

On the R-way to hell

Statistical analysis in the marvelous world of R

Julien Martin



*Do what you think is interesting,
do something that you think is fun and worthwhile,
because otherwise you won't do it well anyway.*

—Brian W. Kernighan

Table of contents

Preface	ii
The aim of this book	ii
Multilingual book	ii
How to use this book	iii
Who are we ?	iv
Thanks	iv
Image credits	v
License	v
Citing the book	vi
Course associated reading	vii
Hex Sticker	ix

I. Using R	1
1. Getting started	2
Some R pointers	3
1.1. Installation	3
1.1.1. Installing R	3
1.1.2. Installing an IDE	4
1.2. IDE orientation	6
1.2.1. RStudio	6
1.2.2. VSCode	10
1.3. Working directories	12
1.4. Directory structure	14
1.5. Projects organisation	16
1.5.1. RStudio	16
1.5.2. VSCode	19
1.6. File names	20
1.7. Script documentation	21
1.8. R style guide	23
1.9. Backing up projects	24
1.10. Citing R and R packages	25
2. Some R basics	27
2.1. Important considerations	27
2.2. First step in the console	28
2.3. Objects in R	29
2.3.1. Creating objects	30
2.3.2. Naming objects	33
2.4. Using functions in R	33
2.5. Working with vectors	37
2.5.1. Extracting elements	37

2.5.2. Replacing elements	40
2.5.3. Ordering elements	41
2.5.4. Vectorisation	43
2.5.5. Missing data	44
2.6. Getting help	46
2.6.1. R help	46
2.6.2. Other sources of help	48
2.7. Saving stuff in R	50
2.8. R packages	51
2.8.1. Using packages	51
2.8.2. Installing R packages	52
3. Data	55
3.1. Data types	55
3.2. Data structures	58
3.2.1. Scalars and vectors	58
3.2.2. Matrices and arrays	58
3.2.3. Lists	63
3.2.4. Data frames	65
3.3. Importing data	68
3.3.1. Saving files to import	68
3.3.2. Import functions	69
3.3.3. Common import frustrations	72
3.3.4. Other import options	73
3.4. Wrangling data frames	75
3.4.1. Positional indexes	77
3.4.2. Logical indexes	80
3.4.3. Ordering data frames	85
3.4.4. Adding columns and rows	90
3.4.5. Merging data frames	94
3.4.6. Reshaping data frames	97
3.5. Introduction to the <code>tidyverse</code>	101
3.6. Summarising data frames	101
3.7. Exporting data	109
3.7.1. Export functions	109
3.7.2. Other export functions	111
4. Figures	112
4.1. Simple base R plots	114
4.2. <code>ggplot2</code>	117
4.3. Simple plots	118
4.3.1. Scatterplots	118
4.3.2. Histograms	119
4.3.3. Box plots	124
4.3.4. Violin plots	129
4.3.5. Dot charts	130
4.3.6. Pairs plots	133
4.3.7. Coplots	135
4.3.8. Summary of plot function	138

4.4.	Multiple graphs	139
4.4.1.	Base R	139
4.4.2.	ggplot	140
4.5.	Customising ggplots	142
4.6.	Exporting plots	142
5.	Programming	145
5.1.	Looking behind the curtain	145
5.2.	Functions in R	148
5.3.	Conditional statements	154
5.4.	Combining logical operators	158
5.5.	Loops	160
5.5.1.	For loop	160
5.5.2.	While loop	165
5.5.3.	When to use a loop?	166
5.5.4.	If not loops, then what?	166
6.	Reproducible reports with Quarto	169
6.1.	What is R markdown / Quarto?	169
6.1.1.	R Markdown	169
6.1.2.	Quarto?	170
6.2.	Why use Quarto?	170
6.3.	Get started with Quarto	173
6.3.1.	Installation	173
6.3.2.	Create a Quarto document, .qmd	173
6.4.	Quarto document (.qmd) anatomy	177
6.4.1.	YAML header	177
6.4.2.	Formatted text	178
6.4.1.	Code chunks	184
6.4.2.	Inline R code	189
6.4.3.	Images and photos	190
6.4.4.	Figures	192
6.4.5.	Tables	196
6.4.6.	Cross-referencing	201
6.4.7.	Citations and bibliography	205
6.5.	Some tips and tricks	207
6.6.	Further Information	209
6.7.	Practical	209
6.7.1.	Context	209
6.7.2.	Questions	210
	Example of output	214
7.	Version control with Git and GitHub	216
7.1.	What is version control?	217
7.2.	Why use version control?	217
7.3.	What is Git and GitHub?	217
7.4.	Getting started	219
7.4.1.	Install Git	219
7.4.2.	Configure Git	220
7.4.3.	Configure RStudio	221

7.4.4. Configure VSCode	221
7.4.5. Register a GitHub account	221
7.5. Setting up a project	222
7.5.1. in RStudio	222
7.5.2. Option 1 - GitHub first	223
7.5.3. Option 2 - RStudio first	225
7.5.4. in VSCode	234
7.6. Using Git with RStudio	235
7.6.1. Tracking changes	239
7.6.2. Commit history	243
7.6.3. Reverting changes	246
7.7. Using Git with VSCode	254
Tracking changes	255
Commit History	255
Reverting changes	255
7.8. Collaborate with Git	255
7.9. Git tips	255
7.10. Further resources	256
7.11. Practical	257
7.11.1. Context	257
7.11.2. Information of the data	257
7.11.3. Questions	259
7.11.4. Solution	260
II. Fundamentals of stats	262
8. Power Analysis	263
8.1. The theory	263
8.1.1. What is power?	263
8.1.2. Why do a power analysis?	263
8.1.3. Factors affecting power	264
8.1.4. Types of power analyses	264
8.1.5. How to calculate effect size	265
8.2. Practical	266
8.2.1. What is G*Power?	266
8.2.2. How to use G*Power	267
8.2.3. Power analysis for a t-test on two independent means	267
8.2.4. Post-hoc analysis	268
8.2.5. A priori analysis	275
8.2.6. Sensitivity analysis - Calculate the detectable effect size	278
8.3. Important points to remember	280
III. Linear models	281
9. Correlation and simple linear regression	282
9.1. R packages and data	282
9.2. Scatter plots	283
9.3. Data transformations and the product-moment correlation	286

9.4. Testing the significance of correlations and Bonferroni probabilities	289
9.5. Non-parametric correlations: Spearman's rank and Kendall's τ	290
9.6. Simple linear regression	292
9.6.1. Testing regression assumptions	295
9.6.2. Formal tests of regression assumptions	298
9.7. Data transformations in regression	300
9.8. Dealing with outliers	304
9.9. Quantifying effect size in regression and power analysis	307
9.9.1. Power to detect a given slope	308
9.9.2. Sample size required to achieve desired power	311
9.10. Bootstrapping the simple linear regression	313
10. Two - sample comparisons	318
10.1. R packages and data	318
10.2. Visual examination of sample data	319
10.3. Comparing means of two independent samples: parametric and non-parametric comparisons	322
10.4. Bootstrap and permutation tests to compare 2 means	326
10.4.1. Bootstrap	326
10.4.2. Permutation	329
10.5. Comparing the means of paired samples	330
10.6. Bibliography	336
11. One-way ANOVA	337
11.1. R packages and data	337
11.2. One-way ANOVA with multiple comparisons	338
11.2.1. Visualiser les données	338
11.2.2. Testing the assumptions of a parametric ANOVA	342
11.2.3. Performing the ANOVA	344
11.2.4. Performing multiple comparisons of means test	345
11.3. Data transformations and non-parametric ANOVA	352
11.4. Dealing with outliers	354
11.5. Permutation test	355
12. Multiway ANOVA: factorial and nested designs	358
12.1. R packages and data needed	358
12.2. Two-way factorial design with replication	359
12.2.1. Fixed effects ANOVA (Model I)	359
12.2.2. Mixed effects ANOVA (Model III)	368
12.3. 2-way factorial ANOVA without replication	369
12.4. Nested designs	373
12.5. Two-way non-parametric ANOVA	377
12.6. Multiple comparisons	380
12.7. Test de permutation pour l'ANOVA à deux facteurs de classification	386
12.8. Bootstrap for two-way ANOVA	389
13. Multiple regression	391
13.1. R packages and data	391
13.2. Points to keep in mind	392
13.3. First look at the data	392
13.4. Multiple regression models from scratch	393
13.5. Stepwise multiple regression procedures	401

13.6. Detecting multicollinearity	404
13.7. Polynomial regression	405
13.8. Checking assumptions of a multiple regression model	412
13.9. Visualizing effect size	414
13.10. Testing for interactions	417
13.11. Dredging and the information theoretical approach	422
13.12. Bootstrapping multiple regression	425
13.13. Permutation test	430
14. ANCOVA and general linear model	431
14.1. R packages and data	431
14.2. Linear models	432
14.3. ANCOVA	432
14.4. Homogeneity of slopes	433
14.4.1. Case 1 - Size as a function of age (equal slopes example)	433
14.4.2. Case 2 - Size as a function of age (different slopes example)	438
14.5. The ANCOVA model	442
14.6. Comparing model fits	449
14.7. Bootstrap	450
14.8. Permutation test	452
IV. Generalized linear models	454
15. Generalized linear model, <code>glm</code>	455
15.1. Lecture	455
15.1.1. Distributions	455
15.2. Practical	456
15.2.1. Logistic regression	456
15.2.2. Poisson regression	464
16. Frequency data and Poisson Regression	472
16.1. R packages and data	472
16.2. Organizing the data: 3 forms	473
16.3. Graphs for contingency tables and testing for independence	476
16.4. Log-linear models as an alternative to Chi-square test for contingency tables	480
16.5. Testing an external hypothesis	482
16.6. Poisson regression to analyze multi-way tables	484
16.7. Exercice	487
V. Mixed models	494
17. Introduction to linear mixed models	495
17.1. Lecture	495
17.1.1. Testing fixed effects	495
17.1.2. Shrinkage	495
17.2. Practical	500
17.2.1. Overview	500
17.2.2. R packages needed	500
17.2.3. The superb wild unicorns of the Scottish Highlands	501

17.2.4. Do unicorns differ in aggressiveness? Your first mixed model	504
17.2.5. Do unicorns differ in aggressiveness? A better mixed model	506
17.2.6. What is the repeatability?	516
17.2.7. A quick note on uncertainty	519
17.2.8. An easy way to mess up your mixed models	520
17.2.9. Happy mixed-modelling	523
18. Introduction to GLMM	524
18.1. Lecture	524
18.2. Practical	524
18.2.1. Packages and functions	525
18.2.2. The data set	525
18.2.3. Specifying fixed and random Effects	526
18.2.4. Look at overall patterns in data	529
18.2.5. Choose an error distribution	531
18.2.6. Fitting group-wise GLM	540
18.2.7. Fitting and evaluating GLMMs	542
18.2.8. Inference	550
18.2.9. Conclusions	557
18.2.10. Happy generalized mixed-modelling	558
19. Random regression and character state approaches	559
19.1. Lecture	559
19.2. Practical	559
19.2.1. R packages needed	560
19.2.2. Refresher on unicorn aggression	560
19.2.3. Random regression	563
19.2.4. Character-State approach	575
19.2.5. From random regression to character-state	586
19.2.6. Conclusions	588
19.2.7. Happy multivariate models	588
20. Multivariate mixed models	589
20.1. Lecture	589
20.2. Practical	589
20.2.1. R packages needed	590
20.2.2. The blue dragon of the East	590
20.2.3. Multiple univariate models	592
20.2.4. Multivariate approach	600
20.2.5. Happy multivariate models	615
VI. Generalized additive models	616
VII. Multivariate analysis	617
VIII Bayesian approach	618
21. Beyond $P < 0.05$	619

22. Introduction to Bayesian Inference	620
22.1. Lecture	620
22.1.1. Bayes' theorem	620
22.1.2. Intro to MCMC	622
22.1.3. Inferences	627
22.2. Practical	628
22.2.1. R packages needed	628
22.2.2. A refresher on unicorn ecology	628
22.2.3. MCMCglmm	630
22.2.4. Inferences	639
22.2.5. brms	640
22.2.6. Inferences	645
22.2.7. Happy Bayesian stats	649
References	650
R packages	650
Bibliography	652
Appendices	657
A. Data used in this book	657
A.1. All in one zip file	657
A.2. All the data files	657
A.3. R code used in the book and slides	658
B. Installing Quarto and LateX	659
B.1. MS Windows	660
B.2. Mac OSX	660
B.3. Linux	660

Preface

⚠ Warning

Book in development. Several sections not developed yet

The aim of this book

The aim of this book is two-fold:

1. introduce you to R, a powerful and flexible interactive environment for statistical computing and research.
2. introduce you to (or reacquaint you with) statistical analysis done in R.

R in itself is not difficult to learn, but as with learning any new language (spoken or computer) the initial learning curve can be steep and somewhat daunting. It is not intended to cover everything (neither with R nor with statistics) but simply to help you climb the initial learning curve (potentially faster) and provide you with the basic skills (and confidence!) needed to start your own journey with R and with specific analysis.

Multilingual book

The book is provided as a multilingual book breaking that language barrier and potentially allow to facilitate the learn of R and its mainly english-speaking environment. We are always looking for volunteers to help developed the book further and add more languages to the growing list. Please [contact us](#) if you want to help

On the web version of the book, use  in the navigation bar to switch from one language to another. After switching to your preferred language, you can of course also download the pdf and epub versions in this language if you want to using .

List of languages:

- english (work in progress publish but need polishing)
- french (in development, waiting for english to be polished)
- spanish (one day maybe)
- volunteers for more ??

How to use this book

For the best experience we recommend that you read the web version of this book which you can find <https://biostats-uottawa.github.io/Rway>.

The web version includes a navbar at the top of the page where you can toggle the sidebars on and off , search through the book , change the page color  and suggest revisions if you spot a typo or mistake . You can also download  a pdf and epub versions of the book.

We use a few typographical conventions throughout this book.

R code and the resulting output are presented in code blocks in our book.

```
2 + 2
```

```
[1] 4
```

Functions in the text are shown with brackets at the end using code font, i.e. `mean()` or `sd()` etc.

Objects are shown using code font without the round brackets, i.e. `obj1`, `obj2` etc.

R packages in the text are shown using code font and followed by the  icon, i.e. `tidyverse` .

A series of actions required to access menu commands in RStudio or VSCode are identified as `File -> New File -> R Script` which translates to ‘click on the File menu, then click on New File and then select R Script’.

When we refer to **IDE** (**I**ntegrated **D**evelopment **E**nvironment software) later in the text we mean either RStudio or VScode.

When we refer to **.[Rq]md**, we mean either R markdown (.Rmd) or Quarto (.qmd) documents and would generally talk of R markdown documents when referring to either .Rmd or .qmd files.

The manual tries to highlight some part of the text using the following boxes and icons.

 Exercises

Stuff for you to do

 Solutions

R code and explanations

 Warning

warnings

 Important

important points

 Note

notes

Who are we ?



Julien Martin is a Professor at the University of Ottawa working on Evolutionary Ecology and has discovered R with version 1.8.1 and teaches R since v2.4.0.

-  uOttawa <https://www.uottawa.ca/faculty-science/professors/juliengamartin/>, lab page <https://juliengamartin.github.io>
-  <https://twitter.com/jgamartin>
-  <https://github.com/juliengamartin>

Thanks

The first part of the book on using R started as a fork on github from the excellent [An introduction to R](#) book by Douglas, Roos, Mancini, Couto and Lusseau (Douglas 2023). It was forked on April 23rd, 2023 from [Alexd106](#) [github repository](#) then modified and updated following my own needs and teaching perspective of R. The content was neither reviewed nor endorsed by any the previous developers.

Several parts in the book were based on previous lab manuals for biostatistics classes at uOttawa written by Martin, Findlay, Morin and Rundle.

Site that provided a lot of information for the book:

- [dplyr introduction](#)
- [Introduction to gam](#)
- [Intoduction to gams by Noam Ross](#)

Image credits

Photos, images and screenshots are from Julien Martin except when indicated in caption.

Cover image was generated via [Nightcafe Ai Art generator](#). Favicon and hex sticker were created from the cover image.

i Note

several screenshot are currently by Alex Douglas and are being redone to abide by the previous statement

License

I share this work under the license [License Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International](#).



Figure 1.: License Creative Commons

If you teach R, feel free to use some or all of the content in this book to help support your own students. The only thing I ask is that you acknowledge the original source and authors. If you find this book useful or have any comments or suggestions I would love to hear from you (contact info).

Citing the book

Julien Martin. (2024). On the R-way to hell. A multilingual introduction to R book. Version: 0.6.0 (2024-10-11).DOI:
[10.5281/zenodo.13801263](https://doi.org/10.5281/zenodo.13801263)

Course associated reading

Table 1.: Course associated reading for biostatistical course at uOttawa

	BioXx58	Bio8940
Chapter		
Using R		
1.-4.	✓✓	😊
5. Programming		✓✓
6. Reproducible reports	✓	✓✓
7. Version control		✓✓
Stats fundamentals		
all chapters	✓✓	😊
Linear models		
all chapters	✓✓	😊
Generalized linear models		
all chapters	✓	✓✓
Mixed models		
all chapters		✓✓
Generalized additive models		
all chapters		✓
Multivariate analysis		
all chapters		✓
Bayesian approach		
all chapters		✓✓

Suggested ✓ ; mandatory ✓✓ ; expected knowledge (might need a refresher) 😊

Hex Sticker



Part I.

Using R

Chapter 1

Getting started

Although R is not new, its popularity has increased rapidly over the last 10 years or so (see [here](#)). It was originally created and developed by Ross Ihaka and Robert Gentleman during the 1990's with the first stable version released in 2000. Nowadays R is maintained by the [R Development Core Team](#). So, why has R become so popular and why should you learn how to use it? Some reasons include:

- open source and freely available.
- available for Windows, Mac and Linux operating systems.
- extensive and coherent set of tools for statistical analysis.
- extensive and highly flexible graphical facility capable of producing publication quality figures.
- expanding set of freely available ‘packages’  to extend R’s capabilities.
- extensive support network with numerous online and freely available documents.

All of the reasons above are great reasons to use R. However, in our opinion, the biggest reason to use R is that it facilitates robust and reproducible research practices. In contrast to more traditional ‘point and click’ software, writing code ensures you have a permanent and accurate record of all the methods you used (and decisions you made) for your data analysis. You are then able to share this code (and your data) with other researchers / colleagues / reviewers who will be able to reproduce your analysis exactly. This is one of the tenets of [open science](#). We will cover other topics to facilitate open science throughout this book, including creating reproducible reports (Chapter 6) and version control (Chapter 7).

In this Chapter we’ll cover:

- how to download and install R and an IDE on your computer
- give you a brief orientation of the 2 most common IDEs used with R
- some good habits to get into when working on projects

- and finally some advice on documenting your workflow and writing nice readable R code.

Some R pointers

- Use R often and use it regularly. This will help build and maintain all important momentum.
- Learning R is not a memory test. One of advantage of a scripting language is that you will always have your (well annotated) code to refer back to when you forget how to do something.
- You don't need to know everything about R to be productive.
- If you get stuck, search online, it's not cheating and writing a good search query is a skill in itself.
- If you find yourself staring at code for hours trying to figure out why it's not working then walk away for a few minutes.
- In R there are many ways to tackle a particular problem. If your code does what you want it to do in a reasonable time and robustly then don't worry about it.
- R is just a tool to help you answer your interesting questions. Don't lose sight of what's important - your research question(s) and your data. No amount of skill using R will help if your data collection is fundamentally flawed or your question vague.
- Recognize that there will be times when things will get a little tough or frustrating. Try to accept these periods as part of the natural process of learning a new skill (we've all been there) and remember, the time and energy you invest now will be more than payed back in the not too distant future.

Good luck and don't forget to have fun.

1.1. Installation

1.1.1. Installing R

To get up and running the first thing you need to do is install R. R is freely available for Windows, Mac and Linux operating systems from the [Comprehensive R Archive Network \(CRAN\) website](#). For Windows and Mac users we suggest you download and install the pre-compiled binary versions. There are reasonably comprehensive instruction to install R for each OS ([Windows](#),[Mac](#) or [linux](#)).

Whichever operating system you're using, once you have installed R you need to check its working properly. The easiest way to do this is to start R by double clicking on the R icon (Windows or Mac) or by typing R into the Console

(Linux). You should see the R Console and you should be able to type R commands into the Console after the command prompt >. Try typing the following R code and then press enter:

```
plot(1)
```

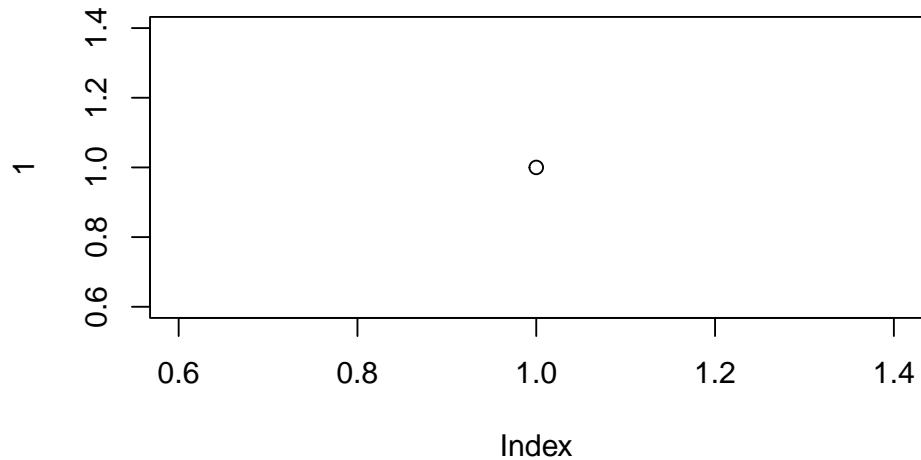


Figure 1.1.: Most amazing plot, just useful to test R

A plot with a single point in the center should appear. If you see this, you're good to go. If not then we suggest you make a note of any errors produced and then use your preferred search engine to troubleshoot.

1.1.2. Installing an IDE

We strongly recommend to use an Integrated Development Environment (IDE) software to work with R. One simple and extremely popular IDE is [RStudio](#). An alternative to RStudio is Visual Studio Code, or VSCode. An IDE can be thought of as an add-on to R which provides a more user-friendly interface, incorporating the R Console, a script editor and other useful functionality (like R markdown and Git Hub integration).

🔥 Caution

You must install R before you install an IDE (see Section 1.1.1 for details).

ℹ Note

When we refer to **IDE** later in the text we mean either RStudio or VScode

1.1.2.1. RStudio

RStudio is freely available for Windows, Mac and Linux operating systems and can be downloaded from the [RStudio site](#). You should select the ‘RStudio Desktop’ version.

1.1.2.2. VSCode

VSCode is freely available for Windows, Mac and Linux operating systems and can be downloaded from the [VS Code site](#).

In addition you need to install the [R extension to VSCode](#). To make VSCode a true powerhouse for working with R we strongly recommend you to also install:

- [radian](#): A modern R console that corrects many limitations of the official R terminal and supports many features such as syntax highlighting and auto-completion.
- [VSCode-R-Debugger](#): A VS Code extension to support R debugging capabilities.
- [httpgd](#): An R package  to provide a graphics device that asynchronously serves SVG graphics via HTTP and WebSockets.

1.1.2.3. Alternatives to RStudio and VSCode

Rather than using an ‘all in one’ IDE many people choose to use R and a separate script editor to write and execute R code. If you’re not familiar with what a script editor is, you can think of it as a bit like a word processor but specifically designed for writing code. Happily, there are many script editors freely available so feel free to download and experiment until you find one you like. Some script editors are only available for certain operating systems and not all are specific to R. Suggestions for script editors are provided below. Which one you choose is up to you: one of the great things about R is that *YOU* get to choose how you want to use R.

1.1.2.3.1. Advanced text editors A light yet efficient way to work with R is using advanced text editors such as:

- [Atom](#) (all operating systems)
- [BBedit](#) (Mac OS)
- [gedit](#) (Linux; comes with most Linux distributions)
- [MacVim](#) (Mac OS)
- [Nano](#) (Linux)

- [Notepad++](#) (Windows)
- [Sublime Text](#) (all operating systems)
- [vim](#) and its extension [NVim-R](#) (Linux)

1.1.2.3.2. Integrated development environments These environments are more powerful than simple text editors, and are similar to RStudio:

- [Emacs](#) and its extension [Emacs Speaks Statistics](#) (all operating systems)
- [RKWard](#) (Linux)
- [Tinn-R](#) (Windows)

1.2. IDE orientation

1.2.1. RStudio

When you open R studio for the first time you should see the following layout (it might look slightly different on a Windows computer).

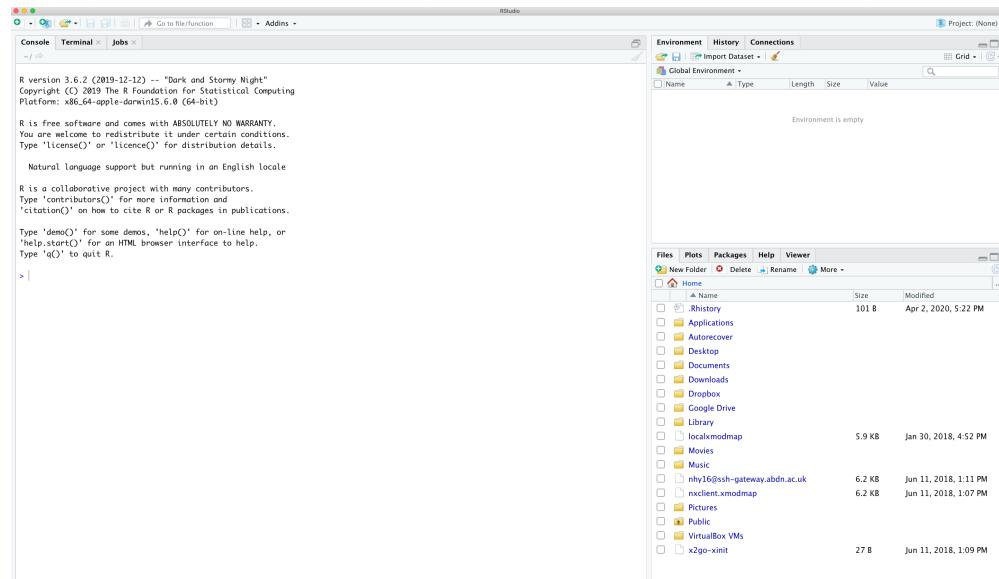


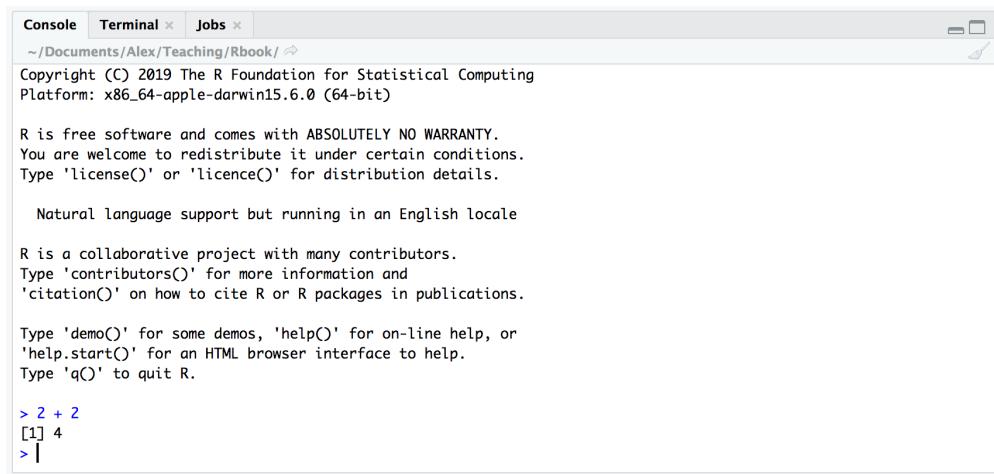
Figure 1.2.: R studio main window

The large window (aka pane) on the left is the **Console** window. The window on the top right is the **Environment / History / Connections** pane and the bottom right window is the **Files / Plots / Packages / Help / Viewer** window. We will discuss each of these panes in turn below. You can customise the location of each pane by clicking on the

'Tools' menu then selecting Global Options → Pane Layout. You can resize the panes by clicking and dragging the middle of the window borders in the direction you want. There are a plethora of other ways to [customise RStudio](#).

1.2.1.1. Console

The Console is the workhorse of R. This is where R evaluates all the code you write. You can type R code directly into the Console at the command line prompt, >. For example, if you type `2 + 2` into the Console you should obtain the answer `4` (reassuringly). Don't worry about the `[1]` at the start of the line for now.



The screenshot shows the R studio interface with the 'Console' tab selected. The window title is 'Console'. The content area displays the R startup message, which includes copyright information, a warning about no warranty, and instructions for redistribution. It also mentions natural language support and lists contributors. At the bottom of the message, it says 'Type 'q()'' to quit R. Below the message, there is a command line input field containing the text '`> 2 + 2`'. The output of the command, '`[1] 4`', is shown in blue text below the input. A cursor is visible at the end of the input line.

Figure 1.3.: R studio console view

However, once you start writing more R code this becomes rather cumbersome. Instead of typing R code directly into the Console a better approach is to create an R script. An R script is just a plain text file with a .R file extension which contains your lines of R code. These lines of code are then sourced into the R Console line by line. To create a new R script click on the 'File' menu then select New File → R Script.

Notice that you have a new window (called the Source pane) in the top left of RStudio and the Console is now in the bottom left position. The new window is a script editor and where you will write your code.

To source your code from your script editor to the Console simply place your cursor on the line of code and then click on the 'Run' button in the top right of the script editor pane.

You should see the result in the Console window. If clicking on the 'Run' button starts to become tiresome you can use the keyboard shortcut 'ctrl + enter' (on Windows and Linux) or 'cmd + enter' (on Mac). You can save your R scripts as a .R file by selecting the 'File' menu and clicking on save. Notice that the file name in the tab will turn red to remind you that you have unsaved changes. To open your R script in RStudio select the 'File' menu and then 'Open File...'. Finally, its worth noting that although R scripts are saved with a .R extension they are actually just plain text files which can be opened with any text editor.

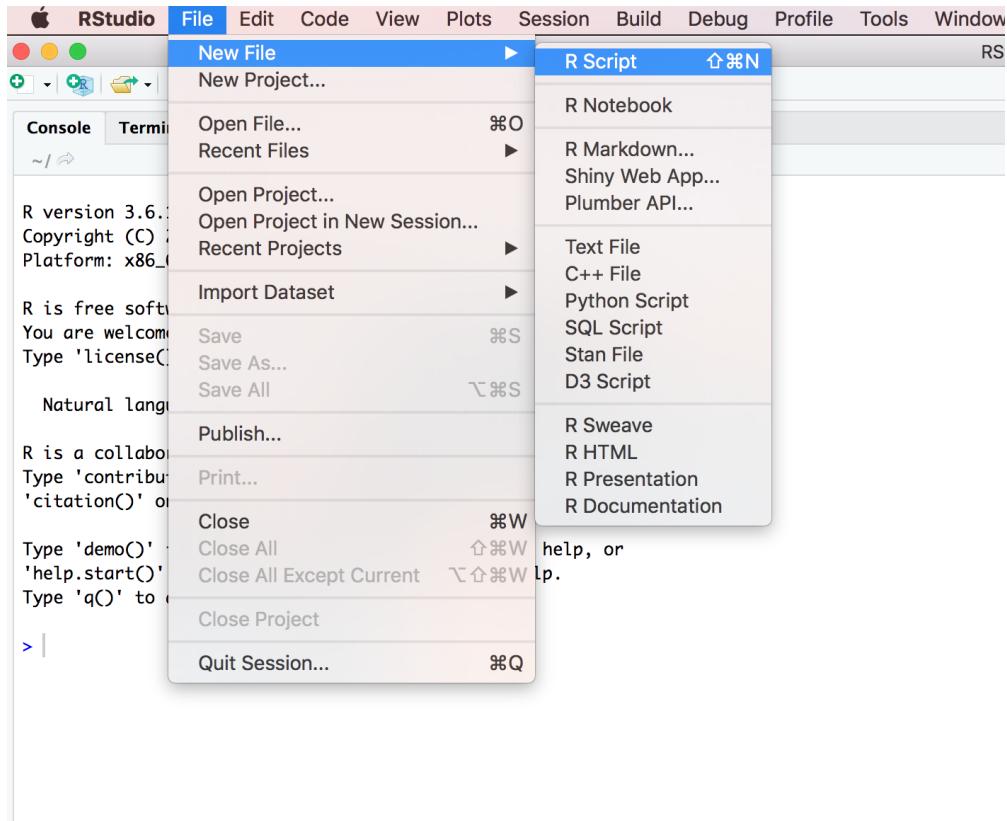


Figure 1.4.: R studio creating a new script file

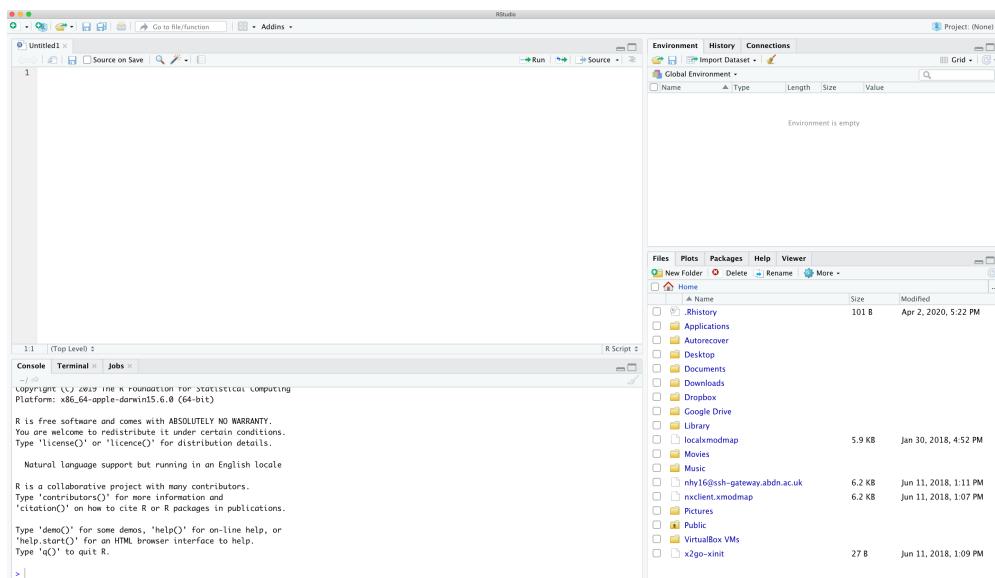


Figure 1.5.: R studio main view with a new script

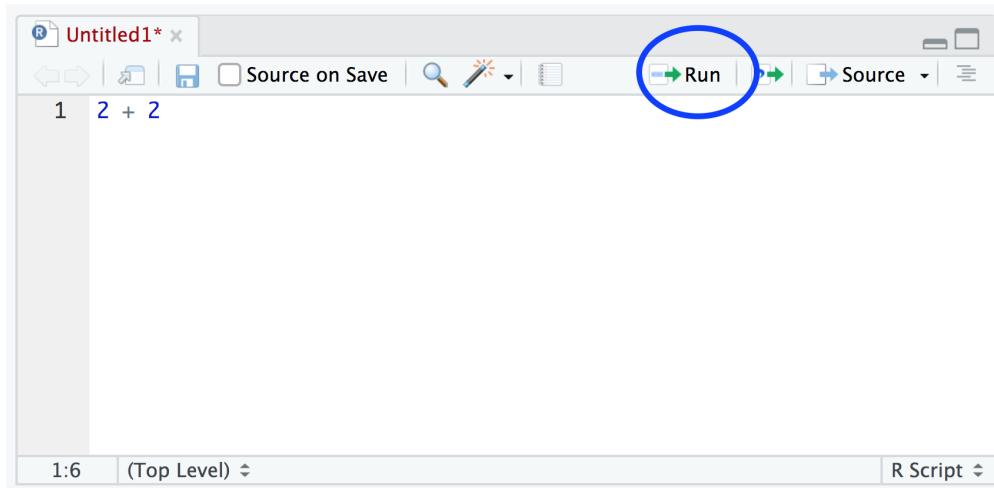


Figure 1.6.: R studio run button

1.2.1.2. Environment/History/Connections

The Environment / History / Connections window shows you lots of useful information. You can access each component by clicking on the appropriate tab in the pane.

- The ‘Environment’ tab displays all the objects you have created in the current (global) environment. These objects can be things like data you have imported or functions you have written. Objects can be displayed as a List or in Grid format by selecting your choice from the drop down button on the top right of the window. If you’re in the Grid format you can remove objects from the environment by placing a tick in the empty box next to the object name and then click on the broom icon. There’s also an ‘Import Dataset’ button which will import data saved in a variety of file formats. However, we would suggest that you don’t use this approach to import your data as it’s not reproducible and therefore not robust (see Chapter 3 for more details).
- The ‘History’ tab contains a list of all the commands you have entered into the R Console. You can search back through your history for the line of code you have forgotten, send selected code back to the Console or Source window. We usually never use this as we always refer back to our R script.
- The ‘Connections’ tab allows you to connect to various data sources such as external databases.

1.2.1.3. Files/Plots/Packages/Help/Viewer

- The ‘Files’ tab lists all external files and directories in the current working directory on your computer. It works like file explorer (Windows) or Finder (Mac). You can open, copy, rename, move and delete files listed in the window.

- The ‘Plots’ tab is where all the plots you create in R are displayed (unless you tell R otherwise). You can ‘zoom’ into the plots to make them larger using the magnifying glass button, and scroll back through previously created plots using the arrow buttons. There is also the option of exporting plots to an external file using the ‘Export’ drop down menu. Plots can be exported in various file formats such as jpeg, png, pdf, tiff or copied to the clipboard (although you are probably better off using the appropriate R functions to do this - see [Chapter 4 for more details]).
- The ‘Packages’ tab lists all of the packages that you have installed on your computer. You can also install new packages and update existing packages by clicking on the ‘Install’ and ‘Update’ buttons respectively.
- The ‘Help’ tab displays the R help documentation for any function. We will go over how to view the help files and how to search for help in Chapter 2).
- The ‘Viewer’ tab displays local web content such as web graphics generated by some packages.

1.2.2. VSCode

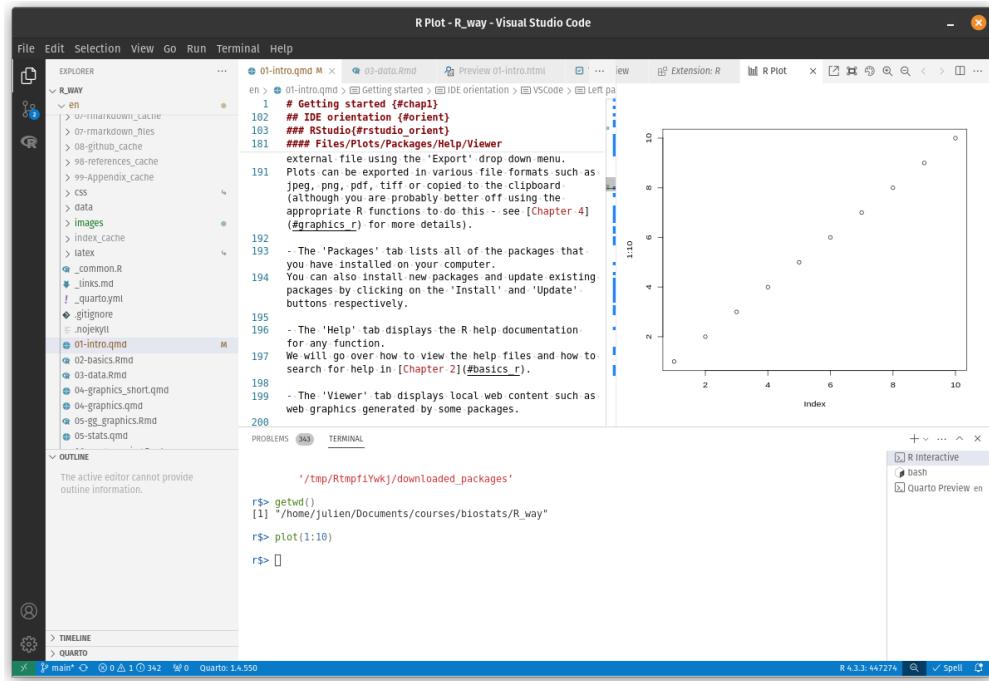


Figure 1.7.: VSCode window overview

1.2.2.1. Left panel

Contains :

- File manager and file outline
- R support including R environment/ R search / R help / install packages
- Github interaction

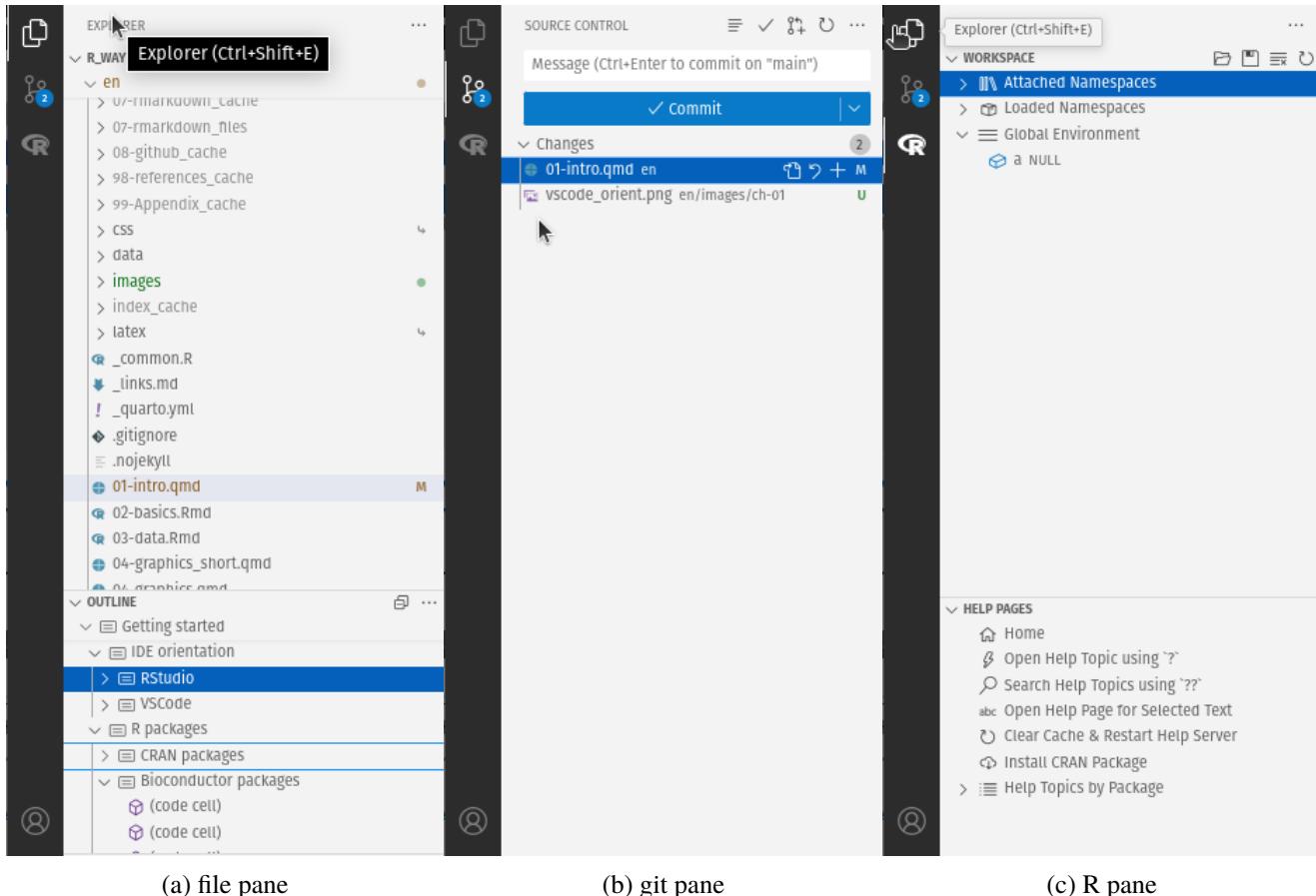


Figure 1.8.: VS Code left panel

1.2.2.2. Editor tabs

Includes:

- plot panel (with history and navigation)
- edition of scripts
- preview panels

1.2.2.3. Terminal window

Contains:

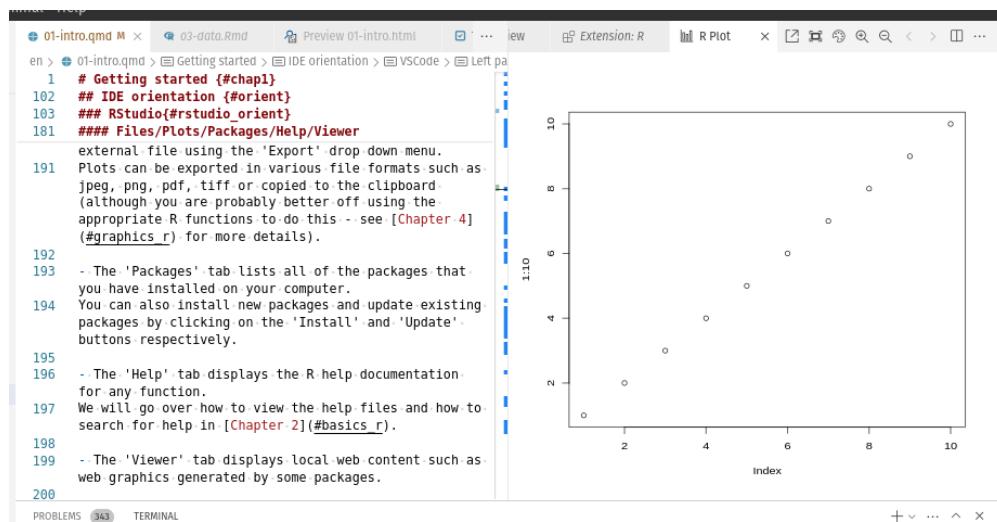


Figure 1.9.: VSCode editor tabs and preview panels

- the terminal allowing to have an R session or any other type of terminals needed (bash/tmux/). It can be split and run multiple sessions at the same time
- a problems pane highlighting both grammar and coding problems

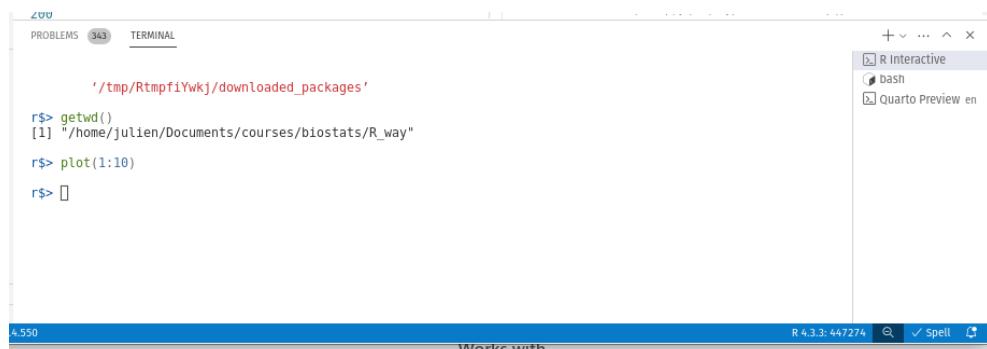


Figure 1.10.: VSCode terminal window

1.3. Working directories

The working directory is the default location where R will look for files you want to load and where it will put any files you save. One of the great things about using RStudio Projects is that when you open a project it will automatically set your working directory to the appropriate location. You can check the file path of your working directory by using either `getwd()` or `here()` functions.

```
getwd()
```

```
[1] "/home/julien/Documents/courses/biostats/livre/Rway"
```

In the example above, the working directory is a folder called ‘Rway’ which is a subfolder of “biostats” in the ‘courses’ folder which in turn is in a ‘Documents’ folder located in the ‘julien’ folder which itself is in the ‘home’ folder. On a Windows based computer our working directory would also include a drive letter (i.e. C:\home\julien\Documents\courses\biostats\Rway).

If you weren’t using an IDE then you would have to set your working directory using the `setwd()` function at the start of every R script (something we did for many years).

```
setwd("/home/julien/Documents/courses/biostats/Rway/")
```

However, the problem with `setwd()` is that it uses an *absolute* file path which is specific to the computer you are working on. If you want to send your script to someone else (or if you’re working on a different computer) this absolute file path is not going to work on your friend/colleagues computer as their directory configuration will be different (you are unlikely to have a directory structure /home/julien/Documents/courses/biostats/ on your computer). This results in a project that is not self-contained and not easily portable. IDEs solves this problem by allowing you to use *relative* file paths which are relative to the *Root* project directory. The Root project directory is just the directory that contains the `.Rproj` file in Rstudio (`first_project.Rproj` in our case) or the base folder of your workspace in VScode. If you want to share your analysis with someone else, all you need to do is copy the entire project directory and send to your collaborator. They would then just need to open the project file and any R scripts that contain references to relative file paths will just work. For example, let’s say that you’ve created a subdirectory called `data` in your Root project directory that contains a csv delimited datile called `mydata.csv` (we will cover directory structures below in Section 1.4). To import this datile in an RStudio project using the `read.csv()` function (don’t worry about this now, we will cover this in much more detail in Chapter 3) all you need to include in your R script is

```
dat <- read.csv("data/mydata.csv")
```

Because the file path `data/mydata.csv` is relative to the project directory it doesn’t matter where you collaborator saves the project directory on their computer it will still work.

If you weren’t using an RStudio project or VScode workspace then you would need to either set the working directory providing the full path to your directory or specify the full path of the data file. Neither option would be reproducible on other computers.

```
setwd("/home/julien/Documents/courses/biostats/Rway")  
  
dat <- read.csv("data/mydata.csv")
```

or

```
dat <- read.csv("/home/julien/Documents/courses/biostats/Rway/data/mydata.csv")
```

For those of you who want to take the notion of relative file paths a step further, take a look at the `here()` function in the [here package](#). The `here()` function allows you to build file paths for any file relative to the project root directory that are also operating system agnostic (works on a Mac, Windows or Linux machine). For example, to import our `mydata.csv` file from the `data` directory just use:

```
library(here) # you may need to install the here package first  
dat <- read.csv(here("data", "mydata.csv"))
```

1.4. Directory structure

In addition to using RStudio Projects, it's also really good practice to structure your working directory in a consistent and logical way to help both you and your collaborators. We frequently use the following directory structure in our R based projects.

In our working directory we have the following directories:

- **Root** - This is your project directory containing your `.Rproj` file. We tend to keep all the R scripts or [Rq]md document necessary for the analysis / report in this root folder or in the `scripts` folder when there are too many.
- **data** - We store all our data in this directory. The subdirectory called `data` contains raw data files and only raw data files. These files should be treated as **read only** and should not be changed in any way. If you need to process/clean/modify your data do this in R (not MS Excel) as you can document (and justify) any changes made. Any processed data should be saved to a separate file and stored in the `processed_data` subdirectory. Information about data collection methods, details of data download and any other useful metadata should be saved in a text document (see `README` text files below) in the `metadata` subdirectory.
- **functions** - This is an optional directory where we save all of the custom R functions we've written for the current analysis. These can then be sourced into R using the `source()` function.

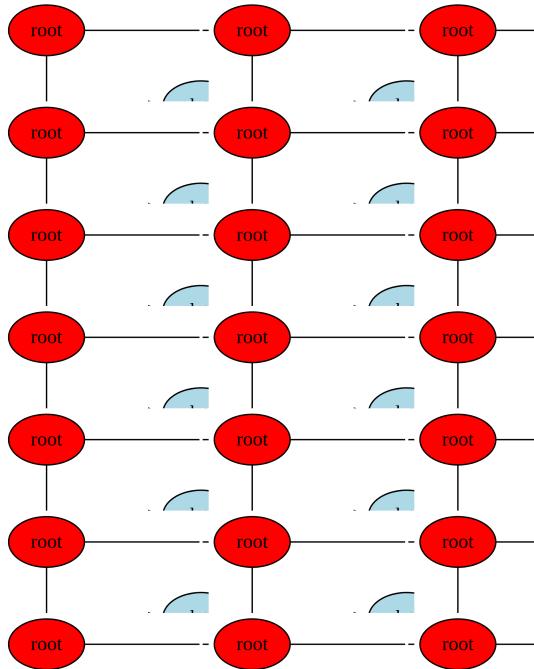


Figure 1.11.: Recommended directory structure for analysis with R

- **scripts** - An optional directory where we save our R markdown documents and/or the main R scripts we have written for the current project are saved here if not in the root folder.
- **output** - Outputs from our R scripts such as plots, HTML files and data summaries are saved in this directory. This helps us and our collaborators distinguish what files are outputs and which are source files.

Of course, the structure described above is just what works for us most of the time and should be viewed as a starting point for your own needs. We tend to have a fairly consistent directory structure across our projects as this allows us to quickly orientate ourselves when we return to a project after a while. Having said that, different projects will have different requirements so we happily add and remove directories as required.

You can create your directory structure using Windows Explorer (or Finder on a Mac) or within your IDE by clicking on the ‘New folder’ button in the ‘Files’ pane.

An alternative approach is to use the `dir.create()` functions in the R Console.

```
# create directory called 'data'
dir.create("data")
```

1.5. Projects organisation

As with most things in life, when it comes to dealing with data and data analysis things are so much simpler if you're organized. Clear project organisation makes it easier for both you (especially the future you) and your collaborators to make sense of what you've done. There's nothing more frustrating than coming back to a project months (sometimes years) later and have to spend days (or weeks) figuring out where everything is, what you did and why you did it. A well documented project that has a consistent and logical structure increases the likelihood that you can pick up where you left off with minimal fuss no matter how much time has passed. In addition, it's much easier to write code to automate tasks when files are well organized and are sensibly named. This is even more relevant nowadays as it's never been easier to collect vast amounts of data which can be saved across 1000's or even 100,000's of separate data files. Lastly, having a well organized project reduces the risk of introducing bugs or errors into your workflow and if they do occur (which inevitably they will at some point), it makes it easier to track down these errors and deal with them efficiently.

There are also a few simple steps you can take right at the start of any project to help keep things shipshape.

A great way of keeping things organized is to use RStudio Projects or VSCode workspaces, referred after as `project`. A `project` keeps all of your R scripts, R markdown documents, R functions and data together in one place. The nice thing about `project` is that each has its own directory, history and source documents so different analyses that you are working on are kept completely separate from each other. This means that you can very easily switch between `projects` without fear of them interfering with each other.

1.5.1. RStudio

To create a project, open RStudio and select `File -> New Project...` from the menu. You can create either an entirely new project, a project from an existing directory or a version controlled project (see the Chapter 7 for further details about this). In this Chapter we will create a project in a new directory.

You can also create a new project by clicking on the 'Project' button in the top right of RStudio and selecting 'New Project...'

In the next window select 'New Project'.

Now enter the name of the directory you want to create in the 'Directory name:' field (we'll call it `first_project` for this Chapter). If you want to change the location of the directory on your computer click the 'Browse...' button and navigate to where you would like to create the directory. We always tick the 'Open in new session' box as well. Finally, hit the 'Create Project' to create the new project.

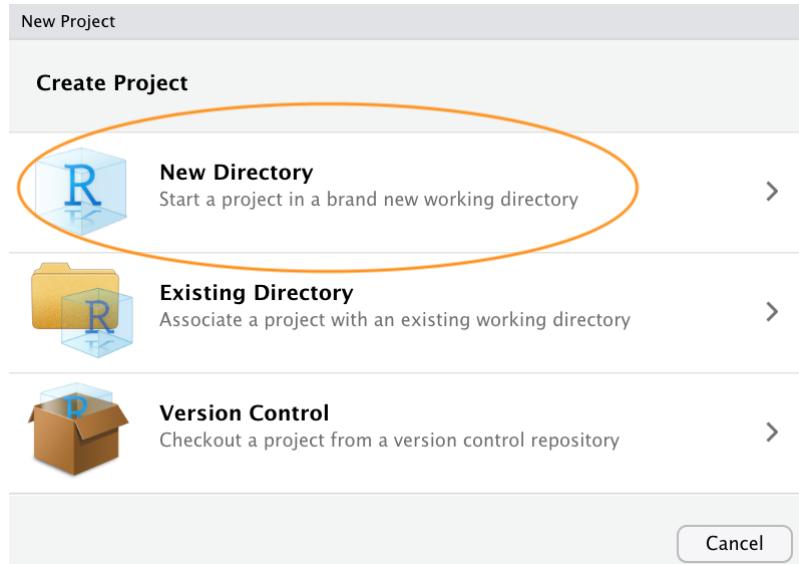


Figure 1.12.: R Studio creating a Project step 1

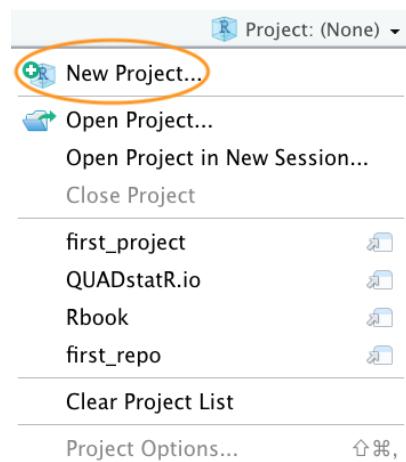


Figure 1.13.: R Studio creating a Project step 2

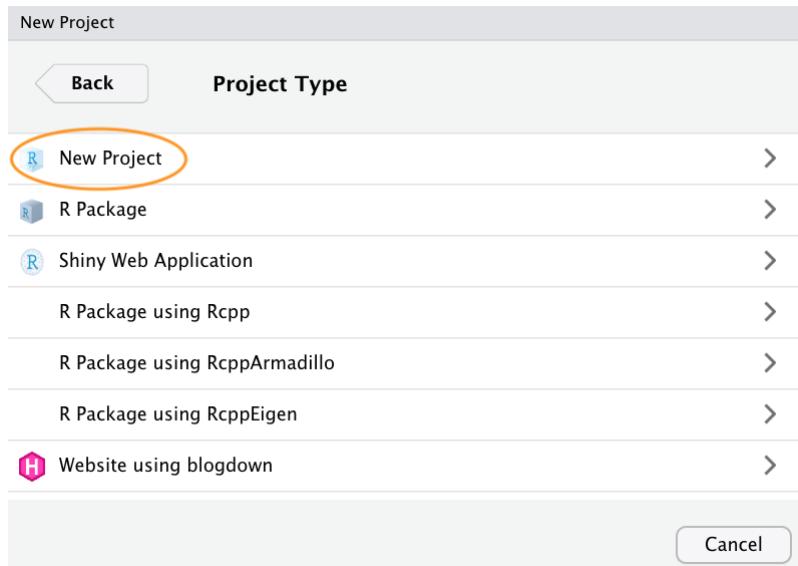


Figure 1.14.: R Studio creating a Project step 3

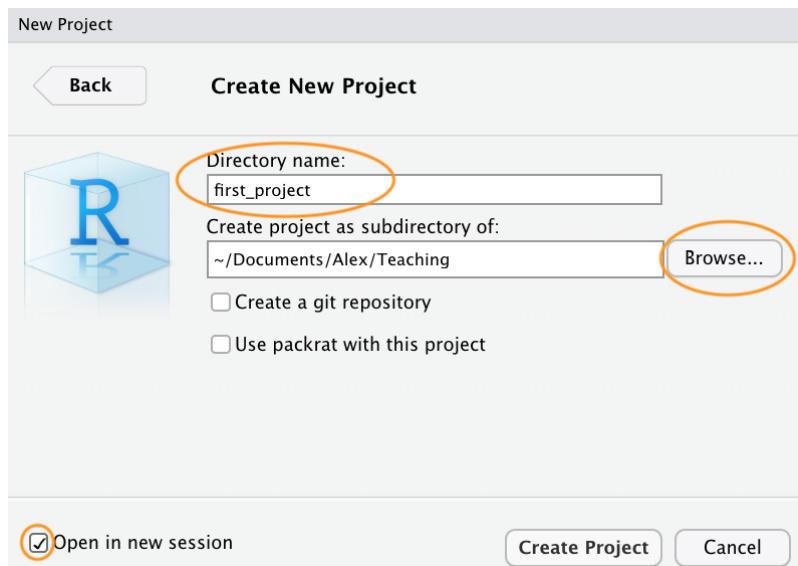


Figure 1.15.: R Studio creating a Project step 4

Once your new project has been created you will now have a new folder on your computer that contains an RStudio project file called `first_project.Rproj`. This `.Rproj` file contains various project options (but you shouldn't really interact with it) and can also be used as a shortcut for opening the project directly from the file system (just double click on it). You can check this out in the 'Files' tab in RStudio (or in Finder if you're on a Mac or File Explorer in Windows).

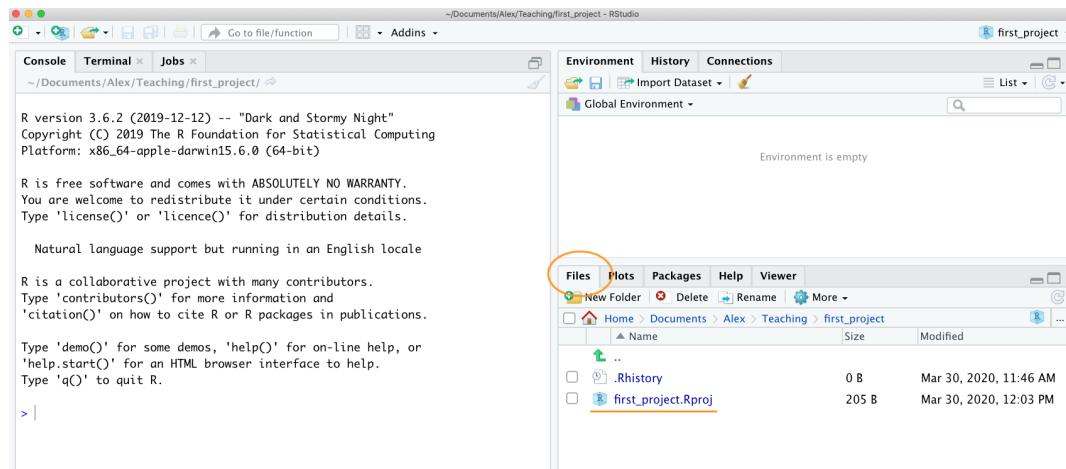


Figure 1.16.: R Studio creating a Project final step

The last thing we suggest you do is select `Tools -> Project Options...` from the menu. Click on the 'General' tab on the left hand side and then change the values for 'Restore .RData into workspace at startup' and 'Save workspace to .RData on exit' from 'Default' to 'No'. This ensures that every time you open your project you start with a clean R session. You don't have to do this (many people don't) but we prefer to start with a completely clean workspace whenever we open our projects to avoid any potential conflicts with things we have done in previous sessions (sometimes leading to surprising results and headaches figuring out the problem). The downside to this is that you will need to rerun your R code every time you open your project.

Now that you have an RStudio project set up you can start creating R scripts (or R markdown /Quarto documents, see Chapter 6) or whatever you need to complete you project. All of the R scripts will now be contained within the RStudio project and saved in the project folder.

1.5.2. VSCode

workspace are similar to RStudio projects. You however need to create a new folder with a R file (or text file) and save as workspace.

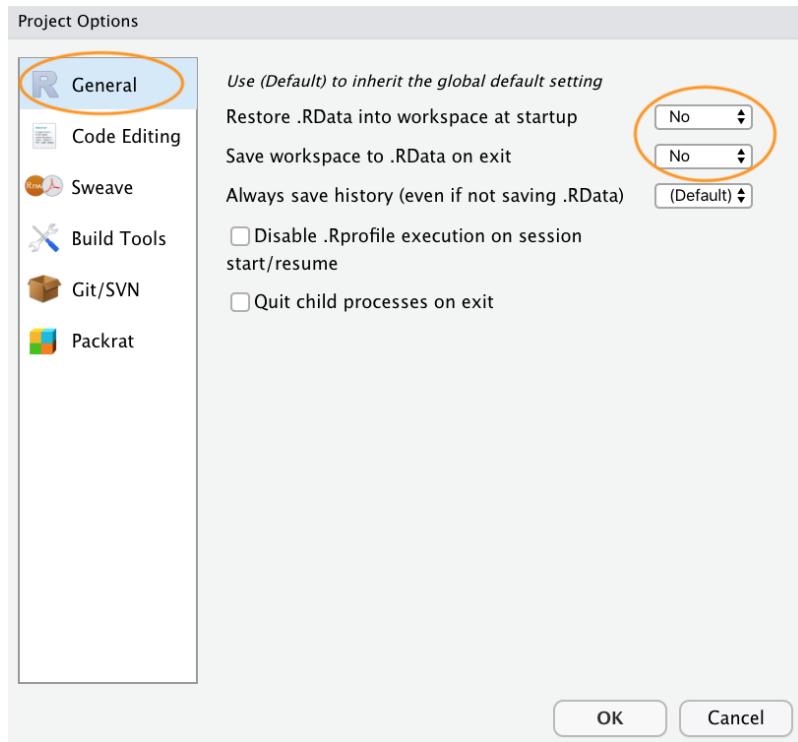


Figure 1.17.: R Studio creating a Project changing options

1.6. File names

What you name your files matters more than you might think. Naming files is also more difficult than you think. The key requirement for a ‘good’ file name is that it’s informative whilst also being relatively short. This is not always an easy compromise and often requires some thought. Ideally you should try to avoid the following!

Although there’s not really a recognized standard approach to naming files (actually [there is](#), just not everyone uses it), there are a couple of things to bear in mind.

- Avoid using spaces in file names by replacing them with underscores or hyphens. Why does this matter? One reason is that some command line software (especially many bioinformatic tools) won’t recognise a file name with a space and you’ll have to go through all sorts of shenanigans using escape characters to make sure spaces are handled correctly. Even if you don’t think you will ever use command line software you may be doing so indirectly. Take R markdown for example, if you want to render an R markdown document to pdf using the `rmarkdown` package you will actually be using a command line L^AT_EX engine under the hood. Another good reason not to use spaces in file names is that it makes searching for file names (or parts of file names) using [regular expressions](#) in R (or any other language) much more difficult.
- Avoid using special characters (i.e. @£\$%^&*(:/)) in your file names.

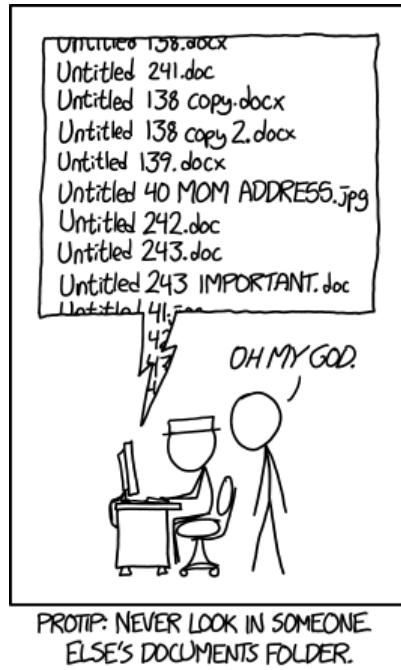


Figure 1.18.: File renaming song (source:<https://xkcd.com/1459/>)

- If you are versioning your files with sequential numbers (i.e. file1, file2, file3 ...). If you plan to have more than 9 files you should use 01, 02, 03, ..., 10 as this will ensure the files are listed in the correct order. If you plan to have more than 99 files then use 001, 002, 003, ...
- For dates, use the ISO 8601 format YYYY-MM-DD (or YYYYMMDD) to ensure your files are listed in proper chronological order.
- Never use the word *final* in any file name - it extremely rarely is!

Whatever file naming convention you decide to use, try to adopt early, stick with it and be consistent.

1.7. Script documentation

A quick note or two about writing R code and creating R scripts. Unless you're doing something really quick and dirty we suggest that you always write your R code as an R script. R scripts are what make R so useful. Not only do you have a complete record of your analysis, from data manipulation, visualisation and statistical analysis, you can also share this code (and data) with friends, colleagues and importantly when you submit and publish your research to a journal. With this in mind, make sure you include in your R script all the information required to make your work reproducible (author names, dates, sampling design etc). This information could be included as a series of comments # or, even better, by mixing executable code with narrative into an R markdown document (Chapter 6). It's also good

practice to include the output of the `sessionInfo()` function at the end of any script which prints the R version, details of the operating system and also loaded packages. A really good alternative is to use the `session_info()` function from the `xfun`  package for a more concise summary of our session environment.

Here's an example of including meta-information at the start of an R script

```
# Title: Time series analysis of lasagna consumption

# Purpose : This script performs a time series analyses on
#           lasagna meals kids want to have each week.
#           Data consists of counts of (dreamed) lasagna meals per week
#           collected from 24 kids at the "Food-dreaming" school
#           between 2042 and 2056.

# data file: lasagna_dreams.csv

# Author: A. Stomach
# Contact details: a.stomach@food.uni.com

# Date script created: Fri Mar 29 17:06:44 2010 -----
# Date script last modified: Thu Dec 12 16:07:12 2019 ----

# package dependencies
library(tidyverse)
library(ggplot2)

print("put your lovely R code here")

# good practice to include session information

xfun::session_info()
```

This is just one example and there are no hard and fast rules so feel free to develop a system that works for you. A really useful shortcut in RStudio is to automatically include a time and date stamp in your R script. To do this, write `ts` where you want to insert your time stamp in your R script and then press the ‘shift + tab’ keys. RStudio

will convert `ts` into the current date and time and also automatically comment out this line with a `#`. Another really useful RStudio shortcut is to comment out multiple lines in your script with a `#` symbol. To do this, highlight the lines of text you want to comment and then press ‘ctrl + shift + c’ (or ‘cmd + shift + c’ on a mac). To uncomment the lines just use ‘ctrl + shift + c’ again.

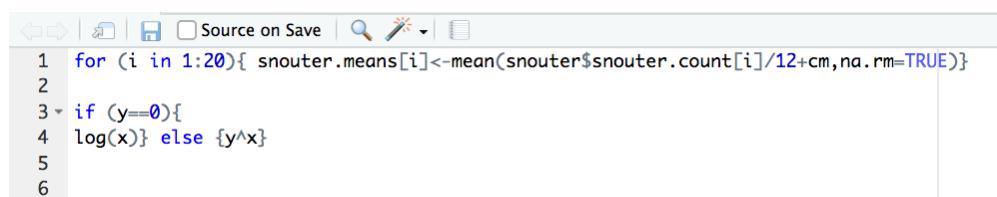
In addition to including metadata in your R scripts it’s also common practice to create a separate text file to record important information. By convention these text files are named `README`. We often include a `README` file in the directory where we keep our raw data. In this file we include details about when data were collected (or downloaded), how data were collected, information about specialised equipment, preservation methods, type and version of any machines used (i.e. sequencing equipment) etc. You can create a `README` file for your project in RStudio by clicking on the `File -> New File -> Text File` menu.

1.8. R style guide

How you write your code is more or less up to you although your goal should be to make it as easy to read as possible (for you and others). Whilst there are no rules (and no code police), we encourage you to get into the habit of writing readable R code by adopting a particular style. We suggest that you follow Google’s [R style guide](#) whenever possible. This style guide will help you decide where to use spaces, how to indent code and how to use square `[]` and curly `{ }` brackets amongst other things.

To help you with code formatting:

- VSCode there is an embedded formatter in the R extension for VSCode. You can just use the keyboard shortcut to reformat the code nicely and automatically.
- RStudio you can install the `styler` 📦 package which includes an RStudio add-in to allow you to automatically restyle selected code (or entire files and projects) with the click of your mouse. You can find more information about the `styler` 📦 package including how to install [here](#). Once installed, you can highlight the code you want to restyle, click on the ‘Addins’ button at the top of RStudio and select the ‘Style Selection’ option. Here is an example of poorly formatted R code



```

1 for (i in 1:20){ snouter.means[i]<-mean(snouter$snouter.count[i]/12+cm,na.rm=TRUE)}
2
3 if (y==0){
4 log(x)} else {y^x}
5
6

```

Figure 1.19.: Poorly styled code

Now highlight the code and use the styler 📦 package to reformat

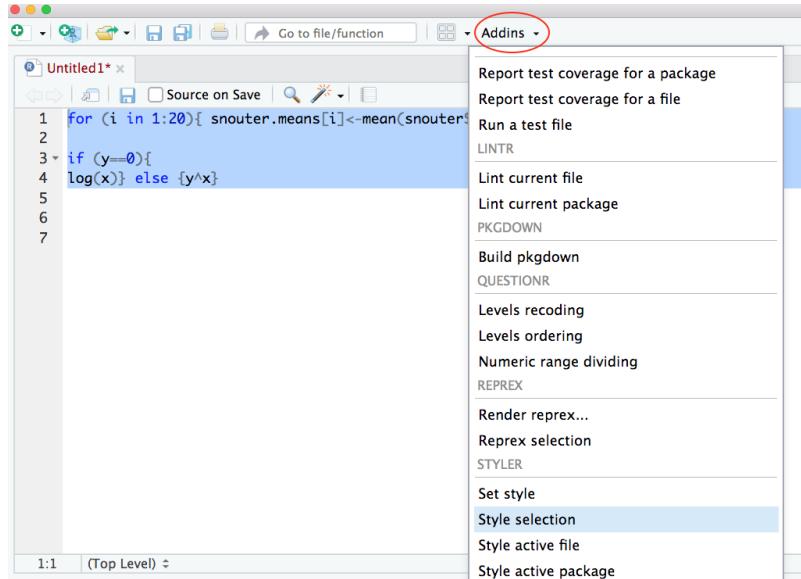


Figure 1.20.: Styling code with styler

To produce some nicely formatted code

The screenshot shows the RStudio interface with the 'Untitled1*' file open. The code has been styled, with improved indentation and spacing. The 'Source on Save' button is checked.

```

1 for (i in 1:20) {
2   snouter.means[i] <- mean(snouter$snouter$count[i] / 12 + cm, na.rm = TRUE)
3 }
4
5 if (y == 0) {
6   log(x)
7 } else {
8   y^x
9 }
10

```

Figure 1.21.: Nicely styled code

1.9. Backing up projects

Don't be that person who loses hard won (and often expensive) data and analyses. Don't be that person who thinks it'll never happen to me - it will! Always think of the absolute worst case scenario, something that makes you wake up in a cold sweat at night, and do all you can to make sure this never happens. Just to be clear, if you're relying on copying your precious files to an external hard disk or USB stick this is **NOT** an effective backup strategy. These things go wrong all the time as you lob them into your rucksack or 'bag for life' and then lug them between your office and home. Even if you do leave them plugged into your computer what happens when the building burns down (we did say worst case!)?

Ideally, your backups should be offsite and incremental. Happily there are numerous options for backing up your files. The first place to look is in your own institute. Most (all?) Universities have some form of network based storage that should be easily accessible and is also underpinned by a comprehensive disaster recovery plan. Other options include cloud based services such as Google Drive and Dropbox (to name but a few), but make sure you're not storing sensitive data on these services and are comfortable with the often eye watering privacy policies.

Whilst these services are pretty good at storing files, they don't really help with incremental backups. Finding previous versions of files often involves spending inordinate amounts of time trawling through multiple files named '*final.doc*', '*final_v2.doc*' and '*final_usethisone.doc*' etc until you find the one you were looking for. The best way we know for both backing up files and managing different versions of files is to use Git and GitHub. To find out more about how you can use RStudio, Git and GitHub together see Chapter 7.

1.10. Citing R and R packages

Many people have invested huge amounts of time and energy making R the great piece of software you're now using. If you use R in your work (and we hope you do) please remember to give appropriate credit by citing not only R but also all the packages you used. To get the most up to date citation for R you can use the `citation()` function.

```
citation()
```

To cite R in publications use:

R Core Team (2024). _R: A Language and Environment for Statistical Computing_. R Foundation for Statistical Computing, Vienna, Austria.
<https://www.R-project.org/>.

A BibTeX entry for LaTeX users is

```
@Manual{,  
  title = {R: A Language and Environment for Statistical Computing},  
  author = {{R Core Team}},  
  organization = {R Foundation for Statistical Computing},  
  address = {Vienna, Austria},  
  year = {2024},
```

```
url = {https://www.R-project.org/},  
}
```

We have invested a lot of time and effort in creating R, please cite it when using it for data analysis. See also 'citation("pkgname")' for citing R packages.

If you want to cite a particular package you've used for your data analysis, you can also use the `citation()` function to get the info.

```
citation(package = "here")
```

To cite package 'here' in publications use:

Müller K (2020). `_here: A Simpler Way to Find Your Files_`. R package version 1.0.1, <<https://CRAN.R-project.org/package=here>>.

A BibTeX entry for LaTeX users is

```
@Manual{,  
  title = {here: A Simpler Way to Find Your Files},  
  author = {Kirill Müller},  
  year = {2020},  
  note = {R package version 1.0.1},  
  url = {https://CRAN.R-project.org/package=here},  
}
```

In our view the most useful tool for citation is the package `grateful`  which allow you to generate the citing information in a file, as well as creating either a sentence or a table citing all packages used. This should become the standard in any manuscript honestly. See Table 22.1 for an example output produced with `grateful`.

Chapter 2

Some R basics

In this Chapter we'll cover how to:

- create objects and assigning values to objects
- exploring different types of objects and how to perform some common operations on objects
- get help in R and highlight some resources to help support your R learning.
- save your work
- use and install packages to extend base R capabilities.

2.1. Important considerations

We provide screenshot of RStudio but everything is really similar when using VSCode.

Before we continue, here are a few things to bear in mind as you work through this Chapter:

- R is case sensitive i.e. A is not the same as a and `anova` is not the same as `Anova`.
- Anything that follows a # symbol is interpreted as a comment and ignored by R. Comments should be used liberally throughout your code for both your own information and also to help your collaborators. Writing comments is a bit of an [art](#) and something that you will become more adept at as your experience grows.
- In R, commands are generally separated by a new line. You can also use a semicolon ; to separate your commands but we strongly recommend to avoid using it.
- If a continuation prompt + appears in the console after you execute your code this means that you haven't completed your code correctly. This often happens if you forget to close a bracket and is especially common

when nested brackets are used (((some command))). Just finish the command on the new line and fix the typo or hit escape on your keyboard (see point below) and fix.

- In general, R is fairly tolerant of extra spaces inserted into your code, in fact using spaces is actively encouraged. However, spaces should not be inserted into operators i.e. <- should not read < - (note the space). See the [style guide](#) for advice on where to place spaces to make your code more readable.
- If your console ‘hangs’ and becomes unresponsive after running a command you can often get yourself out of trouble by pressing the escape key (esc) on your keyboard or clicking on the stop icon in the top right of your console. This will terminate most current operations.

2.2. First step in the console

In Chapter 1, we learned about the R Console and creating scripts and Projects. We also saw how you write your R code in a script and then source this code into the console to get it to run (if you’ve forgotten how to do this, pop back to the console section (-Section 1.2.1.1) to refresh your memory). Writing your code in a script means that you’ll always have a permanent record of everything you’ve done (provided you save your script) and also allows you to make loads of comments to remind your future self what you’ve done. So, while you’re working through this Chapter we suggest that you create a new script (or RStudio Project) to write your code as you follow along.

As we saw in Chapter 1, at a basic level we can use R much as you would use a calculator. We can type an arithmetic expression into our script, then source it into the console and receive a result. For example, if we type the expression 1 + 1 and then source this line of code we get the answer 2 (😊!)

```
1 + 1
```

```
[1] 2
```

The [1] in front of the result tells you that the observation number at the beginning of the line is the first observation. This is not much help in this example, but can be quite useful when printing results with multiple lines (we’ll see an example below). The other obvious arithmetic operators are -, *, / for subtraction, multiplication and division respectively. Matrix multiplication operator is %*%. R follows the usual mathematical convention of [order of operations](#). For example, the expression 2 + 3 * 4 is interpreted to have the value $2 + (3 * 4) = 14$, not $(2 + 3) * 4 = 20$. There are a huge range of mathematical functions in R, some of the most useful include; `log()`, `log10()`, `exp()`, `sqrt()`.

```
log(1) # logarithm to base e
```

```
[1] 0
```

```
log10(1) # logarithm to base 10
```

```
[1] 0
```

```
exp(1) # natural antilog
```

```
[1] 2.718282
```

```
sqrt(4) # square root
```

```
[1] 2
```

```
4^2 # 4 to the power of 2
```

```
[1] 16
```

```
pi # not a function but useful
```

```
[1] 3.141593
```

It's important to realize that when you run code as we've done above, the result of the code (or **value**) is only displayed in the console. Whilst this can sometimes be useful it is usually much more practical to store the value(s) in an object.

2.3. Objects in R

At the heart of almost everything you will do (or ever likely to do) in R is the concept that everything in R is an **object**. These objects can be almost anything, from a single number or character string (like a word) to highly complex structures like the output of a plot, a summary of your statistical analysis or a set of R commands that perform a specific task. Understanding how you create objects and assign values to objects is key to understanding R.

2.3.1. Creating objects

To create an object we simply give the object a name. We can then assign a value to this object using the *assignment operator* `<-` (sometimes called the *gets operator*). The assignment operator is a composite symbol comprised of a ‘less than’ symbol `<` and a hyphen `-`.

```
my_obj <- 32
```

In the code above, we created an object called `my_obj` and assigned it a value of the number 32 using the assignment operator (in our head we always read this as ‘*my_obj is 32*’). You can also use `=` instead of `<-` to assign values but this is bad practice since it can lead to confusion later on when programming in R (see Chapter 5) and we would discourage you from using this notation.

To view the value of the object you simply type the name of the object

```
my_obj
```

```
[1] 32
```

Now that we’ve created this object, R knows all about it and will keep track of it during this current R session. All of the objects you create will be stored in the current workspace and you can view all the objects in your workspace in RStudio by clicking on the ‘Environment’ tab in the top right hand pane.

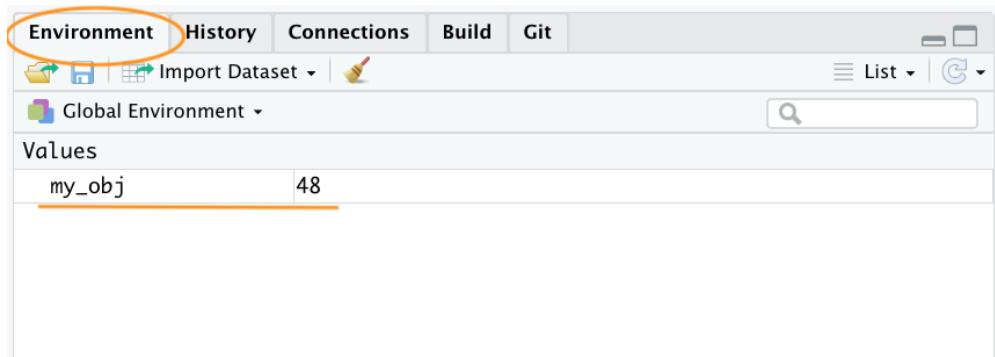


Figure 2.1.: RStudio Environment tab

If you click on the down arrow on the ‘List’ icon in the same pane and change to ‘Grid’ view RStudio will show you a summary of the objects including the type (numeric - it’s a number), the length (only one value in this object), its ‘physical’ size and its value (48 in this case). In VSCode, go on the R extension pane, and you can obtain the same information.

There are many different types of values that you can assign to an object. For example

Name	Type	Length	Size	Value
my_obj	numeric	1	56 B	48

Figure 2.2.: RStudio Environment tab in grid format

```
my_obj2 <- "R is cool"
```

Here we have created an object called `my_obj2` and assigned it a value of `R is cool` which is a character string. Notice that we have enclosed the string in quotes. If you forget to use the quotes you will receive an error message.

Our workspace now contains both objects we've created so far with `my_obj2` listed as type character.

Name	Type	Length	Size	Value
my_obj	numeric	1	56 B	48
my_obj2	character	1	120 B	"R is cool"

Figure 2.3.: RStudio environment tab with `my_obj2` as a character

To change the value of an existing object we simply reassign a new value to it. For example, to change the value of `my_obj2` from `"R is cool"` to the number 1024

```
my_obj2 <- 1024
```

Notice that the Type has changed to numeric and the value has changed to 1024 in the environment

Name	Type	Length	Size	Value
my_obj	numeric	1	56 B	48
my_obj2	numeric	1	56 B	1024

Figure 2.4.: RStudio environment tab with updated `my_obj2` as numeric

Once we have created a few objects, we can do stuff with our objects. For example, the following code creates a new object `my_obj3` and assigns it the value of `my_obj` added to `my_obj2` which is 1072 ($48 + 1024 = 1072$).

```
my_obj3 <- my_obj + my_obj2  
my_obj3
```

```
[1] 1056
```

Notice that to display the value of `my_obj3` we also need to write the object's name. The above code works because the values of both `my_obj` and `my_obj2` are numeric (i.e. a number). If you try to do this with objects with character values (**character class**) you will receive an error

```
char_obj <- "hello"  
char_obj2 <- "world!"  
char_obj3 <- char_obj + char_obj2  
# Error in char_obj+char_obj2:non-numeric argument to binary operator
```

The error message is essentially telling you that either one or both of the objects `char_obj` and `char_obj2` is not a number and therefore cannot be added together.

When you first start learning R, dealing with errors and warnings can be frustrating as they're often difficult to understand (what's an *argument*? what's a *binary operator*?). One way to find out more information about a particular error is to search for a generalised version of the error message. For the above error try searching '[non-numeric argument to binary operator error + r](#)' or even '[common r error messages](#)'.

Another error message that you'll get quite a lot when you first start using R is `Error: object 'XXX' not found`. As an example, take a look at the code below

```
my_obj <- 48  
my_obj4 <- my_obj + no_obj  
# Error: object 'no_obj' not found
```

R returns an error message because we haven't created (defined) the object `no_obj` yet. Another clue that there's a problem with this code is that, if you check your environment, you'll see that object `my_obj4` has not been created.

2.3.2. Naming objects

Naming your objects is one of the most difficult things you will do in R. Ideally your object names should be kept both short and informative which is not always easy. If you need to create objects with multiple words in their name then use either an underscore or a dot between words or capitalise the different words. We prefer the underscore format and never include uppercase in names (called *snake_case*)

```
output_summary <- "my analysis" # recommended#
output.summary <- "my analysis"
outputSummary <- "my analysis"
```

There are also a few limitations when it come to giving objects names. An object name cannot start with a number or a dot followed by a number (i.e. 2my_variable or .2my_variable). You should also avoid using non-alphanumeric characters in your object names (i.e. &, ^, /, ! etc). In addition, make sure you don't name your objects with reserved words (i.e. TRUE, NA) and it's never a good idea to give your object the same name as a built-in function. One that crops up more times than we can remember is

```
data <- read.table("mydatafile", header = TRUE)
```

Yes, `data()` is a function in R to load or list available data sets from packages.

2.4. Using functions in R

Up until now we've been creating simple objects by directly assigning a single value to an object. It's very likely that you'll soon want to progress to creating more complicated objects as your R experience grows and the complexity of your tasks increase. Happily, R has a multitude of functions to help you do this. You can think of a function as an object which contains a series of instructions to perform a specific task. The base installation of R comes with many functions already defined or you can increase the power of R by installing one of the 10,000's of **packages** now available. Once you get a bit more experience with using R you may want to define your own functions to perform tasks that are specific to your goals (more about this in Chapter 5).

The first function we will learn about is the `c()` function. The `c()` function is short for concatenate and we use it to join together a series of values and store them in a data structure called a **vector** (more on vectors in Chapter 3).

```
my_vec <- c(2, 3, 1, 6, 4, 3, 3, 7)
```

In the code above we've created an object called `my_vec` and assigned it a value using the function `c()`. There are a couple of really important points to note here. Firstly, when you use a function in R, the function name is **always** followed by a pair of round brackets even if there's nothing contained between the brackets. Secondly, the argument(s) of a function are placed inside the round brackets and are separated by commas. You can think of an argument as way of customising the use or behaviour of a function. In the example above, the arguments are the numbers we want to concatenate. Finally, one of the tricky things when you first start using R is to know which function to use for a particular task and how to use it. Thankfully each function will always have a help document associated with it which will explain how to use the function (more on this later Section 2.6) and a quick web search will also usually help you out.

To examine the value of our new object we can simply type out the name of the object as we did before

```
my_vec
```

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

Now that we've created a vector we can use other functions to do useful stuff with this object. For example, we can calculate the mean, variance, standard deviation and number of elements in our vector by using the `mean()`, `var()`, `sd()` and `length()` functions

```
mean(my_vec) # returns the mean of my_vec
```

```
[1] 3.625
```

```
var(my_vec) # returns the variance of my_vec
```

```
[1] 3.982143
```

```
sd(my_vec) # returns the standard deviation of my_vec
```

```
[1] 1.995531
```

```
length(my_vec) # returns the number of elements in my_vec
```

```
[1] 8
```

If we wanted to use any of these values later on in our analysis we can just assign the resulting value to another object

```
vec_mean <- mean(my_vec) # returns the mean of my_vec
vec_mean
```

```
[1] 3.625
```

Sometimes it can be useful to create a vector that contains a regular sequence of values in steps of one. Here we can make use of a shortcut using the : symbol.

```
my_seq <- 1:10 # create regular sequence
my_seq
```

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

```
my_seq2 <- 10:1 # in decending order
my_seq2
```

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

Other useful functions for generating vectors of sequences include the `seq()` and `rep()` functions. For example, to generate a sequence from 1 to 5 in steps of 0.5

```
my_seq2 <- seq(from = 1, to = 5, by = 0.5)
my_seq2
```

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Here we've used the arguments `from =` and `to =` to define the limits of the sequence and the `by =` argument to specify the increment of the sequence. Play around with other values for these arguments to see their effect.

The `rep()` function allows you to replicate (repeat) values a specified number of times. To repeat the value 2, 10 times

```
my_seq3 <- rep(2, times = 10) # repeats 2, 10 times  
my_seq3
```

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

You can also repeat non-numeric values

```
my_seq4 <- rep("abc", times = 3) # repeats 'abc' 3 times  
my_seq4
```

```
[1] "abc" "abc" "abc"
```

or each element of a series

```
my_seq5 <- rep(1:5, times = 3) # repeats the series 1 to  
# 5, 3 times  
my_seq5
```

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

or elements of a series

```
my_seq6 <- rep(1:5, each = 3) # repeats each element of the  
# series 3 times  
my_seq6
```

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

We can also repeat a non-sequential series

```
my_seq7 <- rep(c(3, 1, 10, 7), each = 3) # repeats each  
# element of the  
# series 3 times  
my_seq7
```

```
[1] 3 3 3 1 1 1 10 10 10 7 7 7
```

Note in the code above how we've used the `c()` function inside the `rep()` function. Nesting functions allows us to build quite complex commands within a single line of code and is a very common practice when using R. However, care needs to be taken as too many nested functions can make your code quite difficult for others to understand (or yourself some time in the future!). We could rewrite the code above to explicitly separate the two different steps to generate our vector. Either approach will give the same result, you just need to use your own judgement as to which is more readable.

```
in_vec <- c(3, 1, 10, 7)  
my_seq7 <- rep(in_vec, each = 3) # repeats each element of  
# the series 3 times  
my_seq7
```

```
[1] 3 3 3 1 1 1 10 10 10 7 7 7
```

2.5. Working with vectors

Manipulating, summarising and sorting data using R is an important skill to master but one which many people find a little confusing at first. We'll go through a few simple examples here using vectors to illustrate some important concepts but will build on this in much more detail in Chapter 3 where we will look at more complicated (and useful) data structures.

2.5.1. Extracting elements

To extract (also known as indexing or subscripting) one or more values (more generally known as elements) from a vector we use the square bracket `[]` notation. The general approach is to name the object you wish to extract from, then a set of square brackets with an index of the element you wish to extract contained within the square brackets. This index can be a position or the result of a logical test.

Positional index

To extract elements based on their position we simply write the position inside the []. For example, to extract the 3rd value of my_vec

```
my_vec # remind ourselves what my_vec looks like
```

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

```
my_vec[3] # extract the 3rd value
```

```
[1] 1
```

```
# if you want to store this value in another object
```

```
val_3 <- my_vec[3]
```

```
val_3
```

```
[1] 1
```

Note that the positional index starts at 1 rather than 0 like some other other programming languages (i.e. Python).

We can also extract more than one value by using the c() function inside the square brackets. Here we extract the 1st, 5th, 6th and 8th element from the my_vec object

```
my_vec[c(1, 5, 6, 8)]
```

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

Or we can extract a range of values using the : notation. To extract the values from the 3rd to the 8th elements

```
my_vec[3:8]
```

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

2.5.1.1. Logical index

Another really useful way to extract data from a vector is to use a logical expression as an index. For example, to extract all elements with a value greater than 4 in the vector `my_vec`

```
my_vec[my_vec > 4]
```

```
[1] 6 7
```

Here, the logical expression is `my_vec > 4` and R will only extract those elements that satisfy this logical condition. So how does this actually work? If we look at the output of just the logical expression without the square brackets you can see that R returns a vector containing either TRUE or FALSE which correspond to whether the logical condition is satisfied for each element. In this case only the 4th and 8th elements return a TRUE as their value is greater than 4.

```
my_vec > 4
```

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

So what R is actually doing under the hood is equivalent to

```
my_vec[c(FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE)]
```

```
[1] 6 7
```

and only those elements that are TRUE will be extracted.

In addition to the `<` and `>` operators you can also use composite operators to increase the complexity of your expressions. For example the expression for ‘greater or equal to’ is `>=`. To test whether a value is equal to a value we need to use a double equals symbol `==` and for ‘not equal to’ we use `!=` (the `!` symbol means ‘not’).

```
my_vec[my_vec >= 4] # values greater or equal to 4
```

```
[1] 6 4 7
```

```
my_vec[my_vec < 4] # values less than 4
```

```
[1] 2 3 1 3 3
```

```
my_vec[my_vec <= 4] # values less than or equal to 4
```

```
[1] 2 3 1 4 3 3
```

```
my_vec[my_vec == 4] # values equal to 4
```

```
[1] 4
```

```
my_vec[my_vec != 4] # values not equal to 4
```

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

We can also combine multiple logical expressions using [Boolean expressions](#). In R the `&` symbol means AND and the `|` symbol means OR. For example, to extract values in `my_vec` which are less than 6 AND greater than 2

```
val26 <- my_vec[my_vec < 6 & my_vec > 2]
```

```
val26
```

```
[1] 3 4 3 3
```

or extract values in `my_vec` that are greater than 6 OR less than 3

```
val63 <- my_vec[my_vec > 6 | my_vec < 3]
```

```
val63
```

```
[1] 2 1 7
```

2.5.2. Replacing elements

We can change the values of some elements in a vector using our `[]` notation in combination with the assignment operator `<-`. For example, to replace the 4th value of our `my_vec` object from 6 to 500

```
my_vec[4] <- 500  
my_vec
```

```
[1] 2 3 1 500 4 3 3 7
```

We can also replace more than one value or even replace values based on a logical expression

```
# replace the 6th and 7th element with 100  
my_vec[c(6, 7)] <- 100  
my_vec
```

```
[1] 2 3 1 500 4 100 100 7
```

```
# replace element that are less than or equal to 4 with 1000  
my_vec[my_vec <= 4] <- 1000  
my_vec
```

```
[1] 1000 1000 1000 500 1000 100 100 7
```

2.5.3. Ordering elements

In addition to extracting particular elements from a vector we can also order the values contained in a vector. To sort the values from lowest to highest value we can use the `sort()` function

```
vec_sort <- sort(my_vec)  
vec_sort
```

```
[1] 7 100 100 500 1000 1000 1000 1000
```

To reverse the sort, from highest to lowest, we can either include the `decreasing = TRUE` argument when using the `sort()` function

```
vec_sort2 <- sort(my_vec, decreasing = TRUE)  
vec_sort2
```

```
[1] 1000 1000 1000 1000 500 100 100 7
```

or first sort the vector using the `sort()` function and then reverse the sorted vector using the `rev()` function. This is another example of nesting one function inside another function.

```
vec_sort3 <- rev(sort(my_vec))  
vec_sort3
```

```
[1] 1000 1000 1000 1000 500 100 100 7
```

Whilst sorting a single vector is fun, perhaps a more useful task would be to sort one vector according to the values of another vector. To do this we should use the `order()` function in combination with `[]`. To demonstrate this let's create a vector called `height` containing the height of 5 different people and another vector called `p.names` containing the names of these people (so Joanna is 180 cm, Charlotte is 155 cm etc)

```
height <- c(180, 155, 160, 167, 181)  
height
```

```
[1] 180 155 160 167 181
```

```
p.names <- c("Joanna", "Charlotte", "Helen", "Karen", "Amy")  
p.names
```

```
[1] "Joanna"      "Charlotte"    "Helen"        "Karen"        "Amy"
```

Our goal is to order the people in `p.names` in ascending order of their `height`. The first thing we'll do is use the `order()` function with the `height` variable to create a vector called `height_ord`

```
height_ord <- order(height)  
height_ord
```

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

OK, what's going on here? The first value, 2, (remember ignore [1]) should be read as ‘the smallest value of `height` is the second element of the `height` vector’. If we check this by looking at the `height` vector above, you can see that element 2 has a value of 155, which is the smallest value. The second smallest value in `height` is the 3rd element of `height`, which when we check is 160 and so on. The largest value of `height` is element 5 which is 181. Now that we have a vector of the positional indices of heights in ascending order (`height_ord`), we can extract these values from our `p.names` vector in this order

```
names_ord <- p.names[height_ord]  
names_ord
```

```
[1] "Charlotte" "Helen"      "Karen"       "Joanna"     "Amy"
```

You're probably thinking ‘what's the use of this?’ Well, imagine you have a dataset which contains two columns of data and you want to sort each column. If you just use `sort()` to sort each column separately, the values of each column will become uncoupled from each other. By using the ‘order()’ on one column, a vector of positional indices is created of the values of the column in ascending order. This vector can be used on the second column, as the index of elements which will return a vector of values based on the first column. In all honestly, when you have multiple related vectors you need to use a `data.frame` type of object (see Chapter 3) instead of multiple independent vectors.

2.5.4. Vectorisation

One of the great things about R functions is that most of them are vectorised. This means that the function will operate on all elements of a vector without needing to apply the function on each element separately. For example, to multiply each element of a vector by 5 we can simply use

```
# create a vector  
my_vec2 <- c(3, 5, 7, 1, 9, 20)  
  
# multiply each element by 5  
my_vec2 * 5
```

```
[1] 15 25 35  5 45 100
```

Or we can add the elements of two or more vectors

```
# create a second vector  
my_vec3 <- c(17, 15, 13, 19, 11, 0)  
  
# add both vectors  
my_vec2 + my_vec3
```

```
[1] 20 20 20 20 20 20
```

```
# multiply both vectors  
my_vec2 * my_vec3
```

```
[1] 51 75 91 19 99 0
```

However, you must be careful when using vectorisation with vectors of different lengths as R will quietly recycle the elements in the shorter vector rather than throw a wobbly (error).

```
# create a third vector  
my_vec4 <- c(1, 2)  
  
# add both vectors - quiet recycling!  
my_vec2 + my_vec4
```

```
[1] 4 7 8 3 10 22
```

2.5.5. Missing data

In R, missing data is usually represented by an NA symbol meaning ‘Not Available’. Data may be missing for a whole bunch of reasons, maybe your machine broke down, maybe you broke down, maybe the weather was too bad to collect data on a particular day etc etc. Missing data can be a pain in the proverbial both from an R perspective and also a statistical perspective. From an R perspective missing data can be problematic as different functions deal with missing data in different ways. For example, let’s say we collected air temperature readings over 10 days, but our thermometer broke on day 2 and again on day 9 so we have no data for those days

```
temp <- c(7.2, NA, 7.1, 6.9, 6.5, 5.8, 5.8, 5.5, NA, 5.5)
temp
```

```
[1] 7.2 NA 7.1 6.9 6.5 5.8 5.8 5.5 NA 5.5
```

We now want to calculate the mean temperature over these days using the `mean()` function

```
mean_temp <- mean(temp)
mean_temp
```

```
[1] NA
```

If a vector has a missing value then the only possible value to return when calculating a mean is `NA`. R doesn't know that you perhaps want to ignore the `NA` values (R can't read your mind - yet!). If we look at the help file (using `?mean` - see the next section Section 2.6 for more details) associated with the `mean()` function we can see there is an argument `na.rm =` which is set to `FALSE` by default.

`na.rm` - a logical value indicating whether `NA` values should be stripped before the computation proceeds.

If we change this argument to `na.rm = TRUE` when we use the `mean()` function this will allow us to ignore the `NA` values when calculating the mean

```
mean_temp <- mean(temp, na.rm = TRUE)
mean_temp
```

```
[1] 6.2875
```

It's important to note that the `NA` values have not been removed from our `temp` object (that would be bad practice), rather the `mean()` function has just ignored them. The point of the above is to highlight how we can change the default behaviour of a function using an appropriate argument. The problem is that not all functions will have an `na.rm =` argument, they might deal with `NA` values differently. However, the good news is that every help file associated with any function will **always** tell you how missing data are handled by default.

2.6. Getting help

This book is intended as a relatively brief introduction to R and as such you will soon be using functions and packages that go beyond this scope of this introductory text. Fortunately, one of the strengths of R is its comprehensive and easily accessible help system and wealth of online resources where you can obtain further information.

2.6.1. R help

To access R's built-in help facility to get information on any function simply use the `help()` function. For example, to open the help page for our friend the `mean()` function.

```
help("mean")
```

or you can use the equivalent shortcut

```
?mean
```

the help page is displayed in the 'Help' tab in the Files pane (usually in the bottom right of RStudio)

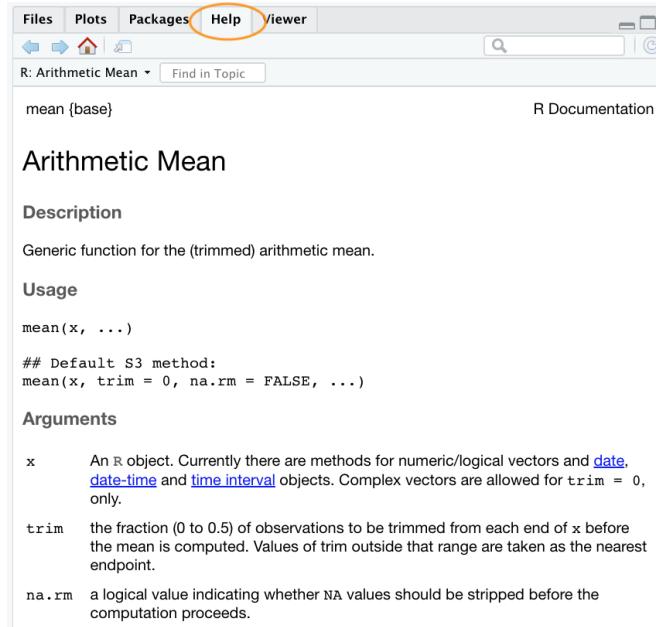


Figure 2.5.: Help page for the `mean()` function in RStudio Help pane

Admittedly the help files can seem anything but helpful when you first start using R. This is probably because they're written in a very concise manner and the language used is often quite technical and full of jargon. Having said that, you do get used to this and will over time even come to appreciate a certain beauty in their brevity (honest!). One

of the great things about the help files is that they all have a very similar structure regardless of the function. This makes it easy to navigate through the file to find exactly what you need.

The first line of the help document contains information such as the name of the function and the package where the function can be found. There are also other headings that provide more specific information such as

Headings	Description
Description:	gives a brief description of the function and what it does.
Usage:	gives the name of the arguments associated with the function and possible default values.
Arguments:	provides more detail regarding each argument and what they do.
Details:	gives further details of the function if required.
Value:	if applicable, gives the type and structure of the object returned by the function or the operator.
See Also:	provides information on other help pages with similar or related content.
Examples:	gives some examples of using the function.

The **Examples** are really helpful, all you need to do is copy and paste them into the console to see what happens. You can also access examples at any time by using the `example()` function (i.e. `example("mean")`)

The `help()` function is useful if you know the name of the function. If you're not sure of the name, but can remember a key word then you can search R's help system using the `help.search()` function.

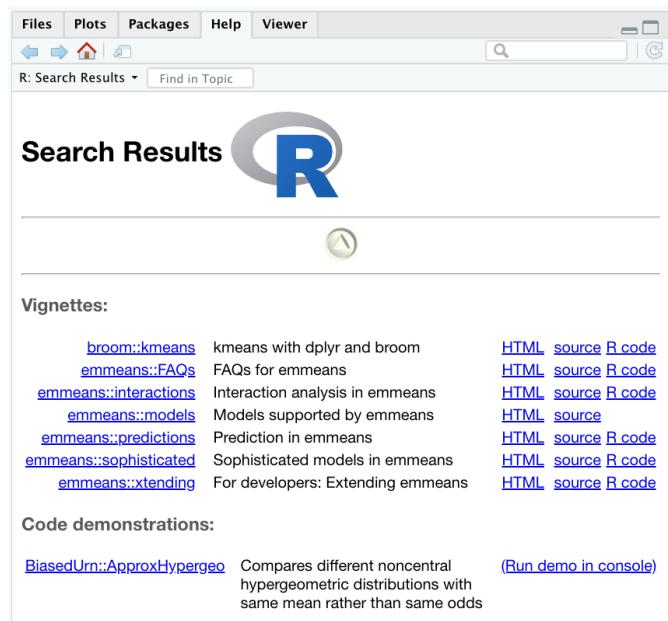
```
help.search("mean")
```

or you can use the equivalent shortcut

```
??mean
```

The results of the search will be displayed in RStudio under the 'Help' tab as before. The `help.search()` function searches through the help documentation, code demonstrations and package vignettes and displays the results as clickable links for further exploration.

Another useful function is `apropos()`. This function can be used to list all functions containing a specified character string. For example, to find all functions with `mean` in their name

Figure 2.6.: Output of the `help.search()` function in RStudio

```
apropos("mean")
```

```
[1] ".colMeans"      ".rowMeans"       "colMeans"        "kmeans"  
[5] "mean"          "mean_temp"       "mean.Date"       "mean.default"  
[9] "mean.difftime" "mean.POSIXct"   "mean.POSIXlt"   "rowMeans"  
[13] "vec_mean"       "weighted.mean"
```

You can then bring up the help file for the relevant function.

```
help("kmeans")
```

Another function is `RSiteSearch()` which enables you to search for keywords and phrases in function help pages and vignettes for all CRAN packages. This function allows you to access the search engine of the R website <https://www.r-project.org/search.html> directly from the Console with the results displayed in your web browser.

```
RSiteSearch("regression")
```

2.6.2. Other sources of help

There really has never been a better time to start learning R. There are a plethora of freely available online resources ranging from whole courses to subject specific tutorials and mailing lists. There are also plenty of paid for options if

that's your thing but unless you've money to burn there really is no need to part with your hard earned cash. Some resources we have found helpful are listed below.

2.6.2.1. General R resources

- [R-Project](#): User contributed documentation
- [The R Journal](#): Journal of the R project for statistical computing
- [Swirl](#): An R package that teaches you R from within R
- [RStudio's printable cheatsheets](#)
- [Rseek](#) A custom Google search for R-related sites

2.6.2.2. Getting help

- [Internet search]: Use your favourite search engine (google, ecosia, duckduckgo, ...) for any error messages you get. It's not cheating and everyone does it! You'll be surprised how many other people have probably had the same problem and solved it.
- [Stack Overflow](#): There are many thousands of questions relevant to R on Stack Overflow. [Here](#) are the most popular ones, ranked by vote. Make sure you search for similar questions before asking your own, and make sure you include a [reproducible example](#) to get the most useful advice. A reproducible example is a minimal example that lets others who are trying to help you to see the error themselves.

2.6.2.3. R markdown resources

- [Basic markdown and R markdown reference](#)
- [A good markdown reference](#)
- [A good 10-minute markdown tutorial](#)
- [RStudio's R markdown cheatsheet](#)
- [R markdown reference sheet](#)
- [The R markdown documentation](#) including a [getting started guide](#), a [gallery of demos](#), and several [articles](#) for more advanced usage.
- [The knitr website](#) has lots of useful reference material about how knitr works.

2.6.2.4. Git and GitHub resources

- [Happy Git](#): Great resource for using Git and GitHub
- [Version control with RStudio](#): RStudio document for using version control
- [Using Git from RStudio](#): Good 10 minute guide
- [The R Class](#): In depth guide to using Git and GitHub with RStudio

2.6.2.5. R programming

- [R Programming for Data Science](#): In depth guide to R programming
- [R for Data Science](#): Fantastic book, tidyverse orientated

2.7. Saving stuff in R

Your approach to saving work in R and RStudio depends on what you want to save. Most of the time the only thing you will need to save is the R code in your script(s). Remember your script is a reproducible record of everything you've done so all you need to do is open up your script in a new RStudio session and source it into the R Console and you're back to where you left off.

Unless you've followed our suggestion about changing the default settings for RStudio Projects (see Section 1.5) you will be asked whether you want to save your workspace image every time you exit RStudio. We suggest that 99.9% of the time that you don't want do this. By starting with a clean RStudio session each time we come back to our analysis we can be sure to avoid any potential conflicts with things we've done in previous sessions.

There are, however, some occasions when saving objects you've created in R is useful. For example, let's say you're creating an object that takes hours (even days) of computational time to generate. It would be extremely inconvenient to have to wait all this time each time you come back to your analysis (although we would suggest exporting this to an external file is a better solution). In this case we can save this object as an external .RData file which we can load back into RStudio the next time we want to use it. To save an object to an .RData file you can use the `save()` function (notice we don't need to use the assignment operator here)

```
save(nameOfObject, file = "name_of_file.RData")
```

or if you want to save all of the objects in your workspace into a single .RData file use the `save.image()` function

```
save.image(file = "name_of_file.RData")
```

To load your .RData file back into RStudio use the `load()` function

```
load(file = "name_of_file.RData")
```

2.8. R packages

The base installation of R comes with many useful packages as standard. These packages will contain many of the functions you will use on a daily basis. However, as you start using R for more diverse projects (and as your own use of R evolves) you will find that there comes a time when you will need to extend R's capabilities. Happily, many thousands of R users have developed useful code and shared this code as installable packages. You can think of a package as a collection of functions, data and help files collated into a well defined standard structure which you can download and install in R. These packages can be downloaded from a variety of sources but the most popular are [CRAN](#), [Bioconductor](#) and [GitHub](#). Currently, CRAN hosts over 15000 packages and is the official repository for user contributed R packages. Bioconductor provides open source software oriented towards bioinformatics and hosts over 1800 R packages. GitHub is a website that hosts git repositories for all sorts of software and projects (not just R). Often, cutting edge development versions of R packages are hosted on GitHub so if you need all the new bells and whistles then this may be an option. However, a potential downside of using the development version of an R package is that it might not be as stable as the version hosted on CRAN (it's in development!) and updating packages won't be automatic.

2.8.1. Using packages

Once you have installed a package onto your computer it is not immediately available for you to use. To use a package you first need to load the package by using the `library()` function. For example, to load the `remotes` 📦 package you previously installed

```
library(remotes)
```

The `library()` function will also load any additional packages required and may print out additional package information. It is important to realize that every time you start a new R session (or restore a previously saved session) you need to load the packages you will be using. We tend to put all our `library()` statements required for our analysis near the top of our R scripts to make them easily accessible and easy to add to as our code develops. If you

try to use a function without first loading the relevant R package you will receive an error message that R could not find the function. For example, if you try to use the `install_github()` function without loading the `remotes`  package first you will receive the following error

```
install_github("tidyverse/dplyr")  
  
# Error in install_github("tidyverse/dplyr") :  
#   could not find function "install_github"
```

Sometimes it can be useful to use a function without first using the `library()` function. If, for example, you will only be using one or two functions in your script and don't want to load all of the other functions in a package then you can access the function directly by specifying the package name followed by two colons and then the function name

```
remotes::install_github("tidyverse/dplyr")
```

This is how we were able to use the `install()` and `install_github()` functions above without first loading the packages `BiocManager`  and `remotes`  . Most of the time we recommend using the `library()` function.

2.8.2. Installing R packages

2.8.2.1. CRAN packages

To install a package from CRAN you can use the `install.packages()` function. For example if you want to install the `remotes` package enter the following code into the Console window of RStudio (note: you will need a working internet connection to do this)

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

You may be asked to select a CRAN mirror, just select ‘0-cloud’ or a mirror near to your location. The `dependencies = TRUE` argument ensures that additional packages that are required will also be installed.

It's good practice to regularly update your previously installed packages to get access to new functionality and bug fixes. To update CRAN packages you can use the `update.packages()` function (you will need a working internet connection for this)

```
update.packages(ask = FALSE)
```

The `ask = FALSE` argument avoids having to confirm every package download which can be a pain if you have many packages installed.

2.8.2.2. Bioconductor packages

To install packages from Bioconductor the process is a [little different](#). You first need to install the `BiocManager`  package. You only need to do this once unless you subsequently reinstall or upgrade R

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

Once the `BiocManager`  package has been installed you can either install all of the ‘core’ Bioconductor packages with

```
BiocManager::install()
```

or install specific packages such as the `GenomicRanges`  and `edgeR`  packages

```
BiocManager::install(c("GenomicRanges", "edgeR"))
```

To update Bioconductor packages just use the `BiocManager::install()` function again

```
BiocManager::install(ask = FALSE)
```

Again, you can use the `ask = FALSE` argument to avoid having to confirm every package download.

2.8.2.3. GitHub packages

There are multiple options for installing packages hosted on GitHub. Perhaps the most efficient method is to use the `install_github()` function from the `remotes`  package (you installed this package previously (Section 2.8.2.1)). Before you use the function you will need to know the GitHub username of the repository owner and also the name of the repository. For example, the development version of `dplyr`  from Hadley Wickham is hosted on the tidyverse GitHub account and has the repository name ‘`github dplyr`’ (just search for ‘`github dplyr`’). To install this version from GitHub use

```
remotes::install_github("tidyverse/dplyr")
```

The safest way (that we know of) to update a package installed from GitHub is to just reinstall it using the above command.

Chapter 3

Data

Until now, you've created fairly simple data in R and stored it as a vector (Section 2.4). However, most (if not all) of you will have much more complicated datasets from your various experiments and surveys that go well beyond what a vector can handle. Learning how R deals with different types of data and data structures, how to import your data into R and how to manipulate and summarize your data are some of the most important skills you will need to master.

In this Chapter we'll go over the main data types in R and focus on some of the most common data structures. We will also cover how to import data into R from an external file, how to manipulate (wrangle) and summarize data and finally how to export data from R to an external file.

3.1. Data types

Understanding the different types of data and how R deals with these data is important. The temptation is to glaze over and skip these technical details, but beware, this can come back to bite you somewhere unpleasant if you don't pay attention. We've already seen an example (Section 2.3.1) of this when we tried (and failed) to add two character objects together using the `+` operator.

R has six basic types of data; numeric, integer, logical, complex and character. The keen eyed among you will notice we've only listed five data types here, the final data type is raw which we won't cover as it's not useful 99.99% of the time. We also won't cover complex numbers, but will let you [imagine](#) that part!

- **Numeric** data are numbers that contain a decimal. Actually they can also be whole numbers but we'll gloss over that.
- **Integers** are whole numbers (those numbers without a decimal point).

- **Logical** data take on the value of either TRUE or FALSE. There's also another special type of logical called NA to represent missing values.
- **Character** data are used to represent string values. You can think of character strings as something like a word (or multiple words). A special type of character string is a *factor*, which is a string but with additional attributes (like levels or an order). We'll cover factors later.

R is (usually) able to automatically distinguish between different classes of data by their nature and the context in which they're used although you should bear in mind that R can't actually read your mind and you may have to explicitly tell R how you want to care a data type. You can find out the type (or class) of any object using the `class()` function.

```
num <- 2.2
```

```
class(num)
```

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

```
char <- "hello"
```

```
class(char)
```

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

```
logi <- TRUE
```

```
class(logi)
```

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

Alternatively, you can ask if an object is a specific class using using a logical test. The `is.[classOfData]()` family of functions will return either a TRUE or a FALSE.

```
is.numeric(num)
```

```
[1] TRUE
```

```
is.character(num)
```

```
[1] FALSE
```

```
is.character(char)
```

```
[1] TRUE
```

```
is.logical(logi)
```

```
[1] TRUE
```

It can sometimes be useful to be able to change the class of a variable using the `as.[className]()` family of coercion functions, although you need to be careful when doing this as you might receive some unexpected results (see what happens below when we try to convert a character string to a numeric).

```
# coerce numeric to character  
class(num)
```

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

```
num_char <- as.character(num)  
num_char
```

```
[1] "2.2"
```

```
class(num_char)
```

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

```
# coerce character to numeric!  
class(char)
```

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

```
char_num <- as.numeric(char)
```

Warning: NAs introduced by coercion

Here's a summary table of some of the logical test and coercion functions available to you.

Type	Logical test	Coercing
Character	<code>is.character</code>	<code>as.character</code>
Numeric	<code>is.numeric</code>	<code>as.numeric</code>
Logical	<code>is.logical</code>	<code>as.logical</code>
Factor	<code>is.factor</code>	<code>as.factor</code>
Complex	<code>is.complex</code>	<code>as.complex</code>

3.2. Data structures

Now that you've been introduced to some of the most important classes of data in R, let's have a look at some of main structures that we have for storing these data.

3.2.1. Scalars and vectors

Perhaps the simplest type of data structure is the vector. You've already been introduced to vectors in Section 2.4 although some of the vectors you created only contained a single value. Vectors that have a single value (length 1) are called scalars. Vectors can contain numbers, characters, factors or logicals, but the key thing to remember is that all the elements inside a vector must be of the same class. In other words, vectors can contain either numbers, characters or logical but not mixtures of these types of data. There is one important exception to this, you can include NA (remember this is special type of logical) to denote missing data in vectors with other data types.

3.2.2. Matrices and arrays

Another useful data structure used in many disciplines such as population ecology, theoretical and applied statistics is the matrix. A matrix is simply a vector that has additional attributes called dimensions. Arrays are just multidimensional matrices. Again, matrices and arrays must contain elements all of the same data class.

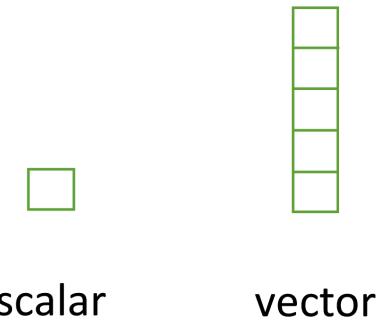


Figure 3.1.: Scalar and vector data structure

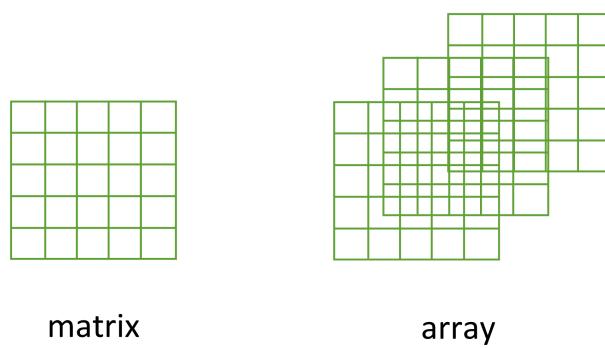


Figure 3.2.: Matrix and array data structure

A convenient way to create a matrix or an array is to use the `matrix()` and `array()` functions respectively. Below, we will create a matrix from a sequence 1 to 16 in four rows (`nrow = 4`) and fill the matrix row-wise (`byrow = TRUE`) rather than the default column-wise. When using the `array()` function we define the dimensions using the `dim =` argument, in our case 2 rows, 4 columns in 2 different matrices.

```
my_mat <- matrix(1:16, nrow = 4, byrow = TRUE)  
my_mat
```

```
[,1] [,2] [,3] [,4]  
[1,] 1 2 3 4  
[2,] 5 6 7 8  
[3,] 9 10 11 12  
[4,] 13 14 15 16
```

```
my_array <- array(1:16, dim = c(2, 4, 2))  
my_array
```

```
, , 1
```

```
[,1] [,2] [,3] [,4]  
[1,] 1 3 5 7  
[2,] 2 4 6 8
```

```
, , 2
```

```
[,1] [,2] [,3] [,4]  
[1,] 9 11 13 15  
[2,] 10 12 14 16
```

Sometimes it's also useful to define row and column names for your matrix but this is not a requirement. To do this use the `rownames()` and `colnames()` functions.

```
rownames(my_mat) <- c("A", "B", "C", "D")
colnames(my_mat) <- c("a", "b", "c", "d")
my_mat
```

	a	b	c	d
A	1	2	3	4
B	5	6	7	8
C	9	10	11	12
D	13	14	15	16

Once you've created your matrices you can do useful stuff with them and as you'd expect, R has numerous built in functions to perform matrix operations. Some of the most common are given below. For example, to transpose a matrix we use the transposition function `t()`

```
my_mat_t <- t(my_mat)
my_mat_t
```

	A	B	C	D
a	1	5	9	13
b	2	6	10	14
c	3	7	11	15
d	4	8	12	16

To extract the diagonal elements of a matrix and store them as a vector we can use the `diag()` function

```
my_mat_diag <- diag(my_mat)
my_mat_diag
```

```
[1] 1 6 11 16
```

The usual matrix addition, multiplication etc can be performed. Note the use of the `%*%` operator to perform matrix multiplication.

```
mat.1 <- matrix(c(2, 0, 1, 1), nrow = 2) # notice that the matrix has been filled  
mat.1 # column-wise by default
```

```
[,1] [,2]  
[1,] 2 1  
[2,] 0 1
```

```
mat.2 <- matrix(c(1, 1, 0, 2), nrow = 2)  
mat.2
```

```
[,1] [,2]  
[1,] 1 0  
[2,] 1 2
```

```
mat.1 + mat.2 # matrix addition
```

```
[,1] [,2]  
[1,] 3 1  
[2,] 1 3
```

```
mat.1 * mat.2 # element by element products
```

```
[,1] [,2]  
[1,] 2 0  
[2,] 0 2
```

```
mat.1 %*% mat.2 # matrix multiplication
```

```
[,1] [,2]  
[1,] 3 2  
[2,] 1 2
```

3.2.3. Lists

The next data structure we will quickly take a look at is a list. Whilst vectors and matrices are constrained to contain data of the same type, lists are able to store mixtures of data types. In fact we can even store other data structures such as vectors and arrays within a list or even have a list of a list. This makes for a very flexible data structure which is ideal for storing irregular or non-rectangular data (see Chapter 5 for an example).

To create a list we can use the `list()` function. Note how each of the three list elements are of different classes (character, logical, and numeric) and are of different lengths.

```
list_1 <- list(
  c("black", "yellow", "orange"),
  c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE),
  matrix(1:6, nrow = 3)
)
```

```
[[1]]
[1] "black"  "yellow" "orange"
```

```
[[2]]
[1] TRUE  TRUE FALSE  TRUE FALSE FALSE
```

```
[[3]]
 [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Elements of the list can be named during the construction of the list

```
list_2 <- list(
  colours = c("black", "yellow", "orange"),
  evaluation = c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE),
  time = matrix(1:6, nrow = 3)
```

```
)  
list_2  
  
  
$colours  
[1] "black"  "yellow" "orange"  
  
$evaluation  
[1] TRUE  TRUE FALSE  TRUE FALSE FALSE  
  
$time  
[,1] [,2]  
[1,]    1    4  
[2,]    2    5  
[3,]    3    6
```

or after the list has been created using the `names()` function

```
names(list_1) <- c("colours", "evaluation", "time")  
list_1
```

```
$colours  
[1] "black"  "yellow" "orange"  
  
$evaluation  
[1] TRUE  TRUE FALSE  TRUE FALSE FALSE  
  
$time  
[,1] [,2]  
[1,]    1    4  
[2,]    2    5  
[3,]    3    6
```

3.2.4. Data frames

By far the most commonly used data structure to store data in is the data frame. A data frame is a powerful two-dimensional object made up of rows and columns which looks superficially very similar to a matrix. However, whilst matrices are restricted to containing data all of the same type, data frames can contain a mixture of different types of data. Typically, in a data frame each row corresponds to an individual observation and each column corresponds to a different measured or recorded variable. This setup may be familiar to those of you who use LibreOffice Calc or Microsoft Excel to manage and store your data. Perhaps a useful way to think about data frames is that they are essentially made up of a bunch of vectors (columns) with each vector containing its own data type but the data type can be different between vectors.

As an example, the data frame below contains the results of an experiment to determine the effect of parental care (with or without) of unicorns (*Unicornus magnificens*) on offsprings growth under 3 different food availability regime. The data frame has 8 variables (columns) and each row represents an individual unicorn. The variables `care` and `food` are factors ([categorical](#) variables). The `p_care` variable has 2 levels (`care` and `no_care`) and the `food` level variable has 3 levels (`low`, `medium` and `high`). The variables `height`, `weight`, `mane_size` and `fluffyness` are numeric and the variable `horn_rings` is an integer representing the number of rings on the horn. Although the variable `block` has numeric values, these do not really have any order and could also be treated as a factor (i.e. they could also have been called A and B).

Table 3.2.: Imported unicorn data

<code>p_care</code>	<code>food</code>	<code>block</code>	<code>height</code>	<code>weight</code>	<code>mane_size</code>	<code>fluffyness</code>	<code>horn_rings</code>
care	medium	1	7.5	7.62	11.7	31.9	1
care	medium	1	10.7	12.14	14.1	46.0	10
care	medium	1	11.2	12.76	7.1	66.7	10
care	medium	1	10.4	8.78	11.9	20.3	1
care	medium	1	10.4	13.58	14.5	26.9	4
care	medium	1	9.8	10.08	12.2	72.7	9
no_care	low	2	3.7	8.10	10.5	60.5	6
no_care	low	2	3.2	7.45	14.1	38.1	4
no_care	low	2	3.9	9.19	12.4	52.6	9
no_care	low	2	3.3	8.92	11.6	55.2	6
no_care	low	2	5.5	8.44	13.5	77.6	9
no_care	low	2	4.4	10.60	16.2	63.3	6

There are a couple of important things to bear in mind about data frames. These types of objects are known as rectangular data (or tidy data) as each column must have the same number of observations. Also, any missing data should be recorded as an NA just as we did with our vectors.

We can construct a data frame from existing data objects such as vectors using the `data.frame()` function. As an example, let's create three vectors `p.height`, `p.weight` and `p.names` and include all of these vectors in a data frame object called `dataaf`.

```
p.height <- c(180, 155, 160, 167, 181)
p.weight <- c(65, 50, 52, 58, 70)
p.names <- c("Joanna", "Charlotte", "Helen", "Karen", "Amy")

dataaf <- data.frame(height = p.height, weight = p.weight, names = p.names)
dataaf
```

	height	weight	names
	180	65	Joanna
	155	50	Charlotte
	160	52	Helen
	167	58	Karen
	181	70	Amy

You'll notice that each of the columns are named with variable name we supplied when we used the `data.frame()` function. It also looks like the first column of the data frame is a series of numbers from one to five. Actually, this is not really a column but the name of each row. We can check this out by getting R to return the dimensions of the `dataaf` object using the `dim()` function. We see that there are 5 rows and 3 columns.

```
dim(dataaf) # 5 rows and 3 columns
```

```
[1] 5 3
```

Another really useful function which we use all the time is `str()` which will return a compact summary of the structure of the data frame object (or any object for that matter).

```
str(dataf)
```

```
'data.frame': 5 obs. of 3 variables:
 $ height: num 180 155 160 167 181
 $ weight: num 65 50 52 58 70
 $ names : chr "Joanna" "Charlotte" "Helen" "Karen" ...
```

The `str()` function gives us the data frame dimensions and also reminds us that `dataf` is a `data.frame` type object. It also lists all of the variables (columns) contained in the data frame, tells us what type of data the variables contain and prints out the first five values. We often copy this summary and place it in our R scripts with comments at the beginning of each line so we can easily refer back to it whilst writing our code. We showed you how to comment blocks in RStudio Section 1.7.

Also notice that R has automatically decided that our `p.names` variable should be a character (`chr`) class variable when we first created the data frame. Whether this is a good idea or not will depend on how you want to use this variable in later analysis. If we decide that this wasn't such a good idea we can change the default behaviour of the `data.frame()` function by including the argument `stringsAsFactors = TRUE`. Now our strings are automatically converted to factors.

```
p.height <- c(180, 155, 160, 167, 181)
p.weight <- c(65, 50, 52, 58, 70)
p.names <- c("Joanna", "Charlotte", "Helen", "Karen", "Amy")
```

```
dataf <- data.frame(
  height = p.height, weight = p.weight, names = p.names,
  stringsAsFactors = TRUE
)
str(dataf)
```

```
'data.frame': 5 obs. of 3 variables:
 $ height: num 180 155 160 167 181
 $ weight: num 65 50 52 58 70
 $ names : Factor w/ 5 levels "Amy","Charlotte",...: 4 2 3 5 1
```

3.3. Importing data

Although creating data frames from existing data structures is extremely useful, by far the most common approach is to create a data frame by importing data from an external file. To do this, you'll need to have your data correctly formatted and saved in a file format that R is able to recognize. Fortunately for us, R is able to recognize a wide variety of file formats, although in reality you'll probably end up only using two or three regularly.

3.3.1. Saving files to import

The easiest method of creating a data file to import into R is to enter your data into a spreadsheet using either Microsoft Excel or LibreOffice Calc and save the spreadsheet as a comma delimited file. We prefer LibreOffice Calc as it's open source, platform independent and free but MS Excel is OK too (but see [here](#) for some gotchas). Here's the data from the petunia experiment we discussed previously displayed in LibreOffice. If you want to follow along you can download the data file ('*unicorn.xlsx*') from Appendix A.

	A	B	C	D	E	F	G	H	I	J
1	treat	nitrogen	block	height	weight	leafarea	shootarea	flowers		
2	tip	medium	1	7.5	7.62	11.7	31.9	1		
3	tip	medium	1	10.7	12.14	14.1	46	10		
4	tip	medium	1	11.2	12.76	7.1	66.7	10		
5	tip	medium	1	10.4	8.78	11.9	20.3	1		
6	tip	medium	1	10.4	13.58	14.5	26.9	4		
7	tip	medium	1	9.8	10.08	12.2	72.7	9		
8	tip	medium	1	6.9	10.11	13.2	43.1	7		
9	tip	medium	1	9.4	10.28	14	28.5	6		
10	tip	medium	2	10.4	10.48	10.5	57.8	5		
11	tip	medium	2	12.3	13.48	16.1	36.9	8		
12	tip	medium	2	10.4	13.18	11.1	56.8	12		
13	tip	medium	2	11	11.56	12.6	31.3	6		
14	tip	medium	2	7.1	8.16	29.6	9.7	2		
15	tip	medium	2	6	11.22	13	16.4	3		
16	tip	medium	2	9	10.2	10.8	90.1	6		
17	tip	medium	2	4.5	12.55	13.4	14.4	6		
18	tip	high	1	12.6	18.66	18.6	54	9		
19	tip	high	1	10	18.07	16.9	90.5	3		
20	tip	high	1	10	13.29	15.8	142.7	12		
21	tip	high	1	8.5	14.33	13.2	91.4	5		
22	tip	high	1	14.1	19.12	13.1	113.2	13		
23	tip	high	1	10.1	15.49	12.6	77.2	12		
24	tip	high	1	8.5	17.82	20.5	54.4	3		

Figure 3.3.: Unicorn data in LibreOffice Calc

For those of you unfamiliar with the tab delimited file format it simply means that data in different columns are separated with a ',' character and is usually saved as a file with a '.csv' extension.

To save a spreadsheet as a comma delimited file in LibreOffice Calc select **File -> Save as ...** from the main menu. You will need to specify the location you want to save your file in the 'Save in folder' option and the name of the file in the 'Name' option. In the drop down menu located above the 'Save' button change the default 'All formats' to 'Text CSV (.csv)'.

Click the Save button and then select the 'Use Text CSV Format' option. Click on OK to save the file.

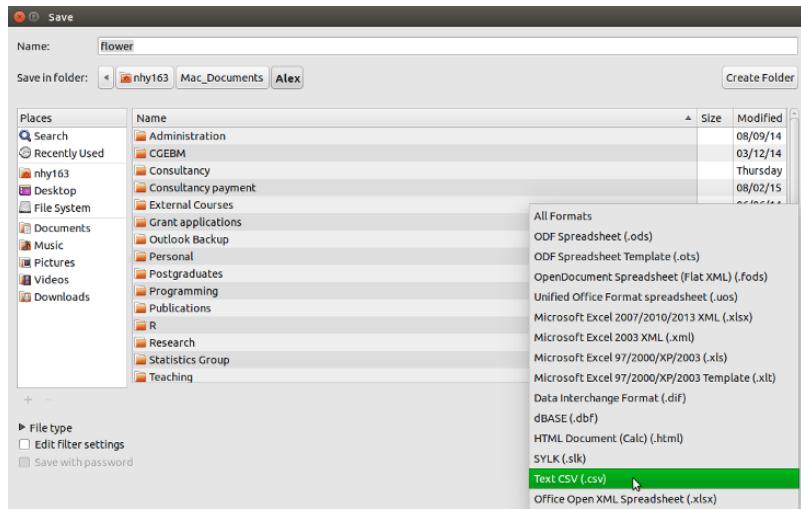


Figure 3.4.: Choosing csv format when saving with LibreOffice Calc

There are a couple of things to bear in mind when saving files to import into R which will make your life easier in the long run. Keep your column headings (if you have them) short and informative. Also avoid spaces in your column headings by replacing them with an underscore or a dot (i.e. replace `mane size` with `mane size` or `mane.size`) and avoid using special characters (i.e. `leaf area (mm^2)` or uppercase to simply your life). Remember, if you have missing data in your data frame (empty cells) you should use an `NA` to represent these missing values or have an empty cell. This will keep the data frame tidy.

3.3.2. Import functions

Once you've saved your data file in a suitable format we can now read this file into R. The workhorse function for importing data into R is the `read.table()` function (we discuss some alternatives later in the chapter). The `read.table()` function is a very flexible function with a shed load of arguments (see `?read.table`) but it's quite simple to use. Let's import a comma delimited file called `unicorns.csv` which contains the data we saw previously in this Chapter (Section 3.2.4) and assign it to an object called `unicorns`. The file is located in a `data` directory which itself is located in our root directory (Section 1.4). The first row of the data contains the variable (column) names. To use the `read.table()` function to import this file

```
unicorns <- read.table(
  file = "data/unicorns.csv", header = TRUE, sep = ",", dec = ".",
  stringsAsFactors = TRUE
)
```

There are a few things to note about the above command. First, the file path and the filename (including the

file extension) needs to be enclosed in either single or double quotes (i.e. the `data/unicorns.txt` bit) as the `read.table()` function expects this to be a character string. If your working directory is already set to the directory which contains the file, you don't need to include the entire file path just the filename. In the example above, the file path is separated with a single forward slash /. This will work regardless of the operating system you are using and we recommend you stick with this. However, Windows users may be more familiar with the single backslash notation and if you want to keep using this you will need to include them as double backslashes.

 **Warning**

Note though that the double backslash notation will **not** work on computers using Mac OSX or Linux operating systems. We thus strongly discourage it since it is not reproducible

The `header = TRUE` argument specifies that the first row of your data contains the variable names (i.e. `food`, `block` etc). If this is not the case you can specify `header = FALSE` (actually, this is the default value so you can omit this argument entirely). The `sep = ", "` argument tells R what is file delimiter.

Other useful arguments include `dec =` and `na.strings =`. The `dec =` argument allows you to change the default character (.) used for a decimal point. This is useful if you're in a country where decimal places are usually represented by a comma (i.e. `dec = ", "`). The `na.strings =` argument allows you to import data where missing values are represented with a symbol other than NA. This can be quite common if you are importing data from other statistical software such as Minitab which represents missing values as a * (`na.strings = "*"`).

Honestly, from the `read.table()` a series of predefined functions are available. They are all using `read.table()` but define format specific options. We can simply `read.csv()` to read a csv file, with “,” separation and “.” for decimals. In countries were “,” is used for decimals, csv files use “;” as a separator. In this case using `read.csv2()` would be needed. When working with tab delimited files, the functions `read.delim()` and `read.delim2()` can be used with “.” and “,” as decimal respectively.

After importing our data into R , to see the contents of the data frame we could just type the name of the object as we have done previously. **BUT** before you do that, think about why you're doing this. If your data frame is anything other than tiny, all you're going to do is fill up your Console with data. It's not like you can easily check whether there are any errors or that your data has been imported correctly. A much better solution is to use our old friend the `str()` function to return a compact and informative summary of your data frame.

```
str(unicorns)
```

```
'data.frame': 96 obs. of 8 variables:
```

```
$ p_care      : Factor w/ 2 levels "care","no_care": 1 1 1 1 1 1 1 1 1 1 ...
$ food        : Factor w/ 3 levels "high","low","medium": 3 3 3 3 3 3 3 3 3 3 ...
$ block       : int  1 1 1 1 1 1 1 1 2 2 ...
$ height      : num  7.5 10.7 11.2 10.4 10.4 9.8 6.9 9.4 10.4 12.3 ...
$ weight      : num  7.62 12.14 12.76 8.78 13.58 ...
$ mane_size   : num  11.7 14.1 7.1 11.9 14.5 12.2 13.2 14 10.5 16.1 ...
$ fluffyness: num  31.9 46 66.7 20.3 26.9 72.7 43.1 28.5 57.8 36.9 ...
$ horn_rings: int  1 10 10 1 4 9 7 6 5 8 ...
```

Here we see that `unicorns` is a ‘`data.frame`’ object which contains 96 rows and 8 variables (columns). Each of the variables are listed along with their data class and the first 10 values. As we mentioned previously in this Chapter, it can be quite convenient to copy and paste this into your R script as a comment block for later reference.

Notice also that your character string variables (`care` and `food`) have been imported as factors because we used the argument `stringsAsFactors = TRUE`. If this is not what you want you can prevent this by using the `stringsAsFactors = FALSE` or from R version 4.0.0 you can just leave out this argument as `stringsAsFactors = FALSE` is the default.

```
unicorns <- read.delim(file = "data/unicorns.txt")
str(unicorns)
```

```
'data.frame': 96 obs. of 8 variables:
 $ p_care      : chr  "care" "care" "care" "care" ...
 $ food        : chr  "medium" "medium" "medium" "medium" ...
 $ block       : int  1 1 1 1 1 1 1 1 2 2 ...
 $ height      : num  7.5 10.7 11.2 10.4 10.4 9.8 6.9 9.4 10.4 12.3 ...
 $ weight      : num  7.62 12.14 12.76 8.78 13.58 ...
 $ mane_size   : num  11.7 14.1 7.1 11.9 14.5 12.2 13.2 14 10.5 16.1 ...
 $ fluffyness: num  31.9 46 66.7 20.3 26.9 72.7 43.1 28.5 57.8 36.9 ...
 $ horn_rings: int  1 10 10 1 4 9 7 6 5 8 ...
```

If we just wanted to see the names of our variables (columns) in the data frame we can use the `names()` function which will return a character vector of the variable names.

```
names(unicorns)
```

```
[1] "p_care"      "food"        "block"        "height"       "weight"  
[6] "mane_size"   "fluffyness"  "horn_rings"
```

You can even import spreadsheet files from MS Excel or other statistics software directly into R but our advice is that this should generally be avoided if possible as it just adds a layer of uncertainty between you and your data. In our opinion it's almost always better to export your spreadsheets as tab or comma delimited files and then import them into R using one of the `read.table()` derivative function. If you're hell bent on directly importing data from other software you will need to install the `foreign` package which has functions for importing Minitab, SPSS, Stata and SAS files. For MS Excel and LO Calc spreadsheets, there are a few packages that can be used.

3.3.3. Common import frustrations

It's quite common to get a bunch of really frustrating error messages when you first start importing data into R. Perhaps the most common is

```
Error in file(file, "rt") : cannot open the connection  
In addition: Warning message:  
In file(file, "rt") :  
  cannot open file 'unicorns.txt': No such file or directory
```

This error message is telling you that R cannot find the file you are trying to import. It usually rears its head for one of a couple of reasons (or all of them!). The first is that you've made a mistake in the spelling of either the filename or file path. Another common mistake is that you have forgotten to include the file extension in the filename (i.e. `.txt`). Lastly, the file is not where you say it is or you've used an incorrect file path. Using RStudio Projects (Section 1.5) and having a logical directory structure (Section 1.4) goes a long way to avoiding these types of errors.

Another really common mistake is to forget to include the `header = TRUE` argument when the first row of the data contains variable names. For example, if we omit this argument when we import our `unicorns.txt` file everything looks OK at first (no error message at least)

```
unicorns_bad <- read.table(file = "data/unicorns.txt", sep = "\t")
```

but when we take a look at our data frame using `str()`

```
str(unicorns_bad)
```

```
'data.frame': 97 obs. of 8 variables:
 $ V1: chr  "p_care" "care" "care" "care" ...
 $ V2: chr  "food"  "medium" "medium" "medium" ...
 $ V3: chr  "block"  "1"    "1"    "1"    ...
 $ V4: chr  "height" "7.5"  "10.7" "11.2" ...
 $ V5: chr  "weight" "7.62" "12.14" "12.76" ...
 $ V6: chr  "mane_size" "11.7" "14.1" "7.1" ...
 $ V7: chr  "fluffyness" "31.9" "46"   "66.7" ...
 $ V8: chr  "horn_rings" "1"   "10"   "10"   ...
```

We can see an obvious problem, all of our variables have been imported as factors and our variables are named V1, V2, V3 ... V8. The problem happens because we haven't told the `read.table()` function that the first row contains the variable names and so it treats them as data. As soon as we have a single character string in any of our data vectors, R treats the vectors as character type data (remember all elements in a vector must contain the same type of data (Section 3.2.1)).

This is just one more argument to use `read.csv()` or `read.delim()` function with appropriate default values for arguments.

3.3.4. Other import options

There are numerous other functions to import data from a variety of sources and formats. Most of these functions are contained in packages that you will need to install before using them. We list a couple of the more useful packages and functions below.

The `fread()` function from the `data.table`  package is great for importing large data files quickly and efficiently (much faster than the `read.table()` function). One of the great things about the `fread()` function is that it will automatically detect many of the arguments you would normally need to specify (like `sep = etc`). One of the things you will need to consider though is that the `fread()` function will return a `data.table` object not a `data.frame` as would be the case with the `read.table()` function. This is usually not a problem as you can pass a `data.table` object to any function that only accepts `data.frame` objects. To learn more about the differences between `data.table` and `data.frame` objects see [here](#).

```
library(data.table)
all_data <- fread(file = "data/unicorns.txt")
```

Various functions from the `readr` package are also very efficient at reading in large data files. The `readr` package is part of the ‘tidyverse’ collection of packages and provides many equivalent functions to base R for importing data. The `readr` functions are used in a similar way to the `read.table()` or `read.csv()` functions and many of the arguments are the same (see `?readr::read_table` for more details). There are however some differences. For example, when using the `read_table()` function the `header = TRUE` argument is replaced by `col_names = TRUE` and the function returns a `tibble` class object which is the tidyverse equivalent of a `data.frame` object (see [here](#) for differences).

 Warning

Some functions are not happy to handle the data format produced by `tidyverse` and might require you to transform them to `data.frame` format using `data.frame()`.

```
library(readr)

# import white space delimited files
all_data <- read_table(file = "data/unicorns.txt", col_names = TRUE)

# import comma delimited files
all_data <- read_csv(file = "data/unicorns.csv")

# import tab delimited files
all_data <- read_delim(file = "data/unicorns.txt", delim = "\t")

# or use
all_data <- read_tsv(file = "data/unicorns.txt")
```

If your data file is ginormous, then the `ff` and `bigmemory` packages may be useful as they both contain import functions that are able to store large data in a memory efficient manner. You can find out more about these functions [here](#) and [here](#).

3.4. Wrangling data frames

Now that you're able to successfully import your data from an external file into R our next task is to do something useful with our data. Working with data is a fundamental skill which you'll need to develop and get comfortable with as you'll likely do a lot of it during any project. The good news is that R is especially good at manipulating, summarising and visualising data. Manipulating data (often known as data wrangling or munging) in R can at first seem a little daunting for the new user but if you follow a few simple logical rules then you'll quickly get the hang of it, especially with some practice.

Let's remind ourselves of the structure of the `unicorns` data frame we imported in the previous section.

```
unicorns <- read.table(file = "data/unicorns.txt", header = TRUE, sep = "\t")
str(unicorns)
```

```
'data.frame': 96 obs. of 8 variables:
 $ p_care    : chr  "care" "care" "care" "care" ...
 $ food       : chr  "medium" "medium" "medium" "medium" ...
 $ block      : int  1 1 1 1 1 1 1 1 2 2 ...
 $ height     : num  7.5 10.7 11.2 10.4 10.4 9.8 6.9 9.4 10.4 12.3 ...
 $ weight     : num  7.62 12.14 12.76 8.78 13.58 ...
 $ mane_size  : num  11.7 14.1 7.1 11.9 14.5 12.2 13.2 14 10.5 16.1 ...
 $ fluffyness: num  31.9 46 66.7 20.3 26.9 72.7 43.1 28.5 57.8 36.9 ...
 $ horn_rings: int  1 10 10 1 4 9 7 6 5 8 ...
```

To access the data in any of the variables (columns) in our data frame we can use the `$` notation. For example, to access the `height` variable in our `unicorns` data frame we can use `unicorns$height`. This tells R that the `height` variable is contained within the data frame `unicorns`.

```
unicorns$height
```

```
[1]  7.5 10.7 11.2 10.4 10.4  9.8  6.9  9.4 10.4 12.3 10.4 11.0  7.1  6.0  9.0
[16] 4.5 12.6 10.0 10.0  8.5 14.1 10.1  8.5  6.5 11.5  7.7  6.4  8.8  9.2  6.2
[31] 6.3 17.2  8.0  8.0  6.4  7.6  9.7 12.3  9.1  8.9  7.4  3.1  7.9  8.8  8.5
[46] 5.6 11.5  5.8  5.6  5.3  7.5  4.1  3.5  8.5  4.9  2.5  5.4  3.9  5.8  4.5
[61] 8.0  1.8  2.2  3.9  8.5  8.5  6.4  1.2  2.6 10.9  7.2  2.1  4.7  5.0  6.5
```

```
[76] 2.6 6.0 9.3 4.6 5.2 3.9 2.3 5.2 2.2 4.5 1.8 3.0 3.7 2.4 5.7  
[91] 3.7 3.2 3.9 3.3 5.5 4.4
```

This will return a vector of the `height` data. If we want we can assign this vector to another object and do stuff with it, like calculate a mean or get a summary of the variable using the `summary()` function.

```
f_height <- unicorns$height  
mean(f_height)
```

```
[1] 6.839583
```

```
summary(f_height)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.200	4.475	6.450	6.840	9.025	17.200

Or if we don't want to create an additional object we can use functions 'on-the-fly' to only display the value in the console.

```
mean(unicorns$height)
```

```
[1] 6.839583
```

```
summary(unicorns$height)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.200	4.475	6.450	6.840	9.025	17.200

Just as we did with vectors (Section 2.5), we also can access data in data frames using the square bracket [] notation. However, instead of just using a single index, we now need to use two indexes, one to specify the rows and one for the columns. To do this, we can use the notation `my_data[rows, columns]` where `rows` and `columns` are indexes and `my_data` is the name of the data frame. Again, just like with our vectors our indexes can be positional or the result of a logical test.

3.4.1. Positional indexes

To use positional indexes we simple have to write the position of the rows and columns we want to extract inside the []. For example, if for some reason we wanted to extract the first value (1st row) of the height variable (4th column)

```
unicorns[1, 4]
```

```
[1] 7.5
```

```
# this would give you the same
unicorns$height[1]
```

```
[1] 7.5
```

We can also extract values from multiple rows or columns by specifying these indexes as vectors inside the []. To extract the first 10 rows and the first 4 columns we simple supply a vector containing a sequence from 1 to 10 for the rows index (1:10) and a vector from 1 to 4 for the column index (1:4).

```
unicorns[1:10, 1:4]
```

p_care	food	block	height
care	medium	1	7.5
care	medium	1	10.7
care	medium	1	11.2
care	medium	1	10.4
care	medium	1	10.4
care	medium	1	9.8
care	medium	1	6.9
care	medium	1	9.4
care	medium	2	10.4
care	medium	2	12.3

Or for non sequential rows and columns then we can supply vectors of positions using the c() function. To extract the 1st, 5th, 12th, 30th rows from the 1st, 3rd, 6th and 8th columns

```
unicorns[c(1, 5, 12, 30), c(1, 3, 6, 8)]
```

	p_care	block	mane_size	horn_rings
1	care	1	11.7	1
5	care	1	14.5	4
12	care	2	12.6	6
30	care	2	11.6	5

All we are doing in the two examples above is creating vectors of positions for the rows and columns that we want to extract. We have done this by using the skills we developed in Section 2.4 when we generated vectors using the `c()` function or using the `:` notation.

But what if we want to extract either all of the rows or all of the columns? It would be extremely tedious to have to generate vectors for all rows or for all columns. Thankfully R has a shortcut. If you don't specify either a row or column index in the `[]` then R interprets it to mean you want all rows or all columns. For example, to extract the first 4 rows and all of the columns in the `unicorns` data frame

```
unicorns[1:4, ]
```

p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
care	medium	1	7.5	7.62	11.7	31.9	1
care	medium	1	10.7	12.14	14.1	46.0	10
care	medium	1	11.2	12.76	7.1	66.7	10
care	medium	1	10.4	8.78	11.9	20.3	1

or all of the rows and the first 3 columns¹.

```
unicorns[, 1:3]
```

	p_care	food	block
1	care	medium	1

¹For space and simplicity we are just showing the first and last five rows

	p_care	food	block
2	care	medium	1
3	care	medium	1
4	care	medium	1
5	care	medium	1
92	no_care	low	2
93	no_care	low	2
94	no_care	low	2
95	no_care	low	2
96	no_care	low	2

We can even use negative positional indexes to exclude certain rows and columns. As an example, lets extract all of the rows except the first 85 rows and all columns except the 4th, 7th and 8th columns. Notice we need to use -() when we generate our row positional vectors. If we had just used -1:85 this would actually generate a regular sequence from -1 to 85 which is not what we want (we can of course use -1:-85).

```
unicorns[-(1:85), -c(4, 7, 8)]
```

	p_care	food	block	weight	mane_size
86	no_care	low	1	6.01	17.6
87	no_care	low	1	9.93	12.0
88	no_care	low	1	7.03	7.9
89	no_care	low	2	9.10	14.5
90	no_care	low	2	9.05	9.6
91	no_care	low	2	8.10	10.5
92	no_care	low	2	7.45	14.1
93	no_care	low	2	9.19	12.4
94	no_care	low	2	8.92	11.6
95	no_care	low	2	8.44	13.5
96	no_care	low	2	10.60	16.2

In addition to using a positional index for extracting particular columns (variables) we can also name the variables

directly when using the square bracket [] notation. For example, let's extract the first 5 rows and the variables `care`, `food` and `mane_size`. Instead of using `unicorns[1:5, c(1, 2, 6)]` we can instead use

```
unicorns[1:5, c("p_care", "food", "mane_size")]
```

	p_care	food	mane_size
	care	medium	11.7
	care	medium	14.1
	care	medium	7.1
	care	medium	11.9
	care	medium	14.5

We often use this method in preference to the positional index for selecting columns as it will still give us what we want even if we've changed the order of the columns in our data frame for some reason.

3.4.2. Logical indexes

Just as we did with vectors, we can also extract data from our data frame based on a logical test. We can use all of the logical operators that we used for our vector examples so if these have slipped your mind maybe have a look at Section 2.5.1.1 and refresh your memory. Let's extract all rows where `height` is greater than 12 and extract all columns by default (remember, if you don't include a column index after the comma it means all columns).

```
big_unicorns <- unicorns[unicorns$height > 12, ]
big_unicorns
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
10	care	medium	2	12.3	13.48	16.1	36.9	8
17	care	high	1	12.6	18.66	18.6	54.0	9
21	care	high	1	14.1	19.12	13.1	113.2	13
32	care	high	2	17.2	19.20	10.9	89.9	14
38	care	low	1	12.3	11.27	13.7	28.7	5

Notice in the code above that we need to use the `unicorns$height` notation for the logical test. If we just named the `height` variable without the name of the data frame we would receive an error telling us R couldn't find the

variable `height`. The reason for this is that the `height` variable only exists inside the `unicorns` data frame so you need to tell R exactly where it is.

```
big_unicorns <- unicorns[height > 12, ]
Error in `^`[.data.frame^(unicorns, height > 12, ) :
  object 'height' not found
```

So how does this work? The logical test is `unicorns$height > 12` and R will only extract those rows that satisfy this logical condition. If we look at the output of just the logical condition you can see this returns a vector containing `TRUE` if `height` is greater than 12 and `FALSE` if `height` is not greater than 12.

```
unicorns$height > 12
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
[13] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
[37] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[49] FALSE FALSE
[61] FALSE FALSE
[73] FALSE FALSE
[85] FALSE FALSE
```

So our row index is a vector containing either `TRUE` or `FALSE` values and only those rows that are `TRUE` are selected.

