

Effective Data Science

Zak Varty

2024-02-11

Table of contents

About this Course	11
Schedule	11
Learning outcomes	12
Allocation of Study Hours	12
Assessment Structure	12
Acknowledgements	13
I Effective Workflows	14
Introduction	15
1 Organising your work	16
1.1 What are we trying to do?	16
1.2 An R Focused Approach	17
1.3 One Project = One Directory	17
1.4 Properties of a Well-Organised Project	18
1.4.1 Portability	19
1.4.2 Version Control Friendly	19
1.4.3 Reproducibility	20
1.4.4 IDE Friendly	20
1.5 Project Structure	21
1.5.1 README.md	23
1.5.2 Inside the README	24
1.5.3 data	25
1.5.4 src	27
1.5.5 tests	28
1.5.6 analyses	29
1.5.7 outputs	30
1.5.8 reports	31
1.5.9 make file	33
1.6 Wrapping up	34
1.7 Session Information	35
2 Naming Files	36
2.1 Introduction	36

2.2	Naming Files	36
2.2.1	What do we want from our file names?	37
2.2.2	Machine Readable	37
2.2.3	Order Friendly	39
2.2.4	Human Readable	40
2.2.5	Naming Files - Summary	41
2.3	File Extensions and Where You Work	41
2.3.1	Open Source vs Proprietary File Types	42
2.3.2	Inside Data Files	43
2.3.3	A Note on Notebooks	46
2.3.4	File Extensions and Where You Code	46
2.3.5	Summary	47
2.4	Session Information	48
3	Code	49
3.1	Introduction	49
3.2	Functional Programming	49
3.2.1	The Pipe Operator	50
3.2.2	When not to pipe	51
3.3	Object Oriented Programming	52
3.3.1	OOP Philosophy	52
3.3.2	OOP Example	53
3.4	Structuring R Script Headers	53
3.5	Portable File paths with <code>{here}</code>	54
3.6	Code Body	55
3.6.1	Comments	55
3.6.2	Objects are Nouns	56
3.6.3	Functions are Verbs	57
3.6.4	Casing Consistantly	58
3.6.5	Style Guide Summary	58
3.7	Further Tips for Friendly Coding	58
3.8	Reduce, Reuse, Recycle	59
3.8.1	DRY Coding	59
3.8.2	Rememer how to use your own code	60
3.8.3	<code>{roxygen2}</code> for documentation	60
3.8.4	An <code>{roxygen2}</code> example	61
3.8.5	Checking Your Code	62
3.8.6	An Informal Testing Workflow	62
3.8.7	A Formal Testing Workflow	63
3.9	Summary	64
3.10	Session Information	64

Workflows Checklist	66
Videos / Chapters	66
Reading	66
Tasks	66
Live Session	67
II Acquiring and Sharing Data	68
Introduction	69
4 Tabular Data	70
4.1 Loading Tabular Data	70
4.1.1 Base R	70
4.1.2 {readr}	71
4.1.3 WTF: Tibbles	72
4.1.4 Other {readr} functions	73
4.1.5 Need for Speed	74
4.2 Tidy Data	74
4.2.1 Wide vs. Tall Data	74
4.2.2 Tidy What?	76
4.2.3 Example - Tidy Longer	77
4.2.4 Example - Tidy Wider	77
4.2.5 Other helpful functions	78
4.2.6 Missing Data	79
4.3 Wrapping Up	79
4.4 Session Information	79
5 Web Scraping	80
5.1 Scraping webpage data using {rvest}	80
5.2 What is a webpage?	80
5.3 HTML	81
5.3.1 HTML elements	81
5.3.2 Important HTML Elements	81
5.4 HTML Attributes	82
5.5 CSS Selectors	82
5.6 Which Attributes and Selectors Do You Need?	83
5.7 Reading HTML with {rvest}	83
5.8 Extracting HTML elements	84
5.9 Extracting Data From HTML Elements	85
5.9.1 Extracting text	85
5.9.2 Extracting Attributes	86
5.9.3 Extracting tables	86
5.10 Tip for Building Tibbles	88

