.1    6.03   19.2   19.7    5.41  10.4   33.9  23.5  0.611
#> 2     2     32   6.19    1.79     6    6.23    2.97    4     8     4   -0.175
#> 3     3     32  231.   124.   196.   223.   140.   71.1  472   401.  0.382
#> # ... with 2 more variables: kurtosis <dbl>, se <dbl>
```

But not here (just numbers in their place):

```
psych::describe(mtcars) %>%
  as_data_frame() %>%
  arrange(mean) %>%
```

```

head(3)
# A tibble: 3 x 13
  vars     n   mean    sd median trimmed   mad   min   max range skew
  <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1     9    32  0.406  0.499     0    0.385  0      0     1     1  0.364
2     8    32  0.438  0.504     0    0.423  0      0     1     1  0.240
3    11    32  2.81   1.62      2    2.65   1.48    1     8     7  1.05
# ... with 2 more variables: kurtosis <dbl>, se <dbl>

```

18.0.0.0.1 Preserving row names

If you want to preserve row names, it's best to convert the names to an extra column in the data. So, the example below does what we probably want:

```

# the var='car.name' argument is optional, but can be useful
mtcars %>%
  rownames_to_column(var="car.name") %>%
  arrange(mpg) %>%
  head(3)

  car.name  mpg cyl disp hp drat    wt  qsec vs am gear carb
1 Cadillac Fleetwood 10.4   8 472 205 2.93 5.250 17.98  0  0    3    4
2 Lincoln Continental 10.4   8 460 215 3.00 5.424 17.82  0  0    3    4
3 Camaro Z28 13.3   8 350 245 3.73 3.840 15.41  0  0    3    4

lm(mpg~wt, data=mtcars) %>%
  broom::tidy() %>%
  pander

```

term	estimate	std.error	statistic	p.value
(Intercept)	37.29	1.878	19.86	8.242e-19
wt	-5.344	0.5591	-9.559	1.294e-10

Working with character strings

18.0.1 Searching and replacing

If you want to search inside a string there are lots of useful functions in the `stringr::` library. These replicate some functionality in base R, but like other packages in the 'tidyverse' they tend to be more consistent and easier to use. For example:

```

cheese <- c("Stilton", "Brie", "Cheddar")
stringr::str_detect(cheese, "Br")
[1] FALSE TRUE FALSE
stringr::str_locate(cheese, "i")
  start end
[1,]    3   3
[2,]    3   3
[3,]   NA  NA
stringr::str_replace(cheese, "Stil", "Mil")
[1] "Milton"  "Brie"    "Cheddar"

```

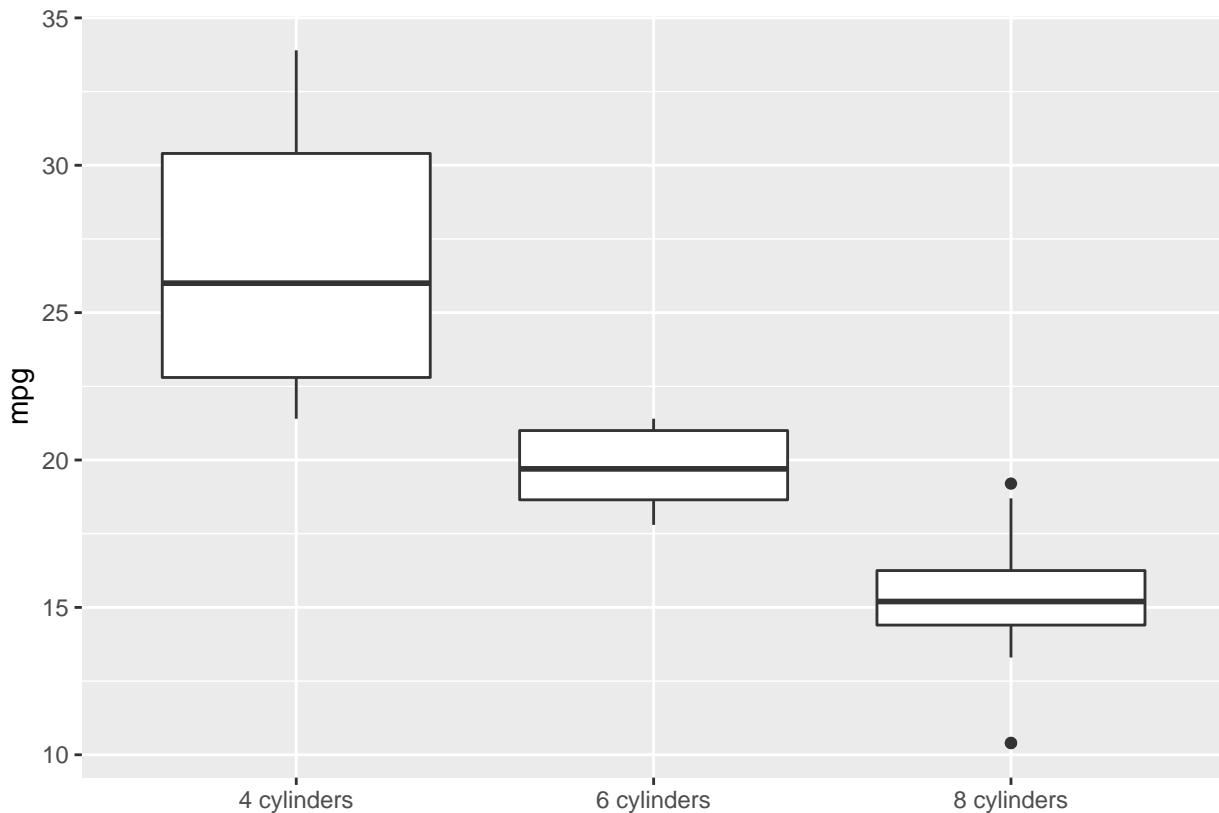
Using paste to make labels

Paste can combine character strings with other types of variable to produce a new vector:

```
paste(mtcars$cyl, "cylinders")[1:10]
[1] "6 cylinders" "6 cylinders" "4 cylinders" "6 cylinders" "8 cylinders"
[6] "6 cylinders" "8 cylinders" "4 cylinders" "4 cylinders" "6 cylinders"
```

Which can be a useful way to label graphs:

```
mtcars %>%
  ggplot(aes(paste(mtcars$cyl, "cylinders"), mpg)) +
  geom_boxplot() + xlab("")
```



Fixing up variable after melting

In this example `melt()` creates a new column called `variable`.

```
sleep.wide %>%
  melt(id.var = "Subject") %>%
  arrange(Subject, variable) %>%
  head
  Subject variable     value
1       1   Day.0 249.5600
2       1   Day.1 258.7047
3       1   Day.2 250.8006
4       1   Day.3 321.4398
```

```

5      1   Day.4 356.8519
6      1   Day.5 414.6901

```

However the contents of `variable` are now a character string (i.e. a list of letters and numbers) rather than numeric values (see column types) but in this instance we know that the values `Day.1`, `Day.2...` are not really separate categories but actually form a linear sequence, from 1 to 9.

We can use the `extract` or `separate` functions to split up `variable` and create a numeric column for `Day`:

```

sleep.long %>%
  separate(variable, c("variable", "Day")) %>%
  mutate(Day=as.numeric(Day)) %>%
  arrange(Subject) %>%
  head %>% pander

```

Subject	variable	Day	value
1	Day	0	249.6
1	Day	1	258.7
1	Day	2	250.8
1	Day	3	321.4
1	Day	4	356.9
1	Day	5	414.7

See the user guide for `separate` and `extract` for more details.

If you are familiar with regular expressions you will be happy to know that you can use regex to separate variables using `extract` and `separate`. See this guide for more details on how `separate` and `extract` work

Colours

Picking colours for plots

See https://www.perceptualedge.com/articles/b-eye/choosing_colors.pdf for an interesting discussion on picking colours for data visualisation.

Also check the ggplots docs for colour brewer and the Colour Brewer website.

Named colours in R

```

print.col <- Vectorize(function(col){
  rgb <- grDevices::col2rgb(col)
  sprintf("<span style='padding:1em; background-color: rgb(%s, %s, %s);'>&nbsp;</span> %s \n\n", rgb[1])
})