Other commonly used operators are shown below

```
unicorns[unicorns$height >= 6, ] # values greater or equal to 6

unicorns[unicorns$height <= 6, ] # values less than or equal to 6

unicorns[unicorns$height == 8, ] # values equal to 8

unicorns[unicorns$height != 8, ] # values not equal to 8
```

We can also extract rows based on the value of a character string or factor level. Let's extract all rows where the food level is equal to `high` (again we will output all columns). Notice that the double equals `==` sign must be used for a logical test and that the character string must be enclosed in either single or double quotes (i.e. `"high"`).

```
food_high <- unicorns[unicorns$food == "high", ]
rbind(head(food_high, n = 10), tail(food_high, n = 10))
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
17	care	high	1	12.6	18.66	18.6	54.0	9
18	care	high	1	10.0	18.07	16.9	90.5	3
19	care	high	1	10.0	13.29	15.8	142.7	12
20	care	high	1	8.5	14.33	13.2	91.4	5
21	care	high	1	14.1	19.12	13.1	113.2	13
22	care	high	1	10.1	15.49	12.6	77.2	12
23	care	high	1	8.5	17.82	20.5	54.4	3
24	care	high	1	6.5	17.13	24.1	147.4	6
25	care	high	2	11.5	23.89	14.3	101.5	12
26	care	high	2	7.7	14.77	17.2	104.5	4
71	no_care	high	1	7.2	15.21	15.9	135.0	14
72	no_care	high	1	2.1	19.15	15.6	176.7	6
73	no_care	high	2	4.7	13.42	19.8	124.7	5
74	no_care	high	2	5.0	16.82	17.3	182.5	15
75	no_care	high	2	6.5	14.00	10.1	126.5	7
76	no_care	high	2	2.6	18.88	16.4	181.5	14
77	no_care	high	2	6.0	13.68	16.2	133.7	2
78	no_care	high	2	9.3	18.75	18.4	181.1	16
79	no_care	high	2	4.6	14.65	16.7	91.7	11
80	no_care	high	2	5.2	17.70	19.1	181.1	8

Or we can extract all rows where food level is not equal to `medium` (using `!=`) and only return columns 1 to 4.

```
food_not_medium <- unicorns[unicorns$food != "medium", 1:4]
rbind(head(food_not_medium, n = 10), tail(food_not_medium, n = 10))
```

	p_care	food	block	height
17	care	high	1	12.6
18	care	high	1	10.0
19	care	high	1	10.0
20	care	high	1	8.5
21	care	high	1	14.1
22	care	high	1	10.1
23	care	high	1	8.5
24	care	high	1	6.5
25	care	high	2	11.5
26	care	high	2	7.7
87	no_care	low	1	3.0
88	no_care	low	1	3.7
89	no_care	low	2	2.4
90	no_care	low	2	5.7
91	no_care	low	2	3.7
92	no_care	low	2	3.2
93	no_care	low	2	3.9
94	no_care	low	2	3.3
95	no_care	low	2	5.5
96	no_care	low	2	4.4

We can increase the complexity of our logical tests by combining them with [Boolean expressions](#) just as we did for vector objects. For example, to extract all rows where `height` is greater or equal to 6 AND `food` is equal to `medium` AND `care` is equal to `no_care` we combine a series of logical expressions with the `&` symbol.

```
low_no_care_heigh6 <- unicorns[unicorns$height >= 6 & unicorns$food == "medium" &
  unicorns$p_care == "no_care", ]
low_no_care_heigh6
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
51	no_care	medium	1	7.5	13.60	13.6	122.2	11

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
54	no_care	medium	1	8.5	10.04	12.3	113.6	4
61	no_care	medium	2	8.0	11.43	12.6	43.2	14

To extract rows based on an ‘OR’ Boolean expression we can use the `|` symbol. Let’s extract all rows where `height` is greater than 12.3 OR less than 2.2.

```
height2.2_12.3 <- unicorns[unicorns$height > 12.3 | unicorns$height < 2.2, ]
height2.2_12.3
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
17	care	high	1	12.6	18.66	18.6	54.0	9
21	care	high	1	14.1	19.12	13.1	113.2	13
32	care	high	2	17.2	19.20	10.9	89.9	14
62	no_care	medium	2	1.8	10.47	11.8	120.8	9
68	no_care	high	1	1.2	18.24	16.6	148.1	7
72	no_care	high	1	2.1	19.15	15.6	176.7	6
86	no_care	low	1	1.8	6.01	17.6	46.2	4

An alternative method of selecting parts of a data frame based on a logical expression is to use the `subset()` function instead of the `[]`. The advantage of using `subset()` is that you no longer need to use the `$` notation when specifying variables inside the data frame as the first argument to the function is the name of the data frame to be subsetted. The disadvantage is that `subset()` is less flexible than the `[]` notation.

```
care_med_2 <- subset(unicorns, p_care == "care" & food == "medium" & block == 2)
care_med_2
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
9	care	medium	2	10.4	10.48	10.5	57.8	5
10	care	medium	2	12.3	13.48	16.1	36.9	8
11	care	medium	2	10.4	13.18	11.1	56.8	12
12	care	medium	2	11.0	11.56	12.6	31.3	6

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
13	care	medium	2	7.1	8.16	29.6	9.7	2
14	care	medium	2	6.0	11.22	13.0	16.4	3
15	care	medium	2	9.0	10.20	10.8	90.1	6
16	care	medium	2	4.5	12.55	13.4	14.4	6

And if you only want certain columns you can use the `select =` argument.

```
uni_p_care <- subset(unicorns, p_care == "care" & food == "medium" & block == 2,
  select = c("p_care", "food", "mane_size")
)
uni_p_care
```

	p_care	food	mane_size
9	care	medium	10.5
10	care	medium	16.1
11	care	medium	11.1
12	care	medium	12.6
13	care	medium	29.6
14	care	medium	13.0
15	care	medium	10.8
16	care	medium	13.4

3.4.3. Ordering data frames

Remember when we used the function `order()` to order one vector based on the order of another vector (way back in Section 2.5.3). This comes in very handy if you want to reorder rows in your data frame. For example, if we want all of the rows in the data frame `unicorns` to be ordered in ascending value of `height` and output all columns by default.

```
height_ord <- unicorns[order(unicorns$height), ]
head(height_ord, n = 10)
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
68	no_care	high	1	1.2	18.24	16.6	148.1	7
62	no_care	medium	2	1.8	10.47	11.8	120.8	9
86	no_care	low	1	1.8	6.01	17.6	46.2	4
72	no_care	high	1	2.1	19.15	15.6	176.7	6
63	no_care	medium	2	2.2	10.70	15.3	97.1	7
84	no_care	low	1	2.2	9.97	9.6	63.1	2
82	no_care	low	1	2.3	7.28	13.8	32.8	6
89	no_care	low	2	2.4	9.10	14.5	78.7	8
56	no_care	medium	1	2.5	14.85	17.5	77.8	10
69	no_care	high	1	2.6	16.57	17.1	141.1	3

We can also order by descending order of a variable (i.e. `mane_size`) using the `decreasing = TRUE` argument.

```
mane_size_ord <- unicorns[order(unicorns$mane_size, decreasing = TRUE), ]
head(mane_size_ord, n = 10)
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
70	no_care	high	1	10.9	17.22	49.2	189.6	17
13	care	medium	2	7.1	8.16	29.6	9.7	2
24	care	high	1	6.5	17.13	24.1	147.4	6
65	no_care	high	1	8.5	22.53	20.8	166.9	16
23	care	high	1	8.5	17.82	20.5	54.4	3
66	no_care	high	1	8.5	17.33	19.8	184.4	12
73	no_care	high	2	4.7	13.42	19.8	124.7	5
80	no_care	high	2	5.2	17.70	19.1	181.1	8
17	care	high	1	12.6	18.66	18.6	54.0	9
49	no_care	medium	1	5.6	11.03	18.6	49.9	8

We can even order data frames based on multiple variables. For example, to order the data frame `unicorns` in ascending order of both `block` and `height`.

```
block_height_ord <- unicorns[order(unicorns$block, unicorns$height), ]
head(block_height_ord, n = 10)
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
68	no_care	high	1	1.2	18.24	16.6	148.1	7
86	no_care	low	1	1.8	6.01	17.6	46.2	4
72	no_care	high	1	2.1	19.15	15.6	176.7	6
84	no_care	low	1	2.2	9.97	9.6	63.1	2
82	no_care	low	1	2.3	7.28	13.8	32.8	6
56	no_care	medium	1	2.5	14.85	17.5	77.8	10
69	no_care	high	1	2.6	16.57	17.1	141.1	3
87	no_care	low	1	3.0	9.93	12.0	56.6	6
53	no_care	medium	1	3.5	12.93	16.6	109.3	3
88	no_care	low	1	3.7	7.03	7.9	36.7	5

What if we wanted to order `unicorns` by ascending order of `block` but descending order of `height`? We can use a simple trick by adding a `-` symbol before the `unicorns$height` variable when we use the `order()` function. This will essentially turn all of the `height` values negative which will result in reversing the order. Note, that this trick will only work with numeric variables.

```
block_revheight_ord <- unicorns[order(unicorns$block, -unicorns$height), ]
rbind(head(block_revheight_ord, n = 10), tail(block_revheight_ord, n = 10))
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
21	care	high	1	14.1	19.12	13.1	113.2	13
17	care	high	1	12.6	18.66	18.6	54.0	9
38	care	low	1	12.3	11.27	13.7	28.7	5
3	care	medium	1	11.2	12.76	7.1	66.7	10
70	no_care	high	1	10.9	17.22	49.2	189.6	17
2	care	medium	1	10.7	12.14	14.1	46.0	10
4	care	medium	1	10.4	8.78	11.9	20.3	1
5	care	medium	1	10.4	13.58	14.5	26.9	4

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
22	care	high	1	10.1	15.49	12.6	77.2	12
18	care	high	1	10.0	18.07	16.9	90.5	3
64	no_care	medium	2	3.9	12.97	17.0	97.5	5
93	no_care	low	2	3.9	9.19	12.4	52.6	9
91	no_care	low	2	3.7	8.10	10.5	60.5	6
94	no_care	low	2	3.3	8.92	11.6	55.2	6
92	no_care	low	2	3.2	7.45	14.1	38.1	4
42	care	low	2	3.1	8.74	16.1	39.1	3
76	no_care	high	2	2.6	18.88	16.4	181.5	14
89	no_care	low	2	2.4	9.10	14.5	78.7	8
63	no_care	medium	2	2.2	10.70	15.3	97.1	7
62	no_care	medium	2	1.8	10.47	11.8	120.8	9

If we wanted to do the same thing with a factor (or character) variable like `food` we would need to use the function `xtfrm()` for this variable inside our `order()` function.

```
block_revheight_ord <- unicorns[order(-xtfrm(unicorns$food), unicorns$height), ]
rbind(head(block_revheight_ord, n = 10), tail(block_revheight_ord, n = 10))
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
62	no_care	medium	2	1.8	10.47	11.8	120.8	9
63	no_care	medium	2	2.2	10.70	15.3	97.1	7
56	no_care	medium	1	2.5	14.85	17.5	77.8	10
53	no_care	medium	1	3.5	12.93	16.6	109.3	3
58	no_care	medium	2	3.9	9.07	9.6	90.4	7
64	no_care	medium	2	3.9	12.97	17.0	97.5	5
52	no_care	medium	1	4.1	12.58	13.9	136.6	11
16	care	medium	2	4.5	12.55	13.4	14.4	6
60	no_care	medium	2	4.5	13.68	14.8	125.5	9
55	no_care	medium	1	4.9	6.89	8.2	52.9	3
29	care	high	2	9.2	13.26	11.3	108.0	9

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
78	no_care	high	2	9.3	18.75	18.4	181.1	16
18	care	high	1	10.0	18.07	16.9	90.5	3
19	care	high	1	10.0	13.29	15.8	142.7	12
22	care	high	1	10.1	15.49	12.6	77.2	12
70	no_care	high	1	10.9	17.22	49.2	189.6	17
25	care	high	2	11.5	23.89	14.3	101.5	12
17	care	high	1	12.6	18.66	18.6	54.0	9
21	care	high	1	14.1	19.12	13.1	113.2	13
32	care	high	2	17.2	19.20	10.9	89.9	14

Notice that the `food` variable has been reverse ordered alphabetically and `height` has been ordered by increasing values within each level of `food`.

If we wanted to order the data frame by `food` but this time order it from `low` -> `medium` -> `high` instead of the default alphabetically (`high`, `low`, `medium`), we need to first change the order of our levels of the `food` factor in our data frame using the `factor()` function. Once we've done this we can then use the `order()` function as usual. Note, if you're reading the pdf version of this book, the output has been truncated to save space.

```
unicorns$food <- factor(unicorns$food,
  levels = c("low", "medium", "high")
)

food_ord <- unicorns[order(unicorns$food), ]
rbind(head(food_ord, n = 10), tail(food_ord, n = 10))
```

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
33	care	low	1	8.0	6.88	9.3	16.1	4
34	care	low	1	8.0	10.23	11.9	88.1	4
35	care	low	1	6.4	5.97	8.7	7.3	2
36	care	low	1	7.6	13.05	7.2	47.2	8
37	care	low	1	9.7	6.49	8.1	18.0	3
38	care	low	1	12.3	11.27	13.7	28.7	5
39	care	low	1	9.1	8.96	9.7	23.8	3

	p_care	food	block	height	weight	mane_size	fluffyness	horn_rings
40	care	low	1	8.9	11.48	11.1	39.4	7
41	care	low	2	7.4	10.89	13.3	9.5	5
42	care	low	2	3.1	8.74	16.1	39.1	3
71	no_care	high	1	7.2	15.21	15.9	135.0	14
72	no_care	high	1	2.1	19.15	15.6	176.7	6
73	no_care	high	2	4.7	13.42	19.8	124.7	5
74	no_care	high	2	5.0	16.82	17.3	182.5	15
75	no_care	high	2	6.5	14.00	10.1	126.5	7
76	no_care	high	2	2.6	18.88	16.4	181.5	14
77	no_care	high	2	6.0	13.68	16.2	133.7	2
78	no_care	high	2	9.3	18.75	18.4	181.1	16
79	no_care	high	2	4.6	14.65	16.7	91.7	11
80	no_care	high	2	5.2	17.70	19.1	181.1	8

3.4.4. Adding columns and rows

Sometimes it's useful to be able to add extra rows and columns of data to our data frames. There are multiple ways to achieve this (as there always is in R!) depending on your circumstances. To simply append additional rows to an existing data frame we can use the `rbind()` function and to append columns the `cbind()` function. Let's create a couple of test data frames to see this in action using our old friend the `data.frame()` function.

```
# rbind for rows
df1 <- data.frame(
  id = 1:4, height = c(120, 150, 132, 122),
  weight = c(44, 56, 49, 45)
)
df1
```

	id	height	weight
	1	120	44
	2	150	56

id	height	weight
3	132	49
4	122	45

```
df2 <- data.frame(
  id = 5:6, height = c(119, 110),
  weight = c(39, 35)
)
df2
```

id	height	weight
5	119	39
6	110	35

```
df3 <- data.frame(
  id = 1:4, height = c(120, 150, 132, 122),
  weight = c(44, 56, 49, 45)
)
df3
```

id	height	weight
1	120	44
2	150	56
3	132	49
4	122	45

```
df4 <- data.frame(location = c("UK", "CZ", "CZ", "UK"))
df4
```

location
UK

location

CZ

CZ

UK

We can use the `rbind()` function to append the rows of data in `df2` to the rows in `df1` and assign the new data frame to `df_rcomb`.

```
df_rcomb <- rbind(df1, df2)  
df_rcomb
```

	id	height	weight
	1	120	44
	2	150	56
	3	132	49
	4	122	45
	5	119	39
	6	110	35

And `cbind` to append the column in `df4` to the `df3` data frame and assign to `df_ccomb`:

```
df_ccomb <- cbind(df3, df4)  
df_ccomb
```

	id	height	weight	location
	1	120	44	UK
	2	150	56	CZ
	3	132	49	CZ
	4	122	45	UK

Another situation when adding a new column to a data frame is useful is when you want to perform some kind of transformation on an existing variable. For example, say we wanted to apply a \log_{10} transformation on the height

variable in the `df_rcomb` data frame we created above. We could just create a separate variable to contains these values but it's good practice to create this variable as a new column inside our existing data frame so we keep all of our data together. Let's call this new variable `height_log10`.

```
# log10 transformation
df_rcomb$height_log10 <- log10(df_rcomb$height)
df_rcomb
```

	<code>id</code>	<code>height</code>	<code>weight</code>	<code>height_log10</code>
	1	120	44	2.079181
	2	150	56	2.176091
	3	132	49	2.120574
	4	122	45	2.086360
	5	119	39	2.075547
	6	110	35	2.041393

This situation also crops up when we want to convert an existing variable in a data frame from one data class to another data class. For example, the `id` variable in the `df_rcomb` data frame is numeric type data (use the `str()` or `class()` functions to check for yourself). If we wanted to convert the `id` variable to a factor to use later in our analysis we can create a new variable called `Fid` in our data frame and use the `factor()` function to convert the `id` variable.

```
# convert to a factor
df_rcomb$Fid <- factor(df_rcomb$id)

df_rcomb
```

	<code>id</code>	<code>height</code>	<code>weight</code>	<code>height_log10</code>	<code>Fid</code>
	1	120	44	2.079181	1
	2	150	56	2.176091	2
	3	132	49	2.120574	3
	4	122	45	2.086360	4
	5	119	39	2.075547	5
	6	110	35	2.041393	6

```
str(df_rcomb)
```

```
'data.frame': 6 obs. of 5 variables:  
 $ id : int 1 2 3 4 5 6  
 $ height : num 120 150 132 122 119 110  
 $ weight : num 44 56 49 45 39 35  
 $ height_log10: num 2.08 2.18 2.12 2.09 2.08 ...  
 $ Fid : Factor w/ 6 levels "1","2","3","4",...: 1 2 3 4 5 6
```

3.4.5. Merging data frames

Instead of just appending either rows or columns to a data frame we can also merge two data frames together. Let's say we have one data frame that contains taxonomic information on some common UK rocky shore invertebrates (called `taxa`) and another data frame that contains information on where they are usually found on the rocky shore (called `zone`). We can merge these two data frames together to produce a single data frame with both taxonomic and location information. Let's first create both of these data frames (in reality you would probably just import your different datasets).

```
taxa <- data.frame(  
  GENUS = c("Patella", "Littorina", "Halichondria", "Semibalanus"),  
  species = c("vulgata", "littoria", "panacea", "balanoides"),  
  family = c("patellidae", "Littorinidae", "Halichondriidae", "Archaeobalanidae")  
)  
taxa
```

GENUS	species	family
Patella	vulgata	patellidae
Littorina	littoria	Littorinidae
Halichondria	panacea	Halichondriidae
Semibalanus	balanoides	Archaeobalanidae

```

zone <- data.frame(
  genus = c(
    "Laminaria", "Halichondria", "Xanthoria", "Littorina",
    "Semibalanus", "Fucus"
  ),
  species = c(
    "digitata", "panacea", "parietina", "littoria",
    "balanoides", "serratus"
  ),
  zone = c("v_low", "low", "v_high", "low_mid", "high", "low_mid")
)
zone

```

genus	species	zone
Laminaria	digitata	v_low
Halichondria	panacea	low
Xanthoria	parietina	v_high
Littorina	littoria	low_mid
Semibalanus	balanoides	high
Fucus	serratus	low_mid

Because both of our data frames contains at least one variable in common (`species` in our case) we can simply use the `merge()` function to create a new data frame called `taxa_zone`.

```

taxa_zone <- merge(x = taxa, y = zone)
taxa_zone

```

species	GENUS	family	genus	zone
balanoides	Semibalanus	Archaeobalanidae	Semibalanus	high
littoria	Littorina	Littorinidae	Littorina	low_mid
panacea	Halichondria	Halichondriidae	Halichondria	low

Notice that the merged data frame contains only the rows that have `species` information in **both** data frames. There are also two columns called `GENUS` and `genus` because the `merge()` function treats these as two different variables that originate from the two data frames.

If we want to include all data from both data frames then we will need to use the `all = TRUE` argument. The missing values will be included as `NA`.

```
taxa_zone <- merge(x = taxa, y = zone, all = TRUE)
taxa_zone
```

species	GENUS	family	genus	zone
balanoides	Semibalanus	Archaeobalanidae	Semibalanus	high
digitata	NA	NA	Laminaria	v_low
littoria	Littorina	Littorinidae	Littorina	low_mid
panacea	Halichondria	Halichondriidae	Halichondria	low
parietina	NA	NA	Xanthoria	v_high
serratus	NA	NA	Fucus	low_mid
vulgata	Patella	patellidae	NA	NA

If the variable names that you want to base the merge on are different in each data frame (for example `GENUS` and `genus`) you can specify the names in the first data frame (known as `x`) and the second data frame (known as `y`) using the `by.x =` and `by.y =` arguments.

```
taxa_zone <- merge(x = taxa, y = zone, by.x = "GENUS", by.y = "genus", all = TRUE)
taxa_zone
```

GENUS	species.x	family	species.y	zone
Fucus	NA	NA	serratus	low_mid
Halichondria	panacea	Halichondriidae	panacea	low
Laminaria	NA	NA	digitata	v_low
Littorina	littoria	Littorinidae	littoria	low_mid
Patella	vulgata	patellidae	NA	NA
Semibalanus	balanoides	Archaeobalanidae	balanoides	high

GENUS	species.x	family	species.y	zone
Xanthoria	NA	NA	parietina	v_high

Or using multiple variable names.

```
taxa_zone <- merge(
  x = taxa, y = zone, by.x = c("species", "GENUS"),
  by.y = c("species", "genus"), all = TRUE
)
taxa_zone
```

species	GENUS	family	zone
balanoides	Semibalanus	Archaeobalanidae	high
digitata	Laminaria	NA	v_low
littoria	Littorina	Littorinidae	low_mid
panacea	Halichondria	Halichondriidae	low
parietina	Xanthoria	NA	v_high
serratus	Fucus	NA	low_mid
vulgata	Patella	patellidae	NA

3.4.6. Reshaping data frames

Reshaping data into different formats is a common task. With rectangular type data (data frames have the same number of rows in each column) there are two main data frame shapes that you will come across: the ‘long’ format (sometimes called stacked) and the ‘wide’ format. An example of a long format data frame is given below. We can see that each row is a single observation from an individual subject and each subject can have multiple rows. This results in a single column of our measurement.

```
long_data <- data.frame(
  subject = rep(c("A", "B", "C", "D"), each = 3),
  sex = rep(c("M", "F", "F", "M"), each = 3),
  condition = rep(c("control", "cond1", "cond2"), times = 4),
```

```
measurement = c(  
  12.9, 14.2, 8.7, 5.2, 12.6, 10.1, 8.9,  
  12.1, 14.2, 10.5, 12.9, 11.9  
)  
)  
long_data
```

subject	sex	condition	measurement
A	M	control	12.9
A	M	cond1	14.2
A	M	cond2	8.7
B	F	control	5.2
B	F	cond1	12.6
B	F	cond2	10.1
C	F	control	8.9
C	F	cond1	12.1
C	F	cond2	14.2
D	M	control	10.5
D	M	cond1	12.9
D	M	cond2	11.9

We can also format the same data in the wide format as shown below. In this format we have multiple observations from each subject in a single row with measurements in different columns (`control`, `cond1` and `cond2`). This is a common format when you have repeated measurements from sampling units.

```
wide_data <- data.frame(  
  subject = c("A", "B", "C", "D"),  
  sex = c("M", "F", "F", "M"),  
  control = c(12.9, 5.2, 8.9, 10.5),  
  cond1 = c(14.2, 12.6, 12.1, 12.9),  
  cond2 = c(8.7, 10.1, 14.2, 11.9)  
)  
wide_data
```

subject	sex	control	cond1	cond2
A	M	12.9	14.2	8.7
B	F	5.2	12.6	10.1
C	F	8.9	12.1	14.2
D	M	10.5	12.9	11.9

Whilst there's no inherent problem with either of these formats we will sometimes need to convert between the two because some functions will require a specific format for them to work. The most common format is the long format.

There are many ways to convert between these two formats but we'll use the `melt()` and `dcast()` functions from the `reshape2` package (you will need to install this package first). The `melt()` function is used to convert from wide to long formats. The first argument for the `melt()` function is the data frame we want to melt (in our case `wide_data`). The `id.vars = c("subject", "sex")` argument is a vector of the variables you want to stack, the `measured.vars = c("control", "cond1", "cond2")` argument identifies the columns of the measurements in different conditions, the `variable.name = "condition"` argument specifies what you want to call the stacked column of your different conditions in your output data frame and `value.name = "measurement"` is the name of the column of your stacked measurements in your output data frame.

```
library(reshape2)
wide_data # remind ourselves what the wide format looks like
```

subject	sex	control	cond1	cond2
A	M	12.9	14.2	8.7
B	F	5.2	12.6	10.1
C	F	8.9	12.1	14.2
D	M	10.5	12.9	11.9

```
# convert wide to long
my_long_df <- melt(
  data = wide_data, id.vars = c("subject", "sex"),
  measured.vars = c("control", "cond1", "cond2"),
  variable.name = "condition", value.name = "measurement"
```

```
)  
my_long_df
```

subject	sex	condition	measurement
A	M	control	12.9
B	F	control	5.2
C	F	control	8.9
D	M	control	10.5
A	M	cond1	14.2
B	F	cond1	12.6
C	F	cond1	12.1
D	M	cond1	12.9
A	M	cond2	8.7
B	F	cond2	10.1
C	F	cond2	14.2
D	M	cond2	11.9

The `dcast()` function is used to convert from a long format data frame to a wide format data frame. The first argument is again is the data frame we want to cast (`long_data` for this example). The second argument is in formula syntax. The `subject + sex` bit of the formula means that we want to keep these columns separate, and the `~ condition` part is the column that contains the labels that we want to split into new columns in our new data frame. The `value.var = "measurement"` argument is the column that contains the measured data.

```
long_data # remind ourselves what the long format look like
```

subject	sex	condition	measurement
A	M	control	12.9
A	M	cond1	14.2
A	M	cond2	8.7
B	F	control	5.2
B	F	cond1	12.6
B	F	cond2	10.1

subject	sex	condition	measurement
C	F	control	8.9
C	F	cond1	12.1
C	F	cond2	14.2
D	M	control	10.5
D	M	cond1	12.9
D	M	cond2	11.9

```
# convert long to wide
my_wide_df <- dcast(
  data = long_data, subject + sex ~ condition,
  value.var = "measurement"
)
my_wide_df
```

subject	sex	cond1	cond2	control
A	M	14.2	8.7	12.9
B	F	12.6	10.1	5.2
C	F	12.1	14.2	8.9
D	M	12.9	11.9	10.5

3.5. Introduction to the tidyverse

it seems it is not super tidy in here and we need to improve that

3.6. Summarising data frames

Now that we're able to manipulate and extract data from our data frames our next task is to start exploring and getting to know our data. In this section we'll start producing tables of useful summary statistics of the variables in our data frame and in the next two Chapters we'll cover visualising our data with base R graphics and using the `ggplot2` package.

A really useful starting point is to produce some simple summary statistics of all of the variables in our `unicorns` data frame using the `summary()` function.

```
summary(unicorns)
```

p_care	food	block	height	weight
Length:96	low :32	Min. :1.0	Min. : 1.200	Min. : 5.790
Class :character	medium:32	1st Qu.:1.0	1st Qu.: 4.475	1st Qu.: 9.027
Mode :character	high :32	Median :1.5	Median : 6.450	Median :11.395
		Mean :1.5	Mean : 6.840	Mean :12.155
		3rd Qu.:2.0	3rd Qu.: 9.025	3rd Qu.:14.537
		Max. :2.0	Max. :17.200	Max. :23.890
mane_size	fluffyness	horn_rings		
Min. : 5.80	Min. : 5.80	Min. : 1.000		
1st Qu.:11.07	1st Qu.: 39.05	1st Qu.: 4.000		
Median :13.45	Median : 70.05	Median : 6.000		
Mean :14.05	Mean : 79.78	Mean : 7.062		
3rd Qu.:16.45	3rd Qu.:113.28	3rd Qu.: 9.000		
Max. :49.20	Max. :189.60	Max. :17.000		

For numeric variables (i.e. `height`, `weight` etc) the mean, minimum, maximum, median, first (lower) quartile and third (upper) quartile are presented. For factor variables (i.e. `care` and `food`) the number of observations in each of the factor levels is given. If a variable contains missing data then the number of `NA` values is also reported.

If we wanted to summarise a smaller subset of variables in our data frame we can use our indexing skills in combination with the `summary()` function. For example, to summarise only the `height`, `weight`, `mane_size` and `fluffyness` variables we can include the appropriate column indexes when using the `[]`. Notice we include all rows by not specifying a row index.

```
summary(unicorns[, 4:7])
```

height	weight	mane_size	fluffyness
Min. : 1.200	Min. : 5.790	Min. : 5.80	Min. : 5.80
1st Qu.: 4.475	1st Qu.: 9.027	1st Qu.:11.07	1st Qu.: 39.05
Median : 6.450	Median :11.395	Median :13.45	Median : 70.05

```
Mean      : 6.840    Mean     :12.155    Mean     :14.05    Mean     : 79.78
3rd Qu.: 9.025    3rd Qu.:14.537    3rd Qu.:16.45    3rd Qu.:113.28
Max.     :17.200    Max.     :23.890    Max.     :49.20    Max.     :189.60
```

```
# or equivalently
# summary(unicorns[, c("height", "weight", "mane_size", "fluffyness")])
```

And to summarise a single variable.

```
summary(unicorns$mane_size)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
5.80	11.07	13.45	14.05	16.45	49.20

```
# or equivalently
# summary(unicorns[, 6])
```

As you've seen above, the `summary()` function reports the number of observations in each level of our factor variables. Another useful function for generating tables of counts is the `table()` function. The `table()` function can be used to build contingency tables of different combinations of factor levels. For example, to count the number of observations for each level of `food`

```
table(unicorns$food)
```

low	medium	high
32	32	32

We can extend this further by producing a table of counts for each combination of `food` and `care` factor levels.

```
table(unicorns$food, unicorns$p_care)
```

	care	no_care
low	16	16
medium	16	16
high	16	16

A more flexible version of the `table()` function is the `xtabs()` function. The `xtabs()` function uses a formula notation (~) to build contingency tables with the cross-classifying variables separated by a + symbol on the right hand side of the formula. `xtabs()` also has a useful `data` = argument so you don't have to include the data frame name when specifying each variable.

```
xtabs(~ food + p_care, data = unicorns)
```

```
      p_care  
food      care no_care  
low       16    16  
medium    16    16  
high      16    16
```

We can even build more complicated contingency tables using more variables. Note, in the example below the `xtabs()` function has quietly coerced our `block` variable to a factor.

```
xtabs(~ food + p_care + block, data = unicorns)
```

```
, , block = 1  
  
      p_care  
food      care no_care  
low       8     8  
medium    8     8  
high      8     8
```

```
, , block = 2  
  
      p_care
```

```
food      care no_care  
low       8     8  
medium    8     8  
high      8     8
```

And for a nicer formatted table we can nest the `xtabs()` function inside the `ftable()` function to ‘flatten’ the table.

```
ftable(xtabs(~ food + p_care + block, data = unicorns))
```

		block 1 2	
food	p_care		
low	care	8	8
	no_care	8	8
medium	care	8	8
	no_care	8	8
high	care	8	8
	no_care	8	8

We can also summarise our data for each level of a factor variable. Let’s say we want to calculate the mean value of `height` for each of our `low`, `medium` and `high` levels of `food`. To do this we will use the `mean()` function and apply this to the `height` variable for each level of `food` using the `tapply()` function.

```
tapply(unicorns$height, unicorns$food, mean)
```

	low	medium	high
5.853125	5.853125	7.012500	7.653125

The `tapply()` function is not just restricted to calculating mean values, you can use it to apply many of the functions that come with R or even functions you’ve written yourself (see Chapter 5 for more details). For example, we can apply the `sd()` function to calculate the standard deviation for each level of `food` or even the `summary()` function.

```
tapply(unicorns$height, unicorns$food, sd)
```

	low	medium	high
2.828425	2.828425	3.005345	3.483323

```
tapply(unicorns$height, unicorns$food, summary)
```

```
$low
  Min. 1st Qu. Median   Mean 3rd Qu.   Max.
1.800  3.600  5.550  5.853  8.000 12.300
```

```
$medium
  Min. 1st Qu. Median   Mean 3rd Qu.   Max.
1.800  4.500  7.000  7.013  9.950 12.300
```

```
$high
  Min. 1st Qu. Median   Mean 3rd Qu.   Max.
1.200  5.800  7.450  7.653  9.475 17.200
```

Note, if the variable you want to summarise contains missing values (NA) you will also need to include an argument specifying how you want the function to deal with the NA values. We saw an example of this in Section 2.5.5 where the `mean()` function returned an NA when we had missing data. To include the `na.rm = TRUE` argument we simply add this as another argument when using `tapply()`.

```
tapply(unicorns$height, unicorns$food, mean, na.rm = TRUE)
```

```
low    medium      high
5.853125 7.012500 7.653125
```

We can also use `tapply()` to apply functions to more than one factor. The only thing to remember is that the factors need to be supplied to the `tapply()` function in the form of a list using the `list()` function. To calculate the mean height for each combination of food and care factor levels we can use the `list(unicorns$food, unicorns$p_care)` notation.

```
tapply(unicorns$height, list(unicorns$food, unicorns$p_care), mean)
```

```
care no_care
low     8.0375 3.66875
medium  9.1875 4.83750
high    9.6000 5.70625
```

And if you get a little fed up with having to write `unicorns$` for every variable you can nest the `tapply()` function inside the `with()` function. The `with()` function allows R to evaluate an R expression with respect to a named data object (in this case `unicorns`).

```
with(unicorns, tapply(height, list(food, p_care), mean))
```

	care	no_care
low	8.0375	3.66875
medium	9.1875	4.83750
high	9.6000	5.70625

The `with()` function also works with many other functions and can save you a lot of typing!

Another really useful function for summarising data is the `aggregate()` function. The `aggregate()` function works in a very similar way to `tapply()` but is a bit more flexible.

For example, to calculate the mean of the variables `height`, `weight`, `mane_size` and `fluffyness` for each level of `food`.

```
aggregate(unicorns[, 4:7], by = list(food = unicorns$food), FUN = mean)
```

food	height	weight	mane_size	fluffyness
low	5.853125	8.652812	11.14375	45.1000
medium	7.012500	11.164062	13.83125	67.5625
high	7.653125	16.646875	17.18125	126.6875

In the code above we have indexed the columns we want to summarise in the `unicorns` data frame using `unicorns[, 4:7]`. The `by =` argument specifies a list of factors (`list(food = unicorns$food)`) and the `FUN =` argument names the function to apply (`mean` in this example).

Similar to the `tapply()` function we can include more than one factor to apply a function to. Here we calculate the mean values for each combination of `food` and `care`

```
aggregate(unicorns[, 4:7], by = list(
  food = unicorns$food,
  p_care = unicorns$p_care
), FUN = mean)
```

food	p_care	height	weight	mane_size	fluffyness
low	care	8.03750	9.016250	9.96250	30.30625
medium	care	9.18750	11.011250	13.48750	40.59375
high	care	9.60000	16.689375	15.54375	98.05625
low	no_care	3.66875	8.289375	12.32500	59.89375
medium	no_care	4.83750	11.316875	14.17500	94.53125
high	no_care	5.70625	16.604375	18.81875	155.31875

We can also use the `aggregate()` function in a different way by using the formula method (as we did with `xtabs()`). On the left hand side of the formula (`~`) we specify the variable we want to apply the mean function on and to the right hand side our factors separated by a `+` symbol. The formula method also allows you to use the `data = argument` for convenience.

```
aggregate(height ~ food + p_care, FUN = mean, data = unicorns)
```

food	p_care	height
low	care	8.03750
medium	care	9.18750
high	care	9.60000
low	no_care	3.66875
medium	no_care	4.83750
high	no_care	5.70625

One advantage of using the formula method is that we can also use the `subset = argument` to apply the function to subsets of the original data. For example, to calculate the mean `height` for each combination of the `food` and `care` levels but only for those unicorns that have less than 7 `horn_rings`.

```
aggregate(height ~ food + p_care, FUN = mean, subset = horn_rings < 7, data = unicorns)
```

food	p_care	height
low	care	8.176923
medium	care	8.570000

food	p_care	height
high	care	7.900000
low	no_care	3.533333
medium	no_care	5.316667
high	no_care	3.850000

Or for only those unicorns in block 1.

```
aggregate(height ~ food + p_care, FUN = mean, subset = block == "1", data = unicorns)
```

food	p_care	height
low	care	8.7500
medium	care	9.5375
high	care	10.0375
low	no_care	3.3250
medium	no_care	5.2375
high	no_care	5.9250

3.7. Exporting data

By now we hope you're getting a feel for how powerful and useful R is for manipulating and summarising data (and we've only really scratched the surface). One of the great benefits of doing all your data wrangling in R is that you have a permanent record of all the things you've done to your data. Gone are the days of making undocumented changes in Excel or Calc! By treating your data as 'read only' and documenting all of your decisions in R you will have made great strides towards making your analysis more reproducible and transparent to others. It's important to realise, however, that any changes you've made to your data frame in R will not change the original data file you imported into R (and that's a good thing). Happily it's straightforward to export data frames to external files in a wide variety of formats.

3.7.1. Export functions

The main workhorse function for exporting data frames is the `write.table()` function. As with the `read.table()` function, the `write.table()` function is very flexible with lots of arguments to help customise its behaviour. As

an example, let's take our original `unicorns` data frame, do some useful stuff to it and then export these changes to an external file.

Similarly to `read.table()`, `write.table()` has a series of function with format specific default values such as `write.csv()` and `write.delim()` which use “,” and tabs as delimiters, respectively, and include column names by default.

Let's order the rows in the data frame in ascending order of `height` within each level `food`. We will also apply a square root transformation on the number of horn rings variable (`horn_rings`) and a \log_{10} transformation on the `height` variable and save these as additional columns in our data frame (hopefully this will be somewhat familiar to you!).

```
unicorns_df2 <- unicorns[order(unicorns$food, unicorns$height), ]  
unicorns_df2$horn_rings_sqrt <- sqrt(unicorns_df2$horn_rings)  
unicorns_df2$log10_height <- log10(unicorns_df2$height)  
str(unicorns_df2)
```

```
'data.frame': 96 obs. of 10 variables:  
 $ p_care       : chr  "no_care" "no_care" "no_care" "no_care" ...  
 $ food         : Factor w/ 3 levels "low","medium",...: 1 1 1 1 1 1 1 1 1 1 ...  
 $ block        : int  1 1 1 2 1 2 2 2 1 2 ...  
 $ height       : num  1.8 2.2 2.3 2.4 3 3.1 3.2 3.3 3.7 3.7 ...  
 $ weight       : num  6.01 9.97 7.28 9.1 9.93 8.74 7.45 8.92 7.03 8.1 ...  
 $ mane_size    : num  17.6 9.6 13.8 14.5 12 16.1 14.1 11.6 7.9 10.5 ...  
 $ fluffyness   : num  46.2 63.1 32.8 78.7 56.6 39.1 38.1 55.2 36.7 60.5 ...  
 $ horn_rings   : int  4 2 6 8 6 3 4 6 5 6 ...  
 $ horn_rings_sqrt: num  2 1.41 2.45 2.83 2.45 ...  
 $ log10_height : num  0.255 0.342 0.362 0.38 0.477 ...
```

Now we can export our new data frame `unicorns_df2` using the `write.table()` function. The first argument is the data frame you want to export (`unicorns_df2` in our example). We then give the filename (with file extension) and the file path in either single or double quotes using the `file =` argument. In this example we're exporting the data frame to a file called `unicorns_transformed.csv` in the `data` directory. The `row.names = FALSE` argument stops R from including the row names in the first column of the file.

```
write.csv(unicorns_df2,
  file = "data/unicorns_transformed.csv",
  row.names = FALSE
)
```

As we saved the file as a comma delimited text file we could open this file in any text editor.

We can of course export our files in a variety of other formats.

3.7.2. Other export functions

As with importing data files into R, there are also many alternative functions for exporting data to external files beyond the `write.table()` function. If you followed the ‘Other import functions’ Section 3.3.4 of this Chapter you will already have the required packages installed.

The `fwrite()` function from the `data.table`  package is very efficient at exporting large data objects and is much faster than the `write.table()` function. It’s also quite simple to use as it has most of the same arguments as `write.table()`. To export a tab delimited text file we just need to specify the data frame name, the output file name and file path and the separator between columns.

```
library(data.table)
fwrite(unicorns_df2, file = "data/unicorns_04_12.txt", sep = "\t")
```

To export a csv delimited file it’s even easier as we don’t even need to include the `sep =` argument.

```
library(data.table)
fwrite(unicorns_df2, file = "data/unicorns_04_12.csv")
```

The `readr` package also comes with two useful functions for quickly writing data to external files: the `write_tsv()` function for writing tab delimited files and the `write_csv()` function for saving comma separated values (csv) files.

```
library(readr)
write_tsv(unicorns_df2, path = "data/unicorns_04_12.txt")

write_csv(unicorns_df2, path = "data/unicorns_04_12.csv")
```

Chapter 4

Figures

Summarising your data, either numerically or graphically, is an important (if often overlooked) component of any data analysis. Fortunately, R has excellent graphics capabilities and can be used whether you want to produce plots for initial data exploration, model validation or highly complex publication quality figures. There are three main systems for producing graphics in R; base R graphics, lattice graphics and ggplot2.

The base R graphics system is the original plotting system that's been around (and has evolved) since the first days of R. When creating plots with base R we tend to use high level functions (like the `plot()` function) to first create our plot and then use one or more low level functions (like `lines()` and `text()` etc) to add additional information to these plots. This can seem a little weird (and time consuming) when you first start creating fancy plots in R, but it does allow you to customise almost every aspect of your plot and build complexity up in layers. The flip side to this flexibility is that you'll often need to make many decisions about how you want your plot to look rather than rely on the software to make these decisions for you. Having said that, it's generally very quick and easy to generate simple exploratory plots with base R graphics.

The lattice system is implemented in the `lattice`  package that comes pre-installed with the standard installation of R. However, it won't be loaded by default so you'll first need to use `library(lattice)` to access all the plotting functions. Unlike base R graphics, lattice plots are mostly generated all in one go using a single function so there's no need to use high and low level plotting functions to customise the look of a plot. This can be a real advantage as things like margin sizes and plot spacing are adjusted automatically. Lattice plots also make a few more decisions for you about how the plots will look but this comes with a slight cost as customising lattice plots to get them to look exactly how you want can become quite involved. Where lattice plots really shine is plotting complex multi-dimensional data using panel plots (also called trellis plots). We'll see a couple of examples of these types of plots later in the Chapter.

ggplot2 was based on a book called *Grammar of Graphics* by Wilkinson (2005). For an interesting summary of Wilkinson's book [here](#). The *Grammar of Graphics* approach breaks figures down into their various components (e.g. the underlying statistics, the geometric arrangement, the theme, see Figure 4.1). Users are thus able to manipulate each of these components (i.e. layers) and produce a tailor-made figure fit for their specific needs.

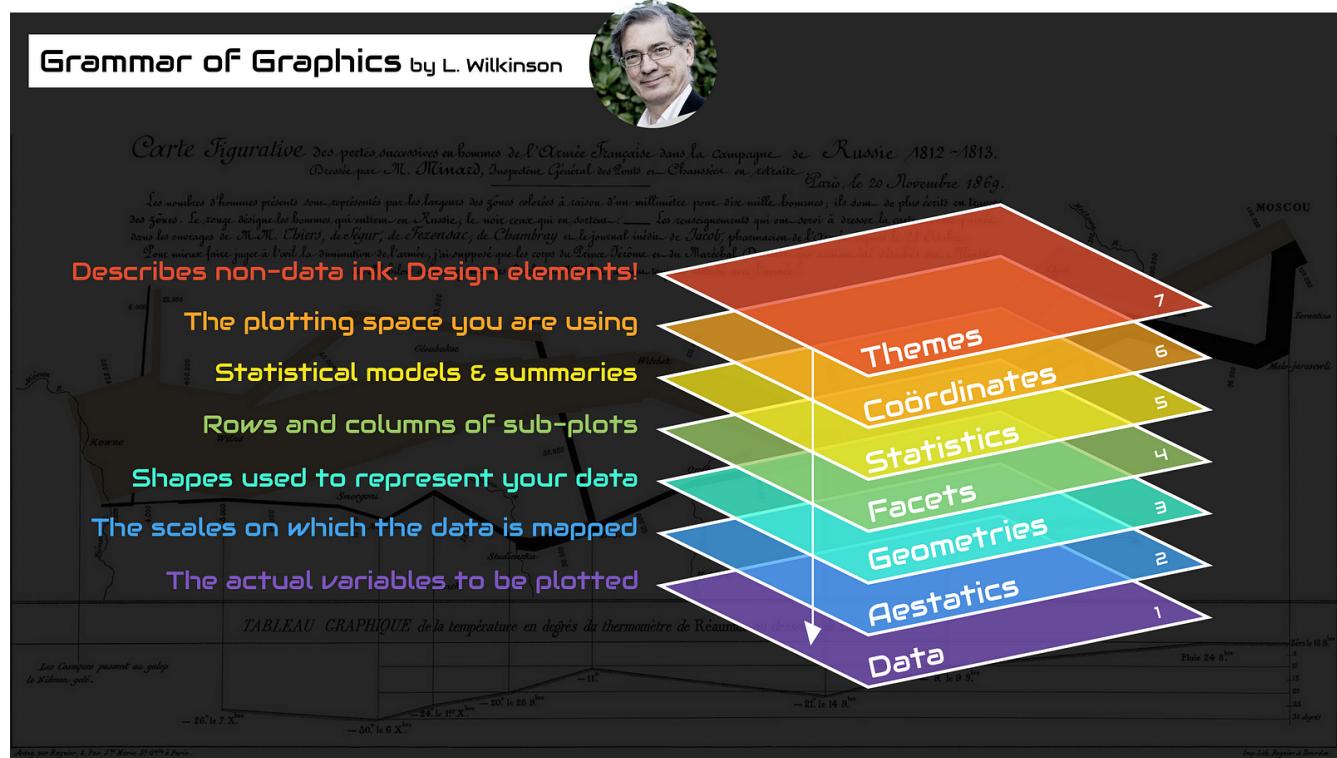


Figure 4.1.: The Grammar of Graphics. Visual by Thomas de Beus

Each of these systems have their strengths and weaknesses and we often use them interchangeably. In this Chapter we'll introduce you to the both base R plotting function and the ggplot2 package. It's important to note that ggplot2 is not **required** to make "fancy" and informative figures in R. If you prefer using base R graphics then feel free to continue as almost all ggplot2 type figures can be created using base R (we often use either approach depending on what we're doing). The difference between ggplot2 and base R is how you *get* to the end product rather than any substantial differences in the end product itself. This is, never-the-less, a common belief probably due to the fact that making a moderately attractive figure is (in our opinion at least), easier to do with ggplot2 as many aesthetic decisions are made for the user, without you necessarily even knowing that a decision was ever made!

With that in mind, let's get started making some figures.

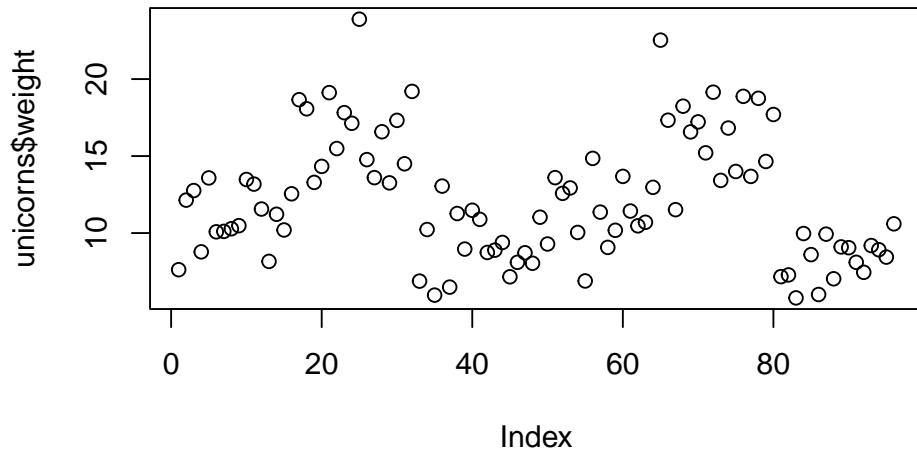
4.1. Simple base R plots

There are many functions in R to produce plots ranging from the very basic to the highly complex. It's impossible to cover every aspect of producing graphics in R in this book so we'll introduce you to most of the common methods of graphing data and describe how to customise your graphs later on in Section 4.5.

The most common high level function used to produce plots in R is (rather unsurprisingly) the `plot()` function. For example, let's plot the `weight` of unicorns from our `unicorns` data frame which we imported in Section 3.3.2.

```
unicorns <- read.csv(file = "data/unicorns.csv")

plot(unicorns$weight)
```



R has plotted the values of `weight` (on the y axis) against an index since we are only plotting one variable to plot. The index is just the order of the `weight` values in the data frame (1 first in the data frame and 97 last). The `weight` variable name has been automatically included as a y axis label and the axes scales have been automatically set.

If we'd only included the variable `weight` rather than `unicorns$weight`, the `plot()` function will display an error as the variable `weight` only exists in the `unicorns` data frame object.

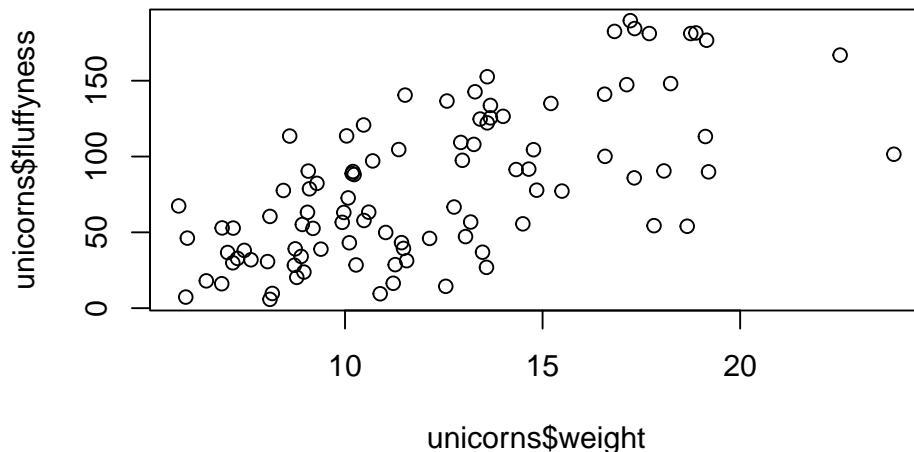
```
plot(weight)
## Error in plot(weight) : object 'weight' not found
```

As many of the base R plotting functions don't have a `data =` argument to specify the data frame name directly we can use the `with()` function in combination with `plot()` as a shortcut.

```
with(unicorns, plot(weight))
```

To plot a scatterplot of one numeric variable against another numeric variable we just need to include both variables as arguments when using the `plot()` function. For example to plot `fluffyness` on the y axis and `weight` of the x axis.

```
plot(x = unicorns$weight, y = unicorns$fluffyness)
```



There is an equivalent approach for these types of plots which often causes some confusion at first. You can also use the formula notation when using the `plot()` function. However, in contrast to the previous method the formula method requires you to specify the y axis variable first, then a `~` and then our x axis variable.

```
plot(fluffyness ~ weight, data = unicorns)
```

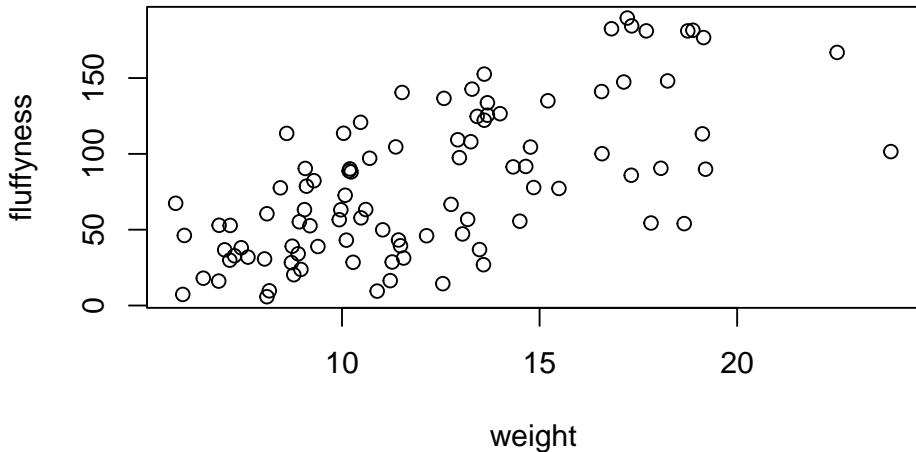


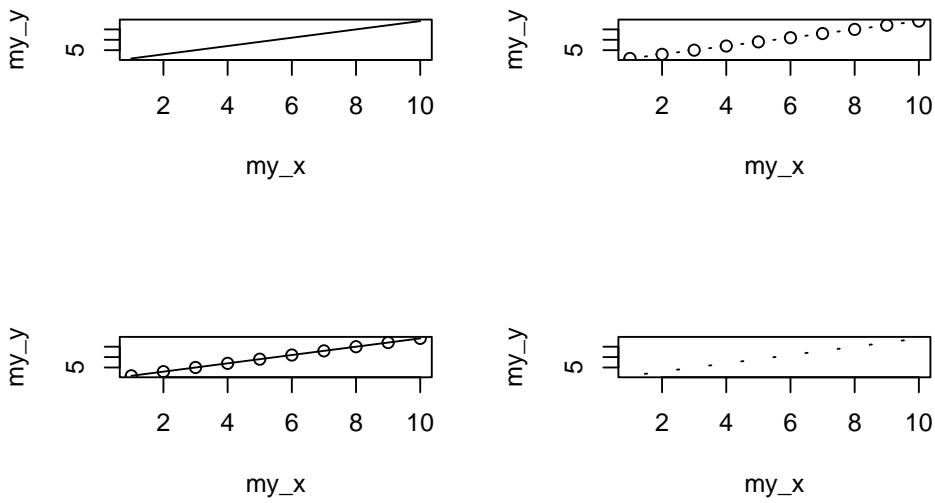
Figure 4.2.

Both of these two approaches are equivalent so we suggest that you just choose the one you prefer and go with it.

You can also specify the type of graph you wish to plot using the argument `type =`. You can plot just the points (`type = "p"`, this is the default), just lines (`type = "l"`), both points and lines connected (`type = "b"`), both points and lines with the lines running through the points (`type = "o"`) and empty points joined by lines (`type = "c"`). For example, let's use our skills from Section 2.4 to generate two vectors of numbers (`my_x` and `my_y`) and then plot one against the other using different `type =` values to see what type of plots are produced. Don't worry about the `par(mfrow = c(2, 2))` line of code yet. We're just using this to split the plotting device so we can fit all four plots on the same device to save some space. See Section 4.4 in the Chapter for more details about this. The top left plot is `type = "l"`, the top right `type = "b"`, bottom left `type = "o"` and bottom right is `type = "c"`.

```
my_x <- 1:10
my_y <- seq(from = 1, to = 20, by = 2)

par(mfrow = c(2, 2))
plot(my_x, my_y, type = "l")
plot(my_x, my_y, type = "b")
plot(my_x, my_y, type = "o")
plot(my_x, my_y, type = "c")
```



Admittedly the plots we've produced so far don't look anything particularly special. However, the `plot()` function is incredibly versatile and can generate a large range of plots which you can customise to your own taste. We'll cover how to customise ggplots in Section 4.5. As a quick aside, the `plot()` function is also what's known as a generic function which means it can change its default behaviour depending on the type of object used as an argument. You will see an example of this in Section 9.6 where we use the `plot()` function to generate diagnostic plots of residuals from a linear model object (bet you can't wait!).

4.2. **ggplot2**

As mentioned earlier `ggplot` grammar requires several elements to produce a graphic (Figure 4.1) and a minimum of 3 are required:

- a data frame
- a mapping system defining x and y
- a geometry layer

The data and mapping are provided within the call to the `ggplot()` function with the `data` and `mapping` arguments. The geometry layer is added using specific functions.

In fact all layers are needed but default simple values of the other layers are automatically provided.

To redo the Figure 4.2, that contain only a scatterplot of point we can use the `geom_point()` function.

```
ggplot(  
  data = unicorns,  
  mapping = aes(x = weight, y = fluffyness)  
) +  
  geom_point()
```

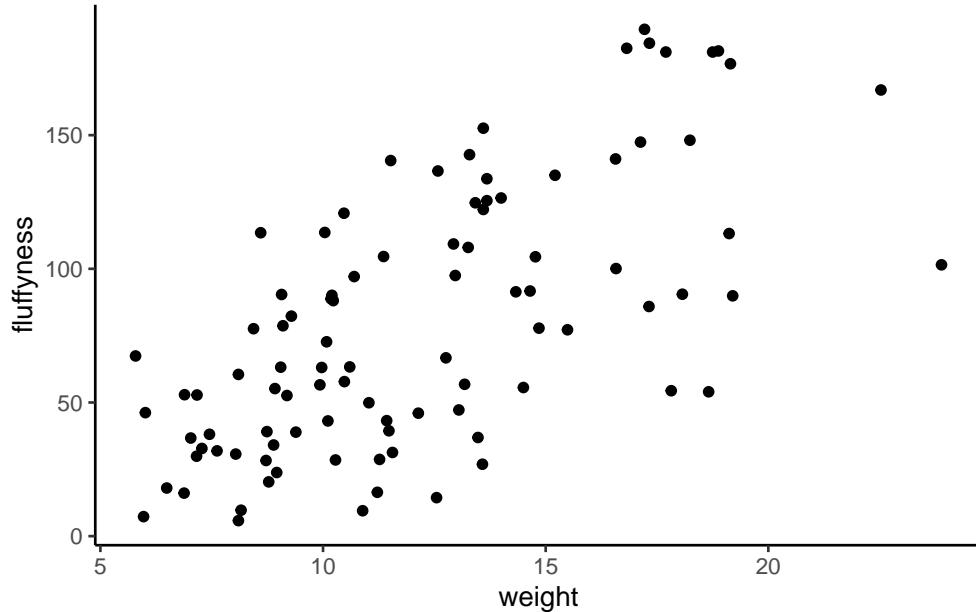


Figure 4.3.

Now that we have basic understanding of ggplotwe can explore some graphics using both base R and ggplot code

4.3. Simple plots

4.3.1. Scatterplots

Simple type of plots really useful to have a look at the relation between 2 variables for example. Here are the code to do it using base R (Figure 4.2)

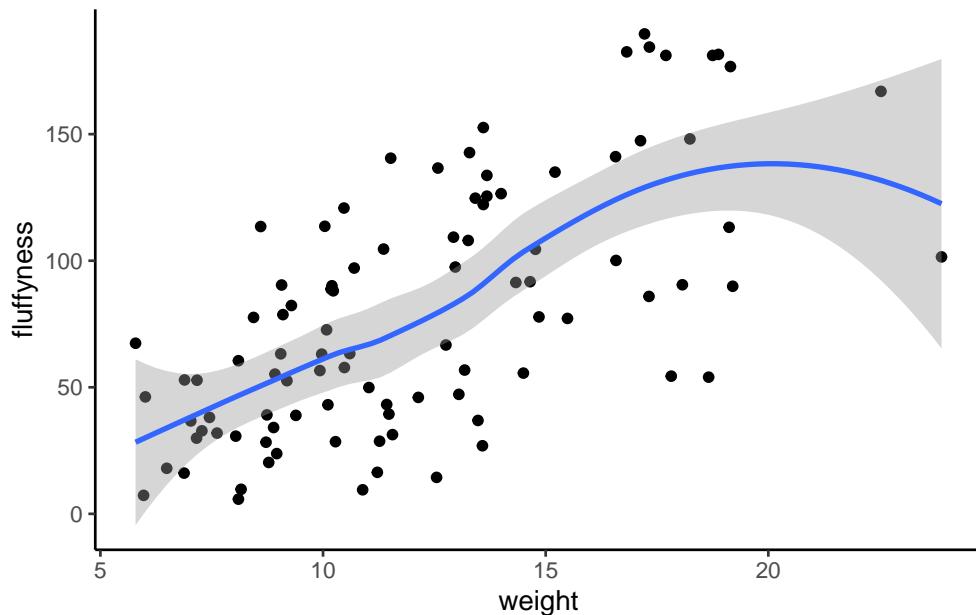
```
plot(fluffyness ~ weight, data = unicorns)
```

or ggplot (Figure 4.3)

```
ggplot(
  data = unicorns,
  mapping = aes(x = weight, y = fluffyness)
) +
  geom_point()
```

One big advantage of ggplot for simple scatterplot is the ease with which we can add a regression, smoother (loes or gam) line to the plot using `geom_smooth()` function to add a statistic layer to the plot.

```
ggplot(
  data = unicorns,
  mapping = aes(x = weight, y = fluffyness)
) +
  geom_point() +
  geom_smooth()
```

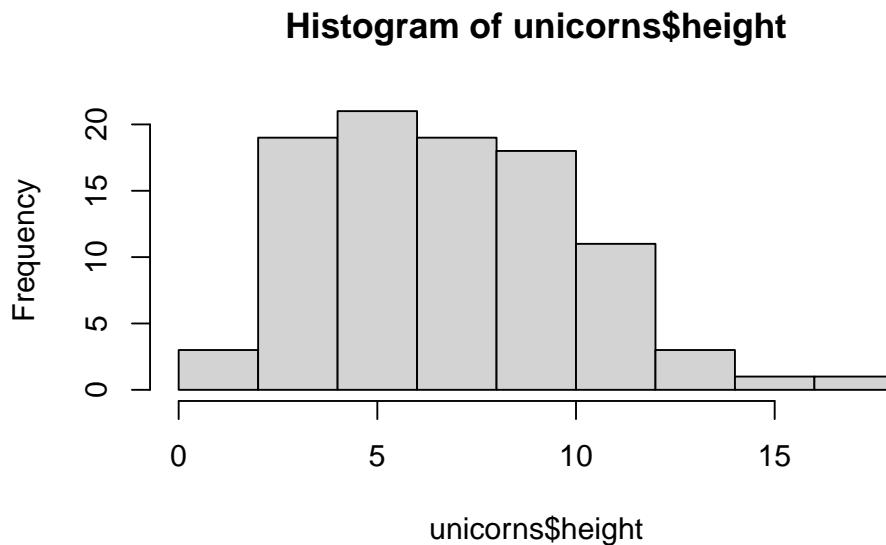


4.3.2. Histograms

Frequency histograms are useful when you want to get an idea about the distribution of values in a numeric variable. Using base R, the `hist()` function takes a numeric vector as its main argument. In ggplot, we need to use `geom_histogram()`. Let's generate a histogram of the height values.

With base R

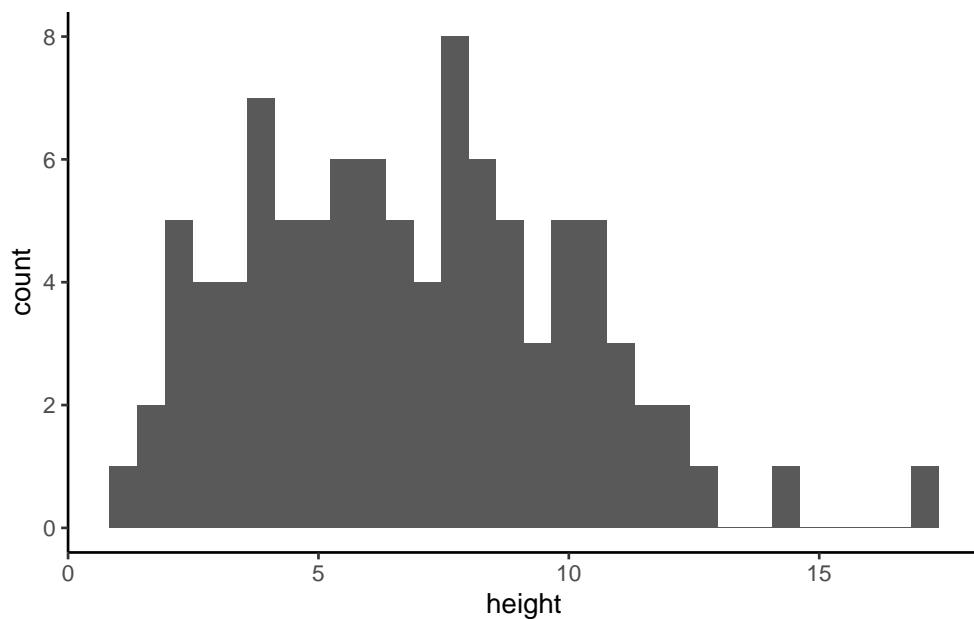
```
hist(unicorns$height)
```



with ggplot2

```
ggplot(unicorns, aes(x = height)) +  
  geom_histogram()
```

`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.

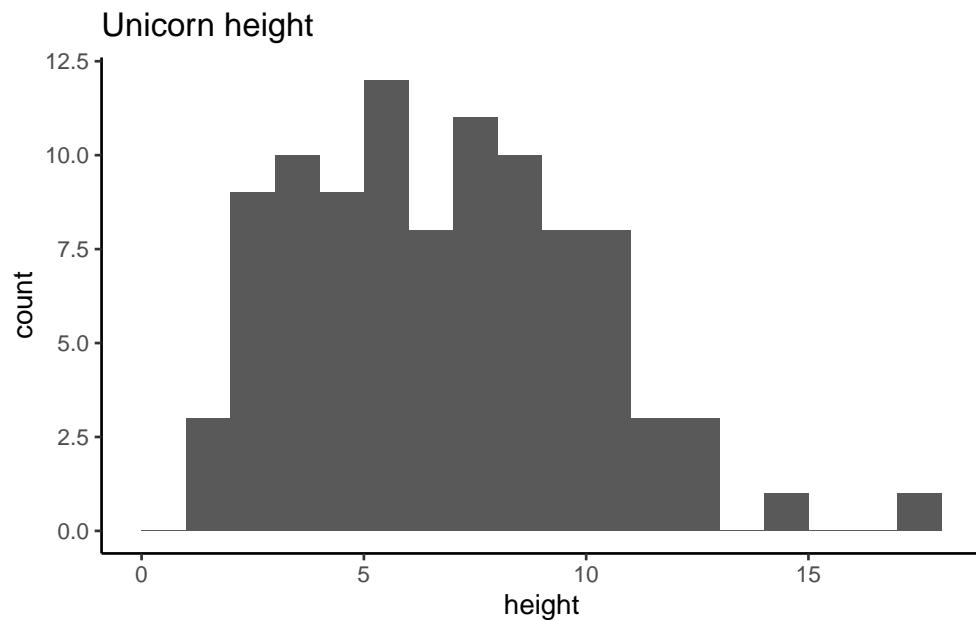


The `hist()` and `geom_histogram()` function automatically creates the breakpoints (or bins) in the histogram unless you specify otherwise by using the `breaks` = argument. For example, let's say we want to plot our histogram with breakpoints every 1 cm unicorns height. We first generate a sequence from zero to the maximum value of height (18 rounded up) in steps of 1 using the `seq()` function. We can then use this sequence with the `breaks` = argument. While we're at it, let's also replace the ugly title for something a little better using the `main` = argument

```
brk <- seq(from = 0, to = 18, by = 1)
hist(unicorns$height, breaks = brk, main = "Unicorn height")
```



```
brk <- seq(from = 0, to = 18, by = 1)
ggplot(unicorns, aes(x = height)) +
  geom_histogram(breaks = brk) +
  ggtitle("Unicorn height")
```

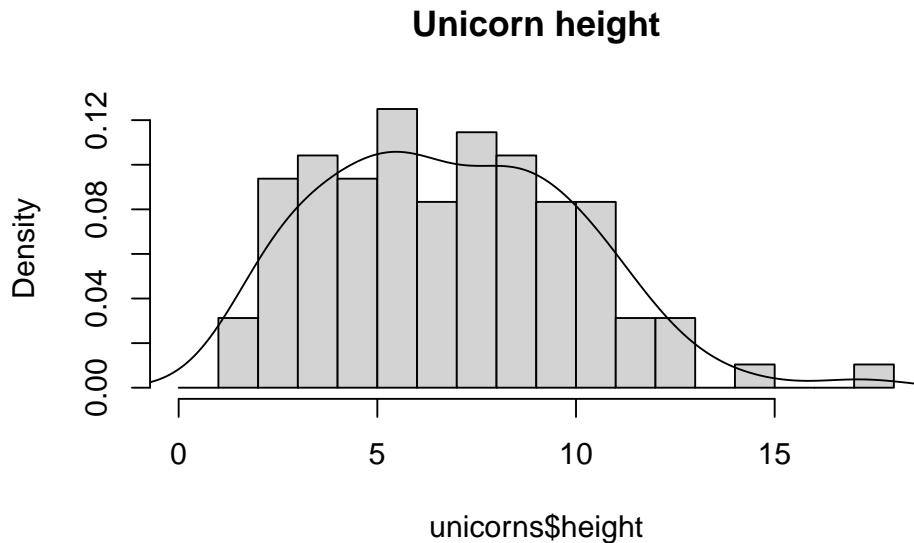


You can also display the histogram as a proportion rather than a frequency by using the `freq = FALSE` argument to `hist()` or indicating `aes(y = after_stat(density))` in `geom_histogram()`.

```
brk <- seq(from = 0, to = 18, by = 1)
hist(unicorns$height,
      breaks = brk, main = "Unicorn height",
      freq = FALSE
)
ggplot(unicorns, aes(x = height)) +
  geom_histogram(aes(y = after_stat(density)), breaks = brk) +
  ggtitle("Unicorn height")
```

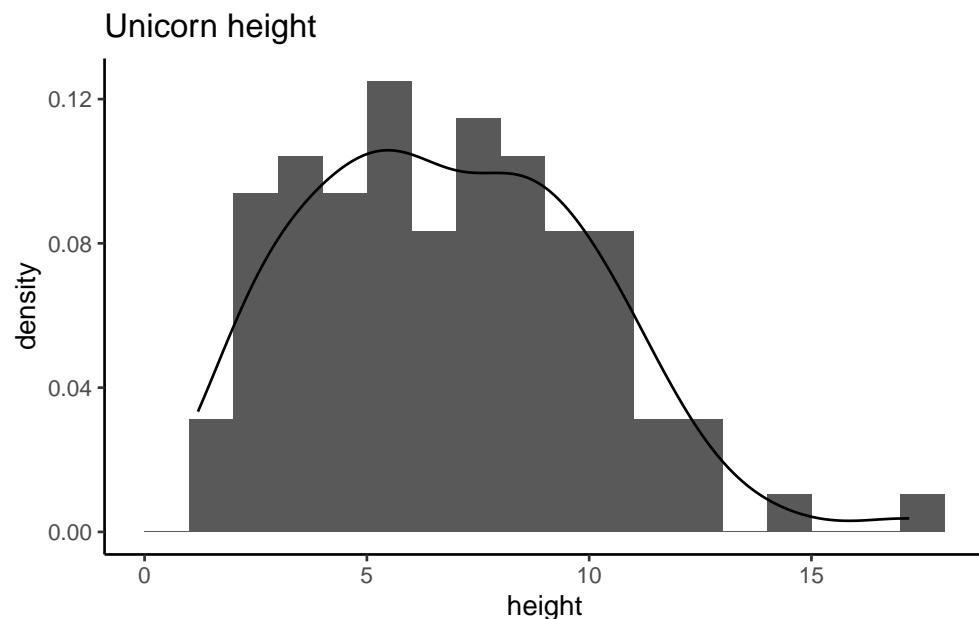
An alternative to plotting just a straight up histogram is to add a **kernel density** curve to the plot. In base R, you first need to compute the kernel density estimates using the `density()` and then add the estimates to plot as a line using the `lines()` function.

```
dens <- density(unicorns$height)
hist(unicorns$height,
      breaks = brk, main = "Unicorn height",
      freq = FALSE
)
lines(dens)
```



With `ggplot`, you can simply add the `geom_density()` layer to the plot

```
ggplot(unicorns, aes(x = height)) +
  geom_histogram(aes(y = after_stat(density)), breaks = brk) +
  geom_density() +
  ggtitle("Unicorn height")
```

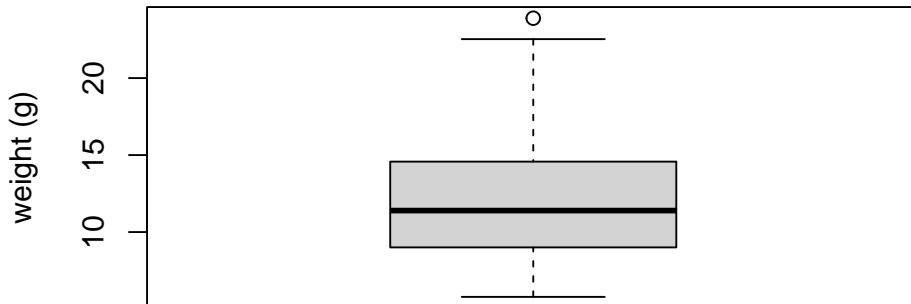


4.3.3. Box plots

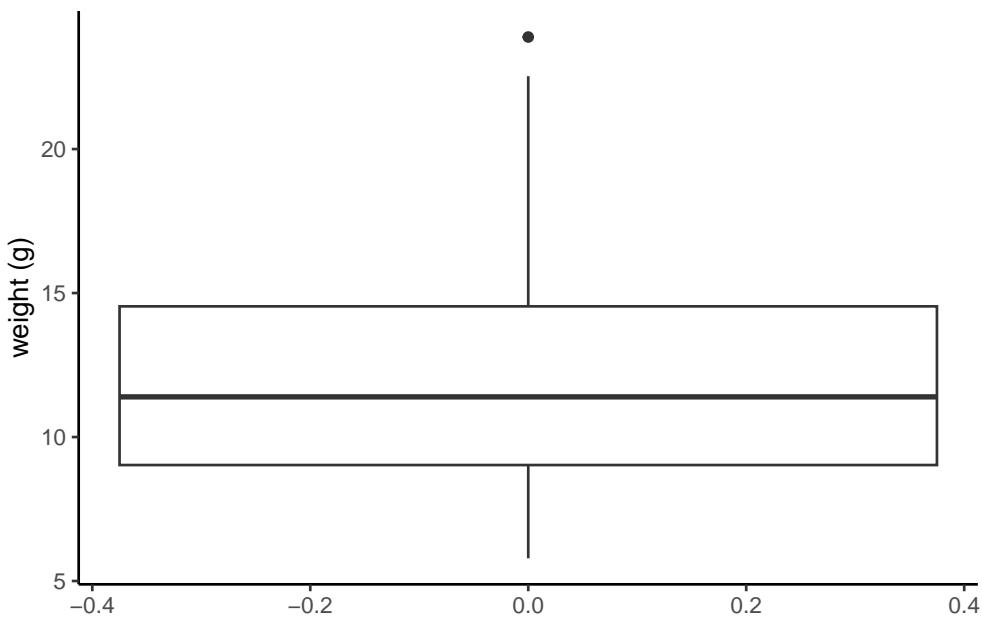
OK, we'll just come and out and say it, we love boxplots and their close relation the violin plot. Boxplots (or box-and-whisker plots to give them their full name) are very useful when you want to graphically summarise the distribution of a variable, identify potential unusual values and compare distributions between different groups. The reason we love them is their ease of interpretation, transparency and relatively high data-to-ink ratio (i.e. they convey lots of information efficiently). We suggest that you try to use boxplots as much as possible when exploring your data and avoid the temptation to use the more ubiquitous bar plot (even with standard error or 95% confidence intervals bars). The problem with bar plots (aka dynamite plots) is that they hide important information from the reader such as the distribution of the data and assume that the error bars (or confidence intervals) are symmetric around the mean. Of course, it's up to you what you do but if you're tempted to use bar plots just search for 'dynamite plots are evil' or see [here](#) or [here](#) for a fuller discussion.

To create a boxplot in R we use the `boxplot()` function. For example, let's create a boxplot of the variable `weight` from our `unicorns` data frame. We can also include a y axis label using the `ylab =` argument.

```
boxplot(unicorns$weight, ylab = "weight (g)")
```



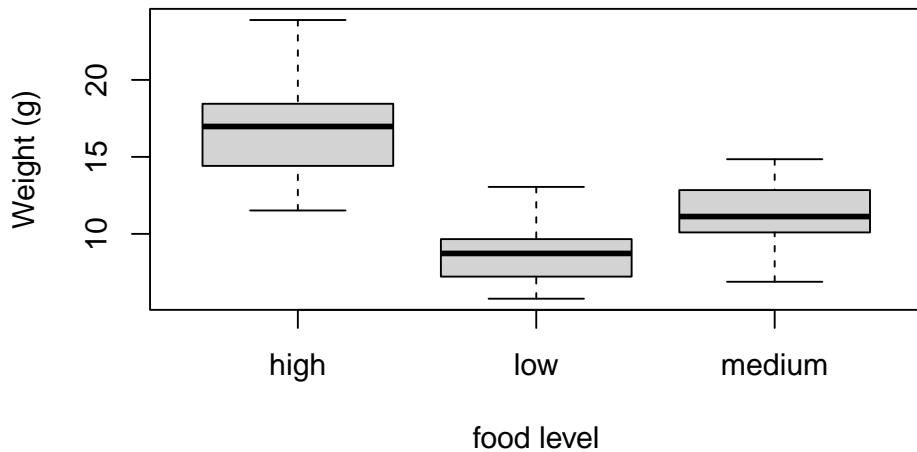
```
ggplot(unicorns, aes(y = weight)) +
  geom_boxplot() +
  labs(y = "weight (g)")
```



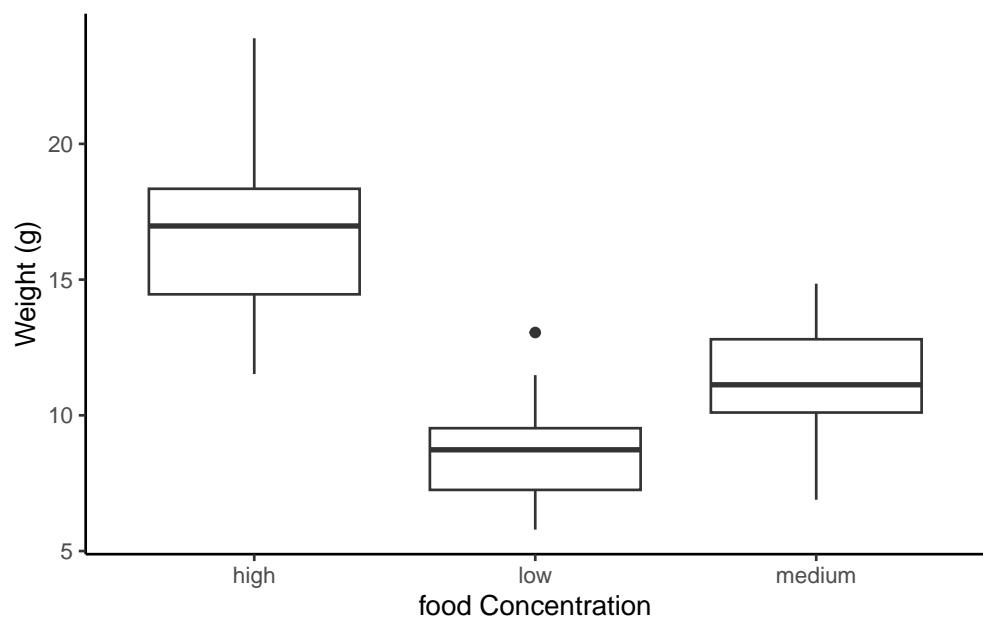
The thick horizontal line in the middle of the box is the median value of `weight` (around 11 g). The upper line of the box is the upper quartile (75th percentile) and the lower line is the lower quartile (25th percentile). The distance between the upper and lower quartiles is known as the inter quartile range and represents the values of `weight` for 50% of the data. The dotted vertical lines are called the whiskers and their length is determined as 1.5 x the inter quartile range. Data points that are plotted outside the the whiskers represent potential unusual observations. This doesn't mean they are unusual, just that they warrant a closer look. We recommend using boxplots in combination with Cleveland dotplots to identify potential unusual observations (see the Section 4.3.5 for more details). The neat thing about boxplots is that they not only provide a measure of central tendency (the median value) they also give you an idea about the distribution of the data. If the median line is more or less in the middle of the box (between the upper and lower quartiles) and the whiskers are more or less the same length then you can be reasonably sure the distribution of your data is symmetrical.

If we want examine how the distribution of a variable changes between different levels of a factor we need to use the formula notation with the `boxplot()` function. For example, let's plot our `weight` variable again, but this time see how this changes with each level of `food`. When we use the formula notation with `boxplot()` we can use the `data =` argument to save some typing. We'll also introduce an x axis label using the `xlab =` argument.

```
boxplot(weight ~ food,
        data = unicorns,
        ylab = "Weight (g)", xlab = "food level"
)
```



```
ggplot(unicorns, aes(y = weight, x = food)) +
  geom_boxplot() +
  labs(y = "Weight (g)", x = "food Concentration")
```



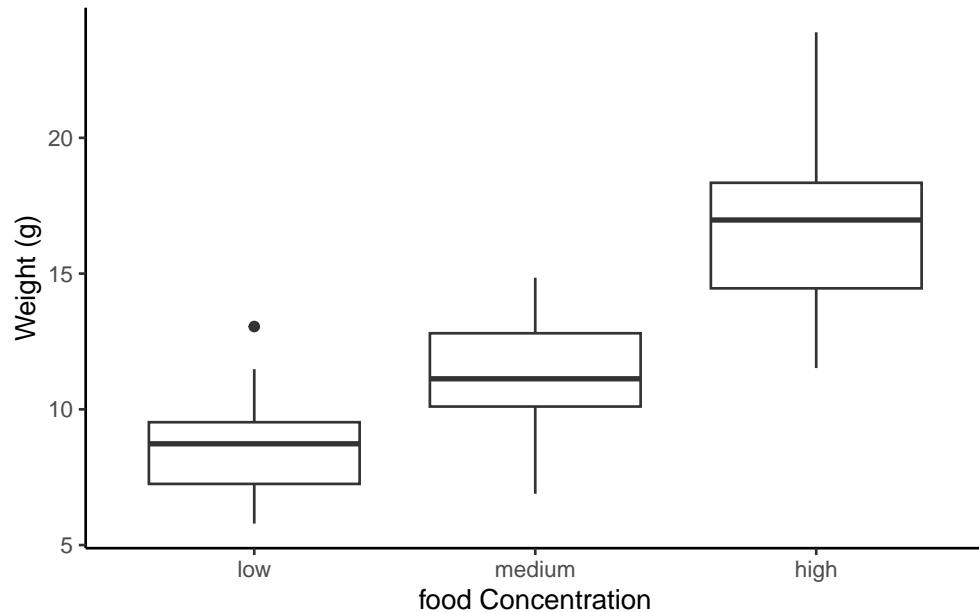
The factor levels are plotted in the same order defined by our factor variable `food` (often alphabetically). To change the order we need to change the order of our levels of the `food` factor in our data frame using the `factor()` function and then re-plot the graph. Let's plot our boxplot with our factor levels going from `low` to `high`.

```

unicorns$food <- factor(unicorns$food,
  levels = c("low", "medium", "high")
)

ggplot(unicorns, aes(y = weight, x = food)) +
  geom_boxplot() +
  labs(y = "Weight (g)", x = "food Concentration")

```

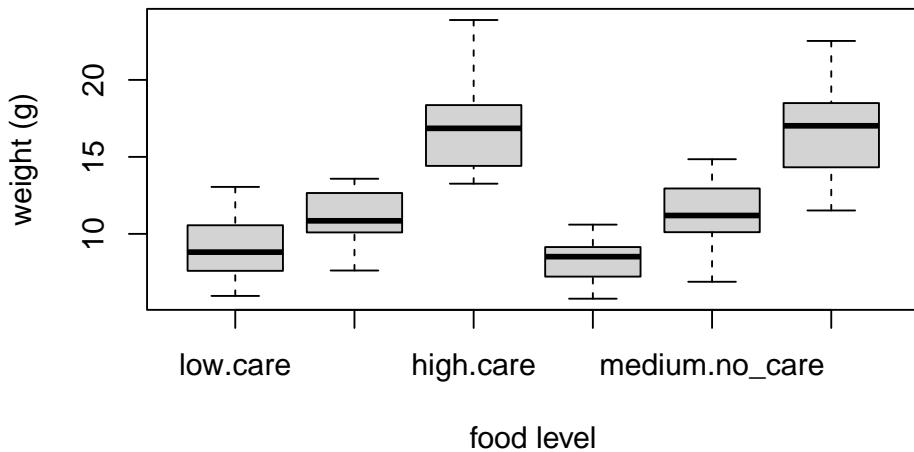


We can also group our variables by two factors in the same plot. Let's plot our `weight` variable but this time plot a separate box for each `food` and `p_care` treatment combination.

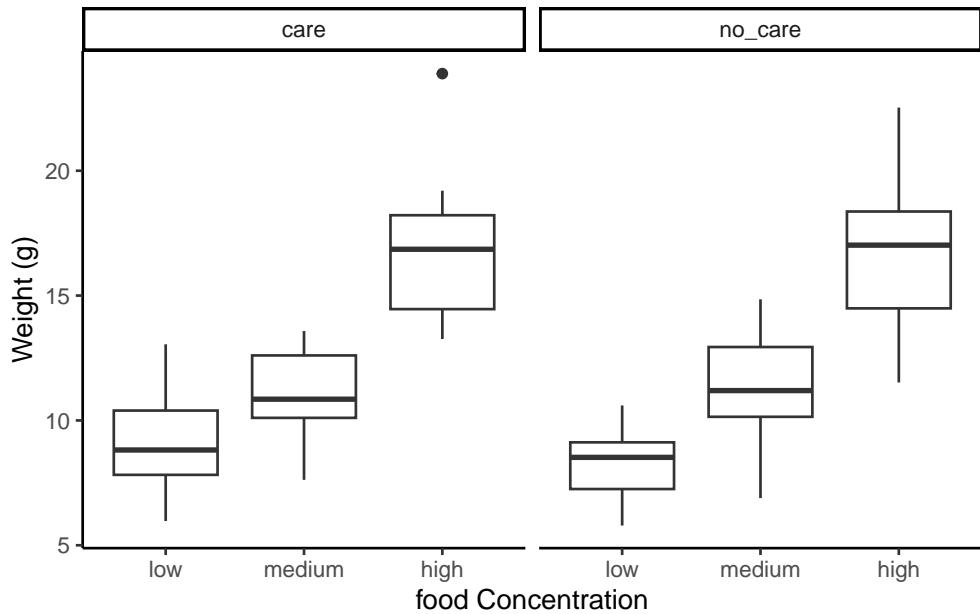
```

boxplot(weight ~ food * p_care,
  data = unicorns,
  ylab = "weight (g)", xlab = "food level"
)

```



```
ggplot(unicorns, aes(y = weight, x = food)) +
  geom_boxplot() +
  labs(y = "Weight (g)", x = "food Concentration") +
  facet_grid(. ~ p_care)
```

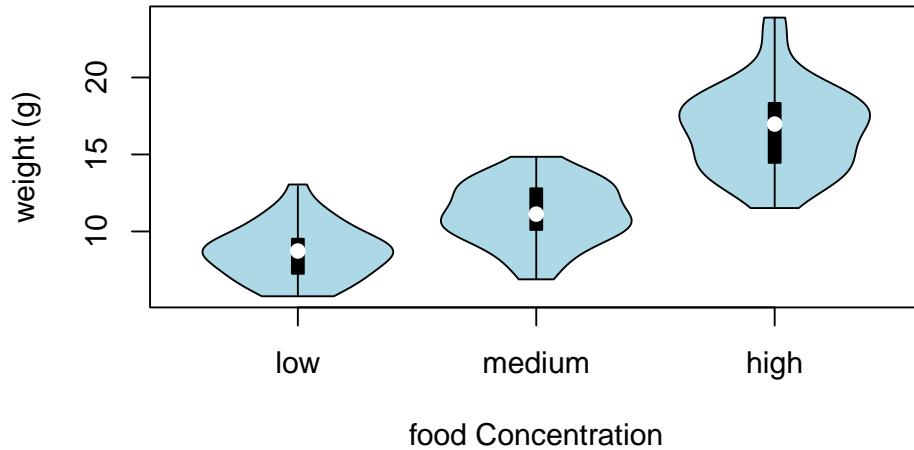


This plot looks much better in ggplot with the use of `facet_grid` allowing to make similar plots as a function of a third (or even fourth) variable.

4.3.4. Violin plots

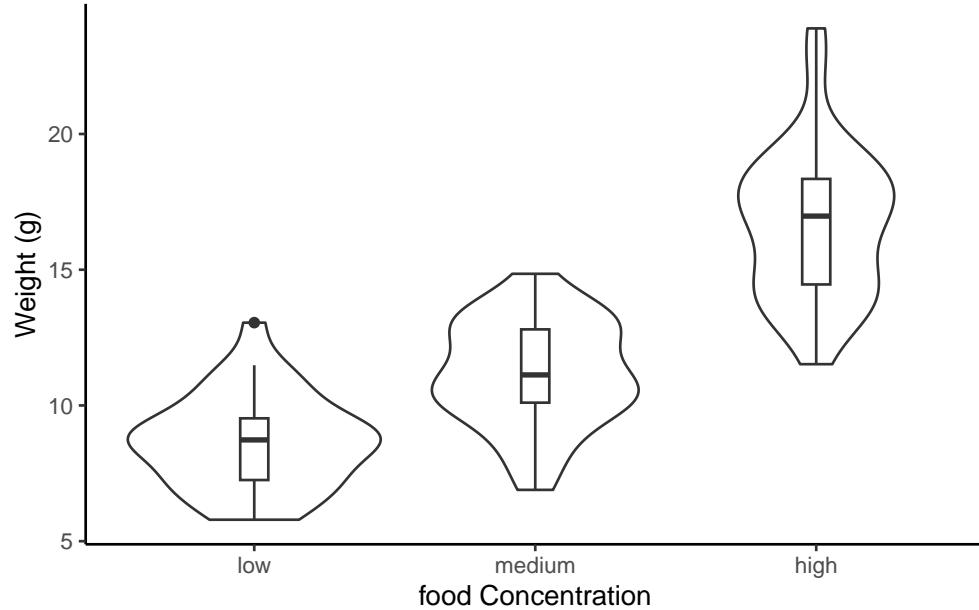
Violin plots are like a combination of a boxplot and a kernel density plot (you saw an example of a kernel density plot in the histogram section above) all rolled into one figure. We can create a violin plot in R using the `vioplot()` function from the `vioplot`  package. You'll need to first install this package using `install.packages('vioplot')` function as usual. The nice thing about the `vioplot()` function is that you use it in pretty much the same way you would use the `boxplot()` function. We'll also use the argument `col = "lightblue"` to change the fill colour to light blue.

```
library(vioplot)
vioplot(weight ~ food,
        data = unicorns,
        ylab = "weight (g)", xlab = "food Concentration",
        col = "lightblue")
)
```



In the violin plot above we have our familiar boxplot for each food level but this time the median value is represented by a white circle. Plotted around each boxplot is the kernel density plot which represents the distribution of the data for each food level.

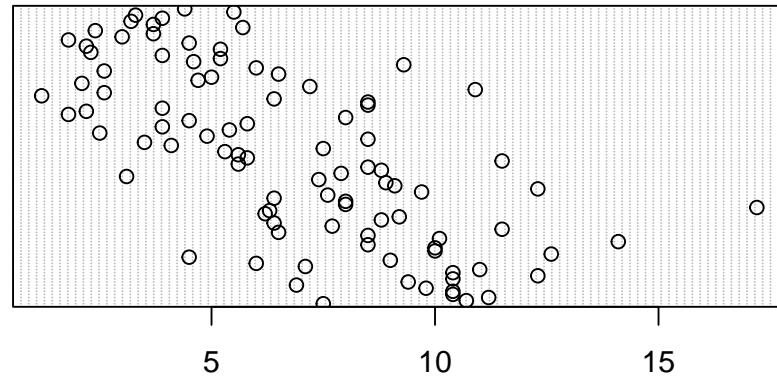
```
ggplot(unicorns, aes(y = weight, x = food)) +  
  geom_violin() +  
  geom_boxplot(width = 0.1) +  
  labs(y = "Weight (g)", x = "food Concentration")
```



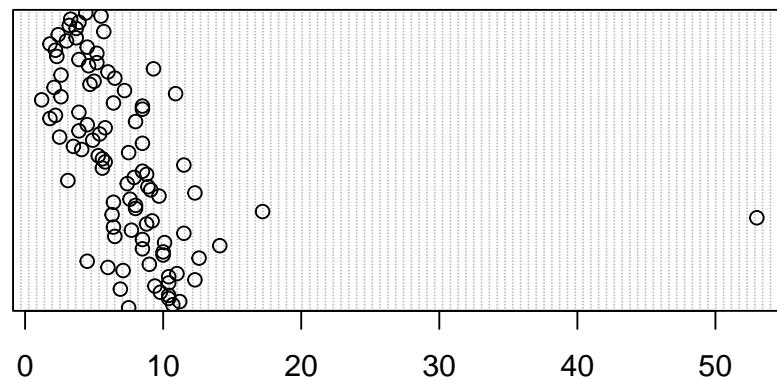
4.3.5. Dot charts

Identifying unusual observations (aka outliers) in numeric variables is extremely important as they may influence parameter estimates in your statistical model or indicate an error in your data. A really useful (if undervalued) plot to help identify outliers is the Cleveland dotplot. You can produce a dotplot in R very simply by using the `dotchart()` function.

```
dotchart(unicorns$height)
```

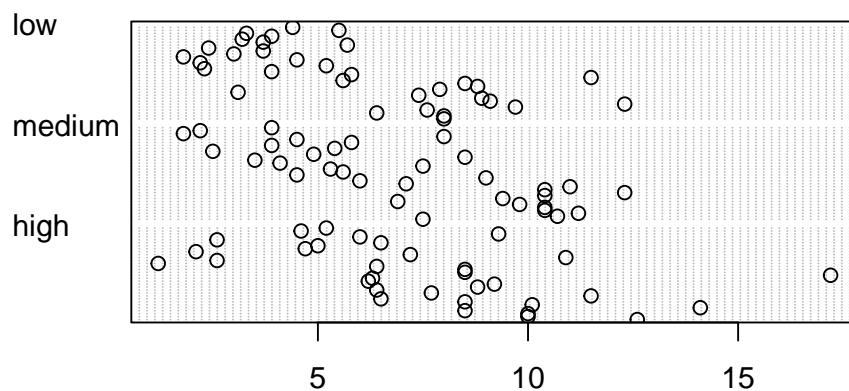


In the dotplot above the data from the `height` variable is plotted along the x axis and the data is plotted in the order it occurs in the `unicorns` data frame on the y axis (values near the top of the y axis occur later in the data frame with those lower down occurring at the beginning of the data frame). In this plot we have a single value extending to the right at about 17 cm but it doesn't appear particularly large compared to the rest. An example of a dotplot with an unusual observation is given below.

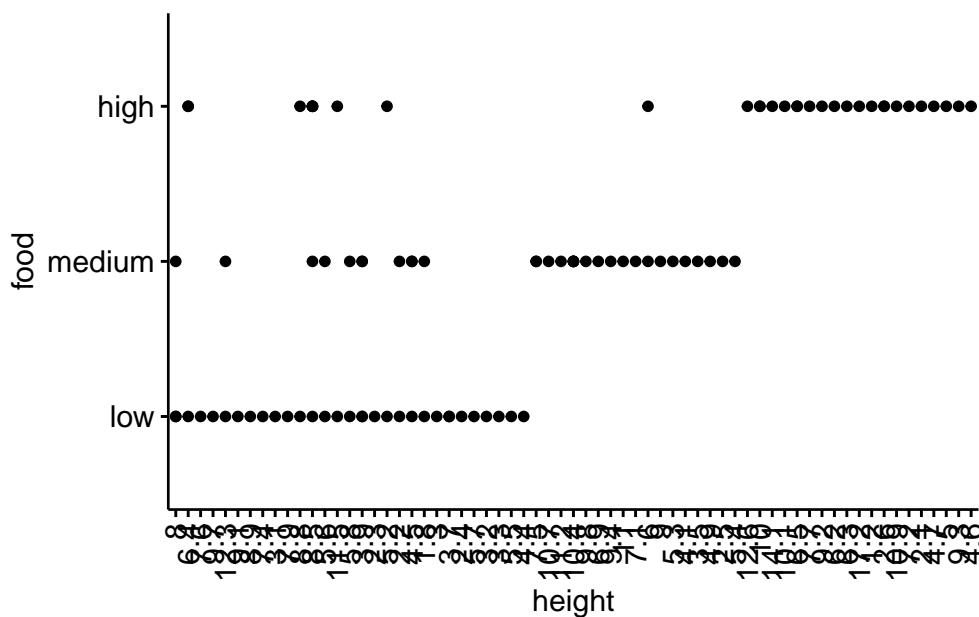


We can also group the values in our `height` variable by a factor variable such as `food` using the `groups =` argument. This is useful for identifying unusual observations within a factor level that might be obscured when looking at all the data together.

```
dotchart(unicorns$height, groups = unicorns$food)
```



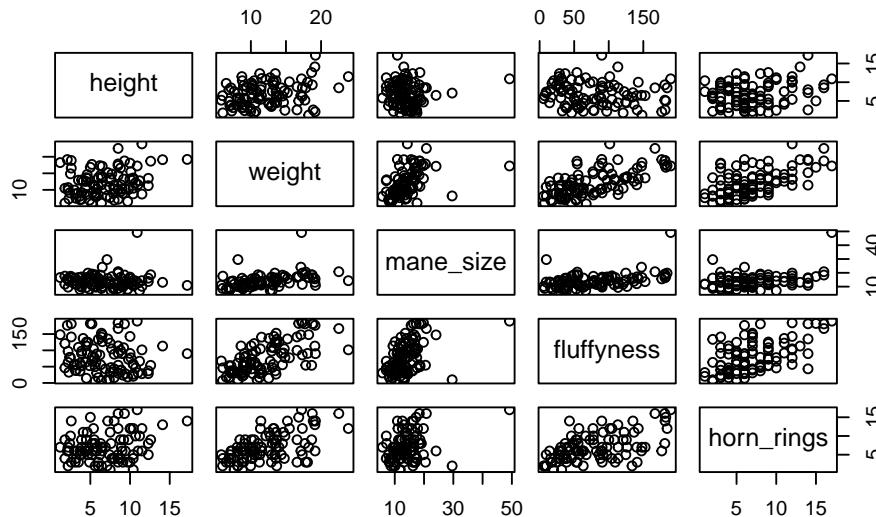
```
ggdotchart(data = unicorns, x = "height", y = "food")
```



4.3.6. Pairs plots

Previously in this Chapter we used the `plot()` function to create a scatterplot to explore the relationship between two numeric variables. With datasets that contain many numeric variables, it's often handy to create multiple scatterplots to visualise relationships between all these variables. We could use the `plot()` function to create each of these plot individually, but a much easier way is to use the `pairs()` function. The `pairs()` function creates a multi-panel scatterplot (sometimes called a scatterplot matrix) which plots all combinations of variables. Let's create a multi-panel scatterplot of all of the numeric variables in our `unicorns` data frame. Note, you may need to click on the 'Zoom' button in RStudio to display the plot clearly.

```
pairs(unicorns[, c(
  "height", "weight", "mane_size",
  "fluffyness", "horn_rings"
)])
```



```
# or we could use the equivalent
# pairs(unicorns[, 4:8])
```

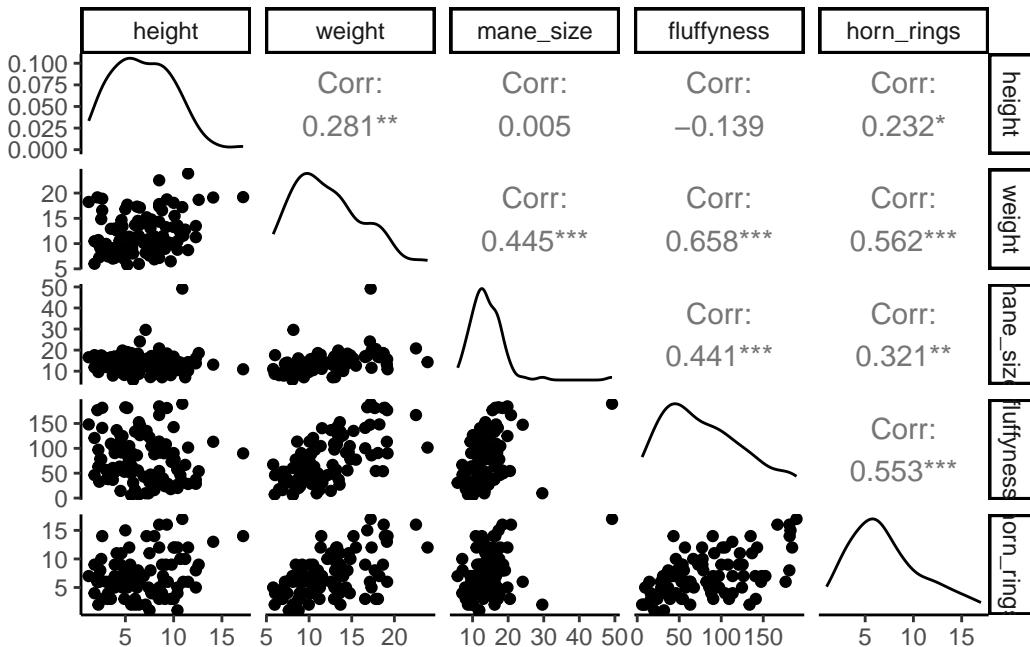
Interpretation of the pairs plot takes a bit of getting used to. The panels on the diagonal give the variable names. The first row of plots displays the `height` variable on the y axis and the variables `weight`, `mane_size`, `fluffyness` and `horn_rings` on the x axis for each of the four plots respectively. The next row of plots have `weight` on the y axis and `height`, `mane_size`, `fluffyness` and `horn_rings` on the x axis. We interpret the rest of the rows in the

same way with the last row displaying the `unicorns` variable on the y axis and the other variables on the x axis. Hopefully you'll notice that the plots below the diagonal are the same plots as those above the diagonal just with the axis reversed.

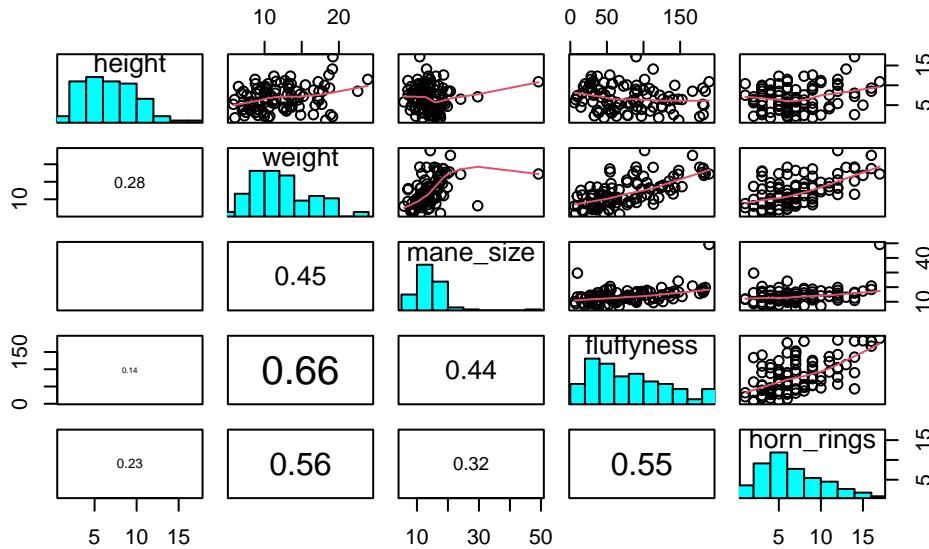
To do pairs plot with `ggplot`, you nee the `ggpairs()` function from `GGally`  package. The output is quite similar but you have only the lower part of the matrix of plots, you get a density plot on the diagonal and the correlations on the upper part of the plot.

```
ggpairs(unicorns[, c(
  "height", "weight", "mane_size",
  "fluffyness", "horn_rings"
)])

```



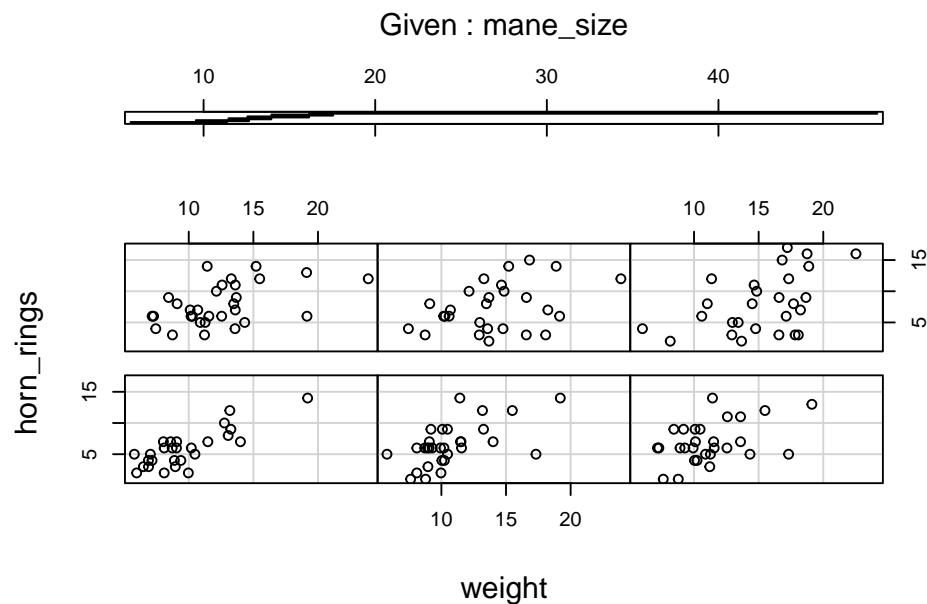
The `pairs()` function can be tweak to do similar things and more but is more involved. Have a look at the great help file for the `pairs()` function (`?pairs`)which provide all the details to do something like the plot below.



4.3.7. Coplots

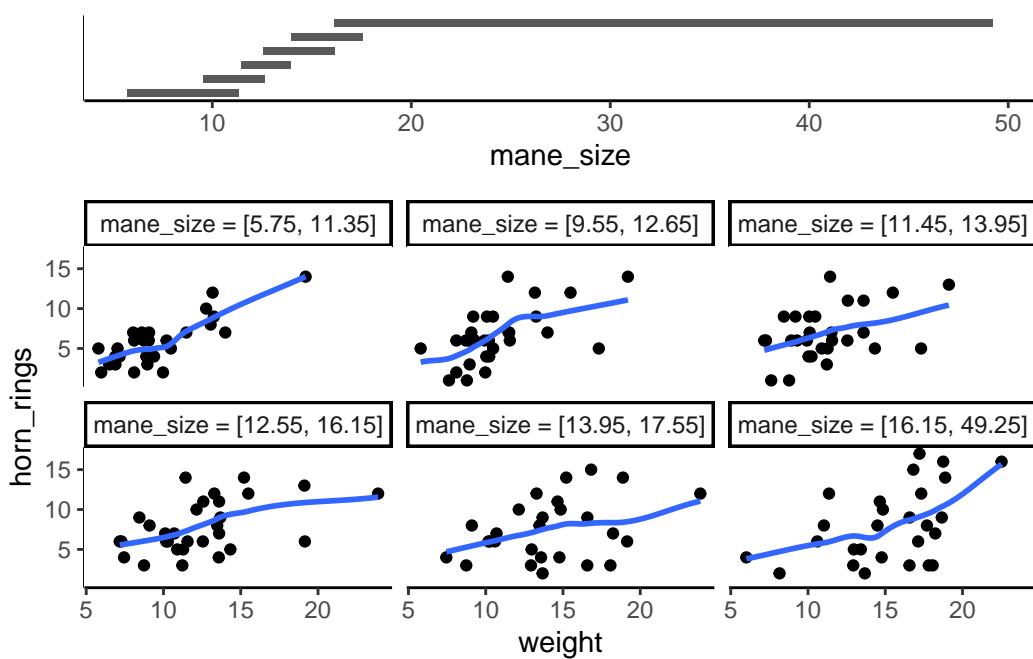
When examining the relationship between two numeric variables, it is often useful to be able to determine whether a third variable is obscuring or changing any relationship. A really handy plot to use in these situations is a conditioning plot (also known as conditional scatterplot plot) which we can create in R by using the `coplot()` function. The `coplot()` function plots two variables but each plot is conditioned (`|`) by a third variable. This third variable can be either numeric or a factor. As an example, let's look at how the relationship between the number of horn rings (`horn_rings` variable) and the `weight` of unicorns changes dependent on `mane_size`. Note the `coplot()` function has a `data =` argument so no need to use the `$` notation.

```
coplot(horn_rings ~ weight | mane_size, data = unicorns)
```



```
gg_coplot(unicorns,
  x = weight, y = horn_rings,
  facetting = mane_size
)
```

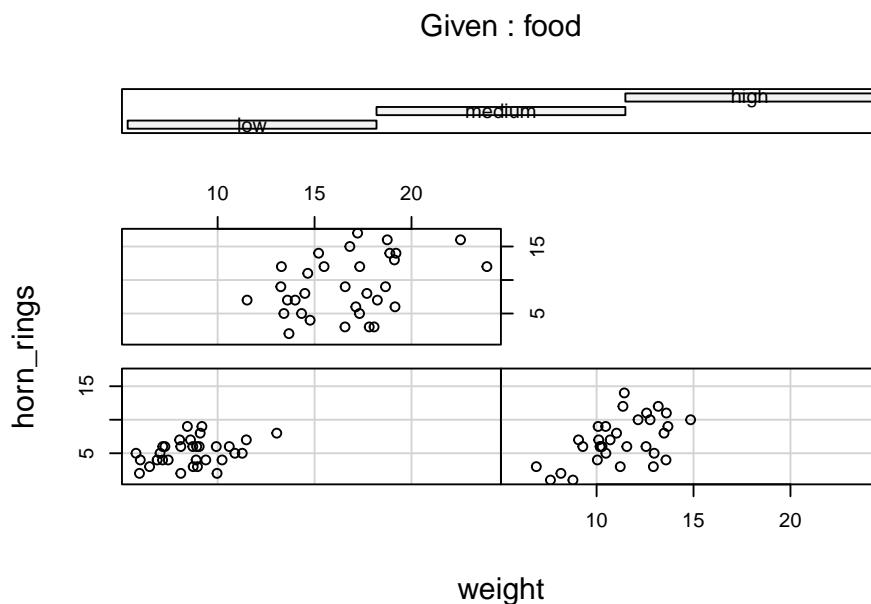
```
`geom_smooth()` using formula = 'y ~ x'
```



It takes a little practice to interpret coplots. The number of horn rings is plotted on the y axis and the weight of unicorns on the x axis. The six plots show the relationship between these two variables for different ranges of mane size. The bar plot at the top indicates the range of mane size values for each of the plots. The panels are read from bottom left to top right along each row. For example, the bottom left panel shows the relationship between number of horn rings and weight for unicorns with the lowest range of mane size (approximately 5 - 11 cm). The top right plot shows the relationship between horn ring and weight for unicorns with a mane size ranging from approximately 16 - 50 cm. Notice that the range of values for mane size differs between panels and that the ranges overlap from panel to panel. The `coplot()` function does it's best to split the data up to ensure there are an adequate number of data points in each panel. If you don't want to produce plots with overlapping data in the panel you can set the `overlap = argument to overlap = 0`

You can also use the `coplot()` function with factor conditioning variables. With `gg_coplot()` you need to first set the factor as numeric before plotting and specify `overlap=0`. For example, we can examine the relationship between `horn_rings` and `weight` variables conditioned on the factor `food`. The bottom left plot is the relationship between `horn_rings` and `weight` for those unicorns in the `low` food treatment. The top left plot shows the same relationship but for unicorns in the `high` food treatment.

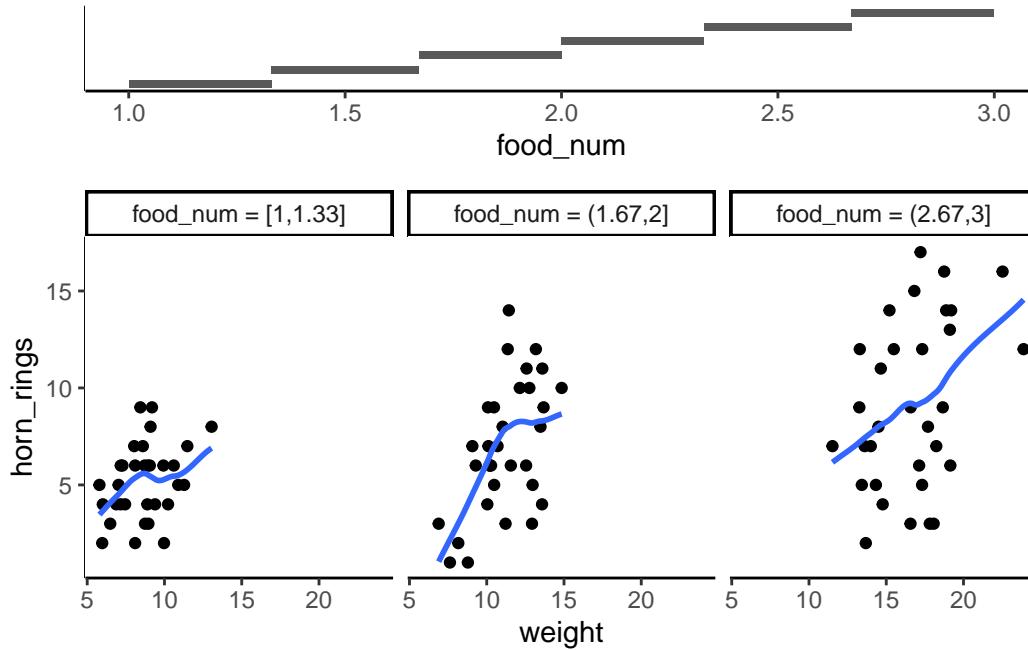
```
coplot(horn_rings ~ weight | food, data = unicorns)
```



```
unicorns <- mutate(unicorns, food_num = as.numeric(food))
gg_coplot(unicorns,
  x = weight, y = horn_rings,
```

```
    facetting = food_num, overlap = 0
)
```

```
`geom_smooth()` using formula = 'y ~ x'
```



4.3.8. Summary of plot function

Graph type	ggplot2	Base R function
scatterplot	<code>geom_point()</code>	<code>plot()</code>
frequency histogram	<code>geom_histogram()</code>	<code>hist()</code>
boxplot	<code>geom_boxplot()</code>	<code>boxplot()</code>
Cleveland dotplot	<code>ggdotchart()</code>	<code>dotchart()</code>
scatterplot matrix	<code>ggpairs()</code>	<code>pairs()</code>
conditioning plot	<code>gg_coplot()</code>	<code>coplot()</code>

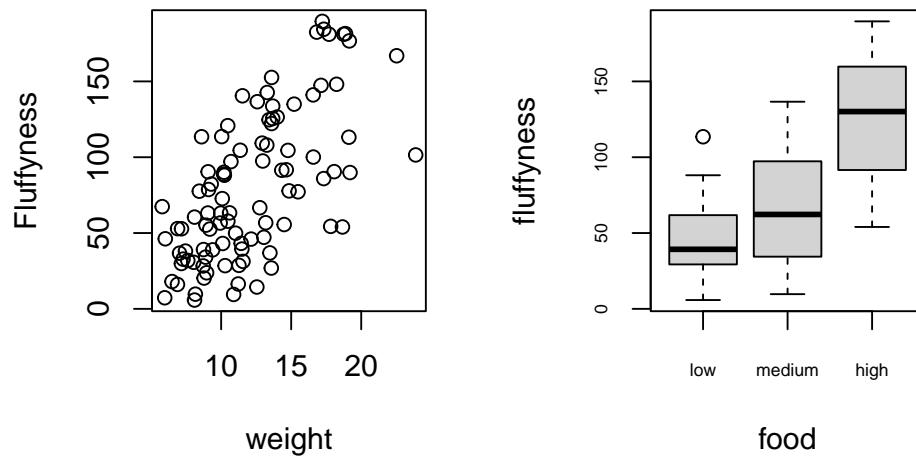
Hopefully, you're getting the idea that we can create really informative exploratory plots quite easily using either base R or ggplot graphics. Which one you use is entirely up to you (that's the beauty of using R, you get to choose) and we happily mix and match to suit our needs. In the next section we cover how to customise your base R plots to get them to look exactly how you want.

4.4. Multiple graphs

4.4.1. Base R

In base R, one of the most common methods to plot multiple graphs is to use the main graphical function `par()` to split the plotting device up into a number of defined sections using the `mfrow =` argument. With this method, you first need to specify the number of rows and columns of plots you would like and then run the code for each plot. For example, to plot two graphs side by side we would use `par(mfrow = c(1, 2))` to split the device into 1 row and two columns.

```
par(mfrow = c(1, 2))
plot(unicorns$weight, unicorns$fluffyness,
     xlab = "weight",
     ylab = "Fluffyness")
)
boxplot(fluffyness ~ food, data = unicorns, cex.axis = 0.6)
```



Once you've finished making your plots don't forget to reset your plotting device back to normal with `par(mfrow = c(1,1))`.

4.4.2. ggplot

Using `ggplot` in addition to the `facet_grid()` and `facet_wrap` functions allowing to easily repeat and organise multiple plots as a function of specific variables, there are multiple way of organising multiple `ggplot` together. The approach we recommend is using the `patchwork`  package.

First you will need to install (if you don't have it yet) and make the `patchwork`  package available.

```
install.packages("patchwork")
library(patchwork)
```

An important note: For those who have used base R to produce their figures and are familiar with using `par(mfrow = c(2,2))` (which allows plotting of four figures in two rows and two columns) be aware that this does not work for `ggplot2` objects. Instead you will need to use either the `patchwork`  package or alternative packages such as `gridArrange`  or `cowplot`  or convert the `ggplot2` objects to grobs.

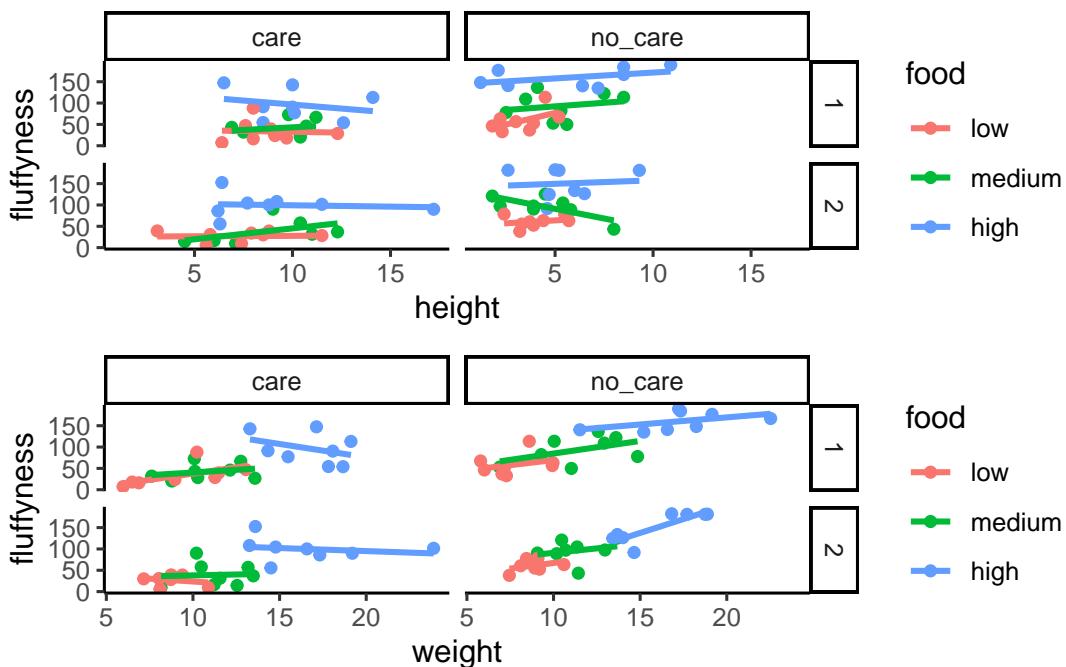
To plot both of the plots together we need to assign each figure to a separate object and then use these objects when we use `patchwork`.

So we can generate 2 figures and assign them to objects. As you can see, the figures do not appear in the plot window. They will appear only when you call the object.

```
first_figure <- ggplot(
  aes(x = height, y = fluffyness, color = food),
  data = unicorns
) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  facet_grid(block ~ p_care)
second_figure <- ggplot(
  aes(x = weight, y = fluffyness, color = food),
  data = unicorns
) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  facet_grid(block ~ p_care)
```

We have two immediate and simple options with patchwork; arrange figures on top of each other (specified with a /) or arrange figures side-by-side (specified with either a + or a |). Let's try to plot both figures, one on top of the other.

```
first_figure / second_figure
```

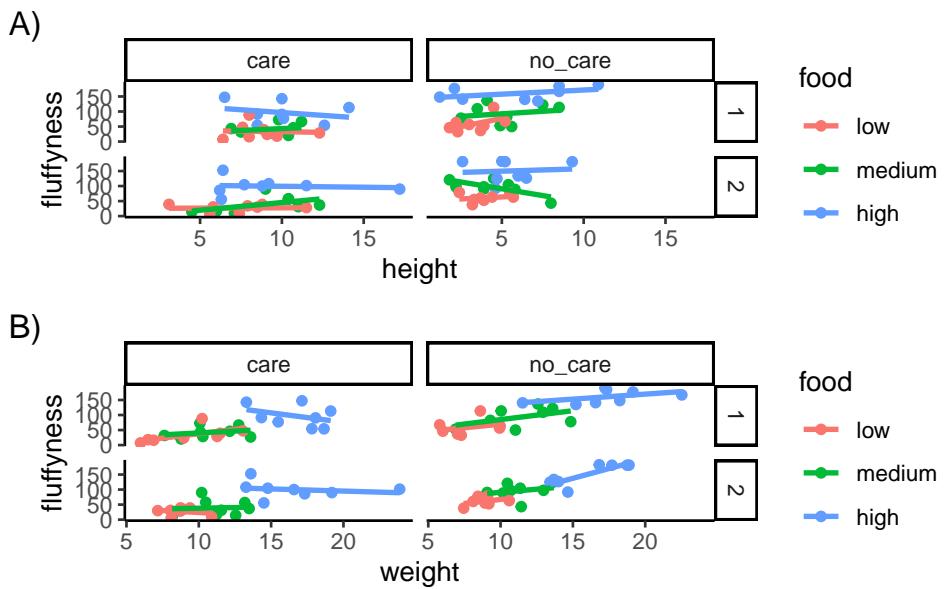


Play around: Try to create a side-by-side version of the above figure (hint: try the other operators).

We can take this one step further and assign nested patchwork figures to an object and use this in turn to create labels for individual figures.

```
nested_compare <- first_figure / second_figure

nested_compare +
  plot_annotation(tag_levels = "A", tag_suffix = ")")
```



4.5. Customising ggplots

Went for a walk to be edited 🦄

4.6. Exporting plots

Creating plots in R is all well and good but what if you want to use these plots in your thesis, report or publication? One option is to click on the ‘Export’ button in the ‘Plots’ tab in RStudio. You can also export your plots from R to an external file by writing some code in your R script. The advantage of this approach is that you have a little more control over the output format and it also allows you to generate (or update) plots automatically whenever you run your script. You can export your plots in many different formats but the most common are, pdf, png, jpeg and tiff.

By default, R (and therefore RStudio) will direct any plot you create to the plot window. To save your plot to an external file you first need to redirect your plot to a different graphics device. You do this by using one of the many graphics device functions to start a new graphic device. For example, to save a plot in pdf format we will use the `pdf()` function. The first argument in the `pdf()` function is the filepath and filename of the file we want to save (don’t forget to include the .pdf extension). Once we’ve used the `pdf()` function we can then write all of the code we used to create our plot including any graphical parameters such as setting the margins and splitting up the plotting device. Once the code has run we need to close the pdf plotting device using the `dev.off()` function.

```

pdf(file = "output/my_plot.pdf")
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs = "i", yaxs = "i")
plot(unicorns$weight, unicorns$fluffyness,
  xlab = "weight (g)",
  ylab = expression(paste("shoot area (cm"^(2), ")")),
  xlim = c(0, 30), ylim = c(0, 200), bty = "l",
  las = 1, cex.axis = 0.8, tcl = -0.2,
  pch = 16, col = "dodgerblue1", cex = 0.9
)
text(x = 28, y = 190, label = "A", cex = 2)
dev.off()

```

If we want to save this plot in png format we simply use the `png()` function in more or less the same way we used the `pdf()` function.

```

png("output/my_plot.png")
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs = "i", yaxs = "i")
plot(unicorns$weight, unicorns$fluffyness,
  xlab = "weight (g)",
  ylab = expression(paste("shoot area (cm"^(2), ")")),
  xlim = c(0, 30), ylim = c(0, 200), bty = "l",
  las = 1, cex.axis = 0.8, tcl = -0.2,
  pch = 16, col = "dodgerblue1", cex = 0.9
)
text(x = 28, y = 190, label = "A", cex = 2)
dev.off()

```

Other useful functions are; `jpeg()`, `tiff()` and `bmp()`. Additional arguments to these functions allow you to change the size, resolution and background colour of your saved images. See `?png` for more details.

`ggplot2`  provide a really useful function `ggsave()` function which simplify saving plots a lot but works only for ggplots.

After producing a plot and seeing it in your IDE, you can simply run `ggsave()` with the adequate argument to save the last ggplot produced. You can of course, also, specify which plot to save.

```
ggsave("file.png")
```

Chapter 5

Programming

After learning the basics, programming in R is the next big step. There are already a vast number of R packages available, surely more than enough to cover everything you could possibly want to do? Why then, would you ever need to create your own R functions? Why not just stick to the functions from a package? Well, in some cases you'll want to customise those existing functions to suit your specific needs. Or you may want to implement a new approach which means there won't be any pre-existing packages that work for you. Both of these are not particularly common. Functions are mainly used to do one thing well in a simple manner without having to type of the code necessary to do that function each time. We can see functions as a short-cut to copy-pasting. If you have to do a similar task 4 times or more, build a function for it, and simply call that function 4 times or call it in a loop .

5.1. Looking behind the curtain

A good way to start learning to program in R is to see what others have done. We can start by briefly peeking behind the curtain. With many functions in R, if you want to have a quick glance at the machinery behind the scenes, we can simply write the function name but without the () .

Note that to view the source code of base R packages (those that come with R) requires some additional steps which we won't cover here (see this [link](#) if you're interested), but for most other packages that you install yourself, generally entering the function name without () will show the source code of the function.

What can have a look at the function to fit a linear model `lm()`

`lm`

```
function (formula, data, subset, weights, na.action, method = "qr",
  model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
  contrasts = NULL, offset, ...)
{
  ret.x <- x
  ret.y <- y
  cl <- match.call()
  mf <- match.call(expand.dots = FALSE)
  m <- match(c("formula", "data", "subset", "weights", "na.action",
    "offset"), names(mf), 0L)
  mf <- mf[c(1L, m)]
  mf$drop.unused.levels <- TRUE
  mf[[1L]] <- quote(stats::model.frame)
  mf <- eval(mf, parent.frame())
  if (method == "model.frame")
    return(mf)
  else if (method != "qr")
    warning(gettextf("method = '%s' is not supported. Using 'qr'", method),
            domain = NA)
  mt <- attr(mf, "terms")
  y <- model.response(mf, "numeric")
  w <- as.vector(model.weights(mf))
  if (!is.null(w) && !is.numeric(w))
    stop("'weights' must be a numeric vector")
  offset <- model.offset(mf)
  mlm <- is.matrix(y)
  ny <- if (mlm)
    nrow(y)
  else length(y)
  if (!is.null(offset)) {
    if (!mlm)
      offset <- as.vector(offset)
    if (NROW(offset) != ny)
```

```

stopgettextf("number of offsets is %d, should equal %d (number of observations)",
             NROW(offset), ny), domain = NA)
}

if (is.empty.model(mt)) {
  x <- NULL
  z <- list(coefficients = if (mlm) matrix(NA_real_, 0,
                                              ncol(y)) else numeric(),
             residuals = y, fitted.values = 0 *
               y, weights = w, rank = 0L, df.residual = if (!is.null(w)) sum(w !=
               0) else ny)
  if (!is.null(offset)) {
    z$fitted.values <- offset
    z$residuals <- y - offset
  }
}
else {
  x <- model.matrix(mt, mf, contrasts)
  z <- if (is.null(w))
    lm.fit(x, y, offset = offset, singular.ok = singular.ok,
            ...)
  else lm.wfit(x, y, w, offset = offset, singular.ok = singular.ok,
                ...)
}

class(z) <- c(if (mlm) "mlm", "lm")
z$na.action <- attr(mf, "na.action")
z$offset <- offset
z$contrasts <- attr(x, "contrasts")
z$xlevels <- .getXlevels(mt, mf)
z$call <- cl
z$terms <- mt
if (model)
  z$model <- mf
if (ret.x)
  z$x <- x

```

```
if (ret.y)
  z$y <- y
if (!qr)
  z$qr <- NULL
z
}

<bytecode: 0x643ece1f28b8>
<environment: namespace:stats>
```

What we see above is the underlying code for this particular function. We could copy and paste this into our own script and make any changes we deemed necessary, although tread carefully and test the changes you've made.

Don't worry overly if most of the code contained in functions doesn't make sense immediately. This will be especially true if you are new to R, in which case it seems incredibly intimidating. Honestly, it can be intimidating even after years of R experience. To help with that, we'll begin by making our own functions in R in the next section.

5.2. Functions in R

Functions are the bread and butter of R, the essential sustaining elements allowing you to work with R. They're made (most of the time) with the utmost care and attention but may end up being something of a Frankenstein's monster - with weirdly attached limbs. But no matter how convoluted they may be they will always faithfully do the same thing.

This means that functions can also be very stupid.

If we asked you to go to the supermarket to get us some ingredients to make *Balmoral chicken*, even if you don't know what the heck that is, you'd be able to guess and bring at least *something* back. Or you could decide to make something else. Or you could ask a chef for help. Or you could pull out your phone and search online for what *Balmoral chicken* is. The point is, even if we didn't give you enough information to do the task, you're intelligent enough to, at the very least, try to find a work around.

If instead, we asked a function to do the same, it would listen intently to our request, and then will simply return an error. It would then repeat this every single time we asked it to do the job when the task is not clear. The point here, is that code and functions cannot find workarounds to poorly provided information, which is great. It's totally reliant on you, to tell it very explicitly what it needs to do step by step.

Remember two things: the intelligence of code comes from the coder, not the computer and functions need exact instructions to work.

To prevent functions from being *too* stupid you must provide the information the function needs in order for it to work. As with the *Balmoral chicken* example, if we'd supplied a recipe list to the function, it would have managed just fine. We call this “fulfilling an argument”. The vast majority of functions require the user to fulfill at least one argument.

This can be illustrated in the pseudocode below. When we make a function we can:

- specify what arguments the user must fulfill (*e.g.* `arg1` and `arg2`)
- provide default values to arguments (*e.g.* `arg2 = TRUE`)
- define what to do with the arguments (`expression`):

```
my_function <- function(arg1, arg2, ...) {
  expression
}
```

The first thing to note is that we've used the function `function()` to create a new function called `my_function`. To walk through the above code; we're creating a function called `my_function`. Within the round brackets we specify what information (*i.e.* arguments) the function requires to run (as many or as few as needed). These arguments are then passed to the expression part of the function. The expression can be any valid R command or set of R commands and is usually contained between a pair of braces `{ }`. Once you run the above code, you can then use your new function by typing:

```
my_function(arg1, arg2)
```

Let's work through an example to help clear things up.

First we are going to create a data frame called `dishes`, where columns `lasagna`, `stovies`, `poutine`, and `tartiflette` are filled with 10 random values drawn from a bag (using the `rnorm()` function to draw random values from a Normal distribution with mean 0 and standard deviation of 1). We also include a “problem”, for us to solve later, by including 3 NA values within the `poutine` column (using `rep(NA, 3)`).

```
dishes <- data.frame(
  lasagna = rnorm(10),
  stovies = rnorm(10),
```

```
poutine = c(rep(NA, 3), rnorm(7)),  
tartiflette = rnorm(10)  
)
```

Let's say that you want to multiply the values in the variables `stovies` and `lasagna` and create a new object called `stovies_lasagna`. We can do this "by hand" using:

```
stovies_lasagna <- dishes$stovies * dishes$lasagna
```

If this was all we needed to do, we can stop here. R works with vectors, so doing these kinds of operations in R is actually much simpler than other programming languages, where this type of code might require loops (we say that R is a vectorised language). Something to keep in mind for later is that doing these kinds of operations with loops can be much slower compared to vectorisation.

But what if we want to repeat this multiplication many times? Let's say we wanted to multiply columns `lasagna` and `stovies`, `stovies` and `tartiflette`, and `poutine` and `tartiflette`. In this case we could copy and paste the code, replacing the relevant information.

```
lasagna_stovies <- dishes$lasagna * dishes$stovies  
stovies_tartiflette <- dishes$stovies * dishes$stovies  
poutine_tartiflette <- dishes$poutine * dishes$tartiflette
```

While this approach works, it's easy to make mistakes. In fact, here we've "forgotten" to change `stovies` to `tartiflette` in the second line of code when copying and pasting. This is where writing a function comes in handy. If we were to write this as a function, there is only one source of potential error (within the function itself) instead of many copy-pasted lines of code (which we also cut down on by using a function).

 Tip

As a rule of thumb if we have to do the same thing (by copy/paste & modify) 3 times or more, we just make a function for it.

In this case, we're using some fairly trivial code where it's maybe hard to make a genuine mistake. But what if we increased the complexity?

```
dishes$lasagna * dishes$stovies / dishes$lasagna + (dishes$lasagna * 10^(dishes$stovies))
-dishes$stovies - (dishes$lasagna * sqrt(dishes$stovies + 10))
```

Now imagine having to copy and paste this three times, and in each case having to change the `lasagna` and `stovies` variables (especially if we had to do it more than three times).

What we could do instead is generalize our code for `x` and `y` columns instead of naming specific dishes. If we did this, we could recycle the `x * y` code. Whenever we wanted to multiple columns together, we assign a `dishes` to either `x` or `y`. We'll assign the multiplication to the objects `lasagna_stovies` and `stovies_poutine` so we can come back to them later.

```
# Assign x and y values
x <- dishes$lasagna
y <- dishes$stovies

# Use multiplication code
lasagna_stovies <- x * y

# Assign new x and y values
x <- dishes$stovies
y <- dishes$poutine

# Reuse multiplication code
stovies_poutine <- x * y
```

This is essentially what a function does. Let's call our new function `multiply_cols()` and define it with two arguments, `x` and `y`. A function in R will simply return its last value. However, it is possible to force the function to return an earlier value if wanted/needed. Using the `return()` function is not strictly necessary in this example as R will automatically return the value of the last line of code in our function. We include it here to make this explicit.

```
multiply_cols <- function(x, y) {
  return(x * y)
}
```

Now that we've defined our function we can use it. Let's use the function to multiple the columns `lasagna` and `stovies` and assign the result to a new object called `lasagna_stovies_func`

```
lasagna_stovies_func <- multiply_cols(x = dishes$lasagna, y = dishes$stovies)
lasagna_stovies_func
```

```
[1] -0.67383476  0.09729587 -2.24091464 -0.20096247  0.02001534  0.11474034
[7]  0.71020002  0.04563115  2.00041809 -1.90902035
```

If we're only interested in multiplying `dishes$lasagna` and `dishes$stovies`, it would be overkill to create a function to do something once. However, the benefit of creating a function is that we now have that function added to our environment which we can use as often as we like. We also have the code to create the function, meaning we can use it in completely new projects, reducing the amount of code that has to be written (and retested) from scratch each time.

To satisfy ourselves that the function has worked properly, we can compare the `lasagna_stovies` variable with our new variable `lasagna_stovies_func` using the `identical()` function. The `identical()` function tests whether two objects are *exactly* identical and returns either a TRUE or FALSE value. Use `?identical` if you want to know more about this function.

```
identical(lasagna_stovies, lasagna_stovies_func)
```

```
[1] TRUE
```

And we confirm that the function has produced the same result as when we do the calculation manually. We recommend getting into a habit of checking that the function you've created works the way you think it has.

Now let's use our `multiply_cols()` function to multiply columns `stovies` and `poutine`. Notice now that argument `x` is given the value `dishes$stovies` and `y` the value `dishes$poutine`.

```
stovies_poutine_func <- multiply_cols(x = dishes$stovies, y = dishes$poutine)
stovies_poutine_func
```

```
[1]          NA          NA          NA  0.005400149 -0.410644593
[6] -0.744696576  0.308796054  0.093830731  1.215142429  1.696310555
```

So far so good. All we've really done is wrapped the code `x * y` into a function, where we ask the user to specify what their `x` and `y` variables are.

Using the function is a bit long since we have to retype the name of the data frame for each variable. For a bit of fun we can modify the function so that, we can specify the data frame as an argument and the column names without quoting them (as in a tidyverse style).

```

1 multiply_cols <- function(data, x, y) {
2   temp_var <- data %>%
3     select({{ x }}, {{ y }}) %>%
4     mutate(xy = prod(.)) %>%
5     pull(xy)
6 }
```

For this new version of the function, we added we added a `data` argument on line 1. On lines 3, we select the `x` and `y` variables provided as arguments. On line 4., we create the product of the 2 selected columns and on line 5. we extract the column we just created. We also remove the `return()` function since it was not needed

Our function is now compatible with the pipe (either native `|>` or magrittr `%>%`) function. However, since the function now uses the pipe from magrittr  and dplyr  functions, we need to load the tidyverse  package for it to work.

```

library(tidyverse)
lasagna_stovies_func <- multiply_cols(dishes, lasagna, stovies)
lasagna_stovies_func <- dishes |> multiply_cols(lasagna, stovies)
```

Now let's add a little bit more complexity. If you look at the output of `poutine_tartiflette` some of the calculations have produced NA values. This is because of those NA values we included in `poutine` when we created the `dishes` data frame. Despite these NA values, the function appeared to have worked but it gave us no indication that there might be a problem. In such cases we may prefer if it had warned us that something was wrong. How can we get the function to let us know when NA values are produced? Here's one way.

```

1 multiply_cols <- function(data, x, y) {
2   temp_var <- data %>%
3     select({{ x }}, {{ y }}) %>%
4     mutate(xy = {
5       .[1] * .[2]
6     }) %>%
```

```
7     pull(xy)
8     if (any(is.na(temp_var))) {
9         warning("The function has produced NAs")
10        return(temp_var)
11    } else {
12        return(temp_var)
13    }
14 }
```

```
stovies_poutine_func <- multiply_cols(dishes, stovies, poutine)
```

Warning in multiply_cols(dishes, stovies, poutine): The function has produced
NAs

```
lasagna_stovies_func <- multiply_cols(dishes, lasagna, stovies)
```

The core of our function is still the same, but we've now got an extra six lines of code (lines 6-11). We've included some conditional statements, `if` (lines 6-8) and `else` (lines 9-11), to test whether any NAs have been produced and if they have we display a warning message to the user. The next section of this Chapter will explain how these work and how to use them.

5.3. Conditional statements

`x * y` does not apply any logic. It merely takes the value of `x` and multiplies it by the value of `y`. Conditional statements are how you inject some logic into your code. The most commonly used conditional statement is `if`. Whenever you see an `if` statement, read it as '*If X is TRUE, do a thing*'. Including an `else` statement simply extends the logic to '*If X is TRUE, do a thing, or else do something different*'.

Both the `if` and `else` statements allow you to run sections of code, depending on a condition is either `TRUE` or `FALSE`. The pseudo-code below shows you the general form.

```
if (condition) {
    Code executed when condition is TRUE
} else {
```

```
Code executed when condition is FALSE
}
```

To delve into this a bit more, we can use an old programmer joke to set up a problem.

A programmer's partner says: '*Please go to the store and buy a carton of milk and if they have eggs, get six.*'

The programmer returned with 6 cartons of milk.

When the partner sees this, and exclaims '*Why the heck did you buy six cartons of milk?*'

The programmer replied '*They had eggs*'

At the risk of explaining a joke, the conditional statement here is whether or not the store had eggs. If coded as per the original request, the programmer should bring 6 cartons of milk if the store had eggs (condition = TRUE), or else bring 1 carton of milk if there weren't any eggs (condition = FALSE). In R this is coded as:

```
eggs <- TRUE # Whether there were eggs in the store

if (eggs == TRUE) { # If there are eggs
  n.milk <- 6 # Get 6 cartons of milk
} else { # If there are not eggs
  n.milk <- 1 # Get 1 carton of milk
}
```

We can then check `n.milk` to see how many milk cartons they returned with.

```
n.milk
```

```
[1] 6
```

And just like the joke, our R code has missed that the condition was to determine whether or not to buy eggs, not more milk (this is actually a loose example of the [Winograd Scheme](#), designed to test the *intelligence* of artificial intelligence by whether it can reason what the intended referent of a sentence is).

We could code the exact same egg-milk joke conditional statement using an `ifelse()` function.

```
eggs <- TRUE  
n.milk <- ifelse(eggs == TRUE, yes = 6, no = 1)
```

This `ifelse()` function is doing exactly the same as the more fleshed out version from earlier, but is now condensed down into a single line of code. It has the added benefit of working on vectors as opposed to single values (more on this later when we introduce loops). The logic is read in the same way; “If there are eggs, assign a value of 6 to `n.milk`, if there isn’t any eggs, assign the value 1 to `n.milk`”.

We can check again to make sure the logic is still returning 6 cartons of milk:

```
n.milk
```

```
[1] 6
```

Currently we’d have to copy and paste code if we wanted to change if eggs were in the store or not. We learned above how to avoid lots of copy and pasting by creating a function. Just as with the simple `x * y` expression in our previous `multiply_cols()` function, the logical statements above are straightforward to code and well suited to be turned into a function. How about we do just that and wrap this logical statement up in a function?

```
milk <- function(eggs) {  
  if (eggs == TRUE) {  
    6  
  } else {  
    1  
  }  
}
```

We’ve now created a function called `milk()` where the only argument is `eggs`. The user of the function specifies if `eggs` is either `TRUE` or `FALSE`, and the function will then use a conditional statement to determine how many cartons of milk are returned.

Let’s quickly try:

```
milk(eggs = TRUE)
```

```
[1] 6
```

And the joke is maintained. Notice in this case we have actually specified that we are fulfilling the `eggs` argument (`eggs = TRUE`). In some functions, as with ours here, when a function only has a single argument we can be lazy and not name which argument we are fulfilling. In reality, it's generally viewed as better practice to explicitly state which arguments you are fulfilling to avoid potential mistakes.

OK, lets go back to the `multiply_cols()` function we created above and explain how we've used conditional statements to warn the user if NA values are produced when we multiple any two columns together.

```
multiply_cols <- function(data, x, y) {
  temp_var <- data %>%
    select({{ x }}, {{ y }}) %>%
    mutate(xy = {
      .[1] * .[2]
    }) %>%
    pull(xy)

  if (any(is.na(temp_var))) {
    warning("The function has produced NAs")
    return(temp_var)
  } else {
    return(temp_var)
  }
}
```

In this new version of the function we still use `x * y` as before but this time we've assigned the values from this calculation to a temporary vector called `temp_var` so we can use it in our conditional statements. Note, this `temp_var` variable is *local* to our function and will not exist outside of the function due something called R's [scoping rules](#). We then use an `if` statement to determine whether our `temp_var` variable contains any NA values. The way this works is that we first use the `is.na()` function to test whether each value in our `temp_var` variable is an NA. The `is.na()` function returns TRUE if the value is an NA and FALSE if the value isn't an NA. We then nest the `is.na(temp_var)` function inside the function `any()` to test whether **any** of the values returned by `is.na(temp_var)` are TRUE. If at least one value is TRUE the `any()` function will return a TRUE. So, if there are any NA values in our `temp_var` variable the condition for the `if()` function will be TRUE whereas if there are no NA values present then the condition will be FALSE. If the condition is TRUE the `warning()` function generates a warning message for the user and then returns the `temp_var` variable. If the condition is FALSE the code below the `else` statement is executed which just returns the `temp_var` variable.

So if we run our modified `multiple_columns()` function on the columns `dishes$stovies` and `dishes$poutine` (which contains NAs) we will receive an warning message.

```
stovies_poutine_func <- multiply_cols(dishes, stovies, poutine)
```

Warning in `multiply_cols(dishes, stovies, poutine)`: The function has produced
NAs

Whereas if we multiple two columns that don't contain NA values we don't receive a warning message

```
lasagna_stovies_func <- multiply_cols(dishes, lasagna, stovies)
```

5.4. Combining logical operators

The functions that we've created so far have been perfectly suited for what we need, though they have been fairly simplistic. Let's try creating a function that has a little more complexity to it. We'll make a function to determine if today is going to be a good day or not based on two criteria. The first criteria will depend on the day of the week (Friday or not) and the second will be whether or not your code is working (TRUE or FALSE). To accomplish this, we'll be using `if` and `else` statements. The complexity will come from `if` statements immediately following the relevant `else` statement. We'll use such conditional statements four times to achieve all combinations of it being a Friday or not, and if your code is working or not.

We also used the `cat()` function to output text formatted correctly.

```
good.day <- function(code.working, day) {  
  if (code.working == TRUE && day == "Friday") {  
    cat(  
      "BEST.  
      DAY.  
      EVER.  
      Stop while you are ahead and go to the pub!"  
    )  
  } else if (code.working == FALSE && day == "Friday") {  
    cat("Oh well, but at least it's Friday! Pub time!")  
  }  
}
```

```

} else if (code.working == TRUE && day != "Friday") {
  cat("
So close to a good day...
shame it's not a Friday"
)
} else if (code.working == FALSE && day != "Friday") {
  cat("Hello darkness.")
}
}

```

```
good.day(code.working = TRUE, day = "Friday")
```

BEST.

DAY.

EVER.

Stop while you are ahead and go to the pub!

```
good.day(FALSE, "Tuesday")
```

Hello darkness.

Notice that we never specified what to do if the day was not a Friday? That's because, for this function, the only thing that matters is whether or not it's Friday.

We've also been using logical operators whenever we've used `if` statements. Logical operators are the final piece of the logical conditions jigsaw. Below is a table which summarises operators. The first two are logical operators and the final six are relational operators. You can use any of these when you make your own functions (or loops).

Operator	Technical Description	What it means	Example
<code>&&</code>	Logical AND	Both conditions must be met	<code>if(cond1 == test && cond2 == test)</code>
<code> </code>	Logical OR	Either condition must be met	<code>if(cond1 == test cond2 == test)</code>
<code><</code>	Less than	X is less than Y	<code>if(X < Y)</code>

Operator	Technical Description	What it means	Example
>	Greater than	X is greater than Y	<code>if(X > Y)</code>
<=	Less than or equal to	X is less/equal to Y	<code>if(X <= Y)</code>
>=	Greater than or equal to	X is greater/equal to Y	<code>if(X >= Y)</code>
==	Equal to	X is equal to Y	<code>if(X == Y)</code>
!=	Not equal to	X is not equal to Y	<code>if(X != Y)</code>

5.5. Loops

R is very good at performing repetitive tasks. If we want a set of operations to be repeated several times we use what's known as a loop. When you create a loop, R will execute the instructions in the loop a specified number of times or until a specified condition is met. There are three main types of loop in R: the *for* loop, the *while* loop and the *repeat* loop.

Loops are one of the staples of all programming languages, not just R, and can be a powerful tool (although in our opinion, used far too frequently when writing R code).

5.5.1. For loop

The most commonly used loop structure when you want to repeat a task a defined number of times is the *for* loop. The most basic example of a *for* loop is:

```
for (i in 1:5) {
  print(i)
}
```

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

But what's the code actually doing? This is a dynamic bit of code were an index `i` is iteratively replaced by each value in the vector `1:5`. Let's break it down. Because the first value in our sequence (`1:5`) is 1, the loop starts by replacing `i` with 1 and runs everything between the `{ }`. Loops conventionally use `i` as the counter, short for iteration, but you are free to use whatever you like, even your pet's name, it really does not matter (except when using nested loops, in which case the counters must be called different things, like `SenorWhiskers` and `HerrFlufferkins`).

So, if we were to do the first iteration of the loop manually

```
i <- 1
print(i)
```

[1] 1

Once this first iteration is complete, the for loop *loops* back to the beginning and replaces `i` with the next value in our `1:5` sequence (2 in this case):

```
i <- 2
print(i)
```

[1] 2

This process is then repeated until the loop reaches the final value in the sequence (5 in this example) after which point it stops.

To reinforce how for loops work and introduce you to a valuable feature of loops, we'll alter our counter within the loop. This can be used, for example, if we're using a loop to iterate through a vector but want to select the next row (or any other value). To show this we'll simply add 1 to the value of our index every time we iterate our loop.

```
for (i in 1:5) {
  print(i + 1)
}
```

[1] 2

[1] 3

[1] 4

[1] 5

[1] 6

As in the previous loop, the first value in our sequence is 1. The loop begins by replacing `i` with 1, but this time we've specified that a value of 1 must be added to `i` in the expression resulting in a value of `1 + 1`.

```
i <- 1  
i + 1
```

```
[1] 2
```

As before, once the iteration is complete, the loop moves onto the next value in the sequence and replaces `i` with the next value (2 in this case) so that `i + 1` becomes `2 + 1`.

```
i <- 2  
i + 1
```

```
[1] 3
```

And so on. We think you get the idea! In essence this is all a `for` loop is doing and nothing more.

Whilst above we have been using simple addition in the body of the loop, you can also combine loops with functions.

Let's go back to our data frame `dishes`. Previously in the Chapter we created a function to multiply two columns and used it to create our `lasagna_stovies`, `stovies_poutine`, and `poutine_tartiflette` objects. We could have used a loop for this. Let's remind ourselves what our data look like and the code for the `multiple_columns()` function.

```
dishes <- data.frame(  
  lasagna = rnorm(10),  
  stovies = rnorm(10),  
  poutine = c(rep(NA, 3), rnorm(7)),  
  tartiflette = rnorm(10)  
)
```

```
multiply_cols <- function(data, x, y) {  
  temp_var <- data %>%  
    select({{ x }}, {{ y }}) %>%  
    mutate(xy = {
```

```

. [1] * . [2]
}) %>%
pull(xy)
if (any(is.na(temp_var))) {
  warning("The function has produced NAs")
  return(temp_var)
} else {
  return(temp_var)
}
}

```

To use a list to iterate over these columns we need to first create an empty list (remember Section 3.2.3?) which we call `temp` (short for temporary) which will be used to store the output of the `for` loop.

```

temp <- list()
for (i in 1:(ncol(dishes) - 1)) {
  temp[[i]] <- multiply_cols(dishes, x = colnames(dishes)[i], y = colnames(dishes)[i + 1])
}

```

```

Warning in multiply_cols(dishes, x = colnames(dishes)[i], y =
colnames(dishes)[i + : The function has produced NAs
Warning in multiply_cols(dishes, x = colnames(dishes)[i], y =
colnames(dishes)[i + : The function has produced NAs

```

When we specify our `for` loop notice how we subtracted 1 from `ncol(dishes)`. The `ncol()` function returns the number of columns in our `dishes` data frame which is 4 and so our loop runs from `i = 1` to `i = 4 - 1` which is `i = 3`.

So in the first iteration of the loop `i` takes on the value 1. The `multiply_cols()` function multiplies the `dishes[, 1]` (`lasagna`) and `dishes[, 1 + 1]` (`stovies`) columns and stores it in the `temp[[1]]` which is the first element of the `temp` list.

The second iteration of the loop `i` takes on the value 2. The `multiply_cols()` function multiplies the `dishes[, 2]` (`stovies`) and `dishes[, 2 + 1]` (`poutine`) columns and stores it in the `temp[[2]]` which is the second element of the `temp` list.

The third and final iteration of the loop `i` takes on the value 3. The `multiply_cols()` function multiplies the `dishes[, 3]` (`poutine`) and `dishes[, 3 + 1]` (`tartiflette`) columns and stores it in the `temp[[3]]` which is the third element of the `temp` list.

