Exploratory Data Analysis

Rens Holmer Mark Sterken Harm Nijveen
Peter Bourke

2026-03-05

Table of contents

Introduction

This online book is the main study material for the course BIF20806: Exploratory Data Analysis in R. This course is taught at Wageningen University and targets 2^{nd} year students in the BSc program Plant Sciences (BPW). The aim of this book and the course is to provide sufficient information and background so that after finishing, students can work independently on data analysis projects with R, using datasets that are typically encountered in the plant sciences.

Data analysis in the plant sciences

There is no plant science without data, and working with data has always been an essential skill for any plant scientist. Historically, this would not have involved talking about programming languages. However, increasing awareness of reproducibility and the ever-growing volume of data in the plant sciences mean that plant scientists today routinely turn to computers when analysing their data.

One of the most famous experiments in plant science is Gregor Mendel's work on pea plants in the mid-19th century [7]. Mendel carefully crossed plants with different observable traits (such as seed colour or shape) and systematically counted how often each trait appeared in the offspring. These experiments later proved foundational for the development of Mendel's theories on genetic inheritance.

What makes Mendel's work particularly relevant here is not the biological theory that later emerged from it, but the way he worked with data. Mendel designed controlled experiments, collected quantitative observations, organised his results in tables, and searched for patterns in those data before proposing any formal explanation. In modern terms, much of this would now be described as exploratory data analysis.

If Mendel were working today, he would likely store his data in a spreadsheet or data frame, visualise trait frequencies with plots, and use simple statistical summaries to explore variation and uncertainty. The tools would be different, but the underlying questions would be the same.

What if Mendel could use R?

In his seminal experiments, Mendel counted inheritance patters and compared them to a theoretical model. At the time, the statistical tools we would use today had not been developed yet, so he mostly eyeballed the difference. These days we would calculate a p-value according to a statistical model to guide our estimation of uncertainty of the outcome. If Mendel used R, and had access to modern statistical tools, his approach might look something like the example below.

```
# Observed counts in F2: dominant vs recessive
obs <- c(dominant = 705, recessive = 224)

# Expected Mendelian proportions (3:1)
p <- c(0.75, 0.25)

# Goodness-of-fit test (Mendel did not have this!)
chisq.test(obs, p = p)
```

```
    Chi-squared test for given probabilities

data:  obs
X-squared = 0.39074, df = 1, p-value = 0.5319
```

From this statistical test we can conclude that the observed counts do not significantly deviate from a 3:1 ratio, confirming Mendel's model of inheritance.

About the course

In this course, we focus on modern tools for asking and answering questions from data. We discuss strategies to acquire, process, explore, visualize, and analyse data for a variety of plant science related data sets. In addition, we work on making data analysis projects reproducible: by using the R programming language, we create reproducible analysis workflows.

Why R?

R is widely used in the plant sciences and related fields because it combines data handling, statistical analysis, and visualisation in a single environment. Unlike point-and-click software, analyses in R are expressed explicitly as code. This makes every step of an analysis transparent and reproducible, and allows analyses to scale from small, simple datasets to large and complex ones. Learning R therefore provides not only practical skills for this course, but also a foundation that students can build on in later courses, internships, and research projects.

This course does not assume prior experience with programming. Instead, we approach R as a tool for thinking with data. Concepts are introduced gradually, starting from basic data manipulation and simple visualisation, and building towards more complex analyses. Throughout the course, the focus remains on understanding the data and the questions being asked, rather than on writing code for its own sake.

About the book

This book is designed to support the learning objectives of the course by providing structured, practice-oriented learning material. Each chapter focuses on a specific set of data analysis skills and introduces these through examples based on datasets commonly encountered in the plant sciences. The book does not aim to be a comprehensive reference for R, but instead concentrates on methods and techniques that are essential for exploratory data analysis.

Throughout the book, code examples are used to demonstrate standard data analysis workflows.

Learning how to program in R

Programming and data analysis are not spectator activities. To develop practical skills, it is essential to actively work through the examples in this book and to apply the demonstrated techniques to new datasets. By writing, running, and modifying code yourself, you will build the proficiency needed to use R effectively and to carry out reproducible data analyses.

Just like the course, the book does not assume any prior knowledge on programming skills or working with R. To gradually build up to a level of independent data analysis, the book is structured into four main sections, aligning with the four weeks of the course (Figure 1).

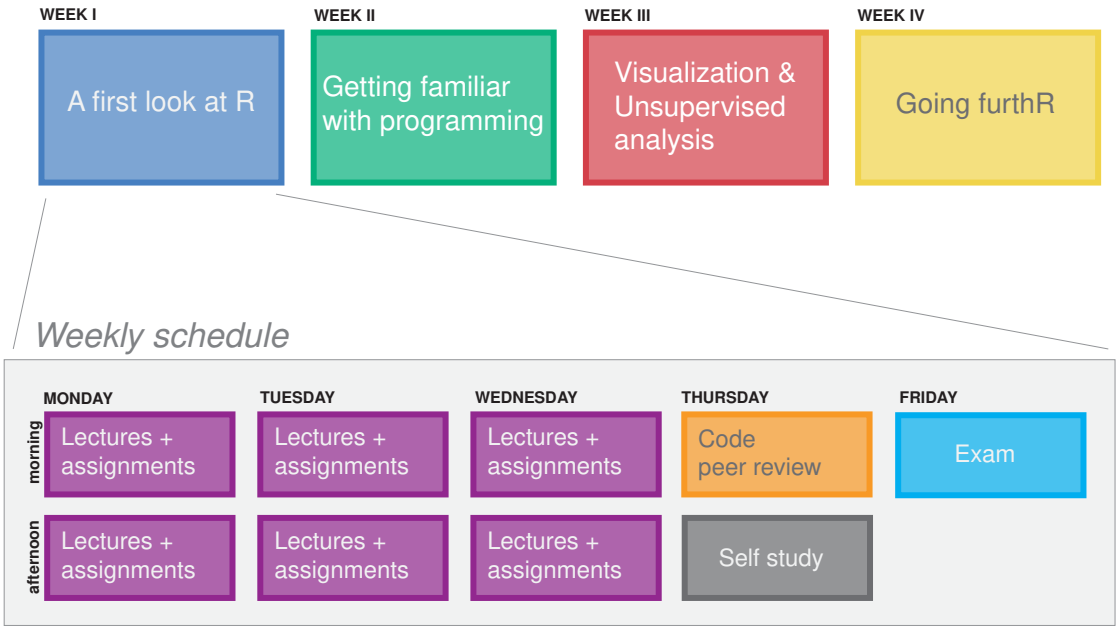


Figure 1: Course overview

Reading guide

Throughout the book, you will find a mix of examples, stories, theory, and assignments. All the study material for the course is available in the book, lectures are intended to solidify concepts and provide additional motivation and examples. This means that by completing the material of one section of the book and carefully reviewing the lecture slides, you should be able to successfully perform the graded assignment of the accompanying week.

To provide additional structure to the material in the book, you will find various 'boxes': these are intended to either highlight important information, or provide additional background material.

Part I

I: A first look at R

The goals for week 1

In this course you will learn how to conduct data analysis in R. As such, the assignments in this book are meant to take you through the steps of data analysis. Furthermore, it introduces concepts and ways to work with R.

In the first week, the goals are:

1. Acquire basic knowledge on using R and R studio
2. Recognize and load common data formats
3. Apply common statistical tools for inspecting and analysing data
4. Document your code in a clear & concise way.
5. Apply the data-science cycle (load, inspect, clean, analyse, present)

> **Tip 1: What is the 'data-science cycle'?**
>
> You are familiar with collecting data, even with some analysis and interpretation. For instance, in Plant Science in Practice you gathered a dataset on wild plants via fieldwork and in Reproduction of Plants you carried out a small experiment in a laboratory setting. During lab work you follow a particular protocol, and it is no different for data-analysis. We use an iterative cycle of loading, inspecting, cleaning, analyzing, and presenting data. It is normal to go back and forth in the cycle. The main assumption is that any data set is imperfect and has issues. In the first week, we will not bother with imperfection in what you use, but we will discuss it.

How week 1 is organized

The material you are supposed to work through each day is a chapter in the book. During the scheduled tutorials background and context will be given. Also, there will be help available if you get stuck with the coding.

There will be three days of working through the chapters (Monday - Wednesday). The exercises in this book require you to use data. In the first week, this data is available via a link on Brightspace. For each day of the course, there is a data set available.

1. For Chapter 1 we all work on the same dataset[11].
2. For Chapter 2 you have your own, personal dataset based on [9]. This data set you find in a folder under your own name.
3. For Chapter 3, we start with a shared dataset[13]. For the assignment, you will also have your own, personal data set[13]. This data set you find in a folder under your own name.

The answers to the assignments will be posted on Brightspace the morning after (so the answers for Chapter 1 will appear on Tuesday).

Next to the chapters, there is some (light) reading to support your view on data analysis. After Chapter 2 you are expected to read a paper by Itai Yanai and Martin Lercher[14]. After Chapter 3 you are expected to read a critique on this paper by Teppo Felin *et al.*[4]. These two papers should give you a firm grip on hypothesis testing and data analysis.

The assignment

At the end of Wednesday (23:59), a coding peer-feedback assignment is due, which you submit via feedback fruits on Brightspace. Your assignment will be reviewed by two students, as you will review the assignment of two other students. The instructions for this assignment you find in Chapter 3. **Completing the assignment and participation in the code-review is mandatory.**

You need to hand in a .html file (generated from a .qmd file). Practice with generating these types of files *before* the assignment is due. Ideally when the tutorials are given.

The code-review (peer-feedback via Brightspace) is due Thursday at 13:00. Instructions for how to give feedback can be found in Chapter 3. The items we expect you to give feedback on as well.

The exam

The exam of week 1 **does not count for your final grade of the course**. It does however show how the exams in this course go, and you will also receive feedback and a grade (again, it will not count) on what you hand in. The assignment on Wednesday will prepare you for the exam on Friday.

At the end of week 1 we expect you to be able to:

- Complete a given .qmd file with answers and completed code-blocks;
- Start a data-analysis project in R (set a work directory, install and activate packages);
- Load the data formats covered in week 1 (.tsv, .csv, .xlsx, .Rdata);
- perform and interpret the outcome of basic checks on loaded data using functions (e.g. `dim()`, `ncol()`, `summary()`, ...);
- Combine objects that in essence fit together (`cbind()`, `rbind()`);
- Test data for normality (e.g. by making a qqplot or by `shapiro.test()`);
- Conduct a t-test (`t.test()`);
- Conduct a Wilcoxon rank sum test (`wilcox.test()`);
- Be familiar with correlation and clustering;
- Translate a p-value to a biological interpretation;

- Complete {ggplot2} code to make a histogram, a qqplot, a boxplot, and a scatterplot;
- Be able read a boxplot, qqplot, a scatterplot, and a histogram and translate these figures to a biological interpretation.

1 The importance of data analysis

1.1 The importance of data analysis

1.1.1 Introduction to day 1

Today we will focus on goal 1 (Built basic knowledge on using R and R studio) and goal 5 [Apply the data-science cycle (load, inspect, ~~clean~~, analyse, present)].

In order to illustrate some concepts, we will make use of data from a paper on nematode infections in *Arabidopsis thaliana*[11].

M. incognita infections in *A. thaliana*

Plant susceptibility to nematodes is a complex trait, with a lot of variation in susceptibility between individuals of the same species. At the Laboratory of Nematology, we study this plant-property. The study by Sonja Warmerdam (who, before her PhD, studied plant science)[11] was conducted on 340 ecotypes of the model plant *A. thaliana*. These were infected with the tropical root knot nematode *Meloidogyne incognita*. This plant-parasitic nematode is the cause of a major agricultural burden globally[5]. We will use data from the initial screening (that took over 1.5 years) that was conducted for this research project.

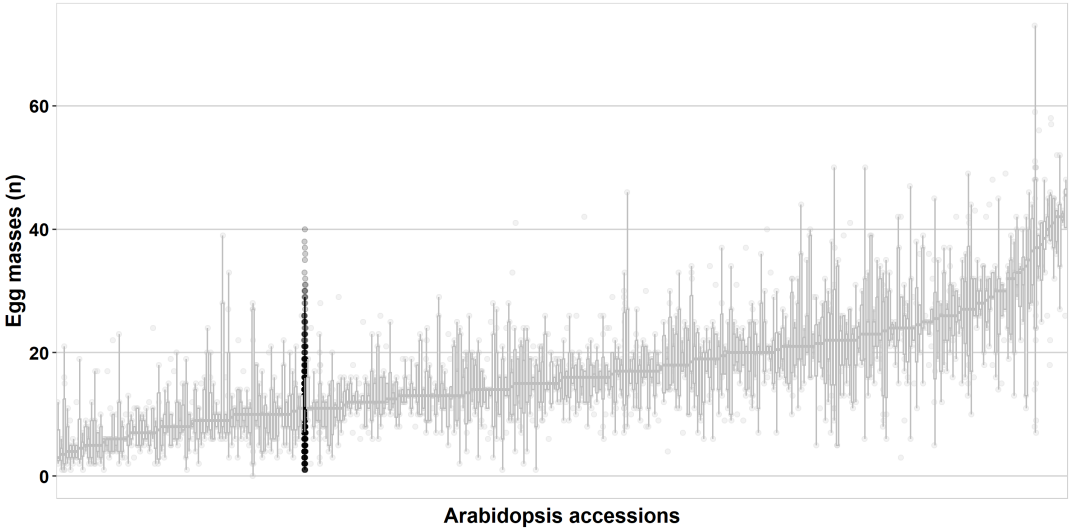


Figure 1.1: Quantitative variation in susceptibility of *Arabidopsis thaliana* to the root-knot nematode *Meloidogyne incognita*. Boxplot of the number of egg masses per plant. The thick bar indicates the median and the box indicates the 25 and 50 percent quantiles. The dots represent the individual observations on 340 ecotypes of Arabidopsis (accessions) at 6 wk after inoculation with 2nd stage juveniles of *M. incognita*. The black boxplot and dots indicate the number of egg masses per plant for Col-0, which was used as a reference throughout this study.

As you gathered by now, this is a course in which we will use the versatile, statistical programming language R. We will teach you how to use R via RStudio. In particular we will teach you how to write and document code for data analysis. By the end of this week, you will have made your first steps towards making documents as you are reading now: Quarto markdown.

1.1.2 Set up your R environment

In this section of the book, the basics of setting up RStudio are explained. You are welcome to revisit this section any day of the course.

A short intro to RStudio and scripts

When you open RStudio, you see four panes.

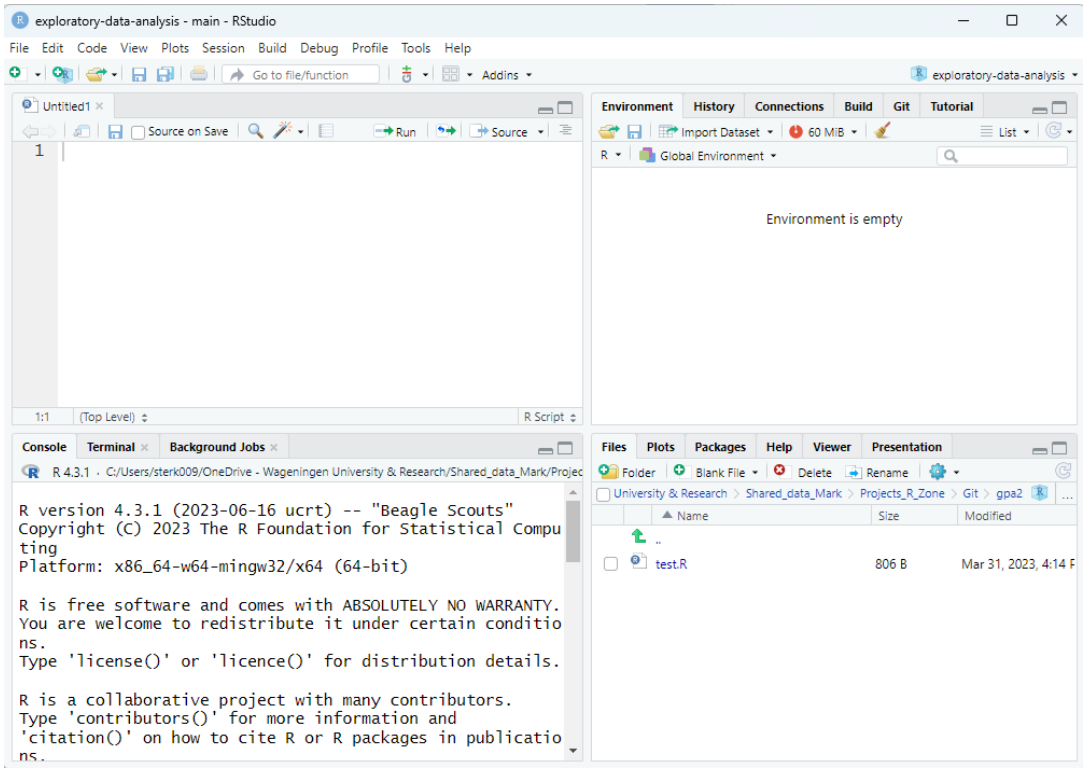


Figure 1.2: The Rstudio layout with four panes: (1) On the top-left a text editor to write and document your code in. (2) On the top-right a pane that - for instance - can show which objects are available in the environment. (3) On the bottom-left the console which can execute your code. (4) On the bottom-right a pane where, among others, you can: browse through files, plots, and get an overview of packages.

Now, check the message you see in the console.The R version displayed is important.

Exercise 1.1 (Getting to grips with R)**.** Start RStudio and browse through to find out: 1. Which version of R are you running? 2. Where can you find which packages are activated?

3. Which version of the stats package is active? 4. Where on your Harddrive is the current (default) working directory?

The scripts you are using the first week should work with R versions 4.3 and onwards. If your version number is lower, please first shut-down RStudio and install a newer version of R. You can find out how to do so here. RStudio should automatically pick up on the newer installed version. You can check if this worked by checking the version number of R after installation.

It is generally advisable to install a new version of R at the start of a new project (e.g. BSc thesis).

1.1.2.1 Documenting your code

Your analysis starts with making a file to document your progress and thereafter you follow the standard steps in data analysis

1. Loading
2. Inspecting
3. Cleaning
4. Analyzing
5. Presenting

For the tutorials, assignments, and exams, it's important that you built a file documenting your code. On the first two days, we will provide a full-scale template for you, that you can find here. This Quarto markdown template you can open into the text editor in R-studio.

Exercise 1.2 (Prepare your documentation)**.** Open the Quarto markdown template in RStudio and fill in your own information.

This type of document is also what you will make for the assignments. Furthermore, you will also make your exam based on this template.

1.1.2.2 Telling R where on the hard drive it should work from.

First, you need to set a work directory (tell R where on the hard drive you'll be working from). This can be done with the function `setwd()`, where you have to fill in the work directory yourself. If you want to know how to use that function type `?setwd()` into the console.

Exercise 1.3 (Set your work directory)**.** Run the script below to get information on the `setwd()` function. thereafeter, use it to set the location where R should work from. It is advisable to make a folder on your HDD for items related to this course.

```
# Set a work directory
?setwd()

###Now type the function to set a location on your HDD to work from
```

Alternatively, you can also use the three dots ... in the top-right corner of the files pane of RStudio to navigate towards the location you will be working from.

1.1.2.3 Installing packages.

Typically R-users use several 'packages'. These packages, give extra functionality to R. The main repositories are:

1. CRAN
2. Bioconductor

In this tutorial we will be making use of various packages. During week 1 we will make use of the {readxl} and {tidyverse} package each day, including during the exam. By the end of this first week, we expect you are able to install and activate packages.

Exercise 1.4 (Install packages)**.** Install the {readxl} and {tidyverse} packages.

```
# Install the package
install.packages("readxl")    # For reading Excel files
install.packages("tidyverse") # For wrangling data and generating plots
```

Note that you only need to install a package once, and only re-install it if the package has a new version or you install a new version of R.

Exercise 1.5 (Activate packages)**.** To activate the package, we need to load it into memory:

```
# Activate the package
library("readxl")
library("tidyverse")
```

Now you are all set to do data-analysis in R!

1.2 Analysis of quantitative phenotypic data from an experiment

1.2.1 Loading data into R

During this course, we will load different types of data files into R. This is an essential skill for working with real-world data. In the following days we'll cover a couple of file-types that are common, especially tomorrow, (Chapter 2 we will cover file-types.

For today, we have provided you with data in tab-delimited text format. Download the data via fileshare link on Brightspace and save it in your course folder.

Load this data using the function `read.delim()`. You see the arrow <-, this indicates that an object is defined within the R environment that will hold the loaded data. In general, you can find information about basic operators in the appendices (base-r-cheatsheet). For the remainder of this chapter, we will continue working on this object `pheno_data`.

```
# Maybe you need to add the path to your tab-delimited file
pheno_data <- read.delim("./BIF20806_Warmerdam_dataset.txt")
```

1.2.2 Inspecting data

It is always important to check your data, this can be done using specific functions or via plots. Always check what you did; when you load in data, check if it conforms to what you expect.

1.2.2.1 Using functions

There are a couple of handy functions you should use to check the data you loaded: `head()`, `tail()`, and `summarise()`.

```
# Check the first 6 rows
head(pheno_data)

# Check the last 6 rows
tail(pheno_data)

# get a summary of the data (per column)
summary(pheno_data)
```

As you see above, these functions give a good overview of the dataset. It is important to do these basic checks in order to confirm that your data is in the right shape.

You can also specifically check items in an object by using `[]`, also see (base-r-cheatsheet).

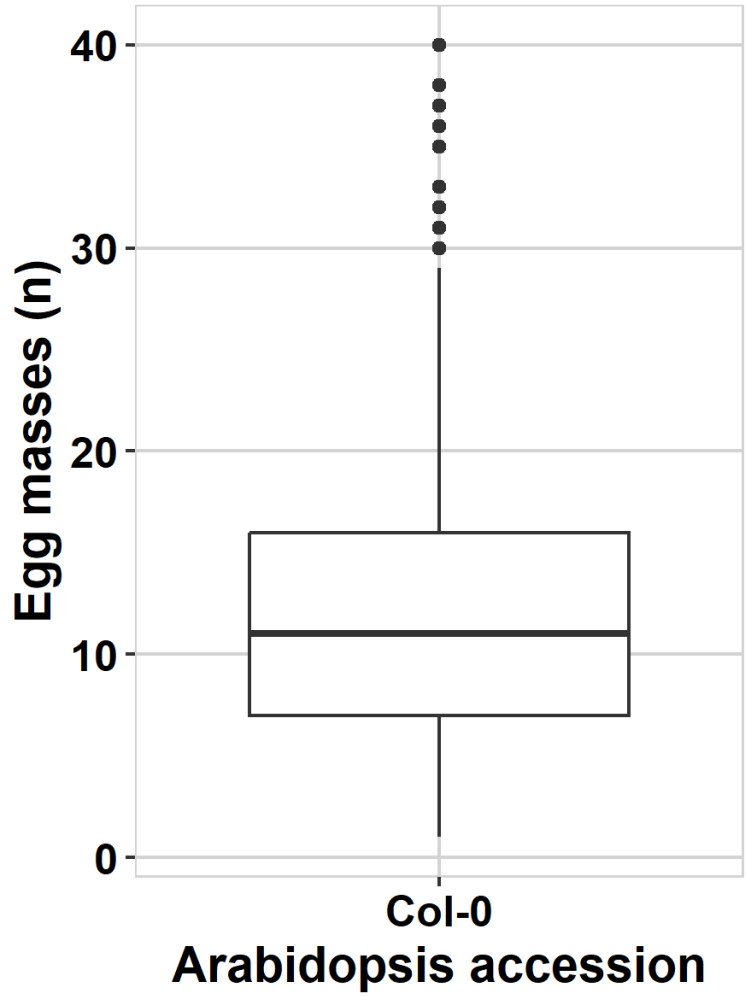
```
# The 18th row
pheno_data[18,]

# The 3rd column
pheno_data[,3]

# Or combined the 18th row of the 3rd column
pheno_data[18,3]

# the 18th value of the Line column; as a vector is one-dimensional only 1 coordinate is
pheno_data$Line[18]
```

1.2.2.2 Inspect and clean data using ggplot2 and tidyverse

To actually clean data, you need to inspect it (take a peak). I typically make a couple of plots. For example, here using ggplot (comes with the tidyverse package). You can read more about the plotting functions on their website

The code below produces a boxplot, great for checking continuous data. A box-plot displays various summary statistics of the data (more on that next week in Chapter 4).

Boxplot



The thick line in the middle of the box represents the median of the data (the value of the number in the middle when all values are placed in order), the outer lines of the box represent the 25% and 75% quantiles (the values of the numbers at 25 and 75 percent of all values). The lines sticking out of the box ('whiskers') represent the values covering ~95% of the data. If you see dots beyond these whiskers, these represent extreme values within the dataset.

Now produce the boxplot of all data from the paper[11].

```
data.plot <- pheno_data

ggplot2::ggplot(data.plot,aes(x=genotype,y=eggmass)) +
```

```
    geom_boxplot()
```

I notice in the plot that there are a couple of single points (the horizontal stripes) per strain, also, there seem to be a couple of really high values. Let's check these.

Now, let's have a look at the distribution of the data.

Exercise 1.6 (Histogram)**.** Try to plot a histogram, for that you need to adjust the boxplot code above a bit. You need to use the `geom_histogram()` function. Use the ggplot2 website to figure out how to make this plot.

```
    data.plot <- pheno_data

    ggplot2::ggplot(data.plot) + aes() + ###select the right set of values to make the plot
    geom_ ###add the right geom
```

1.2.2.3 Use table to inspect observation per genotype

For the single values per genotype, it can be handy to use the `table()` function. Try to use this function on the column *genotype*. You can use `pheno_data$genotype` to access that column.

```
    ### this way you can figure out how the function works
    ?table()

    table(pheno_data$genotype)
```

This way, you can check the number of observations per genotype.

Exercise 1.7 (Make a table)**.** Now use `table()` to check the Line names. For this you need to indicate the column that is accessed

```
    table(pheno_data) ### you need to indicate the column you access
```

1.2.3 Cleaning and preparing the data

In this example, all data is clean and we will not spend time cleaning it up. But it is important to realize that there typically are errors and inconsistencies in datasets. Figuring these out and fixing them is a large chunk of doing data analysis.

1.2.4 Analyzing

1.2.4.1 Normal distribution

Let's figure out whether the data is normally distributed. This is important for determining which kinds of statistical tests can be used.

For this we can use the functions `stat_qq()` and `stat_qq_line()`.

```
data.plot <- pheno_data

ggplot(pheno_data,aes(sample=eggmass)) +
stat_qq() + stat_qq_line()
```

Alternatively, you can use a Shapiro-Wilk test (`shapiro.test`), this tests the deviation from normality.

```
data.test <- pheno_data

shapiro.test(data.test$eggmass)
```

Exercise 1.8 (Does the data follow a normal distribution?)**.** You now conducted two types of analyses that tell you whether the data is normally distributed. (1) what do you see in the qqplot, and what does it mean? (2) what is the p-value from the Shapiro-Wilk test? (3) what does a p-value mean?

1.2.4.2 T-test

Now we can use a t-test to compare between two samples. For that, we first need to generate a smaller set of genotypes. We want to test a genotype versus the *A. thaliana* reference genotype Col-0. Let's take the also popular Cvi-0 genotype to compare.

Using a tidyverse pipeline

In the lines below, we first group (`dplyr::group()`) the batches together (screenings were done at a particular date), then we keep only the screenings with Col-0 and Cvi-0 in them (`dplyr::filter()`). Thereafter we remove the grouping (`dplyr::ungroup()`) and keep only the Col-0 and Cvi-0 observations (again with `dplyr::filter()`).
You may notice that at the end of these lines you find the %>% operator. This operator moves the output of the pervious line into the function on the next line. The code you see below follows a manner of programming that distinct from programming in base-R. Namely, it uses the {tidyverse} packages. More information about this will follow later (and see appendix on tidyverse).

```
data.test <- dplyr::group_by(pheno_data,screening) %>%
             dplyr::filter(("Col-0" %in% Line & "Cvi-0" %in% Line)) %>%
             dplyr::ungroup() %>%
             dplyr::filter(Line %in% c("Col-0","Cvi-0"))
```

Now the data is filtered, we can test. See if you can test whether these two *A. thaliana* genotypes are different in their susceptiblity to *Meloidogyne incognita*. Use `t.test()`

```
t.test(eggmass~Line,data=data.test)
###you should have p-value = 2.588e-06
```

Exercise 1.9 (How to interpret a p-value?)**.** You now have tested a null-hypothesis and found that the chance his data agrees with this null-hypothesis is very small. What is the *biological interpretation* of this p-value?

1.2.5 Presenting

Can you now also make a plot of what you just tested?

1.2.5.1 Boxplot

Use `geom_boxplot()` to visualize the distribution and `geom_jitter()` to show the individual measurements.

```
ggplot(data.test,aes(x=Line,y=eggmass)) +
geom_boxplot() + geom_jitter(height=0,width=0.25)
```

We can make it a bit fancier by plotting the p-value as well. Try to include it into the plot by using the function `annotate()`. We use `broom::tidy()` for convenience, {broom} is also included in the {tidyverse}.

To make the fancy plot, we add statistics with `ggplot2::annotate()`. Also, we've added a layout with `theme_bw()` and axis labels with `ggplot2::labs()`. The function `expression()` allows printing of cursive text and ^{super} and $_{sub}$ script within {ggplot2} axis labels.

```
### Fancier boxplot
    ###I'm getting the statistics here
    statplot <- broom::tidy(t.test(eggmass~Line,data=data.test))

    p1 <- ggplot(data.test,aes(x=Line, y=eggmass)) +
          geom_boxplot() +
          geom_jitter(height=0, width=0.25) +
          theme_bw() +
          annotate("text", x=1.5, y=max(data.test$eggmass),
                   label=paste("p =", signif(statplot$p.value, 2))) +
          labs(x=expression(bolditalic("A. thaliana")~bold("genotype")),
               y=expression(bolditalic("M. incognita")~bold("eggmasses (n)")))

    p1
```

1.2.5.2 Saving plots

Now you have your plot, you need to save it. If you render it within the notebook you are keeping, the plot will be shown when you hit 'render' (you need pandoc installed to be able to do this).

Most of the time, you will make a figure that you can share later.There are multiple 'graphical devices' in `R` that you can use to plot.

The function I use most often is `pdf()`. The reason is that this function produces vectorized pdf's, which means that figures have infinite resolution. When you want to save your plot using this function, either save the figure in a file (as in the previous section), or plot directly within the 'graphical device', if you use {ggplot2}, you often have to explicitly define that you print the plot, by adding the function `print()`. Your entire plotting code than should be included in the brackets.

I also need to define how wide and high I want to make the plot (in inches). I play around with the dimensions a lot, to get the figure in the right proportions. Note that the plotting is ended with `dev.off()`. This shuts down the graphical device. If you do not do that, it remains open and you will not see any plots appear on your plotting window in RStudio.

Exercise 1.10 (Save the plot)**.** Now save the plot running the code below.

```
pdf(file="Figure_Col0-Cvi0.pdf", width=3, height=4)
    print(p1)
dev.off()
```

Confirm that you now have a file on your HDD, within the folder you set as your working directory.

1.2.6 A for-loop

The basics of a loop

Within the data there are 332 unique genotypes. Verify this using the function `unique()` to get the unique genotypes and `length()` to count.

```
genotypes <- unique(pheno_data$Line)

length(genotypes)
```

What if we want to plot all of these versus Col-0? Writing all these plots line-by-line is not doable. For this we can use a for-loop. with the function `for()` you can start a loop. The `for()` function counts along a variable. There are also other loops possible, for instance with `while()` that continues if a condition is met. Let's first look at a simple for-loop.

```
for(i in 1:30){
    cat("",i)
}
```

That probably went too fast, but it printed the values 1:30 in your console. You can loop over any vector:

```
for(i in c("N. americanus","E. vermicularis","M. incognita","A. suum",
           "T. semipenetrans","O. volvulvus","D. dipsaci","E. arcuatus")){
    print(i)
}
```

A 'cleaner' way to write this (at least in our opinion) is to loop over the index number in the vector with species names `1:8` and then select the species for printing.

```
species <- c("N. americanus","E. vermicularis","M. incognita","A. suum",
           "T. semipenetrans","O. volvulvus","D. dipsaci","E. arcuatus")

for(i in 1:length(species)){
    print(species[i])
}
```

Use this to test any *A. thaliana* genotype against Col-0.

1.2.6.1 For-loop over t-tests

Let's automatize the t-test function! This is a bit simpler than plotting. For this, you need to make a for-loop cycling over the genoypes in `pheno_data`. You can remove Col-0 from the vector using `genotypes <- genotypes[genotypes != "Col-0"]`.

Exercise 1.11. Now, with what you read above, write a for-loop doing the t-test of each genotype versus Col-0 and make sure to pint the values in the console (either use `cat()` or `print()`).

For that you need to: (1) replace AAA with the genotype you want to test. Hint: use [i] to access the current genotype. (2) save the p-value in the list. (3) print the p-val (see the `print()` or `cat()` functions above).

```
###Get the unique genotypes to test against Col-0
genotypes <- unique(pheno_data$Line)
genotypes <- genotypes[genotypes != "Col-0"]

for(i in 1:length(genotypes)){

    ### We select the data from one batch
    ### You need to replace AAA (2 times) with a selection of a genotype
    data.test <- dplyr::group_by(pheno_data,screening) %>%
                 dplyr::filter(("Col-0" %in% Line & AAA %in% Line)) %>%
                 dplyr::ungroup() %>%
                 dplyr::filter(Line %in% c("Col-0",AAA))

    ### conduct test and print the p-value
    ### you can select the p-value in the t-test output with $
    pval <- t.test(eggmass~Line,data=data.test)$p.value ### save it into an object

    ### print the p-value

}
```

1.2.6.2 For loop for plotting

Now we can add a plot, for that we need to save the plots in an object (like we did with p1 before). Now, we will make a specific type of object: a `list`. Note that this `list` is not what you think of as a list.

Lists

`Lists` are objects that can contain various other types of objects. For instance, I can place a `vector` or a `dataframe` into a `list`. You can access a location in a list using `[[]]`.

```
###this makes an empty list
output <- as.list(NULL)

###you can add the first item
output[[1]] <- c("N. americanus","E. vermicularis","M. incognita","A. suum",
                 "T. semipenetrans","O. volvulvus","D. dipsaci","E. arcuatus")

###and a second item
output[[2]] <- matrix(runif(100),ncol=4)
```

As we saw before, you can also access specific locations within these items, simply by adding a `[]` on top of the `[[]]`.

```
# The 18th row
output[[2]][18,]

# The 3rd column
output[[2]][,3]

# Or combined the 18th row of the 3rd column
output[[2]][18,3]

# the 8th value of the Line column; as a vector is one-dimensional only 1 coordinate
output[[1]][8]
```

We can also write data into a list as output of a for-loop.

```
###this makes an empty list
output <- as.list(NULL)

###run a for-loop and save within ouput
for(i in 1:30){
    output[[i]] <- i^2
}
```

Exercise 1.12. Now take the plotting function we wrote before and use it to save plots into an object. Note that this takes a while to run (~2 minutes). For this you need to save

the plots into the list. See above for how to use a list.

```
###Get the unique genotypes to test against Col-0
genotypes <- unique(pheno_data$Line)
genotypes <- genotypes[genotypes != "Col-0"]

###this makes an empty list
output <- as.list(NULL)
for(i in 1:length(genotypes)){

    ### We select the data from one batch
    ### The mutate on the final line of this chunk make sure that Col-0 is always on the
    data.test <- dplyr::group_by(pheno_data,screening) %>%
                 dplyr::filter(("Col-0" %in% Line & genotypes[i] %in% Line)) %>%
                 dplyr::ungroup() %>%
                 dplyr::filter(Line %in% c("Col-0",genotypes[i])) %>%
                 dplyr::mutate(Line = factor(Line,levels=c("Col-0",genotypes[i])))

    ### conduct test and clean the result with broom::tidy
    statplot <- broom::tidy(t.test(eggmass~Line,data=data.test))

    ### Make the plot into the list ('output')
    ###note that the object you need to write into is missing.
     <- ggplot(data.test,aes(x=Line, y=eggmass)) +
          geom_boxplot() +
          geom_jitter(height=0, width=0.25) +
          theme_bw() +
          annotate("text", x=1.5, y=max(data.test$eggmass),
                   label=paste("p =", signif(statplot$p.value, 2))) +
          labs(x=expression(bolditalic("A. thaliana")~bold("genotype")),
               y=expression(bolditalic("M. incognita")~bold("eggmasses (n)")))

    }
```

1.2.6.3 Save the plots

We can save the plots in various ways. We can write one big .pdf that contains all the plots.

```
###One big happy pdf
pdf(file="Figure_All_Col0-comparisons.pdf",width=3,height=4)
    for(i in 1:length(output)){
```

```
        print(output[[i]])
    }
dev.off()
```

Exercise 1.13 (Save individual plots)**.** Complete the code below, so you can make a .png of each plot individually.

```
### Here include the for-loop
    png(file=paste("Figure_Col0-",genotypes[i],".png",sep=""),width=3,height=4,units="in"
        print(output[[i]])
    dev.off()
### Here, close the for-loop
```

2 From messy to clean

2.1 From messy to clean

2.1.1 Introduction to day 2

Today we will continue with goal 1 (Built basic knowledge on using R and R studio) and start with goal 2 (Be able to recognize and load common data formats) and goal 3 (Apply common statistical tools for inspecting and analyzing data)

We will make use of a fragmented simulated dataset. You need to puzzle it together using a couple of functions. Therafter you can analyze it further.

2.1.2 Set up your R environment

For the basics of setting up your environment, you can check Chapter 1. The Markdown file to use for the exercises and documentation of your code is here.

2.1.3 Loading data into R

In this tutorial, we will start with loading different types of data files into R. This is an essential skill for working with real-world data. We'll cover a couple of file-types that are common. *Each of you has their data in these formats*, you can find these via the fileshare on Brightspace. If you are familiar with R, this part might be easy.

- Excel files (`.xlsx`)
- Comma-separated values (`.csv`)
- Semicolon-separated CSV files (`.csv`)
- Tab-delimited text files (`.txt` or `.tsv`) (you encountered this filetype yesterday)

To make this part of the course realistic, *we have provided you with data in these four formats*. The reason being that this is also how it works in practice. Finding out the format of the data and loading it into memory is a crucial step in using any data-analysis pipeline.

You now need to load in data from all four data formats.

Exercise 2.1 (Loading data from an excel file (.xlsx))**.** To read Excel files, we use the `read_excel()` function from the readxl package. Excel files are very common but also very tricky. Namely, these files are mostly made by persons and people *do not automatically follow conventions* that are clear for a programming language. Therefore, always check the excel file before loading it in.

```
### Package
install.packages("readxl")    # For reading Excel files
library(readxl) # first activate the package

# Replace with the path to your Excel file
data_excel <- readxl::read_excel("data/YOURFILENAME.xlsx")

### Check after loading
head(data_excel)
```

Exercise 2.2 (Loading data from a CSV File (Comma-separated))**.** CSV files are very common as format for data. Here each column is separated by a comma, rows are typically observations. We use the `read.csv()` function to load the data.

```
# Replace with the path to your csv file
data_csv <- read.csv("data/YOURFILENAME.csv")

### Check after loading
head(data_csv)
```

Exercise 2.3 (Loading data from a CSV file (semicolon-separated))**.** Some CSV files (especially from European systems) use a semicolon (;) instead of a comma. Use `read.csv2()` for these files

```
# Replace with the path to your csv file
data_csv2 <- read.csv2("data/YOURFILENAME.csv")

### Check after loading
head(data_csv2)
```

Exercise 2.4 (Loading a Tab-delimited File)**.** These files use tabs to separate values. They often have a .txt or .tsv extension.

```
# Replace with the path to your csv file
data_tsv <- read.delim("data/YOURFILENAME.txt")

### Check after loading
head(data_tsv)
```

2.2 Analysis of (a)virulence of *Globodera pallida*

This set of instructions follows the data analysis cycle, so it also includes setting up the R environment as well as loading in the data. If all is well, you already prepared this when going through the code above.

2.2.1 Setup R

Before we start, make sure you have set the work directory and have the following packages ready: {tidyverse} and {readxl}. Of course you also need to activate them. If you're not sure how, check Chapter 1.

```
### type the code to install the packages

### type the code to activate the packages
```

2.2.2 Load data

Load your data for the tutorial into R, see the exercise above.

2.2.3 Inspecting data part 1

2.2.3.1 Using functions

Last time we used `head()`, `tail()`, and `summarise()`. Again apply these functions. But now we also want to check which piece is which. For that the functions `length`, `ncol()`, `nrow()`, and `dim()` are handy.

```
# Check the first 6 rows
head(data_csv)

# Check the last 6 rows
tail(data_csv)

# get a summary of the data
summary(data_csv)

# get the length of a vector
length(data_csv)

# get the number of columns
ncol(data_csv)

# get the number of rows
nrow(data_csv)

# get the dimensions of an object
dim(data_csv)
```

Exercise 2.5 (Which file is which puzzle piece?)**.** Inspect the four files you have using `dim()`, `nrow()`, and `ncol()`. Which piece is which?

```
###Type the code to check these three properties of each file
```

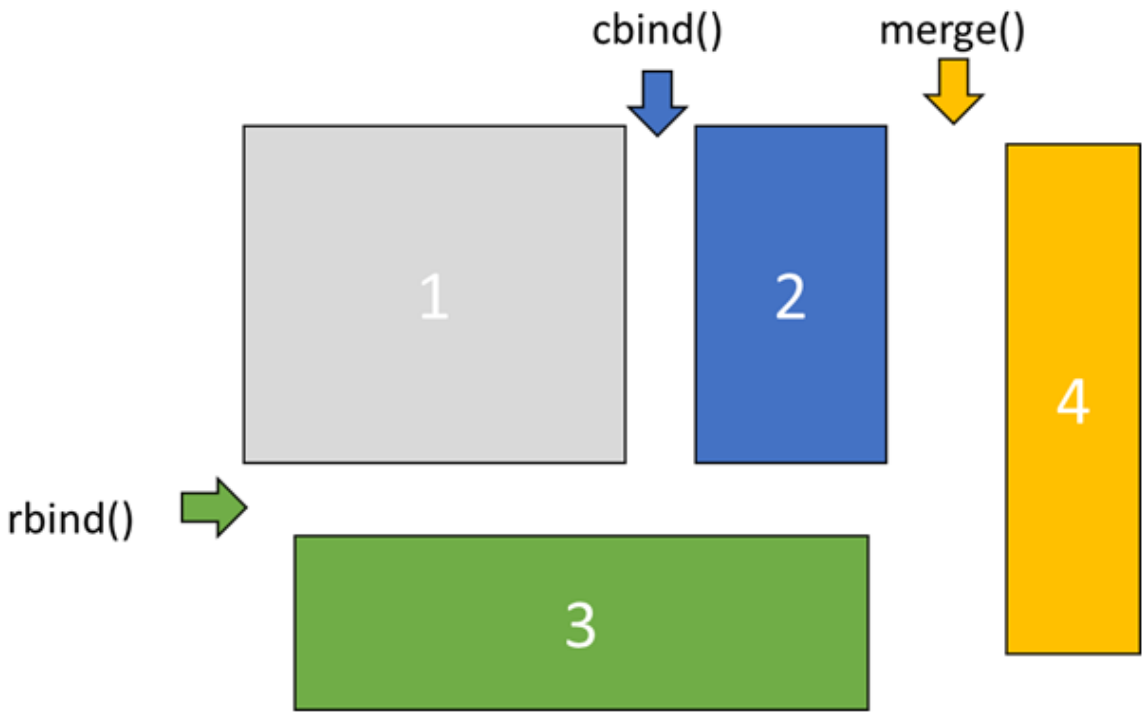


Figure 2.3: The puzzle pieces and the functions you will need to combine them further on in the excercise.