pandoc.p(print.col(colours()))

```

```

white
aliceblue
antiquewhite
antiquewhite1

```

antiquewhite2

antiquewhite3

antiquewhite4

aquamarine

aquamarine1

aquamarine2

aquamarine3

aquamarine4

azure

azure1

azure2

azure3

azure4

beige

bisque

bisque1

bisque2

bisque3

bisque4

black

blanchedalmond

blue

blue1

blue2

blue3

blue4

blueviolet

brown

brown1

brown2

brown3

brown4

burlywood

burlywood1

burlywood2

burlywood3

burlywood4
cadetblue
cadetblue1
cadetblue2
cadetblue3
cadetblue4
chartreuse
chartreuse1
chartreuse2
chartreuse3
chartreuse4
chocolate
chocolate1
chocolate2
chocolate3
chocolate4
coral
coral1
coral2
coral3
coral4
cornflowerblue
cornsilk
cornsilk1
cornsilk2
cornsilk3
cornsilk4
cyan
cyan1
cyan2
cyan3
cyan4
darkblue
darkcyan
darkgoldenrod
darkgoldenrod1

darkgoldenrod2
darkgoldenrod3
darkgoldenrod4
darkgray
darkgreen
darkgrey
darkkhaki
darkmagenta
darkolivegreen
darkolivegreen1
darkolivegreen2
darkolivegreen3
darkolivegreen4
darkorange
darkorange1
darkorange2
darkorange3
darkorange4
darkorchid
darkorchid1
darkorchid2
darkorchid3
darkorchid4
darkred
darksalmon
darkseagreen
darkseagreen1
darkseagreen2
darkseagreen3
darkseagreen4
darkslateblue
darkslategray
darkslategray1
darkslategray2
darkslategray3
darkslategray4

darkslategrey
darkturquoise
darkviolet
deeppink
deeppink1
deeppink2
deeppink3
deeppink4
deepskyblue
deepskyblue1
deepskyblue2
deepskyblue3
deepskyblue4
dimgray
imgrey
dodgerblue
dodgerblue1
dodgerblue2
dodgerblue3
dodgerblue4
firebrick
firebrick1
firebrick2
firebrick3
firebrick4
floralwhite
forestgreen
gainsboro
ghostwhite
gold
gold1
gold2
gold3
gold4
goldenrod
goldenrod1

goldenrod2

goldenrod3

goldenrod4

gray

gray0

gray1

gray2

gray3

gray4

gray5

gray6

gray7

gray8

gray9

gray10

gray11

gray12

gray13

gray14

gray15

gray16

gray17

gray18

gray19

gray20

gray21

gray22

gray23

gray24

gray25

gray26

gray27

gray28

gray29

gray30

gray31

gray32

gray33

gray34

gray35

gray36

gray37

gray38

gray39

gray40

gray41

gray42

gray43

gray44

gray45

gray46

gray47

gray48

gray49

gray50

gray51

gray52

gray53

gray54

gray55

gray56

gray57

gray58

gray59

gray60

gray61

gray62

gray63

gray64

gray65

gray66

gray67

gray68

gray69

gray70

gray71

gray72

gray73

gray74

gray75

gray76

gray77

gray78

gray79

gray80

gray81

gray82

gray83

gray84

gray85

gray86

gray87

gray88

gray89

gray90

gray91

gray92

gray93

gray94

gray95

gray96

gray97

gray98

gray99

gray100

green

green1

green2

green3
green4
greenyellow
grey
grey0
grey1
grey2
grey3
grey4
grey5
grey6
grey7
grey8
grey9
grey10
grey11
grey12
grey13
grey14
grey15
grey16
grey17
grey18
grey19
grey20
grey21
grey22
grey23
grey24
grey25
grey26
grey27
grey28
grey29
grey30
grey31

grey32

grey33

grey34

grey35

grey36

grey37

grey38

grey39

grey40

grey41

grey42

grey43

grey44

grey45

grey46

grey47

grey48

grey49

grey50

grey51

grey52

grey53

grey54

grey55

grey56

grey57

grey58

grey59

grey60

grey61

grey62

grey63

grey64

grey65

grey66

grey67

grey68
grey69
grey70
grey71
grey72
grey73
grey74
grey75
grey76
grey77
grey78
grey79
grey80
grey81
grey82
grey83
grey84
grey85
grey86
grey87
grey88
grey89
grey90
grey91
grey92
grey93
grey94
grey95
grey96
grey97
grey98
grey99
grey100
honeydew
honeydew1
honeydew2

honeydew3
honeydew4
hotpink
hotpink1
hotpink2
hotpink3
hotpink4
indianred
indianred1
indianred2
indianred3
indianred4
ivory
ivory1
ivory2
ivory3
ivory4
khaki
khaki1
khaki2
khaki3
khaki4
lavender
lavenderblush
lavenderblush1
lavenderblush2
lavenderblush3
lavenderblush4
lawngreen
lemonchiffon
lemonchiffon1
lemonchiffon2
lemonchiffon3
lemonchiffon4
lightblue
lightblue1

lightblue2
lightblue3
lightblue4
lightcoral
lightcyan
lightcyan1
lightcyan2
lightcyan3
lightcyan4
lightgoldenrod
lightgoldenrod1
lightgoldenrod2
lightgoldenrod3
lightgoldenrod4
lightgoldenrodyellow
lightgray
lightgreen
lightgrey
lightpink
lightpink1
lightpink2
lightpink3
lightpink4
lightsalmon
lightsalmon1
lightsalmon2
lightsalmon3
lightsalmon4
lightseagreen
lightskyblue
lightskyblue1
lightskyblue2
lightskyblue3
lightskyblue4
lightslateblue
lightslategray

lightslategrey
lightsteelblue
lightsteelblue1
lightsteelblue2
lightsteelblue3
lightsteelblue4
lightyellow
lightyellow1
lightyellow2
lightyellow3
lightyellow4
limegreen
linen
magenta
magenta1
magenta2
magenta3
magenta4
maroon
maroon1
maroon2
maroon3
maroon4
mediumaquamarine
mediumblue
mediumorchid
mediumorchid1
mediumorchid2
mediumorchid3
mediumorchid4
mediumpurple
mediumpurple1
mediumpurple2
mediumpurple3
mediumpurple4
mediumseagreen

mediumslateblue
mediumspringgreen
mediumturquoise
mediumvioletred
midnightblue
mintcream
mistyrose
mistyrose1
mistyrose2
mistyrose3
mistyrose4
moccasin
navajowhite
navajowhite1
navajowhite2
navajowhite3
navajowhite4
navy
navyblue
oldlace
olivedrab
olivedrab1
olivedrab2
olivedrab3
olivedrab4
orange
orange1
orange2
orange3
orange4
orangered
orangered1
orangered2
orangered3
orangered4
orchid

orchid1
orchid2
orchid3
orchid4
palegoldenrod
palegreen
palegreen1
palegreen2
palegreen3
palegreen4
paleturquoise
paleturquoise1
paleturquoise2
paleturquoise3
paleturquoise4
palevioletred
palevioletred1
palevioletred2
palevioletred3
palevioletred4
papayawhip
peachpuff
peachpuff1
peachpuff2
peachpuff3
peachpuff4
peru
pink
pink1
pink2
pink3
pink4
plum
plum1
plum2
plum3

plum4
powderblue
purple
purple1
purple2
purple3
purple4
red
red1
red2
red3
red4
rosybrown
rosybrown1
rosybrown2
rosybrown3
rosybrown4
royalblue
royalblue1
royalblue2
royalblue3
royalblue4
saddlebrown
salmon
salmon1
salmon2
salmon3
salmon4
sandybrown
seagreen
seagreen1
seagreen2
seagreen3
seagreen4
seashell
seashell1

seashell2
seashell3
seashell4
sienna
sienna1
sienna2
sienna3
sienna4
skyblue
skyblue1
skyblue2
skyblue3
skyblue4
slateblue
slateblue1
slateblue2
slateblue3
slateblue4
slategray
slategray1
slategray2
slategray3
slategray4
slategrey
snow
snow1
snow2
snow3
snow4
springgreen
springgreen1
springgreen2
springgreen3
springgreen4
steelblue
steelblue1

steelblue2
steelblue3
steelblue4
tan
tan1
tan2
tan3
tan4
thistle
thistle1
thistle2
thistle3
thistle4
tomato
tomato1
tomato2
tomato3
tomato4
turquoise
turquoise1
turquoise2
turquoise3
turquoise4
violet
violetred
violetred1
violetred2
violetred3
violetred4
wheat
wheat1
wheat2
wheat3
wheat4
whitesmoke
yellow



Figure 24: On windows

yellow1
yellow2
yellow3
yellow4
yellowgreen

ColourBrewer with ggplot

See: http://ggplot2.tidyverse.org/reference/scale_brewer.html

19 Getting help

If you don't know or can't remember what a function does, R provides help files which explain how they work. To access a help file for a function, just type `?command` in the console, or run `?command` command within an R block. For example, running `?mean` would bring up the documentation for the `mean` function.

You can also type `CRTL-Shift-H` while your cursor is over any R function in the RStudio interface.

It's fair to say R documentation isn't always written for beginners. However the 'examples' sections are usually quite informative: you can normally see this by scrolling right to the end of the help file.

Finding the backtick on your keyboard

The backtick `` symbol is unfamiliar to some readers. Here's where it is:

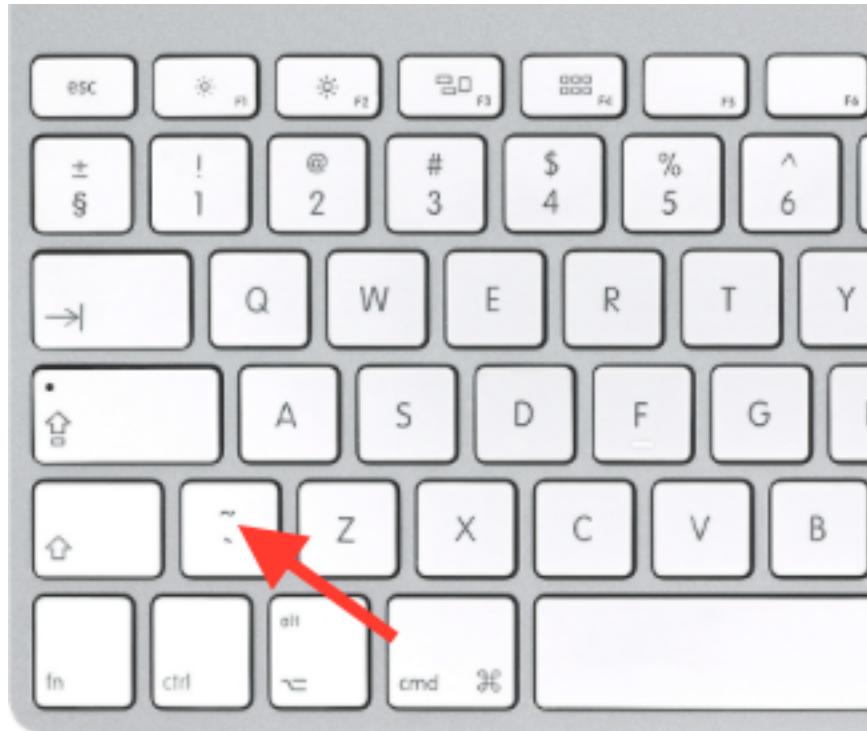


Figure 25: On a Mac

Part V

Explanations

20 Confidence and Intervals

Some quick definitions to begin. Let's say we have made an estimate from a model. To keep things simple, it could just be the sample mean.

A *Confidence interval* is the range within which we would expect the ‘true’ value to fall, 95% of the time, if we replicated the study.

A *Prediction interval* is the range within which we expect 95% of new observations to fall. If we’re considering the prediction interval for a specific point prediction (i.e. where we set predictors to specific values), then this interval would be for new observations *with the same predictor values*.

A Bayesian *Credible interval* is the range of values within which we are 95% sure the true value lies, based on our prior knowledge and the data we have collected.

The problem with confidence intervals

Confidence intervals are helpful when we want to think about how *precise our estimate* is. For example, in an RCT we will want to estimate the difference between treatment groups, and it’s conceivable we would want to know, for example, the range within which the true effect would fall 95% of the time if we replicated our study many times (although in reality, this isn’t a question many people would actually ask).

If we run a study with small N, intuitively we know that we have less information about the difference between our RCT treatments, and so we'd like the CI to expand accordingly.

So — all things being equal — the confidence interval reduces as we collect more data.

The problem with confidence intervals comes about because many researchers and clinicians read them incorrectly. Typically, they either:

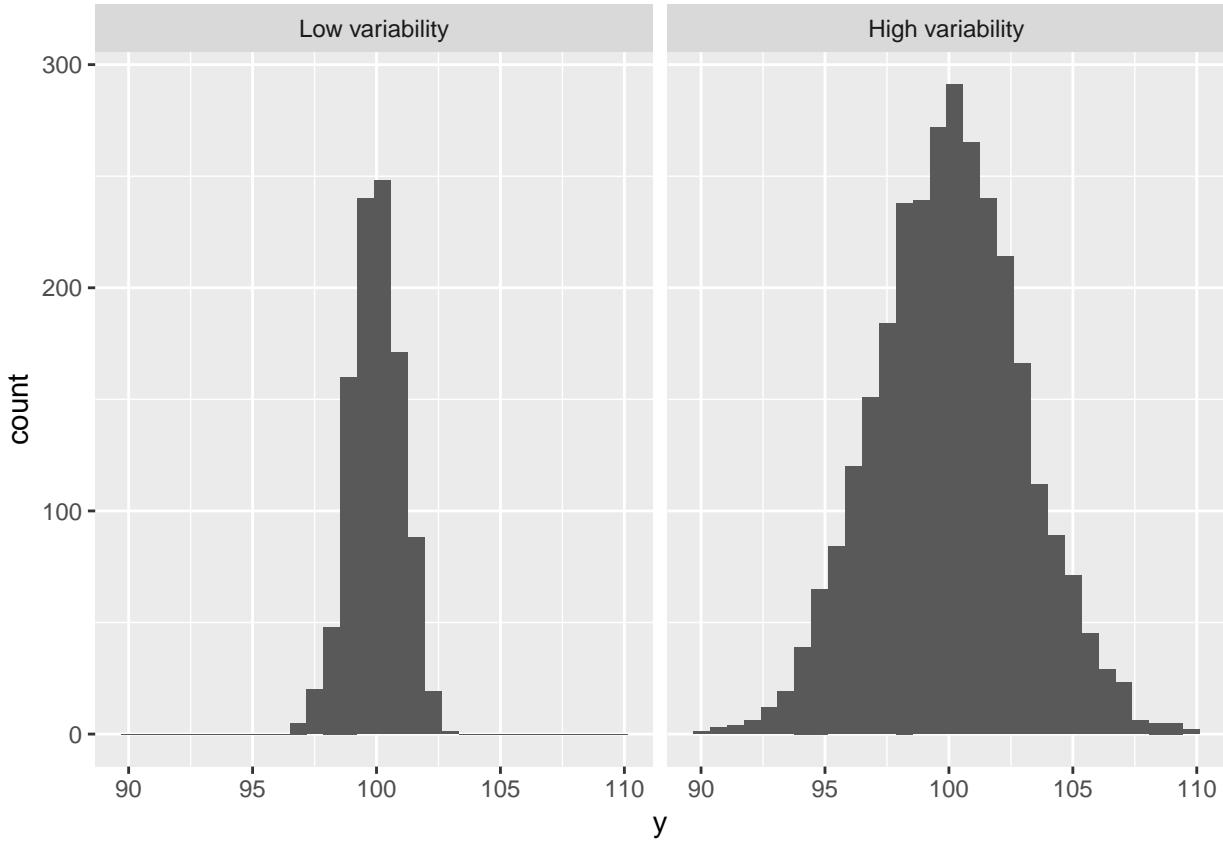
- Forget that the CI represents only the *precision of the estimate*. The CI *doesn't* reflect how good our predictions for new observations will be.
- Misinterpret the CI as the range in which we are 95% sure the true value lies.

Forgetting that the CI depends on sample size.

By forgetting that the CI contracts as the sample size increases, researchers can become overconfident about their ability to predict new observations. Imagine that we sample data from two populations with the same mean, but different variability:

```
set.seed(1234)
df <- expand.grid(v=c(1,3,3,3), i=1:1000) %>%
  as_data_frame %>%
  mutate(y = rnorm(length(.i), 100, v)) %>%
  mutate(samp = factor(v, labels=c("Low variability", "High variability")))
Warning: `as_data_frame()` is deprecated, use `as_tibble()` (but mind the new semantics).
This warning is displayed once per session.