Again, it's a good idea to test that we are getting something sensible from our loop (remember, check, check and check again!). To do this we can use the `identical()` function to compare the variables we created by hand with each iteration of the loop manually.

```
lasagna_stovies_func <- multiply_cols(dishes, lasagna, stovies)
i <- 1
identical(
  multiply_cols(dishes, colnames(dishes)[i], colnames(dishes)[i + 1]),
  lasagna_stovies_func
)
```

[1] TRUE

```
stovies_poutine_func <- multiply_cols(dishes, stovies, poutine)
```

Warning in `multiply_cols(dishes, stovies, poutine)`: The function has produced
NAs

```
i <- 2
identical(
  multiply_cols(dishes, colnames(dishes)[i], colnames(dishes)[i + 1]),
  stovies_poutine_func
)
```

Warning in `multiply_cols(dishes, colnames(dishes)[i], colnames(dishes)[i + :`
The function has produced NAs

[1] TRUE

If you can follow the examples above, you'll be in a good spot to begin writing some of your own for loops. That said there are other types of loops available to you.

5.5.2. While loop

Another type of loop that you may use (albeit less frequently) is the `while` loop. The `while` loop is used when you want to keep looping until a specific logical condition is satisfied (contrast this with the `for` loop which will always iterate through an entire sequence).

The basic structure of the while loop is:

```
while (logical_condition) {
    expression
}
```

A simple example of a while loop is:

```
i <- 0
while (i <= 4) {
    i <- i + 1
    print(i)
}
```

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

Here the loop will only continue to pass values to the main body of the loop (the `expression` body) when `i` is less than or equal to 4 (specified using the `<=` operator in this example). Once `i` is greater than 4 the loop will stop.

There is another, very rarely used type of loop; the `repeat` loop. The `repeat` loop has no conditional check so can keep iterating indefinitely (meaning a break, or “stop here”, has to be coded into it). It’s worthwhile being aware of its existence, but for now we don’t think you need to worry about it; the `for` and `while` loops will see you through the vast majority of your looping needs.

5.5.3. When to use a loop?

Loops are fairly commonly used, though sometimes a little overused in our opinion. Equivalent tasks can be performed with functions, which are often more efficient than loops. Though this raises the question when should you use a loop?

In general loops are implemented inefficiently in R and should be avoided when better alternatives exist, especially when you're working with large datasets. However, loops are sometimes the only way to achieve the result we want.

Some examples of when using loops can be appropriate:

- Some simulations (e.g. the Ricker model can, in part, be built using loops)
- Recursive relationships (a relationship which depends on the value of the previous relationship [“to understand recursion, you must understand recursion”])
- More complex problems (e.g., how long since the last badger was seen at site j , given a pine marten was seen at time t , at the same location j as the badger, where the pine marten was detected in a specific 6 hour period, but exclude badgers seen 30 minutes before the pine marten arrival, repeated for all pine marten detections)
- While loops (keep jumping until you've reached the moon)

5.5.4. If not loops, then what?

In short, use the apply family of functions; `apply()`, `lapply()`, `tapply()`, `sapply()`, `vapply()`, and `mapply()`. The apply functions can often do the tasks of most “home-brewed” loops, sometimes faster (though that won’t really be an issue for most people) but more importantly with a much lower risk of error. A strategy to have in the back of your mind which may be useful is; for every loop you make, try to remake it using an apply function (often `lapply` or `sapply` will work). If you can, use the apply version. There’s nothing worse than realizing there was a small, tiny, seemingly meaningless mistake in a loop which weeks, months or years down the line has propagated into a huge mess. We strongly recommend trying to use the apply functions whenever possible.

lapply

Your go to apply function will often be `lapply()` at least in the beginning. The way that `lapply()` works, and the reason it is often a good alternative to for loops, is that it will go through each element in a list and perform a task (*i.e.* run a function). It has the added benefit that it will output the results as a list - something you’d have to otherwise code yourself into a loop.

An `lapply()` has the following structure:

```
lapply(X, FUN)
```

Here `X` is the vector which we want to do *something* to. `FUN` stands for how much fun this is (just kidding!). It's also short for "function".

Let's start with a simple demonstration first. Let's use the `lapply()` function create a sequence from 1 to 5 and add 1 to each observation (just like we did when we used a for loop):

```
lapply(0:4, function(a) {  
  a + 1  
})
```

```
[[1]]
```

```
[1] 1
```

```
[[2]]
```

```
[1] 2
```

```
[[3]]
```

```
[1] 3
```

```
[[4]]
```

```
[1] 4
```

```
[[5]]
```

```
[1] 5
```

Notice that we need to specify our sequence as `0:4` to get the output `1 ,2 ,3 ,4 , 5` as we are adding 1 to each element of the sequence. See what happens if you use `1:5` instead.

Equivalently, we could have defined the function first and then used the function in `lapply()`

```
add_fun <- function(a) {  
  a + 1  
}  
lapply(0:4, add_fun)
```

```
[[1]]
```

```
[1] 1
```

```
[[2]]
```

```
[1] 2
```

```
[[3]]
```

```
[1] 3
```

```
[[4]]
```

```
[1] 4
```

```
[[5]]
```

```
[1] 5
```

The `sapply()` function does the same thing as `lapply()` but instead of storing the results as a list, it stores them as a vector.

```
sapply(0:4, function(a) {  
  a + 1  
})
```

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

As you can see, in both cases, we get exactly the same results as when we used the for loop.

Chapter 6

Reproducible reports with Quarto

⚠ Warning

screenshot are still with R markdown will be update soon

This chapter will introduce you to creating reproducible reports using R markdown / Quarto to encourage best (or better) practice to facilitate open science. It will first describe what R markdown and Quarto are and why you might want to consider using it, describe how to create a Quarto document using RStudio and then how to convert this document to a html or pdf formatted report. During this Chapter you will learn:

- the different components of a Quarto document
- how to format text, graphics and tables within the document
- how to avoid some of the common difficulties using Quarto.

6.1. What is R markdown / Quarto?

6.1.1. R Markdown

R markdown is a simple and easy to use plain text language used to combine your R code, results from your data analysis (including plots and tables) and written commentary into a single nicely formatted and reproducible document (like a report, publication, thesis chapter or a web page like this one).

Technically, R markdown is a combination of three languages, R, Markdown and YAML (yet another markup language). Both Markdown and YAML are a type of ‘markup’ language. A markup language simply provides a way of creating an easy to read plain text file which can incorporate formatted text, images, headers and links to

other documents. If you're interested you can find more information about markup languages [here](#). Actually, you are exposed to a markup language on a daily basis, as most of the internet content you digest every day is underpinned by a markup language called HTML (**Hypertext Markup Language**). Anyway, the main point is that R markdown is very easy to learn (much, much easier than HTML) and when used with a good IDE (RStudio or VS Code) it's ridiculously easy to integrate into your workflow to produce feature rich content (so why wouldn't you?!).

6.1.2. Quarto?

Quarto is a multi-language, next generation version of R Markdown from Posit, with many new features and capabilities and is compatible not only with R but also with other language like Python and Julia. Like R Markdown, Quarto uses `knitr`  package to execute R code, and is therefore able to render most existing **.Rmd** files without modification. However, it also comes with a plethora of new functionalities. More importantly, it makes it much easier to create different type of output since the coding is homogenize for specific format without having to rely on different r packages each with there own specificity (*e.g* bookdown, hugodown, blogdown, thesisdown, rticles, xaringan, ...).

In the rest of this chapter, we will talk about Quarto but a lot can be done with R markdown. Quarto uses **.qmd** files while R markdown works with **.Rmd** but Quarto can render **.Rmd** files too.

6.2. Why use Quarto?

During the previous Chapters we talked a lot about conducting your research in a robust and reproducible manner to facilitate open science. In a nutshell, open science is about doing all we can to make our data, methods, results and inferences transparent and available to everyone. Some of the main tenets of open science are described [here](#) and include:

- Transparency in experimental methodology, observation, collection of data and analytical methods.
- Public availability and re-usability of scientific data
- Public accessibility and transparency of scientific communication
- Using web-based tools to facilitate scientific collaboration

By now all of you will (hopefully) be using R to explore and analyse your interesting data. As such, you're already well along the road to making your analysis more reproducible, transparent and shareable. However, perhaps your current workflow looks something like this:

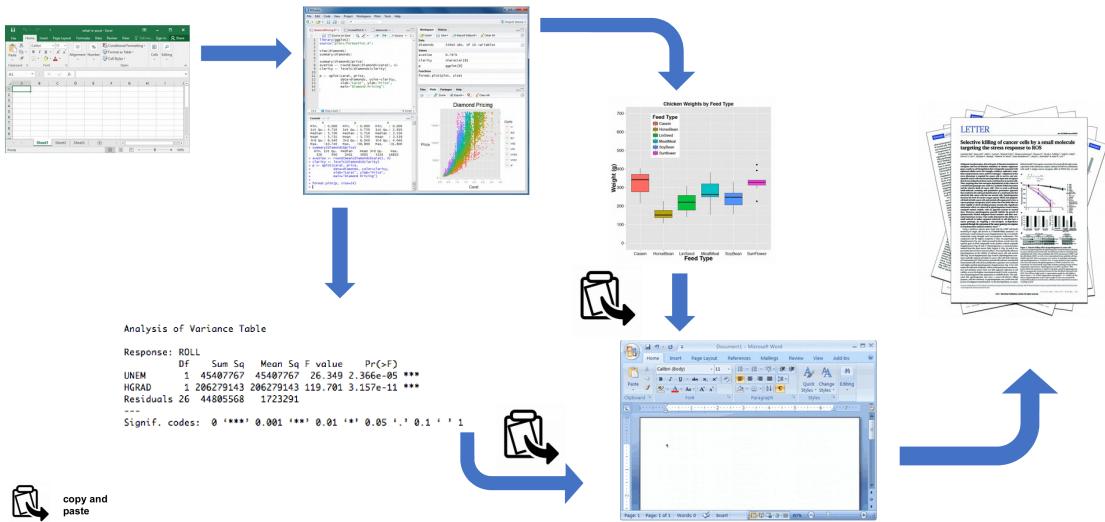


Figure 6.1.: Non-reproducible workflow

Your data is imported from your favourite spreadsheet software into R, you write your R code to explore and analyse your data, you save plots as external files, copy tables of analysis output and then manually combine all of this and your written prose into a single MS Word document (maybe for a paper or thesis chapter). Whilst there is nothing particularly wrong with this approach (and it's certainly better than using point and click software to analyse your data) there are some limitations:

- It's not particularly reproducible. Because this workflow separates your R code from the final document there are multiple opportunities for undocumented decisions to be made (which plots did you use? what analysis did/didn't you include? etc).
- It's inefficient. If you need to go back and change something (create a new plot or update your analysis etc) you will need to create or amend multiple documents increasing the risk of mistakes creeping into your workflow.
- It's difficult to maintain. If your analysis changes you again need to update multiple files and documents.
- It can be difficult to decide what to share with others. Do you share all of your code (initial data exploration, model validation etc) or just the code specific to your final document? It's quite a common (and bad!) practice for researchers to maintain two R scripts, one used for the actual analysis and one to share with the final paper or thesis chapter. This can be both time consuming and confusing and should be avoided.

Perhaps a more efficient and robust workflow would look something like this:

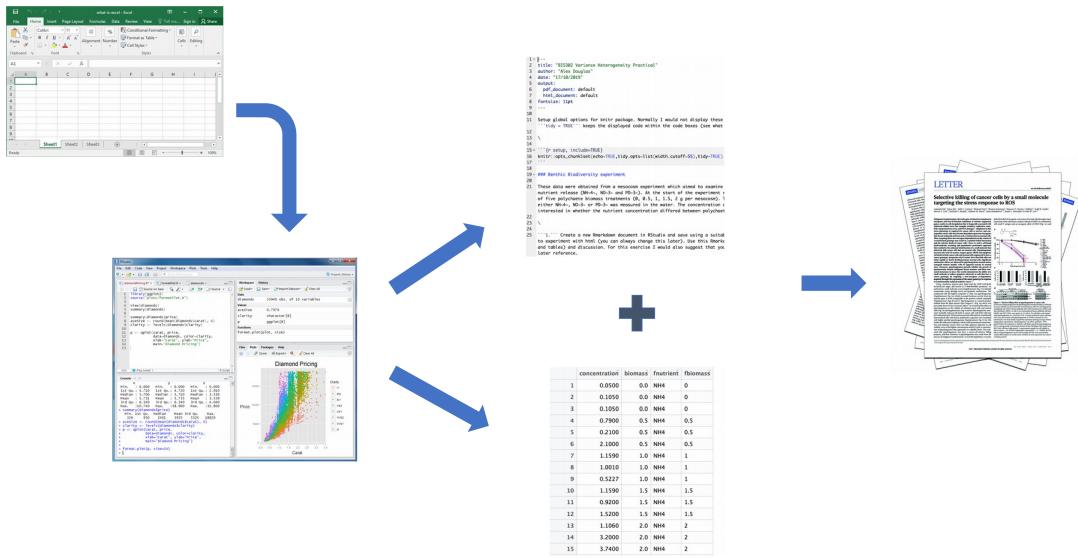


Figure 6.2.: A-reproducible (and more efficient) workflow

Your data is imported into R as before but this time all of the R code you used to analyse your data, produce your plots and your written text (Introduction, Materials and Methods, Discussion etc) is contained within a single Quarto document which is then used (along with your data) to automatically create your final document. This is exactly what Quarto allows you to do.

Some of the advantages of using Quarto include:

- Explicitly links your data with your R code and output creating a fully reproducible workflow. **ALL** of the R code used to explore, summarise and analyse your data can be included in a single easy to read document. You can decide what to include in your final document (as you will learn below) but all of your R code can be included in the Quarto document.
- You can create a wide variety of output formats (pdf, html web pages, MS Word and many others) from a single Quarto document which enhances both collaboration and communication.
- Enhances transparency of your research. Your data and Quarto file can be included with your publication or thesis chapter as supplementary material or hosted on a GitHub repository (see Chapter 7).
- Increases the efficiency of your workflow. If you need to modify or extend your current analysis you just need to update your Quarto document and these changes will automatically be included in your final document.

6.3. Get started with Quarto

Quarto integrates really well with R Studio and VS Code, and provide both a source editor as well as a visual editor providing an experience close to your classic WYSIWYG (what you see is what you write) writing software (e.g. Microsoft Word or LibreOffice writer)

6.3.1. Installation

To use Quarto you will first need to install the Quarto software and the quarto 📦 package (with its dependencies). You can find instructions on how to do this in Section 1.1.1 and on the [Quarto website](#). If you would like to create pdf documents (or MS Word documents) from your Quarto file you will also need to install a version of L^AT_EX on your computer. If you've not installed L^AT_EX before, we recommend that you install [TinyTeX](#). Again, instructions on how to do this can be found at Section 1.1.1.

6.3.2. Create a Quarto document, .qmd

Right, time to create your first Quarto document. Within RStudio, click on the menu File -> New File -> Quarto.... In the pop up window, give the document a ‘Title’ and enter the ‘Author’ information (your name) and select HTML as the default output. We can change all of this later so don’t worry about it for the moment.

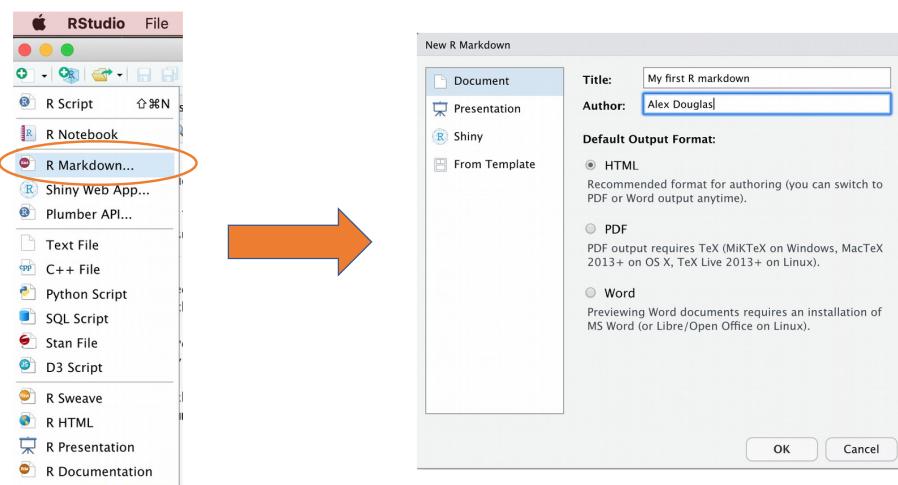


Figure 6.3.: Creating a Quarto document

You will notice that when your new Quarto document is created it includes some example Quarto code. Normally you would just highlight and delete everything in the document except the information at the top between the --- delimiters (this is called the YAML header which we will discuss in a bit) and then start writing your own code. However, just for now we will use this document to practice converting Quarto to both html and pdf formats and check everything is working.

```
1 ---  
2 title: "My first R markdown"  
3 author: "Alex Douglas"  
4 date: "19/02/2020"  
5 output: html_document  
6 ---  
7  
8 ````{r setup, include=FALSE}  
9 knitr::opts_chunk$set(echo = TRUE)  
10 ...  
11  
12 ## R Markdown  
13  
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For  
more details on using R Markdown see <http://rmarkdown.rstudio.com>.  
15  
16 When you click the **Knit** button a document will be generated that includes both content as well as the output of any  
embedded R code chunks within the document. You can embed an R code chunk like this:  
17  
18 ````{r cars}  
19 summary(cars)  
20 ...  
21  
22 ## Including Plots  
23  
24 You can also embed plots, for example:  
25  
26 ````{r pressure, echo=FALSE}  
27 plot(pressure)  
28 ...  
29  
30 Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.  
31
```

Figure 6.4.: A new Quarto document

Once you've created your Quarto document it's good practice to save this file somewhere convenient (Section 1.4 and Figure 1.11). You can do this by selecting File -> Save from RStudio menu (or use the keyboard shortcut ctrl + s on Windows or cmd + s on a Mac) and enter an appropriate file name (maybe call it `my_first_quarto`). Notice the file extension of your new Quarto file is `.qmd`.

Now, to convert your `.qmd` file to a HTML document click on the little black triangle next to the Knit icon at the top of the source window and select `knit to HTML`

RStudio will now ‘knit’ (or render) your `.qmd` file into a HTML file. Notice that there is a new Quarto tab in your console window which provides you with information on the rendering process and will also display any errors if something goes wrong.

If everything went smoothly a new HTML file will have been created and saved in the same directory as your `.qmd` file (ours will be called `my_first_quarto.html`). To view this document simply double click on the file to open in a browser (like Chrome or Firefox) to display the rendered content. RStudio will also display a preview of the



Figure 6.5.: Knitting a Qmd file

rendered file in a new window for you to check out (your window might look slightly different if you're using a Windows computer).

Great, you've just rendered your first Quarto document. If you want to knit your .qmd file to a pdf document then all you need to do is choose `knit to PDF` instead of `knit to HTML` when you click on the knit icon. This will create a file called `my_first_quarto.pdf` which you can double click to open. Give it a go!

You can also knit an .qmd file using the command line in the console rather than by clicking on the knit icon. To do this, just use the `quarto_render()` function from the `quarto` package as shown below. Again, you can change the output format using the `output_format` = argument as well as many other options.

```

library(quarto)

quarto_render('my_first_quarto.qmd', output_format = 'html_document')

# alternatively if you don't want to load the quarto package

quarto::quarto_render('my_first_quarto.Rmd', output_format = 'html_document')

```

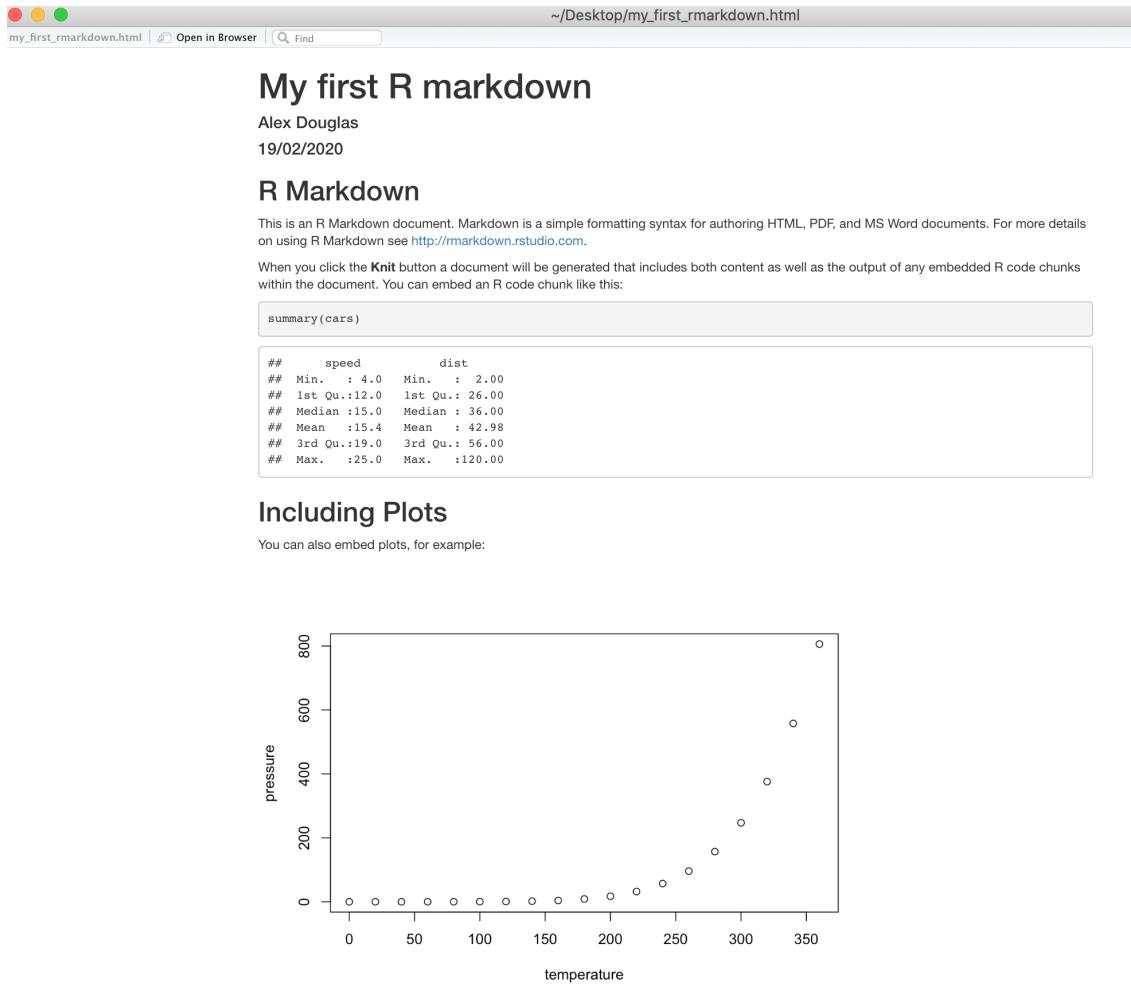


Figure 6.6.: A my first rendered html

6.4. Quarto document (.qmd) anatomy

OK, now that you can render a Quarto file in RStudio into both HTML and pdf formats let's take a closer look at the different components of a typical Quarto document. Normally each Quarto document is composed of 3 main components:

1. a YAML header
2. formatted text
3. code chunks.

```

YAML header
{[

1 ---  

2 title: "BIS502 Variance Heterogeneity Practical"  

3 author: "Alex Douglas"  

4 date: "17/10/2019"  

5 output:  

6   pdf_document: default  

7   html_document: default  

8   fontsize: 11pt  

9 ---  

10  

11 Setup global options for knitr package. Normally I would not display these but I leave them here for your information. The arguments `width.cutoff` and `tidy = TRUE` keeps the displayed code within the code boxes (see what happens if you omit this).  

12 ````r setup, include=TRUE}  

13 knitr::opts_chunk$set(echo=TRUE,tidy.opts=list(width.cutoff=55),tidy=TRUE)  

14 ````  

15  

16  

17 ## Benthic Biodiversity experiment  

18 These data were obtained from a mesocosm experiment which aimed to examine the effect of benthic polychaete (*Nereis diversicolor*) biomass on sediment nutrient release (NH4-, NO3- and PO3-). At the start of the experiment replicate mesocosms were filled with  

19 Import all the packages required for this exercise:  

20  

21 ````r import data}  

22 nereis <- read.table("/Users/nhy163/Documents/Alex/tmp/Nereis2.txt", header = TRUE)  

23 nereis$biomass <- factor(nereis$biomass)  

24 str(nereis)  

25  

26  

27 3. How many replicates are there for each biomass and nutrient combination?  

28

```

Figure 6.7.: Structure of a qmd file

6.4.1. YAML header

YAML stands for ‘**YAML Ain’t Markup Language**’ (it’s an ‘in’ [joke!](#)) and this optional component contains the metadata and options for the entire document such as the author name, date, output format, etc. The YAML header is surrounded before and after by a --- on its own line. In RStudio a minimal YAML header is automatically created for you when you create a new Quarto document as we did above (Section 6.3.2) but you can change this any time. A simple YAML header may look something like this:

```

---
title: My first Quarto document
author: Jane Doe

```

```
date: March 01, 2020
format: html
---
```

In the YAML header above the output format is set to HTML. If you would like to change the output to pdf format then you can change it from `format: html` to `format: pdf` (you can also set more than one output format if you like). You can also change the default font and font size for the whole document and even include fancy options such as a table of contents and inline references and a bibliography. If you want to explore the plethora of other options see [here](#). Just a note of caution, many of the options you can specify in the YAML header will work with both HTML and pdf formatted documents, but not all. If you need multiple output formats for your Quarto document check whether your YAML options are compatible between these formats. Also, indentation in the YAML header has a meaning, so be careful when aligning text. For example, if you want to include a table of contents you would modify the `output:` field in the YAML header as follows

```
---
title: My first Quarto document
author: Bob Hette
date: March 01, 2020
format:
  html:
    toc: true
---
```

6.4.2. Formatted text

As mentioned above, one of the great things about Quarto is that you don't need to rely on your word processor to bring your R code, analysis and writing together. Quarto is able to render (almost) all of the text formatting that you are likely to need such as italics, bold, strike-through, super and subscript as well as bulleted and numbered lists, headers and footers, images, links to other documents or web pages and also equations. However, in contrast to your familiar *What-You-See-Is-What-You-Get* ([WYSIWYG](#)) word processing software you don't see the final formatted text in your Quarto document (as you would in MS Word), rather you need to 'markup' the formatting in your text ready to be rendered in your output document. At first, this might seem like a right pain in the proverbial but it's actually very easy to do and also has many [advantages](#) (do you find yourself spending more time on making your text look pretty in MS Word rather than writing good content?!).

Here is an example of marking up text formatting in an Quarto document

```
#### Tadpole sediment experiment
```

These data were obtained from a mesocosm experiment which aimed to examine the effect of bullfrog tadpoles (*Lithobates catesbeianus*) biomass on sediment nutrient (NH₄, NO₃ and PO₄) release.

At the start of the experiment 15 replicate mesocosms were filled with 20 cm² of **homogenised** marine sediment and assigned to one of five tadpole biomass treatments.

which would look like this in the final rendered document (can you spot the markups?)

Tadpole sediment experiment

These data were obtained from a mesocosm experiment which aimed to examine the effect of bullfrog tadpoles (*Lithobates catesbeianus*) biomass on sediment nutrient (NH₄, NO₃ and PO₄) release. At the start of the experiment 15 replicate mesocosms were filled with 20 cm² of **homogenised** marine sediment and assigned to one of five tadpole biomass treatments.

Emphasis

Some of the most common markdown syntax for providing emphasis and formatting text is given below.

Goal	Quarto	output
bold text	**mytext**	mytext
italic text	*mytext*	<i>mytext</i>
strikethrough	~~mytext~~	mytext
superscript	mytext ²	mytext ²
subscript	mytext ₂	mytext ₂

Interestingly there is no underline in R markdown syntax by default, for more or less esoteric reasons (e.g. an underline is considered a stylistic element (there may well be other [reasons](#))). Quarto fixed that problem, you can simply do [text to underline]{.underline} to underline your text.

White space and line breaks

One of the things that can be confusing for new users of markdown is the use of spaces and carriage returns (the enter key on your keyboard). In markdown, multiple spaces within the text are generally ignored as are carriage returns. For example this markdown text

```
These      data were      obtained from a
mesocosm experiment which      aimed to examine the
effect
of          bullfrog tadpoles (*Lithobates catesbeianus*) biomass.
```

will be rendered as

These data were obtained from a mesocosm experiment which aimed to examine the effect of bullfrog tadpoles (*Lithobates catesbeianus*) biomass.

This is generally a good thing (no more random multiple spaces in your text). If you want your text to start on a new line then you can simply add two blank spaces at the end of the preceding line

```
These data were obtained from a
mesocosm experiment which aimed to examine the
effect bullfrog tadpoles (Lithobates catesbeianus) biomass.
```

If you really want multiple spaces within your text then you can use the **Non breaking space tag**

```
These &nbs; &nbs; &nbs; data were &nbs; &nbs; &nbs; obtained from a
mesocosm experiment which &nbs; &nbs; aimed to examine the
effect &nbs; &nbs; &nbs; bullfrog tadpoles (*Lithobates catesbeianus*) biomass.
```

```
These      data were      obtained from a
mesocosm experiment which      aimed to examine the
effect      bullfrog tadpoles (Lithobates catesbeianus) biomass.
```

Headings

You can add headings and subheadings to your Quarto document by using the # symbol at the beginning of the line. You can decrease the size of the headings by simply adding more # symbols. For example

```
# Header 1  
## Header 2  
### Header 3  
#### Header 4  
##### Header 5  
###### Header 6
```

results in headings in decreasing size order

Header 1

Header 2

Header 3

Header 4

Header 5

Comments

As you can see above the meaning of the # symbol is different when formatting text in an Quarto document compared to a standard R script (which is used to included a comment - remember?!). You can, however, use a # symbol to comment code inside a code chunk (Section 6.4.1) as usual (more about this in a bit). If you want to include a comment in your Quarto document outside a code chunk which won't be included in the final rendered document then enclose your comment between <!-- and -->.

```
<!--  
this is an example of how to format a comment using Quarto.  
-->
```

Lists

If you want to create a bullet point list of text you can format an unordered list with sub items. Notice that the sub-items need to be indented.

```
- item 1
- item 2
  + sub-item 2
  + sub-item 3
- item 3
- item 4
```

- item 1
- item 2
 - sub-item 2
 - sub-item 3
- item 3
- item 4

If you need an ordered list

```
1. item 1
1. item 2
  + sub-item 2
  + sub-item 3
1. item 3
1. item 4
```

1. item 1
2. item 2
 - sub-item 2
 - sub-item 3
3. item 3
4. item 4

Links

In addition to images you can also include links to webpages or other links in your document. Use the following syntax to create a clickable link to an existing webpage. The link text goes between the square brackets and the URL for the webpage between the round brackets immediately after.

```
You can include a text for your clickable [link] (https://www.worldwildlife.org)
```

which gives you:

You can include a text for your clickable [link](https://www.worldwildlife.org)

6.4.1. Code chunks

Now to the heart of the matter. To include R code into your Quarto document you simply place your code into a ‘code chunk’. All code chunks start and end with three backticks ` `` `. Note, these are also known as ‘grave accents’ or ‘back quotes’ and are not the same as an apostrophe! On most keyboards you can [find the backtick](#) on the same key as tilde (~).

```
```{r}
Any valid R code goes here
```
```

You can insert a code chunk by either typing the chunk delimiters ````{r}` and ````` manually or use your IDE option (RStudio toolbar (the Insert button) or by clicking on the menu `Code -> Insert Chunk`. In VS Code you can use code snippets) Perhaps an even better way is to get familiar with the keyboard shortcuts for your IDE or code snippets.

There are many things you can do with code chunks: you can produce text output from your analysis, create tables and figures and insert images amongst other things. Within the code chunk you can place rules and arguments between the curly brackets `{}` that give you control over how your code is interpreted and output is rendered. These are known as chunk options. The only mandatory chunk option is the first argument which specifies which language you're using (`r` in our case but [other](#) languages are supported). Note, chunk options can be written in two ways:

1. either all of your chunk options must be written between the curly brackets on one line with no line breaks
2. or they can be written using a YAML notation within the code chunk using `#|` notation at the beginning of the line.

We are using the YAML notation for code chunk options since we find it much easier to read when you have multiple options of long captions.

You can also specify an optional code chunk name (or label) which can be useful when trying to debug problems and when performing advanced document rendering. In the following block, we name the code chunk `summary-stats`, load the package `ggplot2` , create a dataframe (`dataf`) with two variables `x` and `y`, use the `summary()` function to display some summary statistics and plot a scatterplot of the data with `ggplot()`. When we run the code chunk both the R code and the resulting output are displayed in the final document.

```
```{r, summary-stats, echo = TRUE, fig.cap = "Caption for a simple figure but making the chunk op
library(ggplot)
x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataf <- data.frame(x = x, y = y)
```

```
summary(dataf)
ggplot(dataf, aes(x = x, y = y)) + geom_point()
```
```

```
```{r}
#| label: summary-stats
#| echo: true
#| fig-cap = "Caption for a simple figure but making the chunk options long and hard to read"

x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataf <- data.frame(x = x, y = y)

summary(dataf)
ggplot(dataf, aes(x = x, y = y)) + geom_point()
```
```

Both will output

```
x <- 1:10      # create an x variable
y <- 10:1      # create a y variable
dataf <- data.frame(x = x, y = y)

summary(dataf)
```

| | x | y |
|----------|--------|--------|
| Min. | : 1.00 | : 1.00 |
| 1st Qu.: | 3.25 | 3.25 |
| Median : | 5.50 | 5.50 |
| Mean : | 5.50 | 5.50 |

```
3rd Qu.: 7.75    3rd Qu.: 7.75
Max.     :10.00    Max.     :10.00
```

```
ggplot(dataf, aes(x = x, y = y)) + geom_point()
```

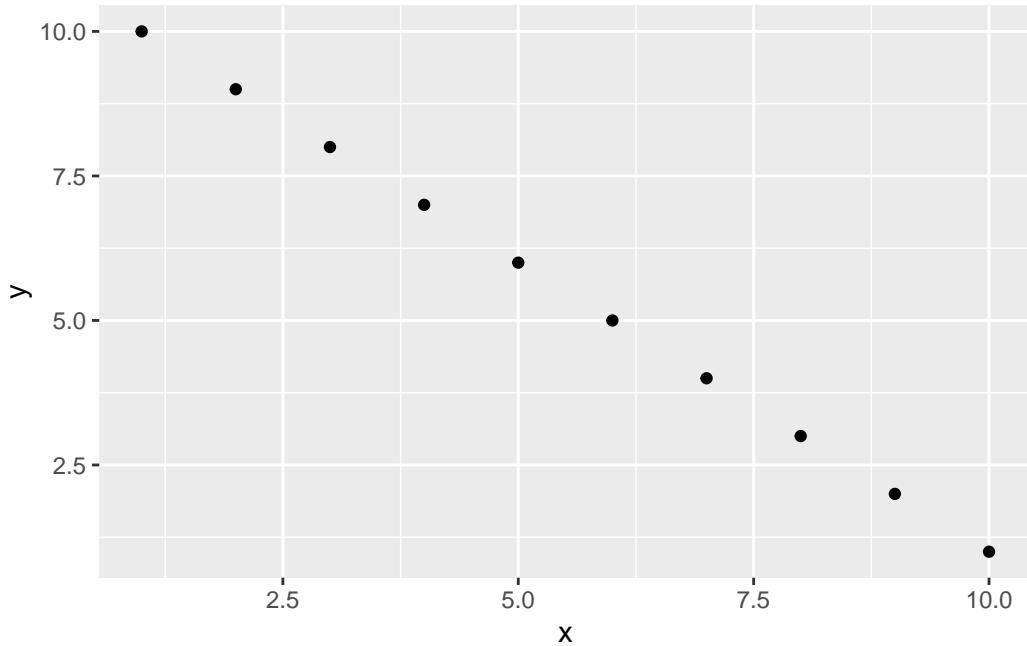


Figure 6.8.: Caption for a simple figure but making the chunk options long and hard to read

When using chunk names make sure that you don't have duplicate chunk names in your Quarto document and avoid spaces and full stops as this will cause problems when you come to knit your document (We use a - to separate words in our chunk names).

If we wanted to only display the output of our R code (just the summary statistics for example) and not the code itself in our final document we can use the chunk option `echo=FALSE`

```
```{r}
#| label: summary-stats2
#| echo: false
x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataf <- data.frame(x = x, y = y)
```

```
summary(dataf)
````
```

| | x | y |
|---------|-------|-------|
| Min. | 1.00 | 1.00 |
| 1st Qu. | 3.25 | 3.25 |
| Median | 5.50 | 5.50 |
| Mean | 5.50 | 5.50 |
| 3rd Qu. | 7.75 | 7.75 |
| Max. | 10.00 | 10.00 |

To display the R code but not the output use the `results='hide'` chunk option.

```
```{r}
#| label: summary-stats
#| results: 'hide'
x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataf <- data.frame(x = x, y = y)
summary(dataf)
````
```

```
x <- 1:10      # create an x variable
y <- 10:1      # create a y variable
dataf <- data.frame(x = x, y = y)
summary(dataf)
```

Sometimes you may want to execute a code chunk without showing any output at all. You can suppress the entire output using the chunk option `include: false`.

```
```{r}
#| label: summary-stats4
#| include: false
x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataf <- data.frame(x = x, y = y)
summary(dataf)
```

```

There are a large number of chunk options documented [here](#) with a more condensed version [here](#). Perhaps the most commonly used are summarised below with the default values shown.

| Chunk option | default value | Function |
|--------------|-------------------|---|
| echo | echo: true | If <code>false</code> , will not display the code in the final document |
| results | results: 'markup' | If 'hide', will not display the code's results in the final document. |

If ‘hold’, will delay displaying all output pieces until the end of the chunk. If ‘asis’, will pass through results without reformatting them. || `include | include: true |` If `false`, will run the chunk but not include the chunk in the final document. || `eval | eval: true |` If `false`, will not run the code in the code chunk. || `message | message: true |` If `false`, will not display any messages generated by the code. || `warning | warning: true |` If `false`, will not display any warning messages generated by the code. |

6.4.2. Inline R code

Up till now we've been writing and executing our R code in code chunks. Another great reason to use Quarto is that we can also include our R code directly within our text. This is known as ‘inline code’. To include your code in

your Quarto text you simply write `r write your code here`. This can come in really useful when you want to include summary statistics within your text. For example, we could describe the `iris` dataset as follows:

```
Morphological characteristics (variable names:  
`r names(iris)[1:4]` were measured from  
`r nrow(iris)` *Iris sp.* plants from  
`r length(levels(iris$Species))` different species.  
The mean Sepal length was  
`r round(mean(iris$Sepal.Length), digits = 2)` mm.
```

which will be rendered as

Morphological characteristics (variable names: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)
were measured from 150 *iris* plants from 3 different species. The mean Sepal length was 5.84 mm.

The great thing about including inline R code in your text is that these values will automatically be updated if your data changes.

6.4.3. Images and photos

A useful feature is the ability to embed images and links to web pages (or other documents) into your Quarto document. You can include images into your Quarto document in a number of different ways. Perhaps the simplest method is to use the Quarto markdown format:

```
! [Image caption] (path/to/you/image){options}
```

Here is an example with an image taking 75% of the width and centered.

! [Waiting for the eclipse] (images/markdown/eclipse_ready.jpg){fig-align="center" width="75%"}
resulting in:



Figure 6.9.: Waiting for the eclipse

An alternative way of including images in your document is to use the `include_graphics()` function from the `knitr` package. The following code will produce similar output.

```
```{r}
#| label: fig-knitr
#| fig-align: center
#| out-width: 75%
#| fig-cap: Waiting for the eclipse
knitr:::include_graphics("images/markdown/eclipse_ready.jpg")
```

```

The code above will only work if the image file (`eclipse_ready.jpg`) is in the right place relative to where you saved your `.qmd` file. In the example the image file is in a sub directory (folder) called `images/markdown` in the directory where we saved our `my_first_quarto.qmd` file. You can embed images saved in many different file types but perhaps the most common are `.jpg` and `.png`.

6.4.4. Figures

By default, figures produced by R code will be placed immediately after the code chunk they were generated from. For example:

```
```{r}
#| label: fig-simple-plot
#| fig-cap: A simple plot
x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataaf <- data.frame(x = x, y = y)
plot(dataaf$x, dataaf$y, xlab = "x axis", ylab = "y axis")
```

```

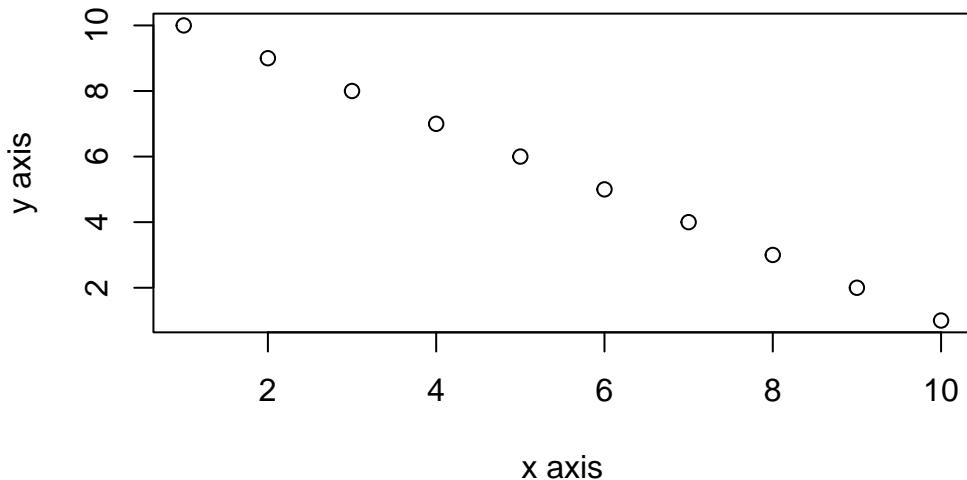


Figure 6.10.: A simple plot

The `fig-cap:` chunk option allow to provide a figure caption recognized by Quarto and using in figure numbering and cross referencing (Section 6.4.6).

If you want to change the plot dimensions in the final document you can use the `fig-width:` and `fig-height:` chunk options (in inches!). You can also change the alignment of the figure using the `fig-align:` chunk option.

```
```{r}
#| label: fig-simple-plot2
#| fig-cap: A shrinked figure
#| fig-width: 4
#| fig-height: 3
#| fig-align: center
x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```

```

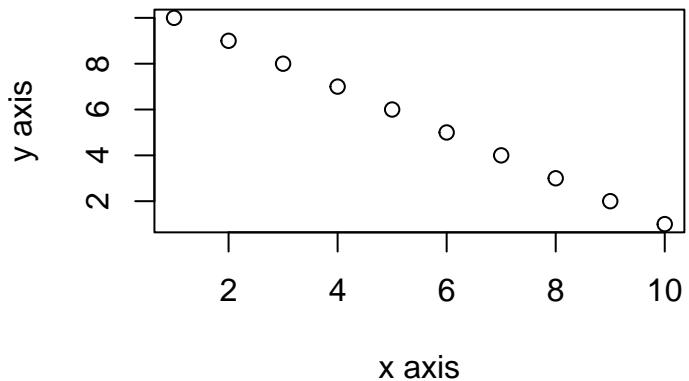


Figure 6.11.: A shrinked figure

You can add a figure caption using the `fig-cap:` option.

```
```{r}
#| label: fig-simple-plot-cap
#| class-source: fold-show
#| fig-cap: A simple plot
#| fig-align: center
x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```
```

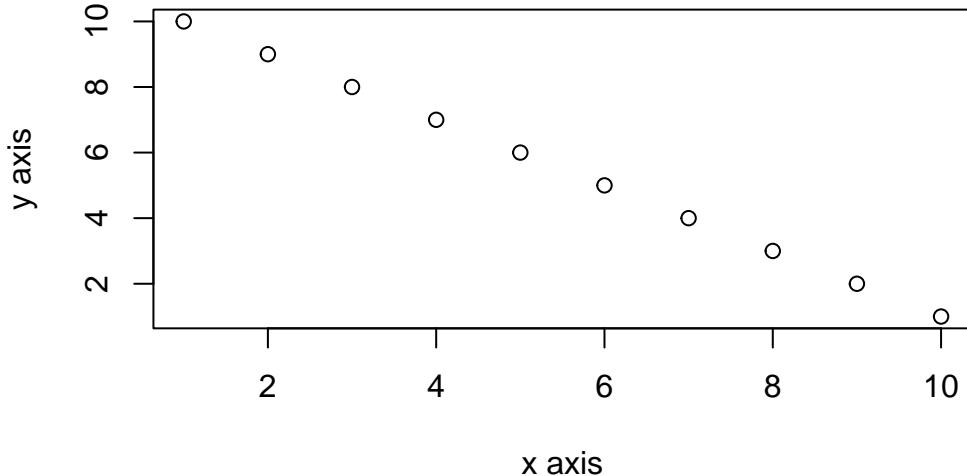


Figure 6.12.: A simple plot

If you want to suppress the figure in the final document use the `fig-show: 'hide'` option.

```
```{r}
#| label: fig-simple-plot5
#| fig-show: hide
x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```

```

If you're using a package like `ggplot2`  to create your plots then don't forget you will need to make the package available with the `library()` function in the code chunk (or in a preceding code chunk).

```
```{r}
#| label: fig-simple-ggplot
#| fig-cap: A simple ggplot

```

```
x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataf <- data.frame(x = x, y = y)

library(ggplot2)
ggplot(dataf, aes(x = x, y = y)) +
 geom_point()
```

```

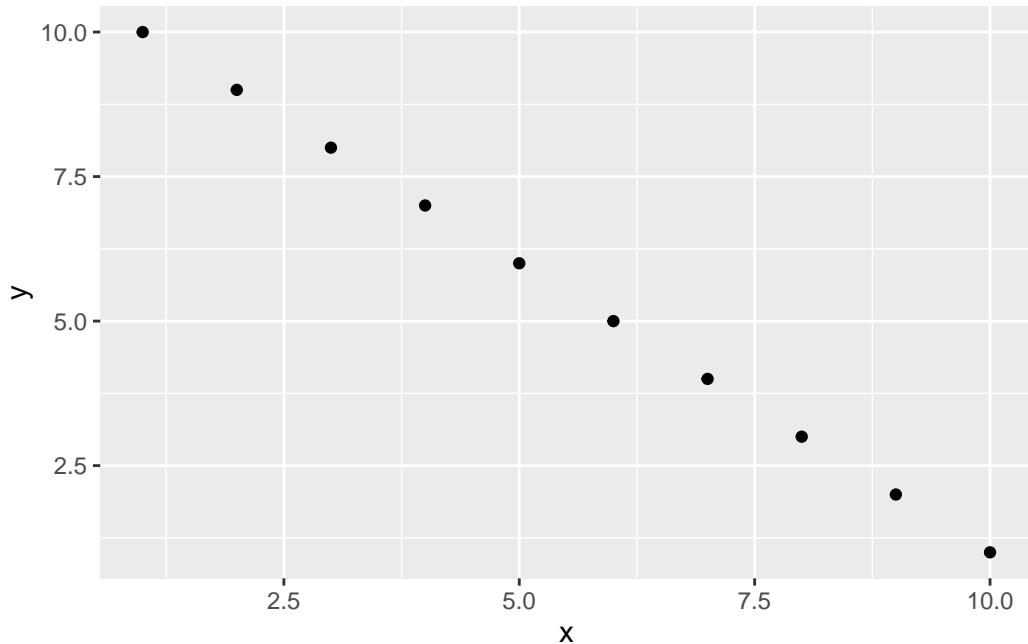


Figure 6.13.: A simple ggplot

Again, there are a large number of chunk options specific to producing plots and figures. See [here](#) for more details.

6.4.5. Tables

In Quarto, you can create tables using native markdown syntax (this doesn't need to be in a code chunk).

| x | y |
|---|----|
| 1 | 10 |

| | |
|---|---|
| 1 | 5 |
| 2 | 4 |
| 3 | 3 |
| 4 | 2 |
| 5 | 1 |

```
: Caption for a simple markdown table
```

Table 6.3.: Caption for a simple markdown table

| x | y |
|---|---|
| 1 | 5 |
| 2 | 4 |
| 3 | 3 |
| 4 | 2 |
| 5 | 1 |

The :-----: lets markdown know that the line above should be treated as a header and the lines below as the body of the table. Alignment within the table is set by the position of the :. To center align use :-----:, to left align :----- and right align -----:. Whilst it can be fun(!) to create tables with raw markup it's only practical for very small and simple tables.

The easiest way we know to include tables in an Quarto document is by using the `kable()` function from the `knitr` package. The `kable()` function can create tables for HTML, PDF and Word outputs.

To create a table of the first 2 rows per species of the `iris` data frame using the `kable()` function simply write

```
library(knitr)
iris %>%
  group_by(Species) %>%
```

```
slice_head(n = 2) %>%
kable()
```

or without loading `knitr`  but indicating where to find the `kable()` function.

```
iris %>%
group_by(Species) %>%
slice_head(n = 2) %>%
knitr::kable()
```

Table 6.4.: A simple kable table

| Sepal.Length | Sepal.Width | Petal.Length | Petal.Width | Species |
|--------------|-------------|--------------|-------------|------------|
| 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| 7.0 | 3.2 | 4.7 | 1.4 | versicolor |
| 6.4 | 3.2 | 4.5 | 1.5 | versicolor |
| 6.3 | 3.3 | 6.0 | 2.5 | virginica |
| 5.8 | 2.7 | 5.1 | 1.9 | virginica |

The `kable()` function offers plenty of options to change the formatting of the table. For example, if we want to round numeric values to one decimal place use the `digits = argument`. To center justify the table contents use `align = 'c'` and to provide custom column headings use the `col.names = argument`. See `?knitr::kable` for more information.

```
iris %>%
group_by(Species) %>%
slice_head(n = 2) %>%
knitr::kable()
```

```

  digits=0,
  align = 'c',
  col.names = c(
    'Sepal length', 'Sepal width',
    'Petal length', 'Petal width', 'Species'
  )
)

```

Table 6.5.: A nicer kable table

| Sepal length | Sepal width | Petal length | Petal width | Species |
|--------------|-------------|--------------|-------------|------------|
| 5 | 4 | 1 | 0 | setosa |
| 5 | 3 | 1 | 0 | setosa |
| 7 | 3 | 5 | 1 | versicolor |
| 6 | 3 | 4 | 2 | versicolor |
| 6 | 3 | 6 | 2 | virginica |
| 6 | 3 | 5 | 2 | virginica |

You can further enhance the look of your `kable` tables using the `kableExtra`  package (don't forget to install the package first!). See [here](#) for more details and a helpful tutorial.

If you want even more control and customisation options for your tables take a look at the `gt`  [package][gt]. `gt` is an acronym for **grammar of tables** and is based on similar principle for tables that are used for plots in `ggplot`.

```

iris %>%
  group_by(Species) %>%
  slice_head(n = 2) %>%
  rename_with(~ gsub("(\\.)", " ", .x)) %>%
  gt()

```

Table 6.6.: A nice gt table

| | Sepal Length | Sepal Width | Petal Length | Petal Width |
|------------|--------------|-------------|--------------|-------------|
| setosa | | | | |
| | 5.1 | 3.5 | 1.4 | 0.2 |
| | 4.9 | 3.0 | 1.4 | 0.2 |
| versicolor | | | | |
| | 7.0 | 3.2 | 4.7 | 1.4 |
| | 6.4 | 3.2 | 4.5 | 1.5 |
| virginica | | | | |
| | 6.3 | 3.3 | 6.0 | 2.5 |
| | 5.8 | 2.7 | 5.1 | 1.9 |

Within most R packages developed to produce tables, there are options to include table captions. However, if you want to add a table caption we recommend to do using the code chunk option in Quarto `tbl-cap`: since it will allow for cross-referencing (Section 6.4.6) and better integration in the document.

```
```{r}
#| label: tbl-gt-table
#| tbl-cap: A nice gt table
#| echo: true

iris %>%
 group_by(Species) %>%
 slice_head(n=2) %>%
 rename_with(~gsub("([._])", " ", .x)) %>%
 gt()
```

```

6.4.6. Cross-referencing

Cross-references make it easier for readers to navigate your document by providing numbered references and hyperlinks to various entities like figures and tables. Once set up, tables and figures numbering happens automatically, so you don't need to re-number all the figures when you add or delete one.

Every cross-referenceable entity requires a label (a unique identifier) prefixed with a cross-reference type e.g. #fig-element

For more details see the [cross-referencing section on Quarto website](#).

6.4.6.1. Document sections

You can make cross-references to other sections of the document. To do so you need to:

1. set up a identifier for the section you want to link to. The identifier should:

- start with #sec-
- be in lower case (Figure 6.3)
- does not have any space, using – instead

2. use the @ symbol and the identifier to refer to the section

```
## Cross-referencing sections {#sec-cross-ref-sections}
```

```
[...]
```

```
As seen before(@sec-cross-ref-sections)
```

6.4.6.2. Images, figures and tables

For tables, images and figures, in addition to the identifier the element also needs a caption for cross-referencing to work.

The prefix for tables is `#tbl-` and `#fig-` for images and figures.

Here is an example for an image included with markdown:

```
! [Rocking the eclipse] (images/markdown/eclipse_ready.jpg){#fig-cute-dog}
```

```
See @fig-cute-dog for an illustration.
```

See Figure 6.14 for an illustration.

For figures and tables produced with R code chunks, simply provide the identifier in the `label` chunk option and the caption also as a chunk option.

Here is the code for a figure and a table.

```
```{r}
#| label: fig-cr-plot
#| fig-cap: A nice figure
x <- 1:10 # create an x variable
y <- 10:1 # create a y variable
dataf <- data.frame(x = x, y = y)

library(ggplot2)
ggplot(dataf, aes(x = x, y = y)) +
 geom_point()
```
```



Figure 6.14.: Rocking the eclipse

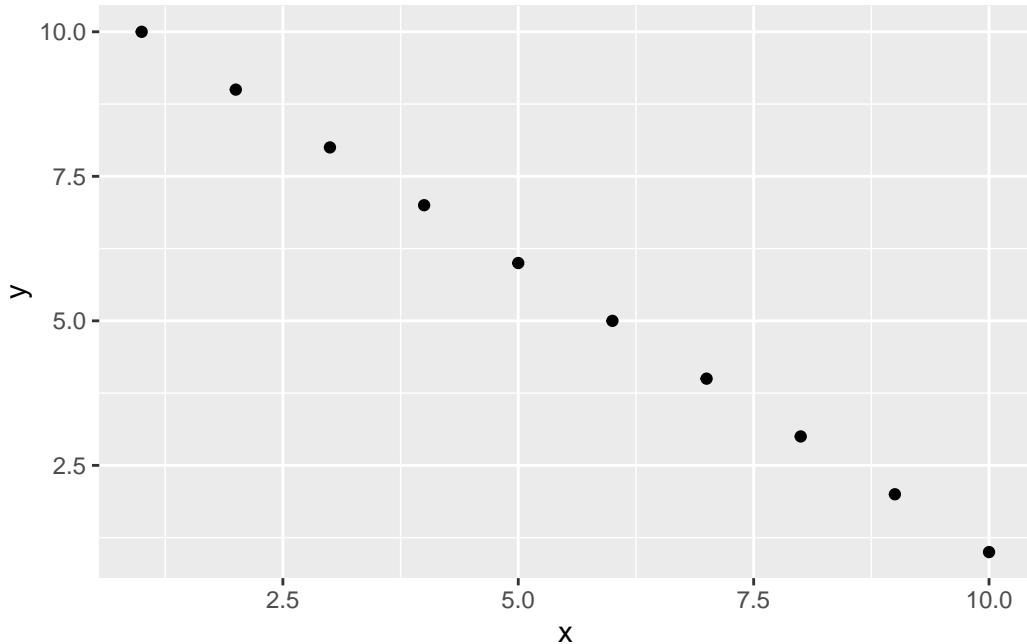


Figure 6.15.: A nice figure

```
```{r}
#| label: tbl-cr-table
#| tbl-cap: A nice table
#| warning: false
library(knitr)
kable(iris[1:5], digits=0, align = 'c', col.names = c('sepal length', 'sepal width', 'petal length',
```

```

Table 6.7.: A nice table

| sepal length | sepal width | petal length | petal width | species |
|--------------|-------------|--------------|-------------|---------|
| 5 | 4 | 1 | 0 | setosa |
| 5 | 3 | 1 | 0 | setosa |
| 5 | 3 | 1 | 0 | setosa |
| 5 | 3 | 2 | 0 | setosa |

Table 6.7.: A nice table

| sepal length | sepal width | petal length | petal width | species |
|--------------|-------------|--------------|-------------|---------|
| 5 | 4 | 1 | 0 | setosa |

Using cross-references, we can write:

As seen on @fig-cr-plot and @tbl-cr-table ...

To get:

As seen on Figure 6.15 and Table 6.7 ...

6.4.7. Citations and bibliography

To generate citations and a bibliography, Quarto requires:

- a properly formatted .qmd document
- a bibliographic source file including all the information for the citations. It works with a wide variatey of format but we suggest using BIBTEX format.
- (optional) a CSL file which specifies the formatting to use when generating the citations and bibliography.

The bibliographic source and the (optional) csl file are specified in the yaml header as :

```
---
title: "My Document"
bibliography: references.bib
csl: ecology.csl
---
```

6.4.7.1. Citations

Quarto uses the standard Pandoc markdown representation for citations (e.g. `[@citation]`) — citations go inside square brackets and are separated by semicolons. Each citation must have a key, composed of ‘@’ + the citation identifier from the database, and may optionally have a prefix, a locator, and a suffix. The citation key must begin with a letter, digit, or `,`, *and may contain alphanumerics*, `,`, and internal punctuation characters.

| Markdown Format | Output (default) |
|---|---|
| Unicorns are the best [see @martin1219, pp. 33-35; also @martin2200, chap. 1] | Unicorns are the best (see Martin 1219 pp. 33–35, also Martin 2200 chap. 1) |
| Unicorns are the best [@martin2200; @martin1219] | Unicorns are the best (Martin 1219, 2200) |
| Martin says unicorns are the best [-@martin2200] | Martin says unicorns are the best (2200) |
| @martin1219 says unicorns are the best. | Martin (1219) says unicorns are the best. |
| @martin1219 [p. 33] says unicorns are the best. | Martin (1219 p. 33) says unicorns are the best. |

6.4.7.2. Create the bibliography

By default, the list of works cited will automatically be generated and placed at the end of document if the style calls for it. It will be placed in a div with the id `refs` if one exists like

```
### Bibliography

::: {#refs}
:::
```

For more details see the [Citation page on Quarto website](#).

6.4.7.3. Integration with Zotero

Quarto integrates really well with [Zotero](#) if you are using the visual editor in either [RStudio](#) or [VS Code](#).

6.5. Some tips and tricks

Problem :

When rendering my Quarto document to pdf my code runs off the edge of the page.

Solution:

Add a global_options argument at the start of your .qmd file in a code chunk:

```
```{r}
#| label: global_options
#| include: false
knitr::opts_chunk$set(message=FALSE, tidy.opts=list(width.cutoff=60), tidy=TRUE)
```
```

This code chunk won't be displayed in the final document due to the `include: false` argument and you should place the code chunk immediately after the YAML header to affect everything below that.

`tidy.opts = list(width.cutoff = 60)`, `tidy=TRUE` defines the margin cutoff point and wraps text to the next line. Play around with this value to get it right (60-80 should be OK for most documents).

With quarto you can also put the global `knitr` options in a `knitrblock` in the YAML header (see [Quarto website](#) for details).

```
---
title: "My Document"
format: html
knitr:
  opts_chunk:
```

```
message: false  
tidy.opts: !expr 'list(width.cutoff=60)'  
tidy: true  
---
```

Problem:

When I load a package in my Quarto document my rendered output contains all of the startup messages and/or warnings.

Solution:

You can load all of your packages at the start of your Quarto document in a code chunk along with setting your global options.

```
```{r}  
#| label: global_options
#| include: false
knitr::opts_chunk$set(
 message = FALSE,
 warning=FALSE,
 tidy.opts=list(width.cutoff=60)
)
suppressPackageStartupMessages(library(ggplot2))
```
```

The `message = FALSE` and `warning = FALSE` arguments suppress messages and warnings. The `suppressPackageStartupMessages` will load the `ggplot2`  package but suppress startup messages.

Problem:

When rendering my Quarto document to pdf my tables and/or figures are split over two pages.

Solution:

Add a page break using the L^AT_EX \pagebreak notation before your offending table or figure

Problem:

The code in my rendered document looks ugly!

Solution:

Add the argument `tidy: true` to your global arguments. Sometimes, however, this can cause problems especially with correct code indentation. The best solution is to write code that looks nice (insert space and use multiple lines)

6.6. Further Information

Although we've covered more than enough to get you quite far using Quarto, as with most things R related, we've really only had time to scratch the surface. Happily, there's a wealth of information available to you should you need to expand your knowledge and experience. A good place to start is the excellent quarto website [here](#).

Another useful and concise Quarto reference guide can be found [here](#)

A quick and easy R Markdown [cheatsheet](#)

6.7. Practical

We will create a new Rmarkdown document and edit it using basic R and Rmarkdown functions.

6.7.1. Context

We will use the awesome `palmerpenguins` dataset  to explore and visualize data.

These data have been collected and shared by [Dr. Kristen Gorman](#) and [Palmer Station, Antarctica LTER](#).

The package was built by Drs Allison Horst and Alison Hill, check out the [official website](#).

The package `palmerpenguins` has two datasets:

- `penguins_raw` has the raw data of penguins observations (see `?penguins_raw` for more info)
- `penguins` is a simplified version of the raw data (see `?penguins` for more info)

For this exercise, we're gonna use the penguins dataset.

```
library(palmerpenguins)  
head(penguins)
```

| species | island | bill_length_mm | bill_depth_mm | flipper_length_mm | body_mass_g | sex | year |
|---------|-----------|----------------|---------------|-------------------|-------------|--------|------|
| Adelie | Torgersen | 39.1 | 18.7 | 181 | 3750 | male | 2007 |
| Adelie | Torgersen | 39.5 | 17.4 | 186 | 3800 | female | 2007 |
| Adelie | Torgersen | 40.3 | 18.0 | 195 | 3250 | female | 2007 |
| Adelie | Torgersen | NA | NA | NA | NA | NA | 2007 |
| Adelie | Torgersen | 36.7 | 19.3 | 193 | 3450 | female | 2007 |
| Adelie | Torgersen | 39.3 | 20.6 | 190 | 3650 | male | 2007 |

6.7.2. Questions

1) Install the package palmerpenguins.

 Solution

```
install.packages("palmerpenguins")
```

2)

- Create a new Quarto document, name it and save it.
- Delete everything after line 12.
- Add a new section title, simple text and text in bold font.
- Compile (“Knit”).

3)

- Add a chunk in which you load the `palmerpenguins`. The corresponding line of code should be hidden in the output.
- Load also the `tidyverse` suite of packages. Modify the defaults to suppress all messages.

Solution

```
```{r}
#| echo: false
#| message:false
library(palmerpenguins)
library(tidyverse)
```
```

4) Add another chunk in which you build a table with the 10 first rows of the dataset.

Solution

```
```{r}
penguins %>%
 slice(1:10) %>%
 knitr::kable()
```
```

5) In a new section, display how many individuals, penguins species and islands we have in the dataset. This info should appear directly in the text, you need to use inline code 😊. Calculate the mean of the (numeric) traits measured on the penguins.

 Solution

```
## Numerical exploration
```

There are `r nrow(penguins)` penguins in the dataset,
and `r length(unique(penguins\$species))` different species.
The data were collected in `r length(unique(penguins\$island))`
islands of the Palmer archipelago in Antarctica.

The mean of all traits that were measured on the penguins are:

```
```{r}
#| echo: false
penguins %>%
 group_by(species) %>%
 summarize(across(where(is.numeric), mean, na.rm = TRUE))
````
```

- 6) In another section, entitled ‘Graphical exploration’, build a figure with 3 superimposed histograms, each one corresponding to the body mass of a species.

 Solution

```
## Graphical exploration
```

A histogram of body mass per species:

```
```{r}
#| fig-cap: Distribution of body mass by species of penguins
ggplot(data = penguins) +
 aes(x = body_mass_g) +
 geom_histogram(aes(fill = species),
 alpha = 0.5,
 position = "identity") +
 scale_fill_manual(values = c("darkorange", "purple", "cyan4")) +
 theme_minimal() +
 labs(x = "Body mass (g)",
 y = "Frequency",
 title = "Penguin body mass")
```

```

7) In another section, entitled *Linear regression*, fit a model of bill length as a function of body size (flipper length), body mass and sex. Obtain the output and graphically evaluate the assumptions of the model. As reminder here is how you fit a linear regression.

```
```{r}
model <- lm(Y ~ X1 + X2, data = data)
summary(model)
plot(model)
```

```

Solution

```
## Linear regression
```

And here is a nice model with graphical output

```
```{r}
#| fig-cap: "Checking assumptions of the model"
m1 <- lm(bill_length_mm ~ flipper_length_mm + body_mass_g + sex, data = penguins)
summary(m1)
par(mfrow= c(2,2))
plot(m1)
```
```

8) Add references manually or using `citr` in RStudio.

1. Pick a recent publication from the researcher who shared the data, Dr Kristen Gorman. Import this publication in your favorite references manager (we use Zotero, no hard feeling), and create a bibtex reference that you will add to the file `mabiblio.bib`.
2. Add bibliography: `mabiblio.bib` at the beginning of your R Markdown document (YAML).
3. Cite the reference in the text using either typing the reference manually or using `citr`. To use `citr`, install it first; if everything goes well, you should see it in the pulldown menu `Addins` . Then simply use `Insert citations` in the pull-down menu `Addins`.
4. Compile.

9) Change the default citation format (Chicago style) into the The American Naturalist format. It can be found here <https://www.zotero.org/styles>. To do so, add `csl: the-american-naturalist.csl` in the YAML.

10) Build your report in html, pdf and docx format. 

Example of output

You can see an example of the [Rmarkdown source file](#) and [pdf output](#)

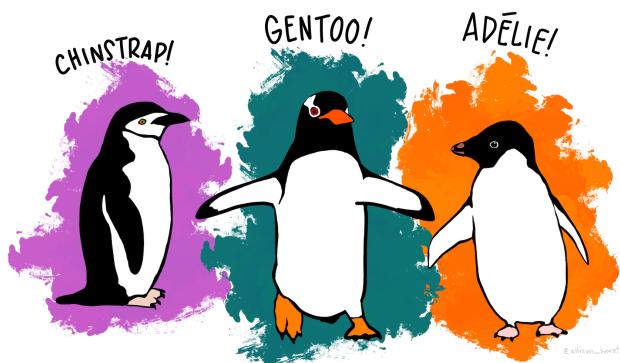


Figure 6.16.: Happy coding

Chapter 7

Version control with Git and GitHub

This Chapter will introduce you to the basics of using a version control system to keep track of all your important R code and facilitate collaboration with colleagues and the wider world. This Chapter will focus on using the software ‘Git’ in combination with the web-based hosting service ‘GitHub’. By the end of the Chapter, you will be able to install and configure Git and GitHub on your computer and setup and work with a version controlled project in RStudio. We won’t be covering more advanced topics such as branching, forking and pull requests in much detail but we do give an overview later in Section 7.8.

Just a few notes of caution. In this Chapter we’ll be using RStudio to interface with Git as it gives you a nice friendly graphical user interface which generally makes life a little bit easier (and who doesn’t want that?). However, one downside to using RStudio with Git is that RStudio only provides pretty basic Git functionality through its menu system. That’s fine for most of what we’ll be doing during this Chapter (although we will introduce a few Git commands as we go along) but if you really want to benefit from using Git’s power you will need to learn some Git commands (see Section 7.10) and syntax and change for another IDE such as [VSCode](#) which is much much better when it comes to integration with github. Git can become a little bewildering and frustrating when you first start using it. This is mostly due to the terminology and liberal use of jargon associated with Git, but there’s no hiding the fact that it’s quite easy to get yourself and your Git repository into a pickle. Therefore, we’ve tried hard to keep things as straight forward as we can during this Chapter and as a result we do occasionally show you a couple of very ‘un-Git’ ways of doing things (mostly about reverting to previous versions of documents). Don’t get hung up about this, there’s no shame to using these low tech solutions and if it works then it works. Lastly, GitHub was not designed to host very large files and will warn you if you try to add files greater than 50 MB and block you adding files greater than 100 MB. If your project involves using large file sizes there are a [few solutions](#) but we find the easiest solution is to host these files elsewhere (Googledrive, Dropbox etc) and create a link to them in a README file or R markdown document on Github.

7.1. What is version control?

A [Version Control System](#) (VCS) keeps a record of all the changes you make to your files that make up a particular project and allows you to revert to previous versions of files if you need to. To put it another way, if you muck things up or accidentally lose important files you can easily roll back to a previous stage in your project to sort things out. Version control was originally designed for collaborative software development, but it's equally useful for scientific research and collaborations (although admittedly a lot of the terms, jargon and functionality are focused on the software development side). There are many different version control systems currently available, but we'll focus on using *Git*, because it's free and open source and it integrates nicely with RStudio. This means that its can easily become part of your usual workflow with minimal additional overhead.

7.2. Why use version control?

So why should you worry about version control? Well, first of all it helps avoid this (familiar?) situation when you're working on a project usually arising from this (familiar?) scenario.

Version control automatically takes care of keeping a record of all the versions of a particular file and allows you to revert back to previous versions if you need to. Version control also helps you (especially the future you) keep track of all your files in a single place and it helps others (especially collaborators) review, contribute to and reuse your work through the GitHub website. Lastly, your files are always available from anywhere and on any computer, all you need is an internet connection.

7.3. What is Git and GitHub?

Git is a version control system originally developed by [Linus Torvalds](#) that lets you track changes to a set of files. These files can be any type of file including the menagerie of files that typically make up a data orientated project (.pdf, .Rmd, .docx, .txt, .jpg etc) although plain text files work the best. All the files that make up a project is called a **repository** (or just **repo**).

GitHub is a web-based hosting service for Git repositories which allows you to create a remote copy of your local version-controlled project. This can be used as a backup or archive of your project or make it accessible to you and to your colleagues so you can work collaboratively.

At the start of a project we typically (but not always) create a **remote** repository on GitHub, then **clone** (think of this as copying) this repository to our **local** computer (the one in front of you). This cloning is usually a one time

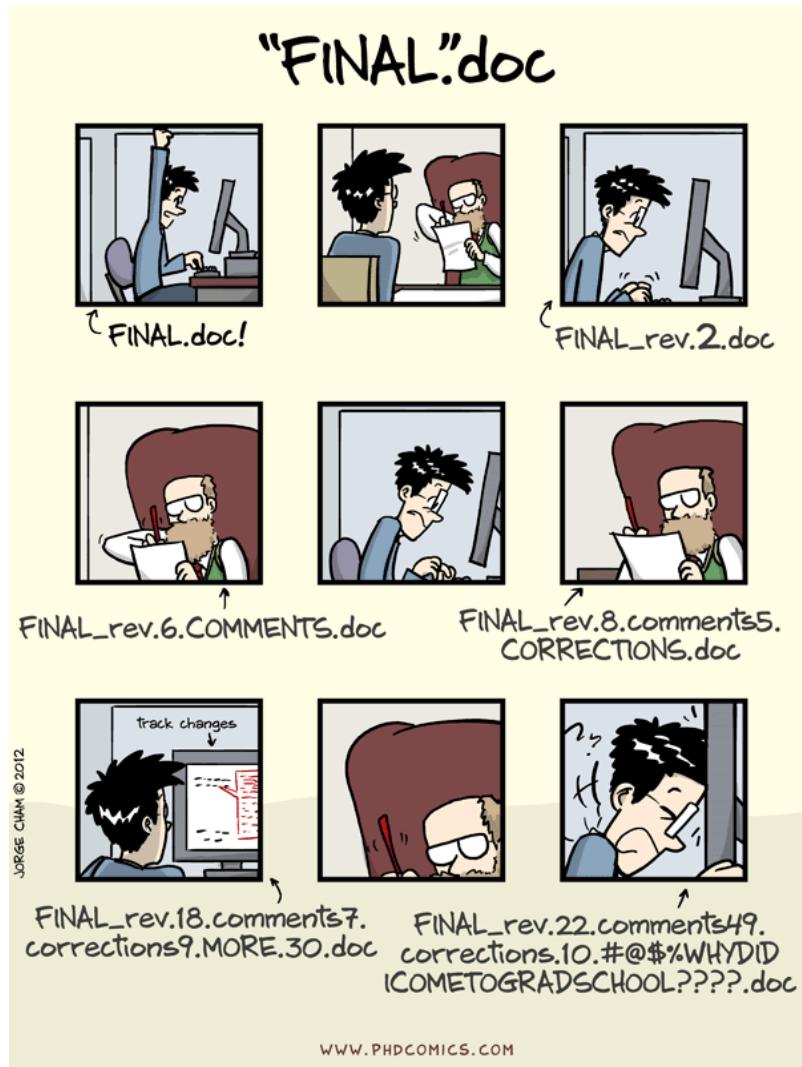


Figure 7.1.: Why you need version control (source: [PhDComics](http://www.phdcomics.com))

event and you shouldn't need to clone this repository again unless you really muck things up. Once you have cloned your repository you can then work locally on your project as usual, creating and saving files for your data analysis (scripts, R markdown documents, figures etc). Along the way you can take snapshots (called **commits**) of these files after you've made important changes. We can then **push** these changes to the remote GitHub repository to make a backup or make available to our collaborators. If other people are working on the same project (**repository**), or maybe you're working on a different computer, you can **pull** any changes back to your local repository so everything is synchronised.

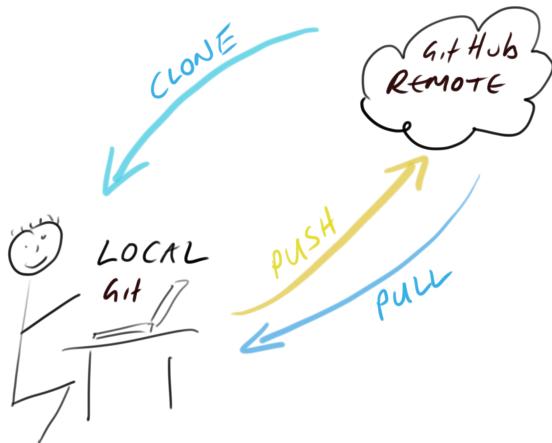


Figure 7.2.: How git works

7.4. Getting started

This Chapter assumes that you have already installed the latest versions of R and an IDE (RStudio or VSCode). If you haven't done this yet you can find instructions in Section 1.1.1.

7.4.1. Install Git

To get started, you first need to install Git. If you're lucky you may already have Git installed (especially if you have a Mac or Linux computer). You can check if you already have Git installed by clicking on the Terminal tab in RStudio and typing `git --version`. If you see something that looks like `git version 2.25.0` (the version number may be different on your computer) then you already have Git installed (happy days). If you get an error (something like `git: command not found`) this means you don't have Git installed (yet!).

You can also do this check outside RStudio by opening up a separate Terminal if you want. On Windows go to the ‘Start menu’ and in the search bar (or run box) type cmd and press enter. On a Mac go to ‘Applications’ in Finder, click on the ‘Utilities’ folder and then on the ‘Terminal’ program. On a Linux machine simply open the Terminal (Ctrl+Alt+T often does it).

To install Git on a **Windows** computer we recommend you download and install Git for Windows (also known as ‘Git Bash’). You can find the download file and installation instructions [here](#).

For those of you using a **Mac** computer we recommend you download Git from [here](#) and install in the usual way (double click on the installer package once downloaded). If you’ve previously installed Xcode on your Mac and want to use a more up to date version of Git then you will need to follow a few more steps documented [here](#). If you’ve never heard of Xcode then don’t worry about it!

For those of you lucky enough to be working on a **Linux** machine you can simply use your OS package manager to install Git from the official repository. For Ubuntu Linux (or variants of) open your Terminal and type

```
sudo apt update  
sudo apt install git
```

You will need administrative privileges to do this. For other versions of Linux see [here](#) for further installation instructions.

Whatever version of Git you’re installing, once the installation has finished verify that the installation process has been successful by running the command git --version in the Terminal tab in RStudio (as described above). On some installations of Git (yes we’re looking at you MS Windows) this may still produce an error as you will also need to setup RStudio so it can find the Git executable (described in Section 7.4.3).

7.4.2. Configure Git

After installing Git, you need to configure it so you can use it. Click on the Terminal tab in the Console window again and type the following:

```
git config --global user.email 'you@youremail.com'  
  
git config --global user.name 'Your Name'
```

substituting 'Your Name' for your actual name and 'you@youremail.com' with your email address. We recommend you use your University email address (if you have one) as you will also use this address when you register for your GitHub account (coming up in a bit).

If this was successful, you should see no error messages from these commands. To verify that you have successfully configured Git type the following into the Terminal

```
git config --global --list
```

You should see both your `user.name` and `user.email` configured.

7.4.3. Configure RStudio

As you can see above, Git can be used from the command line, but it also integrates well with RStudio, providing a friendly graphical user interface. If you want to use RStudio's Git integration (we recommend you do - at least at the start), you need to check that the path to the Git executable is specified correctly. In RStudio, go to the menu `Tools -> Global Options -> Git/SVN` and make sure that 'Enable version control interface for RStudio projects' is ticked and that the 'Git executable:' path is correct for your installation. If it's not correct hit the `Browse...` button and navigate to where you installed git and click on the executable file. You will need to restart RStudio after doing this.

7.4.4. Configure VSCode

to develop

7.4.5. Register a GitHub account

If all you want to do is to keep track of files and file versions on your local computer then Git is sufficient. If however, you would like to make an off-site copy of your project or make it available to your collaborators then you will need a web-based hosting service for your Git repositories. This is where GitHub comes into play (there are also other services like [GitLab](#), [Bitbucket](#) and [Savannah](#)). You can sign up for a free account on GitHub [here](#). You will need to specify a username, an email address and a strong password. We suggest that you use your University email address (if you have one) as this will also allow you to apply for a free [educator or researcher account](#) later on which gives you some useful [benefits](#) (don't worry about this now though). When it comes to choosing a username we suggest you give this some thought. Choose a short(ish) rather than a long username, use all lowercase and hyphenate if you

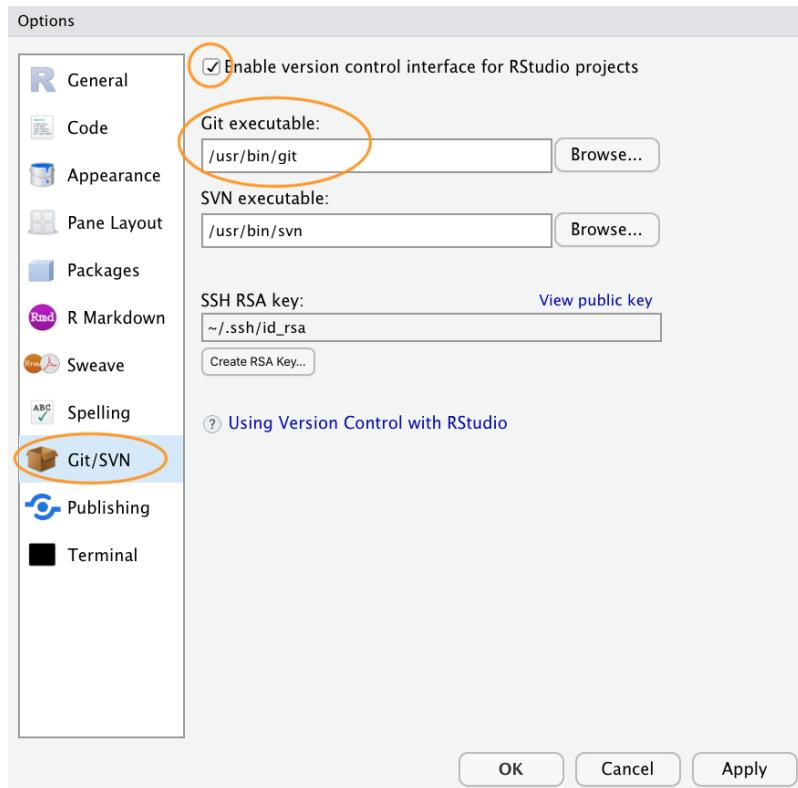


Figure 7.3.: Providing path to git software in RStudio

want to include multiple words, find a way of incorporating your actual name and lastly, choose a username that you will feel comfortable revealing to your future employer!

Next click on the ‘Select a plan’ (you may have to solve a simple puzzle first to verify you’re human) and choose the ‘Free Plan’ option. Github will send an email to the email address you supplied for you to verify.

Once you’ve completed all those steps you should have both Git and GitHub setup up ready for you to use (Finally!).

7.5. Setting up a project

7.5.1. in RStudio

Now that you’re all set up, let’s create your first version controlled RStudio project. There are a couple of different approaches you can use to do this. You can either setup a remote GitHub repository first then connect an RStudio project to this repository (we’ll call this Option 1). Another option is to setup a local repository first and then link a remote GitHub repository to this repository (Option 2). You can also connect an existing project to a GitHub repository but we won’t cover this here. We suggest that if you’re completely new to Git and GitHub then use Option 1 as this approach sets up your local Git repository nicely and you can **push** and **pull** immediately. Option 2 requires

a little more work and therefore there are more opportunities to go wrong. We will cover both of these options below.

7.5.2. Option 1 - GitHub first

To use the GitHub first approach you will first need to create a **repository (repo)** on GitHub. Go to your [GitHub page](#) and sign in if necessary. Click on the ‘Repositories’ tab at the top and then on the green ‘New’ button on the right

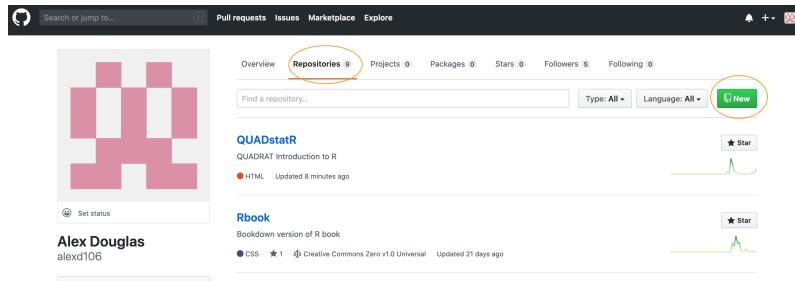


Figure 7.4.: Creating a new repository on Github

Give your new repo a name (let’s call it `first_repo` for this Chapter), select ‘Public’, tick on the ‘Initialize this repository with a README’ (this is important) and then click on ‘Create repository’ (ignore the other options for now).

 A screenshot of the 'Create a new repository' form.
 - **Owner:** Set to 'alex106'.
 - **Repository name:** 'first_repo' is entered in the input field, which is highlighted with a red oval.
 - **Description (optional):** An empty text area.
 - **Visibility:** 'Public' is selected (radio button highlighted with a red oval), while 'Private' is unselected.
 - **Initialization:** 'Initialize this repository with a README' is checked (checkbox highlighted with a red oval).
 - **Buttons:** 'Create repository' is a large green button at the bottom, highlighted with a red oval.
 - **Other Options:** Buttons for '.gitignore' (None) and 'Add a license' (None) are shown at the bottom left.

Figure 7.5.: Configuring a new repository on Github

Your new GitHub repository will now be created. Notice the README has been rendered in GitHub and is in

markdown (.md) format (see Chapter 6 on R markdown if this doesn't mean anything to you). Next click on the green 'Clone or Download' button and copy the [https://... URL](https://github.com/alex106/first_repo) that pops up for later (either highlight it all and copy or click on the copy to clipboard icon to the right).

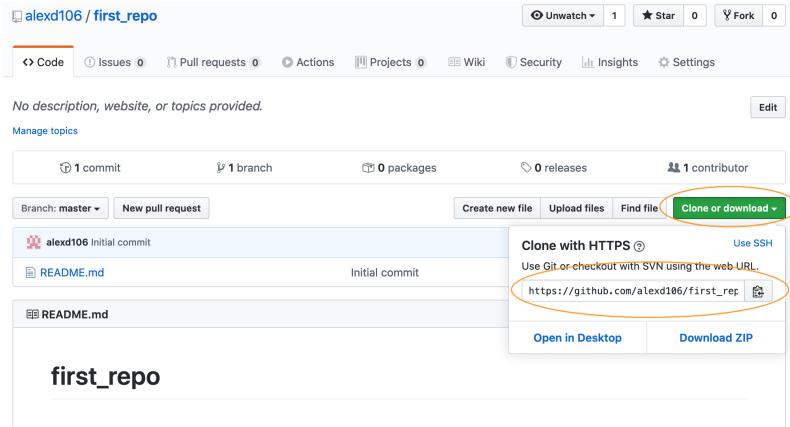


Figure 7.6.: Getting the cloning path for a directory on github

Ok, we now switch our attention to RStudio. In RStudio click on the **File -> New Project** menu. In the pop up window select **Version Control**.

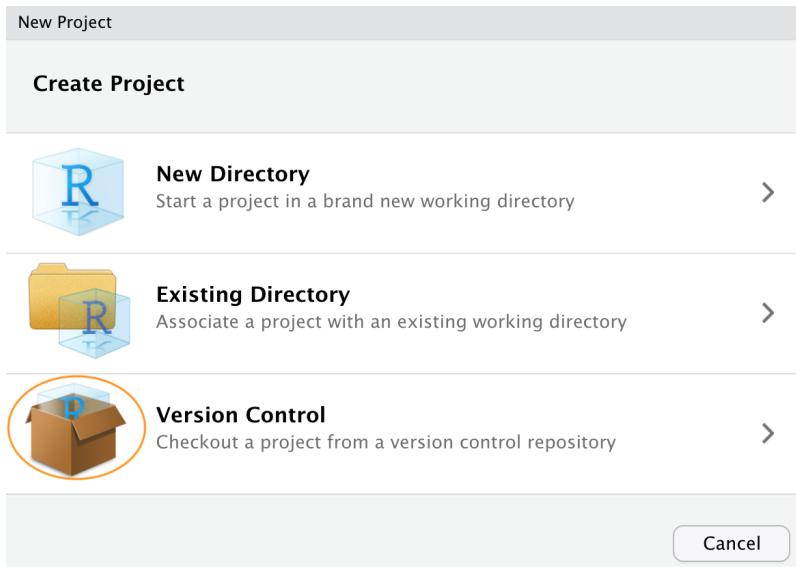


Figure 7.7.: Setting a new Github project in RStudio

Now paste the the URL you previously copied from GitHub into the **Repository URL:** box. This should then automatically fill out the **Project Directory Name:** section with the correct repository name (it's important that the name of this directory has the same name as the repository you created in GitHub). You can then select where you want to create this directory by clicking on the **Browse** button opposite the **Create project as a subdirectory of:** option. Navigate to where you want the directory created and click **OK**. We also tick the **Open**

in new session option.

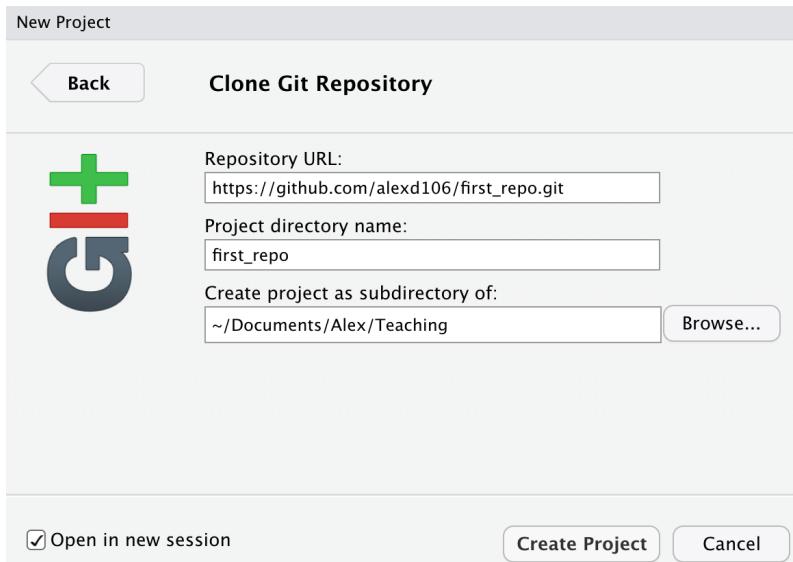


Figure 7.8.

RStudio will now create a new directory with the same name as your repository on your local computer and will then **clone** your remote repository to this directory. The directory will contain three new files; `first_repo.Rproj` (or whatever you called your repository), `README.md` and `.gitignore`. You can check this out in the `Files` tab usually in the bottom right pane in RStudio. You will also have a `Git` tab in the top right pane with two files listed (we will come to this later on in the Chapter). That's it for Option 1, you now have a remote GitHub repository set up and this is linked to your local repository managed by RStudio. Any changes you make to files in this directory will be version controlled by Git.

7.5.3. Option 2 - RStudio first

An alternative approach is to create a local RStudio project first and then link to a remote Github repository. As we mentioned before, this option is more involved than Option 1 so feel free to skip this now and come back later to it if you're interested. This option is also useful if you just want to create a local RStudio project linked to a local Git repository (i.e. no GitHub involved). If you want to do this then just follow the instructions below omitting the GitHub bit.

In RStudio click on the `File -> New Project` menu and select the `New Directory` option.

In the pop up window select the `New Project` option

In the New Project window specify a `Directory name` (choose `second_repo` for this Chapter) and select where you would like to create this directory on you computer (click the `Browse` button). Make sure the `Create a git`

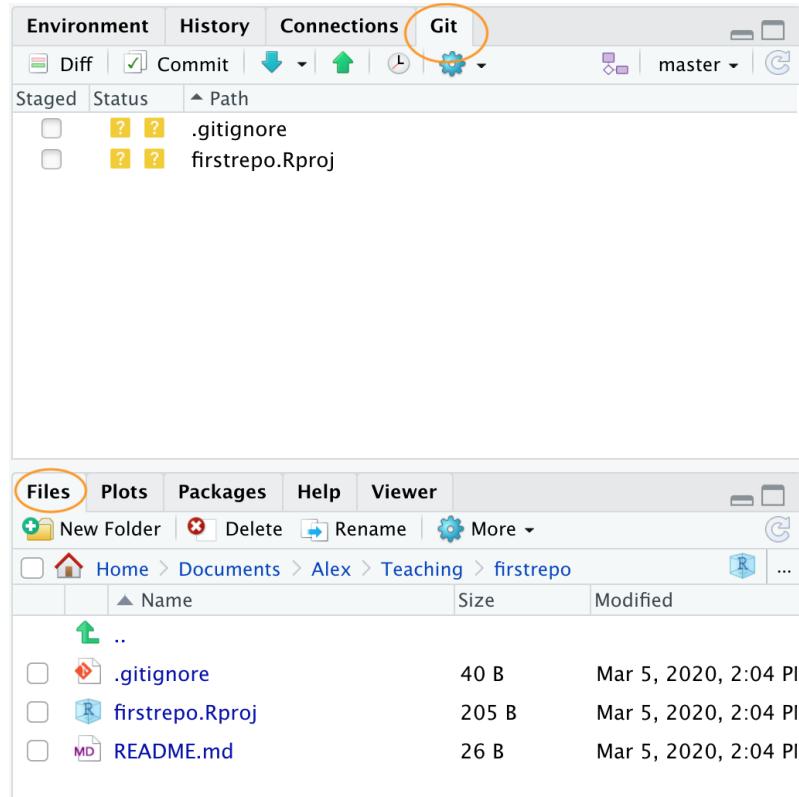


Figure 7.9.

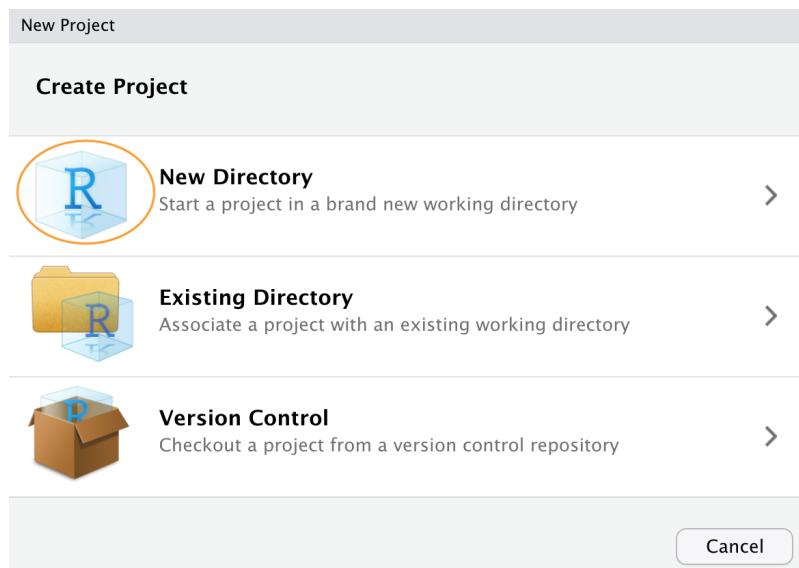


Figure 7.10.

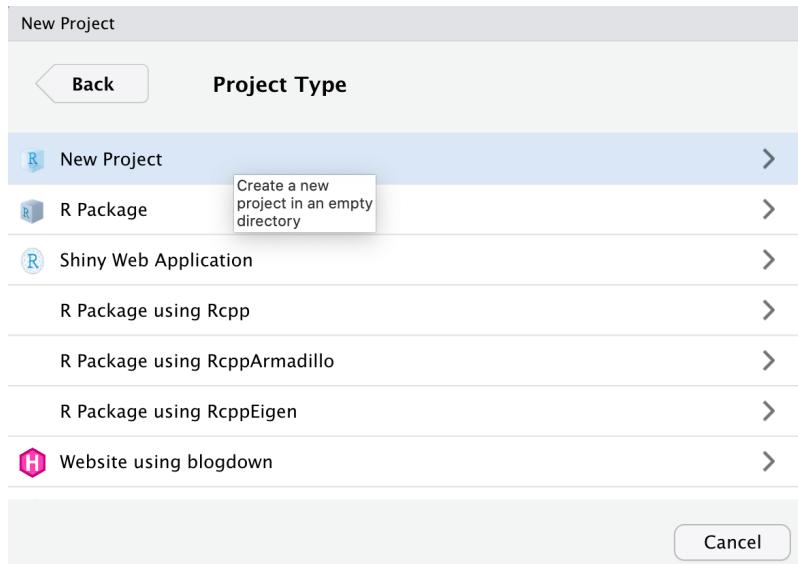


Figure 7.11.

repository option is ticked

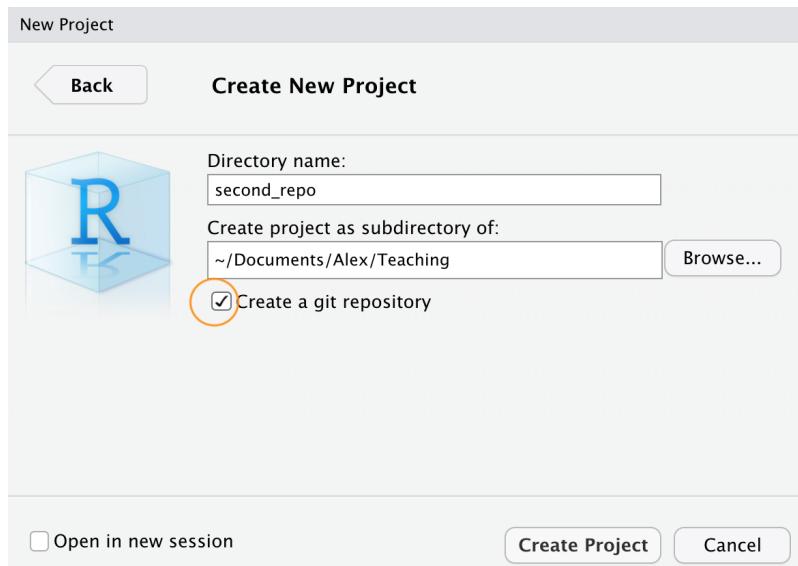


Figure 7.12.

This will create a version controlled directory called `second_repo` on your computer that contains two files, `second_repo.Rproj` and `.gitignore` (there might also be a `.Rhistory` file but ignore this). You can check this by looking in the Files tab in RStudio (usually in the bottom right pane).

OK, before we go on to create a repository on GitHub we need to do one more thing - we need to place our `second_repo.Rproj` and `.gitignore` files under version control. Unfortunately we haven't covered this in detail yet so just follow the next few instructions (blindly!) and we'll revisit them in Section 7.6 of this Chapter.

To get our two files under version control click on the 'Git' tab which is usually in the top right pane in RStudio

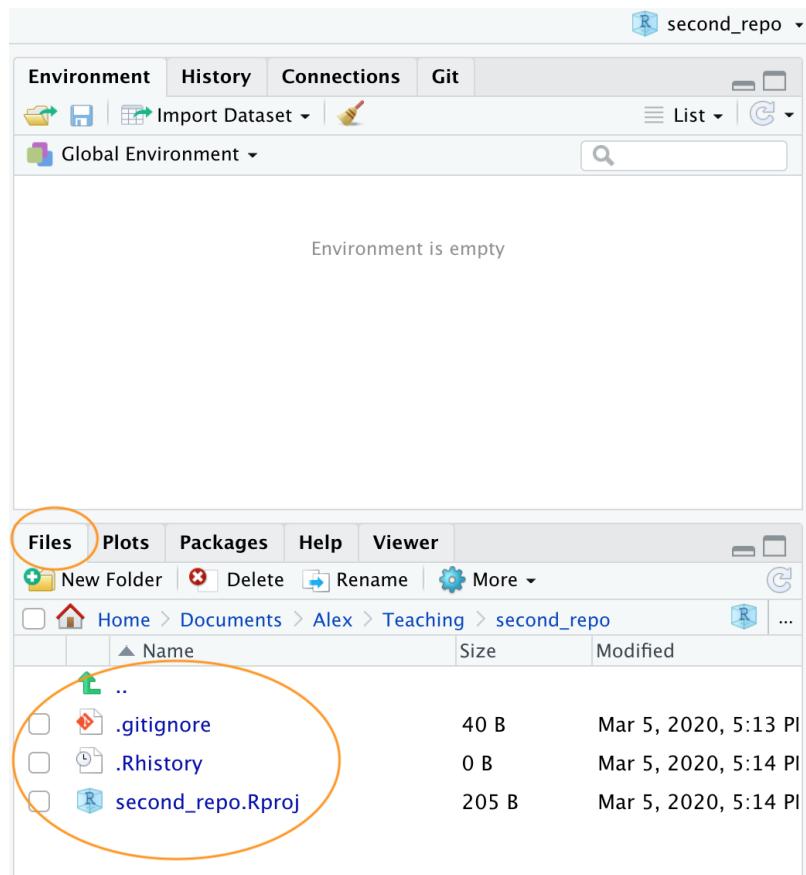


Figure 7.13.

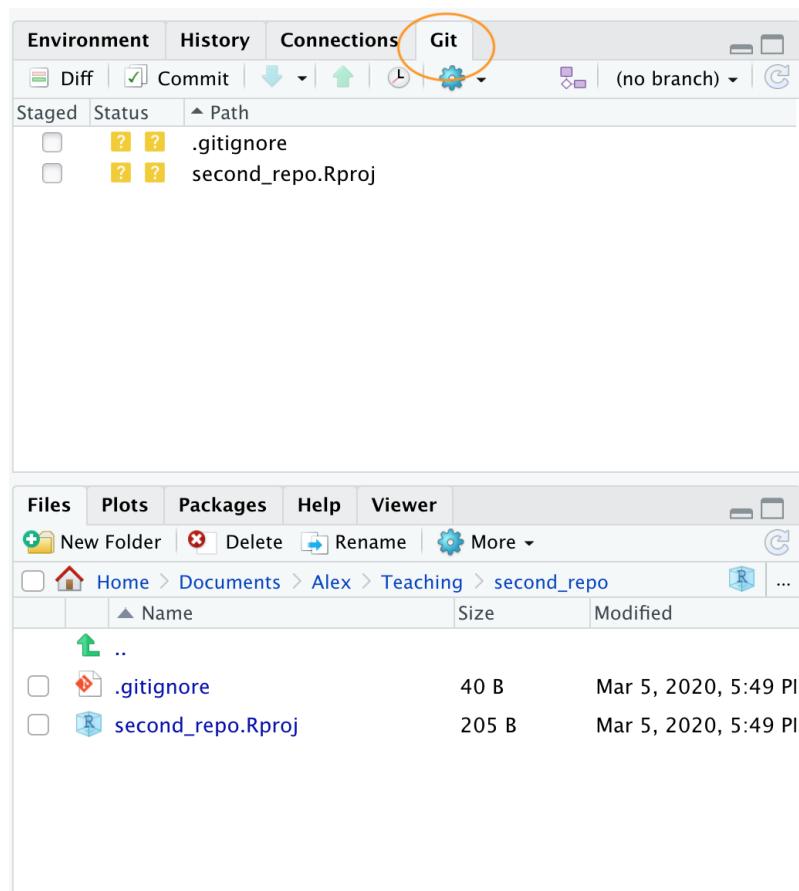


Figure 7.14.

You can see that both files are listed. Next, tick the boxes under the ‘Staged’ column for both files and then click on the ‘Commit’ button.

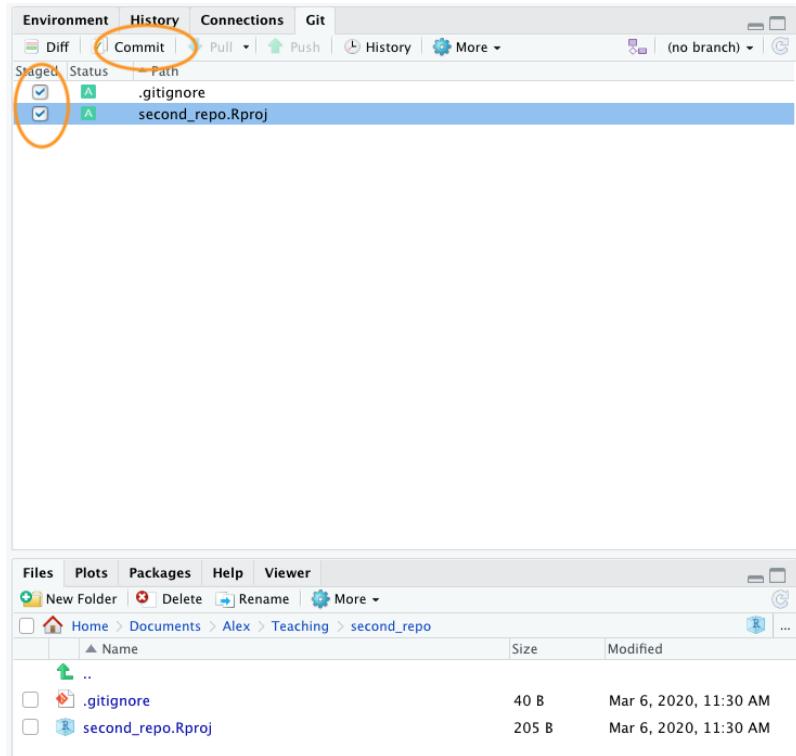


Figure 7.15.

This will take you to the ‘Review Changes’ window. Type in the commit message ‘First commit’ in the ‘Commit message’ window and click on the ‘Commit’ button. A new window will appear with some messages which you can ignore for now. Click ‘Close’ to close this window and also close the ‘Review Changes’ window. The two files should now have disappeared from the Git pane in RStudio indicating a successful commit.

OK, that’s those two files now under version control. Now we need to create a new repository on GitHub. In your browser go to your [GitHub page](#) and sign in if necessary. Click on the ‘Repositories’ tab and then click on the green ‘New’ button on the right. Give your new repo the name `second_repo` (the same as your version controlled directory name) and select ‘Public’. This time **do not** tick the ‘Initialize this repository with a README’ (this is important) and then click on ‘Create repository’.

This will take you to a Quick setup page which provides you with some code for various situations. The code we are interested in is the code under `...or push an existing repository from the command line` heading.

Highlight and copy the first line of code (note: yours will be slightly different as it will include your GitHub username not mine)

```
git remote add origin https://github.com/alexd106/second_repo.git
```

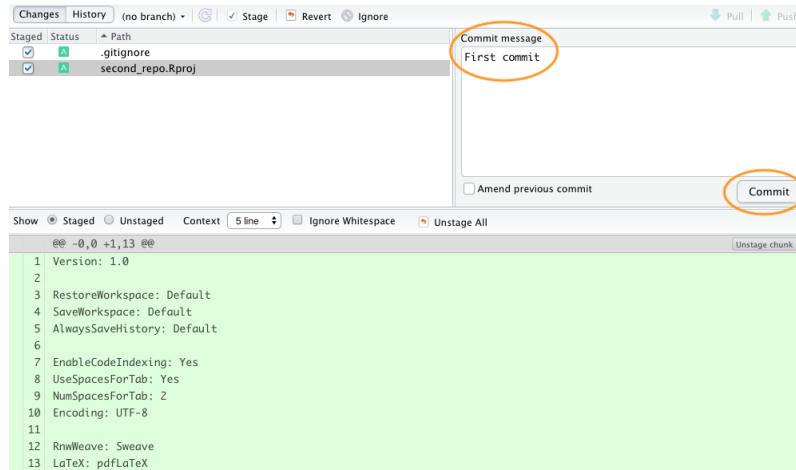


Figure 7.16.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

| | |
|----------|--|
| Owner | Repository name * |
| alexd106 | / <input type="text" value="second_repo"/> |

Great repository names are short and memorable. Need inspiration? How about [fictional-octo-eureka](#)?

Description (optional)

Public
Anyone can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

Initialize this repository with a README
This will let you immediately clone the repository to your computer.

Figure 7.17.

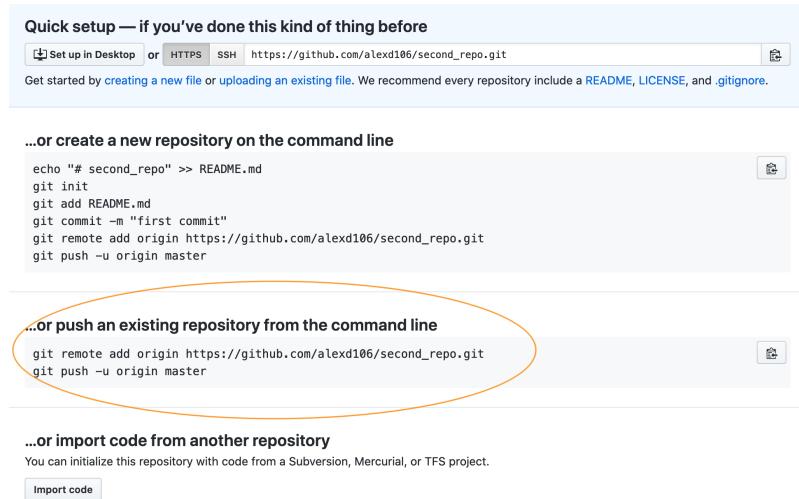


Figure 7.18.

Switch to RStudio, click on the ‘Terminal’ tab and paste the command into the Terminal. Now go back to GitHub and copy the second line of code

```
git push -u origin master
```

and paste this into the Terminal in RStudio. You should see something like this

The image shows a screenshot of RStudio's Terminal tab. The terminal window is titled 'Terminal 1' and shows the path '~ /Documents/Alex/Teaching/second_repo'. The output of the command 'git push -u origin master' is displayed, showing the progress of the push operation and the successful creation of a new branch 'master' on GitHub.

```

Console Terminal x Jobs x
Terminal 1 ~ /Documents/Alex/Teaching/second_repo
md-051770:second_repo nhy163$ git remote add origin https://github.com/alexd106/second_repo.git
md-051770:second_repo nhy163$ git push -u origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 436 bytes | 218.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To https://github.com/alexd106/second_repo.git
 * [new branch] master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
md-051770:second_repo nhy163$
```

Figure 7.19.

If you take a look at your repo back on GitHub (click on the `/second_repo` link at the top) you will see the `second_repo.Rproj` and `.gitignore` files have now been **pushed** to GitHub from your local repository.

The last thing we need to do is create and add a **README** file to your repository. A README file describes your project and is written using the same Markdown language you learned in Chapter 6. A good README file makes it easy for others (or the future you!) to use your code and reproduce your project. You can create a README file in RStudio or in GitHub. Let’s use the second option.

In your repository on GitHub click on the green Add a README button.

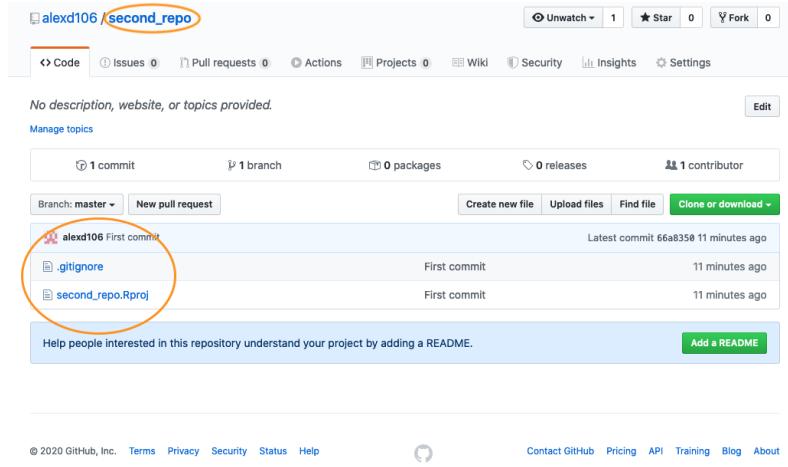


Figure 7.20.

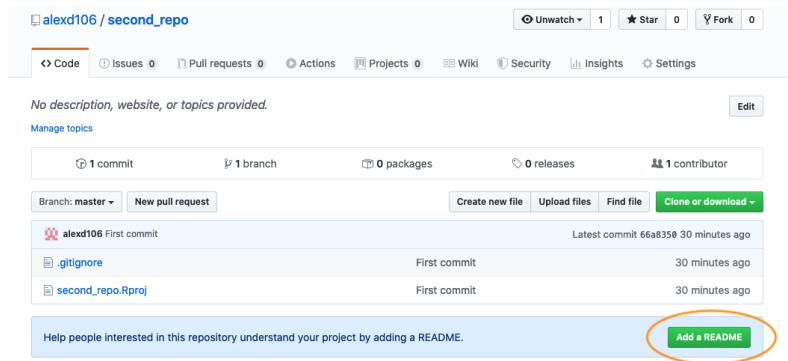


Figure 7.21.

Now write a short description of your project in the <> `Edit new file` section and then click on the green `Commit new file` button.

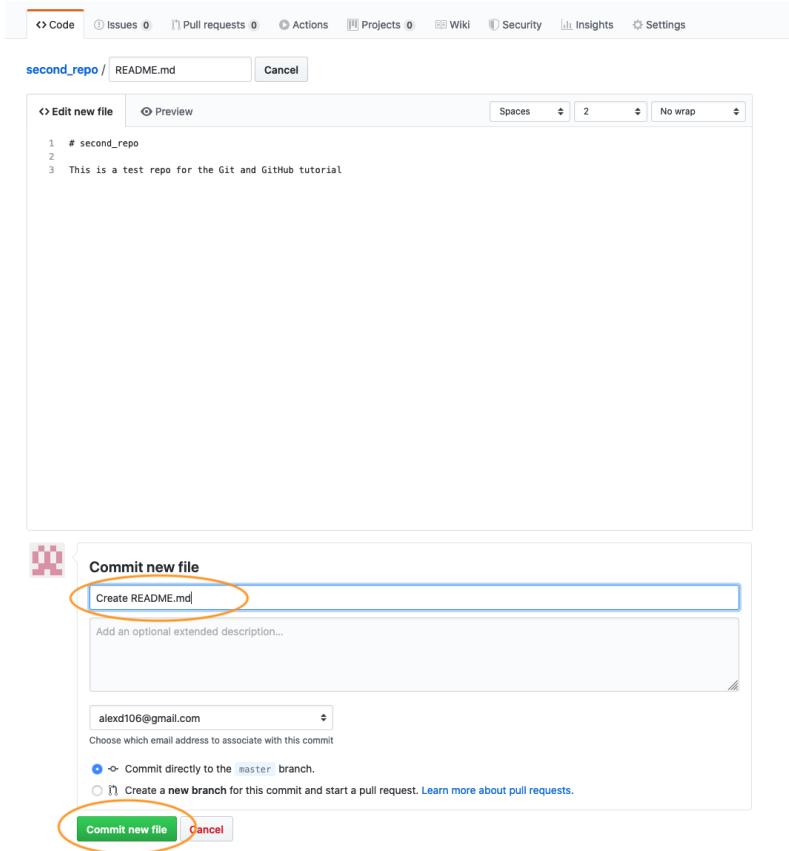


Figure 7.22.

You should now see the `README.md` file listed in your repository. It won't actually exist on your computer yet as you will need to **pull** these changes back to your local repository, but more about that in the next section.

Whether you followed Option 1 or Option 2 (or both) you have now successfully setup a version controlled RStudio project (and associated directory) and linked this to a GitHub repository. Git will now monitor this directory for any changes you make to files and also if you add or delete files. If the steps above seem like a bit of an ordeal, just remember, you only need to do this once for each project and it gets much easier over time.

7.5.4. in VSCode

to develop

7.6. Using Git with RStudio

Now that we have our project and repositories (both local and remote) set up, it's finally time to learn how to use Git in your IDE!

Typically, when using Git your workflow will go something like this:

1. You create/delete and edit files in your project directory on your computer as usual (saving these changes as you go)
2. Once you've reached a natural 'break point' in your progress (i.e. you'd be sad if you lost this progress) you **stage** these files
3. You then **commit** the changes you made to these staged files (along with a useful commit message) which creates a permanent snapshot of these changes
4. You keep on with this cycle until you get to a point when you would like to **push** these changes to GitHub
5. If you're working with other people on the same project you may also need to **pull** their changes to your local computer

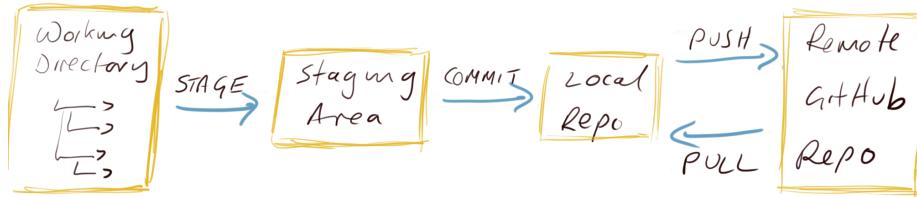


Figure 7.23.

OK, let's go through an example to help clarify this workflow.

Open up the `first_repo.Rproj` you created previously during Option 1. Either use the `File -> Open Project` menu or click on the top right project icon and select the appropriate project.

Create an R markdown document inside this project by clicking on the `File -> New File -> R markdown` menu (remember Chapter 6?).

Once created, we can delete all the example R markdown code (except the YAML header) as usual and write some interesting R markdown text and include a plot. We'll use the inbuilt `cars` dataset to do this. Save this file (cmd + s

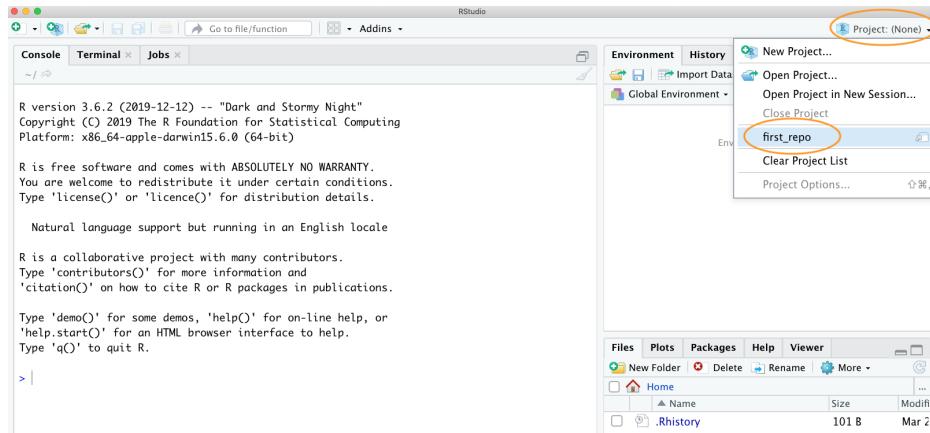


Figure 7.24.

for Mac or **ctrl + s** in Windows). Your R markdown document should look something like the following (it doesn't matter if it's not exactly the same).

```

1 <!--
2   title: "First version control project"
3   author: "Alex Douglas"
4   date: "05/03/2020"
5   output: html_document
6 ---
7
8 ````{r setup, include=FALSE}
9 knitr::opts_chunk$set(echo = TRUE)
10 ```
11
12 My *first* version controlled project in RStudio.
13
14 Let's create a test plot
15
16 ````{r, test-plot}
17 plot(cars)
18 ```
19

```

Figure 7.25.

Take a look at the ‘Git’ tab which should list your new R markdown document (`first_doc.Rmd` in this example) along with `first_repo.Rproj`, and `.gitignore` (you created these files previously when following Option 1).

Following our workflow, we now need to **stage** these files. To do this tick the boxes under the ‘Staged’ column for all files. Notice that there is a status icon next to the box which gives you an indication of how the files were changed. In our case all of the files are to be added (capital A) as we have just created them.

After you have staged the files the next step is to **commit** the files. This is done by clicking on the ‘Commit’ button.

After clicking on the ‘Commit’ button you will be taken to the ‘Review Changes’ window. You should see the three files you staged from the previous step in the left pane. If you click on the file name `first_doc.Rmd` you will see the changes you have made to this file highlighted in the bottom pane. Any content that you have added is highlighted in green and deleted content is highlighted in red. As you have only just created this file, all the content is highlighted

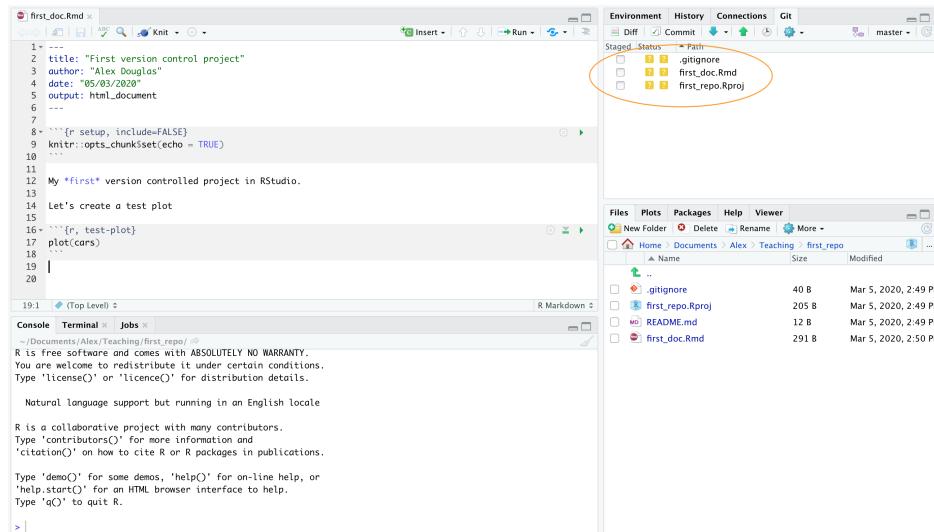


Figure 7.26.

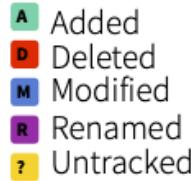


Figure 7.27.

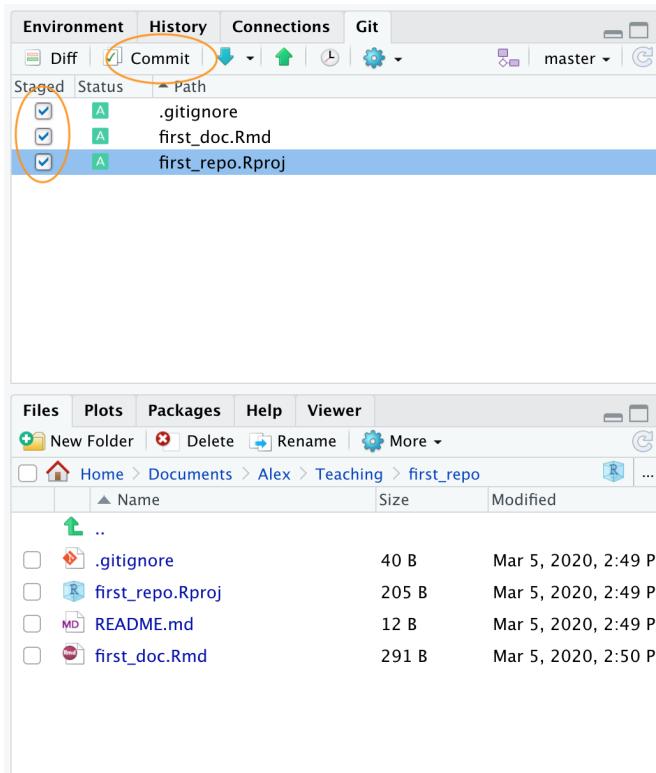


Figure 7.28.

in green. To commit these files (take a snapshot) first enter a mandatory commit message in the ‘Commit message’ box. This message should be relatively short and informative (to you and your collaborators) and indicate why you made the changes, not what you changed. This makes sense as Git keeps track of *what* has changed and so it is best not to use commit messages for this purpose. It’s traditional to enter the message ‘First commit’ (or ‘Initial commit’) when you commit files for the first time. Now click on the ‘Commit’ button to commit these changes.

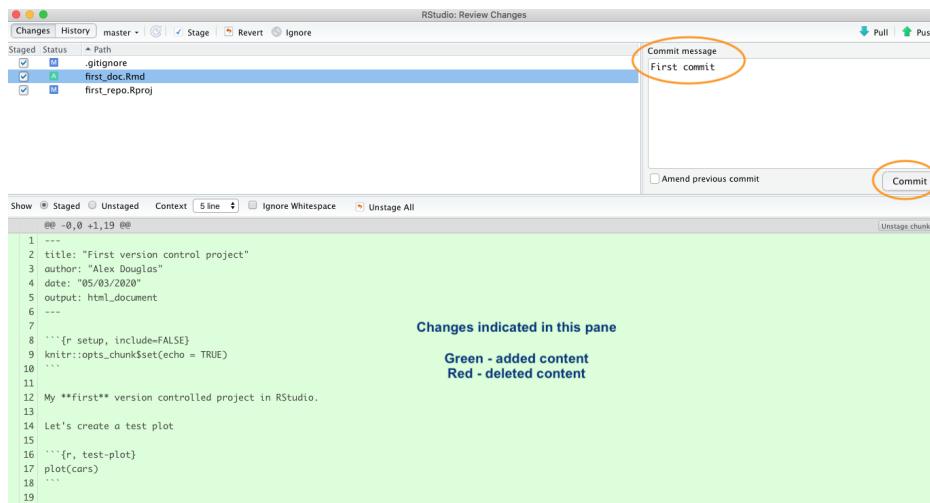


Figure 7.29.

A summary of the commit you just performed will be shown. Now click on the ‘Close’ button to return to the ‘Review Changes’ window. Note that the staged files have now been removed.



Figure 7.30.

Now that you have committed your changes the next step is to **push** these changes to GitHub. Before you push your changes it’s good practice to first **pull** any changes from GitHub. This is especially important if both you and your collaborators are working on the same files as it keeps your local copy up to date and avoids any potential conflicts. In this case your repository will already be up to date but it’s a good habit to get into. To do this, click on the ‘Pull’ button on the top right of the ‘Review Changes’ window. Once you have pulled any changes click on the green ‘Push’ button to push your changes. You will see a summary of the push you just performed. Hit the ‘Close’ button and

then close the ‘Review Changes’ window.

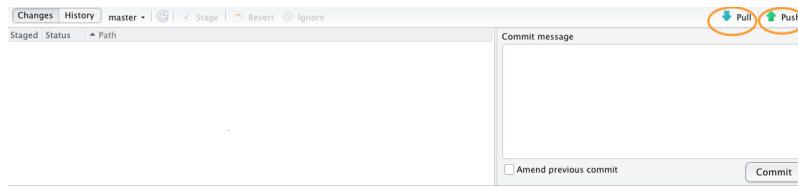


Figure 7.31.

To confirm the changes you made to the project have been pushed to GitHub, open your GitHub page, click on the Repositories link and then click on the `first_repo` repository. You should see four files listed including the `first_doc.Rmd` you just pushed. Along side the file name you will see your last commit message ('First commit' in this case) and when you made the last commit.

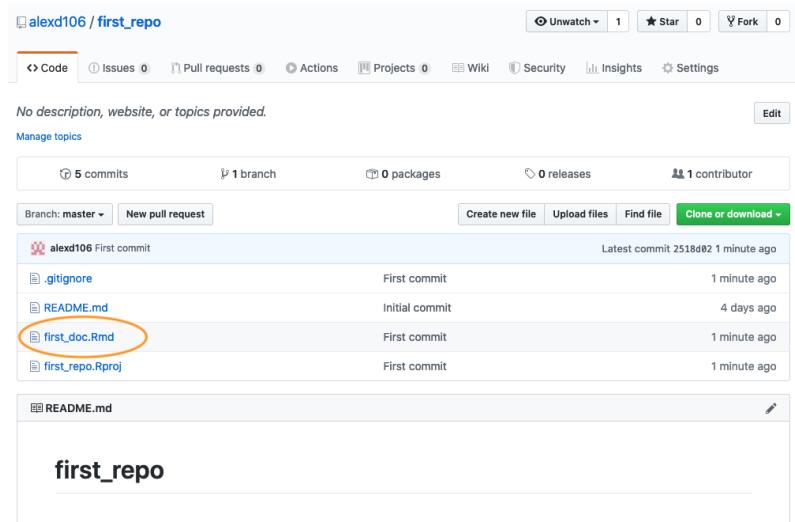


Figure 7.32.

To see the contents of the file click on the `first_doc.Rmd` file name.

7.6.1. Tracking changes

After following the steps outlined above, you will have successfully modified an RStudio project by creating a new R markdown document, staged and then committed these changes and finally pushed the changes to your GitHub repository. Now let’s make some further changes to your R markdown file and follow the workflow once again but this time we’ll take a look at how to identify changes made to files, examine the commit history and how to restore to a previous version of the document.

In RStudio open up the `first_repo.Rproj` file you created previously (if not already open) then open the `first_doc.Rmd` file (click on the file name in the Files tab in RStudio).

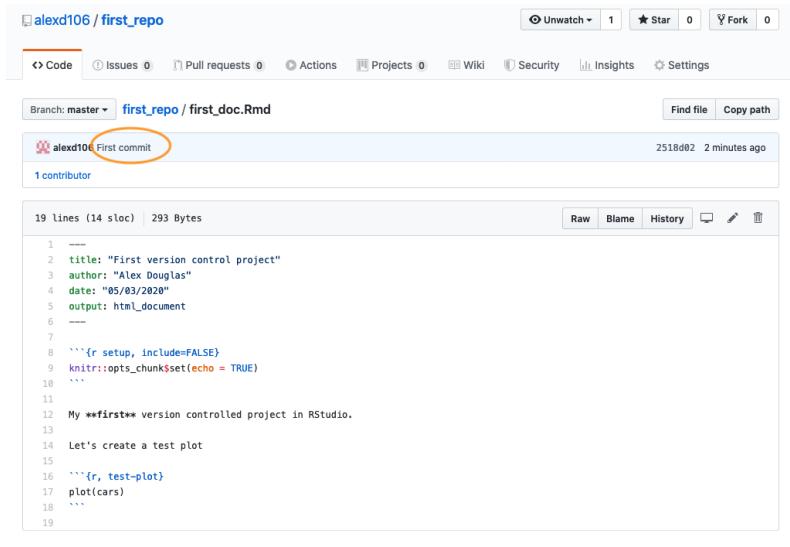


Figure 7.33.

Let's make some changes to this document. Delete the line beginning with 'My first version controlled ...' and replace it with something more informative (see figure below). We will also change the plotted symbols to red and give the plot axes labels. Lastly, let's add a summary table of the dataframe using the `kable()` and `summary()` functions (you may need to install the `knitr` package if you haven't done so previously to use the `kable()` function) and finally render this document to pdf by changing the YAML option to `output: pdf_document`.

The screenshot shows an RStudio editor window with the file 'first_doc.Rmd'. The code has been modified to remove the introductory text and add a summary table using `kable()`. The modified code includes a summary table of the 'cars' data frame.

```

1
2
3
4
5
6
7
8
9
10
11
12 This report documents my first attempts of using Git and GitHub to version control an RStudio project. I will be modifying this report, staging and committing changes and then pushing to GitHub.
13
14 Let's create a test plot of distance (miles) and speed (mph).
15
16 ````{r, test-plot}
17 plot(cars, col = "red", xlab = "speed (mph)", ylab = "distance (miles)")
18
19
20 A summary of the data frame is given below
21
22 ````{r, cars-summary}
23 library(knitr)
24 kable(summary(cars))
25
26
27
28
29

```

Figure 7.34.

Now save these changes and then click the `knit` button to render to pdf. A new pdf file named `first_doc.pdf` will be created which you can view by clicking on the file name in the `Files` tab in RStudio.

Notice that these two files have been added to the `Git` tab in RStudio. The status icons indicate that the `first_doc.Rmd` file has been modified (capital M) and the `first_doc.pdf` file is currently untracked (question mark).

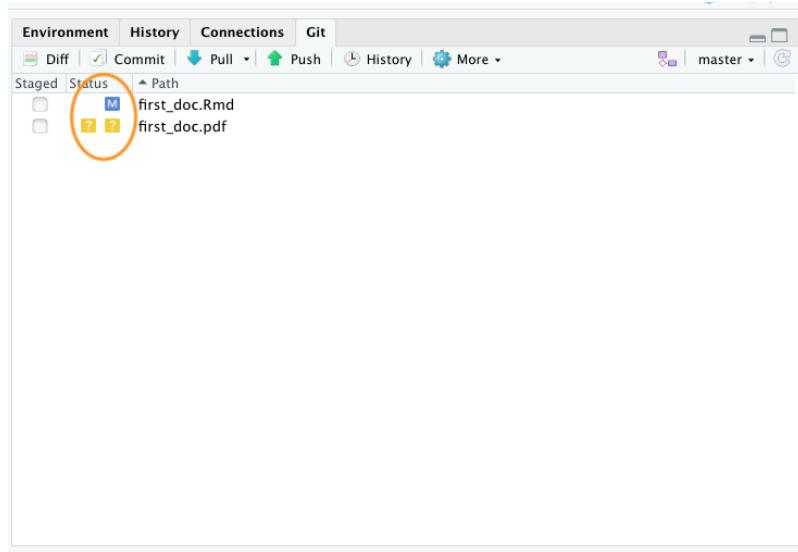


Figure 7.35.

To stage these files tick the ‘Staged’ box for each file and click on the ‘Commit’ button to take you to the ‘Review Changes’ window

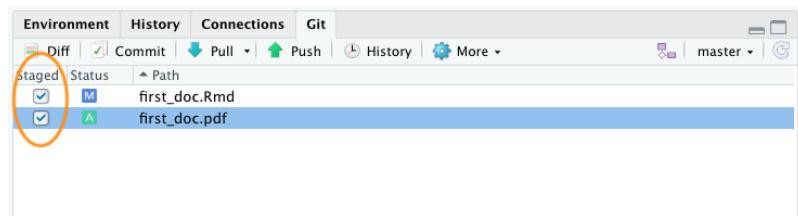


Figure 7.36.

Before you commit your changes notice the status of `first_doc.pdf` has changed from untracked to added (A). You can view the changes you have made to the `first_doc.Rmd` by clicking on the file name in the top left pane which will provide you with a useful summary of the changes in the bottom pane (technically called diffs). Lines that have been deleted are highlighted in red and lines that have been added are highlighted in green (note that from Git’s point of view, a modification to a line is actually two operations: the removal of the original line followed by the creation of a new line). Once you’re happy, commit these changes by writing a suitable commit message and click on the ‘Commit’ button.

To push the changes to GitHub, click on the ‘Pull’ button first (remember this is good practice even though you are only collaborating with yourself at the moment) and then click on the ‘Push’ button. Go to your online GitHub repository and you will see your new commits, including the `first_doc.pdf` file you created when you rendered your R markdown document.

To view the changes in `first_doc.Rmd` click on the file name for this file.

The screenshot shows the RStudio interface for a 'Review Changes' session. The top bar includes 'Changes', 'History', 'master', 'Stage', 'Revert', 'Ignore', 'Pull', and 'Push' buttons. The left panel lists files: 'first_doc.Rmd' (M) and 'first_doc.pdf' (A). The right panel shows the 'Commit message' field with the text 'improved plot and added summary table of dataframe'. Below it is an 'Amend previous commit' checkbox and a 'Commit' button, which is circled in orange.

line numbers

deleted lines

added lines

```

@@ -1,19 +1,28 @@
1 ---  

2 title: "First version control project"  

3 author: "Alex Douglas"  

4 date: "05/03/2020"  

5 output: html_document  

5 output: pdf_document  

6 ---  

7  

8 ```{r setup, include=FALSE}  

9 knitr::opts_chunk$set(echo = TRUE)  

10 ````  

11  

12 My **first** version controlled project in RStudio.  

12 This report documents my first attempts of using Git and Github to version control an RStudio project. I will be  

modifying this report, staging and committing changes and then pushing to GitHub.  

13  

14 Let's create a test plot  

14 Let's create a test plot of distance (miles) and speed (mph).  

15  

16 ```{r, test-plot}  

17 plot(cars)  

17 plot(cars, col = "red", xlab = "speed (mph)", ylab = "distance (miles)")  

18 ````  

19  

20 A summary of the data frame is given below  

21  

22 ```{r, cars-summary}  

23 library(knitr)  

24 kable(summary(cars))  

25 ````  

26  

27  

28

```

Figure 7.37.

The screenshot shows a GitHub repository page for 'alex106/first_repo'. The top navigation bar includes 'Unwatch', '1 star', '0 forks', and 'Edit'. Below the bar, there are tabs for 'Code', 'Issues (0)', 'Pull requests (0)', 'Actions', 'Projects (0)', 'Wiki', 'Security', 'Insights', and 'Settings'. The main area displays a message 'No description, website, or topics provided.' and a 'Manage topics' button. Below this, a summary bar shows '6 commits', '1 branch', '0 packages', '0 releases', '1 contributor', and a 'Clone or download' button. A search bar at the top of the commit list allows for 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The commit list shows the following entries:

- alex106 improved plot and added summary table of dataframe (Latest commit c0f073f 2 hours ago)
- .gitignore First commit (2 days ago)
- README.md Initial commit (6 days ago)
- first_doc.Rmd improved plot and added summary table of dataframe (2 hours ago)
- first_doc.pdf improved plot and added summary table of dataframe (2 hours ago)
- first_repo.Rproj First commit (2 days ago)

The 'first_doc.Rmd' and 'first_doc.pdf' commits are circled in orange.

Figure 7.38.

The screenshot shows a GitHub commit page for the repository 'alex106 / first_repo'. The commit message is: 'alexd106 improved plot and added summary table of dataframe'. The commit was made 2 hours ago by 'alexd106' with 1 contributor. The R code in the commit is:

```

1  ---
2  title: "First version control project"
3  author: "Alex Douglas"
4  date: "05/03/2020"
5  output: pdf_document
6  ---
7
8  ```{r setup, include=FALSE}
9  knitr::opts_chunk$set(echo = TRUE)
10 ```
11
12 This report documents my first attempts of using Git and Github to version control an RStudio project. I will be modifying this
13
14 Let's create a test plot of distance (miles) and speed (mph).
15
16 ```{r, test-plot}
17 plot(cars, col = "red", xlab = "speed (mph)", ylab = "distance (miles)")
18 ```
19
20 A summary of the data frame is given below
21
22 ```{r, cars-summary}
23 library(knitr)
24 kable(summary(cars))
25 ```
26
27
28

```

Figure 7.39.

7.6.2. Commit history

One of the great things about Git and GitHub is that you can view the history of all the commits you have made along with the associated commit messages. You can do this locally using RStudio (or the Git command line) or if you have pushed your commits to GitHub you can check them out on the GitHub website.

To view your commit history in RStudio click on the ‘History’ button (the one that looks like a clock) in the Git pane to bring up the history view in the ‘Review Changes’ window. You can also click on the ‘Commit’ or ‘Diff’ buttons which takes you to the same window (you just need to additionally click on the ‘History’ button in the ‘Review Changes’ window).

The history window is split into two parts. The top pane lists every commit you have made in this repository (with associated commit messages) starting with the most recent one at the top and oldest at the bottom. You can click on each of these commits and the bottom pane shows you the changes you have made along with a summary of the **Date** the commit was made, **Author** of the commit and the commit message (**Subject**). There is also a unique identifier for the commit (**SHA** - Secure Hash Algorithm) and a **Parent** SHA which identifies the previous commit. These SHA identifiers are really important as you can use them to view and revert to previous versions of files (details below Section 7.6.3). You can also view the contents of each file by clicking on the ‘View file @ SHA key’ link (in



Figure 7.40.

our case 'View file @ 2b4693d1').

| SHA | Author | Date | SHA |
|----------|----------------------------------|------------|----------|
| 2b4693d1 | alex106 <alex106@gmail.com> | 2020-03-24 | 2b4693d1 |
| d27e79f1 | alex106 <alex106@gmail.com> | 2020-03-24 | d27e79f1 |
| c80b0c75 | Alex Douglas <alex106@gmail.com> | 2020-03-24 | c80b0c75 |

first_doc.Rmd

```

@@ -2,17 +2,27 @@
 2 | 2 | title: "First version control project"
 3 | 3 | author: "Alex Douglas"
 4 | 4 | date: "05/03/2020"
 5 | output: html_document
 5 | output: pdf_document
 6 | ---
 7 | 
 8 | ````{r setup, include=FALSE}
 9 | knitr::opts_chunk$set(echo = TRUE)
10 | ```
11 | 
12 | My **first** version controlled project in RStudio.
13 | This report documents my first attempts at using Git and GitHub to version control on RStudio project. I will be modifying this report, staging and committing changes and then pushing to GitHub.

```

Figure 7.41.

You can also view your commit history on GitHub website but this will be limited to only those commits you have already pushed to GitHub. To view the commit history navigate to the repository and click on the 'commits' link (in our case the link will be labelled '3 commits' as we have made 3 commits).

You will see a list of all the commits you have made, along with commit messages, date of commit and the SHA identifier (these are the same SHA identifiers you saw in the RStudio history). You can even browse the repository at a particular point in time by clicking on the <> link. To view the changes in files associated with the commit simply click on the relevant commit link in the list.

Which will display changes using the usual format of green for additions and red for deletions.

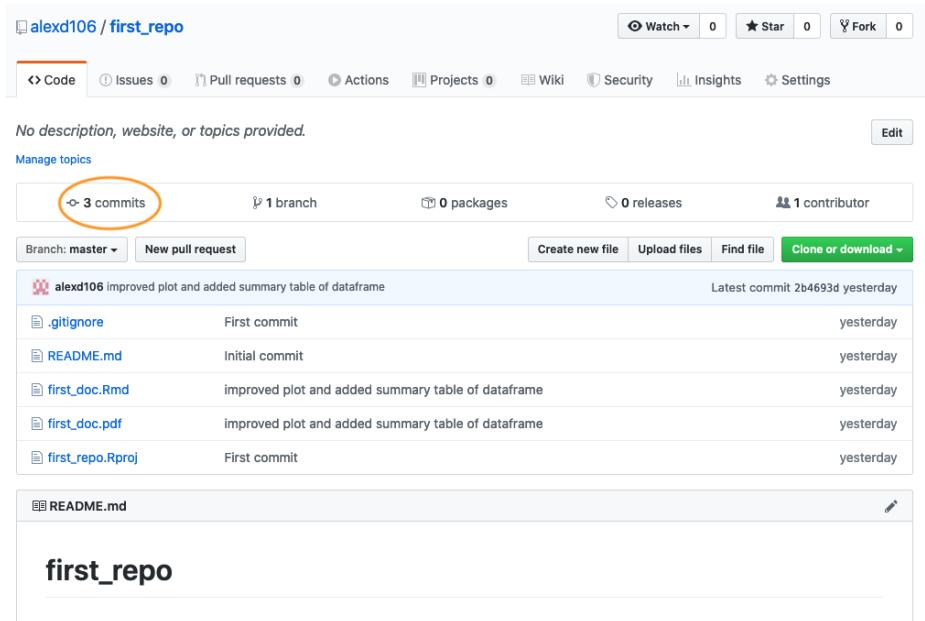


Figure 7.42.

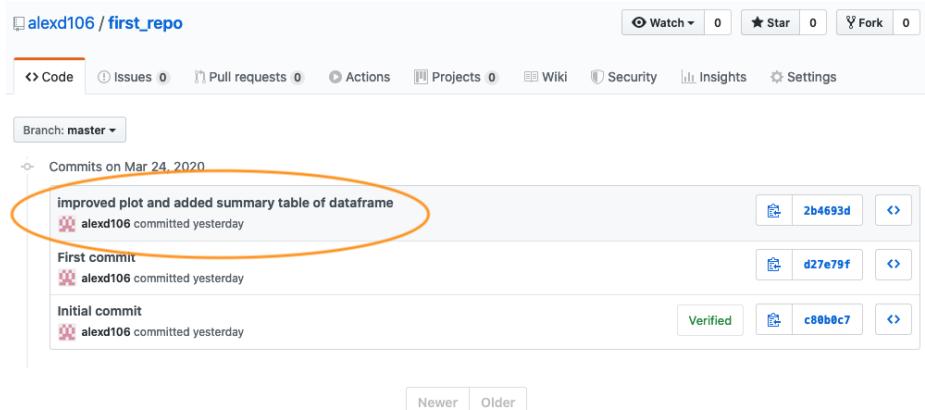


Figure 7.43.

```

improved plot and added summary table of dataframe
alex106 committed yesterday
1 parent d27e79f commit 2b4693d1e89ebc8db6400478a39d02203ba12bb8

Showing 2 changed files with 15 additions and 5 deletions.
Unified Split

20 first_doc.Rmd
@@ -2,17 +2,27 @@
 2   title: "First version control project"
 3   author: "Alex Douglas"
 4   date: "05/03/2020"
 5 - output: html_document
 5 + output: pdf_document
 6   -----
 7
 8   ```{r setup, include=FALSE}
 9   knitr::opts_chunk$set(echo = TRUE)
10
11
12 - My **first** version controlled project in RStudio.
12 + This report documents my first attempts at using Git and GitHub to version control an RStudio project. I will be
13   modifying this report, staging and committing changes and then pushing to GitHub.
14
15
16   ```{r, test-plot}
17 - plot(cars)
18
17 + plot(cars, col = "red", xlab = "speed (mph)", ylab = "distance (miles)")
18 + ```


```

Figure 7.44.

7.6.3. Reverting changes

One of the great things about using Git is that you are able to revert to previous versions of files if you've made a mistake, broke something or just prefer an earlier approach. How you do this will depend on whether the changes you want to discard have been staged, committed or pushed to GitHub. We'll go through some common scenarios below mostly using RStudio but occasionally we will need to resort to using the Terminal (still in RStudio though).

Changes saved but not staged, committed or pushed

If you have saved changes to your file(s) but not staged, committed or pushed these files to GitHub you can right click on the offending file in the Git pane and select ‘Revert …’. This will roll back all of the changes you have made to the same state as your last commit. Just be aware that you cannot undo this operation so use with caution.

You can also undo changes to just part of a file by opening up the ‘Diff’ window (click on the ‘Diff’ button in the Git pane). Select the line you wish to discard by double clicking on the line and then click on the ‘Discard line’ button. In a similar fashion you can discard chunks of code by clicking on the ‘Discard chunk’ button.

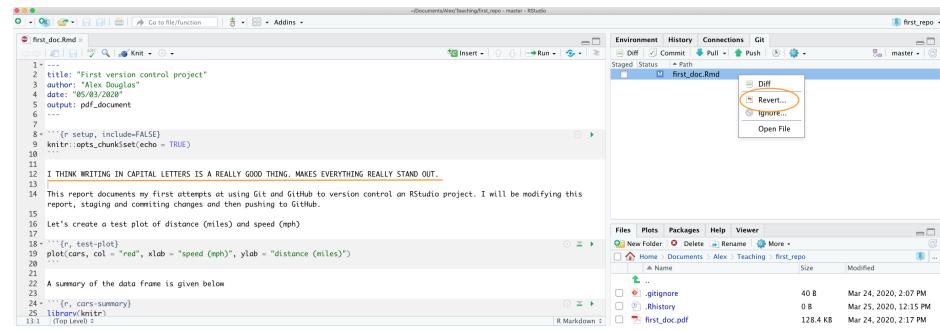


Figure 7.45.

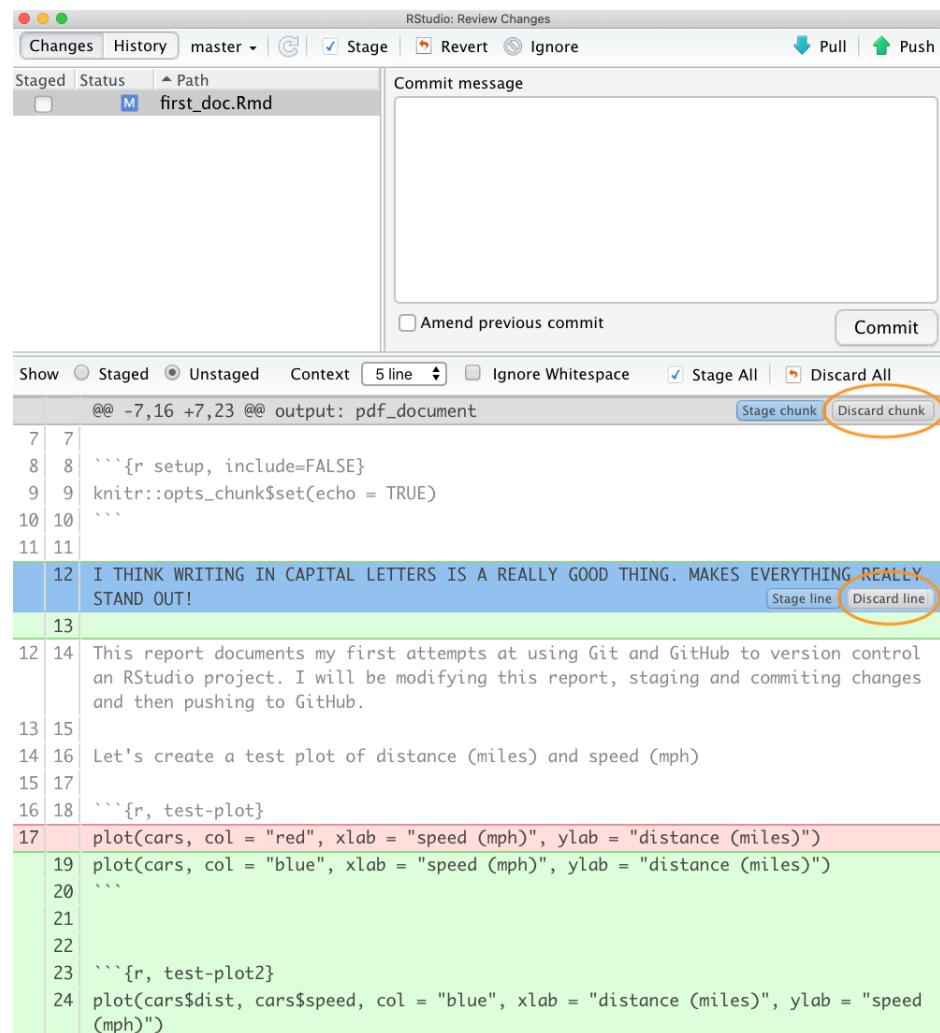


Figure 7.46.

Staged but not committed and not pushed

If you have staged your files, but not committed them then simply unstage them by clicking on the ‘Staged’ check box in the Git pane (or in the ‘Review Changes’ window) to remove the tick. You can then revert all or parts of the file as described in the section above.

Staged and committed but not pushed

If you have made a mistake or have forgotten to include a file in your last commit which you have not yet pushed to GitHub, you can just fix your mistake, save your changes, and then amend your previous commit. You can do this by staging your file and then tick the ‘Amend previous commit’ box in the ‘Review Changes’ window before committing.

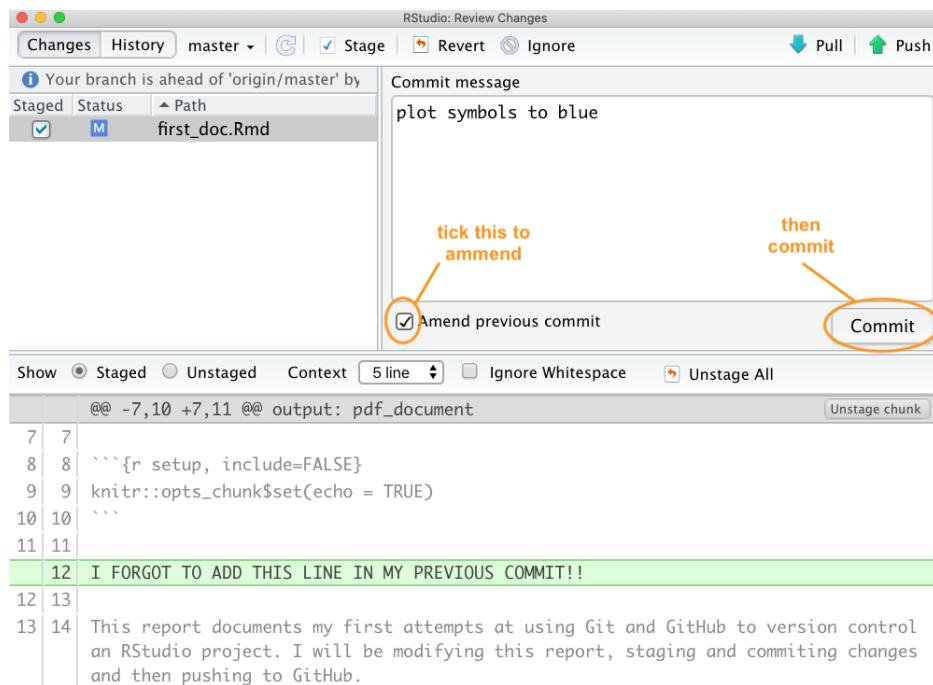


Figure 7.47.

If we check out our commit history you can see that our latest commit contains both changes to the file rather than having two separate commits. We use the amend commit approach a lot but it's important to understand that you should **not** do this if you have already pushed your last commit to GitHub as you are effectively rewriting history and all sorts bad things may happen!

If you spot a mistake that has happened multiple commits back or you just want to revert to a previous version of a document you have a number of options.

RStudio: Review Changes

Changes History master ▾ (all commits) ▾ C Search Pull

| Subject | Author | Date | SHA |
|--|----------------------------------|------------|----------|
| HEAD -> refs/heads/master plot symbols to blue | alex106 <alex106@gmail.com> | 2020-03-26 | ef8449f2 |
| origin/master origin/HEAD improved plot and adde | alex106 <alex106@gmail.com> | 2020-03-24 | 2b4693d1 |
| First commit | alex106 <alex106@gmail.com> | 2020-03-24 | d27e79f1 |
| Initial commit | Alex Douglas <alex106@gmail.com> | 2020-03-24 | c80b0c75 |

Commits 1-4 of 4

SHA ef8449f2
Author alex106 <alex106@gmail.com>
Date 2020-03-26 12:02
Subject plot symbols to blue
Parent 2b4693d1

 first_doc.Rmd

 first_doc.Rmd View file @ ef8449f2

```

@@ -9,12 +9,16 @@ output: pdf_document
 9 | 9 knitr::opts_chunk$set(echo = TRUE)
 10| 10 ``
 11| 11
 12| 12 I FORGOT TO ADD THIS LINE IN MY PREVIOUS COMMIT!!
 13| 13
 14| 14 This report documents my first attempts at using Git and GitHub to version
 15| 15 control an RStudio project. I will be modifying this report, staging and
 16| 16 committing changes and then pushing to GitHub.
 17| 17 Let's create a test plot of distance (miles) and speed (mph)
 18| 18 ````{r, test-plot}
 19| 19 plot(cars, col = "red", xlab = "speed (mph)", ylab = "distance (miles)")
 20| 20 ``

```

single commit includes both changes made to this file

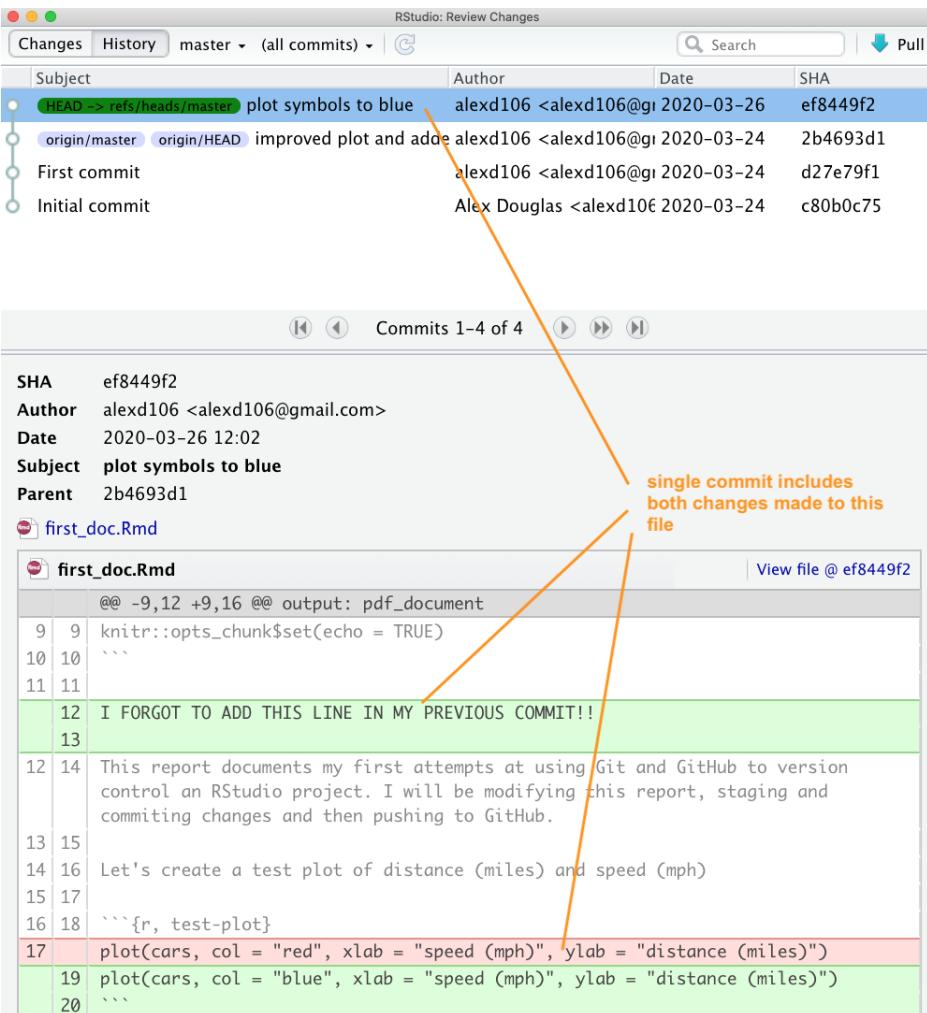


Figure 7.48.

Option 1 - (probably the easiest but very unGit - but like, whatever!) is to look in your commit history in RStudio, find the commit that you would like to go back to and click on the ‘View file @’ button to show the file contents.

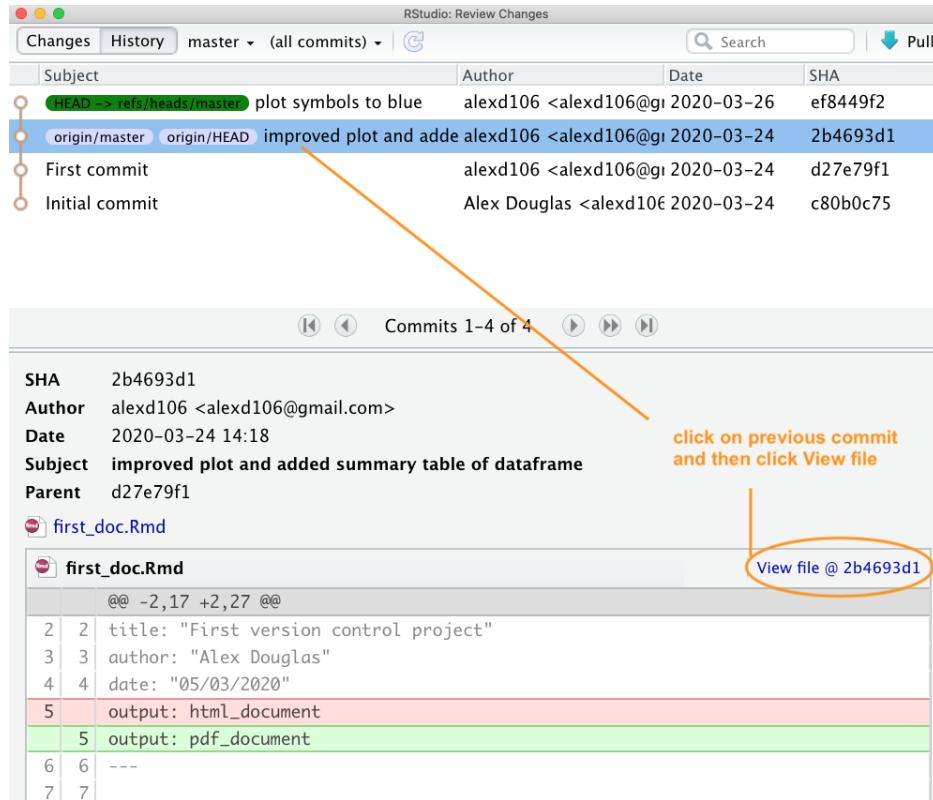


Figure 7.49.

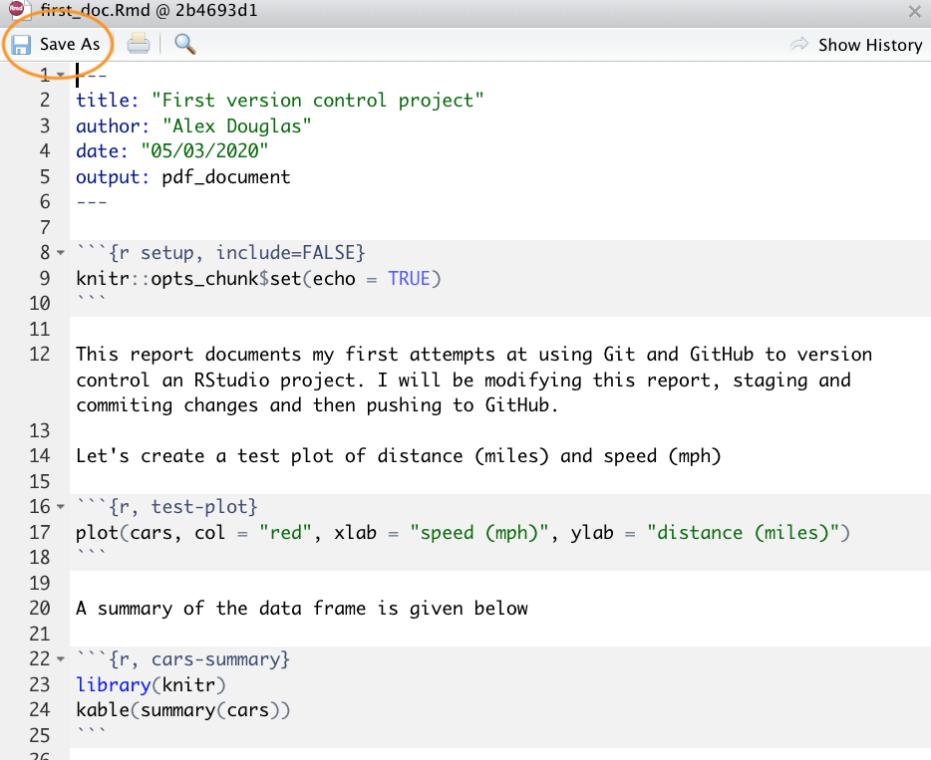
You can then copy the contents of the file to the clipboard and paste it into your current file to replace your duff code or text. Alternatively, you can click on the ‘Save As’ button and save the file with a different file name. Once you have saved your new file you can delete your current unwanted file and then carry on working on your new file. Don’t forget to stage and commit this new file.

Option 2 - (Git like) Go to your Git history, find the commit you would like to roll back to and write down (or copy) its SHA identifier.

Now go to the Terminal in RStudio and type `git checkout <SHA> <filename>`. In our case the SHA key is 2b4693d1 and the filename is `first_doc.Rmd` so our command would look like this:

```
git checkout 2b4693d1 first_doc.Rmd
```

The command above will copy the selected file version from the past and place it into the present. RStudio may ask you whether you want to reload the file as it now changed - select yes. You will also need to stage and commit the file as usual.