Answer the following questions (note this is *random* for each personal dataset!). You can deduce which piece is which via their dimensions.

1. Which 2 files are piece 1 and 2?
2. which file is piece 3?
3. Which file is piece 4?

2.2.4 Cleaning and preparing the data

Now you have the dimensions of the puzzle pieces. You can combine them in only one way.

Exercise 2.6 (Combine the data)**.** First you need to join the two pieces (1 and 2) with the equal number of rows using `cbind()`. Second, you need to add the piece (3) that now has

the same number of columns using `rbind()`. Finally, you need to add the last piece (4) of data using `merge()`.

Complete the code below

```
### bind objects together over columns
pheno_data <- cbind(YOURPIECE1,YOURPIECE2)

### binds objects together over rows
pheno_data <- rbind(pheno_data,YOURPIECE3)

### merges objects, seeks for the same identitier, or you can provide it using by.x and
pheno_data <- merge(pheno_data,YOURPIECE4)
```

2.2.4.1 Inspecting data using histograms

After combining, it is smart to inspect the data again to check if everything went well. You can use the same functions as before or, for instance, use `geom_histogram()`.

```
ggplot2::ggplot(pheno_data,aes(x=Gpal_vir)) +
geom_histogram()

ggplot2::ggplot(pheno_data,aes(x=Gpal_avir)) +
geom_histogram()
```

2.2.5 Analyzing

You now have the data complete. As you can see we have two measurements linked to one plant ID. The reason is that this data represents various inbred lines of potato plants tested for both virulent and avirulent *G. pallida* populations.

2.2.5.1 Normal distribution

Same as before, let's figure out whether the data is normally distributed. For this we can use the functions `stat_qq()` and `stat_qq_line()`. Find the code to do this in Chapter 1 and check the data in the `Gpal_vir` and `Gpal_avir` columns.

```
### Add the code to make the qqplot for Gpal_vir

### Add the code to make the qqplot for Gpal_avir
```

Of course, you can also use a Shapiro-Wilk test (`shapiro.test`), this calculates the likelyhood whether the data follows a normal distribution.

```
shapiro.test(pheno_data$Gpal_vir)

shapiro.test(pheno_data$Gpal_avir)
```

Exercise 2.7 (Does the data follow a normal distribution?)**.** You now conducted two types of analyses that tell you whether the data is normally distributed. (1) what do you see in the qqplots, and what does it mean? (2) what are the p-values from the Shapiro-Wilk test? (3) what does a p-value mean?

2.2.5.2 Wilcoxon test

As the data is not exactly normally distributed, let's deal with that in the proper way. Now we can use a Wilcoxon Rank Sum test to compare between two samples. We first need to transform the dataset, as we do not have a single observation per row currently. For this you can use the `gather()` function from the {tidyr} package. Use functions like `head()` or `tail()` to check what happened.

```
data.test <- tidyr::gather(pheno_data,key="pallida",value="juveniles",-ID,-Experiment)
```

Now we can test how the potato plants performed against virulent versus avirulent *G. pallida*. Using `wilcox.test()`

```
wilcox.test(juveniles~pallida,data=data.test)
###you should have a very low p-value
```

Exercise 2.8 (How to interpret a p-value?)**.** You now have tested a null-hypothesis using a non-parametric test and found that the chance his data agrees with this null-hypothesis is very small. What is the *biological interpretation* of this p-value?

2.2.6 Presenting

2.2.6.1 Boxplot

Can you now also make a plot of what you just tested? Use `geom_boxplot()` to visualize the distribution and `geom_jitter()` to show the individual measurements.

```
ggplot(data.test,aes(x=pallida,y=juveniles)) +
geom_boxplot() + geom_jitter(height=0,width=0.25)
```

Exercise 2.9 (Boxplot with facet_grid()). There is another variable in the data as well: 'Experiment'. Use `facet_grid()` to split this out. Figure out how using the ggplot2 website.

```
ggplot(data.test,aes(x=pallida,y=juveniles)) +
geom_boxplot() + geom_jitter(height=0,width=0.25) +
facet_grid() ### add something in-between the brackets.
```

2.2.6.2 Fancier boxplot

Now we can add p-value as well using `annotate()`.

```
###I'm getting the statistics here
tmp <- tapply(data.test[,c("pallida","juveniles")],data.test$Experiment,function(x){
                            broom::tidy(wilcox.test(x$juveniles~x$pallida))})

### I need to include the facets
statplot <- do.call(rbind,tmp) %>%
            mutate(Experiment=1:3,pallida=1.5,juveniles=max(data.test$juveniles),
                   label=paste("p =",signif(p.value,2)))

###save the plot into an object
p1 <- ggplot(data.test,aes(x=pallida,y=juveniles)) +
      geom_boxplot() + geom_jitter(height=0,width=0.25) +
      facet_grid(~Experiment) + geom_text(aes(label=label),data=statplot)

###plot the plot
p1
```

Exercise 2.10 (Save plot). Now save the plot, check back at Chapter 1. Use a graphical device to save the plot with dimensions of 7 (width) by 4 (height) inch.

```
### use a graphical device (e.g. pdf())
    print(p1) ### we must use print, as we have a ggplot2 object
### close the graphical device
```

2.2.7 Inspecting data part 2

2.2.7.1 Correlation analysis

Correlation analysis is useful to discover patterns in large sets of data or between multiple measurements on the same sample (for instance different measurements on the same plant).

As the data is not exactly normally distributed, we also have to deal with that when calculating correlation. For normal (distribution) correlation, you should use the Pearson method. This is the standard setting for `cor()`. For data that is not normally distributed, the Spearman method should be used. To do this, you need to specify a parameter within a fuction. If you check the function `cor()` using the `?`, you see that you can specify `method`; `cor(x,y,method="Spearman")`.

```
data.test <- pheno_data

### Pearson correlation
cor(data.test$Gpal_vir,data.test$Gpal_avir)

### Spearman correlation
cor(data.test$Gpal_vir,data.test$Gpal_avir,method = "spearman")

### Test of correlation
### it is possible you get a warning here; this is not an error.
cor.test(data.test$Gpal_vir,data.test$Gpal_avir,method = "spearman")
```

Exercise 2.11 (Correlation analysis interpretation)**.** What do you conclude from the correlation analysis, are the two values correlated? What is the *biological* interpretation of this result?

2.2.7.2 Clustering analysis

Clustering is another method to explore data. Instead of patterns that move in the same direction, clustering is useful to apply if you expect consistent value differences in values.

For clustering we use the functions `dist()` and `hclust()`. The first function calculates the distance matrix, there are multiple methods to calculate the distance-matrix, here we use the standard method ("euclidean"). The second performs the clustering. This process groups the measurements that are most similar.

Because the dataset is too large to perform clustering on all the measurements, we summarise the data by taking the mean per experiment for the virulent and avirulent populations. To achieve this, we group the data per experiment using `dplyr::group_by()`, then we summarise using `dplyr::summarise()`. Then we need to transform the data to long format, for this we use `tidyr::gather()`. To see what happens, the code below includes a before and after inspection of the dataframe.

```
### To make the distance matrix, we first need to summarise the data
### otherwise the matrix becomes too large
```

```
data.test <- dplyr::group_by(pheno_data,Experiment) %>%
             dplyr::summarise(Gpal_vir=mean(Gpal_vir),Gpal_avir=mean(Gpal_avir))

###check the shape of the data before gather
head(data.test)

data.test <- tidyr::gather(data.test,key=population,value=reproduction,-Experiment)

###and after gather
head(data.test)


### Here we calculate the distance matrix
dist_mat <- dist(data.test$reproduction,diag=TRUE,upper=TRUE)

### to give understandable names to the distance matrix we re-write it as matrix
dist_mat <- as.matrix(dist_mat)
    ### these two lines replace the column and row names in the matrix
    colnames(dist_mat) <- paste(data.test$population,data.test$Experiment)
    rownames(dist_mat) <- paste(data.test$population,data.test$Experiment)
### to conduct clustering the matrix is re-written als distance matrix
dist_mat <- as.dist(dist_mat)

### here we use the base plot-function to look at the clustering
plot(hclust(dist_mat))
```

Exercise 2.12 (Clustering analysis)**.** What do you conclude from the clustering analysis, are there differences between the two *G. pallida* populations? What is the *biological* interpretation of this result?

2.2.7.3 heatmap

A handy visualization tool for correlation analysis as well as clustering analysis, is a heat map. There is a basic function in `R`, `heatmap()` that works with matrices. When you run the code below, you get a visualization of the distance matrix. The same can be done with a correlation matrix, but we do not use that today.

```
### To make the distance matrix, we first need to summarise the data
### otherwise the matrix becomes too large
data.test <- dplyr::group_by(pheno_data,Experiment) %>%
             dplyr::summarise(Gpal_vir=mean(Gpal_vir),Gpal_avir=mean(Gpal_avir)) %>%
             tidyr::gather(key=population,value=reproduction,-Experiment)
```

```
### Here we calculate the distance matrix
dist_mat <- dist(data.test$reproduction,diag=TRUE,upper=TRUE)

### to give understandable names to the distance matrix we re-write it as matrix
dist_mat <- as.matrix(dist_mat)
    ### these two lines replace the column and row names in the matrix
    colnames(dist_mat) <- paste(data.test$population,data.test$Experiment)
    rownames(dist_mat) <- paste(data.test$population,data.test$Experiment)

heatmap(dist_mat)
```

Exercise 2.13 (Plot of correlation)**.** Correlation can best be visualized with a scatter plot. complete the code below to generate a scatterplot of your *G. pallida* data. Add the correct column name for the y-axis and the right geom to plot all data points.

```
data.plot <- pheno_data

ggplot(data.plot, aes(x = Gpal_vir, y = AAA, colour=Experiment)) + ###replace the AAA wi
geom_ +  ### add the geom to make a scatter plot
scale_color_gradient(low="black",high="grey") + theme_bw()
```

2.2.8 Some reading

In part, this assignment was inspired by a paper in the journal Genome Biology, by Itai Yanai and Martin Lercher[14]. Read this paper as food for thought about what we covered today in p-value testing and its role in data analysis. This editorial paper is part of an interesting series of editorials about data science in biology: Night science[15].

3 Your digital notebook

3.1 Your digital notebook

3.1.1 Introduction to day 3

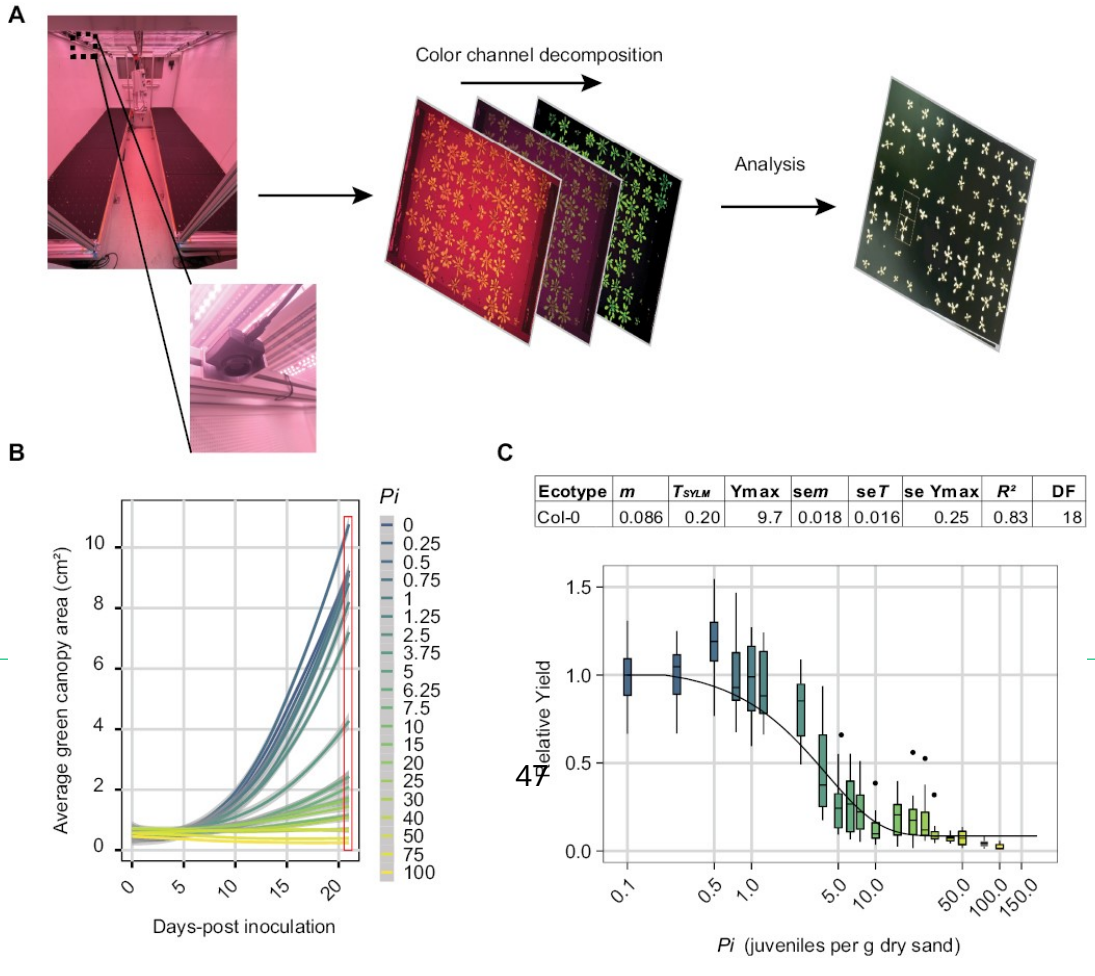
Today we will continue with goal 1 (Built basic knowledge on using R and R studio) and goal 3 (Apply common statistical tools for inspecting and analyzing data). We will also start with goal 4 (Be able to document your code in a clear & concise way). You will find out that we've already covered this by requiring you to write a script. Now we'll add one more layer to it, the notebook.

New today, is that you get an assignment. In this assignment, you are expected to go over goal 5 [Apply the data-science cycle (load, inspect, ~~clean~~, analyse, present)]. Also, this is not new, we've been doing it for the past two days.

As on the previous days, we will be making use of a real data-case.

Plant growth impairment by nematodes.

You already know that plant susceptibility to nematodes is a complex trait, with a lot of variation in susceptibility between individuals of the same species. This variation also exists for the dose-response relation between plants and nematodes. The data you will be using today is derived from a study by Jaap-Jan Willig, who used a camera platform to track the growth of 960 *A. thaliana* plants simultaneously[13].



Ecotype	*m*	T_{SYLM}	Ymax	se*m*	se*T*	se Ymax	R^2	DF
Col-0	0.086	0.20	9.7	0.018	0.016	0.25	0.83	18

Figure 3.1: The relationship between the inoculation density (P_i) of *H. schachtii* and

3.1.2 Set up your R environment

For the basics of setting up your environment, you can check Chapter 1.

3.1.3 Quarto markdown

3.1.3.1 The use of a markdown document

The last two days you have been working with .qmd files (Quarto markdown). This is a flavor of markdown, which is a markup language that relies on simple text to produce a layout of a document. As you have a lab-journal for experimental work in the field, greenhouse, or laboratory, a markdown notebook is useful to have when writing code.

The Quarto markdown file you make today for the assignment, should be handed in on Brightspace today, before the deadline. You hand in the .qmd file. Your reviewer will go through the file and check if it renders and whether your conclusions are correct.

Exercise 3.1 (Make your own markdown file)**.** In RStudio go to File > New File > Quarto document...

Based on the documents provided last two days (Chapter 1 and Chapter 2), make your own markdown file. Include meta information (title, name, registration number) and headings for: setup of the R environment (working directory, packages), and the five steps in the data analysis cycle (see Chapter 1).

You can choose whether it renders a .html or a .pdf document. For now, settle on .html, this typically renders well and does not require many additional programs. You probably will need pandoc.

You can find general tips on how to write in markdown in Appendix QMD notebook cheatsheet.

Typically, when you make a markdown file, you can make errors. RStudio tries to make you aware of these errors by putting a notification at the line numbers. But some errors you only catch when you render the document. Or when you run code line-by-line.
Many errors were made by me - and the other teachers - in building the book you are now reading. Do ask us about our frustrations in working with R.

3.2 Analysis of plant-growth data

Today we want to cover one more aspect of data analysis, namely including the fit of a simple model into a plot. In this case, we will use some plant-growth data. And to be precise, we will use the linear part of the growth curve. It is perfectly possible to fit a growth-model within R, but we want to start a bit simpler with an actual linear curve. you can find the data for this assignment via the fileshare on Brightspace. You need the general file under day 3, not your personal dataset for the code-review assignment.

Exercise 3.2 (Loading data into R)**.** Load the data using `load()`. We have not used this function before. Use `?load()` to figure out how this function works.

```
### here add the code to read the data
```

Note that the name of the object you loaded into memory is not the same as the name of the file you have on your hard disk. The reason is that the filename is inherited from my R-environment. Also note that this can be a major source of annoyance; namely we have been using object name `pheno_data` consistently. If you were to run `load()`, that would also mean you *overwrite* any other file with that name in the R environment.

Up till now, we have been avoiding talking about many of the programming aspects of working with R. But, grasshoppers, I'm afraid it is due.
There are some things you probably noticed by now: any wrong wording, comma's or other 'unintended mistakes' from your side are immediately recognized by R and rewarded with an 'error'. There are also so-called 'warnings'. The difference is that 'error' results in code that does not work (e.g. you will not get the specific calculation/processing executed) and a 'warning' means that you might have done something that was not intended (but you will get output). Next week in Chapter 5 we cover this more thoroughly.
The most important step in data-analysis is checking after an operation. Make sure that what you did was what you intended and that what comes back is correct. How often have we, your teachers, not wept when we 'quickly' skipped this and found out that the output we got was incorrect.
There are some general measures you can take to reduce risks:

1. Always start with a clean environment. At exit, do not save the environment, but start afresh. This ensures that you *know* which packages you loaded, which files you need to collect, which variables you need to define.
2. When writing a pipe (%>%) or any complex set of operations, check each step individually. Was the outcome of your function as intended?
3. Do a pilot. If you run a calculation that takes a long time, run it on a small subset of your data and *thoroughly* check the output.
4. Know what you do, especially with statistics. Always *visualize* the outcome of the analysis if you can. Otherwise it might be that you take a trip into statistical winter-wonder land.
5. Be wary of the class factors. Factors are indexed sets of data. If this indexed set of data is a number, the number on which the calculation is performed is the *index number*, not the actual number you see. In this case, you need to transform the factor to an actual number. You can check if this might be a problem using the function `class()`.

3.2.1 Inspecting data

3.2.1.1 Using functions

Inspect the data using the same functions as in Chapter 2. Also, these functions you can find in Appdix Base R cheatsheet.

```
head(pheno_data)
tail(pheno_data)
summary(pheno_data)
```

```
length(pheno_data)
ncol(pheno_data)
nrow(pheno_data)
dim(pheno_data)
```

Exercise 3.3 (Inspecting data)**.** Answer these three questions:

1. What does the `head()` function tell you about an object?
2. Similarly, what does the `tail()` function tell you about an object?
3. If you would not know this, what is the command you could execute within the R console to find out?

3.2.2 Cleaning and preparing the data

Not needed here today. We keep this heading in for completeness though.

3.2.3 Analyzing

Now the fun starts. We pick up on our old friend `cor()`. To determine which kind of correlation we need to perform, we need to figure out the statistical distribution of the data.

3.2.3.1 Normal distribution

Same as before, let's figure out whether the data is normally distributed. If you don't know the two tests anymore, check back at Chapter 2.

```
###Use the graphical method

###Use the statistical method
```

Exercise 3.4 (Does the data follow a normal distribution?)**.** You now conducted two types of analyses that tell you whether the data is normally distributed. What is your assessment, is it normally distributed?

3.2.3.2 Correlation analysis

Yesterday we conducted an analysis of correlation using the `cor()` function. There is actually another way to conduct this type of analysis, namely using a linear model with `lm()`. The advantage of this method is that we will get a slope and an intercept.

We can use the function `summary()` to get the fit of the model.

```
data.test <- pheno_data

### here the linear model is done
tested <- lm(value~dpi,data=data.test)

### use summary to get the fit
summary(tested)
```

The `summary()` function is a very versatile function, that can generate outputs from many objects. In this case, it presents the analysis from the linear model, which in essence fits a correlation line. The `Estimate` for the intercept of that line (-2.97528) and the slope (0.50663) are reported. Also the fit of the model (the Adjusted R-squared: 0.8274) and of the parameters. For instance, there is a significant correlation between day post infection (dpi) and the leaf area (value); p < 2e-16.

3.2.4 Presenting

3.2.4.1 Scatter plot

Now, let's make a proper plot of correlation. Of course, you can find instructions in both Chapter 1 and Chapter 2. Also, check Appendix GGplot cheatsheet.

```
data.plot <- pheno_data

###I'm getting the statistics here
statplot <- broom::tidy(summary(lm(value~dpi,data=data.plot)))

p1 <- ggplot(data.plot, aes(x = dpi, y = value)) +
      geom_point() + geom_smooth(method = "lm") +
      theme_bw() + annotate("text",
                            x=10,
                            y=max(data.plot$value),
                            label=paste("p =", signif(statplot$p.value[2], 2)))

p1
```

Exercise 3.5 (Is the plant-growth correlated with time?)**.** What is the *biological interpretation* of the correlation you calculated and observed?

1. What does the intercept represent?
2. What does the slope represent?
3. Is a linear model a good fit for plant growth *in general*?

3.3 Assignment for code-review week 1

3.3.1 The assignment

Via the fileshare on Brightspace, you can find your dataset for the code review. Everyone of you has their own folder, with their own dataset. This means that the conclusions you draw will be based on your individual dataset. We expect that you hand in a .html file generated using a Quarto Markdown (QMD) file you wrote yourself in R. Within the .html file you provide the code to:

1. Setup of the R environment
2. Load the data
3. Inspect the data
4. 'Clean' the data
5. Analyse the data
6. Present the data

Also, you should provide a conclusion about the *biological interpretation* of the data. The exercises below take you through the assignment step-by step. Make sure that for the R steps you include code chunks. If you print figures or execute functions within these blocks, the output will be included in the .html.

Do not wait untill you are finished to make the first render. At the very start, when your file is still pristine (e.g without code chunks but after adapting a .qmd file from Chapter 1 or Chapter 2), do a check on whether it renders correctly. You render the document by hitting the 'Render' button on top of the top-left pane in RStudio.
Note that in day 1 and day 2 the parameter `eval` is set to `false` in the yaml header (this is the text at the start of the document between the --- and ---). This means that in the .html generated, none of the code blocks are executed. This is a good option to prevent R coding errors from rendering the document. However, if you want to include a figure, you need to set this option to true within that code-chunk `#| eval: true`.
If you need more tips in general on QMD, check Appendix QMD notebook cheat-sheet.

Exercise 3.6 (Setup of the R environment)**.** Set the working directory and install and activate the required packages to run the code within the file.

While going through the exercises, you might encounter steps requiring you to install additional packages. Simply add them to this section of the code one-by-one. You help yourself by being explicit about which package you use (e.g. `dplyr::` or `tidyr::`).

Exercise 3.7 (Load the data)**.** Load the two data files into the R environment using the appropriate read function. Make sure that your name is included here, so the reviewers can check if your code works. That way your reviewer can find your datafiles.

The data consists of the growth rates of *A. thaliana* Col-0 when inoculated with either of two plant parasitic nematodes[13]. The two nematode species tested are the beet-cyst nematode *Heterodera schachtii* and the root-knot nematode *Meloidogyne incognita*. Within your personal folder you find the data of the experiment conducted at a particular infection density (P_i) in nematodes g^-1. The measured trait is the day-by-day growth rate, measured between two consecutive days post infection (dpi). For instance between day 2 and day 3. The dpi within the data file would be 2.5 in that case. This data is given per plant.

Exercise 3.8 (Inspect)**.** Inspect the data using functions (no plot required). Based on the output indicate which bind function you need to use to combine the two objects into one.

We covered the data inspection functions in Chapter 1 and Chapter 2) under the heading Inspecting data.

Exercise 3.9 (Clean; combine)**.** Use a 'bind' function to combine the two objects into one.

> ℹ We covered the bind functions in Chapter 2) when combining the four objects.

Exercise 3.10 (Analyse)**.** Compare growth rates of the *A. thaliana* plant when infected with either *H. schachtii* or *M. incognita*, analyse whether these differ using a test that does not assume a normal distribution. Report at least the p-value.

We do not ask you to check for normality for this assignment, as that dataset overall is not normally distributed, we give you that your subset is also not. Note that in the paper We deal with this by using a non-parametric test as well[13].

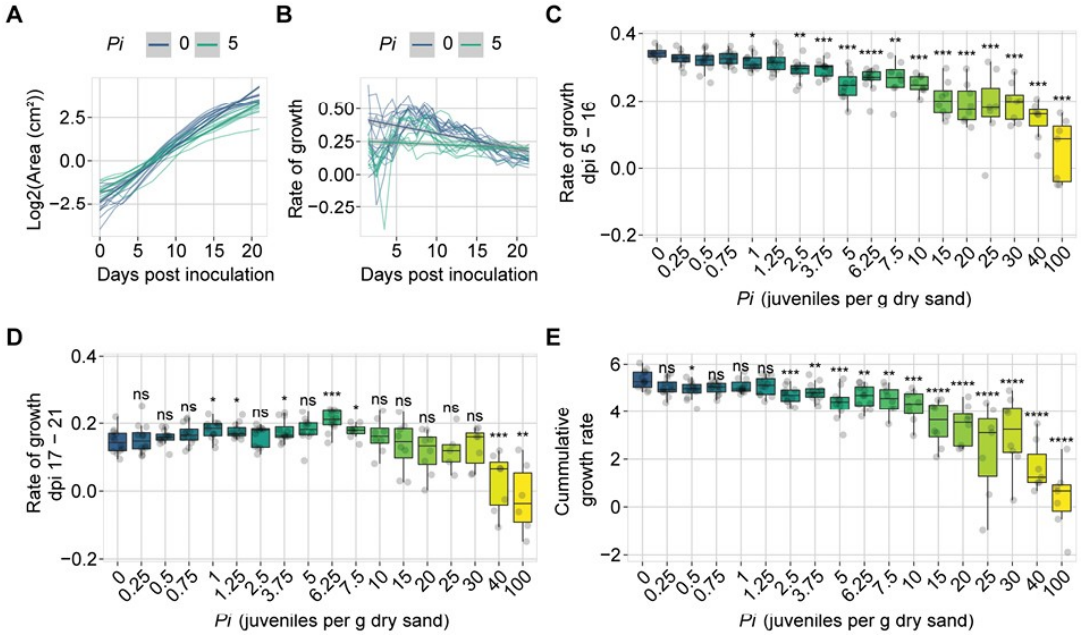


Figure 3.2: Quantification of the tolerance limit of Col-0 to Meloidogyne incognita based on growth rate. Nine-day-old Arabidopsis seedlings were inoculated with 18 different densities (Pi) of M. incognita juveniles (0–100 juveniles g^{-1} of dry sand. (A) Green canopy area of plants treated with P_i=0 (blue line) or P_i=5 (aquamarine line) from 0 to 21 dpi. (B) Rate of growth of plants treated with P_i=0 (blue line) or P_i=5 (aquamarine line) from 0 to 21 dpi was calculated using Equation 2 in the manuscript. (C) Rate of growth of plants inoculated with increasing P_i values growing from 5 to 16 dpi. (D) Rate of growth of plants inoculated with increasing P_i values growing from 17 to 21 dpi. (E) Cumulative growth rate of plants inoculated with increasing Pi values. (C–E) Dots represent individual plants. Data were analysed with a Wilcoxon rank sum test. ns=not significant, * P<0.05, ** P<0.01, *** P<0.001 (n=10–12).

Exercise 3.11 (Present)**.** Make a `ggplot2::` boxplot showing the two nematode species on the x-axis and the growth rate on the y-axis.

> ℹ We covered the boxplot function in Chapter 1) in the presentation section. Also, you can find how to in Appendix GGplot cheatsheet.

Now you have analysed the difference in growth rates between the two species at a particular density at a particular day of the experiment. Now it is time to draw conclusions.

Exercise 3.12 (Interpret)**.** What is the *biological interpretation* of the boxplot and the wilcoxon-test p-value?

Once you have worked through all these excercises related to the assignment, you can render the .html file. Check the file (is everything included?). If you run into errors, the 'Background Jobs' window will indicate where the rendering has halted.

3.3.2 How to do code-review

During code-review, we try to establish if the code that is written:

1. is well-organized
2. is well-documented
3. properly used
4. works

Next to that, we also check whether the conclusions drawn from the analysis and figure are valid.

3.3.2.1 How to give peer-feedback

You need to review 2 coding assignments of other students within the course. Your teachers will check your feedback. In the first week, you will get an indication from us whether you give sufficient and clear feedback. You need to get a 'pass' for the assignments in week 2-4.

3.3.2.1.1 What is feedback?

Feedback literally means 'giving back'. It is not an euphemism for criticism. Rather, feedback can be described as: information regarding the way my message or behaviour (or coding and interpreting the outcome of data-analysis in this case) has been received and interpreted. Feedback stimulates you to think about and implement possibilities to improve.

Feedback is critical for your own improvement. Peer feedback relies on giving and receiving feedback. Receiving feedback allows you to develop your competences and skills. Well-given feedback can help you asses your strengths and weak points, and allows you to address the weak points from the basis of your strengths. However, giving feedback also ask you to be critical and recognize certain aspects of data analysis, which you can implement again in your own process of analyzing data in R.

3.3.2.1.2 How to give feedback

When giving feedback it is important to:

1. Balance your feedback. Mention both positive and constructive aspects (Tops and Tips). It is equally important to know your strengths, as it is to know your weaknesses. Moreover, it is motivating to receive positive feedback.
2. Be specific. Feedback is not helpful to the receiver when the feedback is not specifically formulated. Comments such as: "I think it is bad code and it needs revision" does not tell the receiver why it is a bad code. Instead, "I think you arrive at your result in more steps then necessary, I suggest you use function x instead of function y and function z" is much more helpful.
3. Connect concrete suggestions for improvement when providing constructive feedback. This can be quite challenging, and requires you to place yourself in the shoes of the someone else. This is very helpful for the receiver, as they receive help on how to improve.
4. Provide examples. Link your feedback to specific pieces of code or sentences in the document. In Feedbackfruits you can highlight text and make comments.
5. Formulate your feedback to what you observe (objectively) in an open way (without judgement or premature conclusions). For instance: "I noticed that you used the same object name for two different datasets, did you do this for a certain reason?" You can also describe the effect it has on you: "For me, it would be more logical if you use readxl::read_xlsx() to load an .xlsx file instead of read.delim()."
6. Do not generalize. Avoid words like always/never in your feedback e.g. "You always forget to indicate which package a function comes from..."

3.3.2.2 Well organized code

There are many ways in which you can write well-organized code. What all have in common is consistency. Here, I teach you *a way* of doing this. These are the rules that I *generally* follow. I would like to stress that these are guidelines. These guidelines help others to read and understand your code. However, they also help you when you need to re-use or re-run your code.

Check:

- Are clear section-level headings used?
- are hierarchies indicated with a tab?
- from non-base R functions, is the package indicated (e.g. `dplyr::`)?
- do objects have a clear, understandable, name?

3.3.2.3 Well documented code

When writing code, leave comments. Especially when you're doing something new / special this is very helpful to you in the future. But also to others that will check or even use your code. It can be very handy to indicate why you use a particular function for instance.

Check:

- Are comments included in the code?
- Above code chunks, is there an explanation of what is done and why?

3.3.2.4 Properly used code

This checks whether the functions are used as intended. For instance, you can run a correlation analysis on numbers that are of `class` `factor` (check this by using `class()` on a vector/data frame column). However, correlations should be done using actualy numbers `is.numeric()` should return `TRUE`. Another example is whether the correct statistical test was used given the data?

Check:

- Are the correct functions / inputs used?

3.3.2.5 Working code

Check:

- can you run the code and get the same output?

3.3.2.6 The Biological interpretation

Do you agree with the biological interpretation given? Note that all of you have different days and P_i's to compare. Thus, the conclusions will probably differ amongst yourselves!

3.3.3 Some reading

Yesterday you might have been convinced that a hypothesis is a liability[14], today you may convince yourself otherwise. There is a nice critique written of the provocative paper you read yesterday. The authors in the paper 'The data-hypothesis relationship' will convince you that a hypothesis is not such a bad idea after all[4].

Part II

II: Getting familiar with programming

The goals for week 2

This week is about promoting the basic R you met in week 1 into practical programming skills for data analysis. Where week 1 introduced R and the data-science cycle, week 2 focuses on how you express data checks, summaries, and small analyses as reusable, testable code. The goals are:

- Compute and interpret common summary statistics programmatically.
- Write small functions to encapsulate repeated operations and calculations.
- Find and fix bugs using minimal, systematic debugging techniques and simple unit tests.
- Quantify relationships between variables (correlation) and interpret what those relationships do and do not tell you.
- Apply the core tidyverse verbs to reshape, summarise, and visualise data in reproducible pipelines.

To achieve these goals, we will dive into a few topics you encountered already in week 1 in more detail.

How week 2 is organized

The material for week 2 is divided over four chapters, which you will work through in the first three days of the week.

- Day 1: Chapter Chapter 4
- Day 2: Chapter Chapter 5 and Chapter Chapter 6
- Day 3: Chapter Chapter 7

The **data** for week 2 focuses mostly on ecological traits and environmental measurements: we use two published datasets, and one dataset that was gathered during the course Plant Science in Practice (NEM11305). Since the focus of this week is mostly on programming techniques and analysis methods, the published datasets have been pre-processed into a *data package*.

Note 1: Installing the data package for week 2

The following code lets you install the `PlantEDA` R package that contains the data for week 2. There are three datasets: `corre_continuous` and `corre_categorical`, and `grassland_traits_environment`.

```
# We need the remotes package to install from the WUR bioinformatics github
install.packages('remotes')
# Here we install the PlantEDA package from github, this contains the data for this week
remotes::install_github('wur-bioinformatics/planteda')

# If we load the PlantEDA package, we can load the datafiles from the package
library(planteda)
data("corre_continuous")

# Alternatively, you can specify the package directly in the `data` call
data("corre_continuous", package = 'planteda')
```

BONUS EXERCISE: inspect the PlantEDA data package source code

The source code for the `PlantEDA` data package is available at https://github.com/wur-bioinformatics/plantEDA. Inspect the repository structure and see if you can identify where the data is stored, and what steps are taken to publish it.

The assignment

Like last week, a coding peer-feedback assignment is due on Wednesday (deadline 23:59), which you submit via feedback fruits on Brightspace. Again, your assignment will be reviewed by two students, and you will review the assignment of two other students. The instructions for the assignment of this week can be found in chapter Chapter 7. **Completing the assignment and participation in the code-review is mandatory.**

The exam

At the end of week2 we expect you to be able to:

- Everything from week 1
- Produce your own `.qmd` document
- Compute and interpret a basic summary of a structured dataset (i.e. a dataframe)
- Describe the properties of a few summary statistics
- Perform and interpet a correlation analysis using `cor()` and `cor.test()`

- Systematically debug errors and broken code
- Write and test a small function that performs a specific task
- Create a simple `dplyr` and `tidyr` data processing pipeline
- Create a simple `ggplot` data visualization

4 Functions and summary statistics

In this chapter we expand on two concepts that we encountered in the previous week: functions, and summarizing a dataset with numbers. We start with a few concepts on what functions are and when to use them in Section 4.1, and we apply these concepts in Section 4.2 to implement the calculation of some useful dataset summaries. We will explore several numerical summary techniques, such as computing the mean and standard deviation of a numeric variable, or counts/frequencies of a categorical variable. We finish the chapter with a brief section on various other programming concepts that you typically encounter when doing exploratory data analysis (Section 4.3).

Exercise 4.1 (Inspecting the datasets for this week)**.** Make sure you have installed the `PlantEDA` data package (see Note 1). Inspect the documentation for the three datasets: `corre_continuous`, `corre_categorical`, and `grassland_traits_environment`. Where do these datasets come from? What is the difference between the three datasets? In what type of object are these datasets stored?

```
library(planteda)

# Loading and inspecting the corre_continuous dataset
data('corre_continuous')
?corre_continuous
```

4.1 Functions

In the previous chapters we have encountered *functions*. So far, we have used these functions to our advantage, but we have mostly glossed over what a function is, and when it is most appropriate to use functions. In this section we go into a bit more depth to highlight the most important practical aspects of functions in R, whilst avoiding unnecessary theoretical detail as much as possible.

4.1.1 What is a function?

A function can be concisely described as a **reusable piece of code** that takes input values (arguments), performs an operation, and returns a result. Some clear examples of this are the functions you have used from some of the packages you used (e.g. the `read.csv()` or `ggplot()` functions). It is important to make the distinction between the *function definition*, which is the piece of code that describes what the function does, and *calling a function*, which is where the code is used (See Note 2).

Note 2: R function syntax

4.1.1.1 Function definition

Named functions in R are all functions that are assigned a variable name (see example below).

```
# Named function example
function_name <- function(argument){                    ①
  # function body                                       ②
  return(...)                                           ③
}                                                       ④
```

① First line of a function definition, in this case the function is assigned to a variable named `function_name`. `{` indicates the opening of the function body.
② The function 'body' contains all the logic that will be executed upon function execution
③ The return statement provides the result of the computation performed inside the function

Exercise 4.2 (Inspect some function definitions and documentation)**.** To get familiar with inspecting function documentation and definition, execute the following commands and note down what you find:

- `?read.csv` and `read.csv`
- `?mean` and `mean`
- `?filter` and `filter`

4.1.2 Why (and when) use a function?

As you become more experienced and proficient in performing data analysis in R, you'll find that the amount of code you write to do these analyses expands. The basic steps of a data analysis pipeline (loading, inspecting, cleaning, analyzing, presenting, see Tip 1) will often be repeated multiple times. These are very good cases of where defining a function makes sense! By bundling a sequence of operations into a function, your code improves in three key ways:

1. **Clarity**: instead of a long list of R commands, readers can read more 'high-level' code and more easily understand the main steps being performed
2. **Re-usability**: write the analysis logic once, and reuse it across different datasets without copying and pasting. (Packages are the ultimate example here).
3. **Maintainability**: If something changes in your workflow, updating the logic in one place (i.e. at the function) makes sure all places where the logic is applied are also updated.

A few typical situations where it thus makes sense to implement a function: you're copying and pasting the same code over and over again; a sequence of steps represent a logical subtask; applying the same steps to different variables or datasets. In addition, functions make your code easier to document and test (more details on testing will be discussed in Chapter 5).

Tip 2: Naming functions (is hard)

Should my code be a function? A straightforward way to determine whether a piece of code can be extracted into a function, is how easy it is to give a name to the collection of steps being performed. You'll be surprised how hard it can be to give intuitive names to functions! In fact, there are entire books written about just this.
Base R has some good examples (both good and bad):

- `read.csv()` does what it says: it reads a csv file. The name matches perfectly with what the function does.
- `apply()` is much less straightforward: the name does not tell you what is being applied, or what it is being applied to. You need to read the documentation carefully to figure out these things, and how to actually use this function.

Exercise 4.3 (Revisite some code from last week)**.** Let's take a look at some code from last week:

1. Compare the code for Exercise 1.11 and Exercise 1.12. Which parts are shared? What would the function for doing the shared parts be called?
2. Revisit the code for Exercise 1.12. There are a few steps in the for loop that perform (more or less) isolated tasks. Could you come up with logical names for these steps? Do you think it makes sense to extract these steps into individual functions?

4.2 Summary statistics

To explore new and previously unseen datasets, we tend to follow the data science cycle (Tip 1). Summary statistics play a key role in both the **inspect** and **analyse** parts of this cycle. A summary statistic is something that is easy to calculate, and gives you some insights into what the data in your dataset looks like. By calculating a few different summary statistics, you can get a quick overview of a dataset, and potentially diagnose some easy to spot issues.

In this section, we present the main summary statistics that you will use for summarizing a new dataset. For most summary statistics, we perform some arithmetic and computation. This gives us an excellent opportunity to implement what was covered in the previous section on functions!

4.2.1 Mean

Perhaps one of the simplest and most intuitive summary statistics is the mean (typically indicated with the greek letter μ). The mean can be calculated as the sum of all values in a variable divided by the number of values in that variable (Equation 4.1).

$$Mean(X) = \mu = \frac{1}{N}\sum_{i=1}^{n} x_i \tag{4.1}$$

The simplest and most direct interpretation of the mean is that it represents the average value of a variable. A slightly more statistically oriented interpretation is that the mean represents the expected value. In other words: if you have no other information, the mean would be your best guess[1] on average. In the example below this would mean that your best guess of the leaf dry matter content of a plant, without knowing anything else about that plant, would be $0.25 g/g$ (The blue solid line in Figure 4.1).

