Data Science Workshop

British Society for Proteomic Research Meeting 2018

Alistair Bailey

July 04 2018

Contents

Overview

This book covers:

1. An introduction to R and RStudio
2. An introduction to tidyverse and base R
3. Importing and transforming proteomics data
4. Visualisation of proteomics analysis

The analysis is of an example data set of observations for 7702 proteins from cells in three control experiments and three treatment experiments. The observations are signal intensity measurements from the mass spectrometer. These intensities relate the concentration of protein observed in each experiment and under each condition. The analysis transforms the data to examine the effect of treatment on the cellular proteome and visualise the output using a volcano plot and a heatmap. Click here to download the csv file.

Requirements

An up to date version of R (R Core Team, 2018) and RStudio (RStudio Team, 2018).

If you are new to R, then the first thing to know is that R is a programming language and RStudio is a program for working with R called an integrated development environment (IDE). You can use R without RStudio, but not the other way around. Further details in Chapter 1.1.

Download R here and Download RStudio Desktop here.

These materials were generated using R version 3.5.0.

Once you've installed R and RStudio, you'll also need a few R packages. Packages are collections of functions.

Open RStudio and put the code below into the `Console` window and press `Enter` to install these three packages.

```
install.packages(c("plyr","tidyverse","gplots","pheatmap"))
```

Chapter 1

Introduction

There are many resources for learning R on the web. Much of Chapters 1, 2, 3 and 4 derive from a Data Carpentry lesson using ecological data that I have previously reworked, which in turn takes a lot from Hadley Wickham's R for Data Science aka **R4DS**. Follow the links to access those materials.

Chapter 5 deals with some statistical transformations and visualisation methods in the context of proteomics data.

Whilst finally in Chapter 6 there is some advice about how to build upon the materials covered here.

In terms of philosophy:

1. The primary motivation for using tools such as R is to get more done, in less time and with less pain.
2. And the overall aim is to *understand and communicate* findings from our data.

As shown in Figure 1.1 of typical data analysis workflow, to acheive this aim we need to learn tools that enable us to perform the fundamental tasks of tasks of importing, tidying and often transforming the data. Transformation means for example, selecting a subset of the data to work with, or calculating the mean of a set of observations. We'll cover that in Chapter 5.

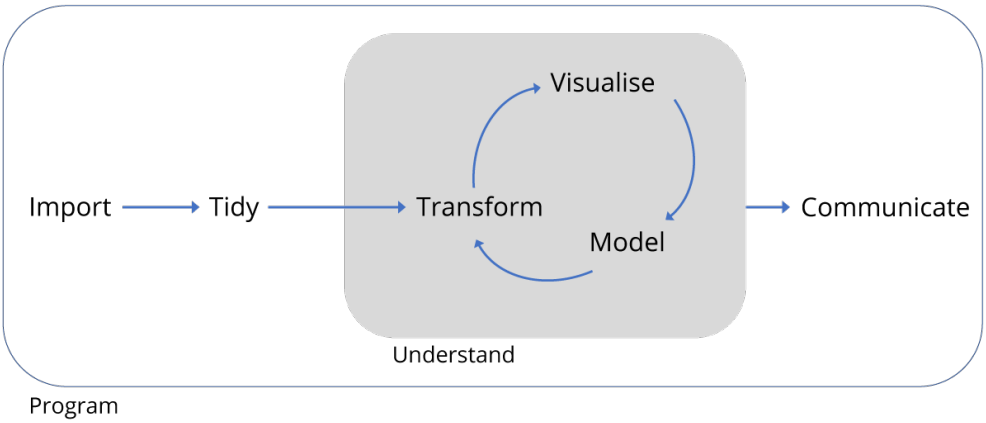


Figure 1.1: Data project workflow.

But first...

1.1 What are R and RStudio?

"There are only two kinds of languages: the ones people complain about and the ones nobody uses"

Bjarne Stroustrup

R is a programming language that follows the philosophy laid down by it's predecessor S. The philosophy being that users begin in an interactive environment where they don't consciously think of themselves as programming. It was created in 1993, and documented in (Ihaka and Gentleman, 1996).

Reasons R has become popular include that it is both open source and cross platform, and that it has broad functionality, from the analysis of data and creating powerful graphical visualisations and web apps.

Like all languages though it has limitations, for example the syntax is initially confusing.

Take for example the word `environment`...

1.1.1 Environments

An environment is where we bring our data to work with it. Here we work in a R envrionment, using the R language as a set of tools. **RStudio** is an integrated development environment, or IDE for R programming. It is regularly updated, and upgrading enables access to the latest features.

The latest version can be downloaded here: http://www.rstudio.com/download

1.2 Why learn R, or any language ?

We can write R code without saving it, but it's generally more useful to write and save our code as a script. Working with scripts makes the steps you used in your analysis clear, and the code you write can be inspected by someone else who can give you feedback and spot mistakes.

Learning R (or any programming language) and working with scripts forces you to have deeper understanding of what you are doing, facilitates your learning and comprehension of the methods you use:

- Writing and publishing code is important for reproducible resarch
- R has many thousands of packages covering many disciplines.
- R can work with many types of data.
- They is a large R community for development and support.
- Using R gives you control over your figures and reports.

1.3 Finding your way around RStudio

Let's begin by learning about RStudio, the Integrated Development Environment (IDE).

We will use R Studio IDE to write code, navigate the files found on our computer, inspect the variables we are going to create, and visualize the plots we will generate. R Studio

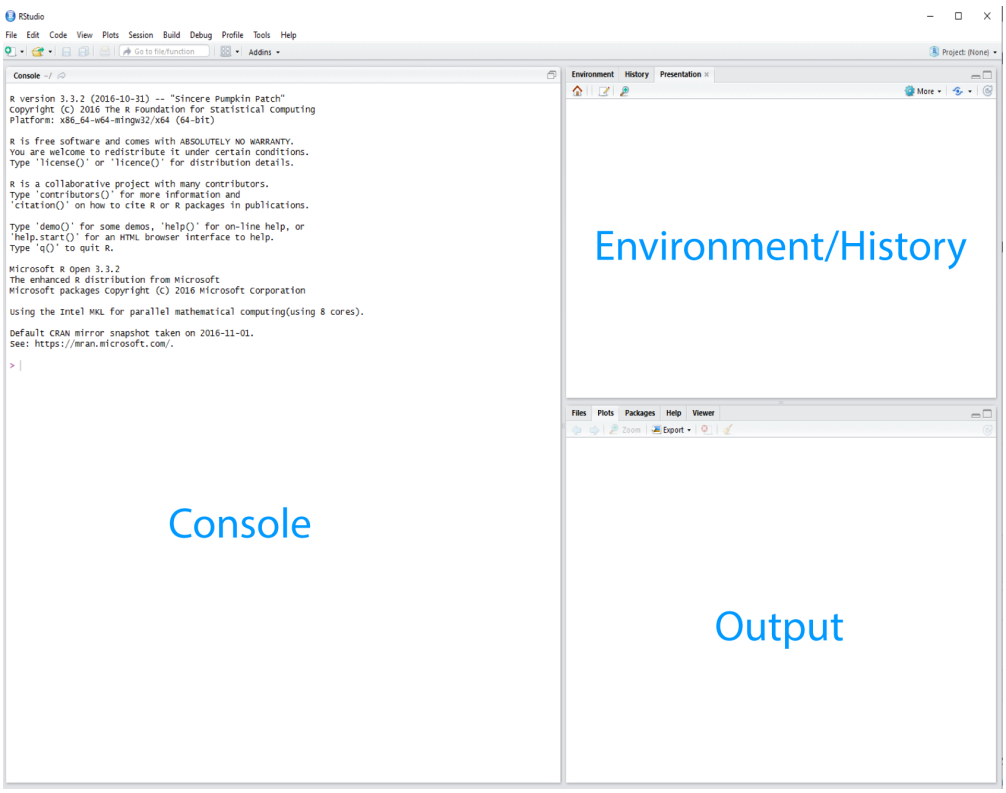


Figure 1.2: The Rstudio Integrated Development Environment (IDE).

can also be used for other things (e.g., version control, developing packages, writing Shiny apps) that we don't have time to cover during this workshop.

R Studio is divided into "Panes", see Figure 1.2.

When you first open it, there are three panes,the console where you type commands, your environment/history (top-right), and your files/plots/packages/help/viewer (bottom-right).

The enivronment shows all the R objects you have created or are using, such as data you have imported.

The output pane can be used to view any plots you have created.

Not opened at first start up is the fourth default pane: the script editor pane, but this will open as soon as we create/edit a R script (or many other document types). *The script editor is where will be typing much of the time.*

The placement of these panes and their content can be customized (see menu, R Studio ->

Tools -> Global Options -> Pane Layout). One of the advantages of using R Studio is that all the information you need to write code is available ina single window. Additionally, with many shortcuts, auto-completion, and highlighting for the major file types you use while developing in R, R Studio will make typing easier and less error-prone.

Time for a philosphical diversion. . .

1.3.1 What is real?

At the start, we might consider our environment "real" - that is to say the objects we've created/loaded and are using are "real". But it's much better in the long run to consider our scripts as "real" - our scripts are where we write down the code that creates our objects that we'll be using in our environment.

As a script is a document, it is reproducible

Or to put it another way: we can easily recreate an environment from our scripts, but not so easily create a script from an enivronment.

To support this notion of thinking in terms of our scripts as real, we recommend turning off the preservation of workspaces between sessions by setting the Tools > Global Options menu in R studio as shown in Figure 1.3:

1.4 Where am I?

R studio tells you where you are in terms of directory address like so:

(ref:working-dir)

If you are unfamiliar with how computers structure folders and files, then consider a tree with a root from which the trunk extends and branches divide. In the image above, the ~ symbol represents a contraction of the path from the root to the 'home' directory (in Windows this is 'Documents') and then the forward slashes are the branches. (Note: Windows uses backslashes, Unix type systems and R use forwardslashes).

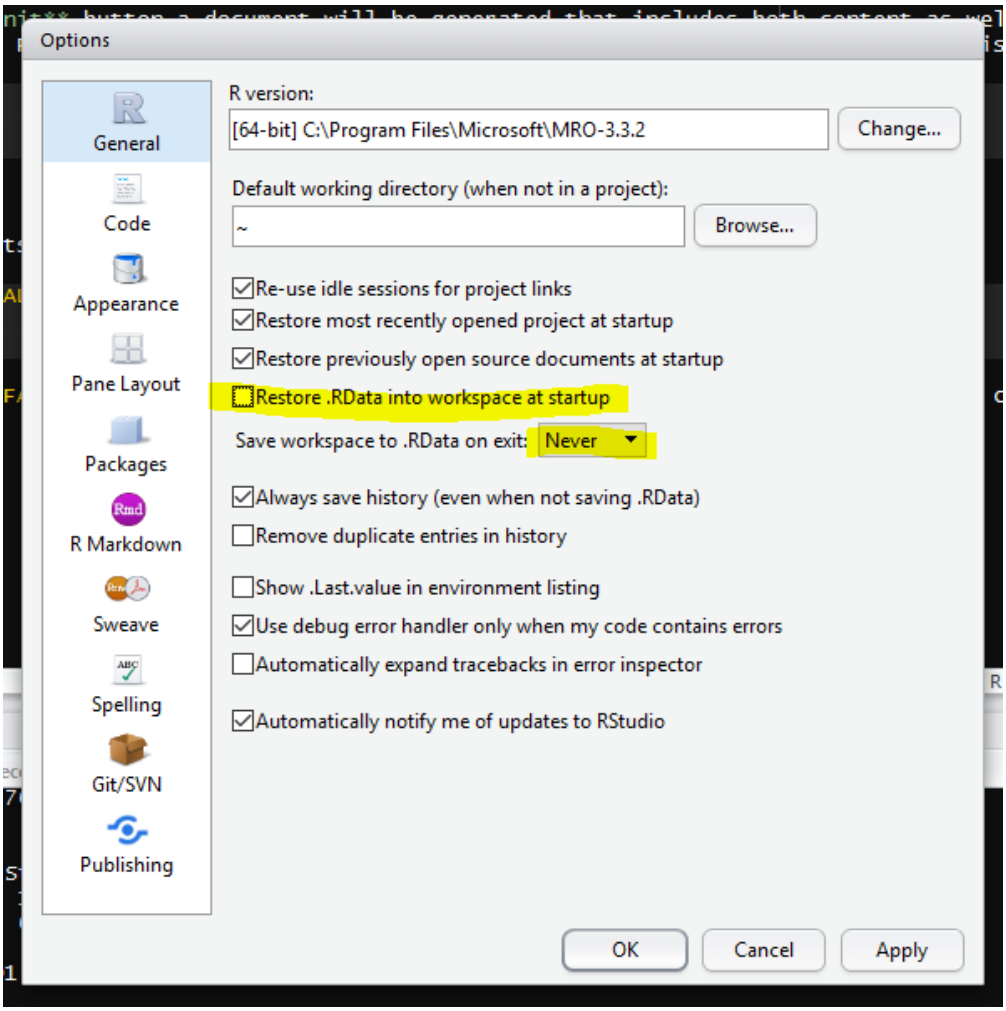


Figure 1.3: Don't save your workspace, save your script instead.

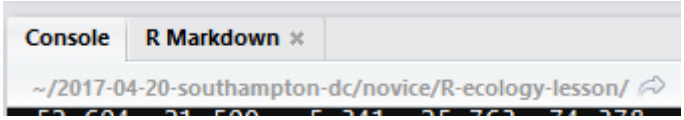


Figure 1.4: (ref:working-dir)

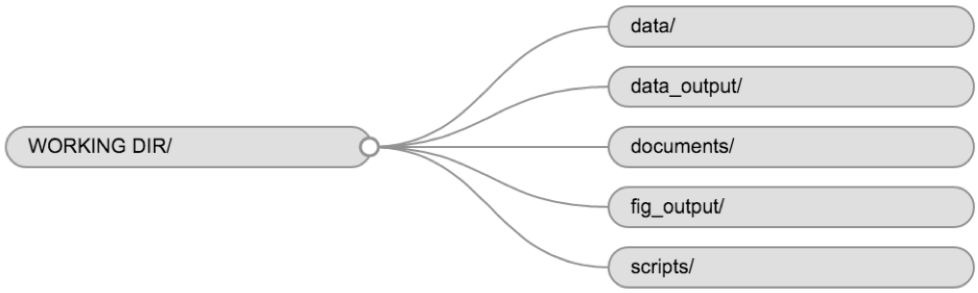


Figure 1.5: A typical directory structure

It is good practice to keep a set of related data, analyses, and text self-contained in a single folder, called the **working directory**. All of the scripts within this folder can then use *relative paths* to files that indicate where inside the project a file is located (as opposed to absolute paths, which point to where a file is on a specific computer). Working this way makes it a lot easier to move your project around on your computer and share it with others without worrying about whether or not the underlying scripts will still work.

1.5 R projects

RStudio also has a facility to keep all files associated with a particular analysis together called a project.

Creating a project creates a working directory for you and also remembers its location (allowing you to quickly navigate to it) and optionally preserves custom settings and open files to make it easier to resume work after a break.

Below, we will go through the steps for creating an “R Project”:

- Start R Studio (presentation of R Studio -below- should happen here)

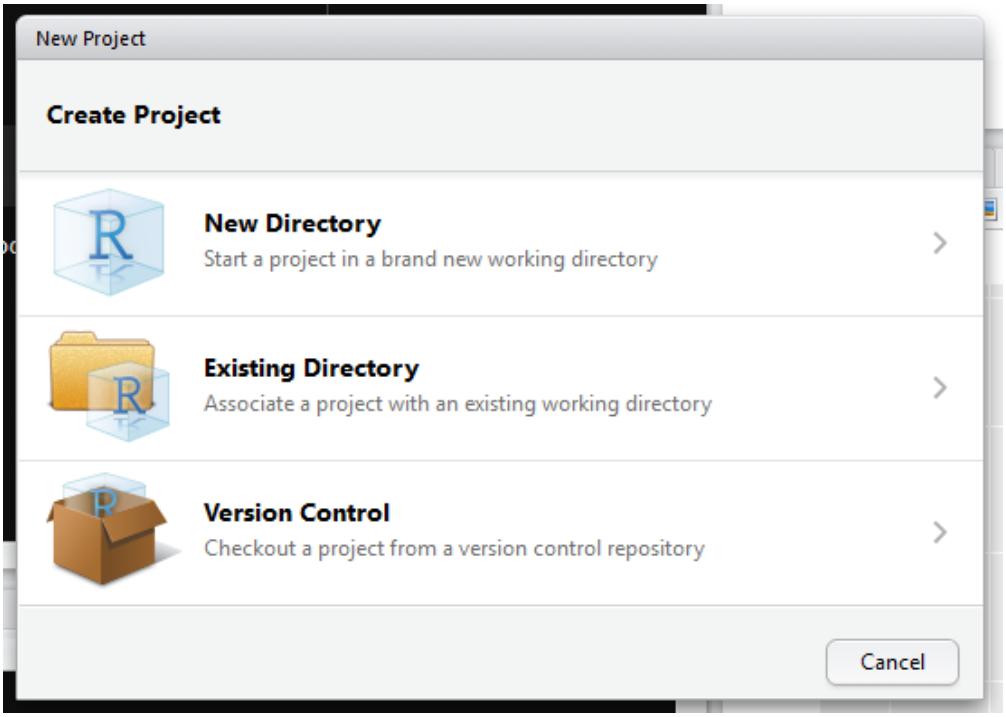


Figure 1.6: Creating a R project

- Under the `File` menu, click on `New project`, choose `New directory`, then `Empty project`
- Enter a name for this new folder (or "directory", in computer science), and choose a convenient location for it. This will be your **working directory** for the rest of the day (e.g., `~/bspr-workshop`)
- Click on "Create project"
- Under the `Files` tab on the right of the screen, click on `New Folder` and create a folder named `data` within your newly created working directory. (e.g., `~/bspr-workshopdata`)
- Create a new R script (File > New File > R script) and save it in your working directory (e.g. `bspr-workshop-script.R`)

1.6 Naming things

Jenny Bryan has three principles for naming things that are well worth remembering.