```

1 -+
2   title: "First version control project"
3   author: "Alex Douglas"
4   date: "05/03/2020"
5   output: pdf_document
6   ---
7
8 - ````{r setup, include=FALSE}
9   knitr::opts_chunk$set(echo = TRUE)
10  ```
11
12 This report documents my first attempts at using Git and GitHub to version
13 control an RStudio project. I will be modifying this report, staging and
14 committing changes and then pushing to GitHub.
15
16 ````{r, test-plot}
17   plot(cars, col = "red", xlab = "speed (mph)", ylab = "distance (miles)")
18  ```
19
20 A summary of the data frame is given below
21
22 ````{r, cars-summary}
23   library(knitr)
24   kable(summary(cars))
25  ```
26

```

Figure 7.50.

If you want to revert all your files to the same state as a previous commit rather than just one file you can use (the single ‘dot’ . is important otherwise your HEAD will detach!):

```

git rm -r .
git checkout 2b4693d1 .

```

Note that this will delete all files that you have created since you made this commit so be careful!

Staged, committed and pushed

If you have already pushed your commits to GitHub you can use the git checkout strategy described above and then commit and push to update GitHub (although this is not really considered ‘best’ practice). Another approach would be to use git revert (Note: as far as we can tell git revert is not the same as the ‘Revert’ option in RStudio). The revert command in Git essentially creates a new commit based on a previous commit and therefore preserves all of your commit history. To rollback to a previous state (commit) you first need to identify the SHA for the commit you wish to go back to (as we did above) and then use the revert command in the Terminal. Let’s say we want to revert back to our ‘First commit’ which has a SHA identifier d27e79f1.

The screenshot shows the RStudio interface for reviewing changes. At the top, the title bar says "RStudio: Review Changes". Below it, the "Changes" tab is selected, showing a list of commits in the "master" branch. One commit is highlighted in blue: "improved plot and added summary table of dataframe" by "alex106 <alex106@gmail.com>" on March 24, 2020, with SHA 2b4693d1. An orange arrow points from this commit to a detailed view below. The detailed view shows the commit's metadata: SHA 2b4693d1, Author alex106 <alex106@gmail.com>, Date 2020-03-24 14:18, Subject improved plot and added summary table of dataframe, and Parent d27e79f1. It also shows the file "first_doc.Rmd" with its content:

```

@@ -2,17 +2,27 @@
2 | 2 | title: "First version control project"
3 | 3 | author: "Alex Douglas"
4 | 4 | date: "05/03/2020"
5 | output: html_document
5 | output: pdf_document
6 |
7 |
8 | ``{r setup, include=FALSE}
9 | knitr::opts_chunk$set(echo = TRUE)
10 |
11 |
12 | My **first** version controlled project in RStudio.
12 This report documents my first attempts at using Git and GitHub to version
control an RStudio project. I will be modifying this report, staging and
committing changes and then pushing to GitHub.

```

Figure 7.51.

This screenshot shows the RStudio interface for reviewing changes, similar to Figure 7.51. The commit history lists several commits, including "First commit" by "alex106 <alex106@gmail.com>" on March 24, 2020, with SHA d27e79f1. An orange arrow points from this commit to a detailed view below. The detailed view shows the commit's metadata: SHA d27e79f1, Author alex106 <alex106@gmail.com>, Date 2020-03-24 14:11, Subject First commit, and Parent c80b0c75. It also shows the files ".gitignore", "first_doc.Rmd", and "first_repo.Rproj" with their content. The "first_doc.Rmd" content is identical to the one in Figure 7.51.

Figure 7.52.

We can use the `revert` command as shown below in the Terminal. The `--no-commit` option is used to prevent us from having to deal with each intermediate commit.

```
git revert --no-commit d27e79f1..HEAD
```

Your `first_doc.Rmd` file will now revert back to the same state as it was when you did your ‘First commit’. Notice also that the `first_doc.pdf` file has been deleted as this wasn’t present when we made our first commit. You can now stage and commit these files with a new commit message and finally push them to GitHub. Notice that if we look at our commit history all of the commits we have made are still present.

The screenshot shows the RStudio interface with the 'Review Changes' tab selected. The commit history table lists seven commits:

| Subject | Author | Date | SHA |
|---|----------------------------------|------------|----------|
| <code>HEAD -> refs/heads/master</code> | alex106 <alex106@gmail.com> | 2020-03-26 | 550640e2 |
| checkout to previous | alex106 <alex106@gmail.com> | 2020-03-26 | 5e4eccb4 |
| rolled back | alex106 <alex106@gmail.com> | 2020-03-26 | 6a4a9a9b |
| plot symbols to blue | alex106 <alex106@gmail.com> | 2020-03-26 | ef8449f2 |
| improved plot and added summary table of datafr | alex106 <alex106@gmail.com> | 2020-03-24 | 2b4693d1 |
| First commit | alex106 <alex106@gmail.com> | 2020-03-24 | d27e79f1 |
| Initial commit | Alex Douglas <alex106@gmail.com> | 2020-03-24 | c80b0c75 |

Below the table, the commit details for the first commit are shown:

```

SHA      550640e2
Author   alex106 <alex106@gmail.com>
Date     2020-03-26 16:04
Subject  tried revert command
Parent   5e4eccb4

```

The file content for `first_doc.Rmd` is displayed in a code editor window:

```

@ first_doc.Rmd
@@ -2,27 +2,17 @@
2 | 2 title: "First version control project"
3 | 3 author: "Alex Douglas"
4 | 4 date: "05/03/2020"
5 | 5 output: pdf_document
5 | 5 output: html_document
6 | 6 ---
7 |
8 | 8 ```{r setup, include=FALSE}
9 | 9 knitr::opts_chunk$set(echo = TRUE)
10| 10 ```
11|
12| 12 This report documents my first attempts at using Git and GitHub to version
control an RStudio project. I will be modifying this report, staging and
committing changes and then pushing to GitHub.
12| 12 My **first** version controlled project in RStudio.

```

Figure 7.53.

and our repo on GitHub also reflects these changes

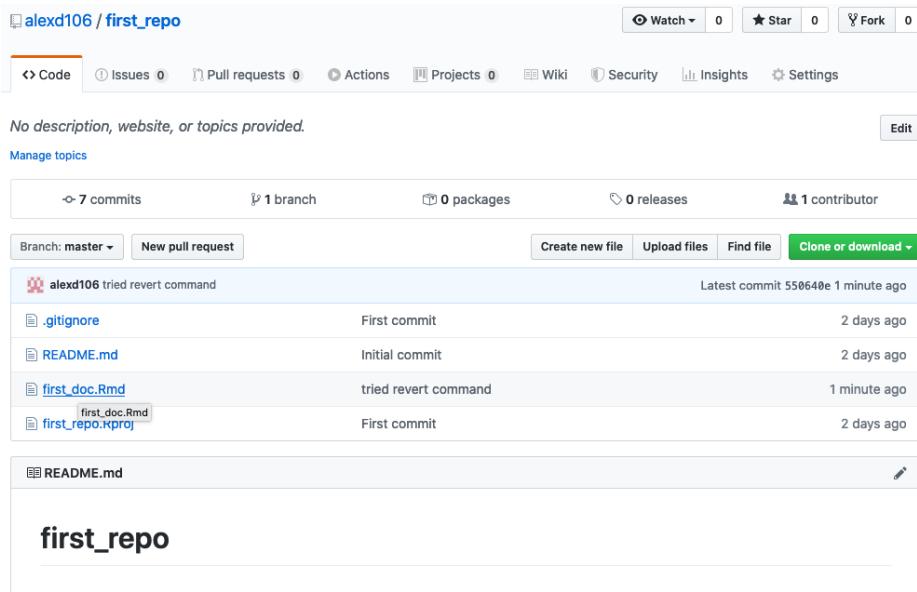


Figure 7.54.

7.7. Using Git with VSCode

Now that we have our project and repositories (both local and remote) set up, it's finally time to learn how to use Git in VSCode!

Typically, when using Git your workflow will go something like this:

1. You create/delete and edit files in your project directory on your computer as usual (saving these changes as you go)
2. Once you've reached a natural 'break point' in your progress (i.e. you'd be sad if you lost this progress) you **stage** these files
3. You then **commit** the changes you made to these staged files (along with a useful commit message) which creates a permanent snapshot of these changes
4. You keep on with this cycle until you get to a point when you would like to **push** these changes to GitHub
5. If you're working with other people on the same project you may also need to **pull** their changes to your local computer

OK, let's go through an example to help clarify this workflow.

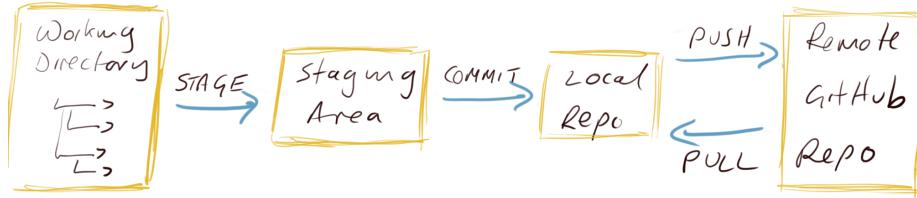


Figure 7.55.

Tracking changes

Commit History

Reverting changes

7.8. Collaborate with Git

GitHub is a great tool for collaboration, it can seem scary and complicated at first, but it is worth investing some time to learn how it works. What makes GitHub so good for collaboration is that it is a *distributed system*, which means that every collaborator works on their own copy of the project and changes are then merged together in the remote repository. There are two main ways you can set up a collaborative project on GitHub. One is the workflow we went through above, where everybody connects their local repository to the same remote one; this system works well with small projects where different people mainly work on different aspects of the project but can quickly become unwieldy if many people are collaborating and are working on the same files (merge misery!). The second approach consists of every collaborator creating a copy (or **fork**) of the main repository, which becomes their remote repository. Every collaborator then needs to send a request (a **pull request**) to the owner of the main repository to incorporate any changes into the main repository and this includes a review process before the changes are integrated. More detail of these topics can be found in Section 7.10.

7.9. Git tips

Generally speaking you should commit often (including amended commits) but push much less often. This makes collaboration easier and also makes the process of reverting to previous versions of documents much more straight forward. We generally only push changes to GitHub when we're happy for our collaborators (or the rest of the world)

to see our work. However, this is entirely up to you and depends on the project (and who you are working with) and what your priorities are when using Git.

If you don't want to track a file in your repository (maybe they are too large or transient files) you can get Git to ignore the file by adding it to the `.gitignore` file. On RStudio, in the git pane, you can right clicking on the filename to exclude and selecting 'Ignore...'

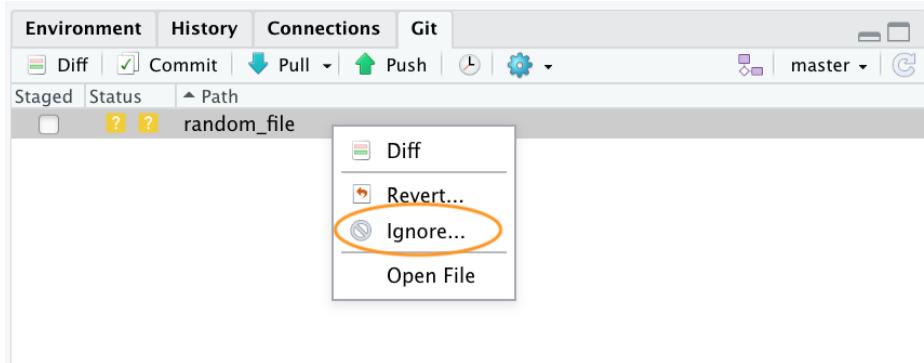


Figure 7.56.

This will add the filename to the `.gitignore` file. If you want to ignore multiple files or a particular type of file you can also include wildcards in the `.gitignore` file. For example to ignore all png files you can include the expression `*.png` in your `.gitignore` file and save.

If it all goes pear shaped and you end up completely trashing your Git repository don't despair (we've all been there!). As long as your GitHub repository is good, all you need to do is delete the offending project directory on your computer, create a new RStudio project and link this with your remote GitHub repository using Option 2 (-Section 7.5.3). Once you have cloned the remote repository you should be good to go.

7.10. Further resources

There are many good online guides to learn more about git and GitHub and as with any open source software there is a huge community that can be a great resource:

- The British Ecological Society guide to [Reproducible Code](#)
- The [GitHub guides](#)
- The Mozilla Science Lab [GitHub for Collaboration on Open Projects guide](#)
- Jenny Bryan's [Happy Git and GitHub](#). We borrowed the idea (but with different content) of RStudio first, RStudio second in the 'Setting up a version controlled Project in RStudio' section.

- Melanie Frazier's [GitHub: A beginner's guide to going back in time \(aka fixing mistakes\)](#). We followed this structure (with modifications and different content) in the ‘Reverting changes’ section.
- If you have done something terribly wrong and don't know how to fix it try [Oh Shit, Git](#) or if you're easily offended [Dangit, Git](#)

These are only a couple of examples, all you need to do is search for “version control with git and GitHub” to see how huge the community around these open source projects is and how many free resources are available for you to become a version control expert.

7.11. Practical

7.11.1. Context

We will configure Rstudio to work with our github account, then create a new project and start using github. To have some data I suggest to use the awesome `palmerpenguins` dataset .

7.11.2. Information of the data

These data have been collected and shared by [Dr. Kristen Gorman](#) and [Palmer Station, Antarctica LTER](#).

The package was built by Drs Allison Horst and Alison Hill, check out the [official website](#).

The package `palmerpenguins` has two datasets.

```
library(palmerpenguins)
```

The dataset `penguins` is a simplified version of the raw data; see `?penguins` for more info:

```
head(penguins)
```

| species | island | bill_length_mm | bill_depth_mm | flipper_length_mm | body_mass_g | sex | year |
|---------|-----------|----------------|---------------|-------------------|-------------|--------|------|
| Adelie | Torgersen | 39.1 | 18.7 | 181 | 3750 | male | 2007 |
| Adelie | Torgersen | 39.5 | 17.4 | 186 | 3800 | female | 2007 |
| Adelie | Torgersen | 40.3 | 18.0 | 195 | 3250 | female | 2007 |
| Adelie | Torgersen | NA | NA | NA | NA | NA | 2007 |

| species | island | bill_length_mm | bill_depth_mm | flipper_length_mm | body_mass_g | sex | year |
|---------|-----------|----------------|---------------|-------------------|-------------|--------|------|
| Adelie | Torgersen | 36.7 | 19.3 | 193 | 3450 | female | 2007 |
| Adelie | Torgersen | 39.3 | 20.6 | 190 | 3650 | male | 2007 |

The other dataset `penguins_raw` has the raw data; see `?penguins_raw` for more info:

```
head(penguins_raw)
```

| Name
ber | Species | In- | | | | | | | | | | | | |
|-------------|----------|------|-------------|-------|--------|-----------|--------|-------|--------|-------------|-------|----------|--------|-------|
| | | Sam- | | di- | Clutch | Cul- | Cul- | Flip- | Delta | | | | | |
| | | stu- | ple | vid- | Com- | men | men | per | Body | 15 | Delta | | | |
| dy- | Num- | Re- | Is- | ual | ple- | Date | Length | Depth | Length | Mass | N | 13 C | Com- | |
| Name | ber | gion | land | Stage | ID | Egg | (mm) | (mm) | (mm) | (g) | Sex | (o/oo) | (o/oo) | ments |
| PAL0708 | Adelie | An- | Torg-Adult, | N1A1 | Yes | 2007-39.1 | 18.7 | 181 | 3750 | MALNA | NA | Not | | |
| | Penguin | vers | ersen1 | | | 11- | | | | | | enough | | |
| | (Py- | | Egg | | | 11 | | | | | | blood | | |
| | goscelis | | Stage | | | | | | | | | for iso- | | |
| | adeliae) | | | | | | | | | | | topes. | | |
| PAL0708 | Adelie | An- | Torg-Adult, | N1A2 | Yes | 2007-39.5 | 17.4 | 186 | 3800 | FE- 8.94956 | - | NA | | |
| | Penguin | vers | ersen1 | | | 11- | | | | | MALE | 24.69454 | | |
| | (Py- | | Egg | | | 11 | | | | | | | | |
| | goscelis | | Stage | | | | | | | | | | | |
| | adeliae) | | | | | | | | | | | | | |
| PAL0708 | Adelie | An- | Torg-Adult, | N2A1 | Yes | 2007-40.3 | 18.0 | 195 | 3250 | FE- 8.36821 | - | NA | | |
| | Penguin | vers | ersen1 | | | 11- | | | | | MALE | 25.33302 | | |
| | (Py- | | Egg | | | 16 | | | | | | | | |
| | goscelis | | Stage | | | | | | | | | | | |
| | adeliae) | | | | | | | | | | | | | |
| PAL0708 | Adelie | An- | Torg-Adult, | N2A2 | Yes | 2007- NA | NA | NA | NA | NA | NA | Adult | | |
| | Penguin | vers | ersen1 | | | 11- | | | | | | not | | |
| | (Py- | | Egg | | | 16 | | | | | | sam- | | |
| | goscelis | | Stage | | | | | | | | | pled. | | |
| | adeliae) | | | | | | | | | | | | | |

| Name | Species | Region | Stage | ID | Re- | Is- | Ual | di- | vid- | Clutch | Com- | Cul- | Cul- | Flip- | per | Body | Delta | | | |
|---------|----------------|--------|------------------|----|------|--------|-------|-----|------|------------------|--------|-----------|--------|-------|--------|------------|----------|-------|------|--|
| | | | | | | | | | | | | | | | | | N | 13 C | Com- | |
| PAL0708 | Adelie Penguin | An- | Torg-Adult, (Py- | | vers | ersen1 | Egg | | | Torg-Adult, (Py- | ersen1 | Date | Length | Depth | Length | Mass | | | | |
| | | | goscelis | | | | Stage | | | | | Egg (mm) | (mm) | (mm) | (g) | Sex (o/oo) | (o/oo) | ments | | |
| PAL0708 | Adelie Penguin | An- | Torg-Adult, (Py- | | vers | ersen1 | Egg | | | Torg-Adult, (Py- | ersen1 | 2007-36.7 | 19.3 | 193 | 3450 | FE- | 8.76651 | - | NA | |
| | | | goscelis | | | | Stage | | | | | | | | | MALE | 25.32426 | | | |
| | | | adeliae) | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |

For this exercise, we're gonna use the penguins dataset.

7.11.3. Questions

- 1) Create a github account if not done yet.
- 2) Configure Rstudio with your github account using the `usethis` package.
- 3) Create and Store your GITHUB Personal Authorisation Token
- 4) Create a new R Markdown project, initialize it for git, and create a new git repository
- 5) Create a new Rmarkdown document, in your project. Then save the file and stage it.
- 6) Create a new commit including the new file and push it to github (Check on github that it works).
- 7) Edit the file. Delete everything after line 12. Add a new section title, simple text and text in bold font. Then knit and compile.
- 8) Make a new commit (with a meaningful message), and push to github.

9) Create a new branch, and add a new section to the rmarkdown file in this branch. Whatever you want. I would suggest a graph of the data.

10) Create a commit and push it to the branch.

11) On github, create a pull request to merge the 2 different branches.

12) Check and accept the pull request to merge the 2 branches.

You have successfully used all the essential tools of git 🎉. You are really to explore 💬 and discover its power 💪



Figure 7.57.: Happy git(hub)-ing

7.11.4. Solution

2)

```
usethis::git_sitrep()  
usethis::use_git_config(  
  user.name = "your_username",  
  user.email = "your_email@address.com"  
)
```

3)

```
usethis::create_github_token()  
gitcreds::gitcreds_set()
```

4)

```
#create R project  
usethis::use_git()  
  
#restart R  
usethis::use_github()  
usethis::git_vaccinate()
```

Part II.

Fundamentals of stats

Chapter 8

Power Analysis

8.1. The theory

8.1.1. What is power?

Power is the probability of rejecting the null hypothesis when it is false

8.1.2. Why do a power analysis?

Assess the strength of evidence

Power analysis, performed after accepting a null hypothesis, can help assess the probability of rejecting the null if it were false, and if the magnitude of the effect was equal to that observed (or to any other given magnitude). This type of *a posteriori* analysis is very common.

Design better experiments

Power analysis, performed prior to conducting an experiment (but most often after a preliminary experiment), can be used to determine the number of observations required to detect an effect of a given magnitude with some probability (the power). This type of *a priori* experiment should be more common.

Estimate minimum detectable effect

Sampling effort is often predetermined (when you are handed data of an experiment already completed), or extremely constrained (when logistics dictates what can be done). Whether it is *a priori* or *a posteriori*, power analysis can help you estimate, for a fixed sample size and a given power, what is the minimum effect size that can be detected.

8.1.3. Factors affecting power

For a given statistical test, there are 3 factors that affect power.

Decision criteria

Power is related to α , the probability level at which one rejects the null hypothesis. If this decision criteria is made very strict (i.e. if critical α is set to a very low value, like 0.1% or $p = 0.001$), then power will be lower than if the critical α was less strict.

Sample size

The larger the sample size, the larger the power. As sample size increases, one's ability to detect small effect sizes as being statistically significant gets better.

Effect size

The larger the effect size, the larger the power. For a given sample size, the ability to detect an effect as being significant is higher for large effects than for small ones. Effect size measures how false the null hypothesis is.

8.1.4. Types of power analyses

First, α is define as the probability level at which one rejects the null hypothesis, and β is $1 - \text{power}$.

A priori

Computes the sample size required given β , α , and the effect size. This type of analysis is useful when planning experiments.

Compromise

Computes α and β for a given α/β ratio, sample size, and effect size. Less commonly used (I have never used it myself) although it can be useful when the α/β ratio has meaning, for example when the cost of type I and type II errors can be quantified.

Criterion

Computes α for a given β , sample size, and effect size. In practice, I see little interest in this. Let me know if you see something I don't!

Post-hoc

Computes the power for a given α , effect size, and sample size. Used frequently to help in the interpretation of a test that is not statistically significant, but only if an effect size that is biologically significant is used (and not the observed effect size). Not relevant when the test is significant.

Sensitivity

Computes the detectable effect size for a given β , α , and sample size. Very useful at the planning stage of an experiment.

8.1.5. How to calculate effect size

The metric for effect size depends on the test. Note that other software packages often use different effect size metrics and that it is important to use the correct one for each package. G*Power, a software to easily perform power analysis, has an effect size calculator for many tests that only requires you to enter the relevant values. The following table lists the effect size metrics used by G*Power for the various tests.

| Test | Effect size | Formula |
|------------------------|-------------|--|
| t-test on means | d | $d = \frac{ \mu_1 - \mu_2 }{\sqrt{(s_1^2 + s_2^2)/2}}$ |
| t-test on correlations | r | |

| Test | Effect size | Formula |
|--------------------|-------------|---|
| other t-tests | d | $d = \frac{\mu}{\sigma}$ |
| F-test
(ANOVA) | f | $f = \frac{\sqrt{\sum_{i=1}^k (\mu_i - \mu)^2}}{\sigma}$ |
| other F-tests | f^2 | $f^2 = \frac{R_p^2}{1-R_p^2}$
R_p is the squared partial correlation coefficient |
| Chi-square
test | w | $w = \sqrt{\sum_{i=1}^m \frac{(p_{0i} - p_{1i})^2}{p_{0i}}}$
p_{0i} and p_{1i} are the proportion in category i predicted by the null, $_0$, and alternative, $_1$, hypothesis |

8.2. Practical

After completing this laboratory, you should :

- be able to compute the power of a t-test with G*Power and R
- be able to calculate the required sample size to achieve a desired power level with a t-test
- be able to calculate the detectable effect size by a t-test given the sample size, the power and α
- understand how power changes when sample size increases, the effect size changes, or when α decreases
- understand how power is affected when you change from a two-tailed to a one-tailed test.

8.2.1. What is G*Power?

G*Power is free software developed by quantitative psychologists from the University of Dusseldorf in Germany. It is available in MacOS and Windows versions. It can be run under Linux using Wine or a virtual machine.

G*Power will allow you to do power analyses for the majority of statistical tests we will cover during the term without making lengthy calculations and looking up long tables and figures of power curves. It is a really useful tool that you need to master.

It is possible to perform all analysis made by G*Power in R, but it requires a bit more code, and a better understanding of the process since everything should be coded by hand. In simple cases, R code is also provided.

 Caution

Download the software [here](#) and install it on your computer and your workstation (if it is not there already).

8.2.2. How to use G*Power

8.2.2.1. General Principle

Using G*Power generally involves 3 steps:

1. Choosing the appropriate test
2. Choosing one of the 5 types of available power analyses
3. Enter parameter values and press the **Calculate** button

8.2.3. Power analysis for a t-test on two independent means

 Important

All the power analysis presented in this chapter can be done using 2 functions in R.

- `pwr.t.test()` when the two samples have a similar size
- `pwr.t2n.test()` when samples have different numbers of observations

The objective of this lab is to learn to use G*Power and understand how the 4 parameters of power analyses (α , β , sample size and effect size) are related to each other. For this, you will only use the standard t-test to compare two independent means. This is the test most used by biologists, you have all used it, and it will serve admirably for this lab. What you will learn today will be applicable to all other power analyses.

Jayne Stephenson studied the productivity of streams in the Ottawa region. She has measured fish biomass in 36 streams, 18 on the Shield and 18 in the Ottawa Valley. She found that fish biomass was lower in streams from the valley (2.64 g/m^2 , standard deviation = 3.28) than from the Shield (3.31 g/m^2 , standard deviation = 2.79.).

When she tested the null hypothesis that fish biomass is the same in the two regions by a t-test, she obtained:

```
Pooled-Variance Two-Sample t-Test
t = -0.5746, df = 34, p-value = 0.5693
```

She therefore accepted the null hypothesis (since p is much larger than 0.05) and concluded that fish biomass is the same in the two regions.

8.2.4. Post-hoc analysis

Using the observed means and standard deviations, we can use G*Power to calculate the power of the two-tailed t-test for two independent means, using the observed effect size (the difference between the two means, weighted by the standard deviations) for $\alpha = 0.05$.

Start G*Power.

1. In ***Test family**, choose: t tests
2. For **Statistical test**, choose: Means: Difference between two independent means (two groups)
3. For **Type of power analysis**, choose: Post hoc: Compute achieved power - given α , sample size, and effect size
4. At **Input Parameters**,
 - in the box **Tail(s)**, chose: Two,
 - check that α **err prob** is equal to 0.05
 - Enter 18 for the **Sample size** of group 1 and of group 2
 - then, to calculate effect size (d), click on **Determine =>**
5. In the window that opens,
 - select **n1 = n2**, then
 - enter the two means (**Mean group 1 et 2**)
 - the two standard deviations(**SD group 1 et 2**)
 - click on **Calculate and transfer to main window**
6. After you click on the **Calculate button** in the main window, you should get the following:

Similar analysis can be done in R. You first need to calculate the effect size d for a t-test comparing 2 means, and then use the `pwr.t.test()` function from the `pwr`  . The easiest is to create anew function in R to estimate the effect size d since we are going to reuse it multiple times during the lab.

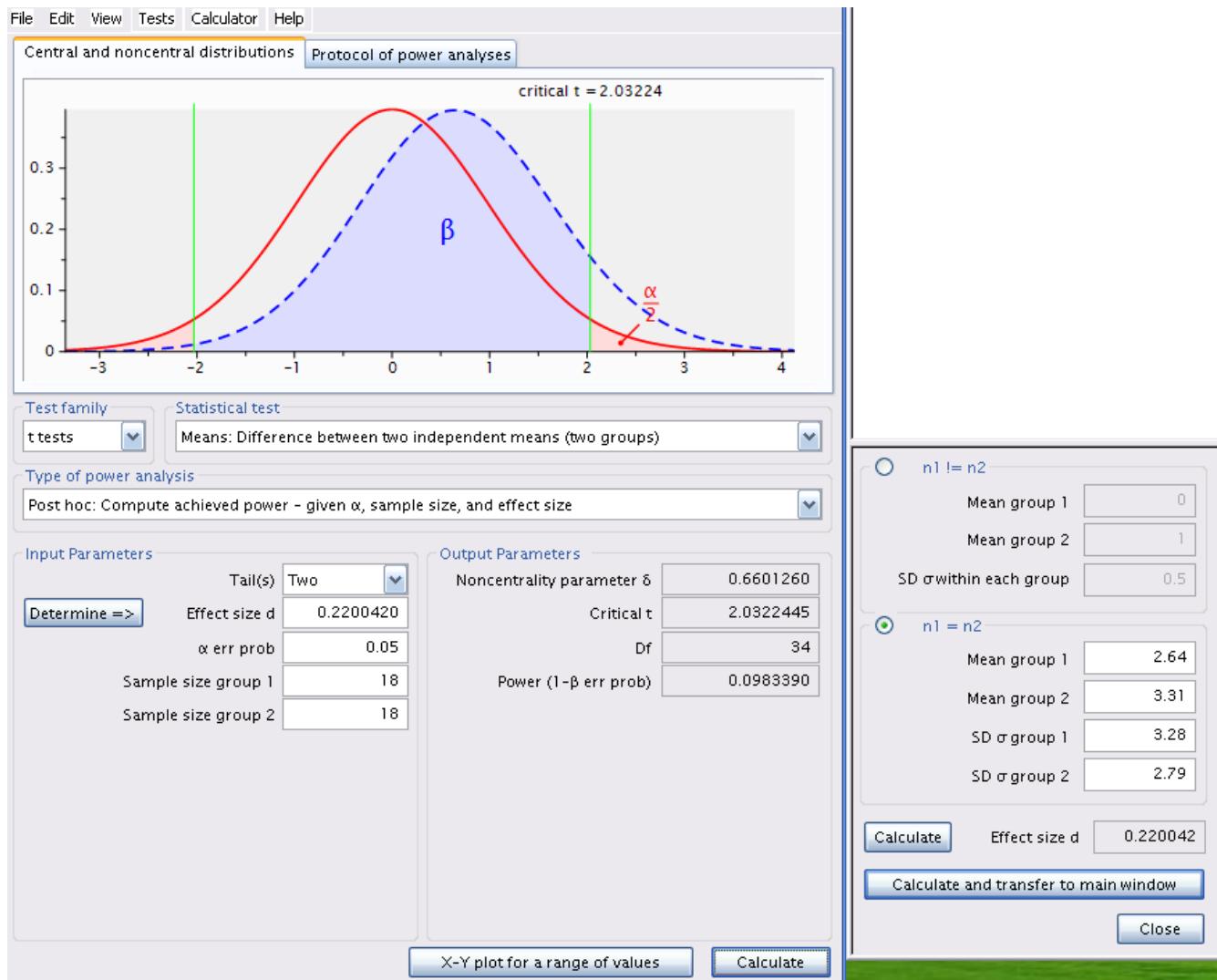


Figure 8.1.: Post-hoc analysis with estimated effect size

```
# load package pwr
library(pwr)

# define d for a 2 sample t-test
d <- function(u1, u2, sd1, sd2) {
  abs(u1 - u2) / sqrt((sd1^2 + sd2^2) / 2)
}

# power analysis
pwr.t.test(
  n = 18,
  d = d(u1 = 2.64, sd1 = 3.28, u2 = 3.31, sd2 = 2.79),
  sig.level = 0.05,
  type = "two.sample"
)
```

Two-sample t test power calculation

```
n = 18
d = 0.220042
sig.level = 0.05
power = 0.09833902
alternative = two.sided
```

NOTE: n is number in *each* group

```
# plot similar to G*Power
d_cohen <- d(u1 = 2.64, sd1 = 3.28, u2 = 3.31, sd2 = 2.79)

x <- seq(-4, 4, length = 200)
y0 <- dt(x, 34)
y1 <- dt(x, 34, ncp = d_cohen * sqrt(36) / 2)
plot(x, y0, type = "l", col = "red", lwd = 2)
qc <- qt(0.025, 34)
```

```

abline(v = qc, col = "green")
abline(v = -qc, col = "green")
lines(x, y1, type = "l", col = "blue", lwd = 2)

# type 2 error corresponds to the shaded area

polygon(
  c(x[x <= -qc], -qc), c(y1[x <= -qc], 0),
  col = rgb(red = 0, green = 0.2, blue = 1, alpha = 0.5)
)

```

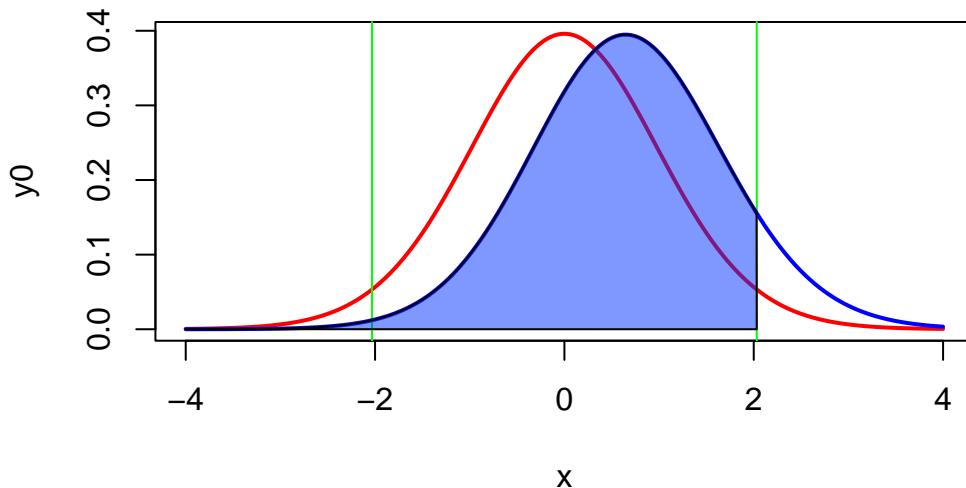


Figure 8.2.: Post-hoc analysis with estimated effect size in R

```

qc <- qt(0.025, 34)
ncp <- d_cohen * sqrt(36) / 2
dat <- data.frame(
  x = seq(-4, 4, length = 200),
  y0 = dt(x, (n - 1) * 2),
  y1 = dt(x, (n - 1) * 2, ncp = ncp)
) %>%
  mutate(
    qc = qt(0.025, 34),
    -qc = -qt(0.025, 34)
)

```

```
area = ifelse(x <= qc, y1, 0)
)

ggplot(dat, aes(x = x)) +
  geom_line(aes(y = y0), color = "red") +
  geom_line(aes(y = y1), color = "blue") +
  geom_vline(xintercept = qcl, color = "green") +
  geom_area(
    aes(x = x, y = area),
    fill = rgb(red = 0, green = 0.2, blue = 1, alpha = 0.5)
  ) +
  theme_classic() +
  ylab("dt(x)")
```

Let's examine the figure Figure 8.2.

- The curve on the left, in red, corresponds to the expected distribution of the t-statistics when H_0 is true (*i.e.* when the two means are equal) given the sample size (18 per region) and the observed standard deviations.
- The vertical green lines correspond to the critical values of t for $\alpha = 0.05$ and a total sample size of 36 (2x18).
- The shaded pink regions correspond to the rejection zones for H_0 . If Jaynie had obtained a *t-value* outside the interval delimited by the critical values ranging from -2.03224 to 2.03224, she would then have rejected H_0 , the null hypothesis of equal means. In fact, she obtained a t-value of -0.5746 and concluded that the biomass is equal in the two regions.
- The curve on the right, in blue, corresponds to the expected distribution of the t-statistics if H_1 is true (here H_1 is that there is a difference in biomass between the two regions equal to $3.33 - 2.64 = 0.69g/m^2$, given the observed standard deviations). This distribution is what we should observe if H_1 was true and we repeated a large number of times the experiment using random samples of 18 streams in each of the two regions and calculated a t-statistic for each sample. On average, we would obtain a t-statistic of about 0.6.
- Note that there is considerable overlap of the two distributions and that a large fraction of the surface under the right curve is within the interval where H_0 is accepted between the two vertical green lines at -2.03224 and 2.03224. This proportion, shaded in blue under the distribution on the right is labeled β and corresponds to the risk of *type II error* (accept H_0 when H_1 is true).
- Power is simply $1 - \beta$, and is here 0.098339. Therefore, if the mean biomass differed by $0.69g/m^2$ between the two regions, Jaynie had only 9.8% chance of being able to detect it as a statistically significant difference

at =5% with a sample size of 18 streams in each region.

Let's recapitulate: The difference in biomass between regions is not statistically significant according to the t-test. It is because the difference is relatively small relative to the precision of the measurements. It is therefore not surprising that that power, i.e. the probability of detecting a statistically significant difference, is small. Therefore, this analysis is not very informative.

Indeed, a post hoc power analysis using the observed effect size is not useful. It is much more informative to conduct a post hoc power analysis for an effect size that is different from the observed effect size. But what effect size to use? It is the biology of the system under study that will guide you. For example, with respect to fish biomass in streams, one could argue that a two fold change in biomass (say from 2.64 to 5.28 g/m²) has ecologically significant repercussions. We would therefore want to know if Jaynie had a good chance of detecting a difference as large as this before accepting her conclusion that the biomass is the same in the two regions. So, what were the odds that Jaynie could detect a difference of 2.64 g/m² between the two regions? G*Power can tell you if you cajole it the right way.

🔥 Exercise

Change the mean of group 2 to 5.28, recalculate effect size, and click on Calculate to obtain figure Figure 8.3.

Same analysis using R (without all the code for the interesting but not really useful plot)

```
pwr.t.test(
  n = 18,
  d = d(u1 = 2.64, sd1 = 3.28, u2 = 5.28, sd2 = 2.79),
  sig.level = 0.05,
  type = "two.sample")
```

Two-sample t test power calculation

```
n = 18
d = 0.8670313
sig.level = 0.05
power = 0.7146763
alternative = two.sided
```

NOTE: n is number in *each* group

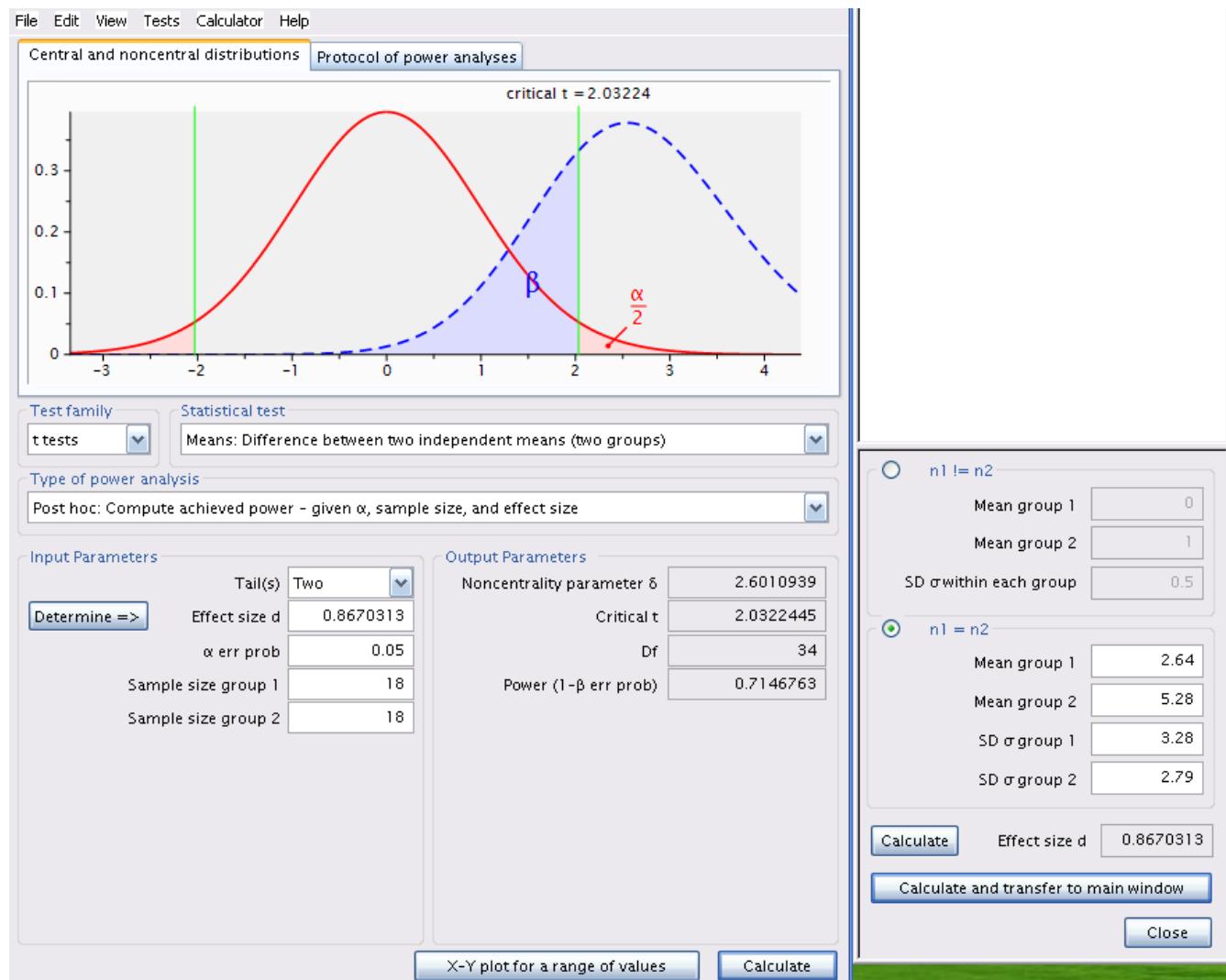


Figure 8.3.: Post-hoc analysis using an effect size different from the one estimated

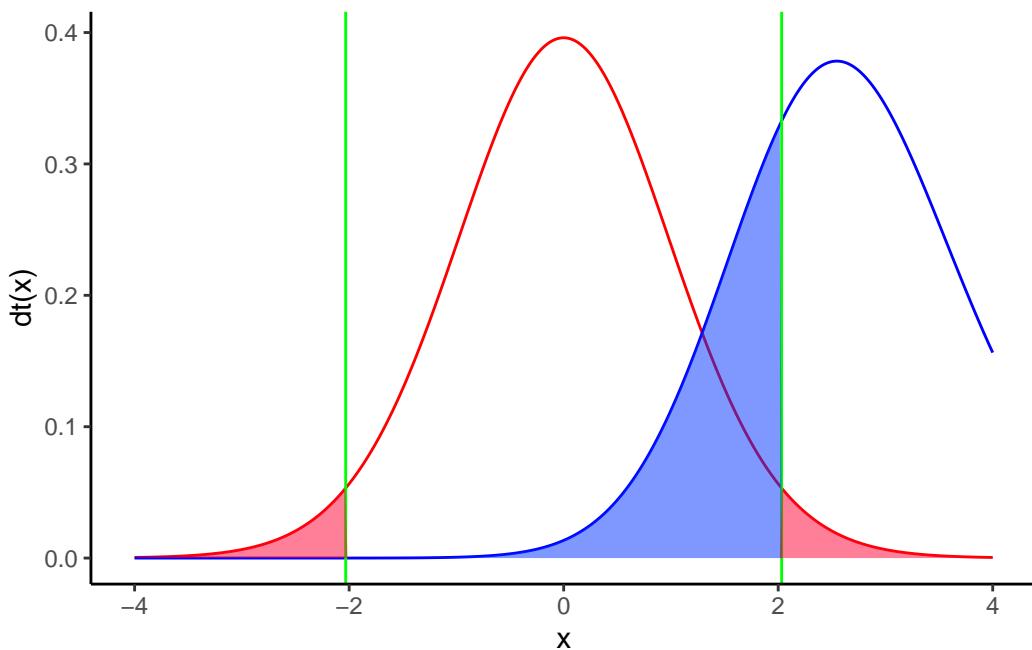


Figure 8.4.

The power is 0.71, therefore Jaynie had a reasonable chance (71%) of detecting a doubling of biomass with 18 streams in each region.

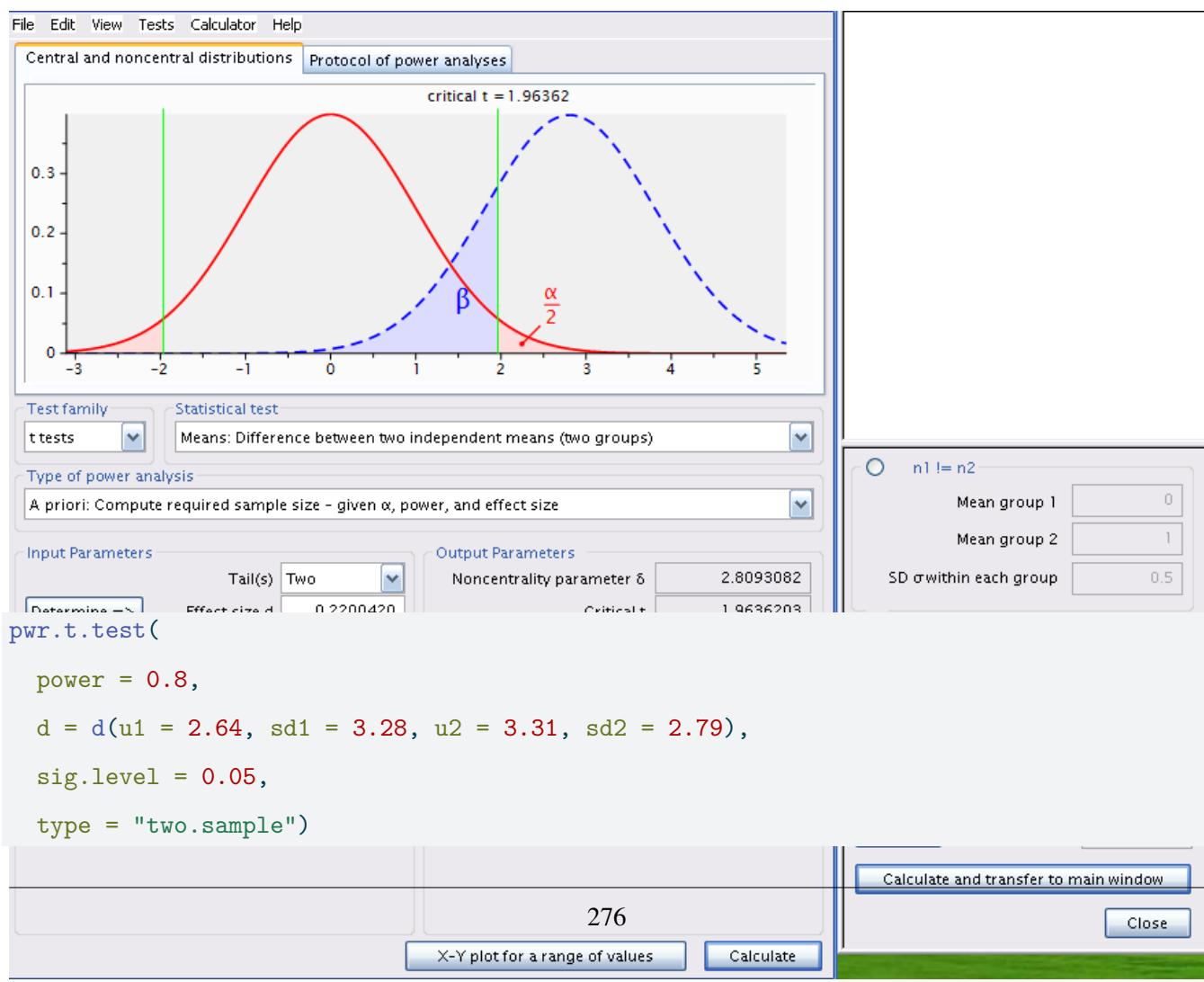
Note that this post hoc power analysis, done for an effect size considered biologically meaningful, is much more informative than the preceding one done with the observed effect size (which is what too many students do because it is the default of so many power calculation programs). Jaynie did not detect a difference between the two regions. There are two possibilities: 1) there is really no difference between the regions, or 2) the precision of measurements is so low (because the sample size is small and/or there is large variability within a region) that it is very unlikely to be able to detect even large differences. The second power analysis can eliminate this second possibility because Jaynie had 71% chances of detecting a doubling of biomass.

8.2.5. A priori analysis

Suppose that a difference in biomass of $3.31 - 2.64 = 0.67 g/m^2$ can be ecologically significant. The next field season should be planned so that Jaynie would have a good chance of detecting such a difference in fish biomass between regions. How many streams should Jaynie sample in each region to have 80% of detecting such a difference (given the observed variability)?

Exercise

Change the type of power analysis in G*Power to **A priori: Compute sample size - given α , power, and effect size**. Ensure that the values for means and standard deviations are those obtained by Jaynie. Recalculate the effect size metric and enter 0.8 for power and you will obtain Figure 8.5.



```
Two-sample t test power calculation
```

```
n = 325.1723
d = 0.220042
sig.level = 0.05
power = 0.8
alternative = two.sided
```

NOTE: n is number in *each* group

Ouch! The required sample would be of 326 streams in each region! It would cost a fortune and require several field teams otherwise only a few dozen streams could be sampled over the summer and it would be very unlikely that such a small difference in biomass could be detected. Sampling fewer streams would probably be in vain and could be considered as a waste of effort and time: why do the work on several dozens of streams if the odds of success are that low?

If we recalculate for a power of 95%, we find that 538 streams would be required from each region. Increasing power means more work!

```
pwr.t.test(
  power = 0.95,
  d = d(u1 = 2.64, sd1 = 3.28, u2 = 3.31, sd2 = 2.79),
  sig.level = 0.05,
  type = "two.sample")
```

```
Two-sample t test power calculation
```

```
n = 537.7286
d = 0.220042
sig.level = 0.05
power = 0.95
alternative = two.sided
```

NOTE: n is number in *each* group

8.2.6. Sensitivity analysis - Calculate the detectable effect size

Given the observed variability, a sampling effort of 18 streams per region, and with $\alpha = 0.05$, what effect size could Jaynie detect with 80% probability $\beta = 0.2$?

🔥 Exercise

Change analysis type in G*Power to **Sensitivity: Compute required effect size - given α , power, and sample size** and size is 18 in each region.

```
pwr.t.test(  
    power = 0.8,  
    n = 18,  
    sig.level = 0.05,  
    type = "two.sample")
```

Two-sample t test power calculation

```
n = 18  
d = 0.9612854  
sig.level = 0.05  
power = 0.8  
alternative = two.sided
```

NOTE: n is number in *each* group

The detectable effect size for this sample size, $\alpha = 0.05$ and $\beta = 0.2$ (or power of 80%) is 0.961296.

⚠️ Warning

Attention, this effect size is the metric d and is dependent on sampling variability.

Here, d is approximately equal to

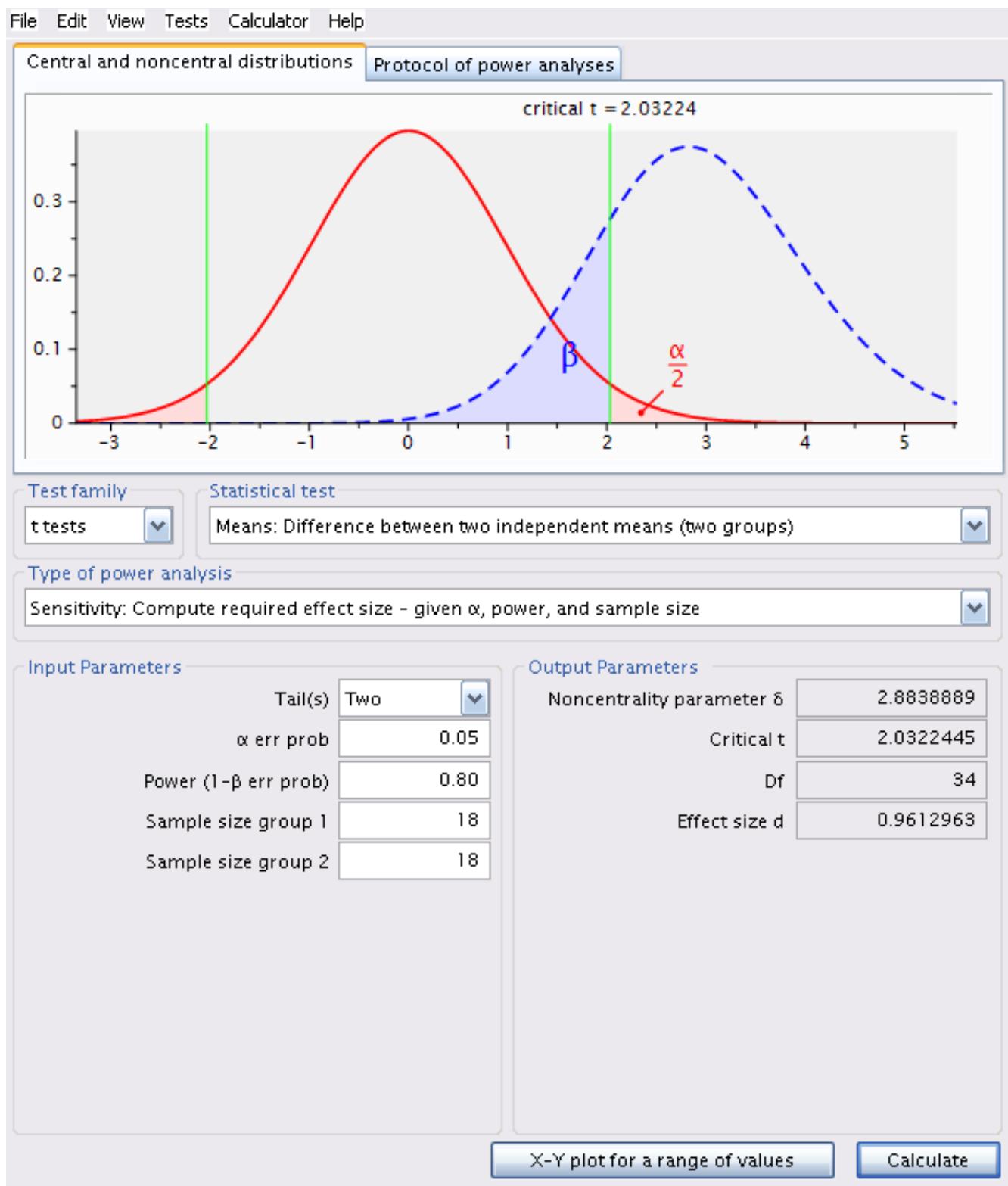


Figure 8.6.: Analyse de sensibilité

$$d = \frac{|\bar{X}_1 - \bar{X}_2|}{\sqrt{\frac{s_1^2 + s_2^2}{2}}}$$

To convert this d value without units into a value for the detectable difference in biomass between the two regions, you need to multiply d by the denominator of the equation.

$$|\bar{X}_1 - \bar{X}_2| = d * \sqrt{\frac{s_1^2 + s_2^2}{2}}$$

In R this can be done with the following code

```
pwr.t.test(  
  power = 0.8,  
  n = 18,  
  sig.level = 0.05,  
  type = "two.sample")$d * sqrt((3.28^2 + 2.79^2) / 2)
```

[1] 2.926992

Therefore, with 18 streams per region, $\alpha = 0.05$ and $\beta = 0.2$ (so power of 80%), Jaynie could detect a difference of 2.93 g/m² between regions, a bit more than a doubling of biomass.

8.3. Important points to remember

- Post hoc power analyses are relevant only when the null hypothesis is accepted because it is impossible to make a *type II error* when rejecting H_0 .
- With very large samples, power is very high and minute differences can be statistically detected, even if they are not biologically significant.
- When using a stricter significance criteria ($\alpha < 0.05$) power is reduced.
- Maximizing power implies more sampling effort, unless you use a more liberal statistical criteria ($\alpha > 0.05$)
- The choice of β is somewhat arbitrary. $\beta = 0.2$ (power of 80%) is considered relatively high by most.

Part III.

Linear models

Chapter 9

Correlation and simple linear regression

After completing this laboratory exercise, you should be able to:

- Use R to produce a scatter plot of the relationship between two variables.
- Use R to carry out some simple data transformations.
- Use R to compute the Pearson product-moment correlation between two variables and assess its statistical significance.
- Use R to compute the correlation between pairs of ranked variables using the Spearman rank correlation and Kendall's tau.
- Use R to assess the significance of pairwise comparisons from a generalized correlation matrix using Bonferroni-adjusted probabilities.
- Use R do a simple linear regression
- Use R to test the validity of the assumptions underlying simple linear regression
- Use R to assess significance of a regression by the bootstrap method
- Quantify effect size in simple regression and perform a power analysis using G*Power

9.1. R packages and data

For this lab you need:

- R packages:
 - car
 - lmtest
 - boot

- pwr
 - ggplot
 - performance
- data:
 - sturgeon.csv

You need to load the packages in R with `library()` and if needed install them first with `install.packages()`. For the data, load them using the `read.csv()` function.

```
library(car)
library(lmtest)
library(performance)
library(boot)
library(ggplot2)
library(pwr)

sturgeon <- read.csv("data/sturgeon.csv")
```

i Note

Note that the command to read the data assumes that the data file is in a folder named `data` within the working directory. Adjust as needed.

9.2. Scatter plots

Correlation and regression analysis should always begin with an examination of the data: this is a critical first step in determining whether such analyses are even appropriate for your data. Suppose we are interested in the extent to which length of male sturgeon in the vicinity of The Pas and Cumberland House covaries with weight. To address this question, we look at the correlation between `fklnght` and `rdwght`. Recall that one of the assumptions in correlation analysis is that the relationship between the two variables is linear. To evaluate this assumption, a good first step is to produce a scatterplot.

- Load the data from `sturgeon.csv` in an `obj=jct` named `sturgeon`. Make a scatter plot of `rdwght` vs `fklnght` fit with a locally weighted regression (Loess) smoother, and a linear regression line.

```
sturgeon <- read.csv("data/sturgeon.csv")
str(sturgeon)
```

```
'data.frame': 186 obs. of 9 variables:
 $ fklnghth : num 37 50.2 28.9 50.2 45.6 ...
 $ totlngth: num 40.7 54.1 31.3 53.1 49.5 ...
 $ drlngth : num 23.6 31.5 17.3 32.3 32.1 ...
 $ rdwgght : num 15.95 NA 6.49 NA 29.92 ...
 $ age      : int 11 24 7 23 20 23 20 7 23 19 ...
 $ girth    : num 40.5 53.5 31 52.5 50 54.2 48 28.5 44 39 ...
 $ sex      : chr "MALE" "FEMALE" "MALE" "FEMALE" ...
 $ location: chr "THE_PAS" "THE_PAS" "THE_PAS" "THE_PAS" ...
 $ year     : int 1978 1978 1978 1978 1978 1978 1978 1978 1978 ...
```

```
mygraph <- ggplot(
  data = sturgeon[!is.na(sturgeon$rdwgght), ], # source of data
  aes(x = fklnghth, y = rdwgght)
)

# plot data points, regression, loess trace
mygraph <- mygraph +
  geom_smooth(method = lm, se = FALSE, color = "green") + # add linear regression, but no SE shade
  geom_smooth(color = "red", se = FALSE) + # add loess
  geom_point() # add data points

mygraph # display graph
```

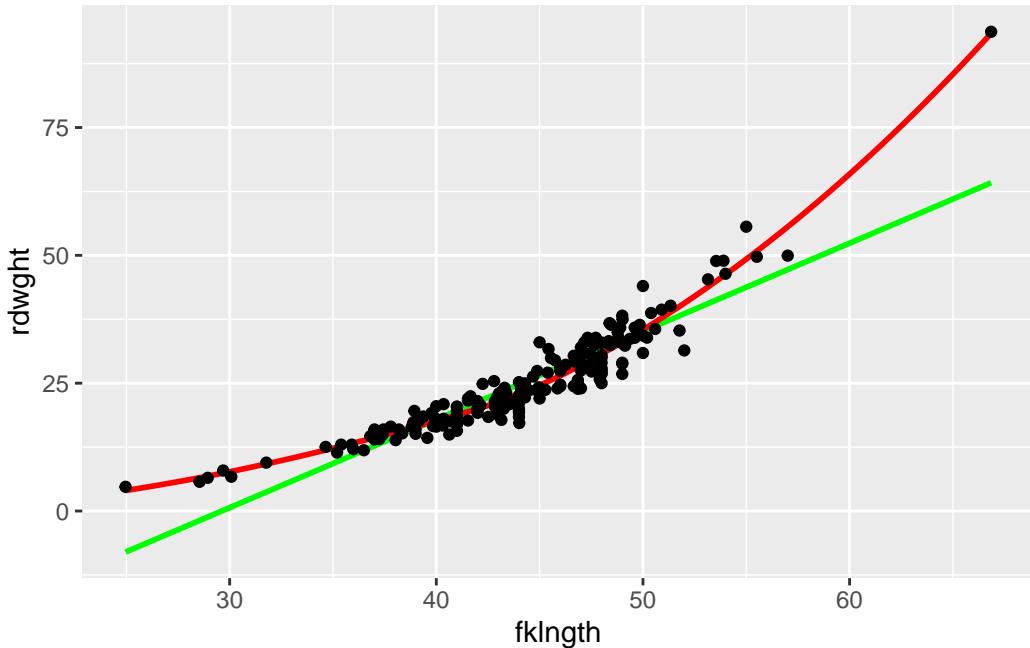


Figure 9.1.: Scatter plot of Weight as a function of length in sturgeons

- Does this curve suggest a good correlation between the two? Based on visual inspection, does the relationship between these two variables appear linear?

There is some evidence of nonlinearity, as the curve appears to have a positive second derivative (concave up). This notwithstanding, it does appear the two variables are highly correlated.

- Redo the scatterplot, but after logtransformation of both axes.

```
# apply log transformation on defined graph
mygraph + scale_x_log10() + scale_y_log10()
```

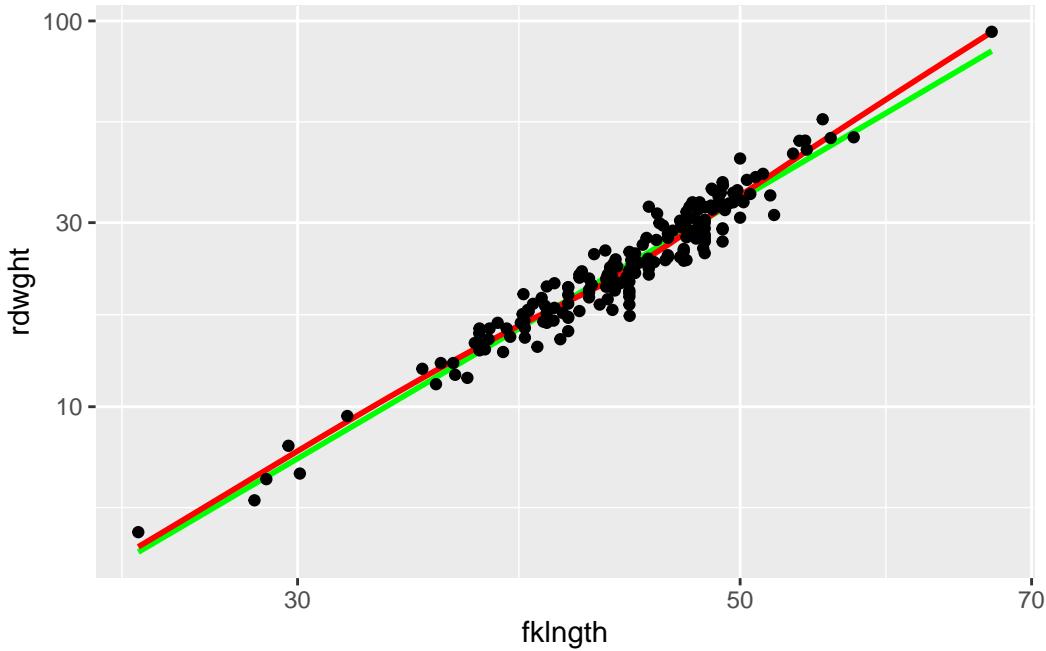


Figure 9.2.: Plot weight-length in sturgeon using a log scale

Compare the diagrams before and after the transformation (Figs Figure 9.1 and Figure 9.2). Since the relationship is more linear after transformation, correlation analysis should be done on the transformed data

9.3. Data transformations and the product-moment correlation

Recall that another assumption underlying significance testing of the product-moment correlation is that the distribution of the two variables in question is bivariate normal. We can test to see whether each of the two variables are normally distributed using the same procedures outlined in the exercise on two-sample comparisons. If the two variables are each normally distributed, then one is usually (relatively) safe in assuming the joint distribution is normal, although this needn't necessarily be true.

- Examine the distribution of the 4 variables (the two original variables and the log-transformed variables).
What do you conclude from visual inspection of these plots?

The following graph contains the 4 QQ plots (`qqplot()`). It was produced by the code below that starts with the `par()` command to ensure that all 4 plots would appear together on the same page in 2 rows and 2 columns:

```
par(mfrow = c(2, 2)) # split graph in 4 (2 rows, 2 cols) filling by rows
qqnorm(sturgeon$fklength, ylab = "fklength")
```

```

qqline(sturgeon$fklnghth)
qqnorm(log10(sturgeon$fklnghth), ylab = "log10(fklnghth)")
qqline(log10(sturgeon$fklnghth))
qqnorm(sturgeon$rdwght, ylab = "rdwght")
qqline(sturgeon$rdwght)
qqnorm(log10(sturgeon$rdwght), ylab = "log10(rdwght)")
qqline(log10(sturgeon$rdwght))
par(mfrow = c(1, 1)) # redefine plotting area to 1 plot

```

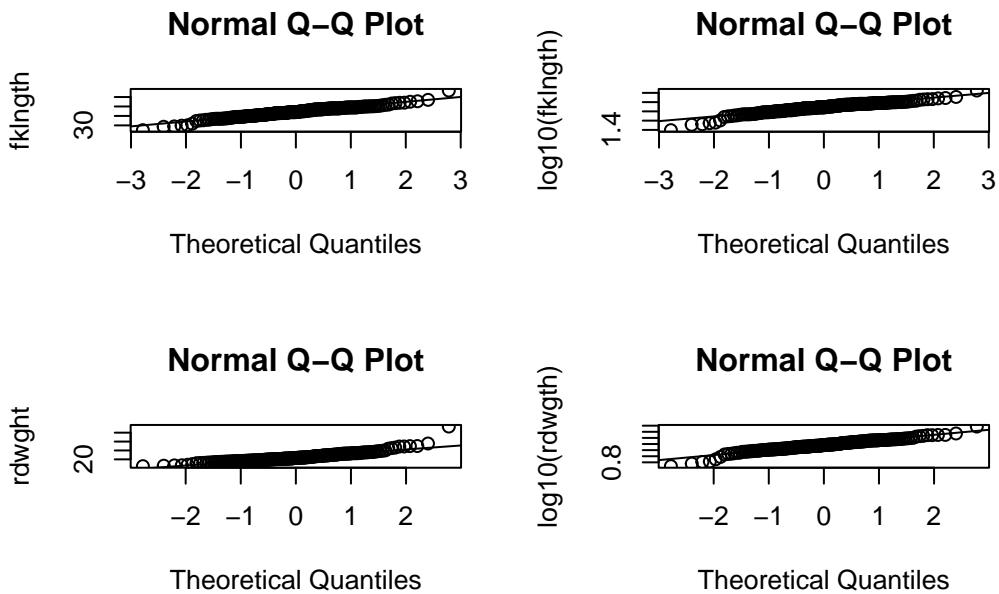


Figure 9.3.

None of these distributions are perfectly normal, but deviations are mostly minor.

- To generate a scatterplot matrix of all pairs of variables, with linear regression and lowess traces, you can use `scatterplotMatrix` from car .

```

scatterplotMatrix(
  ~ fklnghth + log10(fklnghth) + rdwght + log10(rdwght),
  data = sturgeon,
  smooth = TRUE, diagonal = "density"
)

```

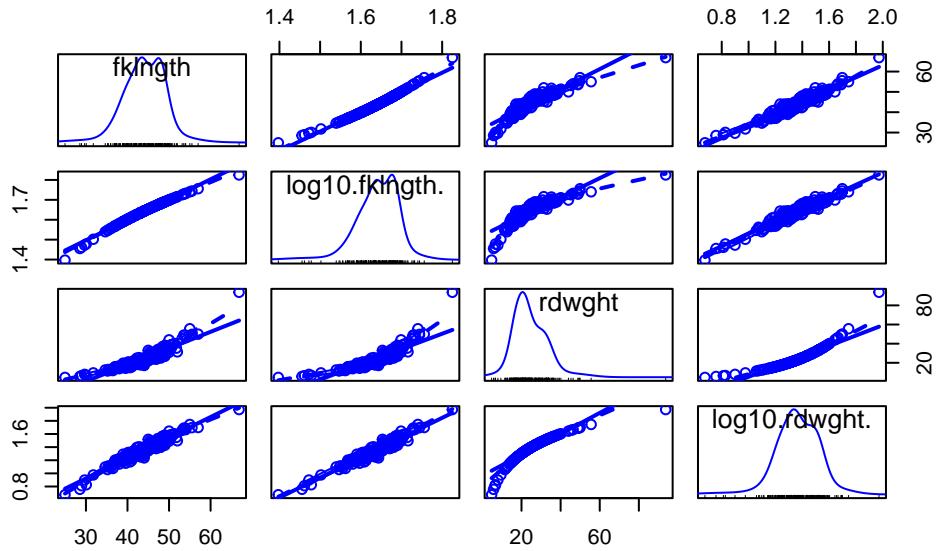


Figure 9.4.

- Next, calculate the Pearson product-moment correlation between each pair (untransformed and log transformed) using the `cor()` command. However, to do this, it will be easier if you first add your transformed data as columns in the `sturgeon` data frame.

```
sturgeon$lfklnth <- with(sturgeon, log10(fklnth))
sturgeon$lrdwght <- log10(sturgeon$rdwght)
```

Then you can get the correlation matrix by:

```
cor(sturgeon[, c("fklnth", "lfklnth", "lrdwght", "rdwght")], use = "complete.obs")
```

Note the `use="complete.obs"` parameter. It tells R to keep only lines of the data frame where all variables were measured. If there are missing data, some lines will be removed, but correlations will be calculated for the same subset of cases for all pairs of variables. One could use, instead, `use="pairwise.complete.obs"`, to tell R to only eliminate observations when values are missing for this particular pair of variables. In this situation, if there are missing values in the data frame, the sample size for pairwise correlations will vary. In general, I recommend you use the option `use="complete.obs"`, unless you have so many missing values that it eliminates the majority of your data.

- Why is the correlation between the untransformed variables smaller than between the transformed variables?

```
cor(sturgeon[, c("fklngth", "lfklnth", "lrdwght", "rdwght")], use = "complete.obs")
```

| | fklngth | lfklnth | lrdwght | rdwght |
|---------|-----------|-----------|-----------|-----------|
| fklngth | 1.0000000 | 0.9921435 | 0.9645108 | 0.9175435 |
| lfklnth | 0.9921435 | 1.0000000 | 0.9670139 | 0.8756203 |
| lrdwght | 0.9645108 | 0.9670139 | 1.0000000 | 0.9265513 |
| rdwght | 0.9175435 | 0.8756203 | 0.9265513 | 1.0000000 |

Several things should be noted here.

1. the correlation between fork length and round weight is high, regardless of which variables are used: so as might be expected, heavier fish are also longer, and vice versa
2. the correlation is greater for the transformed variables than the untransformed variables.

Why? Because the correlation coefficient is inversely proportional to the amount of scatter around a straight line. If the relationship is curvilinear (as it is for the untransformed data), the scatter around a straight line will be greater than if the relationship is linear. Hence, the correlation coefficient will be smaller.

9.4. Testing the significance of correlations and Bonferroni probabilities

It's possible to test the significance of individual correlations using the commands window. As an example, let's try testing the significance of the correlation between lfklnth and rdwght (the smallest correlation in the above table).

- In the R script window, enter the following to test the correlation between lfkgnth and rdwght :

```
cor.test(
  sturgeon$lfklnth, sturgeon$rdwght,
  alternative = "two.sided",
  method = "pearson"
)
```

Pearson's product-moment correlation

data: sturgeon\$lfklnth and sturgeon\$rdwght

```
t = 24.322, df = 180, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.8367345 0.9057199
sample estimates:
cor
0.8756203
```

We see here that the correlation is highly significant ($p < 2.2e - 16$), which is no surprise given how high the correlation coefficient is (0.8756).

It's important to bear in mind that when you are estimating correlations, the probability of finding any one correlation that is “*significant*” by pure chance increases with the number of pairwise correlations examined. Suppose, for example, that you have five variables; there are then a total of 10 possible pairwise correlations, and from this set, you would probably not be surprised to find at least one that is “*significant*” purely by chance. One way of avoiding the problem is to adjust individual α levels for pairwise correlations by dividing by the number of comparisons, k , such that: $\alpha' = \frac{\alpha}{k}$ (Bonferroni probabilities), i.e. if initially, $\alpha = 0.05$ and there are a total of 10 comparisons, then $\alpha' = 0.005$.

In the above example where we examined correlations between `fklngth` and `rdwght` and their `log`, it would be appropriate to adjust the α at which significance is tested by the total number of correlations in the matrix (in this case, 6, so $\alpha' = 0.0083$). Does your decision about the significance of the correlation between `lfklngh` and `rdwght` change?

9.5. Non-parametric correlations: Spearman's rank and Kendall's τ

The analysis done with the sturgeon data in the section above suggests that one of the assumptions of correlation, namely, bivariate normality, may not be valid for `fklngth` and `rdwght` nor for the log transforms of these variables. Finding an appropriate transformation is sometimes like looking for a needle in a haystack; indeed, it can be much worse simply because for some distributions, there is no transformation that will normalize the data. In such cases, the best option may be to go to a non-parametric analysis that does not assume bivariate normality or linearity. All such correlations are based on the ranks rather than the data themselves: two options available in R are Spearman and Kendall's τ (tau).

- Test the correlation between `fklngth` and `rdwght` using both the Spearman and Kendall's tau. The following commands will produce the correlations:

```
cor.test(
  sturgeon$lfklnth, sturgeon$rdwght,
  alternative = "two.sided",
  method = "spearman"
)
```

Spearman's rank correlation rho

```
data: sturgeon$lfklnth and sturgeon$rdwght
S = 47971, p-value < 2.2e-16
alternative hypothesis: true rho is not equal to 0
sample estimates:
rho
0.9522546
```

```
cor.test(
  sturgeon$lfklnth, sturgeon$rdwght,
  alternative = "two.sided",
  method = "kendall"
)
```

Kendall's rank correlation tau

```
data: sturgeon$lfklnth and sturgeon$rdwght
z = 16.358, p-value < 2.2e-16
alternative hypothesis: true tau is not equal to 0
sample estimates:
tau
0.8208065
```

Contrast these results with those obtained using the Pearson product-moment correlation. Why the difference?

Test the non-parametric correlations on pairs of the transformed variables. You should immediately note that the non-parametric correlations are identical for untransformed and transformed variables. This is because we are using the ranks, rather than the raw data, and the rank ordering of the data does not change when a transformation is applied to the raw values.

Note that the correlations for Kendall's tau (0.820) are lower than for the Spearman rank (0.952) correlation. This is because Kendall's gives more weight to ranks that are far apart, whereas Spearman's weights each rank equally. Generally, Kendall's is more appropriate when there is more uncertainty about the reliability of close ranks.

The sturgeons in this sample were collected using nets and baited hooks of a certain size. What impact do you think this method of collection had on the shapes of the distributions of `fklngth` and `rdwght`? Under these circumstances, do you think correlation analysis is appropriate at all?

Note that correlation analysis assumes that each variable is *randomly sampled*. In the case of sturgeon, this is not the case: baited hooks and nets will only catch sturgeon above a certain minimum size. Note that in the sample, there are no small sturgeons, since the fishing gear targets only larger fish. Thus, we should be very wary of the correlation coefficients associated with our analysis, as the inclusion of smaller fish may well change our estimate of these correlations.

9.6. Simple linear regression

In correlation analysis we are interested in how pairs of variables covary: However, in regression analysis, we are attempting to estimate a model that predicts a variable (the dependent variable) from another variable (the independent variable).

As with any statistical analysis, the best way to begin is by looking at your data. If you are interested in the relationship between two variables, say, Y and X, produce a plot of Y versus X just to get a “feel” for the relationship.

- The data file `sturgeon.csv` contains data for sturgeons collected from 1978-1980 at Cumberland House, Saskatchewan and The Pas, Manitoba. Make a scatterplot of `fklngth` (the dependent variable) versus `age` (the independent variable) for males and add a linear regression and a loess smoother. What do you conclude from this plot?

```

sturgeon.male <- subset(sturgeon, subset = sex == "MALE")

mygraph <- ggplot(
  data = sturgeon.male, # source of data
  aes(x = age, y = fklnth)
) # aesthetics: y=fklnth, x=rdwght

# plot data points, regression, loess trace
mygraph <- mygraph +
  geom_smooth(method = lm, se = FALSE, color = "green") + # add linear regression, but no SE shade
  geom_smooth(color = "red") + # add loess
  geom_point() # add data points

mygraph # display graph

```

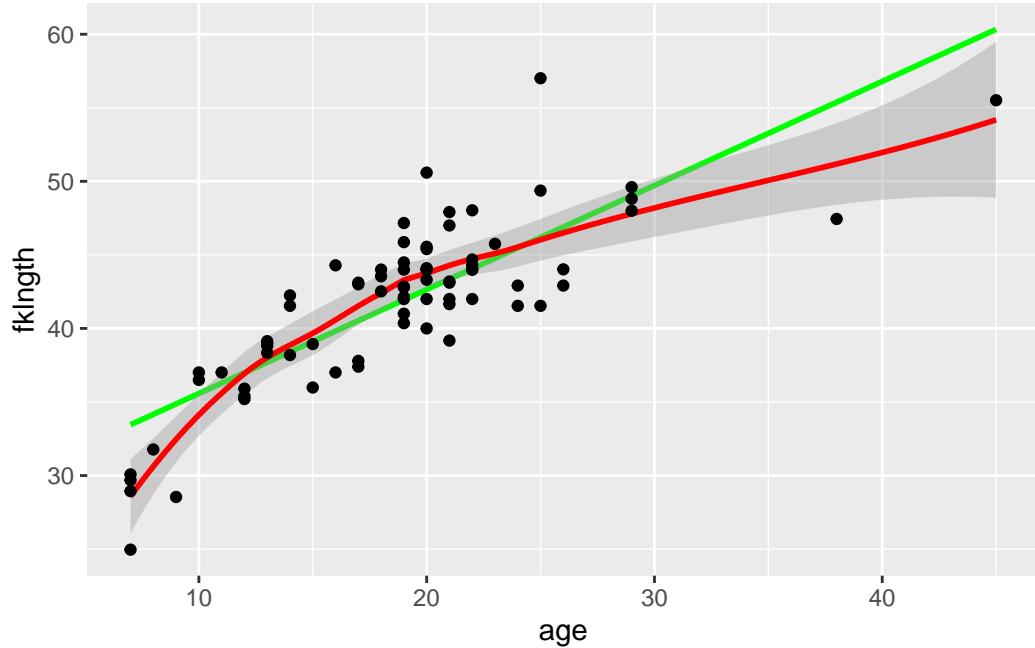


Figure 9.5.

This suggests that the relationship between age and fork length is not linear.

Suppose that we want to know the growth rate of male sturgeon. One estimate (perhaps not a very good one) of the growth rate is given by the slope of the fork length - age regression.

First, let's run the regression with the `lm()` command, and save its results in an object called `RegModel.1`.

```
RegModel.1 <- lm(fklnth ~ age, data = sturgeon.male)
```

Nothing appears on the screen, but don't worry, it all got saved in memory. To see the statistical results, type:

```
summary(RegModel.1)
```

Call:

```
lm(formula = fklnth ~ age, data = sturgeon.male)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|---------|---------|--------|--------|---------|
| -8.4936 | -2.2263 | 0.1849 | 1.7526 | 10.8234 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|----------------|----------|------------|------------|-----------------------------|
| (Intercept) | 28.50359 | 1.16873 | 24.39 | <2e-16 *** |
| age | 0.70724 | 0.05888 | 12.01 | <2e-16 *** |
| --- | | | | |
| Signif. codes: | 0 | '***' | 0.001 '**' | 0.01 '*' 0.05 '.' 0.1 ' ' 1 |

Residual standard error: 3.307 on 73 degrees of freedom

(5 observations deleted due to missingness)

Multiple R-squared: 0.664, Adjusted R-squared: 0.6594

F-statistic: 144.3 on 1 and 73 DF, p-value: < 2.2e-16

R output gives you:

1. **Call:** A friendly reminder of the model fitted and the data used.
2. **Residuals:** General statistics about the residuals around the fitted model.
3. **Coefficients:** Fitted model parameter estimates, standard errors, t values and associated probabilities.
4. **Residual standard error:** Square root of the residual variance.
5. **Multiple R-squared:** Coefficient of determination. It corresponds to the proportion of the total variance of the dependent variable that is accounted for by the regression (i.e. by the independent variable)

6. **Adjusted R-squared:** The adjusted R-squared accounts for the number of parameters in the model. If you want to compare the performance of several models with different numbers of parameters, this is the one to use
7. **F-statistic:** This is the test of the overall significance of the model. In the simple regression case, this is the same as the test of the slope of the regression.

The estimated regression equation is therefore:

$$Fklength = 28.50359 + 0.70724 * age$$

Given the highly significant F-value of the model (or equivalently the highly significant t-value for the slope of the line), we reject the null hypothesis that there is no relationship between fork length and age.

9.6.1. Testing regression assumptions

Simple model I regression makes four assumptions:

1. the X variable is measured without error;
2. the relationship between Y and X is linear;
3. that for any value of X, the Y's are independently and normally distributed;
4. the variance of Y for fixed X is independent of X.

Having done the regression, we can now test the assumptions. For most biological data, the first assumption is almost never valid; usually there is error in both Y and X. This means that in general, slope estimates are biased, but predicted values are unbiased. However, so long as the error in X is small relative to the range of X in your data, the fact that X has an associated error is not likely to influence the outcome dramatically. On the other hand, if there is substantial error in X, regression results based on a model I regression may give poor estimates of the functional relationship between Y and X. In this case, more sophisticated regression procedures must be employed which are, unfortunately, beyond the scope of this course.

The other assumptions of a model I regression can, however, be tested, or at least evaluated visually. The `plot()` command can display diagnostics for linear models.

```
par(mfrow = c(2, 2), las = 1)
plot(RegModel.1)
```

The `par()` command is used here to tell R to display 2 rows and 2 columns of graphs per page (there are 4 diagnostic graphs for linear models generated automatically), and the last statement is to tell R to rotate the labels of the Y axis so that they are perpendicular to the Y axis. (Yes, I know, this is not at all obvious.)

You will get:

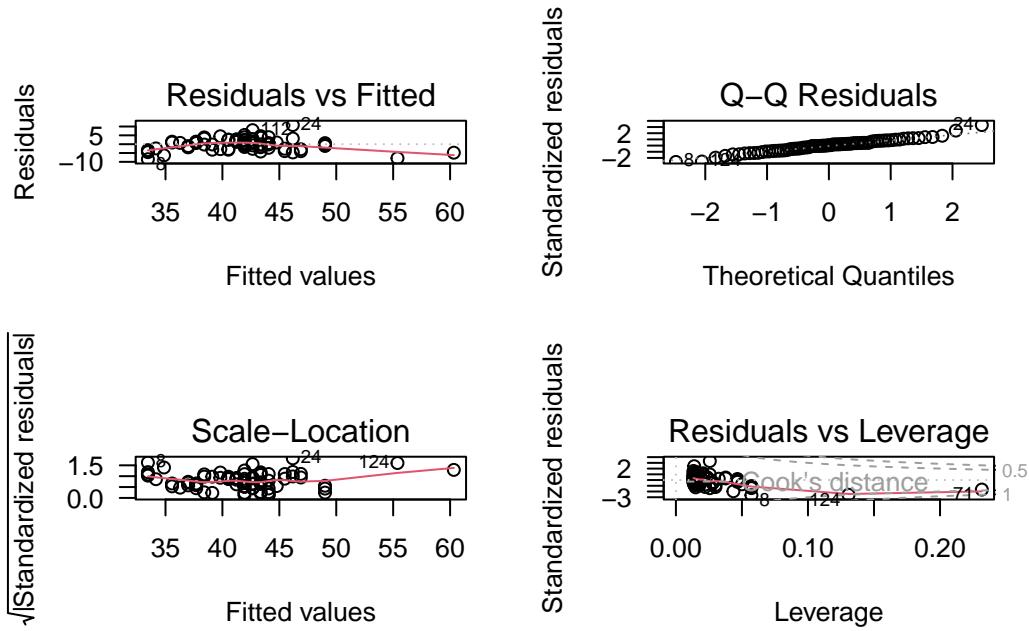


Figure 9.6.

1. **Upper left** tell you about linearity, normality, and homoscedasticity of the residuals. It shows the deviations around the regression vs the predicted values. Remember that the scatterplot (`fklngth` vs `age`) suggested that the relationship between fork length and age is not linear. Very young and very old sturgeons tended to fall under the line, and fish of average age tended to be a bit above the line. This is exactly what the residual vs fitted plot shows. The red line is a lowess trace through these data. If the relationship was linear, it would be approximately flat and close to 0. The scatter of residuals tells you a bit about their normality and homoscedasticity, although this graph is not the best way to look at these properties. The next two are better.
2. **Upper right** is to assess the normality of the residuals. It is a QQ plot of the residuals . If the residuals were normally distributed, they would fall very close to the diagonal line. Here, we see it is mostly the case, except in the tails
3. **Bottom left** titled Scale-Location, helps with assessing homoscedasticity. It plots the square root of the absolute value of the standardized residual (residual divided by the standard error of the residuals, this scales the residuals so that their variance is 1) as a function of the fitted value. This graph can help you visualize whether the spread of the residuals is constant or not. If residuals are homoscedastic, then the average will not

change with increasing fitted values. Here, there is slight variability, but it is not monotonous (i.e. it does not increase or decrease systematically) and there is no strong evidence against the assumption of homoscedasticity.

4. **Bottom right** plots the residuals as a function of leverage and can help detecting the presence of outliers or points that have a very strong influence on the regression results. The leverage of a point measures how far it is from the other points, but only with respect to the independent variable. In the case of simple linear regression, it is a function of the difference between the observation and the mean of the independent variable. You should look more closely at any observation with a leverage value that is greater than: $2(k + 1)/n$, where k is the number of independent variables (here 1), and n is the number of observations. In this case there is 1 independent variable, 75 observations, and points with a leverage higher than 0.053 may warrant particular scrutiny. The plot also gives you information about how the removal of a point from the data set would change the predictions. This is measured by the Cook's distance, illustrated by the red lines on the plot. A data point with a Cook distance larger than 1 has a large influence.

Warning

Note that R automatically labels the most extreme cases on each of these 4 plots. It does not mean that these cases are outliers, or that you necessarily need be concerned with them. In any data set, there will always be a minimum and a maximum residual.

The R package `performance` offers a new and updated version of those graphs with colours and more plots to help visually assess the assumptions with the function `model_check()`

```
check_model(RegModel.1)
```

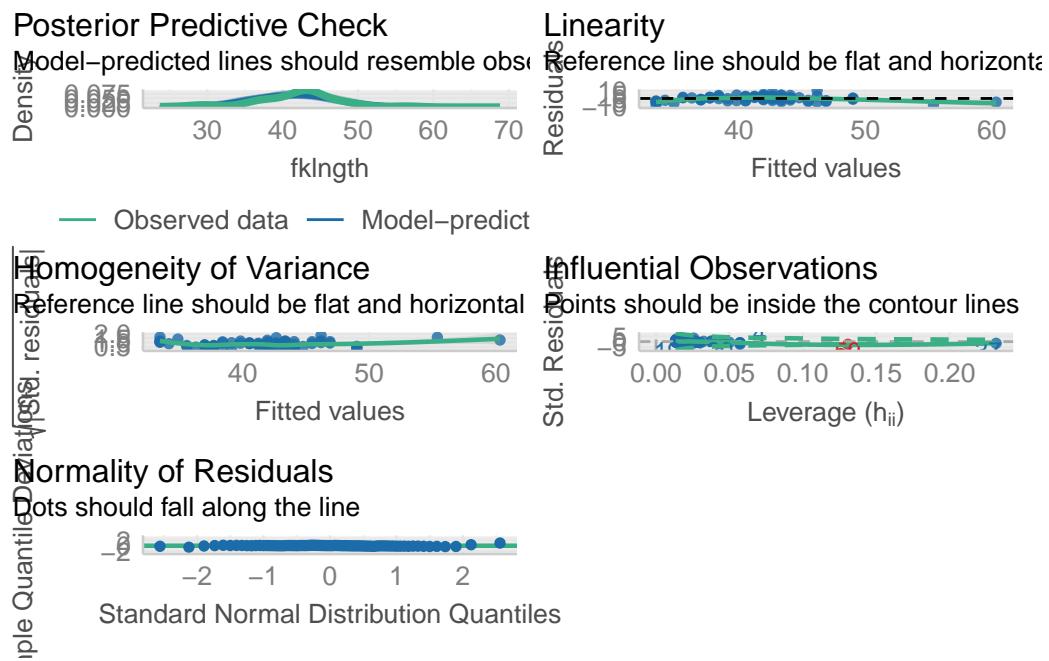


Figure 9.7.

So, what is the verdict about the linear regression between `fklngth` and `age`? It fails the linearity, possibly fails the normality, passes homoscedasticity, and this does not seem to be too strongly affected by some bizarre points.

9.6.2. Formal tests of regression assumptions

In my practice, I seldom use formal tests of regression assumptions and mostly rely on graphs of the residuals to guide my decisions. To my knowledge, this is what most biologists and data analysts do. However, in my early analyst life I was not always confident that I was interpreting these graphs correctly and wished that I had a formal test or a statistic quantifying the degree of deviation from the regression assumptions.

The `lmtest` R package, not part of the base R installation, but available from CRAN, contains a number of tests for linearity and homoscedasticity. And one can test for normality using the Shapiro-Wilk test seen previously.

First, you need to load (and maybe install) the `lmtest` package.

```
library(lmtest)
```

Exercise

Run the following commands

```
bptest(RegModel.1)
```

studentized Breusch-Pagan test

```
data: RegModel.1  
BP = 1.1765, df = 1, p-value = 0.2781
```

The Breusch-Pagan test examines whether the variability of the residuals is constant with respect to increasing fitted values. A low p value is indicative of heteroscedasticity. Here, the p value is high, and supports my visual assessment that the homoscedasticity assumption is met by these data.

```
dwtest(RegModel.1)
```

Durbin-Watson test

```
data: RegModel.1  
DW = 2.242, p-value = 0.8489  
alternative hypothesis: true autocorrelation is greater than 0
```

The Durbin-Watson test can detect serial autocorrelation in the residuals. Under the assumption of no autocorrelation, the D statistic is 2. This test can detect violation of independence of observations (residuals), although it is not foolproof. Here there is no problem identified.

```
resettest(RegModel.1)
```

RESET test

```
data: RegModel.1  
RESET = 14.544, df1 = 2, df2 = 71, p-value = 5.082e-06
```

The RESET test is a test of the assumption of linearity. If the linearity assumption is met, the RESET statistic will be close to 1. Here, the statistic is much larger (14.54), and very highly significant. This confirms our visual assessment that the relationship is not linear.

```
shapiro.test(residuals(RegModel.1))
```

```
Shapiro-Wilk normality test
```

```
data: residuals(RegModel.1)
W = 0.98037, p-value = 0.2961
```

The Shapiro-Wilk normality test on the residual confirms that the deviation from normality of the residuals is not large.

9.7. Data transformations in regression

The analysis above revealed that the linearity assumption underlying regression analysis is not met by the `fklnghth - age` data. If we want to use regression analysis, data transformations are required:

Let's plot the log-transformed data

```
par(mfrow = c(1, 1), las = 1)
ggplot(
  data = sturgeon.male,
  aes(x = log10(age), y = log10(fklnghth))
) +
  geom_smooth(color = "red") +
  geom_smooth(method = "lm", se = FALSE, color = "green") +
  geom_point()
```

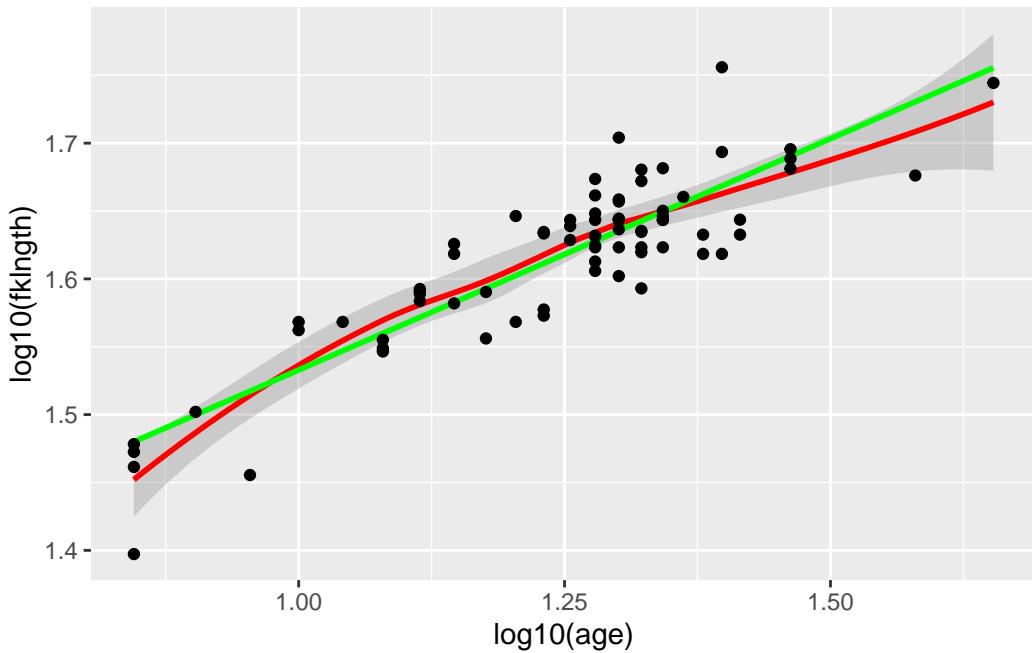


Figure 9.8.

We can fit the linear regression model on the log-transformed variables.

```
RegModel.2 <- lm(log10(fklength) ~ log10(age), data = sturgeon.male)
summary(RegModel.2)
```

Call:

```
lm(formula = log10(fklength) ~ log10(age), data = sturgeon.male)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|-----------|-----------|----------|----------|
| -0.082794 | -0.016837 | -0.000719 | 0.021102 | 0.087446 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|----------------|----------|------------|----------|------------|
| (Intercept) | 1.19199 | 0.02723 | 43.77 | <2e-16 *** |
| log10(age) | 0.34086 | 0.02168 | 15.72 | <2e-16 *** |
| --- | | | | |
| Signif. codes: | 0 '***' | 0.001 '**' | 0.01 '*' | 0.05 '.' |
| | 0.1 ' ' | 1 | | |

Residual standard error: 0.03015 on 73 degrees of freedom

(5 observations deleted due to missingness)

Multiple R-squared: 0.772, Adjusted R-squared: 0.7688

F-statistic: 247.1 on 1 and 73 DF, p-value: < 2.2e-16

Note that by using the log transformed data, the proportion of variation explained by the regression has increased by 10% (from 0.664 to 0.772), a substantial increase. So the relationship has become more linear. Good. Let's look at the residual diagnostic plots:

```
par(mfrow = c(2, 2), las = 1)
plot(RegModel.2)
check_model(RegModel.2)
```

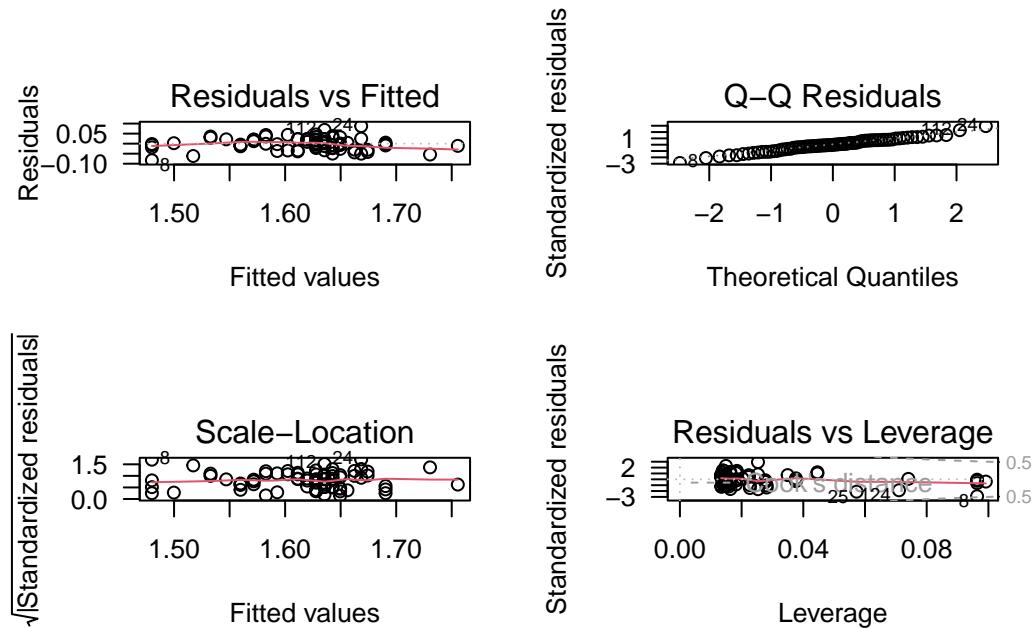


Figure 9.9.

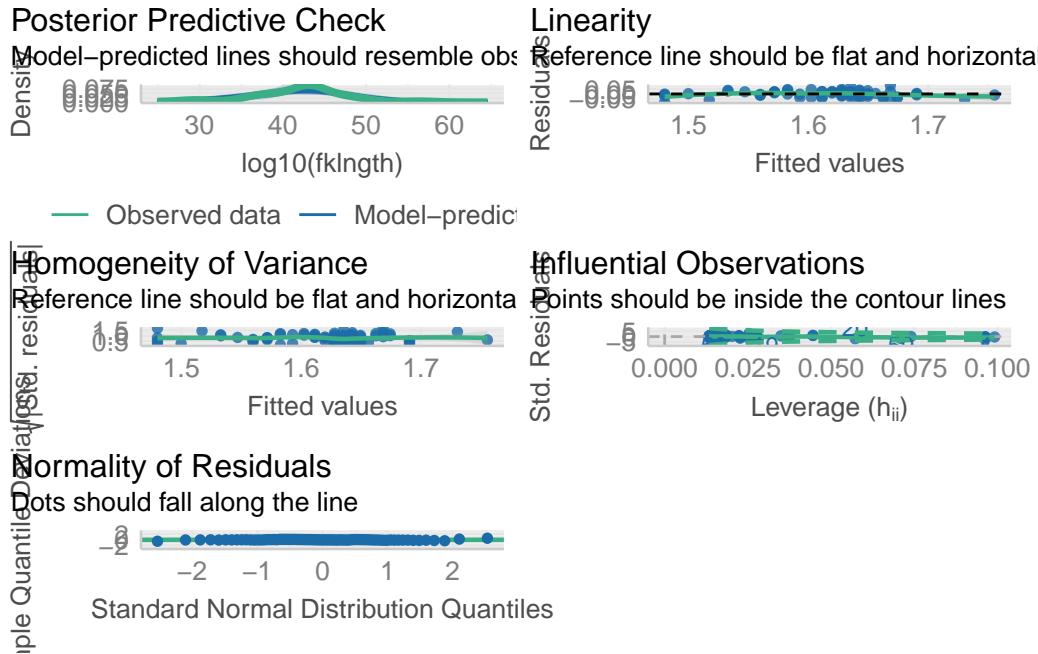


Figure 9.10.

So things appear a little better than before, although still not ideal. For example, the Residual vs fitted plot still suggests a potential nonlinearity. The QQ plot is nicer than before, indicating that residuals are more normally distributed after the log-log transformation. There is no indication of heteroscedasticity. And, although there are still a few points with somewhat high leverage, none have a Cook's distance above 0.5. It thus seems that transforming data improved things: more linear, more normal, less dependence on extreme data. Do the formal tests support this visual assessment?

```
bptest(RegModel.2)
```

```
studentized Breusch-Pagan test
```

```
data: RegModel.2
BP = 0.14282, df = 1, p-value = 0.7055
```

```
dwtest(RegModel.2)
```

```
Durbin-Watson test
```

```
data: RegModel.2  
DW = 2.1777, p-value = 0.6134  
alternative hypothesis: true autocorrelation is greater than 0
```

```
resettest(RegModel.2)
```

RESET test

```
data: RegModel.2  
RESET = 4.4413, df1 = 2, df2 = 71, p-value = 0.01523
```

```
shapiro.test(residuals(RegModel.2))
```

Shapiro-Wilk normality test

```
data: residuals(RegModel.2)  
W = 0.98998, p-value = 0.8246
```

Indeed, they do: residuals are still homoscedastic (Breusch-Pagan test), show no autocorrelation (Durbin-Watson test), are normal (Shapiro-Wilk test), and they are more linear (p value of the RESET test is now 0.015, instead of 0.000005). Linearity has improved, but is still violated somewhat.

9.8. Dealing with outliers

In this case, there are no real clear outliers. Yes, observations 8, 24, and 112 are labeled as the most extreme in the last set of residual diagnostic plots. But they are still within what I consider the “reasonable” range. But how does one define a limit to the reasonable? When is an extreme value a real outlier we have to deal with? Opinions vary about the issue, but I favor conservatism.

My rule is that, unless the value is clearly impossible or an error in data entry, I do not delete “outliers”; rather, I analyze all my data. Why? Because, I want my data to reflect natural or real variability. Indeed, variability is often what interests biologists the most.

Keeping extreme values is the fairest way to proceed, but it often creates other issues. These values will often be the main reason why the data fail to meet the assumptions of the statistical analysis. One solution is to run the analysis with and without the outliers, and compare the results. In many cases, the two analyses will be qualitatively similar: the same conclusions will be reached, and the effect size will not be very different. Sometimes, however, this comparison will reveal that the presence of the outliers changes the story. The logical conclusion then is that the results depend on the outliers and that the data at hand are not very conclusive. As an example, let's rerun the analysis after eliminating observations labeled 8, 24, and 112.

```
RegModel.3 <- lm(log10(fklnth) ~ log10(age), data = sturgeon.male, subset = !(rownames(sturgeon.
summary(RegModel.3)
```

Call:

```
lm(formula = log10(fklnth) ~ log10(age), data = sturgeon.male,
subset = !(rownames(sturgeon.male) %in% c("8", "24", "112")))
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|-----------|----------|----------|----------|
| -0.069163 | -0.017390 | 0.000986 | 0.018590 | 0.047647 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|----------------|----------|------------|---------|------------|
| (Intercept) | 1.22676 | 0.02431 | 50.46 | <2e-16 *** |
| log10(age) | 0.31219 | 0.01932 | 16.16 | <2e-16 *** |
| --- | | | | |
| Signif. codes: | 0 *** | 0.001 ** | 0.01 * | 0.05 . |
| | ' | ' | ' | ' |

Residual standard error: 0.02554 on 70 degrees of freedom

(5 observations deleted due to missingness)

Multiple R-squared: 0.7885, Adjusted R-squared: 0.7855

F-statistic: 261 on 1 and 70 DF, p-value: < 2.2e-16

The intercept, slope, and R squared are about the same, and the significance of the slope is still astronomical. Removing the “outliers” has little effect in this case.

As for the diagnostic residual plots and the formal tests of assumptions:

```
par(mfrow = c(2, 2))
plot(RegModel.3)
bptest(RegModel.3)
```

studentized Breusch-Pagan test

```
data: RegModel.3
BP = 0.3001, df = 1, p-value = 0.5838
```

```
dwtest(RegModel.3)
```

Durbin-Watson test

```
data: RegModel.3
DW = 2.0171, p-value = 0.5074
alternative hypothesis: true autocorrelation is greater than 0
```

```
resettest(RegModel.3)
```

RESET test

```
data: RegModel.3
RESET = 3.407, df1 = 2, df2 = 68, p-value = 0.0389
```

```
shapiro.test(residuals(RegModel.3))
```

Shapiro-Wilk normality test

```
data: residuals(RegModel.3)
W = 0.98318, p-value = 0.4502
```

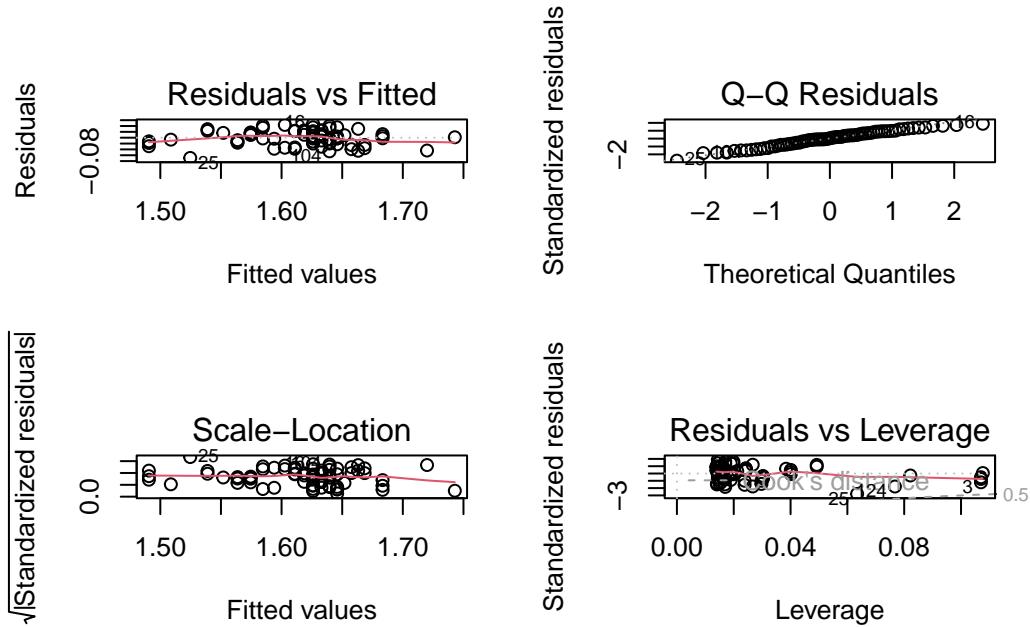


Figure 9.11.

No real difference either. Overall, this suggests that the most extreme values do not have undue influence on the results.

9.9. Quantifying effect size in regression and power analysis

Biological interpretation differs from statistical interpretation. Statistically, we conclude that size increase with age (i.e. the slope is positive and different from 0). But this conclusion alone does not tell if the difference between young and old fish is large. The slope and the scatterplot are more informative than the p-value here. The slope (in log-log space) is 0.34. This means that for each unit increase of X ($\log_{10}(\text{age})$), there is an increase of 0.34 units of $\log_{10}(\text{fkLength})$. In other words, when age is multiplied by 10, fork length is multiplied by about 2 ($10^{0.34}$). Humm, length increases more slowly than age. This slope value (0.34) is an estimate of raw effect size. It measures how much length changes with age.

It would also be important to estimate the confidence interval around the estimate of the slope. This can be obtained using the `confint()` function.

```
confint(RegModel.2)
```

```
2.5 %    97.5 %
(Intercept) 1.1377151 1.246270
log10(age)  0.2976433 0.384068
```

The 95% confidence interval for the slope is 0.29-0.38. It is quite narrow and include only values far from zero.

9.9.1. Power to detect a given slope

You can compute power with G*Power for some slope value that you deem of sufficient magnitude to warrant detection.

1. Go to **t Tests: Linear bivariate regression: One group, size of slope.**
2. Select **Post hoc: Compute achieved power- given α , sample size, and effect size**

For example, suppose that sturgeon biologists deem that a slope of 0.1 for the relationship between `log10(fklngh)` and `log10(age)` is meaningful and you wanted to estimate the power to detect such a slope with a sample of 20 sturgeons. Results from the log-log regression contain most of what you need:

```
summary(RegModel.2)
```

Call:

```
lm(formula = log10(fklngh) ~ log10(age), data = sturgeon.male)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|-----------|-----------|----------|----------|
| -0.082794 | -0.016837 | -0.000719 | 0.021102 | 0.087446 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|-------------|----------|------------|---------|------------|
| (Intercept) | 1.19199 | 0.02723 | 43.77 | <2e-16 *** |
| log10(age) | 0.34086 | 0.02168 | 15.72 | <2e-16 *** |
| <hr/> | | | | |

```
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Residual standard error: 0.03015 on 73 degrees of freedom

(5 observations deleted due to missingness)

Multiple R-squared: 0.772, Adjusted R-squared: 0.7688

F-statistic: 247.1 on 1 and 73 DF, p-value: < 2.2e-16

Note the Residual standard error value (0.03015). You will need this. The other thing you need is an estimate of the standard deviation of `log10(age)`. R can (of course) compute it. Be careful, the `sd()` function will return NA if there are missing values. You can get around this by adding `na.rm=TRUE` as an argument of the `sd()` function.

```
sd(log10(sturgeon.male$age), na.rm = TRUE)
```

```
[1] 0.1616675
```

You can then enter these values (slope to be detected, sample size, alpha, standard deviation of the independent variable) to calculate another quantity that G*Power needs (standard deviation of y) using the Determine panel. Finally you can calculate Power. The filled panels should look like this

Note

Note: The SD of y can't just be taken from the data because if the slope changes (e.g. H1) then this will change the SD of y. SD y needs to be estimated from the observed scatter around the line and the hypothesized slope.

Power to detect a significant slope, if the slope is 0.1, variability of data points around the regression is like in our sample, for a sample of 20 sturgeons, with $\alpha = 0.05$ is 0.621. Only about 2/3 of samples of that size would detect a significant effect of `age` on `fklngth`.

In R, you can do the analysis also but we will use another trick to work with the `pwr.t.test()` function. First we, need to estimate the effect size d. IN this case d is estimated as:

$$d = \frac{b}{s_b \sqrt{n - k - 1}}$$

where b is the slope, s_b is the standard error on the slope, n is the number of observations and k is the number of independent variables (1 for simple liner regression).

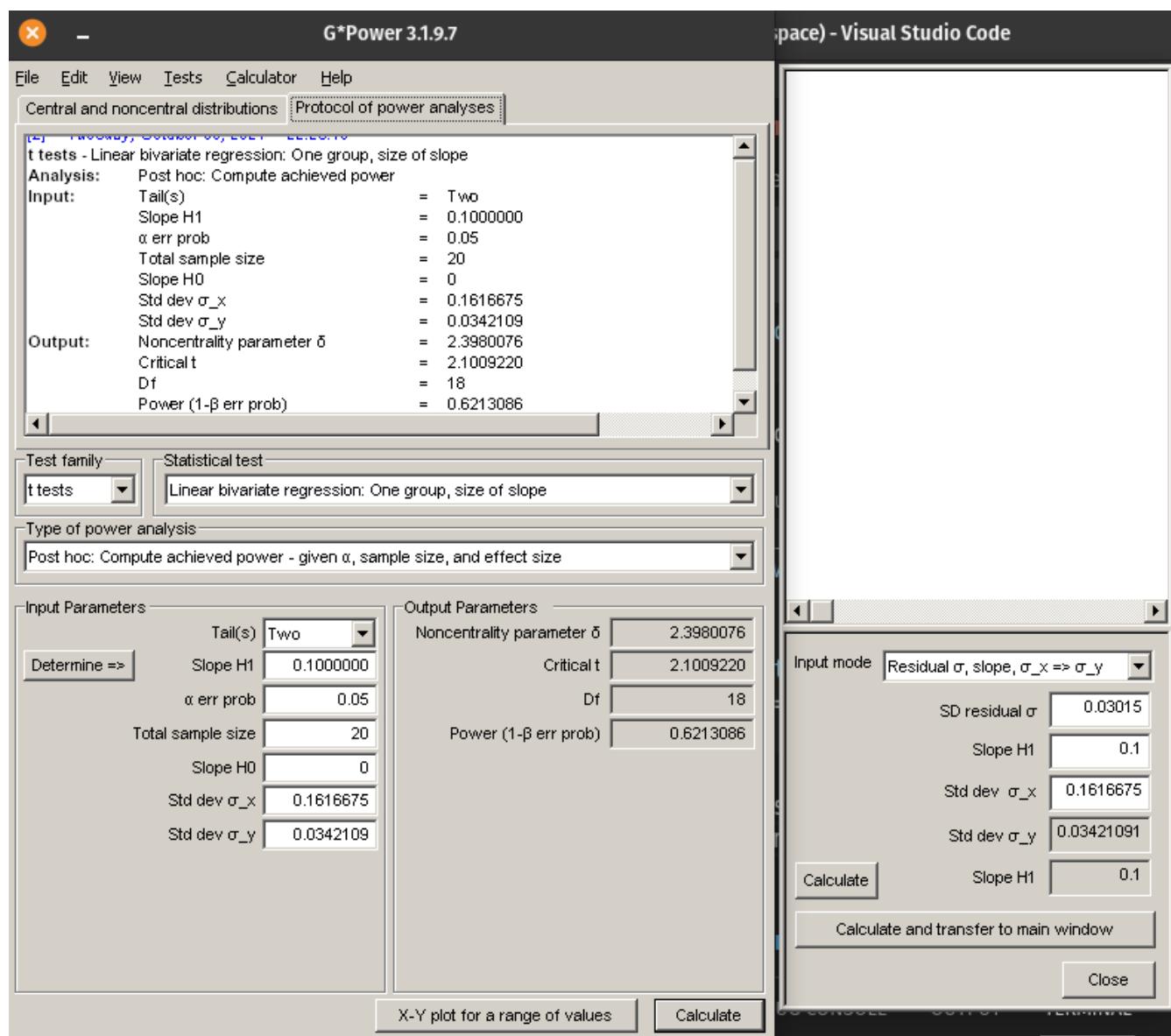


Figure 9.12.: Power analysis for age-length in sturgeon with N = 20 and slope = 0.1

SE of the slope is 0.02168. The model was fitted using 75 fishes (n=75). We can then estimate d.

$$d = \frac{b}{s_b \sqrt{n - k - 1}} = \frac{0.1}{0.02168 \sqrt{74 - 1 - 1}} = 0.54$$

We can simply use the `pwr.t.test()` function to estimate the power.

```
library(pwr)

# analyse de puissance
pwr.t.test(n = 20, d = 0.54, sig.level = 0.05, type = "one.sample")
```

```
One-sample t test power calculation
```

```
n = 20
d = 0.54
sig.level = 0.05
power = 0.6299804
alternative = two.sided
```

You can see that the results is really similar but not exactly the same than with G*power which is normal since we did not use the exact same formula to estimate power.

9.9.2. Sample size required to achieve desired power

To estimate the sample size required to achieve 99% power to detect a slope of 0.1 (in log-log space), with alpha=0.05, you simply change the type of analysis:

In R you can simply do:

```
library(pwr)

# analyse de puissance
pwr.t.test(power = 0.99, d = 0.54, sig.level = 0.05, type = "one.sample")
```

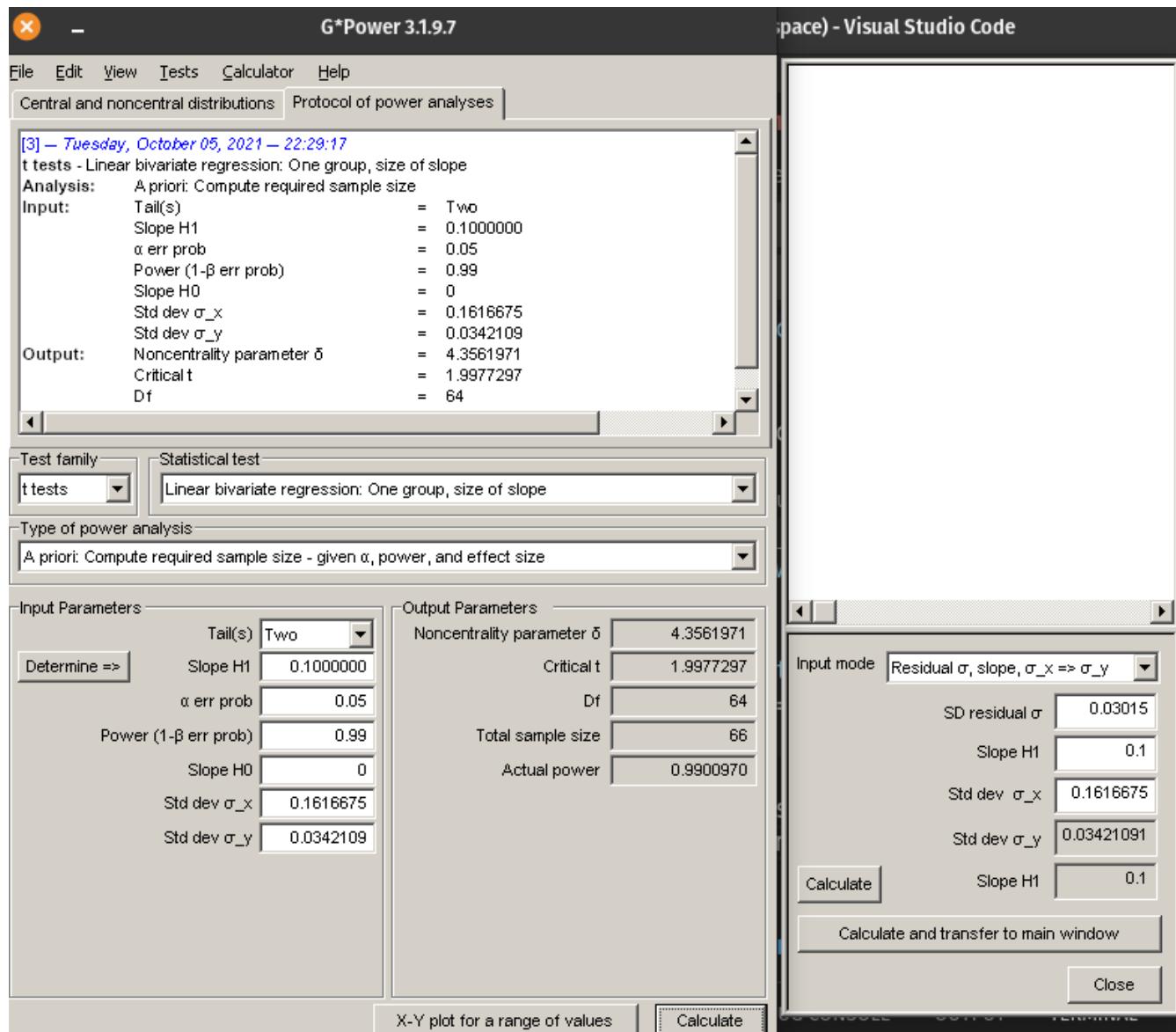


Figure 9.13.: A priori power analysis to estimate the sample size needed to have a power of 0.99

```
One-sample t test power calculation
```

```
n = 64.96719
d = 0.54
sig.level = 0.05
power = 0.99
alternative = two.sided
```

By increasing sample size to 66, with the same assumptions as before, power increases to 99%.

9.10. Bootstrapping the simple linear regression

A non-parametric test for the intercept and slope of a linear regression can be obtained by bootstrapping.

```
# load boot
library(boot)

# function to obtain regression weights
bs <- function(formula, data, indices) {
  d <- data[indices, ] # allows boot to select sample
  fit <- lm(formula, data = d)
  return(coef(fit))
}

# bootstrapping with 1000 replications
results <- boot(
  data = sturgeon.male,
  statistic = bs,
  R = 1000, formula = log10(fklngh) ~ log10(age)
)
# view results
results
```

ORDINARY NONPARAMETRIC BOOTSTRAP

Call:

```
boot(data = sturgeon.male, statistic = bs, R = 1000, formula = log10(fklength) ~  
log10(age))
```

Bootstrap Statistics :

| | original | bias | std. error |
|-----|-----------|--------------|------------|
| t1* | 1.1919926 | 0.001710115 | 0.03414993 |
| t2* | 0.3408557 | -0.001228031 | 0.02715333 |

For each parameter in the model (here the intercept is labeled $t1^*$ and the slope of the regression line is labeled $t2^*$), you obtain:

Pour chaque paramètre du modèle (ici l'ordonnée à l'origine est appelée $t1^*$ et la pente de la régression $t2^*$), R imprime :

1. `original` original parameter estimate (on all non-bootstrapped data)
2. `bias` the difference between the mean value of all bootstrap estimates and the original value
3. `std. error` standard error of the bootstrap estimate

```
par(mfrow = c(2, 2))  
plot(results, index = 1) # intercept  
plot(results, index = 2) # log10(age)
```

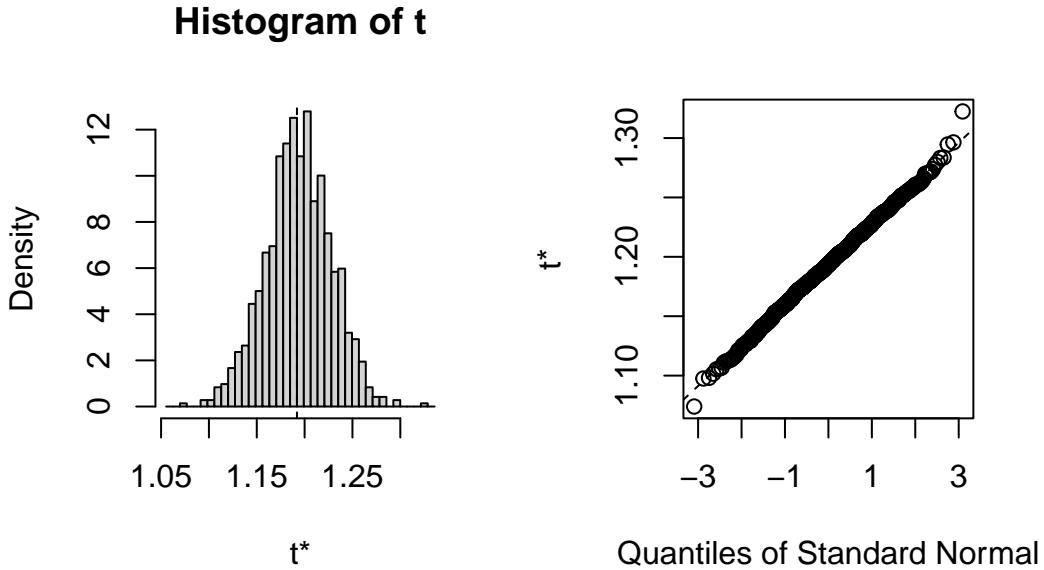


Figure 9.14.

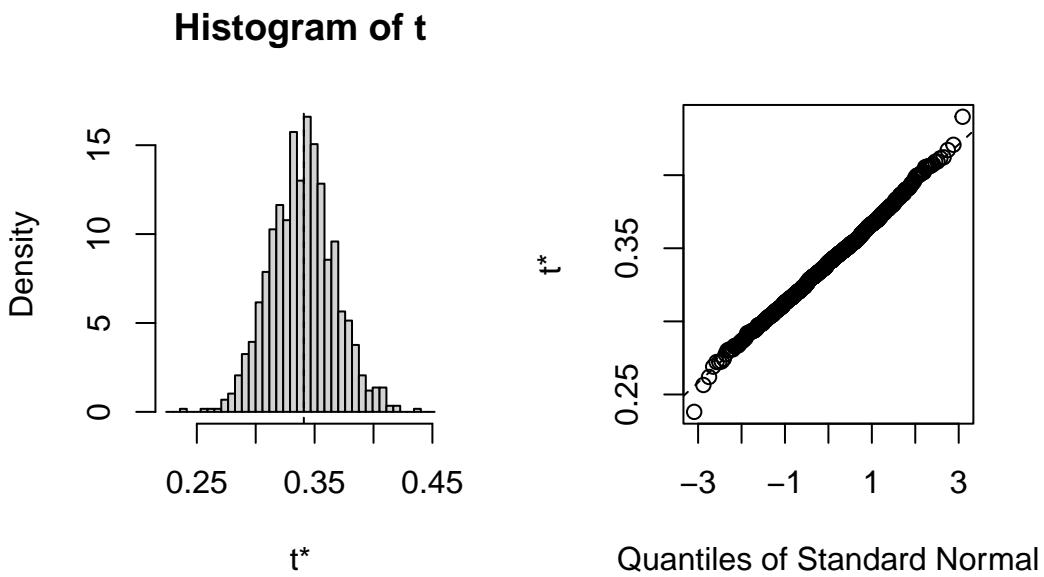


Figure 9.15.

The distribution of the bootstrapped estimates is rather Gaussian, with only small deviations in the tails (where it counts for confidence intervals...). One could use the standard error of the bootstrap estimates to calculate a symmetrical confidence interval as mean \pm SE. But, given that R can as easily calculate a bias-corrected adjusted (BCa) confidence interval, or one based on the actual distribution, (Percentile) why not have it do it all:

```
# interval de confiance pour l'ordonnée à l'origine  
boot.ci(results, type = "all", index = 1)
```

Warning in boot.ci(results, type = "all", index = 1): bootstrap variances
needed for studentized intervals

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS

Based on 1000 bootstrap replicates

CALL :

```
boot.ci(boot.out = results, type = "all", index = 1)
```

Intervals :

| Level | Normal | Basic |
|-------|------------------|------------------|
| 95% | (1.123, 1.257) | (1.126, 1.259) |

| Level | Percentile | BCa |
|-------|------------------|------------------|
| 95% | (1.125, 1.258) | (1.113, 1.252) |

Calculations and Intervals on Original Scale

```
# intervalle de confiance pour la pente  
boot.ci(results, type = "all", index = 2)
```

Warning in boot.ci(results, type = "all", index = 2): bootstrap variances
needed for studentized intervals

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS

Based on 1000 bootstrap replicates

CALL :

```
boot.ci(boot.out = results, type = "all", index = 2)
```

Intervals :

| Level | Normal | Basic |
|-------|--------|-------|
|-------|--------|-------|

95% (0.2889, 0.3953) (0.2863, 0.3942)

| Level | Percentile | BCa |
|-------|--------------------|--------------------|
| 95% | (0.2875, 0.3954) | (0.2934, 0.4027) |

Calculations and Intervals on Original Scale

Here the 4 types of CI that R managed to calculate are essentially the same. Had data been violating more strongly the standard assumptions (normality, homoscedasticity), then the different methods (Normal, Basic, Percentile, and BCa) would have diverged more. In that case, which one is best? BCa has the favor of most, currently.

Chapter 10

Two - sample comparisons

After completing this laboratory exercise, you should be able to:

- Use R to visually examine data.
- Use R to compare the means of two normally distributed samples.
- Use R to compare the means of two non-normally distributed samples.
- Use R to compare the means of two paired samples

10.1. R packages and data

For this lab you need:

- R packages:
 - car
 - lmtest
 - boot
 - pwr
 - ggplot2
 - performance
 - lmPerm
- data:
 - sturgeon.csv
 - skulldat_2020.csv

You need to load the packages in R with `library()` and if need needed install them first with `install.packages()`. For the data, load them using the `read.csv()` function.

10.2. Visual examination of sample data

One of the first steps in any type of data analysis is to visualize your data with plots and summary statistics, to get an idea of underlying distributions, possible outliers, and trends in your data. This often begins with plots of the data, such as histograms, probability plots, and box plots, that allow you to get a feel for whether your data are normally distributed, whether they are correlated one to the other, or whether there are any suspicious looking points that may lead you to go back to the original data file to check for errors.

Suppose we want to test the null hypothesis that the size, as indexed by fork length (`fklngth` in file `sturgeon.csv` - the length, in cm, from the tip of the nose to the base of the fork in the caudal fin), of sturgeon at The Pas and Cumberland House is the same. To begin, we have a look at the underlying distributions of the sample data to get a feel for whether the data are normally distributed in each sample. We will not actually test for normality at this point; the assumption of normality in parametric analyses refers always to the residuals and not the raw data themselves. However, if the raw data are non-normally distributed, then you usually have good reason to suspect that the residuals also will be non-normally distributed.

An excellent way to visually compare a data distribution to a normal distribution is to superimpose a histogram of the data and a normal curve. To do so, we must proceed in two steps:

1. tell R that we want to make a histogram with a density curve superimposed
 2. tell R that we want this to be done for both locations.
- Using the data file `sturgeon.csv`, generate histograms for `fklngth` data at The Pas and Cumberland House.

```
# use "sturgeon" dataframe to make plot called mygraph
# and define x axis as representing fklngth
mygraph <- ggplot(
  data = sturgeon,
  aes(x = fklngth)
) +
  xlab("Fork length (cm)")
# add data to the mygraph ggplot
mygraph <- mygraph +
```

```

geom_density() + # add data density smooth
geom_rug() + # add rug (bars at the bottom of the plot)
geom_histogram( # add black semitransparent histogram
  aes(y = ..density..),
  color = "black", alpha = 0.3
) +
# add normal curve in red, with mean and sd from fklngth
stat_function(
  fun = dnorm,
  args = list(
    mean = mean(sturgeon$fklngth),
    sd = sd(sturgeon$fklngth)
  ),
  color = "red"
)
# display graph, by location
mygraph + facet_grid(. ~ location)

```

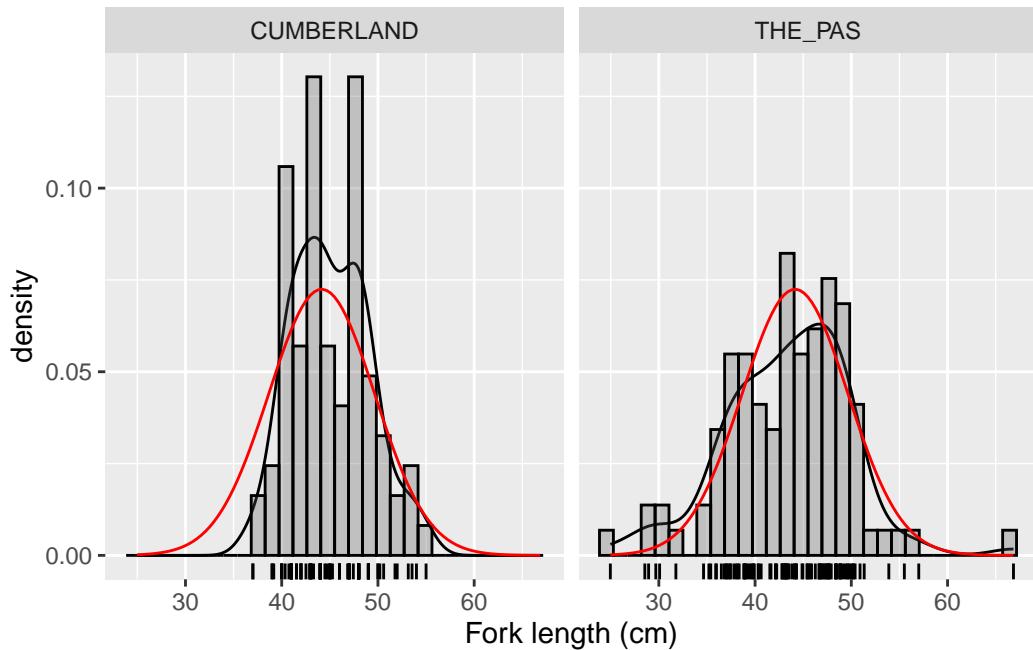


Figure 10.1.: Distribution of sturgeon length at 2 locations

Based on your visual inspection, are the two samples normally distributed? Visual inspection of these plots suggests

that this variable is approximately normally distributed in each sample.

Since we are interested in finding out if mean fish size differs among the two locations, it is probably also a good idea to generate a graph that compares the two groups of data. A box plot works well for this.

- Generate a box plot of `fklngth` grouped by `location`. What do you conclude about differences in size among the two locations?

```
ggplot(data = sturgeon, aes(
  x = location,
  y = fklngth
)) +
  geom_boxplot(notch = TRUE)
```

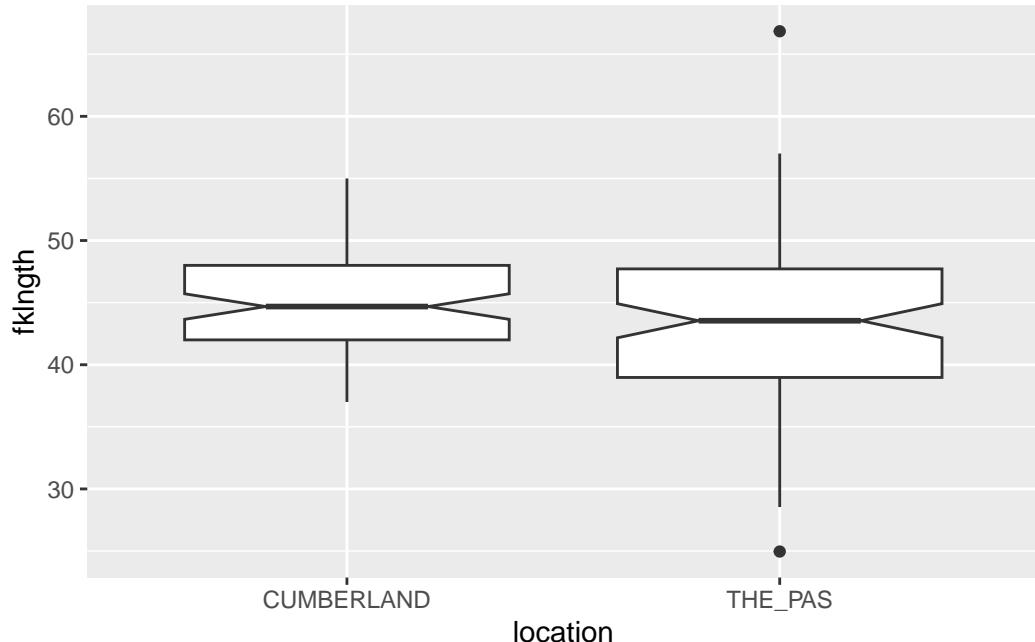


Figure 10.2.: Boxplot of sturgeon length at 2 locations

It would appear as though there are not big differences in fish size among the two locations, although fish size at The Pas looks to be more variable, with a bigger range in size and outliers (defined as values $> 1.5 * \text{inter-quartile range}$) at both ends of the distribution.

10.3. Comparing means of two independent samples: parametric and non-parametric comparisons

Test the null hypothesis that the mean fklngh of The Pas and Cumberland House samples are the same. Using 3 different tests:

1. parametric test with equal variances
2. parametric test with unequal variances
3. non-parametric test

What do you conclude?

```
# t-test assuming equal variances
t.test(
  fklngh ~ location,
  data = sturgeon,
  alternative = "two.sided",
  var.equal = TRUE
)
```

Two Sample t-test

```
data: fklngh by location
t = 2.1359, df = 184, p-value = 0.03401
alternative hypothesis: true difference in means between group CUMBERLAND and group THE_PAS is not
95 percent confidence interval:
 0.1308307 3.2982615
sample estimates:
mean in group CUMBERLAND   mean in group THE_PAS
45.08439                  43.36984
```

```
# t-test assuming unequal variances
t.test(
  fklngh ~ location,
```

```

data = sturgeon,
alternative = "two.sided",
var.equal = FALSE
)

```

Welch Two Sample t-test

```

data: fklngh by location
t = 2.2201, df = 169.8, p-value = 0.02774
alternative hypothesis: true difference in means between group CUMBERLAND and group THE_PAS is no
95 percent confidence interval:
0.1900117 3.2390804
sample estimates:
mean in group CUMBERLAND      mean in group THE_PAS
45.08439                      43.36984

```

```

# test non paramétrique
wilcox.test(
  fklngh ~ location,
  data = sturgeon,
  alternative = "two.sided"
)

```

Wilcoxon rank sum test with continuity correction

```

data: fklngh by location
W = 4973, p-value = 0.06296
alternative hypothesis: true location shift is not equal to 0

```

Based on the *t-test*, we would reject the null hypothesis, *i.e.* there is a significant (but not highly significant) difference in mean fork length between the two populations.

Note that using the Wilcoxon rank sum test, we do not reject the null hypothesis. The two different tests therefore give us two different results. The significant difference obtained using the t-test may, at least in part, be due to deviations from normality or homoscedasticity; on the other hand, the non-significant difference obtained using the U -statistic may be due to the fact that for fixed sample size, the power of a non-parametric test is lower than the corresponding parametric test. Given the p values obtained from both tests, and the fact that for samples of this size (84 and 101), the t-test is comparatively robust with respect to non-normality, I would be inclined to reject the null hypothesis. In practice to avoid P-hacking, you should decide which test is appropriate first and then apply and interpret it, or if you decide to do all you should present results of all and interpret accordingly.

Before accepting the results of the parametric t-test and rejecting the null hypothesis that there is no difference in size between the two locations, one should do some sort of assessment to determine if the model fits the assumption of normally distributed residuals and equal variances. Preliminary examination of the raw data suggested the data appeared roughly normal but there might be problems with variances (since the spread of data for The_Pas was much greater than for Cumberland). We can examine this more closely by looking at the residuals. An easy way to do so, is to fit a linear model and use the residual diagnostic plots:

```
m1 <- lm(fklength ~ location, data = sturgeon)
par(mfrow = c(2, 2))
plot(m1)
```

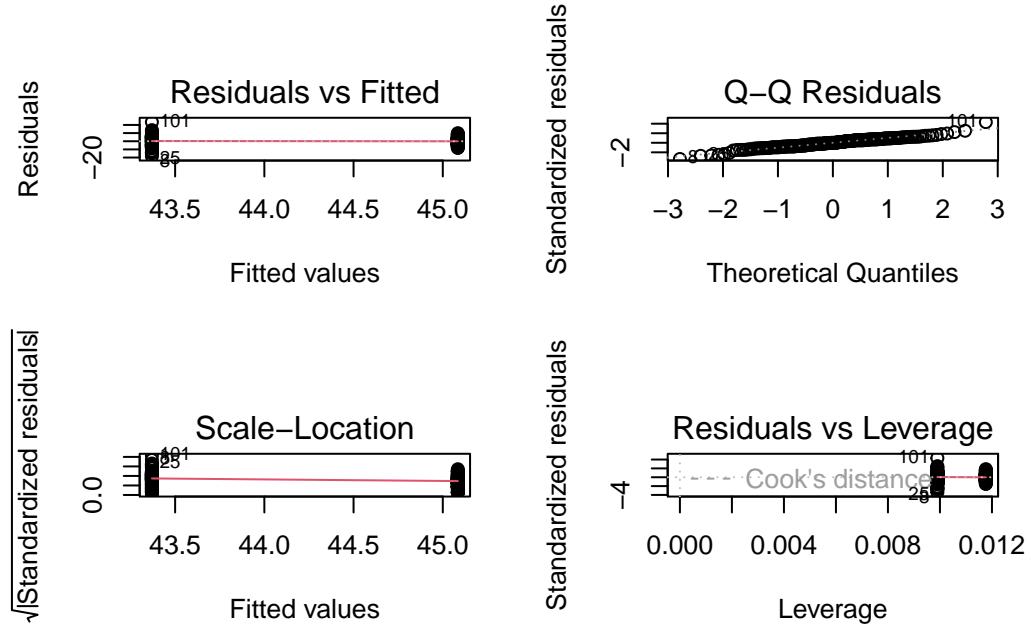


Figure 10.3.: Model assumption checks

The first plot above shows the spread of the residuals around the estimated values for the two groups and allows us to get a feel for whether there are problems with the assumption of homogeneity of variances. If the variances were equal, the vertical spread of the two clusters of points should be about the same. The above plot shows that the vertical spread of the group with the smaller mean is greater than it is for the larger mean, suggesting again that there are problems with the variances. We can test this formally by examining the mean differences in the absolute value of the residuals.

The second graph above is a normal QQ plot (or probability plot) of the residuals of the model. Note that these generally fall on a straight line, suggesting there is no real problem with normality. We can do a formal test for normality on the residuals using the Shapiro-Wilk test.

```
shapiro.test(residuals(m1))
```

```
Shapiro-Wilk normality test
```

```
data: residuals(m1)
W = 0.97469, p-value = 0.001857
```

Hummm. The test indicates that the residuals are not normal. But, given that (a) the distribution is not very far (at least visually) from normal, and that (b) the number of observations in each location is reasonably large (i.e. >30), we do not need to be overly concerned with this violation of the normality assumption.

How about equality of variance?

```
library(car)
leveneTest(m1)
```

```
Warning in leveneTest.default(y = y, group = group, ...): group coerced to
factor.
```

| | Df | F value | Pr(>F) |
|-------|-----|----------|-----------|
| group | 1 | 11.51454 | 0.0008456 |
| | 184 | NA | NA |

```
bptest(m1)
```

```
studentized Breusch-Pagan test

data: m1
BP = 8.8015, df = 1, p-value = 0.00301
```

The above are the results of two tests implemented in R (in the `car` and `lmtest` packages  that can be used to test for equal variances in t-tests or linear models involving only discontinuous or categorical independent variables.

Doing the two of them is overkill. There is not much to prefer one test over another. Levene test is possibly the better known. It tests whether the mean of absolute values of the residuals differs among groups. The Breusch-Pagan test has the advantage of being applicable to more linear models (it can deal with regression-type continuous independent variables, at least to some extent). It tests whether the studentized (i.e. scaled by their sd estimate) squared residuals vary with the independent variables in a linear model. In this case, both indicate that variances are unequal.

On the basis of these results, we conclude that there is evidence (albeit weak) to reject the null hypothesis of no difference in `fklngth` by `location`. We have modified the `t-test` to accommodate unequal variances, and are satisfied that the assumption of normally distributed residuals is sufficiently met. Thus, it appears that `fklngth` at Cumberland is greater than `fklngth` at The Pas.

10.4. Bootstrap and permutation tests to compare 2 means

10.4.1. Bootstrap

Bootstrap and permutation tests can be used to compare means (or other statistics) between pairs of samples. The general idea is simple, and it can be implemented in more ways than I can count. Here, I use existing tools and the fact that a comparison of means can be construed as a test of a linear model. We will be able to use similar code later on when we fit more complex (but fun!) models.

```
library(boot)
```

The first section defines the function that I called `bs` that simply extracts coefficients from a fitted model:

```
# function to obtain model coefficients for each iteration
bs <- function(formula, data, indices) {
  d <- data[indices, ]
  fit <- lm(formula, data = d)
  return(coef(fit))
}
```

The second section with the `boot()` command is where the real work is done: take data in `sturgeon`, bootstrap $R = 1000$ times, each time fit the model `fklngth` vs `location`, and keep the values calculated by the `bs()` function.

```
# bootstrapping with 1000 replications
results <- boot(
  data = sturgeon, statistic = bs, R = 1000,
  formula = fklngth ~ location
)
# view results
results
```

ORDINARY NONPARAMETRIC BOOTSTRAP

Call:

```
boot(data = sturgeon, statistic = bs, R = 1000, formula = fklngth ~
  location)
```

Bootstrap Statistics :

| | original | bias | std. error |
|-----|-----------|-------------|------------|
| t1* | 45.084391 | 0.01203071 | 0.4272893 |
| t2* | -1.714546 | -0.02400977 | 0.7746452 |

So we get the original estimates for the two coefficients in this model: the mean at the first (alphabetical) location,

Cumberland, and the difference in means between Cumberland and The Pas). It is the second parameter, the difference between means, which is of interest here.

```
plot(results, index = 2)
```

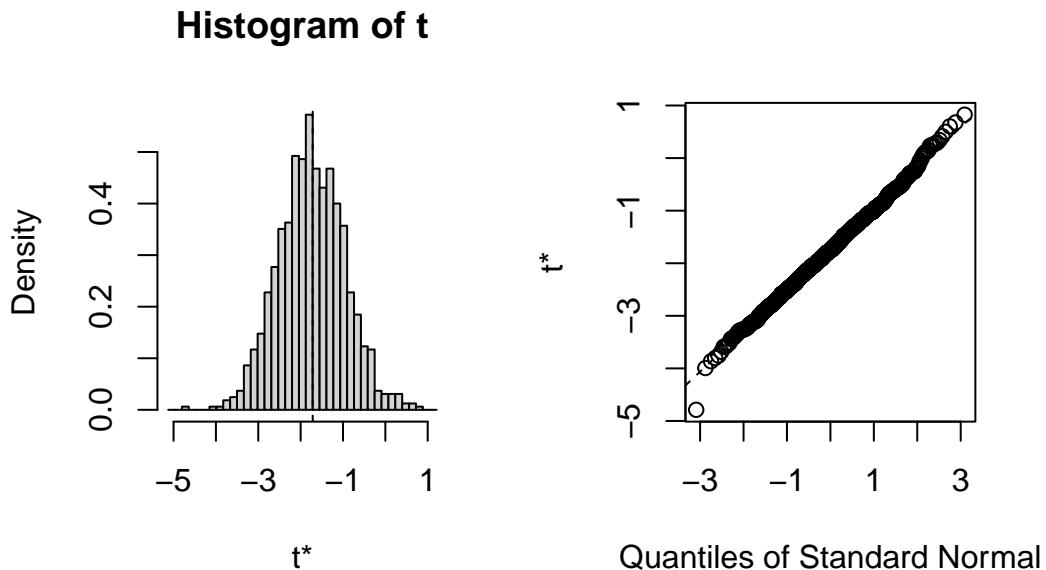


Figure 10.4.: Distribution of bootstrapped mean difference

```
# get 95% confidence intervals
boot.ci(results, type = "bca", index = 2)
```

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS

Based on 1000 bootstrap replicates

CALL :

```
boot.ci(boot.out = results, type = "bca", index = 2)
```

Intervals :

Level BCa

95% (-3.163, -0.050)

Calculations and Intervals on Original Scale

The 95% CI for the difference between the two means does not include 0. Hence, the bootstrap test indicates that the two means are not equals.