Being a statistical programming language, R comes with a default function to calculate the mean:

```
# Calculate the average leaf dry matter content
mean(corre_continuous$LDMC)
```

```
[1] 0.2535559
```

[1] "Best guess" is kept informal here. A full probabilistic treatment exists, but is beyond the scope of this book

Note 3: Example: implementing the mean calculation in a custom function

Since R already comes with a default function to calculate the mean this example might seem redundant. At the same time, this gives us the opportunity to check whether our custom implementation returns the correct result.

```
# Here we define a function to perform the calculation of the mean,
# closely following the mathematical description
custom_mean <- function(input){
  total = sum(input)
  number = length(input)
  return(total / number)
}

# This should give the same result as using the default `mean` from R
custom_mean(corre_continuous$LDMC)
```

```
[1] 0.2535559
```

4.2.2 Variance

Where the mean describes the average value of a variable, the variance describes how much individual data points differ from the mean. More precisely, the variance is defined as the average of the squared differences between each observation and the mean (Equation 4.2). In other words: for each value we compute how far it is from the mean, square this difference, and then average these squared deviations. In our example on leaf dry matter content, the variance quantifies how much individual plant species differ from the average leaf dry matter content. Note that squaring the difference has two important consequences: positive and negative differences contribute equally, and large differences contribute relatively more than small differences. A consequence of this last property is that the variance is relatively sensitive to outliers in the data.

$$Variance(X) = \sigma^2 = \frac{1}{N}\sum_{i=1}^{n}(\mu - x_i)^2 \tag{4.2}$$

4.2.3 Standard deviation

One slight downside in interpreting the variance is that the *units* are squared compared to the data points. In our example, leaf dry matter content is measured in $[g/g]$, and

as a consequence the mean is also in $[g/g]$. However, the variance is in $[(g/g)^2]$, which makes interpretation less straightforward. A common and useful solution is to take the square root of the variance, which results in the standard deviation (See Equation 4.3 and the striped blue lines in Figure 4.1).

$$StandardDeviation(X) = \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{n}(\mu - x_i)^2} \tag{4.3}$$

Exercise 4.4 (Implement the variance calculation in a custom function)**.** Just like in the previous example on the mean, R has a built-in function to calculate the variance (`var`). To practice implementing custom functions, in this exercise you will implement the calculation of the variance yourself. Compare the output of your custom function to the output of the built-in `var` function to check your work.

```
# Implement the calculation of the variance in a custom function
custom_variance <- function(input){
  # ... some initial calculation goes here
  variance <- # ... final calculation goes here
  return(variance)
}
```

4.2.4 Median

The mean and variance summarize the data by combining all values using arithmetic operations. In contrast, some summary statistics depend only on the *relative order* of the values, not on how large or small individual observations are. These order-based statistics often behave more robustly when extreme values are present (in other words: they are less sensitive to outliers). In the following section we discuss the median (the striped blue line in Figure 4.1), its generalization into quantiles, and the accompanying interquartile range (the dotted blue lines in Figure 4.1).

The simplest and most direct interpretation of the median is that it represents the middle value of a variable. More precisely, the median is the value that splits the data into two equally sized parts when the observations are sorted. In other words: half of the observations are smaller than the median, and half are larger. In the example below this would mean that half of the plants have a leaf dry matter content below the median value of 0.24g/g, and half above.

```
# Calculate the median leaf dry matter content
median(corre_continuous$LDMC)
```

```
[1] 0.2397274
```

Exercise 4.5 (Interpret the mean and median)**.** When computing the mean and median of the leaf dry matter content, we observe a (small) difference. How do you interpret this difference? (You can use Figure 4.1 for visual intuition).

Exercise 4.6 (Summarize the corre_continous dataset)**.** Write some R code to calculate the difference between the mean and the median of all continuous variables in the `corre_continuous` dataset.

Tips:

- The function `colnames()` returns the names of all columns in a dataframe
- You can loop over colnames
- You can access an individual column of a dataframe by name and using double square brackets –> `dataframe[[colname]]`
- You can check whether something is numeric with the `is.numeric()` function

4.2.5 Quantiles

Quantiles generalize the idea of the median by describing values at fixed positions in the sorted data. A q-quantile is the value below which a fraction q of the observations fall. In other words: a q-quantile splits the data into a lower part containing a fraction q of the observations and an upper part containing the remaining fraction. For example, the 0.25-quantile (also called the first *quartile*) is the value below which 25% of the data lie.

Exercise 4.7 (Implement quantile calculation in a custom function)**.** In this assignment you will implement the calculation of quantiles for a continuous variable. Once again, you can check your implementation by comparing to the built-in `quantile` function. Unlike the mean, calculating quantiles is not just a sequence of arithmetic. To help you get started, here are some pointers:

- The number of elements in a collection can be calculated with `length`
- A collection can be sorted with `sort`
- Numbers can be rounded with `round`, the resulting integer can be used as an index

```
custom_quantile <- function(data, quantile_percentage){
  # Do stuff here
  quant <- # Some more calculations
  return(quant)
}
```

4.2.6 Interquartile Range (IQR)

A commonly used measure of spread in the data based on quantiles is the interquartile range (IQR). The IQR is defined as the difference between the third quartile and the first quartile. In other words: it measures how much the middle 50% of the data vary. In the example below this would quantify the spread of leaf dry matter content values for the central half of the plants (also see the dotted green lines in Figure 4.1).

```
# Calculate the middle 50% of values (AKA the interquartile range IQR) for leaf dry matter co
IQR(corre_continuous$LDMC)
```

```
[1] 0.1105531
```

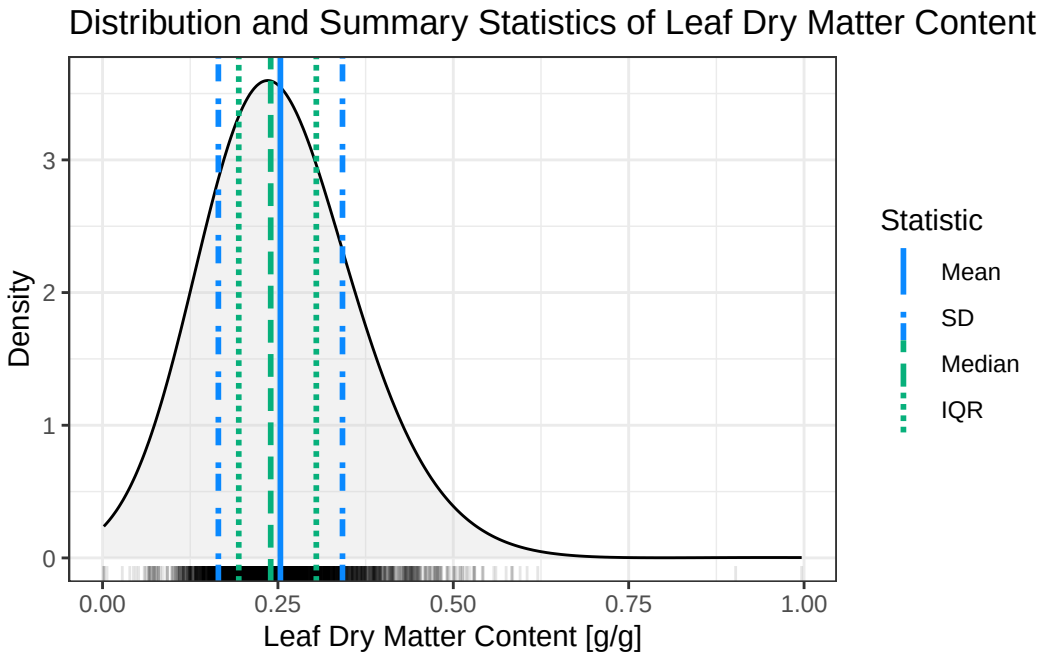


Figure 4.1: A visualization of various summary statistics computed on the Leaf Dry Matter Content variable in the CoRRE dataset. Blue lines represent the mean and standard deviation, green lines represent the median and the interquartile range. Individual datapoints are indicated on the X axis. The effect of a few large values can be observed in the difference between the mean and the median.

Exercise 4.8 (Creating and interpreting a boxplot)**.** Create a boxplot for leaf dry matter content using the `corre_continuous` dataset and compare to Figure 4.1. Which sum-

mary statistics do you recognize? (You can use either base R `boxplot` or `ggplot` with the `geom_boxplot` geom.)

4.2.7 Counts/proportions

All the summary statistics we have discussed so far apply to numeric variables. Of course, not all data is numeric: one commonly encountered type of data is categorical. The most commonly used technique to summarize categorical data is to count the number of elements in each category, and potentially convert these counts into proportions. In the example below we use the built-in `table` and `proportions` functions to calculate counts and proportions respectively for how many species are known to be clonal.

```
# Counting how many plant species are clonal
table(corre_categorical$clonal)
```

```
       no uncertain       yes
     1646       442      1991
```

```
# Calculating the proportions of how many plant species are clonal
proportions(table(corre_categorical$clonal))
```

```
       no uncertain       yes
0.4035303 0.1083599 0.4881098
```

4.2.8 Putting it all together

Throughout this chapter, we introduced summary statistics by first focusing on their individual roles and functions—such as `mean()` for central tendency, `sd()` and `var()` for spread, `median()` and `quantile()` for robustness against outliers, and counting functions for categorical data. While these functions are useful when you want a specific numerical result, R also provides a convenient way to combine several of these descriptive measures in a single step using the function `summary()`. For numeric data, `summary()` computes the minimum, first quartile, median, mean, third quartile, and maximum, bringing together measures of location and spread that you have already seen separately in this chapter. For categorical variables, the same function *switches behaviour* and returns counts per category, which mirrors the tabulation-based summaries discussed earlier. As such, `summary()` acts as a compact overview function: it does not replace the individual summary statistics, but it provides a quick first impression of the data using a consistent and easy-to-interpret output.

Exercise 4.9 (Putting all summary statistics into a single overview)**.** Use the `summary()` function on both the `corre_categorical` and `corre_continuous` datasets. Describe your findings, making note of the following things: (1) what are the data types of the different variables? (2) How many NAs are there in the dataset? (3) Are there any variables with extreme values?

A note on hypothesis testing

In this chapter, we describe summary statistics mostly in the context of the *inspect* step of the data science cycle (Tip 1). However, some of these summary statistics also appear in the *analyse* step! In that context, some of these statistics are used in e.g. statistical hypothesis testing. A clear example is the **t-test** (Equation 4.4), which uses the means and standard deviations to compute a p-value for a observed difference. The exact formulations of the statistical tests belong in a statistics course, and are outside of the scope of this book. However, here we show a formulation of the t-test as an example, and to highlight the appearance of the **mean** and **standard deviation**. Note: here we use $\bar{x}$ and s for the mean and standard deviation respectively. The symbols μ and σ are typically used in formulas and theory but refer to ideal quantities rather than observations of a dataset (which would be part of a follow-up statistics course!).

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{4.4}$$

4.3 Other programming techniques

In Section 4.1 we have discussed the basics of functions in R. In your data analysis practice, this will be the most useful tool that you use to make your code readable, reusable, and maintainable. However, when working with R a bit more (and perhaps also using other programming languages), you likely encounter some other programming techniques as well. In this section we briefly highlight a few common programming techniques to raise awareness, but we do not go into full detail on how these techniques work.

Advanced R

The book Advanced R [12] provides a good and detailed overview of some of the topics in this section. It's not very beginner friendly though!

4.3.1 Object Oriented Programming (OOP)

In addition to functions, R also supports object-oriented programming (OOP). In OOP, data *and the operations that act on that data* are conceptually bundled together into objects.

In practice, you have already been using OOP in R, even if you were not aware of it. For example, when you call `summary()` on different types of objects, R automatically chooses an appropriate method depending on the object:

```
# These all provide useful summaries, but are implemented completely differently under the ho
summary(my_numeric_vector)
summary(my_data_frame)
summary(my_statistical_model)
```

Although the function name is the same, the behavior differs. This is an example of *method dispatch*, where R selects the correct method based on the class of the object.

For most data analysis workflows, it is not necessary to write your own object-oriented code. However, understanding that many R functions behave differently depending on object class helps explain why the same function call can produce different results in different contexts.

4.3.2 Scoping rules

The *scope* of your code determines where R looks for objects such as variables and functions.

When a function is executed, R first looks for variables inside the function itself. If a variable is not found there, R then searches in the surrounding environments. This behavior is known as *lexical scoping*.

```
# Scoping example: var1 is defined in the outside scope
var1 <- 10

# This example function takes no arguments
f <- function() {
  # var2 is defined in the function scope
  var2 <- 5
  # We still have access to the outside scope
  return(var1 + var2)
}
```

```
# Executing the function performs the calculation. Note there are no arguments provided!
f()
```

```
[1] 15
```

Exercise 4.10 (Thinking about scope)**.** In the example above `var1` and `var2` are available in the function scope (Verify how you know this is true!). Is `var2` available outside of the scope of the function (e.g. what happens if you try to print `var2` outside of the function?)

4.3.3 Vectorization

Many quantitative operations and functions in R are *vectorized*. This means that the semantics of the operation are applied to a collection of elements individually, without having to write additional loops or other code

```
# Vectorization example, the square root is calculated for each number individually
numbers <- c(25, 9, 5)
sqrt(numbers)
```

```
[1] 5.000000 3.000000 2.236068
```

There is no single diagnostic for determining whether a function is vectorized or not. Often the semantics of the function will give a hint: if the operation should work on multiple elements, and the number of return values is the same as the number of inputs, a function is a good candidate for being vectorized. Some examples: most arithmetic is vectorized, `read.csv()` is typically not expected to work on a collection of filenames (so not vectorized), and `mean()` summarizes multiple numbers into a single statistic (also not vectorized).

Exercise 4.11 (Vectorization alternatives)**.** What other language constructs have you already encountered that perform the same function as vectorization? (Hint: look back at Exercise 1.11)

5 Debugging and testing

Programming and scripting for data analysis is *not only* about writing code that runs. It is perhaps even more important to write code that produces correct and interpretable results.

Errors in code can become especially dangerous when they silently produce wrong results without throwing errors. In this chapter we will go over different types of errors and warnings and how you can solve and prevent them.

> ⚠ **Warning 1: Data analysis code gone wrong, famous examples**
>
> In the late 2000s, researchers claimed that gene-expression data from cancer cell lines could be used to predict which chemotherapy drugs would work for individual patients. These results were considered so promising that human clinical trials were started based on the predictions. Later, independent scientists tried to reproduce the analysis and found that the problem was not the biological question or the statistical methodology, but simple bugs in the data-analysis code [2]. Gene-expression tables had been misaligned with patient labels, some samples had been accidentally swapped, and categorical variables had been silently reordered by the software. Because the analysis pipeline contained no checks that data and labels still matched, these errors went unnoticed. The models therefore looked highly accurate, while in reality they were making predictions close to random guessing. When the mistakes were corrected, the supposed drug-response signatures disappeared. Several papers were withdrawn and the clinical trials were stopped. This episode is now a classic warning that small programming errors can lead to large scientific and real-world consequences. The take home message: careful debugging and reproducible workflows matter in data-driven biology.

We will work on two complementary skill sets:

- Debugging and error handling: what to do when your code (or someone else's code!) does not work as intended.
- Testing and validating: how to formally verify your code works as intended

5.1 Debugging and error handling

Bugs and errors go hand in hand: a **bug** is an error or flaw in a program that causes it to behave differently from what the programmer intended. Bugs can cause code to crash with an error, produce warnings, or even run without complaint while giving incorrect results.

Bugs arise for many reasons: misunderstandings about how a function works, incorrect assumptions about the data, edge cases that were not considered, or simple typing mistakes. Importantly, a program can be syntactically correct (it runs) and still contain bugs if the logic is wrong.

Debugging is the process of identifying, understanding, and fixing these bugs. Rather than trial-and-error, effective debugging relies on systematically inspecting code, checking assumptions, and narrowing down where the program's behavior diverges from expectations.

> **About the term "bug"**
>
> The word "bug" to describe a programming error is often linked to Grace Hopper. In 1947, while working on the Harvard Mark II computer, her team found that a malfunction was caused by a real moth trapped in a relay (back then computers where large devices that filled entire rooms). The insect was taped into the logbook with the note "First actual case of bug being found".
> Although the term bug was already used informally in engineering, this incident popularized it in computing. It nicely illustrates an important idea: bugs can arise from unexpected causes, and finding them requires careful investigation rather than guesswork.

Tip 3: How to debug

The main approach to debugging (potentially complex) code is to be systematic. The following steps help in identifying the underlying problem:

1. **Reproduce the issue**: make sure you consistently see the bug. Does it only occur on a specific dataset? Or under other specific conditions? These can provide hints.
2. **Isolate the problem**: (especially when working with larger functions or big datasets) Narrow down where the error *actually* comes from. Adding print statements or commenting out specific sections can help. Switching to a small test dataset can be useful too.
3. **Understand/hypothesize**: once you figured out where the error occurs, check what you expected to happen and verify this is (or is not) happening.
4. **Fix and verify**: Implement the fix, then *test* the code by reproducing the original conditions under which the error occurred.

To better understand what can cause your code to misbehave, it is important to understand a little bit about different types of errors in R. The following section covers syntax errors, runtime errors, warnings, and logical errors.

5.1.1 Syntax errors

These errors often pop up when you are in the process of writing some new code. Simply put, a syntax error is any form of mistake where your code does not follow the syntax definitions of the R language. As a consequence, the code cannot be parsed and executed by the interpreter. These error are caught relatively easy, because any syntax error will break the execution of your code. In addition, when viewing the full Quarto error message (see Note 4), a syntax error always starts with `Error in parse(text = input): [...]` indicating that *parsing* of the code failed.

Note 4: Rstudio VS Quarto syntax errors

The syntax errors you see in RStudio will look slightly different from what you see in this book (see Figure 5.1). This has to do with a difference between how RStudio and Quarto (used to create this book) read R code: RStudio reads line by line, Quarto reads an entire chunk. A scenario where this becomes relevant is when you try to render a notebook that contains a syntax error: then you will see the *Quarto* error, and not the *RStudio* error.

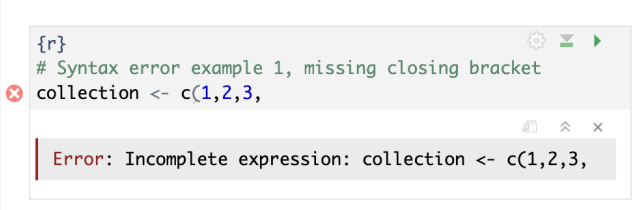


Figure 5.1: Example RStudio error, compare with syntax error example 1 below: the exact error messages are different

Replicating the Quarto error message in RStudio is possible, but requires some trickery: we force RStudio to read an entire chunk at once.

```
# Executing this code in RStudio mimics how Quarto parses code and should produce the sa
code <- "
# Syntax error example 1, missing closing bracket
collection <- c(1,2,3,
"
parse(text = code)
```

```
# Syntax error example 1, missing closing bracket
collection <- c(1,2,3,
```

```
Error in parse(text = input): <text>:3:0: unexpected end of input
1: # Syntax error example 1, missing closing bracket
2: collection <- c(1,2,3,
  ^
```

Note that in R, code is interpreted and potentially executed on a line-by-line basis. Executing syntax error example 2 (below) in a clean environment will crash on the second line, but the first line will already have executed when the crash happens. As a result, in this example the variable `collection` *will* exist when the crash happens.

```
# Syntax error example 2, the last line is missing the closing bracket
collection <- c(1,2,3)
collection[1
```

```
Error in parse(text = input): <text>:4:0: unexpected end of input
2: collection <- c(1,2,3)
3: collection[1
  ^
```

An exception to the line-by-line interpretation and execution is how code blocks are treated. Examples 3 and 4 show examples of syntax errors in code blocks: function definitions and loops are first completely interpreted before they are executed.

```
# Syntax error example 3, using incorrect bracket types
my_function <- function(arg){
  return(arg + 1)
]
```

```
Error in parse(text = input): <text>:4:1: unexpected ']'
3:   return(arg + 1)
4: ]
   ^
```

```
# Syntax error example 4, a syntax error in a loop
for (i in 1:3){
  print(i)
  print('There is a syntax error on this line)
}
```

```
Error in parse(text = input): <text>:4:9: unexpected INCOMPLETE_STRING
4:   print('There is a syntax error on this line)
5: }
           ^
```

5.1.2 Runtime errors

A class of errors that will also break execution of your code are the runtime errors. Unlike syntax errors, the code can be parsed and interpreted, but there is an underlying problem when trying to execute the code that causes the interpreter to crash.

```
# Runtime error example 1: mismatched types
1 + "2"
```

```
Error in 1 + "2": non-numeric argument to binary operator
```

```
# Runtime error example 2: trailing commas/empty arguments are valid syntax but can break at
collection <- c(1,2,3,)
```

```
Error in c(1, 2, 3, ): argument 4 is empty
```

Runtime errors are defined in code and make your code crash when some checks are failing. It is relatively straightforward to implement a check and produce an error (see runtime example 3 below). We will use this more strategically in Section 5.2.

```
# Runtime error example 3: implementing a custom check and producing a runtime error
my_function <- function(arg){
  if (arg > 3) {
    stop("arg must be <= 3, got ", arg)
  }
  print(arg)
}
my_function(2)
```

```
[1] 2
```

```
my_function(4)
```

```
Error in my_function(4): arg must be <= 3, got 4
```

5.1.3 Warnings

In some cases, functions that you use might *warn* you when you are asking for unexpected behavior. One example we previously encountered are the warning messages of functions being overwritten when loading the `tidyverse` package. Warnings do not terminate your code, so just like logical errors you still get results. Unlike logical errors, you get *some* indication that there might be something not going as intended. Sometimes warnings can be safely ignored, sometimes they mean something is critically wrong.

```
# Warning example 1: converting strings to number raises a warning, but you still get a resul
as.numeric(corre_continuous$species[1:5])
```

```
Warning: NAs introduced by coercion
```

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

Exercise 5.1. In the example above (when attempting to convert a character to a number), do you think it is appropriate to raise a warning, or would an error be a better choice?

Implementing warnings in your own functions can be a very powerful tool to keep track of what is happening in your analysis (see example below).

```
# Warning example 2: raising a warning in a function

f <- function(arg){
  if (arg < 3) warning('arg must be >= 3, got ', arg)
  arg ** 2
}

f(2)
```

```
Warning in f(2): arg must be >= 3, got 2
```

```
[1] 4
```

It is important to realize that warnings, just like runtime errors (Section 5.1.2), are implemented in R code by e.g. package maintainers. This means that whether or not a function raises a warning when something unexpected happens always depends on the function, and whether the function author implemented a check or not.

5.1.4 Logical errors

In sections Section 5.1.2 and Section 5.1.3 we have covered what happens when e.g. a package maintainer implements a check in their code. But what about errors in your code that do not cause an error or raise a warning? These errors are potentially the most dangerous, because they do not show any direct visual clues that something is wrong. Any behavior of your code that does not result in the desired/expected result can be seen as a **logical error**. This can range from explicit mistakes in an implementation (e.g. messing up a mathematical calculation), to not realizing some hidden behavior of a certain function, to simple typo's in variable names. The example at the start of the chapter (Warning 1) is an example of logical errors: the code ran to completion and

produced results, but due to some mistakes in the implementation the results were incorrect.

The crucial problem here is that there is nothing in the code that checks assumptions or e.g. enforces certain properties of datasets/results. In Exercise 5.3 and Section 5.2 we explore in more detail how you can implement some of these checks yourself.

Exercise 5.2 (Identify the error)**.** For each of the following 10 exercises, identify the type of error, and fix the code so that it works as expected. The aim of this exercise is to develop some intuition, so first try to do this without executing the code!

```
# Exercise 1: Calculating average leaf Nitrogen content
mean(corre_continuous$leaf_N

# Exercise 2: Finding the most common growth form
mean(corre_categorical$growth_form)

# Exercise 3: Is there a relationship between plant family and leaf N content?
cor.test(corre_continuous$family, corre_continuous$leaf_N)

# Excercise 4: What is the average leaf dry mass?
mean(corre_continuous$leaf_dry_mass, na.rm = TRUE)

# Exercise 5: Select all the entries for the grass family
subset(corre_continuous, family = 'Poaceae')

# Exercise 6: Calculate mean leaf are
mean(corre_continuous$SLA)

# Exercise 7: Select the leaf area for the 1,000th plant
corre_continuous$leaf_area[10000]

# Exercise 8: Compute the average seed dry mass for grasses
subset(corre_continuous, family == 'Poaceae')$seed_dry_mass |> mean(na.rm = TRUE)

# Exercise 9: Compute the mean SLA for a specific family
mean_trait_by_family <- function(dataset, trait){
  mean(dataset[[trait]], na.rm = TRUE)
}
mean_trait_by_family(corre_continuous, 'SLA')

# Exercise 10: log-transform seed dry mass
log(corre_continuous$seed_dry_mass)
```

Exercise 5.3 (Upgrading the implementation of the variance calculation)**.** In Exercise 4.4 you implemented the calculation of the variance. Here, you will make that function a bit safer to use. Add the following checks, decide yourself whether it should be a warning or an error:

- The variance of zero or one numbers is undefined
- The variance of a collection of all the same numbers (e.g. `c(4,4,4)`) is zero

Hints:

- You can implement a check with `if` and `else`:

  ```
  if (condition) {
    # Code to run when condition is TRUE
  } else {
    # Code to run when condition is FALSE
  }
  ```

  In this case `condition` must evaluate to `TRUE` or `FALSE`, e.g. `is.numeric(column)` or `length(input) == 3`. You don't always need the `else` part.
- You can throw an error with the `stop()` function.
- You can raise a warning with the `warning()` function.

5.2 Testing and validating

In Section 4.1 we discussed how and when to use functions. It turns out that by following our rule of thumb for when to create a function[1] we create functions that are often straightforward to *test*. In Section 5.2.1, we discuss what we mean with *testing* a function, and we highlight two ways of testing your code. In addition, we briefly discuss the difference between testing and validating in Section 5.2.2.

5.2.1 Testing code

As an example, we revisit Exercise 4.6: we want to compute the difference between the mean and the median many times, so it makes sense to implement this in a function.

[1] Is there a logical name for a sequence of steps? Then it should be a function. See Tip 2

```
# Example function that we are going to test
mean_median_difference <- function(input){
  # We compute the absolute difference so it doesn\'t matter which number is bigger
  abs(mean(input, na.rm = TRUE) - median(input, na.rm = TRUE))
}

mean_median_difference(grassland_traits_environment$ld)
```

```
[1] 0.0558324
```

Our example function does not throw errors or raise warnings, and it produces a result, so far everything looks alright! But apart from just reading and checking the code, we don't really *know* whether our function behaves as intended. This is where it becomes useful to implement some checks of our code, *using code*. In this book we limit ourselves to unit testing: the process of testing small isolated components (i.e. functions). Approaches for testing integrations or entire systems exist as well, but these are outside of the scope of this course.

The key component of testing functions is to identify a set of inputs for which the output is *known*. This often means coming up with small toy examples. In the case of our example function, the computations are relatively straightforward for small amounts of numbers, so we implement a simple testing procedure below:

```
# We will test our function on a small test dataset for which we know the expected outcome o
test_input <- c(1, 2, 2, 3, 7)

# The mean of this collection of numbers is 3, and the median is 2, so the difference should
known_output <- 1

# We run out function on the test input to acquire the test output
test_output <- mean_median_difference(test_input)

# We check if the acquired output matches the expected output
if (test_output == known_output) {
  print('Great success!')
} else {
  stop(paste('Expected', known_output, 'but got', test_output))
}
```

```
[1] "Great success!"
```

Exercise 5.4 (Experiment with the simple testing example)**.** Create a few alternative testing datasets with known outputs, and use them with the simple testing code above. Also try out what happens when the known output does not match the test output.

A careful observer might already have identified that our simple testing procedure that we describe above always does the same steps for every test you would design. This makes it a very good candidate for extracting the logic into a function, which is exactly what the testthat package does! More precisely, the `testthat` package includes a lot of functionality for quickly and reproducibly writing unit tests, with functions that closely resemble the english language. This latter property makes for very readable testing code:

```
# We implement the same test as before, but now with the testthat package, and including more
library(testthat)

test_that("Correct difference between mean and median is calculated", {
  expect_equal(mean_median_difference(c(1, 2, 2, 3, 7)), 1)
  expect_equal(mean_median_difference(c(1, 2, 3)), 0)
})
```

```
Test passed with 2 successes.
```

In addition to testing expected outputs, the `testthat` package makes it very easy to test other behavior of functions as well, for example whether it correctly produces an error when incorrect input data is provided:

```
library(testthat)

test_that("Error is thrown on incorrect input", {
  expect_error(mean_median_difference('String'))
})
```

```
-- Warning: Error is thrown on incorrect input ---------------------------------
argument is not numeric or logical: returning NA
Backtrace:
    x
 1. +-testthat::expect_error(mean_median_difference("String"))
 2. | \-testthat:::quasi_capture(...)
 3. |   +-testthat (local) .capture(...)
 4. |   | \-base::withCallingHandlers(...)
 5. |   \-rlang::eval_bare(quo_get_expr(.quo), quo_get_env(.quo))
 6. \-global mean_median_difference("String")
```

```
 7.   +-base::mean(input, na.rm = TRUE)
 8.   \-base::mean.default(input, na.rm = TRUE)
Test passed with 1 success.
```

Note that whereas we are incorrectly using the `mean_median_difference()` function and so it raises an error, this is actually part of the test, and so the *test itself* is successful.

In summary: **unit testing** describes a set of tests that are automated in code. Tests are designed to throw an error upon unwanted behavior of the code, so if all tests pass this is a good sign of properly working code.

Exercise 5.5. Note that the *error* test in the example above raises a *warning*? Can you find out where this warning is coming from and what is causing it? Hint: recall Tip 3. Why does the test still pass if a warning is raised? What would be the best way to get rid of this warning: adding additional tests, or changing the function?

Exercise 5.6. In Exercise 5.3 you implemented some warnings and errors for your code that calculates the variance. Write tests with the `testthat` package to verify the calculation, errors, and warnings of your function.

5.2.2 Validating code

In the previous section we looked at formally verifying the correctness of a piece of code. In other words, we answered the question "Does the code do what we programmed it to do?". This an important aspect of producing reliable analysis results, but is not the only thing that matters. In addition to formal correctness, you should also always think about whether it *makes sense* what your code is doing. In this course, we call this **validating**: "Is the code solving the right problem?".

For validation, we typically do not use automated procedures. Rather, we rely on external comparisons: this can be checking plausability of results with literature, but also submitting your code to expert- or peer review (This is in part why we do weekly code review sessions!).

Ultimately, you want your code to be both tested and validated (see Table 5.1), but it takes some time and effort before you get there!

Table 5.1: Different combinations of testing and validating your approach lead to different guarantees on your results

	Tested	Untested
Validated	Provably correct, relevant results	Correct idea, buggy execution
Unvalidated	Bug-free implementation of a flawed idea	Don't go here

Exercise 5.7. Revisit Warning 1 and decide whether testing, validating, or both went wrong. How should plant scientists make sure they do better than this example?

6 Correlation

In this chapter we cover a fundamental approach for comparing two variables: correlation. Calculating correlation is often an important part in the data analysis pipeline (Tip 1). Just like summary statistics, correlation can be relevant in both the inspecting and analyzing parts of the pipeline. Since correlation is about *comparing* things, it is critical for the interpretation of a correlation analysis that it is clear what is being compared.

We first introduce the concept of *co*-variance, from that we derive linear and non-linear correlation measures. We then briefly discuss statistical testing of correlation measures, and we finish with testing and interpreting a correlation analysis (and how to use correlation in building an argument).

6.1 Covariance

In Section 4.2.2 we introduced the concept of *variance*. By calculating the variance you can answer the following question: how much do individual observations differ from the mean? In this chapter we are interested in comparing two variables, and so we generalize the notion of variance to *co*-variance. By calculating the covariance you can answer the following question: if one variable variable varies by some amount, how much does my second variable *co*-vary? In other words: if one variable is high, is my other variable high as well? In a concrete example: do plants that have a large specific leaf area also have a high leaf nitrogen content?

The mathematical derivation for the covariance is relatively straightforward and directly builds on the definition of variance (Equation 6.1).

$$Variance(X) = \frac{1}{N}\sum_{i=1}^{n}(\mu - x_i)^2 \tag{6.1}$$

We start by writing the squared difference from the mean as explicit multiplication (Equation 6.2).

$$Variance(X) = \frac{1}{N}\sum_{i=1}^{n}(\mu - x_i)(\mu - x_i) \tag{6.2}$$

Next we swap out one of the difference terms with the difference from the mean of another variable. Note that we use the respective means for the two variables: μ_x and μ_y (Equation 6.3)!

$$Covariance(X, Y) = \frac{1}{N} \sum_{i=1}^{n} (\mu_x - x_i)(\mu_y - y_i) \tag{6.3}$$

The R language has a built-in function for computing the covariance:

```
# Do plant species with a high specific leaf area have a high leaf nitrogen content?
cov(corre_continuous$SLA, corre_continuous$leaf_N, use='complete.obs')
```

```
[1] 11.78539
```

Exercise 6.1 (Comparing variance and covariance)**.** The formulas in Section 6.1 derive how covariance is a generalization of variance. In this assignment you will verify this empirically using R's built-in `var` and `cov` functions. For a variable of your choice, compute the it's variance, and the covariance of that variable *with itself*. Compare the outputs. How do you explain this? Hint: compare Equation 6.3 with Equation 6.2.

Exercise 6.2 (Interpreting the units of covariance)**.** As discussed in Section 4.2.3, the units of variance are the square of the units of measurement. Verify how you can derive this from the mathematical formulation of the variance (Equation 6.1). Next, describe what the units of covariance are. How interpretable do you think the units of covariance are? Hint: identify what the units of the two separate elements of the product term inside the sum are, and compare those to the units of measurement.

6.2 Linear Pearson correlation

In Section 4.2.3 we discussed how the standard deviation solves a problem of the variance: it expresses the units in which we express variation in the same units as the mean and the original observations. *Correlation* solves a similar problem that occurs with the covariance, but in a slightly different way. In Exercise 6.2 you worked on the units of covariance, and from that is should become obvious that it is not straightforward to transform the units of covariance into the original units of measurement. In addition, the covariance can take on values in the range `-inf` to `+inf`. The **pearson correlation coefficient** solves some of these issues: it *normalizes* the covariance to be restricted to the range `-1` to `+1`, and as a side-effect is a *dimensionless* quantity.

To normalize the covariance, the pearson correlation coefficient divides the covariance by the product of the squared variances (Equation 6.4).

$$PearsonCorrelationCoefficient(X,Y) = rX,Y = \frac{Covariance(X,Y)}{\sqrt{Variance(X)}\sqrt{Variance(Y)}} \tag{6.4}$$

This is also often expressed in terms of standard deviations σ:

$$r_{X,Y} = \frac{Covariance(X,Y)}{\sigma_X \sigma_Y} \tag{6.5}$$

> **Note 5: Useful properties of the pearson correlation coefficient**
>
> The pearson correlation coefficient r has a few useful properties that help interpretation.
>
> - **Range** `-1` to `+1`: correlation coefficients of 1 (either positive or negative) indicate a *perfect* relationship (in other words, all values are exactly equal). A correlation coefficient of 0 indicates *no* relationship.
> - The **sign** indicates the direction of the relationship: positive coefficients indicate a high value in one variable corresponds to a high value in the other variable, negative coefficients indicate a high value in one variable corresponds to a low value in the other variable.
> - It measures **linear** relationships. In other words: it draws a *straight line* through the datapoints. There is a strong connection with Ordinary Least Squares regression, in this course we do not go into detail about this relationship.
> - It is **dimensionless**: since it has no units, and the range is always the same, the interpretation of the correlation coefficient is always the same. In other words: the interpretation of the correlation coefficient doe s not depend on the units of the measured variable

Exercise 6.3 (Verify the calculation of the pearson correlation coefficient.)**.** Equation 6.5 describes the pearson correlation coefficient in terms of covariance and standard deviations. Pick two variables of your choice in the `corre_continuous` dataset and verify the output of `cor()` with calculating the coefficient yourself using `cov()` and `sd()`.

Exercise 6.4. Since the pearson correlation coefficient describes a linear relationship, it is relatively straightforward to draw this relationship as a straight line in a scatterplot. Using `ggplot`, create a scatterplot of your two variables of interest with `geom_point` and add

the correlation line using `geom_smooth`. Make sure to use `method = 'lm'` for `geom_smooth`. What is the relationship between the pearson correlation coefficient and the line you've drawn?

6.3 Non-linear Spearman correlation

Not all real data is best described by a straight line (Figure 6.1)! We will not go into too much detail on how to express non-linear relationships here (there are many options, most are outside the scope of this course). However, a relatively straightforward modification of the spearman correlation coefficient makes it somewhat suitable to express non-linear relationships. The *spearman correlation coefficient* is a non-linear correlation approach that simply calculates the pearson correlation coefficient on the *ranks*[1] of the variables. In doing so, spearman correlation effectively ignores the magnitude of the differences in a variable: big jumps are just as important as small jumps. Spearman correlation keeps most of the properties of pearson correlation (Note 5) except one: the relationship is no longer linear, but instead *monotonic*: the variables consistently increase or decrease together, without requiring a straight-line relationship.

Exercise 6.5 (Verifying the spearman correlation coefficient)**.** Section 6.3 describes how non-linear spearman correlation is a simple modificiation of pearson correlation. Verify this is true: you can get the ranks of a variable by using the `rank()` function. Pick two variables of your choice, and compare spearman correlation (`cor()` function with `method = spearman`) to pearson correlation on the ranks.

Exercise 6.6 (Interpreting the difference between Spearman and Pearson correlation)**.** For two variables of your choice, compute both the spearman and pearson correlation coefficient. Are they different? How different are they? What do you think this means?

Exercise 6.7 (Computing many correlations on the same dataset)**.** For datasets with multiple quantitative variables, a straightforward question is to ask 'which variables are correlated with each other?'. The built-in `cor()` function makes use of vectorization (See Section 4.3.3) to make this a straightforward question to ask.

- Select all numeric columns of the `grassland_traits_environment` dataset:

```
numeric_columns <- sapply(grassland_traits_environment, is.numeric)
grassland_numeric_df <- grassland_traits_environment[numeric_columns]
```

- Compute pairwise correlations of all variable pairs:

[1] i.e. what is the first biggest number, and the second biggest, etc.

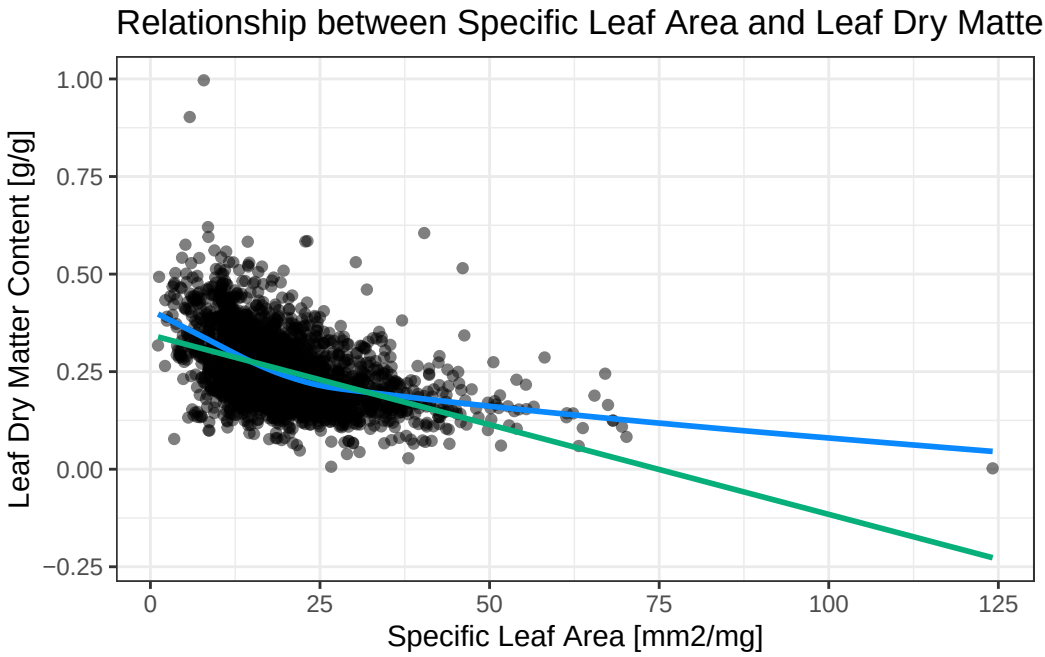


Figure 6.1: The relationship between Specific Leaf Area and Leaf Dry Matter Content in the CoRRE dataset. Green line represents a linear model fit and is directly related to the pearson correlation between these two variables. Blue line represents a non-linear model fit and is probably a better fit for this relationship. Note: the blue line is *not* spearman correlation, but based on a different non-linear model fit that is outside the scope of this book.

```
cor(grassland_numeric_df, use = 'complete.obs')
```

What does the output look like? How do you interpret all these numbers? What does `use = "complete.obs"` do (Hint: check the documentation with `?cor`)?

6.4 Testing and interpretation (and how to build an argument)

From Note 5 it has become clear that a correlation coefficient of zero indicates *no relationship*. But when do we call a non-zero correlation coefficient sufficiently large to indicate a *significant* relationship? This is a question that typically lends itself for a statistical approach! In Section 6.4.1 we will not go into too much detail, but a brief description of how significance testing for correlation coefficients work is necessary to interpret results. Once we have established how to determine which correlation coefficients can be seen as *statistically significant*, we ask the question "how to interpret a significant correlation?" in Section 6.4.2.

6.4.1 Testing

The main points for statistical testing of correlation coefficients are bundled in Note 6.

Note 6: Correlation coefficient significance testing

To test whether a correlation coefficient is (statistically) significantly different from zero, we take into account three things:

- **Sample size** (n): with larger samples we are more confident our observation is 'real'
- **Magnitude** of the correlation coefficient: a larger coefficient is more likely to be significant
- Chosen **significance level** (α): the typical statistical approach, we test a null hypothesis $H_0 : r = 0$, if the p-value is smaller than α we conclude significance.

For **Pearson correlation** the test looks as follows: we compute a t-statistic based on sample size n and correlation coefficient r, and we compare the t-statistic to a t-distribution to get a p-value. Approaches for other correlation coefficients are similar in nature, but differ in details.

$$t = r\sqrt{\frac{n-2}{1-r^2}} \tag{6.6}$$

From Equation 6.6 it can be observed that a large sample size n will *always* lead to a large test statistic t, which in turn will *always* lead to a low p-value. Table 6.1 provides a few examples of which correlation coefficient will be significant at various sample sizes.

Table 6.1: Pearson r significance levels at different sample sizes using $\alpha = 0.05$

Sample size	Smallest |r| that is significant
n = 10	▯ 0.63
n = 30	▯ 0.36
n = 100	▯ 0.20
n = 1000	▯ 0.06

The **take home message** here: A correlation is significant when the data provide enough evidence that the true association is not zero. This depends more on sample size than on the strength of the relationship!

Exercise 6.8 (Exploring Anscombe's dataset)**.** In this assignment you'll explore the limitations of summary statistics and correlation. Understanding these limitations is very important when analysing real datasets. To make the limitations very explicit, you will use a artificial dataset designed in 1973 by statistician Francis Anscombe. There are four pairs

of observations in this dataset, so it's often referred to as 'Anscombe's quartet'. You can load the dataset into your R session with `data(anscombe)`, the corresponding dataframe will be called `anscombe`.

- Calculate the main summary statistics discussed in Section 4.2 using the `summary()` function. What do these look like?
- Calculate correlation coefficients for all pairs of `X` and `Y`, check both Pearson and Spearman. What do these look like?