df %>%
  ggplot(aes(y)) +
  geom_histogram() +
  facet_grid(~samp) +
  scale_color_discrete("")
```



- If we sample 100 individuals from each population the confidence interval around the sample mean would be wider in the high variability group.

If we increase our sample size we would become more confident about the location of the mean, and this confidence interval would shrink.

But imagine taking a single *new sample* from either population. These samples would be new grey squares, which we place on the histograms above. It does not matter how much extra data we have collected in group B or how sure what the mean of the group is: *We would always be less certain making predictions for new observations in the high variability group.*

The important insight here is that *if our data are noisy and highly variable we can never make firm predictions for new individuals, even if we collect so much data that we are very certain about the location of the mean.*

21 Multiple comparisons

People get confused about multiple comparisons and worry about ‘doing things right’.

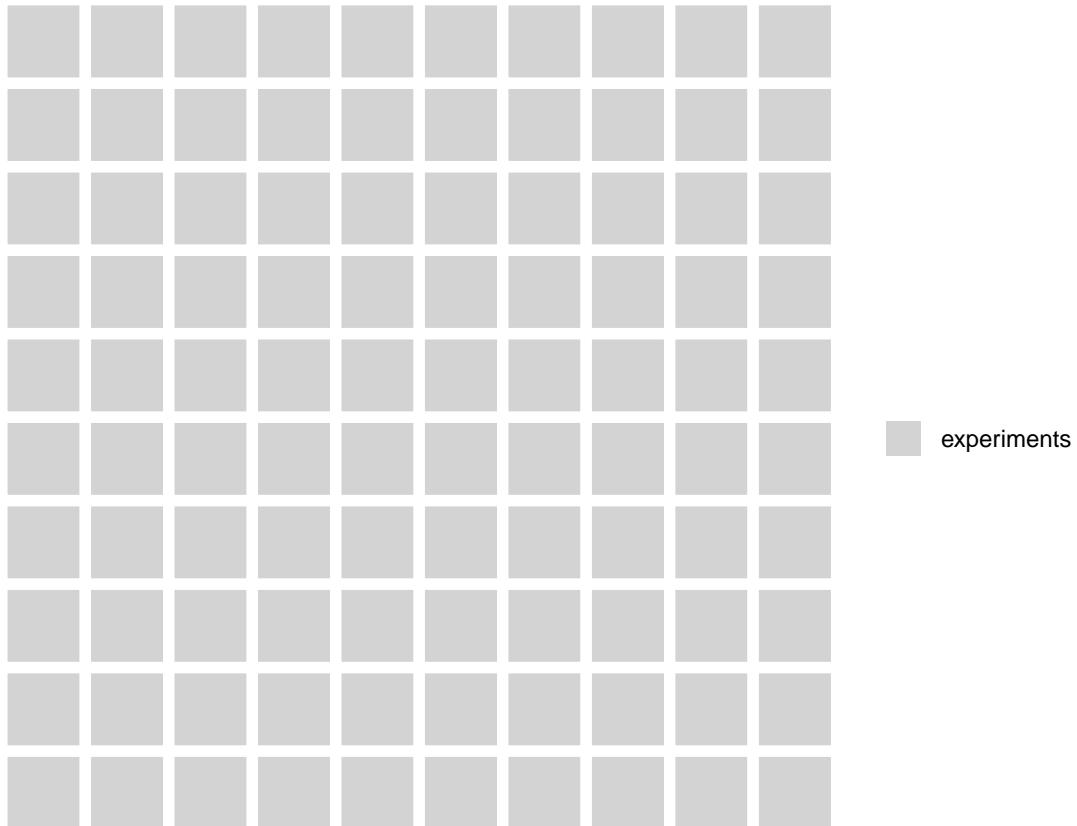
There are many different tests and procedures, and thousands of pages of tutorials and guides each of which recommends a slightly different approach. Textbooks typically describe the tests themselves in detail, and list the assumptions they make, but it sometimes feels like nobody will give a straight answer.

The inconsistency arises because different researchers have different priorities. It might help to re-address what a p value is, and what it is *for*.

p values and ‘false discoveries’

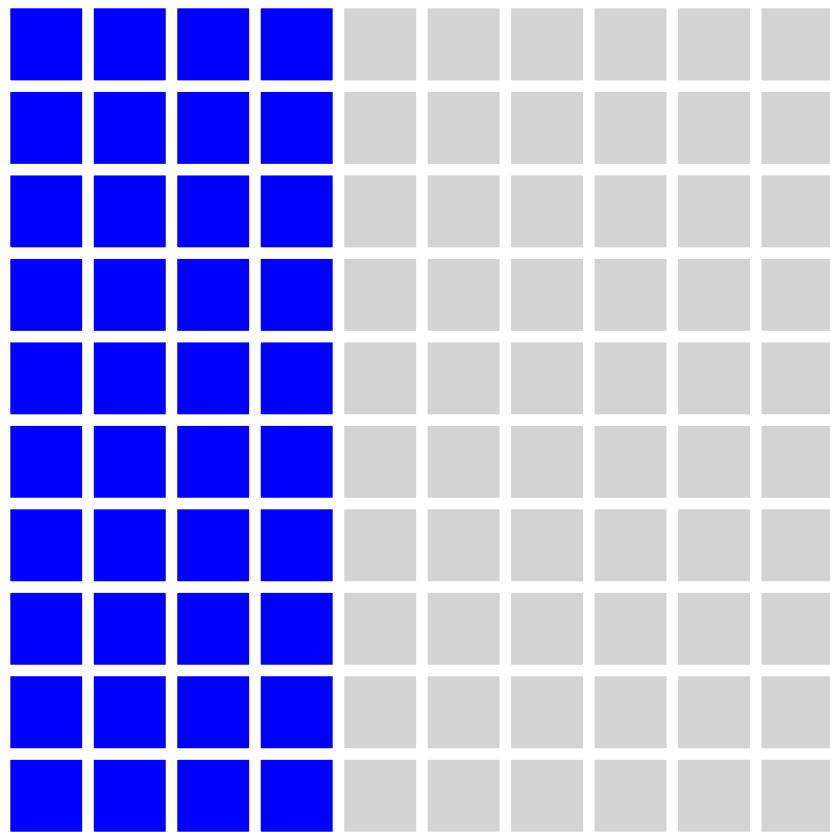
In frequentist statistics, *p* values are defined as *the probability of obtaining a test statistic at least as large as that observed, if the null hypothesis is true. That is, it’s the chance that the data we have collected are atypical and will mislead us into thinking there is a difference, when the true effect is zero.

Let’s pretend we are creative researchers and, over the course of our career, we will develop 100 hypotheses, each of which we test in an experiment, represented by squares in the plot below:

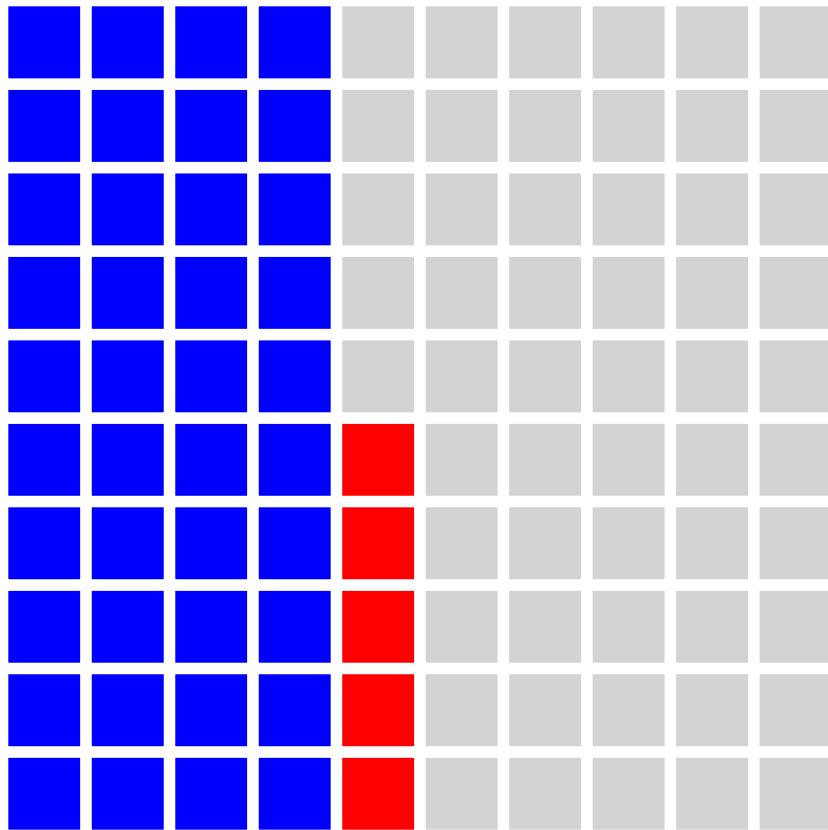


We are conscientious and use sample size calculations for our studies, setting our desired power = .8, and use $p = .05$ as our criterion for rejecting the null hypotheses. As is common, we report tests which reject the null hypothesis *as if our predictions had been supported, and the alternative hypotheses were true* (this is a bad idea, but run with it for the moment).

Let’s be generous and say that, in reality, 50% of our hypotheses are true (the other 50% are nice ideas, but are not correct). Because we set our power to be .8 in our sample size calculation, this means that over the course of our career we will detect around 40 ‘true’ effects, and publish them. These are shown in blue in the figure below:



Because we set our alpha level to 0.05, we will also have some false positives, shown in red:



But what that means is that *for the effects we publish as supporting our hypotheses* (i.e. the blue and red squares) then we will be making false claims $5/40 = 12.5\%$ of the time. This is obviously much higher than the nominal alpha level implies. What's more, if we're not hotshot theorists (or work in an area where theories are less rigorously developed) and only 20% of our hypotheses are in fact true then we will make even more false claims: $5/20 = 25\%$.

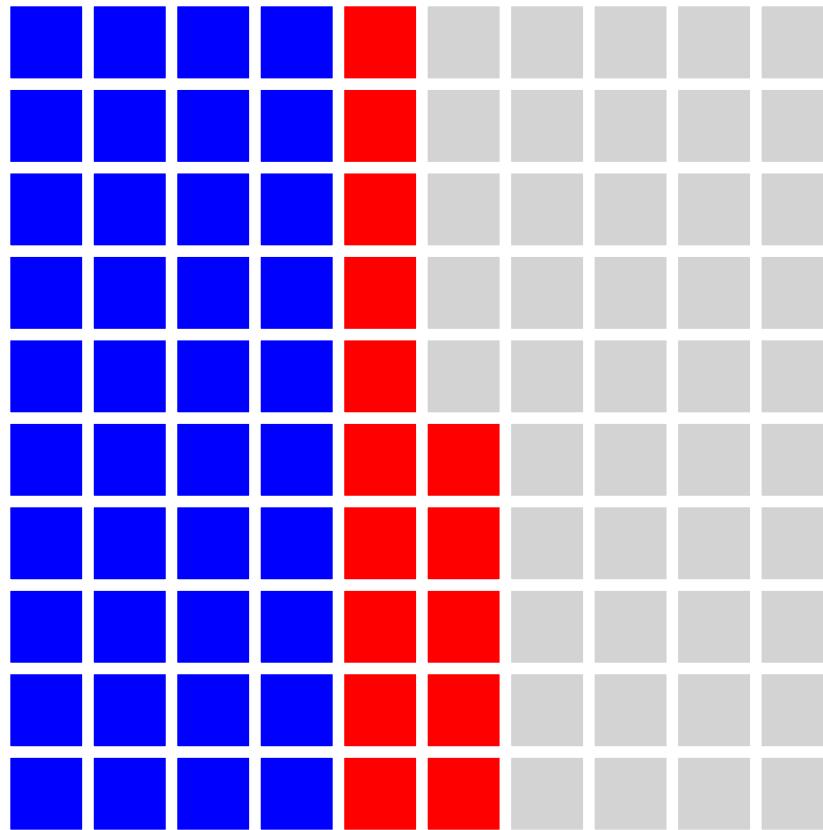
Multiple tests on the same data

Another problem arises when we run multiple statistical tests on the same data.

The most common case is that we have multiple experimental conditions and want to test whether any of the differences between them are significant. If we had experimental 3 groups, then there are 3 possible comparisons between them, for example (A-B, A-C, B-C). This can be compounded if we have multiple outcome measurements (for example, measuring depression *and* anxiety in a clinical trial; see Feise [2002]).

This works like a lucky dip or lottery: if you buy more tickets you have a larger chance of winning a prize. In this case, with three comparisons between the groups (tickets) we have a 15% chance of winning a prize, rather than the 5% we intended.

Assuming each of our 100 experiments allows for 3 tests, any of which would be ‘interesting’ if significant (and we would therefore publish them), then our plot looks like this:



And our ‘false discovery rate’ (at the level of our published papers) is now over one third: $15/40 = 37.5\%$.

21.0.0.0.1 False discovery rates for single experiments

We can also think about a false discovery rate for findings presented within a particular paper, or set of analyses. Imagine we have a 2x4 factorial design, and so we have 8 experimental groups, and 28 possible pairwise combinations:

Table 56: 8 cells in the 2x4 design

Cell.No	A	B
1	1	0
2	2	0
3	3	0
4	4	0
5	1	1
6	2	1
7	3	1
8	4	1

Warning: `as_data_frame()` is deprecated, use `as_tibble()` (but mind the new semantics).
This warning is displayed once per session.

Warning: `funs()` is soft deprecated as of dplyr 0.8.0
please use `list()` instead