When you names something, a file or an object, ideally it should be:

1. Machine readable (no whitespace, punctuation, upper AND lowercase. . .)
2. Human readable (makes sense in 6 months or 2 years time)
3. Plays well with default ordering (numerical or date order)

1.7 Seeking help

If you need help with a specific R function, let's say `barplot()`, you can type:

```
?barplot
```

If you can't find what you are looking for, you can use the rdocumention.org website that searches through the help files across all packages available.

A Google or internet search "R <task>" will often either send you to the appropriate package documentation or a helpful forum question that someone else already asked, such as Stack Overflow or the RStudio Community.

1.7.1 Asking for help

As well as knowing where to ask, the key to get help from someone is for them to grasp your problem rapidly. You should make it as easy as possible to pinpoint where the issue might be.

Try to use the correct words to describe your problem. For instance, a package is not the same thing as a library. Most people will understand what you meant, but others have really strong feelings about the difference in meaning. The key point is that it can make things confusing for people trying to help you. Be as precise as possible when describing your problem.

If possible, try to reduce what doesn't work to a simple *reproducible example* otherwise known as a *reprex*.

For more information on how to write a reproducible example see this article.

Chapter 2

Getting started in R and the tidyverse

Functions are a way to automate common tasks and R comes with a set of functions called the `base` package. We will be using some `base` functions in Chapter 5, but to introduce the concept of using functions we'll begin with the `tidyverse`.

2.1 The tidyverse and tidy data

The tidyverse (Wickham, 2017) is *"an opinionated collection of R packages designed for data science"* .

Tidyverse packages contain functions that *"share an underlying design philosophy, grammar, and data structures."* It's this philiosophy that makes tidyverse functions and packages relatively easy to learn and use.

Tidy data follows three principals for tabular data as proposed in the Tidy Data paper http://www.jstatsoft.org/v59/i10/paper :

1. Every variable has its own column.
2. Every observation has its own row.
3. Each value has its own cell.

	A	C	D	E	F	G	
1	Peptide	Mass	Length	ppm	m/z	RT	I
2	GEPRFIAVGYVDDTQFVRFDSDAASPR	3014.452	27	0.6	754.6208	73.51	
3	GEPHFIAVGYVDDTQFVRFDSDAASPR	2995.41	27	0.1	749.8598	73.47	
4	GEPRFIAVGYVDDTQFVRFDSDAASQR	3045.458	27	2.9	762.374	72.35	
5	GSHSLRYFSTAVSRPGR	1876.966	17	0.3	626.6627	23.45	

Figure 2.1: An example of tidy proteomics data

If our table was proteomics data then, we might have a set of variables such as the peptide sequence, mass or length observed for a number of peptides. Therefore each peptide would have a row with columns for peptide sequence, mass and length with the value for each variable in separate cells, as seen in Figure 2.1.

Often much of the work in any data analysis is getting our data into a tidy form.

We can't do everything in the tidyverse, and everything we can do in the tidyverse can be done in what is called base R or other packages, but the motivation behind the tidyverse is to ease the pain of data manipulation.

With this in mind, the two tasks we are most likely to want to do in data science are:

1. Visualise our data
2. Automate our processes.

Taking our cue from R4DS let's try an example.

2.2 Data visualisation

The `ggplot2` package implements the *grammer of graphics*, for describing and building graphs.

The motivation here is twofold:

1. To begin to grasp the grammar of graphics approach to creating plots. This will be our first example of automating a task using a function.

2. To demonstrate how plotting is often the most useful thing we can do when trying to understand our data.

We'll use the `mpg` dataset that comes with the tidyverse to examine the question *do cars with big engines use more fuel than cars with small engines?*

Try `?mpg` to learn more about the data.

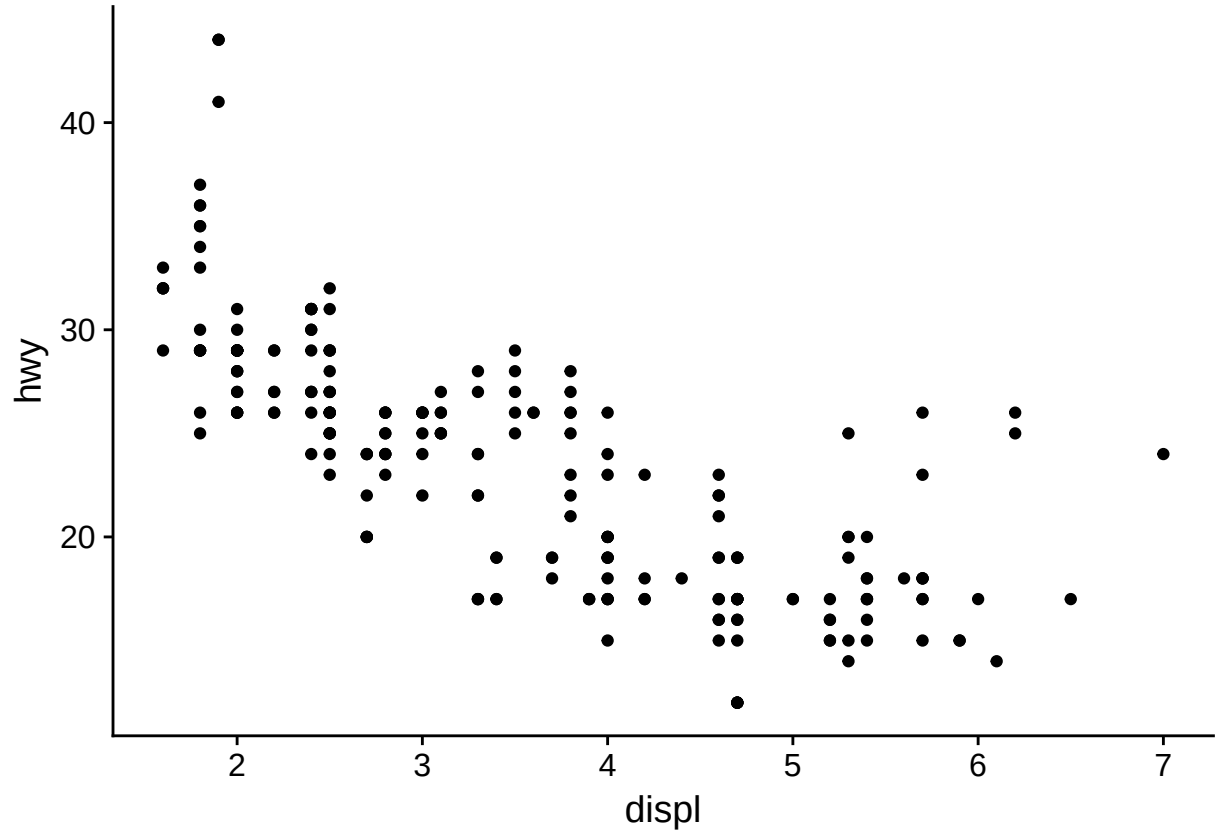
1. Engine size in litres is in the `displ` column.
2. Fuel efficiency on the highway in miles per gallon is given in the `hwy` column.

To create a plot of engine size `displ` (x-axis) against fuel efficiency `hwy` (y-axis) we do the following:

1. Use the `ggplot()` function to create an empty graph.
2. Provide ggplot with a first input or **argument** of the data (here `mpg`).
3. Then we follow the ggplot function with a + sign to indicate we are going to add more code, followed by a `geom_point()` function to add a layer of points mapping some aesthetics for the x and y axes.
4. Mapping is always paired to aesthetics `aes()`. An aesthetic is a visual property of the objects in your plot, such a point size, shape or point colour.

Therefore to plot engine size (x-axis) against fuel efficiency (y-axis) we use the following code:

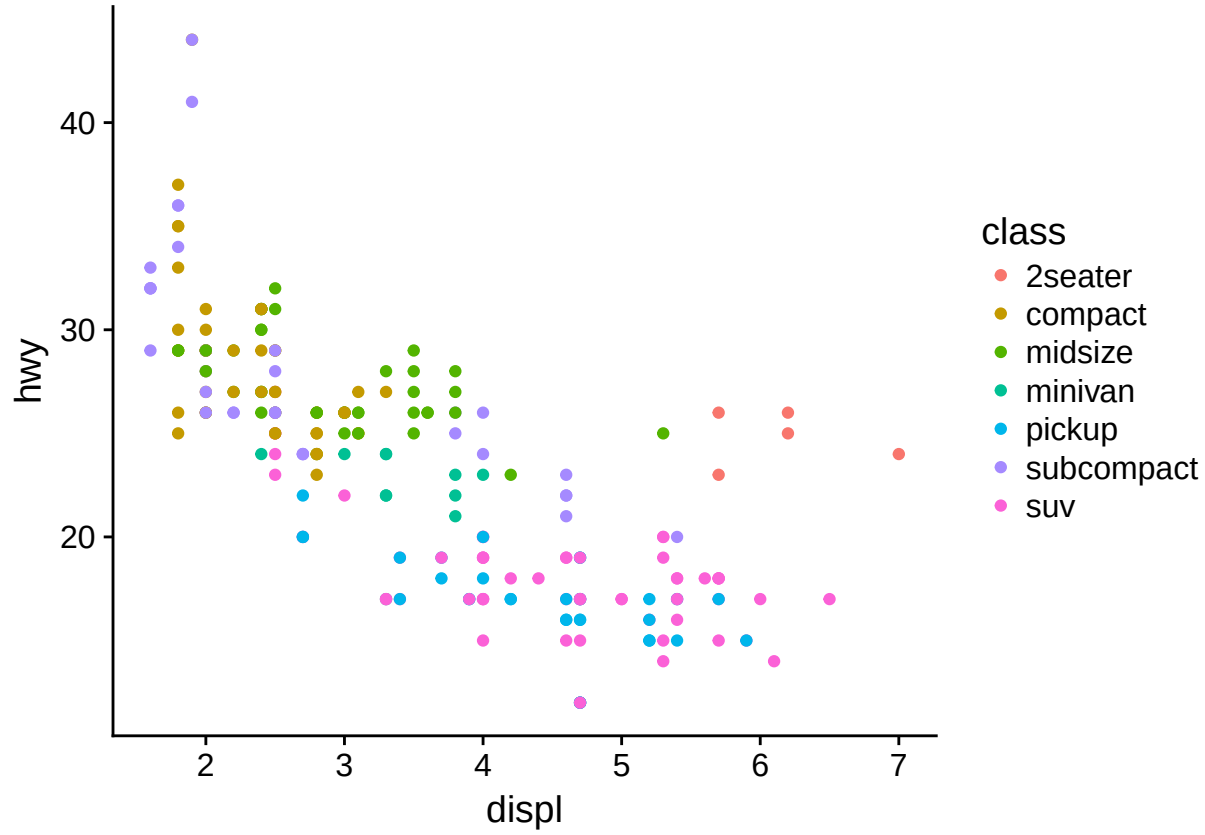
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```



This plot shows a negative relationship between engine size and fuel efficiency.

Now try extending this code to include to add a `colour` aesthetic to the the `aes()` function, let `colour = class`, `class` being the veichle type. This should create a plot with as before but with the points coloured according to the viechle type to expand our understanding.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, colour = class))
```



Now we can see that as we might expect, bigger cars such as SUVs tend to have bigger engines and are also less fuel efficient, but some smaller cars such as 2-seaters also have big engines and greater fuel efficiency. Hence we have a more nuanced view with this additional aesthetic.

Check out the ggplot2 documentation for all the aesthetic possibilities (and Google for examples): http://ggplot2.tidyverse.org/reference/

So now we have re-usable code snippet for generating plots in R:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

Concretely, in our first example `<DATA>` was `mpg`, the `<GEOM_FUNCTION>` was `geom_point()` and the arguments we supplies to map our aesthetics `<MAPPINGS>` were `x = displ, y = hwy`.

As we can use this code for any tidy data set, hopefully you are beginning to see how a small amount of code can do a lot.

2.2.1 Visualisations

Claus Wilke has written a very nice guide to visualising data using R called Fundamentals of Data Visualization.

2.3 Workflow basics

Let's run through the basics of working in R to conclude this chapter.

2.3.1 Assigning objects

Objects are just a way to store data inside the R environment. We create objects using the assignment operator <−:

```
mass_kg <- 55
```

Read this as *"mass_kg gets value 55"* in your head.

Using <− can be annoying to type, so use RStudio's keyboard short cut: Alt + - (the minus sign) to make life easier.

Many people ask why we use this assignment operator when we can use = instead?

Colin Fay had a Twitter thread on this subject, but the reason I favour most is that it provides clarity. The arrow points in the direction of the assigment (it is actually possible to assign in the other direction too) and it distinguishes between creating an object in the workspace and assigning a value inside a function.

Object name style is a matter of choice, but must start with a letter and can only contain letters, numbers, _ and . . We recommend using descriptive names and using _ between words. Some special symbols cannot be used in variable names, so watch out for those.

So here we've used the name to indicate its value represents a mass in kilograms. Look in your environment pane and you'll see the `mass_kg` object containing the (data) value 55.

We can inspect an object by typing it's name:

```
mass_kg
```

```
## [1] 55
```

What's wrong here?

```
mass_KG
```

```
Error: object 'mass_KG' not found
```

This error illustrates that typos matter, everything must be precise and `mass_KG` is not the same as `mass_kg`. `mass_KG` doesn't exist, hence the error.

2.3.2 Function anatomy

Functions in R are objects followed by parentheses, such as `library()`.

Functions have the form:

```
function_name(arg1 = val, arg2 = val2, ...)
```

The use of arguements or inputs allows us to generalise. That is to say not just do something in a specific case, but in many cases. For example not just make a scatter plot for the `mpg` dataset, but for any dataset of observations that can be plotted pairwise.

Let's use `seq()` to create a **seq**uence of numbers, and at the same time practice tab completion.

Start typing `se` in the console and you should see a list of functions appear, add `q` to shorten the list, then use the up and down arrow to highlight the function of interest `seq()` and hit Tab to select.

RStudio puts the cursor between the parentheses to prompt us to enter some arguments. Here we'll use `1` as the start and `10` as the end:

```
seq(1,10)
```

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

If we left off a parentheses to close the function, then when we hit enter we'll see a + indicating RStudio is expecting further code. We either add the missing part or press Escape to cancel the code.

Let's call a function and make an assignment at the same time. Here we'll use the base R function `seq()` which takes three arguments: `from`, `to` and `by`.

Read the following code as *"make an object called my_sequence that stores a sequence of numbers from 2 to 20 by intervals of 2*.