5.11	Session Information	89
6	APIs	91
6.1	Acquiring Data Via an API	91
6.2	Why do I need to know about APIs?	91
6.3	What is an API?	92
6.4	HTTP	93
6.5	HTTP Requests	93
6.5.1	URL	93
6.5.2	Method	94
6.5.3	Header	94
6.5.4	Body	94
6.6	HTTP Responses	94
6.6.1	Status Codes	95
6.7	Authentication	95
6.7.1	Basic Authentication	95
6.7.2	API Key Authentication	96
6.8	API wrappers	96
6.9	{geonames} wrapper	96
6.9.1	Set up	96
6.9.2	Keep it Secret, Keep it Safe	98
6.9.3	Using {geonames}	99
6.10	What if there is no wrapper?	100
6.11	Wrapping up	103
6.12	Session Information	104
	Checklist	105
	Videos / Chapters	105
	Reading	105
	Tasks	105
	Live Session	106
III	Data Exploration and Visualisation	107
	Introduction	108
7	Data Wrangling	109
7.1	What is Data Wrangling?	109
7.2	Example Data Sets	111
7.3	Viewing Your Data	111
7.3.1	view()	111
7.3.2	head()	112
7.3.3	str()	112

7.3.4	<code>names()</code>	112
7.4	Renaming Variables	113
7.4.1	<code>colnames()</code>	113
7.4.2	<code>dplyr::rename()</code>	114
7.5	Subsetting	114
7.5.1	Base R	114
7.5.2	<code>filter()</code> and <code>select()</code>	117
7.6	Creating New Variables	119
7.6.1	Base R	119
7.6.2	<code>dplyr::mutate()</code>	120
7.6.3	<code>rownames_to_column()</code>	120
7.6.4	<code>rowids_to_column()</code>	121
7.7	Summaries	122
7.8	Grouped Operations	123
7.8.1	Ungrouping	124
7.9	Reordering Factors	126
7.10	Be Aware: Factors	127
7.11	Be Aware: Strings	127
7.12	Be Aware: Date-Times	129
7.13	Be Aware: Relational Data	129
7.14	Wrapping up	132
7.15	Session Information	132
8	Exploratory Data Analysis	133
8.1	Introduction	133
8.2	Get to know your data	133
8.3	Start a conversation	134
8.3.1	Communicating with specialists	135
8.3.2	Communicating with project manager	135
8.4	Scope Your Project	135
8.5	Investigate Your Data	136
8.6	What is not EDA?	136
8.7	Issue: Forking Paths	139
8.8	Correction Methods	142
8.9	Learning More	144
9	Data Visualisation	146
9.1	Introduction	146
9.1.1	More than a pretty picture	146
9.2	Your Tools	147
9.2.1	Picking the right tool for the job	147
9.2.2	Analogue or Digital	148
9.2.3	<code>ggplot2</code>	149

9.3	Your Medium	151
9.3.1	Where is your visualisation going?	151
9.3.2	File Types	152
9.4	Your Audience	154
9.4.1	Know Your Audience	154
9.4.2	Preattentive Attributes	156
9.4.3	Example: First Impressions Count	156
9.4.4	Visual Perception	158
9.4.5	Alt-text, Titles and Captions	160
9.5	Your Story	162
9.6	Your Guidelines	165
9.6.1	Standardise and Document	165
9.6.2	Example Style Guides	166
9.7	Wrapping Up	167
	Checklist	168
	Videos / Chapters	168
	Reading	168
	Activities	168
	Live Session	169
	IV Preparing for Production	170
	V Introduction	171
	10 Reproducibility	173
10.1	The Data Scientific Method	173
10.2	Issue: Multiple, Dependent Tests	174
10.3	Issues: p -hacking and Publication Bias	176
10.4	Reproducibility	177
10.5	Replicability	178
10.5.1	Shadow deployment	179
10.5.2	Parallel deployment	180
10.6	Reproduction and Replication in Statistical Data Science	180
10.6.1	Monte Carlo Methods	180
10.6.2	Optimisation	182
10.6.3	(Pseudo-)Random Numbers	183
10.7	Beware	184
10.8	Wrapping Up	184