10.4.2. Permutation

Permutation tests for linear models can easily be done using the `lmPerm` package .

```
m1Perm <- lmP(
  fklngth ~ location,
  data = sturgeon,
  perm = "Prob"
)
```

```
[1] "Settings: unique SS "
```

The `lmP()` function does all the work for us. Here it is run with the option `perm` to control the stopping rule used. Option `Prob` stops the sampling when the estimated standard deviation of the p-value falls below some fraction of the estimated. It is one of many stopping rules that one could use to do permutations on a subset of all the possibilities (because it would take foreeeeever to do them all, even on your fast machine).

```
summary(m1Perm)
```

Call:

```
lmP(formula = fklngth ~ location, data = sturgeon, perm = "Prob")
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|----------|----------|---------|----------|
| -18.40921 | -3.75370 | -0.08439 | 3.76598 | 23.48055 |

Coefficients:

| | Estimate | Iter | Pr(Prob) |
|-----------|----------|------|----------|
| location1 | 0.8573 | 2326 | 0.0413 * |
| --- | | | |

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Residual standard error: 5.454 on 184 degrees of freedom

Multiple R-Squared: 0.02419, Adjusted R-squared: 0.01889

F-statistic: 4.562 on 1 and 184 DF, p-value: 0.03401

1. Iter coefficient: the Prob stopping rule stopped after 2326 iterations. Note that this number will vary each time you run this snippet of code. These are random permutation results, so expect variability.
2. Pr(Prob) coefficient: The estimated probability associated to H0 is 0.0413 . The observed difference in fklngh between the two locations was larger than the permuted differences in about (1 - 0.0413= about 95.9%) of the 2326 cases. Mind you, 2326 permutations is not a large number, so small p values can't be expected to be very precise. If it is critical that you get more precise p values, more permutations would be needed. Two parameters can be tweaked: maxIter, the maximum number of iterations (default=5000), and Ca, that stops iterations when estimated standard error of the estimated p is less than Ca*p. Default 0.1.
3. F-statistic: The rest is the standard output for the model fitted to the data, with the standard parametric test. Here the p-value, assuming all assumptions are met, is 0.034.

10.5. Comparing the means of paired samples

In some experimental designs, individuals are measured twice: common examples are the measurement of the same individual at two different times during development, or of the same individual subjected to two different experimental treatments. In these cases, the two samples are not independent (they include the same individuals), and a paired comparison must be made.

The file `skulldat_2020.csv` shows measurements of lower face width of 15 North American girls measured at age 5 and again at age 6 years (data from Newman and Meredith, 1956).

- Let's first run a standard t-test comparing the face width at age 5 and 6, not taking into account that the data are not independent and that they are consecutive measurements on the same individuals.

```
skull <- read.csv("data/skulldat_2020.csv")  
t.test(width ~ age,  
       data = skull,  
       alternative = "two.sided"  
)
```

Welch Two Sample t-test

```

data: width by age
t = -1.7812, df = 27.93, p-value = 0.08576
alternative hypothesis: true difference in means between group 5 and group 6 is not equal to 0
95 percent confidence interval:
-0.43002624 0.03002624
sample estimates:
mean in group 5 mean in group 6
7.461333      7.661333

```

So far, we specified the t-test using a formula notation as $y \sim x$ where y is the variable for which we want to compare the means and x is a variable defining the groups. This works really well when the samples are not paired and when the data is presented in a *long* format. For example theskull data is presented in a long format and contains 3 variables:

- **width**: head width for each observations
- **age**: age at measurement 5 or 6
- **id**: person identity

```
head(skull)
```

| | width | age | id |
|--|-------|-----|----|
| | 7.33 | 5 | 1 |
| | 7.53 | 6 | 1 |
| | 7.49 | 5 | 2 |
| | 7.70 | 6 | 2 |
| | 7.27 | 5 | 3 |
| | 7.46 | 6 | 3 |

When data are paired, we need to indicate how they are paired. In the skulldata, samples are paired by an individual identity, **id**, with measurement taken at different ages. However, the function **t.test** does not cope well with this

data structure. We need to transpose the data from a *long* to a *wide* format where we have a column per group, with the data of a given individual on the same line. Here is how we can do it.

```
skull_w <- data.frame(id = unique(skull$id))
skull_w$width5 <- skull$width[match(skull_w$id, skull$id) & skull$age == 5]
skull_w$width6 <- skull$width[match(skull_w$id, skull$id) & skull$age == 6]
head(skull_w)
```

| | id | width5 | width6 |
|--|----|--------|--------|
| | 1 | 7.33 | 7.53 |
| | 2 | 7.49 | 7.70 |
| | 3 | 7.27 | 7.46 |
| | 4 | 7.93 | 8.21 |
| | 5 | 7.56 | 7.81 |
| | 6 | 7.81 | 8.01 |

Now, let's run the appropriate paired t-test. What do you conclude? Compare this with the previous result and explain any differences.

```
t.test(skull_w$width5, skull_w$width6,
       alternative = "two.sided",
       paired = TRUE
)
```

Paired t-test

```
data: skull_w$width5 and skull_w$width6
t = -19.72, df = 14, p-value = 1.301e-11
alternative hypothesis: true mean difference is not equal to 0
95 percent confidence interval:
-0.2217521 -0.1782479
sample estimates:
mean difference
```

-0.2

The first analysis above assumes that the two samples of girls at age 5 and 6 are independent samples, whereas the second analysis assumes that the same girl is measured twice, once at age 5 and once at age 6 years.

Note that in the former case, we accept the null based on $p = 0.05$, but in the latter we reject the null. In other words, the appropriate (paired sample) test shows a very significant effect of age, whereas the inappropriate one does not. The reason is because there is a strong correlation between face width at age 5 and face width at age 6:

```
graphskull <- ggplot(data = skull_w, aes(x = width5, y = width6)) +
  geom_point() +
  labs(x = "Skull width at age 5", y = "Skull width at age 6") +
  geom_smooth() +
  scale_fill_continuous(low = "lavenderblush", high = "red")
graphskull
```

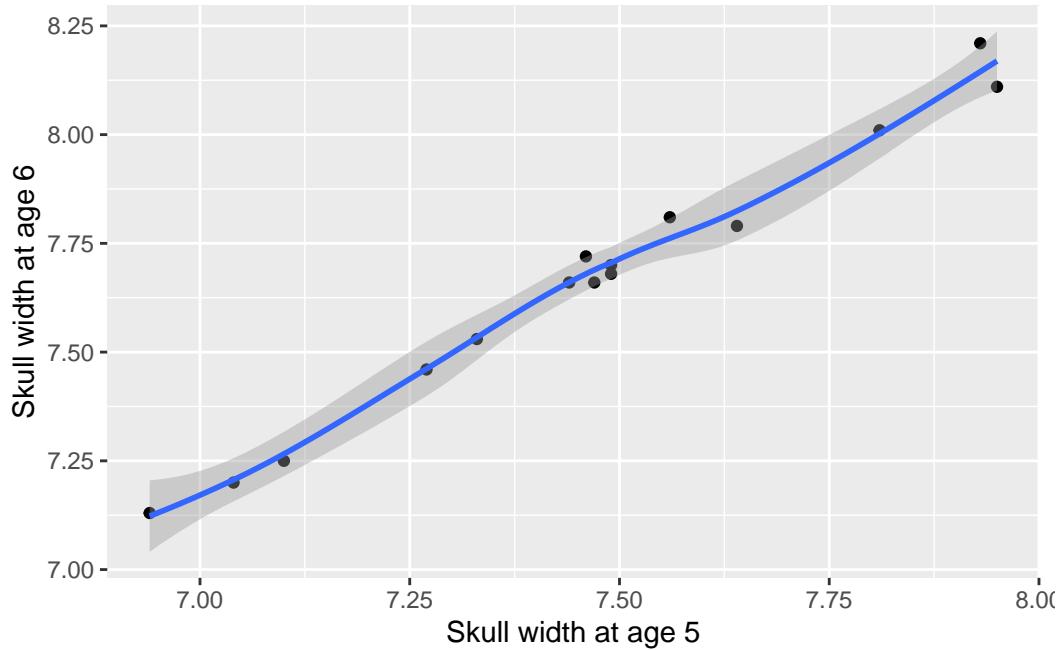


Figure 10.5.: Relation between head width at age 5 and 6

With $r = 0.9930841$. In the presence of correlation, the standard error of the pairwise difference in face width at age 5 and 6 is much smaller than the standard error of the difference between the mean face width at age 5 and 6. Thus, the associated t-statistic will be much larger for a paired sample test, *i.e.* the power of the test is much greater, and the p values are smaller.

- Repeat the above procedure with the nonparametric alternative, the Wilcoxon signed-rank test. What do you conclude?

```
wilcox.test(skull_w$width5, skull_w$width6,  
            alternative = "two.sided",  
            paired = TRUE  
)
```

Warning in wilcox.test.default(skull_w\$width5, skull_w\$width6, alternative =
"two.sided", : cannot compute exact p-value with ties

Wilcoxon signed rank test with continuity correction

```
data: skull_w$width5 and skull_w$width6  
V = 0, p-value = 0.0007193  
alternative hypothesis: true location shift is not equal to 0
```

So, we reach the same conclusion as we did using the paired sample t-test and conclude there are significant differences in skull sizes of girls aged 5 and 6 (what a surprise!).

But, wait a minute. We have used two-tailed tests here. But, given what we know about how children grow, a one-tail hypothesis would be preferable. This can be done by changing the alternative option. One uses the alternative hypothesis to decide if it is “less” or greater”. Here, we expect that if there is an effect (i.e the alternative hypothesis), width5 will be less than width6

```
t.test(skull_w$width5, skull_w$width6,  
       alternative = "less",  
       paired = TRUE  
)
```

Paired t-test

```
data: skull_w$width5 and skull_w$width6
```

```
t = -19.72, df = 14, p-value = 6.507e-12
alternative hypothesis: true mean difference is less than 0
95 percent confidence interval:
-Inf -0.1821371
sample estimates:
mean difference
-0.2

wilcox.test(skull_w$width5, skull_w$width6,
  alternative = "less",
  paired = TRUE
)
```

Warning in wilcox.test.default(skull_w\$width5, skull_w\$width6, alternative = "less", : cannot compute exact p-value with ties

Wilcoxon signed rank test with continuity correction

```
data: skull_w$width5 and skull_w$width6
V = 0, p-value = 0.0003597
alternative hypothesis: true location shift is less than 0
```

Note

Note that instead of rerunning the t-test specifying a one-tailed test, you can:

- if the sign of the estimate goes in the same direction as the alternative hypothesis, simply divide by 2 the probability you obtain with the two-tailed test
- if not the sign of the estimate is in the opposite direction of the alternative hypothesis, use $1 - p/2$

To estimate the power of a paired t-test in R, we can use the function `power.t.test()` as for other t-tests but we need to specify the argument `type = "paired"`. You need to specify the mean difference within the pairs as the `delta` and standard deviance of difference within pairs as `sd`.

```
skull_w$diff <- skull_w$width6 - skull_w$width5  
power.t.test(  
  n = 15,  
  delta = mean(skull_w$diff),  
  sd = sd(skull_w$diff),  
  type = "paired")
```

Paired t test power calculation

```
n = 15  
delta = 0.2  
sd = 0.03927922  
sig.level = 0.05  
power = 1  
alternative = two.sided
```

NOTE: n is number of *pairs*, sd is std.dev. of *differences* within pairs

10.6. Bibliography

Bumpus, H.C. (1898) The elimination of the unfit as illustrated by the introduced sparrow, *Passer domesticus*. Biological Lectures, Woods Hole Biology Laboratory, Woods Hole, 11 th Lecture: 209 - 226.

Newman, K.J. and H.V. Meredith. (1956) Individual growth in skeletal bigonial diameter during the childhood period from 5 to 11 years of age. Amer. J. Anat. 99: 157 - 187.

Chapter 11

One-way ANOVA

After completing this laboratory exercise, you should be able to:

- Use R to do a one-way parametric ANOVA with multiple comparisons
- Use R to test the validity of the parametric ANOVA assumptions
- Use R to perform a one-way non-parametric ANOVA
- Use R to transform your data so that the assumptions of parametric ANOVA are met.

11.1. R packages and data

For this lab you need:

- R packages:
 - ggplot2
 - multcomp
 - car
- data
 - dam10dat.csv

```
library(ggplot2)
library(car)
library(multcomp)
```

11.2. One-way ANOVA with multiple comparisons

The one-way ANOVA is the multi-group analog of the *t*-test, which is used to compare two groups/levels. It makes essentially the same assumptions, and in the case of two groups/levels, is in fact mathematically equivalent to the *t*-test.

In 1960-1962, the Grand Rapids Dam was built on the Saskatchewan River upstream of Cumberland House. There are anecdotal reports that during dam construction, a number of large sturgeon were stranded and died in shallow pools. Surveys of sturgeon were carried out in 1954, 1958, 1965 and 1966 with fork length (fklngh) and round weight (rdwght) being recorded (not necessarily both measurements for each individual). These data are in the data file Dam10dat.csv.

11.2.1. Visualiser les données

- Using Dam10dat.csv, you must first change the data type of the numerical variable year, so that R recognizes that we wish to treat this variable as a factor variable and not a continuous variable.

Solution

```
dam10dat <- read.csv("data/Dam10dat.csv")
dam10dat$year <- as.factor(dam10dat$year)
str(dam10dat)

'data.frame': 118 obs. of 21 variables:
 $ year      : Factor w/ 4 levels "1954","1958",...: 1 1 1 1 1 1 1 1 1 ...
 $ fklngh   : num  45 50 39 46 54.5 49 42.5 49 56 54 ...
 $ totlngth : num  49 NA 43 50.5 NA 51.7 45.5 52 60.2 58.5 ...
 $ drlngth  : logi  NA NA NA NA NA NA ...
 $ drwght   : num  16 20.5 10 17.5 19.7 21.3 9.5 23.7 31 27.3 ...
 $ rdwght   : num  24.5 33 15.5 28.5 32.5 35.5 15.3 40.5 51.5 43 ...
 $ sex       : int  1 1 1 2 1 2 1 1 1 ...
 $ age       : int  24 33 17 31 37 44 23 34 33 47 ...
 $ lfklngh : num  1.65 1.7 1.59 1.66 1.74 ...
 $ ltotl    : num  1.69 NA 1.63 1.7 NA ...
 $ ldrl     : logi  NA NA NA NA NA NA ...
 $ ldrwght : num  1.2 1.31 1 1.24 1.29 ...
```

```
$ lrdwght : num  1.39 1.52 1.19 1.45 1.51 ...
$ lage     : num  1.38 1.52 1.23 1.49 1.57 ...
$ rage     : int   4 6 3 6 7 7 4 6 6 7 ...
$ ryear    : int   1954 1954 1954 1954 1954 1954 1954 1954 1954 1954 ...
$ ryear2   : int   1958 1958 1958 1958 1958 1958 1958 1958 1958 1958 ...
$ ryear3   : int   1966 1966 1966 1966 1966 1966 1966 1966 1966 1966 ...
$ location: int   1 1 1 1 1 1 1 1 1 1 ...
$ girth    : logi  NA NA NA NA NA ...
$ lgirth   : logi  NA NA NA NA NA ...
```

- Next, have a look at the fklngh data, just as we did in the last lab for t-tests. Create a histogram with density line grouped by year to get a feel for what's happening with your data and a boxplot of length per year. What can you say about these data?

 Solution

```
mygraph <- ggplot(dam10dat, aes(x = fklnghth)) +  
  labs(x = "Fork length (cm)") +  
  geom_density() +  
  geom_rug() +  
  geom_histogram(aes(y = ..density..),  
    color = "black",  
    alpha = 0.3  
) +  
  stat_function(  
    fun = dnorm,  
    args = list(  
      mean = mean(dam10dat$fklnghth),  
      sd = sd(dam10dat$fklnghth)  
,  
      color = "red"  
)  
  
# display graph, by year  
mygraph + facet_wrap(~year, ncol = 2)
```

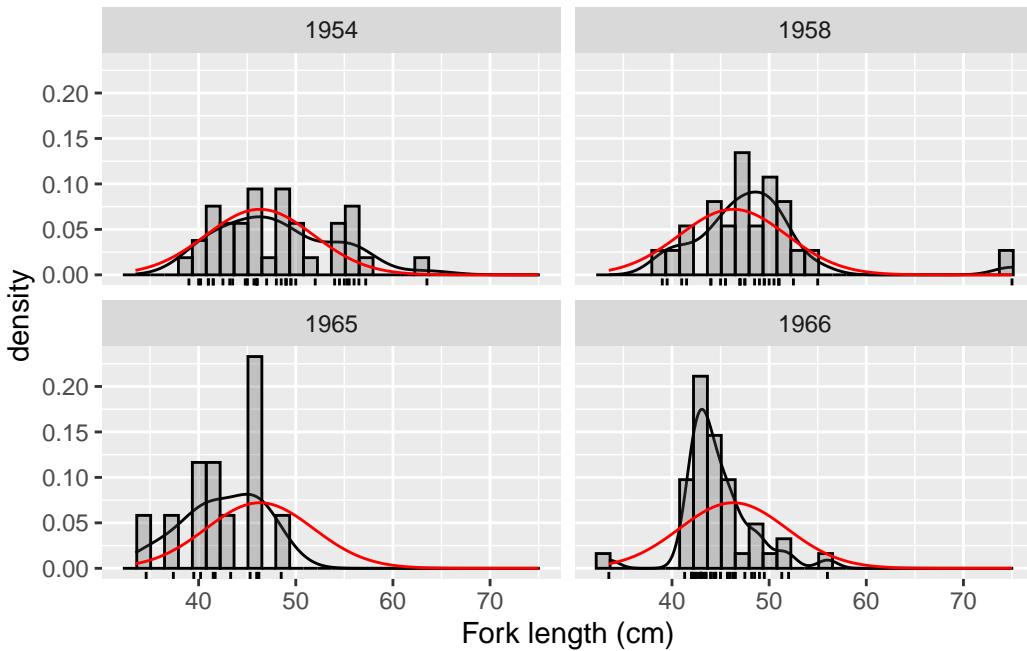


Figure 11.1.: Distribution of sturgeon length per year

```
boxplot(fklnghth ~ year, data = dam10dat)
```

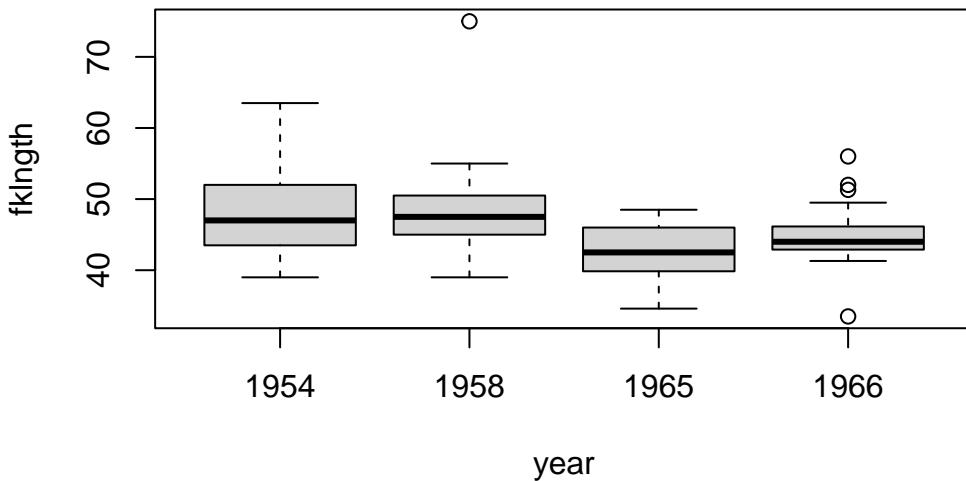


Figure 11.2.: Boxplot of sturgeon length per year

It appears as though there may have been a small drop in fklnghth after the construction of the dam, but the data

are variable and the effects are not clear. There might also be some problems with normality in the 1954 and 1966 samples, and it looks as though there are outliers in the 1958 and 1966 samples. Let's proceed with testing the assumptions of the ANOVA by running the analysis and looking at the residuals.

11.2.2. Testing the assumptions of a parametric ANOVA

Parametric one-way ANOVAs have three major assumptions:

1. the residuals are normally distributed
2. the error variance is the same for all groups (homoscedasticity)
3. the residuals are independent.

These assumptions must be tested before we can accept the results of any parametric ANOVA.

- Carry out a one-way ANOVA on fklngth by year and produce the residual diagnostic plots

```
# Fit anova model and plot residual diagnostics
anova.model1 <- lm(fklngth ~ year, data = dam10dat)
par(mfrow = c(2, 2))
plot(anova.model1)
```

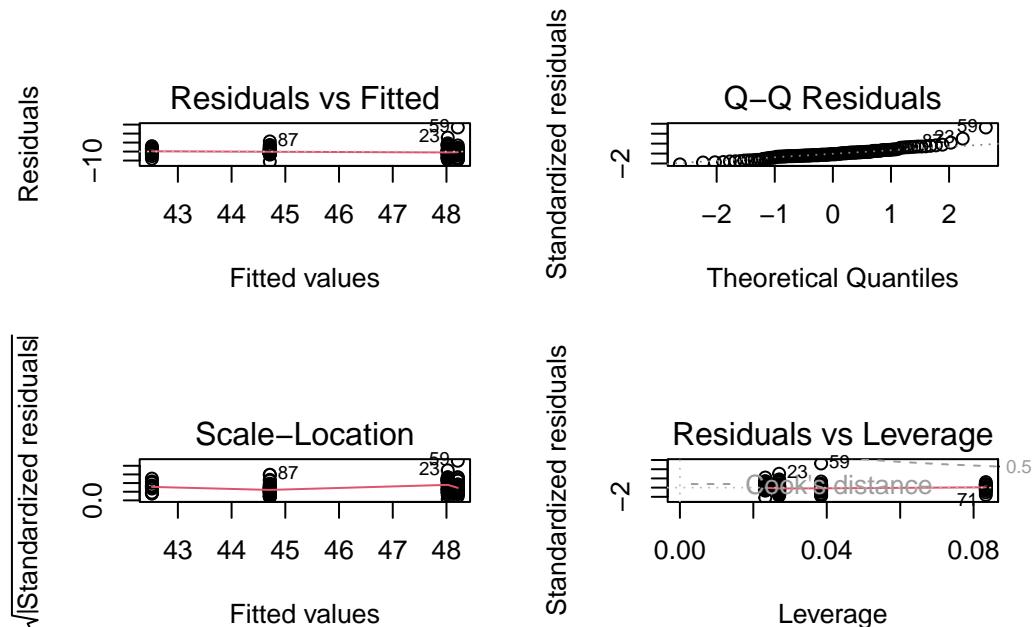


Figure 11.3.: Diagnostic plots for a one-way ANOVA

Warning

Double check that the independent variable is a **factor**. If the dependent variable is a **character**, then you will obtain only 3 graphs and an error message like:

```
'hat values (leverages) are all = 0.1
and there are no factor predictors; no plot no. 5'
```

D'après les graphiques, on peut douter de la normalité et de l'homogénéité des variances. Judging from the plots, it looks as though there may be problems with both normality and variance heterogeneity. Note that there is one point (case 59) with large expected values and a large residual that appear to lie well off the line: this is the outlier we noted earlier. This point might be expected to inflate the variance for the group it belongs to. Formal tests may also provide some insight as to whether we should be concerned about normality and variance heterogeneity.

- Perform a normality test on the residuals from the ANOVA.

```
shapiro.test(residuals(anova.model1))
```

```
Shapiro-Wilk normality test
```

```
data: residuals(anova.model1)
W = 0.91571, p-value = 1.63e-06
```

This test confirms our suspicions from the probability plot: the residuals are not normally distributed. Recall, however, that the power here is high, so only small deviations from normality are required to reject the null.

- Next, test for homoscedasticity:

```
leveneTest(fklngh ~ year, data = dam10dat)
```

| | Df | F value | Pr(>F) |
|-------|-----|---------|-----------|
| group | 3 | 2.8159 | 0.0423438 |
| | 114 | NA | NA |

The probability value tells you that you can reject the null hypothesis that there is no difference in variances among years. Thus, we conclude there is evidence that the variances in the groups are not equal.

11.2.3. Performing the ANOVA

Let's look at the results of the ANOVA, assuming for the moment that assumptions are met well enough.

```
summary(anova.model1)
```

Call:

```
lm(formula = fklngh ~ year, data = dam10dat)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|---------|---------|--------|---------|
| -11.2116 | -2.6866 | -0.7116 | 2.2103 | 26.7885 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|----------------|----------|------------|----------|-------------|
| (Intercept) | 48.0243 | 0.8566 | 56.061 | < 2e-16 *** |
| year1958 | 0.1872 | 1.3335 | 0.140 | 0.88859 |
| year1965 | -5.5077 | 1.7310 | -3.182 | 0.00189 ** |
| year1966 | -3.3127 | 1.1684 | -2.835 | 0.00542 ** |
| --- | | | | |
| Signif. codes: | 0 '***' | 0.001 '**' | 0.01 '*' | 0.05 '.' |
| | 0.1 | ' ' | 1 | |

Residual standard error: 5.211 on 114 degrees of freedom

Multiple R-squared: 0.1355, Adjusted R-squared: 0.1128

F-statistic: 5.957 on 3 and 114 DF, p-value: 0.0008246

- *Coefficients: Estimates* Note the 4 coefficients printed. They can be used to obtain the predicted values for the model (i.e. the group means). The mean fklngh for the first year (1954) is 48.0243. The coefficients for the 3 other years are the difference between the mean for that year and for 1954. So, the mean for 1965 is (48.0243-5.5077=42.5166). For each estimated coefficient, there is a standard error, a t-value and associated probability (for H₀ that the coefficient is 0). Note here that coefficients for 1965 and 1966 are both negative and significantly less than 0. Fish were smaller after the construction of the dam than in 1954. Take these p-values with a grain of salt: these are not corrected for multiple comparisons, and they constitute only a

subset of the possible comparisons. In general, I pay little attention to this part of the output and look more at what comes next.

- *Residual standard error*: The square root of the variance of the residuals (observed minus fitted values) corresponds to the amount of variability that is unexplained by the models (here an estimate of how much size varied among fish, once corrected for differences among years)
 - *Multiple R-squared* The R-squared is the proportion of the variance of the dependent variable that can be explained by the model. Here the model explains only 13.5% of the variability. Size differences among year are relatively small compared to the ranges of sizes that can occur within years. This corresponds well to the visual impression left by the histograms of `fklngth` per year
4. *F-Statistic* This is the p-value for the “omnibus” test, the test that all means are equal. Here it is much smaller than 0.05 and hence we would reject H₀ and conclude that `fklngth` varies among the years

The `anova()` command produces the standard ANOVA table that contains most of the same information:

```
anova(anova.model1)
```

| | Df | Sum Sq | Mean Sq | F value | Pr(>F) |
|-----------|-----|-----------|-----------|---------|-----------|
| year | 3 | 485.2642 | 161.75472 | 5.95744 | 0.0008246 |
| Residuals | 114 | 3095.2955 | 27.15171 | NA | NA |

The total variability in `fklngth` sums of square is partitioned into what can be accounted for by year (485.26) and what is left unexplained as residual variability (3095.30). Year indeed explains $(485.26 / (3095.30 + 485.26)) = .1355$ or 13.55% of the variability). The mean square of the residuals is their variance.

11.2.4. Performing multiple comparisons of means test

- The `pairwise.t.test()` function can be used to compare means and adjust (or not) probabilities for multiple comparisons by choosing one of the options for the argument `p.adj`:

Comparing all means without corrections for multiple comparisons.

```
pairwise.t.test(dam10dat$fklngth, dam10dat$year,
  p.adj = "none"
)
```

Pairwise comparisons using t tests with pooled SD

```
data: dam10dat$fklnngth and dam10dat$year
```

| | 1954 | 1958 | 1965 |
|------|--------|--------|--------|
| 1958 | 0.8886 | - | - |
| 1965 | 0.0019 | 0.0022 | - |
| 1966 | 0.0054 | 0.0079 | 0.1996 |

P value adjustment method: none

Option "bonf" adjusts the p-values according to the Bonferroni correction. In this case, since there are 6 p-values calculated, it amounts to simply multiplying the uncorrected p-values by 6 (unless the result is above 1, in that case the adjusted p-value is 1).

```
pairwise.t.test(dam10dat$fklnngth, dam10dat$year,  
  p.adj = "bonf"  
)
```

Pairwise comparisons using t tests with pooled SD

```
data: dam10dat$fklnngth and dam10dat$year
```

| | 1954 | 1958 | 1965 |
|------|-------|-------|-------|
| 1958 | 1.000 | - | - |
| 1965 | 0.011 | 0.013 | - |
| 1966 | 0.033 | 0.047 | 1.000 |

P value adjustment method: bonferroni

Option "holm" is the sequential Bonferroni correction, where the p-values are ranked from (i=1) smallest to (N) largest. The correction factor for p-values is then $(N - i + 1)$. Here, for example, we have N=6 pairs that are compared. The

lowest uncorrected p-value is 0.0019 for 1954 vs 1965. The corrected p-value becomes $0.0019 * (6 - 1 + 1) = 0.011$. The second lowest p-value is 0.0022. The corrected p-value is therefore $0.0022 * (6 - 2 + 1) = 0.011$. For the highest p-value, the correction is $(N - N + 1) = 1$, hence it is equal to the uncorrected probability.

```
pairwise.t.test(dam10dat$fklnngth, dam10dat$year,
  p.adj = "holm"
)
```

Pairwise comparisons using t tests with pooled SD

data: dam10dat\$fklnngth and dam10dat\$year

| | 1954 | 1958 | 1965 |
|------|-------|-------|-------|
| 1958 | 0.889 | - | - |
| 1965 | 0.011 | 0.011 | - |
| 1966 | 0.022 | 0.024 | 0.399 |

P value adjustment method: holm

The “fdr” option is for controlling the false discovery rate.

```
pairwise.t.test(dam10dat$fklnngth, dam10dat$year,
  p.adj = "fdr"
)
```

Pairwise comparisons using t tests with pooled SD

data: dam10dat\$fklnngth and dam10dat\$year

| | 1954 | 1958 | 1965 |
|------|--------|--------|------|
| 1958 | 0.8886 | - | - |
| 1965 | 0.0066 | 0.0066 | - |

```
1966 0.0108 0.0119 0.2395
```

```
P value adjustment method: fdr
```

The four post-hoc tests here tell us the same thing: differences are all between two groups of years: 1954/58 and 1965/66, since all comparisons show differences between the 50's and 60's but no differences within the 50's or 60's. So, in this particular case, the conclusion is not affected by the choice of adjustment method. But in other situations, you will observe contradictory results.

Which one to choose? Unadjusted p-values are certainly suspect when there are multiple tests. On the other hand, the traditional *Bonferroni* correction is very conservative, and becomes even more so when there are a large number of comparisons. Recent work suggest that the *fdr* approach may be a good compromise when there are a lot of comparisons. The *Tukey* method of multiple comparisons is one of the most popular and is easily performed with R (note, however, that there is a pesky bug that manifests itself when the independent variable can look like a number rather than a factor, hence the little pirouette with `paste0()` to add a letter `m` before the first digit):

```
dam10dat$myyear <- as.factor(paste0("m", dam10dat$year))
TukeyHSD(aov(fklnngth ~ myyear, data = dam10dat))
```

```
Tukey multiple comparisons of means
```

```
95% family-wise confidence level
```

```
Fit: aov(formula = fklnngth ~ myyear, data = dam10dat)
```

```
$myyear
```

| | diff | lwr | upr | p | adj |
|-------------|------------|------------|------------|-----------|-----|
| m1958-m1954 | 0.1872141 | -3.289570 | 3.6639986 | 0.9990071 | |
| m1965-m1954 | -5.5076577 | -10.021034 | -0.9942809 | 0.0100528 | |
| m1966-m1954 | -3.3126964 | -6.359223 | -0.2661701 | 0.0274077 | |
| m1965-m1958 | -5.6948718 | -10.436304 | -0.9534397 | 0.0116943 | |
| m1966-m1958 | -3.4999106 | -6.875104 | -0.1247171 | 0.0390011 | |
| m1966-m1965 | 2.1949612 | -2.240630 | 6.6305526 | 0.5710111 | |

```
par(mar = c(4, 7, 2, 1))
plot(TukeyHSD(aov(fklngh ~ myyear, data = dam10dat)), las = 2)
```

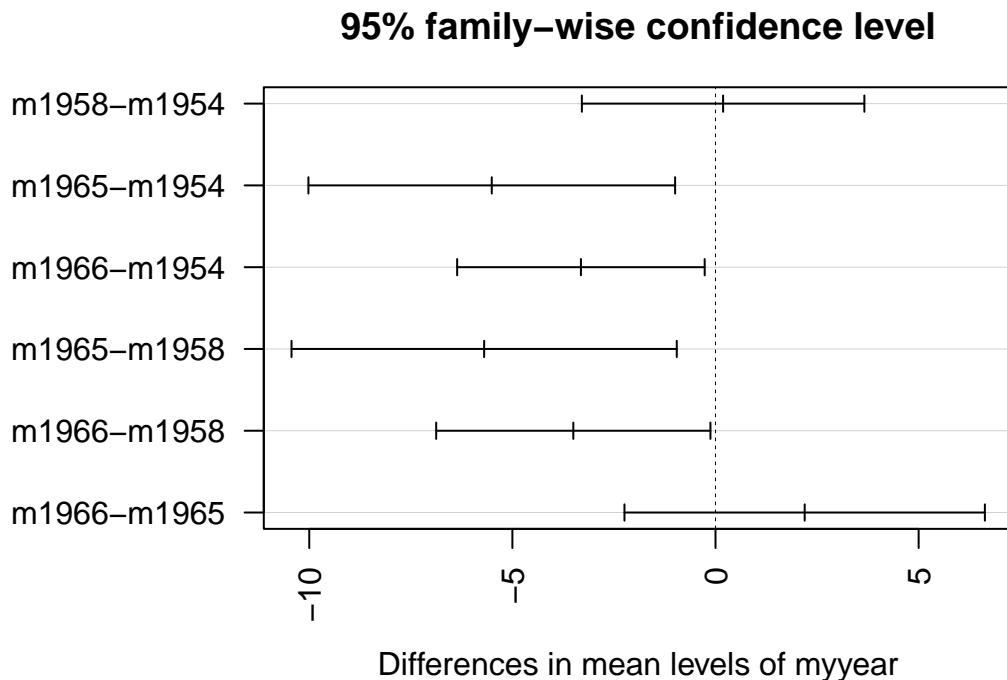


Figure 11.4.: Inter-annual differences in sturgeon length

The confidence intervals, corrected for multiple tests by the Tukey method, are plotted for differences among years. Unfortunately, the labels are not all printed because they would overlap, but the order is the same as in the preceding table. The `multcomp` can produce a better plot version, but requires a bit more code:

```
# Alternative way to compute Tukey multiple comparisons
# set up a one-way ANOVA
anova_fkl_year <- aov(fklngh ~ myyear, data = dam10dat)
# set up all-pairs comparisons for factor `year'

meandiff <- glht(anova_fkl_year, linfct = mcp(
  myyear =
  "Tukey"
))
confint(meandiff)
```

Simultaneous Confidence Intervals

Multiple Comparisons of Means: Tukey Contrasts

```
Fit: aov(formula = fklngh ~ myyear, data = dam10dat)
```

```
Quantile = 2.5955
```

```
95% family-wise confidence level
```

Linear Hypotheses:

| | Estimate | lwr | upr |
|--------------------|----------|----------|---------|
| m1958 - m1954 == 0 | 0.1872 | -3.2738 | 3.6482 |
| m1965 - m1954 == 0 | -5.5077 | -10.0005 | -1.0148 |
| m1966 - m1954 == 0 | -3.3127 | -6.3454 | -0.2800 |
| m1965 - m1958 == 0 | -5.6949 | -10.4148 | -0.9750 |
| m1966 - m1958 == 0 | -3.4999 | -6.8598 | -0.1400 |
| m1966 - m1965 == 0 | 2.1950 | -2.2205 | 6.6104 |

```
par(mar = c(5, 7, 2, 1))  
plot(meandiff)
```

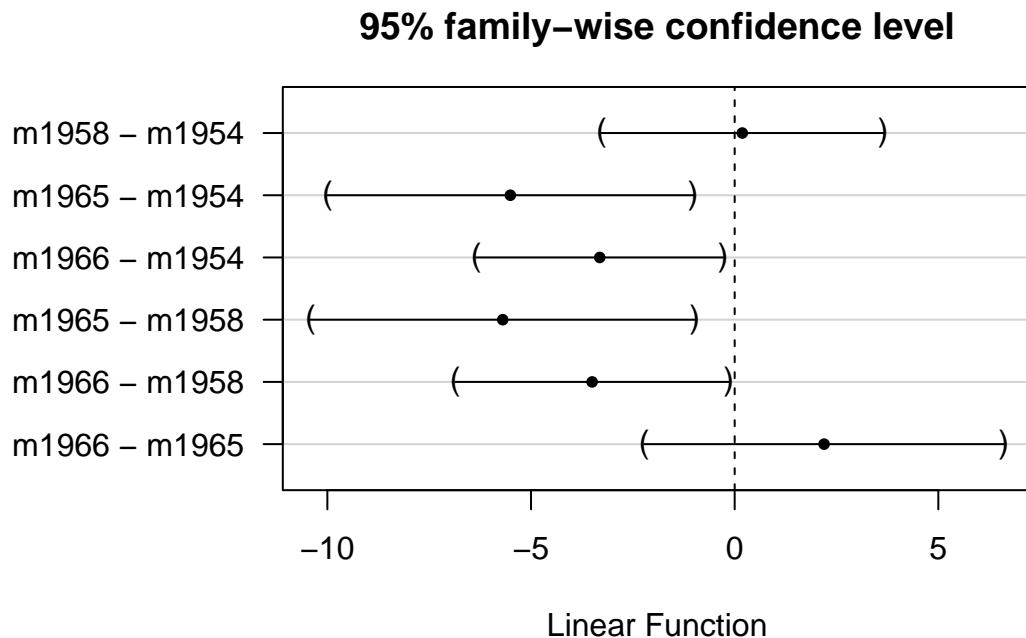


Figure 11.5.: Inter-annual differences in sturgeon length

This is better. Also useful is a plot the means and their confidence intervals with the Tukey groupings shown as letters above:

```
# Compute and plot means and Tukey CI
means <- glht(
  anova_fkl_year,
  linfct = mcp(myyear = "Tukey")
)
cimeans <- cld(means)

# use sufficiently large upper margin
# plot
old_par <- par(mai = c(1, 1, 1.25, 1))
plot(cimeans)
```

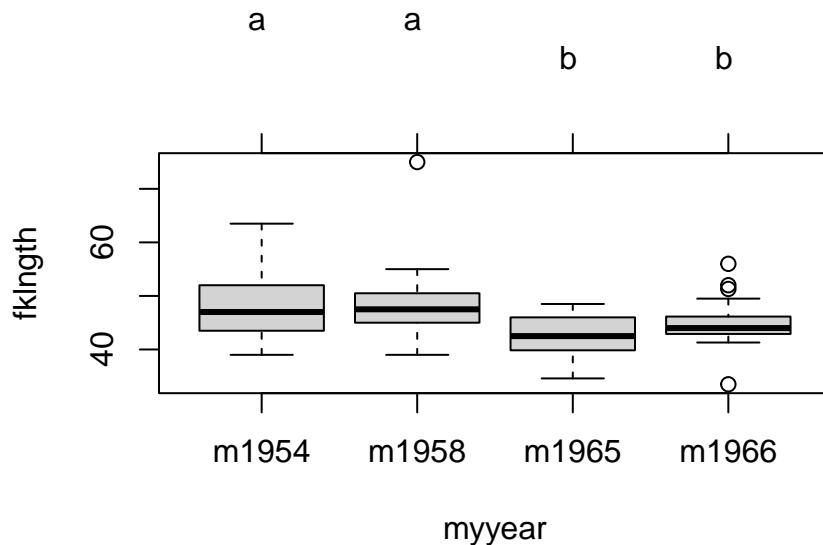


Figure 11.6.: Inter-annual differences in sturgeon length

Note the letters appearing on top. Years labelled with the same letter do not differ significantly.

11.3. Data transformations and non-parametric ANOVA

In the above example to examine differences in fklngh among years , we detected evidence of non-normality and variance heterogeneity. If the assumptions underlying a parametric ANOVA are not valid, there are several options:

1. if sample sizes in each group are reasonably large, parametric ANOVA is reasonably robust with respect to the normality assumption, for the same reason that the t-test is, so the results are probably not too bad;
 2. we can transform the data;
 3. we can go the non-parametric route.
- Repeat the one-way ANOVA in the section above, but this time run the analysis on the log₁₀ fklngh . With this transformation, do some of the problems encountered previously disappear?

```
# Fit anova model on log10 of fklngh and plot residual diagnostics
par(mfrow = c(2, 2))
anova.model2 <- lm(log10(fklngh) ~ year, data = dam10dat)
plot(anova.model2)
```

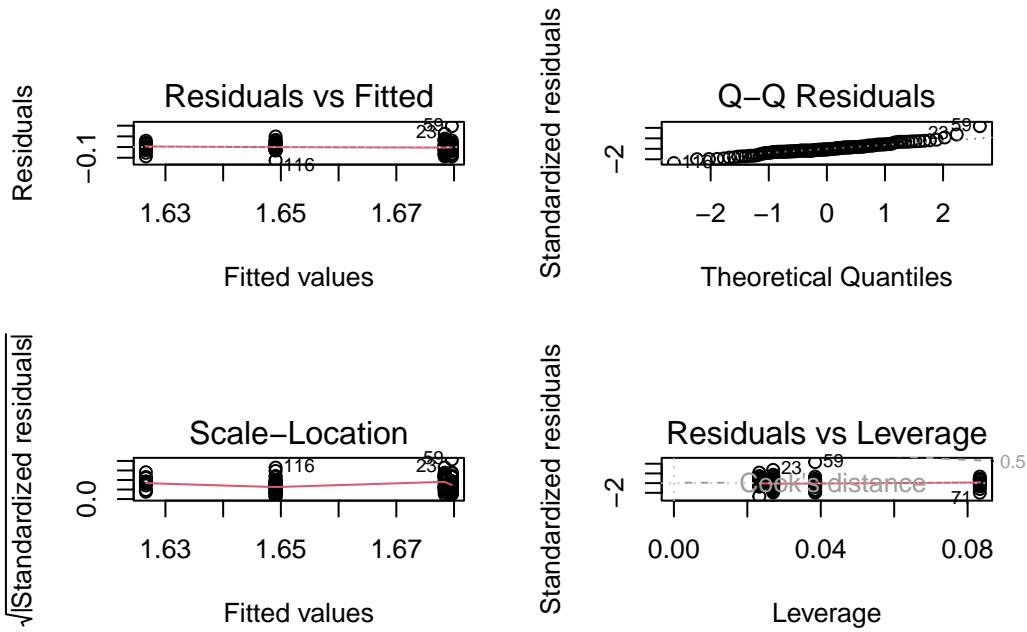


Figure 11.7.: Diagnostic plots for the ANOVA of sturgeon length by year

Looking at the residuals, things look barely better than before without the log transformation. Running the Wilks-Shapiro test for normality on the residuals, we get:

```
shapiro.test(residuals(anova.model2))
```

```
Shapiro-Wilk normality test

data: residuals(anova.model2)
W = 0.96199, p-value = 0.002048
```

So, it would appear that we still have some problems with the assumption of normality and are just on the border line of meeting the assumption of homogeneity of variances. You have several choices here:

1. try to find a different transformation to satisfy the assumptions,
 2. assume the data are close enough to meeting the assumptions, or
 3. perform a non-parametric ANOVA.
- The most commonly used non-parametric analog of the parametric one-way ANOVA is the Kruskall-Wallis one-way ANOVA. Perform a Kruskall-Wallis one-way ANOVA of `fklngh`, and compare these results to the parametric analysis above. What do you conclude?

```
kruskal.test(fklnth ~ year, data = dam10dat)
```

```
Kruskal-Wallis rank sum test

data: fklnth by year
Kruskal-Wallis chi-squared = 15.731, df = 3, p-value = 0.001288
```

So, the conclusion is the same as with the parametric ANOVA: we reject the null that the mean rank is the same for each year. Thus, despite violation of one or more assumptions, the parametric analysis is telling us the same thing as the non-parametric analysis: the conclusion is, therefore, quite robust.

11.4. Dealing with outliers

Our preliminary analysis of the relationship between `fklnth` and `year` suggested there might be some outliers in the data. These were evident in the box plots of `fklnth` by `year` and flagged as cases 59, 23 and 87 in the residual probability plot and residual-fit plot. In general, you have to have very good reasons for removing outliers from a data set (e.g., you know there was a mistake made in the data collection/entry). However, it is often useful to know how the analysis changes if you remove the outliers from the data set.

- Repeat the original ANOVA of `fklnth` by `year` but work with a subset of the data without the outliers. Have any of the conclusions changed?

```
damsubset <- dam10dat[-c(23, 59, 87), ] # removes obs 23, 59 and 87
aov_damsubset <- aov(fklnth ~ as.factor(year), damsubset)
summary(aov_damsubset)
```

```
            Df Sum Sq Mean Sq F value    Pr(>F)
as.factor(year)   3  367.5  122.50   6.894 0.000267 ***
Residuals      111 1972.4   17.77
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
shapiro.test(residuals(aov_damsubset))
```

Shapiro-Wilk normality test

```
data: residuals(aov_damsubset)
W = 0.98533, p-value = 0.2448
```

```
leveneTest(fklngh ~ year, damsubset)
```

| | Df | F value | Pr(>F) |
|-------|-----|----------|-----------|
| group | 3 | 4.623721 | 0.0043666 |
| | 111 | NA | NA |

Elimination of three outliers, in this case, makes things better in terms of the normality assumption, but does not improve the variances. Moreover, the fact that the conclusion drawn from the original ANOVA with outliers retained does not change upon their removal reinforces the fact that there is no good reason to remove the points. Instead of a Kruskall-Wallis rank-based test, a permutation test could be used.

11.5. Permutation test

This is an example for a more complex way of doing permutation that we used when `lmPerm` was not available.

```
#####
# Permutation Test for one-way ANOVA
# modified from code written by David C. Howell
# http://www.uvm.edu/~dhowell/StatPages/
# More_Stuff/Permutation%20Anova/PermTestsAnova.html
# set desired number of permutations
nreps <- 500
# to simplify reuse of this code, copy desired dataframe to mydata
mydata <- dam10dat
```

```
# copy model formula to myformula
myformula <- as.formula("fklngth ~ year")
# copy dependent variable vector to mydep
mydep <- mydata$fklngth
# copy independent variable vector to myindep
myindep <- as.factor(mydata$year)
#####
# You should not need to modify code chunk below
#####
# Compute observed F value for original sample
mod1 <- lm(myformula, data = mydata) # Standard Anova
sum_anova <- summary(aov(mod1)) # Save summary to variable
obs_f <- sum_anova[[1]]$"F value"[1] # Save observed F value
# Print standard ANOVA results
cat(
  " The standard ANOVA for these data follows ",
  "\n"
)

print(sum_anova, "\n")
cat("\n")
cat("\n")
print("Resampling as in Manly with unrestricted sampling of observations. ")

# Now start resampling
boot_f <- numeric(nreps) # initialize vector to receive permuted
values
boot_f[1] <- obs_f
for (i in 2:nreps) {
  newdependent <- sample(mydep, length(mydep)) # randomize dep
  var
  mod2 <- lm(newdependent ~ myindep) # refit model
  b <- summary(aov(mod2))
```

```

boot_f[i] <- b[[1]]$"F value"[1] # store F stats
}

permprob <- length(boot_f[boot_f >= obs_f]) / nreps
cat(
  " The permutation probability value is: ", permprob,
  "\n"
)
# end of code chunk for permutation

```

Version `lmPerm` du test de permutation.

```

## lmPerm version of permutation test
library(lmPerm)

# for generality, copy desired dataframe to mydata
# and model formula to myformula

mydata <- dam10dat

myformula <- as.formula("fklngth ~ year")

# Fit desired model on the desired dataframe

mymodel <- lm(myformula, data = mydata)

# Calculate permutation p-value

anova(lmp(myformula, data = mydata, perm = "Prob", center = FALSE, Ca = 0.001))

```

Chapter 12

Multiway ANOVA: factorial and nested designs

After completing this laboratory exercise, you should be able to:

- Use R to do parametric ANOVAs for 2-way factorial designs with replication.
- Use R to do 2-way factorial design ANOVA without replication
- Use R to do parametric ANOVAs for nested designs with replication.
- Use R to do non-parametric 2-way ANOVAs
- Use R to do multiway pairwise comparisons

Be aware that there are a large number of possible ANOVA designs, many of which can be handled by R: this laboratory is

12.1. R packages and data needed

For this lab you need:

- R packages:
 - tidyverse
 - multicomp
 - car
 - effects
- data files:
 - Stu2wdat.csv
 - Stu2mdat.csv

- nr2wdat.csv
- nestdat.csv
- wmcdat2.csv
- wmc2dat2.csv

```
library(multcomp)
library(car)
library(tidyverse)
library(effects)
```

12.2. Two-way factorial design with replication

Many experiments are designed to investigate the joint effects of several different factors: in a two-way ANOVA, we examine the effect of two factors, but in principle the analysis can be extended to three, four or even five factors, although interpreting the results from 4- and 5-way ANOVAs can be very difficult.

Suppose that we are interested in the effects of two factors: `location` (Cumberland House and The Pas) and `sex` (male or female) on sturgeon size (data can be found in `Stu2wdat.csv`). Note that because the sample sizes are not the same for each group, this is an unbalanced design. Note also that there are missing data for some of the variables, meaning that not every measurement was made on every fish.

12.2.1. Fixed effects ANOVA (Model I)

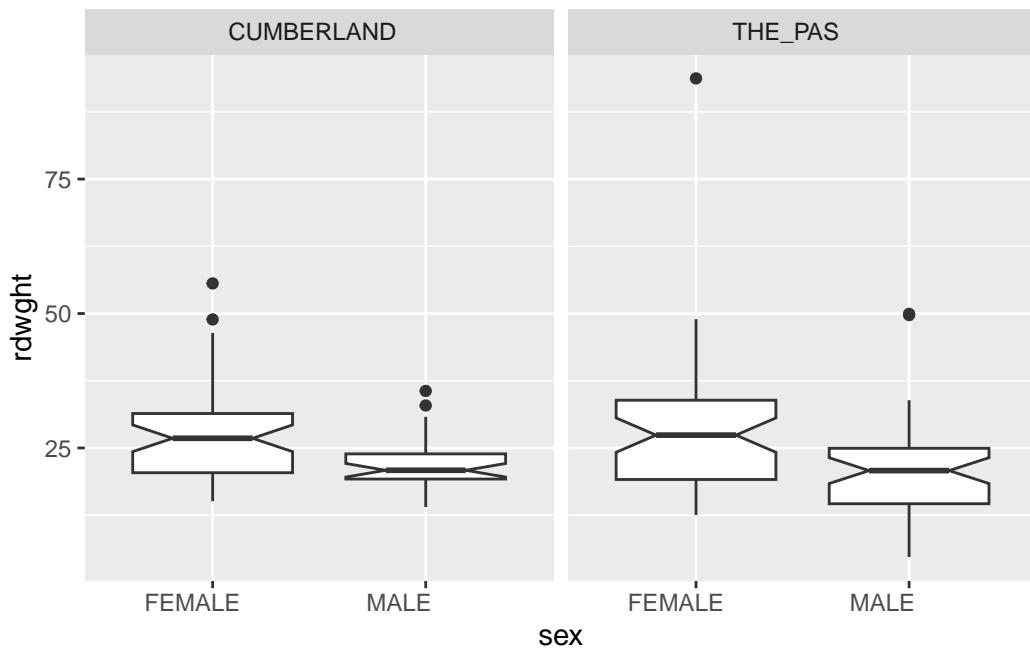
- Begin by having a look at the data by generating box plots of `rdwght` for `sex` and `location` from the file `Stu2wdat.csv`.

Solution

```
Stu2wdat <- read.csv("data/Stu2wdat.csv")

ggplot(Stu2wdat, aes(x = sex, y = rdwght)) +
  geom_boxplot(notch = TRUE) +
  facet_grid(~location)
```

Warning: Removed 4 rows containing non-finite outside the scale range
`stat_boxplot()`.



From this, it appears as though females might be larger at both locations. It's difficult to get an idea of whether fish differ in size between the two locations. The presence of outliers on these plots suggests there might be problems meeting normality assumptions for the residuals.

- Generate summary statistics for rdwght by sex and location .

```
Stu2wdat %>%
  group_by(sex, location) %>%
  summarise(
    mean = mean(rdwght, na.rm = TRUE), sd = sd(rdwght, na.rm = TRUE), n = n()
  )
```

``summarise()`` has grouped output by 'sex'. You can override using the ``.groups`` argument.

| sex | location | mean | sd | n |
|--------|------------|----------|-----------|----|
| FEMALE | CUMBERLAND | 27.37347 | 9.331438 | 51 |
| FEMALE | THE_PAS | 27.97717 | 12.533105 | 55 |
| MALE | CUMBERLAND | 22.14118 | 4.789390 | 34 |
| MALE | THE_PAS | 20.64652 | 9.917066 | 46 |

| sex | location | mean | sd | n |
|-----|----------|------|----|---|
|-----|----------|------|----|---|

The summary statistics confirm our interpretation of the box plots: females appear to be larger than males, and differences in fish size between locations are small.

- Using the file `Stu2wdat.csv`, do a two-way factorial ANOVA:

```
# Fit anova model and plot residual diagnostics
# but first, save current par and set graphic page to hold 4 graphs
opar <- par(mfrow = c(2, 2))

anova.model1 <- lm(rdwght ~ sex + location + sex:location,
  contrasts = list(sex = contr.sum, location = contr.sum),
  data = Stu2wdat
)
anova(anova.model1)
```

| | Df | Sum Sq | Mean Sq | F value | Pr(>F) |
|--------------|-----|--------------|-------------|------------|-----------|
| sex | 1 | 1839.552607 | 1839.552607 | 18.6784735 | 0.0000257 |
| location | 1 | 4.261586 | 4.261586 | 0.0432714 | 0.8354530 |
| sex:location | 1 | 48.692262 | 48.692262 | 0.4944121 | 0.4828844 |
| Residuals | 178 | 17530.359982 | 98.485168 | NA | NA |

⚠ Warning

Be careful here. R gives you the sequential sums of squares (Type I) and associated Mean squares and probabilities. These are not to be trusted unless the design is perfectly balanced. In this case, there are varying numbers of observations across sex and location combinations and therefore the design is not balanced.

What you want are the partial sums of squares (type III). The easiest way to get them is to use the `Anova()` function in the `car` package (note the subtle difference, `Anova()` is not the same as `anova()`, remember case matters in R.). However, this is not enough by itself. To get the proper values for the type III sums of square, one also needs to specify contrasts, hence the cryptic `contrasts = list(sex = contr.sum, location = contr.sum)`.

```
library(car)
Anova(anova.model1, type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|--------------|--------------|-----|--------------|-----------|
| (Intercept) | 1.065073e+05 | 1 | 1081.4551655 | 0.0000000 |
| sex | 1.745358e+03 | 1 | 17.7220422 | 0.0000405 |
| location | 8.778349e+00 | 1 | 0.0891337 | 0.7656296 |
| sex:location | 4.869226e+01 | 1 | 0.4944121 | 0.4828844 |
| Residuals | 1.753036e+04 | 178 | NA | NA |

On the basis of the ANOVA, there is no reason to reject two null hypotheses: (1) that the effect of sex (if any) does not depend on location (no interaction), and (2) that there is no difference in the size of sturgeon (pooled over sex) between the two locations . On the other hand, we reject the null hypothesis that there is no difference in size between male and female sturgeon (pooled over location), precisely as expected from the graphs.

```
par(mfrow = c(2, 2))
plot(anova.model1)
```

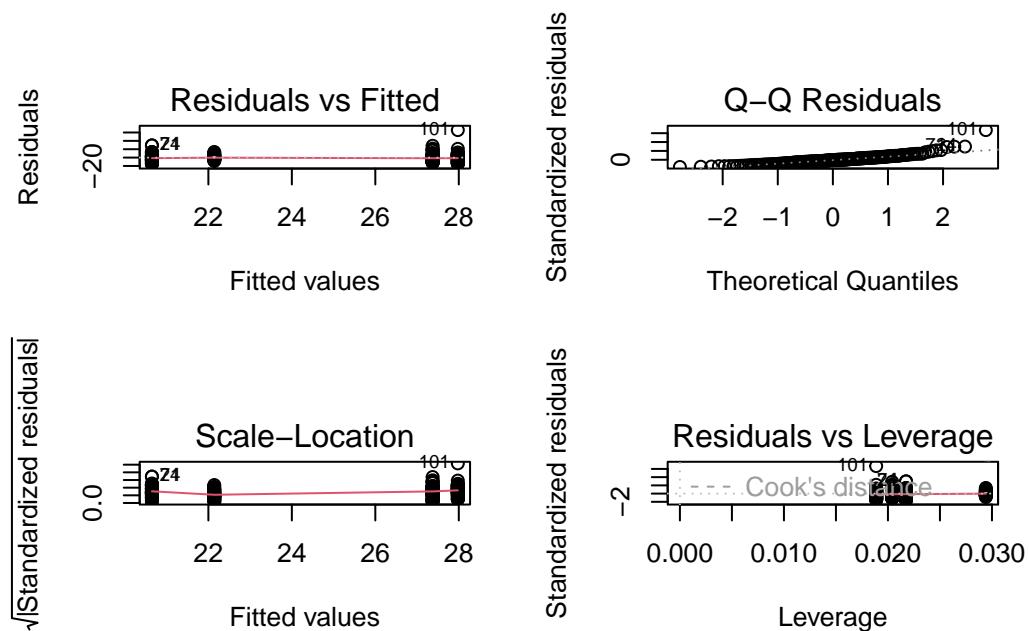


Figure 12.1.: Checking model assumptions for ANOVA model1

As usual, we cannot accept the above results without first ensuring that the assumptions of ANOVA are met. Examination of the residuals plots above shows that the residuals are reasonably normally distributed, with the exception of three potential outliers flagged on the QQ plot (cases 101, 24, & 71; the latter two are on top of one another). However, Cook's distances are not large for these (the 0.5 contour is not even visible on the plot), so there is little indication that these are a concern. The residuals vs fit plot shows that the spread of residuals is about equal over the range of the fitted values, again with the exception of a few cases. When we test for normality of residuals we get:

```
shapiro.test(residuals(anova.model1))
```

```
Shapiro-Wilk normality test
```

```
data: residuals(anova.model1)
W = 0.87213, p-value = 2.619e-11
```

So, there is evidence of non-normality in the residuals.

We will use the Levene's test to examine the assumption of homogeneity of variances, just as we did with the 1-way anova.

```
leveneTest(rdwght ~ sex * location, data = Stu2wdat)
```

| | Df | F value | Pr(>F) |
|-------|-----|----------|-----------|
| group | 3 | 3.852621 | 0.0105483 |
| | 178 | NA | NA |

If the assumption of homogeneity of variances was valid, we would be accepting the null that the mean of the absolute values of residuals does not vary among levels of sex and location (i.e., group). The above table shows that the hypothesis is rejected and we conclude there is evidence of heteroscedasticity. All in all, there is some evidence that several important assumptions have been violated. However, whether these violations are sufficiently large to invalidate our conclusions remains to be seen.

🔥 Exercise

Repeat this procedure using the data file Stu2mdat.Rdata . Now what do you conclude? Suppose you wanted to compare the sizes of males and females: in what way would these comparisons differ between Stu2wdat.Rdata and Stu2mdat.Rdata ?

💡 Solution

```
Stu2mdat <- read.csv("data/Stu2mdat.csv")
anova.model2 <- lm(
  formula = rdwght ~ sex + location + sex:location,
  contrasts = list(sex = contr.sum, location = contr.sum),
  data = Stu2mdat
)
summary(anova.model2)
Anova(anova.model2, type = 3)
```

Call:

```
lm(formula = rdwght ~ sex + location + sex:location, data = Stu2mdat,
  contrasts = list(sex = contr.sum, location = contr.sum))
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|---------|--------|--------|-------|--------|
| -15.917 | -6.017 | -0.580 | 4.445 | 65.743 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|----------------|----------|------------|----------|--------------|
| (Intercept) | 24.5346 | 0.7461 | 32.885 | < 2e-16 *** |
| sex1 | -0.5246 | 0.7461 | -0.703 | 0.483 |
| location1 | 0.2227 | 0.7461 | 0.299 | 0.766 |
| sex1:location1 | 3.1407 | 0.7461 | 4.210 | 4.05e-05 *** |
| --- | | | | |
| Signif. codes: | 0 '***' | 0.001 '**' | 0.01 '*' | 0.05 '.' |
| | 1 | | | |

```
Residual standard error: 9.924 on 178 degrees of freedom
(4 observations deleted due to missingness)

Multiple R-squared:  0.09744,   Adjusted R-squared:  0.08223
F-statistic: 6.405 on 3 and 178 DF,  p-value: 0.0003817
```

Note that in this case, we see that at Cumberland House, females are larger than males, whereas the opposite is true in The Pas (you can confirm this observation by generating summary statistics). What happens with the ANOVA (remember, you want Type III sum of squares)?

| | Sum Sq | Df | F value | Pr(>F) |
|--------------|--------------|-----|--------------|-----------|
| (Intercept) | 1.065073e+05 | 1 | 1081.4551655 | 0.0000000 |
| sex | 4.869226e+01 | 1 | 0.4944121 | 0.4828844 |
| location | 8.778349e+00 | 1 | 0.0891337 | 0.7656296 |
| sex:location | 1.745358e+03 | 1 | 17.7220422 | 0.0000405 |
| Residuals | 1.753036e+04 | 178 | NA | NA |

In this case, the interaction term `sex:location` is significant but the main effects are not significant.

- You might find it useful here to generate plots for the two data files to compare the interactions between `sex` and `location`. The effect plot shows the relationship between means for each combination of factors (also called cell means). Generate an effect plot for the two models using the `allEffects()` command from the `effects`  package:

```
library(effects)
allEffects(anova.model1)
```

```
model: rdwght ~ sex + location + sex:location
```

```
sex*location effect
      location
sex      CUMBERLAND  THE_PAS
FEMALE      27.37347  27.97717
MALE       22.14118  20.64652
```

```
plot(allEffects(anova.model1), "sex:location")
```

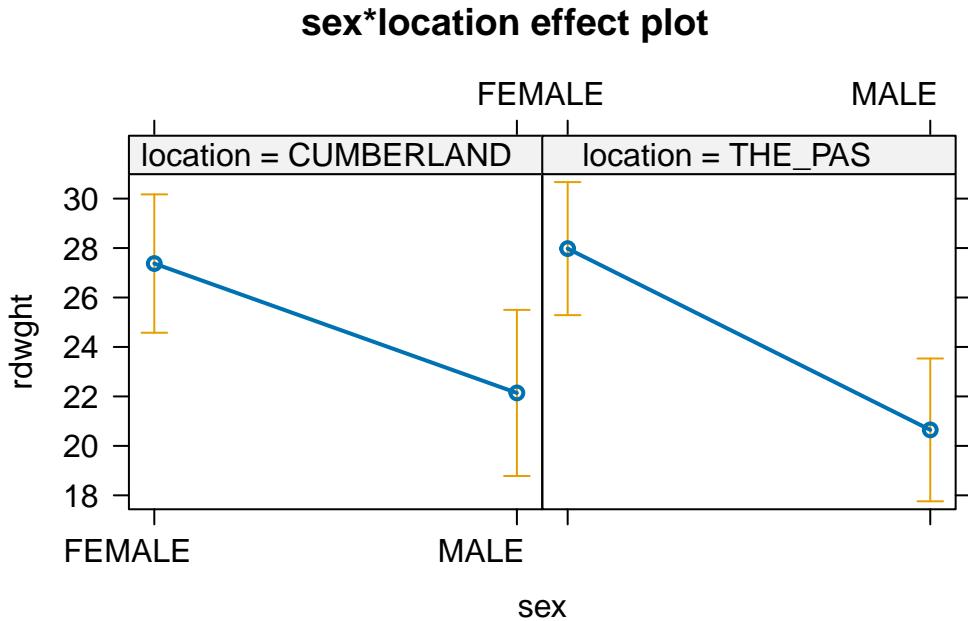


Figure 12.2.: Effet du sexe et du lieu sur le poids des esturgeons

```
allEffects(anova.model2)
```

```
model: rdwght ~ sex + location + sex:location
```

```
sex*location effect
```

| | | location | |
|--------|--|------------|----------|
| sex | | CUMBERLAND | THE_PAS |
| FEMALE | | 27.37347 | 20.64652 |
| MALE | | 22.14118 | 27.97717 |

```
plot(allEffects(anova.model2), "sex:location")
```

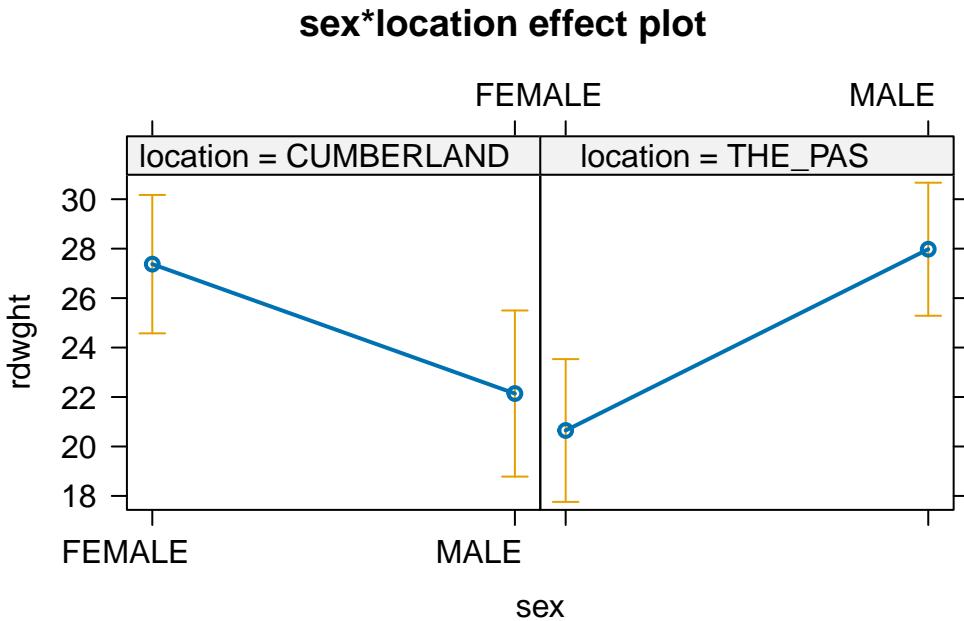


Figure 12.3.: Effet du sexe et du lieu sur le poids des esturgeons

There is a very large difference between the results from Stu2wdat and Stu2mdat. In the former case, because there is no significant interaction, we can essentially pool over the levels of factor 1 (sex, say) to test for the effects of location , or over the levels of factor 2 (location) to test for the effects of sex . In fact, if we do so and simply run a one-way ANOVA on the Stu2wdat data with sex as the grouping variable, we get:

```
Anova(aov(rdwght ~ sex, data = Stu2wdat), type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|-------------|-----------|-----|-----------|----------|
| (Intercept) | 78191.023 | 1 | 800.43979 | 0.00e+00 |
| sex | 1839.553 | 1 | 18.83146 | 2.38e-05 |
| Residuals | 17583.314 | 180 | NA | NA |

Note that here the residual sum of squares (17583) is only slightly higher than for the 2-way model (17530), simply because, in the 2-way model, only a small fraction of the explained sums of squares is due to the location main effect or the sex:LOCATION interaction. On the other hand, if you try the same trick with stu2mdat, you get:

```
Anova(aov(rdwght ~ sex, data = Stu2mdat), type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|-------------|------------|-----|------------|-----------|
| (Intercept) | 55251.2030 | 1 | 515.043549 | 0.0000000 |
| sex | 113.3992 | 1 | 1.057091 | 0.3052593 |
| Residuals | 19309.4672 | 180 | NA | NA |

Here, the residuals sum of squares (19309) is much larger than in the 2-way model (17530), because most of the explained sums of squares is due to the interaction. Note that if we did this, we would conclude that male and female sturgeons don't differ in size. But in fact they do: it's just that the difference is in different directions, depending on location. This is why it is always dangerous to try and make too much of main effects in the presence of interactions!

12.2.2. Mixed effects ANOVA (Model III)

We have neglected an important component in the above analyses, and that is related to the type of ANOVA model we wish to run. In this example, Location could be considered a random effect, whereas sex is a fixed effect (because it is “fixed” biologically), and so this model could be treated as a mixed model (Model III) ANOVA. Note that in these analyses, R treats analyses by default as Model I ANOVA, so that the main effects and the interaction are tested over the residuals mean square. Recall, however, that in a Model III ANOVA, main effects are tested over the interaction mean square or the pooled interaction mean square and residual mean square (depending on which statistician you consult!)

- Working with the Stu2wdat data, rebuild the ANOVA table for rdwght for the situation in which location is a random factor and sex is a fixed factor. To do this, you need to recalculate the F-ratio for sex using the sex:location interaction mean square instead of the residual mean square. This is most easily accomplished by hand, making sure you are working with the Type III Sums of squares ANOVA table.

Solution

```
Anova(anova.model1, type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|--------------|--------------|----|--------------|-----------|
| (Intercept) | 1.065073e+05 | 1 | 1081.4551655 | 0.0000000 |
| sex | 1.745358e+03 | 1 | 17.7220422 | 0.0000405 |
| location | 8.778349e+00 | 1 | 0.0891337 | 0.7656296 |
| sex:location | 4.869226e+01 | 1 | 0.4944121 | 0.4828844 |

| | Sum Sq | Df | F value | Pr(>F) |
|-----------|--------------|-----|---------|--------|
| Residuals | 1.753036e+04 | 178 | NA | NA |

For `sex`, the new ratio of mean squares is

$$F = \frac{(1745/1)}{(49/1)} = 35.6$$

To assign a probability to the new F-value, enter the following in the commands window: `pf(F, df1, df2, lower.tail = FALSE)`, where `F` is the newly calculated F-value, and `df1` and `df2` are the degrees of freedom of the numerator (`sex`) and denominator (`SEX:location`), respectively.

```
pf(35.6, 1, 1, lower.tail = FALSE)
```

[1] 0.1057152

Note that the *p value* for `sex` is now non-significant. This is because the error MS of the initial ANOVA is smaller than the interaction MS, but mostly because the number of degrees of freedom of the denominator of the F test has dropped from 178 to 1. In general, a drop in the denominator degrees of freedom makes it much more difficult to reach significance.

i Note

Mixed model which are a generalisation of mixed-effect ANOVA are now really developed and are to be favoured instead of doing it by hand.

12.3. 2-way factorial ANOVA without replication

In some experimental designs, there are no replicates within data cells: perhaps it is simply too expensive to obtain more than one datum per cell. A 2-way ANOVA is still possible under these circumstances, but there is an important limitation.

⚠ Warning

Because there is no replication within cells, there is no error variance: we have simply a row sum of squares, a column sum of squares, and a remainder sum of squares. This has important implications: if there is an

interaction in a Model III ANOVA, only the fixed effect can be tested (over the remainder MS); for Model I ANOVAs, or for random effects in Model III ANOVAs, it is not appropriate to test main effects over the remainder unless we are sure there is no interaction.

A limnologist studying Round Lake in Algonquin Park takes a single temperature (`temp`) reading at 10 different depths (`depth` , in m) at four times (`date`) over the course of the summer. Her data are shown in `Nr2wdat.csv`.

- Do a two-way unreplicated ANOVA using `temp` as the dependent variable, `date` and `depth` as the factor variables (you will need to recode `depth` to tell R to treat this variable as a factor). Note that there is no interaction term included in this model.

```
nr2wdat <- read.csv("data/nr2wdat.csv")
nr2wdat$depth <- as.factor(nr2wdat$depth)
anova.model4 <- lm(temp ~ date + depth, data = nr2wdat)
Anova(anova.model4, type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|-------------|-----------|----|------------|---------|
| (Intercept) | 1511.9940 | 1 | 125.565158 | 0.0e+00 |
| date | 591.1467 | 3 | 16.364138 | 2.9e-06 |
| depth | 1082.8202 | 9 | 9.991552 | 1.5e-06 |
| Residuals | 325.1207 | 27 | NA | NA |

Assuming that this is a Model III ANOVA (`date` random, `depth` fixed), what do you conclude? (Hint: you may want to generate an interaction plot of `temp` versus `depth` and `month`, just to see what's going on.)

```
interaction.plot(nr2wdat$depth, nr2wdat$date, nr2wdat$temp)
```

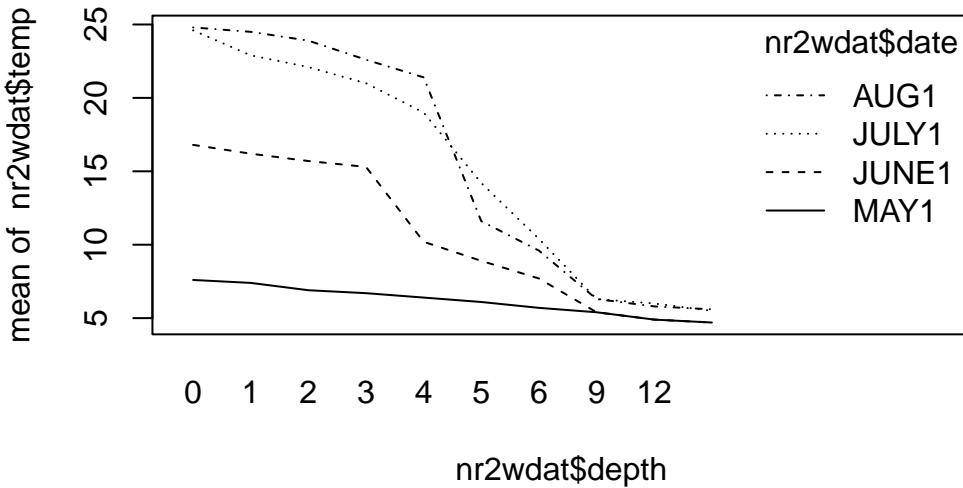


Figure 12.4.: Effet du mois et de la profondeur sur la température

There is a highly significant decrease in temperature as depth increases. To test the effect of month (the (assumed) random factor), we must assume that there is no interaction between depth and month, i.e. that the change in temperature with depth is the same for each month. This is a dubious assumption: if you plot temperature against depth for each month, you should see that the temperature profile becomes increasingly non-linear as the summer progresses (i.e. the thermocline develops), from almost a linear decline in early spring to what amounts to a step decline in August. In other words, the relationship between temperature and depth does change with month, so that if you were to use the above fitted model to estimate, say, the temperature at a depth of 5 m in July, you would not get a particularly good estimate.

In terms of residual diagnostics, have a look at the residuals probability plot and residuals vs fitted values plot.

```
par(mfrow = c(2, 2))
plot(anova.model4)
```

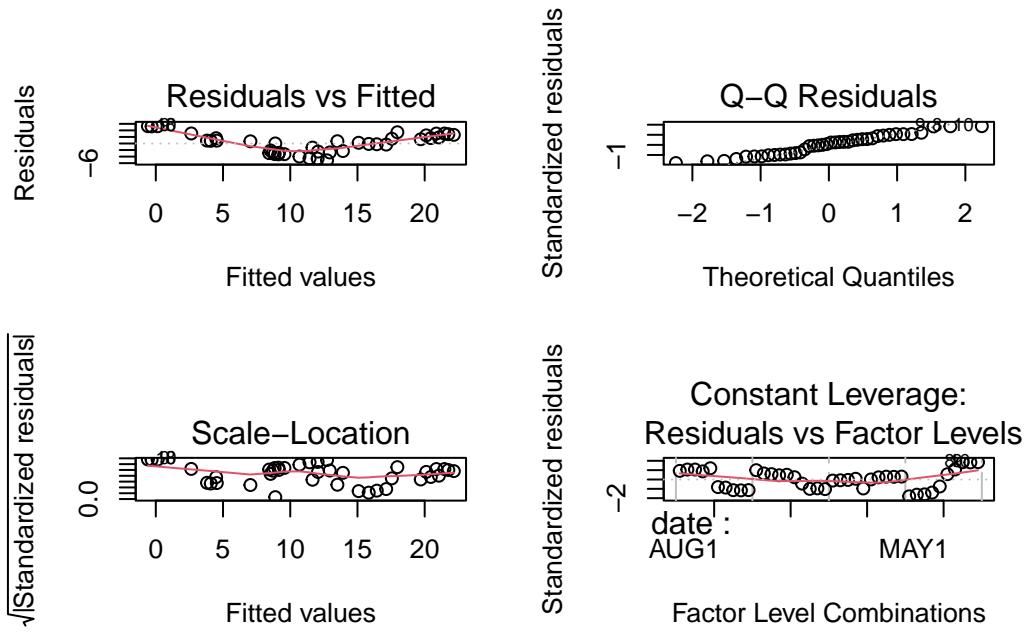


Figure 12.5.: Conditions d'applications du modèle anova.model4

```
shapiro.test(residuals(anova.model4))
```

Shapiro-Wilk normality test

```
data: residuals(anova.model4)
W = 0.95968, p-value = 0.1634
```

Testing the residuals for normality, we get $p = 0.16$, so that the normality assumption seems to be O.K. In terms of heteroscedasticity, we can only test among months, using depths as replicates (or among depths using months as replicates). Using depths as replicates within months, we find

```
leveneTest(temp ~ date, data = nr2wdat)
```

Warning in leveneTest.default(y = y, group = group, ...): group coerced to factor.

| | Df | F value | Pr(>F) |
|-------|----|---------|--------|
| group | 3 | 17.9789 | 3e-07 |
| | 36 | NA | NA |

So there seems to be some problem here, as can be plainly seen in the above plot of residuals vs fit. All in all, this analysis is not very satisfactory: there appears to be some problems with the assumptions, and the assumption of no interaction between depth and date would appear to be invalid.

12.4. Nested designs

A common experimental design occurs when each major group (or treatment) is divided into randomly chosen subgroups. For example, a geneticist interested in the effects of genotype on desiccation resistance in fruit flies might conduct an experiment with larvae of three different genotypes. For each genotype (major group), she sets up three environmental chambers (sub-groups, replicates within groups) with a fixed temperature humidity regime, and in each chamber, she has five larvae for which she records the number of hours each larvae survived.

- The file `Nestdat.csv` contains the results of just such an experiment. The file lists three variables: genotype , chamber and survival . Run a nested ANOVA with survival as the dependent variable, genotype/chamber as the independent variables (this is the shorthand notation for a chamber effect nested under genotype).

```
nestdat <- read.csv("data/nestdat.csv")
nestdat$chamber <- as.factor(nestdat$chamber)
nestdat$genotype <- as.factor(nestdat$genotype)
anova.nested <- lm(survival ~ genotype / chamber, data = nestdat)
```

What do you conclude from this analysis? What analysis would (should) you do next? (Hint: if there is a non-significant effect of chambers within genotypes, then you can increase the power of between-genotype comparisons by pooling over chambers within genotypes, although not everyone (Dr. Rundle included) agrees with such pooling.) Do it! Make sure you check your assumptions!

Solution

```
anova(anova.nested)
```

| | Df | Sum Sq | Mean Sq | F value | Pr(>F) |
|------------------|----|------------|-------------|------------|-----------|
| genotype | 2 | 2952.22044 | 1476.110222 | 292.608079 | 0.0000000 |
| genotype:chamber | 6 | 40.65467 | 6.775778 | 1.343157 | 0.2638893 |
| Residuals | 36 | 181.60800 | 5.044667 | NA | NA |

Conditions d'applications du modèle anova.nested

```
par(mfrow = c(2, 2))
plot(anova.nested)
```

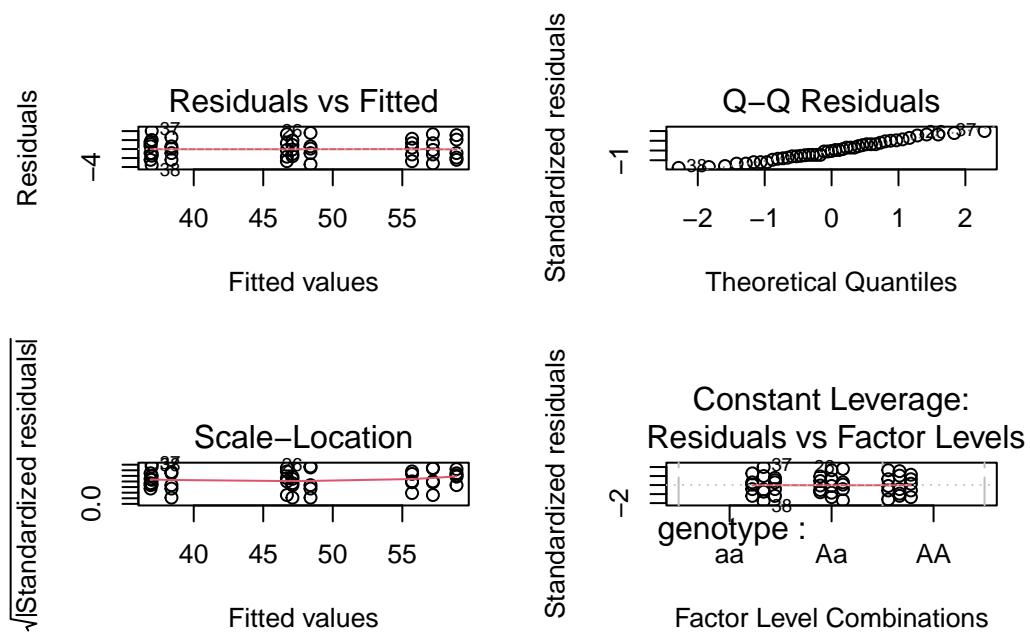


Figure 12.6.: Conditions d'applications du modèle anova.nested

We conclude from this analysis that there is no (significant) variation among chambers within genotypes, but that the null hypothesis that all genotypes have the same desiccation resistance (as measured by survival) is rejected (Test of genotype using MS genotype:chamber as denominator: $F = 1476.11/6.78 = 217.7153$, $P < 0.0001$). In other words, genotypes differ in their survival.

Since the chambers within genotypes effect is non-significant, we may want to pool over chambers to increase our

degrees of freedom:

```
anova.simple <- lm(survival ~ genotype, data = nestdat)
anova(anova.simple)
```

| | Df | Sum Sq | Mean Sq | F value | Pr(>F) |
|-----------|----|-----------|-------------|----------|--------|
| genotype | 2 | 2952.2204 | 1476.110222 | 278.9341 | 0 |
| Residuals | 42 | 222.2627 | 5.291968 | NA | NA |

Thus, we conclude that there is significant variation among the three genotypes in dessiccation resistance.

A box plot of survival across genotypes shows clearly that there is significant variation among the three genotypes in dessiccation resistance. This can be combined with a formal Tukey multiple comparison test:

```
par(mfrow = c(1, 1))
# Compute and plot means and Tukey CI
means <- glht(anova.simple, linfct = mcp(
  genotype =
  "Tukey"
))
cimeans <- cld(means)
# use sufficiently large upper margin
old.par <- par(mai = c(1, 1, 1.25, 1))
# plot
plot(cimeans, las = 1) # las option to put y-axis labels as God intended them
```

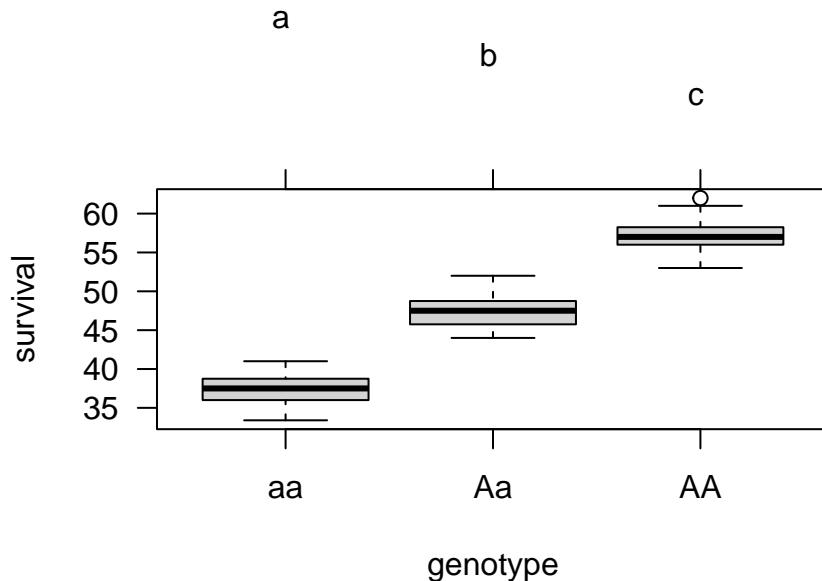


Figure 12.7.: Effet du genotype sur la résistance à la dessication avec un test de Tukey

So, we conclude from the Tukey analysis and plot that dessiccation resistance (R) , as measured by larval survival under hot, dry conditions, varies significantly among all three genotypes with $R(AA) > R(Aa) > R(aa)$.

Before concluding this, however, we must test the assumptions. Here are the residual plots and diagnostics for the one-way (unnested) design:

Solution

```
par(mfrow = c(2, 2))
plot(anova.simple)
```

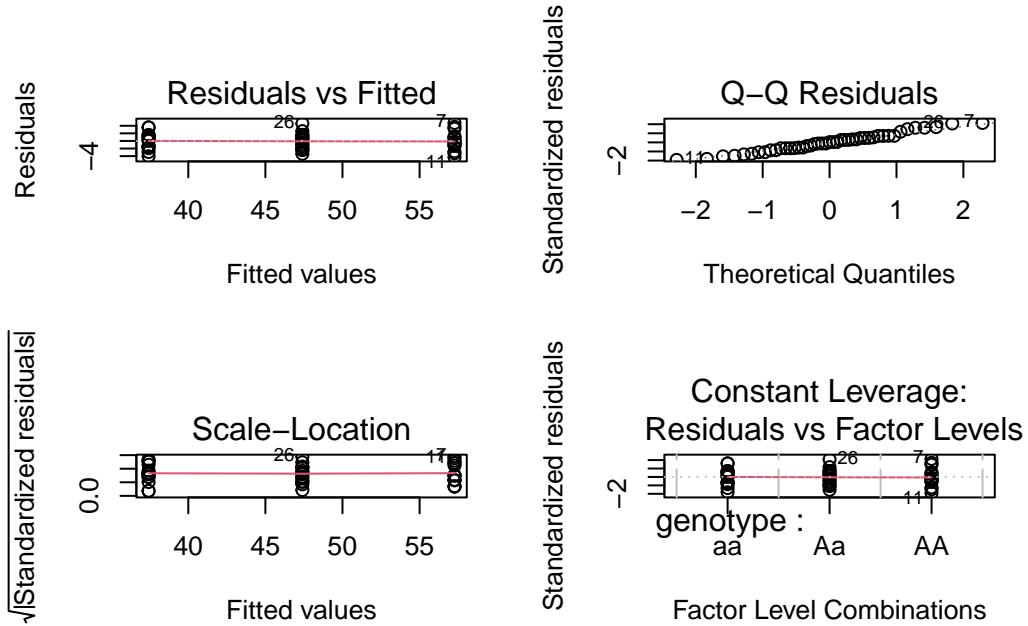


Figure 12.8.: Conditions d'applications du modèle anova.simple

So, all the assumptions appear to be valid, and the conclusion reached above still holds. Note that if you compare the residual mean squares of the nested and one-way ANOVAs (5.04 vs 5.29), they are almost identical. This is not surprising, given the small contribution of the chamber %in% genotype effect to the explained sum of squares.

12.5. Two-way non-parametric ANOVA

Two-way non-parametric ANOVA is an extension of the non-parametric one-way methods discussed previously. The basic procedure is to rank all the data in the sample from smallest to largest, then carry out a 2-way ANOVA on the ranks. This can be done either for replicated or unreplicated data.

Using the data file `Stu2wdat.csv`, do a two-factor ANOVA to examine the effects of `sex` and `location` on `rank(rdwght)`.

```
aov.rank <- aov(
  rank(rdwght) ~ sex * location,
  contrasts = list(
    sex = contr.sum, location = contr.sum
  ),
```

```
  data = Stu2wdat  
)
```

The Scheirer-Ray-Hare extension of the Kruskall-Wallis test is done by computing a statistic H given by the effect sums of squares (SS) divided by the total MS. The latter can be calculated as the variance of the ranks. We compute an H statistic for each term. The H-statistics are then compared to a theoretical χ^2 (chi-square) distribution using the command line: `pchisq(H, df, lower.tail = FALSE)`, where H and df are the calculated H-statistics and associated degrees of freedom, respectively.

- Use the ANOVA table based on ranks to test the effects of `sex` and on `rdwght`. What do you conclude? How does this result compare with the result obtained with the parametric 2-way ANOVA done before?

```
Anova(aov.rank, type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|--------------|-------------|-----|-------------|-----------|
| (Intercept) | 1499862.414 | 1 | 577.8672627 | 0.0000000 |
| sex | 58393.631 | 1 | 22.4979086 | 0.0000042 |
| location | 1128.212 | 1 | 0.4346776 | 0.5105359 |
| sex:location | 1229.819 | 1 | 0.4738251 | 0.4921091 |
| Residuals | 472383.499 | 182 | NA | NA |

To calculate the Scheirer-Ray-Hare extension to the Kruskall-Wallis test, you must first calculate the total mean square (MS), i.e. the variance of the ranked data. In this case, there are 186 observations, their ranks are therefore the series 1, 2, 3, ..., 186. The variance can be calculated simply as `var(1:186)` (Isn't R neat? Cryptic maybe, but neat). So we can compute the H statistic for each term:

```
Hsex <- 58394 / var(1:186)  
Hlocation <- 1128 / var(1:186)  
Hsexloc <- 1230 / var(1:186)
```

And convert these statistics into p-values:

```
# sex
Hsex

[1] 20.14628

pchisq(Hsex, 1, lower.tail = FALSE)

[1] 7.173954e-06

# location
Hlocation

[1] 0.3891668

pchisq(Hlocation, 1, lower.tail = FALSE)

[1] 0.5327377

# sex:location
Hsexloc

[1] 0.4243574

pchisq(Hsexloc, 1, lower.tail = FALSE)

[1] 0.5147707
```

Note that these results are the same as those obtained in our original two-way parametric ANOVA. Despite the reduced power, we still find significant differences between the sexes, but still no interaction and no effect due to location.

There is, however, an important difference. Recall that in the original parametric ANOVA, there was a significant effect of sex when we considered the problem as a Model I ANOVA. However, if we consider it as Model III, the significant sex effect could in principle disappear, because the df associated with the interaction MS are much smaller than the df associated with the Model I error MS. In this case, however, the interaction MS is about half that of the error MS. So, the significant sex effect becomes even more significant if we analyze the problem as a Model III ANOVA. Once again, we see the importance of specifying the appropriate ANOVA design.

12.6. Multiple comparisons

Further hypothesis testing in multiway ANOVAs depends critically on the outcome of the initial ANOVA. If you are interested in comparing groups of marginal means (that is, means of treatments for one factor pooled over levels of the other factor, e.g., between male and female sturgeon pooled over location), this can be done exactly as outlined for multiple comparisons for one-way ANOVAs. For comparison of individual cell means, you must specify the interaction as the group variable.

The file `wmcdat2.csv` shows measured oxygen consumption (`o2cons`) of two species (`species = A, B`) of limpets at three different concentrations of seawater (`conc = 100, 75, 50%`) taken from Sokal and Rohlf, 1995, p. 332.

- Run a 2-way factorial ANOVA on `wmcdat2` data, using `o2cons` as the dependent variable and `species` and `conc` as the factors. What do you conclude?

Solution

```
wmcdat2 <- read.csv("data/wmcdat2.csv")
wmcdat2$species <- as.factor(wmcdat2$species)
wmcdat2$conc <- as.factor(wmcdat2$conc)
anova.model5 <- lm(o2cons ~ species * conc, data = wmcdat2)
Anova(anova.model5, type = 3)
```

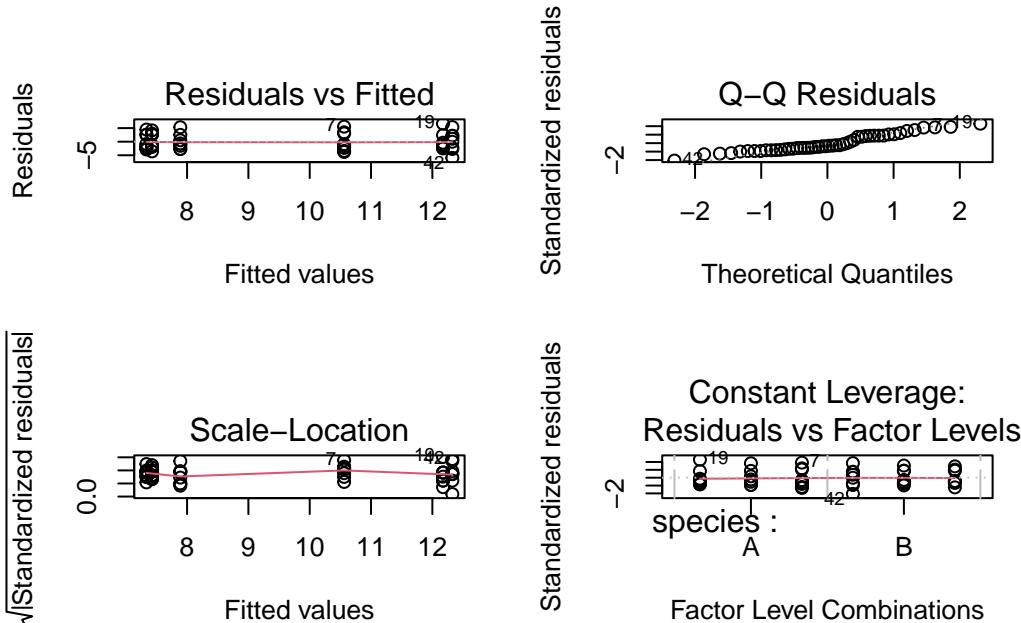
The ANOVA table is shown below. Technically, because the sample sizes in individual cells are rather small, this analysis should be repeated using a non-parametric ANOVA. For the moment, let's stick with the parametric analysis.

| | Sum Sq | Df | F value | Pr(>F) |
|--------------|-------------|----|-------------|-----------|
| (Intercept) | 1185.601512 | 1 | 124.0164931 | 0.0000000 |
| species | 0.093025 | 1 | 0.0097306 | 0.9218903 |
| conc | 74.896658 | 2 | 3.9171766 | 0.0275505 |
| species:conc | 23.926200 | 2 | 1.2513662 | 0.2965616 |
| Residuals | 401.521300 | 42 | | NA |

Look at the diagnostic plots:

Solution

```
par(mfrow = c(2, 2))
plot(anova.model5)
```



Homoscedasticity looks ok, but normality less so.. Testing for normality, we get:

Solution

```
shapiro.test(residuals(anova.model5))
```

```
Shapiro-Wilk normality test

data: residuals(anova.model5)
W = 0.93692, p-value = 0.01238
```

So there is evidence of non-normality, but otherwise everything looks O.K. Since the ANOVA is relatively robust with respect to non-normality, we proceed, but if we wanted to reassure ourselves, we could run a non-parametric ANOVA, and get the same answer.

- On the basis of the ANOVA results obtained above, which means would you proceed to compare? Why?

 Solution

Need to add an explanation here

Overall, we conclude that there are no differences among species, and that the effect of concentration does not depend on species (no interaction). Since there is no interaction and no main effect due to species, the only comparison of interest is among salinity concentrations:

```
# fit simplified model
anova.model6 <- aov(o2cons ~ conc, data = wmcdat2)
# Make Tukey multiple comparisons
TukeyHSD(anova.model6)
```

```
Tukey multiple comparisons of means
95% family-wise confidence level

Fit: aov(formula = o2cons ~ conc, data = wmcdat2)

$conc
      diff      lwr      upr      p adj
75-50 -4.63625 -7.321998 -1.9505018 0.0003793
100-50 -3.25500 -5.940748 -0.5692518 0.0141313
100-75  1.38125 -1.304498  4.0669982 0.4325855
```

```
par(mfrow = c(1, 1))
# Graph of all comparisons for conc
tuk <- glht(anova.model6, linfct = mcp(conc = "Tukey"))
# extract information
tuk.cld <- cld(tuk)
# use sufficiently large upper margin
old.par <- par(mai = c(1, 1, 1.25, 1))
# plot
plot(tuk.cld)
```

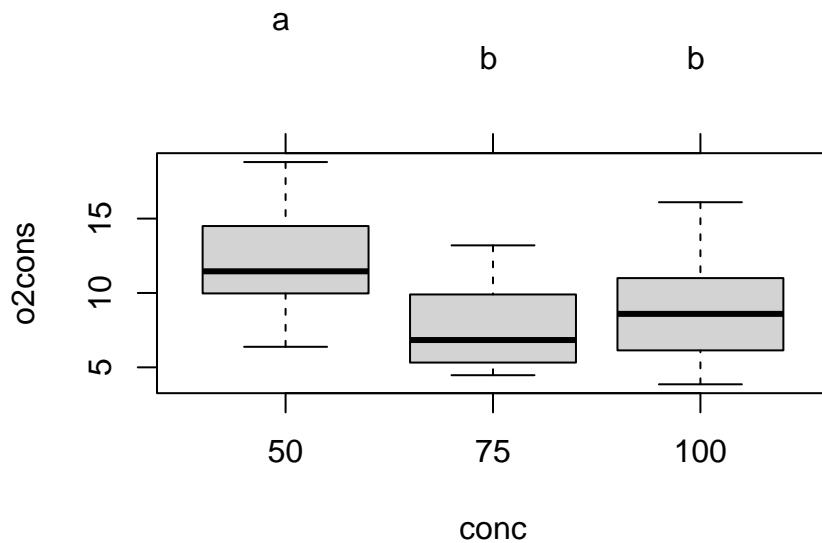


Figure 12.9.: Comparaison de Tukey des moyennes de consommation d'oxygène en fonction de la concentration

```
par(old.par)
```

So there is evidence of a significant difference in oxygen consumption at a reduction in salinity to 50% of regular seawater, but not at a reduction of only 25%.

- Repeat the analysis described above using `wmc2dat2.csv`. How do your results compare with those obtained for `wmcdat2.csv` ?

Solution

```
wmc2dat2 <- read.csv("data/wmc2dat2.csv")
wmc2dat2$species <- as.factor(wmc2dat2$species)
wmc2dat2$conc <- as.factor(wmc2dat2$conc)
anova.model7 <- lm(o2cons ~ species * conc, data = wmc2dat2)
```

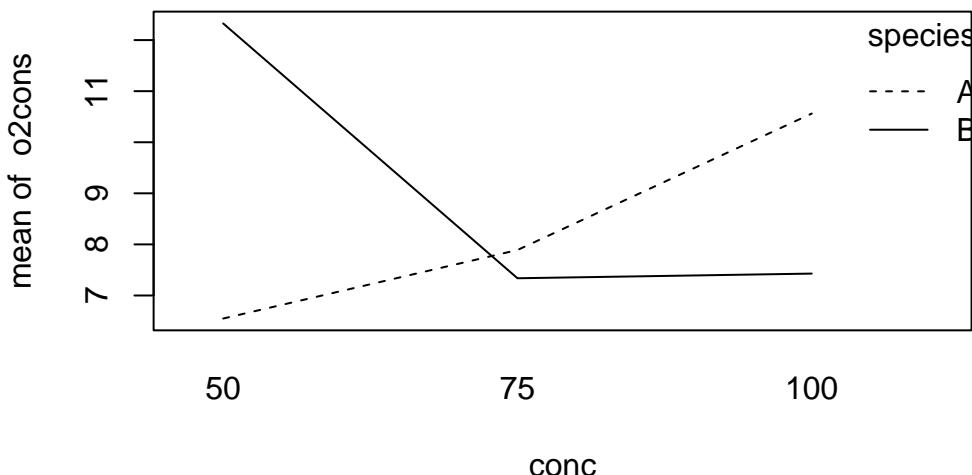
Using `wmc2dat2.csv`, we get:

| | Sum Sq | Df | F value | Pr(>F) |
|-------------|-----------|----|-----------|-----------|
| (Intercept) | 343.08901 | 1 | 36.213216 | 0.0000004 |
| species | 133.51802 | 1 | 14.092894 | 0.0005286 |

| | Sum Sq | Df | F value | Pr(>F) |
|--------------|-----------|----|----------|-----------|
| conc | 66.75916 | 2 | 3.523231 | 0.0385011 |
| species:conc | 168.15120 | 2 | 8.874222 | 0.0006101 |
| Residuals | 397.91380 | 42 | NA | NA |

Here there is a large interaction effect, and consequently, there is no point in comparing marginal means. This is made clear by examining an interaction plot:

```
with(wmc2dat2, interaction.plot(conc, species, o2cons))
```



- Working still with the wmc2dat2 data set, compare individual cell means (6 in all), with the Bonferroni adjustment. To do this, it is helpful to create a new variable to indicate all the combinations of species and conc:

```
wmc2dat2$species.conc <- as.factor(paste0(wmc2dat2$species, wmc2dat2$conc))
```

Then we can conduct pairwise bonferroni comparisons:

```
with(wmc2dat2, pairwise.t.test(o2cons, species.conc, p.adj = "bonf"))
```

Pairwise comparisons using t tests with pooled SD

data: o2cons and species.conc

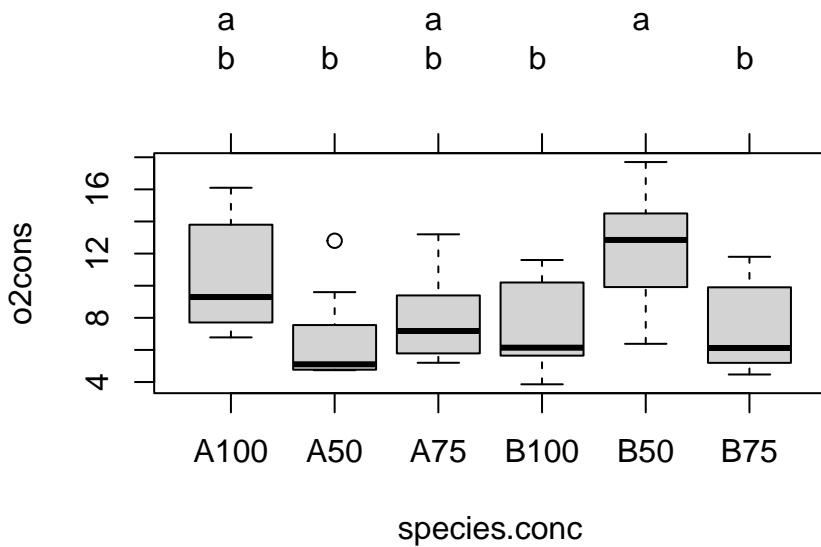
| | A100 | A50 | A75 | B100 | B50 |
|------|--------|--------|--------|--------|--------|
| A50 | 0.1887 | - | - | - | - |
| A75 | 1.0000 | 1.0000 | - | - | - |
| B100 | 0.7223 | 1.0000 | 1.0000 | - | - |
| B50 | 1.0000 | 0.0079 | 0.0929 | 0.0412 | - |
| B75 | 0.6340 | 1.0000 | 1.0000 | 1.0000 | 0.0350 |

P value adjustment method: bonferroni

These comparisons are a little more difficult to interpret, but the analysis essentially examines for differences among seawater concentrations within species A and for differences among concentrations within species B. We see here that the o2Cons at 50% seawater for species B is significantly different from that of 75% and 100% seawater for species B, whereas there are no significant differences in o2cons for species A across all seawater concentrations.

I find these outputs rather unsatisfying because they show only p-values, but no indication of effect size. One can get both the conclusion from the multiple comparison procedure and an indication of effect size from the graph produced with the following code:

```
# fit one-way anova comparing all combinations of species.conc combinations
anova.modelx <- aov(o2cons ~ species.conc, data = wmc2dat2)
tuk2 <- glht(anova.modelx, linfct = mcp(species.conc = "Tukey"))
# extract information
tuk2.cld <- cld(tuk2)
# use sufficiently large upper margin
old.par <- par(mai = c(1, 1, 1.25, 1))
# plot
plot(tuk2.cld)
```



```
par(old.par)
```

Note that in this analysis, we have used the error MS = 9.474 from the original model to contrast cell means. Recall, however, that this assumes that in fact we are dealing with a Model I ANOVA, which may or may not be the case (conc is certainly a fixed factor, but species might be either fixed or random).

12.7. Test de permutation pour l'ANOVA à deux facteurs de classification

When data do not meet the assumptions of the parametric analysis in two- and multiway ANOVA, as an alternative to the non-parametric ANOVA, it is possible to run permutation tests to calculate p-values. The lmPerm package does this easily.

```
#####
## lmPerm version of permutation test
library(lmPerm)
# for generality, copy desired dataframe to mydata
# and model formula to myformula
mydata <- Stu2wdat
myformula <- as.formula("rdwght ~ sex+location+sex:location")
# Fit desired model on the desired dataframe
```

```
mymodel <- lm(myformula, data = mydata)

# Calculate permutation p-value

anova(lmp(myformula, data = mydata, perm = "Prob", center = FALSE, Ca = 0.001))
```

`lmPerm` was orphaned for a while and the code below, while clunkier, provided an alternative way of doing it. You would have to adapt it for other situations.

```
#####
# Permutation test for two way ANOVA
# Ter Braak creates residuals from cell means and then permutes across
# all cells
# This can be accomplished by taking residuals from the full model
# modified from code written by David C. Howell
# http://www.uvm.edu/~dhowell/StatPages/More_Stuff/Permutation%20Anova/PermTestsAnova.html
nreps <- 500
dependent <- Stu2wdat$rdwght
factor1 <- as.factor(Stu2wdat$sex)
factor2 <- as.factor(Stu2wdat$location)
my.dataframe <- data.frame(dependent, factor1, factor2)
my.dataframe.noNA <- my.dataframe[complete.cases(my.dataframe), ]
mod <- lm(dependent ~ factor1 + factor2 + factor1:factor2,
           data = my.dataframe.noNA
)
res <- mod$residuals
TBint <- numeric(nreps)
TB1 <- numeric(nreps)
TB2 <- numeric(nreps)
ANOVA <- summary(aov(mod))
cat(
  " The standard ANOVA for these data follows ",
  "\n"
)
F1 <- ANOVA[[1]]$"F value"[1]
F2 <- ANOVA[[1]]$"F value"[2]
```

```
Finteract <- ANOVA[[1]]$"F value"[3]
print(ANOVA)
cat("\n")
cat("\n")
TBint[1] <- Finteract
for (i in 2:nreps) {
  newdat <- sample(res, length(res), replace = FALSE)
  modb <- summary(aov(newdat ~ factor1 + factor2 +
    factor1:factor2,
  data = my.dataframe.noNA
))
  TBint[i] <- modb[[1]]$"F value"[3]
  TB1[i] <- modb[[1]]$"F value"[1]
  TB2[i] <- modb[[1]]$"F value"[2]
}
probInt <- length(TBint[TBint >= Finteract]) / nreps
prob1 <- length(TB1[TB1 >= F1]) / nreps
prob2 <- length(TB2[TB2 >= F2]) / nreps
cat("\n")
cat("\n")
print("Resampling as in ter Braak with unrestricted sampling
of cell residuals. ")
cat(
  "The probability for the effect of Interaction is ",
  probInt, "\n"
)
cat(
  "The probability for the effect of Factor 1 is ",
  prob1, "\n"
)
cat(
  "The probability for the effect of Factor 2 is ",
  prob2, "\n"
```

)

12.8. Bootstrap for two-way ANOVA

In most cases, permutation tests will be more appropriate than bootstrap in ANOVA designs. However, for the sake of completeness, I have a snippet of code to do bootstrap for you::

```
#####
#####
# Bootstrap for two-way ANOVA
# You possibly want to edit bootfunction.mod1 to return other values
# Here it returns the standard coefficients of the fitted model
# Requires boot library
#
nreps <- 5000
dependent <- Stu2wdat$rdwght
factor1 <- as.factor(Stu2wdat$sex)
factor2 <- as.factor(Stu2wdat$location)
my.dataframe <- data.frame(dependent, factor1, factor2)
my.dataframe.noNA <- my.dataframe[complete.cases(my.dataframe), ]
library(boot)
# Fit model on observed data
mod1 <- aov(dependent ~ factor1 + factor2 + factor1:factor2,
  data = my.dataframe.noNA
)
#
#
#
#
# Bootstrap 1000 time using the residuals bootstrapping methods to
# keep the same unequal number of observations for each level of the indep. var.
fit <- fitted(mod1)
e <- residuals(mod1)
X <- model.matrix(mod1)
bootfunction.mod1 <- function(data, indices) {
```

```
y <- fit + e[indices]
bootmod <- lm(y ~ X)
coefficients(bootmod)
}

bootresults <- boot(my.dataframe.noNA, bootfunction.mod1,
R = 1000
)

bootresults
## Calculate 90% CI and plot bootstrap estimates separately for each model parameter
boot.ci(bootresults, conf = 0.9, index = 1)
plot(bootresults, index = 1)
boot.ci(bootresults, conf = 0.9, index = 3)
plot(bootresults, index = 3)
boot.ci(bootresults, conf = 0.9, index = 4)
plot(bootresults, index = 4)
boot.ci(bootresults, conf = 0.9, index = 5)
plot(bootresults, index = 5)
```

Chapter 13

Multiple regression

After completing this laboratory exercise, you should be able to:

- Use R to fit a multiple regression model, and compare the adequacy of several models using inferential and information theoretic criteria
- Use R to test hypotheses about the effects of different independent variables on the dependent variable of interest.
- Use R to evaluate multicollinearity among (supposedly) independent variables and its effects.
- Use R to do curvilinear (polynomial) regression.

13.1. R packages and data

For this lab you need:

- R packages:
 - ggplot2
 - car
 - lmtest
 - simpleboot
 - boot
 - MuMIn
- data files:
 - Mregdat.csv

13.2. Points to keep in mind

Multiple regression models are used in cases where there is one dependent variable and several independent, continuous variables. In many biological systems, the variable of interest may be influenced by several different factors, so that accurate description or prediction requires that several independent variables be included in the regression model. Before beginning, be aware that multiple regression takes time to learn well. Beginners should keep in mind several important points:

1. An overall regression model may be statistically significant even if none of the individual regression coefficients in the model are (caused by multicollinearity)
2. A multiple regression model may be “nonsignificant” even though some of the individual coefficients are “significant” (caused by overfitting)
3. Unless “independent” variables are uncorrelated in the sample, different model selection procedures may yield different results.

13.3. First look at the data

The file Mregdat.Rdata contains data collected in 30 wetlands in the Ottawa-Cornwall-Kingston area. The data included are

- the richness (number of species) of:
 - birds (`bird`, and its log transform `logbird`),
 - plants (`plant`, `logpl`),
 - mammals (`mammal`, `logmam`),
 - herptiles (`herptile`, `logherp`)
 - total species richness of all four groups combined (`totsp`, `logtot`)
- GPS coordinates of the wetland (`lat`, `long`)
- its area (`logarea`)
- the percentage of the wetland covered by water at all times during the year (`swamp`)
- the percentage of forested land within 1 km of the wetland (`cpfor2`)
- the density (in m/hectare) of hard-surface roads within 1 km of the wetland (`thtden`).

We will focus on herptiles for this exercise, so we better first have a look at how this variable is distributed and correlated to the potential independent variables:

```
mydata <- read.csv("data/Mregdat.csv")
scatterplotMatrix(
  ~ logherp + logarea + cpfor2 + thtden + swamp,
  regLine = TRUE, smooth = TRUE, diagonal = TRUE,
  data = mydata
)
```

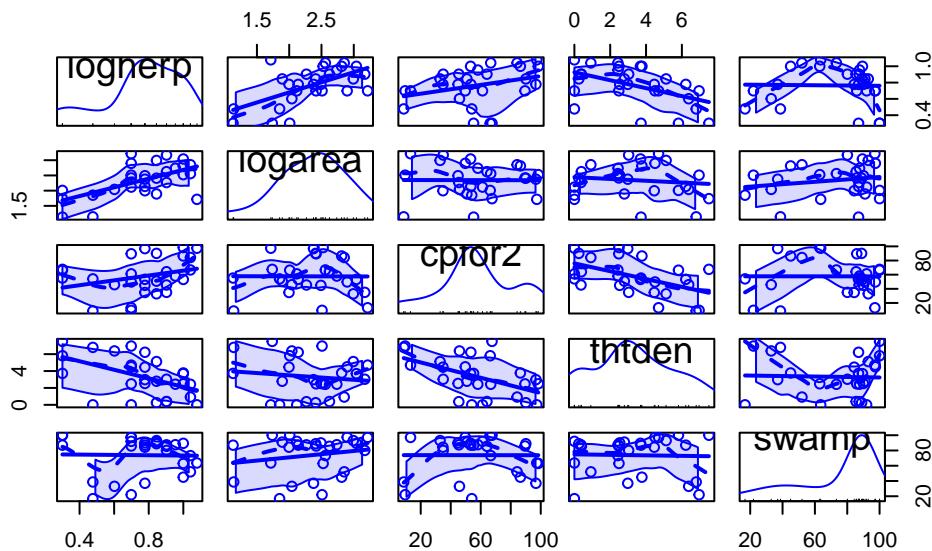


Figure 13.1.: Matrice de relation et densité pour la richesse spécifique des amphibiens et reptiles

13.4. Multiple regression models from scratch

We begin the multiple regression exercise by considering a situation with one dependent variable and three (possibly) independent variables. First, we will start from scratch and build a multiple regression model based on what we know from building simple regression models. Next, we will look at automated methods of building multiple regressions models using simultaneous, forward, and backward stepwise procedures.

🔥 Exercise

Using the subset of the Mregdat.csv data file, regress logherp on logarea.

On the basis of the regression, what do you conclude?

```
model_loga <- lm(logherp ~ logarea, data = mydata)
summary(model_loga)
```

Call:

```
lm(formula = logherp ~ logarea, data = mydata)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -0.38082 | -0.09265 | 0.00763 | 0.10409 | 0.46977 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|-------------|----------|------------|---------|--------------|
| (Intercept) | 0.18503 | 0.15725 | 1.177 | 0.249996 |
| logarea | 0.24736 | 0.06536 | 3.784 | 0.000818 *** |

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1856 on 26 degrees of freedom

(2 observations deleted due to missingness)

Multiple R-squared: 0.3552, Adjusted R-squared: 0.3304

F-statistic: 14.32 on 1 and 26 DF, p-value: 0.0008185

```
par(mfrow = c(2, 2))
plot(model_loga)
```

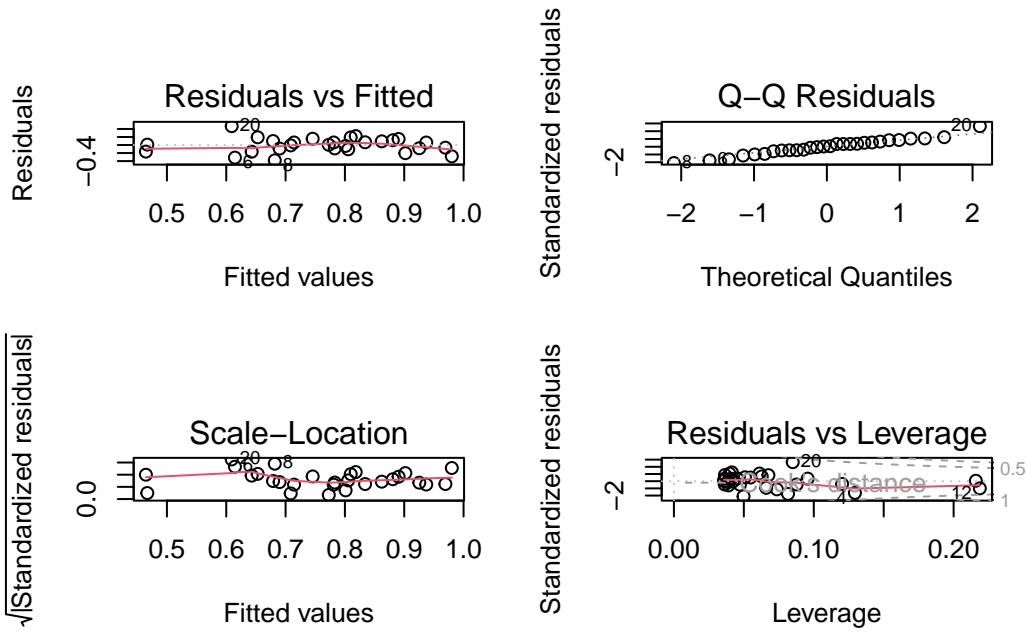


Figure 13.2.: Checking model assumptions for regression of *logherp* as a function of *logarea*

It looks like there is a positive relationship between herptile species richness and wetland area: the larger the wetland, the greater the number of species. Note, however, that about 2/3 of the observed variability in species richness among wetlands is not “explained” by wetland area ($R^2 = 0.355$). Residual analysis shows no major problems with normality, heteroscedasticity or independence of residuals.

Exercise

Rerun the above regression, this time replacing `logarea` with `cpfor2` as the independent variable, such that the expression in the formula field reads: `logherp ~ cpfor2`. What do you conclude?

Solution

```
model_logcp <- lm(logherp ~ cpfor2, data = mydata)
summary(model_logcp)
```

Call:

```
lm(formula = logherp ~ cpfor2, data = mydata)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----|----|--------|----|-----|
|-----|----|--------|----|-----|

```
-0.49095 -0.10266 0.05881 0.16027 0.25159
```

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) | | | | | | | |
|----------------|----------|------------|---------|--------------|------|-----|------|------|-----|-------|---|
| (Intercept) | 0.609197 | 0.104233 | 5.845 | 3.68e-06 *** | | | | | | | |
| cpfor2 | 0.002706 | 0.001658 | 1.632 | 0.115 | | | | | | | |
| --- | | | | | | | | | | | |
| Signif. codes: | 0 | '***' | 0.001 | '**' | 0.01 | '*' | 0.05 | '. ' | 0.1 | ' . ' | 1 |

Residual standard error: 0.2202 on 26 degrees of freedom

(2 observations deleted due to missingness)

Multiple R-squared: 0.09289, Adjusted R-squared: 0.058

F-statistic: 2.662 on 1 and 26 DF, p-value: 0.1148

According to this result, we would accept the null hypothesis, and conclude that there is no relationship between herptile density and the proportion of forest on adjacent lands. But what happens when we enter both variables into the regression simultaneously?

🔥 Exercise

Rerun the above regression one more time, this time adding both independent variables into the model at once, such that `logherp ~ logarea + cpfor2`. What do you conclude?

💡 Solution

```
model_mcp <- lm(logherp ~ logarea + cpfor2, data = mydata)
summary(model_mcp)
```

Call:

```
lm(formula = logherp ~ logarea + cpfor2, data = mydata)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -0.40438 | -0.11512 | 0.01774 | 0.08187 | 0.36179 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) | | | | | | | |
|----------------|----------|------------|---------|--------------|------|-----|------|------|-----|-----|---|
| (Intercept) | 0.027058 | 0.166749 | 0.162 | 0.872398 | | | | | | | |
| logarea | 0.247789 | 0.061603 | 4.022 | 0.000468 *** | | | | | | | |
| cpfor2 | 0.002724 | 0.001318 | 2.067 | 0.049232 * | | | | | | | |
| --- | | | | | | | | | | | |
| Signif. codes: | 0 | '***' | 0.001 | '**' | 0.01 | '*' | 0.05 | '. ' | 0.1 | ' ' | 1 |

Residual standard error: 0.175 on 25 degrees of freedom

(2 observations deleted due to missingness)

Multiple R-squared: 0.4493, Adjusted R-squared: 0.4052

F-statistic: 10.2 on 2 and 25 DF, p-value: 0.0005774

Now we reject both null hypotheses that the slope of the regression of logherp on logarea is zero and that the slope of the regression of logherp on cpfor2 is zero.

Why is cpfor2 a significant predictor of logherp in the combined model when it was not significant in the simple linear model? The answer lies in the fact that it is sometimes necessary to control for one variable in order to detect the effect of another variable. In this case, there is a significant relationship between logherp and logarea that masks the relationship between logherp and cpfor2. When both variables are entered into the model at once, the effect of logarea is controlled for, making it possible to detect a cpfor2 effect (and vice versa).

🔥 Exercise

Run another multiple regression, this time substituting thtden for cpfor2 as an independent variable (logherp ~ logarea + thtden).

💡 Solution

```
model_mden <- lm(logherp ~ logarea + thtden, data = mydata)
summary(model_mden)
```

Call:

```
lm(formula = logherp ~ logarea + thtden, data = mydata)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -0.31583 | -0.12326 | 0.02095 | 0.13201 | 0.31674 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) | | | | | | | |
|----------------|----------|------------|---------|--------------|------|-----|------|------|-----|-----|---|
| (Intercept) | 0.37634 | 0.14926 | 2.521 | 0.018437 * | | | | | | | |
| logarea | 0.22504 | 0.05701 | 3.947 | 0.000567 *** | | | | | | | |
| thtden | -0.04196 | 0.01345 | -3.118 | 0.004535 ** | | | | | | | |
| --- | | | | | | | | | | | |
| Signif. codes: | 0 | '***' | 0.001 | '**' | 0.01 | '*' | 0.05 | '. ' | 0.1 | ' ' | 1 |

Residual standard error: 0.1606 on 25 degrees of freedom

(2 observations deleted due to missingness)

Multiple R-squared: 0.5358, Adjusted R-squared: 0.4986

F-statistic: 14.43 on 2 and 25 DF, p-value: 6.829e-05

In this case we reject the null hypotheses that there are no effects of wetland area (`logarea`) and road density (`thtden`) on herptile richness (`logherp`). Note here that road density has a negative effect on richness, whereas wetland area and forested area (`cpfor2`; results from previous regression) both have positive effects on herptile richness.

The R^2 of this model is even higher than the previous multiple regression model, reflecting a higher correlation between `logherp` and `thtden` than between `logherp` and `cpfor2` (if you run a simple regression between `logherp` and `thtden` and compare it to the `cpfor2` regression you should be able to detect this).

Thus far, it appears that herptile richness is related to wetland area (`logarea`), road density (`thtden`), and possibly forest cover on adjacent lands (`cpfor2`). But, does it necessarily follow that if we build a regression model with all three independent variables, that all three will show significant relationships? No, because we have not yet examined the relationship between `Logarea`, `cpfor2` and `thtden`. Suppose, for example, two of the variables (say, `cpfor2` and `thtden`) are perfectly correlated. Then the `thtden` effect is nothing more than the `cpfor2` effect (and vice versa), so that once we include one or the other in the regression model, none of the remaining variability would be explained by the third variable.

🔥 Exercise

Fit a regression model with logherp as the dependent variable and logarea , cpfor2 and thtden as the independent variables. What do you conclude?

💡 Solution

```
model_mtri <- lm(logherp ~ logarea + cpfor2 + thtden, data = mydata)
summary(model_mtri)
```

Call:

```
lm(formula = logherp ~ logarea + cpfor2 + thtden, data = mydata)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -0.30729 | -0.13779 | 0.02627 | 0.11441 | 0.29582 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|----------------|-----------|------------|---------|--------------|
| (Intercept) | 0.284765 | 0.191420 | 1.488 | 0.149867 |
| logarea | 0.228490 | 0.057647 | 3.964 | 0.000578 *** |
| cpfor2 | 0.001095 | 0.001414 | 0.774 | 0.446516 |
| thtden | -0.035794 | 0.015726 | -2.276 | 0.032055 * |
| --- | | | | |
| Signif. codes: | 0 *** | 0.001 ** | 0.01 * | 0.05 . |
| | '' | ' | ' | ' |

Residual standard error: 0.1619 on 24 degrees of freedom

(2 observations deleted due to missingness)

Multiple R-squared: 0.5471, Adjusted R-squared: 0.4904

F-statistic: 9.662 on 3 and 24 DF, p-value: 0.0002291

Several things to note here:

1. The regression coefficient for cpfor2 has become non-significant: once the variability explained by logarea and thtden is removed, a non-significant part of the remaining variability is explained by cpfor2.

2. The R^2 for this model (.547 is only marginally larger than the R^2 for the model with only logarea and thtden (.536, which is again consistent with the non-significant coefficient for cpfor2.

Note also that although the regression coefficient for thtden has not changed much from that obtained when just thtden and logarea were included in the fitted model (-.036 vs -.042, the standard error for the regression coefficient for thtden has increased slightly, meaning the estimate is less precise. If the correlation between thtden and cpfor2 was greater, the change in precision would also be greater.

We can compare the fit of the last two models (*i.e.*, the model with all 3 variables and the model with only logarea and thtden to decide which model is best to include.

```
anova(model_mtri, model_mden)
```

| Res.Df | RSS | Df | Sum of Sq | F | Pr(>F) |
|--------|-----------|----|------------|-----------|-----------|
| 24 | 0.6293717 | NA | NA | NA | NA |
| 25 | 0.6450798 | -1 | -0.0157081 | 0.5990024 | 0.4465163 |

Note that this is the identical result we obtained via the t-test of the effect of cpfor2 in the model with all 3 variables above as they are testing the same thing (this should make sense to you). From this analysis, we would conclude that the full model with all three variables included does not offer a significant improvement in fit over the model with only logarea and thtden. This isn't surprising given that we already know that we cannot reject the null hypothesis of no effect of cpfor2 in the full model. Overall, we would conclude, on the basis of these analyses, that:

- Given the three variables thtden , logarea and cpfor2 , the best model is one that includes the first two variables.
- There is evidence of a negative relationship between herptile richness and the density of roads on adjacent lands.
- There is evidence that the larger the wetland area, the greater the herptile species richness. Note that by "best", I don't mean the best possible model, I mean the best one given the three predictor variables we started with. It seems pretty clear that there are other factors controlling richness in wetlands, since even with the "best" model, almost half of the variability in richness is unexplained.

13.5. Stepwise multiple regression procedures

There are a number of techniques available for selecting the multiple regression model that best suits your data. When working with only three independent variables it is often sufficient to work through the different combinations of possible variables yourself, until you are satisfied you have fit the best model. This is, essentially, what we did in the first section of this lab. However, the process can become tedious when dealing with numerous independent variables, and you may find an automatic procedure for fitting models to be easier to work with.

Stepwise regression in R relies on the Akaike Information Criterion, as a measure of goodness of fit

$$AIC = 2k + 2\ln(L)$$

where k is the number of regressors, and L is the maximized value of the likelihood function for the model). This is a statistic that rewards prediction precision while penalizing model complexity. If a new model has an AIC lower than that of the current model, the new model is a better fit to the data.

🔥 Exercise

Still working with the Mregdat data, run a stepwise multiple regression on the same set of variables:

```
# Stepwise Regression
step_mtri <- step(model_mtri, direction = "both")
```

Start: AIC=-98.27

`logherp ~ logarea + cpfor2 + thtden`

| | Df | Sum of Sq | RSS | AIC |
|-----------|----|-----------|---------|---------|
| - cpfor2 | 1 | 0.01571 | 0.64508 | -99.576 |
| <none> | | | 0.62937 | -98.267 |
| - thtden | 1 | 0.13585 | 0.76522 | -94.794 |
| - logarea | 1 | 0.41198 | 1.04135 | -86.167 |

Step: AIC=-99.58

`logherp ~ logarea + thtden`

| | Df | Sum of Sq | RSS | AIC |
|--|----|-----------|-----|-----|
|--|----|-----------|-----|-----|

```

<none>           0.64508 -99.576
+ cpfor2    1   0.01571  0.62937 -98.267
- thtden    1   0.25092  0.89600 -92.376
- logarea    1   0.40204  1.04712 -88.013

```

```
step_mtri$anova # display results
```

| Step | Df | Deviance | Resid. Df | Resid. Dev | AIC |
|----------|----|-----------|-----------|------------|-----------|
| | NA | NA | 24 | 0.6293717 | -98.26666 |
| - cpfor2 | 1 | 0.0157081 | 25 | 0.6450798 | -99.57640 |

Examining the output, we find:

1. R calculated the AIC for the starting model (here the full model with the 3 independent variables).
2. The AIC for models where terms are deleted. Note here that the only way to reduce the AIC is to drop 2.
3. The AIC for models where terms are added or deleted from the model selected in the first step (i.e. `logherp ~ logarea + thtden`). Note that none of these models are better.

Instead of starting from the full (saturated) model and removing and possibly re-adding terms (i.e. `direction = "both"`), one can start from the null model and only add terms:

```

# Forward selection approach
model_null <- lm(logherp ~ 1, data = mydata)
step_f <- step(
  model_null,
  scope = ~ . + logarea + cpfor2 + thtden, direction = "forward"
)

```

Start: AIC=-82.09

`logherp ~ 1`

| | Df | Sum of Sq | RSS | AIC |
|-----------|----|-----------|--------|---------|
| + logarea | 1 | 0.49352 | 0.8960 | -92.376 |
| + thtden | 1 | 0.34241 | 1.0471 | -88.013 |

```
+ cpfor2 1 0.12907 1.2605 -82.820
<none>           1.3895 -82.091
```

Step: AIC=-92.38

logherp ~ logarea

| | Df | Sum of Sq | RSS | AIC |
|----------|----|-----------|---------|---------|
| + thtden | 1 | 0.25093 | 0.64508 | -99.576 |
| + cpfor2 | 1 | 0.13078 | 0.76522 | -94.794 |
| <none> | | 0.89600 | -92.376 | |

Step: AIC=-99.58

logherp ~ logarea + thtden

| | Df | Sum of Sq | RSS | AIC |
|----------|----|-----------|---------|---------|
| <none> | | 0.64508 | -99.576 | |
| + cpfor2 | 1 | 0.015708 | 0.62937 | -98.267 |

```
step_f$anova # display results
```

| Step | Df | Deviance | Resid. Df | Resid. Dev | AIC |
|-----------|----|-----------|-----------|------------|-----------|
| | NA | NA | 27 | 1.3895281 | -82.09073 |
| + logarea | -1 | 0.4935233 | 26 | 0.8960048 | -92.37639 |
| + thtden | -1 | 0.2509250 | 25 | 0.6450798 | -99.57640 |

You should first notice that the final result is the same as the default stepwise regression and as what we got building the model from scratch. In forward selection, R first fits the least complex model (i.e., with only an intercept), and then adds variables, one by one, according to AIC statistics. Thus, in the above example, the model was first fit with only an intercept. Next, `logarea` was added, followed by `thtden`. `cpfor2` was not added because it would make AIC increase to above that of the model fit with the first two variables. Generally speaking, when doing multiple regressions, it is good practice to try several different methods (e.g. all regressions, stepwise, and backward elimination, etc.) and see whether you get the same results. If you don't, then the "best" model may not be so obvious, and you will have to think very carefully about the inferences you draw. In this case, regardless of whether we use automatic, or forward/backward stepwise regression, we arrive at the same model.

When doing multiple regression, always bear in mind the following:

1. Different procedures may produce different “best” models, i.e. the “best” model obtained using forward stepwise regression needn’t necessarily be the same as that obtained using backward stepwise. It is good practice to try several different methods and see whether you end up with the same result. If you don’t, it is almost invariably due to multicollinearity among the independent variables.
2. Be wary of stepwise regression. As the authors of SYSTAT, another commonly used statistical package, note:

Stepwise regression is probably the most abused computerized statistical technique ever devised. If you think you need automated stepwise regression to solve a particular problem, you probably don’t. Professional statisticians rarely use automated stepwise regression because it does not necessarily find the “best” fitting model, the “real” model, or alternative “plausible” models. Furthermore, the order in which variables enter or leave a stepwise program is usually of no theoretical significance. You are always better off thinking about why a model could generate your data and then testing that model.

3. Remember that just because there is a significant regression of Y on X doesn’t mean that X causes Y: correlation does not imply causation!

13.6. Detecting multicollinearity

Multicollinearity is the presence of correlations among independent variables. In extreme cases (perfect collinearity) it will prevent you from fitting some models.

Warning

When collinearity is not perfect, it reduces your ability to test for the effect of individual variables, but does not affect the ability of the model to predict.

The help file for the HH  package contains this clear passage about one of the indices of multicollinearity, the variance inflation factors:

A simple diagnostic of collinearity is the variance inflation factor, VIF one for each regression coefficient (other than the intercept). Since the condition of collinearity involves the predictors but not the response, this measure is a function of the X’s but not of Y. The VIF for predictor i is

$$1/(1 - R_i^2)$$

where R_i^2 is the R^2 from a regression of predictor i against the remaining predictors. If R_i^2 is close to 1, this means that predictor i is well explained by a linear function of the remaining predictors, and, therefore, the presence of predictor i in the model is redundant. Values of VIF exceeding 5 are considered evidence of collinearity: The information carried by a predictor having such a VIF is contained in a subset of the remaining predictors. If, however, all of a model's regression coefficients differ significantly from 0 (p-value < .05), a somewhat larger VIF may be tolerable.

VIFs indicate by how much the variance of each regression coefficient is increased by the presence of collinearity.

Note

There are several `vif()` functions (I know of at least three in the packages `car`, `HH` and `DAAG`) and I do not know if and how they differ.

To quantify multicollinearity, one can simply call the `vif()` function from the package `car`:

```
library(car)
vif(model_mtri)
```

```
logarea    cpfor2    thtden
1.022127  1.344455  1.365970
```

Here there is no evidence that multicollinearity is a problem since all vif are close to 1.

13.7. Polynomial regression

In the regression models considered so far, we have assumed that the relationship between the dependent and independent variables is linear. If not, in some cases it can be made linear by transforming one or both variables. On the other hand, for many biological relationships no transformation in the world will help, and we are forced to go with some sort of non-linear regression method.

The simplest type of nonlinear regression method is polynomial regression, in which you fit regression models that include independent variables raised to some power greater than one, e.g. X^2 , X^3 , etc.

Exercise

Plot the relationship between the residuals of the `logherp ~ logarea` regression and `swamp`.

💡 Solution

```
# problème avec les données de manquantes dans logherp
mysub <- subset(mydata, !is.na(logherp))
# ajouter les résidus dans les données
mysub$resloga <- residuals(model_loga)
ggplot(data = mysub, aes(y = resloga, x = swamp)) +
  geom_point() +
  geom_smooth()
```

`geom_smooth()` using method = 'loess' and formula = 'y ~ x'

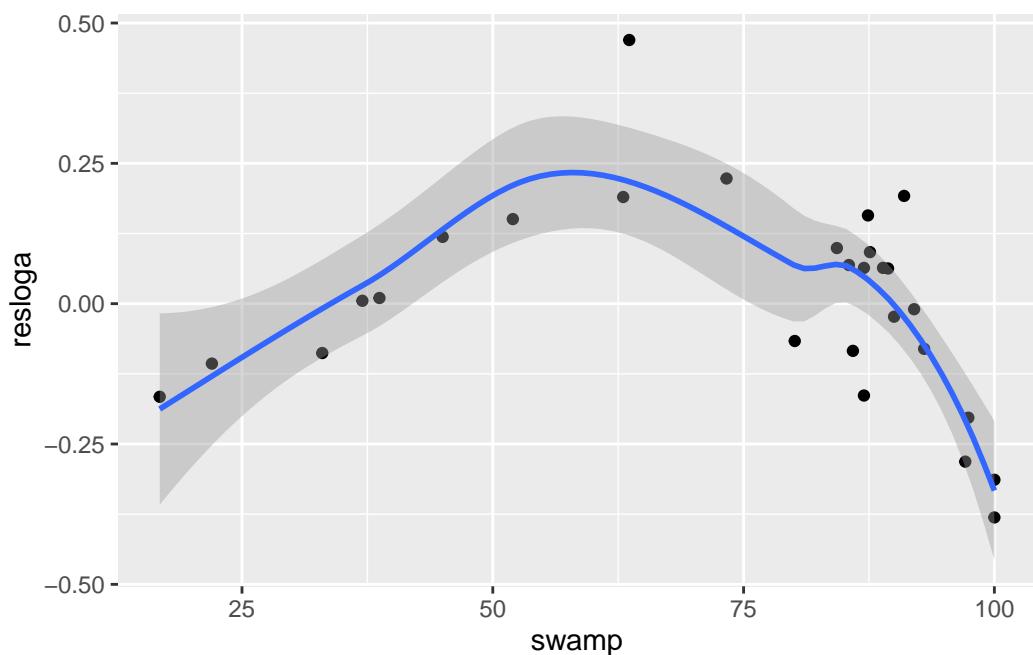


Figure 13.3.: Relation entre swamp et les résidus de la régression entre logherp et logarea

Visual inspection of this graph suggests that there is a strong, but highly nonlinear, relationship between these two variables.

🔥 Exercise

Try regressing the residuals of the `logherp ~ logarea` regression on `swamp`. What do you conclude?

Solution

```
model_resloga <- lm(resloga ~ swamp, mysub)
summary(model_resloga)
```

Call:

```
lm(formula = resloga ~ swamp, data = mysub)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -0.35088 | -0.13819 | 0.00313 | 0.10849 | 0.45802 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|-------------|-----------|------------|---------|----------|
| (Intercept) | 0.084571 | 0.109265 | 0.774 | 0.446 |
| swamp | -0.001145 | 0.001403 | -0.816 | 0.422 |

Residual standard error: 0.1833 on 26 degrees of freedom

Multiple R-squared: 0.02498, Adjusted R-squared: -0.01252

F-statistic: 0.666 on 1 and 26 DF, p-value: 0.4219

In other words, the fit is terrible, even though you can see from the graph that there is in fact quite a strong relationship between the two - it's just that it is a non-linear relationship. (If you look at model assumptions for this model, you will see strong evidence of nonlinearity, as expected) The pattern might be well described by a quadratic relation.

Exercise

Rerun the above regression but add a second term in the Formula field to represent swamp^2 . If you simply add swamp^2 in the model R won't fit a quadratic effect, you need to use the function `I()` which indicates that the formula within should be evaluated before fitting the model.

The expression should appear as:

residuals `swamp + I(swamp2)`

What do you conclude? What does examination of the residuals from this multiple regression tell you?

Solution

```
model_resloga2 <- lm(resloga ~ swamp + I(swamp^2), mysub)
summary(model_resloga2)
```

Call:

```
lm(formula = resloga ~ swamp + I(swamp^2), data = mysub)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|-----------|----------|----------|----------|
| -0.181185 | -0.085350 | 0.007377 | 0.067327 | 0.242455 |

Coefficients:

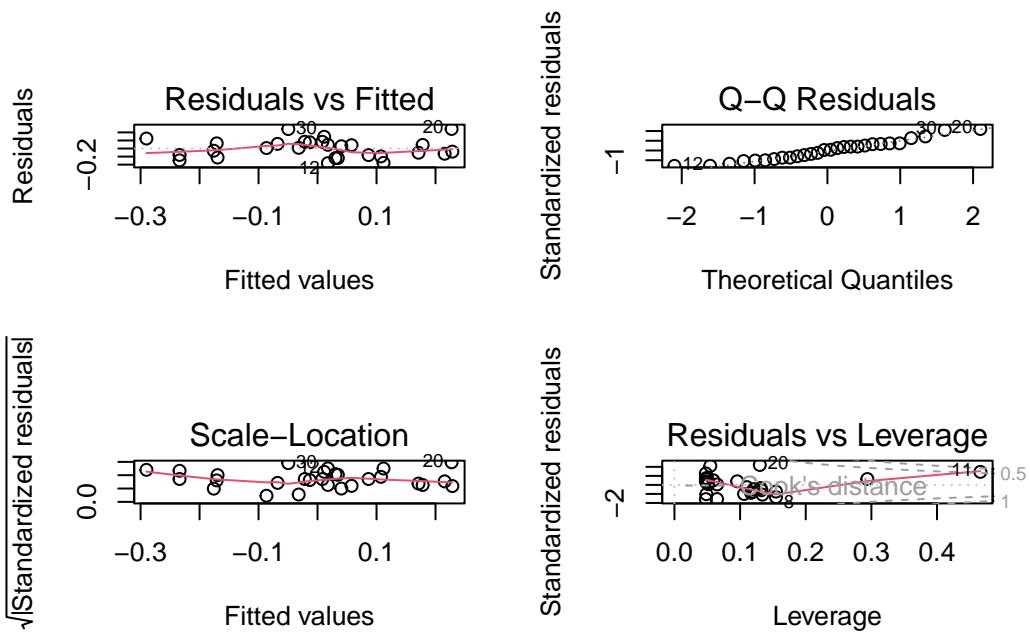
| | Estimate | Std. Error | t value | Pr(> t) | | | | | | | |
|----------------|------------|------------|---------|--------------|------|-----|------|------|-----|-----|---|
| (Intercept) | -7.804e-01 | 1.569e-01 | -4.975 | 3.97e-05 *** | | | | | | | |
| swamp | 3.398e-02 | 5.767e-03 | 5.892 | 3.79e-06 *** | | | | | | | |
| I(swamp^2) | -2.852e-04 | 4.624e-05 | -6.166 | 1.90e-06 *** | | | | | | | |
| --- | | | | | | | | | | | |
| Signif. codes: | 0 | '***' | 0.001 | '**' | 0.01 | '*' | 0.05 | '. ' | 0.1 | ' ' | 1 |

Residual standard error: 0.1177 on 25 degrees of freedom

Multiple R-squared: 0.6132, Adjusted R-squared: 0.5823

F-statistic: 19.82 on 2 and 25 DF, p-value: 6.972e-06

```
par(mfrow = c(2, 2))
plot(model_resloga2)
```



It is clear that once the effects of area are controlled for, a considerable amount of the remaining variability in herptile richness is explained by `swamp`, in a nonlinear fashion. If you examine model assumptions, you will see that compared to the linear model, the fit is much better.

Based on the results from the above analyses, how would you modify the regression model arrived at above? What, in your view, is the “best” overall model? Why? How would you rank the various factors in terms of their effects on herptile species richness?

In light of these results, we might want to try and fit a model which includes `logarea`, `thtden`, `cpfor2`, `swamp` and `swamp^2`:

Solution

```
model_poly1 <- lm(
  logherp ~ logarea + cpfor2 + thtden + swamp + I(swamp^2),
  data = mydata
)
summary(model_poly1)
```

Call:

```
lm(formula = logherp ~ logarea + cpfor2 + thtden + swamp + I(swamp^2),
```

```

data = mydata)

Residuals:
    Min      1Q   Median      3Q     Max 
-0.201797 -0.056170 -0.002072  0.051814  0.205626 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -3.203e-01  1.813e-01 -1.766   0.0912 .  
logarea      2.202e-01  3.893e-02  5.656 1.09e-05 *** 
cpfor2       -7.864e-04  9.955e-04 -0.790   0.4380    
thtden      -2.929e-02  1.048e-02 -2.795   0.0106 *  
swamp        3.113e-02  5.898e-03  5.277 2.70e-05 *** 
I(swamp^2)  -2.618e-04  4.727e-05 -5.538 1.45e-05 *** 
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1072 on 22 degrees of freedom
(2 observations deleted due to missingness)
Multiple R-squared:  0.8181,    Adjusted R-squared:  0.7767 
F-statistic: 19.78 on 5 and 22 DF,  p-value: 1.774e-07

```

Note that on the basis of this analysis, we could potentially drop cpfor2 and refit using the remaining variables:

Solution

```

model_poly2 <- lm(
  logherp ~ logarea + thtden + swamp + I(swamp^2),
  data = mydata
)
summary(model_poly2)

```

Call:

```
lm(formula = logherp ~ logarea + thtden + swamp + I(swamp^2),
```

```

data = mydata)

Residuals:
    Min      1Q  Median      3Q     Max 
-0.19621 -0.05444 -0.01202  0.07116  0.21295 

Coefficients:
                Estimate Std. Error t value Pr(>|t|)    
(Intercept) -3.461e-01  1.769e-01 -1.957   0.0626 .  
logarea       2.232e-01  3.842e-02  5.810 6.40e-06 *** 
thtden        -2.570e-02  9.364e-03 -2.744   0.0116 *  
swamp         2.956e-02  5.510e-03  5.365 1.89e-05 *** 
I(swamp^2)   -2.491e-04  4.409e-05 -5.649 9.46e-06 *** 
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1063 on 23 degrees of freedom
(2 observations deleted due to missingness)
Multiple R-squared:  0.8129,    Adjusted R-squared:  0.7804 
F-statistic: 24.98 on 4 and 23 DF,  p-value: 4.405e-08

```

How about multicollinearity in this model?

```
vif(model_poly2)
```

| | logarea | thtden | swamp | I(swamp^2) |
|--|----------|----------|-----------|------------|
| | 1.053193 | 1.123491 | 45.845845 | 45.656453 |

VIF for the two swamp terms are much higher than the standard threshold of 5. However, this is expected for polynomial terms, and not really a concern given that both terms are highly significant in the model. The high VIF means that these two coefficients are not estimated precisely, but using both in the model still allows to make a good prediction (i.e. account for the response to swamp).

13.8. Checking assumptions of a multiple regression model

All the model selection techniques or the manual model crafting assumes that the standard assumptions (independence, normality, homoscedasticity, linearity) are met. Given that a large number of models can be fitted, it may seem that testing the assumptions at each step would be an herculean task. However, it is generally sufficient to examine the residuals of the full (saturated) model and of the final model. Terms not contributing significantly to the fit do not affect residuals much, and therefore, the residuals to the full model, or the residuals to the final model, are generally sufficient.

Let's have a look at the diagnostic plots for the final model. Here we use the `check_model()` function from the `performance` .

Solution

```
library(performance)
check_model(model_poly2)
```

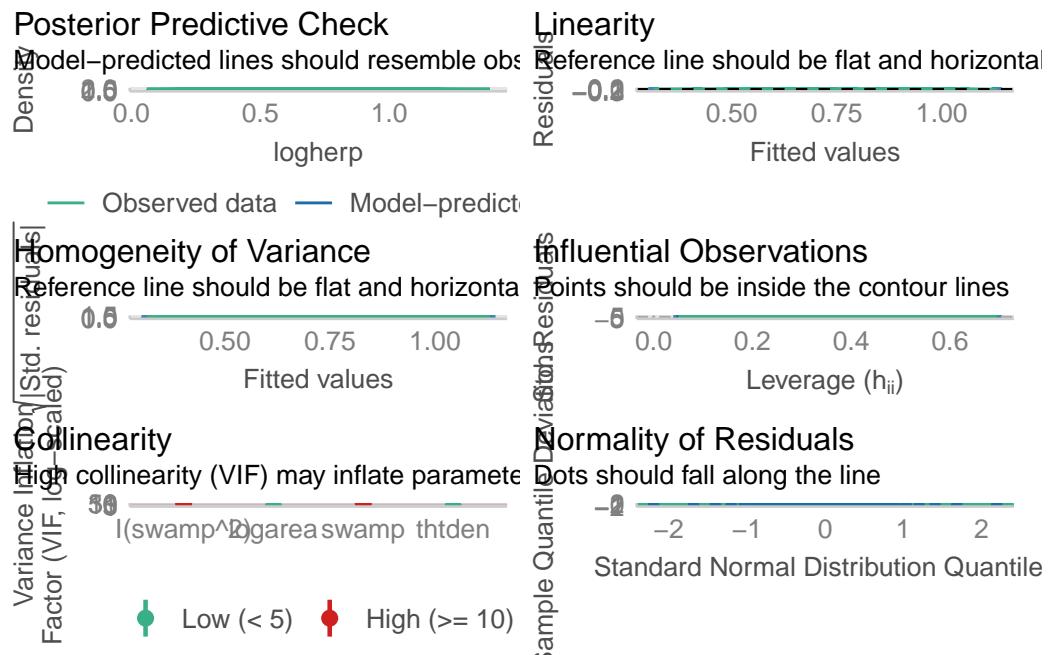


Figure 13.4.: Conditions d'application du modèle `model_poly2`

Alternatively it can be done with the classic method

 Code

```
par(mfrow = c(2, 2))
plot(model_poly2)
```

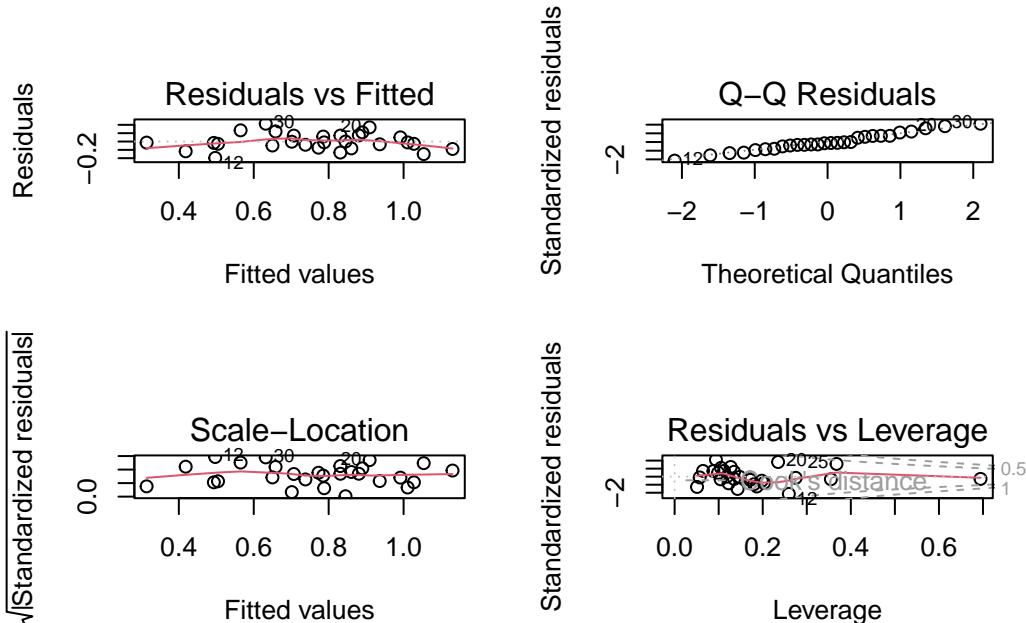


Figure 13.5.: Conditions d'application du modèle `model_poly2`

Everything looks about right here. For the skeptic, let's run the formal tests.

```
shapiro.test(residuals(model_poly2))
```

Shapiro-Wilk normality test

```
data: residuals(model_poly2)
W = 0.9837, p-value = 0.9278
```

The residuals do not deviate from normality. Good.

```
library(lmtest)
bptest(model_poly2)
```

```
studentized Breusch-Pagan test
```

```
data: model_poly2  
BP = 3.8415, df = 4, p-value = 0.4279
```

No deviation from homoscedasticity either. Good.

```
dwtest(model_poly2)
```

```
Durbin-Watson test
```

```
data: model_poly2  
DW = 1.725, p-value = 0.2095  
alternative hypothesis: true autocorrelation is greater than 0
```

No serial correlation in the residuals, so no evidence of non-independence.

```
resettest(model_poly2, type = "regressor", data = mydata)
```

```
RESET test
```

```
data: model_poly2  
RESET = 0.9823, df1 = 8, df2 = 15, p-value = 0.4859
```

And no significant deviation from linearity. So it seems that all is fine.

13.9. Visualizing effect size

How about effect size? How is that measured or viewed? The regression coefficients can be used to measure effect size, although it may be better to standardize them so that they become independent of measurement units. But a graph is often useful as well. In this context, some of the most useful graphs are called partial residual plots (or

component + residual plots). These plots show how the dependent variable, corrected for other variables in the model, varies with each individual variable. Let's have a look:

```
# Evaluate visually linearity and effect size
# component + residual plot
crPlots(model_poly2)
```

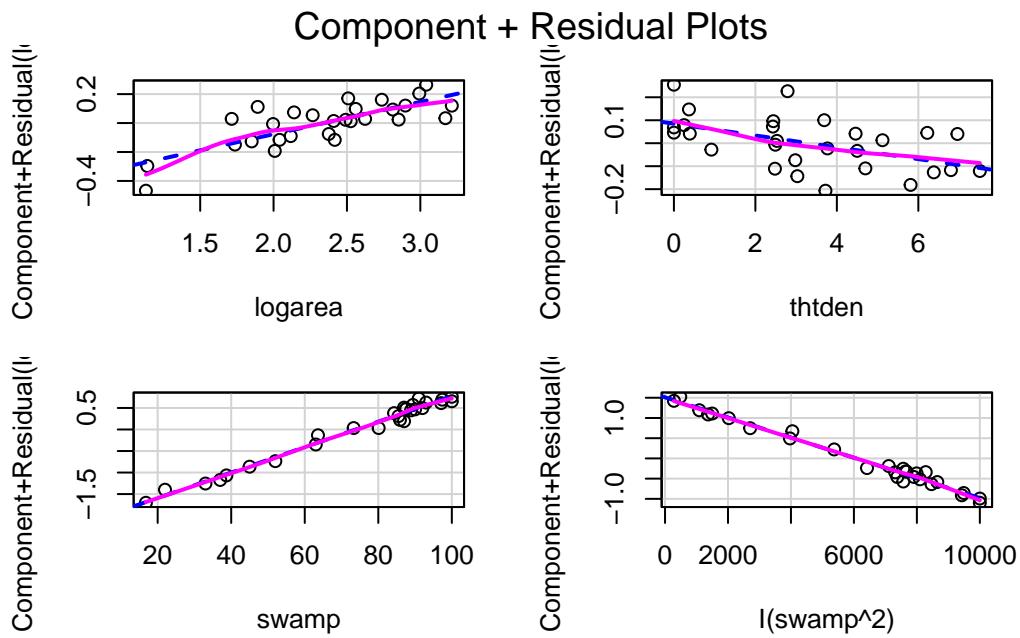


Figure 13.6.: Graphiques de résidus partiels du modèle `model_poly2`

Note that the vertical scale varies among plots. For `thtden`, the dependent variable ($\log_{10}(\text{herptile richness})$) varies by about 0.4 units over the range of `thtden` in the sample. For `logarea`, the variation is about 0.6 log units. For `swamp`, it is a bit tricky since there are two terms and they have opposite effect (leading to a peaked relationship), so the plots are less informative. However, there is no deviation from linearity to be seen.