```
my_sequence <- seq(2,20,2)
```

This time nothing was returned to the console, but we now have an object called `my_sequence` in our environment.

Can you remember how to inspect it?

If we want to subset elements of `my_sequence` we use square brackets `[]`.

For example element five would be subset by:

```
my_sequence[5]
```

```
## [1] 10
```

Here the number five is the index of the vector, not the value of the fifth element. The value of the fifth element is 10.

And returning multiple elements uses a colon `:`, like so

```
my_sequence[5:8]
```

```
## [1] 10 12 14 16
```

2.3.3 Atomic vectors

We actually made an atomic vector already when we made `my_sequence`. We made a a one dimensional group of numbers, in a sequence from two to twenty.

We're not going to be working much with atomic vectors in this workshop, but to make you aware of how R stores data, atomic vector types are:

- Doubles: regular numbers, +ve or -ve and with or without decimal places. AKA numerics.
- Integers: whole numbers, specified with an upper-case L, e.g. `int <- 2L`
- Characters: Strings of text
- Logicals: these store `TRUE`s and `FALSE`s which are useful for comparisons.
- Complex: this would be a vector of numbers with imaginary terms.
- Raw: these vectors store raw bytes of data.

Let's make a character vector and check the type:

```
cards <- c("ace", "king", "queen", "jack", "ten")


cards
```

```
## [1] "ace"   "king"  "queen" "jack"  "ten"
```

```
typeof(cards)
```

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

2.3.4 Attributes

An attribute is a piece of information you can attach to an object, such as names or dimensions. Attributes such as dimensions are added when we create an object, but others such as names can be added.

Let's look at the `mpg` data frame dimensions:

```
# mpg has 234 rows (observations) and 11 columns (variables)
dim(mpg)
```

```
## [1] 234  11
```

2.3.5 Factors

Factors are Rs way of storing categorical information such as eye colour or car type. A factor is something that can only have certain values, and can be ordered (such as `low`,`medium`,`high`) or unordered such as types of fruit.

Factors are useful as they code string variables such as "red" or "blue" to integer values e.g. 1 and 2, which can be used in statistical models and when plotting, but they are confusing as they look like strings.

Factors look like strings, but behave like integers.

Historically R converts strings to factors when we load and create data, but it's often not what we want as a default. Fortunately, in the tidyverse strings are not treated as factors by default.

2.3.6 Lists

Lists also group data into one dimensional sets of data. The difference being that list group objects instead of individual values, such as several atomic vectors.

For example, let's make a list containing a vector of numbers and a character vector

```
list_1 <- list(1:110,"R")


list_1
```

```
## [[1]]
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
##  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
##  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102
## [103] 103 104 105 106 107 108 109 110
##
## [[2]]
## [1] "R"
```

Note the double brackets to indicate the list elements, i.e. element one is the vector of numbers and element two is a vector of a single character.

We won't be working with lists in this workshop, but they are a flexible way to store data of different types in R.

Accessing list elements uses double square brackets syntax, for example `list_1[[1]]` would return the first vector in our list.

And to access the first element in the first vector would combine double and single square brackets like so: `list_1[[1]][1]`.

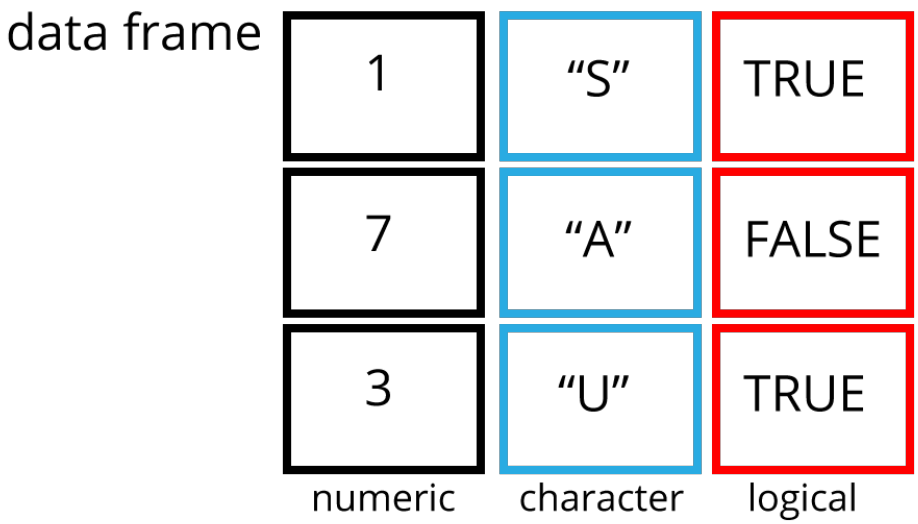


Figure 2.2: An example data frame `df`.

Don't worry if you find this confusing, everyone does when they first start with R.

2.3.7 Matrices and arrays

Matrices store values in a two dimensional array, whilst arrays can have n dimensions. We won't be using these either, but they are also valid R objects.

2.3.8 Data frames

Data frames are two dimensional versions of lists, and this is form of storing data we are going to be using. In a data frame each atomic vector type becomes a column, and a data frame is formed by columns of vectors of the same length. Each column element must be of the same type, but the column types can vary.

Figure 2.2 shows an example data frame we'll refer to as saved as the object `df` consisting of three rows and three columns. Each column is a different atomic data type of the same length.

Packages in the tidyverse create a modified form of data frame called a tibble. You can

read about tibbles here. One advantage of tibbles is that they don't default to treating strings as factors. We deal with modifying data frames when we work with our example data set.

Sub-setting data frames can also be done with square bracket syntax, but as we have both rows and columns, we need to provide index values for both row and column.

For example `df[1,2]` means **return the value of `df` row 1, column 2**. This corresponds with the value `A`.

We can also use the colon operator to choose several rows or columns, and by leaving the row or column blank we return all rows or all columns.

```
# Subset rows 1 and 2 of column 1
df[1:2,1]


# Subset all rows of column 3
df[,3]
```

Again don't worry too much about this for now, we won't be doing to much of this in this lesson, but it's important to be aware of the basic syntax.

2.4 Learning more R

There are many places to start, but swirl can teach you interactively, and at your own pace in RStudio.

Just follow the instructions via this link: http://swirlstats.com/students.html

Hands-On Programming with R by Garrett Grolemund is another great resource for learning R.

Plus all the tidyverse links.

Chapter 3

Creating scripts and importing data

Our analysis is of an example data set of observations for 7702 proteins from cells in three control experiments and three treatment experiments. The observations are signal intensity measurements from the mass spectrometer. These intensities relate the concentration of protein observed in each experiment and under each condition.

We consider raw data as the data as we receive it. This doesn't mean it hasn't be processed in some way, it just means it hasn't been processed by us. Generally speaking we don't change the raw data file, what we do is import it and create an object in R which we then transform.

So let's understand how to import some data.

3.1 Some definitions

- **Importing** means getting data into our R environment by creating an object that we can then manipulate. The raw data file remains unchanged.
- **Inspecting** means looking at the dataset to understand what it contains.
- **Tidying** refers to getting data into a consistent format that makes it easy to use in later steps.

3.1.1 Rectangular data and flat formats

Two further things to note:

1. Here we are only considering **rectangular data**, the sort that comes in rows and columns such as in a spreadsheet. Lots of our data types exist, such as images, but can also be handled by R. As mentioned in 3.5.1 genomic data in particular has led to a project called Bioconductor for the development of analysis tools primarily in R, many of which deal with non-rectangular data, but this is beyond the scope here.

2. **Flat formats** are files that only contain plain text, with each line representing a set of observations and the variables separated by delimiters such as tabs, commas or spaces. Therefore there aren't multiple tables such as we'd get in an Excel file, or meta-data such as the colour highlighting of a cell in an Excel file. The advantages of flat files is that they can be opened and used by many different computing languages or programs. So unless there is a good reason not to use a flat format, and there are good reasons, they are the best way to store data in many situations.

3.2 Using scripts

Using the console is useful, but as we build up a workflow, that is to say, writing code to:

- load packages
- load data
- explore the data
- and output some results

Then it's much more useful to contain this in a script: a document of our code.

Why? When we write and save our code in scripts, we can re-use it, share it or edit it. But **most importantly a script is a record**.

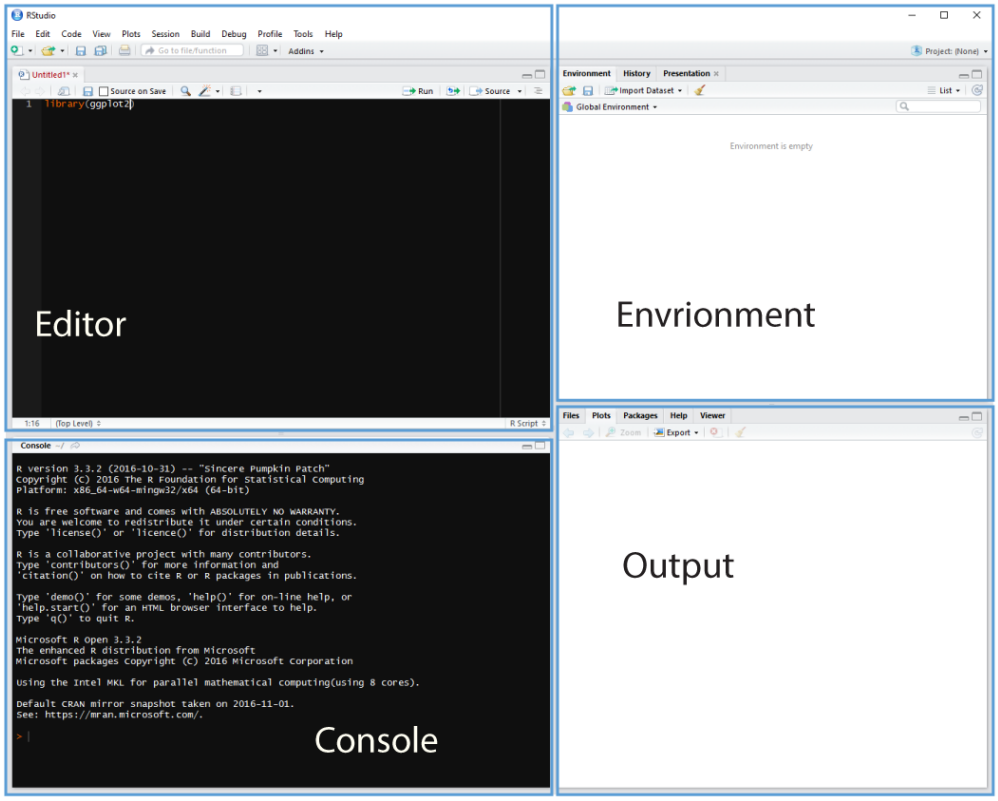


Figure 3.1: Rstudio with the script editor pane open.

Cmd/Ctrl + Shift + N will open a new script file up and you should see something like Figure 3.1 with the script editor pane open:

3.3 Running code

We can run a highlighted portion of code in your script if you click the Run button at the top of the scripts pane as shown in Figure 3.2.

You can run the entire script by clicking the Source button.

Or we can run chunks of code if we split our script into sections, see below.

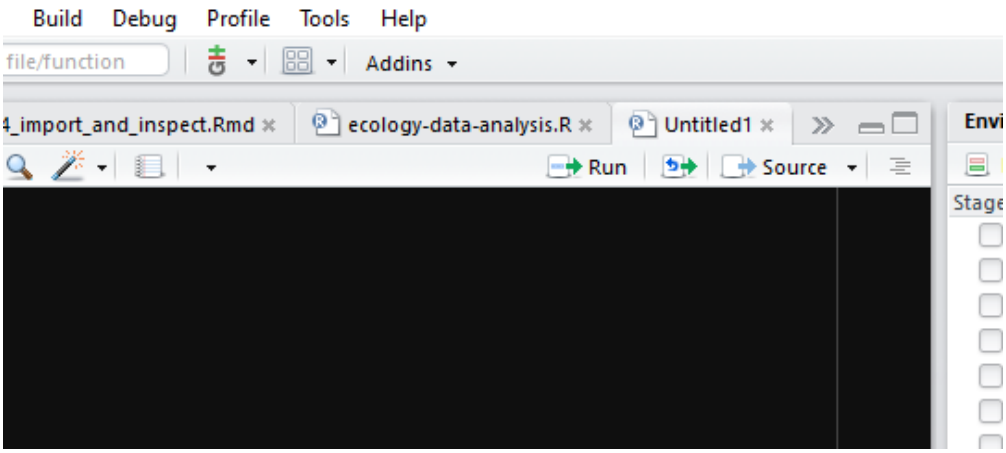


Figure 3.2: Scripts can be run by clicking the Source button.

3.4 Creating a R script

We first need to create a script that will form the basis of our analysis.

Go to the file menu and select New Files > R script. This should open the script editor pane.

Now let's save the script, by going to File > Save and we should find ourselves prompted to save the script in our Project Directory.

Following the advice about naming things we can create a new R script called `01-bspr-workshop-july-2018`.

This name is machine readable (no spaces or special characters), human readable, and works well with default ordering by beginning with `01`.

3.5 Setting up our environment

At the head of our script it's common to put a title, the name of the author and the date, and any other useful information. This is created as comments using the # at the start of each line.

It's then usual to follow this by code to load the packages we need into our our R

environment using the `library()` function and providing the name of the package we wish to load. Packages are collections of R functions.

Often we break the code up into regions by adding dashes (or equals symbols) to the comment line. This enables us to run chunks of the script separately from running the whole script when using our code.

Here is a typical head for a script:

```
# My workshop script
# 7th July 2018
# Alistair Bailey

# Load packages ----------------------------------------------------------------
library(plyr)
library(tidyverse)
library(gplots)
library(pheatmap)
```

3.5.1 Bioconductor

As an aside there are many proteomics specific R packages, these are generally found through Bioconductor which is a project that was initiated in 2001 to create tools for the analysis of high-throughput genomic data, but also includes other 'omics data tools (Gentleman et al., 2004, Huber et al. (2015)).

Exploring Bioconductor is beyond our scope here, but well worth exploring for manipulation and analysis of raw data formats such as mzxml files.

3.6 Importing data

Assuming our data is in a flat format, we can import it into our environment using the tidyverse `readr` package.

If our data was an excel file, we can use the tidyverse `readxl` package to import the data, but it will remove any meta-data and each table in the excel file will become a separate R object as per tidy data principles.

For the purposes of this workshop we have a `csv` (comma separated variable) file.

If you haven't done so already Click here to download the example data and save it to our project directory. Check the `Files` pane to see it's there.

We then import data and assign it to an object we'll call `data` like so:

```
# Import example data -------------------------------------------------------
# Import the example data with read_csv from the readr package
dat <- readr::read_csv("data/070718-proteomics-example-data.csv")
```

```
## Parsed with column specification:
## cols(
##   protein_accession = col_character(),
##   protein_description = col_character(),
##   control_1 = col_double(),
##   control_2 = col_double(),
##   control_3 = col_double(),
##   treatment_1 = col_double(),
##   treatment_2 = col_double(),
##   treatment_3 = col_double()
## )
```

3.7 Exploring the data

3.7.1 `glimpse`, `head` and `str`

The first thing to do with any data set is to actually look at it. Here are four ways to have look at the data in the `Console`: calling the object directly, `glimpse`, `head` and `str`.

1. We can just call the object and return it to the `Console`, which may or may not be useful depending on the size and type of object we call.

2 .`glimpse` is a tidyverse function that tries to show us as much data in a data.frame or tibble as possible, telling us the atomic types of data in the table, the number of observations and the number of variables, and importantly shows all the column variable names by transposing the table.

3. `head` is a base function that shows us the 6 lines of a R object by default.

4. `str` is a base function that show the structure of a R object, so it provides a lot of information, but is not so easy to read.

The outputs for these four functions is shown below:

```
# call object
dat
```

```
## # A tibble: 7,702 x 8
##    protein_accession protein_description     control_1 control_2 control_3
##    <chr>             <chr>                       <dbl>     <dbl>     <dbl>
##  1 VATA_HUMAN_P38606 V-type proton ATPase c~     0.811     0.858     1.04
##  2 RL35A_HUMAN_P180~ 60S ribosomal protein ~     0.367     0.385     0.409
##  3 MYH10_HUMAN_P355~ Myosin-10 OS=Homo sapi~     2.98      4.62      2.87
##  4 RHOG_HUMAN_P84095 Rho-related GTP-bindin~     0.142     0.224     0.128
```

```
##  5 PSA1_HUMAN_P25786 Proteasome subunit alp~     1.07      0.945     0.803
##  6 PRDX5_HUMAN_P300~ Peroxiredoxin-5_ mitoc~     0.566     0.540     0.488
##  7 ACLY_HUMAN_P53396 ATP-citrate synthase O~     5.00      4.22      5.03
##  8 VDAC2_HUMAN_P458~ Voltage-dependent anio~     1.35      1.33      1.14
##  9 LRC47_HUMAN_Q8N1~ Leucine-rich repeat-co~     0.927     0.770     1.17
## 10 CH60_HUMAN_P10809 60 kDa heat shock prot~     9.45      8.41     10.4
## # ... with 7,692 more rows, and 3 more variables: treatment_1 <dbl>,
## #   treatment_2 <dbl>, treatment_3 <dbl>
```