After calculating the summary statistics, let's take a look at a plot of these four pairs of observations. Copy-paste the code below[2] and run it in your R session. What do you observe? How does this affect the interpretation of the summary statistics and correlation coefficients?

```
# Plotting anscombes quartet
library(ggplot2)

# Load the built-in dataset
data(anscombe)

# Reshape to long format
anscombe_long <- data.frame(
  x = c(anscombe$x1, anscombe$x2, anscombe$x3, anscombe$x4),
  y = c(anscombe$y1, anscombe$y2, anscombe$y3, anscombe$y4),
  set = factor(rep(1:4, each = nrow(anscombe)))
)

# Plot
ggplot(anscombe_long, aes(x, y)) +
  geom_point(size = 2) +
  geom_smooth(method = "lm", se = FALSE, color = "red") +
  facet_wrap(~ set, ncol = 2) +
  labs(
    title = "Anscombe's Quartet",
    x = "x",
    y = "y"
  ) +
  theme_minimal()
```

[2] We will cover 'long' and 'wide' dataframes and how to convert between these in the next chapter

6.4.2 Interpreting correlations (and building strong arguments)

The main interpretation of finding a (significant) correlation between two variables, is that these two variables are *somehow* related. This might seem obvious at first, but there are a few subtle nuances that are important to realize that might lead to *over interpretation* when ignored. The most important nuance of correlation, is that it is not causation (See Warning 2).

⚠ Warning 2: Correlation is not causation

Correlation describes **that** two variables are related, it does *not* tell you **why** the variables are related. In many cases you will be interested in asking the 'why' question, in which case correlation might give you a hint. However, a correlation coefficient alone is never enough to make a causal statement.
A good visual aid in remembering this, is that correlation is an *undirected* relationship, whereas causation describes a *directed* relationship: note the arrow vs straight line in Figure 6.2 below.

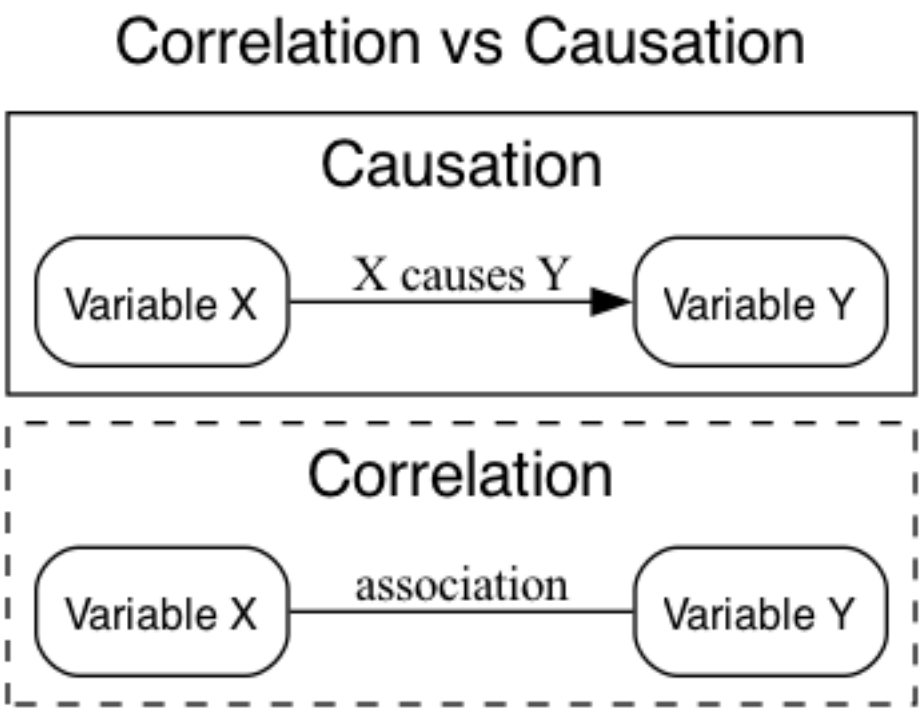


Figure 6.2: Correlation is undirected, causation implies a direction.

There are many examples of correlations that are not causal (also called *spurious* correlations), for example those collected by Tyler Vigen. Perhaps counter intuitively, there are also clear cases where a causal relationship does *not* lead to a correlation (See Figure 6.3 for an example in the plant sciences).

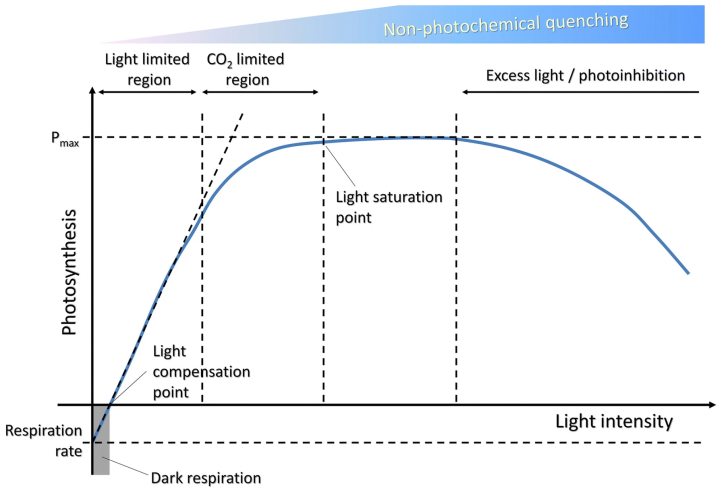


Figure 6.3: Light intensity and photosynthesis efficiency are clearly causally related, but show no inherent correlation when measured: the relationship is neither linear nor monotonic. Figure from [3]

Tip 4: Building a scientific argument based on correlation

A strong scientific argument based on correlation builds on a few principles, which are outlined here. Building scientific arguments *in general* is a fundamental aspect of good science, but outside of the scope of this course.
When describing correlations, consider the following:

- A high correlation supports an argument, but never proves a causal mechanism (See Warning 2).
- Inspect the scatterplot: Exercise 6.8 provides clear examples of what can go wrong.
- Check you assumptions: linear versus non-linear? What do you know about the data that would suggest either of these to be a better fit?
- Effect size is more important than significance: with enough observations, *every* correlation is significant.
- Robustness: what happens if you remove obvious outliers?
- Provide additional evidence: are there existing theories or mechanisms known?

Take home message: A correlation coefficient is evidence, not a verdict. Strong scientific arguments combine visual inspection, use appropriate methods, consider effect size, and include (biological) reasoning about the context.

Exercise 6.9. Revisit Exercise 6.7: Are there any groups of variables that are all correlated to eachother? How would you interpret such a group of variables? Which of the steps of Tip 4 can you apply, and which are not feasible? How does this affect your conclusions?

7 Diving into the tidyverse

Throughout the book you have encountered packages from the **tidyverse**: one of the first things you did in Section 1.1 was importing the `tidyverse` package. In this chapter we dive into more detail about what the tidyverse actually is, how it relates to important data analysis concepts, and how it differs from some base R concepts.

The main reasons we dedicate a chapter to this are: 1. The tidyverse is used *a lot*: it will show up in many published data analysis approaches. 2. The approaches in the tidyverse are *quite different* from what you see when using base R: especially the use of the pipe operator deserves some attention.

Figure 7.1: The tidyverse is a *universe* of R packages for working with *tidy* data

7.1 What is the tidyverse?

In essence, the tidyverse is not *just* a package, it is a **collection of packages**. The main application of the tidyverse is working with *tidy* data: any data that can naturally be represented in tables where columns are variables and rows are observations. Coincidentally, many data science and statistics approaches work with such tabular data, making the tidyverse a natural fit. An example of a very common problem that is elegantly solved in the tidyverse is going back and forth between *long* and *wide* representations of tabular data (Note 7).

Note 7: Long VS wide tables

The tidyverse is designed to work with tables where rows are observations and columns are variables. However, some of the language features of the tidyverse are difficult to implement when the columns of a table represent the *actual* variables [1]. A common solution in the tidyverse is to introduce two *abstract* columns: a `name` and a `value` column, and to map the original variable names to the `name` column and the variable value to the `value` column. A table with the actual variables as columns is called a **wide** table, a table with abstract columns `name` and `value` is called a **long** table. A straightforward example can be a series of height measurement of various plants:

Table 7.1: Wide format table of a time-series plant growth experiment

plant_id	day_0	day_1	day_3	day_7
1	1	2	5.5	7
2	1.2	1.3	4	8

Table 7.2: Long format table of the same time-series plant growth experiment

plant_id	name	value
1	“day0”	1
1	“day1”	2
1	“day3”	5.5
1	“day7”	7
2	“day0”	1.2
2	“day1”	1.3
2	“day3”	4
2	“day7”	8

Wide tables are more readable for humans, long tables are more easy to perform computations on. Crucially, the *information* contained in a long or wide table is identical. The tidyverse implements very clean logic to alternate between these two representations with `pivot_wider()` and `pivot_longer()`.

The overarching design principle can be described as making data analysis code readable for humans. To achieve this, several packages implement very specific language features

[1] This happens when e.g. variables are compared, or used for subsetting: in both cases variables are treated as data, which is easier when they are actually *in* the table

that bring the R language closer to the English language [2]. In the following sections, we outline how `dplyr`, `tidyr`, and `ggplot2` implement their own specific language features.

7.2 Dplyr and tidyr

Within the tidyverse, dplyr and tidyr implement most tidy language features for *transforming* tabular data. Although they are distributed as separate packages, they are best understood as two parts of a single language for describing how a table should change.

The difference between them is not the type of data they work on, but the aspect of the table they modify:

- `dplyr` transforms the **contents** of a table (rows and columns)
- `tidyr` transforms the **structure** of a table (long to wide and vice-versa)

Both rely on a small set of *verbs* (implemented as functions) that describe common data manipulation tasks, such as selecting rows (`filter()`) or columns (`select()`), creating or transforming variables (`mutate()`), computing summaries (`summarise()`), or reshaping data (`arrange()`,`group_by()`). More complex operations are expressed by *combining* these verbs.

These verbs are typically connected using the pipe operator `|>`[3], which passes the result of one step directly to the next (thus creating a *pipe*-line). The trick here is that by using the pipe operator, you automatically inject the *output* of the first function into the *first argument* of the second function. Most tidyverse functions are designed to take a `tibble` or `dataframe` as first argument, and also return a `tibble`, making it straightforward to chain operations. This allows data manipulation code to be written as a sequence of transformations, read from top to bottom.

While `dplyr` keeps the overall table structure intact, `tidyr` is used when the structure itself gets in the way of analysis. In particular, it provides tools to convert between wide and long tables by making information encoded in column names explicit.

In practice, `dplyr` and `tidyr` are almost always used together. From the user's perspective, they form a single, coherent language for data transformation, well suited for exploratory data analysis.

[2] A (part of a) programming language that is aimed at a very specific case is often referred to as a Domain Specific Language

[3] Modern R versions include the pipe operator as part of the default language. In older versions of R you might see %>% being used, which does mostly the same thing but was introduced by the `magritrr` package

Tip 5: Comparing dplyr with base R

Any operation that can be performed with the tidyverse can in principle also be performed with base R. The tidyverse should mostly result in code that is more natural to read for humans, making it easier to follow which steps are being taken. The following examples compute the exact same summaries, once in base R, and twice using dplyr verbs (once with and once without the pipe operator).

7.2.0.1 Base R

```
# Base R code to compute mean leaf tissue density for annuals and perennials seperately
# 1. Remove rows with any NA
dat <- grassland_traits_environment[complete.cases(grassland_traits_environment), ]

# 2. Split data by life_history
split_dat <- split(dat, dat$life_history)

# 3. Compute mean ld per group
result <- data.frame(
  life_history = names(split_dat),
  mean_ld = sapply(split_dat, function(x) mean(x$ld))
)

result
```

```
          life_history   mean_ld
Annual          Annual 0.3368054
Perennial    Perennial 0.3759089
```

7.2.0.2 dplyr


```
# dplyr code to compute mean leaf tissue density for annuals and perennials seperately
library(tidyverse)

grassland_traits_environment |>     # Dataset
  drop_na() |>                      # Remove rows with NAs
  group_by(life_history) |>         # Group by life history
  summarise(mean_ld = mean(ld))     # Compute mean ld per group
```

Exercise 7.1 (Compare dplyr and base R results)**.** In Tip 5 there are three examples of the same analysis, one in base R and two using dplyr. You'll notice the numbers are the same, but that one returns a regular dataframe and the others return a 'tibble' (if you want you can verify this by copying and pasting the corresponding code). Tibbles are also part of the tidyverse, take a look at the tibble homepage and describe the main differences with regular dataframes.

Which `dplyr` version of the code do you prefer and why?

7.3 GGplot2

Just like `dplyr` and `tidyr` introduce specific language elements for transforming data, `ggplot` introduces specific language elements for creating data visualizations. More specifically, it uses a hierarchical *Grammar of Graphics*[4].

A data visualization according to the grammar of graphics in ggplot2 is built up around the following elements:

- Defaults: used for all other layers unless otherwise specified
  - Dataset (typically a dataframe)
  - Aesthetic mappings (e.g. what goes on the x and y axes, colors, fills, etc.)
- Layers:
  - Geometric objects (e.g. a point, line, box, ribbon, etc.)
  - Statistical transformation
  - Position adjustment
- Scales: mapping of data to aesthetic attributes (e.g. choosing specific palettes)
- Coordinates: mapping of data to the plane of the plot (mostly x,y but could also be e.g. polar coordinates)
- Facets: split up the data
- Themes: style choices for e.g. background colors, line widths, etc.

The different elements of a ggplot2 visualization are implemented in corresponding functions: the `ggplot()` function takes care of the defaults, the other elements often have *prefixed* functions indicating there role: `geom_*()`, `facet_*()`, `coord_*()`, or `theme_*()`.

Unlike `dplyr` and `tidyr`, `ggplot2` elements are *not* combined with the pipe operator. This follows the convention of making tidyverse language match natural language: different elements are **added** to the plot, so `ggplot2` elements are combined with the `+` operator. Furthermore, where there are often one-to-one base R implementations of `dplyr` and

[4]Indeed, the GG in GGplot stands for Grammar of Graphics

`tidyr` pipelines, the same is not true for `ggplot2`: base R hase good default plotting functions, but `ggplot2` offers clear improvements in capability and flexibility.

Exercise 7.2 (Revisit tidyverse code from week 1)**.** In Exercise 1.12 you were presented with code that made extensive use of the tidyverse. Look at this code again, and see whether you can identify the different tidyverse concepts that have been introduced in this chapter. Next, we are going to modify the code from the earlier assignment, so that we can create a *single* ggplot2 plot of a few of the accessions.

Using tidyverse concepts, do the following:

- Load the day1 dataset (`BIF20806_Warmerdam_dataset.txt`), you can find this on the week1 BrightSpace
- Select only the following plant genotypes: 'Col-0','Bay-0','Ler-1','Hey-1'. Hint: you can use the `filter()` function and make use of the `%in%` operator.
- Make a ggplot2 boxplot with the datapoints added as points, put the plant genotype on the X-axis and use the plant genotype for color.

Which genotype has the highest eggmass? Which has the most observations? What would you do if you wanted to plot a few other genotypes (Hint: Section 4.1)?

- Next, add the following to your ggplot: `facet_wrap(~screening)`.

How does the plot change? What additional difference between plant genotypes do you notice? What does the `~` in `facet_wrap()` do?

7.4 Assignment for code review week 2

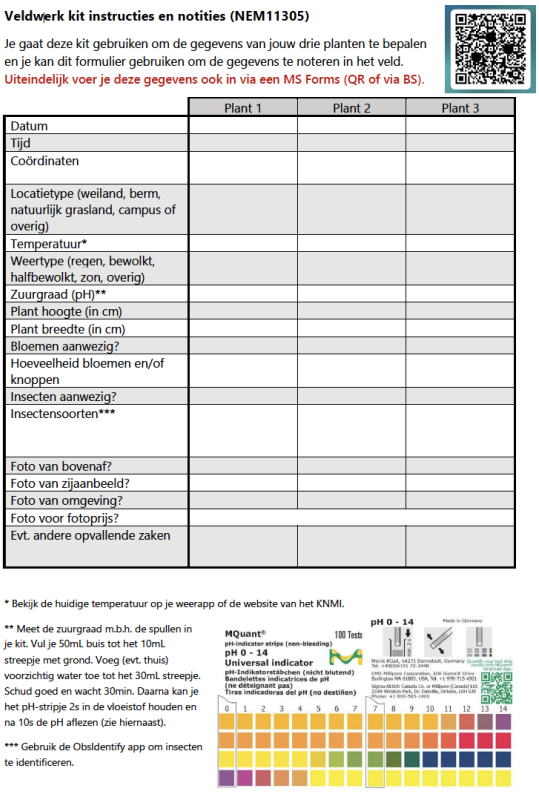
In the course Plant Science in Practice (NEM11305) you previously worked on collecting your own plant trait and environmental data. Specifically, you collected field observations for *Jacobaea vulgaris*, and recorded your observations using a form with standardized questions (Figure 7.2). For this assignment, you will (re-)analyse the data that you (and your fellow students) gathered yourself.

The dataset for this assignment is available on BrightSpace (Content –> Week 2 –> Assignment –> NEM11305 Dataset).

Exercise 7.3 (Analyse student-collected plant trait and environment data)**.** For this assignment you will hand in a quarto notebook (`.qmd` file) that contains your code and interpretation. The work you hand in should contain the following:

Code for the following operations:

- Installing and loading the relevant packages



Veldwerk kit instructies en notities (NEM11305)

Je gaat deze kit gebruiken om de gegevens van jouw drie planten te bepalen en je kan dit formulier gebruiken om de gegevens te noteren in het veld.
Uiteindelijk voer je deze gegevens ook in via een MS Forms (QR of via BS).

	Plant 1	Plant 2	Plant 3
Datum			
Tijd			
Coördinaten			
Locatietype (weiland, berm, natuurlijk grasland, campus of overig)			
Temperatuur*			
Weertype (regen, bewolkt, halfbewolkt, zon, overig)			
Zuurgraad (pH)**			
Plant hoogte (in cm)			
Plant breedte (in cm)			
Bloemen aanwezig?			
Hoeveelheid bloemen en/of knoppen			
Insecten aanwezig?			
Insectensoorten***			
Foto van bovenaf?			
Foto van zijaanbeeld?			
Foto van omgeving?			
Foto voor fotoprijs?			
Evt. andere opvallende zaken			

* Bekijk de huidige temperatuur op je weerapp of de website van het KNMI.

** Meet de zuurgraad m.b.h. de spullen in je kit. Vul je 50mL buis tot het 10mL streepje met grond. Voeg (evt. thuis) voorzichtig water toe tot het 30mL streepje. Schud goed en wacht 30min. Daarna kan je het pH-stripje 2s in de vloeistof houden en na 10s de pH aflezen (zie hiernaast).

*** Gebruik de ObsIdentify app om insecten te identificeren.

Figure 7.2: Instructions for gathering field observations used in the course Plant Science in Practice (NEM11305). The standardized form should make downstream analysis more straightforward.

- Loading the dataset
- Creating a summary of the dataset
- A correlation analysis of stem length and plant width (‘Stengellengte’ and ‘Plantbreedte’ in the dataset)
- A boxplot visualizing the distribution of soil acidity (‘Zuurgraad’) across different soil types (‘Bodemtype’)

Text with the following interpretations/explanations:

- A brief description of what the dataset looks like: size, variables, summary statistics
- Interpretations for the correlation analysis and visualization

Make use of the quarto markdown elements to structure your work in a readable fashion.

Part III

Visualization & (un)supervised analysis

Part IV

III: Visualisation and unsupervised analysis

The goals for week 3

In the third week, the goals are:

1. Use classed objects
2. Use visualizations to explore large data sets
3. Apply clustering methods
4. Analyse a data set using PCA
5. Understand Differential gene expression

> **Tip 6: What are classed objects'?**
>
> In week 2 you already encountered object oriented programming (OOP). From Advanced R by Hadley Wickham "A class defines the behavior of objects by describing their attributes and their relationship to other classes. The class is also used when selecting methods, functions that behave differently depending on the class of their input."

How week 3 is organized

The material you are supposed to work through each day is a chapter in the book. During the scheduled tutorials background and context will be given. Also, there will be help available if you get stuck with the coding.

There will be three days of working through the chapters (Monday - Wednesday).

Monday Chapter 8 Tuesday Chapter 9 Wednesday Chapter 10,

We will mostly work on data associated with the paper ABA Is Required for Plant Acclimation to a Combination of Salt and Heat Stress by Suzuki et al. from 2016.

The assignment

Like before, at the end of Wednesday (23:59), a coding peer-feedback assignment is due, which you submit via feedback fruits on Brightspace. Your assignment will be reviewed by two students, as you will review the assignment of two other students. For this week the assignment consists of creating a set of R functions in a separate R-script, that can be used to perform RNA-seq analyses. Like before **Completing the assignment and participation in the code-review is mandatory.**

You need to hand in a script file called **rnaseq_functions.R** that should contain the definition of four functions you will write during the coming three days:

- `load_atlas_se(experiment_id, from_disk)`
- `get_normalized_counts(se, log, min_count, min_samples)`
- `two_gene_scatter_plot(genes, logcounts, coldata, title)`
- `pca_plot(logcounts, coldata, center, scale)`

The script should also contain a function called `run_tests()` that runs each of the functions with appropriate input to test them.

The code-review (peer-feedback via Brightspace) is due Thursday at 13:00. Instructions for how to give feedback can be found in Chapter 3. The items we expect you to give feedback on as well.

The exam

The exam of week 3 **does count for your final grade of the course**.

At the end of week 3 we expect you to be able to:

- Determine the class and structure of a classed object.
- Construct various types of plots to visually analyse your data.
- Perform hierarchical clustering and explain the key characteristics.
- Perform k-means clustering and explain the key characteristics.
- Explain the difference between supervised and unsupervised analysis.
- Perform a Principal Component Analysis and explain key characteristics of PCA.
- Explain Differential Gene Expression analysis.

8 Gene expression analysis: salt and heat stress in Arabidopsis

8.1 Introduction

Gene expression analyses can shed light on all kinds of biological processes and environmental responses, through identifying genes that are activated or silenced. In this session we will analyze RNA-seq gene expression data from *Arabidopsis thaliana* plants exposed to salt and heat stress and a combination of salt and heat stress.

The data originate from the study:
ABA Is Required for Plant Acclimation to a Combination of Salt and Heat Stress
https://doi.org/10.1371/journal.pone.0147625

We could start with the raw sequencing data, but that is beyond the scope of this course. Fortunately, the gene-level counts are already available through the **EMBL-EBI Expression Atlas** at https://www.ebi.ac.uk/gxa/experiments/E-GEOD-72806/

We will use this data set for:

- exploratory RNA-seq analysis
- clustering and principle component analysis
- differential gene expression (DGE)

8.2 Installing and loading required packages

First we need to install and load some packages:

```
if (!requireNamespace("BiocManager", quietly = TRUE))
  install.packages("BiocManager")

if (!requireNamespace("ExpressionAtlas", quietly = TRUE))
  BiocManager::install("ExpressionAtlas")

if (!requireNamespace("SummarizedExperiment", quietly = TRUE))
  BiocManager::install("SummarizedExperiment")

if (!requireNamespace("edgeR", quietly = TRUE))
  BiocManager::install("edgeR")

if (!requireNamespace("pheatmap", quietly = TRUE))
  BiocManager::install("pheatmap")

if(!requireNamespace("HDF5Array", quietly = TRUE))
  BiocManager::install("HDF5Array")

if(!requireNamespace("GenomeInfoDb", quietly = TRUE))
  BiocManager::install("GenomeInfoDb")

library(ExpressionAtlas)
library(SummarizedExperiment)
library(tidyverse)
```

> **i** Installing these packages can take quite some time. If you encounter errors during installation, these can be challenging to solve. It is usually a good start to properly read the error and look for clues. If the error is a version conflict, for instance on the `rlang` or `xfun` packages, it can help to re-install these: `install.packages("xfun")` `install.packages("rlang")`
> Clearing the workspace (under the Session menu in RStudio) and restarting the R session can help.
> If you get asked whether you want to update certain packages, you can select **a** for all.

8.3 Loading RNA-seq expression data

we can download the gene expression data directly from the EMBL-EBI Expression Atlas website using the getaAtlasExperiment method in the ExpressionAtlas library:

```
exp <- getAtlasExperiment("E-GEOD-72806")
```

This returns a list (actually a `SimpleList`) of data structures, in this case only one, let's extract that:

```
se <- exp[[1]]
```

> If the Expression Atlas site does not work, you can also download a copy from our website as a zip file, unzip it and load the data from there.
>
> ```
> download.file(
>   url = "https://www.bioinformatics.nl/courses/BIF-20806/E-GEOD-72806.zip",
>   destfile = "./E-GEOD-72806.zip",
>   mode = "wb")
> unzip("E-GEOD-72806.zip")
> library(HDF5Array)
> se = loadHDF5SummarizedExperiment(dir="E-GEOD-72806")
> ```

The gene expression data are in this `se` object.

Exercise 8.1. To learn more about the resulting data in `se` use the `class()` function to see what kind we are dealing with.

To further explore the structure of `se`, use the `str()` function to get more information about the content. `str` stands for "**st**ructure of an **R** object".

It is a `RangedSummarizedExperiment` from the package SummarizedExperiment. Try this to get more information about these:

```
?RangedSummarizedExperiment
?SummarizedExperiment
```

8.4 Storing the expression data in Hierarchical Data Format files

The Expression Atlas website can be unresponsive at times, so best to store the data structure locally. Because it is a complex data set, we cannot simply save it as a csv file. Instead, we can use the Hierarchical Data Format (HDF) file format. In the `HDF5Array` library there are specific functions to store and load `SummarizedExperiment` files:

```
library(HDF5Array)
saveHDF5SummarizedExperiment(se,dir="E-GEOD-72806")
```

Using the `dir()` function you can check that a directory was created called **E-GEOD-72806**.

From now on we can load the data as:

```
se = loadHDF5SummarizedExperiment(dir="E-GEOD-72806")
```

Exercise 8.2. Create a function called `load_atlas_se` in your **rnaseq_functions.R** script that returns a `RangedSummarizedExperiment` object either from disk or directly from the EMBL-EBI Expression Atlas. The function should take the accession as input and a Boolean (TRUE/FALSE) to determine whether to load the data from disk or from the website. The default value for the Boolean should be TRUE.

```
load_atlas_se <- function(accession,from_disk = TRUE) {
  ...
  return(se)
}
```

The different kinds of data in the **se** object are stored in different slots. To see which these are, you can use the **slotNames()** method on the **se** object. You can access these slots use the **@** character. For instance, to see how the raw sequencing data were filtered you can look in the **metadata** slot, which returns a list, and then the **filtering** element like this:

```
se@metadata$filtering
# or
se@metadata[["filtering"]]
```

Notice that for accessing an element of a list you do not use an **@** character, but instead a dollar sign or double square brackets.

However, accessing slots directly is like opening the bakery's display case and taking a cake without going through the counter. It might work, but you are bypassing the process that keeps inventory, presentation, and handling consistent. The recommended way to access data in a complex data structure like this is via accessor functions:

```
metadata(se)$filtering
```

To find the proper accessor function, the help text for the class is usually the best place to look.

8.5 Exploring the count data

To get the actual expression data in the `RangedSummarizedExperiment`, we can use the `assay()` function:

```
counts <- assay(se) # genes x samples
counts
```

This returns a matrix with rows representing genes and columns representing samples. The class of counts is `DelayedMatrix`, which behaves like a regular numeric matrix, but computations are performed lazily. This means for instance that operations like log-transformation or scaling are not executed immediately; instead, they are applied only when the data are accessed. This allows efficient handling of large datasets that may reside on disk rather than in computer memory. To view the counts matrix in RStudio, try the following commands:

```
View(counts)
View(as.matrix(counts))
```

Let's look at the dimensions of counts:

```
dim(counts)
```

There are more than 32000 genes and 12 samples with various treatments. The columns are the samples, to get information about the samples we can use the `colData()` function:

```
coldata <- colData(se) # sample level information
coldata
```

The samples have non-descriptive names like "SRR2302908", but the **environment_stress** column tells us which stress the sample was subjected to. The other columns do not help to differentiate between the samples. We can clean up the coldata DataFrame to only keep the enviromental_stress column. You probably did not notice it, but coldata is a Bioconductor `DataFrame` object and not a base R `data.frame`. We can fix that in the same command:

```
coldata <- as.data.frame(coldata[, "environmental_stress", drop = FALSE])
```

In case you wonder about the `drop = FALSE` in `coldata[, "environmental_stress", drop = FALSE]`: By selecting only the "environmental_stress" column, we get a `data.frame` with only one column. Default R behavior is to 'simplify' this to a vector `c()`, which means we lose the row names which represent the sample labels. `drop = FALSE` prevents this default behavior, so the result is still a `data.frame`.

Now to get some idea of the gene expression values (the counts) for the RNA-seq experiment, we can first sum the counts per sample (so per column). We can use the `apply()` function for this, by allowing us to apply the `sum()` function to each column. `DelayedArray` objects also have a specific function called `colSums()` to do the same.

```
apply(counts,2,sum) # or colSums(counts)
```

apply (1)

The `apply(X, MARGIN, FUN)` function performs the function given by FUN on all rows or columns (indicated by the MARGIN value: 1 for rows and 2 for column) of the matrix X. So in the code above, we calculated the `sum()` per column of `counts`.

In the Suzuki et al. paper we can read that they generated on average 14 million sequencing reads per sample, the sums you see here are after filtering to remove bad quality reads. Do the sums match the number from the paper?

One sequencing read corresponds to one mRNA (fragment), so they can be used to determine the abundance of the different mRNAs and quantify the expression of each Arabidopsis gene.

To get an idea of the overall properties of the counts, we can plot their distribution. For this we look at the counts of all genes and samples together, so let's flatten the `counts` matrix into one vector:

```
expr <- as.vector(counts)
```

And then plot this as a histogram with 100 bins:

```
df_expr <- data.frame(expr = expr) # ggplot needs a data.frame

library(ggplot2)

ggplot(df_expr, aes(x = expr)) +
  geom_histogram(bins = 100) +
  labs(title = "Distribution of raw gene counts",
  x = "Counts", y = "Frequency")
```

This is not very informative, is it? That is because there is very large difference in expression between genes. A common trick to bring values in the same range is to performa a log transformation of the counts. We can plot the log10 transformed counts with a simple addition:

```
ggplot(df_expr, aes(x = expr)) +
  geom_histogram(bins = 100) +
  scale_x_log10() +
  labs(title = "Distribution of raw gene counts (log10 x-axis)",
    x = "Counts", y = "Frequency")
```

Exercise 8.3. You may see a warning that the `scale_x_log10()` introduced infinite values, can you explain why and for which expression values this happened?

We can avoid the "infinite values" warning by using the the `log1p` method:

```
ggplot(df_expr, aes(x = log1p(expr))) +
  geom_histogram(bins = 100) +
  labs(title = "Distribution of log1p-transformed gene counts",
    x = "log1p(count)", y = "Frequency")
```

Exercise 8.4. What is the difference with the previous plot? In which two ways does `log1p()` differ from `log10()`?

For further analyses, we are not interested in genes that have little to no expression, so let's filter these out:

```
keep <- rowSums(counts >= 10) >= 3
counts_filt <- as.matrix(counts[keep, ])
```

Let's unpack what happens here: the `counts >= 10` results in a matrix with Boolean/logical (TRUE or FALSE) values. `rowSums()` counts a TRUE as 1 and a FALSE as 0. The resulting `keep` variable is a vector of logical values, which is used to select rows (genes) of the counts matrix.

Exercise 8.5. How many genes are left? Describe which criteria genes have to meet to be kept.

The differences in expression level per gene can vary a lot. If we create a line plot of the (sorted) mean expression values per gene, we can see that a small proportion of genes are very highly expressed:

```
gene_means <- rowMeans(counts_filt)
gene_means_sorted <- sort(gene_means)

plot(
  gene_means_sorted,
  type = "l", #line
  xlab = "Genes (sorted by mean expression)",
  ylab = "Mean expression",
  main = "Sorted gene mean expression"
)
```

By now it should be clear that for plotting gene expression values a log transformation is advised to not let these highly expressed genes dominate the analyses

```
logcounts = log1p(counts_filt)
```

8.6 Top 10 most highly expressed genes

Now let's look at the top 10 most highly expressed genes, which are actually the bottom 10 of the sorted list that we just plotted.

```
top_n <- 10
top_genes <- names(tail(gene_means_sorted,top_n))
mat_top <- logcounts[top_genes,] # select the rows by the row names of logcounts
```

To plot the expression of these genes in all samples we can use a fancy heatmap that also allows us to add annotation from the metadata of the samples.

```
library(pheatmap)

anno_col <- coldata  # sample annotations

pheatmap(
  mat_top,
  cluster_rows = FALSE,
  cluster_cols = FALSE,
  annotation_col = anno_col,
  show_colnames = FALSE,
  fontsize_row = 7,
  scale = "none",
  main = paste("Top", top_n, "most highly expressed genes")
)
```

Exercise 8.6. In this plot you can already discern a pattern, at least for the 2 or 3 most highly expressed genes. Still the plot can be improved in a number of ways:

- the most highly expressed genes still take most of the attention, the *scale* argument can improve that by converting the values to z-scores per row or column. Set the scale argument to the appropriate direction (row or column) to see the relative expression over the different samples for the less highly expressed genes.
- with clustering you can group samples and/or genes based on how similar their expression values are, more about that tomorrow in the clustering session. Try out what happens if you cluster genes and samples in this plot.

```
anno_col <- coldata

pheatmap(
  mat_top,
  cluster_rows = FALSE,
  cluster_cols = FALSE,
  annotation_col = anno_col,
  show_colnames = FALSE,
  fontsize_row = 7,
  scale = "none",
  main = paste("Top", top_n, "most highly expressed genes")
)
```

8.7 Normalization

So we have looked at the expression at the gene level, let us get back to the total number counts per sample and address the question do all samples have the same number of counts? The total number of counts per sample is also called *library size*:

```
libsize <- colSums(counts_filt)

barplot(
  libsize,
  las = 2,
  ylab = "Total counts",
  main = "Library size per sample"
)
```

The total counts are pretty similar but not identical. If we compare the sample with the most counts and the sample with the fewest counts, there is a 20% difference:

```
min(libsize)/max(libsize)
```

The total number of counts (reads) per sample is more a technical than a biological aspect of the data, you typically ask the DNA sequencing facility for a certain number of reads per sample. Deviation from this requested number is usually due to variation in sample quality and/or sequencing efficiency.

If we look at the correlation between the counts for the top 10 most highly expressed genes with the library size, some are very strongly correlated.

```
apply(mat_top, 1, cor, y = libsize, method = "pearson")
```

apply (2)

This use of `apply()` is a bit more complicated than before. That is because the `cor()` function requires more information than just the rows or columns of the given matrix. The additional arguments of `apply()` are passed on to `cor()`. So for every row (second argument is a `1`) of `mat_top`, the function calculates the Pearson correlation with the `libsize` vector.

We just discovered something uncomfortable: different samples have quite different total numbers of reads. That means our analyses so far were biased. How do we deal with this? A simple and reasonably effective method is to just correct the counts for the total number of reads per sample. But just dividing each count by the total counts per sample would create very small numbers, so the result after dividing is commonly multiplied by a million to generate so-called **Counts Per Million** or CPM.

$$CPM_i = \frac{counts_i}{\sum_{n=1}^{n} counts_n} \times 10^6$$

Summing all CPM normalized counts in a sample would then give a uniform total of 10^6

In practice, this does not remove all bias, so more sophisticated normalization methods are used in RNA-seq analysis. To understand what the problem is with the simple method, think of the following scenario: a treatment causes a very strong increase in the expression of only the top 10 most highly expressed genes, but does not affect any other gene. The top 10 genes now take an even larger proportion of the total counts, which effectively lowers the relative counts for all the other genes making it seem like their expression decreased due to the treatment. The problem is that the simple scaling-based normalization is sensitive to a change in composition.

A normalization method that addresses this is called TMM, or “Trimmed Mean of M-values”. This specifically ignores genes that change a lot between different samples in

calculating a scaling factor. It is good to realize that TMM only works well if the majority of genes do not change much in expression.

We can perform TMM normalization step-by-step, but we can also use an R library that was designed for RNA-seq analyses called **edgeR**.

```
library(edgeR)

dge <- DGEList(counts = counts_filt) # create a DGEList object
dge <- calcNormFactors(dge) # calculate the normalization factors and include these in the DG

cpm <- cpm(dge, log = FALSE) # extract the CPM normalized counts from the DGEList object

barplot(
  colSums(cpm),
  las = 2,
  main = "Effective library sizes after TMM",
  ylab = "Normalized library size"
)
```

Did that help?

We will come back to `edgeR` on Wednesday when we will use it to find which genes are significantly up- or down-regulated in response to the stress treatments. These genes are called Differentially Expressed Genes or **DEG**, and the analysis is called Differential Gene Expression or **DGE**.

Exercise 8.7. Create a function called **get_normalized_counts** in your **rnaseq_functions.R** script that does the following:

- extract the counts from a `RangedSummarizedExperiment`
- removes lowly expressed genes
- normalizes the counts using edgeR's `cpm()` function
- optionally log transforms the counts

It should take as input a **RangedSummarizedExperiment** object, a logical value **log** indicating whether the counts should be log2 transformed (default TRUE) and two numeric values that are used for filtering: **min_count** and **min_samples**. These should have default values 10 and 3. It should return the cpm values:

```
get_normalized_counts<- function(se, log = TRUE, min_count = 10, min_samples = 3) {
  ...
  return(cpm)
}
```

9 Clustering

9.1 Introduction

In this session we continue with the gene expression dataset from https://www.ebi.ac.uk/gxa/experiments/E-GEOD-72806/

First load the proper libraries and filter/transform the data using the functions that you created yesterday. To make the functions available in the current R session, you can use the `source()` function to run the R script.

```
library(HDF5Array)
library(edgeR)
library(SummarizedExperiment)
library(ExpressionAtlas)

source("./rnaseq_functions.R")
se <- load_atlas_se("E-GEOD-72806")
logcounts <- get_normalized_counts(se)
coldata <- as.data.frame(colData(se)[, c("environmental_stress"), drop = FALSE])
```

We have already seen that the 12 samples represent 4 treatments, with 3 replicates per treatment. We expect the replicates to have similar gene expression values. To check this, we can cluster samples based on these expression values. You were already introduced to clustering on day 2 of the course. Then you used hierarchical clustering, today we will also look at another common clustering method called *k*-means.

In clustering we group samples together that are similar, but how do we determine how similar samples are? We need a similarity measure. This is usually derived from a distance measure, which is inversely related to the similarity.

9.1.1 Euclidean distance.

An often-used distance measure is the Euclidean distance, which is simply the straight-line distance between two points. We can consider the samples to be points in a multidimensional space, where each gene is a dimension and the position of the samples is

determined by the expression values. This becomes quite difficult to picture, so let's start with defining a sample by only two genes, x and y. For the Euclidean distance we can use **Pythagoras**:

$$d(\text{sample}_1, \text{sample}_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Where x_1 is the expression value of gene x in sample 1, etc.

For all genes this would look like this:

$$d(\text{sample}_1, \text{sample}_2) = \sqrt{\sum_{i=1}^{p} (a_i - b_i)^2}$$

Where: p = number of genes, and a_i and b_i is the expression of gene i in sample a and sample b

Starting with the two gene case for clarity, we can select the two genes that have a stable expression in the dataset (genes having the smallest variance):

```
gene_var <- apply(logcounts, 1, var)
genes <- names(sort(gene_var, decreasing = FALSE))[1:2]
```

and plot the samples using ggplot according to the expression of these two genes.

```
library(ggplot2)

# Extract expression of the two genes and transpose
expr_df <- as.data.frame(t(logcounts[genes, ]))

# Make sure expr_df has the sample names as a column
expr_df$sample = rownames(expr_df)

# Make sure coldate has the sample names as a column
coldata$sample = rownames(coldata)

# Merge expression data with metadata, on the sample column
df <- merge(
  expr_df,
  coldata,
  by = "sample",
  all.x = TRUE   # left join
)
```

```
p2 <- ggplot(
  df,
  aes(
    x = .data[[genes[1]]],
    y = .data[[genes[2]]],
    color = environmental_stress
  )
) +
  geom_point(size = 4) +
  geom_text(aes(label = sample), vjust = -0.8, size = 3) +
  labs(
    title = "Samples projected onto the two least variable genes",
    x = paste(genes[1], "(log(cpm) expression)"),
    y = paste(genes[2], "(log(cpm) expression)")
  ) +
  theme_bw()

p2
```

This shows the samples in a 2-dimensional scatter plot, but does not really help to group the samples according to their treatment...

We can try the same for the two most variable genes:

```
genes <- names(sort(gene_var, decreasing = TRUE))[1:2]
```

Exercise 9.1. Now we could copy the plotting code from above, but copying a block of code is generally not a good idea, it is better to turn that into a function that you can call. So please go ahead and create a function that takes the list of genes as input, as well as the data that is needed to make the plot (logcounts and coldata). It is also a good idea to include the title of the plot as parameter. Add this function to your **rnaseq_functions.R** script

```
two_gene_scatter_plot(genes, logcounts, coldata, title)
```

Now we can see that the samples form two groups.

Exercise 9.2. Based on their treatments, which stress response would you say that the genes are involved in?

The gene IDs are not really informative to learn more about the kinds of genes that we are dealing with, but you can look them up at the TAIR website https://www.arabidopsis.org. Does it make sense that you find these genes upregulated in the stress response?

9.2 hierarchical clustering

We can use the 2-gene representation of the samples to find clusters of similar samples (like you already did on day 2 of the course). Hierarchical clustering starts with each sample in its own cluster and then step-wise merges the two most similar clusters based on a distance measure (Euclidean, correlation-based, etc.) until all samples are in one cluster. This is typically depicted as a tree/dendrogram. This merging can be stopped if a certain number of clusters is reached. How the distance is calculated depends on the distance measure, but also on the positions in the clusters the distance is calculated between. For this several so-called **agglomeration** or **linkage methods** are used that give different results. With **average linkage**, the distance between two cluster is calculated as the average of all distances between the members of both cluster. In **complete linkage**, the largest distance between any two members of two cluster is used, while **single linkage** uses the smallest distance between any two members of both clusters. You can imagine that both the distance measure as well as the linkage methods affect the clustering.

```
expr_df <- as.data.frame(t(logcounts[genes, , drop = FALSE]))
expr_df$sample <- rownames(expr_df)

df <- merge(
  expr_df,
  coldata,
  by = "sample",
  all.x = TRUE
)
X <- df[, genes] # select only the gene expression columns for clustering
d_samples <- dist(X, method = "euclidean") # calculate Euclidean distance between all samples
hc_samples <- hclust(d_samples, method = "complete") # cluster using complete-linkage
plot(hc_samples, hang=-1) # this actually calls plot.hclust, hang=-1 makes sure the tree sta
```

This should show a tree (upside down?!) with two very distinct branches of samples. The height value on the y-axis is actually the distance between clusters. We can check this by looking at the content of the distance matrix `d_samples`. The distance between samples 11 and 12 is a bit larger than 12 and that is also where the horizontal line is that merges both clusters. The root of the tree is slightly below 14, which agrees with the largest distance between any pair of samples (try `max(d_samples)`) which is 13.6.