```

# Before:
funks(name = f(.))

# After:
list(name = ~ f(.))
This warning is displayed once per session.

```

Table 57: Pairwise comparisons

Comparison	Cells		
1	1	vs.	2
2	1	vs.	3
3	1	vs.	4
...
26	6	vs.	7
27	6	vs.	8
28	7	vs.	8

Assuming there are ‘real’ differences between only one-third of these pairings, then we have the same problem as we did when considering all the experiments in a researchers’ career: Using $p < .05$ as our criterion, and power of .8 then we will report around 9 significant differences, but 1 or 2 of these will be false discoveries (16%).

What to do about it?

There are two decisions you need to make: First, which technique do you want to use to mitigate the risks of multiple comparisons? Second, what is the ‘family’ of tests you are going to apply this to?

If you find the whole language around null hypothesis testing and p values unhelpful, and the detail of multiple comparison adjustment confusing, there is another way: Multiple comparison problems are largely a non-issue for Bayesian analyses [Gelman et al., 2012], and recent developments in the software make simple models like Anova and regression easy to implement in a Bayesian framework in R.

21.0.0.1 Which technique?

Inevitably, it depends!

- If your worry is ever making a type 1 error then you need to *control the familywise error rate*. The Bonferroni or Tukey correction are likely the best solutions (see details below). Both have the disadvantage of increasing the type 2 error rate: you will falsely accept the null more frequently, and so neglect findings which are ‘true’ and might be theoretically or clinically relevant.
- If your main concern is about the number of ‘false discoveries’ you make, and you think it’s important to you to avoid rejecting ‘good’ hypotheses, then you need a technique to control the *false discovery rate*. The Benjamini and Hochberg method, often abbreviated as BH or FDR correction, is what you need.

Principled arguments can be made for both approaches; there is no ‘correct choice’. Which you pick will depend on which concerns are most pressing to you, and also (inevitably) what is conventional in your field.

It’s worth noting that in specific fields (for example neuroscience, where huge numbers of comparisons must be made between voxels or regions in brain imaging studies) more sophisticated methods to control for multiple comparisons are often commonplace and expected [Nichols and Hayasaka, 2003]. This guide only deals with some of the simpler but common cases.

21.0.0.2 How big is the family?



Whichever technique we employ we do need to determine how many tests we want to correct for having run; i.e. what size the ‘family’ of tests is. Unfortunately, deciding which set of tests constitutes a ‘family’ is an unresolved problem, and reasonable people will come up with different solutions. As Feise [2002] put it:

It is unclear how wide the operative term “family” should be [...] Does “family” include tests that were performed, but not published? Does it include a meta-analysis upon those tests? Should future papers on the same data set be accounted for in the first publication? Should each researcher have a career-wise adjusted p-value, or should there be a discipline-wise adjusted p-value? Should we publish an issue-wise adjusted p-value and a year-end-journal-wise adjusted p-value? Should our studies examine only one association at a time, thereby wasting valuable resources? No statistical theory provides answers for these practical issues, because it is impossible to formally account for an infinite number of potential inferences.

The ‘right’ answer, then, will vary from case to case, and local conventions in particular disciplines will dictate what is normal and expected by supervisors and reviewers. However, there are a few common cases for which we can make some recommendations on what might constitute ‘good practice’. We do not claim that these are the ‘correct’ answers, but if you deviate from these recommendations it would be worth having a reasonable justification to hand!

21.0.0.2.1 Multiple outcome variables

In a clinical trial or other applied study you might want to measure more than one variable to detect changes across a variety of domains (e.g. both depression and anxiety).

Although no firm consensus exists, it is common practice to:

- Analyse multiple outcomes without correction (and indicate this in the text)
- Designate one outcome from the study to be ‘primary’, and report this both in pre-trial protocols and reports of the results.

See Feise [2002] for more details.

21.0.0.2.2 Multiple ‘timepoints’ for the outcome

Another common case is where the same outcome variable is measured on multiple occasions to track progress over time. For example, an RCT might measure outcome at baseline, at the end of treatment, and again 12 months later.

In the cases where there are only a few measurements of outcome made then you could choose between:

1. Bonferroni correction of p values for the between-group difference at each timepoint or
2. Designating one of the timepoints as the ‘primary’ outcome.

In practice option 2 is more common, and probably makes more sense in applied settings where it is the longer-term prognosis of participants which matters most.

21.0.0.2.3 Pairwise comparisons in factorial designs

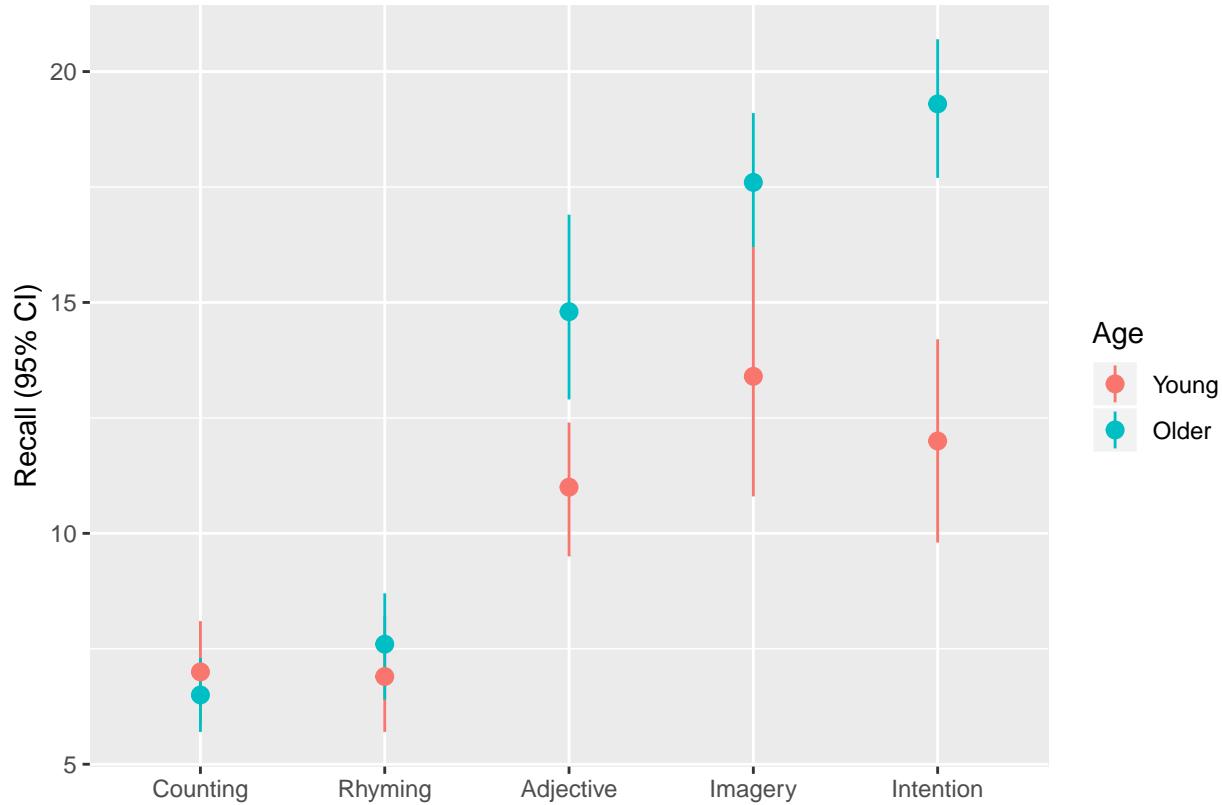
Let’s say you have a complex factorial design and so multiple pairwise comparisons and other contrasts are possible.

- If you are only interested in a small number of the possible pairwise comparisons or specific contrasts then specify this up front. Either report the comparisons without correction (noting this in the text) or use FDR/Bonferroni correction for this small family of tests.
- If you are only interested in specific main effects or interactions from the full factorial model, then you should specify this in advance and report these F-tests or t-tests and associated p values without correction.
- It is perfectly reasonable to be interested only in specific pairwise comparisons contained within an Anova F-test. It is normal to report the F from the Anova, but it is *not* necessary for the F test to be significant before reporting the pairwise comparison of interest (with or without FDR/Bonferroni correction).
- If you have not specified the comparisons of interest in advance, then use FDR or Tukey correction for the complete set of pairwise tests. If you feel able to claim that some of these pairwise comparisons would *never* have been of interest then you could reduce the family size for this correction accordingly, but you should make this justification in your report.

Practical examples

21.0.0.3 Factorial Anova and pairwise comparisons

In the Anova cookbook, we used a dataset from Howell’s textbook which recorded Recall among young v.s. older adults (**Age**) for each of 5 conditions:



In an ideal world we would have published a trial protocol before collecting data, or at the least specified which comparisons were of interest to us. However for the purposes of this example I'll assume you didn't do this, and need to address the potential for multiple comparisons accordingly.

The design of the study suggests that we were interested in the effect of the experimental conditions on recall, and we might have predicted that these experimental manipulations would affect older and younger adults differently. Because we didn't pre-specify our analyses, we should acknowledge that we would likely report any differences between any of the 5 experimental conditions (10 possible comparisons) and any differences between younger and older adults in any of those conditions (45 possible pairwise comparisons).

Because replicating this experiment is relatively cheap (testing might only take 30 minutes per participant), and we are confident that other labs would want to replicate any significant findings we report, we are less concerned with the absolute rate of type 1 errors, but would like to limit our 'false discovery rate' (we have a reputation to maintain). We set our acceptable false discovery rate to 5%.

We run an Anova model, including main effects for Age and Condition and their interaction:

```
eysenck.model <- lm(Recall ~ Age * Condition,
                      data=eysenck)
```

```
car::Anova(eysenck.model, type=3) %>%
  pander()
Registered S3 methods overwritten by 'car':
  method                  from
  influence.merMod        lme4
  cooks.distance.influence.merMod lme4
  dfbeta.influence.merMod   lme4
  dfbetas.influence.merMod  lme4
```

Table 58: Anova Table (Type III tests)

	Sum Sq	Df	F value	Pr(>F)
(Intercept)	490	1	61.05	9.85e-12
Age	1.25	1	0.1558	0.694
Condition	351.5	4	10.95	2.8e-07
Age:Condition	190.3	4	5.928	0.0002793
Residuals	722.3	90	NA	NA

We report that there are significant main effects for Age, Condition, and a significant interaction for Age:Condition. At this point, if we hadn't already plotted the raw data, we would certainly want to do that, or to make predictions for each cell in the design and plot them.

Because the plot of the raw data (see above) suggested that the counting and rhyming conditions produced lower recall rates than the other conditions, we might want to report the relevant pairwise tests. We can use the `lsmeans()` function from the `lsmeans::` package to compute these:

```
# use the lsmeans function which returns an lsm.list object.
eysenck.lsm <- lsmeans::lsmeans(eysenck.model, pairwise~Condition)
NOTE: Results may be misleading due to involvement in interactions

# the `contrasts` element in this list is what we want for now
eysenck.lsm$contrasts
  contrast      estimate    SE df t.ratio p.value
Counting - Rhyming   -0.50 0.896 90 -0.558  0.9807
Counting - Adjective -6.15 0.896 90 -6.865 <.0001
Counting - Imagery   -8.75 0.896 90 -9.767 <.0001
Counting - Intention -8.90 0.896 90 -9.935 <.0001
Rhyming - Adjective -5.65 0.896 90 -6.307 <.0001
Rhyming - Imagery   -8.25 0.896 90 -9.209 <.0001
Rhyming - Intention -8.40 0.896 90 -9.377 <.0001
Adjective - Imagery -2.60 0.896 90 -2.902  0.0367
Adjective - Intention -2.75 0.896 90 -3.070  0.0231
Imagery - Intention -0.15 0.896 90 -0.167  0.9998

Results are averaged over the levels of: Age
P value adjustment: tukey method for comparing a family of 5 estimates
```

By default Tukey correction is applied for multiple comparisons, which is a reasonable choice. If you wanted to adjust for the false discovery rate instead, you can use the `adjust` argument:

```
# not run, just shown as an example
lsmeans::lsmeans(eysenck.model, pairwise~Condition, adjust="fdr")
```

The plot also suggested that older and younger adults appeared to differ for the 'adjective', 'imagery', and 'intention' conditions. We can compute these detailed pairwise comparisons by including the interaction term, Age:Condition in the call to `lsmeans()`:

```
eysenck.age.condition.lsm <-
  lsmeans::lsmeans(eysenck.model,
                  pairwise~Age:Condition,
                  adjust="fdr")

eysenck.age.condition.lsm$contrasts %>%
  broom::tidy() %>%
```

```
snip.middle(., 4) %>%
pander(missing="...", caption="8 of the 45 possible contrasts, FDR corrected", split.tables=Inf)
```

Table 59: 8 of the 45 possible contrasts, FDR corrected

level1	level2	estimate	std.error	df	statistic	p.value
Young,Counting	Older,Counting	0.5	1.267	90	0.3947	0.7263
Young,Counting	Young,Rhyming	0.1	1.267	90	0.07893	0.9373
Young,Counting	Older,Rhyming	-0.6	1.267	90	-0.4736	0.6824
Young,Counting	Young,Adjective	-4	1.267	90	-3.157	0.003251
...
Young,Imagery	Older,Intention	-5.9	1.267	90	-4.657	2.612e-05
Older,Imagery	Young,Intention	5.6	1.267	90	4.42	5.888e-05
Older,Imagery	Older,Intention	-1.7	1.267	90	-1.342	0.2288
Young,Intention	Older,Intention	-7.3	1.267	90	-5.762	3.972e-07

You should note that the FDR adjusted p values do not represent probabilities in the normal sense. Instead, the p value now indicates the *false discovery rate at which the p value should be considered statistically significant*. So, for example, if the adjusted p value = 0.09, then this indicates the contrast *would* be significant if the acceptable false discovery rate is 10%. Researchers often set their acceptable false discovery rate to be 5% out of habit or confusion with conventional levels for alpha, but it would be reasonable to set it to 10% or even higher in some circumstances. Whatever level you set, you need to be able to satisfy both yourself and a reviewer that the result is of interest: the particular values for alpha and FDR you agree on are arbitrary conventions, provided you both agree the finding is worthy of consideration!

Extracting pairwise comparisons to display in your report

In the code above we request FDR-adjusted p values, and then use the `broom::tidy()` function to convert the table into a dataframe, and then use `pander` to display the results as a table. For more information on extracting results from models and other R objects and displaying them properly see the section: models are data too.

21.0.0.3.1 Adjusting for the FDR by hand

If you have a set of tests for which you want to adjust for the FDR it's easy to use R's `p.adjust()` to do this for you.

```
set.seed(5)
p.examples <-
  data_frame(p = runif(4, min=.001, max=.07)) %>%
  mutate(p.fdr = p.adjust(p, method="fdr"),
        p.sig = ifelse(p < .05, "*", ""),
        p.fdr.sig = ifelse(p.fdr < .05, "*", ""))
Warning: `data_frame()` is deprecated, use `tibble()` .
This warning is displayed once per session.

p.examples %>%
  arrange(p) %>%
  pander(caption="Randomly generated 'p values', with and without FDR correction applied.")
```

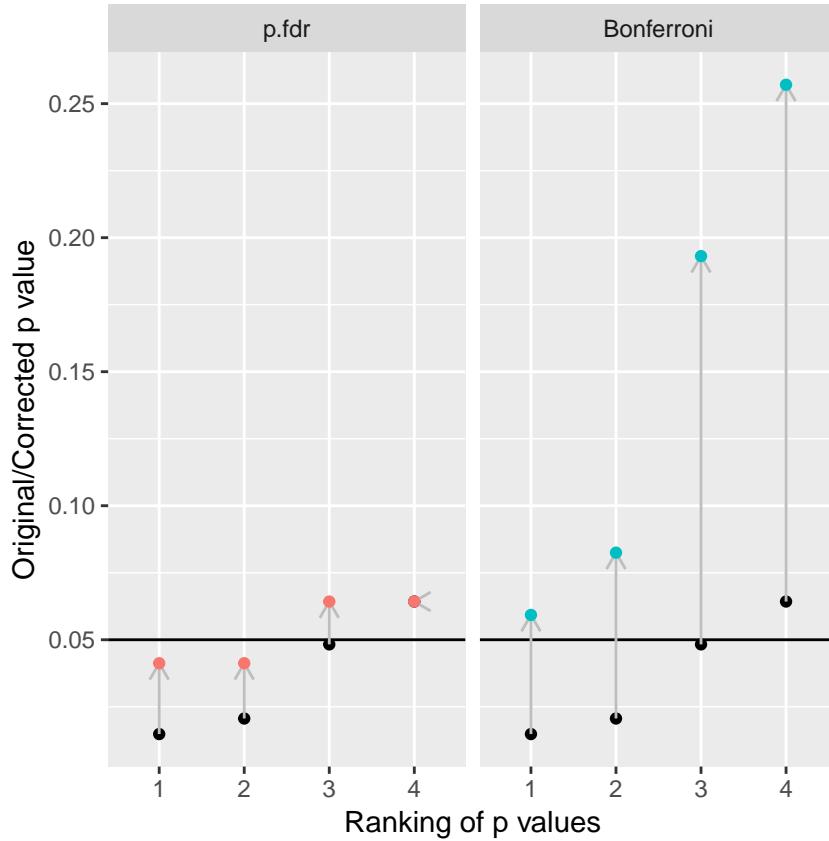


Figure 26: Example of the adjustments made to p values by FDR and Bonferroni methods. Unadjusted p values are shown in black. Note how conservative Bonferroni is with even this small number of comparisons to correct for.

Table 60: Randomly generated ‘p values’, with and without FDR correction applied.

p	p.fdr	p.sig	p.fdr.sig
0.01481	0.04125	*	*
0.02062	0.04125	*	*
0.04828	0.06426	*	
0.06426	0.06426		

We can plot these values to see the effect the adjustment has:

21.0.0.4 Other contrasts

The `lsmeans::` package provides tools to compute contrasts for Anova and other linear models:

```
library(lsmeans)
```

We can see a number of these different contrasts below. First, we run a model including the factor Days:

```
slope.model <- lmer(Reaction ~ factor(Days) + (1|Subject),
                     data=lme4::sleepstudy)
```

And then we create the `lsmeans` object:

```
slope.model.lsm <- lsmeans::lsmeans(slope.model, ~Days)
```

We can use this `lsm` object to calculate various contrasts we might want to see. First, consecutive contrasts where each level is compared with the next:

```
lsmeans::contrast(slope.model.lsm, "consec")
  contrast estimate SE df t.ratio p.value
  1 - 0      7.844 10.5 153 0.749  0.9887
  2 - 1      0.866 10.5 153 0.083  1.0000
  3 - 2     17.630 10.5 153 1.683  0.5276
  4 - 3      5.657 10.5 153 0.540  0.9988
  5 - 4     19.869 10.5 153 1.897  0.3777
  6 - 5      3.660 10.5 153 0.349  1.0000
  7 - 6      6.572 10.5 153 0.627  0.9965
  8 - 7     17.879 10.5 153 1.707  0.5102
  9 - 8     14.222 10.5 153 1.358  0.7628
```

P value adjustment: mvt method for 9 tests

Polynomial contrasts allow us to test whether the increase in RT's over the days is linear or curved:

```
lsmeans::contrast(slope.model.lsm, "poly", adjust="fdr", max.degree=3)
  contrast estimate SE df t.ratio p.value
  linear     1727.1 134.6 153 12.835 <.0001
  quadratic   89.0  85.1 153  1.046  0.4462
  cubic      -65.3 686.1 153 -0.095  0.9243
```

P value adjustment: fdr method for 3 tests

If we don't specify a contrasts argument, or a type of correction, we will get Tukey contrasts for every pairwise comparison:

```
# 45 tests - results not shown because similar to those shown above
lsmeans::contrast(slope.model.lsm, "tukey")
```

If we want to compare particular levels of `Days` with a specific reference level we can use the `trt.vs.ctrl` contrast type. In this example we are comparing each day with day 0, because day 0 is the first in the list of days and we specified `ref=1` (try running the comparisons against day 5).

```
lsmeans::contrast(slope.model.lsm, "trt.vs.ctrl", ref=1, adjust="fdr")
  contrast estimate SE df t.ratio p.value
  1 - 0      7.84 10.5 153 0.749  0.4551
  2 - 0      8.71 10.5 153 0.831  0.4551
  3 - 0     26.34 10.5 153 2.515  0.0167
  4 - 0     32.00 10.5 153 3.055  0.0040
  5 - 0     51.87 10.5 153 4.951  <.0001
  6 - 0     55.53 10.5 153 5.301  <.0001
  7 - 0     62.10 10.5 153 5.928  <.0001
  8 - 0     79.98 10.5 153 7.635  <.0001
  9 - 0     94.20 10.5 153 8.993  <.0001
```

P value adjustment: fdr method for 9 tests

And if we want to compare each level against the grand mean of the sample, the `eff` contrast type is short for `effect`. A way of thinking of this (in English) is that we are asking 'what is the effect of being on a particular day, relative the other average of all days'.

```

lsmeans::contrast(slope.model.lsm, "eff", adjust="fdr")
  contrast estimate SE df t.ratio p.value
  0 effect   -41.86 7.03 153 -5.956 <.0001
  1 effect   -34.01 7.03 153 -4.840 <.0001
  2 effect   -33.15 7.03 153 -4.717 <.0001
  3 effect   -15.52 7.03 153 -2.208 0.0410
  4 effect    -9.86 7.03 153 -1.403 0.1627
  5 effect   10.01 7.03 153 1.425 0.1627
  6 effect   13.67 7.03 153 1.945 0.0670
  7 effect   20.24 7.03 153 2.881 0.0076
  8 effect   38.12 7.03 153 5.425 <.0001
  9 effect   52.34 7.03 153 7.449 <.0001

```

P value adjustment: fdr method for 10 tests

To list all the available contrast types, or see the relevant help files, you can type:

```

# help file describing all the contrasts families
help(pairwise.lsmc)

```

21.0.0.5 Contrasts for interactions

Let's recall our example of factorial Anova:

```

car:::Anova(eysenck.model, type=3) %>%
  pande

```

Table 61: Anova Table (Type III tests)

	Sum Sq	Df	F value	Pr(>F)
(Intercept)	490	1	61.05	9.85e-12
Age	1.25	1	0.1558	0.694
Condition	351.5	4	10.95	2.8e-07
Age:Condition	190.3	4	5.928	0.0002793
Residuals	722.3	90	NA	NA

We can compute pairwise comparisons within the interaction as follows:

```

eysenck.lsm <- lsmeans:::lsmeans(eysenck.model, ~Age:Condition)

```

And the various contrasts, here comparing each cell against the grand mean:

```

lsmeans:::contrast(eysenck.lsm, "eff", adjust="fdr")
  contrast estimate SE df t.ratio p.value
  Young,Counting effect    -4.61 0.85 90 -5.424 <.0001
  Older,Counting effect    -5.11 0.85 90 -6.013 <.0001
  Young,Rhyming effect    -4.71 0.85 90 -5.542 <.0001
  Older,Rhyming effect    -4.01 0.85 90 -4.718 <.0001
  Young,Adjective effect  -0.61 0.85 90 -0.718 0.5275
  Older,Adjective effect   3.19 0.85 90 3.753 0.0004
  Young,Imagery effect     1.79 0.85 90 2.106 0.0475
  Older,Imagery effect     5.99 0.85 90 7.048 <.0001
  Young,Intention effect   0.39 0.85 90 0.459 0.6474
  Older,Intention effect   7.69 0.85 90 9.048 <.0001

```

```
P value adjustment: fdr method for 10 tests
```

Or all of the pairwise tests with Tukey adjustment:

```
lsmeans::contrast(eysenck.lsm, "tukey") %>%
  broom::tidy() %>%
  snip.middle(4) %>%
  pander(missing = "...", caption = "Some of the Tukey corrected pairwise contrasts (table abbreviated)")
```

Table 62: Some of the Tukey corrected pairwise contrasts (table abbreviated)

level1	level2	estimate	std.error	df	statistic	p.value
Young,Counting	Older,Counting	0.5	1.267	90	0.3947	1
Young,Counting	Young,Rhyming	0.1	1.267	90	0.07893	1
Young,Counting	Older,Rhyming	-0.6	1.267	90	-0.4736	1
Young,Counting	Young,Adjective	-4	1.267	90	-3.157	0.06328
...
Young,Imagery	Older,Intention	-5.9	1.267	90	-4.657	0.0004513
Older,Imagery	Young,Intention	5.6	1.267	90	4.42	0.001095
Older,Imagery	Older,Intention	-1.7	1.267	90	-1.342	0.9409
Young,Intention	Older,Intention	-7.3	1.267	90	-5.762	5.006e-06

22 Non-independence

Psychological data often contains natural *groupings*. In intervention research, multiple patients may be treated by individual therapists, or children taught within classes, which are further nested within schools; in experimental research participants may respond on multiple occasions to a variety of stimuli.

Although disparate in nature, these groupings share a common characteristic: they induce *dependency* between the observations we make. That is, our data points are *not independently sampled* from one another.

What this means is that observations *within* a particular grouping will tend, all other things being equal, be more alike than those from a different group.

22.0.0.1 Why does this matter?

Think of the last quantitative experiment you read about. If you were the author of that study, and were offered 10 additional datapoints for ‘free’, which would you choose:

1. 10 extra datapoints from existing participants.
2. 10 data points from 10 new participants.

In general you will gain more *new information* from data from a new participant. Intuitively we know this is correct because an extra observation from someone we have already studies is *less likely to surprise us* or be different from the data we already have than an observation from a new participant.

Most traditional statistical models assume that data *are* sampled independently however. And the precision of the inferences we can draw from statistical models is based on the *amount of information we have available*. This means that if we violate this assumption of independent sampling we will trick our model into thinking we have more information than we really do, and our inferences may be wrong.

23 Fixed and random effects

As noted by Gelman (and summarised here), the terms ‘fixed’ and ‘random’ are used very loosely in both the methodological and applied literature. Gelman identifies 5 different senses in which the distinction between fixed and random effects can be drawn, and this inconsistency can lead to confusion.

23.0.0.1

For practical purposes, if you think that you have some form of grouping in your data and that it makes sense to think of variation in outcomes between these groups then you should probably include it as a random intercept in your model.

Likewise, if you include a predictor in your model and it is reasonable to think that the effect of this predictor would vary between groups in the data (e.g., between individuals) then you should include a random slope effect for this variable.

23.0.0.1.1 Random intercepts

Some example of groupings which should be included as random intercepts:

- Participants
- Classes and Schools
- Therapists or treatment providers (e.g. in cluster randomised trial)
- Stimuli or ‘items’

Groupings which are not clear cut in either direction:

- A smallish number of experimental conditions which could be thought of as ‘sampled’ from a larger population of possible groupings [Gelman et al., 2005]. An example here would be groups which receive different doses of a drug.

Examples of groupings which are probably not best handled as random intercepts:

- Experimental conditions especially where the conditions are qualitatively different (although the interventions might warrant inclusion as a random slope, see below).

23.0.0.1.2 Random slopes

Where the effect of a variable might vary between individuals (or other grouping) should be considered for inclusion as a random slopes. Some examples might include:

- Time (or some function of time)
- An experimental intervention (e.g. in a factorial design)

For a more in depth discussion of when to include a random slope this presentation and transcript from the Bristol CMM is excellent.

24 Scaling predictor variables

When predictors have a natural scale, interpreting them can be relatively straightforward. However when predictors are on an arbitrary scale, or when multiple predictors are on different scales, then interpreting the model (or comparing between models) can be hard. In these cases scaling or standardising predictors in the model can make it easier to interpret the coefficients that are estimated.

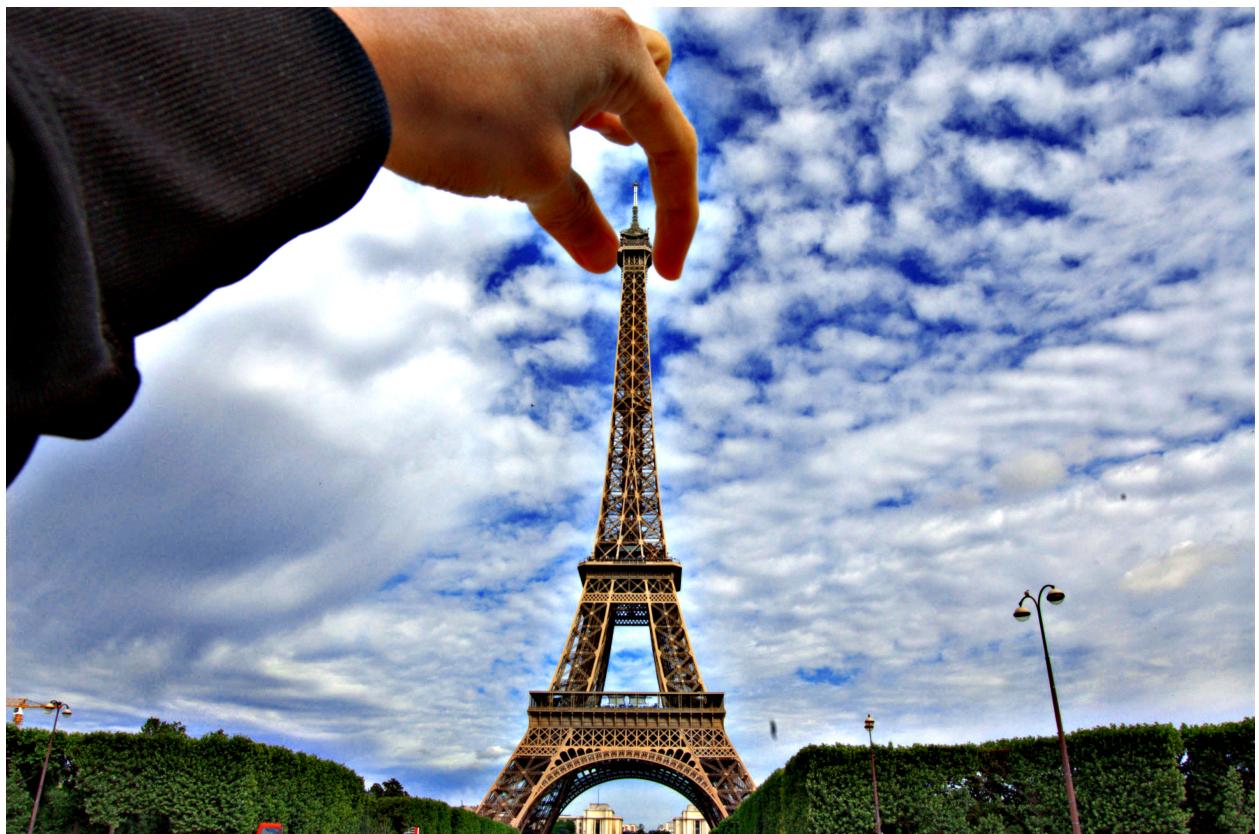


Figure 27: Image: Wikimedia

Standardising

‘Standardising’ predictors, by subtracting the mean and dividing by the standard deviation, is a common way to make interpreting regression models easier, and particularly to make comparisons between predictors — e.g. regarding their relative importance in predicting the outcome.

?scaling_2008 covers in detail the advantages and disadvantages of standardising regression coefficients. Based on the observation that we often wish to compare continuous with binary predictors, they recommend standardisation by subtracting the mean and dividing by two standard deviations (rather than the normal one SD). The arm package implements this procedure, and makes it easy to automatically scale the predictors in a linear model.

First, we run the linear model:

```
m1 <- lm(mpg ~ wt + am, data=mtcars)
m1

Call:
lm(formula = mpg ~ wt + am, data = mtcars)

Coefficients:
(Intercept)          wt            am
 37.32155     -5.35281    -0.02362
```

And then use arm::standardize to standardize the coefficients:

```
arm::standardize(m1)

Call:
lm(formula = mpg ~ z.wt + c.am, data = mtcars)

Coefficients:
(Intercept)      z.wt        c.am
 20.09062    -10.47500   -0.02362
```

This automatically scales the data for m1 and re-fits the model.

An alternative is to use the MuMIn::stdizeFit although this applies scaling rules slightly differently to arm, in this case standardising by a single SD:

```
MuMIn::stdizeFit(m1, mtcars)
Registered S3 method overwritten by 'MuMIn':
  method           from
  predict.merMod lme4

Call:
lm(formula = mpg ~ wt + am, data = mtcars)