```
# tidyverse glimpse function
glimpse(dat)
```

```
## Observations: 7,702
## Variables: 8
## $ protein_accession   <chr> "VATA_HUMAN_P38606", "RL35A_HUMAN_P18077",...
## $ protein_description <chr> "V-type proton ATPase catalytic subunit A ...
## $ control_1           <dbl> 0.8114, 0.3672, 2.9815, 0.1424, 1.0748, 0....
## $ control_2           <dbl> 0.8575, 0.3853, 4.6176, 0.2238, 0.9451, 0....
## $ control_3           <dbl> 1.0381, 0.4091, 2.8709, 0.1281, 0.8032, 0....
## $ treatment_1         <dbl> 0.6448, 0.4109, 7.1670, 0.1643, 0.7884, 0....
## $ treatment_2         <dbl> 0.7190, 0.4634, 2.0052, 0.2466, 0.8798, 1....
## $ treatment_3         <dbl> 0.4805, 0.3561, 0.8995, 0.1268, 0.7631, 0....
```

```
# head function
head(dat)
```

```
## # A tibble: 6 x 8
##   protein_accession protein_description      control_1 control_2 control_3
##   <chr>             <chr>                        <dbl>     <dbl>     <dbl>
## 1 VATA_HUMAN_P38606 V-type proton ATPase ca~     0.811     0.858     1.04
## 2 RL35A_HUMAN_P180~ 60S ribosomal protein L~     0.367     0.385     0.409
```

```
## 3 MYH10_HUMAN_P355~ Myosin-10 OS=Homo sapie~     2.98      4.62      2.87
## 4 RHOG_HUMAN_P84095 Rho-related GTP-binding~     0.142     0.224     0.128
## 5 PSA1_HUMAN_P25786 Proteasome subunit alph~     1.07      0.945     0.803
## 6 PRDX5_HUMAN_P300~ Peroxiredoxin-5_ mitoch~     0.566     0.540     0.488
## # ... with 3 more variables: treatment_1 <dbl>, treatment_2 <dbl>,
## #   treatment_3 <dbl>
```

```
# str function
str(dat)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    7702 obs. of  8 variables:
##  $ protein_accession  : chr  "VATA_HUMAN_P38606" "RL35A_HUMAN_P18077" "MYH10_HUMAN_P3
##  $ protein_description: chr  "V-type proton ATPase catalytic subunit A OS=Homo sapien
##  $ control_1          : num  0.811 0.367 2.982 0.142 1.075 ...
##  $ control_2          : num  0.858 0.385 4.618 0.224 0.945 ...
##  $ control_3          : num  1.038 0.409 2.871 0.128 0.803 ...
##  $ treatment_1        : num  0.645 0.411 7.167 0.164 0.788 ...
##  $ treatment_2        : num  0.719 0.463 2.005 0.247 0.88 ...
##  $ treatment_3        : num  0.48 0.356 0.899 0.127 0.763 ...
##  - attr(*, "spec")=List of 2
##   ..$ cols   :List of 8
##   .. ..$ protein_accession  : list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ protein_description: list()
##   .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
##   .. ..$ control_1          : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   .. ..$ control_2          : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   .. ..$ control_3          : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
```

```
##   .. ..$ treatment_1        : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   .. ..$ treatment_2        : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   .. ..$ treatment_3        : list()
##   .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##   ..$ default: list()
##   .. ..- attr(*, "class")= chr  "collector_guess" "collector"
##   ..- attr(*, "class")= chr "col_spec"
```

To see the data in a *spreadsheet* fashion use `View(dat)`, note the capital V and a new tab will open. This can also be launched from the `Environment` tab by clicking on `dat`.

Although this provides us with some useful information, such as the number of observations and variables, to understand more plotting the data will be helpful as we'll see in Section 5.3.

3.7.2 Summary statisitics

Another useful way to quickly get a sense of the data is to use the `summary` function, which will return summary of the spread of the data and importantly if there are missing values. We can see immediately below that the experimental replicates have different distributions, and missing values that we need to deal with in Chapter 5.

```
summary(dat)
```

```
##  protein_accession  protein_description   control_1          control_2
##  Length:7702        Length:7702         Min.   : 0.001   Min.   : 0.000
##  Class :character   Class :character    1st Qu.: 0.143   1st Qu.: 0.132
##  Mode  :character   Mode  :character    Median : 0.345   Median : 0.322
##                                         Mean   : 0.933   Mean   : 0.845
```

```
##                                                3rd Qu.: 0.959   3rd Qu.: 0.845
##                                                Max.   :31.944   Max.   :31.697
##                                                NA's   :4888     NA's   :4828
##    control_3       treatment_1      treatment_2      treatment_3
##  Min.   : 0.001   Min.   : 0.000   Min.   : 0.002   Min.   : 0.002
##  1st Qu.: 0.149   1st Qu.: 0.112   1st Qu.: 0.135   1st Qu.: 0.101
##  Median : 0.388   Median : 0.286   Median : 0.319   Median : 0.254
##  Mean   : 0.977   Mean   : 0.795   Mean   : 0.856   Mean   : 0.675
##  3rd Qu.: 0.999   3rd Qu.: 0.780   3rd Qu.: 0.880   3rd Qu.: 0.682
##  Max.   :31.320   Max.   :41.686   Max.   :28.234   Max.   :21.428
##  NA's   :5087     NA's   :4739     NA's   :4902     NA's   :5074
```

Chapter 4

`dplyr` verbs and piping

A core package in the tidyverse is `dplyr` for transforming data, which is often used in conjunction with the `magrittr` package that allows us to pipe multiple operations together.

The R4DS dplyr chapter is here and for magrittr here.

The figures in this chapter we made for use with an ecological dataset on rodent surveys, but the principles they illustrate are generic and show the use of each function with or without the use of a pipe.

From R4DS:

"*All `dplyr` verbs work similarly:*

1. The first argument is a data frame.

2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).

3. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result."

4.1 Pipes

A pipe in R looks like this %>% and allows us to send the output of one operation into another. This saves time and space, and can make our code easier to read.

For example we can pipe the output of calling the `dat` object into the `glimpse` function like so:

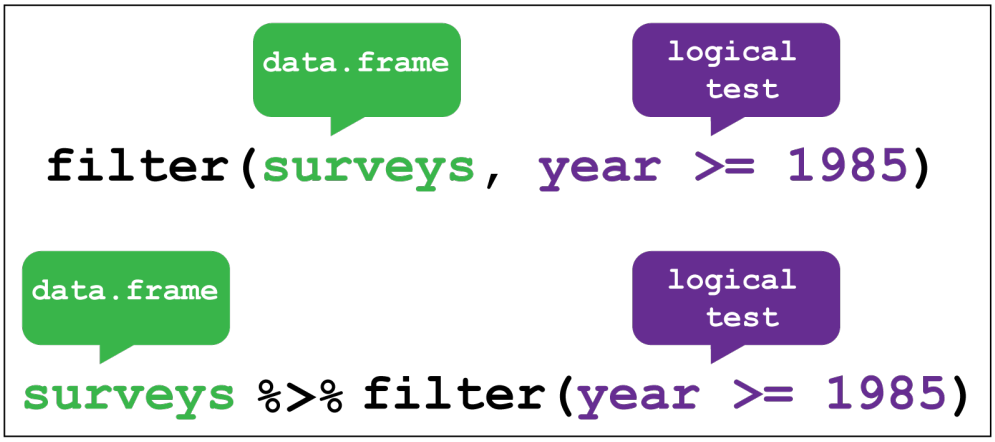
```
dat %>% glimpse()
```

```
## Observations: 7,702
## Variables: 8
## $ protein_accession   <chr> "VATA_HUMAN_P38606", "RL35A_HUMAN_P18077",...
## $ protein_description <chr> "V-type proton ATPase catalytic subunit A ...
## $ control_1           <dbl> 0.8114, 0.3672, 2.9815, 0.1424, 1.0748, 0....
## $ control_2           <dbl> 0.8575, 0.3853, 4.6176, 0.2238, 0.9451, 0....
## $ control_3           <dbl> 1.0381, 0.4091, 2.8709, 0.1281, 0.8032, 0....
## $ treatment_1         <dbl> 0.6448, 0.4109, 7.1670, 0.1643, 0.7884, 0....
## $ treatment_2         <dbl> 0.7190, 0.4634, 2.0052, 0.2466, 0.8798, 1....
## $ treatment_3         <dbl> 0.4805, 0.3561, 0.8995, 0.1268, 0.7631, 0....
```

This becomes even more useful when we combine pipes with `dplyr` functions.

4.2 Filter rows

The `filter` function enables us to filter the rows of a data frame according to a logical test (one that is `TRUE` or `FALSE`). Here it filters rows in the surveys data where the year variable is greater or equal to 1985.



Let's try this with `dat` to filter the rows for proteins in `control_1` and `control_2` experiments where the observations are greater than 20:

```
dat %>% filter(control_1 > 20, control_2 > 20)
```

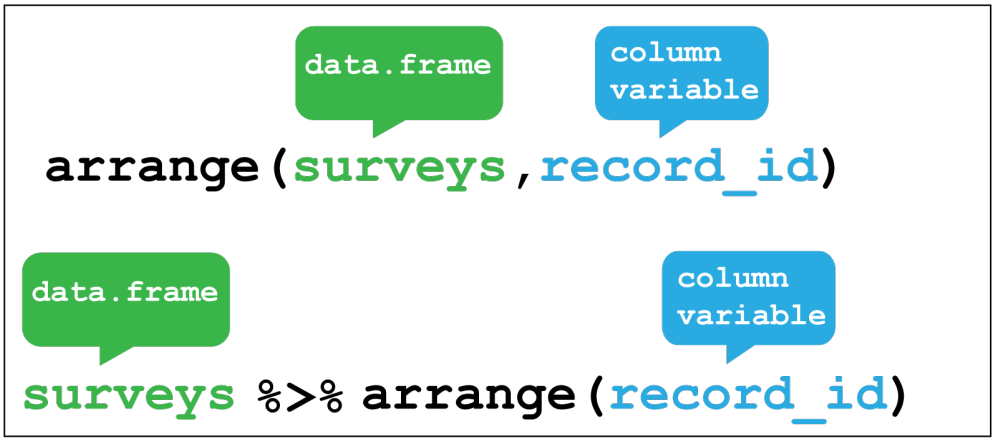
```
## # A tibble: 2 x 8
##   protein_accession   protein_description    control_1 control_2 control_3
##   <chr>               <chr>                      <dbl>     <dbl>     <dbl>
## 1 MYH9_HUMAN_P35579   Myosin-9 OS=Homo sapi~      29.2      31.7      24.6
## 2 A0A087WWY3_HUMAN_A~ Filamin-A OS=Homo sap~      31.9      27.8      31.3
## # ... with 3 more variables: treatment_1 <dbl>, treatment_2 <dbl>,
## #   treatment_3 <dbl>
```

Filtering is done with the following operators >,<,>=,<=, != (not equal) and == for equal. Not the double equal sign.

4.3 Arrange rows

Arranging is similar to filter except that it changes the row order according to the columns in ascending order. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.

Here we arrange the surveys data according to the record identification number.



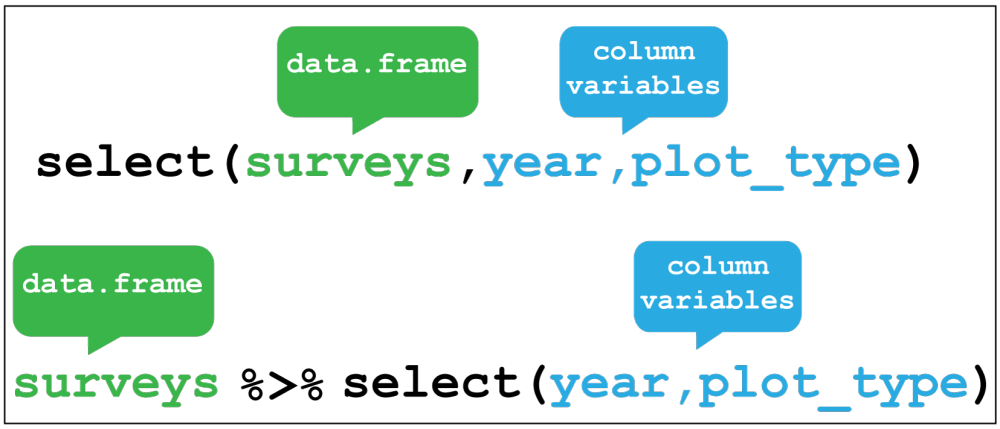
To try that with `dat` let's arrange the data according to `control_1`:

```
dat %>% arrange(control_1)
```

```
## # A tibble: 7,702 x 8
##    protein_accession protein_description     control_1 control_2 control_3
##    <chr>             <chr>                       <dbl>     <dbl>     <dbl>
##  1 PAL4G_HUMAN_P0DN~ Peptidyl-prolyl cis-tr~   0.001      0.0177    NA
##  2 E5RGV5_HUMAN_E5R~ Nucleolysin TIA-1 isof~   0.0011    NA          0.093
##  3 E5RJP4_HUMAN_E5R~ Glutamine--fructose-6-~   0.002     NA         NA
##  4 I3L3U1_HUMAN_I3L~ Myosin light chain 4 O~   0.00240   NA         NA
##  5 ENPLL_HUMAN_Q58F~ Putative endoplasmin-l~   0.0026    NA         NA
##  6 K1C15_HUMAN_P190~ Keratin_ type I cytosk~   0.00290    0.0615     0.122
##  7 B5ME44_HUMAN_B5M~ Outer dense fiber prot~   0.00290   NA         NA
##  8 PANK3_HUMAN_Q9H9~ Pantothenate kinase 3 ~   0.0033    NA         NA
##  9 RRS1_HUMAN_Q15050 Ribosome biogenesis re~   0.0035    NA         NA
## 10 NFL_HUMAN_P07196  Neurofilament light po~   0.0035     0.315      0.564
## # ... with 7,692 more rows, and 3 more variables: treatment_1 <dbl>,
## #   treatment_2 <dbl>, treatment_3 <dbl>
```

4.4 Select columns

Selecting is the verb we use to select columns of interest in the data. Here we select only the `year` and `plot_type` columns and discard the rest.



Let's use select with `dat` to drop the protein description and control experiments using negative indexing and keep everything else:

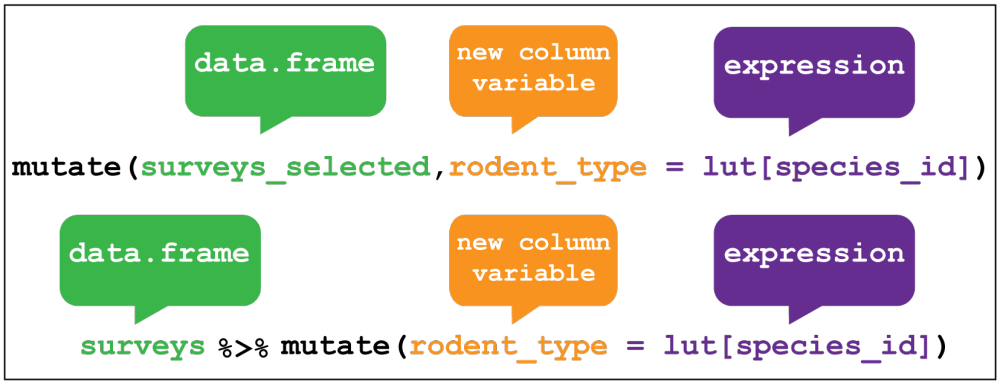
```
dat %>% select(-protein_description,-(control_1:control_3))
```

```
## # A tibble: 7,702 x 4
##    protein_accession  treatment_1 treatment_2 treatment_3
##    <chr>                    <dbl>       <dbl>       <dbl>
##  1 VATA_HUMAN_P38606        0.645       0.719       0.480
##  2 RL35A_HUMAN_P18077       0.411       0.463       0.356
##  3 MYH10_HUMAN_P35580       7.17        2.01        0.900
##  4 RHOG_HUMAN_P84095        0.164       0.247       0.127
##  5 PSA1_HUMAN_P25786        0.788       0.880       0.763
##  6 PRDX5_HUMAN_P30044       0.545       1.69        0.821
##  7 ACLY_HUMAN_P53396        4.67        5.01        3.57
##  8 VDAC2_HUMAN_P45880       1.01        1.04        0.904
##  9 LRC47_HUMAN_Q8N1G4       1.22        1.01        0.593
```

```
## 10 CH60_HUMAN_P10809        8.31        8.31        5.73
## # ... with 7,692 more rows
```

4.5 Create new variables

Creating new variables uses the `mutate` verb. Here I am creating a new variable called `rodent_type` that will create a new column containing the type of rodent observed in each row.



Let's create a new variable for `dat` called `prot_id` that use the `str_extract` function from the `stringr` package to take the last 6 characters of the `protein_accession` variable, the `".{6}$"` part is called a regular expression, to keep just the UNIPROT id part of the string.

We'll use select to drop the other variables except the protein accession afterwards via another pipe.