To get the actual clusters, we can use `cutree()` to cut the tree at a certain height. Alternatively you can specify the number of clusters you want and let `cutree()` determine the appropriate cut position.

Exercise 9.3. Choose a height cutoff that should result in two clusters (this is usually not as easy) and check the result.

```
h = ##

sample_clusters <- cutree(hc_samples, h = h)
sample_clusters
```

The resulting clusters are numbered starting at 1. We can add these cluster numbers to the data frame and show them in the figure to check that the clustering worked. environmental_stress is now shown as the shape of the sample.

```
df$cluster <- factor(sample_clusters)
p2 <- ggplot(
  df,
  aes(
    x = .data[[genes[1]]],
    y = .data[[genes[2]]],
    color = cluster,
    shape = environmental_stress
  )
) +
  geom_point(size = 4) +
  geom_text(aes(label = sample), vjust = -0.8, size = 3) +
  labs(
    title = "hierarchical clustering in 2-gene expression space",
    subtitle = paste("h =", h),
    x = paste(genes[1], "(log(cpm) expression)"),
    y = paste(genes[2], "(log(cpm) expression)")
  ) +
  theme_bw()

p2
```

Does it look okay?

What if we include all genes in the distance calculation?

```
d_samples <- dist(t(logcounts), method = "euclidean")
hc_samples <- hclust(d_samples, method = "complete")

plot(hc_samples)
```

Picking a proper height to cut this tree into four clusters (one for each stress) is difficult, but the `cutree()` function can do that for you. Instead of specifying the `h` argument, you can specify `k` for the number of clusters.

```
sample_clusters <- cutree(hc_samples, k = 4)
```

In the plot it is hard to see how this worked, but we can just look whether the clusters match the stress, by combining the cluster information with the sample annotation into one dataframe like this:

```
cluster_df <- data.frame(
  sample = names(sample_clusters),
  cluster = factor(sample_clusters)
)

cluster_annot <- merge(
  cluster_df,
  coldata,
  by.x = "sample",
  by.y = 0,
  all.x = TRUE
)

print(cluster_annot)
```

Not quite there yet? Especially one of the two stresses does not seem to make enough of a difference to clearly tell the samples apart.

We can vary a number of choices in the different steps to try and cluster the samples according to their stress treatment, but you should be careful not to take this too far.

"If you torture the data long enough, it will confess to anything"

You could also conclude that there is little effect of one of these stresses, or even that samples could have been swapped during the experiment. Still, we can check how different choices for the distance measure, the number of genes, or the clustering agglomeration method affect the clustering.

Exercise 9.4. Try out different distance measures and agglomeration/linkage methods (see `?dist` and `?hclust`).

One distance measure that is not included in `dist()` is correlation. But you can easily do that using the `cor()` function and then calculate the distance by subtracting the correlation from 1:

```
cor_samples <- cor(logcounts, method = "pearson")
d_samples <- as.dist(1 - cor_samples)
```

Do you find a combination of parameter choices that gives the expected clustering?

9.3 *K*-means

Next we will look at the other common clustering method called *K*-means clustering. *K*-means clustering works by dropping a fixed number of points at random positions in your gene space. These points become the centers of the clusters, and hence are called centroids. So the number of centroids determines that number of clusters you get. The procedure is iterative, it repeats a number of steps until a certain stop condition is reached. These steps are: - assign all samples to their nearest centroid, using Euclidean distance

- move the centroids to the center of the cluster, by averaging over all points that are assigned to it
- repeat until the centroids no longer change position (or run a fixed number of steps).

Again, for simplicity we can start with two genes, I selected a couple that better separate the samples. With four groups of samples, I suggest we try setting k to 4.

K-means clustering starts by randomly positioning centroids in the gene space, which means that it is not-deterministic and might produce different clusterings when you run it multiple times. To still make it reproducible, we can use a trick involving a so called Random seed that makes sure we get the same random choices every we run the code. This works because random numbers in R (and generally in computers) are not really random. An algorithm called the pseudorandom number generator generates sequences of numbers that appear to be random, but if we fix the starting point, we always get the same sequence of 'random' numbers. Normally the starting point is set using dynamic information, like the current time in seconds.

```
set.seed(421) # set the starting point for the random number generator

genes = c("AT4G27670","AT1G56600")
expr_df <- as.data.frame(t(logcounts[genes, , drop = FALSE]))
expr_df$sample <- rownames(expr_df)

df <- merge(
  expr_df,
  coldata,
```

```
  by = "sample",
  all.x = TRUE
)

k <- 4
X <- df[, genes] # select only the gene expression columns for clustering

km <- kmeans(
  X,
  centers = k,
)

# add the clusters to the data frame
df$cluster <- factor(km$cluster)
```

And like before we can plot samples in the 2-gene expression space showing the clusters

```
p2 <- ggplot(
  df,
  aes(
    x = .data[[genes[1]]],
    y = .data[[genes[2]]],
    color = cluster,
    shape = environmental_stress
  )
) +
  geom_point(size = 4) +
  geom_text(aes(label = sample), vjust = -0.8, size = 3) +
  labs(
    title = "K-means clustering in 2-gene expression space",
    subtitle = paste("k =", k),
    x = paste(genes[1], "(log(cpm) expression)"),
    y = paste(genes[2], "(log(cpm) expression)")
  ) +
  theme_bw()

p2
```

How does it look?

We can print the cluster assignment in a table with the cluster numbers as column headers and the sample count per cluster as value

```
print(table(km$cluster))
```

Given the randomness of the initialization of the centroid locations, this might not be the best we can get.

Exercise 9.5. Write a for-loop to repeat the *K*-means clustering 10 times and report the number of samples per cluster for each repetition

```
X <- df[, genes]


km <- kmeans(
  X,
  centers = k,
)
print(table(km$cluster))
```

Does it get better?

Exercise 9.6. Ideally we would want to run *K*-means many times and keep only the best result. As it happens, `kmeans()` has a built-in argument for this, called `nstart`. Try setting that to 25 and check the result by plotting the samples in 2-gene space like above with their cluster annotation

Choosing *K* is important for *K*-means clustering. We used the number of treatments, but we can also let the data inform us of the optimal *K*. For this we use the total sum of the the squared distance of every sample to the center of its cluster, the total Within Sum of Squares or total WSS. Total WSS will be largest when *K* is 1 and smallest (zero) if we set *K* to the number of samples. If we plot a trend line of the total WSS (an elbow plot), we can look for a bend that indicates a point after which adding more clusters does not improve the total WSS as much anymore.

```
set.seed(42)
X <- t(logcounts) # now we take all genes into account. With the transpose t() we would clust
wss <- numeric(10) # vector to keep track of the total Within Sum of Squares values per K
for (k in 1:10) {
  km <- kmeans(X, centers = k, nstart = 25)
  wss[k] <- km$tot.withinss # the kmeans object already has the total WSS
}

plot(1:10, wss,
     type = "b", # a line plot with points
     pch = 19,
     xlab = "Number of clusters (k)",
     ylab = "Total within-cluster sum of squares",
     main = "Elbow Plot")
```

9.4 Clustering genes

Next to clustering the samples, we can also cluster the genes based on their expression patterns. If genes have similar expression, they may have related functions. We will use a different data set for this, a seed germination time series from *Arabidopsis thaliana* from this paper: "Extensive transcriptomic and epigenomic remodelling occurs during Arabidopsis thaliana germination."

We can again load it from the Expression Atlas like before

```
se <- load_atlas_se("E-GEOD-94457",from_disk=FALSE)

# store it for later use
saveHDF5SummarizedExperiment(se,dir="E-GEOD-94457",replace=TRUE)

coldata = as.data.frame(colData(se)[,c("growth_condition","time")])
logcounts <- get_normalized_counts(se)
```

And now we take the top 1000 most variable genes:

```
n_top <- 1000

gene_var <- rowVars(logcounts)
top_genes <- names(sort(gene_var, decreasing = TRUE))[1:n_top]
mat_top <- as.matrix(logcounts[top_genes,])  # genes x samples (numeric matrix)

# distance between genes
cor_mat <- cor(t(mat_top), method = "pearson")
d_genes <- as.dist(1 - cor_mat)

# hierarchical clustering
hc_genes <- hclust(d_genes, method = "complete")
```

We could plot the tree, but with 1,000 genes this will not be very helpful. We can instead show the clusters in a heatmap to see the corresponding expression patterns. The `pheatmap()` function that we used before can already do the clustering for you, but we will provide the clustering results that we obtained above.

```
library(pheatmap)

pheatmap( mat_top,
          cluster_rows = hc_genes,
          cluster_cols = FALSE,
          show_rownames = FALSE,
```

```
          show_colnames = FALSE,
          annotation_col = coldata,
          scale = "row",
          main = paste("Top",n_top,"most variable genes") )
```

This shows a few clear patterns of genes with similar expression. Some of the genes are tuned down during germination while others are activated in the middle or at the end. We could group the genes into three clusters: early, middle and late.

```
gene_clusters = cutree(hc_genes, k=3)

for (cluster in 1:3) {
  pheatmap( mat_top[gene_clusters == cluster,],
            cluster_rows = FALSE,
            cluster_cols = FALSE,
            show_rownames = FALSE,
            show_colnames = FALSE,
            annotation_col = coldata,
            scale = "row",
            main = paste("Cluster",cluster) )
}
```

As a bonus (so not part of the exam) we can investigate which biological processes these genes are involved in to dive into the biology. For this a common approach is Gene Ontology enrichment analysis, which basically comes down to finding biological progresses that occur more frequently in the set of genes in a cluster than would expected by chance. The code below does that for all three clusters. The dotplots show the top 20 most enriched biological processes per cluster.

```
if (!requireNamespace("org.At.tair.db", quietly = TRUE))
  BiocManager::install("org.At.tair.db")

if (!requireNamespace("clusterProfiler", quietly = TRUE))
  BiocManager::install("clusterProfiler")

library(clusterProfiler) # for the enrichment analysis
library(org.At.tair.db) # contains gene function annotation

background_genes <- rownames(logcounts) # the background set of genes

for (cluster in 1:3) {
  genes <- names(gene_clusters)[gene_clusters == cluster] # the list of genes in the cluster

  ego <- enrichGO(
```

```
    gene          = genes,
    universe      = background_genes,
    OrgDb         = org.At.tair.db,
    keyType       = "TAIR",
    ont           = "BP",
    pAdjustMethod = "BH",
    pvalueCutoff  = 0.05,
    qvalueCutoff  = 0.05,
    readable      = TRUE
  )

  print(dotplot(ego, showCategory = 20, title = paste("Cluster",cluster)))
}
```

One cluster is clearly composed of seed expressed genes, another more focused on cell wall organisation, and one clearly enriched in photosynthesis related genes.

10 Principle Component Analysis and Differential Gene Expression

10.1 Dimensionality reduction

We looked at clustering the samples based on the expression of 2 genes, which worked pretty good when this was based on the two most variable genes. With two genes, we could also plot the samples in a 2 dimensional figure and 'see' the clusters. However, if we take all genes into account, this will not work anymore: with three genes we reach the limit of the number of dimensions we can plot and that third dimension is already less easy to interpret than the first two. Still, in exploratory data analysis, being able to 'look' at your data can be very informative.

A common way to address this is to perform dimensionality reduction, where the data are transformed into a lower dimensional space, while trying to preserve the main characteristics of the original data. Plotting the samples using the two most variable genes could be considered dimensionality reduction (through feature selection). Dimensionality reduction methods can be separated into linear and non-linear approaches. In this session we will look at a specific method for linear dimensionality reduction called **principle component analysis** or PCA, which works quite well for log-transformed bulk RNA-seq data. If the structure in de data becomes more complex, for instance in single-cell RNA-seq data which can consist of dozens of different cell types that each has its own unique expression profile, non-linear methods like t-SNE and UMAP are popular alternatives. It is good to realize that flattening the data points to only two dimensions will unavoidably lead to a simplification of the overall structure, which may lead to wrong conclusions.

10.2 Principle Component Analysis

To understand the algorithm for doing PCA involves leaning about matrix calculations like eigenvectors and eigenvalues, which is beyond the scope of this course, and fortunately this is not essential for developing a basic understanding of PCA and an intuition to interpret the results.

In this session we continue with the heat and salt stress gene expression data set from https://www.ebi.ac.uk/gxa/experiments/E-GEOD-72806/

First load the proper libraries and filter/transform the data like before

```
library(HDF5Array)
library(edgeR)
library(SummarizedExperiment)

source("./rnaseq_functions.R")
se <- load_atlas_se("E-GEOD-72806")
logcounts <- get_normalized_counts(se)
coldata <- as.data.frame(colData(se)[, c("environmental_stress"), drop = FALSE])
```

Let's plot the samples based on the expression three genes in three dimensions. For that we will make use of the plotly library that offers a plethora of interactive graphs. Here we will use the `scatter3d` plot. First install and load plotly:

```
if(!requireNamespace("plotly", quietly = TRUE))
    install.packages("plotly")
library(plotly)
```

We extend the gene list with a third, and create a data frame with the counts and sample annotation.

```
genes3 <- c("AT4G27670", "AT1G56600", "AT4G09020")
logcounts_genes3 <- t(logcounts[genes3, , drop = FALSE]) # transpose to have samples as rows
expr_df <- as.data.frame(logcounts_genes3)
expr_df$sample <- rownames(expr_df)

# add sample annotation
expr_df$environmental_stress <- coldata[expr_df$sample, "environmental_stress"]

p3 <- plot_ly(
  expr_df,
  x = ~ .data[[genes3[1]]],
  y = ~ .data[[genes3[2]]],
  z = ~ .data[[genes3[3]]],
  color = ~ environmental_stress,
  colors = "Set2",
  type = "scatter3d",
  mode = "markers",
  marker = list(size = 6)
) %>%
  layout(
```

```
    title = "Samples in 3-gene expression space",
    scene = list(
      xaxis = list(title = genes3[1]),
      yaxis = list(title = genes3[2]),
      zaxis = list(title = genes3[3])
    )
  )

p3
```

Drag/rotate the gene space in such a way that the samples are best spread out in the 2 dimensions that you effectively see. This is what PCA tries to accomplish. To see how that works in on this data, we first use the `prcomp()` function:

```
pca <- prcomp(logcounts_genes3) # the PCA should be performed on the counts
summary(pca)
```

In the summary you see the three principle components (PCs) of the data and the proportion of the total variance they explain. The first PC (PC1) already explains most of the variance, with a little left for PC2. We can visualize this as a barplot. This type of plot is called a screeplot.

```
screeplot(pca)
```

To see how the samples separate in the space defined by the first two PCs, we can make a so called biplot.

```
biplot(pca)
```

This is a bit hard to make sense of, so let us make a more pretty plot including the sample annotations. As before we can use `ggplot()`, we need to store the data in a data frame.

```
pca_df <- as.data.frame(pca$x) # ggplot needs a data.frame
pca_df$sample <- rownames(pca$x) # create a sample column to merge on
coldata$sample <- rownames(coldata) # create a sample column to merge on
pca_df <- merge(pca_df, # add sample annotation
                coldata,
                by = "sample",
                all.x = TRUE)

var_expl <- summary(pca)$importance[2, 1:2] * 100 # variance explained by PC1 and PC2

p <- ggplot(pca_df, aes(PC1, PC2, color = environmental_stress)) +
  geom_point(size = 4) +
  labs(title = "PCA of Arabidopsis heat/salt stress RNA-seq",
```

```
       x = paste0("PC1 (", round(var_expl[1], 1), "%)"),
       y = paste0("PC2 (", round(var_expl[2], 1), "%)")) +
  theme_bw()
p
```

To see how the 3D gene space was actually rotated, we can look at the loadings of the genes on the PCs. These indicate how much each gene contributes to each PC. The loadings are stored in the **rotation** component of the PCA object.

```
# loadings: variables (genes) x PCs
loadings <- as.data.frame(pca$rotation[, 1:2])
loadings
```

These loadings per PC represent a so-called unit vector (a line in the gene-space of length one) originating from the center (0,0), projecting the original gene axes. PC1 is strongly aligned with the first gene, while PC2 is largely determined by gene 2. We can add these vectors to the plot,

```
loadings$gene <- rownames(loadings)

p + geom_segment( # p still contains the PCA plot from above
    data = loadings,
    aes(
      x = 0, y = 0,
      xend = PC1, yend = PC2
    ),
    arrow = arrow(length = unit(0.25, "cm")),
    color = "red",
    inherit.aes = FALSE
  ) +
    # loading labels
  geom_text(
    data = loadings,
    aes(
      x = PC1,
      y = PC2,
      label = gene
    ),
    color = "red",
    size = 4,
    vjust = 0,
    hjust = -0.2,
    inherit.aes = FALSE
  )
```

Looking at the direction of the AT4G27670 arrow, it seems this gene contributes equally to PC1 and PC2?

Exercise 10.1. Check this with the loadings for AT4G27670 for PC1 and PC2. What is misleading in the axes of this plot?

Based on these three genes, the PCs separate the samples quite well. However, the results can be quite different if we change the way the data are preprocessed. For instance, if we do not log-transform the data, or if we center and/or scale the data before performing PCA, this can have a big impact on the results. The prcomp() function has options for centering and scaling the data, if you do not set them, the defaults are taken.

Exercise 10.2. To see the effect of centering and scaling, complete the `pca_plot()` function that performs PCA and plots the samples in PC1 and PC2 for different combinations of centering and scaling. Then plot it using the `patchwork` library to arrange the four plots in a 2x2 grid.

Add the `pca_plot()` function to your **rnaseq_functions.R** script

```
pca_plot <- function(logcounts, coldata, center, scale) {
  # add code here to construct pca_df for the given center and scale
  p <- ggplot(pca_df, aes(PC1, PC2, color = environmental_stress)) +
    geom_point(size = 4) +
    labs(title = paste0("Center:", center, " Scale:", scale),
         x = paste0("PC1 (", round(var_expl[1], 1), "%)"),
         y = paste0("PC2 (", round(var_expl[2], 1), "%)")) +
    theme_bw()

  return(p)
}

# Build the 4 plots for the different combinations of centering and scaling, to be plotted i

pFF <- make_pca_plot(logcounts_genes3, coldata, center = F, scale = F)
pTF <- make_pca_plot(logcounts_genes3, coldata, center = T, scale = F)
pFT <- make_pca_plot(logcounts_genes3, coldata, center = F, scale = T)
pTT <- make_pca_plot(logcounts_genes3, coldata, center = T, scale = T)

# Arrange in a 2x2 grid using the patchwork library
if (!requireNamespace("patchwork", quietly = TRUE))
  install.packages("patchwork")
library(patchwork)
(pFF | pFT) / (pTF | pTT) # patchwork magic
```

If you see 4 different plots, great! If the plots are all the same, check the code of your function and make sure you are actually using the center and scale arguments in the `prcomp()` function. You should see that centering has a big impact on the results, while scaling does not have much effect. This is because the data are already on the same scale (log-transformed counts). PCA analyzes variance around the mean of the data. The principal component axes always pass through the origin (0,0,...). By centering the data, we ensure that the origin corresponds to the mean of the data.

Which combination of center/scale gives the best separation of the samples? In general, it is good practice to center the data before performing PCA, while scaling is not always necessary, especially if the data are already on a similar scale. How does this match the defaults of `prcomp()`?

Next, we can see what happens if we base the PCA on all genes (so not just on log-counts_genes3).

```
pca <- prcomp(t(logcounts))
summary(pca)
pca_df <- as.data.frame(pca$x)
pca_df$sample <- rownames(pca$x)
pca_df <- merge(pca_df,coldata,
                by = "sample",
                all.x = TRUE) # add sample annotation

var_expl <- summary(pca)$importance[2, 1:2] * 100 # variance explained by PC1 and PC2
```

This gives us more PCs, but the first one still explain most of the variance.

To see how the samples separate now, plot them again using PC1 and PC2 of this PCA.

What do you conclude about the separation of the samples based on all genes compared to the three genes we selected? Do you think this is a better representation of the data? Why or why not?

We can again look at the loadings of the genes for the PCs, but since we included all genes in the analyses and have many PCs we cannot plot all of them. We can focus on the genes that contribute most to PC1, which are the ones with the highest absolute loadings.

```
pc = "PC1"
loadings <- pca$rotation[, pc]
top50 <- names(sort(abs(loadings), decreasing = TRUE))[1:50]
logcounts_top50 <- logcounts[top50,]
annotation_col <- coldata[,"environmental_stress",drop=FALSE]

library(pheatmap)
```

```
pheatmap( logcounts_top50,
          annotation_col = annotation_col,
          show_rownames=TRUE,
          show_colnames=FALSE,
          scale="none",
          main=paste0("Top 50 genes by absolute ",pc," loading" ))
```

We can use the PCs also for clustering the samples. This gives us the opportunity to select the most important aspects of the data. Starting with the top 10 PCs to do the *K*-means clustering gives a pretty good result.

```
pca_scores <- pca$x[, 1:10]
set.seed(42)
km_pca <- kmeans(pca_scores, centers = 4, nstart=250)

pca_df$cluster <- factor(km_pca$cluster[match(pca_df$sample, rownames(pca_scores))])

ggplot(pca_df, aes(PC1, PC2, color = environmental_stress, shape = cluster)) +
  geom_point(size = 4) +
  labs(
    title = "PCA of Arabidopsis heat/salt stress RNA-seq",
    x = paste0("PC1 (", round(var_expl[1], 1), "%)"),
    y = paste0("PC2 (", round(var_expl[2], 1), "%)"),
    shape = "k-means cluster",
    color = "Stress"
  ) +
  theme_bw()
```

Exercise 10.3. Try out what happens if you try different numbers of top PCs than 10, does the clustering still match the treatments?

10.3 Differential Gene Expression

Up to now we have been doing true exploratory data analysis (EDA), looking at the data without a strong hypothesis. This is what is typically called unsupervised analysis. We only used the treatment information to interpret the results, not to guide the analysis. Now we will do a supervised analysis, where we take the sample information into account to do the analysis. The question we will address is: **For which genes does the expression significantly change in response to the stress treatments?**. As already mentioned on Monday, these genes are called Differentially Expressed Genes or DEG, and the analysis

is called Differential Gene Expression or DGE. For this we group samples based on their treatment. For instance, all three samples that were only treated with heat are replicates of each other where we expect the same genes to be responding to the treatment. You may have noticed the term **significantly**, which implies we do a statistical test and get a *p*-value. The t-test is not appropriate for RNA-seq data, because these are discrete counts instead of continuous values. RNA-seq counts for a gene are generally assumed to follow a negative binomial distribution. Libraries like edgeR are specifically developed to use the count information from the replicates to predict the mean and variance for the expression of a gene and use these in a statistical test comparing different treatments or time points. Below the code for this analysis, not need to understand every step.

```
counts <- assay(se) # we start with the raw counts
group <- factor(coldata$environmental_stress)
levels(group) <- make.names(levels(group))
group <- relevel(group, ref = "none") # compare all treatments to the untreated samples

dge <- DGEList(counts = counts, group = group) # create the DGEList object

keep <- filterByExpr(dge, group = group) # using the edgeR count filtering
dge <- dge[keep, , keep.lib.sizes = FALSE] # remove genes based on the keep logical vector
dge <- calcNormFactors(dge) # for normalization
design <- model.matrix(~ 0 + group) # design of the experiement
colnames(design) <- levels(group)
fit <- glmQLFit(dge, design) # # fit model

heat_contrast <- makeContrasts( # extract results for heat treatment
  heat_stress = `heat.stress` - none,
  levels = design
)

qlf_heat <- glmQLFTest(fit, contrast = heat_contrast) # perform the statistical tests
res_heat <- topTags(qlf_heat, n = Inf)$table

salt_contrast <- makeContrasts( # extract results for salt treatment
  salt_stress = `salt.stress` - none,
  levels = design
)

qlf_salt <- glmQLFTest(fit, contrast = salt_contrast) # perform the statistical tests
res_salt <- topTags(qlf_salt, n = Inf)$table

salt_heat_contrast <- makeContrasts( # extract results for salt + heat treatment
  heat_stress = `salt.and.heat.stress` - none,
```

```
  levels = design
)

qlf_salt_heat <- glmQLFTest(fit, contrast = salt_heat_contrast)
res_salt_heat <- topTags(qlf_salt_heat, n = Inf)$table

res_list <- list( # combine in one list
  heat = res_heat,
  salt = res_salt,
  salt_heat = res_salt_heat
)

sig_counts <- sapply(res_list, function(x) sum(x$FDR < 0.05))
sig_counts # significant counts per stress
```

This shows the number of genes that have significantly different expression comparing samples that did not experience stress with samples that did. The counts include both genes that have higher expression (upregulated) as well as genes that have lower expression (downregulated) due to the stress treatment. We can zoom into the upregulated genes and check the overlap between the different stress treatments using a Venn diagram.

```
if(!requireNamespace("VennDiagram", quietly = TRUE))
  BiocManager::install(c("VennDiagram"))
library(VennDiagram)
library(grid)

# upregulated genes
deg_heat <- rownames(res_heat)[res_heat$FDR < 0.05 & res_heat$logFC >= 0]
deg_salt <- rownames(res_salt)[res_salt$FDR < 0.05 & res_salt$logFC >= 0]
deg_salt_heat <- rownames(res_salt_heat)[res_salt_heat$FDR < 0.05 & res_salt_heat$logFC >= 0

draw.triple.venn(
  area1 = length(deg_heat),
  area2 = length(deg_salt),
  area3 = length(deg_salt_heat),
  n12 = length(intersect(deg_heat, deg_salt)),
  n13 = length(intersect(deg_heat, deg_salt_heat)),
  n23 = length(intersect(deg_salt, deg_salt_heat)),
  n123 = length(Reduce(intersect, list(deg_heat, deg_salt, deg_salt_heat))),
  category = c("Heat", "Salt", "Salt+Heat")
)
```

Exercise 10.4. Look at figure 3A in the Suzuki et al paper.

How do the Venn diagrams from the paper compare to this one?

What would be a good next comparison to make?

Part V

IV: Going furthR

The goals for week 4

In this final week, we will consolidate a lot of what was learnt in the first three weeks, with the emphasis on increasing your confidence in writing your own R code to solve data-related issues.

The learning objectives of week 4 are:

1. Apply data science competences to write your own R code to solve data analytic tasks
2. Understand common sources of error in data, and apply approaches to deal with these errors
3. Understand the importance of colour choice for visualisations of different types of data
4. Apply data science competences to reproduce publication-quality data visualisations
5. Apply string operations for character string manipulation

How week 4 is organised

Similar to previous weeks, the first three days are intended to work through these objectives, while Thursday is kept for code peer-review and exam preparation, and Friday (morning) the exam.

The content of the first three days is as follows:

- **Monday**: Dealing with errors and outliers
- **Tuesday**: Exploring genetic and phenotypic diversity in *Arabidopsis thaliana*
- **Wednesday**: (if needed) Completion of the data analysis from Tuesday / String operations

The assignment

As for the previous weeks, there is an assignment due on Wednesday evening (23:59), which forms the basis of the peer-review exercise on Thursday.

The assignment instructions are given at the end of day 3 (Section 13.5), with an accompanying dataset that can be downloaded from Brightspace in the Week 4 Content Assignment section.

The exam

The exam of week 4 **counts towards your final grade for the course**.

At the end of week 4 we expect you to be able to:

- give a practical (“rule of thumb”) definition of what a data outlier is, and what to do with such data points if they arise
- apply different methods for identifying errors in data, and be able to correct / curate these before performing an analysis
- evaluate the advantages and disadvantages of different approaches to dealing with “big data” in R
- understand the importance of different colour palettes for different data types and visualisation purposes
- apply basic string operations such as string concatenation, string splitting, as well as be familiar with regular expressions for pattern matching and string manipulation.

11 Errors and outliers

11.1 Introduction

The focus of today will be on errors - data points that for one reason or another do not fit in our dataset. We often need to be able to identify errors so that we can remove (or correct) them before running an analysis. This is generally the case for types of analyses or calculations that assume data is error-free (there are more advanced approaches that can explicitly model the possibility of errors in the data e.g. Bayesian methods, but we will not be using them here). Errors could lead to false conclusions being drawn, or a loss of statistical power for testing differences (by inflating the estimates of the variance in the data for example). We will spend some time considering in particular what an "outlier" means, and what to do with them.

We will also be dealing with simulations, using a simulator to generate datasets with errors in them. You will have to don your detectives' cap to figure out the error rates in the data.

We will start with a warm-up exercise, detecting errors in data from a study on salt stress in tomato (*Solanum lycopersicum*).

11.1.1 Tomato salinity experiment

An experimenter investigated the response of three varieties of tomato (Moneymaker, Recordsmen, Rio Grande) to salt stress. Tomato plants were grown in pots with four different salinity levels (0, 50, 100, 150 mM NaCl), with three replicates per treatment.

- **Q. What is the experimental unit? How many experimental units are there?**

The experimenter measured the following traits after 60 days' growth:

- total fruit number
- total fruit weight (g)
- leaf necrosis

The last of these traits was measured on a scale of 0-5, with 0 for healthy leaves and 5 for dead leaves. Per plant, the average necrosis of the top 5 leaves was calculated and recorded.

The dataset that resulted is available to download from Brightspace under Content / Week 4 / Datasets and R scripts for exercises / Errors and outliers / tomato_salinity.csv.

Download this file and save it to your working directory. Let's read in the file and check its dimension and structure using `dim()` and `str`.

```
data <- read.csv("tomato_salinity.csv")

dim(data)
str(data)
```

Exercise 11.1 (Detecting tomato outliers)**.**

- There are four obvious (typing) errors in the dataset. Can you identify them? Can you think of what the values (probably) *should* be?

11.2 Outliers

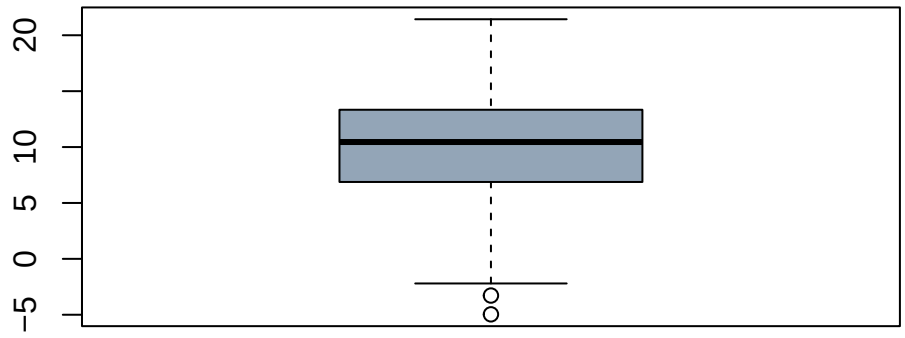
There is no strict definition of an outlier, although there are some rules of thumb that are commonly used. For example, Tukey proposed that an outlier is a value that falls beyond 1.5 times the inter-quartile range. If the data is normally-distributed, then values that are more than 3 standard deviations from the mean are very unlikely to occur by chance and are often considered as outliers (the "3σ rule").

- **Question: With *scaled* data, what would the 3σ rule imply?**

11.2.1 Boxplots

You have already encountered boxplots earlier in the course, here we return to them in the context of identifying outliers. A typical boxplot looks something like the following:

```
set.seed(42) #for reproducibility
v <- rnorm(100,10,5)
boxplot(v, col = rgb(0.4,0.5,0.6,0.7))
```



- **Q. What does the boxplot show? In particular, how are the whiskers defined, and how are the points that are shown identified? Are these outliers?**

If you are unsure, have a look at the `boxplot()` documentation for the details:

```
?boxplot
```

If you need some reminders on what terms like "median" and "quantile" mean, you can also check back to Section 4.2.

We also mentioned the "3σ rule", do any of the values fall outside this range? How could you check this? Try this yourself before looking at the **> Code**!

```
range(scale(v))
```

- **Q. Which outlier criterion is stricter: Tukey's or the 3σ rule?**

We next look at at toy dataset of car fuel consumption, available via Brightspace: Week 4 / Datasets and R scripts for exercises / Errors and outliers / car_mpg.RData

Load the data using `load()`:

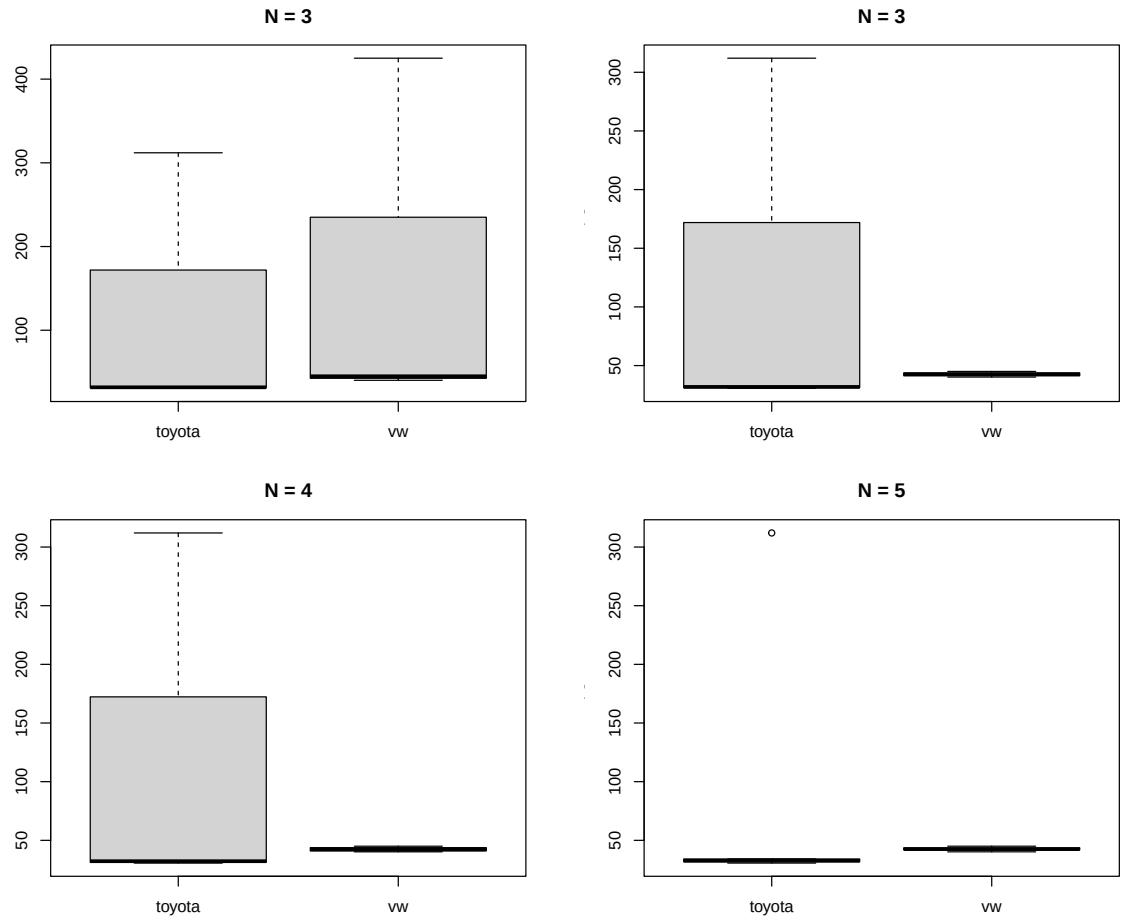
```
print(load("car_mpg.RData"))
```

```
[1] "car_mpg"
```

- **Q. Why did I wrap the `load()` call with `print()`?**

Using the `boxplot()` function, produce boxplots of each of the 4 list elements of the `car_mpg` data. If you are unsure, use the code hidden below.

```
par(mar = c(3,3,3,3), mfrow = c(2,2))
boxplot(mpg ~ car, data = car_mpg$data1,
        main = "N = 3")
boxplot(mpg ~ car, data = car_mpg$data2,
        main = "N = 3")
boxplot(mpg ~ car, data = car_mpg$data3,
        main = "N = 4")
boxplot(mpg ~ car, data = car_mpg$data4,
        main = "N = 5")
```



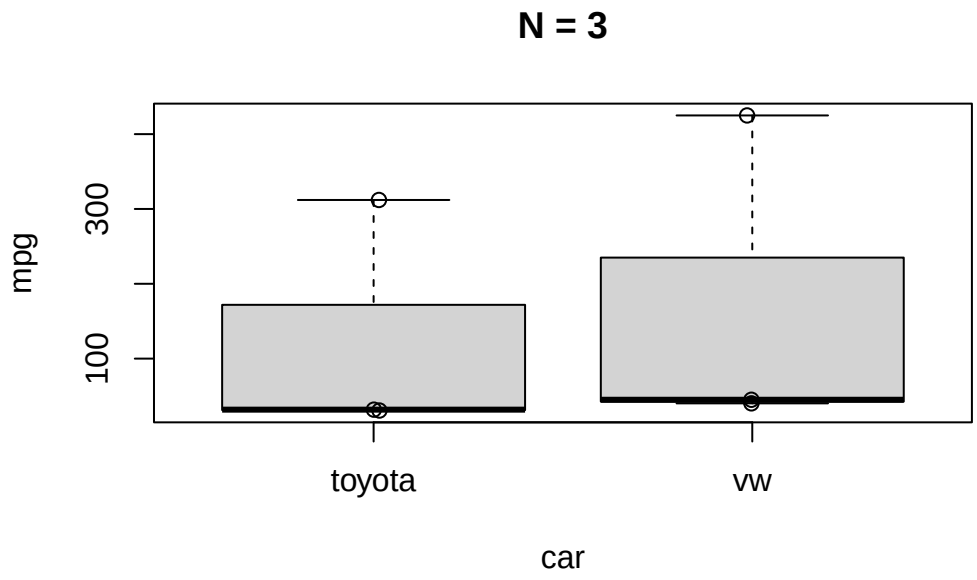
In these boxplots a main title was added with the number of observations underlying each box-and-whiskers plot. What do you notice about the data? Why are the bottom

two plots so different?

Perhaps this is labouring the point here, but it is always useful to know what you are looking at with such visualisations!

Even in the top left-hand plot (N = 3), it should have been apparent that something was wrong when the median line was at the very bottom of the plot. Boxplots are often now accompanied by the (jittered) underlying datapoints, which would have highlighted the issue better:

```
boxplot(mpg ~ car, data = car_mpg$data1,
        main = "N = 3")
points(jitter(as.numeric(factor(car_mpg$data1$car)),
              amount = 0.02),
       car_mpg$data1$mpg)
```



11.2.1.1 Final note

If you wanted to extract the "outliers" identified in a boxplot (data points beyond the whiskers), you can do this by first capturing the output of the function (it is silently returned by the function, probably using `invisible()`) and extracting relevant parts of the output:

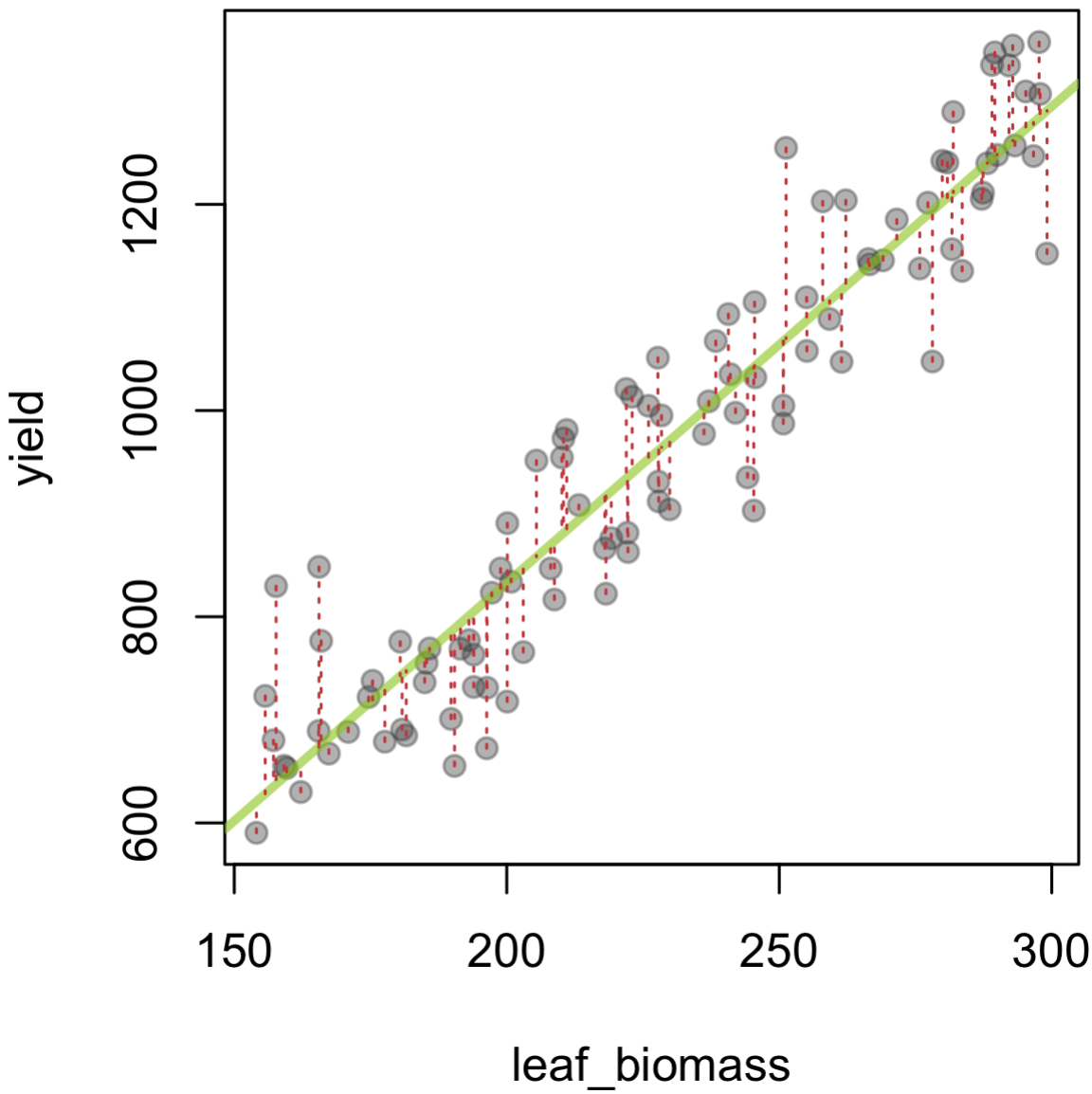
```
bp <- boxplot(mpg ~ car, data = car_mpg$data4,
              plot = FALSE) #we don't want the plot again

setNames(bp$out,bp$names[bp$group]) #read the boxplot documentation!
```

```
toyota
   312
```

> **Extra final note - `setNames()`**
>
> `setNames()` is a useful little function to know, it sets the names of a vector:
>
> ```
> vect <- setNames(1:5,letters[1:5])
> vect
> ```
>
> ```
> a b c d e
> 1 2 3 4 5
> ```

11.2.2 Residuals

As a last topic in this section, we will briefly consider how residuals can also tell us something about data quality and the potential presence of outliers. As you may recall, a residual is what is left over after fitting a model, which is often visualised for the case of a simple linear regression as follows:



In this figure, leaf biomass is plotted against yield, and a fitted trendline (fitted using linear regression) has been added in green. The *predicted* yield given a leaf biomass value is the value the line takes at this point. So we can see the residual as the difference between the observed yield values, and those predicted by the model (in this case, a single line). These quantities are represented above as dashed red lines. Large deviations from the green line (= larger residuals) suggest that the particular observation in question is not in line (literally) with the trend from the rest of the data. This could be by chance, or due to a problematic data-point.