To illustrate what these graphs would look like if there was deviation from linearity, let's drop `swamp^2` term and produce the graphs and run the RESET test

Solution

```
model_nopoly <- lm(
  logherp ~ logarea + thtden + swamp,
  data = mydata
)
crPlots(model_nopoly)
```

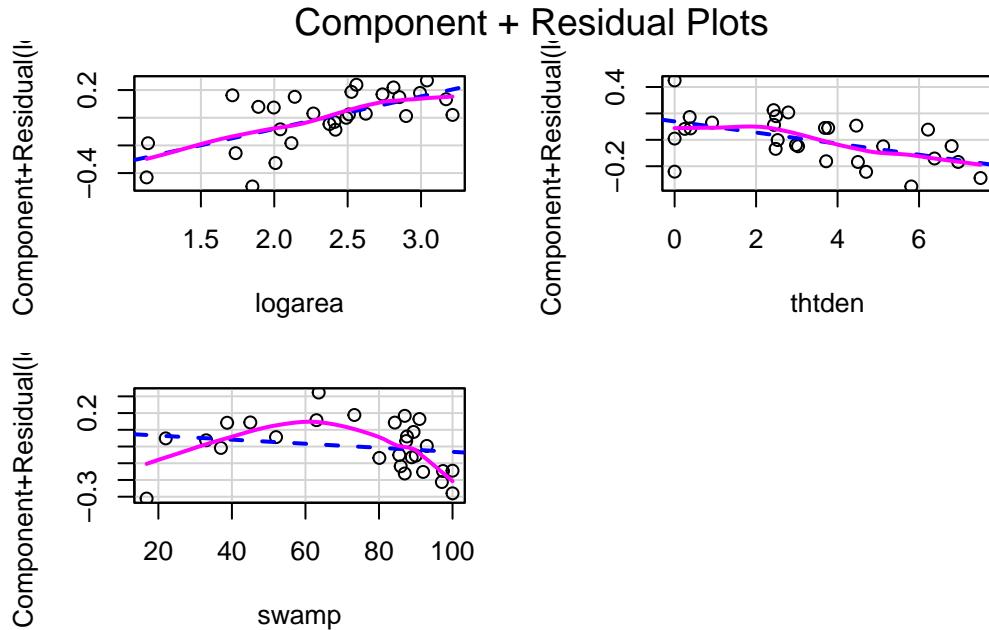


Figure 13.7.: Graphiques de résidus partiels du modèle `model_nopoly`

The lack of linearity along the gradient of `swamp` becomes obvious. The RESET test also detects a violation from linearity:

```
resettest(model_nopoly, type = "regressor")
```

RESET test

```
data: model_nopoly
RESET = 6.7588, df1 = 6, df2 = 18, p-value = 0.0007066
```

13.10. Testing for interactions

When there are multiple independent variables one should always be ready to assess interactions. In most multiple regression contexts this is somewhat difficult because adding interaction terms increases overall multicollinearity and because in many cases there are not enough observations to test all interactions, or the observations are not well balanced to make powerful tests for interactions. Going back to our final model, see what happens if one tries to fit the fully saturated model with all interactions:

```
fullmodel_withinteractions <- lm(
  logherp ~ logarea * cpfor2 * thtden * swamp * I(swamp^2),
  data = mydata
)
summary(fullmodel_withinteractions)
```

Call:

```
lm(formula = logherp ~ logarea * cpfor2 * thtden * swamp * I(swamp^2),
  data = mydata)
```

Residuals:

ALL 28 residuals are 0: no residual degrees of freedom!

Coefficients: (4 not defined because of singularities)

| | Estimate | Std. Error | t value | Pr(> t) |
|----------------|------------|------------|---------|----------|
| (Intercept) | -5.948e+03 | NaN | NaN | NaN |
| logarea | 3.293e+03 | NaN | NaN | NaN |
| cpfor2 | 7.080e+01 | NaN | NaN | NaN |
| thtden | 9.223e+02 | NaN | NaN | NaN |
| swamp | 1.176e+02 | NaN | NaN | NaN |
| I(swamp^2) | -3.517e-01 | NaN | NaN | NaN |
| logarea:cpfor2 | -3.771e+01 | NaN | NaN | NaN |
| logarea:thtden | -4.781e+02 | NaN | NaN | NaN |
| cpfor2:thtden | -1.115e+01 | NaN | NaN | NaN |
| logarea:swamp | -7.876e+01 | NaN | NaN | NaN |

| | | | | |
|--|------------|-----|-----|-----|
| cpfor2:swamp | -1.401e+00 | NaN | NaN | NaN |
| thtden:swamp | -1.920e+01 | NaN | NaN | NaN |
| logarea:I(swamp^2) | 5.105e-01 | NaN | NaN | NaN |
| cpfor2:I(swamp^2) | 3.825e-03 | NaN | NaN | NaN |
| thtden:I(swamp^2) | 7.826e-02 | NaN | NaN | NaN |
| swamp:I(swamp^2) | -2.455e-03 | NaN | NaN | NaN |
| logarea:cpfor2:thtden | 5.359e+00 | NaN | NaN | NaN |
| logarea:cpfor2:swamp | 8.743e-01 | NaN | NaN | NaN |
| logarea:thtden:swamp | 1.080e+01 | NaN | NaN | NaN |
| cpfor2:thtden:swamp | 2.620e-01 | NaN | NaN | NaN |
| logarea:cpfor2:I(swamp^2) | -5.065e-03 | NaN | NaN | NaN |
| logarea:thtden:I(swamp^2) | -6.125e-02 | NaN | NaN | NaN |
| cpfor2:thtden:I(swamp^2) | -1.551e-03 | NaN | NaN | NaN |
| logarea:swamp:I(swamp^2) | -4.640e-04 | NaN | NaN | NaN |
| cpfor2:swamp:I(swamp^2) | 3.352e-05 | NaN | NaN | NaN |
| thtden:swamp:I(swamp^2) | 2.439e-04 | NaN | NaN | NaN |
| logarea:cpfor2:thtden:swamp | -1.235e-01 | NaN | NaN | NaN |
| logarea:cpfor2:thtden:I(swamp^2) | 7.166e-04 | NaN | NaN | NaN |
| logarea:cpfor2:swamp:I(swamp^2) | NA | NA | NA | NA |
| logarea:thtden:swamp:I(swamp^2) | NA | NA | NA | NA |
| cpfor2:thtden:swamp:I(swamp^2) | NA | NA | NA | NA |
| logarea:cpfor2:thtden:swamp:I(swamp^2) | NA | NA | NA | NA |

Residual standard error: NaN on 0 degrees of freedom

(2 observations deleted due to missingness)

Multiple R-squared: 1, Adjusted R-squared: NaN

F-statistic: NaN on 27 and 0 DF, p-value: NA

Indeed, it is not possible to include all 32 terms with only 28 observations. There are not enough data points, R square is one, and the model perfectly overfits the data.

If you try to use an automated routine to “pick” the best model out of this soup, R complains:

```
step(fullmodel_withinteractions)
```

Error in step(fullmodel_withinteractions): AIC is -infinity for this model, so 'step' cannot proceed.

Does this mean you can forget about potential interactions and simply accept the final model without a thought? No. You simply do not have enough data to test for all interactions. But there is a compromise worth attempting, comparing the final model to a model with a subset of the interactions, say all second order interactions, to check whether the inclusion of these interactions improves substantially the fit:

```
full_model_2ndinteractions <- lm(
  logherp ~ logarea + cpfor2 + thtden + swamp + I(swamp^2) +
  logarea:cpfor2 +
  logarea:thtden +
  logarea:swamp +
  cpfor2:thtden +
  cpfor2:swamp +
  thtden:swamp,
  data = mydata
)
summary(full_model_2ndinteractions)
```

Call:

```
lm(formula = logherp ~ logarea + cpfor2 + thtden + swamp + I(swamp^2) +
  logarea:cpfor2 + logarea:thtden + logarea:swamp + cpfor2:thtden +
  cpfor2:swamp + thtden:swamp, data = mydata)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|-----------|----------|----------|----------|
| -0.216880 | -0.036534 | 0.003506 | 0.042990 | 0.175490 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|-------------|-----------|------------|---------|----------|
| (Intercept) | 4.339e-01 | 6.325e-01 | 0.686 | 0.502581 |

```

logarea      -1.254e-01  2.684e-01  -0.467  0.646654
cpfor2       -9.344e-03  7.205e-03  -1.297  0.213032
thtden       -1.833e-01  9.035e-02  -2.028  0.059504 .
swamp        3.569e-02  7.861e-03   4.540  0.000334 ***
I(swamp^2)    -3.090e-04  7.109e-05  -4.347  0.000500 ***
logarea:cpfor2 2.582e-03  2.577e-03   1.002  0.331132
logarea:thtden 7.017e-02  3.359e-02   2.089  0.053036 .
logarea:swamp -5.290e-04  2.249e-03  -0.235  0.816981
cpfor2:thtden -2.095e-04  6.120e-04  -0.342  0.736544
cpfor2:swamp   4.651e-05  5.431e-05   0.856  0.404390
thtden:swamp   2.248e-04  4.764e-04   0.472  0.643336
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Residual standard error: 0.108 on 16 degrees of freedom

(2 observations deleted due to missingness)

Multiple R-squared: 0.8658, Adjusted R-squared: 0.7735

F-statistic: 9.382 on 11 and 16 DF, p-value: 4.829e-05

This model fits the data slightly better than the “final” model (it explains 86.6% of the variance in logherp, compared to 81.2% for the “final” model without interactions), but has twice as many parameters.

If you look at the individual coefficients, some weird things happen: for example, the sign for logarea has changed. This is one of the symptoms of multicollinearity. Let’s look at the variance inflation factors:

```
vif(full_model_2ndinteractions)
```

there are higher-order terms (interactions) in this model

consider setting type = 'predictor'; see ?vif

| | | | | |
|----------------|----------------|---------------|---------------|--------------|
| logarea | cpfor2 | thtden | swamp | I(swamp^2) |
| 49.86060 | 78.49622 | 101.42437 | 90.47389 | 115.08457 |
| logarea:cpfor2 | logarea:thtden | logarea:swamp | cpfor2:thtden | cpfor2:swamp |
| 66.97792 | 71.69894 | 67.27034 | 14.66814 | 29.41422 |
| thtden:swamp | | | | |

```
20.04410
```

Ouch. All VIF are above 5, not only the ones involving the swamp terms. This model is not very satisfying it seems. Indeed the AIC for the two models indicate that the model with interactions has less information than the full model (remember, models with the lowest AIC value are to be preferred):

```
AIC(model_poly1)
```

```
[1] -38.3433
```

```
AIC(full_model_2ndinteractions)
```

```
[1] -34.86123
```

The `anova()` command can be used to test whether the addition of all interaction terms improves the fit significantly:

```
anova(model_poly1, full_model_2ndinteractions)
```

| Res.Df | RSS | Df | Sum of Sq | F | Pr(>F) |
|--------|-----------|----|-----------|-----------|-----------|
| 22 | 0.2528203 | NA | NA | NA | NA |
| 16 | 0.1865067 | 6 | 0.0663136 | 0.9481497 | 0.4890062 |

This test indicates that the addition of interaction terms did not reduce significantly the residual variance around the full model. How about a comparison with the final model without `cpfor2`?

```
anova(model_poly2, full_model_2ndinteractions)
```

| Res.Df | RSS | Df | Sum of Sq | F | Pr(>F) |
|--------|-----------|----|-----------|-----------|-----------|
| 23 | 0.2599923 | NA | NA | NA | NA |
| 16 | 0.1865067 | 7 | 0.0734856 | 0.9005955 | 0.5294411 |

And this comparison suggests that our final model does not make worse predictions than the full model with interactions.

13.11. Dredging and the information theoretical approach

One of the main critiques of stepwise methods is that the p-values are not strictly correct because of the large number of tests that are actually done. This is the multiple testing problem. In building linear models (multiple regression for example) from a large number of independent variables, and possibly their interactions, there are so many possible combinations that if one were to use Bonferroni type corrections, it would make tests very conservative.

An alternative, very elegantly advocated by Burnham and Anderson (2002, Model selection and multimodel inference: a practical information-theoretic approach. 2nd ed), is to use AIC (or better the AICc that is more appropriate for samples where the number of observations is less than about 40 times the number of variables) to rank potential models, and identify the set of models that are the best ones. One can then average the parameters across models, weighting using the probability that it is the best model to obtain coefficients that are more robust and less likely to be unduly affected by multicollinearity.

Warning

To compare models using *AIC*, models need to be fitted using the exact same data for each model. You thus need to be careful that there are no missing data when using an AIC based approach to model selection

The approach of comparing model fit using AIC was first developed to compare a set of models carefully built and chosen by the person doing the analysis based on a-priori knowledge and biological hypotheses. Some, however, developed an approach that I consider brainless and brutal to fit all potential models and then compare them using *AIC*. This approach has been implemented in the package MuMIn.

Note

I do not support the use of stepwise AIC or data dredging which are going against the philosophy of AIC and parsimony. Develop a model based on biological hypothesis and report all the results significant or not without dredging the data.

```
# redo the model double chekcing there are no "NA"
# specifying na.action

full_model_2ndinteractions <- update(
  full_model_2ndinteractions,
  . ~ .,
  data = mysub,
```

```

na.action = "na.fail"
)

library(MuMIn)
dd <- dredge(full_model_2ndinteractions)

```

Fixed term is "(Intercept)"

Object dd will contain all possible models using the terms of our full model with 2nd order interactions. Then, we can have a look at the subset of models that have an AICc within 4 units from the lowest AICc model. (Burnham and Anderson suggest that models that deviate by more than 2 AICc units have very little empirical support):

```

# get models within 4 units of AICc from the best model
top_models_1 <- get.models(dd, subset = delta < 4)
avgmodel1 <- model.avg(top_models_1) # compute average parameters
summary(avgmodel1) # display averaged model

```

Call:

```
model.avg(object = top_models_1)
```

Component model call:

```
lm(formula = logherp ~ <8 unique rhs>, data = mysub, na.action =
na.fail)
```

Component models:

| | df | logLik | AICc | delta | weight |
|--------|----|--------|--------|-------|--------|
| 23457 | 7 | 27.78 | -35.95 | 0.00 | 0.34 |
| 2345 | 6 | 25.78 | -35.56 | 0.39 | 0.28 |
| 123457 | 8 | 28.30 | -33.02 | 2.93 | 0.08 |
| 234578 | 8 | 28.26 | -32.95 | 3.00 | 0.08 |
| 12345 | 7 | 26.17 | -32.74 | 3.21 | 0.07 |
| 23458 | 7 | 26.06 | -32.51 | 3.44 | 0.06 |
| 234567 | 8 | 27.88 | -32.17 | 3.78 | 0.05 |

23456 7 25.79 -31.99 3.97 0.05

Term codes:

| cpfor2 | I(swamp^2) | logarea | swamp | thtden |
|--------|------------|---------------|----------------|--------------|
| 1 | 2 | 3 | 4 | 5 |
| | | logarea:swamp | logarea:thtden | swamp:thtden |
| 6 | 7 | 8 | | |

Model-averaged coefficients:

(full average)

| | Estimate | Std. Error | Adjusted SE | z value | Pr(> z) |
|----------------|------------|------------|-------------|---------|-----------|
| (Intercept) | -2.075e-01 | 2.484e-01 | 2.593e-01 | 0.800 | 0.424 |
| logarea | 1.314e-01 | 1.185e-01 | 1.222e-01 | 1.076 | 0.282 |
| swamp | 3.193e-02 | 6.125e-03 | 6.438e-03 | 4.960 | 7e-07 *** |
| I(swamp^2) | -2.676e-04 | 4.904e-05 | 5.154e-05 | 5.193 | 2e-07 *** |
| thtden | -6.843e-02 | 5.324e-02 | 5.459e-02 | 1.254 | 0.210 |
| logarea:thtden | 2.139e-02 | 2.506e-02 | 2.565e-02 | 0.834 | 0.404 |
| cpfor2 | -1.202e-04 | 4.710e-04 | 4.886e-04 | 0.246 | 0.806 |
| swamp:thtden | -3.277e-05 | 1.419e-04 | 1.475e-04 | 0.222 | 0.824 |
| logarea:swamp | 4.378e-05 | 5.378e-04 | 5.676e-04 | 0.077 | 0.939 |

(conditional average)

| | Estimate | Std. Error | Adjusted SE | z value | Pr(> z) |
|----------------|------------|------------|-------------|---------|-----------|
| (Intercept) | -2.075e-01 | 2.484e-01 | 2.593e-01 | 0.800 | 0.4236 |
| logarea | 1.314e-01 | 1.185e-01 | 1.222e-01 | 1.076 | 0.2820 |
| swamp | 3.193e-02 | 6.125e-03 | 6.438e-03 | 4.960 | 7e-07 *** |
| I(swamp^2) | -2.676e-04 | 4.904e-05 | 5.154e-05 | 5.193 | 2e-07 *** |
| thtden | -6.843e-02 | 5.324e-02 | 5.459e-02 | 1.254 | 0.2100 |
| logarea:thtden | 3.924e-02 | 2.125e-02 | 2.251e-02 | 1.743 | 0.0813 . |
| cpfor2 | -8.187e-04 | 9.692e-04 | 1.027e-03 | 0.797 | 0.4253 |
| swamp:thtden | -2.402e-04 | 3.127e-04 | 3.313e-04 | 0.725 | 0.4684 |
| logarea:swamp | 4.462e-04 | 1.664e-03 | 1.762e-03 | 0.253 | 0.8001 |

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
confint(avgmodel1) # display CI for averaged coefficients
```

| | 2.5 % | 97.5 % |
|----------------|---------------|---------------|
| (Intercept) | -0.7157333646 | 0.3007147516 |
| logarea | -0.1080048582 | 0.3708612563 |
| swamp | 0.0193158426 | 0.0445532538 |
| I(swamp^2) | -0.0003686653 | -0.0001666418 |
| thtden | -0.1754184849 | 0.0385545120 |
| logarea:thtden | -0.0048800385 | 0.0833595106 |
| cpfor2 | -0.0028313465 | 0.0011940283 |
| swamp:thtden | -0.0008894138 | 0.0004090457 |
| logarea:swamp | -0.0030067733 | 0.0038991294 |

1. **components models:** You first get the list of the models with an AICc within the desired 4 units of the best model. The variables that are included in the model are coded with the key just below.
2. For each model, in addition to the AICc, the Akaike weights are calculated. They represent the relative likelihood of a model, and indicate the relative importance of a model compared to the other models tested.
3. **Mode-averaged coefficients:** For the subset of models, weighted averages (using Akaike weights) for model parameters are calculated, with 95% CI. Note that, by default, terms missing from a model are assumed to have a coefficient of 0.

13.12. Bootstrapping multiple regression

When data do not meet the assumptions of normality and homoscedasticity and it is not possible to transform the data to meet the assumptions, bootstrapping can be used to compute confidence intervals for coefficients. If the distribution of the bootstrapped coefficients is symmetrical and approximately Gaussian, then empirical percentiles can be used to estimate the confidence limits.

The following code, using the `simpleboot`  has been designed to be easily modifiable and will compute CI using empirical percentiles. Following this is an easier approach using the library `boot` that will calculate several different bootstrap confidence limits.

```
#####
#####
# Bootstrap analysis the simple way with library simpleboot
# Define model to be bootstrapped and the data source used
mymodel <- lm(logherp ~ logarea + thtden + swamp + I(swamp^2), data = mydata)
# Set the number of bootstrap iterations
nboot <- 1000
library(simpleboot)
# R is the number of bootstrap iterations
# Setting rows to FALSE indicates resampling of residuals
mysimpleboot <- lm.boot(mymodel, R = nboot, rows = FALSE)
# Extract bootstrap coefficients
myresults <- sapply(mysimpleboot$boot.list, function(x) x$coef)
# Transpose matrix so that lines are bootstrap iterations
# and columns are coefficients
tmyresults <- t(myresults)
```

You can then plot the results using the following code. When run, it will pause to let you have a look at the distribution for each coefficient in the model by producing plots like:

```
# Plot histograms of bootstrapped coefficients
ncoefs <- length(data.frame(tmyresults))
par(mfrow = c(1, 2), mai = c(0.5, 0.5, 0.5, 0.5), ask = TRUE)
for (i in 1:ncoefs) {
  lab <- colnames(tmyresults)[i]
  x <- tmyresults[, i]
  plot(density(x), main = lab, xlab = "")
  abline(v = mymodel$coef[i], col = "red")
  abline(v = quantile(x, c(0.025, 0.975)))
  hist(x, main = lab, xlab = "")
  abline(v = quantile(x, c(0.025, 0.975)))
  abline(v = mymodel$coef[i], col = "red")
}
```

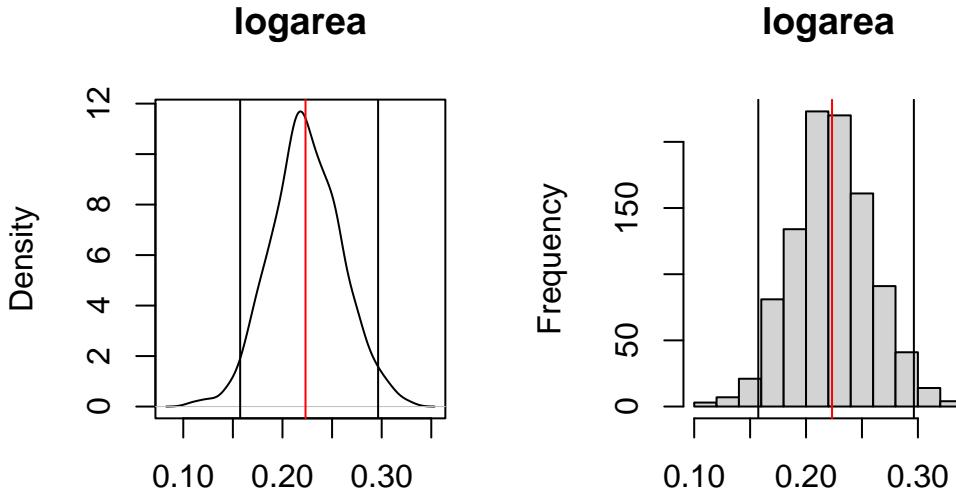


Figure 13.8.: Distribution of bootstrapped estimates for `logarea`

The top plot is the probability density function and the bottom one is the histogram of the bootstrap estimates for the coefficient. On these plots, the red line indicate the value of the parameter in the ordinary analysis, and the two vertical black lines mark the limits of the 95% confidence interval. Here the CI does not include 0 and one can conclude that the effect of `logarea` on `logherp` is significantly positive.

Precise values for the limits can be obtained by:

```
# Display empirical bootstrap quantiles (not corrected for bias)
p <- c(0.005, 0.01, 0.025, 0.05, 0.95, 0.975, 0.99, 0.995)
apply(tmyresults, 2, quantile, p)
```

| | (Intercept) | <code>logarea</code> | <code>thtden</code> | <code>swamp</code> | <code>I(swamp^2)</code> |
|-------|--------------|----------------------|---------------------|--------------------|-------------------------|
| 0.5% | -0.735452920 | 0.1258150 | -0.045932448 | 0.01619705 | -0.0003497515 |
| 1% | -0.708485440 | 0.1409869 | -0.044642449 | 0.01797894 | -0.0003372905 |
| 2.5% | -0.677381966 | 0.1573293 | -0.041669156 | 0.02024060 | -0.0003271927 |
| 5% | -0.619973091 | 0.1663988 | -0.038357265 | 0.02171437 | -0.0003142408 |
| 95% | -0.090752129 | 0.2830946 | -0.012386321 | 0.03771589 | -0.0001877812 |
| 97.5% | -0.042671614 | 0.2964062 | -0.009732258 | 0.03918373 | -0.0001742667 |
| 99% | 0.009413056 | 0.3079643 | -0.006918576 | 0.04035550 | -0.0001559284 |
| 99.5% | 0.066649913 | 0.3123665 | -0.005976149 | 0.04163342 | -0.0001392665 |

These confidence limits are not reliable when the distribution of the bootstrap estimates deviate from Gaussian. If they do, then it is preferable to compute so-called bias-corrected accelerated (BCa) confidence limits. The following code does just that:

```
#####
# Bootstrap analysis in multiple regression with BCa confidence intervals
# Preferable when parameter distribution is far from normal
# Bootstrap 95% BCa CI for regression coefficients

library(boot)

# function to obtain regression coefficients for each iteration
bs <- function(formula, data, indices) {
  d <- data[indices, ] # allows boot to select sample
  fit <- lm(formula, data = d)
  return(coef(fit))
}

# bootstrapping with 1000 replications
results <- boot(
  data = mydata, statistic = bs, R = 1000,
  formula = logherp ~ logarea + thtden + swamp + I(swamp^2)
)
# view results
```

To get the results, the following code will produce the standard graph for each coefficient and the resulting BCa interval.

```
plot(results, index = 1) # intercept
plot(results, index = 2) # logarea
plot(results, index = 3) # thtden
plot(results, index = 4) # swamp
plot(results, index = 5) # swamp2

# get 95% confidence intervals
boot.ci(results, type = "bca", index = 1)
```

```
boot.ci(results, type = "bca", index = 2)
boot.ci(results, type = "bca", index = 3)
boot.ci(results, type = "bca", index = 4)
boot.ci(results, type = "bca", index = 5)
```

For logarea, we get:

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS

Based on 1000 bootstrap replicates

CALL :

```
boot.ci(boot.out = results, type = "bca", index = 2)
```

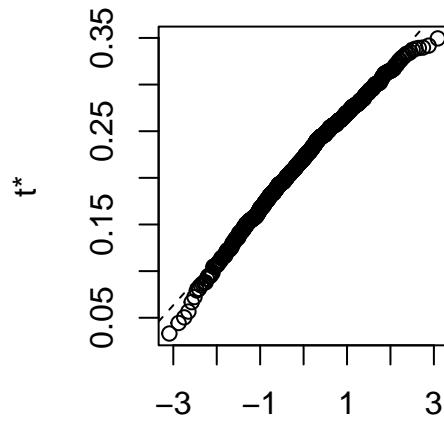
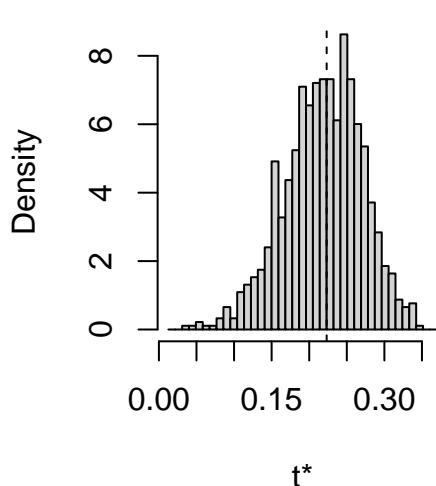
Intervals :

Level BCa

95% (0.1222, 0.3256)

Calculations and Intervals on Original Scale

Histogram of t



Quantiles of Standard Normal

Note that the BCa interval is from 0.12 to 0.32, whereas the simpler percentile interval is 0.16 to 0.29. BCa interval here is longer on the low side, and shorter on the high side, which it should be given the distribution of bootstrap estimates.

13.13. Permutation test

Permutation tests are more rarely performed in multiple regression contexts than bootstrap. But here is code to do it.

```
#####
#####
# Permutation in multiple regression
#
# using lmperm library
library(lmPerm)
# Fit desired model on the desired dataframe
my_model <- lm(logherp ~ logarea + thtden + swamp + I(swamp^2),
  data = mydata
)
my_model_prob <- lmp(
  logherp ~ logarea + thtden + swamp + I(swamp^2),
  data = mydata, perm = "Prob"
)
summary(my_model)
summary(my_model_prob)
```

Chapter 14

ANCOVA and general linear model

After competing this laboratory exercise, you should be able to:

- Use R to do an analysis of covariance (ANCOVA) and interpret statistical models that have both continuous and categorical independent variables LMs
- Use R to test the assumptions underlying LMs
- Use R to compare model fits
- Use R to do bootstrap and permutation tests on LM type models including both continuous and categorical independent variables.

14.1. R packages and data

This laboratory requires the following:

- R packages:
 - ggplot2
 - car
 - lmtest
- data files
 - anc1dat.csv
 - anc3dat.csv

Loading required package: carData

```
Loading required package: zoo
```

```
Attaching package: 'zoo'
```

```
The following objects are masked from 'package:base':
```

```
as.Date, as.Date.numeric
```

14.2. Linear models

GLM sometimes stands for General Linear Model, however, it is much more frequently used for *Generalized linear models*. Thus I always rather talk about *Linear models* or *LMs* instead of *General linear models* to avoid confusion in the acronym. LMs are statistical models that can be written as $Y = XB + E$, where Y is a vector (or matrix) containing the dependent variable, X is a matrix of independent variables, B is a matrix of estimated parameters, and E is the vector (or matrix) of independent, normally distributed and homoscedastic residuals. All tests we have seen to date (t-test, simple linear regression, One-Way ANOVA, Multiway ANOVA, and multiple regression) are LMs. Note that all models we have encountered until now contain only one type of variable (either continuous or categorical) as independent variables. In this laboratory exercise, you will fit models that have both type of independent variables. These models are also LMs.

14.3. ANCOVA

ANCOVA stands for Analysis of Covariance. It is a type of LM where there is one (or more) continuous independent variable (sometimes called a covariate) and one (or more) categorical independent variable. In the traditional treatment of ANCOVA in biostatistical textbooks, the ANCOVA model does not contain interaction terms between the continuous and categorical independent variables. Hence, the traditional ANCOVA analysis assumes that there is no interaction, and is preceded by a test of significance of interactions, equivalent to testing that the slopes (coefficients for the continuous independent variables) do not differ among level of the categorical independent variables (a test for homogeneity of slopes). Some people, me included, use the term ANCOVA a bit more loosely for any LM that involves both continuous and categorical variables. Be aware that, depending on the author, ANCOVA may refer to a model with or without interaction terms.

14.4. Homogeneity of slopes

In many biological problems, a question arises as to whether the slopes of two or more regression lines are significantly different; for example, whether two different insecticides differ in their efficacy, whether males and females differ in their growth curves, etc. These problems call for direct comparisons of slopes. GLMs (ANCOVAs) can test for equality of slopes (homogeneity of slopes).

Remember that there are two parameters that describe a regression line, the intercept and the slope. The ANCOVA model (*sensu stricto*) tests for homogeneity of intercepts, but the starting point for the analysis is a test for homogeneity of slopes. This test can be performed by fitting a model with main effects for both the categorical and continuous independent variables, plus the interaction term(s), and testing for significance of the addition of the interaction terms.

14.4.1. Case 1 - Size as a function of age (equal slopes example)

Exercise

Using the file `anc1dat.csv`, test the hypothesis that female and male sturgeon at The Pas over the years 1978-1980 have the same observed growth rate, defined as the slope of the regression of \log_{10} of fork length, `lfk1`, on the \log_{10} age, `lage`.

First, let's have a look at the data. It would help to draw the regression line and a lowess trace to better assess linearity. Being fancy, one could also use more of R magic to spruce up the axis legends (note the use of `expression()` to get subscripts):

```
anc1dat <- read.csv("data/anc1dat.csv")
anc1dat$sex <- as.factor(anc1dat$sex)
myplot <- ggplot(data = anc1dat, aes(x = lage, y = log10(fklength))) +
  facet_grid(. ~ sex) +
  geom_point()
myplot <- myplot +
  geom_smooth(method = lm, se = FALSE) +
  geom_smooth(se = FALSE, color = "red") +
  labs(
    y = expression(log[10] ~ (Fork ~ length)),
```

```
x = expression(log[10] ~ (Age))
)
myplot
```

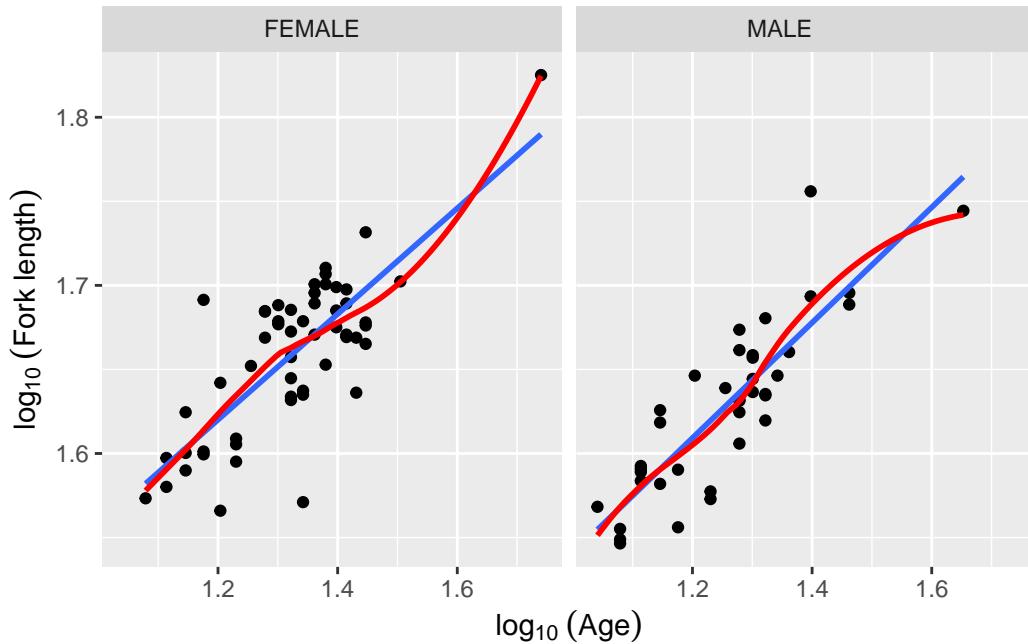


Figure 14.1.: Sturgeon length as a function of age

The log-log transformation makes the relationship linear and, at first glance, there is no issues with assumptions of LMs (although this should be confirmed by appropriate examination of the residuals). Let's fit the full model with both main effects and the interaction:

```
model.full1 <- lm(lfkl ~ sex + lage + sex:lage, data = anc1dat)
Anova(model.full1, type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|-------------|-----------|----|-------------|-----------|
| (Intercept) | 0.6444361 | 1 | 794.8182143 | 0.0000000 |
| sex | 0.0004089 | 1 | 0.5043251 | 0.4794836 |
| lage | 0.0725916 | 1 | 89.5311705 | 0.0000000 |
| sex:lage | 0.0002730 | 1 | 0.3366921 | 0.5632277 |
| Residuals | 0.0713501 | 88 | NA | NA |

In the previous output, on line 4, 0.5632277 is the probability of observing an `lage:sex` interaction this strong or stronger under the null hypothesis that slope of the relationship between fork length and age does not vary between the sexes, or equivalently that the difference in fork length between males and females (if it exists) does not vary with age (and providing the assumptions of the analysis have been met).

Note

Note that I used the `Anova()` function with an uppercase “a” (from the `car` library) instead of the “built in” `anova()` (with a lowercase “a”) command to get the results using Type III sums of squares. The type III (partial) sums of squares are calculated as if each variable was the last entered in the model and correspond to the difference in explained SS between the full model and a reduced model where only that variable is excluded. The standard `anova()` function returns Type I (sequential) SS, calculated as each variable is added sequentially to a null model with only an intercept. In rare cases, the type I and type III SS are equal (when the design is perfectly balanced and there is no multicollinearity). In the vast majority of cases, they will differ, and I recommend that you always use the Type III SS in your analyses.

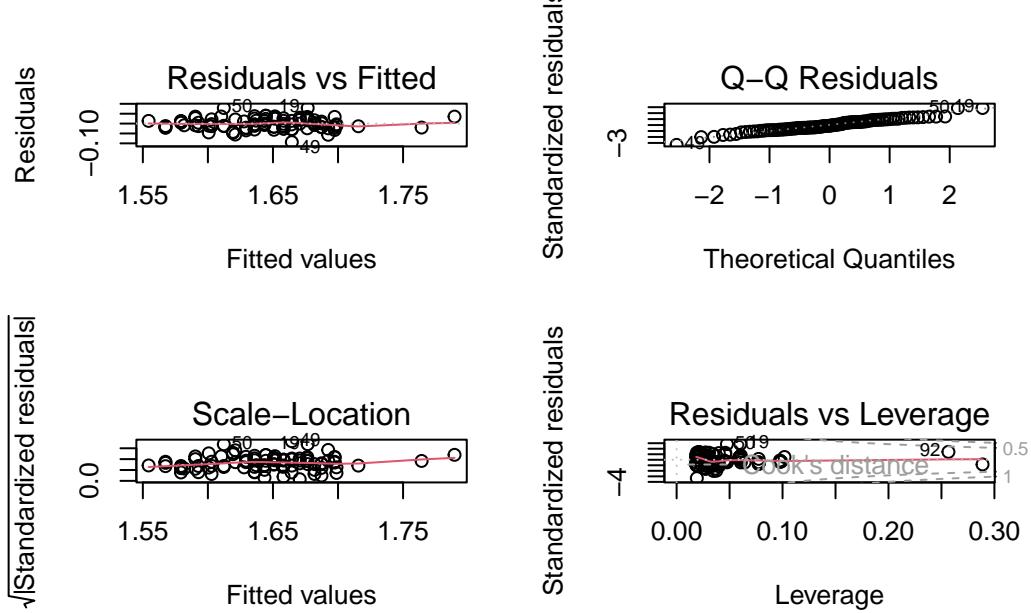
On the basis of this analysis, we would accept the null hypotheses that:

1. the slope of the regression of $\log(\text{fork length})$ on $\log(\text{age})$ is the same for males and females (the interaction term is not significant)
2. that the intercepts are also the same for the two sexes (the sex term is also not significant).

But before accepting these conclusions, we should test the assumptions in the usual way

Solution

```
par(mfrow = c(2, 2))
plot(model.full11)
```

Figure 14.2.: Model assumptions for `model.full1`

With respect to normality, things look O.K., although there are several points in the top right corner that appear to lie off the line. We could also run a Wilk-Shapiro normality test and find $W = 0.9764$, $p = 0.09329$, also suggesting this assumption is valid. Homoscedasticity seems fine too, but if you want further evidence of this, you can run one of the tests. Here I use the Breusch-Pagan test, which is appropriate when some of the independent variables are continuous (Levene's test is for categorical independent variables only):

```
bptest(model.full1)
```

```
studentized Breusch-Pagan test
```

```
data: model.full1
BP = 0.99979, df = 3, p-value = 0.8013
```

Since the null is that the residuals are homoscedastic, and p is rather large, the test confirms the visual assessment.

Further, there is no obvious pattern in the residuals, which implies there is no problem with the assumption of linearity. This too can be formally tested:

```
resettest(model.full1, power = 2:3, type = "regressor", data = anc1dat)
```

```
RESET test

data: model.full1
RESET = 0.59861, df1 = 2, df2 = 86, p-value = 0.5519
```

The last assumption in this sort of analysis is that the covariate (in this case, lage) has no measurement error. We really have no way of knowing whether this assumption is justified, although multiple aging of fish by several different investigators usually gives ages that are within 1-2 years of each other, which is within the 10% considered by most to be the maximum for Type I modelling. Note that there is no “test” that you can do with the data to determine what the error is, at least in this case. If we had replicate ages for individual fish, it could be estimated quantitatively

🔥 Exercise

You will notice that there is one datum with a large studentized residual, i.e. an outlier (case 49). Eliminate this datum from your data file and rerun the analysis. Do your conclusions change?

💡 Tip

```
model.full.no49 <- lm(lfkl ~ sex + lage + sex:lage, data = anc1dat[c(-49), ])
Anova(model.full.no49, type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|-------------|-----------|----|-------------|-----------|
| (Intercept) | 0.6425466 | 1 | 895.9393626 | 0.0000000 |
| sex | 0.0003782 | 1 | 0.5273066 | 0.4696905 |
| lage | 0.0737792 | 1 | 102.8745821 | 0.0000000 |
| sex:lage | 0.0002248 | 1 | 0.3134569 | 0.5770053 |
| Residuals | 0.0623943 | 87 | NA | NA |

So the conclusion does not change if the outlier is deleted (not surprising as Cook’s distance is low for this point reflecting its low leverage). Since there is no good reason to delete this data point, and since (at least qualitatively) our conclusions do not change, it is probably best to go with the full data set. A test of the assumptions for the refit

model (with the outlier removed) shows that all are met, and no more outliers are detected. (I won't report these analyses, but you can and should do them just to assure yourself that everything is O.K.)

14.4.2. Case 2 - Size as a function of age (different slopes example)

🔥 Exercise

The file `anc3dat.csv` records data on male sturgeon collected at two locations (`locate`), Lake of the Woods, in northwestern Ontario, and the Churchill River in northern Manitoba. Using the same procedure as outlined above (with `locate` as the categorical variable instead of `sex`), test the null hypothesis that the slope of the regression of `lfk1` on `lage` is the same in the two locations. What do you conclude?

```
anc3dat <- read.csv("data/anc3dat.csv")

myplot <- ggplot(data = anc3dat, aes(x = lage, y = log10(fklength))) +
  facet_grid(. ~ locate) +
  geom_point() +
  geom_smooth(method = lm, se = FALSE) +
  geom_smooth(se = FALSE, color = "red") +
  labs(
    y = expression(log[10] ~ (Fork ~ length)),
    x = expression(log[10] ~ (Age)))
  )
myplot
```

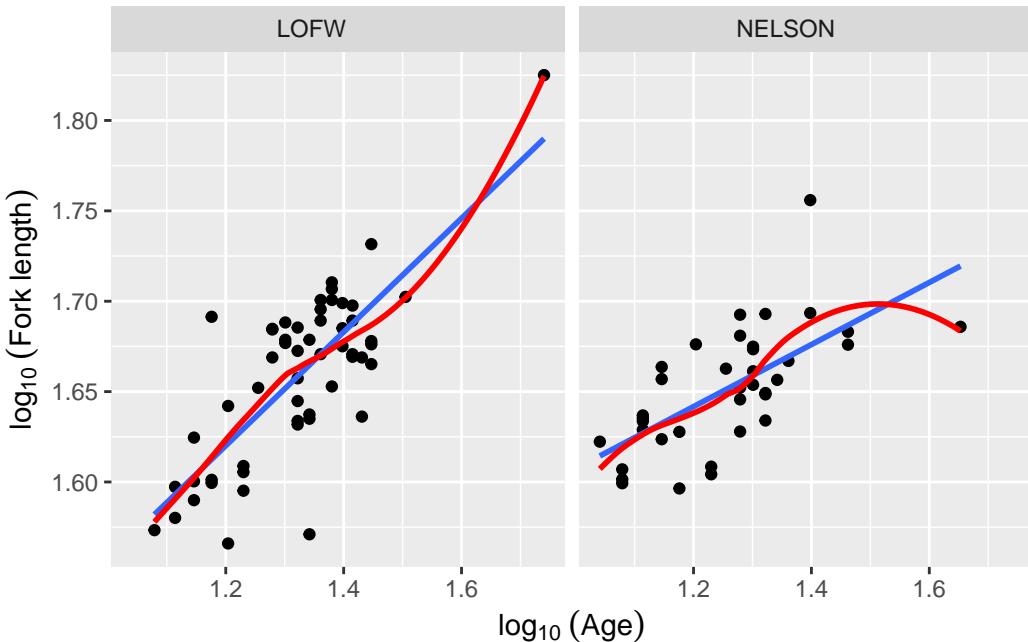


Figure 14.3.: Longueur des esturgeons en fonction de l'age d'après anc3dat

```
model.full2 <- lm(lfkl ~ lage + locate + lage:locate, data = anc3dat)
Anova(model.full2, type = 3)
```

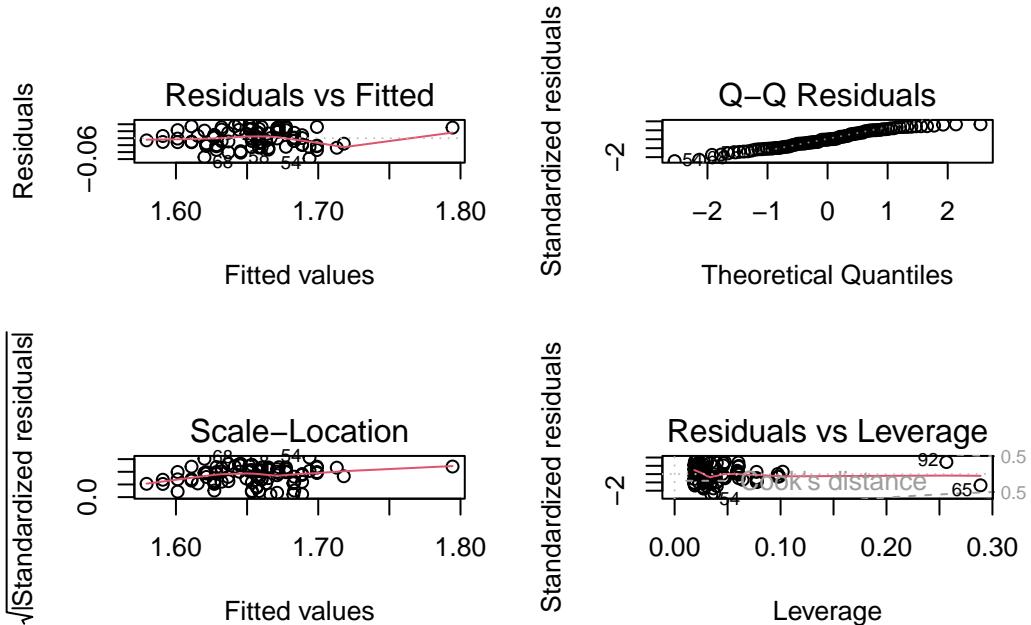
| | Sum Sq | Df | F value | Pr(>F) |
|-------------|-----------|----|------------|-----------|
| (Intercept) | 0.6295064 | 1 | 1078.63230 | 0.0000000 |
| lage | 0.0777287 | 1 | 133.18483 | 0.0000000 |
| locate | 0.0096826 | 1 | 16.59072 | 0.0001012 |
| lage:locate | 0.0090901 | 1 | 15.57541 | 0.0001592 |
| Residuals | 0.0513582 | 88 | NA | NA |

Longueur des esturgeons en fonction de l'age d'après anc3dat

In this case, we reject the null hypotheses that (1) the slopes of the regressions are the same in the two locations; and (2) that the intercepts are the same in the two locations. In other words, if we want to predict the fork length of a sturgeon of a particular age (accurately) we need to know from which location it came. The fact that we reject the null hypothesis that the slopes of the lfkl - lage regressions are the same in both locations means that we should be doing individual regressions for each location separately (that is in fact what the full model is fitting). But we are jumping the gun here. Before you can trust these p values, you need to confirm that assumptions are met:

 Tip

```
par(mfrow = c(2, 2))
plot(model.full12)
```

Figure 14.4.: Conditions d'applications du modèle `model.full12`

If you analyze the residuals (in the way described above), you will find that there is no problem with the linearity assumption, nor the homoscedasticity assumption ($\text{BP} = 1.2267$, $p = 0.268$). However, Wilk-Shapiro test of normality of residuals is suspicious ($W=0.97$, $p = 0.03$). Given the relatively large sample size ($N = 92$), this normality test has high power and the magnitude of deviation from normality does not appear to be large. Considering the robustness of GLM to non-normality with large samples, we should not be overly concerned with this violation.

Given that the assumptions appear sufficiently met, we can accept the results as calculated by R. All terms in the model are significant (location, lage, and the interaction). This full model is equivalent to fitting separate regressions for each location. To get the coefficients of these regression, one can either fit the two regressions on data subsets for each location, or extract the fitted coefficients from the full model:

```
model.full12
```

Call:

```
lm(formula = lfk1 ~ lage + locate + lage:locate, data = anc3dat)
```

Coefficients:

| (Intercept) | lage | locateNELSON |
|-------------------|--------|--------------|
| 1.2284 | 0.3253 | 0.2207 |
| lage:locateNELSON | | |
| -0.1656 | | |

By default, the variable `locate` in the model is internally encoded as 0 for the location that comes first alphabetically (LofW) and 1 for the other (Nelson). So the regression equations for each location become:

For LofW:

$$\begin{aligned} lfk1 &= 1.2284 + 0.3253 \times lage + 0.2207 \times 0 - 0.1656 \times 0 \times lage \\ &= 1.2284 + 0.3253 \times lage \end{aligned}$$

For Nelson:

$$\begin{aligned} lfk1 &= 1.2284 + 0.3253 \times lage + 0.2207 \times 1 - 0.1656 \times 1 \times lage \\ &= 1.4491 + 0.1597 \times lage \end{aligned}$$

You can convince yourself that this is the same as fitting 2 regressions separately.

```
by(anc3dat, anc3dat$locate, function(x) lm(lfk1 ~ lage, data = x))
```

anc3dat\$locate: LOFW

Call:

```
lm(formula = lfk1 ~ lage, data = x)
```

Coefficients:

| (Intercept) | lage |
|-------------|--------|
| 1.2284 | 0.3253 |

anc3dat\$locate: NELSON

Call:

```
lm(formula = lfkl ~ lage, data = x)
```

Coefficients:

| (Intercept) | lage |
|-------------|--------|
| 1.4491 | 0.1597 |

14.5. The ANCOVA model

If the test for homogeneity of slopes indicates that the two or more slopes are not significantly different, i.e. there is no significant interaction between the categorical and continuous variable, then a single slope parameter can be fit. How about the intercepts? Do they differ among levels of the categorical variable? There are two school of thoughts on how to proceed to test for equality of intercepts when slopes are equal:

- The old school fits a reduced model, with the categorical and continuous variables, but no interactions (this is the ANCOVA model, *sensus stricto*) and uses the partitioned sums of squares to test for significance, say with the `Anova()` function. This approach is the one presented in many statistical textbooks.
- Others simply use the full model results, and test significance of each term from the partial sums of squares. This approach has the advantage of being faster as only one model needs to be fitted to make all inferences. However, this approach is less powerful.

In most practical cases, it does not matter unless one has very complex models with a large number of terms and higher level interactions and that many of these terms are not significant. My suggestion is that you use the faster approach first, and use the traditional approach only when you accept the null hypothesis for equal intercepts. Why? Since the faster approach is less powerful, if you nevertheless reject H₀, then this conclusion will not be changed, only reinforced, by using the traditional approach.

Here I will compare the old school and the other approach. Recall that we want to assess equality of intercepts once we determined that slopes are equal. Test for equality of intercepts when slopes differ (or, if you prefer, when there is a significant interaction) are rarely directly meaningful, are often misinterpreted, and should rarely be conducted.

Going back to `anc1dat.csv`, comparing the relationships between `lfkl` and `lage` among sexes, we obtained the following results for the full model with interactions

```
Anova(model.full1, type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|-------------|-----------|----|-------------|-----------|
| (Intercept) | 0.6444361 | 1 | 794.8182143 | 0.0000000 |
| sex | 0.0004089 | 1 | 0.5043251 | 0.4794836 |
| lage | 0.0725916 | 1 | 89.5311705 | 0.0000000 |
| sex:lage | 0.0002730 | 1 | 0.3366921 | 0.5632277 |
| Residuals | 0.0713501 | 88 | NA | NA |

We already concluded that the slope of the regression for males and females does not differ (the interaction sex:lage is not significant). Note that the p-values associated with sex (0.4795) is not significant either.

For the old-school approach, one would fit a reduced model (the *sensus stricto* ANCOVA model):

```
model.ancova <- lm(lfkl ~ sex + lage, data = anc1dat)
Anova(model.ancova, type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|-------------|-----------|----|-------------|-----------|
| (Intercept) | 1.1348025 | 1 | 1410.123239 | 0.0000000 |
| sex | 0.0014899 | 1 | 1.851324 | 0.1770653 |
| lage | 0.1433772 | 1 | 178.162744 | 0.0000000 |
| Residuals | 0.0716231 | 89 | NA | NA |

```
summary(model.ancova)
```

Call:

```
lm(formula = lfkl ~ sex + lage, data = anc1dat)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|-----------|-----------|----------|----------|
| -0.093992 | -0.018457 | -0.000876 | 0.022491 | 0.081161 |

Coefficients:

| Estimate | Std. Error | t value | Pr(> t) |
|----------|------------|---------|----------|
|----------|------------|---------|----------|

```
(Intercept) 1.225533 0.032636 37.552 <2e-16 ***
sexMALE -0.008473 0.006228 -1.361 0.177
lage 0.327253 0.024517 13.348 <2e-16 ***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Residual standard error: 0.02837 on 89 degrees of freedom

Multiple R-squared: 0.696, Adjusted R-squared: 0.6892

F-statistic: 101.9 on 2 and 89 DF, p-value: < 2.2e-16

According to this test, sex is not significant and therefore we can conclude that the intercept does not vary significantly between males and females. Note that the p-value is lower this time (0.1771 vs 0.4795), reflecting the higher power of this old-school approach. However, the conclusion remains qualitatively the same: intercepts do not differ.

So we accept the null hypothesis that the intercepts are the same for the two sexes. Running the residual diagnostics, we find no problems with linearity, independence, homogeneity of variances, and normality.

🔥 Exercise

You will notice, in the above analysis that the residuals plots flag three data points (cases 19, 49, and 50) as having high residuals. These points are a bit worrisome, and may be having a disproportionate effect on your analysis. Eliminate these “outliers” and re-run the analysis. Now what do you conclude?

```
model.ancova.nooutliers <- lm(lfkl ~ sex + lage, data = anc1dat[c(-49, -50, -19), ])
Anova(model.ancova.nooutliers, type = 3)
```

| | Sum Sq | Df | F value | Pr(>F) |
|-------------|-----------|----|-------------|-----------|
| (Intercept) | 1.0916027 | 1 | 1896.520424 | 0.0000000 |
| sex | 0.0023238 | 1 | 4.037371 | 0.0476388 |
| lage | 0.1399208 | 1 | 243.094594 | 0.0000000 |
| Residuals | 0.0495000 | 86 | NA | NA |

```
summary(model.ancova.nooutliers)
```

Call:

```
lm(formula = lfk1 ~ sex + lage, data = anc1dat[c(-49, -50, -19),
])
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|-----------|-----------|----------|----------|
| -0.058397 | -0.018469 | -0.000976 | 0.020696 | 0.040288 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|----------------|-----------|------------|---------|------------|
| (Intercept) | 1.224000 | 0.028106 | 43.549 | <2e-16 *** |
| sexMALE | -0.010823 | 0.005386 | -2.009 | 0.0476 * |
| lage | 0.328604 | 0.021076 | 15.591 | <2e-16 *** |
| --- | | | | |
| Signif. codes: | 0 *** | 0.001 ** | 0.01 * | 0.05 . |
| | ' | ' | ' | ' |
| | ' | ' | ' | ' |

Residual standard error: 0.02399 on 86 degrees of freedom

Multiple R-squared: 0.7706, Adjusted R-squared: 0.7653

F-statistic: 144.4 on 2 and 86 DF, p-value: < 2.2e-16

Well, well. Now we would, according to convention, reject the null hypothesis, and conclude that in fact, the intercepts of the regressions for the two sexes are different! This is a qualitatively different result from that obtained using all the data. Why? There are two possible reasons:

1. the “outliers” have significant impacts on the fitted regression lines, so that the intercepts of the lines change depending on whether the “outliers” are included (or not);
2. the exclusion of the outliers increases the precision, i.e. reduces the standard error of the intercept estimates, and therefore increases the likelihood that the two intercepts will in fact be “statistically” different.

(1) is unlikely, since none of the outliers had high leverage (hence Cook’s distances were not large), so (2) is more likely, and you can verify this by fitting separate regressions for each sex with and without these three outliers. If you do, you will notice that the estimated intercepts for each sex do not change very much, but the standard errors of these intercepts change quite a lot.

🔥 Exercise

Fit separate regressions by sex with vs. without the outliers. Pay attention to the intercepts.

Including all data.

```
by(
  anc1dat,
  anc1dat[, "sex"],
  function(x) {
    summary(lm(lfkl ~ lage, data = x))
  }
)
```

anc1dat[, "sex"]: FEMALE

Call:

```
lm(formula = lfkl ~ lage, data = x)
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|-----------|-----------|----------|----------|
| -0.093728 | -0.020510 | -0.000618 | 0.024066 | 0.078844 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|----------------|----------|------------|----------|--------------|
| (Intercept) | 1.24264 | 0.04660 | 26.664 | < 2e-16 *** |
| lage | 0.31431 | 0.03512 | 8.949 | 4.16e-12 *** |
| --- | | | | |
| Signif. codes: | 0 '***' | 0.001 '**' | 0.01 '*' | 0.05 '.' |
| | 0.1 ' | ' 1 | | |

Residual standard error: 0.03011 on 52 degrees of freedom

Multiple R-squared: 0.6063, Adjusted R-squared: 0.5987

F-statistic: 80.09 on 1 and 52 DF, p-value: 4.16e-12

```

anc1dat[, "sex"]: MALE

Call:
lm(formula = lfk1 ~ lage, data = x)

Residuals:
    Min      1Q  Median      3Q     Max 
-0.046663 -0.014875 -0.004275  0.013489  0.078910 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 1.19730   0.04209  28.45 < 2e-16 ***
lage        0.34300   0.03337  10.28 2.97e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.02594 on 36 degrees of freedom
Multiple R-squared:  0.7458,    Adjusted R-squared:  0.7388 
F-statistic: 105.6 on 1 and 36 DF,  p-value: 2.972e-12

```

Difference in intercept is indeed really small. Now let's have a look when we exclude outliers.

```

by(
  anc1dat,
  anc1dat[, "sex"],
  function(x) {
    summary(lm(lfk1 ~ lage, data = x[c(-49, -50, -19), ]))
  }
)

```

```

anc1dat[, "sex"]: FEMALE

Call:
lm(formula = lfk1 ~ lage, data = x[c(-49, -50, -19), ])

```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|-----------|-----------|----------|----------|
| -0.092746 | -0.020176 | -0.000078 | 0.023779 | 0.079995 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) | | | | | | | |
|----------------|----------|------------|---------|--------------|------|-----|------|------|-----|-----|---|
| (Intercept) | 1.24029 | 0.04815 | 25.760 | < 2e-16 *** | | | | | | | |
| lage | 0.31533 | 0.03614 | 8.724 | 1.53e-11 *** | | | | | | | |
| --- | | | | | | | | | | | |
| Signif. codes: | 0 | '***' | 0.001 | '**' | 0.01 | '*' | 0.05 | '..' | 0.1 | ' ' | 1 |

Residual standard error: 0.03021 on 49 degrees of freedom

Multiple R-squared: 0.6083, Adjusted R-squared: 0.6003

F-statistic: 76.11 on 1 and 49 DF, p-value: 1.526e-11

anc1dat[, "sex"]: MALE

Call:

```
lm(formula = lfkl ~ lage, data = x[c(-49, -50, -19), ])
```

Residuals:

| Min | 1Q | Median | 3Q | Max |
|-----------|-----------|-----------|----------|----------|
| -0.047429 | -0.012818 | -0.005274 | 0.013495 | 0.077538 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) | | | | | | | |
|----------------|----------|------------|---------|--------------|------|-----|------|------|-----|-----|---|
| (Intercept) | 1.19361 | 0.04188 | 28.50 | < 2e-16 *** | | | | | | | |
| lage | 0.34662 | 0.03325 | 10.42 | 2.83e-12 *** | | | | | | | |
| --- | | | | | | | | | | | |
| Signif. codes: | 0 | '***' | 0.001 | '**' | 0.01 | '*' | 0.05 | '..' | 0.1 | ' ' | 1 |

```
Residual standard error: 0.02574 on 35 degrees of freedom
Multiple R-squared:  0.7563,    Adjusted R-squared:  0.7494
F-statistic: 108.6 on 1 and 35 DF,  p-value: 2.835e-12
```

Differences in intercepts are really small and similar than in previous models but now the precision (i.e. standard error) is much smaller for the models without outliers.

Note

It is often the case that by eliminating outliers, new outliers appear. This is simply because the “outlier” designation is usually based on a standardized residual: if you eliminate a couple of outliers, then the residual sums of squares decreases, i.e. the “average” (absolute) residual decreases. Thus, points which were not “far from the average” when the original average residual was comparatively large (i.e. were not “outliers”), may now become so because the average residual has been decreased. Remember also that as you eliminate outliers, N decreases, and the increase in R^2 may be more than compensated for by decreased power. So be wary of eliminating outliers!

14.6. Comparing model fits

As we have just seen, the process of fitting models to data is usually an iterative one. That is, there are, more often than not, several competing models that may be used to fit the data and it is left to the analyst to decide which model best balances goodness of fit (which we are usually trying to maximize) and complexity (which we are usually trying to minimize). In general, the strategy to use in regression and anova is to choose a simpler model when doing so does not reduce the goodness-of-fit by a significant amount. R can compute an F-statistic to compare the fit of two models. The null hypothesis in this situation is that there is no difference in goodness of fit between the models.

Exercise

Working with the `Anc1dat` data set, compare the fit of the ANCOVA and common simple regression models::

```
model.linear <- lm(lfkl ~ lage, data = anc1dat)
anova(model.ancova, model.linear)
```

| Res.Df | RSS | Df | Sum of Sq | F | Pr(>F) |
|--------|-----------|----|-----------|----|--------|
| 89 | 0.0716231 | NA | NA | NA | NA |

| Res.Df | RSS | Df | Sum of Sq | F | Pr(>F) |
|--------|-----------|----|------------|----------|-----------|
| 90 | 0.0731130 | -1 | -0.0014899 | 1.851324 | 0.1770653 |

The `anova()` function can compare the differences in sum of squares and degrees of freedom between the simpler and more complex models, takes the ratio of these two values to generate a mean square, and divides this by the mean square of the more complex model to generate an F-statistic. In the above case, there is insufficient evidence to reject the null hypothesis and we conclude that the simpler model, which is the simple linear regression, is the best model for these data. (Because these models differ by only the presence vs. absence of a single factor (sex), the P-value is the same as the p-value for sex in model 1.)

🔥 Exercise

Repeat the above procedure with the ANC3DAT data, rerunning the full ANCOVA with interaction (`lfkl ~ lage + locate + lage:locate`) and without interaction (`lfkl ~ lage + locate`), saving the model objects as you did above. Compare the fits of the two models. What do you conclude?

```
model.full.anc3dat <- lm(lfkl ~ lage + locate + lage:locate, data = anc3dat)
model.ancova.anc3dat <- lm(lfkl ~ lage + locate, data = anc3dat)
anova(model.full.anc3dat, model.ancova.anc3dat)
```

| Res.Df | RSS | Df | Sum of Sq | F | Pr(>F) |
|--------|-----------|----|------------|----------|-----------|
| 88 | 0.0513582 | NA | NA | NA | NA |
| 89 | 0.0604482 | -1 | -0.0090901 | 15.57541 | 0.0001592 |

In this case there is sufficient evidence to reject the null hypothesis and conclude that the full model with interaction is the best model to fit to the Anc3dat data. This is as we expected, given the fact that we found the interaction to be significant the in original analysis of the data. While no new information is gain from this model comparison in this case, this approach can be more usefully employed to compared nested models that differ in more than one term.

14.7. Bootstrap

```
#####
#####

# Bootstrap analysis

#
# Bootstrap analysis BCa confidence intervals
# Preferable when parameter distribution is far from normal
# Bootstrap 95% BCa CI for regression coefficients
library(boot)

# To simplify future modifications of the code in this file,
# copy the data to a generic mydata dataframe
mydata <- anc3dat

# create a myformula variable containing the formula for the model to be fitted
myformula <- as.formula(lfkl ~ lage + locate + lage:locate)

# function to obtain regression coefficients for each iteration
bs <- function(formula, data, indices) {
  d <- data[indices, ]
  fit <- lm(formula, data = d)
  return(coef(fit))
}

# bootstrapping with 1000 replications
results <- boot(data = mydata, statistic = bs, R = 1000, formula = myformula)

# view results
results
boot_res <- summary(results)
rownames(boot_res) <- names(results$t0)
boot_res

op <- par(ask = TRUE)
for (i in 1:length(results$t0)) {
```

```
plot(results, index = i)
title(names(results$t0)[i])
}
par(op)

# get 95% confidence intervals
for (i in 1:length(results$t0)) {
  cat("\n", names(results$t0)[i], "\n")
  print(boot.ci(results, type = "bca", index = i))
}
```

14.8. Permutation test

```
#####
#####
# Permutation test
#
# using lmperm library
# To simplify future modifications of the code in this file,
# copy the data to a generic mydata dataframe
mydata <- anc3dat
# create a myformula variable containing the formula for the
# model to be fitted
myformula <- as.formula(lfkl ~ lage + locate + lage:locate)
library(lmPerm)
# Fit desired model on the desired dataframe
mymodel <- lm(myformula, data = mydata)
# Calculate p-values for each term by permutation
# Note that lmp centers numeric variable by default, so to
# get results that are
# consistent with standard models, it is necessary to set
# center=FALSE
```

```
mymodelProb <- lmp(myformula,  
  data = mydata, center = FALSE,  
  perm = "Prob"  
)  
summary(mymodel)  
summary(mymodelProb)
```

Part IV.

Generalized linear models

Chapter 15

Generalized linear model, `glm`

15.1. Lecture



Figure 15.1.: Dream pet dragon

```
m1 <- glm(fish ~ french_captain, data = dads_joke, family = poisson)
```

15.1.1. Distributions

15.1.1.1. Continuous linear

- Gaussian

15.1.1.2. Count data

- poisson
- negative binomial
- quasi-poisson
- generalized poisson
- conway-maxwell poisson

15.1.1.3. censored distribution

15.1.1.4. zero-inflated / hurdle distribution

- zero-inflated/zero-truncated poisson
- censored poisson

15.1.1.5. zero-truncated distribution

15.1.1.6. zero-one-inflated distribution

see https://cran.r-project.org/web/packages/brms/vignettes/brms_families.html see also MCMCglmm course notes

for help on description and to add some plots about those distribution

15.2. Practical



This section need to be severely updated

15.2.1. Logistic regression

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate  1.9.3     v tidyverse  1.3.1
v purrr     1.0.2

-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()

i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become er
```

```
library(DHARMa)
```

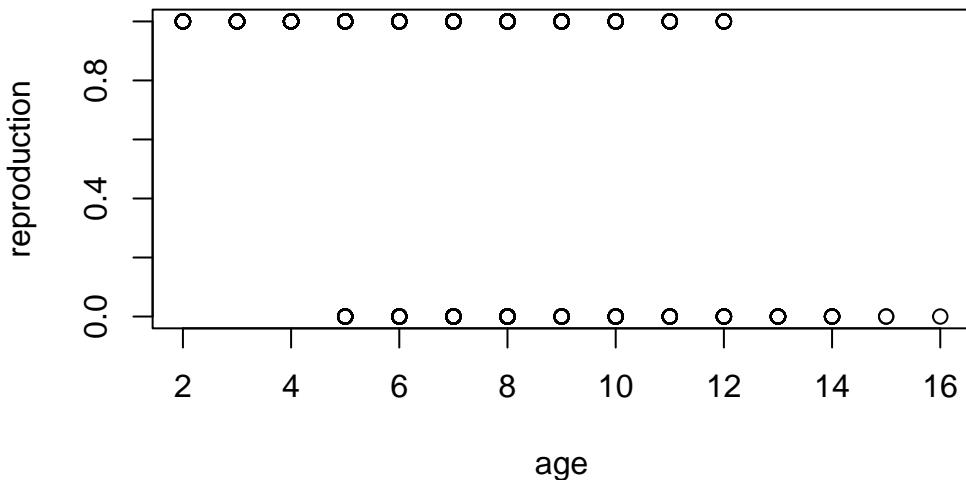
This is DHARMa 0.4.6. For overview type '?DHARMa'. For recent changes, type news(package = 'DHARMa')

```
library(performance)
```

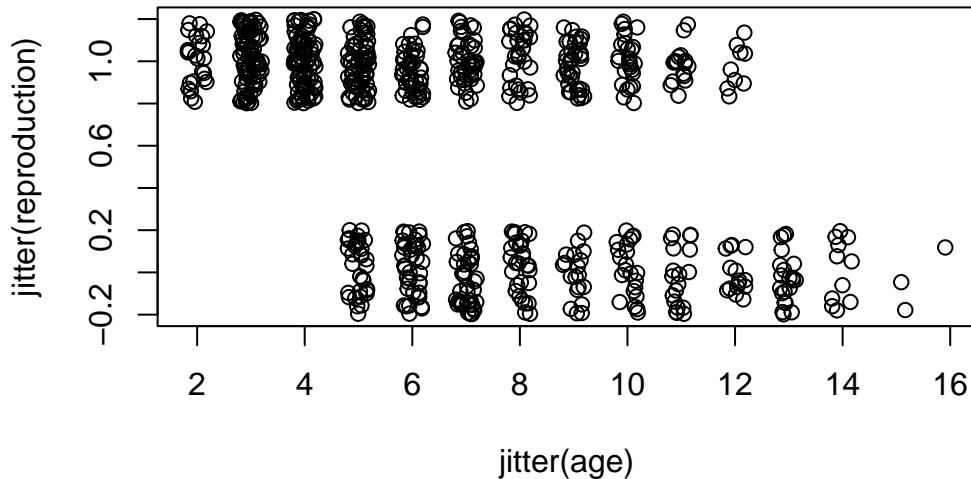
```
mouflon <- read.csv("data/mouflon.csv")
mouflonc <- mouflon[order(mouflon$age),]

mouflonc$reproduction <- ifelse(mouflonc$age < 13, mouflonc$reproduction, 0)
mouflonc$reproduction <- ifelse(mouflonc$age > 4, mouflonc$reproduction, 1)

plot(reproduction ~ age, mouflonc)
```



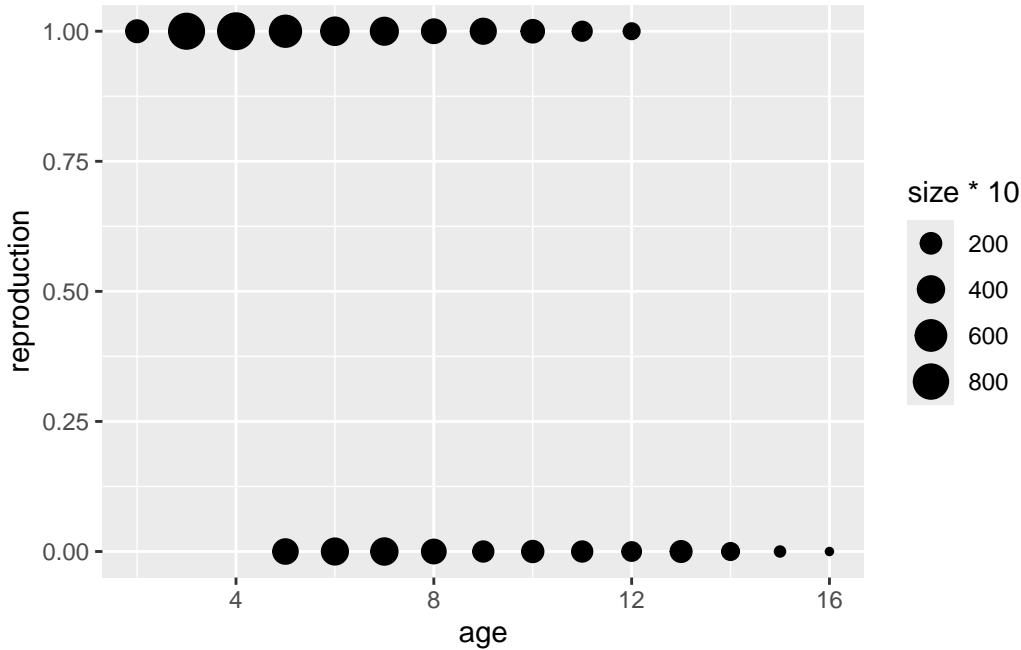
```
plot(jitter(reproduction) ~ jitter(age), mouflonc)
```



```
bubble <- data.frame(age = rep(2:16, 2),
                      reproduction = rep(0:1, each = 15),
                      size = c(table(mouflonc$age, mouflonc$reproduction)))
bubble$size <- ifelse(bubble$size == 0, NA, bubble$size)
ggplot(data = bubble, aes(x = age, y = reproduction)) +
```

```
geom_point(aes(size = size*10))
```

Warning: Removed 7 rows containing missing values or values outside the scale range
`geom_point()`).



```
m1 <- glm(reproduction ~ age,
           data = mouflonc,
           family = binomial)
summary(m1)
```

Call:

```
glm(formula = reproduction ~ age, family = binomial, data = mouflonc)
```

Coefficients:

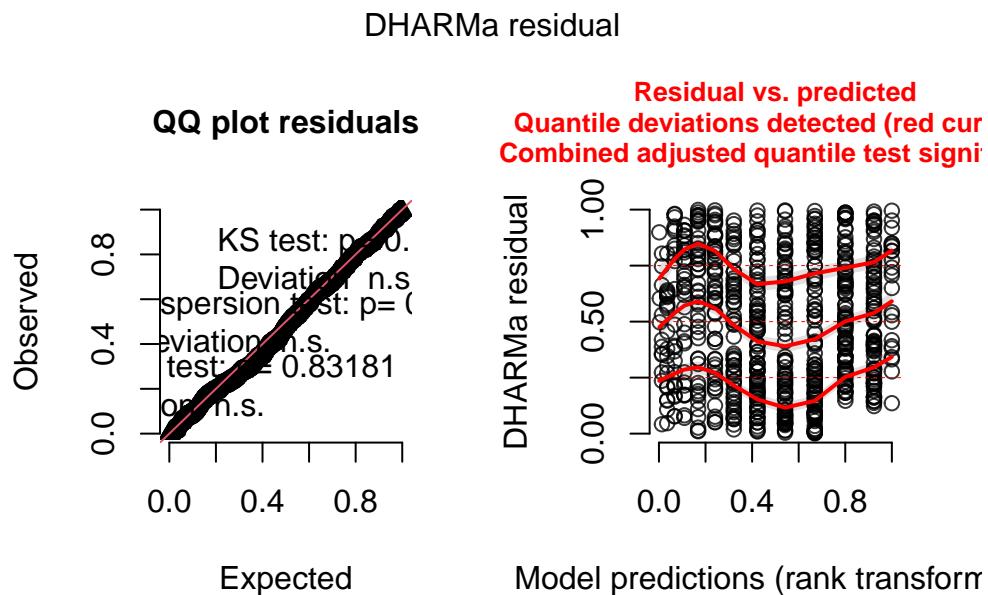
| | Estimate | Std. Error | z value | Pr(> z) | | |
|----------------|----------|------------|----------|------------|---------|---|
| (Intercept) | 3.19921 | 0.25417 | 12.59 | <2e-16 *** | | |
| age | -0.36685 | 0.03287 | -11.16 | <2e-16 *** | | |
| --- | | | | | | |
| Signif. codes: | 0 '***' | 0.001 '**' | 0.01 '*' | 0.05 '.' | 0.1 ' ' | 1 |

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 928.86 on 715 degrees of freedom
 Residual deviance: 767.51 on 714 degrees of freedom
 (4 observations deleted due to missingness)
 AIC: 771.51

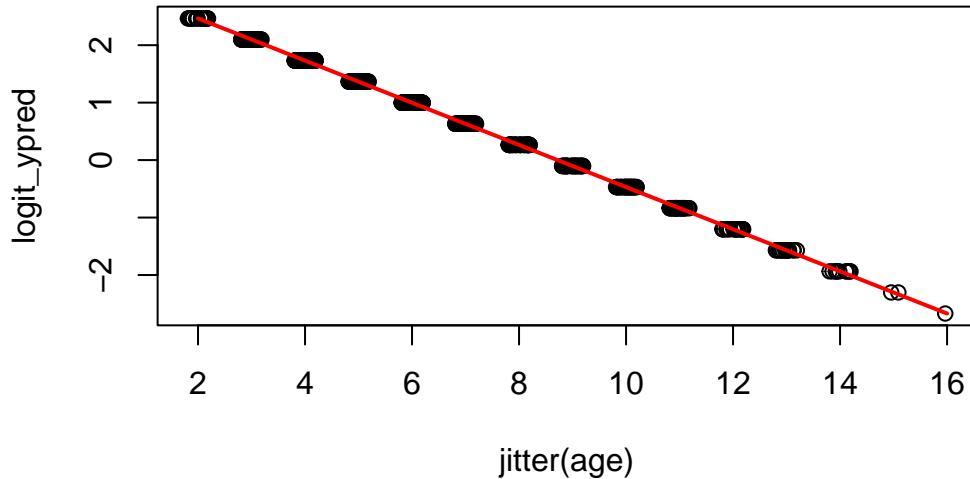
Number of Fisher Scoring iterations: 4

```
simulationOutput <- simulateResiduals(m1)
plot(simulationOutput)
```



plotting the model prediction on the link (latent) scale

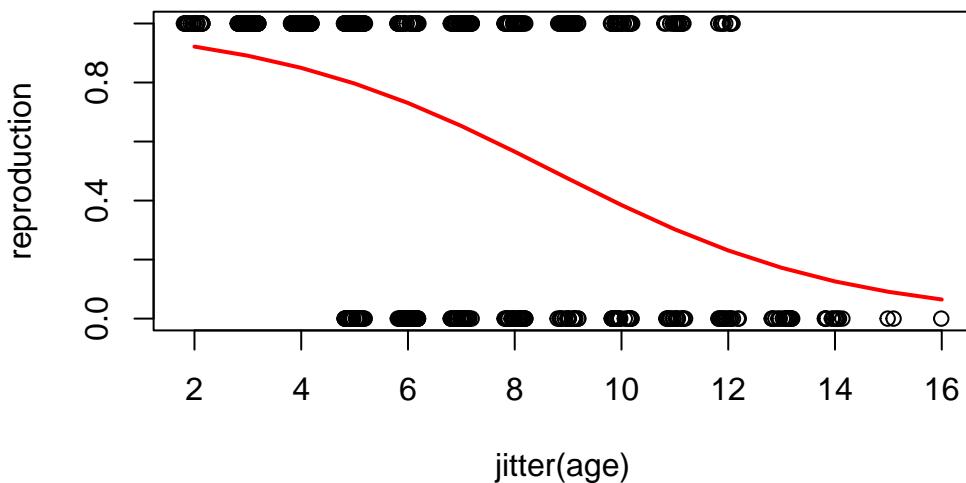
```
mouflonc$logit_ypred <- 3.19921 -0.36685 * mouflonc$age
plot(logit_ypred ~ jitter(age), mouflonc)
points(mouflonc$age, mouflonc$logit_ypred, col="red", type = "l", lwd = 2)
```



plotting on the observed scale

```
mouflonc$ypred <- exp(mouflonc$logit_ypred) / (1 + exp(mouflonc$logit_ypred)) # inverse of logit

plot(reproduction ~ jitter(age), mouflonc)
points(mouflonc$age, mouflonc$ypred, col="red", type = "l", lwd = 2)
```



Enfin, pour se simplifier la vie, il est aussi possible de récupérer les valeurs prédictes de y directement

```
plot(x,y)
myreg <- glm(y~x, family=binomial(link=logit))
ypredit <- myreg$fitted
o=order(x)
points(x[o],ypredit[o], col="red", type="l", lwd=2)
```

```
m2 <- glm(reproduction ~ age + mass_sept + as.factor(sex_lamb) + mass_gain + density + temp,
           data = mouflon,
           family = binomial)

summary(m2)
```

Call:

```
glm(formula = reproduction ~ age + mass_sept + as.factor(sex_lamb) +
    mass_gain + density + temp, family = binomial, data = mouflon)
```

Coefficients:

| | Estimate | Std. Error | z value | Pr(> z) |
|----------------------|-----------|------------|---------|--------------|
| (Intercept) | 1.622007 | 1.943242 | 0.835 | 0.403892 |
| age | -0.148567 | 0.033597 | -4.422 | 9.78e-06 *** |
| mass_sept | 0.029878 | 0.016815 | 1.777 | 0.075590 . |
| as.factor(sex_lamb)1 | -0.428169 | 0.166156 | -2.577 | 0.009969 ** |
| mass_gain | -0.094828 | 0.026516 | -3.576 | 0.000348 *** |
| density | -0.018132 | 0.003518 | -5.154 | 2.55e-07 *** |
| temp | 0.037244 | 0.138712 | 0.269 | 0.788313 |
| --- | | | | |

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 916.06 on 674 degrees of freedom

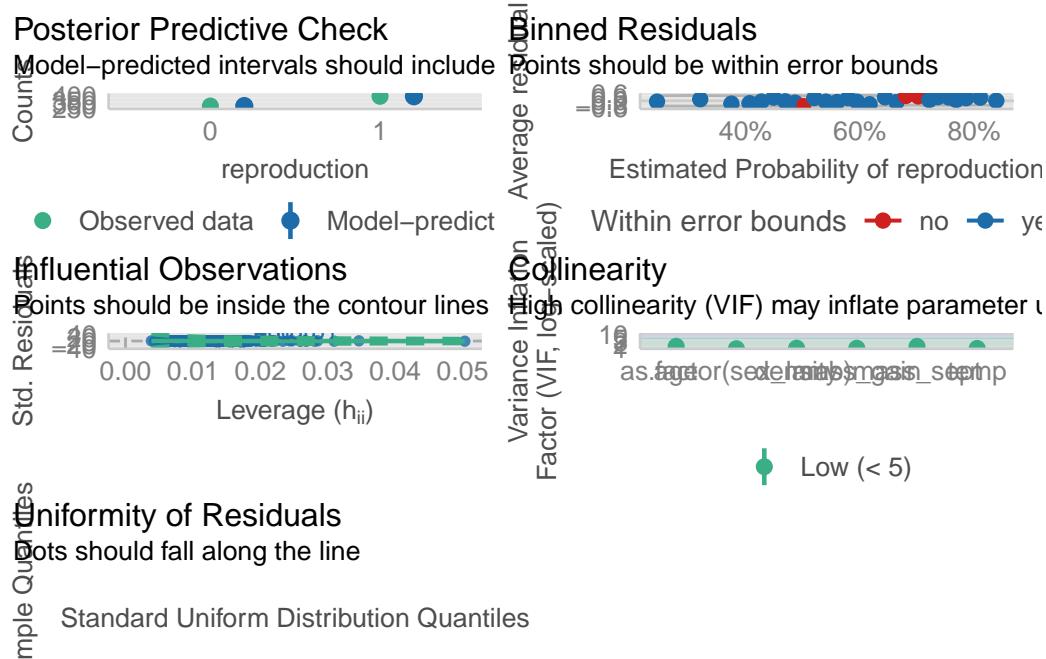
Residual deviance: 845.82 on 668 degrees of freedom

(45 observations deleted due to missingness)

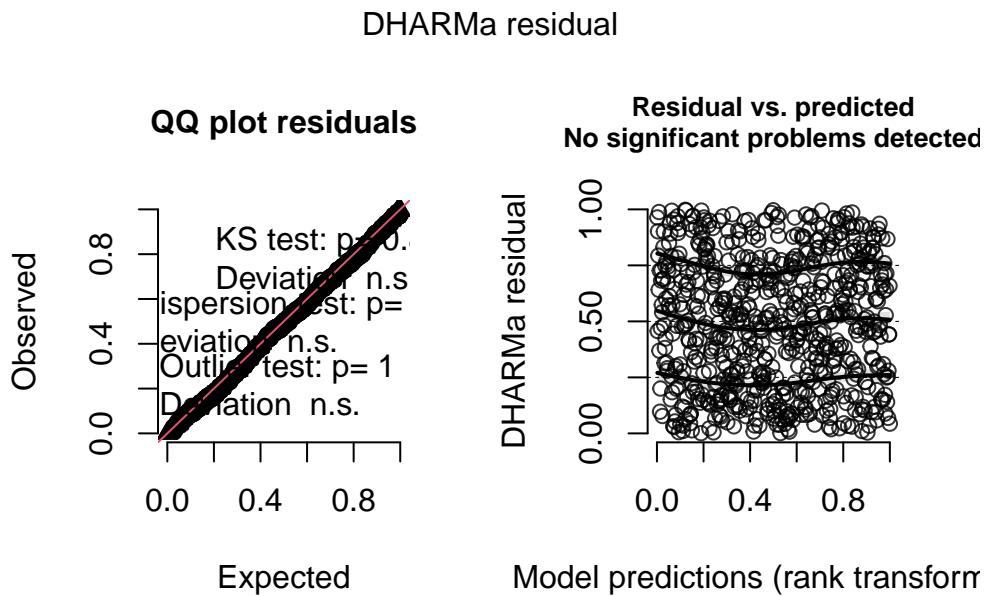
AIC: 859.82

Number of Fisher Scoring iterations: 4

```
check_model(m2)
```



```
simulationOutput <- simulateResiduals(m2)
plot(simulationOutput)
```



15.2.1.1. previous offspring sex effect

```
pred.data <- data.frame(
  age = mean(mouflon$age),
  mass_sept = mean(mouflon$mass_sept),
  sex_lamb = c(0,1),
  mass_gain = mean(mouflon$mass_gain),
  density = mean(mouflon$density),
  temp = mean(mouflon$temp, na.rm =TRUE))

predict(m2, newdata = pred.data)
```

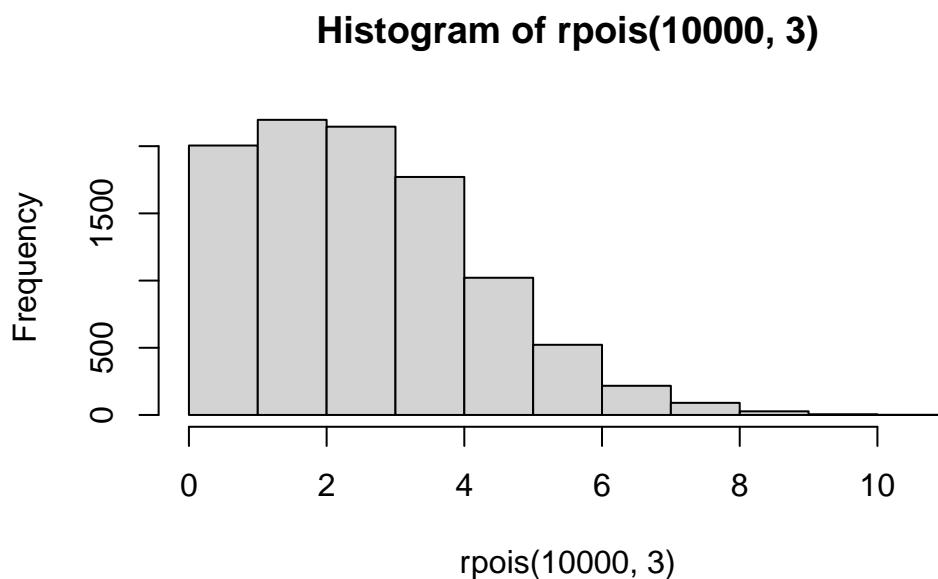
```
1          2
0.6225895 0.1944205
```

15.2.2. Poisson regression

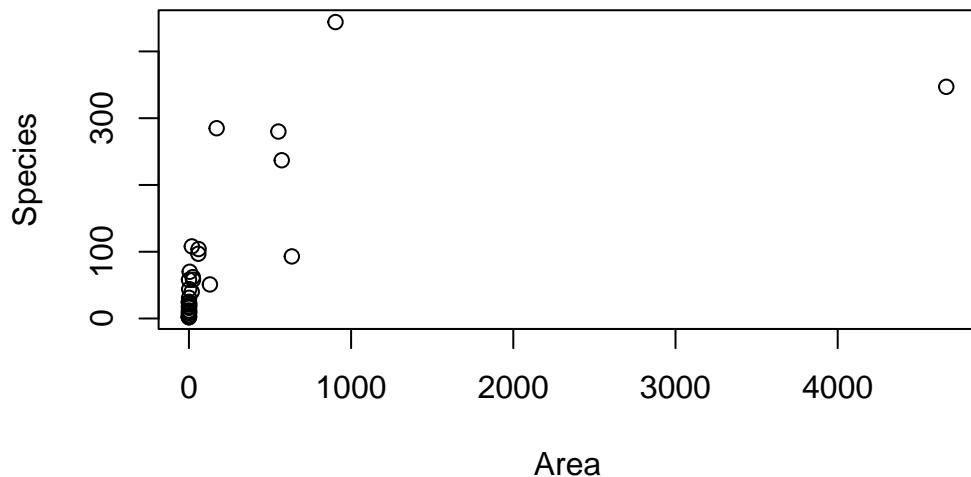
data on galapagos islands species richness model of total number of species model of proportion of native model of density of species

Fit 3 models - model of total number of species - model of proportion of endemics to total - model of species density

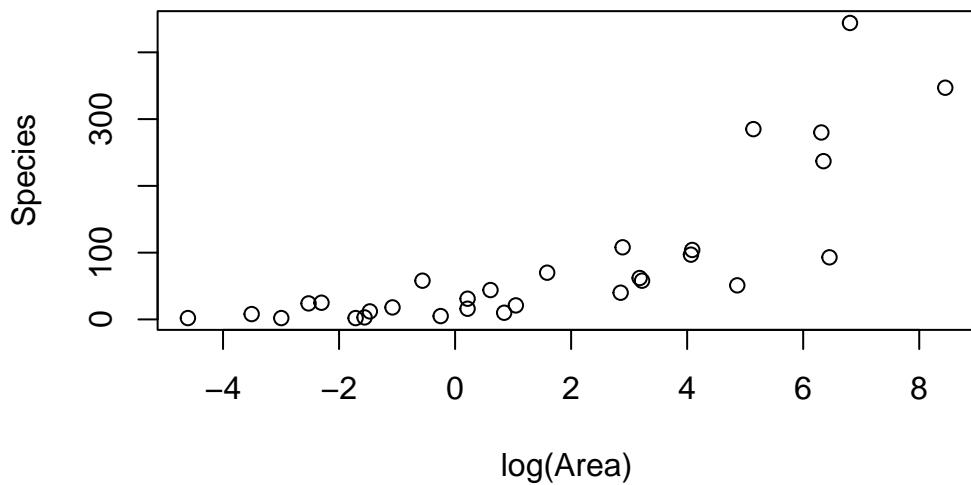
```
hist(rpois(10000,3))
```



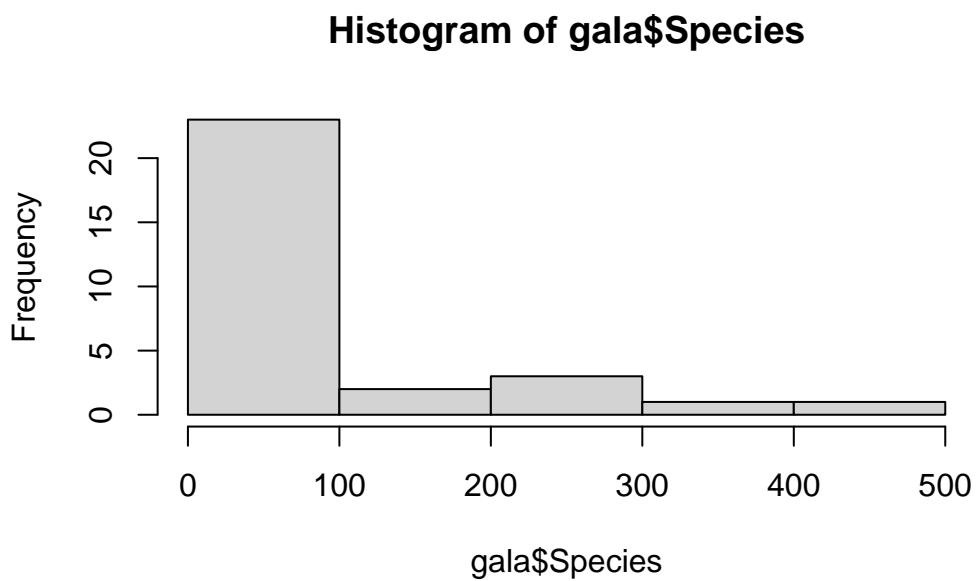
```
#  
gala <- read.delim2("data/gala.txt")  
plot(Species ~ Area, gala)
```



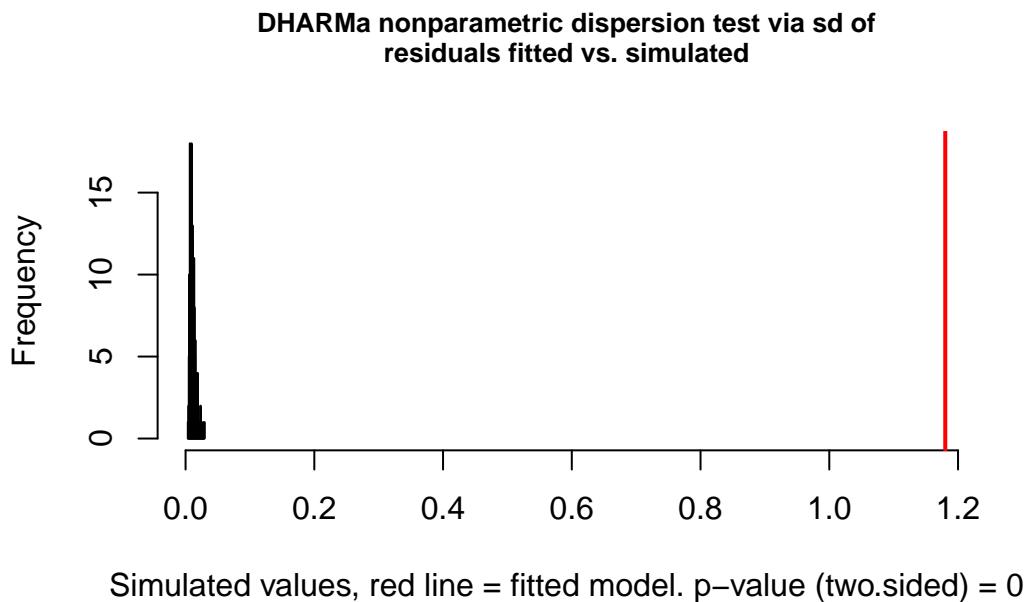
```
plot(Species ~ log(Area), gala)
```



```
hist(gala$Species)
```



```
modpl <- glm(Species ~ Area + Elevation + Nearest, family=poisson, gala)
res <- simulateResiduals(modpl)
testDispersion(res)
```

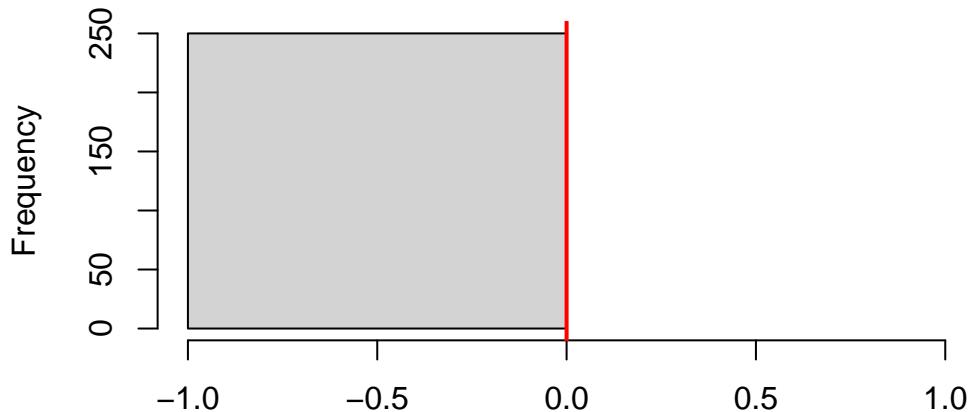


DHARMA nonparametric dispersion test via sd of residuals fitted vs.
simulated

```
data: simulationOutput  
dispersion = 110.32, p-value < 2.2e-16  
alternative hypothesis: two.sided
```

```
testZeroInflation(res)
```

DHARMA zero-inflation test via comparison to expected zeros with simulation under H0 = fitted model



Simulated values, red line = fitted model. p-value (two.sided) = 1

DHARMA zero-inflation test via comparison to expected zeros with simulation under H0 = fitted model

```
data: simulationOutput  
ratioObsSim = NaN, p-value = 1  
alternative hypothesis: two.sided
```

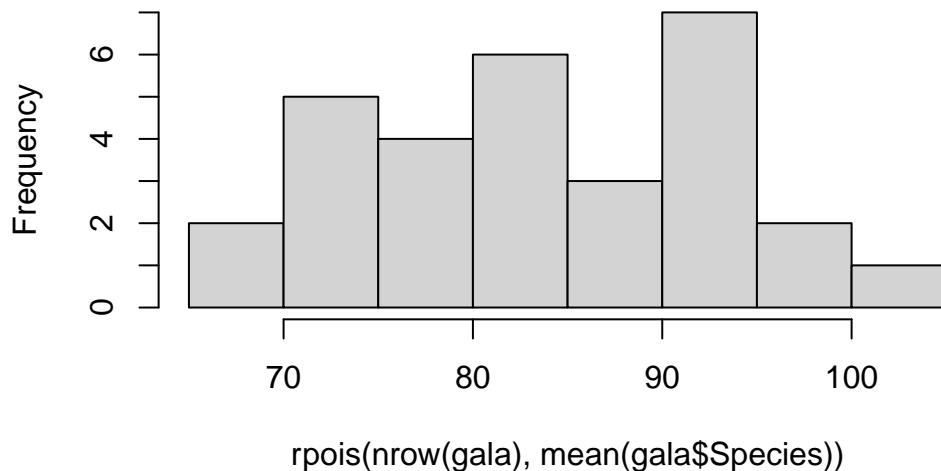
```
mean(gala$Species)
```

```
[1] 85.23333
```

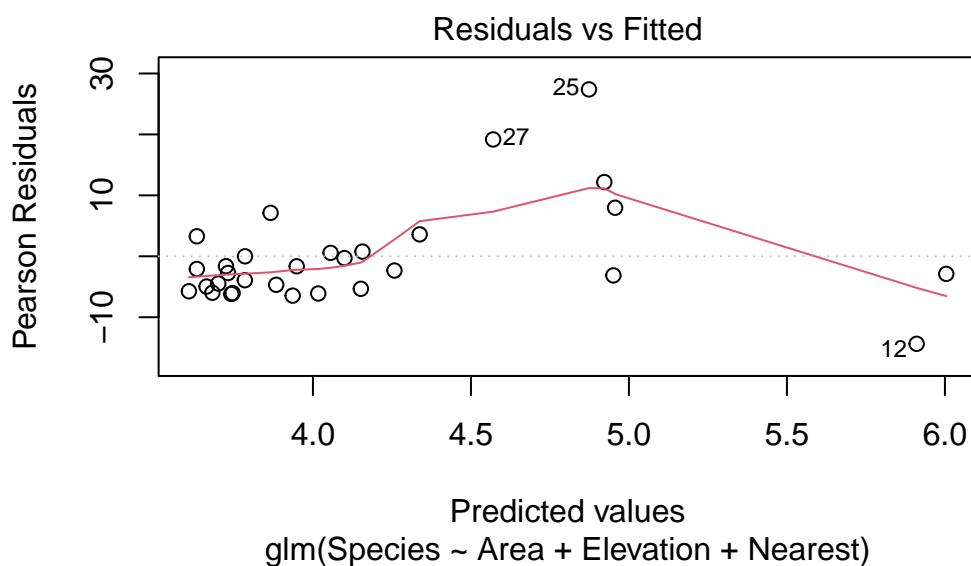
```
var(gala$Species)
```

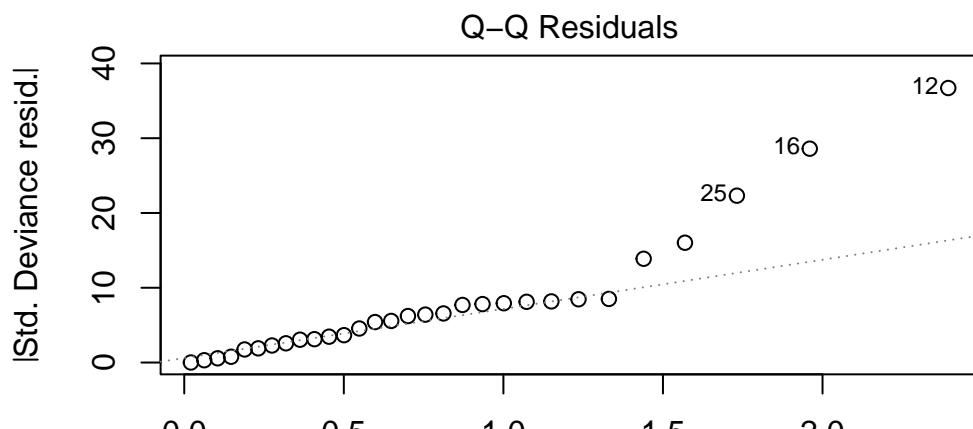
```
[1] 13140.74
```

```
hist(rpois(nrow(gala),mean(gala$Species)))
```

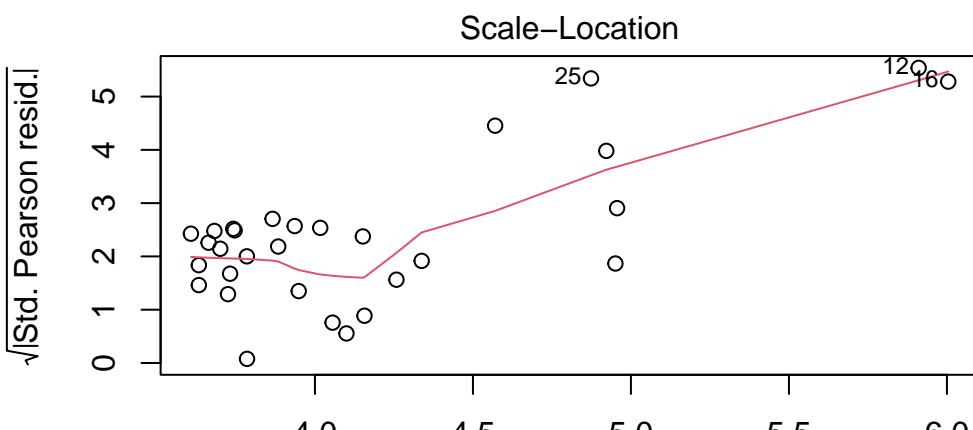
Histogram of rpois(nrow(gala), mean(gala\$Species))

```
plot(modp1)
```





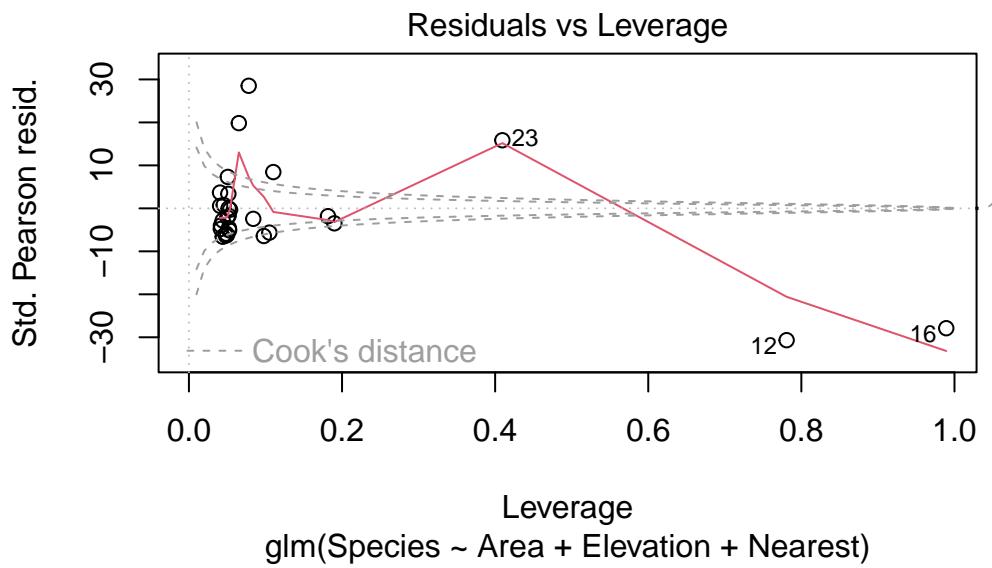
Theoretical Quantiles
`glm(Species ~ Area + Elevation + Nearest)`



Predicted values
`glm(Species ~ Area + Elevation + Nearest)`

Warning in `sqrt(crit * p * (1 - hh)/hh)`: NaNs produced

Warning in `sqrt(crit * p * (1 - hh)/hh)`: NaNs produced



Chapter 16

Frequency data and Poisson Regression

After completing this laboratory exercise, you should be able to:

- Create and manipulate data files in R to analyze count data
- Use R to test an external hypothesis about a particular population using count data.
- Use R to test for independence in two-way tables
- Use R to fit Poisson regression and log-linear models to count data

16.1. R packages and data

For this lab you need:

- R packages:
 - vcd
 - vcdExtra
 - car
- data files
 - USPopSurvey.csv
 - loglin.csv
 - sturgdat.csv

16.2. Organizing the data: 3 forms

Some biological experiments yield count data, e.g., the number of plants infected by a plant pathogen under different exposure regimes, the number of male and female turtles hatched under different incubation temperature treatments (in turtles, sex is temperature dependent!), etc. Usually the statistical issue here is whether the proportion of individuals in different categories (e.g., infected versus uninfected, male versus female, etc.) differs significantly among treatments. To examine this question, we can set up a data file that lists the number of individuals in each category. There are 3 ways to do this. You should be able to decide which one is appropriate, and how to convert between them with R.

The file `USPopSurvey.csv` contains the results of a 1980 U.S population survey of a mid-eastern town:

```
USPopSurvey <- read.csv("data/USPopSurvey.csv")
USPopSurvey
```

| ageclass | sex | frequency |
|----------|--------|-----------|
| 0-9 | female | 17619 |
| 10-19 | female | 17947 |
| 20-29 | female | 21344 |
| 30-39 | female | 19138 |
| 40-49 | female | 13135 |
| 50-59 | female | 11617 |
| 60-69 | female | 11053 |
| 70-79 | female | 7712 |
| 80+ | female | 4114 |
| 0-9 | male | 17538 |
| 10-19 | male | 18207 |
| 20-29 | male | 21401 |
| 30-39 | male | 18837 |
| 40-49 | male | 12568 |
| 50-59 | male | 10661 |
| 60-69 | male | 9374 |
| 70-79 | male | 5348 |
| 80+ | male | 1926 |

Note that there are 18 lines and 3 columns in this file. Each line lists the number of individuals (`frequency`) of a given sex and age class. There are (`sum(USPopSurvey$frequency)`) 239539 individuals that were classified into the 18 (2 sexes x 9 age classes) categories. This way of presenting data is the **frequency form**. It is a compact way to present the data when there are only categorical variables.

When there are continuous variables, the frequency form can't be utilized (or provides no gain since each observation could possibly have a different values for the continuous variable(s)). Data have therefore to be stored in **case form** where each observation (`individual`) represents one line in the data file, and each variable is a column. Conveniently, the `vcdExtra`  includes the `expand.dft()` function to convert from the frequency to case form. For example, to create a data frame with 239539 lines and 2 columns (sex and ageclass):

```
USPopSurvey.caseform <- expand.dft(USPopSurvey, freq = "frequency")
head(USPopSurvey.caseform)
```

| ageclass | sex |
|----------|--------|
| 0-9 | female |

```
tail(USPopSurvey.caseform)
```

| | ageclass | sex |
|--------|----------|------|
| 239534 | 80+ | male |
| 239535 | 80+ | male |
| 239536 | 80+ | male |
| 239537 | 80+ | male |
| 239538 | 80+ | male |
| 239539 | 80+ | male |

Finally, these data can also be represented in **table form** (contingency table) where each variable is represented by a dimension of the n-dimensional table (here, for example, rows could represent each age class, and columns each sex),

and the cells of the resulting table contain the frequencies. The table form can be created from the case or frequency form by the `xtabs()` command with slightly different syntax:

```
# convert case form to table form
xtabs(~ ageclass + sex, USPopSurvey.caseform)
```

```
sex
ageclass female male
0-9      17619 17538
10-19    17947 18207
20-29    21344 21401
30-39    19138 18837
40-49    13135 12568
50-59    11617 10661
60-69    11053  9374
70-79    7712   5348
80+      4114   1926
```

```
# convert frequency form to table form
xtabs(frequency ~ ageclass + sex, data = USPopSurvey)
```

```
sex
ageclass female male
0-9      17619 17538
10-19    17947 18207
20-29    21344 21401
30-39    19138 18837
40-49    13135 12568
50-59    11617 10661
60-69    11053  9374
70-79    7712   5348
80+      4114   1926
```

Table 16.4.: (#tab:unnamed-chunk-1)Tools for converting among different forms for categorical data.

| From (Row) \ To (column) | Case form | Frequency form | Table form |
|--------------------------|----------------------------|-------------------------------|-----------------------------------|
| Case form | | <code>xtabs(~ A + B)</code> | <code>table(A, B)</code> |
| Frequency form | <code>expand.dft(X)</code> | | <code>xtabs(count ~ A + B)</code> |
| Table form | <code>expand.dft(X)</code> | <code>as.data.frame(X)</code> | |

16.3. Graphs for contingency tables and testing for independence

Contingency tables can be used to test for independence. By this we mean to answer the question: Is the classification of observations according to one variable (say, sex) independent from the classification by another variable (say, ageclass). In other words, is the proportion of males and females independent of age, or does it vary among age classes?

The vcd  includes a `mosaic()` function useful to graphically display contingency tables:

```
library(vcd)
USTable <- xtabs(frequency ~ ageclass + sex, data = USPopSurvey) # save the table form as USTable
# Mosaic plot of the contingency table
mosaic(USTable)
```

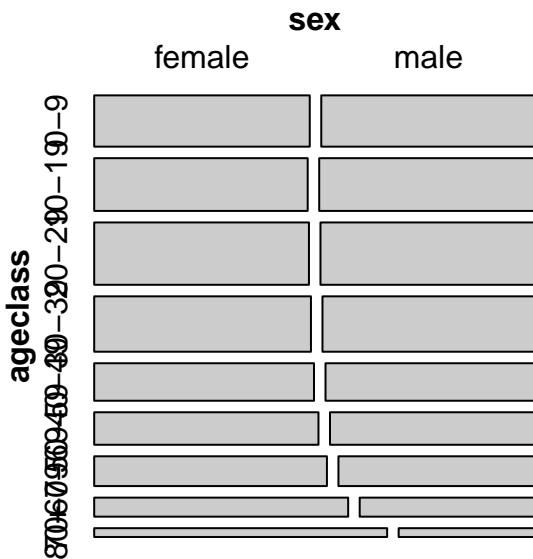


Figure 16.1.: Mosaic plot of sex classes per age

Mosaic plots represent the proportion of observations in each combination of categories (here there are 18 categories, 2 sexes x 9 age classes). Categories with a higher proportion of observations are represented by larger rectangles. Visually, one can see that males and females are approximately equal for young age classes, but that the proportion of females increases quite a bit amongst the elders.

The Chi square test can be used to test the null hypothesis that the proportion of males and females does not differ among age classes:

```
# Test of independence
chisq.test(USTable) # runs chi square test of independence of sex and age class
```

Pearson's Chi-squared test

```
data: USTable
X-squared = 1162.6, df = 8, p-value < 2.2e-16
```

From this we conclude there is ample evidence to reject the null hypothesis that ageclass and sex are independent, which isn't particularly surprising.

The mosaic plot from the vcd  can be shaded to show the categories that contribute most to the lack of independence:

```
# Mosaic plot of the contingency table with shading
mosaic(USTable, shade = TRUE)
```

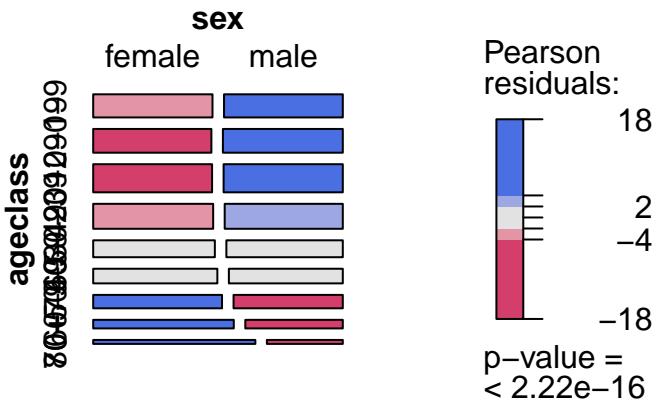


Figure 16.2.: Mosaic plot of sex by age with colours

The shading of each rectangle is proportional to the extent that observed frequencies deviate from what would be expected if sex and age class were independent. The age classes 40-49 and 50-59 have a sex ratio about equal to the overall sex:ratio for the entire dataset, and appear in grey. There are more young males and old females than expected if sex ratio did not change with age, and these rectangles are coded in blue. On the other hand, there are fewer young females and old males than if sex ratio did not change with age, and these rectangles are red coded. Note that the p-value printed on the right of the graph is for the chi-square test that assumes that observations are independent.

The estimation of p-value associated with the chi square statistic is less than ideal when expected frequencies are small in some of the cells, particularly for 2x2 contingency tables. Two options are then preferred, depending on the number of observations. For large samples, like in this example with more than 200,000 cases(!), a Monte Carlo approach is suggested and can be obtained by adding `simulate.p.value=TRUE` as an argument to the `chisq.test()` function

```
# Monte-carlo estimation of p value (better for small n)
chisq.test(USTable, simulate.p.value = TRUE, B = 10000)
```

Pearson's Chi-squared test with simulated p-value (based on 10000 replicates)

```
data: USTable
X-squared = 1162.6, df = NA, p-value = 9.999e-05
```

Here, the simulation was done B=10000 times, and the chi square value observed with the data was never exceeded so p is estimated as $1/10001=9.999e-05$, which is much larger than the p-value estimated from the theoretical chi square distribution ($p < 2.2e-16$). This difference in p-value is at least partly an artifact of the number of simulations. To estimate p values as small as $1e-16$, at least 10^{16} simulations must be run. And I am not THAT patient. For small tables with relatively low expected frequencies, Fisher's exact test can be run to test for independence. This result is unbiased if row and column totals are fixed, but is conservative (i.e. it will incorrectly fail to reject the null more often than expected) if row and/ or column totals are not fixed.

But this test will fail for large samples, like in this example:

```
# Fisher exact test for contingency tables (small samples and small tables)
fisher.test(USTable) # fails here because too many observations
```

Error in fisher.test(USTable): FEXACT error 40.

Out of workspace.

```
fisher.test(USTable, simulate.p.value = TRUE, B = 10000)
```

Fisher's Exact Test for Count Data with simulated p-value (based on 10000 replicates)

```
data: USTable
p-value = 9.999e-05
alternative hypothesis: two.sided
```

16.4. Log-linear models as an alternative to Chi-square test for contingency tables

By now, hopefully, you have learned to appreciate the flexibility and generality of general linear models and you realize that the t-test is a special, simple, case of a linear model with one categorical independent variable. The analysis of contingency tables by chi square test can similarly be generalized. Indeed, generalized linear models for poisson distributed data can be used when the dependent variable are frequencies (count data) and the independent variables can be categorical only (like for contingency tables, these are also called log- linear models), continuous only (Poisson regression), or a combination of categorical and continuous independent variables (this, too is a Poisson regression, but with added categorical variables, analogous to an ANCOVA sensu largo).

Such models predict the natural log frequency of observations given the independent variables. Like for linear models assuming normality of residuals, one can assess the overall quality of the fit (by AIC for example), and the significance of terms (say by comparing the fit of models including or excluding particular terms). One can even, if desired, obtain estimates of the parameters for each model term, with confidence intervals and p-values for the null hypothesis that the value of the parameter is 0.

The `glm()` function with the option `family=poisson()` allows the estimation, by maximum likelihood, of linear models for count data. One “peculiarity” of fitting such models to contingency table data is that generally the only terms of interest are the interactions. Going back to the population survey data in frequency form, with sex and ageclass as independent variables, one can fit a `glm` model by:

```
mymodel <- glm(frequency ~ sex * ageclass, family = poisson(), data = USPopSurvey)
summary(mymodel)
```

Call:

```
glm(formula = frequency ~ sex * ageclass, family = poisson(),
  data = USPopSurvey)
```

Coefficients:

| | Estimate | Std. Error | z value | Pr(> z) |
|---------------|-----------|------------|----------|-------------|
| (Intercept) | 9.776733 | 0.007534 | 1297.730 | < 2e-16 *** |
| sexmale | -0.004608 | 0.010667 | -0.432 | 0.6657 |
| ageclass10-19 | 0.018445 | 0.010605 | 1.739 | 0.0820 . |

| | | | | | | | | | | | |
|-----------------------|-----------|----------|---------|----------|------|-----|------|------|-----|-----|---|
| ageclass20-29 | 0.191793 | 0.010179 | 18.842 | < 2e-16 | *** | | | | | | |
| ageclass30-39 | 0.082698 | 0.010441 | 7.921 | 2.36e-15 | *** | | | | | | |
| ageclass40-49 | -0.293697 | 0.011528 | -25.477 | < 2e-16 | *** | | | | | | |
| ageclass50-59 | -0.416508 | 0.011951 | -34.850 | < 2e-16 | *** | | | | | | |
| ageclass60-69 | -0.466276 | 0.012134 | -38.428 | < 2e-16 | *** | | | | | | |
| ageclass70-79 | -0.826200 | 0.013654 | -60.511 | < 2e-16 | *** | | | | | | |
| ageclass80+ | -1.454582 | 0.017316 | -84.004 | < 2e-16 | *** | | | | | | |
| sexmale:ageclass10-19 | 0.018991 | 0.014981 | 1.268 | 0.2049 | | | | | | | |
| sexmale:ageclass20-29 | 0.007275 | 0.014400 | 0.505 | 0.6134 | | | | | | | |
| sexmale:ageclass30-39 | -0.011245 | 0.014803 | -0.760 | 0.4475 | | | | | | | |
| sexmale:ageclass40-49 | -0.039519 | 0.016416 | -2.407 | 0.0161 | * | | | | | | |
| sexmale:ageclass50-59 | -0.081269 | 0.017136 | -4.742 | 2.11e-06 | *** | | | | | | |
| sexmale:ageclass60-69 | -0.160154 | 0.017633 | -9.083 | < 2e-16 | *** | | | | | | |
| sexmale:ageclass70-79 | -0.361447 | 0.020747 | -17.422 | < 2e-16 | *** | | | | | | |
| sexmale:ageclass80+ | -0.754343 | 0.029598 | -25.486 | < 2e-16 | *** | | | | | | |
| --- | | | | | | | | | | | |
| Signif. codes: | 0 | '***' | 0.001 | '**' | 0.01 | '*' | 0.05 | '..' | 0.1 | ' ' | 1 |

(Dispersion parameter for poisson family taken to be 1)

```

Null deviance: 5.3611e+04 on 17 degrees of freedom
Residual deviance: 6.5463e-12 on 0 degrees of freedom
AIC: 237.31

```

Number of Fisher Scoring iterations: 2

Fitting the full model, with the sex:ageclass interaction, allows the proportion of males and females to vary among ageclass levels, and hence to estimate exactly the frequencies for each combination of sex and ageclass (note that the deviance residuals are all 0's and that the Residual deviance is also approximately zero).

A masochist can use the coefficient table to obtain the predicted values for sex and ageclass categories by summing the appropriate coefficients. The predicted values, like for multiway ANOVA model, are obtained by combining the coefficients. Remembering that the first level of a factor (alphabetically) is used as a reference, here the coefficient for the intercept (9.776733) is the predicted value for the natural log of the number of observations for females in the

first alphabetical ageclass (0 to 9). Indeed $e^{9.776733}$ is approximately equal to 17619, the observed number of females in that age class. For example, for males in the 80+ ageclass, calculate the antilog of the coefficient for the intercept (for female in the youngest age class) plus the coefficient for sexmale (equal to the difference between ln frequency of females and males overall), plus the coefficient for the ageclass 80+ corresponding in the difference in frequency on average between the oldest and reference ageclass, plus the coefficient for the interaction terms sexmale:ageclass80+ (corresponding to the difference in the proportion of male for this ageclass compared to the youngest ageclass), so $\ln(\text{frequency}) = 9.776733 - 0.004608 - 1.454582 - 0.754343 = 7.5632$, and the frequency is equal to $e^{7.5632} = 1926$

Although there are numerous p values in this output, they are not really helpful. To test whether the effect of sex on observed frequency is the same across ageclass levels, i.e is sex and age are independent, one needs to fit a model where the interaction sex:ageclass is removed, and see how badly this affects the fit. The `Anova()` function of the `car` package provides a handy shortcut:

```
Anova(mymodel, type = 3, test = "LR")
```

| | LR Chisq | Df | Pr(>Chisq) |
|--------------|--------------|----|------------|
| sex | 1.866202e-01 | 1 | 0.6657446 |
| ageclass | 2.107464e+04 | 8 | 0.0000000 |
| sex:ageclass | 1.182152e+03 | 8 | 0.0000000 |

The use of `type=3` and `test="LR"` ensures that the test performed to compare the full and reduced models is the Likelihood Ratio Chi-Square using the Residual deviance, and that it is a partial test, not a sequential one.

According to these tests, there is no main effect of sex ($p=0.667$) but there is a main effect of ageclass and a significant sex:ageclass interaction. The significant interaction means that the effect of sex on frequency varies with ageclass, or that the sex ratio varies with age. The main effect of ageclass means that the frequency of individuals varies with age (i.e some ageclass are more populous than others). The absence of a main effect of sex suggests that there are approximately the same frequency of males and females in this sample (although, since there is an interaction, you have to be careful in making this assertion. It is “true” overall, but appears incorrect for individual age categories).

16.5. Testing an external hypothesis

The above test of independence is that of an internal hypothesis because the proportions used to calculate the expected frequencies assuming independence of sex and ageclass come from the data (i.e. the overall proportion of males in

females in the entire dataset, and the proportions of individuals in each ageclass, males and females combined.

To test the (external) null hypothesis that the sex ratio is 1:1 for the youngest individuals (ageclass 0-9), one has to compute the 2 X 2 table of observed and expected frequencies. The expected frequencies are obtained simply by summing male and female frequencies and dividing by two.

R program to create and analyze a 2x2 table to test an external hypothesis

```
### Produce a table of obs vs exp for 0-9 age class
Popn0.9 <- rbind(c(17578, 17578), c(17619, 17538))

### Run X2 test on above table
chisq.test(Popn0.9, correct = F) ### X2 without Yates
chisq.test(Popn0.9) ### X2 with Yates
```

🔥 Exercise

Test the null hypothesis that the proportion of male and female at birth is equal. What is your conclusion? Do you think the data is appropriate to test this hypothesis?

💡 Solution

```
chisq.test(Popn0.9, correct = F)
```

Pearson's Chi-squared test

```
data: Popn0.9
X-squared = 0.093309, df = 1, p-value = 0.76
```

```
chisq.test(Popn0.9)
```

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

```
data: Popn0.9
X-squared = 0.088758, df = 1, p-value = 0.7658
```

In the past, for 2 X 2 tables Yates's correction was frequently employed (first test above, but it has since been shown to be overly conservative and is no longer recommended (although it doesn't affect the results in this particular instance). Better is a Fisher's exact test if the total number of cases is <200 (which is not the case here), or a randomization. Given that we cannot use a Fisher's exact test here we are using a Yate's correction.

These data are not particularly good for testing the null hypothesis that the sex ratio at birth is 1:1 because the first age category is too coarse. It is entirely possible that at birth there is an unequal sex-ratio, but there is compensatory age-specific mortality (e.g. more males at birth, but reduced survivorship among males in the first 9 years of life relative to females). In this case, the sex ratio at birth is NOT 1:1, but we still accept the null hypothesis based on age class 0-9.

16.6. Poisson regression to analyze multi-way tables

```
loglin <- read.csv("data/loglin.csv")

# Convert from frequency form to table form for mosaic plot
loglinTable <- xtabs(frequency ~ temperature + light + infected, data = loglin)

# Create mosaic plot to look at data
mosaic(loglinTable, shade = TRUE)
```

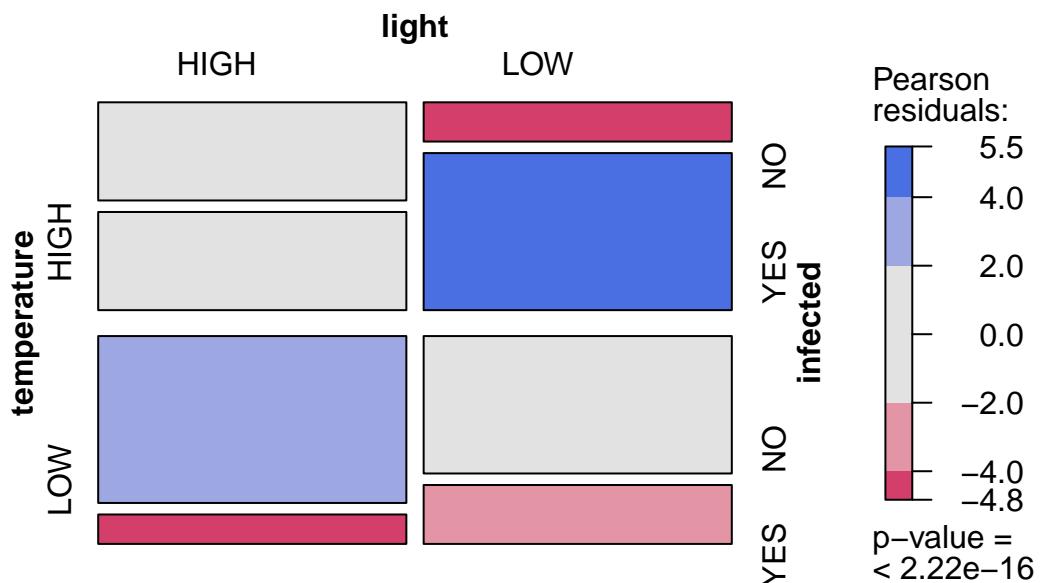


Figure 16.3.: Proportion de plantes infectées en fonction de la température et la lumière

The principle of testing for independence through interactions can be extended to multi-way tables, that is, tables in which more than two criteria are used to classify observations. For example, suppose that we wanted to test the effect of temperature (two levels: high and low) and light (two levels: high irradiance and low irradiance) on the number of plants infected by a plant pathogen (two levels: infected and non-infected). In this case we would need a three-way table with three criteria (infection status, temperature, and light).

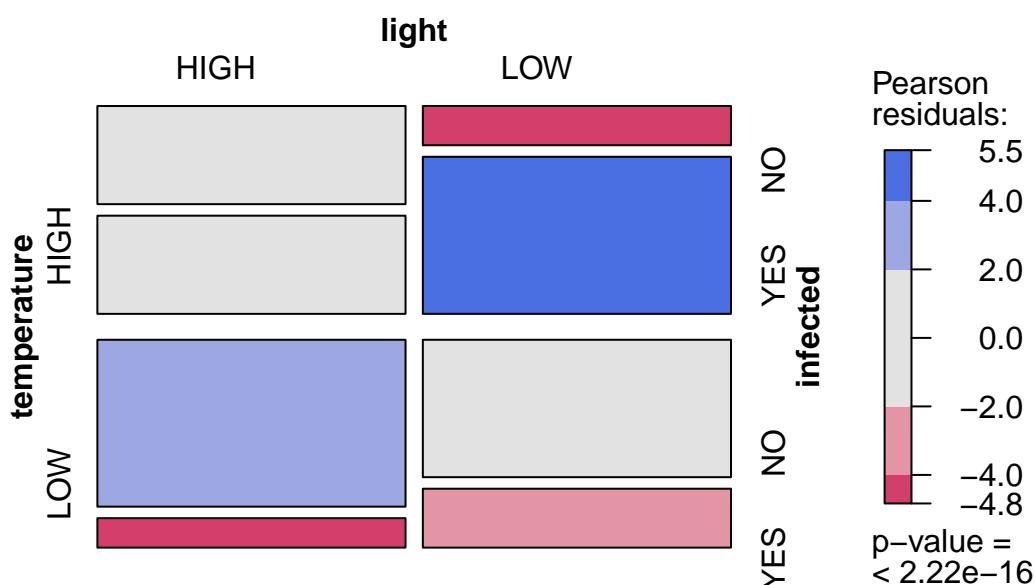
Fitting log linear models to frequency data involves testing of different models by comparing them with the full (saturated) model. A series of simplified models is produced, each model missing one of the interactions of interest, and the fit of each simplified model is compared to that of the full model. If the fit does not change much, then the term eliminated does not have much influence on the frequencies, whereas if the resulting model provides a significantly worse fit, then the term is important. As with two-way tables, the terms of interest are the interactions, not the main effects, if what we are testing for is independence of different factors.

The file `loglin.csv` contains the frequencies (`frequency`) of infected and non-infected plants (`infected`) at low and high temperature (`temperature`) and low and high light (`light`). To graph the data and determine if infected status depends on light and temperature, one can construct a mosaic plot and a loglinear model.

```
# Convert from frequency form to table form for mosaic plot
loglinTable <- xtabs(frequency ~ temperature + light + infected, data = loglin)

# Create mosaic plot to look at data
library(vcd)

mosaic(loglinTable, shade = TRUE)
```



The symmetrical experimental design with the same number of observations made at the two levels of light and of temperature is apparent in the above plot in the overall equal area occupied by the observations in each of the four quadrants. What is of interest, the infected status, appears to vary among the quadrants (i.e. levels of light and temperature). For example, the red rectangles in the lower left and upper right quadrants indicates that there were fewer infected plants at high light and low temperature (bottom left), and fewer uninfected plants at low light and high temperature than if the infected level was not affected by light and temperature. The p-value at the bottom of the color scale represents a test of independence equivalent to testing the full model against a reduced model including only the main effect of temperature, light, and infected status on the (ln) number of observations.

```
# Fit full model
full.model <- glm(frequency ~ temperature * light * infected, family = poisson(), data = loglin)
# Test partial effect of terms in full model
Anova(full.model, type = 3, test = "LR")
```

| | LR Chisq | Df | Pr(>Chisq) |
|----------------------------|-----------|----|------------|
| temperature | 9.178563 | 1 | 0.0024487 |
| light | 13.282863 | 1 | 0.0002678 |
| infected | 0.000000 | 1 | 0.9999999 |
| temperature:light | 5.675769 | 1 | 0.0172008 |
| temperature:infected | 29.061158 | 1 | 0.0000001 |
| light:infected | 20.268735 | 1 | 0.0000067 |
| temperature:light:infected | 1.083963 | 1 | 0.2978126 |

The probabilities associated with each term in the full model are here calculated by comparing the fit of the full model to that of a model with this particular term removed. As is typical in log-linear model analyses, many of the tests here are not interesting. If the biological question is about how infected status varies with other conditions, then the only informative terms are the interaction terms involving infected status.

There are therefore only 3 terms of interest:

1. `temperature:infected` significant interaction implies that infection status is not independent of temperature.
Indeed the mosaic plot shows that the proportion of infected cases is higher at high temperature.
2. `light:infected` significant interaction implies that infection status is not independent of light. The mosaic plot also indicates that the proportion of infected plants is larger at low light levels.

3. `temperature:light:infected` 3 way-interaction is not significant. This implies that the previous 2 effects do not vary between levels of the third variable. So there is no evidence that the effect of light on infection status varies at the two temperatures, or that the effect of temperature on infection status varies between the two light levels. We should therefore drop this term and refit before evaluating the 2-way interactions (small increase in power).

16.7. Exercice

We will now work with the `sturgdat` data set to test the hypothesis that number of fish caught is independent of location, year, and gender. Before the analysis, the data will have to be reshaped to be in suitable format for fitting a log-linear model.

Exercise

Open `sturgdat.csv`, then use the `table()` function to summarize the data according to number of individuals by `sex`, `location`, and `year`. Save this object as `sturgdat.table`. Make a mosaic plot of the data.

```
sturgdat <- read.csv("data/sturgdat.csv")
# Reorganize data from case form to table form
sturgdat.table <- with(sturgdat, table(sex, year, location))
# display the table
sturgdat.table
```

, , location = CUMBERLAND

| | year | | |
|--------|------|------|------|
| sex | 1978 | 1979 | 1980 |
| FEMALE | 10 | 30 | 11 |
| MALE | 14 | 14 | 6 |

, , location = THE_PAS

| | year | | |
|-----|------|------|------|
| sex | 1978 | 1979 | 1980 |

| | | | |
|--------|----|----|----|
| FEMALE | 5 | 12 | 38 |
| MALE | 16 | 12 | 18 |

```
# Create data frame while converting from table form to frequency form
sturgdat.freq <- as.data.frame(sturgdat.table)
# display data frame
sturgdat.freq
```

| sex | year | location | Freq |
|--------|------|------------|------|
| FEMALE | 1978 | CUMBERLAND | 10 |
| MALE | 1978 | CUMBERLAND | 14 |
| FEMALE | 1979 | CUMBERLAND | 30 |
| MALE | 1979 | CUMBERLAND | 14 |
| FEMALE | 1980 | CUMBERLAND | 11 |
| MALE | 1980 | CUMBERLAND | 6 |
| FEMALE | 1978 | THE_PAS | 5 |
| MALE | 1978 | THE_PAS | 16 |
| FEMALE | 1979 | THE_PAS | 12 |
| MALE | 1979 | THE_PAS | 12 |
| FEMALE | 1980 | THE_PAS | 38 |
| MALE | 1980 | THE_PAS | 18 |

Frequency of female and male sturgeon as a function of year and location

```
# Look at the data as mosaic plot
# mosaic using the table created above
mosaic(sturgdat.table, shade = TRUE)
```

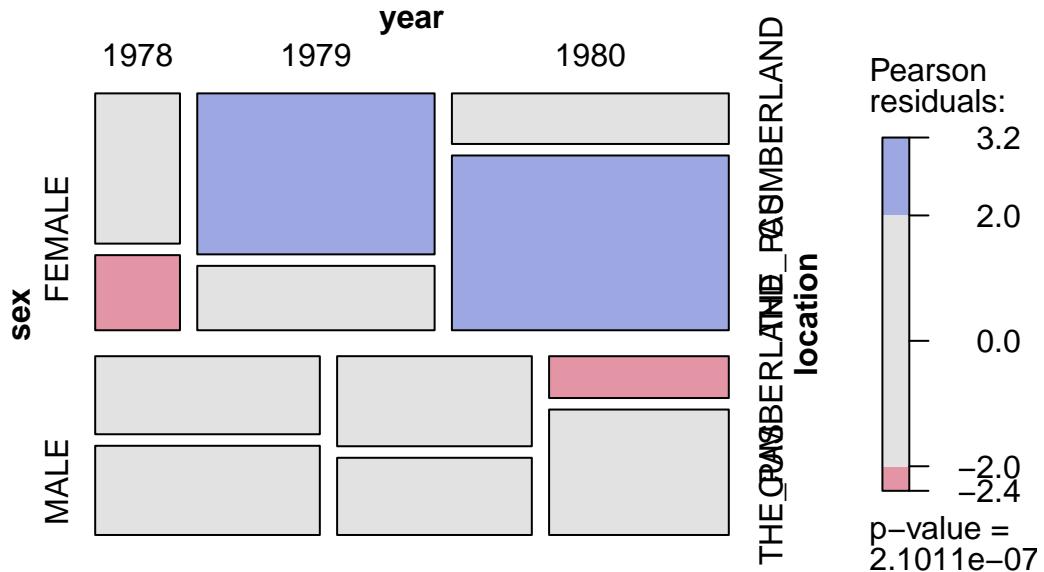


Figure 16.4.: Frequency of female and male sturgeon as a function of year and location

callout-caution # Exercise Using the frequency form of the table, fit the full log-linear model just as we did with the loglin data set and produce the anova table with chi square statistics for the terms in the model. Is the 3-way interaction significant (location:year:sex)? Does sex ratio change between locations or among years? ::

```
# Fit full model
full.model <- glm(Freq ~ sex * year * location, data = sturgdat.freq, family = "poisson")
summary(full.model)
```

Call:

```
glm(formula = Freq ~ sex * year * location, family = "poisson",
  data = sturgdat.freq)
```

Coefficients:

| | Estimate | Std. Error | z value |
|-------------|----------|------------|---------|
| (Intercept) | 2.30259 | 0.31623 | 7.281 |
| sexMALE | 0.33647 | 0.41404 | 0.813 |
| year1979 | 1.09861 | 0.36515 | 3.009 |
| year1980 | 0.09531 | 0.43693 | 0.218 |

| | | | | |
|--------------------------|--|-------------|---------|--------|
| locationTHE_PAS | | -0.69315 | 0.54772 | -1.266 |
| sexMALE | :year1979 | -1.09861 | 0.52554 | -2.090 |
| sexMALE | :year1980 | -0.94261 | 0.65498 | -1.439 |
| sexMALE | :locationTHE_PAS | 0.82668 | 0.65873 | 1.255 |
| year1979:locationTHE_PAS | | -0.22314 | 0.64550 | -0.346 |
| year1980:locationTHE_PAS | | 1.93284 | 0.64593 | 2.992 |
| sexMALE | :year1979:locationTHE_PAS | -0.06454 | 0.83986 | -0.077 |
| sexMALE | :year1980:locationTHE_PAS | -0.96776 | 0.87942 | -1.100 |
| | | Pr(> z) | | |
| (Intercept) | | 3.3e-13 *** | | |
| sexMALE | | 0.41641 | | |
| year1979 | | 0.00262 ** | | |
| year1980 | | 0.82732 | | |
| locationTHE_PAS | | 0.20569 | | |
| sexMALE | :year1979 | 0.03658 * | | |
| sexMALE | :year1980 | 0.15011 | | |
| sexMALE | :locationTHE_PAS | 0.20950 | | |
| year1979:locationTHE_PAS | | 0.72957 | | |
| year1980:locationTHE_PAS | | 0.00277 ** | | |
| sexMALE | :year1979:locationTHE_PAS | 0.93875 | | |
| sexMALE | :year1980:locationTHE_PAS | 0.27114 | | |
| --- | | | | |
| Signif. codes: | 0 *** 0.001 ** 0.01 * 0.05 . 0.1 ' ' 1 | | | |

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 5.7176e+01 on 11 degrees of freedom
Residual deviance: -2.6645e-15 on 0 degrees of freedom
AIC: 77.28

Number of Fisher Scoring iterations: 3

```
Anova(full.model, type = 3)
```

| | LR Chisq | Df | Pr(>Chisq) |
|-------------------|------------|----|------------|
| sex | 0.6697879 | 1 | 0.4131256 |
| year | 13.8894797 | 2 | 0.0009637 |
| location | 1.6989904 | 1 | 0.1924201 |
| sex:year | 4.6930229 | 2 | 0.0957024 |
| sex:location | 1.6322742 | 1 | 0.2013888 |
| year:location | 25.2580075 | 2 | 0.0000033 |
| sex:year:location | 1.6677328 | 2 | 0.4343666 |

This is a three-way table, with three factors: sex, location and year . Thus, the “staturated” or “full” loglinear model includes 7 terms: the three main effects (sex, location and year), the three 2-way interactions (sex:year, sex:location and year: location) and the one 3-way interaction (sex:year:location). The null deviance is 57.17574, the residual deviance of the full model is, not surprisingly, 0. The deviance explained by the three-way interaction, 1.66773 (which is, in fact, a chi square statistic with two degrees of freedom), is not significant, and we are therefore justified in fitting the model without this term.

What does this mean? It means that, if there are any 2-way interactions, they do not depend on the level of the third variable. For example, if indeed the sex ratio of sturgeon varies among years (a sex:year interaction), that it varies in the same manner at the two locations. This in turn means that in testing for two-way interactions, we are (statistically) justified in pooling (summing) over the levels of the third variable. This is analog to what can be done in multiway ANOVA when high order interactions are not significant. For example, in testing for a sex:location effect, we can pool over year , to produce a 2 X 2 table whose cell counts are the total number of sturgeon of a given sex at a given location captured over the three years 1978-1980. By increasing cell counts, we increase statistical power, which is desirable.

- If we adjust the model without the 3-way interaction, we get:

Solution

```
o2int.model <- glm(Freq ~ sex + year + location + sex:year + sex:location + year:location, data = stur)
Anova(o2int.model, type = 3)
```

| | LR Chisq | Df | Pr(>Chisq) |
|---------------|-----------|----|------------|
| sex | 1.869079 | 1 | 0.1715807 |
| year | 15.128861 | 2 | 0.0005186 |
| location | 1.544449 | 1 | 0.2139568 |
| sex:year | 15.584729 | 2 | 0.0004129 |
| sex:location | 2.176220 | 1 | 0.1401583 |
| year:location | 28.349871 | 2 | 0.0000007 |

We can see that the sex:location interaction does not explain a significant portion of the deviance, whereas the two others do. Sex ratio does not vary among locations, but it does among years. The year:location is also significant (see below for its meaning).

Should you try to simplify the model further? Real statisticians are divided on this question. All agree that keeping insignificant terms in the model may cost some power. On the other hand, removing non significant interactions can lead to difficulty interpreting answers when observations are not well balanced (i.e. there is collinearity among model terms).

- Refit the model, this time excluding the sex:location interaction.

Solution

```
o2int.model2 <- glm(Freq ~ sex + year + location + sex:year + year:location, data = sturgdat.fr)
Anova(o2int.model2, type = 3)
```

| | LR Chisq | Df | Pr(>Chisq) |
|---------------|------------|----|------------|
| sex | 5.0969711 | 1 | 0.0239677 |
| year | 16.1226325 | 2 | 0.0003155 |
| location | 0.2001484 | 1 | 0.6546011 |
| sex:year | 13.9882526 | 2 | 0.0009173 |
| year:location | 26.7533952 | 2 | 0.0000016 |

Now the remaining two interactions are significant. It looks as though this is the “best” model. On the basis of the above analysis, the simplest model is:

$$\ln[f_{(ijk)}] = \text{location} + \text{sex} + \text{year} + \text{sex : year} + \text{location : year}$$

How are these effects interpreted biologically? Remember, as in tests of independence, we are not interested in main effects, only the interactions. For example, the main effect location tells us that the total number of sturgeon caught (pooled over both sexes and all years 1978-1980) varied between the two locations. This is not surprising and uninteresting given that we have no information on the sampling effort. However, the sex:year interaction tells us that over the 3 year period, the sex-ratio of the harvest changed, and it changed in more or less the same fashion in the two locations, which is a rather interesting result. The location:year effect tells us that the total number of sturgeon harvested not only changed over the years, but that this change varied between locations. This could be caused by a different fishing effort at one station over the years, or to a negative impact at one station only on one year. Whatever the cause, it affected males and females similarly since the 3 way interaction is not significant.

Part V.

Mixed models

Chapter 17

Introduction to linear mixed models

17.1. Lecture

17.1.1. Testing fixed effects

making a note that LRT on fixed effects should not be the preferred method and more importantly should eb done using ML and not REML Fitsee pinheiro & Bates 2000 p76

17.1.2. Shrinkage

The following is an example of **shrinkage**, sometimes called **partial-pooling**, as it occurs in **mixed effects models**.

It is often the case that we have data such that observations are clustered in some way (e.g. repeated observations for units over time, students within schools, etc.). In mixed models, we obtain cluster-specific effects in addition to those for standard coefficients of our regression model. The former are called **random effects**, while the latter are typically referred to as **fixed effects or population-average** effects.

In other circumstances, we could ignore the clustering, and run a basic regression model. Unfortunately this assumes that all observations behave in the same way, i.e. that there are no cluster-specific effects, which would often be an untenable assumption. Another approach would be to run separate models for each cluster. However, aside from being problematic due to potentially small cluster sizes in common data settings, this ignores the fact that clusters are not isolated and potentially have some commonality.

Mixed models provide an alternative where we have cluster specific effects, but ‘borrow strength’ from the population-average effects. In general, this borrowing is more apparent for what would otherwise be more extreme clusters, and those that have less data. The following will demonstrate how shrinkage arises in different data situations.

17.1.2.1. Analysis

For the following we run a basic mixed model with a random intercept and random slopes for a single predictor variable. There are a number of ways to write such models, and the following does so for a single cluster c and observation i . y is a function of the covariate x , and otherwise we have a basic linear regression model. In this formulation, the random effects for a given cluster (u_{*c}) are added to each fixed effect (intercept b_0 and the effect of x , b_1). The random effects are multivariate normally distributed with some covariance. The per observation noise σ is assumed constant across observations.

$$\mu_{ic} = (b_0 + u_{0c}) + (b_1 + u_{1c}) * x_{ic}$$

$$u_0, u_1 \sim \mathcal{N}(0, \Sigma)$$

$$y \sim \mathcal{N}(\mu, \sigma^2)$$

Such models are highly flexible and have many extensions, but this simple model is enough for our purposes.

17.1.2.2. Data

Default settings for data creation are as follows:

- `obs_per_cluster` (observations per cluster) = 10
- `n_cluster` (number of clusters) = 100
- `intercept` (intercept) = 1
- `beta` (coefficient for x) = .5
- `sigma` (observation level standard deviation) = 1
- `sd_int` (standard deviation for intercept random effect)= .5
- `sd_slope` (standard deviation for x random effect)= .25
- `cor` (correlation of random effect) = 0
- `balanced` (fraction of overall sample size) = 1
- `seed` (for reproducibility) = 1024

In this setting, x is a standardized variable with mean zero and standard deviation of 1. Unless a fraction is provided for `balanced`, the N , i.e. the total sample size, is equal to `n_cluster * obs_per_cluster`. The following is the

function that will be used to create the data, which tries to follow the model depiction above. It requires the tidyverse package to work.

17.1.2.3. Run the baseline model

We will use **lme4** to run the analysis. We can see that the model recovers the parameters fairly well, even with the default of only 1000 observations.

```
df <- create_data()
```

```
library(lme4)
```

Loading required package: Matrix

Attaching package: 'Matrix'

The following objects are masked from 'package:tidyঃ':

expand, pack, unpack

```
mod <- lmer(y ~ x + (x | cluster), df)
summary(mod, cor = F)
```

Linear mixed model fit by REML ['lmerMod']

Formula: y ~ x + (x | cluster)

Data: df

REML criterion at convergence: 3012.2

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|---------|---------|---------|--------|--------|
| -2.9392 | -0.6352 | -0.0061 | 0.6156 | 2.8721 |

Random effects:

| Groups | Name | Variance | Std.Dev. | Corr |
|---------|-------------|----------|----------|------|
| cluster | (Intercept) | 0.29138 | 0.5398 | |
| | x | 0.05986 | 0.2447 | 0.30 |
| | Residual | 0.99244 | 0.9962 | |

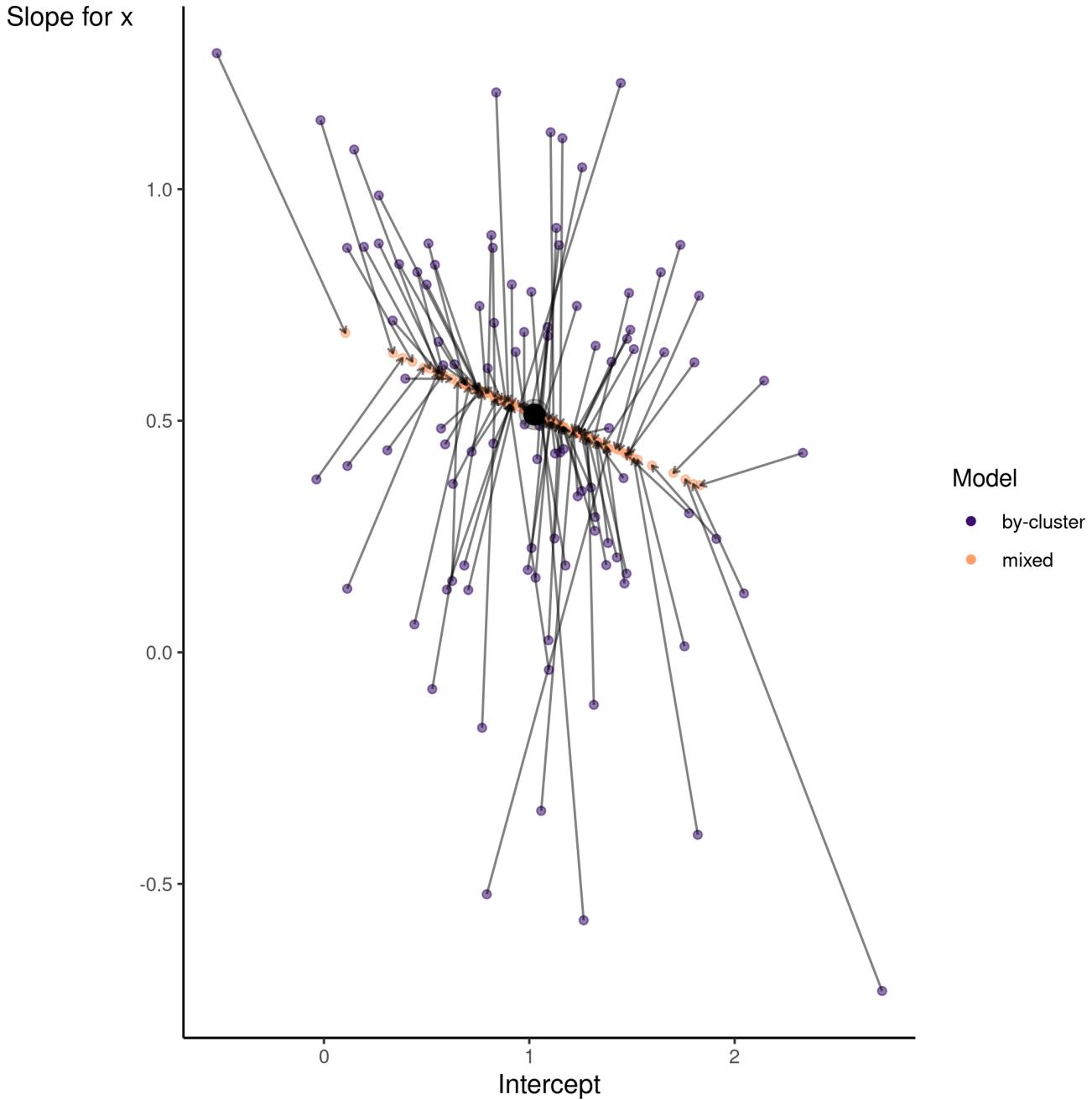
Number of obs: 1000, groups: cluster, 100

Fixed effects:

| | Estimate | Std. Error | t value |
|-------------|----------|------------|---------|
| (Intercept) | 0.93647 | 0.06282 | 14.91 |
| x | 0.54405 | 0.04270 | 12.74 |

17.1.2.4. Visualize the baseline model

Now it is time to visualize the results. We will use **gganimate** to bring the shrinkage into focus. We start with the estimates that would be obtained by a ‘regression-by-cluster’ approach or a linear regression for each cluster. The movement shown will be of those cluster-specific estimates toward the mixed model estimates. On the x axis is the estimate for the intercepts, on the y axis are the estimated slopes of the x covariate.



We see more clearly what the mixed model does. The general result is that cluster-specific effects (lighter color) are shrunk back toward the population-average effects (the ‘black hole’), as the imposed normal distribution for the random effects makes the extreme values less probable. Likewise, those more extreme cluster-specific effects, some of which are not displayed as they are so far from the population average, will generally have the most shrinkage imposed. In terms of prediction, it is akin to introducing bias for the cluster specific effects while lowering variance for prediction of new data, and allows us to make predictions on new categories we have not previously seen - we just assume an ‘average’ cluster effect, i.e. a random effect of 0.

17.1.2.5. Summary

Mixed models incorporate some amount of shrinkage for cluster-specific effects. Data nuances will determine the relative amount of ‘strength borrowed’, but in general, such models provide a good way for the data to speak for itself when it should, and reflect an ‘average’ when there is little information. An additional benefit is that thinking about models in this way can be seen as a precursor to Bayesian approaches, which can allow for even more flexibility via priors, and more control over how shrinkage is added to the model.