```
dat %>%
  group_by(protein_accession) %>%
  mutate(prot_id = str_extract(protein_accession,".{6}$")) %>%
  select(protein_accession, prot_id)
```

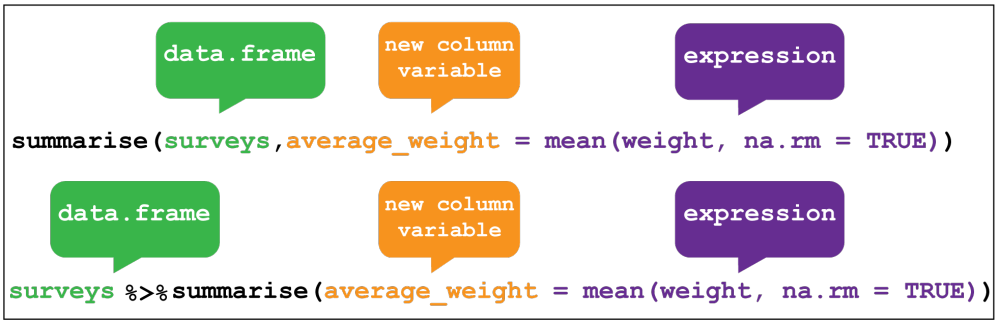
```
## # A tibble: 7,702 x 2
## # Groups:   protein_accession [7,702]
```

```
##    protein_accession  prot_id
##    <chr>              <chr>
##  1 VATA_HUMAN_P38606  P38606
##  2 RL35A_HUMAN_P18077 P18077
##  3 MYH10_HUMAN_P35580 P35580
##  4 RHOG_HUMAN_P84095  P84095
##  5 PSA1_HUMAN_P25786  P25786
##  6 PRDX5_HUMAN_P30044 P30044
##  7 ACLY_HUMAN_P53396  P53396
##  8 VDAC2_HUMAN_P45880 P45880
##  9 LRC47_HUMAN_Q8N1G4 Q8N1G4
## 10 CH60_HUMAN_P10809  P10809
## # ... with 7,692 more rows
```

4.6 Create grouped summaries

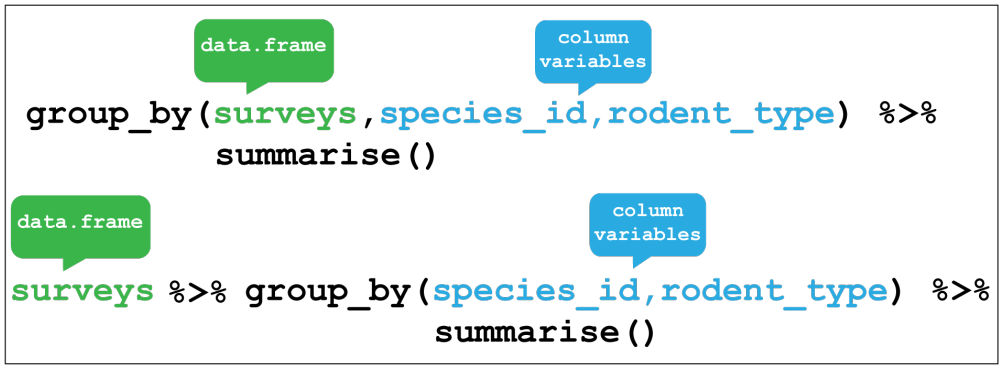
The last key verb is `summarise` which collapses a data frame into a single row.

For example, we could use it to find the average weight of all the animals surveyed in the surveys data using `mean()`. (Here the `na.rm = TRUE` argument is given to remove missing values from the data, otherwise R would return `NA` when trying to average.)



`summarise` is most useful when paired with `group_by` which defines the variables upon which we operate upon.

Here if we group by `species_id` and `rodent_type` together and then used `summarise` without any arguments we return these two variables only.



We'll use the `mtcars` dataset again to illustrate a grouped summary. Here I'll group according gearbox type `am` where 0 is automatic and 1 is manual. Then using summarise to calculate the mean miles per gallon, the tables is collapsed to two rows, one for each gearbox type.

```
# mtcars, am is gearbox, 0 = automatic, 1 = manual
mtcars %>%
  group_by(am) %>%
  summarise(mean_mpg = mean(mpg, na.rm = T))
```

```
## # A tibble: 2 x 2
##      am mean_mpg
##   <dbl>    <dbl>
## 1     0     17.1
## 2     1     24.4
```

We'll use `dplyr` and pipes in Chapter 5.

Chapter 5

Transforming and visualising proteomics data

Having imported our data set of observations for 7702 proteins from cells in three control experiments and three treatment experiments. Remember, the observations are signal intensity measurements from the mass spectrometer, and these intensities relate to the amount of protein in each experiment and under each condition.

Now we will transform the data to examine the effect of the treatment on the cellular proteome and visualise the output using a volcano plot and a heatmap. The hypothesis we are testing is that treatment changes the concentration of protein we observe.

A volcano plot is commonly used way of plotting changes in observed values on the x-axis against the likelihood of observing that change due to chance on the y-axis. Heatmaps are another way of visualising the relative (increase and decrease of) amounts of observed values.

5.1 Fold change and log-fold change

Fold changes are ratios, the ratio of say protein expression before and after treatment, where a value larger than 1 for a protein implies that protein expression was greater after

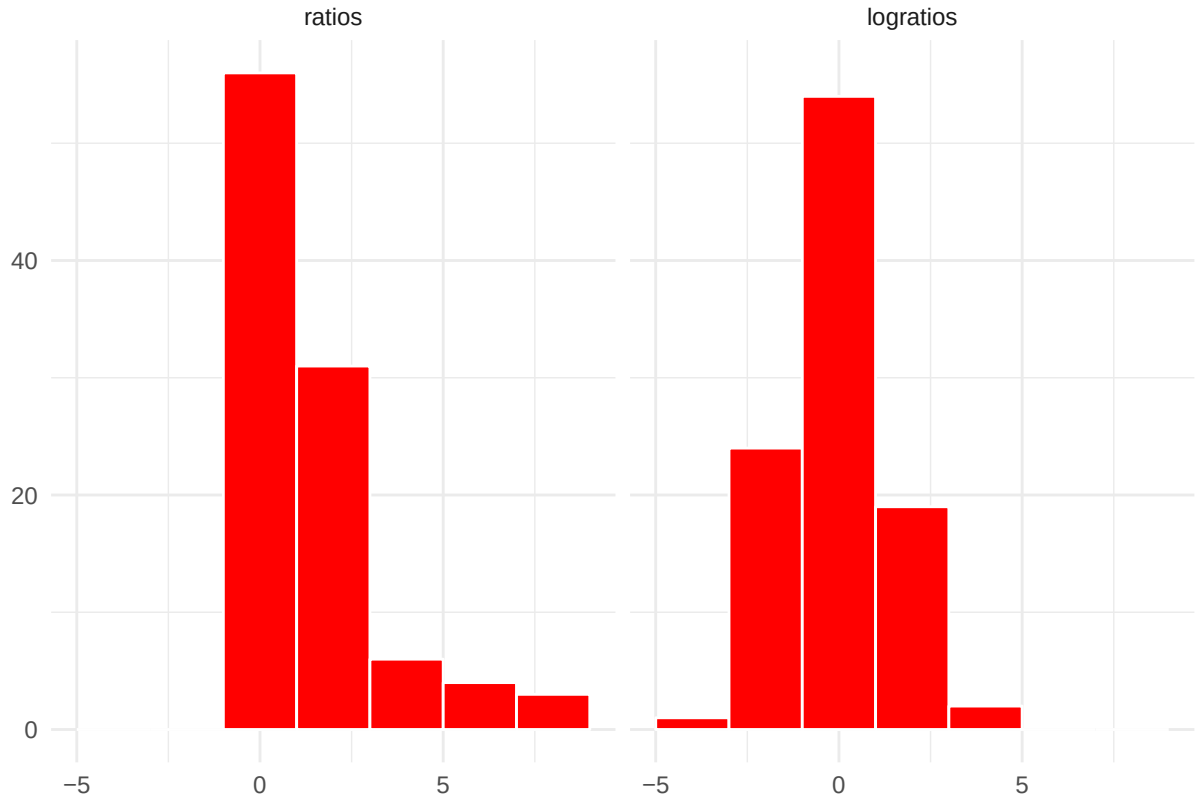


Figure 5.1: Ratios are not symmetric around one, logratios are symmetric around zero.

the treatment.

In life sciences, fold change is often reported as log-fold change. Why is that? There are at least two reasons which can be shown by plotting.

One is that ratios are not symmetrical around 1, so it's difficult to observe both changes in the forwards and backwards direcion i.e. proteins where expression went up and proteins where expression went down due to treatment. When we transform ratios on a log scale, the scale becomes symmetric around 0 and thus we can now observe the distribution of ratios in terms of positive, negative or no change.

A second reason is that transforming values onto a log scale changes where the numbers actually occur when plotted on that scale. If we consider the log scale to represent magnitudes, then we can more easily see changes of small and large magnitudes when we plot the data.

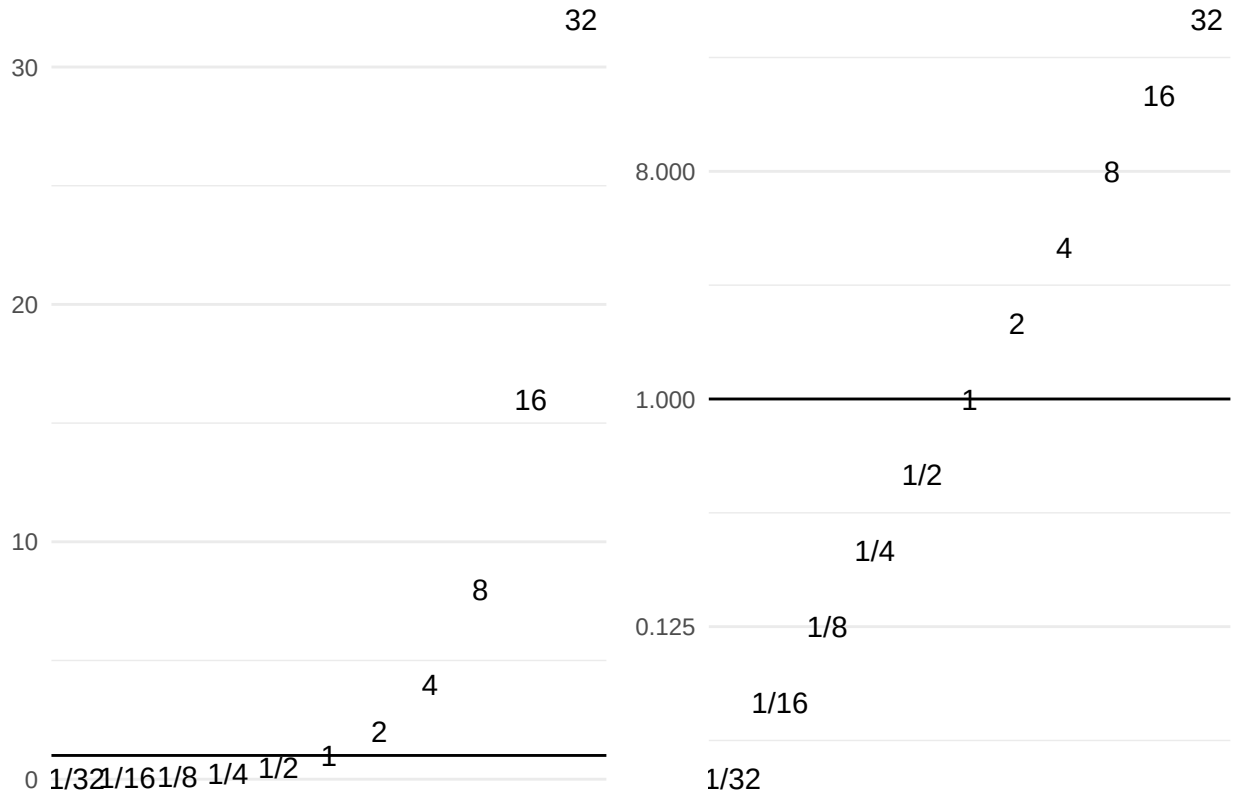


Figure 5.2: Transformation of scales using log transformation.

For example, a fold change of 32 times can be either a ratio 1/32 or 32/1.

As shown in Figure 5.2, 1/32 is much closer to 1 than 32/1, but transformed to a log scale we see that in terms of magnitude of difference it is the same as 32/1.

Often the log transformation is to a base of 2 as each increment of 1 represents a doubling, but sometimes a base of 10 is used, for example for p-values.

5.2 Dealing with missing values

Unless we're really lucky, it's unlikely that we'll get observations for the same numbers of proteins in all replicated experiments. This means there will be missing values for some proteins when looking at all the experiments together. This then raises the question of what to do about the missing values? We have two choices:

1. Only analyse the proteins that we have observations for in all experiments.
2. Impute values for the missing values from the existing observations.

There are pros and cons to either approach. Here for simplicity we'll use only the proteins for which we have observations in all assays.

We can drop the proteins with missing values by piping our data set to the `drop_na()` function from the `tidyr` package like so. We assign this to a new object called `dat_tidy`.

We'll use the summarise function to compare the number of proteins before and after dropping the missing values using the `n()` counting function.

```
# Remove the missing values
dat_tidy <- dat %>% drop_na()
# Nunber of proteins in original data
dat %>% summarise(Number_of_proteins = n())
```

```
## # A tibble: 1 x 1
##   Number_of_proteins
##                <int>
## 1               7702
```

```
# Nunber of proteins without missing values
dat_tidy %>% summarise(Number_of_proteins = n())
```

```
## # A tibble: 1 x 1
##   Number_of_proteins
##                <int>
## 1               1145
```

This shrinks the dataset from 7,702 proteins to 1,145 proteins, so we can see why imputing the missing values might be more atrractive.

One approach you might like to try is to impute the data by replacing the missing values with the mean observation for each protein under each condition.

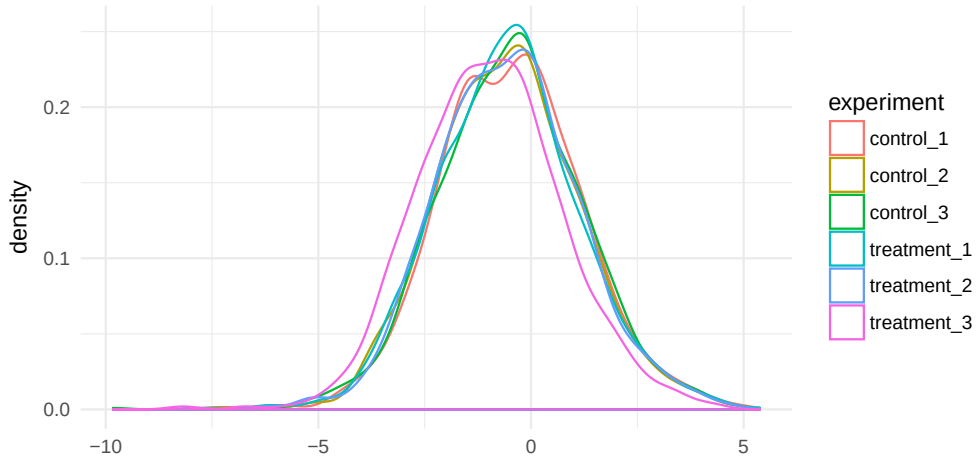


Figure 5.3: Protein data for six assays plotted as a distributions.

5.3 Data normalization

To perform statistical inference, for example whether treatment increases or decreases protein abundance, we need to account for the variation that occurs from run to run on our spectrometers and each give rise to a different distribution. This is as opposed to variation arising from treatment versus control which we are interested in understanding. Hence normalisation seeks to reduce the run-to-run sources of variation.

A method of normalization introduced for DNA microarray analysis is quantile normalisation (Bolstad et al., 2003). There are various ways to normalise data, so using quantile normalisation here is primarily to demonstate the approach in R, you should consider what is best for your data.

If we consider our proteomics data as a distribution of values, one value for the concentration of each protein in our experiment that together form a distribution. Figure 5.3 shows the distribution of protein concentrations observed for the three control and three treatment assays. As we can see the distributions are different for each assay.

A quantile represents a region of distribution, for example the 0.95 quantile is the value such that 95% of the data lies below it. To normalize two or more distributions with each other without recourse to a reference distribution we:

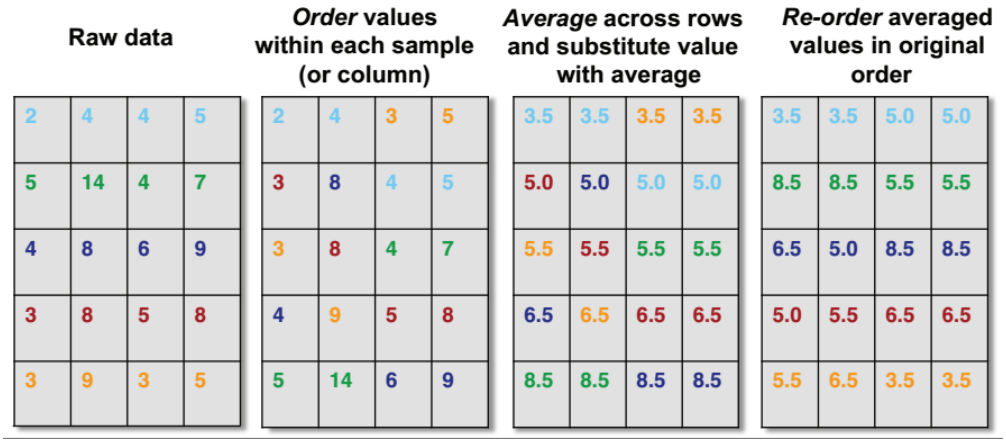


Figure 5.4: Quantile Normalisation from Rafael Irizarry's tweet.

(i) Rank the value in each experiment (represented in the columns) from lowest to highest. In other words identify the quantiles for each protein in each experiment.

(ii) Sort each experiment (the columns) from lowest to highest value.

(iii) Calculate the mean across the rows for the sorted values.

(iv) Then substitute these mean values back according to rank for each experiment to restore the original order.

This results in the highest ranking observation in each experiment becoming the mean of the highest observations across all experiments, the second ranking observation in each experiment becoming the mean of the second highest observations across all experiments. Therefore the distributions for each each experiment are now the same.

Dave Tang's Blog:Quantile Normalisation in R has more details on this approach.

These result of quantile normalisation is that our distributions become statisitcally identitical, which we can see by plotting the densities of the normalized data. As shown in Figure 5.5 the distributions all overlay.

We do this by creating a function. This takes a data frame as the arguement and pefrorms the steps described to iterate through the data frame.

The code below is probably quite tricky to understand if you've not seen `map` functions before, but they enable a function such as `rank` or `sort` to be used on each column

iteratively. What's important here is to understand the aim, even if understanding the code requires some more reading. You can read about map functions in R4DS.