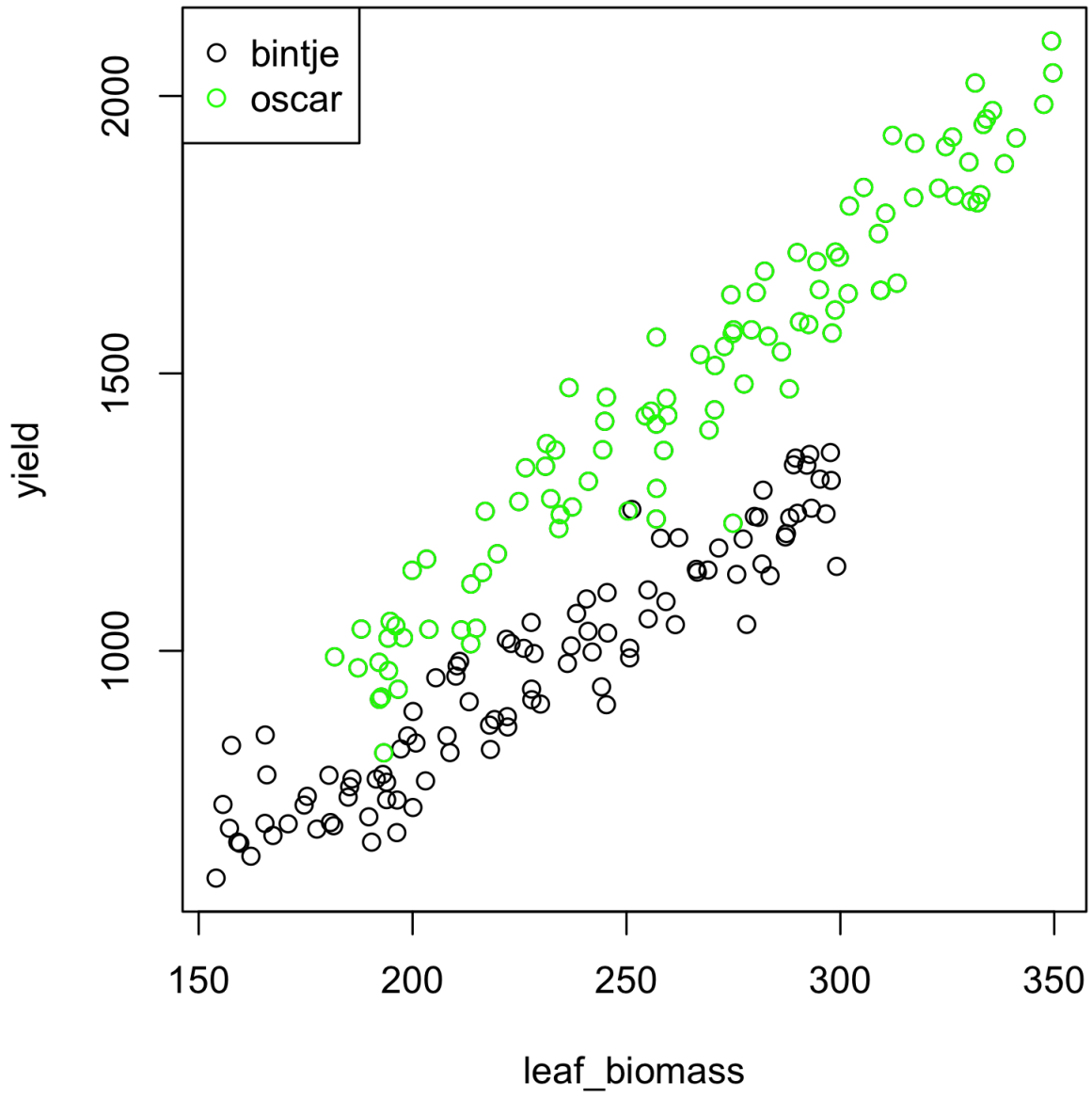
When fitting an ordinary linear model, one assumes certain things about the residuals (often shortened as i.i.d. - that they are independent of each other and identically distributed, namely that they are drawn from a $N(0, \sigma_e^2)$ distribution - a Normal distribution with mean 0 and a shared variance). It is good practice to check the residuals (we often do this using residual plots) after running a regression or ANOVA model, to make sure the residuals are distributed as expected. These checks can also highlight potentially problematic data-points (with high residuals, or perhaps with high leverage i.e. having an unduly large influence on the model fit).

We will use a dataset of leaf biomass and yield collected from a field trial in potato (*Solanum tuberosum*) to demonstrate the point.

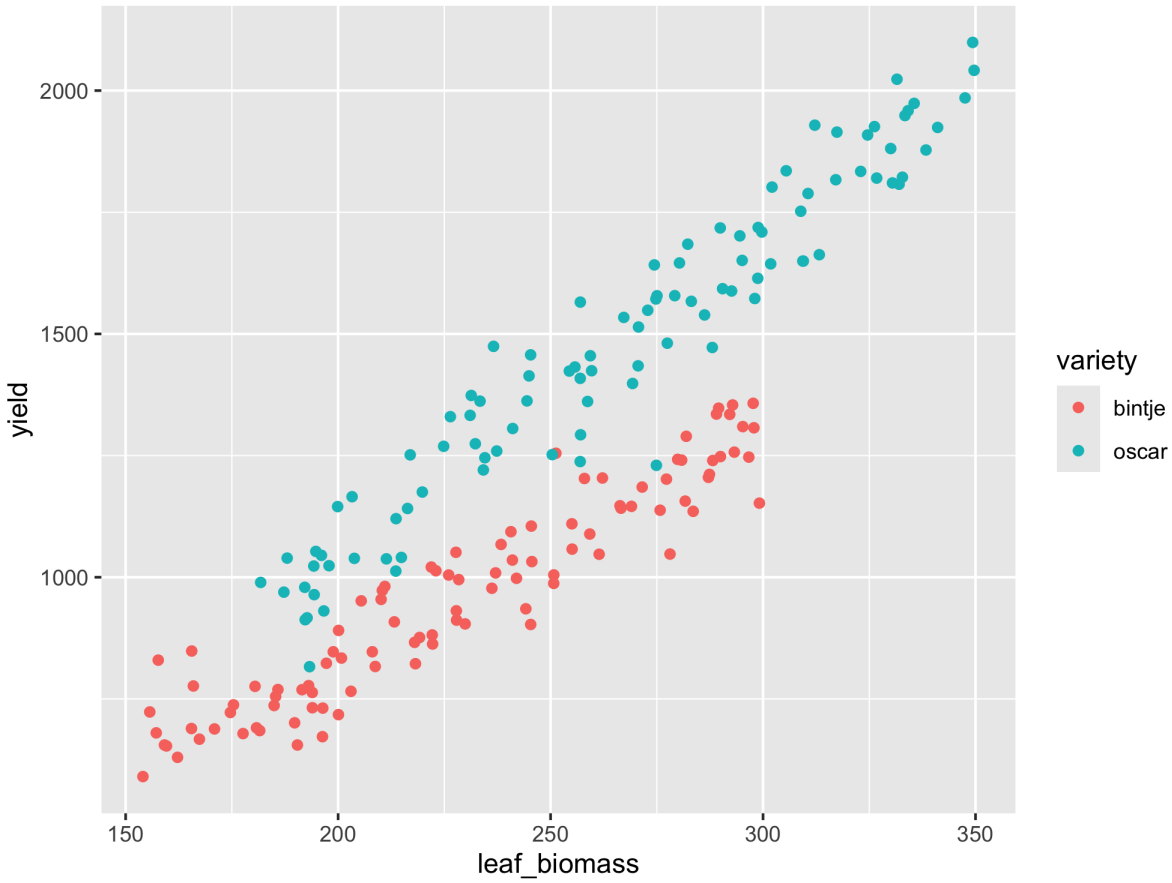
Exercise 11.2 (Potato yield (part 1))**.**

- Load the tab-delimited 'potato_yield.dat' dataset provided on Brightspace using the `read.table()` function.
- Check the **str**ucture of the dataset using `str()`. If there is an experimental factor that is coded as a character string, convert it to a factor (e.g. using `factor()` or `as.factor()`).
- Generate 2 boxplots, showing yield and leaf biomass, split across potato variety. Is there any evidence of outliers?
- Create a scatter plot of the leaf biomass (x axis) versus yield (y axis) using `plot()`. Can you spot any strange data points / potential outliers?
- Make the same plot, but now colour the points by potato variety. Can you spot any strange data points / potential outliers?

For the last of these exercises, you could either produce the plot using the base R `plot()` function, giving something like:



... or using `ggplot2::ggplot()`:



If you were unable to complete the steps in the previous exercise after a reasonable attempt, please check Brightspace week 4 for a script file. If it is not yet visible, please raise your hand.

From the above scatter plots, it is not immediately obvious that there are any outlying data-points present. There also seems to be a clear linear relationship between leaf biomass and yield, so this data would appear to be suitable for linear regression. We can already observe that variety 'Oscar' produces more yield for the same leaf biomass as 'Bintje'. We will fit a model with separate lines as we are not yet sure whether separate slopes or separate intercepts (or both) are needed. We can check this after fitting the model.

Our main interest for today is to check whether there are any outliers after model fitting, i.e. data points that lead to high residuals.

To run a linear model in R we will use the `lm()` function, followed by a visual check on the residuals.

Finally, we will check whether any of the Studentised residuals[1] exceeds 3 (another rule of thumb for identifying potential outliers among the residuals).

The steps to do this in R are as follows:

```
set.seed(1221)

## Make a toy dataset with 2 factors a and b:
test_data <- data.frame(x = rep(1:10,2),
                        n = c(rep("a",10),rep("b",10)),
                        y = c(1:10 + rnorm(10,sd = 3),
                              seq(1,30,3) + rnorm(10,mean = 4,sd = 3))
)

test_data$n <- as.factor(test_data$n) #good habit to convert experimental factors to data ty

## Run a linear model, allowing for separate fitted lines (allow interaction of n and x):
lm1 <- lm(y ~ n*x, data = test_data)

summary(lm1)
```

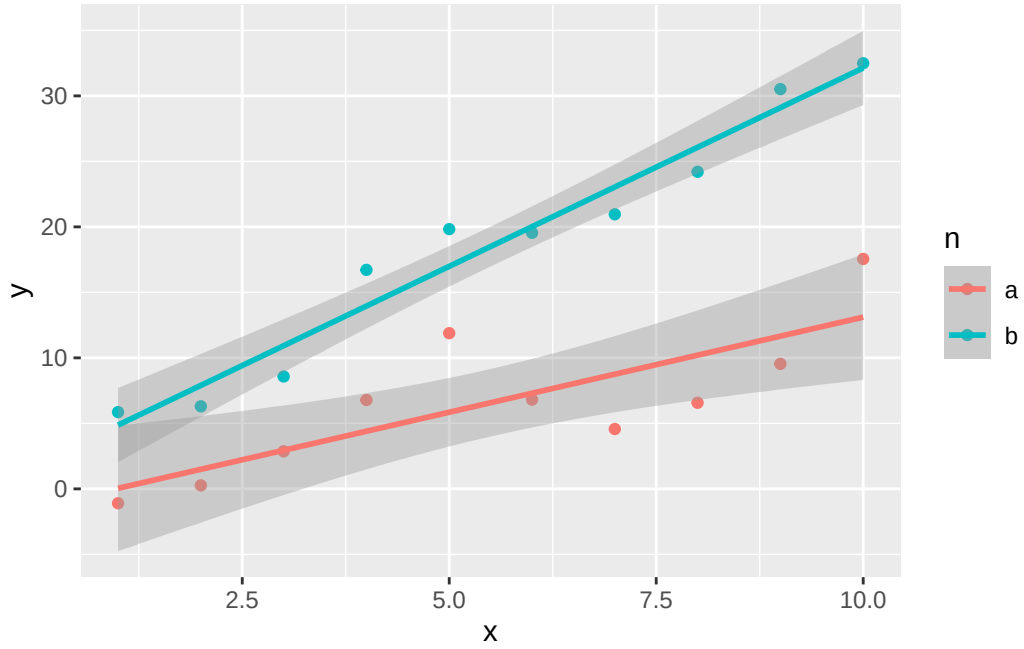
```
Call:
lm(formula = y ~ n * x, data = test_data)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1848 -1.9225 -0.4774  1.6564  6.0345

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -1.4113     1.9845  -0.711 0.487215
nb            3.2516     2.8065   1.159 0.263630
x             1.4520     0.3198   4.540 0.000335 ***
nb:x          1.5770     0.4523   3.486 0.003049 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.905 on 16 degrees of freedom
Multiple R-squared:  0.924, Adjusted R-squared:  0.9098
```

[1] Note that I have deliberately written `[[` and not `[[()` when referring to the function `[[` here. This goes a bit against convention: elsewhere in these notes functions are written like `plot()` to explicitly refer to the function `plot`.

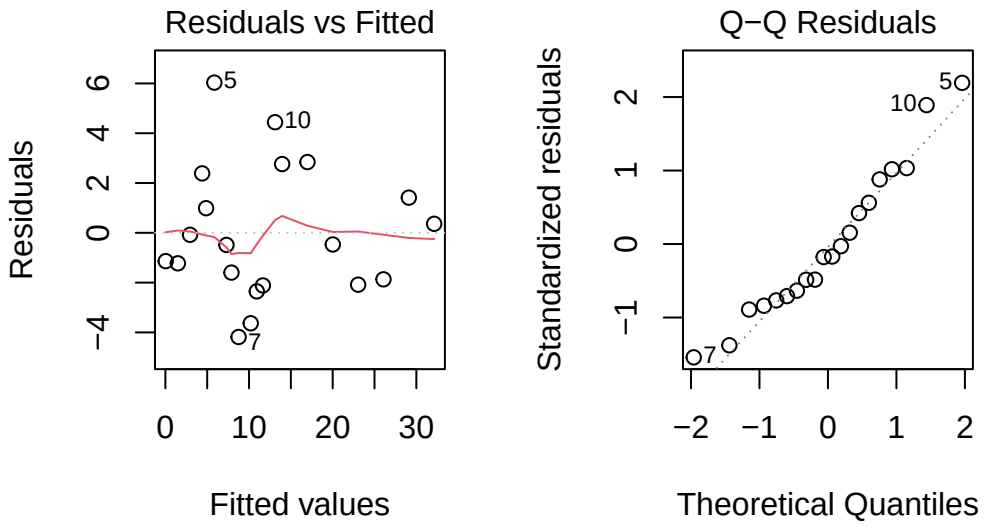
```
F-statistic: 64.85 on 3 and 16 DF,  p-value: 3.584e-09
```

```
## ggplot2 offers useful functionality for plotting the regression lines:
library(ggplot2)
ggplot(data = test_data,
       aes(x = x, y = y, color = n)) +
  geom_point() +
  geom_smooth(method = "lm", se = TRUE)
```

```
`geom_smooth()` using formula = 'y ~ x'
```




```
## Check the residual plots:
par(mfrow = c(1,2))
plot(lm1, which = c(1,2))
```



Exercise 11.3 (Potato yield (part 2))**.**

- Using a similar approach to the steps outlined above, run a linear model using the `lm()` function for the potato yield dataset, with `yield` as the response variable, allowing for separate fitted lines per variety.
- Check the residual plots. Are the model assumptions satisfied? Is there any evidence of high residuals?
- Apply the rule of thumb on the Studentised residuals (*Hint*: use the `rstudent()` function). Are any of the data points potential outliers using this criterion? Which one(s)?

11.3 Detecting errors in a simulated dataset

We are now going to move on to a "fun activity", namely generating a simulated dataset with errors in it for somebody else to puzzle over.

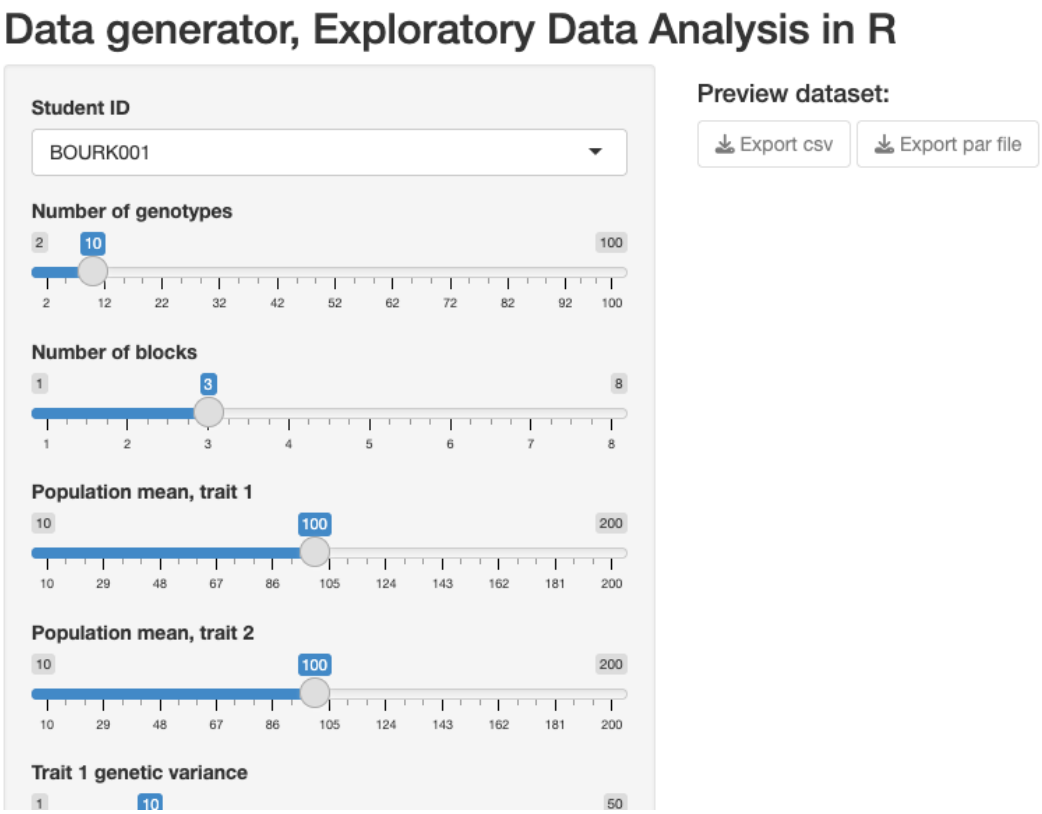
The simulator is hosted on the shiny.wur.nl server.

The simulator uses Shiny, an R package for developing web applications with R (or Python) running under the hood. The simulation tool itself is quite basic: it allows the user to generate a dataset with user-defined numbers of genotypes & experimental blocks,

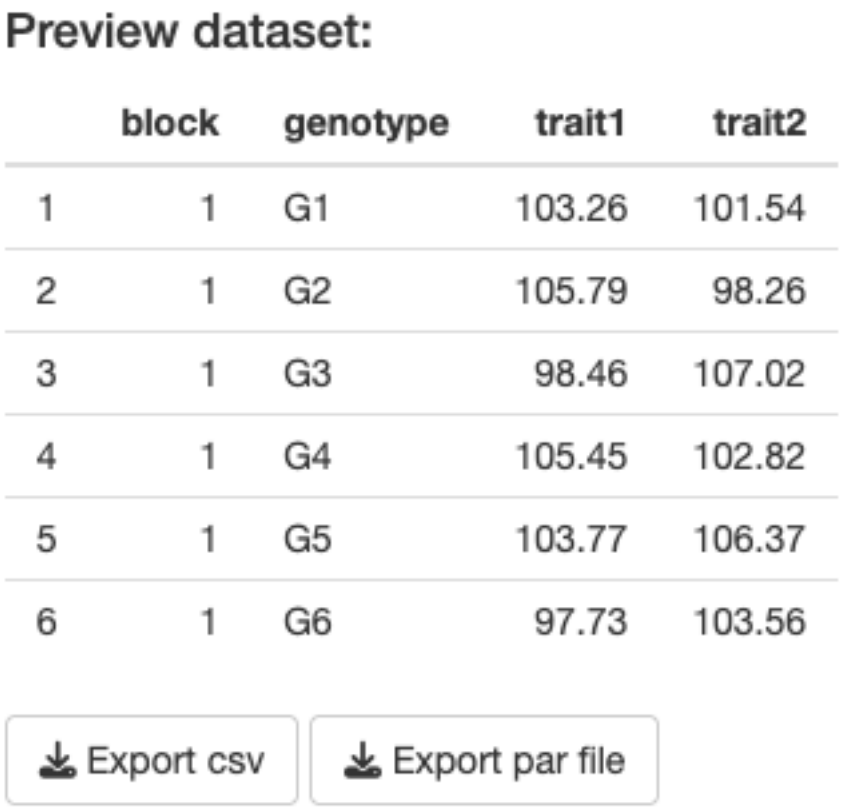
various genetic, environmental and residual variances, for two measured traits (with the exciting names of 'trait1' and 'trait2'). Finally, you can introduce outliers (generated according to the 3σ rule) and missing values.

11.3.1 Running the simulation

The interface is in standard shiny layout, with a side panel for input and a main panel for output.

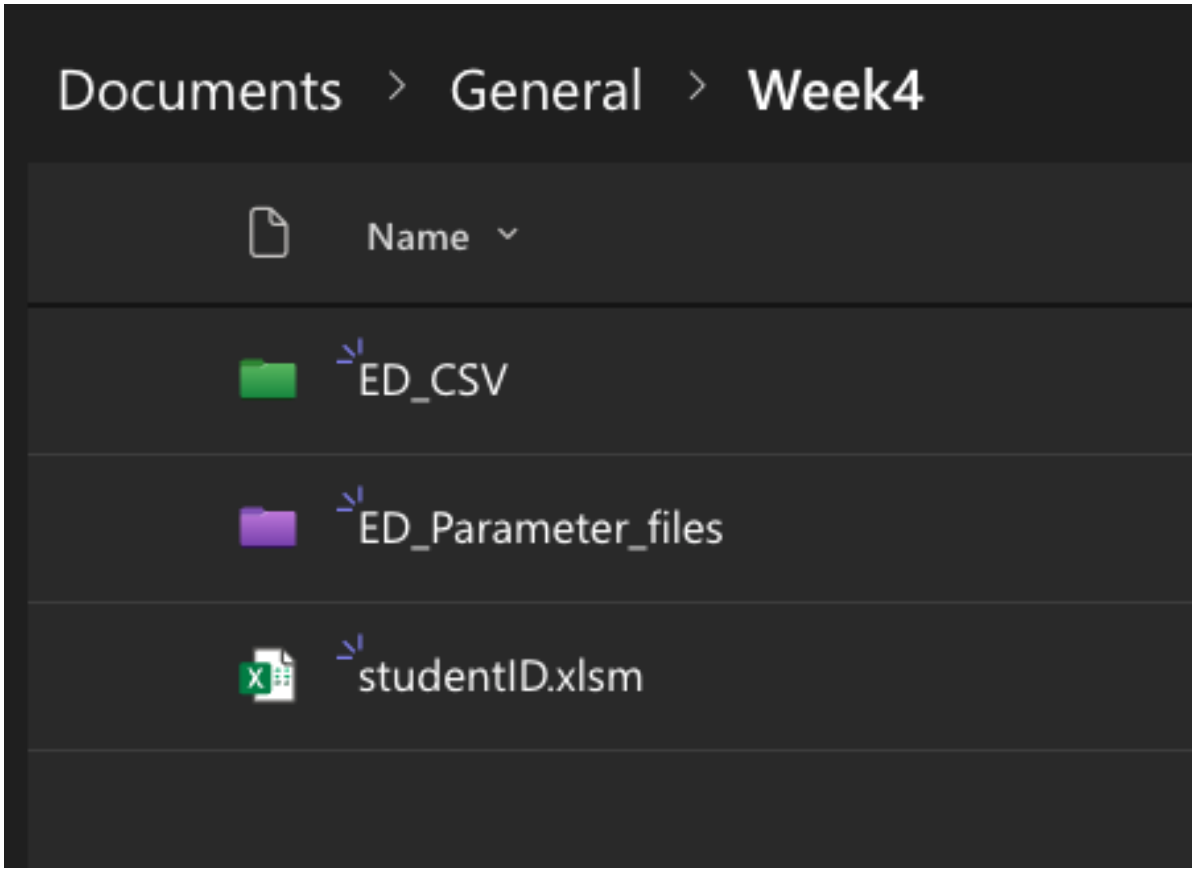


- **Step 1**: Select your student ID in the drop-down menu
- **Step 2**: Choose some interesting parameter settings using the input sliders.
- **Step 3**: Click on the "Run Simulation" button.
- **Step 4**: Export your data (both the CSV which has the simulated dataset, and a parameter file in .RDS format which contains a list of the "true" simulation settings that you chose). You need to click *both* download (Export) buttons shown:



If all has gone well, there should be two data files present in your downloads folder.

These need to be uploaded on MS Teams to the appropriate folder (one for the CSV datafile and one for the parameter file):



As you may have noticed, the CSV filename has your student ID embedded - this is to help the detective find the correct file (a dataset is assigned based on student ID - this will be done in class). Student ID is then followed by a four-character code (a "filetag"), for example 'AG27'. Note that the accompanying parameter file has a different filename, which is an encrypted version of the same four-character code. This is to avoid possible cheating (otherwise it would be very simple and perhaps tempting to look up the 'true' simulation parameters and correct your answers before submitting).

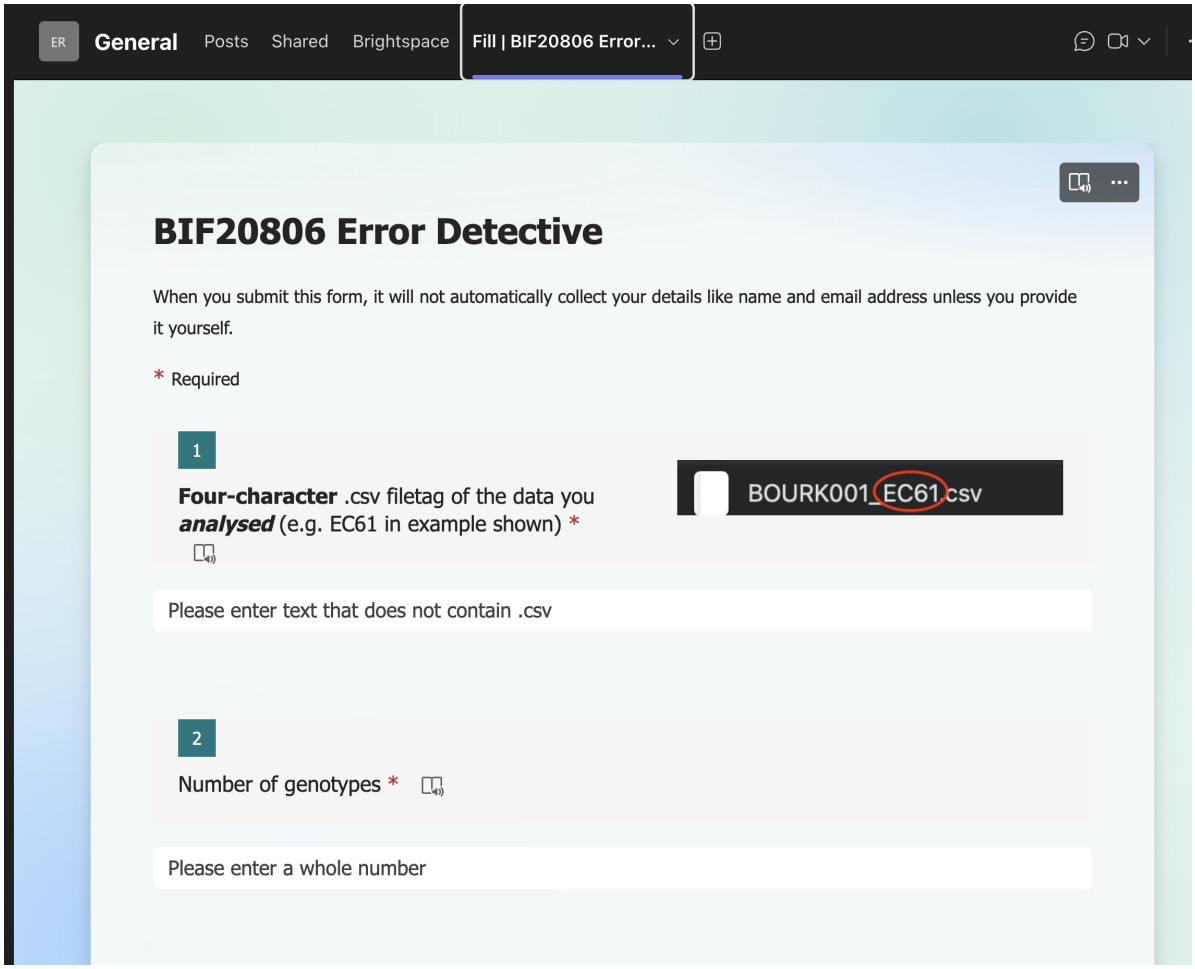
11.3.2 Analysing the data (Error Detective)

In class you will be randomly assigned another student's simulated dataset to work with.

It is now time to become an Error Detective!

Download the CSV file assigned to you (you will need to look for this file via the studentID - make sure you analyse the correct file, and **not your own one**!).

On MS Teams there is a form that needs to be filled in (only visible during week 4 of the course):



There are in total 10 questions to fill in:

Question	Entry
1	Four_character_filetag
2	N_genotypes
3	N_blocks
4	mean_trait1
5	mean_trait2
6	rho
7	outlier_rate_trait1
8	outlier_rate_trait2
9	NA_rate_trait1

Question	Entry
10	NA_rate_trait2

The first of these is simply the four-character (filetag) of the file you analysed. Make sure you type this correctly! Typos here will mean your data will most likely be filtered out and not used in the plenary analysis.

The number of genotypes and blocks should be fairly straightforward to work out (have you already used the function `unique()`? If not, check it out!), as well as the mean trait values for trait1 and trait2 (the function `mean()` might help!).

11.3.3 Genetic correlation, ρ

The next item is more tricky - "rho", the (Pearson) correlation co-efficient between the traits. Whoever generated the data you are now analysing selected a value for ρ between -1 and 1, but in the simulation app, this was actually the *genetic* correlation between traits, not the correlation between the phenotypes directly.

In many instances we are more interested in the genetic rather than the phenotypic correlation, as phenotypes are often affected by other effects, e.g. block effects, other environmental effects, experimental treatments etc. The usual way to estimate the genetic correlation is to fit a multivariate model with an explicit variance-covariance structure. If genome-wide marker information is available, this can also be used to get a better estimation of the genetic correlation, taking into account different levels of genetic relatedness between individuals.

Here we will keep it very simple - we will estimate ρ by calculating the correlation between the estimated genetic effects themselves (the "predicted values" from the fitted model of the genotypes). Once you have done that, check for yourself how ρ compares to the correlation between the raw phenotypes (i.e. `cor(data$trait1,data$trait2)`). The correlation should be broadly similar but probably not identical (unless the block variance was 0 and there was no noise in the data!).

You are not expected to be able to write a function to do this yourself - you may use the `estimate_rho()` function given here:

```
estimate_rho <- function(data){
  ## Start-up checks -- -- -- -- -- -- -- -- --
  if(!"data.frame" %in% class(data)){
    warning("input data is not a data frame. Trying to convert..")
    data <- as.data.frame(data) #try to coerce to a data frame
  }
```

```
  if(!all(c("block","genotype","trait1","trait2") %in% colnames(data))){
    stop("Expected colnames 'block', 'genotype', 'trait1' and 'trait2'.\nPlease check the inp
  }
  ## End of start-up checks -- -- -- -- -- -- -- -- --

  ## Convert factors to factors:
  data$block <- as.factor(data$block) #this is important - block is not numeric!
  data$genotype <- as.factor(data$genotype) #this is just for good habit forming

  ## Fit linear model (no interaction):
  lm1 <- lm(trait1 ~ block + genotype, data = data)
  lm2 <- lm(trait2 ~ block + genotype, data = data)

  ## Extract the fitted coefficients (intercept and slopes)
  coef1 <- lm1$coefficients
  coef2 <- lm2$coefficients

  ## Append 0 for the first genotype; the rest are contrasts:
  fixef1 <- c(0,coef1[grep("genotype",names(coef1))])
  fixef2 <- c(0,coef2[grep("genotype",names(coef2))])

  return(cor(fixef1,fixef2))
}
```

Note that in the exam you will be expected to be able to write a simple function, including the correct use of input arguments in round brackets, function body in curly brackets, `return` at the end etc.

The last four items for the Error Detective are also relatively straightforward - for example you can identify `NA` values with the `is.na()` function. But you may want to give some thought to how to identify outliers. Should you use a boxplot, the 3σ rule, or via the residuals? Maybe try all these approaches and make an informed decision before submitting the form. When all students have submitted their Detective forms, we will collectively analyse the results, to see how well (or poorly) you were able to describe the attributes of the datasets. Afterwards, you can delve more into the group data to see whether you can identify factors that affected prediction accuracy.

Exercise 11.4 (Final exercise)**.** We have put the focus here on identifying errors in the dataset. As an additional exercise, see whether any of your data parameter estimates change if you "clean up" the dataset before estimating them.

12 Genotypic data, colour schemes and GWAS

12.1 Introduction

This chapter follows a data-driven approach: we will be working with a large genotypic dataset (SNP markers) with accompanying meta-data, and a linked phenotypic dataset of flowering time in *A. thaliana*. You will gain some experience with data subsetting, data curation and visualisation, with some focus on the choice of colour schemes. Familiar techniques like PCA will be used (already covered in week 3), followed by some possibly less familiar techniques (genome-wide association study). As we will be using data from a previously-published study, part of today's exercises will try to replicate some of the figures from the publication itself.

12.2 Install R packages (dependencies)

We will be using a number of packages that should ideally be installed at this stage so that any installation issues can be handled together.

For reading HDF5 files, there are some packages available via CRAN, but the `rhdf5` package from Bioconductor seems to work well (we first install the `BiocManager` package to handle the `rhdf5` installation).

Note that in all of the following calls, we first check whether the package has already been installed on our computer (`!requireNamespace()` returns `TRUE`or `FALSE` depending on whether the package is already installed or not).

```
## To install packages from BioConductor, first install BiocManager:
if(!requireNamespace("BiocManager", quietly = TRUE)) install.packages("BiocManager")

## Install rhdf5 package from Bioconductor:
if(!requireNamespace("rhdf5", quietly = TRUE)) BiocManager::install("rhdf5")

## The rest of the packages are available via CRAN:
if(!requireNamespace("RColorBrewer", quietly = TRUE)) install.packages("RColorBrewer")
if(!requireNamespace("mapview", quietly = TRUE)) install.packages("mapview")
```

```r
if(!requireNamespace("ggplot2", quietly = TRUE)) install.packages("ggplot2")
if(!requireNamespace("statgenGWAS", quietly = TRUE)) install.packages("statgenGWAS")
```

12.3 Downloading Data

All the data that we will be using today is publicly available and free to download, and is connected to the 2016 Cell paper "1,135 genomes reveal the global pattern of polymorphism in *Arabidopsis thaliana*" [1]. The genotype data in particular is quite large (> 900 Mb), most of which we won't use. If storage space is an issue for you, you can also work with the subset(s) of the genotype files that we will be generating.

12.3.1 Genotypes

You can download the SNP genotype data from the 1001genomes.org website via the following link: https://1001genomes.org/data/GMI-MPI/releases/v3.1/SNP_matrix_imputed_hdf5/

The large tarball (1001_SNP_MATRIX.tar.gz) is the relevant directory to download. You will need to extract this compressed folder to access the 'imputed_snps_binary.hdf5' file that we will be using. These will be made available via Brightspace in the coming hour.

12.3.2 Accessions list

The list of accessions associated with the SNP data is on the same website:

https://1001genomes.org/accessions.html

Scroll down to the bottom of the '1135 Accessions Final Set' tab and click on "Download complete list (CSV)"

12.3.3 Phenotypes

The flowering time phenotypes for this *Arabidopsis* panel that we will be working with are available here:

https://arapheno.1001genomes.org/study/12/

Download both the FT10 and FT16 datasets; this can be done in one step using the download button above the table (and select the CSV option):

Phenotype Name	Trait Ontology (TO)	Environmental Ontoloy (EO)	Unit Ontology (UO)	# values
FT16	days to flowering trait	growth chamber study	day	1123
FT10	days to flowering trait	growth chamber study	day	1163

12.4 Data overview

As usual, it's convenient to make a new folder where all datasets has been saved, and set your working directory to that folder using `setwd()`.

- First, check out the structure of the SNP data using the `h5ls()` function:

```
rhdf5::h5ls("imputed_snps_binary.hdf5")
```

So there are 10,709,949 SNPs scored per accession, which is more than we can easily handle. We can read in the SNP positions and the accession names using the `h5read()` function:

```
accessions <- rhdf5::h5read("imputed_snps_binary.hdf5","accessions")
positions <- rhdf5::h5read("imputed_snps_binary.hdf5","positions")
```

12.5 Big data in R

R is not famed for its capacities to deal with big data. In R, "big" usually means taking up lots of working memory. For example, if we try to allocate a very large vector within R, at some point we will get an error message (on my computer this occurs for i = 10 in the following call):

```
for(i in 1:10){
  t <- system.time(temp <- rep("memorytest",10^i))
  message(paste0("A vector with 10^",i," elements takes ",round(t[3],1)," seconds to generate
  cat("_____________________________________________________________________\n")
}
```

In today's example, we will be looking at the effect of different subsetting / data simplification strategies on the results we obtain (mainly in terms of population genetics).

12.6 Subsetting the SNP data

To handle this "big-dataset" and generate results quickly, we are going to use a subset of the SNP data today, sampling 10k SNPs from the total set to work with.

For consistency between our results, let's set a random number generator seed e.g. 1234:

```
set.seed(1234)
```

This ensures that our results will be identical (i.e. the results generated here are reproducible).

Our first approach is to randomly `sample()` ten thousand SNPs.

Save this (pseudo-random) selection in an object called `snp_index`, using something like the following code (completed of course!):

```
nSNP <- 1e4 #target total number of SNPs.
snp_index <- NA #you need to replace this with R code that samples 1e4 SNPs!
```

We are going to read in these positions using the same function as before. Out of interest, try to time this step on your computer, by wrapping the code with the `system.time()` function:

```
system.time(SNP <- rhdf5::h5read("imputed_snps_binary.hdf5","snps",
                                 index = list(NULL, snp_index)))
```

- Give meaningful row and column names to `SNP` and save it as an .RDS file using the `saveRDS()` function.

We are going to repeat this procedure, but instead of randomly taking 10k separate SNP positions, we are going to sample 10K SNPs in **chunks** instead, with these chunks evenly spaced over the whole genome. We will subsequently be comparing the results between using the original random set of SNPs, or having the same number of SNPs, but sampled in non-random chunks.

Exercise 12.1 (Subsetting data)**.**

- Try to generate these alternative subsets yourself, while noting how much time it takes to read in 10K SNPs when they are read in chunks rather than singly.
- Also make a version of the random 10K SNPs with a **M**inor **A**llele **F**requency of at least 0.05 (minor allele frequency refers to the frequency of the less common allele at a locus, in the range from 0 to 1). *Hint*: check that the SNP allele frequency >= `0.05`. How many SNPs remain after filtering?

If you are unable to complete these steps yourself (after a reasonable attempt), you can check Brightspace for code to complete exercise 1. If it is not yet visible, please raise your hand.

12.7 Accession data

We have also downloaded information on the accessions in comma-separated format. Supposing you have saved them in a file called "metadata.txt", we can read in this data as follows:

```
meta <- read.table("metadata.txt", header = FALSE, sep = ",")
head(meta)
```

We need to provide column names, using information from the website (as these were not included in the downloadable data).

Three of these columns are ambiguous:

```
colnames(meta) <- c("ID","seq_by","name","country",
                    "name_abbrev","lat","long","collector",
                    "dunno1","CS_num","admixture_group","dunno2","dunno3")
```

- Check that the IDs listed here are in the same order as the accession numbers from the SNP data.

Hint: use == and wrap the result using the `all()` function.

We check what the 3 unknown columns are using `table()`, and save the meta data:

```
table(meta$dunno1)[1:10] #listing the first 10 only; collection dates presumably
table(meta$dunno2) # name this column Y, not sure what it stands for
table(meta$dunno3) # Perhaps something to do with the colours used for admixture groups?

table(meta$dunno3, meta$admixture_group)

## Fill in the missing 3 column names:
colnames(meta) <- c("ID","seq_by","name","country","name_abbrev",
                    "lat","long","collector","date","CS_num",
                    "admixture_group","Y","circle_col")
head(meta) #looks OK
saveRDS(meta, file = "meta.RDS")
```

Out of interest, do you recognise any of the collectors?

```
table(meta$collector)
```

Exercise 12.2 (Extra question)**.**

- Which accession carries the reference allele at all sites?

12.8 PCA

A Principal Component Analysis is a useful way to visualise variation in a dataset. When applied to genomic data like the SNP genotype matrix, it can give insights into the genetic similarity between individuals in the population, i.e. the structure of the population being studied.

Apart from running the PCA and interpreting the results, we would also like to check whether there are differences between our results depending on how we subsetted / filtered the data, as well as checking how the PCA results align with the admixture analysis performed by the authors of the 2016 Cell paper.