11 Explainability	186
11.1 What are we explaining and to whom?	186
11.2 Explaining a Decision Tree	187
11.3 Explaining Regression Models	188
11.4 Example: Cherrywood regression	188
11.5 Simpson’s Paradox	192
11.6 What hope do we have?	192
11.6.1 Permutation Testing	194
11.6.2 Meta-modelling	194
11.6.3 Aggregating Meta-models	196
11.7 Wrapping Up	196
12 Scalability	199
12.1 Scalability and Production	199
12.1.1 Example: Bayesian Inference	199
12.1.2 Knowing when to worry	200
12.1.3 Our Focus	200
12.2 Basics of Code Profiling	201
12.2.1 R as a stopwatch	201
12.3 Profiling Your Code	202
12.3.1 Profiling: Toy Example	202
12.4 Notes on Time Profiling	204
12.4.1 Source code and compiled functions	204
12.5 Memory Profiling	206
12.6 Tips to work at scale	208
12.6.1 Vectorise	208
12.6.2 Linear Algebra	208
12.7 For loops in disguise	209
12.7.1 The apply family	209
12.7.2 {purrr}	209
12.8 Easy parallelisation with furr	211
12.9 Sometimes R doesn’t cut it	212
12.10 Wrapping up	213
Summary	213
Help!	213
Checklist	214
Videos / Chapters	214
Reading	214
Activities	214
Live Session	215

VI Data Science Ethics	216
VII Introduction	217
13 Privacy	219
13.1 Privacy and Data Science	219
13.2 Privacy as a Human Right	220
13.2.1 Article 12 of the Universal Declaration of Human Rights	220
13.3 Data Privacy and the European Union	220
13.3.1 General Date Protection Regulation (2018)	220
13.4 Privacy: Key Terms	221
13.5 Measuring Privacy	222
13.5.1 Pseudo-identifiers and k -anonymity	222
13.5.2 k -anonymity example	222
13.6 Improving Privacy	223
13.7 Breaking k -anonymity	224
13.8 Cautionary tales	225
13.8.1 Massachussets Medical Data	225
13.8.2 Netflix Competition	225
13.9 Wrapping Up	226
14 Fairness	229
14.1 Fairness and the Data Revolution	229
14.2 You are Your Data	230
14.3 Forbidden Predictors	232
14.4 Measuring Fairness	233
14.4.1 Demographic Parity	233
14.4.2 Equal Opportunity	234
14.4.3 Equal Odds	234
14.4.4 Predictive Parity	234
14.5 Metric Madness	235
14.6 Modelling Fairly	236
14.6.1 Fairness Aware Loss Functions	236
14.6.2 Other Approaches	237
14.7 Wrapping Up	238
15 Codes of Conduct	239
15.1 Data Science: Miracle Cure and Sexiest Job	239
15.2 What could go wrong?	239
15.3 That's not <i>my</i> type of data science	243
15.4 Technological Adoption Relies on Public Trust	244
15.5 A Hippocratic Oath for Data Scientists	245

15.6 Codes of Conduct	246
15.7 Wrapping Up	247
Checklist	248
Videos / Chapters	248
Reading	248
Activities	248
Live Session	249
Appendices	250
A Reading List	250
A.1 Effective Data Science Workflows	250
Core Materials	250
Reference Materials	250
Materials of Interest	251
A.2 Acquiring and Sharing Data	251
Core Materials	251
Reference Materials	252
Materials of Interest	252
A.3 Data Exploration and Visualisation	252
Core Materials	252
Referene Materials	253
Materials of Interest	253
A.4 Preparing for Production	253
Core Materials	253
Reference Materials	253
Materials of Interest	254
A.5 Data Science Ethics	254
Core Materials	254
Reference Materials	254
Materials of Interest	255
References	256

About this Course

Model building and evaluation are necessary but not sufficient skills for the effective practice of data science. In this module you will develop the technical and personal skills that are required to work successfully as a data scientist within an organisation.

During this module you will critically explore how to:

- effectively scope and manage a data science project;
- work openly and reproducibly;
- efficiently acquire, manipulate, and present data;
- interpret and explain your work for a variety of stakeholders;
- ensure that your work can be put into production;
- assess the ethical implications of your work as a data scientist.

This interdisciplinary course will draw from fields including statistics, computing, management science and data ethics. Each topic will be investigated through a selection of lecture videos, conference presentations and academic papers, supported by hands-on lab exercises and readings on industry best-practices as published by recognised professional bodies.