```
# Quantile normalisation : the aim is to give different distributions the
# same statistical properties
quantile_normalisation <- function(df){

  # Find rank of values in each column
  df_rank <- map_df(df,rank,ties.method="average")
  # Sort observations in each column from lowest to highest
  df_sorted <- map_df(df,sort)
  # Find row mean on sorted columns
  df_mean <- rowMeans(df_sorted)

  # Function for substiting mean values according to rank
  index_to_mean <- function(my_index, my_mean){
    return(my_mean[my_index])
  }

  # Replace value in each column with mean according to rank
  df_final <- map_df(df_rank,index_to_mean, my_mean=df_mean)

  return(df_final)
}
```

The normalisation function is used by piping `dat_tidy` first to `select` to exclude the first two columns with the protein accession and description in, and then to the normalisation function. We re-bind the protein accession and description afterwards from `dat_tidy` by piping the output to `bind_cols()`.

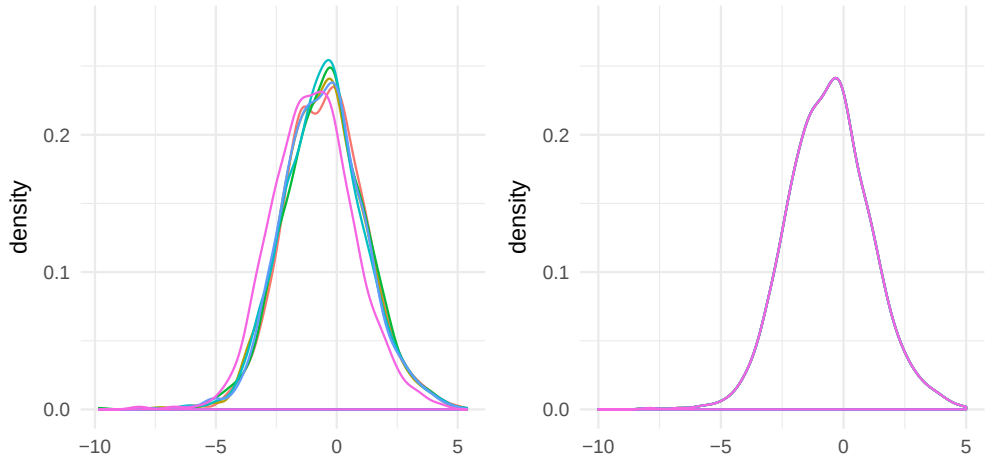


Figure 5.5: Comparison of the protein distributions before normalization (left) and after quantile normalization (right).

```
dat_norm <- dat_tidy %>% select(-c(protein_accession:protein_description)) %>%
  quantile_normalisation() %>%
  bind_cols(dat_tidy[,1:2],.)
```

5.4 Hypothesis testing with the t-test

Having removed missing values and normalised the data, we can consider our hypothesis: treatement changes the amount of protein we observe in the cells.

In practice then, what we would like to know is whether the mean value for each protein in our control and treatment assays differs due to chance or due a real effect. We therefore need to calculate the difference for each protein between treatment and control, and the probability that any difference occurs due to chance. This is what the p-value from the output of a t-test seeks to do. We need to perform 1145 t-tests.

Note There are biocondutor packages that contain functions written to do this. However as a learning exercise we are going to work through the problem.

Here I assume the reader is familiar with t-tests, but just to re-cap some important points:

- We assume that the true population from which our data samples are indpendent, identically distributed and follow a normal distribution. This is not in fact true in practice, but t-test is robust to this assumption.
- We assume unequal variances between the control and treatment for each protein. Hence we will perform a Welch's t-test for unequal variances.
- We don't know whether the effect of the treatment is to increase or decrease the concentration of the protein, hence we will perform a two-sided t-test.
- The observations for the proteins are for proteins of the same type but from independent experiments, rather than observations of the same individuals before and after treatment. Hence we test the observations as unpaired samples.

In R we use the base function `t.test` to perform Welch Two Sample t-test and this outputs the p-values we need for each protein. However, the challenge here is that our data has three observations for each condition for each protein, hence we need to group the observations for each protein according to the experimental condition as inputs to each t-test.

We're going to follow what is called the *split-apply-combine* approach to deal with this problem:

1. Split the data into control and treatment groups.
2. Apply the t-test function to each protein using the grouped inputs and store the p-value.
3. Combine all the p-values for each protein into a single vector.

To this end I've created a function called `t_test` that takes a data frame and two group vectors as inputs. It splits the data into `x` and `y` by subsetting the the data frame according to the columns defined by the groups. The extra steps here are that the subset data has to be unlisted and converted to numeric type for input to the `t.test` function. We then perform the t-test, which will calculate the mean of `x` and `y` and store the result in a new

object, and finally the function creates a data frame with a single variable`p_val` which is then returned as the function output.

```
# T-test function for multiple experiments
t_test <- function(dt,grp1,grp2){
  # Subset control group and convert to numeric
  x <- dt[grp1] %>% unlist %>% as.numeric()
  # Subset treatment group and convert to numeric
  y <- dt[grp2] %>% unlist %>% as.numeric()
  # Perform t-test using the mean of x and y
  result <- t.test(x, y)
  # Extract p-values from the results
  p_vals <- tibble(p_val = result$p.value)
  # Return p-values
  return(p_vals)
}
```

To use the `t_test` function to perform many t-tests and not just one t-test, we need to pass our `t_test` function as an arguement to another function.

This probably seems quite confusing, but the point here is that we want to loop through every row in our table, and group the three control and three treatment columns separately. Our `t_test` function deals with the latter problem, and by passing it to `adply` from the `plyr` package we can loop through each row and it adds the calculated p-values to our original table.

Concretely then, `adply` takes an array and applies the `t_test` function to each row and we supply the column group indices arguments to the `t_test` funcition. Here the indicies are columns 3 to 5 for the control experiments and columns 6 to 8 for the treatment functions. The function returns the input data with an additional corresponding p-value column. **Note** I've piped the output to `as.tibble()` to transform the data.frame output of `adply` to tibble form to prevent errors that can occur if we try to bind data frames and

tibbles.

An important point here is that we can use this function for any number of columns and rows providing our data is in the same tidy form by changing the grouping indices.

```
# Apply t-test function to data using plyr adply
#  .margins = 1, slice by rows, .fun = t_test plus t_test arguements
dat_pvals <- plyr::adply(dat_norm,.margins = 1, .fun = t_test,
                grp1 = c(3:5), grp2 = c(6:8)) %>% as.tibble()
```

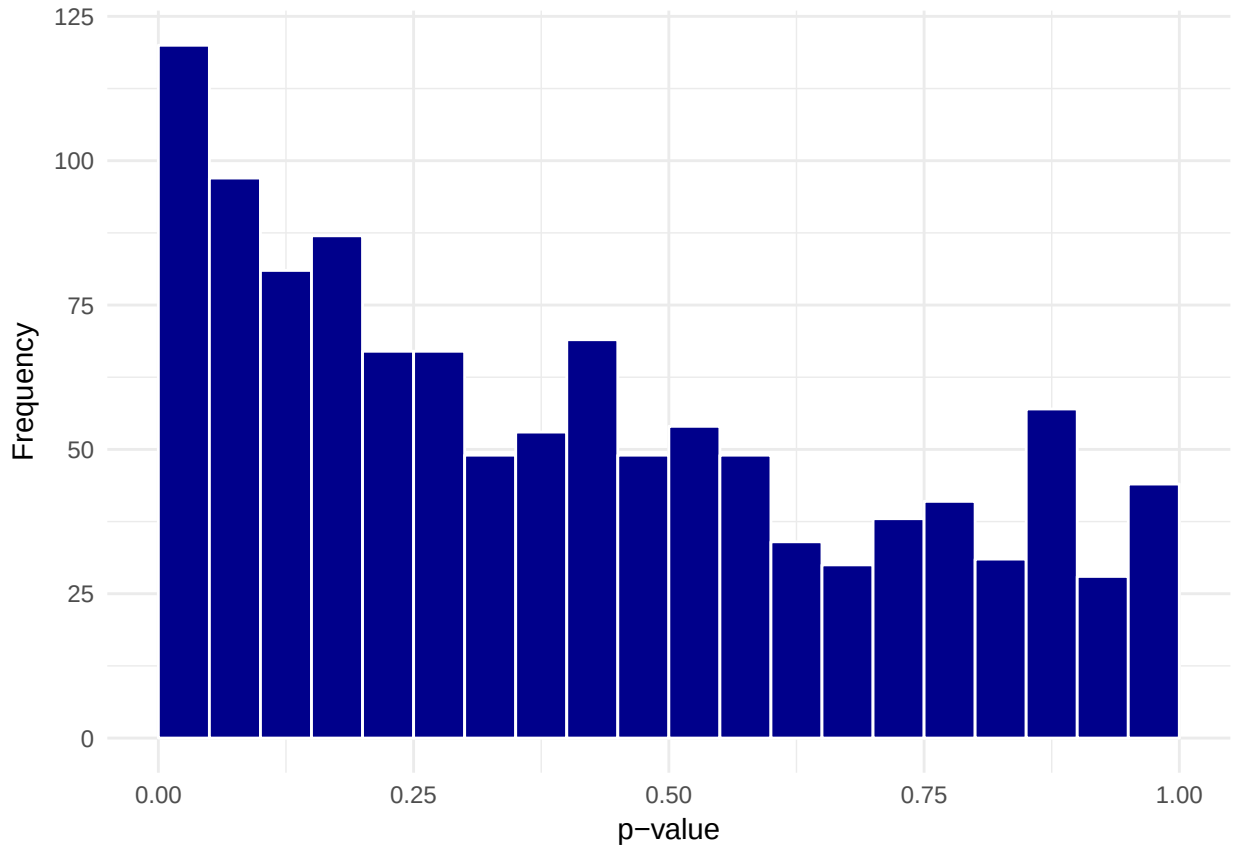
To check our function, here's a comparision of calculating the first protein p-value as a single t-test as shown in the following code and the output of the function.

```
# Perform t-test on first protein
t.test(as.numeric(dat_norm[1,3:5]),
                    as.numeric(dat_norm[1,6:8]))$p.value
```

t_test p-val	t.test p-val
0.0927	0.0927

We can plot a histogram of the p-values:

```
# Plot histogram
dat_pvals %>%
  ggplot(aes(p_val)) +
  geom_histogram(binwidth = 0.05,
            boundary = 0.5,
            fill = "darkblue",
            colour = "white") +
  xlab("p-value") +
  ylab("Frequency") +
  theme_minimal()
```



5.5 Calculating fold change

To perform log transformation of the observations for each protein we take our data and use select to exlude the columns of character vectors and the pipe the output to `log2()` and use the pipe again to create a data frame.

Then we use `bind_cols` to bind the first two columns of `dat_pvals` followed by `dat_log` and the last column of `dat_pvals`. This maintains the original column order.

```
# Select columns and log data
dat_log <- dat_pvals %>%
  select(-c(protein_accession,protein_description,p_val)) %>%
  log2()
```

```
# Bind columns to create transformed data frame
dat_combine <- bind_cols(dat_pvals[,c(1:2)], dat_log, dat_pvals[,9])
```

The log fold change is then the difference between the log mean control and log mean treatment values. By use of grouping by the protein accession we can then use `mutate` to create new variables that calculate the mean values and then calculate the `log_fc`. Whilst we're about it, we can also calculate a -log10(p-value). As with fold change, transforming the p-value on a log10 scale means that a p-value of 0.05 or below is transformed to 1.3 or above and a p-value of 0.01 is equal to 2.

```
dat_fc <- dat_combine %>%
  group_by(protein_accession) %>%
  mutate(mean_control = mean(c(control_1,
                               control_2,
                               control_3)),
                             mean_treatment= mean(c(treatment_1,
                                                    treatment_2,
                                                    treatment_3)),
         log_fc = mean_control - mean_treatment,
         log_pval = -1*log10(p_val))
```

The next step is not necessary, but for ease of viewing we subset `dat_fc` to create a new data frame called `dat_tf` that contains only four variables. We could potentially write this to a csv file for sharing.

```
# Final transformed data
dat_tf <- dat_fc %>% select(protein_accession,
                            protein_description,
                            log_fc, log_pval)
```

Let's look at the head of the final table:

protein_accession	protein_description
VATA_HUMAN_P38606	V-type proton ATPase catalytic subunit A OS=Homo sapiens GN=ATP6V1A PE
RL35A_HUMAN_P18077	60S ribosomal protein L35a OS=Homo sapiens GN=RPL35A PE=1 SV=2
MYH10_HUMAN_P35580	Myosin-10 OS=Homo sapiens GN=MYH10 PE=1 SV=3
RHOG_HUMAN_P84095	Rho-related GTP-binding protein RhoG OS=Homo sapiens GN=RHOG PE=1 S
PSA1_HUMAN_P25786	Proteasome subunit alpha type-1 OS=Homo sapiens GN=PSMA1 PE=1 SV=1

5.6 Visualising the transformed data

Plotting a histogram of the log fold change gives an indication of whether the treatment has an effect on the cells. Most values are close to zero, but there are some observations far above and below zero suggesting the treatment does have an effect.