Coefficients:
(Intercept)          wt            am
 37.32155     -5.35281    -0.02362
```

Check the help file for MuMIn::stdize for a detailed discussion of the differences with arm::standardize.

Dichotomising continuous predictors (or outcomes)

Dichotomising a continuous scale is almost always a bad idea. Although it is sometimes done to aid interpretation or presentation, there are better alternatives (for example estimating means from a model using Stata's `margins` command and plotting them, something we will do in the next session). As the Cochrane collaboration puts it:

The down side of converting other forms of data to a dichotomous form is that information about the size of the effect may be lost. For example a participant's blood pressure may have lowered when measured on a continuous scale (mmHg), but if it has not lowered below the cut point they will still be in the 'high blood pressure group' and you will not see this improvement. In addition the process of dichotomising continuous data requires the setting of an appropriate clinical point about which to 'split' the data, and this may not be easy to determine.

See <http://www.cochrane-net.org/openlearning/html/mod11-2.htm> and also Peacock et al.. Also note that trichotomising (splitting into 3) is likely to be a better/better/more efficient approach, see ?.

25 Non-scale outcomes

Standard regression models are linear combinations of parameters estimated from the data. Multiplying these parameters by different values of the predictor variables gives estimates of the outcome.

However, because there's no hard limit on the range of predictor variables (at least, no limit coded into the model itself) the predictions of a linear model in theory range between negative $-\infty$ (infinity) and $+\infty$. Although values approaching infinity might be very unlikely, there is no hard limit on either the parameters we fit (the regression coefficients) or the predictor values themselves.

Where outcome data are continuous or somewhat like a continuous variable this isn't usually a problem. Although our models might predict some improbable values (for example, that someone is 8 feet tall), they will not often be strictly impossible.

It might occur to you at this point that, if a model predicted a height of -8 feet or a temperature below absolute zero, then this would be impossible. And this is true, and a theoretical violation of the assumption of the linear model that the outcome can range between $-\infty$ (infinity) and $+\infty$. However researchers use linear regression to predict many outcomes which have this type of range restriction and, although models can make strange predictions in edge cases, they are useful and can make good predictions most of the time.

However for other types of outcome — including binary or count data, or other quantities like the duration-to-failure — this often won't be the case, and standard linear regression may fail to make sensible predictions even in cases that are not unusual.

For binary data we want to predict the probability of a positive response, and this can range between zero and 1. For count data, predicted outcomes must always be non-negative (i.e. zero or greater). For these data, the lack of constraint on predictions from linear regression are a problem.

25.1 Link functions

Logistic and poisson regression extend regular linear regression to allow us to *constrain* linear regression to predict within the range of possible outcomes. To achieve this, logistic regression, poisson regression and other members of the family of 'generalised linear models' use different 'link functions'.

Link functions are used to connect the outcome variable to the linear model (that is, the linear combination of the parameters estimated for each of the predictors in the model). This means we can use linear models which still predict between $-\infty$ and $+\infty$, but without making inappropriate predictions.

For linear regression the link is simply the identity function — that is, the linear model directly predicts the outcome. However for other types of model different functions are used.

A good way to think about link functions is as a *transformation* of the model's predictions. For example, in logistic regression predictions from the linear model are transformed in such a way that they are constrained to fall between 0 and 1. Thus, although the underlying linear model allows values between $-\infty$ and $+\infty$, the link function ensures predictions fall between 0 and 1.

Logistic regression

When we have binary data, we want to be able run something like regression, but where we predict a *probability* of the outcome.

Because probabilities are limited to between 0 and 1, to link the data with the linear model we need to transform so they range from $-\infty$ (infinity) to $+\infty$.

You can think of the solution as coming in two steps:

25.1.0.1 Step 1

We can transform a probability on the 0—1 scale to a $0 \rightarrow \infty$ scale by converting it to *odds*, which are expressed as a ratio:

$$\text{odds} = \frac{p}{1 - p}$$

Probabilities and odds ratios are two *equivalent* ways of expressing the same idea.

So a probability of .5 equates to an odds ratio of 1 (i.e. 1 to 1); $p=.6$ equates to odds of 1.5 (that is, 1.5 to 1, or 3 to 2), and $p = .95$ equates to an odds ratio of 19 (19 to 1).

Odds convert or *map* probabilities from 0 to 1 onto the real numbers from 0 to ∞ .

We can reverse the transformation too (which is important later) because:

$$\text{probability} = \frac{\text{odds}}{1 + \text{odds}}$$

25.1.0.1.1

- If a bookie gives odds of 66:1 on a horse, what probability does he think it has of winning?
- Why do bookies use odds and not probabilities?
- Should researchers use odds or probabilities when discussing with members of the public?

25.1.0.2 Step 2

When we convert a probability to odds, the odds will always be $>$ zero.

This is still a problem for our linear model. We'd like our 'regression' coefficients to be able to vary between $-\infty$ and ∞ .

To avoid this restriction, we can take the *logarithm* of the odds — sometimes called the *logit*.

The figure below shows the transformation of probabilities between 0 and 1 to the log-odds scale. The logit has two nice properties:

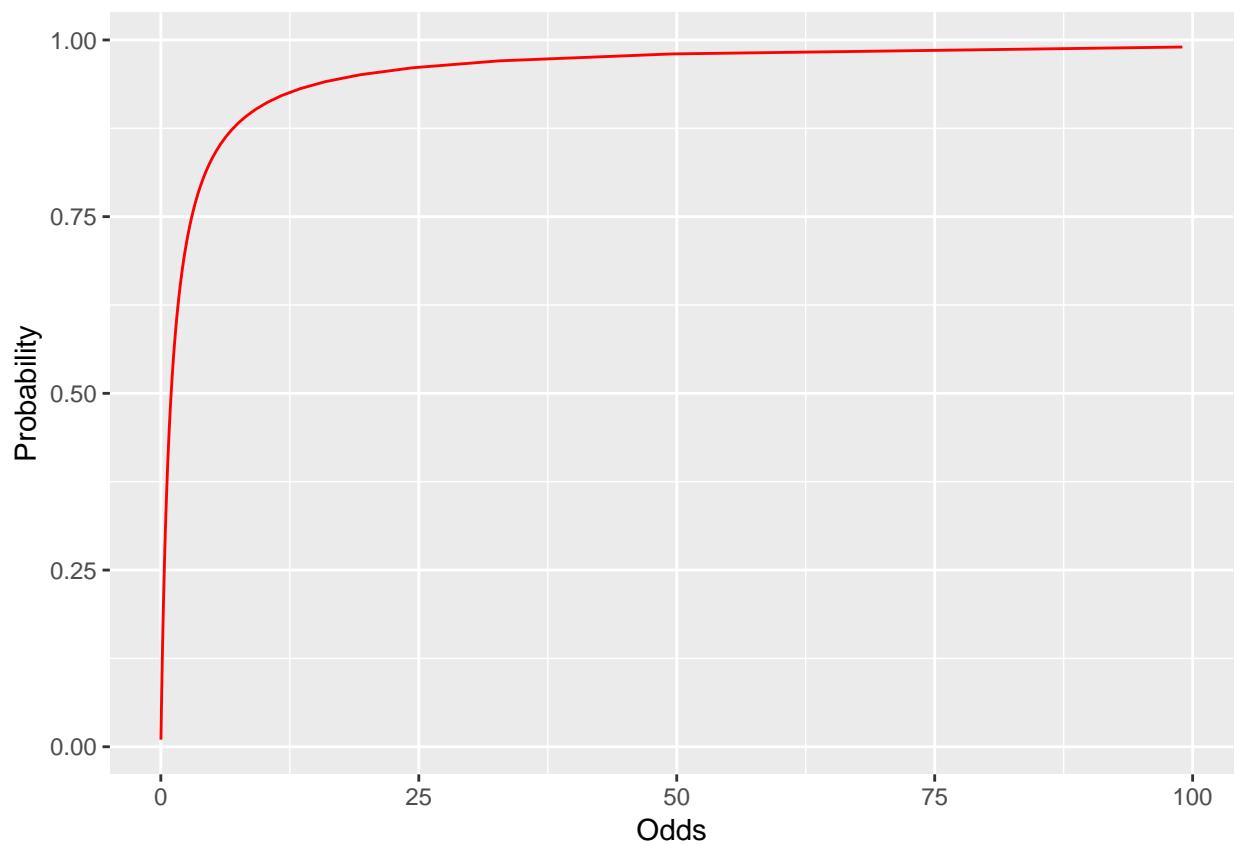


Figure 28: Probabilities converted to the odds scale. As p approaches 1 Odds goes to infinity.

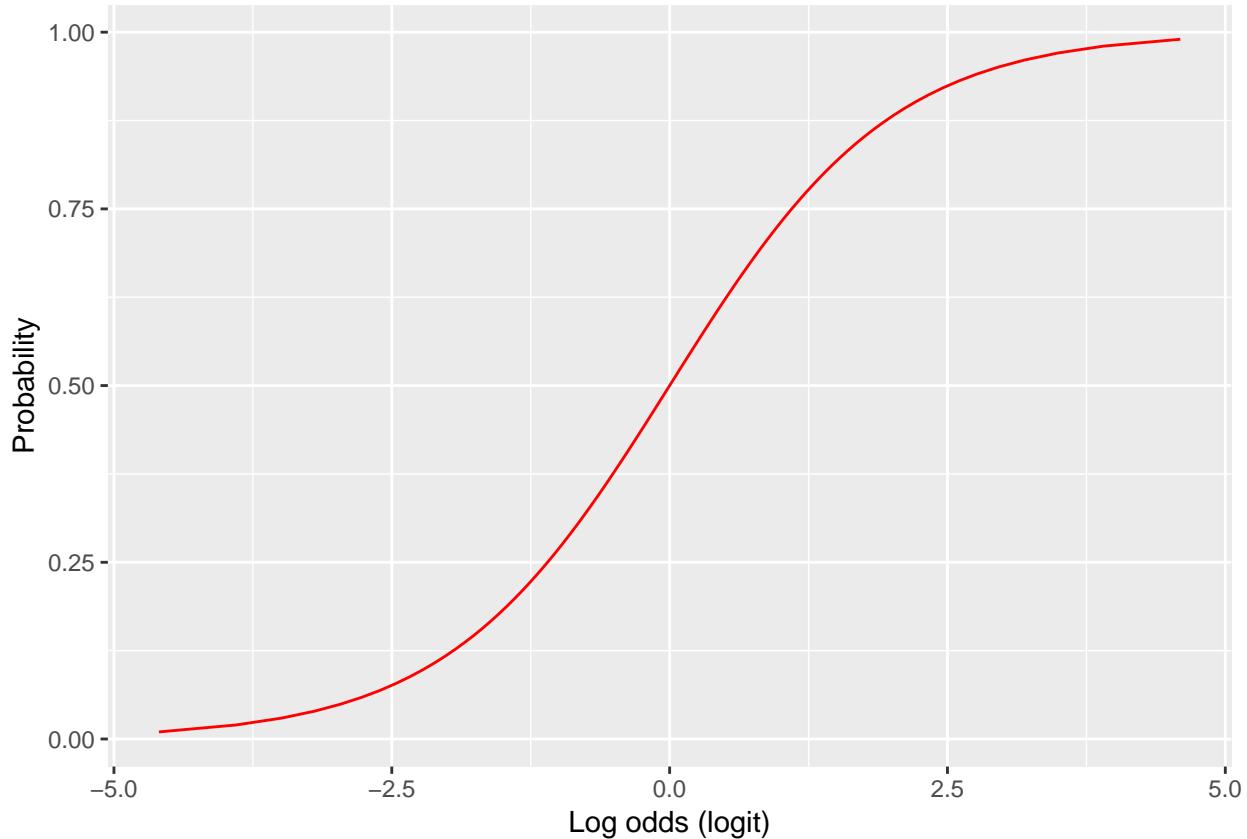


Figure 29: Probabilities converted to the logit (log-odds) scale. Notice how the slope implies that as probabilities approach 0 or 1 then the logit will get very large.

1. It converts odds of less than one to negative numbers, because the *log* of a number between 0 and 1 is always negative[¹].
2. It flattens the rather square curve for the odds in the figure above, and

25.1.0.3 Reversing the process to interpret the model

As we've seen here, the logit or logistic link function transforms probabilities between 0/1 to the range from negative to positive infinity.

This means logistic regression coefficients are in log-odds units, so we must interpret logistic regression coefficients differently from regular regression with continuous outcomes.

- In linear regression, the coefficient is the change the outcome for a unit change in the predictor.
- For logistic regression, the coefficient is the *change in the log odds of the outcome being 1, for a unit change in the predictor*.

If we want to interpret logistic regression in terms of probabilities, we need to undo the transformation described in steps 1 and 2. To do this:

1. We take the exponent of the logit to 'undo' the log transformation. This gives us the *predicted odds*.
2. We convert the odds back to probability.

25.1.0.4 A hypothetical example

Imagine if we have a model to predict whether a person has any children. The outcome is binary, so equals 1 if the person has any children, and 0 otherwise.

The model has an intercept and one predictor, age in years. We estimate two parameters: $\beta_0 = 0.5$ and $\beta_1 = 0.02$.

The outcome (y) of the linear model is the log-odds.

The model prediction is: $\hat{y} = \beta_0 + \beta_1 age$

So, for someone aged 30:

- the predicted log-odds = $0.5 + 0.02 * 30 = 1.1$
- the predicted odds = $exp(1.1) = 3.004$
- the predicted probability = $3.004/(1 + 3.004) = .75$

For someone aged 40:

- the predicted log-odds = $0.5 + 0.02 * 40 = 1.3$
- the predicted odds = $exp(1.3) = 3.669$
- the predicted probability = $3.669/(1 + 3.669) = .78$

25.1.0.5 Regression coefficients are odds ratios

One final twist: In the section above we said that in logistic regression the coefficients are the *change in the log odds of the outcome being 1, for a unit change in the predictor*.

Without going into too much detail, one nice fact about logs is that if you take the log of two numbers and subtract them to take the difference, then this is equal to dividing the same numbers and then taking the log of the result:

```
A <- log(1)-log(5)
B <- log(1/5)
```

```
# we have to use rounding because of limitations in
# the precision of R's arithmetic, but A and B are equal
round(A, 10) == round(B, 10)
[1] TRUE
```

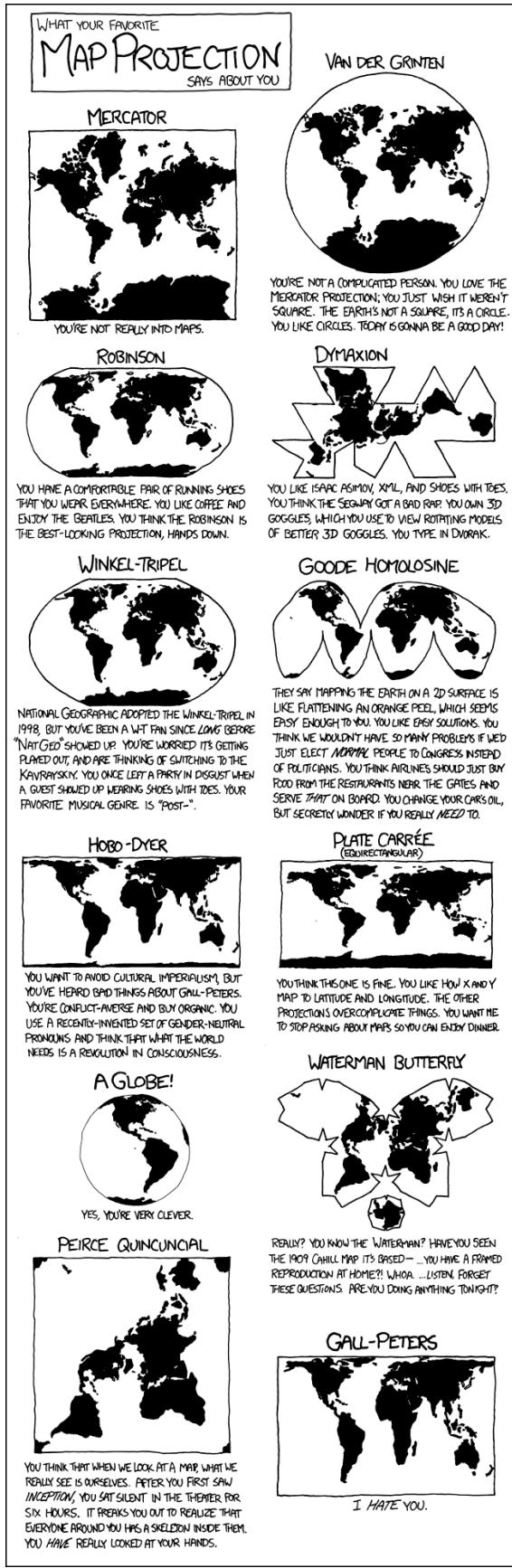
This means that *change in the log-odds* is the same as the *ratio of the odds*

So, once we undo the log transformation by taking the exponent of the coefficient, we are left with the *odds ratio*.

You can now jump back to running logistic regression.

26 Building and choosing models

Like maps, models are imperfect but useful



Maps are always imperfect, and the decisions we make as we build them entail different visions of how the world is.

For example, when making maps cartographers must decide at what size to print it, and how much detail to include. The smaller the map the more convenient it is to carry, but a smaller map can never represent reality as accurately. Similarly, we must leave out details to make the map practical to use; but *what* we omit depends on the planned use of the map.

Stretching the analogy a little, we could evaluate maps according to their performance: did the map help us navigate to a desired location, for example? This is likely to be a function of two things:

- *Veracity*: that is, how faithfully does the map represent the ‘reality’ of the terrain it covers?
- *Simplicity*: does the map include details which are ephemeral, or irrelevant to the task in hand?

At one extreme, we might take an aerial photo of a city, which would accurately represent the terrain, but would include many details which are ephemeral or transient and not necessary to the task of, for example, finding the city’s railway station. Indeed the photo is literally a snapshot of the city on a given day, and some elements might be unhelpful in navigating the city several months later; the photo might include temporary roadworks, for example, which could lead the reader to assume a street was blocked to traffic when this was not normally the case. At the other extreme, we might consider a highly stylised cartoon map of the same city, which includes only major roads and the station itself, but none of the secondary roads or other landmarks. Here the map is so lacking in detail that it may be hard to orient ourselves to the actual terrain.

Statistical models are not exactly like maps, but we can evaluate them using similar criteria. A good model enables the user to make sense of the data they collect (i.e. navigate the city) and this involves making reliable predictions. Reliable predictions are possible when the model accurately represents the underlying reality of a phenomena, but also when the model is simple enough to ignore ephemeral or irrelevant information.

Evaluating and selecting statistical models involves just this kind of tradeoff between *veracity* and *simplicity*.

To stretch the analogy above, when we collect data, we are taking a photo of the city, making a snapshot of a given process at a particular place and time. This snapshot (hopefully) contains valuable information about how the process works in general, but it also includes noise and irrelevant information which could distract us, preventing us from using the data to accurately predict the future.

Overfitting/underfitting

To stealuse a different analogy, imagine you want to predict who will win the next US presidential election:

In total there have only been 57 presidential elections and 44 presidents in the US, which isn’t a lot of data to learn from. The point of the cartoon is that if we try and include ephemeral predictors like beards and false teeth then the space of possible combinations becomes huge. This means that some of these combinations will start to fit the data because of chance variations, and not because they represent some fundamental aspect of the process generating the outcome (election wins).

If you allow statistical models to expand in this way (that is, driven only by patterns in the data you have) then you will find that, inevitably, the model fits the data you have, but the you won’t be able to predict new data at all.

This is one of the reasons why we prefer simpler models, all other things being equal. It also explains why practices like stepwise regression, where we add or remove variables depending on whether they are ‘statistically significant’, are almost always a bad idea. We should only accept a complex model (e.g. one including false teeth as a predictor of the presidency) if we have enough evidence to be confident this will predict new data.

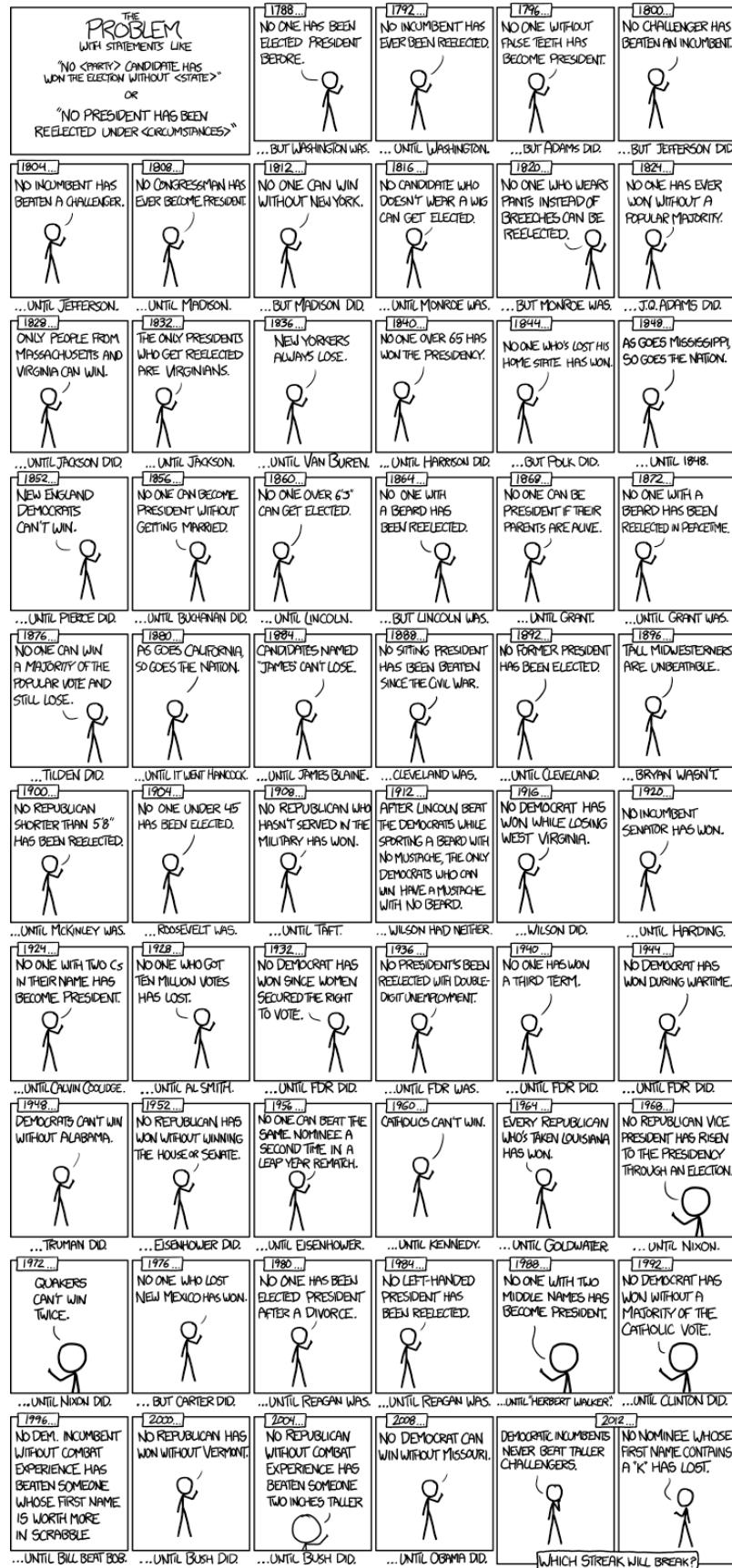


Figure 30: Cartoon: xkcd.com

Choosing the ‘right variables’

As an aside, if you do find yourself in a situation with a large number of predictors and hope to build a simpler model — either with the goal of prediction or theory development — then it’s hopefully obvious from the discussion above that simply dropping or selecting predictors based on p values is unlikely to lead to good science. Not only will you tend to over-fit the data you have, but you may also discount valid predictor variables because of problems like multicollinearity (correlations between predictors).

In other disciplines where inference from observational data are more common (e.g. ecology and biological anthropology) the concepts of model averaging or multimodal inference have gained currency, and may be useful to applied psychologists. The core concept is that if we have several models, all of which are approximately as good as each other, then we should make predictions and inferences by averaging across the models, weighting each model by how well it fits the data: that is, we give the most weight to the best fitting model, but don’t completely discount other models which fit almost as well.

See McElreath [2016], chapter 6, for a good introduction. This video explains the concept well. The R package MuMIn package implements most of the methods needed. See also this interesting post on model selection problems.

References

References

- Thom Baguley. Standardized or simple effect size: What should be reported? *British Journal of Psychology*, 100(3):603–617, 2009. URL <http://onlinelibrary.wiley.com/doi/10.1348/000712608X377117/abstract>.
- Roger Bakeman. Recommended effect size statistics for repeated measures designs. *Behavior research methods*, 37(3):379–384, 2005.
- Reuben M Baron and David A Kenny. The moderator–mediator variable distinction in social psychological research: Conceptual, strategic, and statistical considerations. *Journal of personality and social psychology*, 51(6):1173, 1986.
- Dale J Barr, Roger Levy, Christoph Scheepers, and Harry J Tily. Random effects structure for confirmatory hypothesis testing: Keep it maximal. *Journal of memory and language*, 68(3):255–278, 2013. URL <http://talklab.psy.gla.ac.uk/KeepItMaximalR2.pdf>.
- Ronald J Feise. Do multiple outcome measures require p-value adjustment? *BMC medical research methodology*, 2(1):8, 2002.
- Andrew Gelman. Analysis of variance—why it is more important than ever. 33(1):1–53.
- Andrew Gelman. Scaling regression inputs by dividing by two standard deviations. *Statistics in medicine*, 27(15):2865–2873, 2008. URL <http://www.stat.columbia.edu/~gelman/research/published/standardizing7.pdf>.
- Andrew Gelman, Jennifer Hill, and Masanao Yajima. Why we (usually) don’t have to worry about multiple comparisons. *Journal of Research on Educational Effectiveness*, 5(2):189–211, 2012. URL <http://www.stat.columbia.edu/~gelman/research/published/multiple2f.pdf>.
- Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian data analysis*, volume 2. CRC press Boca Raton, FL, 2014.
- Andrew Gelman et al. Analysis of variance—why it is more important than ever. *The annals of statistics*, 33(1):1–53, 2005.

- Gerd Gigerenzer and Adrian Edwards. Simple tools for understanding risks: from innumeracy to insight. *BMJ: British Medical Journal*, 327(7417):741, 2003.
- Garrett Grolemund. *Hands-On Programming with R: Write Your Own Functions and Simulations.* " O'Reilly Media, Inc.", 2014.
- David C Howell. *Statistical methods for psychology.* Cengage Learning, 2012.
- David C Howell. *Fundamental Statistics for the Behavioral Sciences.* Cengage Learning, 2016.
- Kara Maria Kockelman and Young-Jun Kweon. Driver injury severity: an application of ordered probit models. *Accident Analysis & Prevention*, 34(3):313–321, 2002.
- R McElreath. Statistical rethinking: A bayesian course with examples in r and stan, 2016.
- George E Newman and Brian J Scholl. Bar graphs depicting averages are perceptually misinterpreted: The within-the-bar bias. *Psychonomic bulletin & review*, 19(4):601–607, 2012.
- Thomas Nichols and Satoru Hayasaka. Controlling the familywise error rate in functional neuroimaging: a comparative review. *Statistical methods in medical research*, 12(5):419–446, 2003.
- J.l. Peacock, O. Sauzet, S.m. Ewings, and S.m. Kerry. Dichotomising continuous data while retaining statistical power using a distributional approach. 31(26):3089–3103. ISSN 1097-0258. doi: 10.1002/sim.5354. URL <http://onlinelibrary.wiley.com/doi/10.1002/sim.5354/abstract>.
- Kristopher J. Preacher and Andrew F. Hayes. SPSS and SAS procedures for estimating indirect effects in simple mediation models. 36(4):717–731. ISSN 0743-3808, 1532-5970. doi: 10.3758/BF03206553. URL <https://link.springer.com/article/10.3758/BF03206553>.
- Graeme D Ruxton. The unequal variance t-test is an underused alternative to student's t-test and the mann-whitney u test. *Behavioral Ecology*, 17(4):688–690, 2006. URL <https://doi.org/10.1093/beheco/ark016>.
- James B. Schreiber, Amaury Nora, Frances K. Stage, Elizabeth A. Barlow, and Jamie King. Reporting structural equation modeling and confirmatory factor analysis results: A review. 99(6):323–338.
- James B Schreiber, Amaury Nora, Frances K Stage, Elizabeth A Barlow, and Jamie King. Reporting structural equation modeling and confirmatory factor analysis results: A review. *The Journal of educational research*, 99(6):323–338, 2006.
- Alan R Tait, Terri Voepel-Lewis, Brian J Zikmund-Fisher, and Angela Fagerlin. Presenting research risks and benefits to parents: does format matter? *Anesthesia and analgesia*, 111(3):718, 2010.
- WN Venables. Exegeses on linear models. In *S-Plus User's Conference, Washington DC*, 1998. URL <http://www.stats.ox.ac.uk/pub/MASS3/Exegeses.pdf>.
- Zhongheng Zhang. Missing data exploration: highlighting graphical presentation of missing pattern. *Annals of translational medicine*, 3(22), 2015. URL <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4701517/>.
- Xinshu Zhao, John G. Lynch Jr, and Qimei Chen. Reconsidering baron and kenny: Myths and truths about mediation analysis. 37(2):197–206.