Schedule

These notes are intended for students on the course **MATH70076: Data Science** in the academic year 2023/24.

As the course is scheduled to take place over five weeks, the suggested schedule is:

- 1st week: effective data science workflows;
- 2nd week: acquiring and sharing data;
- 3rd week: exploratory data analysis and visualisation;
- 4th week: preparing for production;
- 5th week: ethics and context of data science.

An alternative pdf version of these notes may be downloaded [here](#). Please be aware that this pdf version is secondary to this course webpage and will be updated less frequently.

Learning outcomes

On successful completion of this module students should be able to:

1. Independently scope and manage a data science project;
2. Source data from the internet through web scraping and APIs;
3. Clean, explore and visualise data, justifying and documenting the decisions made;
4. Evaluate the need for (and implement) approaches that are explainable, reproducible and scalable;
5. Appraise the ethical implications of a data science projects, particularly the risks of compromising privacy or fairness and the potential to cause harm.

Allocation of Study Hours

Lectures: 10 Hours (2 hours per week)

Group Teaching: 5 Hours (1 hour per week)

Lab / Practical: 10 hours (2 hours per week)

Independent Study: 100 hours (11 hours per week + 45 hours coursework)

Drop-In Sessions: Each week there will be a 2-hour optional drop-in session to address any questions about the course or material. This is where you can get support from the course lecturer or GTA on the topics covered each week, either individually or in small groups.

These will be held on Fridays 15:00-17:00 in Huxley 711C.

Office Hours: Additionally, there will be an office hour each week. This is a weekly opportunity for 1-1 discussion with the course lecturer to address any individual questions, concerns or problems that you might have. These meetings can be in person or on Teams and can be academic (relating to course content or progress) or pastoral (relating to student well-being) in nature. To book a 1-1 meeting please use the link on the course blackboard page.

Office hours will be held on Mondays 15:00-16:00 in Huxley 6M20. (Week 3 alteration, 14:00-15:00)

Assessment Structure

The course will be assessed entirely by coursework, reflecting the practical and pragmatic nature of the course material.

Coursework 1 (30%): To be completed during the final week of the course, the week commencing 2024-02-12.

Coursework 2 (70%): To be released during the third week of the course and submitted following the examination period in Summer term (2024-01-29 until 2024-05-15 at 13:00).

Acknowledgements

These notes were created by Dr Zak Varty. They were inspired by a previous lecture series by Dr Purvasha Chakravarti at Imperial College London and draw from many resource that were made available by the R community, which are attributed throughout.

Part I

Effective Workflows

Introduction

As a data scientist you will never work alone.

Within a single project as a data scientist it is likely that you will interact with a range of other people, including but not limited to: one or more project managers, stakeholders and subject matter experts. Depending on the type of work that you are doing, these experts might come from a single specialism or form a multi-disciplinary team. To get your project put into production and working at scale you will likely have to collaborate with data engineers. You're also likely to work closely with other data scientists, reviewing one another's work or collaborating on larger projects. Familiarity with the skills, processes and practices that make for effective collaboration is therefore instrumental to being a successful as a data scientist.

The aim for this part of the course is to provide you with a structure on how you organise and perform your work, so that you can be a good collaborator to current colleagues and your future self.

This is going to require a bit more effort upfront, but the benefits will compound over time. You will get more done by wasting less time staring quizzically at messy folders of indecipherable code. You will also gain a reputation of someone who is good to work with. This promotes better professional relationships and greater levels of trust, which can in turn lead to working on more exciting and impactful projects.

1 Organising your work

Welcome to this course on effective data science. This week we'll be considering effective data science workflows. These workflows are ways of progressing a project that will help you to produce high quality work and help to make you a better collaborator.

In this chapter, we'll kick things off by looking at how you can structure data science projects and organise your work. Familiarity with these skills, processes and practices for collaborative working will be instrumental to you become a successful data scientist.

1.1 What are we trying to do?

First, let's consider why we want to provide our data science projects with some sense of structure and organization.

As a data scientist you'll never work alone. Within a single project you'll interact with a whole range of other people. This might be a project manager, one or more business stakeholders or a variety of subject matter experts. These experts might be trained as sociologists, chemists, or civil servants depending on the exact type of data science work that you're doing.