You should have four different genotype datasets:

- the original random 10K SNPs
- 10K SNPs in 200 chunks
- 10K SNPs in 1000 chunks

- the original random 10K that was filtered on MAF

We read them in here again so that we are working with standard object names (you may have saved the objects under different filenames; please adjust as necessary):

```
SNP <- readRDS("SNP.RDS")
SNP2 <- readRDS("SNP_200regions.RDS")
SNP3 <- readRDS("SNP_1000regions.RDS")
SNP_maf <- readRDS("SNP_maf.RDS")
```

If you were unable to produce these yourself (even with the previous script provided, e.g. you could not download the original data) then please check Brightspace for the files. Can't find them? Maybe they have not yet been made visible - if so, please raise your hand!

Do you expect that our method of subsetting 10k SNPs will have a large effect on the PCA results?

What about the filtering step for Minor Allele Frequency, do you expect that should have any effect?

Before running the analysis, think about these points and discuss with your neighbour(s).

We will be using the `prcomp()` function available in base R, which is relatively simple to use, but does not automatically scale the input data before running the PCA (i.e. the default is that `scale = FALSE`). In almost all cases it is advisable to scale (if you have a variable that is constant then scaling will be problematic (division by 0), but a constant variable doesn't tell us much anyway and should probably have been removed beforehand..)

Bring up the documentation using `?prcomp` for more details of the function:

```
?stats::prcomp
```

You can briefly read through this if interested, but first let's start with the random 10K set, using `prcomp()` to run the PCA:

```
fit <- prcomp(SNP, scale. = TRUE)
```

This may take some time to run, depending on how fast your computer is. You can use this time to further read the function documentation. If `prcomp()` is taking *inordinately* long (more than 2 minutes say), something might be wrong - please raise your hand!

Once we have run the PCA analysis, we would like to visualise the results.

Here is a small script to take the output of the `prcomp()` function and plot it:

```r
pca_plot <- function(fit, # output of prcomp()
                     x = 1, # which component to plot on the x axis
                     y = 2, # which component to plot on the y axis
                     screeplot = FALSE, # option to additionally plot a scree plot
                     addpoints = TRUE, # if FALSE, only the axes will be plotted
                     ...){ # additional arguments passed to plot() or points()

  if(screeplot){
    screecols <- rep("grey",10)
    screecols[c(x,y)] <- "green"
    plot(fit, col = screecols)
  }

  scores <- fit$x[,c(x, y)]

  ## Extract the explained variances:
  ve_x <- fit$sdev[x]^2/sum(fit$sdev^2)
  ve_y <- fit$sdev[y]^2/sum(fit$sdev^2)

  plot(NULL,
       xlim = c(min(scores[,1]), max(scores[,1])),
       ylim = c(min(scores[,2]), max(scores[,2])),
       xlab = paste0('PC', x, ' (', round(ve_x*100, 1), '%)'),
       ylab = paste0('PC', y, ' (', round(ve_y*100, 1), '%)'),
       ...)

  if(addpoints) points(x = scores[,1], y = scores[,2], ...)
}
```

Feel free to adapt or improve this function as you like.

> **i** Aside: Passing arguments using ...
>
> You may have noticed that we used an ellipsis (...) as an argument of the function `pca_plot()`. This allows us to pass extra arguments to `plot()` in our function call to `pca_plot()`. So for example, if we want to add the plot title 'PCA' we would normally add `main = "PCA"` as an extra argument to `plot()`. We can still do this in our new function.
> You may also have observed that the `...` was passed to the `points()` function as well. This is somewhat bad / sloppy programming practice, as we may not want to pass certain arguments to either `plot()` or `points()`. For example, `main` is not an argument of `points()`, but we have not told R where we wanted `main` to be passed

on to. But thankfully in this example, both `plot()` and `points()` are quite forgiving of being passed spurious (named) arguments, as they are closely-related functions. Other functions may not be so forgiving and will probably give warnings if you did this! If you want to test this, try running the following code line by line, and seeing what happens:

```
plot(1:10)
points(1:10, main = "test") #main is not an argument of points...
points(1:10, test = "test") #test is not an argument either!
```

If we wanted / needed to be more careful, we could extract all the extra arguments by checking against the named arguments of `pca_plot()`, storing these as a list, and then check to which internal function they should be passed (by checking names). In situations where two or more sub-functions both use the same argument names, we would then have to decide what we would like the function to do (pass to both sub-functions, or define sub-function specific ellipses..). But for our purposes here, we will ignore all these concerns.
One final note of interest - in `pca_plot()`, we have already specified some arguments, for example the x and y axis labels, and the x and y axis limits. If we try to override these, we will get an error. Try this for yourself, and see if you understand what the Error message means.

Exercise 12.3 (PCA plots)**.**

- Plot the output of the PCA analysis on the random 10k SNPs, giving an appropriate plot title.
- Make a second plot of PC1 versus PC3. Do you notice any outliers? Can you identify in which country this accession was collected?
- Repeat this exercise by also looking at PC1 versus PC4, and identifying the origin of the outlier.

These steps may take a little while to complete. A small R script to do this will be made visible on Brightspace (*after* you have tried to do this yourself first). If it is not yet visible, please raise your hand.

12.9 Choosing colour palettes

One important topic when working with data is how best to visualise the results of some analysis (or the raw data itself), particularly if this is to be shared with a broader audience. Colour is one way to denote differences in such visualisations (alternatives are symbol

shapes, symbol sizes, line types, line weights etc.). There are many resources available to choose appropriate colour sets or palettes, and we will be using a few of them today.

12.9.1 Data types

If you start searching for information on colour palettes and data visualisation, you will probably see that three basic types of data are generally recognised:

- **Sequential** - where ordering from low to high is possible
- **Diverging** - data with two extremes and a meaningful midpoint (like 0)
- **Qualitative** - i.e. categorical data

For each of these types of data, different colour palettes are available. You may already have some experience with these types of colour palettes if you have ever used 'Conditional Formatting' in Excel.

When choosing a colour palette, there are a number of important considerations:

- What is the data type?
- How many categories are present (for categorical data)?
- Is the choice of colours distinct and colour-blind friendly?

For the last of these points, quite some effort has been made to develop colours that can still be distinguished by people with various forms of colour blindness. At the start of this session, we already installed a colour package, namely `RColorBrewer`. This is one of the most popular colour palette generators, and has a nice website that can also be used for inspiration and exploration (colorbrewer2.org). To look at all the palettes available in the package, use the `display.brewer.all()` function:

```
par(mar = c(1,4,1,1)) #decrease the whitespace padding / margins
RColorBrewer::display.brewer.all()
```

12.9.2 Admixture results

Now that you have some experience with running a PCA, let's investigate how well the PCA results align with the admixture assignments from the original 2016 Cell analysis. We will be using `Set1` from the `RColorBrewer` package, which corresponds closely to the colour palette used in the 2016 Cell paper for the admixure groups.

Exercise 12.4 (Colour palette)**.**

- First, check how many admixture groups are in the data. Also check how many colours are provided by the RColorBrewer 'Set1' palette.
- Amend the 'Set1' palette as needed, to create a **named** colour vector with the *same* colours in the *same* order as was used in the Cell 2016 paper (e.g. Relicts should be red, Admixed should be black etc.)

To visualise the PCA results and colour by admixture group, we will be using a follow-on function that calls on `pca_plot()`:

```
admix_pca <- function(pcafit,
                      cols,
                      metadata = meta,
                      ... #arguments passed through to plot()
){
  pca_plot(fit = pcafit,
           addpoints = FALSE,
           ...)

  points(pcafit$x[,1], pcafit$x[,2],
         col = cols[metadata$admixture_group],
         pch = 19)

  ## position of the legend is hard-programmed here, not ideal..
  legend("topright", pch = 19,
         col = cols, legend = names(cols),
         cex = 0.7, bty = "n")
}
```

Exercise 12.5 (PCA plots)**.**

- Using the function we have just defined, produce a PCA plot from a PCA fit for each of the 4 genotype datasets. Use the colour palette that you developed, making sure the order is consistent with the publication. Can you identify (some of) the admixture groups easily in the PCA plot?

Are the PCA results consistent among themselves, i.e. across the four datasets? If not, why?

12.10 Visualising geo-reference data

One of the nice extra features of this dataset is the inclusion of latitute and longitude information for each of the accessions. We will be using the R package `mapview` to visualise the data, to see where the accessions were originally collected, and colour these by e.g. the admixture group or (out of interest) the sequence provided (who did the sequencing or genotyping work).

You should already have installed the `mapview` package; load the package and test it using a sample dataset provided with the package:

```
library(mapview)

mapView(breweries)
```

First, try to map the latitude and longitude columns from the meta data. We will use the argument `crs = 4326` to specify that the co-ordinates are using the standard geo-referencing system (an assumption!). We also turn off the grid using `grid = FALSE`. Note that the column names for longitude and latitude are `long` and `lat`:

```
mapView(x = meta,
        xcol = "long", ycol = "lat",
        crs = 4326, grid = FALSE)
```

Unfortunately, the `mapview::mapView` function is not able to handle missing values!

Exercise 12.6 (mapview plot)**.**

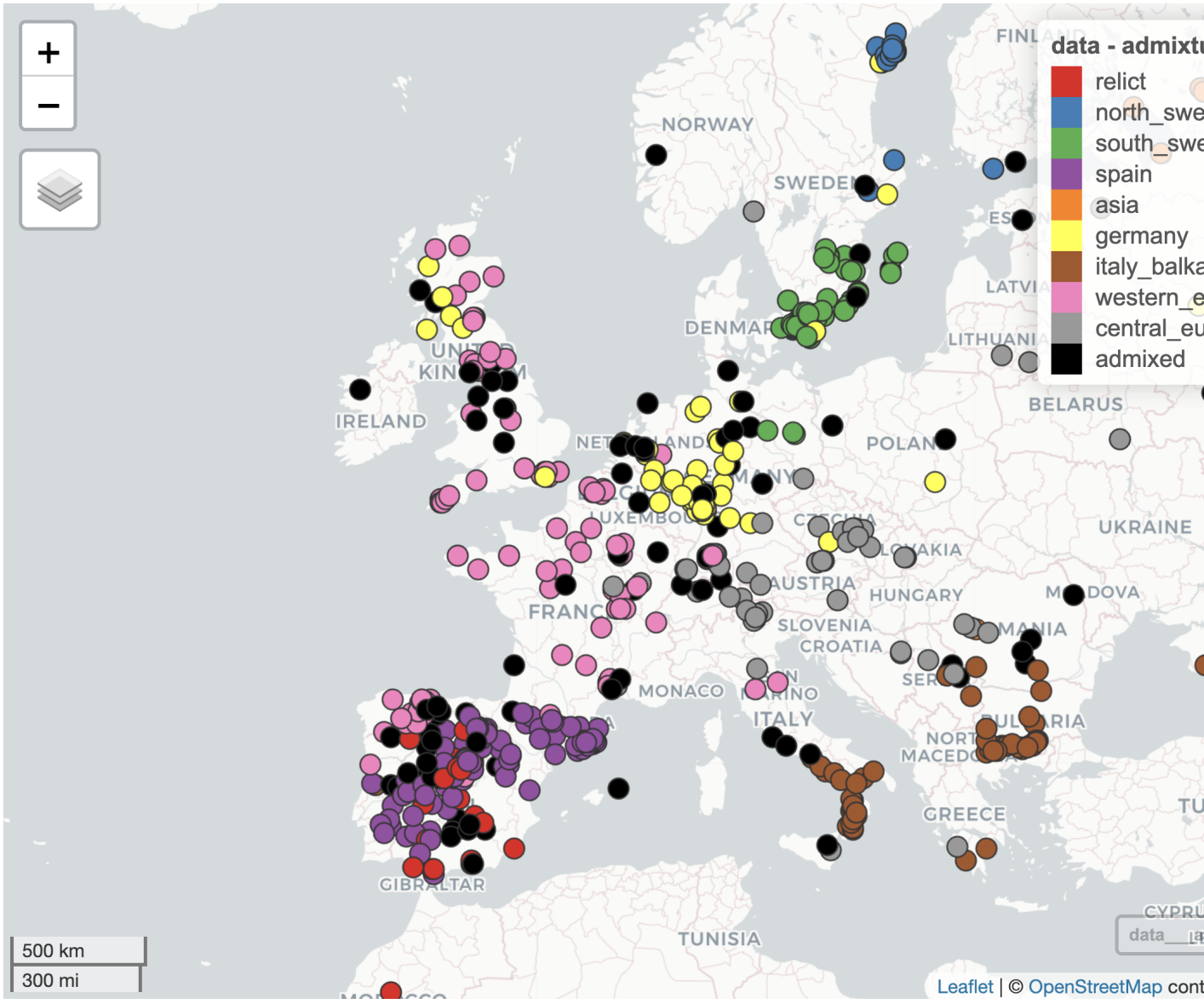
- Make a copy of the meta dataset without missing values for the GPS co-ordinates. Use the `mapview::mapView` function to visualise the accessions, using default colour settings (using function call above).
- If this is (still) not working, try to figure out what is going wrong. For example, Google your Error message to try to solve the issue.

As you can see, there may still be some unexpected issues that come up when trying to use this package (although these may also have been solved by a package update by the time you attempt this exercise). If you are still stuck here *after* a decent attempt at trying to solve the issue yourself, you can use the R code available via Brightspace. If it is not yet visible, please raise your hand.

Exercise 12.7 (mapview plot)**.**

- Re-plot the geographic distribution of accessions using the `mapview::mapView()` function, colouring the points by admixture group using the same colour palette as before (i.e. consistent with the publication). Adjust the settings so that the points are fully opaque.

Your resulting plot should look something like the following:



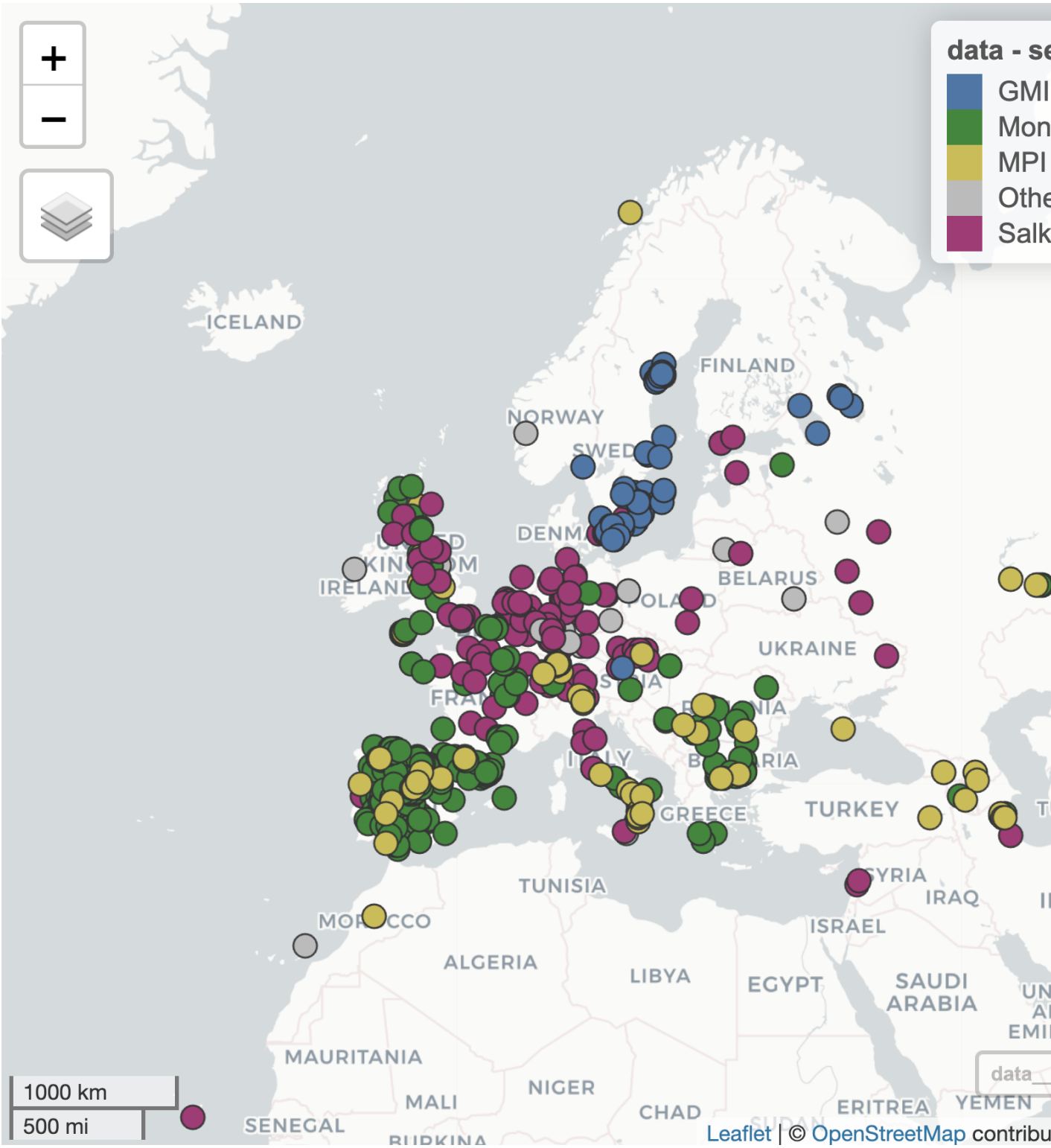
To get some more practice on data manipulation and plotting, we now turn to the sequence provider column. As you will see, there are many different sequence providers, but we are only interested in looking at the distribution of providers that genotyped **at least 100** accessions. Everything else can be included in an extra category "Other".

Regarding colours to use, you are asked to use the "Bright" qualitative palette developed by Paul Tol (sronpersonalpages.nl/~pault/), namely the colours **blue (4477AA)**, **green (228833)**, **yellow (CCBB44)**, **purple (AA3377)** and **grey (BBBBBB)**.

Exercise 12.8 (Sequence providers)**.**

- How many sequence providers appear more than 100 times in the dataset? What are their names?
- Create a new column in your dataset for sequence provider, in which all providers that appear less than 100 times are combined into a single category "Other"
- Using the (subsetted) 'Bright' palette in which the "Other" group is coloured grey, make a `mapview::mapView()` plot of the *Arabidopsis* accessions coloured by sequence provider.

Your plot should look as follows:


data - se
GMI
Mon
MPI
Othe
Salk
ICELAND
FINLAND
NORWAY
SWED
DENMA
IRELAND
BELARUS
UKRAINE
FRA
GREECE
TURKEY
TUNISIA
SYRIA
IRAQ
ISRAEL
MOR CCO
ALGERIA
LIBYA
EGYPT
SAUDI ARABIA
MAURITANIA
MALI
NIGER
CHAD
ERITREA
YEMEN
SENEGAL
1000 km
500 mi
Leaflet | © OpenStreetMap contribu

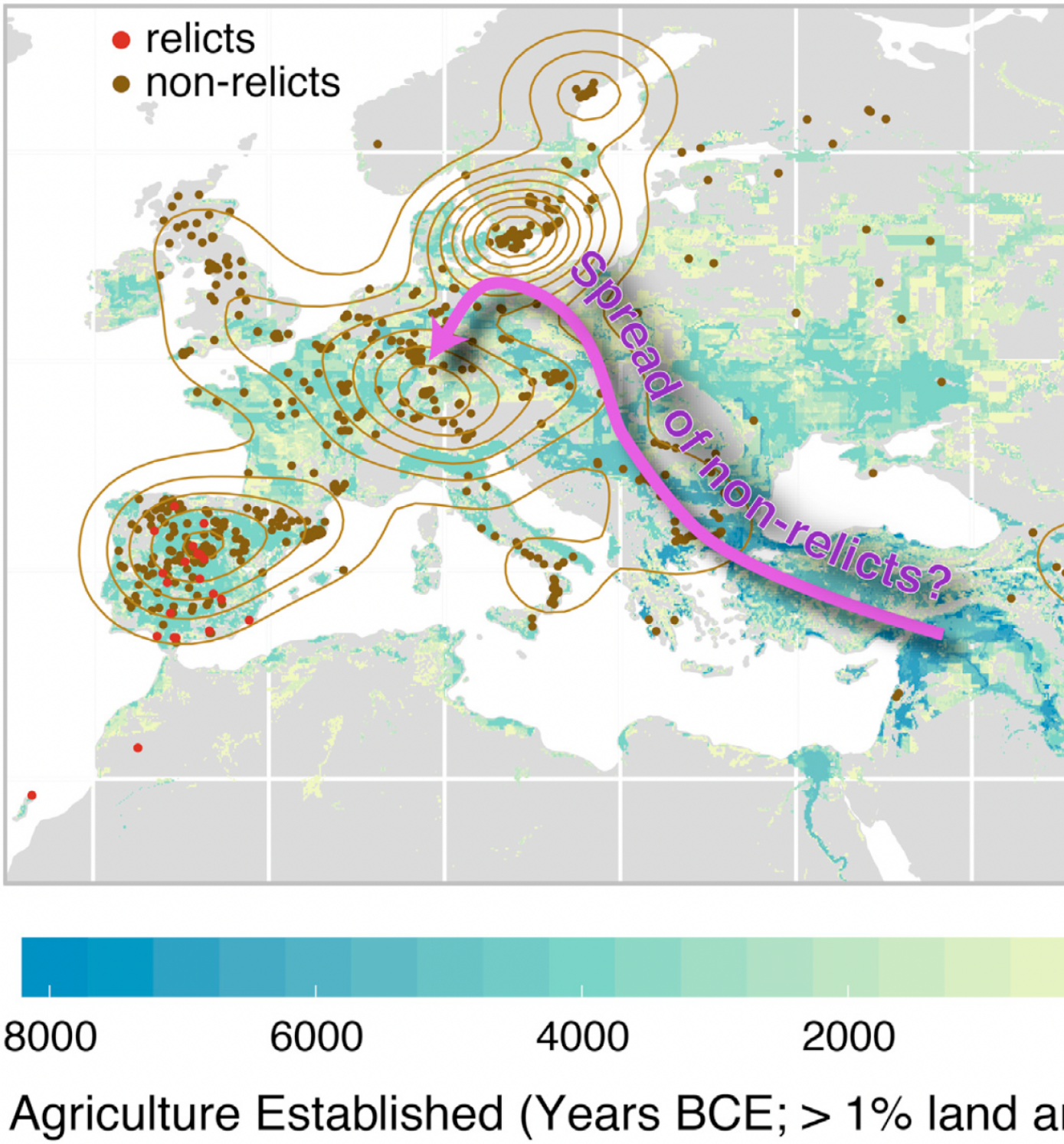
12.11 Reproducibility of results

It is often important to be able to reproduce results from previous research. The usual meaning here is that a new / independent experiment or study is performed, confirming (or not) the previous findings. However, at its very least reproducibility should hold true within a dataset, ie. another researcher that tries to re-run your analysis with your data should be able to generate the same outcomes. In many cases a workflow or script is provided as part of a publication so that the results can be easily generated.

In the following two exercises you are asked to reproduce two of the figures from the 2016 Cell paper.

12.11.1 Graphical Abstract

The first figure we will try to re-create is relatively straightforward, relying on the `mapview::mapView()` function as before but this time only distinguishing relicts from non-relicts:

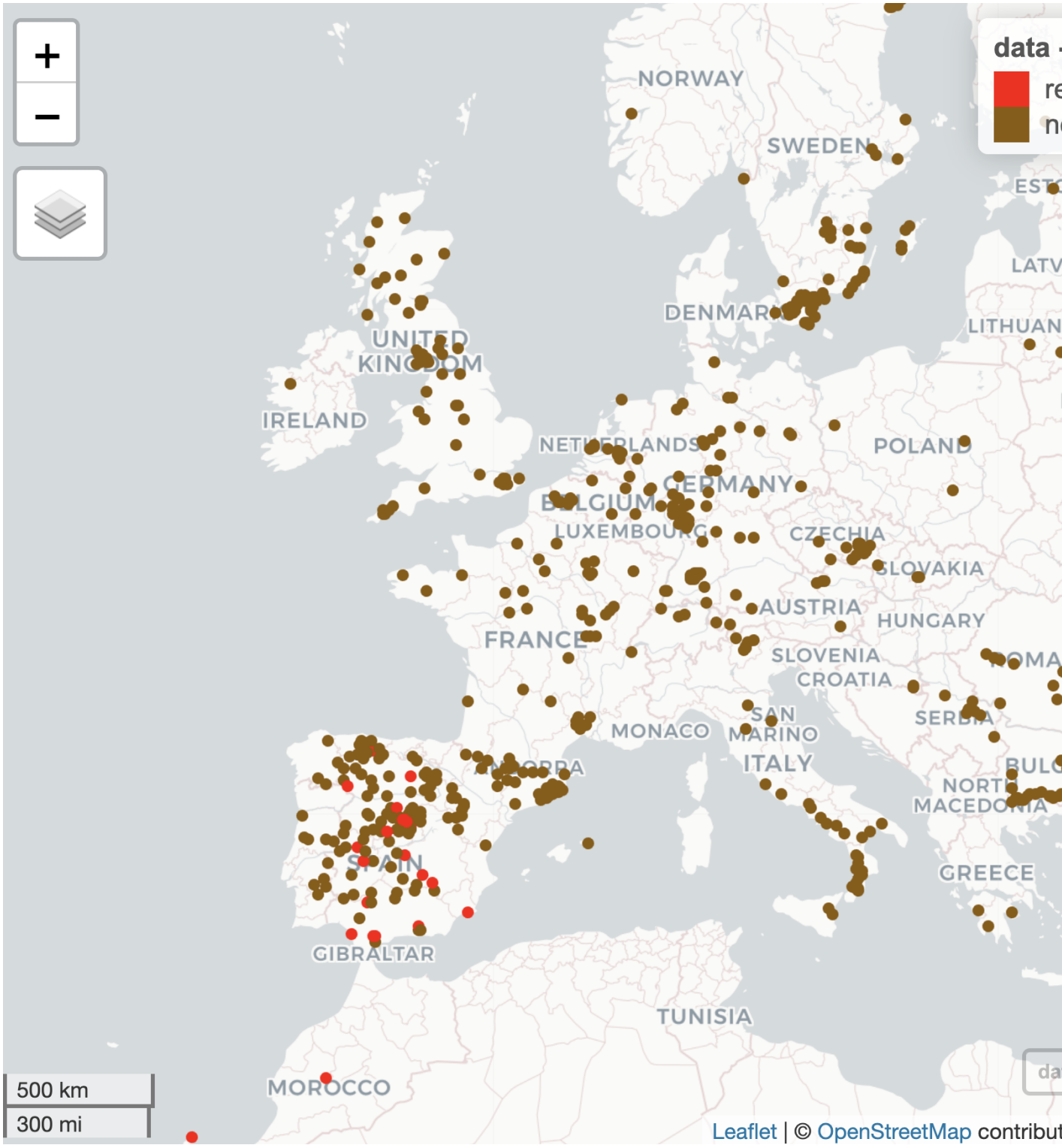

relicts
non-relicts
Spread of non-relicts?
8000
6000
4000
2000
Agriculture Established (Years BCE; > 1% land a

The authors probably used Adobe Illustrator or Powerpoint (or perhaps Coral draw etc etc.) to overlay the pink arrow showing the possible spread of non-relicts back into continental Europe following the retreat of the glaciers at the end of the last Ice Age. We are not interested in overlaying this arrow and text, but we would like to create the underlying plot that distinguishes between relicts and non-relicts only.

Exercise 12.9 (mapview relicts)**.**

- Add a new column to your dataset that distinguishes samples as either “relicts” or “non-relicts”
- Using the `mapview::mapView()` function, generate a plot that shows the geographical distribution of ‘relicts’ and ‘non-relicts’ in red and brown, respectively. Ensure that ‘relicts’ are listed first in the legend.
- Change any other plot attributes to get as close to the original plot as possible.

The resulting plot should look something like this:


NORWAY
SWEDEN
DENMARK
UNITED KINGDOM
IRELAND
NETHERLANDS
POLAND
GERMANY
BELGIUM
LUXEMBOURG
CZECHIA
SLOVAKIA
AUSTRIA
HUNGARY
FRANCE
SLOVENIA
CROATIA
SERBIA
MONACO
SAN MARINO
ITALY
NORTH MACEDONIA
GREECE
SPAIN
GIBRALTAR
TUNISIA
MOROCCO
500 km
300 mi
Leaflet | © OpenStreetMap contribut

12.11.2 Supplementary Figure S5

As a follow-up exercise, you are now asked to reproduce (part of) a supplementary figure of the 2016 Cell paper (Figure S5), showing the distribution of Variants by Type and Group:

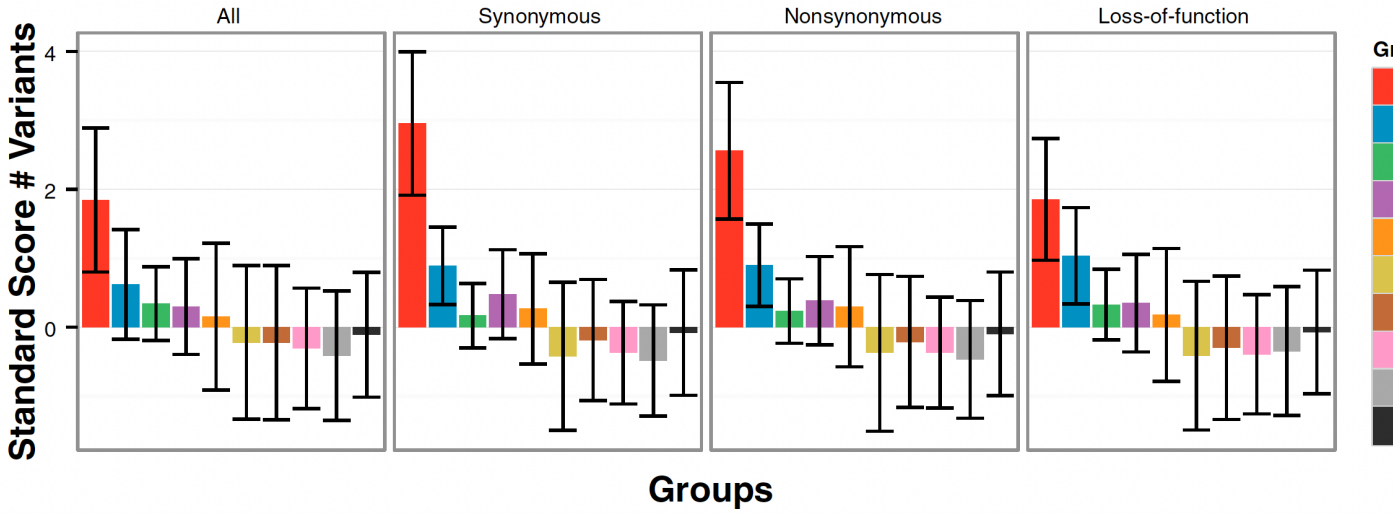


Figure S5. Variants by Type and Group, Related to Figure 5
Mean and standard deviation of the standard score (Z-score) of the number of variants of each type, by group. Relicts show the greatest norm
variants, especially of synonymous variants. Bars indicate means, whiskers represent one standard deviation.

To keep things simple, we will only reproduce the first panel (the bar chart on the left) where we do not make any distinction between synonymous, non-synonymous or loss-of-function variants (it is not obvious how we could categorise SNPs in this way without having extra information on the flanking sequence of the SNP, from which we could determine whether it is in a coding area of a gene and whether the base-pair change would lead to any differences in the protein when transcribed).

As you can see from the figure caption, the plot shows the mean and standard deviation of the standard score (Z-score) of the number of variants by group. Whiskers represent one standard deviation.

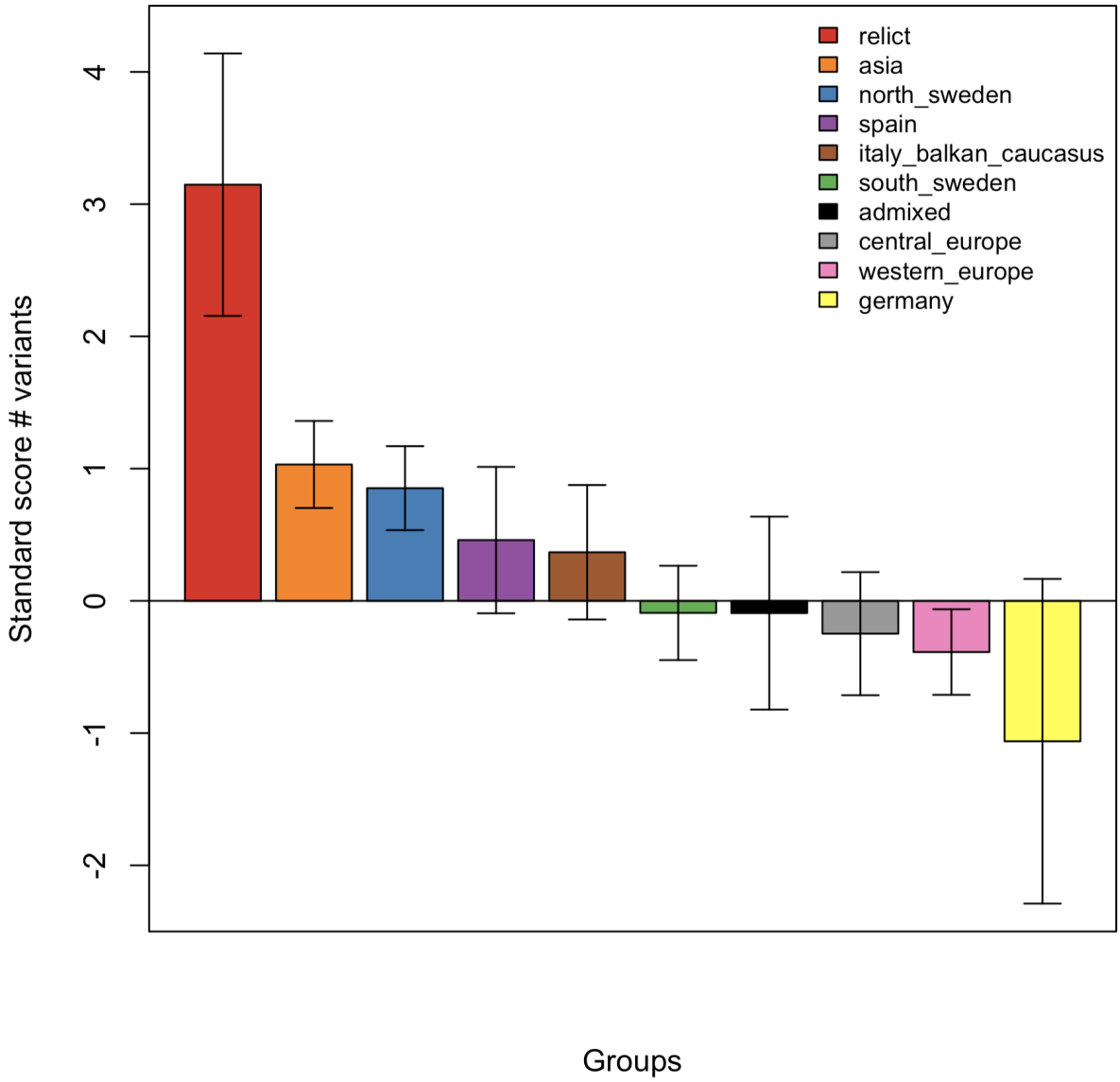
- **Can you recall what a Z-score is, how it is calculated, and why they are useful?**

Exercise 12.10 (Reproduce Figure S5)**.**

- Using the random 10K SNPs, first calculate the number of variants (i.e. SNP alternative allele counts) per accession.

- Convert these counts into standard (Z) scores using the `scale()` function.
- Summarise these Z-scores per admixture group, calculating both the `mean()` and `sd()` Z-score of each group.
- Using `barplot()` (or if you prefer, `ggplot2::geom_bar()`), generate a bar chart showing the means. Order the barplot in descending order (so, the admixture group with most SNPs appears first etc.). Colour the bars using the same colours as before (consistent with the publication figure).
- Add the standard deviations per group as error bars, and finally, add a legend.

If you were able to perform all these steps, you should have produced a plot looking something like this:



Although not exactly reproducing Figure S5, the trend shown is broadly similar (this was using data from 10K rather than 10M SNPs).

12.12 Phenotypic diversity

We leave the SNP data for now to have a look at the phenotype dataset we downloaded earlier (on my computer this is a CSV file called 'values.csv'):

```
pheno <- read.csv('values.csv', header = TRUE)
```

Exercise 12.11 (FT trait comparison)**.**

- What is the (Pearson) correlation coefficient between the two traits (FT10 and FT16)?
- Make a suitable visualisation to compare values of FT10 versus FT16 across the population (ideally a scatter plot). Are there any visible outliers?
- One of the accessions has quite a high FT10 value (greater than 150 days). What is the name of the accession and where was it collected?

We are going to once more visualise the geographic distribution of accessions, but this time coloured by flowering time.

Before proceeding, reflect on what *sort* of data flowering time is? (i.e. is it Sequential, Diverging, or Qualitative)?

Have a look again at the website of Paul Tol. A tweaked version of the *YlOrBr* scheme is suggested; choose this or another suitable colour scheme that is mentioned.

If you downloaded the flowering time data in a single file, you only have the phenotypic data and the accession IDs available (as well as a replicate ID that we do not use):

```
head(pheno)
```

In order to combine the phenotypic data with the geographic co-ordinates, we need to `merge()` the two datasets. Have a look at the help menu for the function and see whether you can create a merged dataset yourself.

Exercise 12.12 (mapview FT)**.**

- Using an appropriate colour palette, produce a visual display of the geographic distribution of *Arabidopsis* accessions coloured by flowering time (either FT10 or FT16), again using `mapview::mapView()`.

Hint: use the argument `zcol` to specify the column name you would like to use to define the colours on the points, and `col.regions` argument to supply the colour palette.

Reflect on what you see - is there an association between flowering time and geographic distribution? What explanatory variables might you include in the model? What would the response variable be?

As an additional exercise, fit a model that tries to predict FT16 from geographic location. Is this a good model? Why / why not? Could you suggest any improvements?

12.13 Genome-wide Association Study (GWAS)

A GWAS is a well-established method to uncover genetic variation that underlies phenotypic variation. Although originally applied to human genetics studies, it has also had a big impact on the identification of important genetic variants in both plant and animal populations.

Although there are quite a number of different methodologies to perform GWAS, they all share similar elements:

- a statistical model to relate genetic variation at a locus with phenotypic variation,
- a method to control for differences in relatedness between samples,
- a method for declaring that an association is "significant", in particular controlling for false-positive associations due to multiple testing.

One of the most common GWAS models applied to plant and animal populations is the linear mixed model with kinship correction [6]. We will be using this model as implemented in the `statgenGWAS` package [10] to look for significant SNPs underlying variation in flowering time in *Arabidopsis*.

Although the genetics underlying flowering time are of course of interest, we are also interested here in arranging the data we have been working with to be able to run the GWAS. In many cases with more advanced analytical methods that have been implemented into software packages, half the work can be in organising and transforming your input files to the expected format that the package requires. Hence, the emphasis here is on practicing the manipulation of a dataset into a suitable format.

12.13.1 Package manual / vignette

There are a number of manuals available for the `statgenGWAS` package, either the very concise package overview which gives a short overview of what is possible, or the more complete package vignette which explains more of the theory as well as more extensive worked examples. If you read the documentation, you will see that the central object used in the package is an object of class `gData`. There is a function `statgenGWAS::createGData()` to create a `gData` object, with a number of arguments. In our case, we will be supplying SNP data via the argument `geno`, marker positions via `map` and the phenotypes via the argument `pheno`.

For the purposes of this exercise, we will use the 10K randomly-selected SNPs for our genotypic data. If you recall, you (should have) given meaningful row and column names to your SNP dataset. Let's double-check this:

```
SNP <- readRDS("SNP.RDS")

## Have a look at the first 10 columns (and 6 rows):
head(SNP[,1:10])
```

As we can see, the rownames are the accession IDs, and the column names are bp positions.

Exercise 12.13 (Prepare GWAS data)**.**

- Run `?createGData` and read the **Arguments** section. In what format should `map` be in?
- Make a suitable map object for the markers contained in `SNP` to pass to the `map` argument in the `statgenGWAS::createGData()` function.
- Make any other changes as needed to adapt the objects passed to the function arguments `geno` and `pheno`, and then create the `gData` object by running the function `statgenGWAS::createGData()`.

Once you have completed these steps, you can remove duplicate markers using the `statgenGWAS::codeMarkers()` function. You are now in a position to run a single trait GWAS model.

Exercise 12.14 (Run GWAS)**.**

- Use the `statgenGWAS::runSingleTraitGwas()` function to perform a GWAS analysis using the SNP data for the traits 'FT10' and 'FT16'.
- Have a look at the `summary()` of the output. Did you detect any significant QTL for either of the traits? On which chromosome(s)?
- Generate QQ-plots and Manhattan plots for both analyses.
- If time permits, try using a more advanced approach by correcting for kinship chromosome-by-chromosome, as described in the vignette. Does this result in any more QTL being detected? What about (possible) p-value inflation?

13 String operations

13.1 Introduction

Although R is primarily a tool for dealing with numeric data, it is also possible to perform string operations. In this final chapter, we will be practicing string manipulation as a final part of the course.

> Tip
>
> A **string** is a separate data type in R, which you have probably already come across in earlier chapters. Strings are often used for descriptive variables (e.g. the name of a tomato variety like 'Moneymaker'), but can also more generally be any group of alphanumeric characters, e.g. "abc123_z". To denote a string we use either `''` or `""`, which are interchangeable, although double quotation marks are preferred.
>
> ```
> ?Quotes # same information as via the link above
> ```

"String operations" could mean any number of things. For example, concatenating strings together, splitting a character string into a number of parts, replacing a capital letter with a small letter, removing a `.` character from a name etc.

Once you start working with data more regularly, you will find it is not always in the form(at) you want. String operations are often needed to "get something to work". Here, we introduce a few of the most common and useful functions for performing string operations, before briefly introducing Regular Expressions.

13.2 R functions for strings

13.2.1 String concatenation with `paste()`

You have probably already seen this function, or its cousin `paste0()`. Like many R functions it has a self-describing name, it pastes (strings) together. `paste0` pastes with zero spaces between.

Exercise 13.1 (Pasting)**.**

- Paste the first 10 letters and the first 10 positive integers together (without any separating space).
- Paste the first 10 letters and the first 10 positive integers together, this time separated by an underscore (so "a_1" would be the first element of the resulting vector).
- Repeat the first exercise, but this time pasting the first 10 positive integers together with the next 10 numbers (i.e. paste 1 - 10 together with 11 - 20). What is the data type of the output? *Hint*: use `class()` to check.