17.2. Practical

17.2.1. Overview

This practical is intended to get you started fitting some simple mixed models with so called *random intercepts*. The tutorial is derived from one that accompanied the paper (Houslay and Wilson 2017), “[Avoiding the misuse of BLUP in behavioral ecology](#)”. Here, you will be working through a simplified version in which I have taken more time to cover the basic mixed models and don’t cover multivariate models which were really the main point of that paper. So if you find this material interesting don’t worry we will go through a more advanced version of the original paper on multivariate models in chapter XX. The original version will be worth a work through to help you break into multivariate mixed models anyway! Here we will:

- Learn how to fit - and interpret the results of - a simple univariate mixed effect model
- See how to add fixed and random effects to your model, and to test their significance in the normal frequentists sense

We are going to use the  `lme4` (Bates et al. 2015) which is widely used and great for simple mixed models. However, since, for philosophical reasons, `lme4` does not provide any p-values for either fixed or random effects, we are going to use the  `lmerTest` (Kuznetsova et al. 2017), which add a bunch a nice goodies to `lme4` For slightly more complex models, including multivariate ones, generalised models, and random effects of things like shared space, pedigree, phylogeny I tend to use different  like `MCMCglmm` (Hadfield 2010) (which is Bayesian, look at Jarrod Hadfield’s excellent course notes (Hadfield 2010)) or `ASReml-R` (The VSNi Team 2023) (which is likelihood based/frequentist but sadly is not free).

17.2.2. R packages needed

First we load required libraries

```
library(lmerTest)
library(performance)
library(tidyverse)
library(rptR)
```

17.2.3. The superb wild unicorns of the Scottish Highlands

Unicorns, a legendary animal and also symbol of Scotland, are frequently described as extremely wild woodland creature but also a symbol of purity and grace. Here is one of most accurate representation of the legendary animal.



Figure 17.1.: The superb unicorn of the Scottish Highlands

Despite their image of purity and grace, unicorns (*Unicornus legendaricus*) are raging fighters when it comes to compete for the best sweets you can find at the bottom of rainbows (unicorn favourite source of food).

We want to know:

- If aggressiveness differs among individuals
- If aggressive behaviour is plastic (change with the environment)
- If aggressive behaviour depends on body condition of focal animal

With respect to plasticity, we will focus on rival size as an ‘environment’. Common sense, and animal-contest theory, suggest a small animal would be wise not to escalate an aggressive contest against a larger, stronger rival. However, there are reports in the legendary beastly literature that they get more aggressive as rival size increases. Those reports are based on small sample sizes and uncontrolled field observations by foreigners Munro baggers enjoying their whisky after a long day in the hills.

17.2.3.1. Experimental design

Here, we have measured aggression in a population of wild unicorns. We brought some ($n=80$) individual into the lab, tagged them so they were individually identifiable, then repeatedly observed their aggression when presented with model ‘intruders’ (animal care committee approved). There were three models; one of average unicorn (calculated as the population mean body length), one that was build to be 1 standard deviation below the population mean, and one that was 1 standard deviation above.

Data were collected on all individuals in two block of lab work. Within each block, each animal was tested 3 times, once against an ‘intruder’ of each size. The test order in which each animal experienced the three intruder sizes was randomised in each block. The body size of all focal individuals was measured at the beginning of each block so we know that too (and have two separate measures per individual).

17.2.3.2. looking at the data

Let’s load the data file `unicorns_aggression.csv` in a R object named `unicorns` and make sure we understand what it contains

Solution

```
unicorns <- read.csv("data/unicorns_aggression.csv")
```

You can use `summary(unicorns)` to get an overview of the data and/or `str(unicorns)` to see the structure in the first few lines. This data frame has 6 variables:

```
str(unicorns)
```

```
'data.frame': 480 obs. of 6 variables:  
 $ ID       : chr  "ID_1" "ID_1" "ID_1" "ID_1" ...  
 $ block    : num  -0.5 -0.5 -0.5 0.5 0.5 0.5 -0.5 -0.5 -0.5 0.5 ...  
 $ assay_rep: int  1 2 3 1 2 3 1 2 3 1 ...
```

```
$ opp_size : int -1 1 0 0 1 -1 1 -1 0 1 ...
$ aggression: num 7.02 10.67 10.22 8.95 10.51 ...
$ body_size : num 206 206 206 207 207 ...
```

```
summary(unicorns)
```

| | ID | block | assay_rep | opp_size | aggression |
|------------------|---------------|--------------|-----------|------------|----------------|
| Length:480 | | Min. :-0.5 | Min. :1 | Min. :-1 | Min. : 5.900 |
| Class :character | | 1st Qu.:-0.5 | 1st Qu.:1 | 1st Qu.:-1 | 1st Qu.: 8.158 |
| Mode :character | | Median : 0.0 | Median :2 | Median : 0 | Median : 8.950 |
| | | Mean : 0.0 | Mean :2 | Mean : 0 | Mean : 9.002 |
| | | 3rd Qu.: 0.5 | 3rd Qu.:3 | 3rd Qu.: 1 | 3rd Qu.: 9.822 |
| | | Max. : 0.5 | Max. :3 | Max. : 1 | Max. :12.170 |
| | body_size | | | | |
| | Min. :192.0 | | | | |
| | 1st Qu.:229.7 | | | | |
| | Median :250.0 | | | | |
| | Mean :252.5 | | | | |
| | 3rd Qu.:272.0 | | | | |
| | Max. :345.2 | | | | |

So the different columns in the data set are:

- Individual **ID**
- Experimental **Block**, denoted for now as a continuous variable with possible values of -0.5 (first block) or +0.5 (second block)
- Individual **body_size**, as measured at the start of each block in kg
- The repeat number for each behavioural test, **assay_rep**
- Opponent size (**opp_size**), in standard deviations from the mean (i.e., -1,0,1)
- **aggression**, our behavioural trait, measured 6 times in total per individual (2 blocks of 3 tests)

maybe add something on how to look at data structure closely using tables

17.2.4. Do unicorns differ in aggressiveness? Your first mixed model

Fit a first mixed model with `lmer` that have only individual identity as a random effect and only a population mean.

Why, so simple? Because we simply want to partition variance around the mean into a component that among-individual variance and one that is within-individual variance.

! Important

We are going to use the function `lmer()` from the  `lme4` package. The notation of the model formula is similar as the notation for a linear model but now we also add random effects using the notation `(1 | r_effect)` which indicates that we want to fit the variable `r_effect` as a random effect for the intercept. Thus, in `lmer` notation a simple model would be :

```
lmer(Y ~ x1 + x2 + (1 | r_effect), data = data)
```

💡 Solution

A sensible researcher would probably take the time to do some exploratory data plots here. So let's write a mixed model. This one is going to have no fixed effects except the mean, and just one random effect - individual identity.

```
m_1 <- lmer(aggression ~ 1 + (1 | ID), data = unicorns)
```

```
boundary (singular) fit: see help('isSingular')
```

There is a warning... something about “singularities”. Ignore that for a moment.

Now you need to get the model output. By that I just mean use `summary(model_name)`.

💡 Solution

```
summary(m_1)
```

```
Linear mixed model fit by REML. t-tests use Satterthwaite's method [  
lmerModLmerTest]  
Formula: aggression ~ 1 + (1 | ID)  
Data: unicorns
```

```
REML criterion at convergence: 1503.7
```

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|----------|---------|---------|
| -2.68530 | -0.73094 | -0.04486 | 0.71048 | 2.74276 |

Random effects:

| Groups | Name | Variance | Std.Dev. |
|----------|-------------|----------|----------|
| ID | (Intercept) | 0.000 | 0.000 |
| Residual | | 1.334 | 1.155 |

Number of obs: 480, groups: ID, 80

Fixed effects:

| | Estimate | Std. Error | df | t value | Pr(> t) | | | | | | |
|---|----------|------------|-----------|---------|------------|-----|------|------|-----|-----|---|
| (Intercept) | 9.00181 | 0.05272 | 479.00000 | 170.7 | <2e-16 *** | | | | | | |
| --- | | | | | | | | | | | |
| Signif. codes: | 0 | '***' | 0.001 | '**' | 0.01 | '*' | 0.05 | '. ' | 0.1 | ' ' | 1 |
| optimizer (nloptwrap) convergence code: | 0 | (OK) | | | | | | | | | |
| boundary (singular) fit: see help('isSingular') | | | | | | | | | | | |

In the summary you will find a table of fixed effects.

Fixed effects:

| | Estimate | Std. Error | df | t value | Pr(> t) |
|-------------|----------|------------|-----------|---------|------------|
| (Intercept) | 9.00181 | 0.05272 | 479.00000 | 170.7 | <2e-16 *** |

The intercept (here the mean) is about 9 and is significantly >0 - fine, but not very interesting to us.

You will also find a random effect table that contains estimates of the among individual (ID) and residual variances.

Random effects:

| Groups | Name | Variance | Std.Dev. |
|----------|-------------|----------|----------|
| ID | (Intercept) | 0.000 | 0.000 |
| Residual | | 1.334 | 1.155 |

Number of obs: 480, groups: ID, 80

The among individual (ID) is estimated as zero. In fact this is what the cryptic warning was about: in most situations the idea of a random effect explaining less than zero variance is not sensible (strangely there are exceptions!). So by default the variance estimates are constrained to lie in positive parameter space. Here in trying to find the maximum likelihood solution for among-individual variance, our model has run up against this constraint.

17.2.4.1. Testing for random effects

We can test the statistical significance of the random effect using the `ranova()` command in `lmerTest`. This function is actually doing a *likelihood ratio test* (LRT) of the random effect. The premise of which is that twice the difference in log-likelihood of the full and reduced (i.e. with the random effect dropped) is itself distributed as χ^2 with DF equal to the number of parameters dropped (here 1). Actually, there is a good argument that this is too conservative, but we can discuss that later. So let's do the LRT for the random effect using `ranova()`

Solution

```
ranova(m_1)
```

| | npar | logLik | AIC | LRT | Df | Pr(>Chisq) |
|----------|------|-----------|----------|-----|----|------------|
| | 3 | -751.8278 | 1509.656 | NA | NA | NA |
| (1 ID) | 2 | -751.8278 | 1507.656 | 0 | 1 | 1 |

There is apparently no among-individual variance in aggressiveness.

So this is a fairly rubbish and underwhelming model. Let's improve it.

17.2.5. Do unicorns differ in aggressiveness? A better mixed model

The answer we got from our first model is **not wrong**, it estimated the parameters we asked for and that might be informative or not and that might be representative or not of the true biology. Anyway all models are **wrong** but as models go this one is fairly rubbish. In fact we have explained no variation at all as we have no fixed effects (except the mean) and our random effect variance is zero. We would have seen just how pointless this model was if we'd plotted it

```
plot(m_1)
```

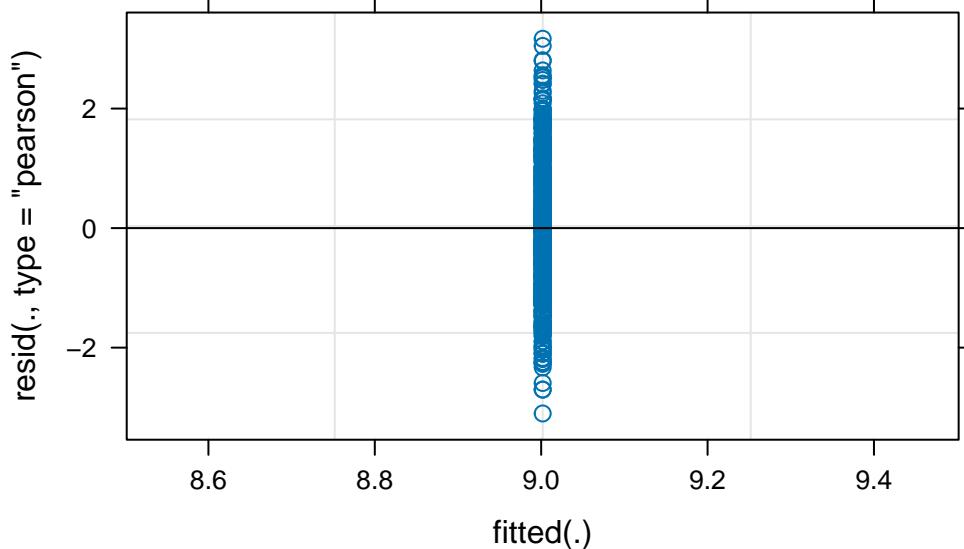


Figure 17.2.: Fitted values vs residuals for a simple mixed model of unicorn aggression

So we can probably do better at modelling the data, which may or may not change our view on whether there is any real variation among unicorns in aggressiveness.

For instance, we can (and should have started with) an initial plot of the phenotypic data against opponent size indicates to have a look at our prediction.

Solution

The code below uses the excellent ggplot2 but the same figure can be done using base R code.

```
ggplot(unicorns, aes(x = opp_size, y = aggression)) +
  geom_jitter(
    alpha = 0.5,
    width = 0.05
  ) +
  scale_x_continuous(breaks = c(-1, 0, 1)) +
  labs(
    x = "Opponent size (SD)",
    y = "Aggression"
  ) +
  theme_classic()
```

```
ggplot(unicorns, aes(x = opp_size, y = aggression)) +
  geom_jitter(
    alpha = 0.5,
    width = 0.05
  ) +
  scale_x_continuous(breaks = c(-1, 0, 1)) +
  labs(
    x = "Opponent size (SD)",
    y = "Aggression"
  ) +
  theme_classic()
```

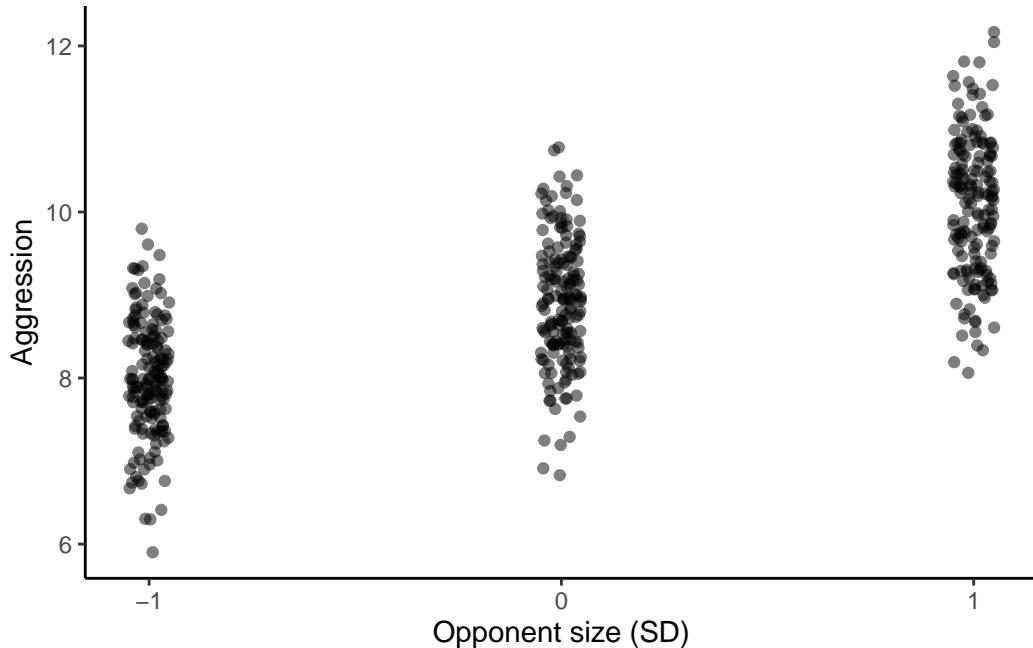


Figure 17.3.: Unicorn aggressivity as a function of opponent size when fighting for sweets

As predicted, there is a general increase in aggression with opponent size (points are lightly jittered on the x-axis to show the spread of data a little better)

You can see the same thing from a quick look at the population means for aggression at opponent size. Here we do it with the `kable` function that makes nice tables in `rmarkdown` documents.

```
unicorns %>%
  group_by(opp_size) %>%
  summarise(mean_aggr = mean(aggression)) %>%
  knitr::kable(digits = 2)
```

| opp_size | mean_aggr |
|----------|-----------|
| -1 | 8.00 |
| 0 | 8.91 |
| 1 | 10.09 |

So, there does appear to be plasticity of aggression with changing size of the model opponent. But other things may explain variation in aggressiveness too - what about block for instance? Block effects may not be the subject of any biologically interesting hypotheses, but accounting for any differences between blocks could remove noise.

There may also be systematic change in behaviour as an individual experiences more repeat observations (i.e. exposure to the model). Do they get sensitised or habituated to the model intruder for example?

So let's run a mixed model with the same random effect of individual, but with a fixed effects of opponent size (our predictor of interest) and experimental block.

💡 Solution

```
m_2 <- lmer(aggression ~ opp_size + block + (1 | ID), data = unicorns)
```

17.2.5.1. Diagnostic plots

Run a few diagnostic plots before we look at the answers. In diagnostic plots, we want to check the condition of applications of the linear mixed model which are the same 4 as the linear model plus 2 extra:

1. Linearity of the relation between covariates and the response

💡 Solution

Done with data exploration graph (i.e. just plot the data see if it is linear) - see previous graph Figure 17.3.

2. No error on measurement of covariates

 Solution

assumed to be correct if measurement error is lower than 10% of variance in the variable - I know this sounds pretty bad

3. Residual have a Gaussian distribution

 Solution

using quantile-quantile plot or histogram of residuals

```
par(mfrow = c(1, 2)) # multiple graphs in a window
qqnorm(residuals(m_2)) # a q-q plot
qqline(residuals(m_2))
hist(resid(m_2)) # are the residuals roughly Gaussian?
```

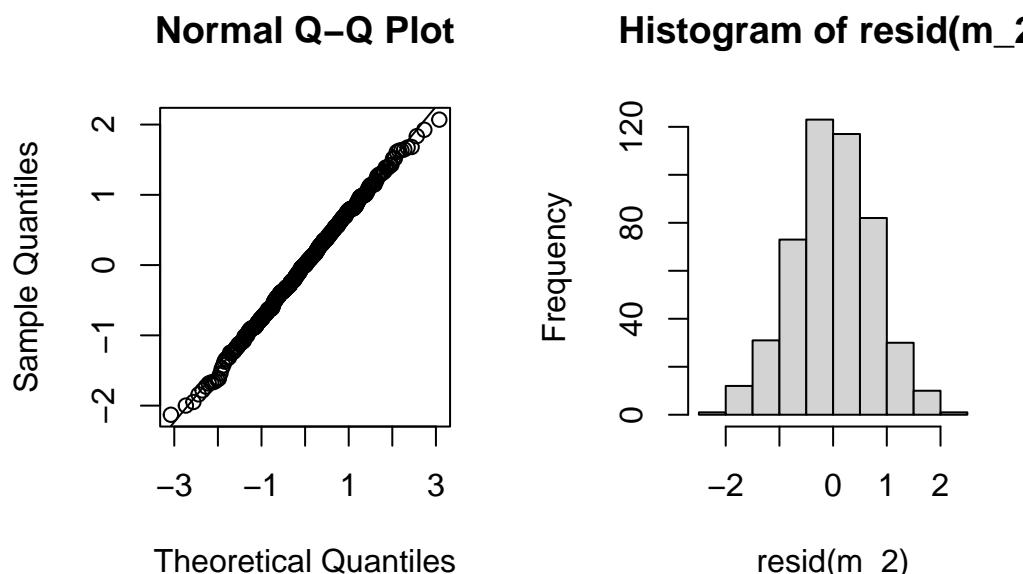


Figure 17.4.: Checking residuals have Gaussian distribution

4. Homoscedasticity (variance of residuals is constant across covariates)

 Solution

Using plot of residuals by fitted values

```
plot(m_2)
```

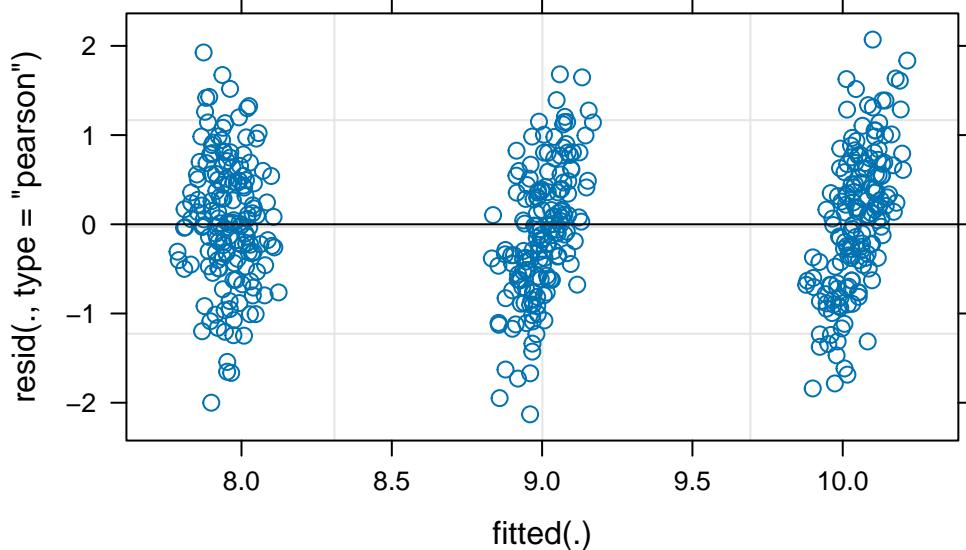


Figure 17.5.: Residuals by fitted values for model m_2 to check homoscedasticity

5. Random effects have a Gaussian distribution

Solution

histogram of the predictions for the random effects (BLUPs)

```
# extracting blups
r1 <- as.data.frame(ranef(m_2, condVar = TRUE))
par(mfrow = c(1, 2))
hist(r1$condval)
qqnorm(r1$condval)
qqline(r1$condval)
```

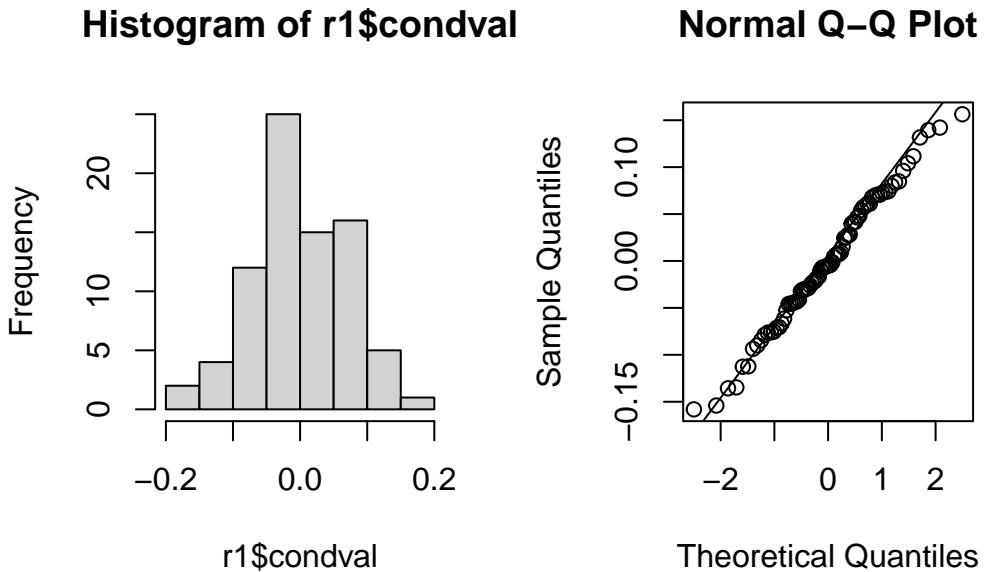


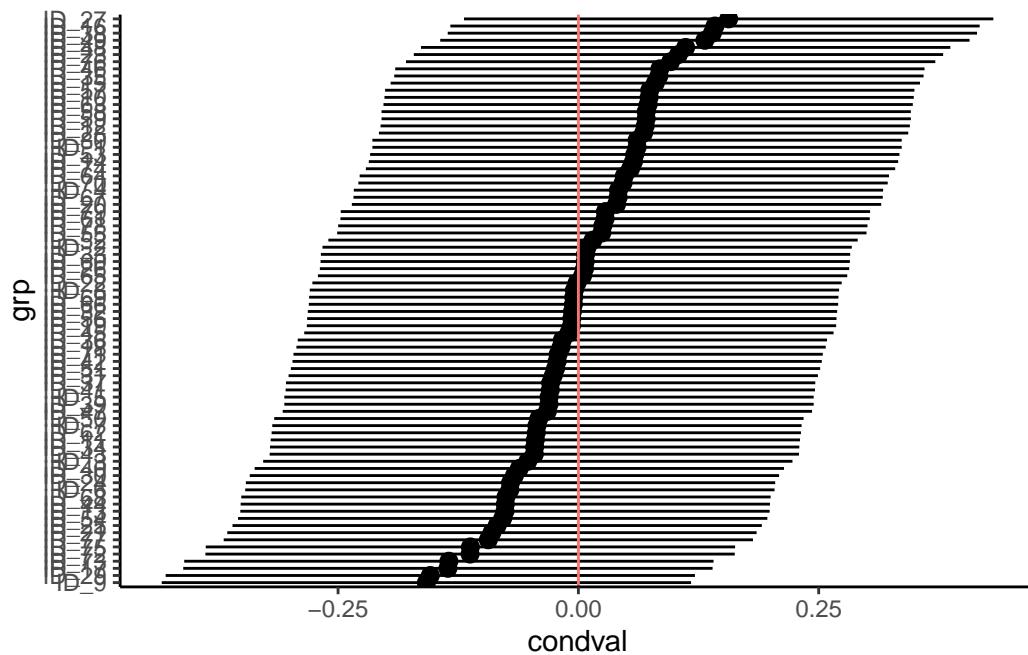
Figure 17.6.: Checking random effects are gaussian

6. Residual variance is constant across all levels of a random effect

Solution

No straightforward solution to deal with that. We can just do a plot is absolutely not-informative for that problem but I always like to look at. It is the plot of the sorted BLUPs with their associated errors.

```
r1 <- r1[order(r1$condval), ] # sorting the BLUPs
ggplot(r1, aes(y = grp, x = condval)) +
  geom_point() +
  geom_pointrange(
    aes(xmin = condval - condstd * 1.96, xmax = condval + condstd * 1.96)
  ) +
  geom_vline(aes(xintercept = 0, color = "red")) +
  theme_classic() +
  theme(legend.position = "none")
```



Here is a great magic trick 💥 because 3-5 and more can be done in one step

💡 Solution

You need to use the function `check_model()` from the 📦 `performance` package.

```
check_model(m_2)
```

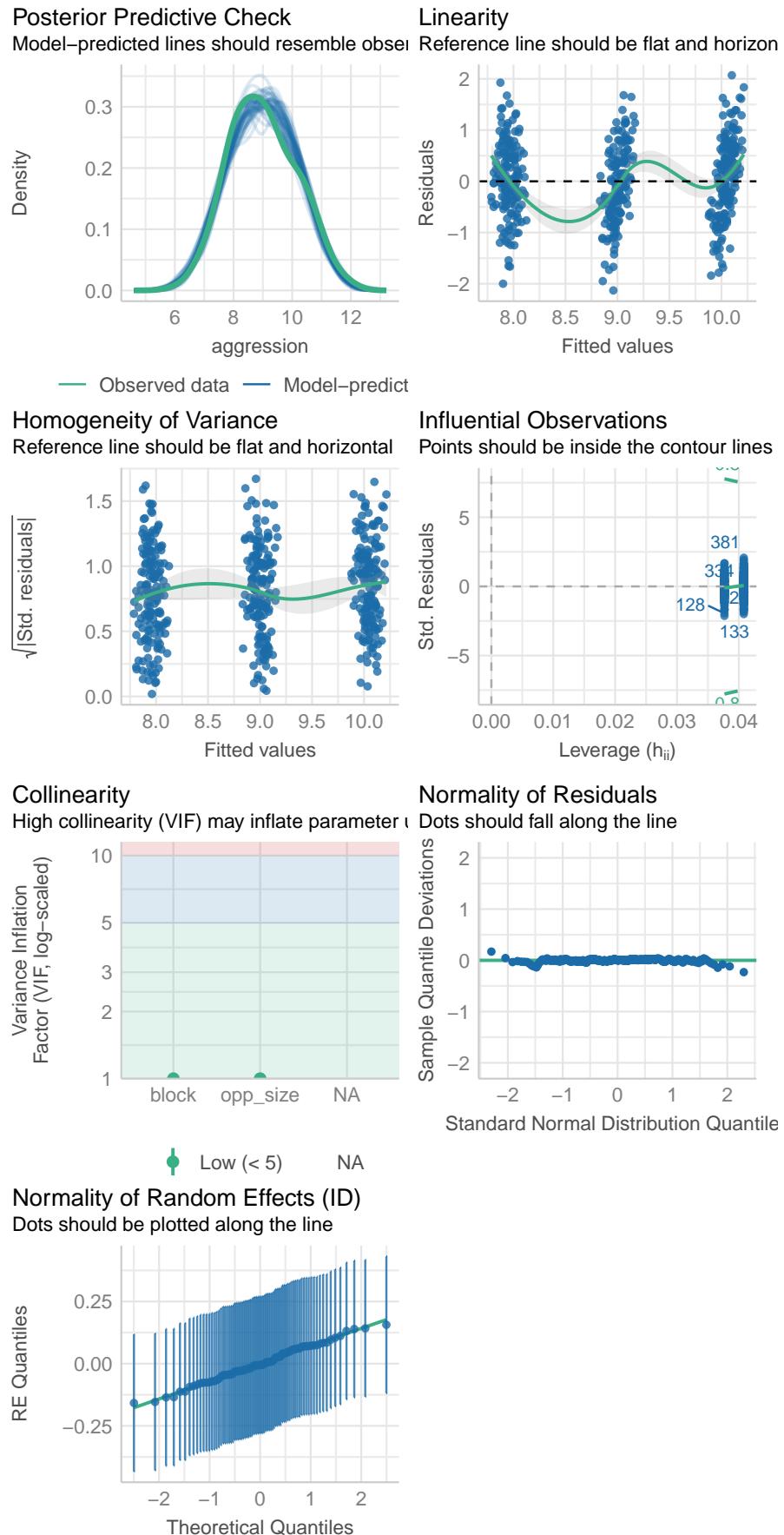


Figure 17.7.: Graphical check of model assumptions

17.2.5.2. Inferences

Now summarise this model. We will pause here for you to think about and discuss a few things: * What can you take from the fixed effect table? * How do you interpret the intercept now that there are other effects in the model? * What would happen if we scaled our fixed covariates differently? Why?

Solution

```
summary(m_2)
```

```
Linear mixed model fit by REML. t-tests use Satterthwaite's method [  
lmerModLmerTest]  
Formula: aggression ~ opp_size + block + (1 | ID)  
Data: unicorns
```

REML criterion at convergence: 1129.9

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -2.79296 | -0.64761 | 0.00155 | 0.67586 | 2.71456 |

Random effects:

| Groups | Name | Variance | Std.Dev. |
|--------|-------------|----------|----------|
| ID | (Intercept) | 0.02478 | 0.1574 |
| | Residual | 0.58166 | 0.7627 |

Number of obs: 480, groups: ID, 80

Fixed effects:

| | Estimate | Std. Error | df | t value | Pr(> t) |
|----------------|----------|------------|-----------|----------|------------|
| (Intercept) | 9.00181 | 0.03901 | 79.00000 | 230.778 | <2e-16 *** |
| opp_size | 1.04562 | 0.04263 | 398.00000 | 24.525 | <2e-16 *** |
| block | -0.02179 | 0.06962 | 398.00000 | -0.313 | 0.754 |
| --- | | | | | |
| Signif. codes: | 0 '***' | 0.001 '**' | 0.01 '*' | 0.05 '.' | 0.1 ' ' |

Correlation of Fixed Effects:

```
(Intr) opp_sz
opp_size 0.000
block    0.000 0.000
```

🔥 Exercise

Try tweaking the fixed part of your model:

- What happens if you add more fixed effects? Try it!
- Could focal body size also matter? If so, should you rescale before adding it to the model?
- Should you add interactions (e.g. block:opp_size)?
- Should you drop non-significant fixed effects?

🔥 Exercise

Having changed the fixed part of your model, do the variance estimates change at all?

- Is among-individual variance always estimated as zero regardless of fixed effects?
- Is among-individual variance significant with some fixed effects structures but not others?

17.2.6. What is the repeatability?

As a reminder, repeatability is the proportion of variance explained by a random effect and it is estimate as the ratio of the variance associated to a random effect by the total variance, or the sum of the residual variance and the different variance components associated with the random effects. In our first model among-individual variance was zero, so R was zero. If we have a different model of aggression and get a non-zero value of the random effect variance, we can obviously calculate a repeatability estimate (R). So we are all working from the same starting point, let's all stick with a common set of fixed effects from here on:

```
m_3 <- lmer(
  aggression ~ opp_size + scale(body_size, center = TRUE, scale = TRUE)
  + scale(assay_rep, scale = FALSE) + block
  + (1 | ID),
  data = unicorns
```

```
)
summary(m_3)
```

Linear mixed model fit by REML. t-tests use Satterthwaite's method [
lmerModLmerTest]

Formula:

```
aggression ~ opp_size + scale(body_size, center = TRUE, scale = TRUE) +
scale(assay_rep, scale = FALSE) + block + (1 | ID)
```

Data: unicorns

REML criterion at convergence: 1136.5

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -2.85473 | -0.62831 | 0.02545 | 0.68998 | 2.74064 |

Random effects:

| Groups | Name | Variance | Std.Dev. |
|--------|-------------|----------|----------|
| ID | (Intercept) | 0.02538 | 0.1593 |
| | Residual | 0.58048 | 0.7619 |

Number of obs: 480, groups: ID, 80

Fixed effects:

| | Estimate | Std. Error | df |
|---|----------|------------|-----------|
| (Intercept) | 9.00181 | 0.03907 | 78.07315 |
| opp_size | 1.05141 | 0.04281 | 396.99857 |
| scale(body_size, center = TRUE, scale = TRUE) | 0.03310 | 0.03896 | 84.21144 |
| scale(assay_rep, scale = FALSE) | -0.05783 | 0.04281 | 396.99857 |
| block | -0.02166 | 0.06955 | 397.00209 |
| | t value | Pr(> t) | |
| (Intercept) | 230.395 | <2e-16 *** | |
| opp_size | 24.562 | <2e-16 *** | |
| scale(body_size, center = TRUE, scale = TRUE) | 0.850 | 0.398 | |

```

scale(assay_rep, scale = FALSE)           -1.351    0.177
block                                -0.311    0.756
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Correlation of Fixed Effects:

| | (Intr) | opp_sz | sc=Ts=T | s(_s=F |
|-------------|--------|--------|---------|--------|
| opp_size | 0.000 | | | |
| s(_,c=TRs=T | 0.000 | 0.000 | | |
| s(_,s=FALSE | 0.000 | -0.100 | 0.000 | |
| block | 0.000 | 0.000 | 0.002 | 0.000 |

So we'd probably calculate R using the individual and residual variance simply as:

```
0.02538 / (0.02538 + 0.58048)
```

```
[1] 0.04189087
```

🔥 Exercise

Do you see where I took the numbers ?

We can use some more fancy coding to extract the estimates and plugged them in a formula to estimate the repeatability

```

v_id <- VarCorr(m_3)$ID[1, 1]
v_r <- attr(VarCorr(m_3), "sc")^2
r_man <- v_id / (v_id + v_r)
r_man

```

```
[1] 0.04188879
```

Which yields an estimate of approximately R=4%. Strictly speaking we should make clear this a **conditional repeatability** estimate.

Conditional on what you might ask... on the fixed effects in your model. So our best estimate of 4% refers to the proportion of variance in aggressiveness *not explained by fixed effects* that is explained by individual identity. This

isn't much and still won't be significant, but illustrates the point that conditional repeatabilities often have a tendency to go up as people explain more of the residual variance by adding fixed effects. This is fine and proper, but can mislead the unwary reader. It also means that decisions about which fixed effects to include in your model need to be based on how you want to interpret R not just on, for instance, whether fixed effects are deemed significant.

17.2.7. A quick note on uncertainty

Using `lmer` in the `lme4` there isn't a really simple way to put some measure of uncertainty (SE or CI) on derived parameters like repeatabilities. This is a bit annoying. Such things are more easily done with other mixed model like `MCMCglmm` and `asreml` which are a bit more specialist. If you are using `lmer` for models you want to publish then you could look into the `rptR` (Stoffel et al. 2017). This acts as a ‘wrapper’ for `lmer` models and adds some nice functionality including options to bootstrap confidence intervals. Regardless, of how you do it, if you want to put a repeatability in one of your papers as a key result - it really should be accompanied by a measure of uncertainty just like any other effect size estimate.

Here I am estimating the repeatability and using bootstrap to estimate a confidence interval and a probability associated with the repeatability with the `rptR`. For more information about the use of the package and the theory behind it suggest the excellent paper associated with the package (Stoffel et al. 2017)

```
r_rpt <- rptGaussian(
  aggression ~ opp_size + block + (1 | ID),
  grname = "ID", data = unicorns
)
```

Bootstrap Progress:

```
r_rpt
```

Repeatability estimation using the lmm method

Repeatability for ID

R = 0.041

SE = 0.03

```
CI = [0, 0.103]
P = 0.0966 [LRT]
NA [Permutation]
```

17.2.8. An easy way to mess up your mixed models

We will try some more advanced mixed models in a moment to explore plasticity in aggressiveness a bit more. First let's quickly look for among-individual variance in focal body size. Why not? We have the data handy, everyone says morphological traits are very repeatable and - lets be honest - who wouldn't like to see a small P value after striking out with aggressiveness.

Include a random effect of ID as before and maybe a fixed effect of block, just to see if the beasties were growing a bit between data collection periods.

```
lmer_size <- lmer(body_size ~ block + (1 | ID),
  data = unicorns
)
```

Summarise and test the random effect.

Solution

```
summary(lmer_size)
```

```
Linear mixed model fit by REML. t-tests use Satterthwaite's method [
lmerModLmerTest]
```

```
Formula: body_size ~ block + (1 | ID)
```

```
Data: unicorns
```

```
REML criterion at convergence: 3460.7
```

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -1.80452 | -0.71319 | 0.00718 | 0.70280 | 1.81747 |

Random effects:

```

Groups     Name          Variance Std.Dev.
ID         (Intercept) 936.01   30.594
Residual              34.32    5.858
Number of obs: 480, groups: ID, 80

```

Fixed effects:

| | Estimate | Std. Error | df | t value | Pr(> t) | |
|----------------|----------|------------|----------|----------|------------|---|
| (Intercept) | 252.5031 | 3.4310 | 79.0000 | 73.595 | <2e-16 *** | |
| block | -0.1188 | 0.5348 | 399.0000 | -0.222 | 0.824 | |
| --- | | | | | | |
| Signif. codes: | 0 '***' | 0.001 '**' | 0.01 '*' | 0.05 '.' | 0.1 ' ' | 1 |

Correlation of Fixed Effects:

| | (Intr) |
|-------|--------|
| block | 0.000 |

```
ranova(lmer_size)
```

| | npar | logLik | AIC | LRT | Df | Pr(>Chisq) |
|----------|------|-----------|----------|----------|----|------------|
| | 4 | -1730.369 | 3468.738 | NA | NA | NA |
| (1 ID) | 3 | -2325.551 | 4657.103 | 1190.364 | 1 | 0 |

🔥 Exercise

What might you conclude, and why would this be foolish?

💡 Solution

Hopefully you spotted the problem here. You have fed in a data set with 6 records per individual (with 2 sets of 3 identical values per unicorns), when you know size was only measured twice in reality. This means you'd expect to get a (potentially very) upwardly biased estimate of R and a (potentially very) downwardly biased P value when testing among-individual variance.

🔥 Exercise

How can we do it properly?

Solution

We can prune the data to the two actual observations per unicorns by just selecting the first assay in each block.

```
unicorns2 <- unicorns[unicorns$assay_rep == 1, ]  
  
lmer_size2 <- lmer(body_size ~ block + (1 | ID),  
  data = unicorns2  
)  
  
summary(lmer_size2)
```

```
Linear mixed model fit by REML. t-tests use Satterthwaite's method [  
lmerModLmerTest]  
Formula: body_size ~ block + (1 | ID)  
Data: unicorns2
```

REML criterion at convergence: 1373.4

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -1.54633 | -0.56198 | 0.01319 | 0.56094 | 1.42095 |

Random effects:

| Groups | Name | Variance | Std.Dev. |
|--------|-------------|----------|----------|
| ID | (Intercept) | 912.84 | 30.213 |
| | Residual | 57.78 | 7.601 |

Number of obs: 160, groups: ID, 80

Fixed effects:

| | Estimate | Std. Error | df | t value | Pr(> t) |
|----------------|----------|------------|----------|----------|------------|
| (Intercept) | 252.5031 | 3.4310 | 79.0000 | 73.595 | <2e-16 *** |
| block | -0.1188 | 1.2019 | 79.0000 | -0.099 | 0.922 |
| --- | | | | | |
| Signif. codes: | 0 '***' | 0.001 '**' | 0.01 '*' | 0.05 '.' | 0.1 ' ' |

```
Correlation of Fixed Effects:
```

```
(Intr)
```

```
block 0.000
```

```
ranova(lmer_size2)
```

| | npar | logLik | AIC | LRT | Df | Pr(>Chisq) |
|----------|------|-----------|----------|----------|----|------------|
| | 4 | -686.6751 | 1381.350 | NA | NA | NA |
| (1 ID) | 3 | -771.9312 | 1549.862 | 170.5121 | 1 | 0 |

Summarise and test your random effect and you'll see the qualitative conclusions will actually be very similar using the pruned data set. Of course this won't generally be true, so just be careful. Mixed models are intended to help you model repeated measures data with non-independence, but they won't get you out of trouble if you mis-represent the true structure of observations on your dependent variable.

17.2.9. Happy mixed-modelling



Figure 17.8.: The superb unicorn

Chapter 18

Introduction to GLMM

18.1. Lecture

theoretical intro to glmm and introduce DHarma package to evaluate fit of glmm



Figure 18.1.: Dream pet dragon

18.2. Practical

This is an adapted version largely inspired by the tutorial in (Bolker et al. 2009). Spatial variation in nutrient availability and herbivory is likely to cause population differentiation and maintain genetic diversity in plant populations. Here we measure the extent to which mouse-ear cress (*Arabidopsis thaliana*) exhibits population and genotypic variation in their responses to these important environmental factors. We are particularly interested in whether these populations exhibit nutrient mediated compensation, where higher nutrient levels allow genotypes to better tolerate

herbivory (Banta et al. 2010). We use GLMMs to estimate the effect of nutrient levels, simulated herbivory, and their interaction on fruit production in *Arabidopsis thaliana*(fixed effects), and the extent to which populations vary in their responses(random effects, or variance components)

18.2.1. Packages and functions

You need to download the “extra_funs.R” script for some functions used in the Practical

```
library(lme4)
library(tidyverse)
library(patchwork)
library(lattice)
library(DHARMa)
source("code/extral_funs.R")
```

18.2.2. The data set

In this data set, the response variable is the number of fruits (i.e. seed capsules) per plant. The number of fruits produced by an individual plant(the experimental unit) was hypothesized to be a function of fixed effects,including nutrient levels (low vs. high), simulated herbivory (none vs. apical meristem damage), region (Sweden, Netherlands, Spain), and interactions among these. Fruit number was also a function of random effects including both the population and individual genotype. Because *Arabidopsis* is highly selfing, seeds of a single individual served as replicates of that individual. There were also nuisance variables, including the placement of the plant in the greenhouse, and the method used to germinate seeds. These were estimated as fixed effects but interactions were excluded.

- X observation number (we will use this observation number later, when we are accounting for overdispersion)
- `reg` a factor for region (Netherlands, Spain, Sweden).
- `popu` a factor with a level for each population.
- `gen` a factor with a level for each genotype.
- `rack` a nuisance factor for one of two greenhouse racks.
- `nutrient` a factor with levels for minimal or additional nutrients.
- `amd` a factor with levels for no damage or simulated herbivory (apical meristem damage; we will sometimes refer to this as “clipping”)

- `status` a nuisance factor for germination method.
- `total.fruits` the response; an integer count of the number of fruits per plant.

18.2.3. Specifying fixed and random Effects

Here we need to select a realistic full model, based on the scientific questions and the data actually at hand. We first load the data set and make sure that each variable is appropriately designated as numeric or factor (i.e.categorical variable).

```
dat_tf <- read.csv("data/Banta_TotalFruits.csv")
str(dat_tf)

'data.frame': 625 obs. of 9 variables:
 $ X           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ reg         : chr  "NL" "NL" "NL" "NL" ...
 $ popu        : chr  "3.NL" "3.NL" "3.NL" "3.NL" ...
 $ gen          : int  4 4 4 4 4 4 4 4 4 5 ...
 $ rack         : int  2 1 1 2 2 2 2 1 2 1 ...
 $ nutrient     : int  1 1 1 1 8 1 1 1 8 1 ...
 $ amd          : chr  "clipped" "clipped" "clipped" "clipped" ...
 $ status        : chr  "Transplant" "Petri.Plate" "Normal" "Normal" ...
 $ total.fruits: int  0 0 0 0 0 0 0 3 2 0 ...
```

The `X`, `gen`, `rack` and `nutrient` variables are coded as integers, but we want them to be factors. We use `mutate()` dplyr , which operates within the data set, to avoid typing lots of commands like `dat_tf$rack <- factor(dat_tf$rack)`. At the same time, we reorder the `clipping` variable so that "unclipped" is the reference level (we could also have used `relevel(amd, "unclipped")`).

```
dat_tf <- mutate(
  dat_tf,
  X = factor(X),
  gen = factor(gen),
  rack = factor(rack),
  amd = factor(amd, levels = c("unclipped", "clipped")),
```

```

nutrient = factor(nutrient, label = c("Low", "High"))
)

```

Now we check replication for each genotype (columns) within each population (rows).

```
(reptab <- with(dat_tf, table(popu, gen)))
```

| | gen | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|------|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|----|----|----|---|---|---|
| popu | 4 | 5 | 6 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 27 | 28 | 30 | 34 | 35 | 36 | | | | | | | |
| 1.SP | 0 | 0 | 0 | 0 | 0 | 39 | 26 | 35 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1.SW | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 28 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 2.SW | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 18 | 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3.NL | 31 | 11 | 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5.NL | 0 | 0 | 0 | 35 | 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5.SP | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 43 | 22 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6.SP | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 13 | 24 | 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7.SW | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 45 | 47 | 45 | 0 | 0 | |
| 8.SP | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 13 | 16 | 35 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

🔥 Exercise

Exercise: this mode of inspection is OK for this data set but might fail for much larger data sets or for more levels of nesting. See if you can think of some other numerical or graphical methods for inspecting the structure of data sets.

1. plot(reptab) gives a mosaic plot of the two-way table; examine this, see if you can figure out how to interpret it, and decide whether you think it might be useful
2. try the commands colSums(reptab>0) (and the equivalent for rowSums) and figure out what they are telling you.
3. Using this recipe, how would you compute the range of number of genotypes per treatment combination?

💡 Solution

1. Do you find the mosaic plot you obtained ugly and super hard to read? Me too 😊

```
plot(reptab)
```

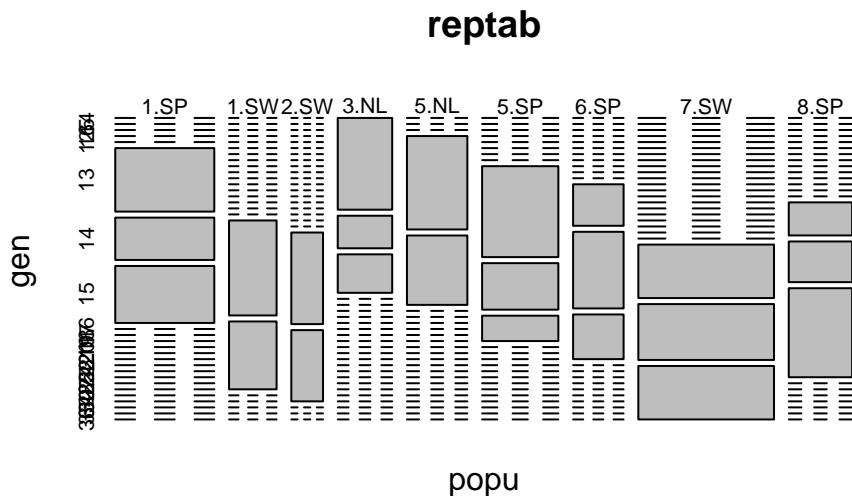


Figure 18.2.: A truly useless plot no one can understand

2. `colSums()` do the sum of all the rows for each columns of a table. So `colSums(reptab>0)` gives you for each genotype the number of populations (lines) where you have at least 1 observations.

```
colSums(reptab > 0)
```

```
4   5   6  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  27  28  30  34  35  36
1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

```
rowSums(reptab > 0)
```

```
1.SP 1.SW 2.SW 3.NL 5.NL 5.SP 6.SP 7.SW 8.SP
3     2     2     3     2     3     3     3     3
```

3. You first need to create a new table of number of observations per treatment and genotypes

```
reptab2 <- with(dat_tf, table(paste(amd, nutrient, sep = "_"), gen))
range(reptab2)
```

```
[1] 2 13
```

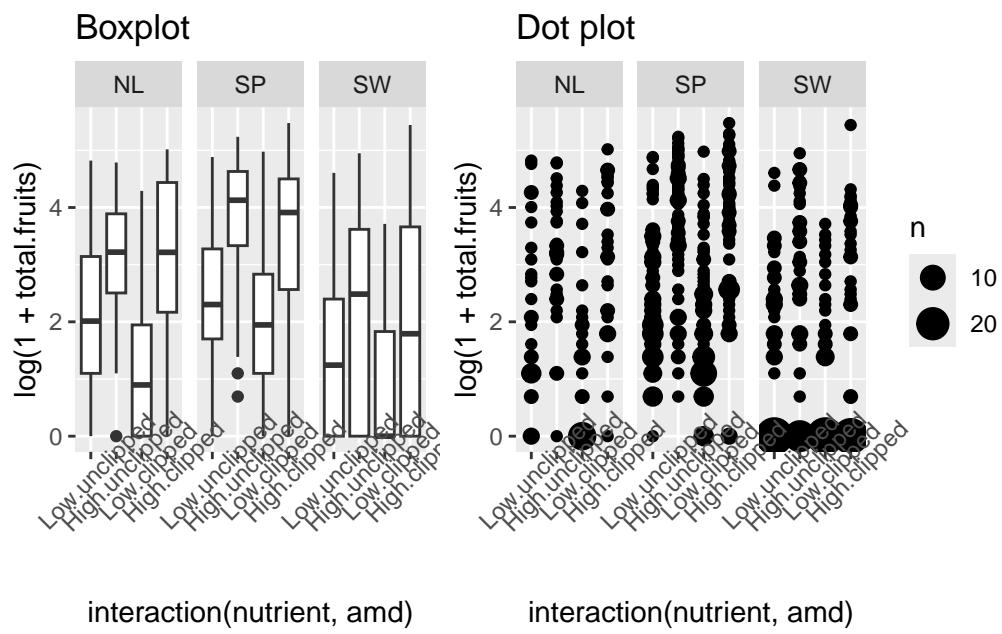
This reveals that we have only 2–4 populations per region and 2–3 genotypes per population. However, we also have 2–13 replicates per genotype for each treatment combination (four unique treatment combinations: 2 levels of nutrients by 2 levels of simulated herbivory). Thus, even though this was a reasonably large experiment (625 plants), there were a very small number of replicates with which to estimate variance components, and many more potential interactions than our data can support. Therefore, judicious selection of model terms, based on both biology and the data, is warranted. We note that we don't really have enough levels per random effect, nor enough replication per unique treatment combination. Therefore, we decide to omit the fixed effect of “region”, although we recognize that populations in different regions are widely geographically separated.

However, as in all GLMMs where the scale parameter is treated as fixed and deviations from the fixed scale parameter would be identifiable (i.e. Poisson and binomial ($N > 1$), but not binary, models) we may have to deal with overdispersion.

18.2.4. Look at overall patterns in data

I usually like to start with a relatively simple overall plot of the data, disregarding the random factors, just to see what's going on. For reasons to be discussed below, we choose to look at the data on the log (or $\log(1 + x)$) scale. Let's plot either box-and-whisker plots (useful summaries) or dot plots (more detailed, good for seeing if we missed anything).

Warning: `qplot()` was deprecated in ggplot2 3.4.0.

Figure 18.3.: Number of fruits ($\log + 1$) as a function of treatments

🔥 Exercise

Generate these plots and figure out how they work before continuing. Try conditioning/faceting on population rather than region: for `facet_wrap` you might want to take out the `nrow = 1` specification. If you want try reorder the subplots by overall mean fruit set and/or colour the points according to the region they come from.

Solution

```
p1 <- qplot(
  interaction(nutrient, amd),
  log(1 + total.fruits),
  data = dat_tf, geom = "boxplot") +
  facet_wrap(~reg, nrow = 1) +
  theme(axis.text.x = element_text(angle = 45)) +
  ggtitle("Boxplot")

p2 <- qplot(
  interaction(nutrient, amd),
  log(1 + total.fruits),
  data = dat_tf) +
  facet_wrap(~reg, nrow = 1) +
  stat_sum() +
  theme(axis.text.x = element_text(angle = 45)) +
  ggtitle("Dot plot")

p1 + p2
```

18.2.5. Choose an error distribution

The data are non-normal in principle (i.e., count data, so our first guess would be a Poisson distribution). If we transform total fruits with the canonical link function (`log`), we hope to see relatively homogeneous variances across categories and groups.

First we define a new factor that represents every combination of genotype and treatment ($\text{nutrient} \times \text{clipping}$) treatment, and sort it in order of increasing mean fruit set.

```
dat_tf <- dat_tf %>%
  mutate(
    gna = reorder(interaction(gen, nutrient, amd), total.fruits, mean)
  )
```

Now time to plot it

```
ggplot(dat_tf, aes(x = gna, y = log(1 + total.fruits))) +
  geom_boxplot() +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 90))
```

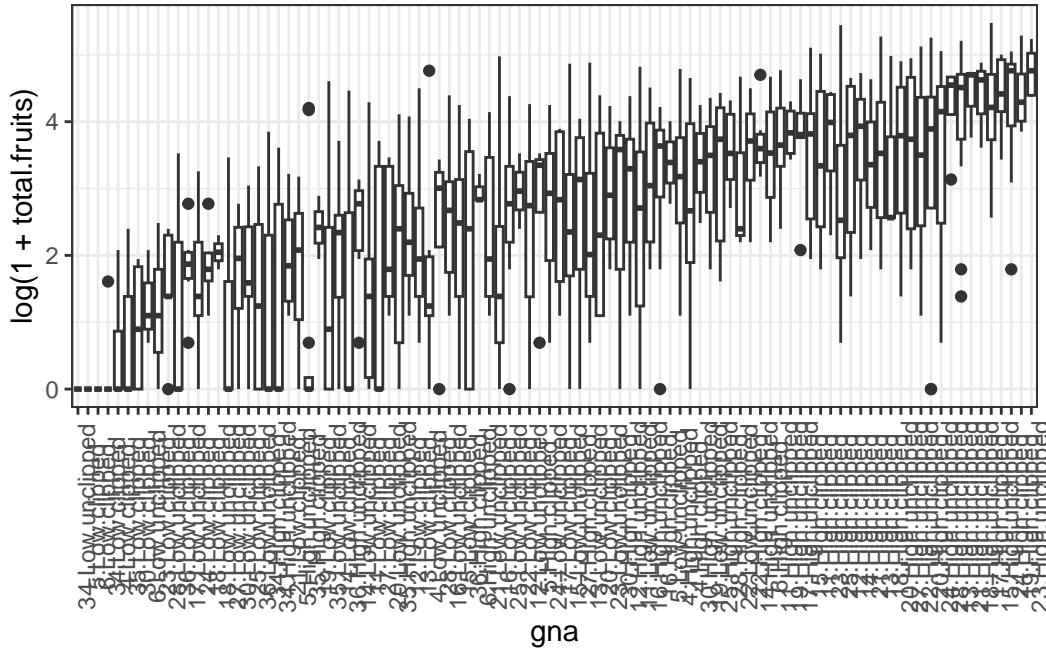


Figure 18.4.: Boxplot of total fruits ($\log + 1$) per genotypes and treatments

We could also calculate the variance for each genotype \times treatment combination and provide a statistical summary of these variances. This reveals substantial variation among the sample variances on the transformed data. In addition to heterogeneous variances across groups, Figure 1 reveals many zeroes in groups, and some groups with a mean and variance of zero, further suggesting we need a non-normal error distribution, and perhaps something other than a Poisson distribution.

We could calculate λ (mean) for each genotype \times treatment combination and provide a statistical summary of each group's λ .

```
grp_means <- with(dat_tf, tapply(total.fruits, list(gna), mean))
summary(grp_means)
```

| | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|--|------|---------|--------|-------|---------|--------|
| | 0.00 | 11.35 | 23.16 | 31.86 | 49.74 | 122.40 |

A core property of the Poisson distribution is that the variance is equal to the mean. A simple diagnostic is a plot of the group variances against the group means:

- Poisson-distributed data will result in a linear pattern with slope = 1
- as long as the variance is generally greater than the mean, we call the data overdispersed. Overdispersion comes in various forms:
 - a linear mean-variance relationship with $\text{Var} = \varphi\mu$ (a line through the origin) with $\varphi > 1$ is called a quasi-Poisson pattern (this term describes the mean-variance relationship, not any particular probability distribution); we can implement it statistically via quasilielihood (Venables and Ripley, 2002) or by using a particular parameterization of the negative binomial distribution (“NB1” in the terminology of Hardin and Hilbe (2007))
 - a semi-quadratic pattern, $\text{Var} = \mu(1 + \alpha\mu)$ or $\mu(1 + \mu/k)$, is characteristic of overdispersed data that is driven by underlying heterogeneity among samples, either the negative binomial (gamma-Poisson) or the lognormal-Poisson (Elston et al. 2001)

We've already calculated the group (genotype \times treatment) means, we calculate the variances in the same way.

```
grp_vars <- with(
  dat_tf,
  tapply(
    total.fruits,
    list(gna), var
  )
)
```

We can get approximate estimates of the quasi-Poisson (linear) and negative binomial (linear/quadratic) pattern using lm.

```
lm1 <- lm(grp_vars ~ grp_means - 1) ## `quasi-Poisson' fit
phi_fit <- coef(lm1)
lm2 <- lm((grp_vars - grp_means) ~ I(grp_means^2) - 1)
k_fit <- 1 / coef(lm2)
```

Now we can plot them.

```

plot(grp_vars ~ grp_means, xlab = "group means", ylab = "group variances")
abline(c(0, 1), lty = 2)
text(105, 500, "Poisson")
curve(phi_fit * x, col = 2, add = TRUE)
## bquote() is used to substitute numeric values
## in equations with symbols
text(110, 3900,
  bquote(paste("QP: ", sigma^2 == .(round(phi_fit, 1)) * mu)),
  col = 2
)
curve(x * (1 + x / k_fit), col = 4, add = TRUE)
text(104, 7200, paste("NB: k=", round(k_fit, 1), sep = ""), col = 4)
l_fit <- loess(grp_vars ~ grp_means)
mvec <- 0:120
lines(mvec, predict(l_fit, mvec), col = 5)
text(100, 2500, "loess", col = 5)

```

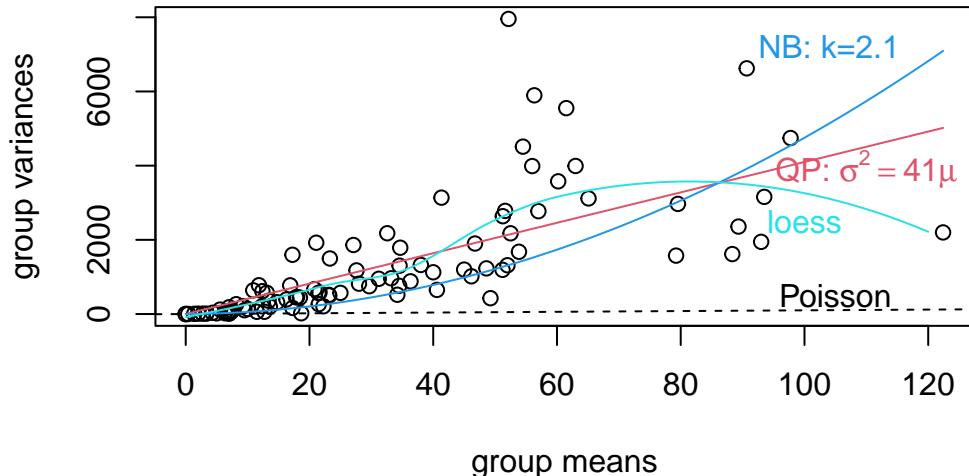


Figure 18.5.: Graphical evaluation of distribution to use

Same with ggplot

```

ggplot(
  data.frame(grp_means, grp_vars),
  aes(x = grp_means, y = grp_vars)) +
  geom_point() +
  geom_smooth(
    aes(colour = "Loess"), se = FALSE) +
  geom_smooth(
    method = "lm", formula = y ~ x - 1, se = FALSE,
    aes(colour = "Q_Pois")) +
  stat_function(
    fun = function(x) x * (1 + x / k_fit),
    aes(colour = "Neg_bin")
  ) +
  geom_abline(
    aes(intercept = 0, slope = 1, colour = "Poisson")) +
  scale_colour_manual(
    name = "legend",
    values = c("blue", "purple", "black", "red")) +
  scale_fill_manual(
    name = "legend",
    values = c("blue", "purple", "black", "red")) +
  guides(fill = FALSE)

```

Warning: The `<scale>` argument of `guides()` cannot be `FALSE`. Use "none" instead as of ggplot2 3.3.4.

`geom_smooth()` using method = 'loess' and formula = 'y ~ x'

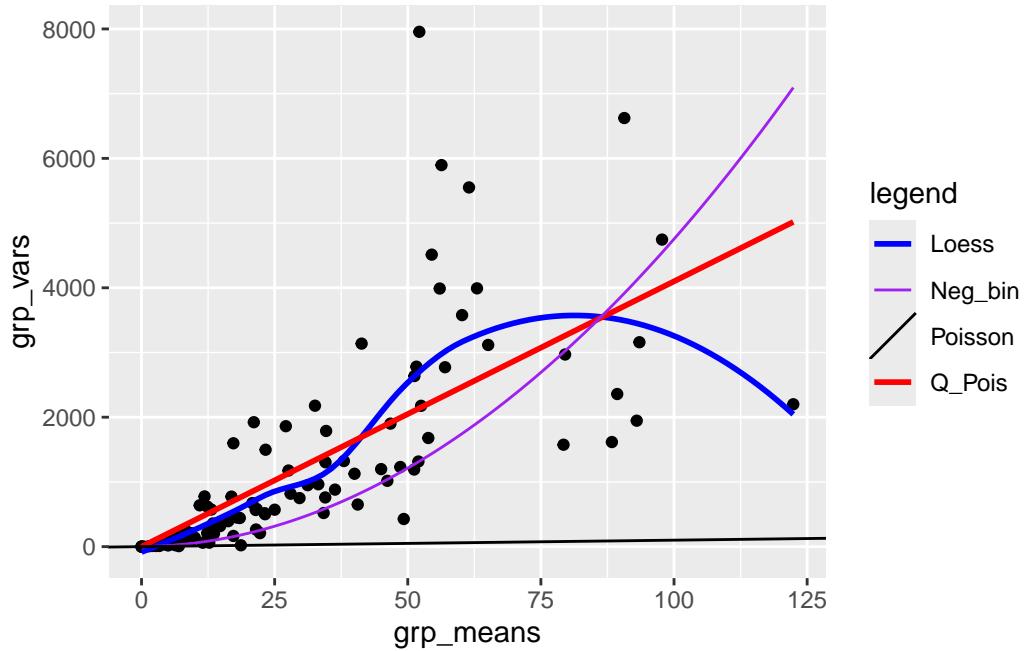


Figure 18.6.: Graphical evaluation of distribution to use with ggplot

These fits are not rigorous statistical tests — they violate a variety of assumptions of linear regression (e.g. constant variance, independence), but they are good enough to give us an initial guess about what distributions we should use.

Exercise

- compare a simple quadratic fit to the data (i.e., without the linear part) with the negative binomial and quasipoisson fits

Solution

```

lm3 <- lm(grp_vars ~ I(grp_means)^2 - 1) ## quadratic fit
quad_fit <- coef(lm3)

ggplot(
  data.frame(grp_means, grp_vars),
  aes(x = grp_means, y = grp_vars)) +
  geom_point() +
  geom_smooth(
    method = "lm", formula = y ~ x - 1, se = FALSE,
    aes(colour = "Q_Pois")) +
  stat_function(
    fun = function(x) x * (1 + x / k_fit),
    aes(colour = "Neg_bin"))
) +
  geom_smooth(
    method = "lm", formula = y ~ I(x^2) - 1, se = FALSE,
    aes(colour = "Quad")) +
  scale_colour_manual(
    name = "legend",
    values = c("blue", "purple", "black")) +
  scale_fill_manual(
    name = "legend",
    values = c("blue", "purple", "black")) +
  guides(fill = FALSE)

```

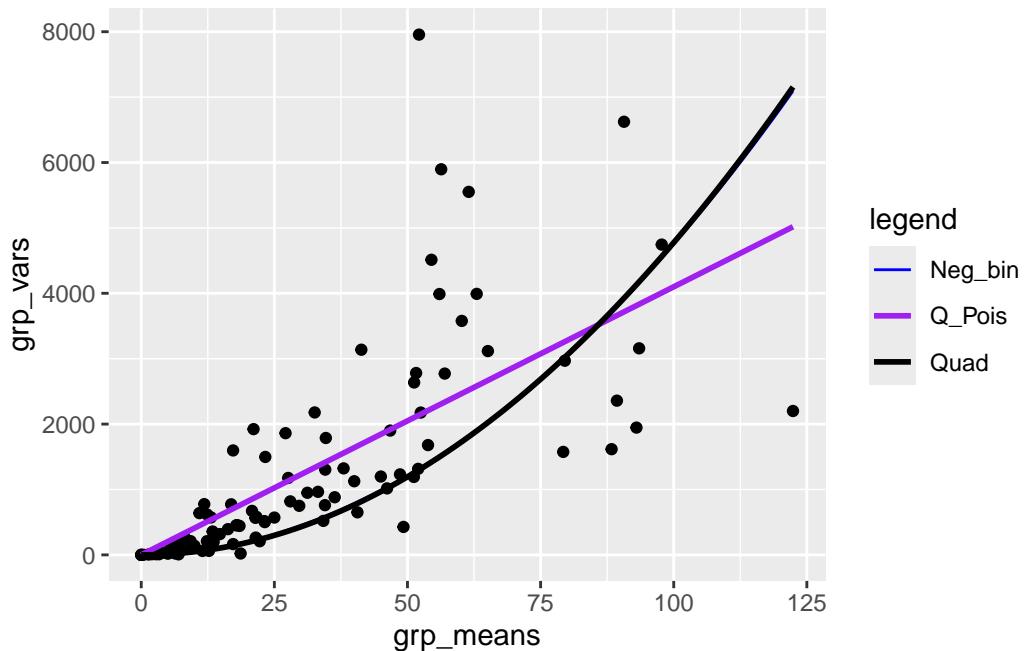


Figure 18.7.: Graphical evaluation of distribution to use including quadratic effect

18.2.5.1. Plotting the response vs treatments

Just to avoid surprises

```
ggplot(dat_tf, aes(x = amd, y = log(total.fruits + 1), colour = nutrient)) +
  geom_point() +
  ## need to use as.numeric(amd) to get lines
  stat_summary(aes(x = as.numeric(amd)), fun = mean, geom = "line") +
  theme_bw() +
  theme(panel.spacing = unit(0, "lines")) +
  facet_wrap(~popu)
```

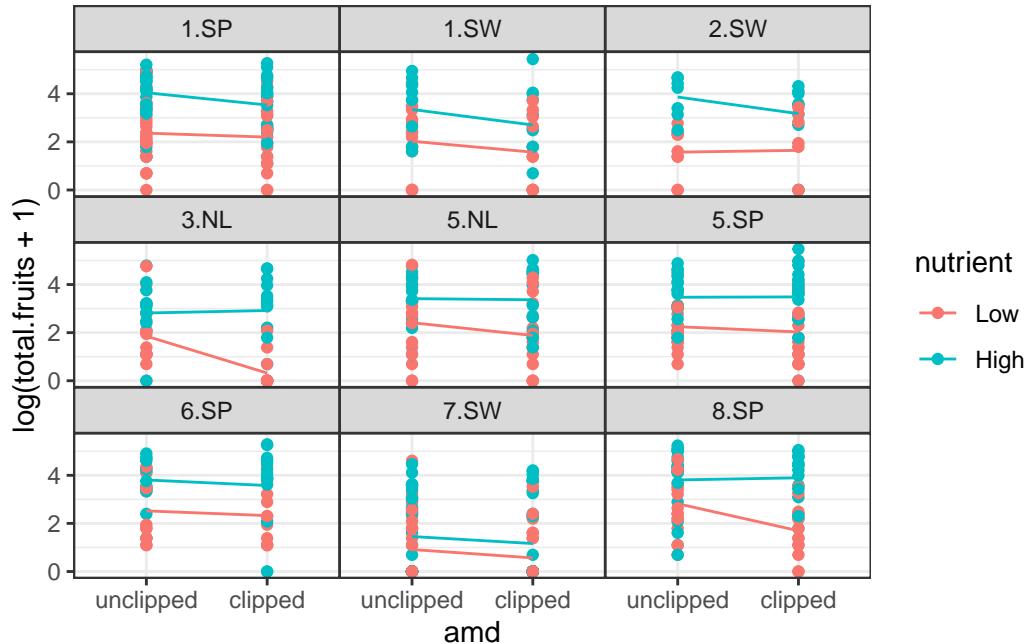


Figure 18.8.: Fruit production by treatments by population

```
ggplot(dat_tf, aes(x = amd, y = log(total.fruits + 1), colour = gen)) +
  geom_point() +
  stat_summary(aes(x = as.numeric(amd)), fun = mean, geom = "line") +
  theme_bw() +
  ## label_both adds variable name ('nutrient') to facet labels
  facet_grid(. ~ nutrient, labeller = label_both)
```

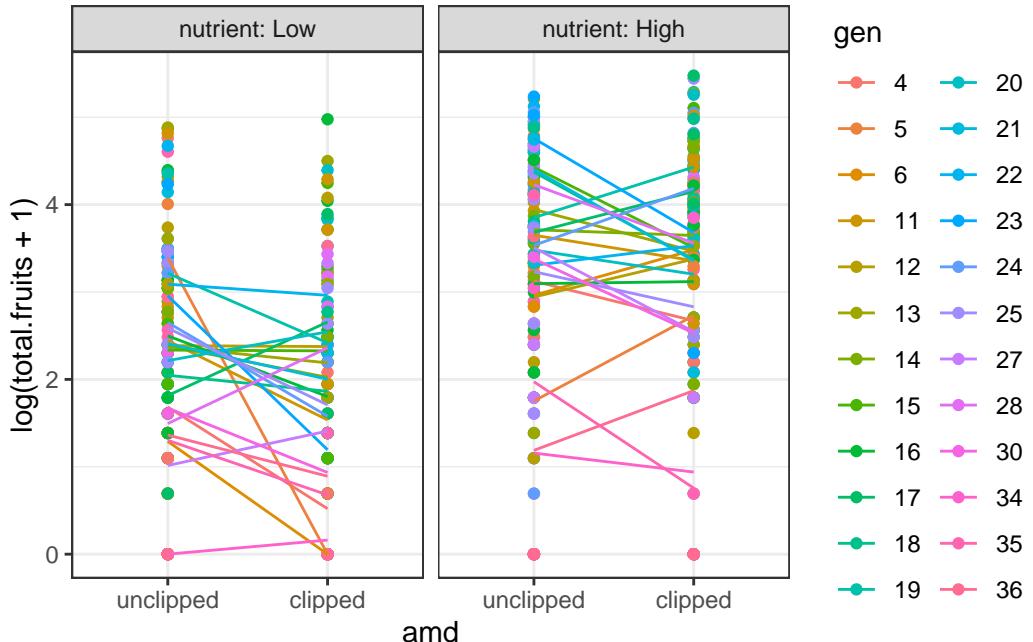


Figure 18.9.: Fruit production by genotype by treatments

18.2.6. Fitting group-wise GLM

Another general starting approach is to fit GLMs to each group of data separately, equivalent to treating the grouping variables as fixed effects. This should result in reasonable variation among treatment effects. We first fit the models, and then examine the coefficients.

```
glm_lis <- lmList(
  total.fruits ~ nutrient * amd | gen,
  data = dat_tf,
  family = "poisson")
plot.lmList(glm_lis)
```

Loading required package: reshape2

Attaching package: 'reshape2'

The following object is masked from 'package:tidyর':

smiths

Using grp as id variables

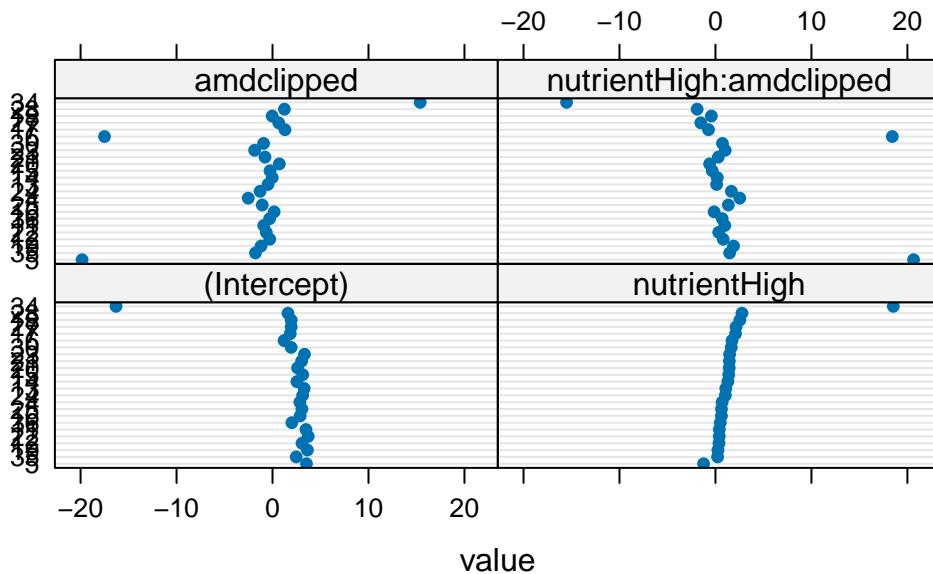


Figure 18.10.: Model coefficients for GLM fits on each genotype

Three genotypes (5, 6, 34) have extreme coefficients (Fig. 5). A mixed model assumes that the underlying random effects are normally distributed, although we shouldn't take these outliers too seriously at this point — we are not actually plotting the random effects, or even estimates of random effects (which are not themselves guaranteed to be normally distributed), but rather separate estimates for each group. Create a plotting function for Q-Q plots of these coefficients to visualize the departure from normality.

```
qqmath.lmList(glm_lis)
```

No id variables; using all as measure variables

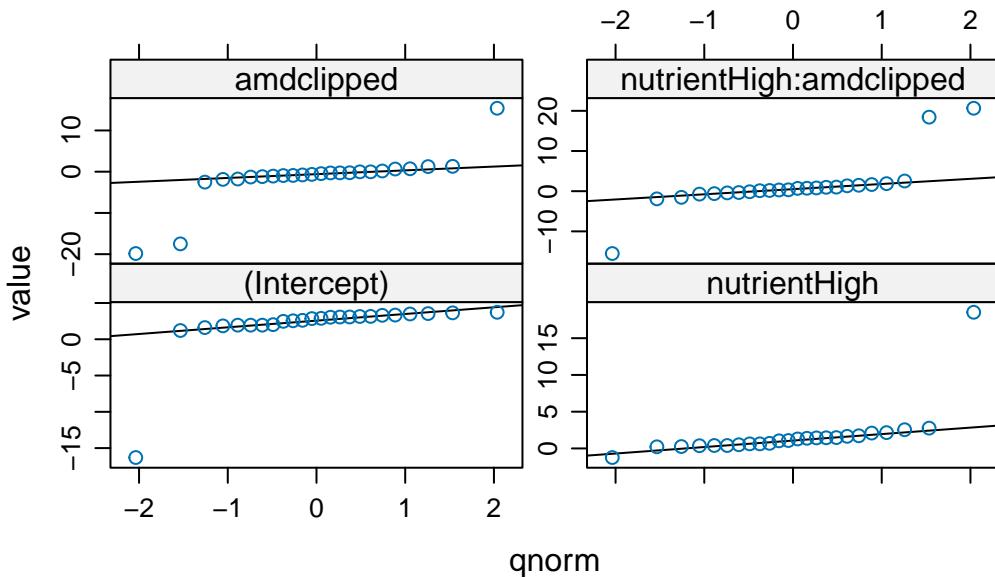


Figure 18.11.: Q-Q plots of model coefficients for GLM fits on each genotype

We see that these extreme coefficients fall far outside a normal error distribution. We shouldn't take these outliers too seriously at this point — we are not actually plotting the random effects, or even estimates of random effects, but rather separate estimates for each group. Especially if these groups have relatively small sample sizes, the estimates may eventually be “shrunk” closer to the mean when we do the mixed model. We should nonetheless take care to see if the coefficients for these genotypes from the GLMM are still outliers, and take the same precautions as we usually do for outliers. For example, we can look back at the original data to see if there is something weird about the way those genotypes were collected, or try re-running the analysis without those genotypes to see if the results are robust.

18.2.7. Fitting and evaluating GLMMs

Now we (try to) build and fit a full model, using `glmer` in the `emojif::emoji("pacakage") lme4`. This model has random effects for all genotype and population \times treatment random effects, and for the nuisance variables for the rack and germination method (status). (Given the mean-variance relationship we saw it's pretty clear that we are going to have to proceed eventually to a model with overdispersion, but we fit the Poisson model first for illustration.)

```
mp1 <- glmer(total.fruits ~ nutrient * amd +
  rack + status +
  (amd * nutrient | popu) +
```

```
(amd * nutrient | gen),
data = dat_tf, family = "poisson"
)
```

Warning in checkConv(attr(opt, "derivs"), opt\$par, ctrl = control\$checkConv, :
Model failed to converge with max|grad| = 0.0135718 (tol = 0.002, component 1)

```
overdisp_fun(mp1)
```

| chisq | ratio | p |
|-------------|----------|---------|
| 13909.47073 | 23.25998 | 0.00000 |

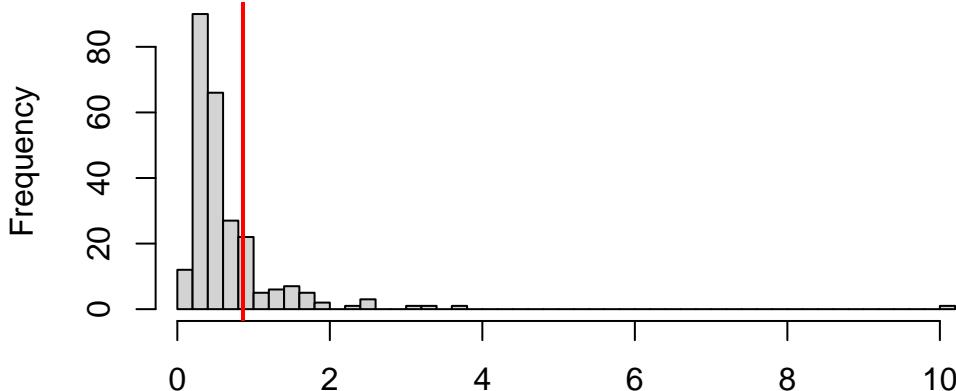
The `overdisp_fun()` is described [here](https://bbolker.github.io/mixedmodels-misc/glmmFAQ.html#testing-for-overdispersioncomputing-overdispersion-factor) <https://bbolker.github.io/mixedmodels-misc/glmmFAQ.html#testing-for-overdispersioncomputing-overdispersion-factor> on the absolutely fantastic FAQ about GLMMs by Ben Bolker <https://bbolker.github.io/mixedmodels-misc/glmmFAQ.html>

We can ignore the model convergence for the moment. This shows that the data are (extremely) over-dispersed, given the model.

We can also use the excellent DHARMA 📦 (Hartig 2022) to evaluate fit of `glm` and `glmm`. So instead of using the function `overdisp_fun()`, we can simply use the function `testDispersion()`.

```
testDispersion(mp1)
```

DHARMA nonparametric dispersion test via sd of residuals fitted vs. simulated



Simulated values, red line = fitted model. p-value (two.sided) = 0.384

```
DHARMA nonparametric dispersion test via sd of residuals fitted vs.  
simulated  
  
data: simulationOutput  
dispersion = 1.3003, p-value = 0.384  
alternative hypothesis: two.sided
```

As you can see, DHARMA suggests that there is no overdispersion based on the distribution of residuals from simulated data. We are going to consider that we have overdispersion and adjust the model accordingly.

Now we add the observation-level random effect to the model to account for overdispersion (Elston et al. 2001).

```
mp2 <- update(mp1, . ~ . + (1 | X))
```

```
Warning in (function (fn, par, lower = rep.int(-Inf, n), upper = rep.int(Inf, :  
failure to converge in 10000 evaluations
```

```
Warning in optwrap(optimizer, devfun, start, rho$lower, control = control, :  
convergence code 4 from Nelder_Mead: failure to converge in 10000 evaluations
```

```
Warning in checkConv(attr(opt, "derivs"), opt$par, ctrl = control$checkConv, :  
Model failed to converge with max|grad| = 0.173075 (tol = 0.002, component 1)
```

The model takes much longer to fit (and gives warnings). We look just at the variance components. In particular, if we look at the correlation matrix among the genotype random effects, we see a perfect correlation.

```
attr(VarCorr(mp2)$gen, "correlation")
```

| | (Intercept) | amdclipped | nutrientHigh |
|-------------------------|-------------|------------|--------------|
| (Intercept) | 1.0000000 | -0.9979733 | -0.9861333 |
| amdclipped | -0.9979733 | 1.0000000 | 0.9883406 |
| nutrientHigh | -0.9861333 | 0.9883406 | 1.0000000 |
| amdclipped:nutrientHigh | 0.8199582 | -0.8326069 | -0.9033829 |
| amdclipped:nutrientHigh | | | |

| | |
|-------------------------|------------|
| (Intercept) | 0.8199582 |
| amdclipped | -0.8326069 |
| nutrientHigh | -0.9033829 |
| amdclipped:nutrientHigh | 1.0000000 |

We'll try getting rid of the correlations between clipping (amd) and nutrients, using `amd+nutrient` instead of `amd*nutrient` in the random effects specification (here it seems easier to re-do the model rather than using `update` to add and subtract terms).

```
mp3 <- glmer(total.fruits ~ nutrient * amd +
  rack + status +
  (amd + nutrient | popu) +
  (amd + nutrient | gen) + (1 | X),
  data = dat_tf, family = "poisson"
)
```

Warning in checkConv(attr(opt, "derivs"), opt\$par, ctrl = control\$checkConv, :
 Model failed to converge with max|grad| = 0.225224 (tol = 0.002, component 1)

```
attr(VarCorr(mp3)$gen, "correlation")
```

| | | | |
|--------------|-------------|------------|--------------|
| | (Intercept) | amdclipped | nutrientHigh |
| (Intercept) | 1.0000000 | -0.9118731 | -0.9966458 |
| amdclipped | -0.9118731 | 1.0000000 | 0.9123919 |
| nutrientHigh | -0.9966458 | 0.9123919 | 1.0000000 |

```
attr(VarCorr(mp3)$popu, "correlation")
```

| | | | |
|--------------|-------------|------------|--------------|
| | (Intercept) | amdclipped | nutrientHigh |
| (Intercept) | 1.0000000 | 0.9947027 | 0.9969663 |
| amdclipped | 0.9947027 | 1.0000000 | 0.9909861 |
| nutrientHigh | 0.9969663 | 0.9909861 | 1.0000000 |

Unfortunately, we still have perfect correlations among the random effects terms. For some models (e.g. random-slope models), it is possible to fit random effects models in such a way that the correlation between the different parameters (intercept and slope in the case of random-slope models) is constrained to be zero, by fitting a model like $(1|f)+(0+x|f)$; unfortunately, because of the way lme4 is set up, this is considerably more difficult with categorical predictors (factors).

We have to reduce the model further in some way in order not to overfit (i.e., in order to not have perfect ± 1 correlations among random effects). It looks like we can't allow both nutrients and clipping in the random effect model at either the population or the genotype level. However, it's hard to know whether we should proceed with amd or nutrient, both, or neither in the model.

A convenient way to proceed if we are going to try fitting several different combinations of random effects is to fit the model with all the fixed effects but only observation-level random effects, and then to use update to add various components to it.

```
mp_obs <- glmer(total.fruits ~ nutrient * amd +
  rack + status +
  (1 | X),
  data = dat_tf, family = "poisson"
)
```

Now, for example, `update(mp_obs, .~.+(1|gen)+(amd|popu))` fits the model with intercept random effects at the genotype level and variation in clipping effects across populations.

🔥 Exercise

Exercise using `update`, fit the models with

1. clipping variation at both genotype and population levels;
2. nutrient variation at both genotype and populations; convince yourself that trying to fit variation in either clipping or nutrients leads to overfitting (perfect correlations).
3. Fit the model with only intercept variation at the population and genotype levels, saving it as mp4; show that there is non-zero variance estimated

💡 Solution

- 1.

```
mpcli <- update(mp_obs, . ~ . + (amd | gen) + (amd | popu))
```

Warning in checkConv(attr(opt, "derivs"), opt\$par, ctrl = control\$checkConv, :
Model failed to converge with max|grad| = 0.0833404 (tol = 0.002, component 1)

```
VarCorr(mpcli)
```

| Groups | Name | Std.Dev. | Corr |
|--------|-------------|----------|--------|
| X | (Intercept) | 1.431394 | |
| gen | (Intercept) | 0.293549 | |
| | amdclipped | 0.032813 | -0.944 |
| popu | (Intercept) | 0.750993 | |
| | amdclipped | 0.125908 | 0.998 |

2.

```
mpnut <- update(mp_obs, . ~ . + (nutrient | gen) + (nutrient | popu))
```

Warning in checkConv(attr(opt, "derivs"), opt\$par, ctrl = control\$checkConv, :
Model failed to converge with max|grad| = 0.0161168 (tol = 0.002, component 1)

```
VarCorr(mpnut)
```

| Groups | Name | Std.Dev. | Corr |
|--------|--------------|----------|--------|
| X | (Intercept) | 1.41918 | |
| gen | (Intercept) | 0.47719 | |
| | nutrientHigh | 0.32402 | -1.000 |
| popu | (Intercept) | 0.74716 | |
| | nutrientHigh | 0.12001 | 1.000 |

3.

```
mp4 <- update(mp_obs, . ~ . + (1 | gen) + (1 | popu))
```

Warning in checkConv(attr(opt, "derivs"), opt\$par, ctrl = control\$checkConv, :
Model failed to converge with max|grad| = 0.0212496 (tol = 0.002, component 1)

```
VarCorr(mp4)
```

| Groups | Name | Std.Dev. |
|--------|-------------|----------|
| X | (Intercept) | 1.43199 |
| gen | (Intercept) | 0.28669 |
| popu | (Intercept) | 0.80575 |

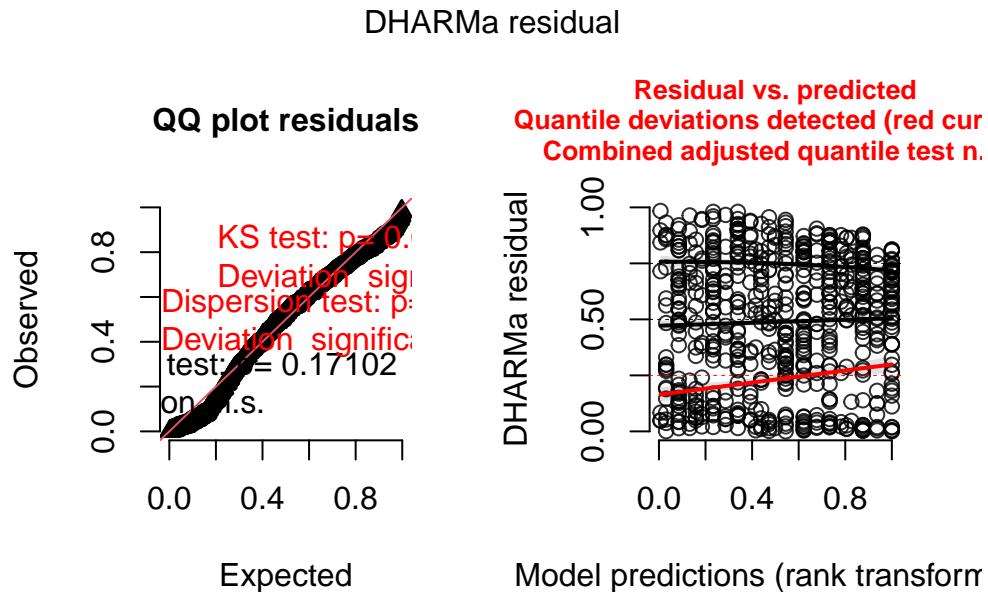
In other words, while it's biologically plausible that there is some variation in the nutrient or clipping effect at the genotype or population levels, with this modeling approach we really don't have enough data to speak confidently about these effects. Let's check that mp4 no longer incorporates overdispersion (the observationlevel random effect should have taken care of it):

```
overdisp_fun(mp4)
```

| chisq | ratio | p |
|-------------|-----------|-----------|
| 177.3714518 | 0.2884089 | 1.0000000 |

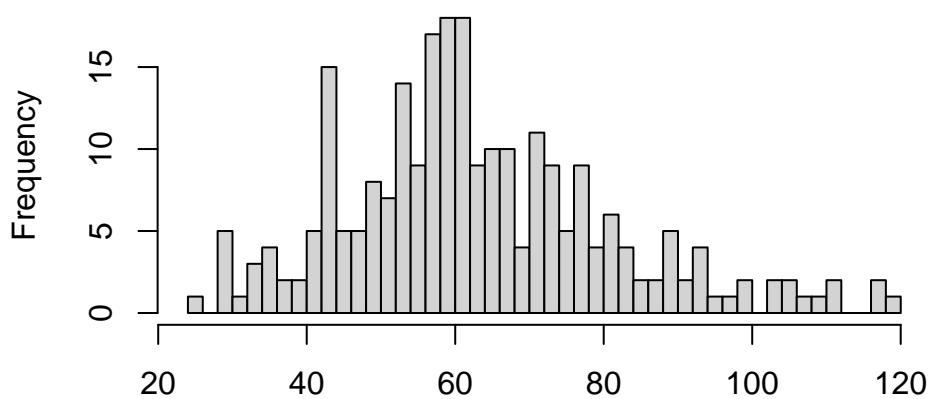
Using the DHARMA 📦, we will also check the model. To do so we first need to simulate some data and get the *scaled residuals* following the DHARMA notation. Then we can check the distributional properties of the *scaled residuals* and see if they follow the classic assumption using the different functions provided.

```
scaled_res <- simulateResiduals(mp4)  
plot(scaled_res)
```



```
testZeroInflation(mp4, plot = TRUE)
```

DHARMA zero-inflation test via comparison to expected zeros with simulation under H0 = fitted model



Simulated values, red line = fitted model. p-value (two.sided) = 0

DHARMA zero-inflation test via comparison to expected zeros with simulation under H0 = fitted model

```
data: simulationOutput
```

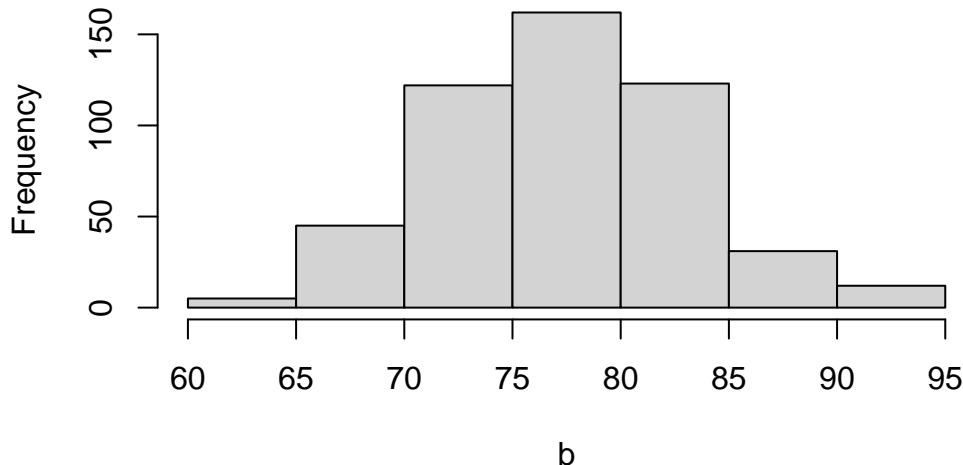
```
ratioObsSim = 1.9823, p-value < 2.2e-16
alternative hypothesis: two.sided
```

```
# note about overdispersion
sum(dat_tf$total.fruits == 0)
```

```
[1] 126
```

```
a <- predict(mp4, type = "response")
b <- rep(0, 500)
for (j in 1:500) {
  b[j] <- sum(sapply(seq(nrow(dat_tf)), function(i) rpois(1, a[i]))) == 0
}
hist(b)
```

Histogram of b



18.2.8. Inference

18.2.8.1. Random effects

`glmer` (`lmer`) does not return information about the standard errors or confidence intervals of the variance components.

[VarCorr\(mp4\)](#)

| Groups | Name | Std.Dev. |
|--------|-------------|----------|
| X | (Intercept) | 1.43199 |
| gen | (Intercept) | 0.28669 |
| popu | (Intercept) | 0.80575 |

18.2.8.1.1. Testing for random Effects If we want to test the significance of the random effects we can fit reduced models and run likelihood ratio tests via anova, keeping in mind that in this case (testing a null hypothesis of zero variance, where the parameter is on the boundary of its feasible region) the reported p value is approximately twice what it should be.

```
mp4v1 <- update(mp_obs, . ~ . + (1 | popu)) ## popu only (drop gen)
mp4v2 <- update(mp_obs, . ~ . + (1 | gen)) ## gen only (drop popu)
```

Warning in checkConv(attr(opt, "derivs"), opt\$par, ctrl = control\$checkConv, :
unable to evaluate scaled gradient

Warning in checkConv(attr(opt, "derivs"), opt\$par, ctrl = control\$checkConv, :
Model failed to converge: degenerate Hessian with 2 negative eigenvalues

[anova\(mp4, mp4v1\)](#)

| | npar | AIC | BIC | logLik | deviance | Chisq | Df | Pr(>Chisq) |
|-------|------|----------|----------|-----------|----------|----------|----|------------|
| mp4v1 | 9 | 5017.437 | 5057.377 | -2499.718 | 4999.437 | NA | NA | NA |
| mp4 | 10 | 5015.374 | 5059.751 | -2497.687 | 4995.374 | 4.063087 | 1 | 0.0438303 |

[anova\(mp4, mp4v2\)](#)

| | npar | AIC | BIC | logLik | deviance | Chisq | Df | Pr(>Chisq) |
|-------|------|----------|----------|-----------|----------|----------|----|------------|
| mp4v2 | 9 | 5031.585 | 5071.525 | -2506.793 | 5013.585 | NA | NA | NA |
| mp4 | 10 | 5015.374 | 5059.751 | -2497.687 | 4995.374 | 18.21153 | 1 | 1.98e-05 |

For various forms of linear mixed models, the RLRsim package can do efficient simulation-based hypothesis testing of variance components — unfortunately, that doesn't include GLMMs. If we are sufficiently patient we can do hypothesis testing via brute-force parametric bootstrapping where we repeatedly simulate data from the reduced (null) model, fit both the reduced and full models to the simulated data, and compute the distribution of the deviance (change in $-2 \log \text{likelihood}$). The code below took about half an hour on a reasonably modern desktop computer.

```
simdev <- function() {  
  newdat <- simulate(mp4v1)  
  reduced <- lme4::refit(mp4v1, newdat)  
  full <- lme4::refit(mp4, newdat)  
  2 * (c(logLik(full) - logLik(reduced)))  
}  
  
set.seed(101)  
nulldist0 <- replicate(2, simdev())  
## zero spurious (small) negative values  
nulldist[nulldist < 0 & abs(nulldist) < 1e-5] <- 0  
obsdev <- 2 * c(logLik(mp4) - logLik(mp4v1))  
  
mean(c(nulldist, obsdev) >= obsdev)
```

```
[1] 0.01492537
```

The true p-value is actually closer to 0.05 than 0.02. In other words, here the deviations from the original statistical model from that for which the original “p value is inflated by 2” rule of thumb was derived — fitting a GLMM instead of a LMM, and using a moderate-sized rather than an arbitrarily large (asymptotic) data set — have made the likelihood ratio test liberal (increased type I error) rather than conservative (decreased type I error).

We can also inspect the random effects estimates themselves (in proper statistical jargon, these might be considered “predictions” rather than “estimates” (Robinson, 1991)). We use the built-in dotplot method for the random effects extracted from glmer fits (i.e. ranef(model,condVar=TRUE)), which returns a list of plots, one for each random effect level in the model.

```

r1 <- as.data.frame(ranef(mp4, condVar = TRUE, whichel = c("gen", "popu")))
p1 <- ggplot(subset(r1, grpvar == "gen"), aes(y = grp, x = condval)) +
  geom_point() +
  geom_pointrange(
    aes(xmin = condval - condstd * 1.96, xmax = condval + condstd * 1.96)
  ) +
  geom_vline(aes(xintercept = 0, color = "red")) +
  theme_classic() +
  theme(legend.position = "none")
p2 <- ggplot(subset(r1, grpvar == "popu"), aes(y = grp, x = condval)) +
  geom_point() +
  geom_pointrange(
    aes(xmin = condval - condstd * 1.96, xmax = condval + condstd * 1.96)
  ) +
  geom_vline(aes(xintercept = 0, color = "red")) +
  theme_classic() +
  theme(legend.position = "none")
p1 + p2

```

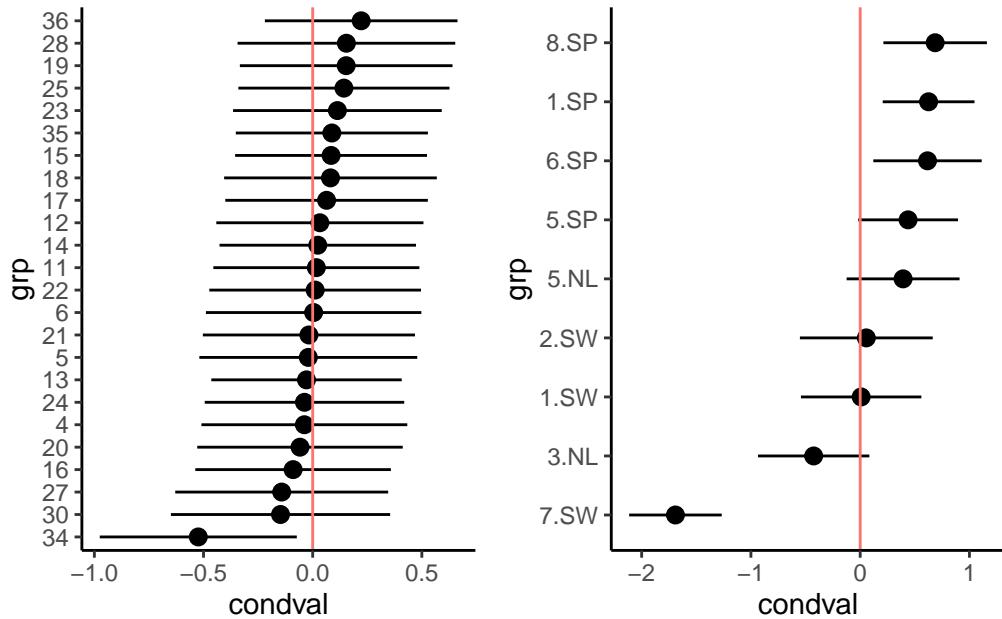


Figure 18.12.: Distribution of BLUPs for genotypes and populations

As expected from the similarity of the variance estimates, the population-level estimates (the only shared component) do not differ much between the two models. There is a hint of regional differentiation — the Spanish populations have higher fruit sets than the Swedish and Dutch populations. Genotype 34 again looks a little bit unusual.

18.2.8.2. Fixed effects

Now we want to do inference on the fixed effects. We use the drop1 function to assess both the AIC difference and the likelihood ratio test between models. (In glmm_funs.R we define a convenience function dfun to convert the AIC tables returned by drop1 (which we will create momentarily) into AIC tables.) Although the likelihood ratio test (and the AIC) are asymptotic tests, comparing fits between full and reduced models is still more accurate than the Wald (curvature-based) tests shown in the summary tables for glmer fits.

```
(dd_aic <- dfun(drop1(mp4)))
```

| | npar | dAIC |
|--------------|------|-----------|
| | NA | 0.000000 |
| rack | 1 | 55.081760 |
| status | 2 | 1.611490 |
| nutrient:amd | 1 | 1.443013 |

```
(dd_lrt <- drop1(mp4, test = "Chisq"))
```

| | npar | AIC | LRT | Pr(Chi) |
|--------------|------|----------|-----------|-----------|
| | NA | 5015.374 | NA | NA |
| rack | 1 | 5070.456 | 57.081760 | 0.0000000 |
| status | 2 | 5016.985 | 5.611491 | 0.0604617 |
| nutrient:amd | 1 | 5016.817 | 3.443013 | 0.0635198 |

On the basis of these comparisons, there appears to be a very strong effect of rack and weak effects of status and of the interaction term. Dropping the nutrient:amd interaction gives a (slightly) increased AIC (AIC = 1.4), so the full model has the best expected predictive capability (by a small margin). On the other hand, the p-value is slightly above 0.05 ($p = 0.06$). At this point we remove the non-significant interaction term so we can test the main effects.

(We don't worry about removing status because it measures an aspect of experimental design that we want to leave in the model whether it is significant or not.) Once we have fitted the reduced model, we can run the LRT via anova.

```
mp5 <- update(mp4, . ~ . - amd:nutrient)
anova(mp5, mp4)
```

| | npar | AIC | BIC | logLik | deviance | Chisq | Df | Pr(>Chisq) |
|-----|------|----------|----------|-----------|----------|----------|----|------------|
| mp5 | 9 | 5016.817 | 5056.757 | -2499.408 | 4998.817 | NA | NA | NA |
| mp4 | 10 | 5015.374 | 5059.751 | -2497.687 | 4995.374 | 3.443013 | 1 | 0.0635198 |

Exercise Test now the reduced model.

In the reduced model, we find that both nutrients and clipping have strong effects, whether measured by AIC or LRT. If we wanted to be still more careful about our interpretation, we would try to relax the asymptotic assumption. In classical linear models, we would do this by doing F tests with the appropriate denominator degrees of freedom. In “modern” mixed model approaches, we might try to use denominator-degree-of-freedom approximations such as the Kenward-Roger (despite the controversy over these approximations, they are actually available in `lmerTest`, but they do not apply to GLMMs). We can use a parametric bootstrap comparison between nested models to test fixed effects, as we did above for random effects, with the caveat that is computationally slow.

In addition, we can check the normality of the random effects and find they are reasonable (Fig. 10).

```
r5 <- as.data.frame(ranef(mp5))
ggplot(data = r5, aes(sample = condval)) +
  geom_qq() + geom_qq_line() +
  facet_wrap(~ grpvar) +
  theme_classic()
```

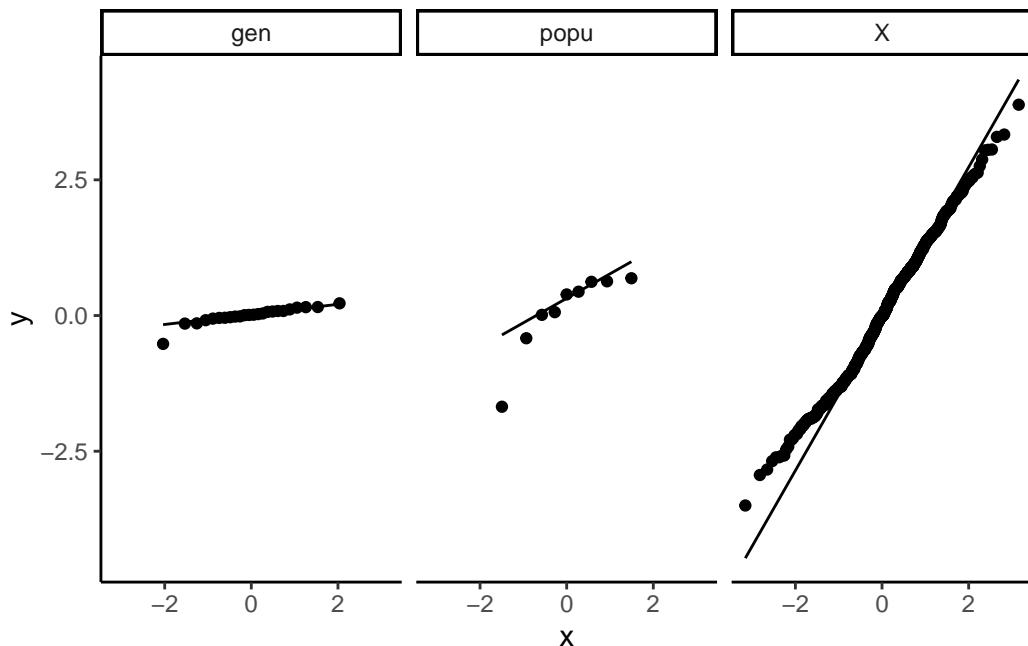
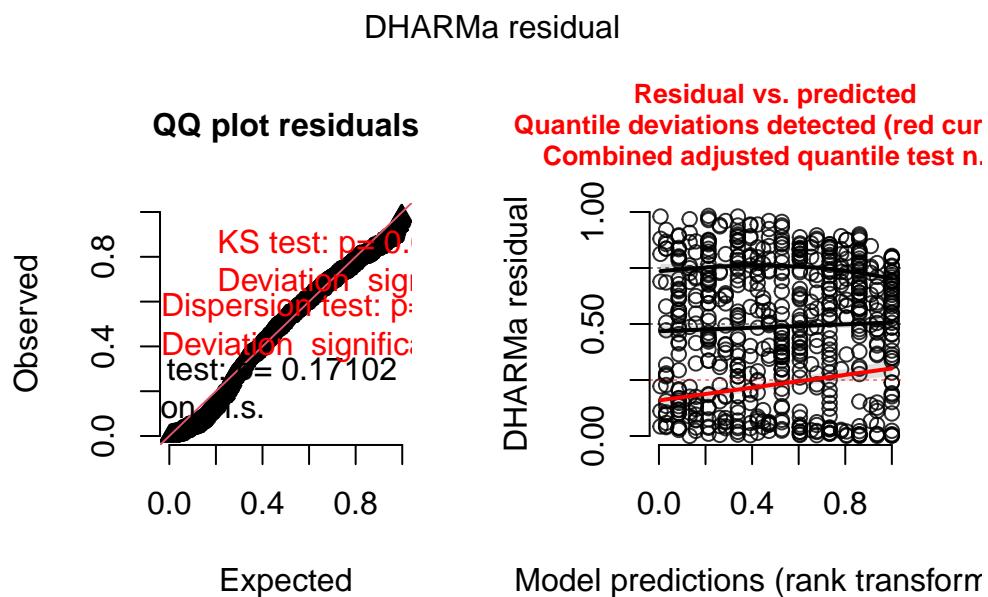


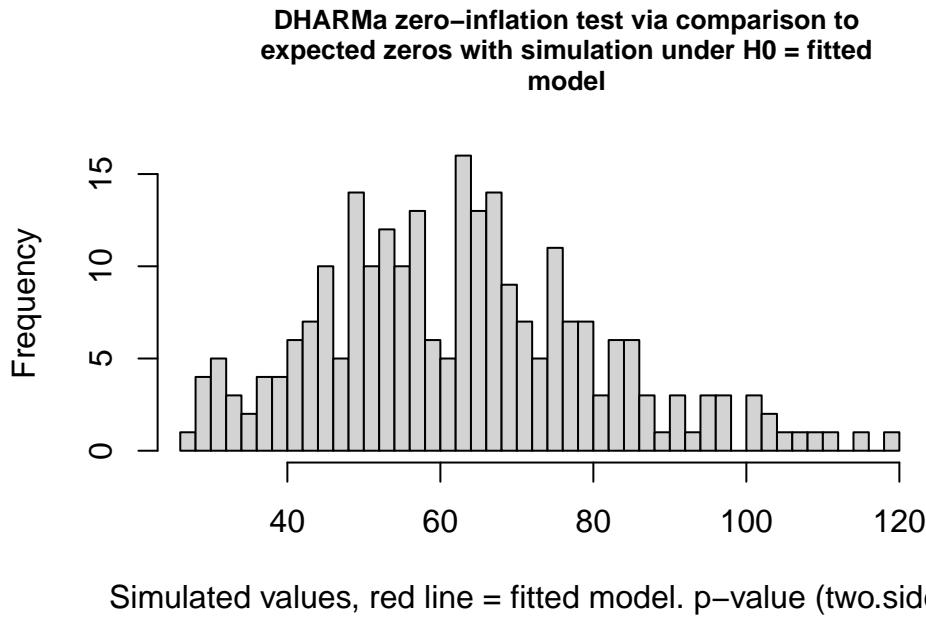
Figure 18.13.: Q-Q plot of BLUPs from model mp5

Checking everything with DHARMA also

```
scaled_res <- simulateResiduals(mp5)
plot(scaled_res)
```



```
testZeroInflation(mp5, plot = TRUE)
```



```
DHARMA zero-inflation test via comparison to expected zeros with
simulation under H0 = fitted model
```

```
data: simulationOutput
ratioObsSim = 1.9918, p-value < 2.2e-16
alternative hypothesis: two.sided
```

It is better than before but not perfect. I think this is completely OK and that it will extremely rarely be perfect. You need to learn what is acceptable (by that I mean you find acceptable) and be happy to justify and discuss your decisions.

18.2.9. Conclusions

Our final model includes fixed effects of nutrients and clipping, as well as the nuisance variables rack and status; observation-level random effects to account for overdispersion; and variation in overall fruit set at the population and genotype levels. However, we don't (apparently) have quite enough information to estimate the variation in clipping and nutrient effects, or their interaction, at the genotype or population levels. There is a strong overall

positive effect of nutrients and a slightly weaker negative effect of clipping. The interaction between clipping and nutrients is only weakly supported (i.e. the p-value is not very small), but it is positive and about the same magnitude as the clipping effect, which is consistent with the statement that “nutrients cancel out the effect of herbivory”.

🔥 Exercise

Exercise

- Re-do the analysis with region as a fixed effect.
- Re-do the analysis with a one-way layout as suggested above

18.2.10. Happy generalized mixed-modelling



Figure 18.14.: A GLMM character

Chapter 19

Random regression and character state approaches

19.1. Lecture

And here there would be dragons



Figure 19.1.: Dream pet dragon

19.2. Practical

In this practical, we will revisit our analysis on unicorn aggressivity. Honestly, we can use any other data with repeated measures for this exercise but I just ❤️ unicorns.

19.2.1. R packages needed

First we load required libraries

```
library(lme4)
library(tidyverse)
library(broom.mixed)
library(asreml)
library(MCMCglmm)
library(bayesplot)
library(patchwork)
```

19.2.2. Refresher on unicorn aggression

In the previous, practical on linear mixed models, we simply explored the differences among individuals in their mean aggression (Intercept), but we assumed that the response to the change in aggression with the opponent size (i.e. plasticity) was the same for all individuals. However, this plastic responses can also vary amon individuals. This is called IxE, or individual by environment interaction. To test if individuals differ in their plasticity we can use a random regression, whcih is simply a mixed-model where we fit both a random intercept and a random slope effect.

Following analysis from the previous pratical, our model of interest using scaled covariate was:

```
aggression ~ opp_size + body_size_sc + assay_rep_sc + block
+ (1 | ID)
```

We should start by loading the data and refitting the model using `lmer()`.

```
unicorns <- read.csv("data/unicorns_aggression.csv")
unicorns <- unicorns %>%
  mutate(
    body_size_sc = scale(body_size),
    assay_rep_sc = scale(assay_rep, scale = FALSE)
  )
```

```
m_mer <- lmer(
  aggression ~ opp_size + body_size_sc + assay_rep_sc + block
  + (1 | ID),
  data = unicorns
)
summary(m_mer)
```

Linear mixed model fit by REML ['lmerMod']
 Formula: aggression ~ opp_size + body_size_sc + assay_rep_sc + block +
 (1 | ID)
 Data: unicorns

REML criterion at convergence: 1136.5

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -2.85473 | -0.62831 | 0.02545 | 0.68998 | 2.74064 |

Random effects:

| Groups | Name | Variance | Std.Dev. |
|--------|-------------|----------|----------|
| ID | (Intercept) | 0.02538 | 0.1593 |
| | Residual | 0.58048 | 0.7619 |

Number of obs: 480, groups: ID, 80

Fixed effects:

| | Estimate | Std. Error | t value |
|--------------|----------|------------|---------|
| (Intercept) | 9.00181 | 0.03907 | 230.395 |
| opp_size | 1.05141 | 0.04281 | 24.562 |
| body_size_sc | 0.03310 | 0.03896 | 0.850 |
| assay_rep_sc | -0.05783 | 0.04281 | -1.351 |
| block | -0.02166 | 0.06955 | -0.311 |

Correlation of Fixed Effects:

```
(Intr) opp_sz bdy_s_ assy__  
opp_size      0.000  
body_siz_sc   0.000  0.000  
assay_rp_sc   0.000 -0.100  0.000  
block         0.000  0.000  0.002  0.000
```

We can now plot the predictions for each of our observations and plot for the observed and the fitted data for each individuals. Todo so we will use the `augment()` function from the  `broom.mixed`.

Below, we plot the raw data for each individual in one panel, with the fitted slopes in a second panel. Because we have 2 blocks of data, and block is fitted as a fixed effect, for ease of presentation we need to either select only 1 block for representation, take the average over the block effect or do a more complex graph with the two blocks. Here I have selected only one of the blocks for this plot

```
pred_m_mer <- augment(m_mer) %>%  
  select(ID, block, opp_size, .fitted, aggression) %>%  
  filter(block == -0.5) %>%  
  gather(  
    type, aggression,  
    `~.fitted`~aggression  
  )  
  
ggplot(pred_m_mer, aes(x = opp_size, y = aggression, group = ID)) +  
  geom_line(alpha = 0.3) +  
  theme_classic() +  
  facet_grid(. ~ type)
```

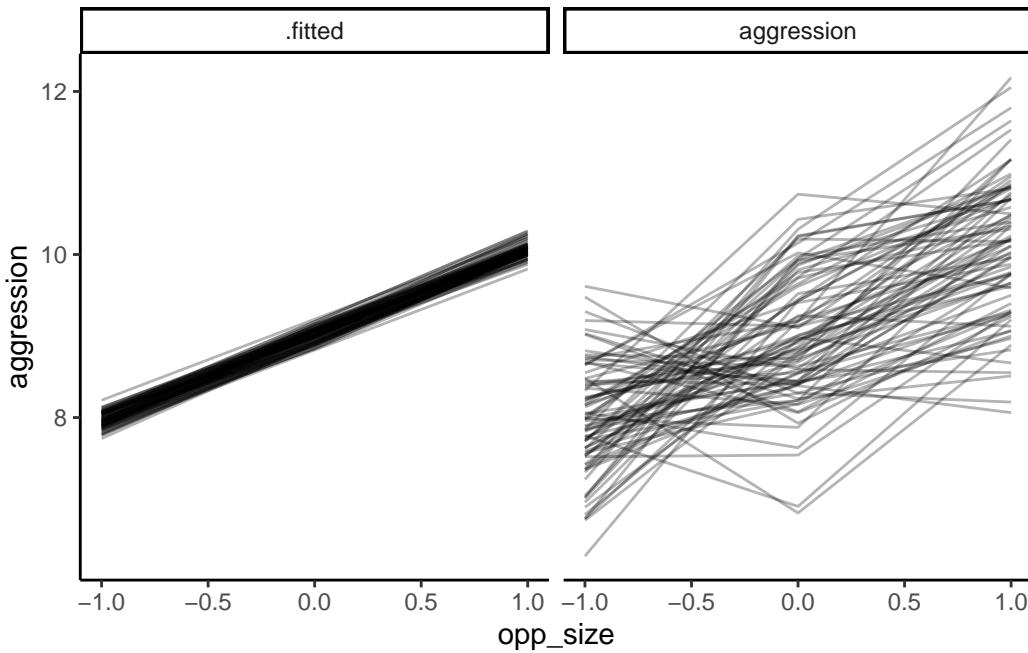


Figure 19.2.: Predicted (from `m_mer`) and observed value of aggression as a function of opponent size in unicorns

This illustrates the importance of using model predictions to see whether the model actually fits the individual-level data well or not — while the diagnostic plots looked fine, and the model captures mean plasticity, here we can see that the model really doesn't fit the actual data very well at all.

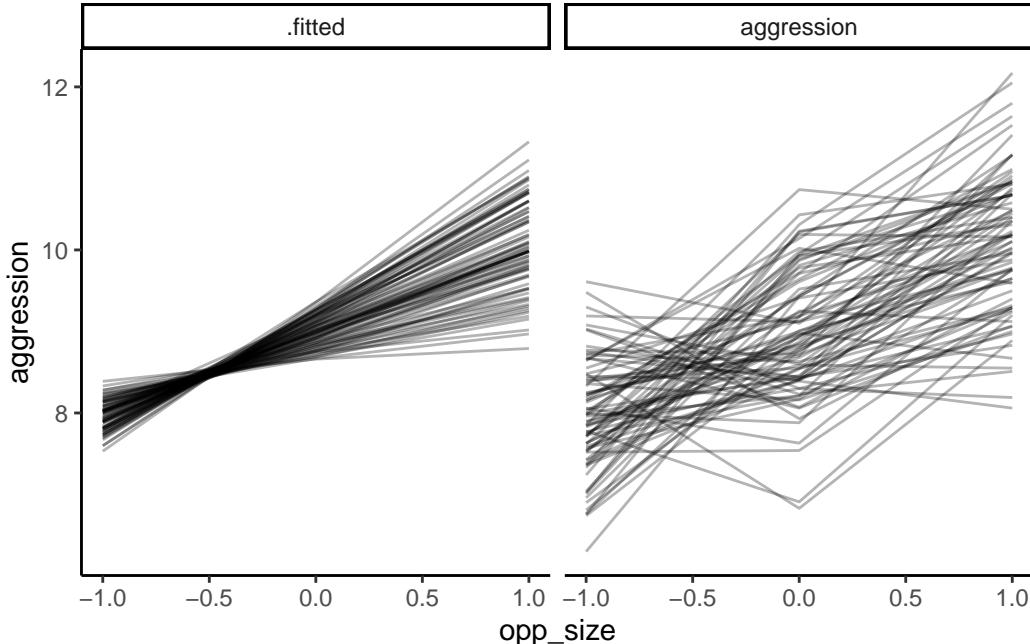
19.2.3. Random regression

19.2.3.1. with `lme4`

```
rr_mer <- lmer(
  aggression ~ opp_size + body_size_sc + assay_rep_sc + block
  + (1 + opp_size | ID),
  data = unicorns
)
```

```
pred_rr_mer <- augment(rr_mer) %>%
  select(ID, block, opp_size, .fitted, aggression) %>%
  filter(block == -0.5) %>%
  gather(type, aggression, `^.fitted`:`aggression`)
```

```
ggplot(pred_rr_mer, aes(x = opp_size, y = aggression, group = ID)) +
  geom_line(alpha = 0.3) +
  theme_classic() +
  facet_grid(. ~ type)
```



We can test the improvement of the model fit using the overloaded `anova` function in R to perform a likelihood ratio test (LRT):

```
anova(rr_mer, m_mer, refit = FALSE)
```

| | npar | AIC | BIC | logLik | deviance | Chisq | Df | Pr(>Chisq) |
|--------|------|----------|----------|-----------|----------|---------|----|------------|
| m_mer | 7 | 1150.477 | 1179.693 | -568.2383 | 1136.477 | NA | NA | NA |
| rr_mer | 9 | 1092.356 | 1129.920 | -537.1780 | 1074.356 | 62.1206 | 2 | 0 |

We can see here that the LRT uses a chi-square test with 2 degrees of freedom, and indicates that the random slopes model shows a statistically significant improvement in model fit. The 2df are because there are two additional (co)variance terms estimated in the random regression model: a variance term for individual slopes, and the covariance (or correlation) between the slopes and intercepts. Let's look at those values, and also the fixed effects parameters, via the model summary:

```
summary(rr_mer)
```

Linear mixed model fit by REML ['lmerMod']
 Formula: aggression ~ opp_size + body_size_sc + assay_rep_sc + block +
 (1 + opp_size | ID)
 Data: unicorns

REML criterion at convergence: 1074.4

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|----------|---------|---------|
| -3.04932 | -0.59780 | -0.02002 | 0.59574 | 2.68010 |

Random effects:

| Groups | Name | Variance | Std.Dev. | Corr |
|--------|-------------|----------|----------|------|
| ID | (Intercept) | 0.05043 | 0.2246 | |
| | opp_size | 0.19167 | 0.4378 | 0.96 |
| | Residual | 0.42816 | 0.6543 | |

Number of obs: 480, groups: ID, 80

Fixed effects:

| | Estimate | Std. Error | t value |
|--------------|----------|------------|---------|
| (Intercept) | 9.00181 | 0.03902 | 230.707 |
| opp_size | 1.05033 | 0.06123 | 17.153 |
| body_size_sc | 0.02725 | 0.03377 | 0.807 |
| assay_rep_sc | -0.04702 | 0.03945 | -1.192 |
| block | -0.02169 | 0.05973 | -0.363 |

Correlation of Fixed Effects:

| (Intr) | opp_sz | bdy_s_ | assy__ |
|-------------|--------|--------|--------|
| opp_size | 0.495 | | |
| body_siz_sc | 0.000 | 0.000 | |
| assay_rp_sc | 0.000 | -0.064 | -0.006 |

```
block      0.000  0.000  0.002  0.000
```

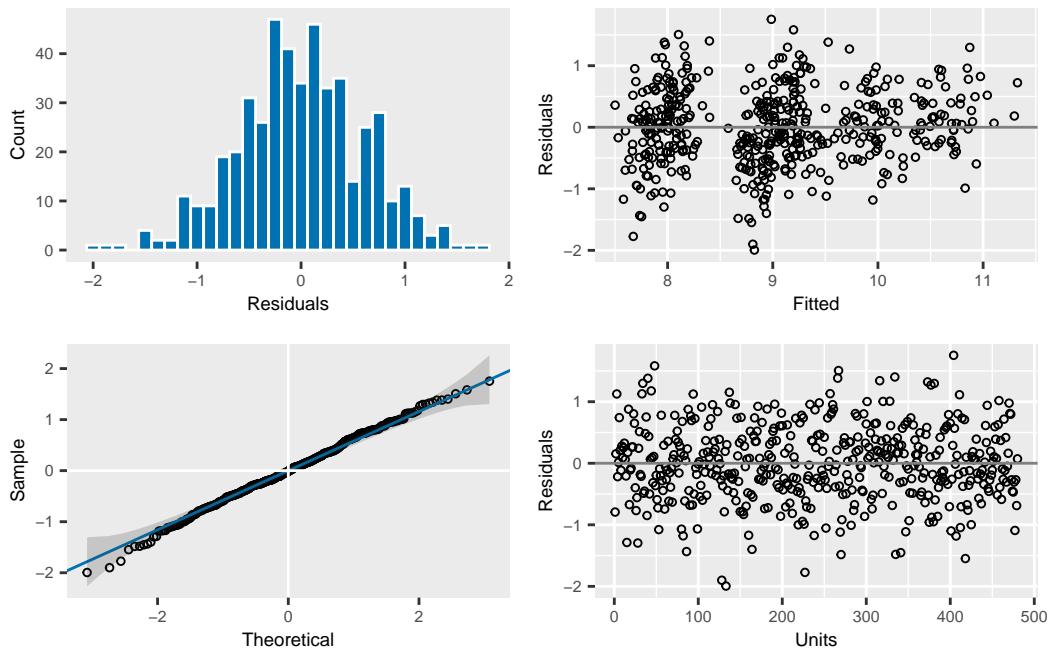
19.2.3.2. with asreml

```
unicorns <- unicorns %>%
  mutate( ID = as.factor(ID))
rr_asr <- asreml(
  aggression ~ opp_size + body_size_sc + assay_rep_sc + block,
  random = ~str(~ ID + ID:opp_size, ~us(2):id(ID)),
  residual = ~ units,
  data = unicorns,
  maxiter = 200
)
```

ASReml Version 4.2 11/10/2024 13:38:22

| | LogLik | Sigma2 | DF | wall |
|---|-----------|-----------|-----|----------|
| 1 | -109.4261 | 0.4632316 | 475 | 13:38:22 |
| 2 | -105.0501 | 0.4545934 | 475 | 13:38:22 |
| 3 | -101.8142 | 0.4436619 | 475 | 13:38:22 |
| 4 | -100.8141 | 0.4338731 | 475 | 13:38:22 |
| 5 | -100.6827 | 0.4285963 | 475 | 13:38:22 |
| 6 | -100.6821 | 0.4281695 | 475 | 13:38:22 |

```
plot(rr_asr)
```



```
summary(rr_asr, coef = TRUE)$coef.fixed
```

| | solution | std error | z.ratio |
|--------------|-------------|------------|-------------|
| (Intercept) | 9.00181250 | 0.03901766 | 230.7112239 |
| opp_size | 1.05032703 | 0.06123110 | 17.1534907 |
| body_size_sc | 0.02725092 | 0.03377443 | 0.8068506 |
| assay_rep_sc | -0.04702032 | 0.03944594 | -1.1920191 |
| block | -0.02168725 | 0.05973354 | -0.3630665 |

```
wa <- wald(rr_asr, ssType = "conditional", denDF = "numeric")
```

ASReml Version 4.2 11/10/2024 13:38:23

| | LogLik | Sigma2 | DF | wall |
|---|-----------|-----------|-----|----------|
| 1 | -100.6821 | 0.4281680 | 475 | 13:38:23 |
| 2 | -100.6821 | 0.4281680 | 475 | 13:38:23 |

```
attr(wa$Wald, "heading") <- NULL
```

```
wa
```

```
$Wald
```

| | Df | denDF | F.inc | F.con | Margin | Pr |
|--------------|----|-------|-------|-------|--------|---------|
| (Intercept) | 1 | 78.3 | 65490 | 53230 | | 0.00000 |
| opp_size | 1 | 79.5 | 293 | 294 | A | 0.00000 |
| body_size_sc | 1 | 84.3 | 1 | 1 | A | 0.42202 |
| assay_rep_sc | 1 | 387.6 | 1 | 1 | A | 0.23398 |
| block | | 318.1 | 0 | 0 | A | 0.71680 |

\$stratumVariances

| | df | Variance | ID+ID:opp_size!us(2)_1:1 |
|--------------------------|-----------|--------------------------|--------------------------|
| ID+ID:opp_size!us(2)_1:1 | 78.00483 | 0.4790737 | 5.216311 |
| ID+ID:opp_size!us(2)_2:1 | 0.00000 | 0.0000000 | 0.000000 |
| ID+ID:opp_size!us(2)_2:2 | 78.94046 | 1.1937287 | 0.000000 |
| units!R | 318.05470 | 0.4281680 | 0.000000 |
| | | ID+ID:opp_size!us(2)_2:1 | ID+ID:opp_size!us(2)_2:2 |
| ID+ID:opp_size!us(2)_1:1 | | -3.301137 | 0.5221955 |
| ID+ID:opp_size!us(2)_2:1 | | 0.000000 | 0.0000000 |
| ID+ID:opp_size!us(2)_2:2 | | 0.000000 | 3.9943993 |
| units!R | | 0.000000 | 0.0000000 |
| | | units!R | |
| ID+ID:opp_size!us(2)_1:1 | 1 | | |
| ID+ID:opp_size!us(2)_2:1 | 1 | | |
| ID+ID:opp_size!us(2)_2:2 | 1 | | |
| units!R | 1 | | |

```
summary(rr_asr)$varcomp
```

| | component | std.error | z.ratio | bound | %ch |
|--------------------------|-----------|-----------|-----------|-------|-----|
| ID+ID:opp_size!us(2)_1:1 | 0.0504293 | 0.0202756 | 2.487187 | P | 0 |
| ID+ID:opp_size!us(2)_2:1 | 0.0945834 | 0.0240074 | 3.939751 | P | 0 |
| ID+ID:opp_size!us(2)_2:2 | 0.1916592 | 0.0483206 | 3.966410 | P | 0 |
| units!R | 0.4281695 | 0.0339532 | 12.610582 | P | 0 |

```
rio_asr <- asreml(
  aggression ~ opp_size + body_size_sc + assay_rep_sc + block,
  random = ~ ID,
  residual = ~units,
  data = unicorns,
  maxiter = 200
)
```

ASReml Version 4.2 11/10/2024 13:38:23

| | LogLik | Sigma2 | DF | wall |
|---|-----------|-----------|-----|----------|
| 1 | -132.6114 | 0.5603527 | 475 | 13:38:23 |
| 2 | -132.1061 | 0.5670427 | 475 | 13:38:23 |
| 3 | -131.7956 | 0.5751571 | 475 | 13:38:23 |
| 4 | -131.7426 | 0.5807624 | 475 | 13:38:23 |
| 5 | -131.7425 | 0.5804802 | 475 | 13:38:23 |

```
pchisq(2 * (rr_asr$loglik - rio_asr$loglik), 2,
lower.tail = FALSE
)
```

[1] 3.241026e-14

```
vpredict(rr_asr, cor_is ~ V2 / (sqrt(V1) * sqrt(V3)))
```

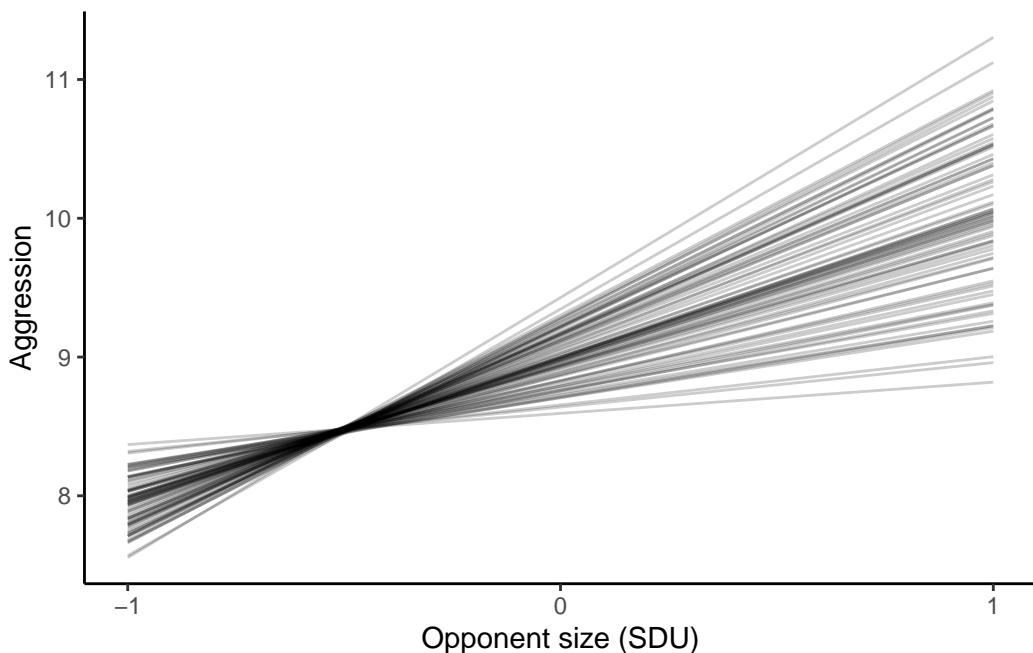
| | Estimate | SE |
|--------|-----------|-----------|
| cor_is | 0.9620736 | 0.1773965 |

```
pred_rr_asr <- as.data.frame(predict(rr_asr,
  classify = "opp_size:ID",
  levels = list(
    "opp_size" =
      c(opp_size = -1:1)
  )
)$pvals)
```

ASReml Version 4.2 11/10/2024 13:38:24

| | LogLik | Sigma2 | DF | wall |
|---|-----------|-----------|-----|----------|
| 1 | -100.6821 | 0.4281680 | 475 | 13:38:24 |
| 2 | -100.6821 | 0.4281680 | 475 | 13:38:24 |
| 3 | -100.6821 | 0.4281680 | 475 | 13:38:24 |

```
p_rr <- ggplot(pred_rr_asr, aes(  
  x = opp_size,  
  y = predicted.value,  
  group = ID  
) +  
  geom_line(alpha = 0.2) +  
  scale_x_continuous(breaks = c(-1, 0, 1)) +  
  labs(  
    x = "Opponent size (SDU)",  
    y = "Aggression"  
) +  
  theme_classic()  
  
p_rr
```



19.2.3.3. with MCMCglmm

```

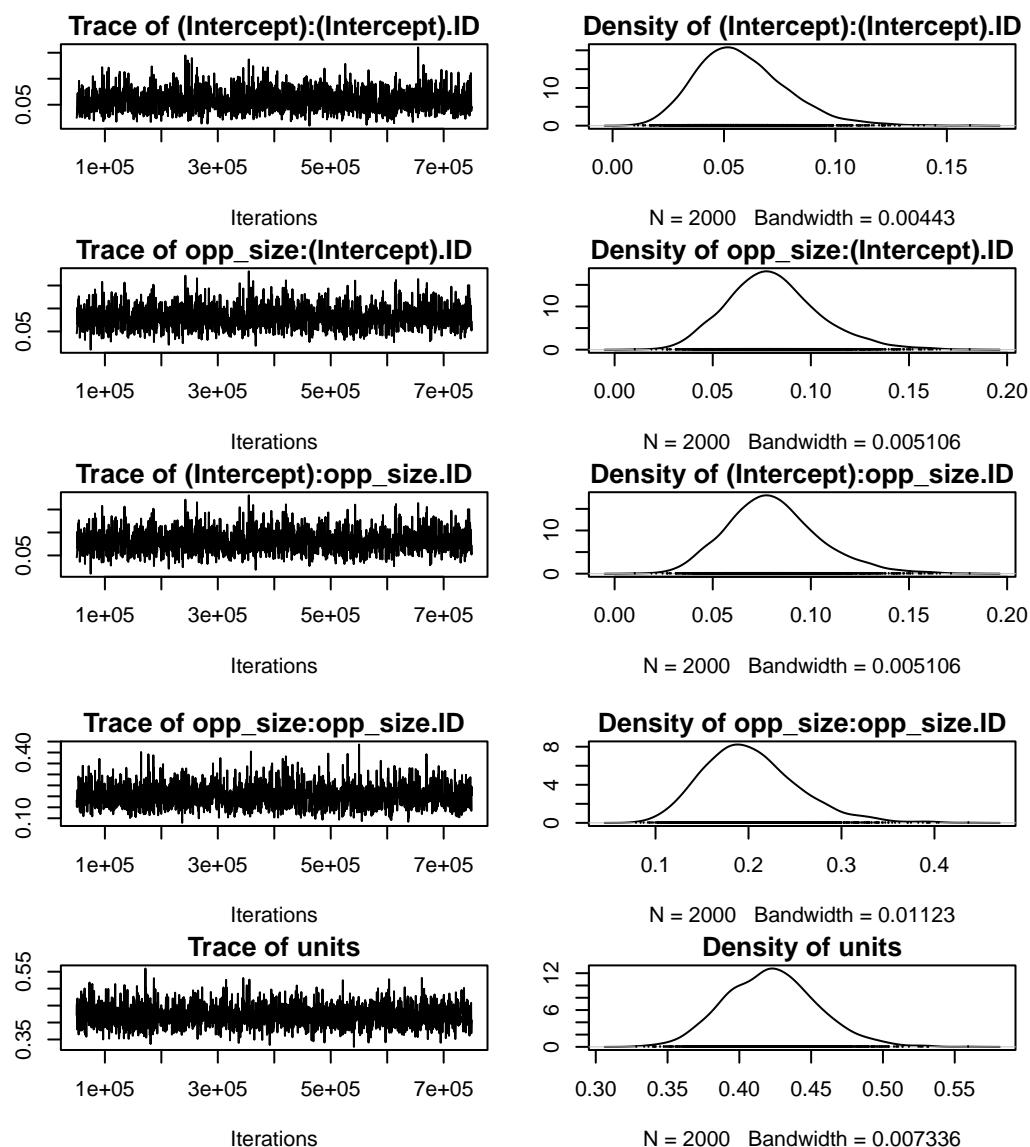
prior_RR <- list(
  R = list(V = 1, nu = 0.002),
  G = list(
    G1 = list(V = diag(2)*0.02, nu = 3,
alpha.mu = rep(0, 2),
alpha.V= diag(1000, 2, 2)))
rr_mcmc <- MCMCglmm(
  aggression ~ opp_size + assay_rep_sc + body_size_sc + block,
  random = ~ us(1 + opp_size):ID,
  rcov = ~ units,
  family = "gaussian",
  prior = prior_RR,
  nitt=750000,
  burnin=50000,
  thin=350,
  verbose = FALSE,
  data = unicorns,
  pr = TRUE,
  saveX = TRUE, saveZ = TRUE)

```

```

omar <- par()
par(mar = c(4, 2, 1.5, 2))
plot(rr_mcmc$VCV)

```



```
par(omar)
```

Warning in par(omar): graphical parameter "cin" cannot be set

Warning in par(omar): graphical parameter "cra" cannot be set

Warning in par(omar): graphical parameter "csi" cannot be set

```
Warning in par(omar): graphical parameter "cxy" cannot be set
```

```
Warning in par(omar): graphical parameter "din" cannot be set
```

```
Warning in par(omar): graphical parameter "page" cannot be set
```

```
posterior.mode(rr_mcmc$VCV[, "opp_size:opp_size.ID"]) # mean
```

```
var1
```

```
0.188436
```

```
HPDinterval(rr_mcmc$VCV[, "opp_size:opp_size.ID"])
```

| lower | upper |
|-------|-------|
|-------|-------|

```
var1 0.1072895 0.2998701
```

```
attr(,"Probability")
```

```
[1] 0.95
```

```
rr_cor_mcmc <- rr_mcmc$VCV[, "opp_size:(Intercept).ID"] /
```

```
(sqrt(rr_mcmc$VCV[, "(Intercept):(Intercept).ID"])) *
```

```
sqrt(rr_mcmc$VCV[, "opp_size:opp_size.ID"]))
```

```
posterior.mode(rr_cor_mcmc)
```

```
var1
```

```
0.8206534
```

```
HPDinterval(rr_cor_mcmc)
```

| lower | upper |
|-------|-------|
|-------|-------|

```
var1 0.5211717 0.9836348
```

```
attr(,"Probability")
```

```
[1] 0.95
```

```
df_rand <- cbind(unicorns,
  rr_fit = predict(rr_mcmc, marginal = NULL)
) %>%
  select(ID, opp_size, rr_fit, aggression) %>%
  group_by(ID, opp_size) %>%
  summarise(
    rr_fit = mean(rr_fit),
    aggression = mean(aggression)
) %>%
gather(
  Type, Value,
  rr_fit:aggression
)
```

`summarise()` has grouped output by 'ID'. You can override using the ` `.groups` argument.

```
# Plot separate panels for individual lines of each type
ggplot(df_rand, aes(x = opp_size, y = Value, group = ID)) +
  geom_line(alpha = 0.3) +
  scale_x_continuous(breaks = c(-1, 0, 1)) +
  theme_classic() +
  facet_grid(. ~ Type)
```

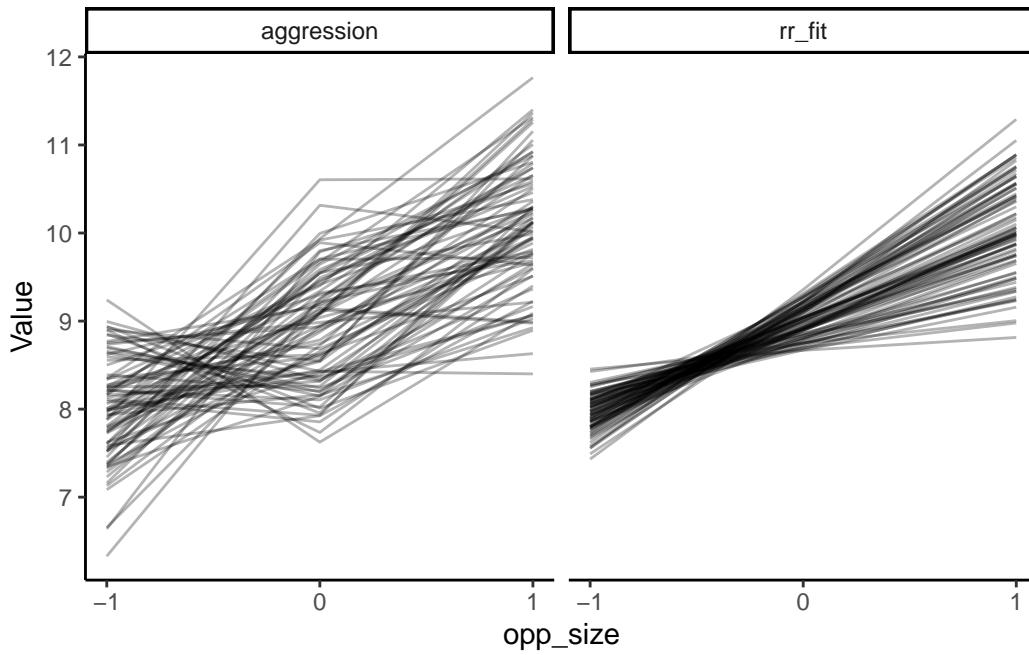


Table 19.4.: Variance estimated from random regression models using 3 different softwares

| Method | v_int | cov | v_sl | v_r |
|----------|-----------|-----------|-----------|-----------|
| lmer | 0.0504347 | 0.0945863 | 0.1916653 | 0.4281625 |
| asreml | 0.0504293 | 0.0945834 | 0.1916592 | 0.4281695 |
| MCMCglmm | 0.0562018 | 0.0859172 | 0.1884360 | 0.4201010 |

19.2.4. Character-State approach

Need to pivot to a wider format

```
unicorns_cs <- unicorns %>%
  select(ID, body_size, assay_rep, block, aggression, opp_size) %>%
  mutate(
    opp_size = recode(as.character(opp_size), "-1" = "s", "0" = "m", "1" = "l")
  ) %>%
  dplyr::rename(agg = aggression) %>%
  pivot_wider(names_from = opp_size, values_from = c(agg, assay_rep)) %>%
  mutate(
    body_size_sc = scale(body_size),
```

```

  opp_order = as.factor(paste(assay_rep_s, assay_rep_m, assay_rep_l, sep = "_"))

)
str(unicorns_cs)

```

```

tibble [160 x 11] (S3: tbl_df/tbl/data.frame)

$ ID      : Factor w/ 80 levels "ID_1","ID_10",...: 1 1 2 2 3 3 4 4 5 5 ...
$ body_size : num [1:160] 206 207 283 288 229 ...
$ block    : num [1:160] -0.5 0.5 -0.5 0.5 -0.5 0.5 -0.5 0.5 -0.5 0.5 ...
$ agg_s    : num [1:160] 7.02 8.44 7.73 8.08 8.06 8.16 8.16 8.51 7.59 6.67 ...
$ agg_l    : num [1:160] 10.67 10.51 10.81 10.67 9.77 ...
$ agg_m    : num [1:160] 10.22 8.95 9.43 9.46 7.63 ...
$ assay_rep_s : int [1:160] 1 3 2 2 1 1 3 3 1 1 ...
$ assay_rep_l : int [1:160] 2 2 1 1 2 2 2 1 2 2 ...
$ assay_rep_m : int [1:160] 3 1 3 3 3 3 1 2 3 3 ...
$ body_size_sc: num [1:160, 1] -1.504 -1.456 0.988 1.143 -0.76 ...
..- attr(*, "scaled:center")= num 253
..- attr(*, "scaled:scale")= num 31.1
$ opp_order   : Factor w/ 6 levels "1_2_3","1_3_2",...: 2 5 4 4 2 2 5 6 2 2 ...