To then get your project put into use and working at scale you'll have to collaborate with data engineers. You'll also likely work closely with other data scientists. For smaller projects this might be to act as reviewers for one another's work. For larger projects, working collaboratively will allow you to tackle larger challenges. These are the sorts of project that wouldn't be feasible alone, because of the inherent limitations on the time and skill of any one individual person.

Even if you work in a small organization, where you're the only data scientist, then working in a way that is optimised for collaboration will pay dividends over time. This is because when you inevitably return to your current project in several weeks, months or years. At that point you will have forgotten almost everything that you did the first time around, let alone why you chose to do it that way. You'll have forgotten about not only the decisions that you made, but also alternative options that you decided against.

This is exactly like working with a current colleague who has poor communication skills and working practices. Nobody wants to be that colleague to somebody else, let alone to their future self. Treating that future self like a current collaborator will make you a kind colleague and a pleasure to work with.

In addition to this, you'll get more done by wasting less time staring quizzically at a mess of folders and indecipherable code. You'll also get a reputation as someone who is well-organized and enjoyable to work with. This promotes better professional relationships and greater levels of trust within your team. These can then, in turn, lead to you working on more exciting and more impactful projects in the future.

The aim of this week is to provide you with a guiding structure on how to organize and perform your work. None of this is going to be particularly difficult or onerous. However it will require some up-front effort and daily discipline to maintain. As with brushing your teeth and flossing, the daily effort required is not large but the benefits compound over time.

1.2 An R Focused Approach

The structures and workflows that are recommended here and throughout the rest of this module are focused strongly on a workflow that predominantly uses R, markdown and LaTeX.

Similar techniques, code and software can achieve the same results that I show you here when coding in Python or C, or when writing up projects in Jupyter notebooks or using some other markup language. Similarly, different organizations have their own variations on these best practices that we'll go through together. Often organisations will have extensive guidance on these topics.

The important thing is that once you understand what good habits are and have built them in one programming language or business, then transferring these skills to a new setting is largely a matter of learning some new vocabulary or slightly different syntax.

With that said, let's get going!

1.3 One Project = One Directory

If there's one thing you should take away from this chapter, it's this one Golden Rule:

Every individual project you work on as a data scientist should be in a single, self-contained directory or folder.

This is worth repeating. Every single project that you work on should be self-contained and live in a single directory. An analogy here might be having a separate ring-binder folder for each of your modules on a degree program.

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```



1 project = 1 directory

This one golden rule is deceptively simple.

The first issue here is that it requires a predetermined scope of what is and what isn't going to be covered by this particular project. This seems straightforward but at the outset of the project you often don't know exactly where your project will go, or how it will link to other pieces of work within your organization.

The second issue is that the second law of Thermodynamics applies equally well to project management as it does to tidying your bedroom or the heatdeath of the universe. It takes continual external effort to prevent the contents of this one folder from becoming chaotic and disordered over time.

That being said, having a single directory does have several benefits that more than justify this additional work.

1.4 Properties of a Well-Organised Project

What are the properties that we would like this single, well-organized project to have?

Ideally, we'd like to organize our project so that it has the following properties:

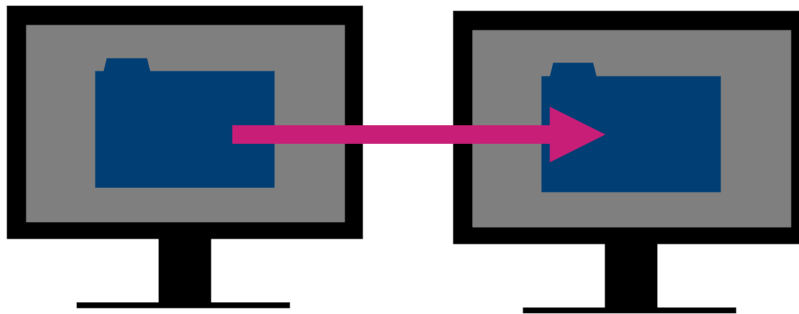
- Portable
- Version Control Friendly
- Reproducible
- IDE friendly.

Don't worry if you haven't heard of some of these terms already. We're going to look at each of them in a little bit of detail.