Leading zeros

If you are assigning names to things like experimental treatments, you sometimes combine character strings with numbers quite naturally, for example "genotype1", "genotype2", "genotype3".
We have already seen how to do this using `paste0()`:

```
paste0("genotype", 1:3)
```

```
[1] "genotype1" "genotype2" "genotype3"
```

Suppose we had 20 genotypes in an experiment, we might reasonably call them genotype1 - genotype20:

```
genos <- paste0("genotype", 1:20)
```

But if we are doing an analysis or plotting the data afterwards, you will notice that the order of the genotypes is not g1:g20, instead they are ordered as follows:

```
levels(as.factor(genos))
```

```
 [1] "genotype1"  "genotype10" "genotype11" "genotype12" "genotype13"
 [6] "genotype14" "genotype15" "genotype16" "genotype17" "genotype18"
[11] "genotype19" "genotype2"  "genotype20" "genotype3"  "genotype4"
[16] "genotype5"  "genotype6"  "genotype7"  "genotype8"  "genotype9"
```

Why do we go from genotype1 to genotype10? Doesn't R know how to count?
The problem of course is that "genotype1" is now a string, and strings are ordered alphanumerically, meaning that genotype10 comes before genotype2.
We could force the ordering of the genotypes by converting it to a factor with defined levels:

```
genos <- factor(paste0("genotype", 1:20), levels = paste0("genotype", 1:20))
```

This can be a useful approach if you want a specific order - we already encountered this when trying to visualise the geographic distribution of admixture groups in the same order as the publication (Section 12.10).
Here, we'll mention a different approach - using **leading zeros**. There are a number of ways to pad a number with zeros, but here we will only consider one, using the function `sprintf()`. Have a look at the documentation and see whether you can work out how to add a leading zero so that the genotype names order as we would like. Try this before looking at the **Code** below.

```
genos <- paste0("genotype", sprintf("%02d",1:20))

## The first argument starts with % and ends with d (for integers). In between we tell the

genos
```

```
 [1] "genotype01" "genotype02" "genotype03" "genotype04" "genotype05"
 [6] "genotype06" "genotype07" "genotype08" "genotype09" "genotype10"
[11] "genotype11" "genotype12" "genotype13" "genotype14" "genotype15"
[16] "genotype16" "genotype17" "genotype18" "genotype19" "genotype20"
```

13.2.2 Manipulating strings

Suppose we have received a dataset from another researcher where plant heights were measured in centimeters. Unfortunately, the researcher recorded everything in Excel with the letters "cm" following the numbers. We would like to extract the actual heights as a **numeric** data type so that we can produce a plot.

We can generate an example of such a dataset as follows:

```
sampledata <- data.frame(Genotype = c("Warwick","Hopkins","Venture","Hopewell","Whitby","Bess
                         Height = c("36.5cm","28.9cm","26.5cm","33.4cm","39.1cm","32cm"))
```

Genotype	Height
Warwick	36.5cm
Hopkins	28.9cm
Venture	26.5cm
Hopewell	33.4cm
Whitby	39.1cm
Bess	32cm

Your initial reaction might be to go back to the Excel file you received and delete the "cm" by hand. That might be a good solution in this example, but it would be less good if there were 6000 or 60000 rows in the dataset!

Find and Replace is another option, but this can have unwanted consequences if applied to a whole file (by perhaps deleting "cm" from other columns unintentionally).

Of course, by now you should know that you can do pretty much everything in R! There are a number of ways to achieve this. We will demonstrate the functionality of the functions `substr()` and `strsplit()`, using this example.

13.2.3 Extracting a substring with `substr()`

`substr()` creates a sub-string of the input, which is exactly what we want here.

Have a look at the function documentation first:

```
?substr
```

From reading this, you should see that you need to know the **start** and **stop** position of the characters you want to retain. This causes a slight problem here, as the last height measurement (32cm) was recorded without specifying the decimal place, while the rest were recorded to 1 decimal place. We want to *subtract* two characters from each string, rather than count from the start (which would go wrong for 32cm if we stopped at the 4th character).

To get around this, we use another (string) function `nchar()`, which counts the **n**umber of **char**acters of a string:

```
nchar(sampledata$Height)
```

```
[1] 6 6 6 6 6 4
```

We can tell `substr()` that we want to stop 2 characters from the end as follows:

```
substr(sampledata$Height,
       start = 1,
       stop = nchar(sampledata$Height) - 2)
```

```
[1] "36.5" "28.9" "26.5" "33.4" "39.1" "32"
```

> **Recycling**
>
> Notice that in this function, we were able to specify the start position (= 1) only once, and the function understood that we wanted this start position to be used for *all* 6 heights. This is an example of **recycling**, which is built into many but not all R functions.
> If it were not possible to recycle here you would get an Error (something like argument lengths don't match), and we would have had to use `start = rep(1,6)` to **rep**eat the number 1, 6 times (so, make a vector `c(1,1,1,1,1,1)` as the start positions).

Finally, if we want to get the heights as numeric data, we would wrap the whole expression with `as.numeric()`, and perhaps add it on to the data-frame as a new column:

```
sampledata$hgt <- as.numeric(substr(sampledata$Height,
                                    start = 1,
                                    stop = nchar(sampledata$Height) - 2))
```

Genotype	Height	hgt
Warwick	36.5cm	36.5
Hopkins	28.9cm	28.9
Venture	26.5cm	26.5
Hopewell	33.4cm	33.4
Whitby	39.1cm	39.1
Bess	32cm	32.0

13.2.4 Splitting strings with `strsplit()`

One of the most basic operations is being able to split a character string at a particular symbol (or combination of characters) such as a ".", "_" or "-" for example. These symbols often turn up where they are not wanted in datasets, and might need to be removed to proceed with an analysis or operation.

As the function name suggests, `strsplit()` splits (character) strings at a symbol or combination of characters. You can again check out `?strsplit` for the details, but we can apply `strsplit()` in our example above by asking `R` to split our heights at the offending characters "cm":

```
strsplit(sampledata$Height,split = "cm")
```

```
[[1]]
[1] "36.5"

[[2]]
[1] "28.9"

[[3]]
[1] "26.5"

[[4]]
[1] "33.4"

[[5]]
[1] "39.1"

[[6]]
[1] "32"
```

The output is a list, and because there was nothing after the "cm", each list element is of length 1. Splitting a string like "21_583" at _ would result in a vector of length two containing the string before and after _.

A list is not always the most convenient structure for downstream applications, so we can turn it into a vector either using the `unlist()` function (collapses the list) or, more elegantly, using `sapply()` with `[[` as follows:

```
sapply(strsplit(sampledata$Height,split = "cm"),"[[",1)
```

```
[1] "36.5" "28.9" "26.5" "33.4" "39.1" "32"
```

This approach picks out the *first* element of each list item (the 1 after "[[" specifies this). In another data-set you might want to pick out the *second* result (for example the number after the _), in which case you would replace the 1 with a 2. Alternatively, you could specify the column name instead of the column number.

> **Note**
>
> You have already encountered the `apply` family of functions in earlier weeks, for example in week 2 (Section 4.1) when you learned about functions.
> The functions `apply()`, `sapply()`, `lapply()` and `tapply()` are the most commonly-used (`mapply()` is also available, but I have never used it).
> Check out the documentation for `sapply()`, which is arguably the most-used function in the family:

```
?sapply
```

The **Usage** section mentions that this function has 3 main arguments, namely `X` (the input object), `FUN` (the function to apply to that object) and `...` (ellipses, to pass extra arguments to `FUN`).
In the code above, `[[` is actually a function we are applying to the output of `strsplit()`, and `1` is the extra argument we are supplying to `[[`[1].

13.3 Regular expressions

One extra "programming tool" topic that can get quite complicated but nonetheless is incredibly useful when you need it is **Regular expressions**. At the very least, just knowing this term may help you successfully Google a solution to a tricky data formatting problem in the future.

Regular expressions are used for character string searches, in a somewhat similar way to Find (and Replace) in MS Word. If you are familiar with Word, you may have noticed that in an Advanced Search, it is possible to "Match case", only matching capital letters with capitals etc.

Regular expressions can be thought of as a way to describe such matching rules in more detail. Different programming languages have their own syntax for Regular Expressions, including `R`, which by default uses so-called *extended* regular expressions (although it is possible to follow the conventions used in Perl for "Perl-like" regular expressions).

A good description of Regular Expression usage in R can be found by looking at the documentation via `?regex`.

Associated R functions are `grep()`,`gsub()` etc - these are functions that accept regular expressions as matching arguments (usually called the "pattern", but check function documentation to be sure).

Here we present an example to give you an idea of how regular expressions can be used in R.

We first generate a vector of character strings that mix letters, numbers and an underscore:

[1] Note that I have deliberately written `[[` and not `[[()` when referring to the function `[[` here. This goes a bit against convention: elsewhere in these notes functions are written like `plot()` to explicitly refer to the function `plot`.

```
set.seed(42) #for reproducibility

vect <- paste0(sample(c("M","Tr","Px","Z"),100,replace=TRUE),
               sample(1:10000,100),"_",sample(1:100,100))

vect[1:10] #show the first 10 elements of vect
```

```
 [1] "M8274_1"   "M9106_17"  "M2344_33"  "M149_28"   "Tr7140_2"  "Z5689_31"
 [7] "Tr100_8"   "Tr2346_80" "M2450_3"   "Z91_13"
```

Note that by setting the same seed, you should see the same set of strings as shown here.

We suppose these are genetic marker names, with some prefix depending on the population in which the marker was identified, followed by the contig number and bp position in the contig. We would like to separate out the contig number, the base-pair position and also the prefix itself (so, we do not want to throw away the information, just separate it).

If we performed a string split at the "_", we could get the base-pair numbers simply enough, but how to separate the letters from the contig numbers? There is no separating symbol like an underscore to split on. To do this, we probably need to use a regular expression approach.

Letters can be represented by "[A-Za-z]" (this considers both upper and lower case letters a match. For only lower case you would use "[a-z]").

However, we also want to be able to match **one or more** letters (some of the prefixes are single letters, some are 2 letters). We therefore need to add "+" directly afterwards (otherwise we are asking R to match *exactly* one letter). So the way to ask R to match any of the prefixes in `vect` is with the regular expression **"[A-Za-z]+"**.

Similarly, to match numbers we would use the regular expression "[0-9]+", which covers (integer) numbers from 0 to infinity (or close enough). Putting this together, we could split up the vector in a couple of steps as follows:

```
split1.ls <- strsplit(vect, split = "[A-Za-z]+") #split on the prefix letters i.e. remove the

split1.vect <- sapply(split1.ls,"[[",2) #extract the 2nd list elements, like 8274_1 etc

split2 <- strsplit(split1.vect,split = "_") #now split on the underscore, using output splitt

contig <- sapply(split2,"[[",1) #extract the contig number

bp <- sapply(split2,"[[",2) #extract the base pair number
```

```
prefix <- sapply(strsplit(vect, split = "[0-9]+"),"[[",1) #re-split by number and extract the
```

Try this yourself by copying and pasting the above code into R, and see whether contig, bp and prefix are correct. Notice that in the last line of code, we perform two operations: first splitting `vect` at the numbers, and then pulling out the first element of the resulting list. As you use `R` more, you will find that it becomes easier to build up a few operations in a single line by nesting bits of code within other code, passing the output from one function as the input to the next function wrapping it.

So we read (base) `R` code not so much from left to right, but from inside to out (and this is also how the code is written, from inside out). This is often presented as one of the main arguments for using `tidyverse` syntax and piping, making it easier for others to read as well by following a natural textual order from left to right and top to bottom[2].

If you are unsure about how part of the code works, just highlight that part of the code (e.g. `strsplit(vect, split = "[0-9]+")`) and Run that (Ctrl + Enter) to make sure it works.

A more elegant solution would be to split the regular expression into "groups" (defined using round brackets), and access each of these groups in turn using the `gsub()` function:

```
myregex <- "([A-Za-z]+)([0-9]+)(_)([0-9]+)" #here we define 4 "groups", each contained by rou

prefix <- gsub(myregex, replacement = "\\1",vect) #replace the whole string with the match fi
contig <- gsub(myregex, replacement = "\\2",vect) #similarly, group 2 (denoted \\2) is the co

bp <- gsub(myregex, replacement = "\\4",vect) #we skipped group 3, the underscore
```

If you want to do all this in a single line of code, you will need to install the `stringr` package which extends the capabilities of R a bit further in this direction:

```
install.packages("stringr")

result <- stringr::str_match(vect,"([A-Za-z]+)([0-9]+)(_)([0-9]+)")

head(result) #show the first 6 rows of the result
```

```
     [,1]        [,2] [,3]   [,4] [,5]
[1,] "M8274_1"   "M"  "8274" "_"  "1"
[2,] "M9106_17"  "M"  "9106" "_"  "17"
```

[2] As an enthusiast of `base R` (for its stability, few dependencies etc), I find the readability argument somewhat spurious. Personally, I primarily write code for a computer to read. If I want to write something for a human to read, I document my code via comments, as should you!

```
[3,] "M2344_33" "M"  "2344" "_"  "33"
[4,] "M149_28"  "M"  "149"  "_"  "28"
[5,] "Tr7140_2" "Tr" "7140" "_"  "2"
[6,] "Z5689_31" "Z"  "5689" "_"  "31"
```

Each of the groups are given in a separate column, along with the original string that needed to be split (useful for comparing to see it worked).

Exercise 13.2 (Practicing string manipulations)**.**

- Using the `substr()` function, remove the starting letter from elements of the vector `v`:

```
v <- c("M1234567","N23456","O345678","P4567","Q567890","R67")
```

- Extract the words "COVID" and "SARS" from the vector `cov`, ie. extract the word before the first dash symbol:

```
cov <- c("COVID-19","SARS-CoV-2")
```

- Standardise the vector 'test' below to have an underscore between the prefix letters and numbers (so, all elements having the form Na_123 for example):

```
test <- c("Br 10992","Fe  9056","Io   999","C    14","Ka 167")
```

- Convert the following string into lower-case letters only:

```
book_string <- "Down and out in Paris and London. Ed. Penguin Books Ltd."
```

As usual, please attempt these exercises yourself first! If you are unable to complete them after a reasonable attempt, a sample script will be made available on Brightspace week 4. If it is not yet visible, please raise your hand.

13.4 Unexpected data formats and `readLines()`

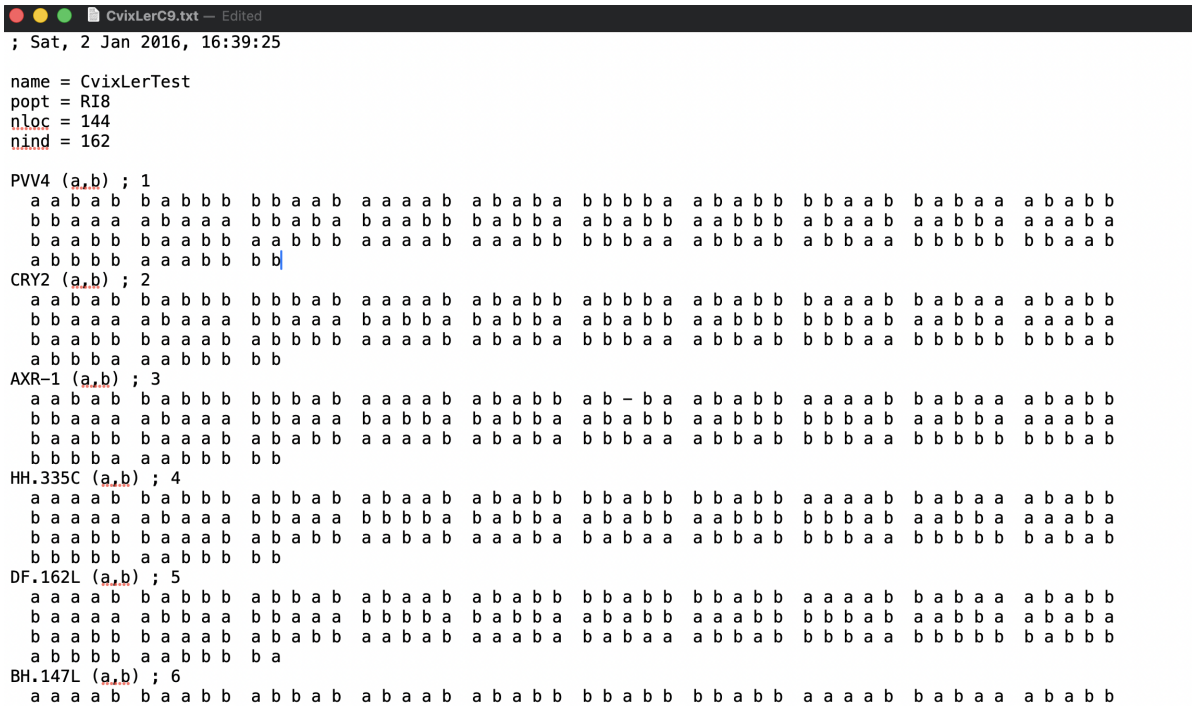
As a final topic in this section, we will briefly look at what to do when the data you would like to work with is not loading properly.

You have been practicing the data-science cycle during the past four weeks, but occasionally data is not in an easy-to-load format. This can often happen with older datasets that were prepared with older (and perhaps obsolete) software in mind, each with its own syntax (you've already encountered some syntax differences in this course - for example,

comments in R are preceded with a hashtag (#), while in markdown files hashtags denote Headers!).

I was recently asked by a former colleague to advise on R software packages for QTL mapping. The datafile the researcher wanted to use was downloaded from TAIR - Arabidopsis again! - and was in JoinMap format (if you are interested, you could visit Kyazma's website).

Looking at the file in a text editor shows the difficulty in accessing the data in R:



The initial line in the file is preceded by ";" denoting a comment, followed by a time-stamp. The next line is blank, there are then 4 lines with some descripive information on the name of the population, the population type, the number of loci and individuals, followed by another blank line and then the marker information.

If we try to read this data in using `read.table()` or `read.csv()` we will get an error!

```
data <- read.table("CvixLerC9.loc")
```

```
Error in scan(file = file, what = what, sep = sep, quote = quote, dec = dec, : line 2 did not
```

The end of this rather long Error message is informative - line 2 did not have 6 elements. The function `read.table()` expects data to be in a sqare format, not with unequal numbers of potential columns from one line to the next.

If you dig into the documentation for the function, you can see that there is an argument `skip` that we might use, skipping over perhaps the first seven or eight lines lines. Let's try it...

```
data <- read.table("CvixLerC9.loc", skip = 7)
```

```
Error in scan(file = file, what = what, sep = sep, quote = quote, dec = dec, : line 1 did not
```

```
## or maybe:
data <- read.table("CvixLerC9.loc", skip = 8)
```

```
Error in scan(file = file, what = what, sep = sep, quote = quote, dec = dec, : line 4 did not
```

`read.tables()` keeps getting stuck. Not only are there unequal numbers of elements, the data has been organised in chunks of 50 data-points (a or b denoting allele a from parent 1, or b from parent 2). With 162 individuals, we are left with an overhang of 12 elements after the first 150..

How can we proceed? Do we throw out the data? No need! `R` has this covered, namely with the function `readLines()`!

```
attempt1 <- readLines("CvixLerC9.loc")
```

```
dim(attempt1)
```

```
NULL
```

```
attempt1[1:10]
```

```
 [1] "; Sat, 2 Jan 2016, 16:39:25"
 [2] ""
 [3] "name = CvixLerTest"
 [4] "popt = RI8"
 [5] "nloc = 144"
 [6] "nind = 162"
 [7] ""
 [8] "PVV4 (a,b) ; 1"
 [9] "  a a b a b  b a b b b  b b a a b  a a a a b  a b a b a  b b b b a  a b a b b  b b a a
[10] "  b b a a a  a b a a a  b b a b a  b a a b b  b a b b a  a b a b b  a a b b b  a b a a
```

`attempt1` is a character vector, not something that you usually have to work with as genetic marker data, but here it is the first step at getting the data into `R`. A number of string operations will be necessary to extract the genotypic data and convert it into a numeric matrix.

This is left as a final exercise to practice on (slightly advanced level). A sample script to complete this exercise will be made available via Brightspace after a reasonable attempt has been made by most students.

Exercise 13.3 (Converting output of readLines)**.** The steps described below provide a suggested approach to solve this problem. You may think of another way, which is also fine.

- Use the `readLines()` function to read in the .loc file. Check how many lines have been read in.
- Use `grep()` with a suitable "pattern" to find the lines with the marker names.
- Apply a suitable string splitting approach to extract the marker name without the extra information or comment (e.g. `PVV4 (a,b) ; 1` should be simply be `PVV4` etc.)
- Use the marker name information to also work out the line numbers of where the genotypic data starts. *Hint* After each markername there is always 4 lines of marker information.
- Convert the dataset to numeric format in a matrix, with 0 in place of 'a' and 1 in place of 'b'. Have markers in rows and individuals in columns.
- Use the marker names as the rownames of the matrix.
- Double-check the data for characters other than 'a' or 'b', and make sure these have been correctly handled in the conversion step.

13.5 Assignment for week 4 code peer-review

Finally, the following exercise needs to be submitted by the end of the day today, in .qmd format. Please check that the file actually Renders correctly before submitting!

Note that you are not provided with a markdown / quarto template for the assignment. This means you will also need to pay attention to the layout and structure of the document yourself.

To get started, create a new .qmd file in `R` for your work. Choose suitable section headings (e.g. using question numbers given below) for each sub-question.

Exercise 13.4 (Week 4 (peer-review) assignment)**.**

1. Download the data file ("week4_assignment.dat") available from Brightspace week 4. Read it into `R` using a suitable function.
2. Describe the dataset along the following lines:
    - What is the structure of the dataset (data types in each column)
    - What is the dimension of the dataset
    - How many genotypes
    - How many replicates
    - Presence of missing values in each column.
3. Use appropriate string operational function(s) to extract numeric values from the height and biomass columns.
4. Make a scatter plot of the height (x axis) versus biomass (y axis), colouring the points by genotype. If you like, choose a colour-blind friendly colour palette.
5. Fit an appropriate linear model with biomass as the response variable and height as the explanatory variable, allowing for separate fitted lines per genotype.
6. Check the residual plots after model fitting. Do you find evidence of outliers in the data? If yes, remove these data points from the dataset and re-run the model. How do the residual plots look now?
7. Calculate the correlation coefficient between plant height and biomass. Compare this to the R^2_{adj} from the fitted model. Can you account for the discrepancy between these numbers?

You can submit the completed **.qmd** file via Brightspace for feedback on Thursday. Please do so before Wednesday night 23:59.

Part VI

Appendices

Base R cheatsheet

Atomic Data Types

Type	Example	Check
logical	`TRUE`	`is.logical(x)`
integer	`5L`	`is.integer(x)`
double	`3.14`	`is.double(x)`
character	`"a"`	`is.character(x)`
complex	`1+2i`	`is.complex(x)`
raw	`as.raw(1)`	`is.raw(x)`

Special Values

Value	Meaning
`NA`	Missing
`NaN`	Undefined number
`Inf`	Infinity
`NULL`	No object
`length(NULL)`	0

Data structures

Structure	Dim	Homogeneous?[3]	Example
vector	$1D$	Yes	`c(1, 2, 3)`
list	$1D$	No	`list(1, "a")`
matrix	$2D$	Yes	`matrix(1:4, 2)`
array	nD	Yes	`array(1:8, c(2, 2, 2))`

Structure	Dim	Homogeneous?[3]	Example
data.frame	$2D$	Yes (columns) / No (rows)	`data.frame(a = 1, b = "x")`
tibble[4]	$2D$	No	`tibble(a = 1, b = "x")`
factor	$1D$	Yes	`factor(c("a", "b"))`

Subsetting rules

Object	`[ ]`	`[[ ]]`
vector	subset	element
list	sublist	element
data.frame	df slice	column
tibble	tibble	vector

Type Coercion

logical –> integer –> double –> character

Examples

```
c(TRUE, 2)
# becomes: c(1, 2)

c(1, "a")
# becomes: c("1", "a")
```

Writing functions

(Named) function definition

[4] Tidyverse only
[3] i.e. all values are of the same type

```r
#' Arbitrary function
#'
#'@description
#'  `function_name` is an example function to highlight function syntax details.
#'  This function calls `other_function` as part of its implementation
#'  The documentation style is part of the roxygen package
#'
#'@param argument1 Description of the first argument (with it's type).
#'@param argument2 Description of the second argument (with it's type).
#'@param argument3 Description of the third argument (with it's type).
#'@param ... Arbitrary additional arguments pass on to `other_function`.
#'
#'@returns Brief description of type (and e.g. shape) of what the function returns
function_name <- function(argument1, argument2, argument3 = 'default_value', ...){
  # Function body
  # `...` are arbitrary arguments that can be passed on to other functions
  some_val <- other_function(argument1, ...)
  return(return_value)
}
```

Using (AKA calling) a (named) function

```r
# Positional arguments only
value <- function_name(value1, value2)

# Combining positional and named arguments (in arbitrary order)
value <- function_name(argument3 = 'other_value', value1, value2)

# Adding arbitrary arguments that will be passed on
# (only if function definition allows it)
value <- function_name(
  value1, # goes to argument1
  value2, # goes to argument2
  value3, # goes to argument3
  arbitrary_value4, # goes to ..., so (in this case) gets passed on to `other_function`
)
```

Anonymous functions

```r
# Some functions take other functions as arguments, these other function can be
# supplied in-line and are called 'anonymous' functions
value <- some_function(
  input,
  function(el) {
```

```
    # ...do something with el here
    # Note that `some_function` determines how this anonymous function will be called
  }
)

# Example
sapply(1:4, function(i){ i**2 })
# This returns: [1]  1  4  9 16
```

Control flow

Conditional execution

```
# Classic if/else statement (val is a single value)
if (val > 1) {
  # Do something
} else {
  # Do something else
}

# Vectorized version (val is a vector)
# new_val will be a vector with values 'large' and 'small'
new_val <- ifelse(val > 1, 'large', 'small')
```

For loops

```
for (i in x) {
  # do something with element i from collection x
}
```

While loops

```
while (condition){
  # Do something while condition evaluates to TRUE
  # Stops when `condition` changes, when `break` is used, or (when inside a function)
  # when a `return` statement is used
  if (other_condition){
    break
  }
}
```

Repeat loops

```r
repeat {
  # Do something repeatedly
  # When not in a function body, the only way to stop this is by using `break`
  # Inside a function body, a `return` statement also terminates the loop
  if (condition) {
    break
  }
}
```

The apply functions

Error handling

```r
# If no error occurs, `try` returns the normal return value, otherwise an error object
result_or_error <- try(some_function_call(arg1, arg2))

# `tryCatch` provides more fine grained control over what to do with errors or warnings
result <- tryCatch(
  function_call(arg1),
  error = function(e){
    # Optionally do something with the error object `e`
    return('Something to return on error')
  },
  warning = function(w){
    # Optionally do something with the warning object `w`
    return('Something to return on warning')
  }
)
```

Operators

Category	Operator	Name / Meaning	Notes
ARITHMETIC OPERATORS			
	+	Addition	Element-wise
	-	Subtraction	Element-wise
	*	Multiplication	Element-wise
	/	Division	Element-wise
	^, **	Power	Right associative
	%%	Modulo	Remainder
	%/%	Integer division	Floor division
COMPARISON OPERATORS			

Category	Operator	Name / Meaning	Notes
	`==`	Equal	Vectorised
	`!=`	Not equal	Vectorised
	`<`	Less than	Vectorised
	`<=`	Less or equal	Vectorised
	`>`	Greater than	Vectorised
	`>=`	Greater or equal	Vectorised
LOGICAL OPERATORS			
	`&`	AND	Vectorised
	`|`	OR	Vectorised
	`!`	NOT	Vectorised
	`&&`	AND	Short-circuit
	`||`	OR	Short-circuit
ASSIGNMENT OPERATORS			
	`<-`	Assign	Standard
	`=`	Assign	Contextual
	`->`	Right assign	Rare
	`<<-`	Super assign	Parent env
	`->>`	Super right assign	Parent env
SUBSETTING / EXTRACTION			
	`[`	Subset	Preserves type
	`[[`	Extract	Single element
	`$`	Named extract	Lists / data frames
SPECIAL INFIX OPERATORS			
	`%in%`	Membership	Set inclusion
	`%*%`	Matrix multiply	Linear algebra
	`%%`	Modulo	Arithmetic
	`%/%`	Integer division	Arithmetic
	`%>%`	Pipe (magrittr)	Tidyverse
	`%<>%`	Compound pipe	Tidyverse
PIPE (BASE R)			
	`|>`	Base pipe	R 4.1
FORMULA / MODELING			
	`~`	Formula	Symbolic relation
NAMESPACE OPERATORS			
	`::`	Exported access	Package API
	`:::`	Internal access	Non-exported
INTROSPECTION / OOP / Help			
	`@`	S4 slot access	Formal classes
	`?`	Help	Documentation
	`??`	Search help	Full-text

Basic functions covered in the course

R Functions by Category

File system

Function / Syntax	Description
setwd()	Set the working directory.

Package management

Function / Syntax	Description
install.packages()	Install R packages from CRAN.
library()	Load an installed package.

Data import

Function / Syntax	Description
read.delim()	Read tab-delimited text files.
read.csv()	Read comma-separated CSV files.
read.csv2()	Read semicolon-separated CSV files.
readxl::read_xlsx()	Read Excel .xlsx files.

Data exploration

Function / Syntax	Description
head()	Show the first 6 rows of data.
tail()	Show the last 6 rows of data.
summary()	Show descriptive statistics.
table()	Create frequency tables.

Dimensions

Function / Syntax	Description
ncol()	Number of columns.
nrow()	Number of rows.
dim()	Dimensions (rows × columns).

Statistics

Function / Syntax	Description
shapiro.test()	Shapiro–Wilk normality test.
t.test()	Student's t-test.
wilcox.test()	Wilcoxon non-parametric test.
cor()	calculates the correlation coefficient
cor.test()	calculates the significance of correlation
lm()	runs a linear model

Visualization

Function / Syntax	Description
heatmap()	Generate heatmap from a numeric matrix.
plot()	Base R generic plotting function.

Tidyverse Cheatsheet

All examples use built-in data (`data('iris')`) so you can copy and paste the examples. This is not an exhaustive manual, but describes the main approaches used in the book.

Installing and loading the entire tidyverse

```
install.packages('tidyverse') # only needed once
library(tidyverse)
```

The pipe operator |>

The pipe operator is used in the tidyverse to chain together actions described by the verbs. It works by injecting the output of one function as first argument in the next function:

```
# Pipe operator
# Output of function1 is injected into first argument of position2
function1(input, arg1) |> function2(arg2)

# Explicit nesting
# This performs the exact same computation as the previous example
function2(function1(input, arg1), arg2)

# Slightly more abstract
x |> f(y) == f(x, y)
```

Core `dplyr` verbs used in this course

`filter()`: select rows (observations) in a tidy dataset

Typically used with a *logical expression* using *comparison operators* (e.g. ==, >=, etc., see 'comparison operators' in the base R cheatsheet)

```r
# Select all observations of the setosa iris species
iris |>
  filter(Species == 'setosa')
```

`select()`: select columns (variables) in a tidy dataset

```r
# Select only petal width and petal length variables
iris |>
  select(Petal.Width, Petal.Length)
```

`group_by()`: create groups based on variables in the data

```r
# Create groups for individual species
iris |>
  group_by('Species')
```

`mutate()`: create new columns/variables (based on existing ones)

```r
# Create a new variable of the ratio between petal length and width
iris |>
  mutate(
    petal_ratio = Petal.Length / Petal.Width
  )
```

`summarise()`: create summaries of existing variables

```r
# Compute the mean and median petal width
iris |>
  summarise(
    mean_petal_width = mean(Petal.Width),
    median_petal_width = median(Petal.Width)
  )
```

GGplot cheatsheet

ggplot2 Beginner Cheat Sheet (R)

This quick reference focuses on the essentials you listed and a few logical additions for beginners. All examples use built-in datasets so you can copy–paste and run immediately.

```r
install.packages("ggplot2")   # once
library(ggplot2)
```

Building a plot

`ggplot()`

Creates a plotting object. Combine with + to add layers.

```r
ggplot(data = mtcars)  # creates an empty canvas using mtcars
```

`aes()` (aesthetics)

Maps variables to visual properties such as `x`, `y`, `color`, `fill`, `size`, `shape`.

```r
ggplot(mtcars, aes(x = wt, y = mpg, color = factor(cyl)))
```

Tip: Put mappings that apply to many layers in the top-level `aes()`. Put layer-specific mappings inside that layer.

Geoms (layers that draw things)

`geom_point()` – scatterplot

```
ggplot(mtcars, aes(wt, mpg, color = factor(cyl))) +
  geom_point(size = 2)
```

`geom_smooth()` – trend/fit lines

Adds model-based smooths (default: LOESS for n < 1000, otherwise GAM). Use `se = FALSE` to hide the ribbon.

```
ggplot(mtcars, aes(wt, mpg, color = factor(cyl))) +
  geom_point() +
  geom_smooth(se = TRUE, method = "loess")
```

Common options: `method = "lm"` for a straight regression line, `formula = y ~ x` for custom formulas.

`geom_histogram()` – distributions of a numeric variable

```
ggplot(mtcars, aes(x = mpg)) +
  geom_histogram(binwidth = 2, color = "white", fill = "steelblue")
```

Tip: Control binning with `binwidth` or `bins`.

`geom_jitter()` – jittered points to reduce overplotting

```
ggplot(iris, aes(Species, Sepal.Length, color = Species)) +
  geom_jitter(width = 0.15 ,height = 0)
```

Often used instead of (or alongside) `geom_point()` when x is categorical. Note that you have to 0, otherwise it also jitters in height!

Faceting (separating plots over a variable)

`facet_wrap()` – wrap a single faceting variable into multiple rows/cols

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point() +
  facet_wrap(~ cyl)
```

`facet_grid()` – 2D grid by rows and columns

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point() +
  facet_grid(gear ~ cyl)
```

Tip: Use `scales = "free"` to allow axes to vary per panel when ranges differ.

Labels, themes, and polishing

`labs()` – titles and axis labels

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point() +
  labs(
    title = "Fuel efficiency vs. weight",
    subtitle = "mtcars dataset",
    x = "Weight (1000 lbs)", y = "Miles per gallon",
    color = "Cylinders"
  )
```

`theme_bw()` – black-and-white theme

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point() +
  theme_bw()
```

Other useful themes: `theme_minimal()`, `theme_classic()`. Modify elements with `theme()`.

Some basic additions to plots

Scales (`scale_*`) – control colors, fills, axes, legends

```
ggplot(mtcars, aes(wt, mpg, color = factor(cyl))) +
  geom_point() +
  scale_color_brewer(palette = "Dark2", name = "Cyl") +
  scale_x_continuous(breaks = seq(1.5, 5.5, 0.5))
```

Why: Scales are how you fine-tune appearance and legends.

Position adjustments – dodge/stack/jitter

```
ggplot(iris, aes(Species, Sepal.Length, color = Species)) +
  geom_point(position = position_jitter(width = 0.15), alpha = 0.7)
```

Why: Helps manage overlapping points and grouped geoms (e.g., bars with `position = "dodge"`).

Coordinate systems (`coord_*`)

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point() +
  coord_cartesian(xlim = c(2, 5), ylim = c(10, 35))
```

Why: Zoom without dropping data (contrast with `xlim`/`ylim`).

A minimal template to reuse

```
library(ggplot2)
p <- ggplot(mtcars, aes(wt, mpg, color = factor(cyl))) +
  geom_point(size = 2, alpha = 0.8) +
  geom_smooth(se = FALSE, method = "lm") +
  facet_wrap(~ gear) +
  labs(title = "MPG vs Weight by Gears", x = "Weight (1000 lbs)", y = "MPG", color = "Cyl") +
  theme_bw()
print(p)
```

Quick summary

Function	Purpose
ggplot()	Create a plot object and define data.
aes()	Map variables to aesthetics (x, y, color, fill, size, shape).
geom_point()	Scatterplot points.
geom_smooth()	Trend lines (LOESS/LM) with optional SE ribbon.
geom_histogram()	Histogram for numeric variables.
geom_jitter()	Jittered points for categorical x.
geom_boxplot()	boxplot for numeric variables.
facet_wrap()	Facet by one variable (wrapped layout).
facet_grid()	Facet by row and column variables.
labs()	Titles, subtitles, captions, axis labels, legend titles.
theme_bw()	Black-and-white theme.
scale_*()	Control color/fill palettes, axes, legends.
coord_*()	Coordinate systems and zooming.
ggsave()	Save plots to files.

Datasets

QMD notebook cheatsheet

Setting up QMD

You start a QMD with a yaml header, this is information in-between three dashes (---). The first three dashes indicate the start of the header, the last three the end. There are some guidelines to writing yaml. But for this course it suffices to copy the header from the first two days

```
---
title: "BIF20806_week1day1_notebook"
editor: source
date: ADD
author: ADD
registration_number: ADD
format:
  html
execute:
  eval: false
---
```

Making a code chunk in QMD

Code is written in QMD between three back ticks ‘‘‘. After these backticks you should state {r}, which indicates that the code chunk is in the r-language. The yaml header you use states that this code is not evaluated (eval: false). You can force evaluation by either setting this to true (eval: true) or setting it to true in a specific code block using “#| eval: true”.

R setup in QMD

The first code chunk in the document should be {r setup}, this chunk specifies some general conditions. In particular, on some systems it is required to set the root directory

(your workdirectory) here as well (in addition to `setwd()`).

```
options(repos = c(CRAN = "https://cloud.r-project.org"))
### Some of you will have problems with setting the workdirectory (setwd())
### Here you need to set your work directory as well
knitr::opts_knit$set(root.dir = "YOUR_WD")
```

Writing text in QMD

Feature	Syntax	Explanation
Bold	**text**	Makes text bold
Italic	*text*	Makes text italic
Bold + Italic	***text***	Bold and italic
Inline code	‘code‘	Formats inline code
Strikethrough	~~text~~	Strikes through text
Headings	#, ##, ###, ####	Creates headings (H1–H4)
Bullet list	- item	Creates unordered lists
Numbered list	1. item	Creates ordered lists
Task list	- [ ] item	Creates checkboxes
Link	https://example.com	Creates a clickable link
Link	[text](https://example.com)	Creates a clickable link with custom text
Image	image.png	Inserts an image
Image	![text](image.png)	Inserts an image with a caption
Blockquote	> text	Creates an indented quote
Horizontal rule	---	Inserts a horizontal line
Footnote	text[^1] / [^1]: note	Adds a footnote
Line break	text (two spaces)	Forces a new line
Escape characters	*, #, _	Shows literal symbols

References

[1] Carlos Alonso-Blanco et al. “1,135 genomes reveal the global pattern of polymorphism in Arabidopsis thaliana”. In: *Cell* 166.2 (2016), pp. 481–491.

[2] Keith A Baggerly and Kevin R Coombes. “Deriving chemosensitivity from cell lines: Forensic bioinformatics and reproducible research in high-throughput biology”. In: *The Annals of Applied Statistics* 4 (Dec. 2009), pp. 1309–1334. arXiv: 1010.1092 [stat.AP].

[3] Manuel Benedetti et al. “Biomass from microalgae: the potential of domestication towards sustainable biofactories”. en. In: *Microbial Cell Factories* 17.1 (Nov. 2018), p. 173. ISSN: 1475-2859. DOI: 10.1186/s12934-018-1019-3.

[4] Teppo Felin et al. “The data-hypothesis relationship”. In: *Genome Biology* 22.1 (2021), p. 57. DOI: 10.1186/s13059-021-02276-4.

[5] John T. Jones et al. “Top 10 plant-parasitic nematodes in molecular plant pathology”. In: *Molecular Plant Pathology* 14.9 (2013), pp. 946–961. DOI: https://doi.org/10.1111/mpp.12057. eprint: https://bsppjournals.onlinelibrary.wiley.com/doi/pdf/10.1111/mpp.12057. URL: https://bsppjournals.onlinelibrary.wiley.com/doi/abs/10.1111/mpp.12057.

[6] Arthur Korte and Ashley Farlow. “The advantages and limitations of trait analysis with GWAS: a review”. In: *Plant methods* 9.1 (2013), p. 29.

[7] Gregor Mendel. “Experiments in Plant Hybridization (English translation)”. English. In: *Verhandlungen des Naturforschenden Vereines in Brünn* 4 (1866). Translation based on Druery and Bateson (1901), pp. 3–47.

[8] A.S. Schaveling et al. “Globodera pallida virulence on major potato resistance has a common genetic basis across Western Europe”. In: *bioRxiv* (2025). DOI: 10.64898/2025.12.22.695896. eprint: https://www.biorxiv.org/content/early/2025/12/23/2025.12.22.695896.full.pdf. URL: https://www.biorxiv.org/content/early/2025/12/23/2025.12.22.695896.

[9] A.S. Schaveling et al. “The potato cyst nematode Globodera pallida overcomes major potato resistance through selection on standing variation at a single locus”. In: *New Phytologist* n/a.n/a (). DOI: https://doi.org/10.1111/nph.70886. eprint: https://nph.onlinelibrary.wiley.com/doi/pdf/10.1111/nph.70886. URL: https://nph.onlinelibrary.wiley.com/doi/abs/10.1111/nph.70886.

[10] Bart-Jan van Rossum and Willem Kruijer. *statgenGWAS: Genome Wide Association Studies*. R package version 1.0.12. 2025. URL: https://biometris.github.io/statgenGWAS/index.html.

[11] S. Warmerdam et al. “Genome-wide association mapping of the architecture of susceptibility to the root-knot nematode Meloidogyne incognita in Arabidopsis thaliana”. In: *New Phytol* 218.2 (2018). Warmerdam, Sonja Sterken, Mark G van Schaik, Casper Oortwijn, Marian E P Sukarta, Octavina C A Lozano-Torres, Jose L Dicke, Marcel Helder, Johannes Kammenga, Jan E Goverse, Aska Bakker, Jaap Smant, Geert eng Research Support, Non-U.S. Gov’t England 2018/02/23 New Phytol. 2018 Apr;218(2):724-737. doi: 10.1111/nph.15034. Epub 2018 Feb 22., pp. 724–737. ISSN: 1469-8137 (Electronic) 0028-646X (Print) 0028-646X (Linking). DOI: 10.1111/nph.15034. URL: https://www.ncbi.nlm.nih.gov/pubmed/29468687.

[12] Hadley Wickham. *Advanced R*. Chapman and Hall/CRC, May 2019. ISBN: 9781351201315. DOI: 10.1201/9781351201315.

[13] J. Willig et al. “From root to shoot; Quantifying nematode tolerance in Arabidopsis thaliana by high-throughput phenotyping of plant development”. In: *Journal of Experimental Botany* (July 2023), erad266. ISSN: 0022-0957. DOI: 10.1093/jxb/erad266. eprint: https://academic.oup.com/jxb/advance-article-pdf/doi/10.1093/jxb/erad266/50854660/erad266.pdf. URL: https://doi.org/10.1093/jxb/erad266.

[14] Itai Yanai and Martin Lercher. “A hypothesis is a liability”. In: *Genome biology* 21.1 (2020), p. 231. DOI: 10.1186/s13059-020-02133-w.

[15] Itai Yanai and Martin Lercher. “Night science”. In: *Genome Biology* 20.1 (2019), p. 179. DOI: 10.1186/s13059-019-1800-6.