```
# Plot a histogram to look at the distribution.
dat_tf %>%
  ggplot(aes(log_fc)) +
  geom_histogram(binwidth = 0.5,
                 boundary = 0.5,
           fill = "darkblue",
           colour = "white") +
  xlab("log2 fold change") +
  ylab("Frequency") +
  theme_minimal()
```

However, we don't know if these fold changes are dueto chance or not, which is why we calculated the p-values. A volcano plot will include the p-value information.

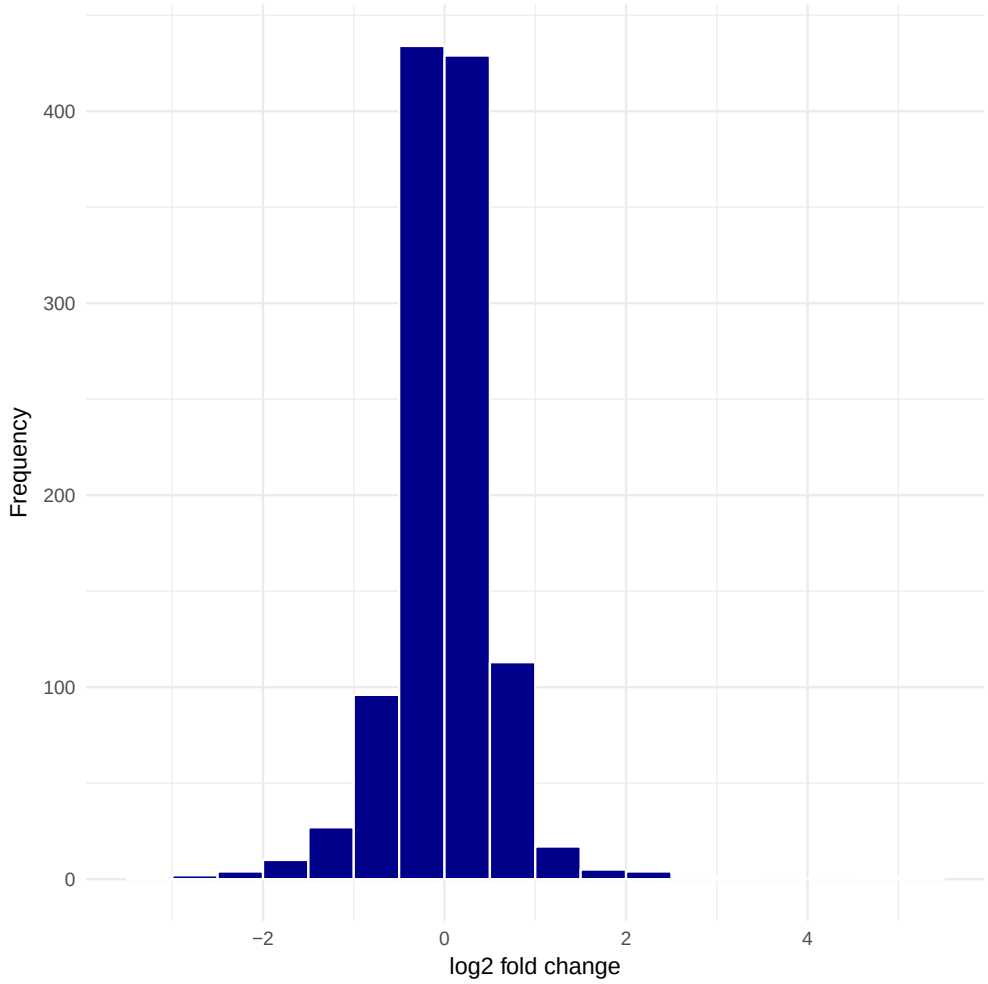


Figure 5.6: Histogram of log fold change.

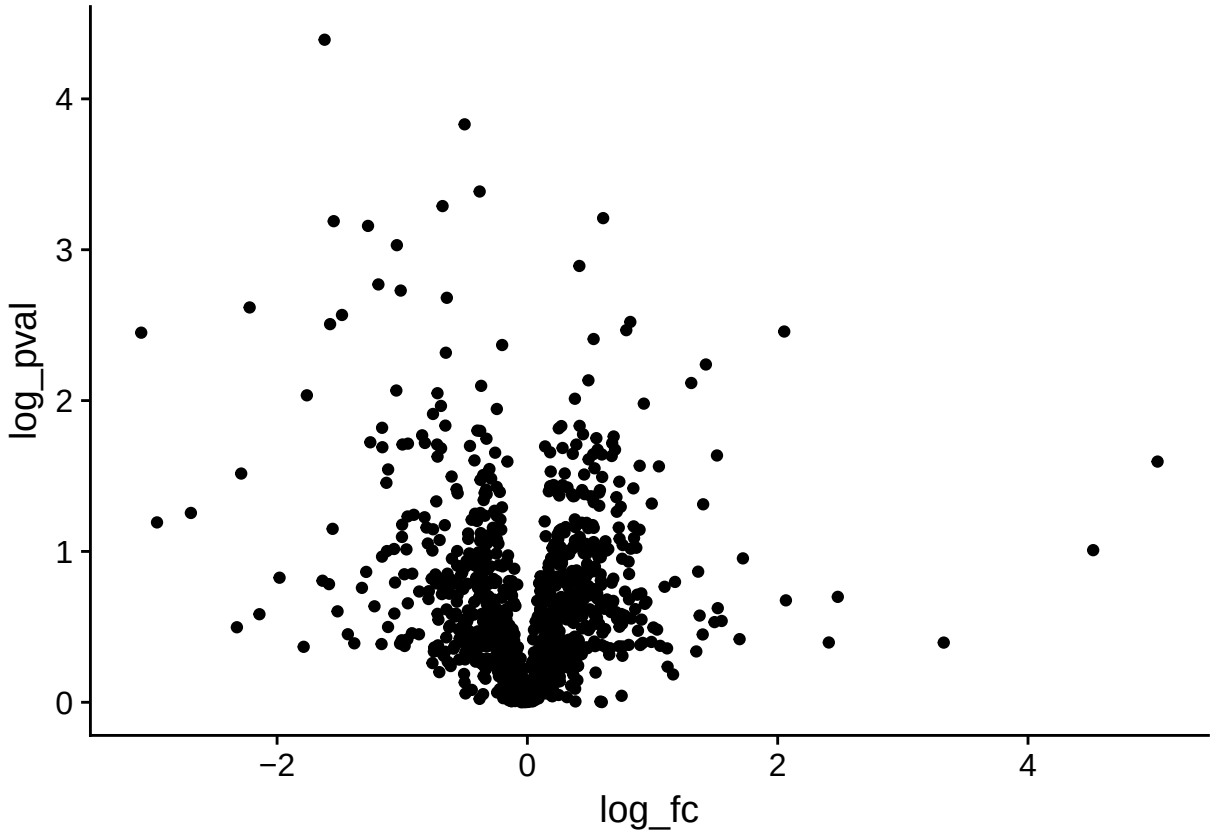
5.7 Volcano plot

A volcano plot is a plot of the log fold change in the observation between two conditions on the x-axis, for example the protein expression between treatment and control conditions. On the y-axis is the corresponding p-value for each observation, representing the likelihood that an observed change is due to the different conditions rather than arising from a natural variation in the fold change that might be observed if we performed many replications of the experiment.

The aim of a volcano plot is to enable the viewer to quickly see the effect (if any) of an experiment with two conditions on many species (i.e. proteins) in terms of both an increase and decrease of the observed value.

Like all plots it has it's good and bad points, namely it's good that we can visualise a lot of complex information in one plot. However this is also it's main weakness, it's rather complicated to understand in one glance.

```
dat_tf %>% ggplot(aes(log_fc,log_pval)) + geom_point()
```



However it would be much more useful with some extra formatting, so the code below shows one way to transform the data to include a threshold which can then be used by ggplot to create an additional aesthetic. The code below also includes some extra formatiing which the reader can explore.

```
dat_tf %>%
  # Add a threhold for significant observations
  mutate(threshold = if_else(log_fc >= 2 & log_pval >= 1.3 |
                               log_fc <= -2 & log_pval >= 1.3,"A", "B")) %>%
  # Plot with points coloured according to the threshold
  ggplot(aes(log_fc,log_pval, colour = threshold)) +
  geom_point(alpha = 0.5) + # Alpha sets the transparency of the points
  # Add dotted lines to indicate the threshold, semi-transparent
  geom_hline(yintercept = 1.3, linetype = 2, alpha = 0.5) +
  geom_vline(xintercept = 2, linetype = 2, alpha = 0.5) +
```

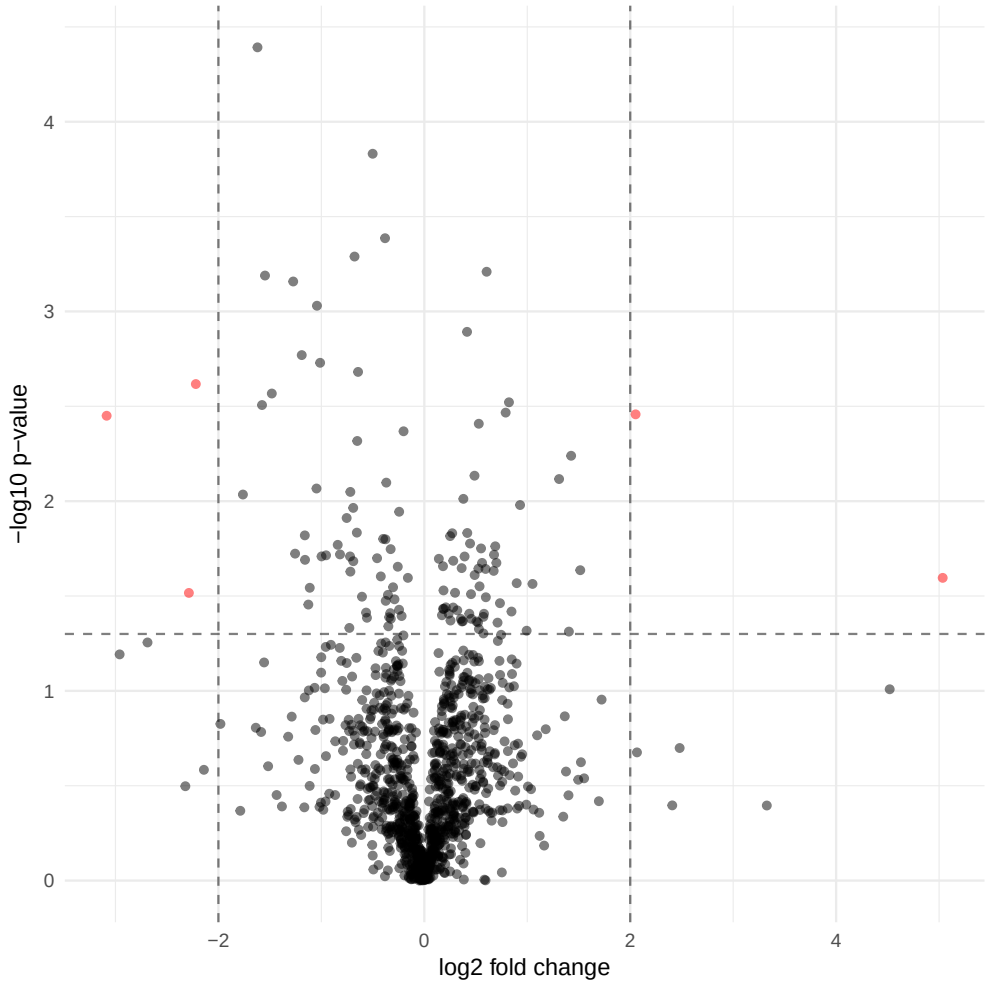


Figure 5.7: A volcano plot with formatting to highlight the significant proteins

```
geom_vline(xintercept = -2, linetype = 2, alpha = 0.5) +
# Set the colour of the points
scale_colour_manual(values = c("A"= "red", "B"= "black")) +
xlab("log2 fold change") + ylab("-log10 p-value") + # Relabel the axes
theme_minimal() + # Set the theme
theme(legend.position="none") # Hide the legend
```

5.7.1 But which proteins are the significant observations?

To extract the proteins in red in Figure 5.7 we filter `dat_tf` according to our threshold and then create a new variable using the `str_extract` function used in Section 4.5.

Note We need to ungroup the data we grouped when calculating the log_fc to be able to select columns without keeping the grouping variable column too.

```
dat_tf %>%
  # Filter for significant observations
  filter(log_pval >= 1.3 & (log_fc >= 2 | log_fc <= -2)) %>%
  # Get last six characters
  mutate(prot_id = str_extract(protein_accession,".{6}$")) %>%
  # Ungroup the data
  ungroup() %>%
  # Select columns of interest
  select(prot_id,protein_description,log_fc,log_pval)
```

```
## # A tibble: 5 x 4
##   prot_id protein_description                                   log_fc log_pval
##   <chr>   <chr>                                                  <dbl>    <dbl>
## 1 Q02952  A-kinase anchor protein 12 OS=Homo sapiens GN=A~  -2.29     1.52
## 2 O94808  Glutamine--fructose-6-phosphate aminotransferas~  -3.09     2.45
## 3 H7BYV1  Interferon-induced transmembrane protein 2 (Fra~   2.05     2.46
## 4 P06756  Integrin alpha-V OS=Homo sapiens GN=ITGAV PE=1 ~  -2.22     2.62
## 5 Q8TDI0  Chromodomain-helicase-DNA-binding protein 5 OS=~   5.04     1.60
```

5.8 Creating a heatmap

Here we'll create a heatmap using the `heatmap.2` function from the `gplots` package and the `pheatmap` function from the `pheatmap` package.

To create a heatmap we need to perform a few more transformations:

1. Filter the data according to a threshold of significance. This time we'll use a more relaxed log_fc cut-off to ensure we have enough proteins to plot. At the same time we'll extract the protein ids as before.
2. We then have to transform our filtered data into a `matrix.data.frame` object for use with `pheatmap`. We name the rows with the protein ids
3. We'll use base R function `scale` to centre our log transformed data around zero. To do this per experiment we transpose the matrix as scale centres rows, and the flip the matrix back again.

```
# Keep the same p-val cut-off, but relax the log_fc to 1 which represents a
# doubling
dat_filt <- dat_fc %>%
  filter(log_pval >= 1.3 & (log_fc >= 1 | log_fc <= -1)) %>%
  mutate(prot_id = str_extract(protein_accession,".{6}$"))

# Convert to matrix data frame
dat_matrix <- as.matrix.data.frame(dat_filt[,3:8])
# Name the rows with protein ids
row.names(dat_matrix) <- dat_filt$prot_id
# Transpose and scale the data to a mean of zero and sd of one
dat_scaled <- scale(t(dat_matrix)) %>% t()
```

5.8.1 Calculating similarity and clustering

At this point we could just plot the data, but to understand what the heatmap functions do to cluster the data, let's step through the process.

Our data here as log fold change in concentrations, but how do we group them? The simplest thing to do is to turn the data into distances, as a measure of similarity, where close things are similar and distant things are dissimilar.

The Euclidean distance d between a pair of observations x_i and y_i is defined as:

$d = \sqrt{\sum_i (x_i - y_i)^2}$

Lets calculate the distance between the columns in `dat_scaled`.

In `dat_scaled` the experiments are in the columns. In calculating the distance is between the experiments for all the proteins in each experiment. What would we expect?

We'd expect the controls to be close to each other and the treated to be close to each other, right?

Let's do this in detail, for example the distance between `control_1` and `control_2` is `sqrt(sum((dat_scaled[,1] - dat_scaled[,2])^2))`.

This means we take the column 2 values from column 1 values, squaring the results and summing them all to a single value and taking the square root to find the linear distance between these rows, which is 3.26.

You can check this against the first value in `d1` that we calculate below in using `dist`.

We do the same for the proteins, but we don't know what to expect. Here's the code for calculating both distance matrices

```
# Transpose the matrix to calculate distance between experiments, row-wise
d1 <- dat_scaled %>% t() %>%
  dist(.,method = "euclidean", diag = FALSE, upper = FALSE)
# Calculate the distance between proteins row-wise
d2 <- dat_scaled %>%
  dist(.,method = "euclidean", diag = FALSE, upper = FALSE)

# Show the values for d1
round(d1,2)
```

```
##             control_1 control_2 control_3 treatment_1 treatment_2
## control_2        3.26
## control_3        3.20      3.27
## treatment_1      8.97      8.60      8.65
## treatment_2      9.40      8.98      8.86        2.35
## treatment_3      9.04      8.56      8.50        2.46        1.71
```

Having calculated the distance matrices, we can cluster proteins and experiments accordingly.

There are lots of flavours of clustering, and no clear way to say which is best. Here we'll use the Ward criterion for clustering which attempts to minimise the variance within clusters as it merges the data into clusters, using the distances we've calculated. The data is merged from the bottom up (aka agglomeration) adding data points to a cluster and splitting them according to the variance criterion.

See Wikipedia for more detail: Hierarchical clustering

```
# Clustering distance between experiments using Ward linkage
c1 <- hclust(d1, method = "ward.D2", members = NULL)
# Clustering distance between proteins using Ward linkage
c2 <- hclust(d2, method = "ward.D2", members = NULL)
```

Now lets look at the dendrograms made by clustering our distance matrices `d1` and `d2`:

```
# Check clustering by plotting dendrograms
par(mfrow=c(2,1),cex=0.5) # Make 2 rows, 1 col plot frame and shrink labels
plot(c1); plot(c2) # Plot both cluster dendrograms
```

As we'd expect, Figure 5.8 shows the controls and treatments cluster respectively.

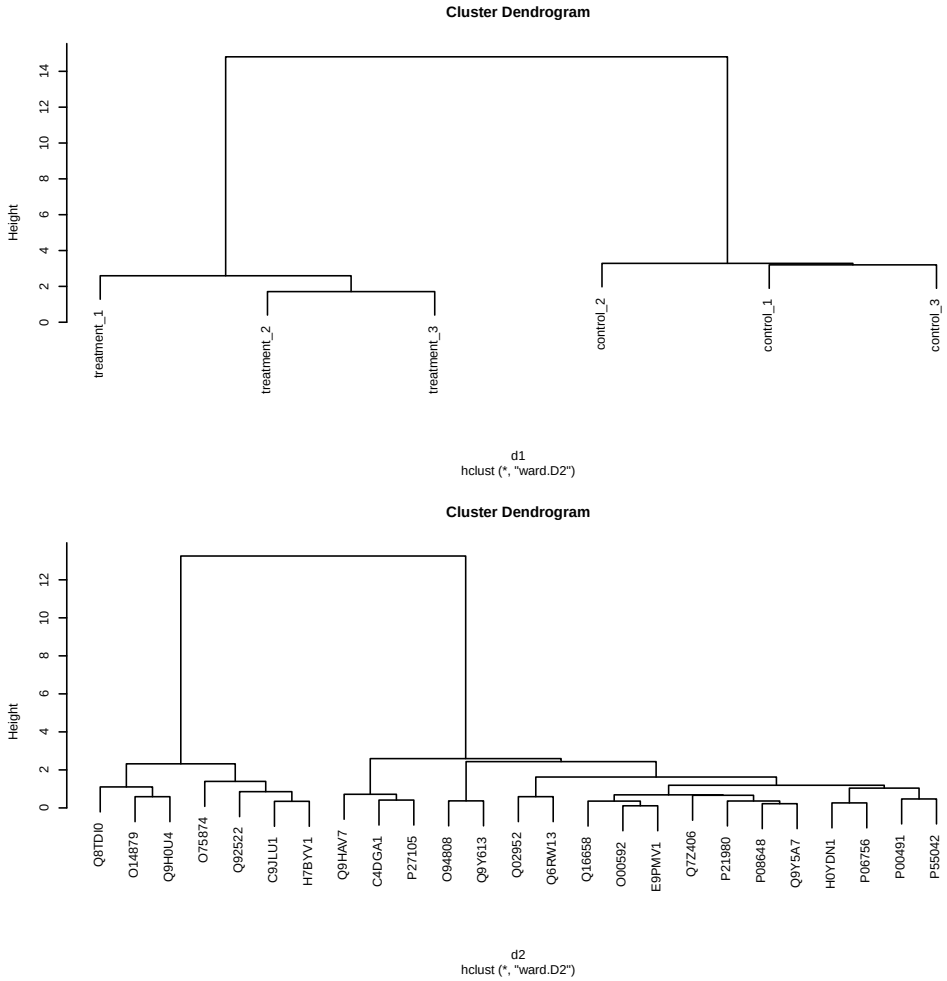


Figure 5.8: Dendrograms of Ward clustering of distance matrices

5.8.2 Plotting the heatmap

The `heatmap.2` function from the `gplots` package will automatically perform the distance calculation and clustering we performed, and it can also do the scaling we did. It only requires the matrix as an input by default. It will use a different clustering method by default.

However, as we've performed scaling and calculated the clusters, we can pass them to heatmap function.

I'll leave it to the reader to explore all the options here, but the concept in the code below to create Figure 5.9 is:

- Create a 25 increment blue/white/red colour pallette
- Pipe `dat_scaled` to a function that renames the colums
- Pipe this to the `heatmap.2` function
- Pass the clusters `c1` and `c2` to the plot
- Change some aesthetics such as the colours, and the font sizes

```
# Set colours for heatmap, 25 increments
my_palette <- colorRampPalette(c("blue","white","red"))(n = 25)

# Plot heatmap with heatmap.2
par(cex.main=0.75) # Shrink title fonts on plot
dat_scaled %>%
  # Rename the comlums
  magrittr::set_colnames(c("Ctl 1", "Ctl 2", "Ctl 3",
                           "Trt 1", "Trt 2", "Trt 3")) %>%
  # Plot heatmap
  gplots::heatmap.2(.,                    # Tidy, normalised data
          Colv=as.dendrogram(c1),     # Experiments clusters in cols
          Rowv=as.dendrogram(c2),     # Protein clusters in rows
```

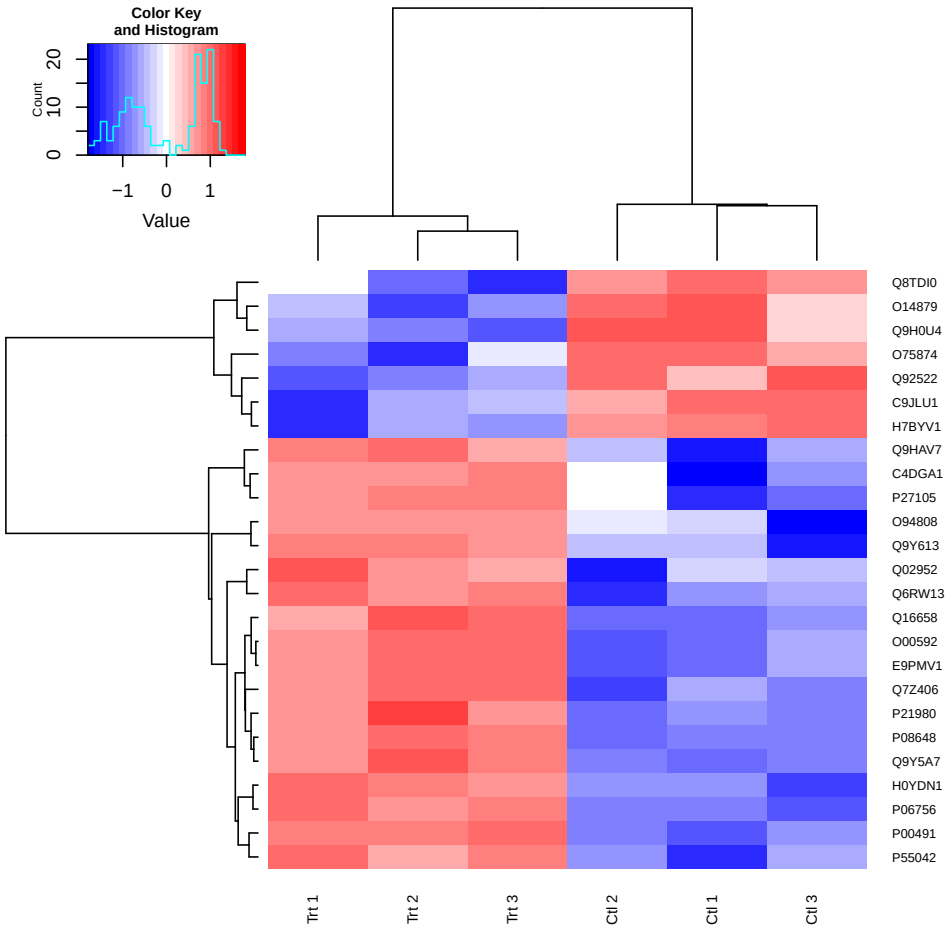


Figure 5.9: Heatmap created with `heatmap.2` using the clusters calculated.

```
        revC=TRUE,                  # Flip plot to match pheatmap
        density.info="histogram",   # Plot histogram of data and colour key
        trace="none",               # Turn of trace lines from heat map
        col = my_palette,           # Use my colour scheme
        cexRow=0.6,cexCol=0.75)     # Amend row and column label fonts
```

An alternative and more `ggplot` style is to use the `pheatmap` package and function.

In Figure 5.10 `dat_scaled` is piped to `set_columns` again to rename the experiments for aesthetic reasons. The output is the piped to `pheatmap` which performs the distance and clustering automatically. The only additional arguements used here are to change the

fontsize and create some breaks in the plot to highlight the clustering.

There is lots more that `pheatmap` can do in terms of aesthetics, so do explore.

```
dat_scaled %>%
  # Rename the comlums
  magrittr::set_colnames(c("Ctl 1", "Ctl 2", "Ctl 3",
                                    "Trt 1", "Trt 2", "Trt 3")) %>%
  # Plot heatmap
  pheatmap(.,
           fontsize = 7,
           cutree_rows = 2, # Create breaks in heatmap
           cutree_cols = 2) # Create breaks in heatmap
```

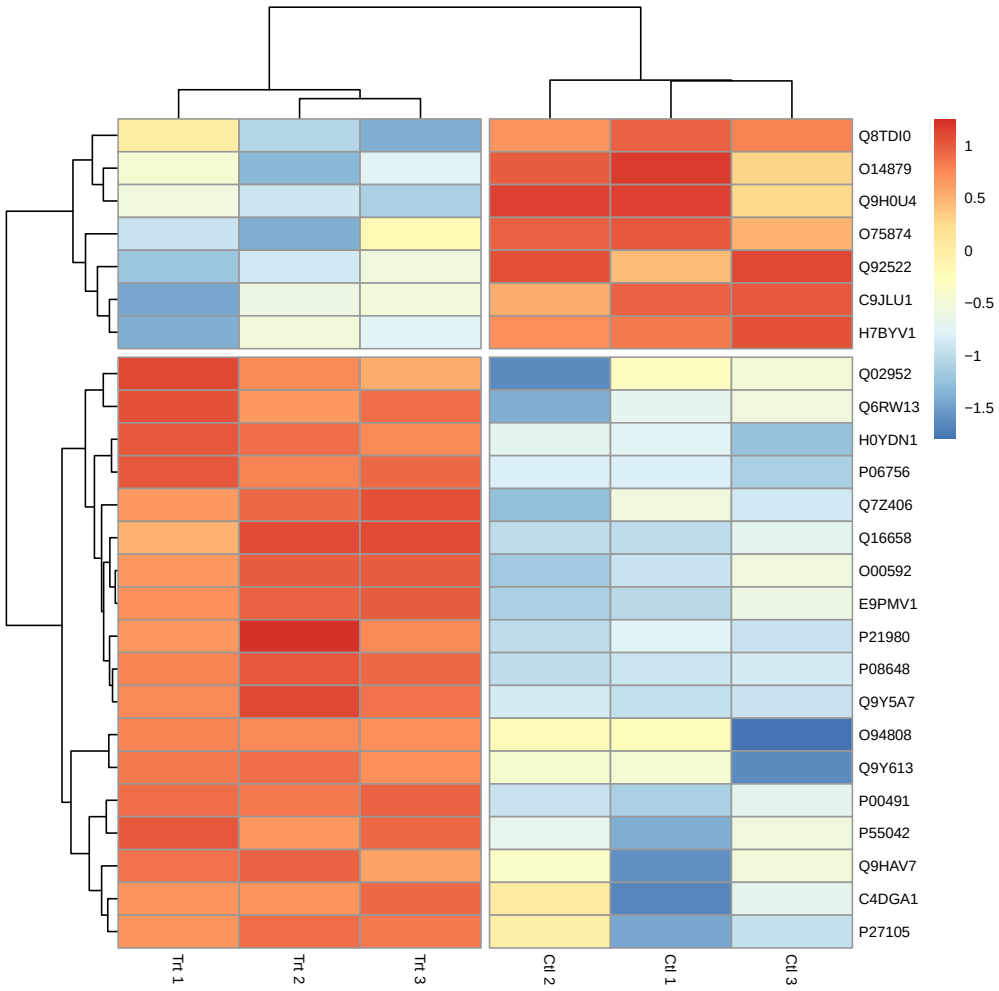


Figure 5.10: Heatmap created using `pheatmap` with breaks to highlight clusters.

Chapter 6

Going further

Here are a few links and suggestions about what else you might like to do with R.

6.1 Exporting figures

Exporting figures is best done using the following structure:

```
# Open up a blank plot file, pdf,jpeg etc.
<plot_function("file",...)>
# Write the plot to the file
<plot_object>
# Close the file
dev.off()
```

For example to export the volcano plot from Figure 5.7 to a pdf, we do:

```
# Open up a blank plot file, pdf,jpeg etc.
pdf("volcano_plot.pdf")


# Write the plot to the file
```

```
dat_tf %>%
  # Add a threhold for significant observations
  mutate(threshold = if_else(log_fc >= 2 & log_pval >= 1.3 |
                               log_fc <= -2 & log_pval >= 1.3,"A", "B")) %>%
  # Plot with points coloured according to the threshold
  ggplot(aes(log_fc,log_pval, colour = threshold)) +
  geom_point(alpha = 0.5) + # Alpha sets the transparency of the points
  # Add dotted lines to indicate the threshold, semi-transparent
  geom_hline(yintercept = 1.3, linetype = 2, alpha = 0.5) +
  geom_vline(xintercept = 2, linetype = 2, alpha = 0.5) +
  geom_vline(xintercept = -2, linetype = 2, alpha = 0.5) +
  # Set the colour of the points
  scale_colour_manual(values = c("A"= "red", "B"= "black")) +
  xlab("log2 fold change") + ylab("-log10 p-value") + # Relabel the axes
  theme_minimal() + # Set the theme
  theme(legend.position="none") # Hide the legend

# Close the file
dev.off()
```

If I had saved the plot to an object called `vplot` I would call that object instead of making the plot using `dat_tf` as shown here.

Here is a general guide to the various formats you can export to.

Alternatively, if you are working in `ggplot` you can use the `ggsave` function as described in R4DS 28.7.

6.2 Exporting data

There is a full manual for the import and export of data in R. However here are few pointers:

6.2.1 Writing to a file

One of the most portables way to share data is by writing to a csv file. These files can be opened in many programs. The tidyverse package contains two functions for csv files, `write_csv` and for Excel `write_excel_csv`. The latter form adds a bit of metadata that tells Excel about the file encoding. See R4DS writing to a file

For example to write a csv file of `dat_tf` to a file called `04072018_transformed_data.csv` to our working directory for sharing with a colleague using excel, the code is of the form `<function>(<r-object>,"filename")` like so:

```
write_excel_csv(dat_tf,"04072018_transformed_data.csv")
```

Note that the file name is a string and is in quotes.

6.2.2 For R

If you are exporting data to use yourself in R, the custom `.rds` format is a good choice and preserves the R structure.

In the tidyverse `write_rds` follows the same structure as `write_csv`.

You can read back in using `read_rds`.

6.3 Joining the R community

It's worth joining the RStudio Community and following community members on Twitter such as Jenny Bryan, Hadley Wickham, Yihui Xie, Mara Averick, David Robinson and Julia

Silge.

If you can afford DataCamp then this is my preferred learning platform.

And if you can't, then swirl is free.

6.4 Communication: creating reports, presentations and websites

R Markdown enables us to do literate programming, saving time as we can create analysis, reports, dashboards or web apps at the same time as writing code. R Markdown can use multiple programming languages. See also R4DS R Markdown and R4DS R Markdown formats.

You can use blogdown to build websites. I created s guide to buidling an academic website with blogdown.

6.5 Machine Learning

If you are interested in machine learning, then TensorFlow is a good place to start, for example Leon Eyrich Jessen's Deep Learning for Cancer Immunotherapy tutorial.

References

Bolstad, B. M., Irizarry, R. A., Astrand, M., and Speed, T. P. (2003). A comparison of normalization methods for high density oligonucleotide array data based on variance and bias. *Bioinformatics (Oxford, England)*, 19:185–193.

Gentleman, R. C., Carey, V. J., Bates, D. M., Bolstad, B., Dettling, M., Dudoit, S., Ellis, B., Gautier, L., Ge, Y., Gentry, J., Hornik, K., Hothorn, T., Huber, W., Iacus, S., Irizarry, R., Leisch, F., Li, C., Maechler, M., Rossini, A. J., Sawitzki, G., Smith, C., Smyth, G., Tierney, L., Yang, J. Y. H., and Zhang, J. (2004). Bioconductor: open software development for computational biology and bioinformatics. *Genome biology*, 5:R80.

Huber, W., Carey, V. J., Gentleman, R., Anders, S., Carlson, M., Carvalho, B. S., Bravo, H. C., Davis, S., Gatto, L., Girke, T., Gottardo, R., Hahne, F., Hansen, K. D., Irizarry, R. A., Lawrence, M., Love, M. I., MacDonald, J., Obenchain, V., Ole, A. K., Pagès, H., Reyes, A., Shannon, P., Smyth, G. K., Tenenbaum, D., Waldron, L., and Morgan, M. (2015). Orchestrating high-throughput genomic analysis with bioconductor. *Nature methods*, 12:115–121.

Ihaka, R. and Gentleman, R. (1996). R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

RStudio Team (2018). *RStudio: Integrated Development Environment for R*. RStudio, Inc., Boston, MA.

Wickham, H. (2017). *tidyverse: Easily Install and Load the 'Tidyverse'*. R package version 1.2.1.