```

```
head(unicorns_cs)
```

| ID | body_size | block | agg_s | agg_l | agg_m | say_rep_s | say_rep_l | say_rep_m | body_size_sc | as- | as- | as- | opp_or- |
|-------|-----------|-------|-------|-------|-------|-----------|-----------|-----------|--------------|--------------|--------------|--------------|---------------------|
| | | | | | | | | | | as-
rep_s | as-
rep_l | as-
rep_m | body_size_sc
der |
| ID_1 | 205.8 | -0.5 | 7.02 | 10.67 | 10.22 | | 1 | 2 | 3 | -1.5038017 | 1_3_2 | | |
| ID_1 | 207.3 | 0.5 | 8.44 | 10.51 | 8.95 | | 3 | 2 | 1 | -1.4555029 | 3_1_2 | | |
| ID_10 | 283.2 | -0.5 | 7.73 | 10.81 | 9.43 | | 2 | 1 | 3 | 0.9884138 | 2_3_1 | | |
| ID_10 | 288.0 | 0.5 | 8.08 | 10.67 | 9.46 | | 2 | 1 | 3 | 1.1429698 | 2_3_1 | | |
| ID_11 | 228.9 | -0.5 | 8.06 | 9.77 | 7.63 | | 1 | 2 | 3 | -0.7600009 | 1_3_2 | | |
| ID_11 | 236.2 | 0.5 | 8.16 | 10.84 | 8.23 | | 1 | 2 | 3 | -0.5249470 | 1_3_2 | | |

```

cs_asr <- asreml(
  cbind(agg_s, agg_m, agg_l) ~ trait + trait:body_size_sc +
  trait:block +

```

```

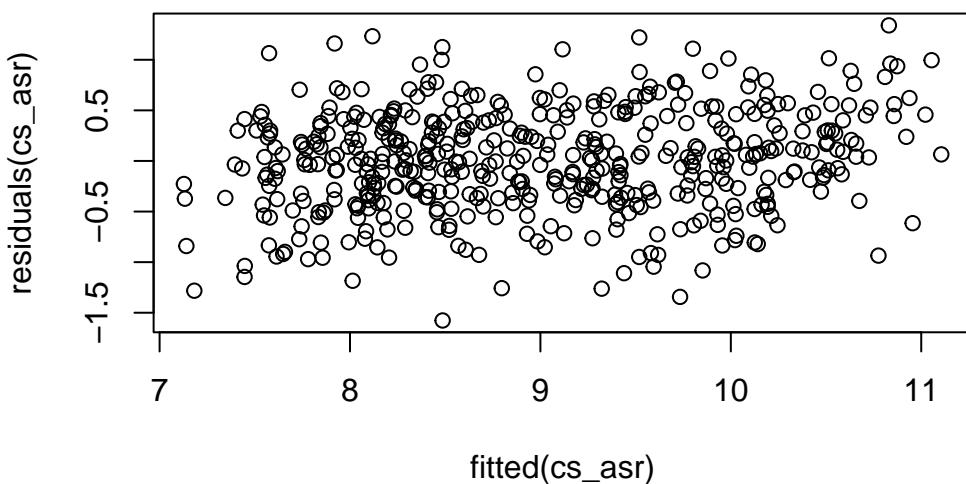
trait:opp_order,
random =~ ID:us(trait),
residual =~ units:us(trait),
data = unicorns_cs,
maxiter = 200
)

```

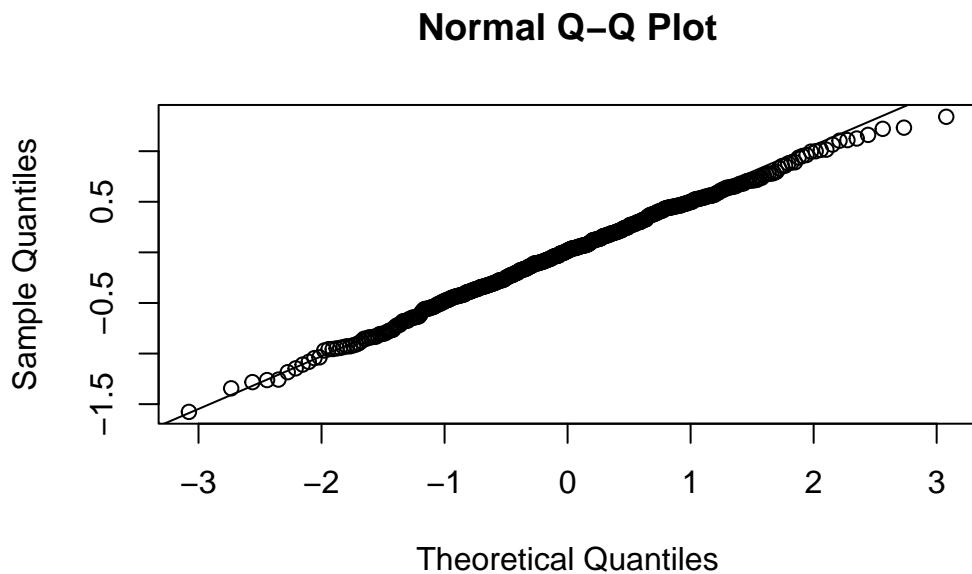
ASReml Version 4.2 11/10/2024 13:40:42

| | LogLik | Sigma2 | DF | wall |
|---|-----------|--------|-----|----------|
| 1 | -150.1721 | 1.0 | 456 | 13:40:42 |
| 2 | -129.6584 | 1.0 | 456 | 13:40:42 |
| 3 | -110.4540 | 1.0 | 456 | 13:40:42 |
| 4 | -101.8792 | 1.0 | 456 | 13:40:42 |
| 5 | -100.0917 | 1.0 | 456 | 13:40:43 |
| 6 | -100.0545 | 1.0 | 456 | 13:40:43 |
| 7 | -100.0544 | 1.0 | 456 | 13:40:43 |

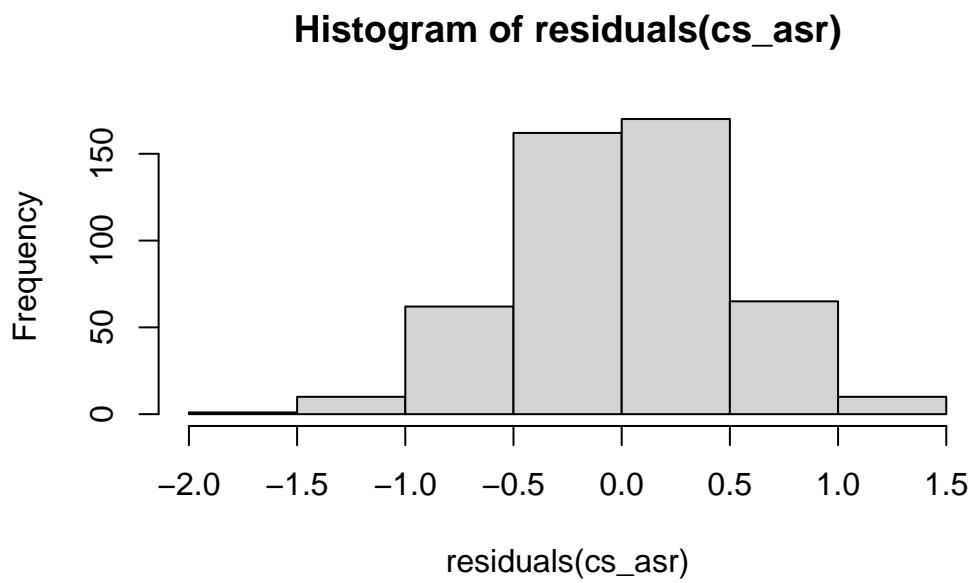
```
plot(residuals(cs_asr) ~ fitted(cs_asr))
```



```
qqnorm(residuals(cs_asr))
qqline(residuals(cs_asr))
```



```
hist(residuals(cs_asr))
```



```
summary(cs_asr, all = T)$coef.fixed
```

NULL

```
wa <- wald(cs_asr, ssType = "conditional", denDF = "numeric")
```

ASReml Version 4.2 11/10/2024 13:40:43

| | LogLik | Sigma2 | DF | wall |
|---|-----------|--------|-----|----------|
| 1 | -100.0544 | 1.0 | 456 | 13:40:43 |
| 2 | -100.0544 | 1.0 | 456 | 13:40:43 |

```
attr(wa$Wald, "heading") <- NULL
```

```
wa
```

\$Wald

| | Df | denDF | F.inc | F.con | Margin | Pr |
|--------------------|----|-------|---------|---------|--------|---------|
| trait | 3 | 73.2 | 21080.0 | 21080.0 | | 0.00000 |
| trait:body_size_sc | 3 | 86.6 | 0.4 | 0.5 | B | 0.68324 |
| trait:block | 3 | 75.2 | 0.6 | 0.3 | B | 0.82418 |
| trait:opp_order | 15 | 240.5 | 1.3 | 1.3 | B | 0.23282 |

\$stratumVariances

NULL

```
summary(cs_asr)$varcomp[, c("component", "std.error")]
```

| | component | std.error |
|----------------------------|------------|-----------|
| ID:trait!trait_agg_s:agg_s | 0.1929600 | 0.0632187 |
| ID:trait!trait_agg_m:agg_s | -0.1685196 | 0.0508558 |
| ID:trait!trait_agg_m:agg_m | 0.2455944 | 0.0709632 |
| ID:trait!trait_agg_l:agg_s | -0.1519902 | 0.0566075 |
| ID:trait!trait_agg_l:agg_m | 0.1584186 | 0.0637500 |

| | component | std.error |
|-------------------------------|------------|-----------|
| ID:trait!trait_agg_l:agg_l | 0.3125481 | 0.0912517 |
| units:trait!R | 1.0000000 | NA |
| units:trait!trait_agg_s:agg_s | 0.3180900 | 0.0519814 |
| units:trait!trait_agg_m:agg_s | 0.0103624 | 0.0369548 |
| units:trait!trait_agg_m:agg_m | 0.3223799 | 0.0524829 |
| units:trait!trait_agg_l:agg_s | -0.0093117 | 0.0416845 |
| units:trait!trait_agg_l:agg_m | 0.1592405 | 0.0456931 |
| units:trait!trait_agg_l:agg_l | 0.4059421 | 0.0667970 |

```
cs_idh_asr <- asreml(
  cbind(agg_s, agg_m, agg_l) ~ trait + trait:body_size_sc +
  trait:block +
  trait:opp_order,
  random = ~ ID:idh(trait),
  residual = ~ units:us(trait),
  data = unicorns_cs,
  maxiter = 200
)
```

ASReml Version 4.2 11/10/2024 13:40:43

| | LogLik | Sigma2 | DF | wall |
|---|-----------|--------|-----|----------|
| 1 | -147.0682 | 1.0 | 456 | 13:40:43 |
| 2 | -131.2680 | 1.0 | 456 | 13:40:43 |
| 3 | -116.9080 | 1.0 | 456 | 13:40:43 |
| 4 | -110.9955 | 1.0 | 456 | 13:40:43 |
| 5 | -109.9048 | 1.0 | 456 | 13:40:43 |
| 6 | -109.8659 | 1.0 | 456 | 13:40:43 |
| 7 | -109.8626 | 1.0 | 456 | 13:40:43 |

```
pchisq(2 * (cs_asr$loglik - cs_idh_asr$loglik), 3,
lower.tail = FALSE
)
```

```
[1] 0.0002038324
```

```
vpredict(cs_asr, cor_S_M ~ V2 / (sqrt(V1) * sqrt(V3)))
```

| | Estimate | SE |
|---------|------------|-----------|
| cor_S_M | -0.7741189 | 0.1869789 |

```
vpredict(cs_asr, cor_M_L ~ V5 / (sqrt(V3) * sqrt(V6)))
```

| | Estimate | SE |
|---------|-----------|-----------|
| cor_M_L | 0.5717926 | 0.1469504 |

```
vpredict(cs_asr, cor_S_L ~ V4 / (sqrt(V1) * sqrt(V6)))
```

| | Estimate | SE |
|---------|------------|-----------|
| cor_S_L | -0.6189044 | 0.1912133 |

```
vpredict(cs_asr, prop_S ~ V1 / (V1 + V8))
```

| | Estimate | SE |
|--------|-----------|-----------|
| prop_S | 0.3775756 | 0.0995031 |

```
vpredict(cs_asr, prop_M ~ V3 / (V3 + V10))
```

| | Estimate | SE |
|--------|----------|-----------|
| prop_M | 0.432404 | 0.0934477 |

```
vpredict(cs_asr, prop_L ~ V6 / (V6 + V13))
```

| | Estimate | SE |
|--------|-----------|-----------|
| prop_L | 0.4350067 | 0.0949851 |

```

init_CS_cor1_tri <- c(
  0.999,
  0.999, 0.999,
  1, 1, 1
)
names(init_CS_cor1_tri) <- c(
  "F",
  "F", "F",
  "U", "U", "U"
)
cs_asr_cor1_tri <- asreml(
  cbind(agg_s, agg_m, agg_l) ~ trait + trait:body_size_sc +
  trait:block +
  trait:opp_order,
  random = ~ ID:corgb(trait, init = init_CS_cor1_tri),
  residual = ~ units:us(trait),
  data = unicorns_cs,
  maxiter = 500
)

```

ASReml Version 4.2 11/10/2024 13:40:43

| | LogLik | Sigma2 | DF | wall | |
|---|-----------|--------|-----|----------|-----------------|
| 1 | -228.0158 | 1.0 | 456 | 13:40:43 | (3 restrained) |
| 2 | -150.0138 | 1.0 | 456 | 13:40:43 | |
| 3 | -129.5803 | 1.0 | 456 | 13:40:43 | |
| 4 | -119.9924 | 1.0 | 456 | 13:40:43 | (1 restrained) |
| 5 | -116.9067 | 1.0 | 456 | 13:40:43 | (1 restrained) |
| 6 | -115.7721 | 1.0 | 456 | 13:40:43 | |
| 7 | -115.6466 | 1.0 | 456 | 13:40:43 | |
| 8 | -115.5882 | 1.0 | 456 | 13:40:43 | |

| | | | | |
|----|-----------|-----|-----|--------------------------|
| 9 | -115.5334 | 1.0 | 456 | 13:40:43 |
| 10 | -115.4795 | 1.0 | 456 | 13:40:43 |
| 11 | -115.4273 | 1.0 | 456 | 13:40:43 |
| 12 | -115.3777 | 1.0 | 456 | 13:40:43 |
| 13 | -115.3314 | 1.0 | 456 | 13:40:43 |
| 14 | -115.2892 | 1.0 | 456 | 13:40:43 |
| 15 | -115.2511 | 1.0 | 456 | 13:40:43 |
| 16 | -115.2174 | 1.0 | 456 | 13:40:43 |
| 17 | -115.1879 | 1.0 | 456 | 13:40:43 |
| 18 | -115.1624 | 1.0 | 456 | 13:40:43 |
| 19 | -115.1406 | 1.0 | 456 | 13:40:43 |
| 20 | -115.1221 | 1.0 | 456 | 13:40:43 |
| 21 | -115.1065 | 1.0 | 456 | 13:40:43 |
| 22 | -115.0934 | 1.0 | 456 | 13:40:43 |
| 23 | -115.0825 | 1.0 | 456 | 13:40:43 |
| 24 | -115.0731 | 1.0 | 456 | 13:40:43 (1 restrained) |
| 25 | -115.0640 | 1.0 | 456 | 13:40:43 |
| 26 | -115.0637 | 1.0 | 456 | 13:40:43 |

```
pchisq(2 * (cs_asr$loglik - cs_asr_cor1_tri$loglik),
      3,
      lower.tail = FALSE
)
```

[1] 1.367792e-06

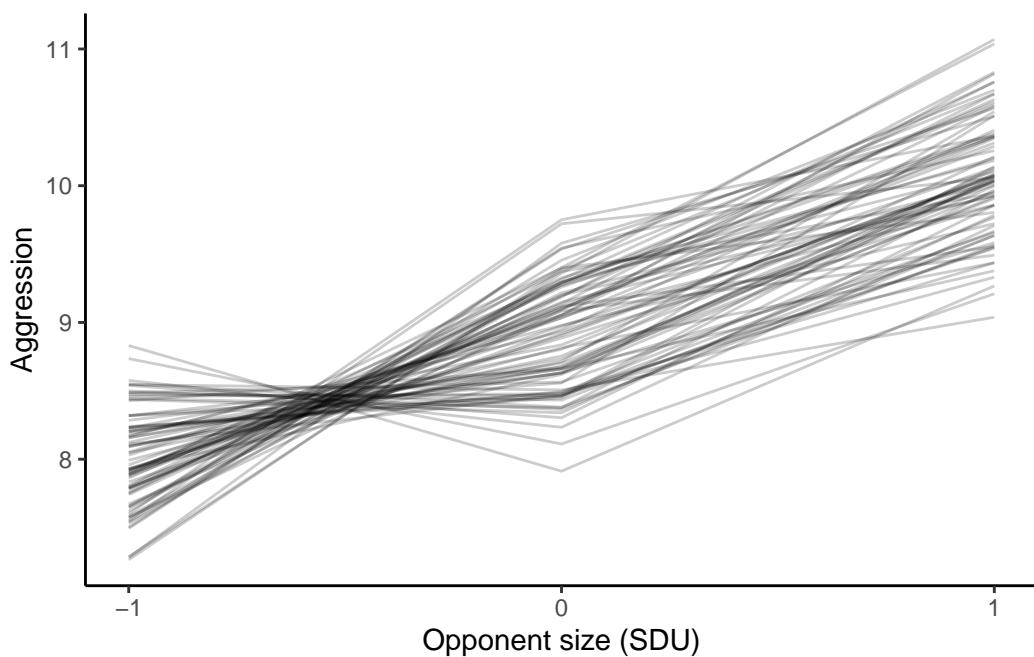
```
df_CS_pred <- as.data.frame(predict(cs_asr,
  classify = "trait:ID"
)$pvals)
```

ASReml Version 4.2 11/10/2024 13:40:43

| | LogLik | Sigma2 | DF | wall |
|---|-----------|--------|-----|----------|
| 1 | -100.0544 | 1.0 | 456 | 13:40:43 |
| 2 | -100.0544 | 1.0 | 456 | 13:40:43 |

```
3      -100.0544      1.0      456  13:40:43
```

```
# Add numeric variable for easier plotting
# of opponent size
df_CS_pred <- df_CS_pred %>%
  mutate(sizeNum = ifelse(trait == "agg_s", -1,
    ifelse(trait == "agg_m", 0, 1)
  ))
p_cs <- ggplot(df_CS_pred, aes(
  x = sizeNum,
  y = predicted.value,
  group = ID
)) +
  geom_line(alpha = 0.2) +
  scale_x_continuous(breaks = c(-1, 0, 1)) +
  labs(
    x = "Opponent size (SDU)",
    y = "Aggression"
  ) +
  theme_classic()
p_cs
```



```

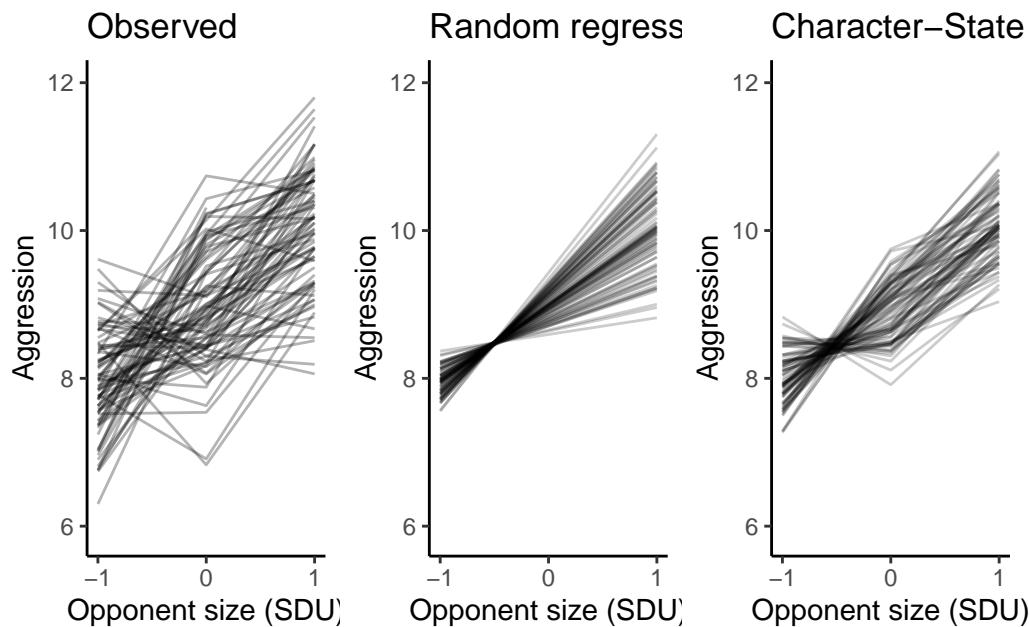
unicorns <- arrange(unicorns, opp_size, by_group = ID)

p_obs <- ggplot(unicorns[unicorns$block==0.5,], aes(x = opp_size, y = aggression, group = ID)) +
  geom_line(alpha = 0.3) +
  scale_x_continuous(breaks = c(-1, 0, 1)) +
  labs(
    x = "Opponent size (SDU)",
    y = "Aggression"
  ) +
  ggtitle("Observed") +
  ylim(5.9, 12) +
  theme_classic()

p_rr <- p_rr + ggtitle("Random regression") + ylim(5.9, 12)
p_cs <- p_cs + ggtitle("Character-State") + ylim(5.9, 12)
p_obs + p_rr + p_cs

```

Warning: Removed 2 rows containing missing values or values outside the scale range
`geom_line()`).



19.2.5. From random regression to character-state

```

var_mat_asr <- function(model, var_names, pos){

  size <- length(var_names)

  v_out <- matrix(NA, ncol = size, nrow = size)

  rownames(v_out) <- var_names

  colnames(v_out) <- var_names

  v_out[upper.tri(v_out, diag = TRUE)] <- summary(model)$varcomp[pos, 1]

  v_out <- forceSymmetric(v_out, uplo = "U")

  as.matrix(v_out)

}

v_id_rr <- var_mat_asr(rr_asr, c("v_int", "v_sl"), 1:3)

knitr::kable(v_id_rr, digits = 3)

```

| | v_int | v_sl |
|-------|-------|-------|
| v_int | 0.050 | 0.095 |
| v_sl | 0.095 | 0.192 |

```

v_id_cs <- var_mat_asr(cs_asr, c("v_s", "v_m", "v_l"), 1:6)

knitr::kable(v_id_cs, digits = 3)

```

| | v_s | v_m | v_l |
|-----|--------|--------|--------|
| v_s | 0.193 | -0.169 | -0.152 |
| v_m | -0.169 | 0.246 | 0.158 |
| v_l | -0.152 | 0.158 | 0.313 |

We also need to make a second matrix, let's call it \mathbf{Q} (no particular reason, pick something else if you want). This is going to contain the values needed to turn an individual's intercept (mean) and slope (plasticity) deviations into estimates of environment-specific individual merit in a character state model.

What do we mean by this? Well if an individual i has an intercept deviation of $ID_{int(i)}$ and a slope deviation of $ID_{slp(i)}$ for a given value of the environment opp_size we might be interested in:

$$ID_i = (1 \times ID_{int(i)}) + (opp_size \times ID_{slp(i)})$$

We want to look at character states representing the three observed values of `opp_size` here so

```
Q <- as.matrix(cbind(c(1, 1, 1),
                      c(-1, 0, 1)))
```

Then we can generate our among-individual covariance matrix environment specific aggressiveness, which we can call `ID_cs_rr` by matrix multiplication:

```
ID_cs_rr<- Q %*% v_id_rr %*% t(Q)      #where t(Q) is the transpose of Q
                                              #and %*% is matrix multiplication
```

```
ID_cs_rr  #rows and columns correspond to aggressiveness at opp_size=-1,0,1 in that order
```

```
[,1]          [,2]          [,3]
[1,]  0.05292184 -0.04415404 -0.1412299
[2,] -0.04415404  0.05042932  0.1450127
[3,] -0.14122993  0.14501267  0.4312553
```

```
cov2cor(ID_cs_rr)  #Converting to a correlation scale
```

```
[,1]          [,2]          [,3]
[1,]  1.0000000 -0.8546956 -0.9348503
[2,] -0.8546956  1.0000000  0.9833253
[3,] -0.9348503  0.9833253  1.0000000
```

```
cov2cor(v_id_cs)
```

```
v_s          v_m          v_l
v_s  1.0000000 -0.7741189 -0.6189044
v_m -0.7741189  1.0000000  0.5717926
v_l -0.6189044  0.5717926  1.0000000
```

19.2.6. Conclusions

19.2.7. Happy multivariate models



Figure 19.3.: A female blue dragon of the West

Chapter 20

Multivariate mixed models

20.1. Lecture

Amazing beasties and crazy animals



Figure 20.1.: Dream pet dragon

add a comparison of lrt

20.2. Practical

In this practical, we have collected data on the amazing blue dragon of the East that roam the sky at night.

We will use two different to fit more complex models that are not possible with `lmer()` from `lme4` (Bates et al. 2015). We will use:

- **asreml-R** which is a commercial software developed by VSNi (The VSNi Team 2023). ASReml fit models using a maximum likelihood approach, is quite flexible and fast.
- **MCMCglmm** which is free and open-source and fit model using a Bayesian approach (Hadfield 2010). It is super flexible and allow to fit a wide diversity of distribution.

The aims of the practical are to learn:

- How to phrase questions of interest in terms of variances and covariances (or derived correlations or regressions);
- How to incorporate more advanced model structures, such as:
 - Fixed effects that apply only to a subset of the response traits;
 - Traits which are measured a different number of times (e.g., repeated measures of behaviour and a single value of breeding success);
- Hypothesis testing using likelihood ratio tests.

20.2.1. R packages needed

First we load required libraries

```
library(lmerTest)
library(tidyverse)
library(asreml)
library(MCMCglmm)
library(nadiv)
```

20.2.2. The blue dragon of the East

For this practical, we have collected data on the amazing blue dragon of the East that roam the sky at night.



Figure 20.2.: Blue dragon male

We tagged all dragons individually when they hatch from their eggs. Here, we concentrate on female dragon that produce a single clutch of eggs per mating seasons. Adult female blue dragons need to explore vast amount of land to find a compatible male. We thus hypothesized that maximum flight speed as well as exploration are key traits to determine fitness. We were able to obtain repeated measures of flying speed and exploration on 80 adult females during one mating season and also measure the number of egg laid at the end of the season.

Each female was captured 4 times during the season and each time we measured the maximum flying speed using a wind tunnel and exploration using a openfield test.

The data frame has 6 variables:

- ID: Individual identity
- assay_rep: the repeat number of the behavioural assay
- max_speed: maximum flying speed
- exploration:
- eggs: measure of reproductive success measured only once per individual
- body_size: individual body size measured on the day of the test

```
df_dragons <- read.csv("data/dragons.csv")
str(df_dragons)
```

```
'data.frame': 320 obs. of 6 variables:
 $ ID       : chr  "S_1" "S_1" "S_1" "S_1" ...
 $ assay_rep : int  1 2 3 4 1 2 3 4 1 2 ...
 $ max_speed : num  58.7 57.9 64.3 61.4 65.5 ...
 ...
```

```
$ exploration: num 126 125 127 127 125 ...
$ eggs       : int 39 NA NA NA 56 NA NA NA 51 NA ...
$ body_size  : num 21.7 21.5 21.3 20.8 25.7 ...
```

To help with convergence of the model, and also help with parameter interpretation, we will first scale our covariates.

```
df_dragons <- df_dragons %>%
  mutate(
    body_size_sc = scale(body_size),
    assay_rep_sc = scale(assay_rep, scale = FALSE)
  )
```

20.2.3. Multiple univariate models

We first use the `lme4`  to determine the proportion of phenotypic variation (adjusted for fixed effects) that is due to differences among individuals, separately for each trait with repeated measures.

20.2.3.1. Flying speed

Our model includes fixed effects of the assay repeat number (centred) and individual body size (centred and scaled to standard deviation units), as we wish to control for any systematic effects of these variables on individual behaviour. Be aware that controlling variables are at your discretion — for example, while we want to characterise among-individual variance in flying speed after controlling for size effects in this study, others may wish to characterise among-individual variance in flying speed without such control. Using techniques shown later in the practical, it would be entirely possible to characterise both among-individual variance in flying speed and in size, and the among-individual covariance between these measurements.

```
lmer_f <- lmer(max_speed ~ assay_rep_sc + body_size_sc + (1 | ID),
  data = df_dragons
)
par(mfrow = c(1, 3))
plot(resid(lmer_f, type = "pearson") ~ fitted(lmer_f))
qqnorm(residuals(lmer_f))
qqline(residuals(lmer_f))
hist(residuals(lmer_f))
```

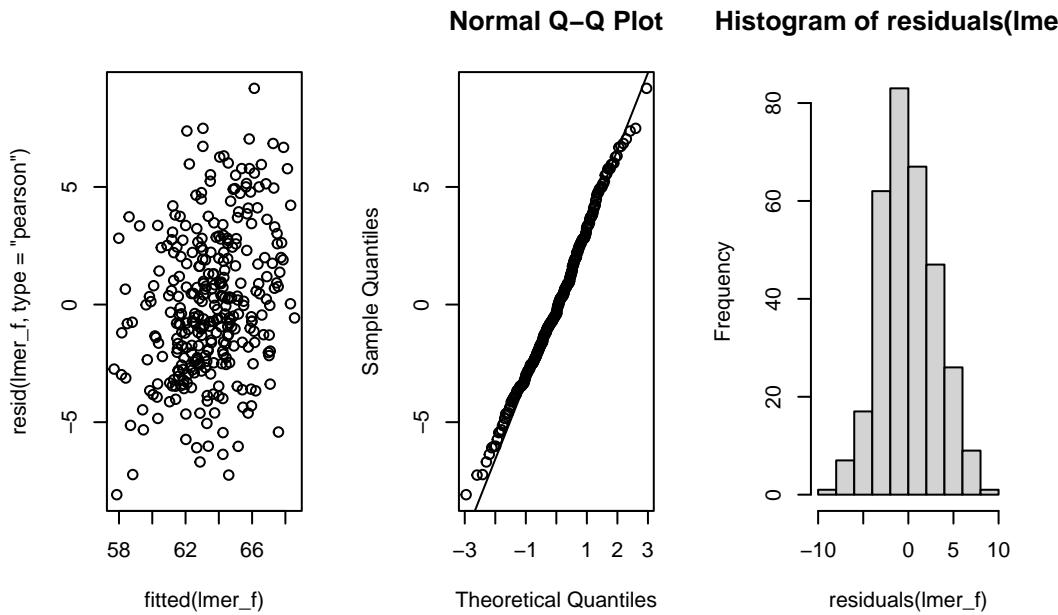


Figure 20.3.: Checking assumptions of model lmer_f

```
summary(lmer_f)
```

```
Linear mixed model fit by REML. t-tests use Satterthwaite's method [  
lmerModLmerTest]  
Formula: max_speed ~ assay_rep_sc + body_size_sc + (1 | ID)  
Data: df_dragons
```

REML criterion at convergence: 1791.4

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|---------|---------|---------|--------|--------|
| -2.3645 | -0.6496 | -0.1154 | 0.6463 | 2.6894 |

Random effects:

| Groups | Name | Variance | Std.Dev. |
|--------|-------------|----------|----------|
| ID | (Intercept) | 6.951 | 2.636 |
| | Residual | 11.682 | 3.418 |

Number of obs: 320, groups: ID, 80

Fixed effects:

| | Estimate | Std. Error | df | t value | Pr(> t) | | | | | | |
|----------------|----------|------------|----------|---------|------------|-----|------|------|-----|-----|---|
| (Intercept) | 63.5344 | 0.3513 | 78.0954 | 180.870 | <2e-16 *** | | | | | | |
| assay_rep_sc | -0.1519 | 0.1709 | 238.9807 | -0.889 | 0.375 | | | | | | |
| body_size_sc | 0.4468 | 0.3445 | 88.0328 | 1.297 | 0.198 | | | | | | |
| --- | | | | | | | | | | | |
| Signif. codes: | 0 | '***' | 0.001 | '**' | 0.01 | '*' | 0.05 | '. ' | 0.1 | ' ' | 1 |

Correlation of Fixed Effects:

| (Intr) | assy__ |
|-------------|--------------|
| assay_rp_sc | 0.000 |
| body_siz_sc | 0.000 -0.002 |

Having examined diagnostic plots of the model fit, we can check the model summary. We are interested in the random effects section of the lme4 model output (specifically the variance component — note that the standard deviation here is simply the square root of the variance). Evidence for ‘animal personality’ (or ‘consistent among-individual differences in behaviour’) in the literature is largely taken from the repeatability of behavioral traits: we can compute this repeatability (also known as the intraclass correlation coefficient) by dividing the variance in the trait due to differences among individuals (V_{ID}) by the total phenotypic variance after accounting for the fixed effects ($V_{ID} + V_{residual}$).

```
rep_flying <- as.data.frame(VarCorr(lmer_f)) %>%  
  select(grp, vcov) %>%  
  spread(grp, vcov) %>%  
  mutate(repeatability = ID / (ID + Residual))  
rep_flying
```

Table 20.1.: Variance components and repeatability for the maximum flying speed of blue dragons

| ID | Residual | repeatability |
|-------|----------|---------------|
| 6.951 | 11.682 | 0.373 |

So we can see that 37.31% of the phenotypic variation in boldness (having controlled for body size and assay repeat number) is due to differences among individuals.

20.2.3.2. Exploration

```
lmer_e <- lmer(exploration ~ assay_rep_sc + body_size_sc + (1 | ID),
  data = df_dragons
)
par(mfrow = c(1, 3))
plot(resid(lmer_e, type = "pearson") ~ fitted(lmer_e))
qqnorm(residuals(lmer_e))
qqline(residuals(lmer_e))
hist(residuals(lmer_e))
```

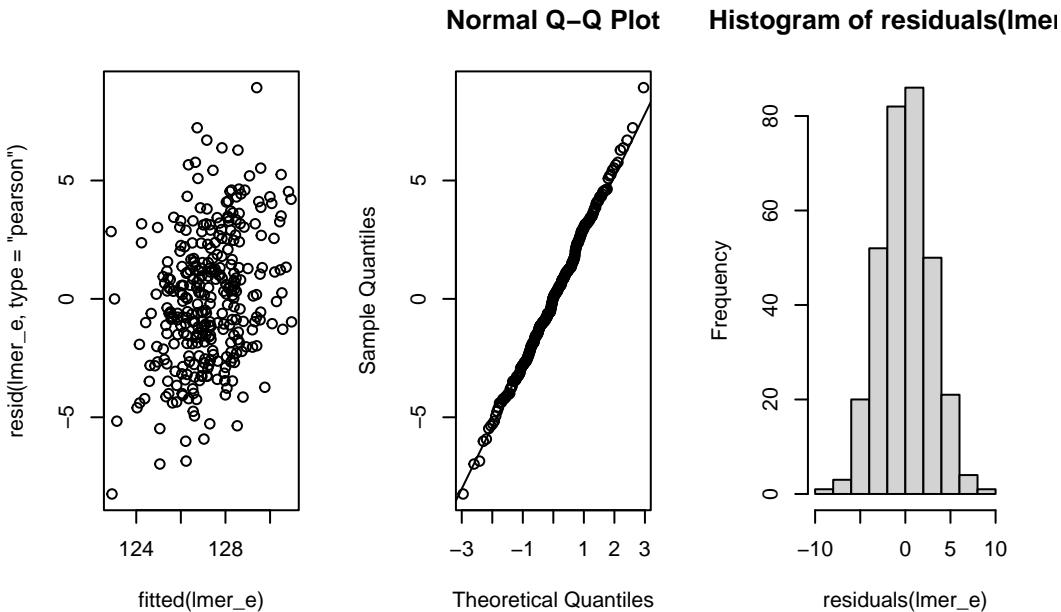


Figure 20.4.: Checking assumptions of model lmer_e

```
summary(lmer_e)
```

```
Linear mixed model fit by REML. t-tests use Satterthwaite's method [  
lmerModLmerTest]  
Formula: exploration ~ assay_rep_sc + body_size_sc + (1 | ID)  
Data: df_dragons
```

REML criterion at convergence: 1691.2

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -2.73290 | -0.62520 | 0.01635 | 0.55523 | 2.95896 |

Random effects:

| Groups | Name | Variance | Std.Dev. |
|--------|-------------|----------|----------|
| ID | (Intercept) | 3.623 | 1.903 |
| | Residual | 9.091 | 3.015 |

Number of obs: 320, groups: ID, 80

Fixed effects:

| | Estimate | Std. Error | df | t value | Pr(> t) |
|--------------|-----------|------------|-----------|---------|------------|
| (Intercept) | 127.22524 | 0.27148 | 78.08871 | 468.639 | <2e-16 *** |
| assay_rp_sc | -0.07811 | 0.15076 | 238.99943 | -0.518 | 0.605 |
| body_size_sc | 0.26114 | 0.26806 | 85.68180 | 0.974 | 0.333 |

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Correlation of Fixed Effects:

| (Intr) | assy__ |
|-------------|--------------|
| assay_rp_sc | 0.000 |
| body_siz_sc | 0.000 -0.002 |

So the model looks good and we can see our estimates for both fixed and random effects. We can now estimate the repeatability of exploration.

```
rep_expl <- as.data.frame(VarCorr(lmer_e)) %>%  
  select(grp, vcov) %>%  
  spread(grp, vcov) %>%  
  mutate(repeatability = ID / (ID + Residual))  
rep_expl
```

Table 20.2.: Variance components and repeatability for exploration behaviour of blue dragons

| ID | Residual | repeatability |
|-------|----------|---------------|
| 3.623 | 9.091 | 0.285 |

Both of traits of interest are repeatable at the among-individual level. So, the remaining question is estimating the relation between these two traits. Are individuals that are consistently faster than average also more exploratory than average (and vice versa)?

20.2.3.3. Correlation using BLUPs

Using BLUPs to estimate correlations between traits or to further investigate biological associations can lead to spurious results and anticonservative hypothesis tests and narrow confidence intervals. Hadfield et al. (2010) discuss the problem as well as present some alternative method to avoid the problem using Bayesian methods. However, it is always preferable to use multivariate models when possible.

We need to create a data frame that contain the BLUPs from both univariate models.

```
df_blups_fe <- merge(
  as.data.frame(ranef(lmer_f)),
  as.data.frame(ranef(lmer_e)),
  by = "grp"
) %>%
  mutate(
    speed = condval.x,
    exploration = condval.y
)
```

We can now test the correlation among-individual between flying speed and exploration.

```
(cor_blups <- with(df_blups_fe, cor.test(speed, exploration)))
```

Pearson's product-moment correlation

```

data: speed and exploration
t = 3.2131, df = 78, p-value = 0.00191
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
0.1320924 0.5223645
sample estimates:
cor
0.3418867

```

```

ggplot(df_blups_fe, aes(x = exploration, y = speed)) +
  geom_point() +
  labs(xlab = "Exploration (BLUP)", ylab = "Flying speed (BLUP)") +
  theme_classic()

```

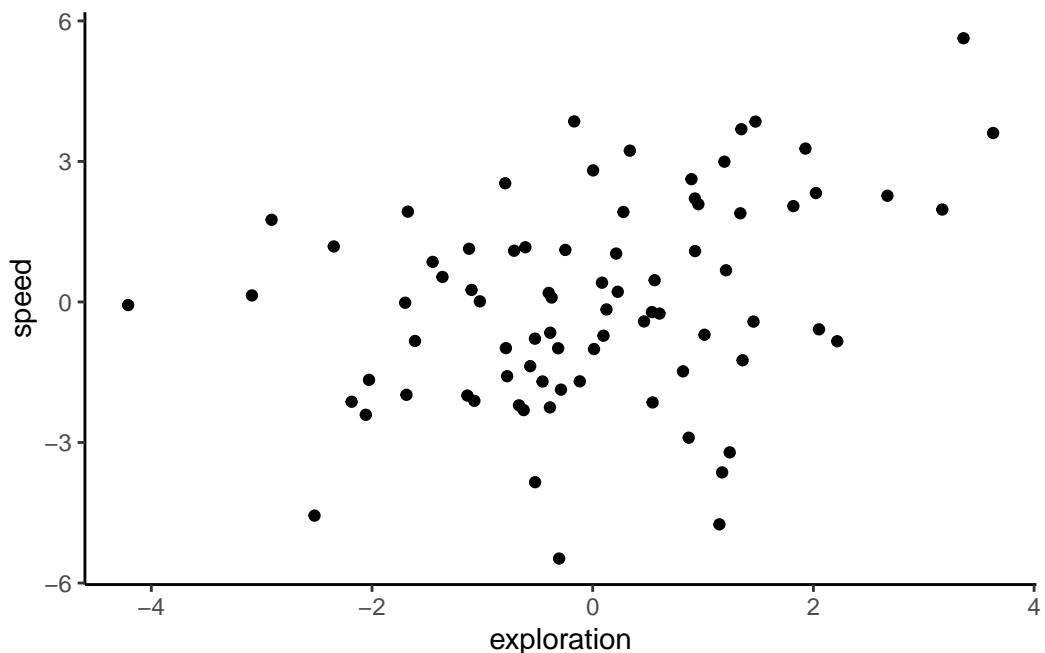


Figure 20.5.: Relation between exploration and flying speed using BLUPs from univariate models

As you can see, we get a positive correlation with a very small p-value ($P = 0.00191$), indicating that these traits are involved in a behavioural syndrome. While the correlation itself is fairly weak ($r = 0.342$), it appears to be highly significant, and suggests that individuals that are faster than average also tend to be more exploratory than average. However, as discussed in Hadfield et al. (2010) and Houslay and Wilson (2017), using BLUPs in this way leads to anticonservative significance tests. This is because the error inherent in their prediction is not carried forward from

the lmer models to the subsequent analysis (in this case, a correlation test). To illustrate this point quickly, below we plot the individual estimates along with their associated standard errors.

```
ggplot(df_blups_fe, aes(x = exploration, y = speed)) +
  geom_point() +
  geom_linerange(aes(
    xmin = exploration - condstd.x,
    xmax = exploration + condstd.x
  )) +
  geom_linerange(aes(
    ymin = speed - condstd.y,
    ymax = speed + condstd.y
  )) +
  labs(
    xlab = "Exploration (BLUP +/- SE)",
    ylab = "Flying speed (BLUP +/- SE)"
  ) +
  theme_classic()
```

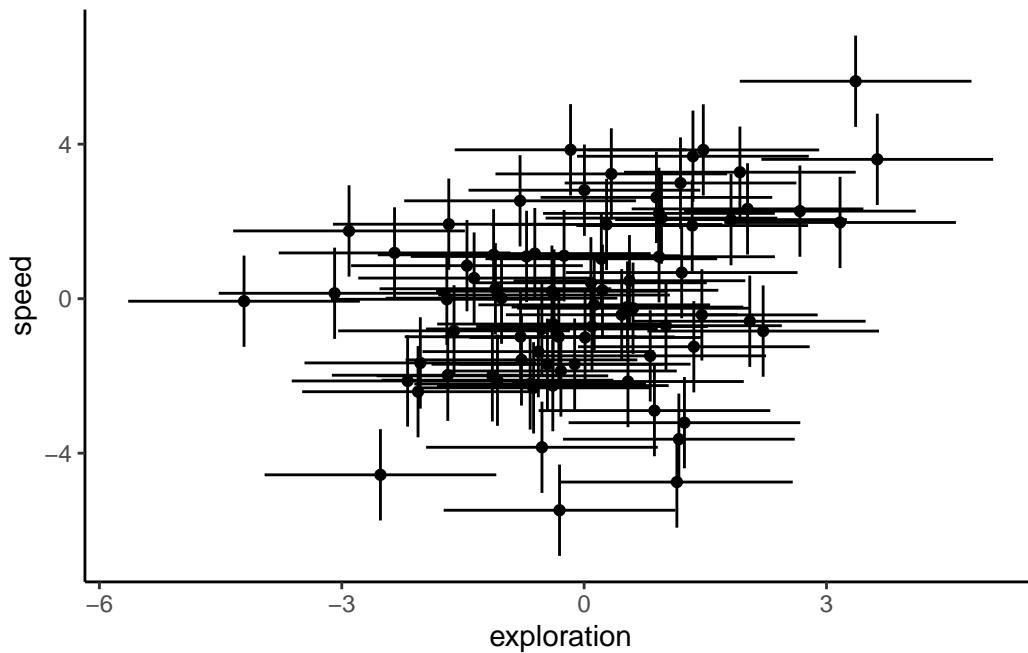


Figure 20.6.: Relation between exploration and flying speed using BLUPs from univariate models including +/- SE as error bars

20.2.4. Multivariate approach

20.2.4.1. Based on ASRemlR

The correct approach for testing the hypothesised relation between speed and exploration uses both response variables in a two-trait ('bivariate') mixed model. This model estimates the among-individual variance for each response variable (and the covariance between them). Separate (co)variances are also fitted for the residual variation. The bivariate model also allows for fixed effects to be fitted on both response variables. We set up our model using the `asreml` function call, with our bivariate response variable being `exploration` and `flying speed` bound together using `cbind`. You will also note that we scale our response variables, meaning that each is centred at their mean value and standardised to units of 1 standard deviation. This is not essential, but simply makes it easier for the model to be fit. Scaling the response variables also aids our understanding of the output, as both flying speed and exploration are now on the same scale.

`asreml` can be a bit specific sometime and random effects should absolutely be `factor` and not `character` or `integer`

```
df_dragons <- df_dragons %>%
  mutate(
    ID = as.factor(ID),
    speed_sc = scale(max_speed),
    exploration_sc = scale(exploration)
  )

asr_us <- asreml(
  cbind(speed_sc, exploration_sc) ~ trait +
  trait:assay_rep_sc + trait:body_size_sc,
  random = ~ ID:us(trait),
  residual = ~ units:us(trait),
  data = df_dragons,
  maxiter = 100
)
```

ASReml Version 4.2 11/10/2024 13:44:46

| LogLik | Sigma2 | DF | wall |
|--------|--------|----|------|
|--------|--------|----|------|

| | | | | |
|---|-----------|-----|-----|----------|
| 1 | -333.1053 | 1.0 | 634 | 13:44:46 |
| 2 | -303.6372 | 1.0 | 634 | 13:44:46 |
| 3 | -274.8492 | 1.0 | 634 | 13:44:46 |
| 4 | -260.2431 | 1.0 | 634 | 13:44:46 |
| 5 | -256.1178 | 1.0 | 634 | 13:44:46 |
| 6 | -255.8906 | 1.0 | 634 | 13:44:46 |
| 7 | -255.8893 | 1.0 | 634 | 13:44:46 |

On the right hand side of our model formula, we use the `trait` keyword to specify that this is a multivariate model — `trait` itself tells the model to give us the intercept for each trait. We then interact `trait` with the fixed effects, `assay_rep_sc` and `body_size_sc`, so that we get estimates for the effect of these variables on each of the 2 traits. The random effects structure starts with the random effects, where we tell the model to fit an *unstructured* (us) covariance matrix for the grouping variable ID. This means that the variance in exploration due to differences among individuals, the variance in boldness due to differences among individuals, and the covariance between these variances will be estimated. Next, we set a structure for the residual variation (`residual`), which is also sometimes known as the *within-individual variation*. As we have repeated measures for both traits at the individual level, we also set an *unstructured* covariance matrix, which estimates the residual variance for each trait and also allows the residuals to covary across the two traits. Finally, we provide the name of the data frame, and a maximum number of iterations for ASReml to attempt to fit the model. After the model has been fit by ASReml, we can check the fit using the same type of model diagnostic plots as we use for `lme4`:

```
par(mfrow = c(1, 3))
plot(residuals(asr_us) ~ fitted(asr_us))
qqnorm(residuals(asr_us))
qqline(residuals(asr_us))
hist(residuals(asr_us))
```

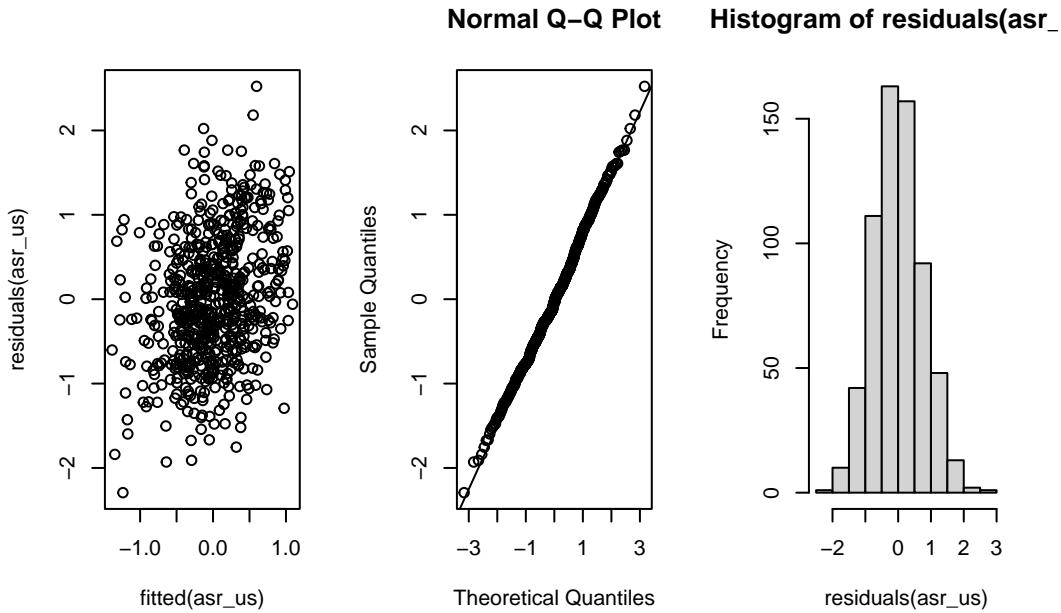


Figure 20.7.: Checking assumptions of model asr_us

The summary part of the ASReml model fit contains a large amount of information, so it is best to look only at certain parts of it at a single time. While we are not particularly interested in the fixed effects for current purposes, you can inspect these using the following code to check whether there were any large effects of assay repeat or body size on either trait:

```
summary(asr_us, coef = TRUE)$coef.fixed
```

| | solution | std error | z.ratio |
|-----------------------------------|---------------|------------|---------------|
| trait_speed_sc | -7.150609e-17 | 0.08140684 | -8.783795e-16 |
| trait_exploration_sc | -2.138998e-16 | 0.07631479 | -2.802861e-15 |
| trait_speed_sc:assay_rep_sc | -3.521261e-02 | 0.03960492 | -8.890967e-01 |
| trait_exploration_sc:assay_rep_sc | -2.195541e-02 | 0.04238056 | -5.180538e-01 |
| trait_speed_sc:body_size_sc | 1.040579e-01 | 0.07972962 | 1.305135e+00 |
| trait_exploration_sc:body_size_sc | 7.269022e-02 | 0.07533421 | 9.649033e-01 |

```
wa <- wald(asr_us, ssType = "conditional", denDF = "numeric")
```

ASReml Version 4.2 11/10/2024 13:44:46

| LogLik | Sigma2 | DF | wall |
|--------|--------|----|------|
|--------|--------|----|------|

```
1      -255.8893      1.0    634 13:44:46
2      -255.8893      1.0    634 13:44:46
```

```
attr(wa$Wald, "heading") <- NULL
wa
```

\$Wald

| | Df | denDF | F.inc | F.con | Margin | Pr |
|--------------------|----|-------|--------|--------|--------|---------|
| trait | 2 | 77.1 | 0.0000 | 0.0000 | | 1.00000 |
| trait:assay_rep_sc | 2 | 237.9 | 0.3955 | 0.3984 | B | 0.67184 |
| trait:body_size_sc | 2 | 86.6 | 0.9871 | 0.9871 | B | 0.37679 |

\$stratumVariances

NULL

We can see that there is a separate intercept for both personality traits (no surprise that these are very close to zero, given that we mean-centred and scaled each trait before fitting the model), and an estimate of the effect of assay repeat and body size on both traits. None of these appear to be large effects, so let's move on to the more interesting parts — the random effects estimates:

```
summary(asr_us)$varcomp
```

| | component | std.error | z.ratio | bound | %ch |
|---|-----------|-----------|-----------|-------|-----|
| ID:trait!trait_speed_sc:speed_sc | 0.3733306 | 0.0860712 | 4.337461 | P | 0 |
| ID:trait!trait_exploration_sc:speed_sc | 0.0883864 | 0.0606701 | 1.456837 | P | 0 |
| ID:trait!trait_exploration_sc:exploration_sc | 0.2863101 | 0.0763725 | 3.748865 | P | 0 |
| units:trait!R | 1.0000000 | NA | NA | F | 0 |
| units:trait!trait_speed_sc:speed_sc | 0.6274169 | 0.0574028 | 10.930073 | P | 0 |
| units:trait!trait_exploration_sc:speed_sc | 0.3263211 | 0.0482917 | 6.757286 | P | 0 |
| units:trait!trait_exploration_sc:exploration_sc | 0.7184419 | 0.0657278 | 10.930563 | P | 0 |

In the above summary table, we have the among-individual (co)variances listed first (starting with ID), then the residual (or within-individual) (co)variances (starting with R). You will notice that the variance estimates here are

actually close to the `lme4` repeatability estimates, because our response variables were scaled to phenotypic standard deviations. We can also find the ‘adjusted repeatability’ (i.e., the repeatability conditional on the fixed effects) for each trait by dividing its among-individual variance estimate by the sum of its among-individual and residual variances. Here, we use the `vpredict` function to estimate the repeatability and its standard error for each trait, conditional on the effects of assay repeat and body size. For this function, we provide the name of the model object, followed by a name that we want to give the estimate being returned, and a formula for the calculation. Each ‘V’ term in the formula refers to a variance component, using its position in the model summary shown above.

```
vpredict(asr_us, rep_speed ~ V1 / (V1 + V5))
```

| | Estimate | SE |
|-----------|-----------|-----------|
| rep_speed | 0.3730518 | 0.0612403 |

```
vpredict(asr_us, rep_expl ~ V3 / (V3 + V7))
```

| | Estimate | SE |
|----------|----------|-----------|
| rep_expl | 0.284956 | 0.0611354 |

We can also use this function to calculate the estimate and standard error of the correlation from our model (co)variances. We do this by specifying the formula for the correlation:

```
(cor_fe <- vpredict(asr_us, cor_expl_speed ~ V2 / (sqrt(V1 * V3))))
```

| | Estimate | SE |
|----------------|-----------|-----------|
| cor_expl_speed | 0.2703462 | 0.1594097 |

In this case, the estimate is similar (here, slightly lower) than our correlation estimate using BLUPs. However, if we consider confidence intervals as $\pm 1.96 \text{ SE}$ around the estimate, the lower bound of the confidence interval would actually be $| \text{Estimate} | - 1.96 \text{ SE} = 0.2703462 - 0.1594097 = 0.1109365$. With confidence intervals straddling zero, we would conclude that this correlation is likely non-significant. As the use of standard errors in this way is only approximate, we should also test our hypothesis formally using likelihood ratio tests.

20.2.4.1.1. Hypothesis testing We can now test the statistical significance of this correlation directly, by fitting a second model without the among-individual covariance between our two traits, and then using a likelihood ratio test to determine whether the model with the covariance produces a better fit. Here, we use the `idh` structure for our random effects. This stands for ‘identity matrix’ (i.e., with 0s on the off-diagonals) with heterogeneous variances (i.e., the variance components for our two response traits are allowed to be different from one another). The rest of the model is identical to the previous version.

```
asr_idh <- asreml(
  cbind(speed_sc, exploration_sc) ~ trait +
  trait:assay_rep_sc + trait:body_size_sc,
  random = ~ ID:idh(trait),
  residual = ~ units:us(trait),
  data = df_dragons,
  maxiter = 100
)
```

ASReml Version 4.2 11/10/2024 13:44:47

| | LogLik | Sigma2 | DF | wall |
|---|-----------|--------|-----|----------|
| 1 | -327.0510 | 1.0 | 634 | 13:44:47 |
| 2 | -299.8739 | 1.0 | 634 | 13:44:47 |
| 3 | -273.6894 | 1.0 | 634 | 13:44:47 |
| 4 | -260.8385 | 1.0 | 634 | 13:44:47 |
| 5 | -257.3307 | 1.0 | 634 | 13:44:47 |
| 6 | -257.1202 | 1.0 | 634 | 13:44:47 |
| 7 | -257.1176 | 1.0 | 634 | 13:44:47 |

The likelihood ratio test is calculated as twice the difference between model log-likelihoods, on a single degree of freedom (the covariance term):

```
(p_biv <- pchisq(2 * (asr_us$loglik - asr_idh$loglik),
  df = 1,
  lower.tail = FALSE
))
```

[1] 0.1170385

In sharp contrast to the highly-significant P-value given by a correlation test using BLUPs, here we find no evidence for a correlation between flying speed and exploration. To better understand why BLUPs produce an anticonservative p-value in comparison to multivariate models, we should plot the correlation estimates and their confidence intervals. The confidence intervals are taken directly from the cor.test function for BLUPs, and for ASReml they are calculated as 1.96 times the standard error from the vpredict function.

```
df_cor <- data.frame(
  Method = c("ASReml", "BLUPs"),
  Correlation = c(as.numeric(cor_fe[1]), cor_blups$estimate),
  low = c(as.numeric(cor_fe[1] - 1.96 * cor_fe[2]), cor_blups$conf.int[1]),
  high = c(as.numeric(cor_fe[1] + 1.96 * cor_fe[2]), cor_blups$conf.int[2])
)
ggplot(df_cor, aes(x = Method, y = Correlation)) +
  geom_point() +
  geom_linerange(aes(ymin = low, ymax = high)) +
  ylim(-1, 1) +
  geom_hline(yintercept = 0, linetype = 2) +
  theme_classic()
```

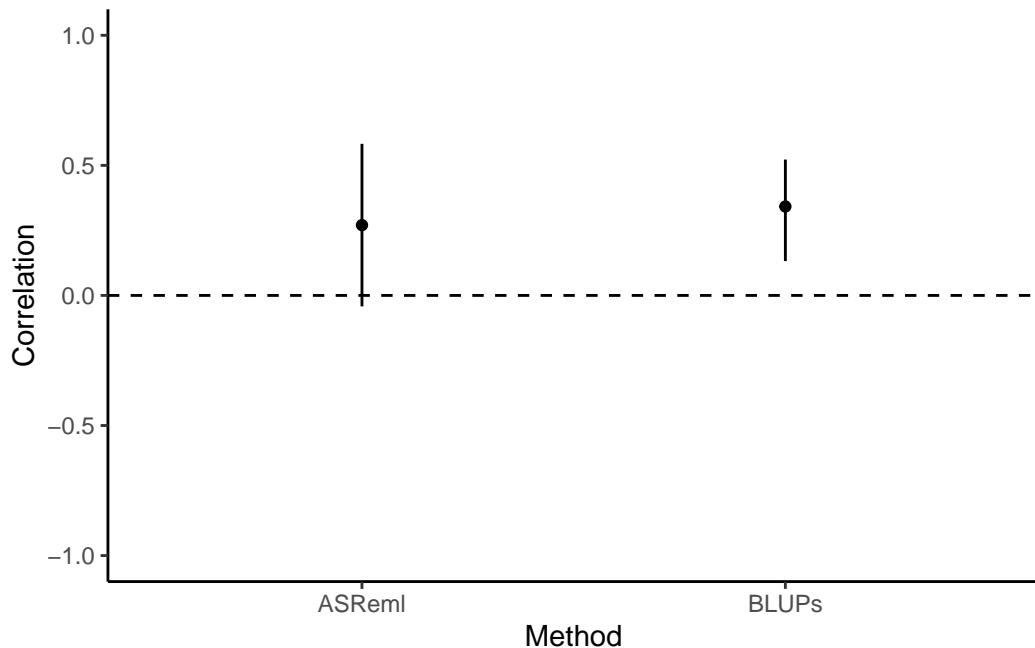


Figure 20.8.: Correlation estimates (with CI) using 2 different methods

Here we can clearly see that the BLUPs method - having failed to carry through the error around the predictions of

individual-level estimates - is anticonservative, with small confidence intervals and a correspondingly small P-value ($P = 0.00191$). Testing the syndrome directly in a bivariate model that retains all the data, by comparison, enables us to capture the true uncertainty about the estimate of the correlation. This is reflected in the larger confidence intervals and, in this case, the non-significant P-value ($P = 0.117$).

20.2.4.1.2. Conclusions To conclude, then: we found that the correlation between flying speed and exploration tends to be positive among female blue dragon. This correlation is not statistically significant, and thus does not provide strong evidence. However, inappropriate analysis of BLUP extracted from univariate models would lead to a different (erroneous) conclusion.

20.2.4.2. Using MCMCglmm

In this section I present the code needed to fit the model and explain only the specific aspect of fitting and evaluating the models with `MCMCglmm`.

To be completed. with more details

First, we need to create a ‘prior’ for our model. We recommend reading up on the use of priors (see the course notes of `MCMCglmm` Hadfield 2010); briefly, we use a parameter-expanded prior here that should be uninformative for our model. One of the model diagnostic steps that should be used later is to check that the model is robust to multiple prior specifications.

```
prior_1ex <- list(
  R = list(V = diag(2), nu = 0.002),
  G = list(G1 = list(
    V = diag(2) * 0.02, nu = 3,
    alpha.mu = rep(0, 2),
    alpha.V = diag(1000, 2, 2)
  )))
)
```

```
mcmc_us <- MCMCglmm(cbind(speed_sc, exploration_sc) ~ trait - 1 +
  trait:assay_rep_sc +
  trait:body_size_sc,
  random = ~ us(trait):ID,
```

```

rcov = ~ us(trait):units,
family = c("gaussian", "gaussian"),
prior = prior_1ex,
nitt = 420000,
burnin = 20000,
thin = 100,
verbose = FALSE,
data = df_dragons
)

```

```

omar <- par()
par(mar = c(4, 2, 1.5, 2))
plot(mcmc_us$VCV[, c(1, 2, 4)])

```

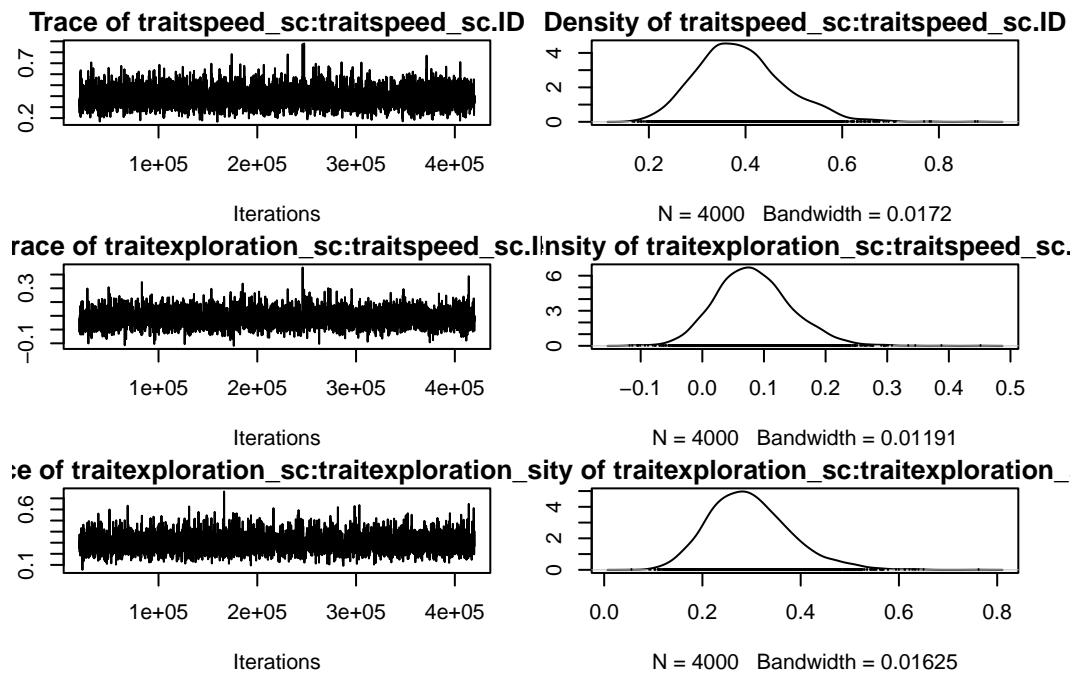
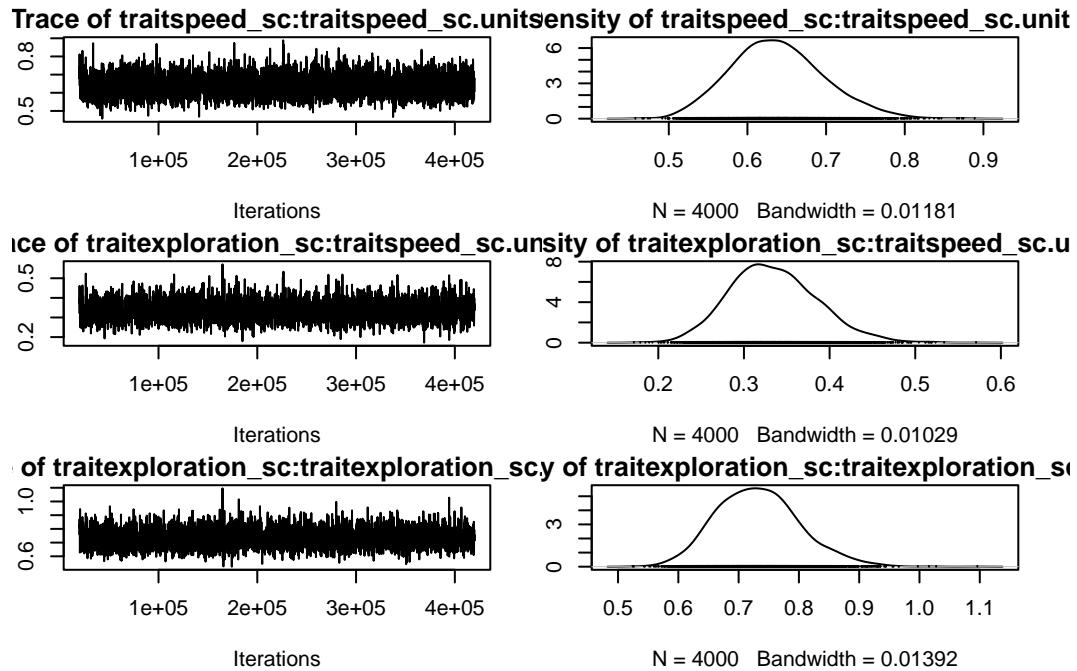


Figure 20.9.: MCMC trace and Posterior distribution of the (co)variance estimates of model mcmc_us

```
plot(mcmc_us$VCV[, c(5, 6, 8)])
```



```
par(omar)
```

```
summary(mcmc_us)
```

Iterations = 20001:419901

Thinning interval = 100

Sample size = 4000

DIC: 1596.597

G-structure: ~us(trait):ID

| | post.mean | l-95% CI | u-95% CI | eff.samp |
|--|-----------|----------|----------|----------|
| traitspeed_sc:traitspeed_sc.ID | 0.38963 | 0.23132 | 0.5751 | 4000 |
| traitexploration_sc:traitspeed_sc.ID | 0.08324 | -0.02864 | 0.2071 | 4000 |
| traitspeed_sc:traitexploration_sc.ID | 0.08324 | -0.02864 | 0.2071 | 4000 |
| traitexploration_sc:traitexploration_sc.ID | 0.29618 | 0.14854 | 0.4693 | 4000 |

R-structure: ~us(trait):units

| | post.mean | l-95% CI | u-95% CI |
|---|-----------|----------|----------|
| traitspeed_sc:traitspeed_sc.units | 0.6385 | 0.5310 | 0.7605 |
| traitexploration_sc:traitspeed_sc.units | 0.3347 | 0.2345 | 0.4325 |
| traitspeed_sc:traitexploration_sc.units | 0.3347 | 0.2345 | 0.4325 |
| traitexploration_sc:traitexploration_sc.units | 0.7325 | 0.6020 | 0.8703 |
| | eff.samp | | |
| traitspeed_sc:traitspeed_sc.units | 3759 | | |
| traitexploration_sc:traitspeed_sc.units | 4000 | | |
| traitspeed_sc:traitexploration_sc.units | 4000 | | |
| traitexploration_sc:traitexploration_sc.units | 4000 | | |

Location effects: cbind(speed_sc, exploration_sc) ~ trait - 1 + trait:assay_rep_sc + trait:body_size_sc

| | post.mean | l-95% CI | u-95% CI | eff.samp |
|----------------------------------|------------|------------|-----------|----------|
| traitspeed_sc | -0.0005335 | -0.1546478 | 0.1639974 | 4000 |
| traitexploration_sc | -0.0008756 | -0.1534328 | 0.1420720 | 4000 |
| traitspeed_sc:assay_rep_sc | -0.0350202 | -0.1152836 | 0.0439971 | 4000 |
| traitexploration_sc:assay_rep_sc | -0.0219956 | -0.1064651 | 0.0603338 | 4053 |
| traitspeed_sc:body_size_sc | 0.1052286 | -0.0477453 | 0.2668186 | 4000 |
| traitexploration_sc:body_size_sc | 0.0730324 | -0.0758619 | 0.2203125 | 4000 |
| | pMCMC | | | |
| traitspeed_sc | 0.982 | | | |
| traitexploration_sc | 0.993 | | | |
| traitspeed_sc:assay_rep_sc | 0.392 | | | |
| traitexploration_sc:assay_rep_sc | 0.613 | | | |
| traitspeed_sc:body_size_sc | 0.189 | | | |
| traitexploration_sc:body_size_sc | 0.332 | | | |

```
mcmc_prop_f <- mcmc_us$VCV[, 1] /
(mcmc_us$VCV[, 1] + mcmc_us$VCV[, 5])
plot(mcmc_prop_f)
```

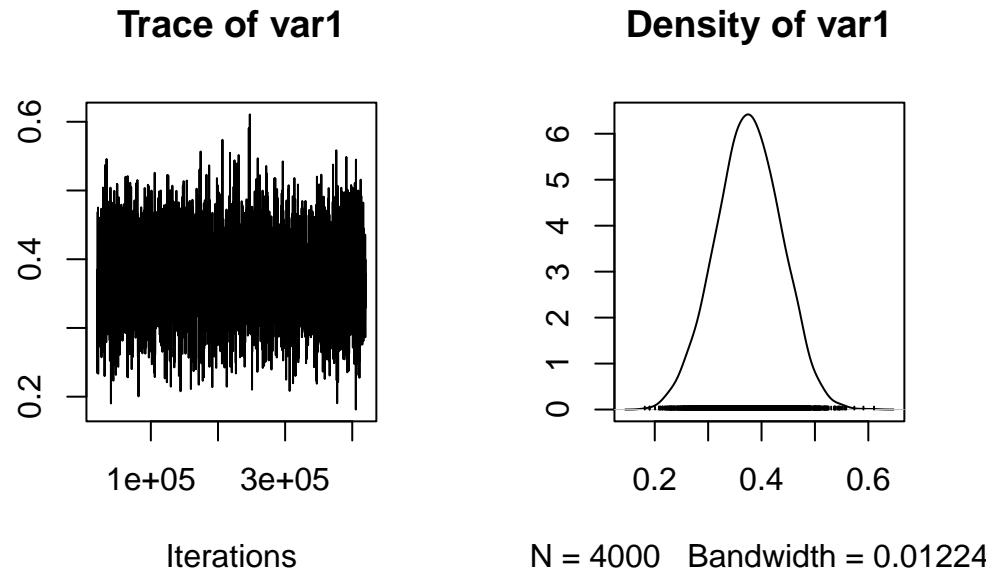


Figure 20.11.: Posterior trace and distribution of the repeatability in flying speed

```
posterior.mode(mcmc_prop_f)
```

```
var1
0.3640202
```

```
HPDinterval(mcmc_prop_f)
```

```
lower      upper
var1 0.2542384 0.4885375
attr(,"Probability")
[1] 0.95
```

```
mcmc_prop_e <- mcmc_us$VCV[, 4] /
  (mcmc_us$VCV[, 4] + mcmc_us$VCV[, 8])
plot(mcmc_prop_e)
```

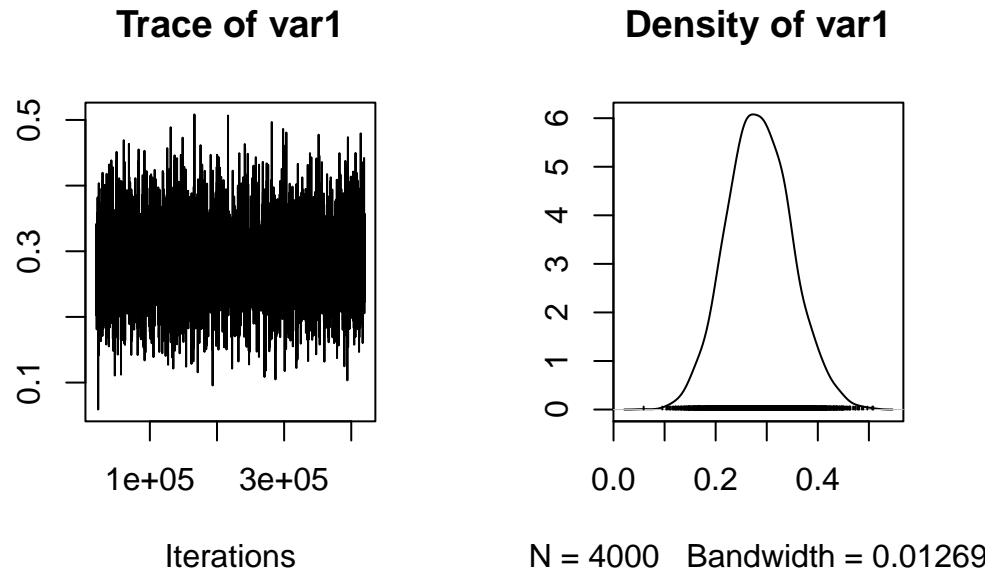


Figure 20.12.: Posterior trace and distribution of the repeatability of exploration

```
posterior.mode(mcmc_prop_e)
```

```
var1
0.2600335
```

```
HPDinterval(mcmc_prop_e)
```

| | lower | upper |
|----------------------|----------|-----------|
| var1 | 0.163859 | 0.4112182 |
| attr(,"Probability") | | |
| [1] | 0.95 | |

```
mcmc_cor_fe <- mcmc_us$VCV[, 2] /
sqrt(mcmc_us$VCV[, 1] * mcmc_us$VCV[, 4])
plot(mcmc_cor_fe)
```

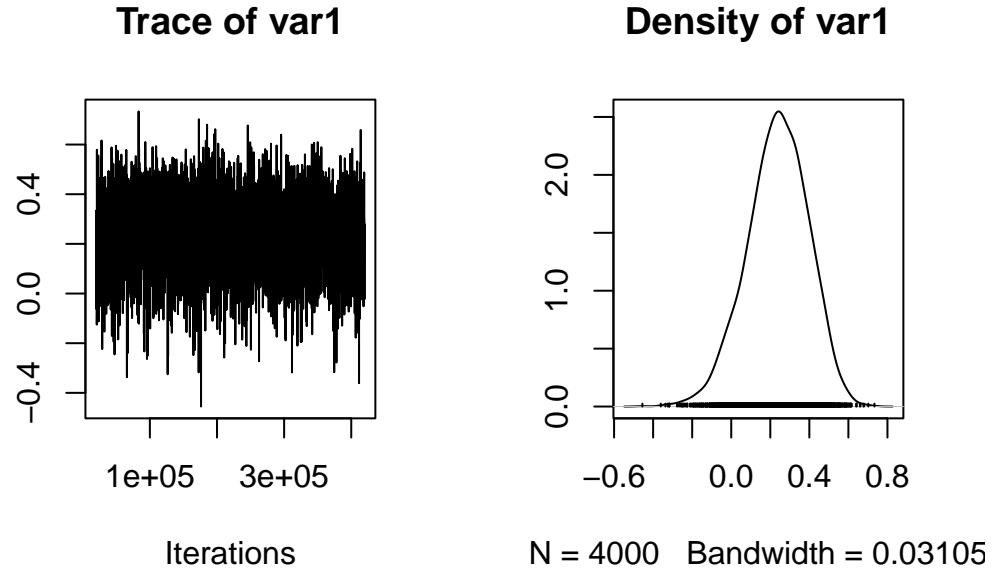


Figure 20.13.: Posterior trace and distribution of the correlation between flying speed and exploration

```
posterior.mode(mcmc_cor_fe)
```

```
var1
0.2228099
```

```
HPDinterval(mcmc_cor_fe)
```

| | lower | upper |
|------|-------------|-----------|
| var1 | -0.07008243 | 0.5308713 |

```
attr(),"Probability")
[1] 0.95
```

```
df_cor[3, 1] <- "MCMCglmm"
df_cor[3, -1] <- c(posterior.mode(mcmc_cor_fe), HPDinterval(mcmc_cor_fe))
rownames(df_cor) <- NULL

ggplot(df_cor, aes(x = Method, y = Correlation)) +
  geom_point() +
  geom_linerange(aes(ymin = low, ymax = high)) +
```

```

ylim(-1, 1) +
geom_hline(yintercept = 0, linetype = 2) +
theme_classic()

```

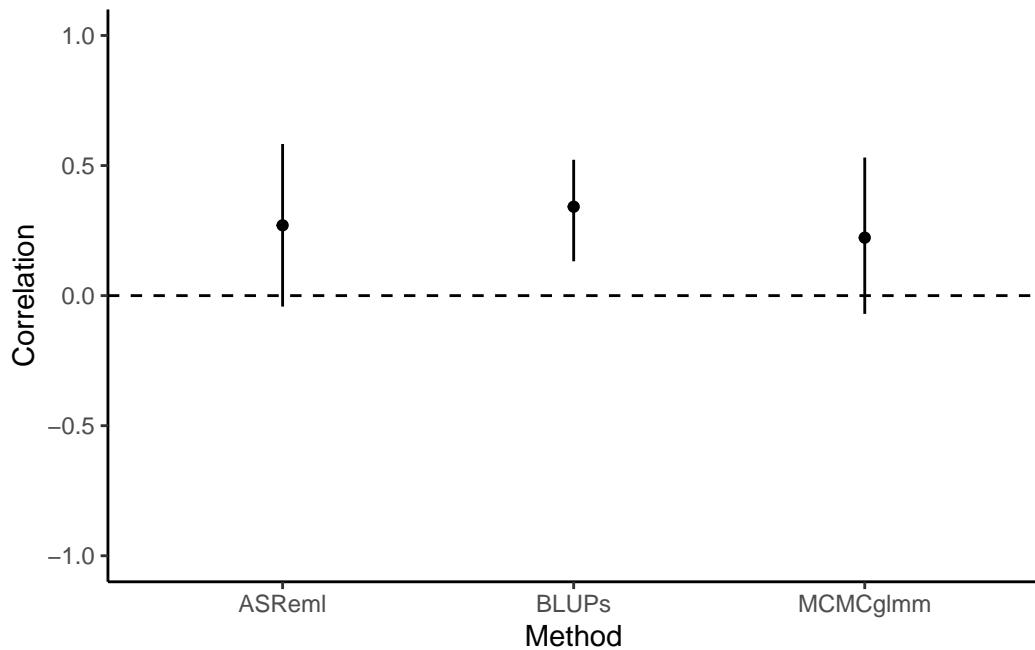


Figure 20.14.: Correlation estimates (with CI) using 3 different methods

Table 20.7.: Correlation (with 95% intervals) between flying speed and exploration estimated with 3 different methods

| Method | Correlation | low | high |
|----------|-------------|--------|-------|
| ASReml | 0.270 | -0.042 | 0.583 |
| BLUPs | 0.342 | 0.132 | 0.522 |
| MCMCglmm | 0.223 | -0.070 | 0.531 |

20.2.5. Happy multivariate models



Figure 20.15.: A female blue dragon of the West

Part VI.

Generalized additive models

Part VII.

Multivariate analysis

Part VIII.

Bayesian approach

Chapter 21

Beyond $P < 0.05$

cite a bunch a must read paper on the subject and maybe summarize the big point of **Do** and **Don't**

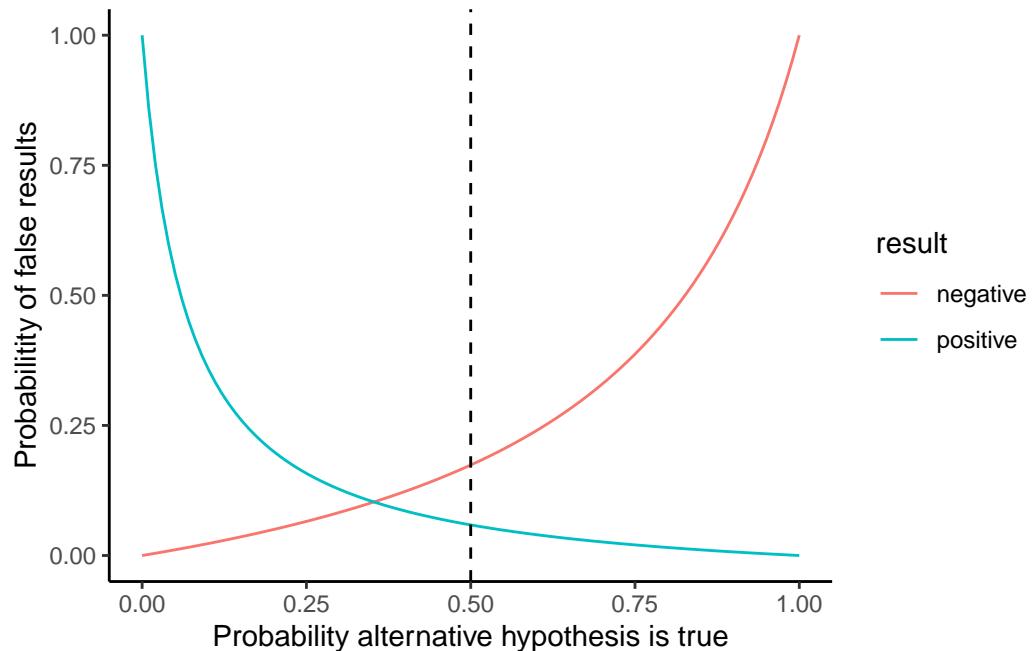


Figure 21.1.: Reflection on the meaning of probabilities in biology

Chapter 22

Introduction to Bayesian Inference

22.1. Lecture

Amazing beasties and crazy animals



Figure 22.1.: Dream pet dragon

need to add stuff here

22.1.1. Bayes' theorem

First, let's review the theorem. Mathematically, it says how to convert one conditional probability into another one.

$$P(B | A) = \frac{P(A | B) * P(B)}{P(A)}$$

The formula becomes more interesting in the context of statistical modeling. We have some model that describes a data-generating process and we have some *observed* data, but we want to estimate some *unknown* model parameters. In that case, the formula reads like:

$$P(\text{hypothesis} \mid \text{data}) = \frac{P(\text{data} \mid \text{hypothesis}) * P(\text{hypothesis})}{P(\text{data})}$$

These terms have conventional names:

$$\text{posterior} = \frac{\text{likelihood} * \text{prior}}{\text{evidence}}$$

Prior and *posterior* describe when information is obtained: what we know pre-data is our prior information, and what we learn post-data is the updated information (“posterior”).

The *likelihood* in the equation says how likely the data is given the model parameters. I think of it as *fit*: How well do the parameters fit the data? Classical regression’s line of best fit is the maximum likelihood line. The likelihood also encompasses the data-generating process behind the model. For example, if we assume that the observed data is normally distributed, then we evaluate the likelihood by using the normal probability density function. You don’t need to know what that last sentence means. What’s important is that the likelihood contains our built-in assumptions about how the data is distributed.

The *evidence* (sometimes called *average likelihood*) is harder to grasp. I am not sure how to describe it in an intuitive way. It’s there to make sure the math works out so that the posterior probabilities sum to 1. Some presentations of Bayes’ theorem gloss over it and I am not the exception 😊. The important thing to note is that the posterior is proportional to the likelihood and prior information.

$$\text{posterior information} \propto \text{likelihood of data} * \text{prior information}$$

So simply put, **you update your prior information in proportion to how well it fits the observed data**. So essentially you are doing that on a daily basis for everything except when you are doing frequentist stats 😊.

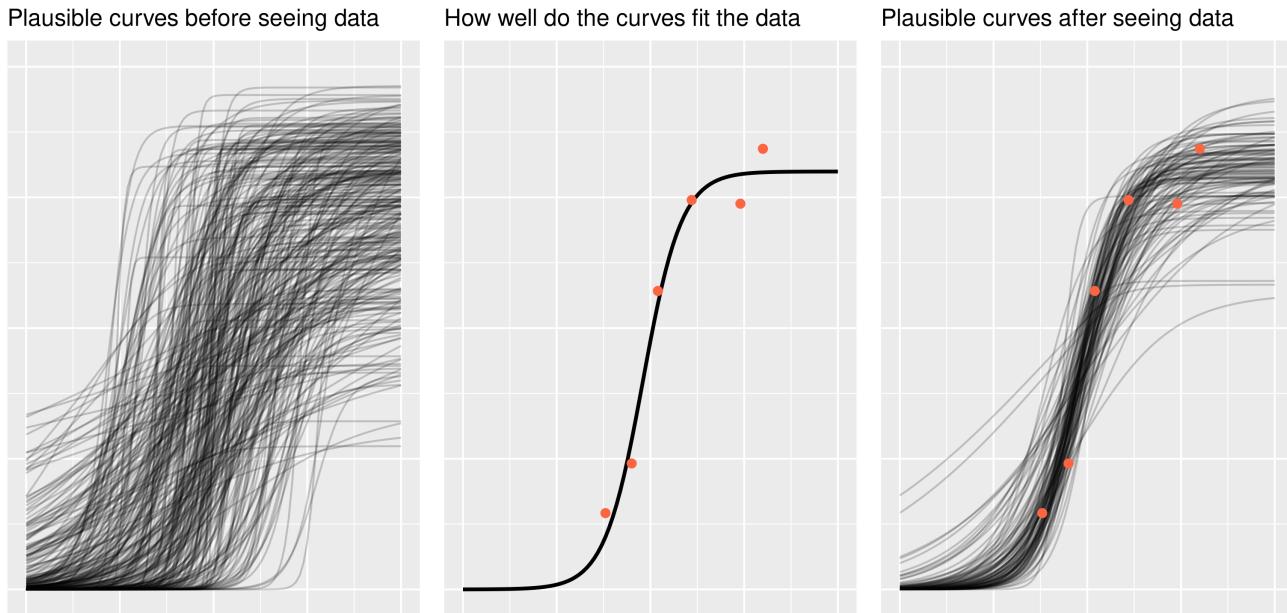


Figure 22.2.: Bayesian Triptych

⚠️ Warning

A word of encouragement! The prior is an intimidating part of Bayesian statistics. It seems highly subjective, as though we are pulling numbers from thin air, and it can be overwhelming for complex models. But if we are familiar with the kind of data we are modeling, we have prior information. We can have the model simulate new observations using the prior distribution and then plot the hypothetical data. Does anything look wrong or implausible about the simulated data? If so, then we have some prior information that we can include in our model. Note that we do not evaluate the plausibility of the simulated data based on the data we have in hand (the data we want to model); that's not

22.1.2. Intro to MCMC

We will now walk through a simple example coded in R to illustrate how an MCMC algorithm works.

Suppose you are interested in the mean heart rate of students when asked a question in a stat course. You are not sure what the exact mean value is, but you know the values are normally distributed with a standard deviation of 15. You have observed 5 individuals to have heart rate of 104, 120, 160, 90, 130. You could use MCMC sampling to draw samples from the target distribution. We need to specify:

1. the starting value for the chain.
2. the length of the chain. In general, more iterations will give you more accurate output.

```
library(coda)
library(bayesplot)
```

This is bayesplot version 1.11.1

- Online documentation and vignettes at mc-stan.org/bayesplot
- bayesplot theme set to `bayesplot::theme_default()`
 - * Does `_not_` affect other `ggplot2` plots
 - * See `?bayesplot_theme_set` for details on theme setting

```
set.seed(170)

hr_obs <- c(104, 112, 132, 115, 110)

start_value <- 250

n_iter <- 2500 # define number of iterations

pd_mean <- numeric(n_iter) # create vector for sample values

pd_mean[1] <- start_value # define starting value

for (i in 2:n_iter) {
  proposal <- pd_mean[i - 1] + MASS::mvrnorm(1, 0, 5) # proposal
  lprop <- sum(dnorm(proposal, hr_obs, 15)) # likelihood of proposed parameter
  lprev <- sum(dnorm(pd_mean[i - 1], hr_obs, 15))
  if (lprop / lprev > runif(1)) { # if likelihood of proposed > likelihood previous accept
    # and if likelihood is lower accept with random noise
    pd_mean[i] <- proposal
  } # if true sample the proposal
  else {
    (pd_mean[i] <- pd_mean[i - 1])
```

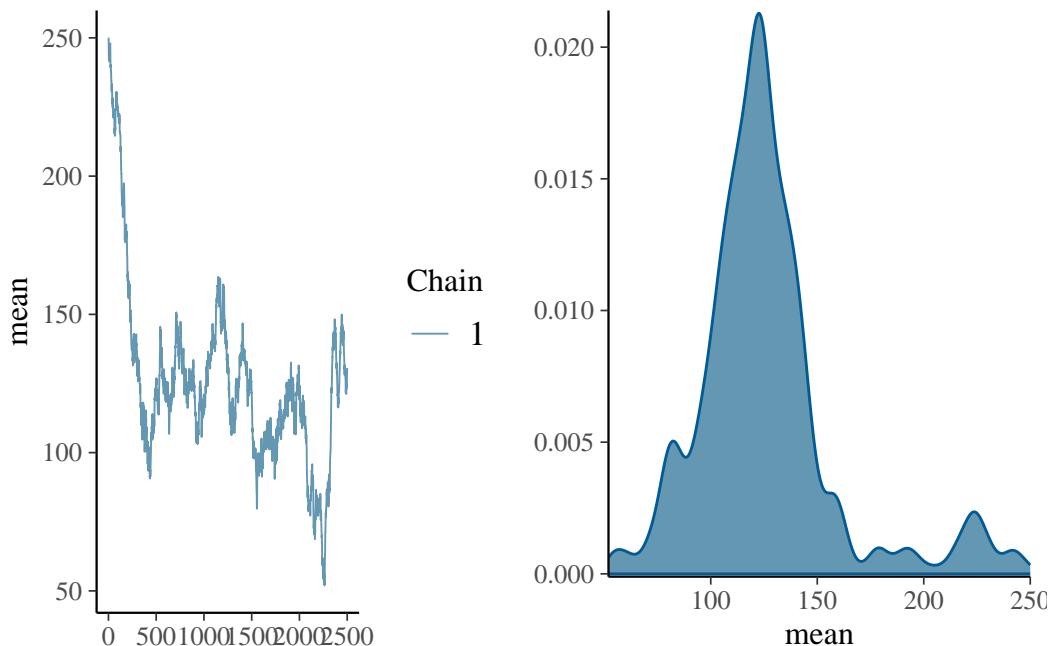
```

} # if false sample the current value
}

pd_mean <- as.mcmc(data.frame(mean = pd_mean))

mcmc_combo(pd_mean, combo = c("trace", "dens"))

```



```
summary(pd_mean)
```

```

Iterations = 1:2500
Thinning interval = 1
Number of chains = 1
Sample size per chain = 2500

```

1. Empirical mean and standard deviation for each variable,
plus standard error of the mean:

| Mean | SD | Naive SE | Time-series SE |
|----------|---------|----------|----------------|
| 125.8105 | 32.8672 | 0.6573 | 13.3046 |

2. Quantiles for each variable:

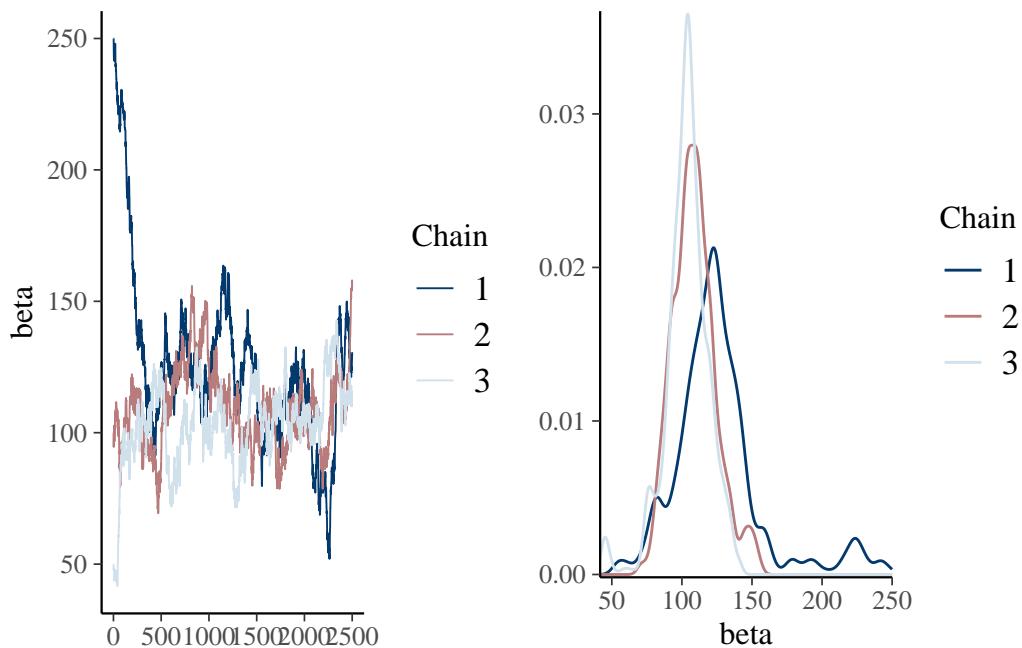
```
2.5%     25%     50%     75%   97.5%
75.53 108.03 122.19 136.12 225.46
```

```
set.seed(170)
hr_obs <- c(104, 112, 132, 115, 110)
n_iter <- 2500 # define number of iterations

n_chain <- 3
start_value <- c(250, 100, 50)

pd_mean <- array(NA, dim = c(n_iter, n_chain, 1), dimnames = list(iter = NULL, chain = NULL, para

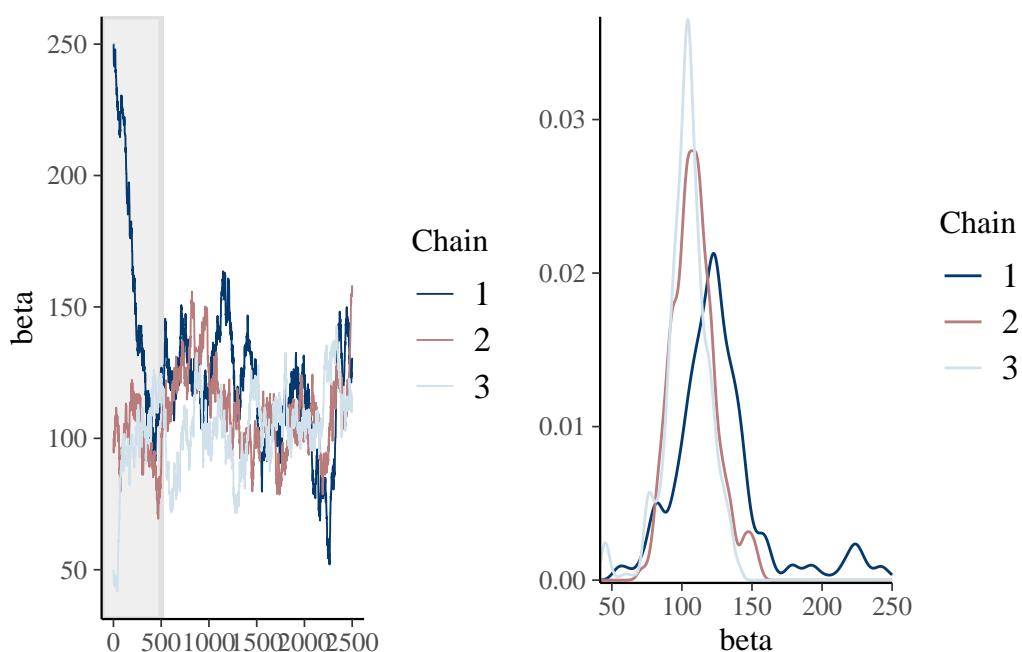
for (j in seq_len(n_chain)) {
  pd_mean[1, j, 1] <- start_value[j] # define starting value
  for (i in 2:n_iter) {
    proposal <- pd_mean[i - 1, j, 1] + MASS::mvrnorm(1, 0, 5) # proposal
    if (sum(dnorm(proposal, hr_obs, 15)) # likelihood of proposed parameter
    / sum(dnorm(pd_mean[i - 1, j, 1], hr_obs, 15)) > runif(1, 0, 1)) {
      pd_mean[i, j, 1] <- proposal
    } # if true sample the proposal
    else {
      (pd_mean[i, j, 1] <- pd_mean[i - 1, j, 1])
    } # if false sample the current value
  }
}
color_scheme_set("mix-blue-red")
mcmc_combo(pd_mean, combo = c("trace", "dens_overlay"))
```



```
summary(pd_mean)
```

| | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|--|-------|---------|--------|--------|---------|--------|
| | 41.65 | 99.32 | 109.68 | 112.71 | 122.52 | 250.00 |

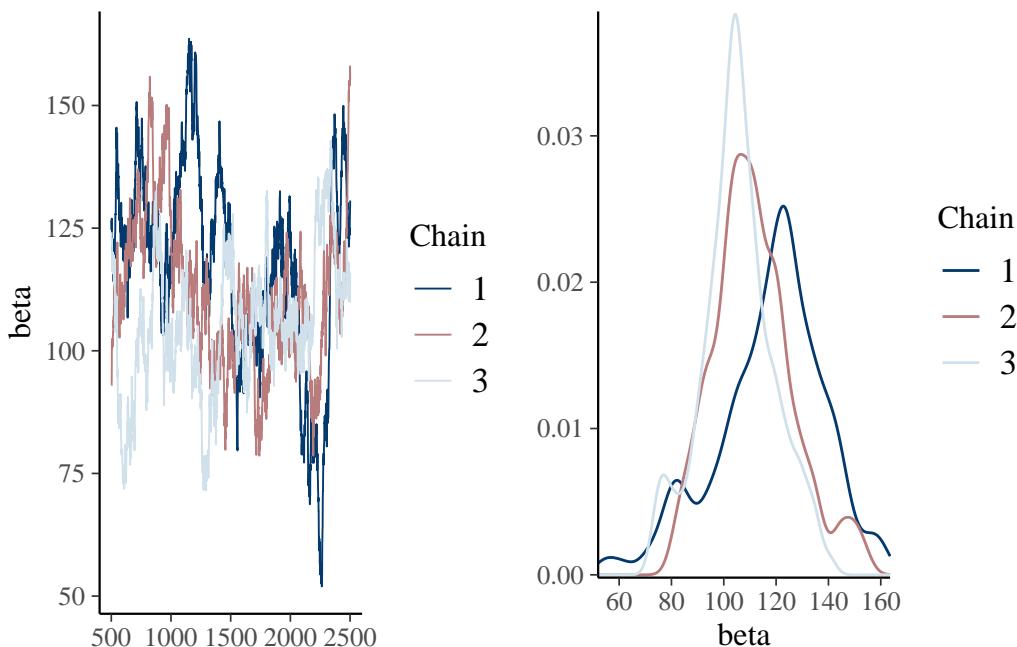
```
mcmc_combo(pd_mean, combo = c("trace", "dens_overlay"), n_warmup = 500)
```



```
pd_burn <- pd_mean[-c(1:500), , , drop = FALSE]
summary(pd_burn)
```

| | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|--|-------|---------|--------|--------|---------|--------|
| | 51.98 | 100.71 | 110.38 | 111.42 | 122.69 | 163.58 |

```
mcmc_combo(pd_burn, combo = c("trace", "dens_overlay"), iter1 = 501)
```



22.1.3. Inferences

22.1.3.1. Fixed effects

Easy peazy lemon squeezy just have a look at the posteriro distribution, does it overlap 0 yes or no.

talk about mean, median and mode of a distribution as well as credible intervals

22.1.3.2. Random effects

Quite a bit more harder. because constrained to be positive

- Interpreting posterior distribution
- DIC

- WAIC

22.2. Practical

In this practical, we will revisit our analysis on unicorn aggressivity. Honestly, we can use any other data with repeated measures for this exercise but I just love unicorns ❤️. However, instead of fitting the model using `lmer()` from the `lmerTest` 📦 (Kuznetsova et al. 2017), we will refit the model using 2 excellent softwares fitting models with a Bayesian approach: `MCMCglmm` (Hadfield 2010) and `brms` (Bürkner 2021).

22.2.1. R packages needed

First we load required libraries

```
library(lmerTest)
library(tidyverse)
library(rptR)
library(brms)
library(MCMCglmm)
library(bayesplot)
```

22.2.2. A refresher on unicorn ecology

The last model on unicorns was:

```
aggression ~ opp_size + scale(body_size, center = TRUE, scale = TRUE)
           + scale(assay_rep, scale = FALSE) + block
           + (1 | ID)
```

Those scaled terms are abit a sore for my eyes and way too long if we need to type them multiple times in this practical. So first let's recode them. -

```
unicorns <- read.csv("data/unicorns_aggression.csv")
unicorns <- unicorns %>%
  mutate(
```

```

body_size_sc = scale(body_size),
assay_rep_sc = scale(assay_rep, scale = FALSE)
)

```

Ok now we can fit the same model by just using:

```

aggression ~ opp_size + body_size_sc + assay_rep_sc + block
+ (1 | ID)

```

We can now fit a model using `lmer()`. Since we want to compare a bit REML and Bayesian aproaches, I am going to wrap the model function in a function called `system.time()`. This function simply estimate the user and computer time use by the function.

```

mer_time <- system.time(
  m_mer <- lmer(
    aggression ~ opp_size + body_size_sc + assay_rep_sc + block
    + (1 | ID),
    data = unicorns
  )
)
mer_time

```

| user | system | elapsed |
|-------|--------|---------|
| 0.064 | 0.000 | 0.064 |

```
summary(m_mer)
```

```

Linear mixed model fit by REML. t-tests use Satterthwaite's method [
lmerModLmerTest]
Formula: aggression ~ opp_size + body_size_sc + assay_rep_sc + block +
(1 | ID)
Data: unicorns

```

```
REML criterion at convergence: 1136.5
```

Scaled residuals:

| Min | 1Q | Median | 3Q | Max |
|----------|----------|---------|---------|---------|
| -2.85473 | -0.62831 | 0.02545 | 0.68998 | 2.74064 |

Random effects:

| Groups | Name | Variance | Std.Dev. |
|--------|-------------|----------|----------|
| ID | (Intercept) | 0.02538 | 0.1593 |
| | Residual | 0.58048 | 0.7619 |

Number of obs: 480, groups: ID, 80

Fixed effects:

| | Estimate | Std. Error | df | t value | Pr(> t) |
|--------------|----------|------------|-----------|---------|------------|
| (Intercept) | 9.00181 | 0.03907 | 78.07315 | 230.395 | <2e-16 *** |
| opp_size | 1.05141 | 0.04281 | 396.99857 | 24.562 | <2e-16 *** |
| body_size_sc | 0.03310 | 0.03896 | 84.21144 | 0.850 | 0.398 |
| assay_rep_sc | -0.05783 | 0.04281 | 396.99857 | -1.351 | 0.177 |
| block | -0.02166 | 0.06955 | 397.00209 | -0.311 | 0.756 |
| --- | | | | | |

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Correlation of Fixed Effects:

| | (Intr) | opp_sz | bdy_s_ | assy__ |
|-------------|--------|--------|--------|--------|
| opp_size | 0.000 | | | |
| body_siz_sc | 0.000 | 0.000 | | |
| assay_rp_sc | 0.000 | -0.100 | 0.000 | |
| block | 0.000 | 0.000 | 0.002 | 0.000 |

Ok so it took no time at all to do it and we got our “classic” results.

22.2.3. MCMCglmm

What makes MCMCglmm so useful and powerful 🍍 in ecology and for *practical Bayesian people* is that:

1. it is blazing fast ➡ (for Bayesian analysis) for some models particularly models with structured covariances
2. it is fairly intuitive to code

but it also has some inconvenients:

1. it is blazing fast for **Bayesian analysis** meaning it is 🐾 compared to *maximum likelihood* approaches
2. it has some limitations in terms of functionality, distribution availability and model specifications compared to other *Bayesian* softwares
3. the priors, *oh, the priors* 🤯, are a bit tricky to code and understand 🤓.

22.2.3.1. Fitting the Model

So here is how we can code the model in `MCMCglmm()`. It is fairly similar to `lmer()` except that the random effects are specified in a different *argument*.

```
mcglm_time <- system.time(
  m_mcmcglmm <- MCMCglmm(
    aggression ~ opp_size + body_size_sc + assay_rep_sc + block,
    random = ~ID,
    data = unicorns
  )
)
```

MCMC iteration = 0

MCMC iteration = 1000

MCMC iteration = 2000

MCMC iteration = 3000

MCMC iteration = 4000

MCMC iteration = 5000

```
MCMC iteration = 6000  
  
MCMC iteration = 7000  
  
MCMC iteration = 8000  
  
MCMC iteration = 9000  
  
MCMC iteration = 10000  
  
MCMC iteration = 11000  
  
MCMC iteration = 12000  
  
MCMC iteration = 13000
```

```
summary(m_mcmcglmm)
```

```
Iterations = 3001:12991  
Thinning interval = 10  
Sample size = 1000
```

```
DIC: 1128.004
```

```
G-structure: ~ID
```

| | post.mean | l-95% CI | u-95% CI | eff.samp |
|----|-----------|-----------|----------|----------|
| ID | 0.003686 | 9.807e-14 | 0.0262 | 45.81 |

```
R-structure: ~units
```

| | post.mean | l-95% CI | u-95% CI | eff.samp |
|--|-----------|----------|----------|----------|
|--|-----------|----------|----------|----------|

```
units      0.6044    0.5228    0.6819      1000
```

Location effects: aggression ~ opp_size + body_size_sc + assay_rep_sc + block

| | post.mean | l-95% CI | u-95% CI | eff.samp | pMCMC | | | | | | |
|----------------|-----------|----------|----------|----------|------------|-----|------|------|-----|-----|---|
| (Intercept) | 9.00152 | 8.93150 | 9.07158 | 1000 | <0.001 *** | | | | | | |
| opp_size | 1.04940 | 0.96813 | 1.12946 | 1000 | <0.001 *** | | | | | | |
| body_size_sc | 0.03154 | -0.03985 | 0.09563 | 1000 | 0.410 | | | | | | |
| assay_rep_sc | -0.05620 | -0.13196 | 0.03546 | 893 | 0.184 | | | | | | |
| block | -0.02069 | -0.16186 | 0.11553 | 1000 | 0.774 | | | | | | |
| --- | | | | | | | | | | | |
| Signif. codes: | 0 | '***' | 0.001 | '**' | 0.01 | '*' | 0.05 | '. ' | 0.1 | ' ' | 1 |

```
mcglm_time
```

| user | system | elapsed |
|-------|--------|---------|
| 1.237 | 0.000 | 1.237 |

Model is slow and not good. We need more iteration and maybe even a longer burnin, and honestly maybe better priors.

We can still take the time to have a look at the R object output from `MCMCglmm()`. The 2 main parts we are interested in are:

- `Sol` which stand for the model solution and includes the posteriro distribution of the fixed effects
- `VCV`, for the variance covariance estimates, which includes the posterior distribution of all (co)variances estimates for both random effects and residual variance.

```
omar <- par()
par(mar = c(4, 2, 1.5, 2))
plot(m_mcmcglmm$Sol)
```

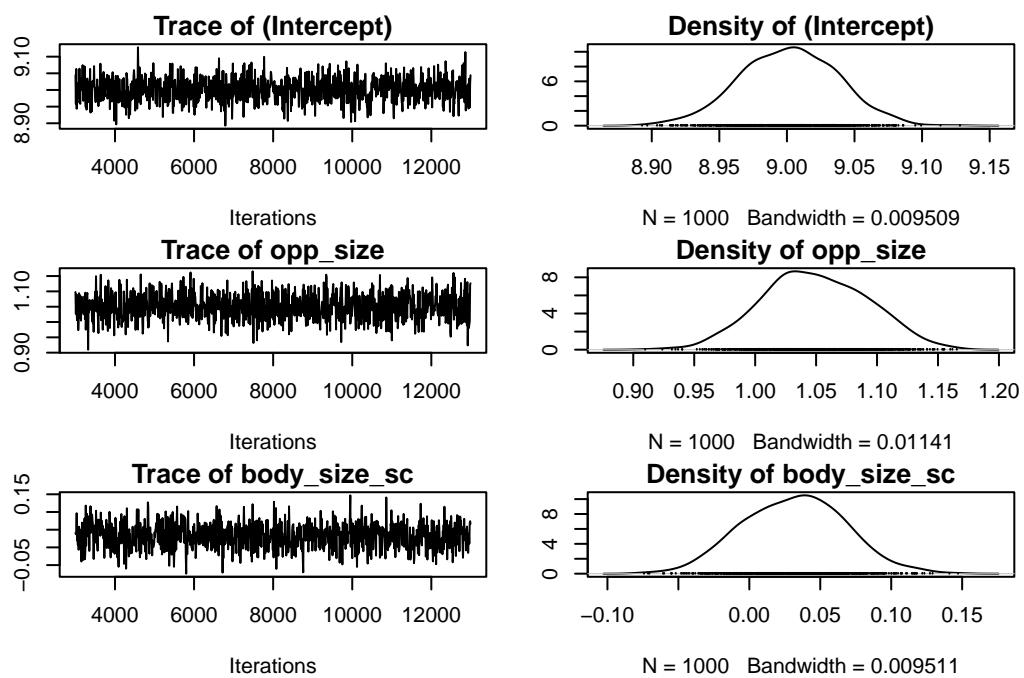


Figure 22.3.: Posterior trace and distribution of the parameters in `m_mcmcglmm` using default settings

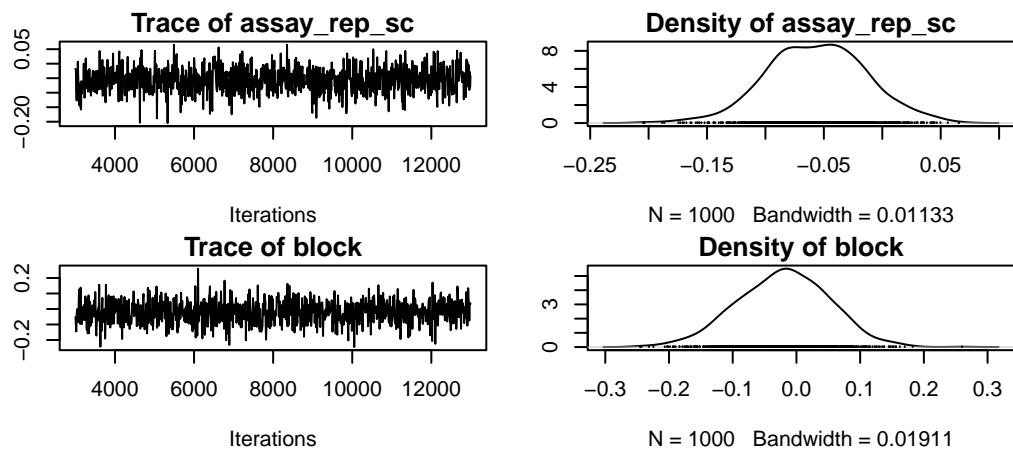


Figure 22.4.: Posterior trace and distribution of the parameters in `m_mcmcglmm` using default settings

```
plot(m_mcmcglmm$VCV)
```

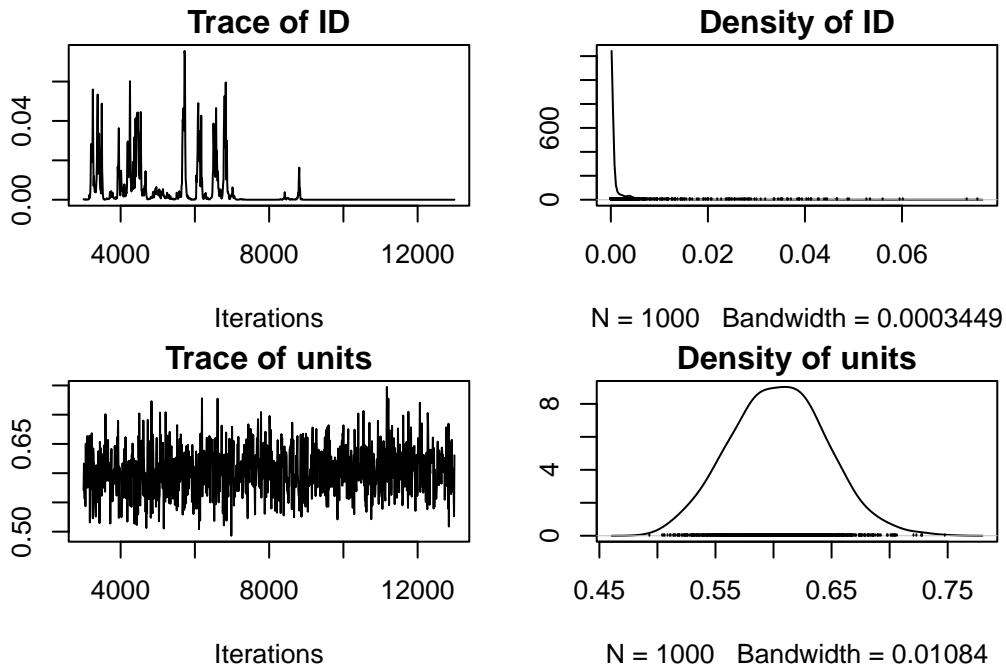


Figure 22.5.: Posterior trace and distribution of the parameters in `m_mcmcglmm` using default settings

```
par(omar)
autocorr.diag(m_mcmcglmm$VCV)
```

| | ID | units |
|---------|-----------|-------------|
| Lag 0 | 1.0000000 | 1.0000000 |
| Lag 10 | 0.8042405 | -0.02074155 |
| Lag 50 | 0.4807583 | -0.04264317 |
| Lag 100 | 0.1951356 | 0.04422296 |
| Lag 500 | 0.1254589 | 0.04401956 |

Talk about autocorrelation, mixing, convergence and priors here

```
n_samp <- 1000
thin <- 500
burnin <- 20000
mcglm_time <- system.time(
  m_mcmcglmm <- MCMCglmm(
    aggression ~ opp_size + body_size_sc + assay_rep_sc + block,
    random = ~ID,
```

```
data = unicorns,
nitt = n_samp * thin + burnin, thin = thin, burnin = burnin,
verbose = FALSE,
prior = list(
  R = list(V = 1, nu = 0.002),
  G = list(
    G1 = list(V = 1, nu = 0.002)
  )
)
)
summary(m_mcmcglmm)
```

Iterations = 20001:519501

Thinning interval = 500

Sample size = 1000

DIC: 1126.66

G-structure: ~ID

| | post.mean | l-95% CI | u-95% CI | eff.samp |
|----|-----------|-----------|----------|----------|
| ID | 0.01987 | 0.0002904 | 0.05458 | 1000 |

R-structure: ~units

| | post.mean | l-95% CI | u-95% CI | eff.samp |
|-------|-----------|----------|----------|----------|
| units | 0.5917 | 0.5188 | 0.6763 | 1000 |

Location effects: aggression ~ opp_size + body_size_sc + assay_rep_sc + block

| | post.mean | l-95% CI | u-95% CI | eff.samp | pMCMC |
|--|-----------|----------|----------|----------|-------|
|--|-----------|----------|----------|----------|-------|

```
(Intercept) 9.00136 8.92221 9.07383      1000 <0.001 ***
opp_size     1.05363 0.96382 1.13650      1000 <0.001 ***
body_size_sc 0.03373 -0.03781 0.10686      1000  0.396
assay_rep_sc -0.05861 -0.14186 0.02882      1000  0.182
block        -0.02709 -0.16061 0.11441      1000  0.698
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
mcglm_time
```

```
user  system elapsed
50.678 0.000 50.717
```

evaluate model here

```
omar <- par()
par(mar = c(4, 2, 1.5, 2))
plot(m_mcmcglmm$Sol)
```

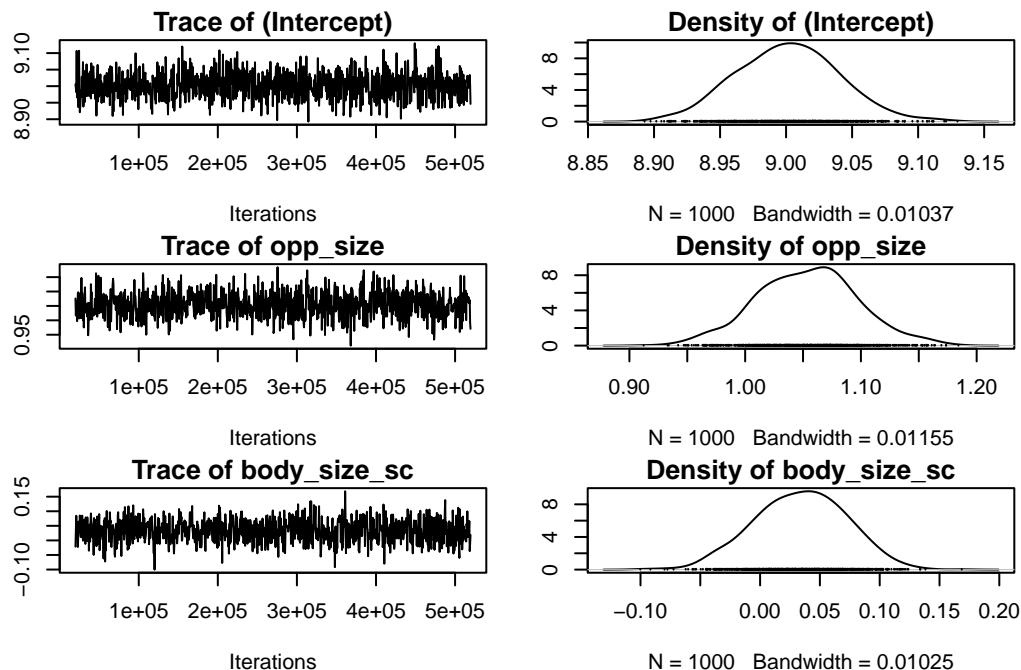


Figure 22.6.: Posterior trace and distribution of the parameters in m_mcmcglmm with better settings

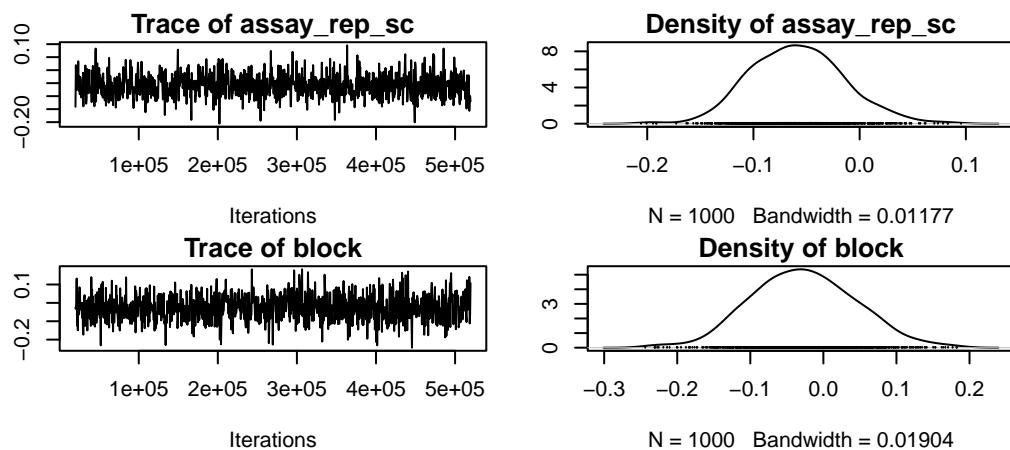


Figure 22.7.: Posterior trace and distribution of the parameters in m_mcmcglmm with better settings

```
plot(m_mcmcglmm$VCV)
```

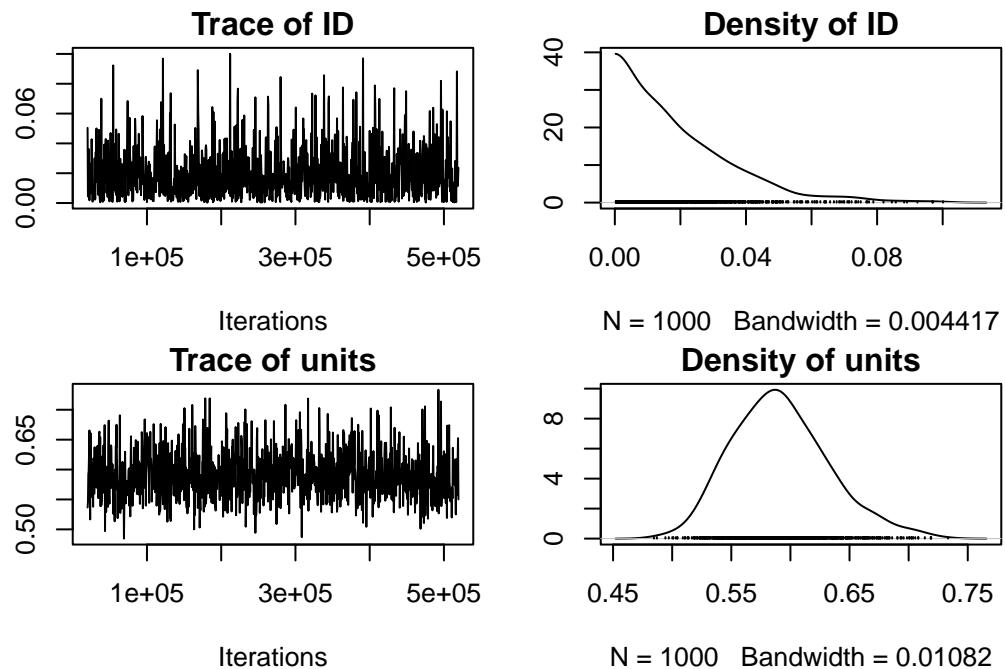


Figure 22.8.: Posterior trace and distribution of the parameters in m_mcmcglmm with better settings

```
par(omar)
autocorr.diag(m_mcmcglmm$VCV)
```

| | ID | units |
|-----------|--------------|--------------|
| Lag 0 | 1.000000000 | 1.000000000 |
| Lag 500 | 0.013876043 | -0.044235206 |
| Lag 2500 | 0.026120260 | -0.048012241 |
| Lag 5000 | -0.049357725 | 0.021158672 |
| Lag 25000 | 0.002544256 | -0.003722595 |

22.2.4. Inferences

22.2.4.1. Fixed effects

Easy peazy lemon squeezy just have a look at the posterior distribution, does it overlap 0 yes or no.

```
posterior.mode(m_mcmcglmm$Sol)
```

| | opp_size | body_size_sc | assay_rep_sc | block |
|-------------|------------|--------------|--------------|-------------|
| (Intercept) | 9.00632282 | 1.07353252 | 0.03500916 | -0.04048582 |
| | | | | -0.03276275 |

```
HPDinterval(m_mcmcglmm$Sol)
```

| | lower | upper |
|----------------------|-------------|------------|
| (Intercept) | 8.92221005 | 9.07383400 |
| opp_size | 0.96382086 | 1.13649873 |
| body_size_sc | -0.03781276 | 0.10685606 |
| assay_rep_sc | -0.14185602 | 0.02882443 |
| block | -0.16060691 | 0.11440706 |
| attr(,"Probability") | | |
| [1] | 0.95 | |

22.2.4.2. Random effects

Quite a bit more harder. because constrained to be positive

```
posterior.mode(m_mcmcglmm$VCV)
```

| ID | units |
|------------|------------|
| 0.00096263 | 0.59129362 |

```
HPDinterval(m_mcmcglmm$VCV)
```

| | lower | upper |
|----------------------|--------------|------------|
| ID | 0.0002903938 | 0.05458376 |
| units | 0.5188238599 | 0.67634529 |
| attr(,"Probability") | | |
| [1] | 0.95 | |

22.2.5. brms

brms is an acronym for *Bayesian Regression Models using ‘Stan’* (Bürkner 2021). It is a package developed to fit regression models with a Bayesian approach using the amazing `stan` software (Stan Development Team 2021).

What makes **brms** so useful and powerful 💪 in ecology is that:

1. it is really intuitive to code (same syntax as `glmer()`)
2. it is incredibly flexible since it is essentially a front end for `stan` via its `rstan` interface (Stan Development Team 2024)

but with *great powers come great responsibility* 🦖

```
brm_time <- system.time(
  m_brm <- brm(
    aggression ~ opp_size + body_size_sc + assay_rep_sc + block
    + (1 | ID),
    data = unicorns, iter = 4750, warmup = 1000, thin = 15, cores = 4
    # refresh = 0
```

```
)
)
```

Compiling Stan program...

Start sampling

```
brm_time
```

| user | system | elapsed |
|--------|--------|---------|
| 90.998 | 4.822 | 81.741 |

```
summary(m_brm)
```

Family: gaussian

Links: mu = identity; sigma = identity

Formula: aggression ~ opp_size + body_size_sc + assay_rep_sc + block + (1 | ID)

Data: unicorns (Number of observations: 480)

Draws: 4 chains, each with iter = 4750; warmup = 1000; thin = 15;

total post-warmup draws = 1000

Multilevel Hyperparameters:

~ID (Number of levels: 80)

| | Estimate | Est.Error | l-95% | CI | u-95% | CI | Rhat | Bulk_ESS | Tail_ESS |
|---------------|----------|-----------|-------|------|-------|------|------|----------|----------|
| sd(Intercept) | 0.14 | 0.07 | 0.01 | 0.27 | 1.00 | 1.00 | 1054 | 1037 | |

Regression Coefficients:

| | Estimate | Est.Error | l-95% | CI | u-95% | CI | Rhat | Bulk_ESS | Tail_ESS |
|--------------|----------|-----------|-------|------|-------|------|------|----------|----------|
| Intercept | 9.00 | 0.04 | 8.92 | 9.08 | 1.01 | 1.01 | 957 | 936 | |
| opp_size | 1.05 | 0.04 | 0.97 | 1.14 | 1.01 | 1.01 | 954 | 986 | |
| body_size_sc | 0.03 | 0.04 | -0.04 | 0.10 | 1.01 | 1.01 | 1038 | 916 | |
| assay_rep_sc | -0.06 | 0.04 | -0.15 | 0.03 | 1.00 | 1.00 | 967 | 1033 | |
| block | -0.02 | 0.07 | -0.16 | 0.11 | 1.00 | 1.00 | 1073 | 846 | |

Further Distributional Parameters:

| | Estimate | Est.Error | l-95% CI | u-95% CI | Rhat | Bulk_ESS | Tail_ESS |
|-------|----------|-----------|----------|----------|------|----------|----------|
| sigma | 0.77 | 0.03 | 0.72 | 0.82 | 1.00 | 982 | 949 |

Draws were sampled using sampling(NUTS). For each parameter, Bulk_ESS and Tail_ESS are effective sample size measures, and Rhat is the potential scale reduction factor on split chains (at convergence, Rhat = 1).

```
mcmc_acf_bar(m_brm, regex_pars = c("sd"))
```

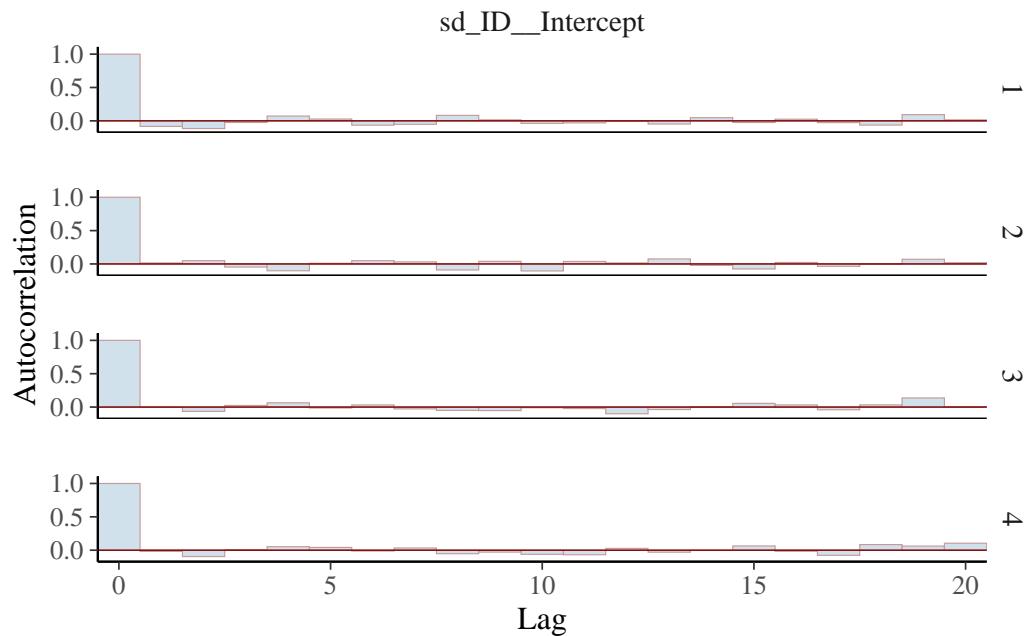


Figure 22.9.: Autocorrelation in the chain for variance parameters in model m_brm

22.2.5.1. Under the hood

have a look at the stan code

```
stancode(m_brm)
```

```
// generated with brms 2.21.0
functions {
```

```

data {
    int<lower=1> N; // total number of observations
    vector[N] Y; // response variable
    int<lower=1> K; // number of population-level effects
    matrix[N, K] X; // population-level design matrix
    int<lower=1> Kc; // number of population-level effects after centering
    // data for group-level effects of ID 1
    int<lower=1> N_1; // number of grouping levels
    int<lower=1> M_1; // number of coefficients per level
    array[N] int<lower=1> J_1; // grouping indicator per observation
    // group-level predictor values
    vector[N] Z_1_1;
    int prior_only; // should the likelihood be ignored?
}

transformed data {
    matrix[N, Kc] Xc; // centered version of X without an intercept
    vector[Kc] means_X; // column means of X before centering
    for (i in 2:K) {
        means_X[i - 1] = mean(X[, i]);
        Xc[, i - 1] = X[, i] - means_X[i - 1];
    }
}

parameters {
    vector[Kc] b; // regression coefficients
    real Intercept; // temporary intercept for centered predictors
    real<lower=0> sigma; // dispersion parameter
    vector<lower=0>[M_1] sd_1; // group-level standard deviations
    array[M_1] vector[N_1] z_1; // standardized group-level effects
}

transformed parameters {
    vector[N_1] r_1_1; // actual group-level effects
    real lprior = 0; // prior contributions to the log posterior
    r_1_1 = (sd_1[1] * (z_1[1]));
}

```

```
lprior += student_t_lpdf(Intercept | 3, 8.9, 2.5);
lprior += student_t_lpdf(sigma | 3, 0, 2.5)
- 1 * student_t_lccdf(0 | 3, 0, 2.5);
lprior += student_t_lpdf(sd_1 | 3, 0, 2.5)
- 1 * student_t_lccdf(0 | 3, 0, 2.5);
}

model {
    // likelihood including constants
    if (!prior_only) {
        // initialize linear predictor term
        vector[N] mu = rep_vector(0.0, N);
        mu += Intercept;
        for (n in 1:N) {
            // add more terms to the linear predictor
            mu[n] += r_1_1[J_1[n]] * Z_1_1[n];
        }
        target += normal_id_glm_lpdf(Y | Xc, mu, b, sigma);
    }
    // priors including constants
    target += lprior;
    target += std_normal_lpdf(z_1[1]);
}
generated quantities {
    // actual population-level intercept
    real b_Intercept = Intercept - dot_product(means_X, b);
}
```

22.2.5.2. using shiny

```
launch_shinystan(m_brm)
```

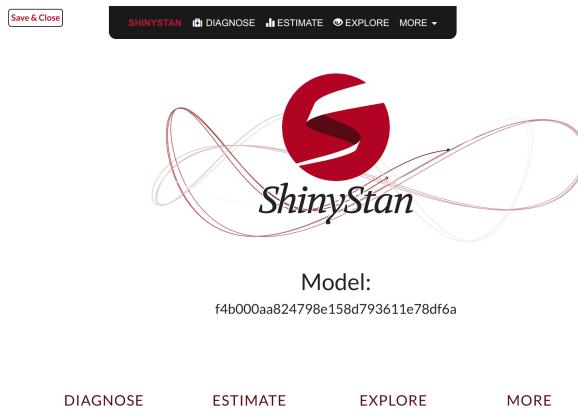


Figure 22.10.: Shinystan interface

22.2.6. Inferences

22.2.6.1. Fixed effects

```
summary(m_brm)
```

Family: gaussian

Links: mu = identity; sigma = identity

Formula: aggression ~ opp_size + body_size_sc + assay_rep_sc + block + (1 | ID)

Data: unicorns (Number of observations: 480)

Draws: 4 chains, each with iter = 4750; warmup = 1000; thin = 15;

total post-warmup draws = 1000

Multilevel Hyperparameters:

~ID (Number of levels: 80)

| | Estimate | Est.Error | l-95% | CI | u-95% | CI | Rhat | Bulk_ESS | Tail_ESS |
|--|----------|-----------|-------|----|-------|----|------|----------|----------|
|--|----------|-----------|-------|----|-------|----|------|----------|----------|

| | | | | | | | | |
|---------------|------|------|------|------|------|------|------|--|
| sd(Intercept) | 0.14 | 0.07 | 0.01 | 0.27 | 1.00 | 1054 | 1037 | |
|---------------|------|------|------|------|------|------|------|--|

Regression Coefficients:

| | Estimate | Est.Error | l-95% | CI | u-95% | CI | Rhat | Bulk_ESS | Tail_ESS |
|--|----------|-----------|-------|----|-------|----|------|----------|----------|
|--|----------|-----------|-------|----|-------|----|------|----------|----------|

| | | | | | | | | |
|-----------|------|------|------|------|------|-----|-----|--|
| Intercept | 9.00 | 0.04 | 8.92 | 9.08 | 1.01 | 957 | 936 | |
|-----------|------|------|------|------|------|-----|-----|--|

| | | | | | | | |
|--------------|-------|------|-------|------|------|------|------|
| opp_size | 1.05 | 0.04 | 0.97 | 1.14 | 1.01 | 954 | 986 |
| body_size_sc | 0.03 | 0.04 | -0.04 | 0.10 | 1.01 | 1038 | 916 |
| assay_rep_sc | -0.06 | 0.04 | -0.15 | 0.03 | 1.00 | 967 | 1033 |
| block | -0.02 | 0.07 | -0.16 | 0.11 | 1.00 | 1073 | 846 |

Further Distributional Parameters:

| | Estimate | Est.Error | l-95% | CI | u-95% | CI | Rhat | Bulk_ESS | Tail_ESS |
|-------|----------|-----------|-------|------|-------|-----|------|----------|----------|
| sigma | 0.77 | 0.03 | 0.72 | 0.82 | 1.00 | 982 | 949 | | |

Draws were sampled using sampling(NUTS). For each parameter, Bulk_ESS and Tail_ESS are effective sample size measures, and Rhat is the potential scale reduction factor on split chains (at convergence, Rhat = 1).

```
mcmc_plot(m_brm, regex_pars = "b_")
```

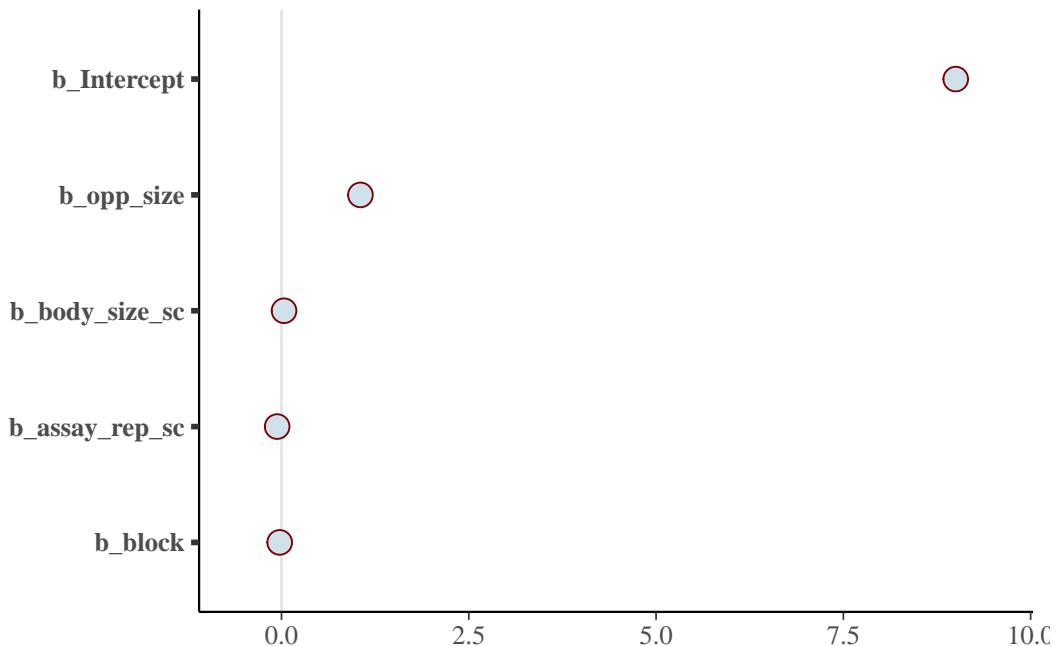


Figure 22.11.: Fixed effect estimates (with 95% credible intervals) from model m_brm

22.2.6.2. Random effects

```
summary(m_brm)
```

Family: gaussian

Links: mu = identity; sigma = identity

Formula: aggression ~ opp_size + body_size_sc + assay_rep_sc + block + (1 | ID)

Data: unicorns (Number of observations: 480)

Draws: 4 chains, each with iter = 4750; warmup = 1000; thin = 15;

total post-warmup draws = 1000

Multilevel Hyperparameters:

~ID (Number of levels: 80)

| | Estimate | Est.Error | l-95% CI | u-95% CI | Rhat | Bulk_ESS | Tail_ESS |
|---------------|----------|-----------|----------|----------|------|----------|----------|
| sd(Intercept) | 0.14 | 0.07 | 0.01 | 0.27 | 1.00 | 1054 | 1037 |

Regression Coefficients:

| | Estimate | Est.Error | l-95% CI | u-95% CI | Rhat | Bulk_ESS | Tail_ESS |
|--------------|----------|-----------|----------|----------|------|----------|----------|
| Intercept | 9.00 | 0.04 | 8.92 | 9.08 | 1.01 | 957 | 936 |
| opp_size | 1.05 | 0.04 | 0.97 | 1.14 | 1.01 | 954 | 986 |
| body_size_sc | 0.03 | 0.04 | -0.04 | 0.10 | 1.01 | 1038 | 916 |
| assay_rep_sc | -0.06 | 0.04 | -0.15 | 0.03 | 1.00 | 967 | 1033 |
| block | -0.02 | 0.07 | -0.16 | 0.11 | 1.00 | 1073 | 846 |

Further Distributional Parameters:

| | Estimate | Est.Error | l-95% CI | u-95% CI | Rhat | Bulk_ESS | Tail_ESS |
|-------|----------|-----------|----------|----------|------|----------|----------|
| sigma | 0.77 | 0.03 | 0.72 | 0.82 | 1.00 | 982 | 949 |

Draws were sampled using sampling(NUTS). For each parameter, Bulk_ESS and Tail_ESS are effective sample size measures, and Rhat is the potential scale reduction factor on split chains (at convergence, Rhat = 1).

```
mcmc_plot(m_brm, pars = c("sd_ID__Intercept", "sigma"))
```

Warning: Argument 'pars' is deprecated. Please use 'variable' instead.

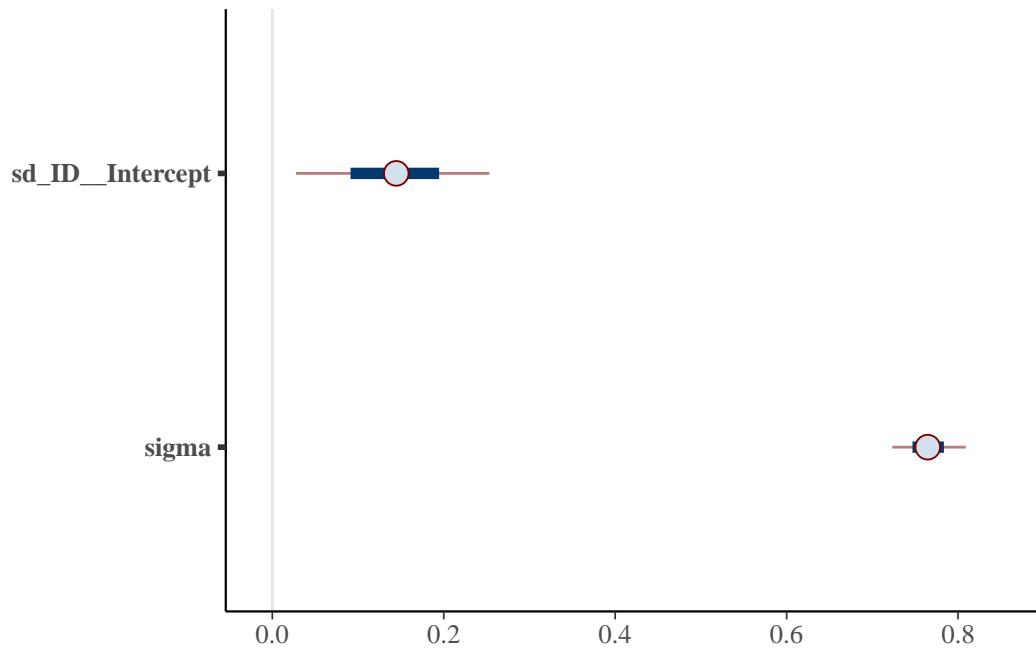


Figure 22.12.: Among-individual and residual standard deviance (with 95% credible intervals) estimated from model m_brm

22.2.7. Happy Bayesian stats



Figure 22.13.: Sherlock Holmes, a truly bayesian detective

References

R packages

This book was produced all packages (excluding dependencies) listed in table Table 22.1. As recommended by the ‘tidyverse’ team, all citations to tidyverse packages are collapsed into a single citation.

Table 22.1.: R packages used in this book

| Package | Version | Citation |
|-------------|------------|--|
| asreml | 4.2.0.302 | The VSNI Team (2023) |
| babardown | 0.0.0.9000 | Salmon (2024) |
| base | 4.4.1 | R Core Team (2024) |
| bayesplot | 1.11.1 | Gabry et al. (2019); Gabry and Mahr (2024) |
| boot | 1.3.31 | A. C. Davison and D. V. Hinkley (1997); Angelo Canty and B. D. Ripley (2024) |
| brio | 1.1.5 | Hester and Csárdi (2024) |
| brms | 2.21.0 | Bürkner (2017); Bürkner (2018); Bürkner (2021) |
| broom.mixed | 0.2.9.5 | Bolker and Robinson (2024) |
| car | 3.1.2 | Fox and Weisberg (2019a) |
| coda | 0.19.4.1 | Plummer et al. (2006) |
| DHARMa | 0.4.6 | Hartig (2022) |
| digest | 0.6.37 | Eddelbuettel (2024) |
| effects | 4.2.2 | Fox (2003); Fox and Hong (2009); Fox and Weisberg (2018); Fox and Weisberg (2019b) |
| emoji | 15.0 | Hvitfeldt (2022) |
| factoextra | 1.0.7 | Kassambara and Mundt (2020) |
| FactoMineR | 2.11 | Lê et al. (2008) |

Table 22.1.: R packages used in this book

| Package | Version | Citation |
|----------------|----------|---|
| GGally | 2.2.1 | Schloerke et al. (2024) |
| ggcleveland | 0.1.0 | Prunello and Mari (2021) |
| ggfortify | 0.4.17 | Tang et al. (2016); Horikoshi and Tang (2018) |
| ggpubr | 0.6.0 | Kassambara (2023) |
| grateful | 0.2.10 | Rodriguez-Sanchez and Jackson (2023) |
| gt | 0.11.0 | Iannone et al. (2024) |
| hexbin | 1.28.4 | Carr et al. (2024) |
| kableExtra | 1.4.0 | Zhu (2024) |
| knitr | 1.48 | Xie (2014); Xie (2015); Xie (2024) |
| lattice | 0.22.6 | Sarkar (2008) |
| lme4 | 1.1.35.5 | Bates et al. (2015) |
| lmerTest | 3.1.3 | Kuznetsova et al. (2017) |
| lmPerm | 2.1.0 | Wheeler and Torchiano (2016) |
| lmtest | 0.9.40 | Zeileis and Hothorn (2002) |
| MASS | 7.3.61 | Venables and Ripley (2002) |
| MCMCglmm | 2.36 | Hadfield (2010) |
| memoise | 2.0.1 | Wickham et al. (2021) |
| multcomp | 1.4.26 | Hothorn et al. (2008) |
| MuMIn | 1.48.4 | Bartoń (2024) |
| mvtnorm | 1.3.1 | Genz and Bretz (2009) |
| nadiv | 2.18.0 | Wolak (2012) |
| palmerpenguins | 0.1.1 | Horst et al. (2020) |
| patchwork | 1.2.0 | Pedersen (2024) |
| performance | 0.12.3 | Lüdecke et al. (2021) |
| pwr | 1.3.0 | Champely (2020) |
| quantreg | 5.98 | Koenker (2024) |
| reshape2 | 1.4.4 | Wickham (2007) |
| rmarkdown | 2.28 | Xie et al. (2018); Xie et al. (2020); Allaire et al. (2024) |
| rptR | 0.9.22 | Stoffel et al. (2017) |
| shiny | 1.9.1 | Chang et al. (2024) |

Table 22.1.: R packages used in this book

| Package | Version | Citation |
|------------|------------|---|
| simpleboot | 1.1.8 | Peng (2024) |
| tidyverse | 2.0.0 | Wickham et al. (2019) |
| tinkr | 0.2.0.9000 | Salmon et al. (2024) |
| vcd | 1.4.13 | Meyer et al. (2006); Zeileis et al. (2007); Meyer et al. (2024) |
| vcdExtra | 0.8.5 | Friendly (2023) |
| vioplot | 0.5.0 | Adler et al. (2024) |
| withr | 3.0.1 | Hester et al. (2024) |
| yaml | 2.3.10 | Garbett et al. (2024) |

Bibliography

- A. C. Davison, and D. V. Hinkley. 1997. [Bootstrap methods and their applications](#). Cambridge University Press, Cambridge.
- Adler, D., S. T. Kelly, T. Elliott, and J. Adamson. 2024. [vioplot: Violin plot](#).
- Allaire, J., Y. Xie, C. Dervieux, J. McPherson, J. Luraschi, K. Ushey, A. Atkins, H. Wickham, J. Cheng, W. Chang, and R. Iannone. 2024. [rmarkdown: Dynamic documents for r](#).
- Angelo Canty, and B. D. Ripley. 2024. [boot: Bootstrap r \(s-plus\) functions](#).
- Banta, J. A., M. H. H. Stevens, and M. Pigliucci. 2010. [A comprehensive test of the “limiting resources” framework applied to plant tolerance to apical meristem damage](#). Oikos 119:359–369.
- Bartoń, K. 2024. [MuMIn: Multi-model inference](#).
- Bates, D., M. Mächler, B. Bolker, and S. Walker. 2015. [Fitting linear mixed-effects models using lme4](#). Journal of Statistical Software 67:1–48.
- Bolker, B. M., M. E. Brooks, C. J. Clark, S. W. Geange, J. R. Poulsen, M. H. H. Stevens, and J.-S. S. White. 2009. Generalized linear mixed models: A practical guide for ecology and evolution. Trends in Ecology and Evolution 24:127–135.
- Bolker, B., and D. Robinson. 2024. [broom.mixed: Tidying methods for mixed models](#).
- Bürkner, P.-C. 2017. [brms: An R package for Bayesian multilevel models using Stan](#). Journal of Statistical Software 80:1–28.
- Bürkner, P.-C. 2018. [Advanced Bayesian multilevel modeling with the R package brms](#). The R Journal 10:395–411.

- Bürkner, P.-C. 2021. [Bayesian item response modeling in R with brms and Stan](#). Journal of Statistical Software 100:1–54.
- Carr, D., ported by Nicholas Lewin-Koh, M. Maechler, and contains copies of lattice functions written by Deepayan Sarkar. 2024. [hexbin: Hexagonal binning routines](#).
- Champely, S. 2020. [pwr: Basic functions for power analysis](#).
- Chang, W., J. Cheng, J. Allaire, C. Sievert, B. Schloerke, Y. Xie, J. Allen, J. McPherson, A. Dipert, and B. Borges. 2024. [shiny: Web application framework for r](#).
- Douglas, A. 2023. [An introduction to r](#).
- Eddelbuettel, D. 2024. [digest: Create compact hash digests of r objects](#).
- Elston, D. A., R. Moss, T. Boulinier, C. Arrowsmith, and X. Lambin. 2001. Analysis of aggregation, a worked example: Numbers of ticks on red grouse chicks. *Parasitology* 122:563–569.
- Fox, J. 2003. [Effect displays in R for generalised linear models](#). Journal of Statistical Software 8:1–27.
- Fox, J., and J. Hong. 2009. [Effect displays in R for multinomial and proportional-odds logit models: Extensions to the effects package](#). Journal of Statistical Software 32:1–24.
- Fox, J., and S. Weisberg. 2018. [Visualizing fit and lack of fit in complex regression models with predictor effect plots and partial residuals](#). Journal of Statistical Software 87:1–27.
- Fox, J., and S. Weisberg. 2019a. [An R companion to applied regression](#). Third. Sage, Thousand Oaks CA.
- Fox, J., and S. Weisberg. 2019b. [An r companion to applied regression](#). 3rd edition. Sage, Thousand Oaks CA.
- Friendly, M. 2023. [vcdExtra: “vcd” extensions and additions](#).
- Gabry, J., and T. Mahr. 2024. [bayesplot: Plotting for bayesian models](#).
- Gabry, J., D. Simpson, A. Vehtari, M. Betancourt, and A. Gelman. 2019. [Visualization in bayesian workflow](#). *J. R. Stat. Soc. A* 182:389–402.
- Garbett, S. P., J. Stephens, K. Simonov, Y. Xie, Z. Dong, H. Wickham, J. Horner, reikoch, W. Beasley, B. O’Connor, G. R. Warnes, M. Quinn, Z. N. Kamvar, and C. Gao. 2024. [yaml: Methods to convert r data to YAML and back](#).
- Genz, A., and F. Bretz. 2009. Computation of multivariate normal and t probabilities. Springer-Verlag, Heidelberg.
- Hadfield, J. D. 2010. [MCMC methods for multi-response generalized linear mixed models: The MCMCglmm R package](#). Journal of Statistical Software 33:1–22.
- Hadfield, J. D., A. J. Wilson, D. Garant, B. C. Sheldon, and L. E. Kruuk. 2010. The Misuse of BLUP in Ecology and Evolution. *American Naturalist* 175:116–125.
- Hartig, F. 2022. [DHARMA: Residual diagnostics for hierarchical \(multi-level / mixed\) regression models](#).
- Hester, J., and G. Csárdi. 2024. [brio: Basic r input output](#).
- Hester, J., L. Henry, K. Müller, K. Ushey, H. Wickham, and W. Chang. 2024. [withr: Run code “With” temporarily modified global state](#).

- Horikoshi, M., and Y. Tang. 2018. [ggfortify: Data visualization tools for statistical analysis results](#).
- Horst, A. M., A. P. Hill, and K. B. Gorman. 2020. [palmerpenguins: Palmer archipelago \(antarctica\) penguin data](#).
- Hothorn, T., F. Bretz, and P. Westfall. 2008. Simultaneous inference in general parametric models. *Biometrical Journal* 50:346–363.
- Houslay, T. M., and A. J. Wilson. 2017. [Avoiding the misuse of BLUP in behavioural ecology](#). *Behavioral Ecology* 28:948–952.
- Hvitfeldt, E. 2022. [emoji: Data and function to work with emojis](#).
- Iannone, R., J. Cheng, B. Schloerke, E. Hughes, A. Lauer, J. Seo, K. Brevoort, and O. Roy. 2024. [gt: Easily create presentation-ready display tables](#).
- Kassambara, A. 2023. [ggpubr: “ggplot2” based publication ready plots](#).
- Kassambara, A., and F. Mundt. 2020. [factoextra: Extract and visualize the results of multivariate data analyses](#).
- Koenker, R. 2024. [quantreg: Quantile regression](#).
- Kuznetsova, A., P. B. Brockhoff, and R. H. B. Christensen. 2017. [lmerTest package: Tests in linear mixed effects models](#). *Journal of Statistical Software* 82:1–26.
- Lê, S., J. Josse, and F. Husson. 2008. [FactoMineR: A package for multivariate analysis](#). *Journal of Statistical Software* 25:1–18.
- Lüdecke, D., M. S. Ben-Shachar, I. Patil, P. Waggoner, and D. Makowski. 2021. [performance: An R package for assessment, comparison and testing of statistical models](#). *Journal of Open Source Software* 6:3139.
- Martin, J. 1219. Another lasagna recipe from medieval times. *Journal of Lasagna* 4:1686.
- Martin, J. 2200. A silly example. Chapman; Hall/CRC, Boca Raton, Florida.
- Meyer, D., A. Zeileis, and K. Hornik. 2006. [The strucplot framework: Visualizing multi-way contingency tables with vcd](#). *Journal of Statistical Software* 17:1–48.
- Meyer, D., A. Zeileis, K. Hornik, and M. Friendly. 2024. [vcd: Visualizing categorical data](#).
- Pedersen, T. L. 2024. [patchwork: The composer of plots](#).
- Peng, R. D. 2024. [simpleboot: Simple bootstrap routines](#).
- Plummer, M., N. Best, K. Cowles, and K. Vines. 2006. [CODA: Convergence diagnosis and output analysis for MCMC](#). *R News* 6:7–11.
- Prunello, M., and G. Mari. 2021. [ggcleveland: Implementation of plots from cleveland’s visualizing data book](#).
- R Core Team. 2024. [R: A language and environment for statistical computing](#). R Foundation for Statistical Computing, Vienna, Austria.
- Rodriguez-Sanchez, F., and C. P. Jackson. 2023. [grateful: Facilitate citation of r packages](#).
- Salmon, M. 2024. [babardown: Helpers for automatic translation of markdown-based content](#).
- Salmon, M., Z. N. Kamvar, and J. Ooms. 2024. [tinkr: Cast “\(R\)Markdown” files to “XML” and back again](#).

- Sarkar, D. 2008. [Lattice: Multivariate data visualization with r](#). Springer, New York.
- Schloerke, B., D. Cook, J. Larmarange, F. Briatte, M. Marbach, E. Thoen, A. Elberg, and J. Crowley. 2024. [GGally: Extension to “ggplot2”](#).
- Stan Development Team. 2021. [Stan modeling language users guide and reference manual, 2.26](#).
- Stan Development Team. 2024. [RStan: The R interface to Stan](#).
- Stoffel, M. A., S. Nakagawa, and H. Schielzeth. 2017. [rptR: Repeatability estimation and variance decomposition by generalized linear mixed-effects models](#). Methods in Ecology and Evolution 8:1639???1644.
- Tang, Y., M. Horikoshi, and W. Li. 2016. [ggfortify: Unified interface to visualize statistical result of popular r packages](#). The R Journal 8:474–485.
- The VSNI Team. 2023. [asreml: Fits linear mixed models using REML](#).
- Venables, W. N., and B. D. Ripley. 2002. [Modern applied statistics with s](#). Fourth. Springer, New York.
- Wheeler, B., and M. Torchiano. 2016. [lmPerm: Permutation tests for linear models](#).
- Wickham, H. 2007. [Reshaping data with the reshape package](#). Journal of Statistical Software 21:1–20.
- Wickham, H., M. Averick, J. Bryan, W. Chang, L. D. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester, M. Kuhn, T. L. Pedersen, E. Miller, S. M. Bache, K. Müller, J. Ooms, D. Robinson, D. P. Seidel, V. Spinu, K. Takahashi, D. Vaughan, C. Wilke, K. Woo, and H. Yutani. 2019. [Welcome to the tidyverse](#). Journal of Open Source Software 4:1686.
- Wickham, H., J. Hester, W. Chang, K. Müller, and D. Cook. 2021. [memoise: “Memoisation” of functions](#).
- Wilkinson, L. 2005. [The Grammar of Graphics](#). Springer Science & Business Media.
- Wolak, M. E. 2012. [nadv: An R package to create relatedness matrices for estimating non-additive genetic variances in animal models](#). Methods in Ecology and Evolution 3:792–796.
- Xie, Y. 2014. knitr: A comprehensive tool for reproducible research in R. *in* V. Stodden, F. Leisch, and R. D. Peng, editors. [Implementing reproducible computational research](#). Chapman; Hall/CRC.
- Xie, Y. 2015. [Dynamic documents with R and knitr](#). 2nd edition. Chapman; Hall/CRC, Boca Raton, Florida.
- Xie, Y. 2024. [knitr: A general-purpose package for dynamic report generation in r](#).
- Xie, Y., J. J. Allaire, and G. Grolemund. 2018. [R markdown: The definitive guide](#). Chapman; Hall/CRC, Boca Raton, Florida.
- Xie, Y., C. Dervieux, and E. Riederer. 2020. [R markdown cookbook](#). Chapman; Hall/CRC, Boca Raton, Florida.
- Zeileis, A., and T. Hothorn. 2002. [Diagnostic checking in regression relationships](#). R News 2:7–10.
- Zeileis, A., D. Meyer, and K. Hornik. 2007. [Residual-based shadings for visualizing \(conditional\) independence](#). Journal of Computational and Graphical Statistics 16:507–525.
- Zhu, H. 2024. [kableExtra: Construct complex table with “kable” and pipe syntax](#).

Appendix A

Data used in this book

A.1. All in one zip file

All the data, code, and r-objects used in the book in a [zip file](#)

A.2. All the data files

- [age.csv](#)
- [anc1dat.csv](#)
- [anc3dat.csv](#)
- [atmosphere.txt](#)
- [Banta_TotalFruits.csv](#)
- [Biston_pd_1.csv](#)
- [Biston_pd_2.csv](#)
- [Biston_student.csv](#)
- [Biston.postdoc.csv](#)
- [Biston.prof.csv](#)
- [Dam10dat.csv](#)
- [dragons.csv](#)
- [ErablesGatineau.csv](#)
- [gala.txt](#)
- [hypoxia.uottawa.csv](#)
- [JacobsenDangles_1.csv](#)

- JacobsenDangles_2.csv
- loglin.csv
- mouflon.csv
- Mregdat.csv
- nematodes.csv
- nestdat.csv
- nr2wdat.csv
- pollution.txt
- Rway_enfR_data_code.zip
- salmonella.csv
- simulies.csv
- simuliidae.csv
- skulldat_2020.csv
- skulldat-rm.csv
- smoking.txt
- Stu2mdat.csv
- Stu2wdat.csv
- sturgdat.csv
- sturgeon.csv
- unicorns_aggression.csv
- unicorns.csv
- unicorns.txt
- unicorns.xlsx
- USPopSurvey.csv
- wmc2daf2.csv
- wmcdat2.csv

A.3. R code used in the book and slides

- book-fnc.R
- extra_funs.R
- glmm_simdev.rda

Appendix B

Installing Quarto and LateX

Installing Quarto on your computer is pretty straight forward and should be painless. If you're getting started with Quarto we suggest that you use RStudio but of course RStudio is not required and there are other options available. You will also need to install some additional software if you want to render your Quarto documents to PDF format.

This guide assumes you have already installed [R](#) and an IDE ([RStudio IDE](#) or [VSCode](#)). An IDE is not required but recommended, because it makes it easier to work with Quarto. If you don't have RStudio IDE installed, you will also have to install [Pandoc](#). If you have RStudio installed there is no need to install Pandoc separately because it's bundled with RStudio.

Next you need to install [Quarto](#). It is really straightforward, just download and install from the files specific to your OS (Windows, Mac or linux).

You should also install the `quarto` 📦 package using:

```
install.packages("quarto", dep = TRUE)
```

The `dep = TRUE` argument will also install a bunch of additional R packages on which the `quarto` 📦 depends.

Finally you can install the `rmarkdown` 📦 package but it is not strictly required if using Quarto.

```
install.packages("rmarkdown", dep = TRUE)
```

If you want to generate PDF output, you will need to install [LATEX](#). For R Markdown users who have not installed [LATEX](#) before, we recommend that you install [TinyTeX](#). You can install TinyTeX from within R using the `tinytex` 📦 package with the following code:

```
install.packages("tinytex")
tinytex::install_tinytex() # install TinyTeX
```

TinyTeX is a lightweight, portable, cross-platform, and easy-to-maintain \LaTeX distribution. The R companion package `tinytex`  can help you automatically install missing \LaTeX packages when compiling \LaTeX or R Markdown documents to PDF.

B.1. MS Windows

An alternative option would be to install MiKTeX instead. You can download the latest distribution of [MiKTeX](#). Installing MiKTeX is pretty straight forward, but it can sometimes be a pain to get it to play nicely with RStudio. If at all possible we recommend that you use TinyTeX.

B.2. Mac OSX

If for some reason TinyTeX does not work on your Mac computer then you can try to install MacTeX instead. You can download the latest version of [MacTeX here](#).

B.3. Linux

An alternative to TinyTeX on linux would be to use a full fledge distribution of \LaTeX such as [TexLive](#)