1.4.1 Portability

A project is said to be portable if it can be easily moved without breaking.

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire  
knitr::include_graphics(path = image_path)
```



This might be a small move, like relocating the directory to a different location on your own computer. It might also mean a moderate move, say to another machine if yours dies just before a big deadline. Alternatively, it might be a large shift - to be used by another person who is running a different operating system.

From this small thought experiment, you can see that there's a full spectrum of how portable a project may or may not need to be.

1.4.2 Version Control Friendly

A project under Version Control has all changes tracked either manually or automatically. This means that snapshots of the project are taken regularly as it gradually develops and evolves over time. Having these snapshots as many, incremental changes are made to the project allow it to be rolled back to a specific previous state if something goes wrong.

A version controlled pattern of working helps to avoid the horrendous state that we have all found ourselves in - renaming `final_version.doc` to `final_final_version.doc` and so on.

By organising your workflow around incremental changes, you acknowledge that no work is ever finally complete. There will always be small changes that need to be done in the future.

1.4.3 Reproducibility

A study is reproducible if you can take the original data and the computer code used to analyze the data and recreate all of the numerical findings from the study.

Broman et al (2017). “Recommendations to Funding Agencies for Supporting Reproducible Research”

In their paper, Broman et al define reproducibility as a project where you can take the original data and code used to perform the analysis and using these we create all of the numerical findings of the study.

This definition leads naturally to several follow-up questions.

Who exactly is *you* in this definition? Does it specifically mean yourself in the future or should someone else with access to all that data and code be able to recreate your findings too? Also, should this reproducibility be limited to just the numerical results? Or should they also be able to create the associated figures, reports and press releases?

Another important question is *when* this project needs to be reproduced. Will it be in a few weeks time or in 10 years time? Do you need to protect your project from changes in dependencies, like new versions of packages or modules? How about different versions of R or Python? Taking this time-scale out even further, what about different operating systems and hardware?

It’s unlikely that you would consider someone handing you a [floppy disk](#) of code that only runs on Windows XP to be acceptably reproducible. Sure, you could probably find a way to get it to work, but that would be an awful lot of effort on your end.

That’s perhaps a bit of an extreme example, but it emphasizes the importance of clearly defining the level of reproducibility that you’re aiming for within every project you work on. This example also highlights the amount of work that can be required to reproduce an analysis, especially after quite some time. It’s important to explicitly think about how we are dividing that effort between ourselves as the original developer and the person trying to reproduce the analysis in the future.

1.4.4 IDE Friendly

Our final desirable property is that we’d like our projects to play nicely with integrated development environments.

When you’re coding document and writing your data science projects it would be possible for you to work entirely in either a plain text editor or typing code directly at the command line. While these approaches to a data science workflow have the benefit of simplicity, they also expect a great deal from you as a data scientist.

These workflows expect you to type everything perfectly accurately every time, that you recall the names and argument orders of every function you use, and that you are constantly aware of the current state of all objects within your working environment.

Integrated Development Environments (IDEs) are applications to help reduce this burden, allowing you to be more effective programmer and data scientist. IDEs offer tools like code completion and highlighting that make your code easier to write and to read. They offer tools for debugging, to fix your code when (not if!) things are going wrong. IDEs also offer environment panes so that you don't have to hold everything in your head all at once. Many IDEs have additional templating facilities. These let you save and reuse snippets of code so that you can avoid typing out repetitive, boilerplate code and introducing errors in the process.

Even if you haven't heard of IDEs before, you've likely already used one. Some common examples might be RStudio for R-users, PyCharm for python users, or Visual Studio as a more language agnostic coding environment. Whichever of these we use, we'd like our project to play nicely with them. This lets us reap their benefits while keeping our project portable, version controlled, and reproducible for someone working with a different set-up.

1.5 Project Structure

We have given a pretty exhaustive argument for why having a single directory for each project is a good idea. Let's now take a look *inside* that directory and define a common starting layout for the content of all of your projects.

Having this sort of project directory template will mean that you'll always know where to find what you're looking for and other members of your team will too. Again, before we start I'll reiterate that we're taking an opinionated approach here and providing a sensible starting point for organizing many projects.

Every project is going to be slightly different and some might require slight alterations to what I suggest here. Indeed, even if you start as I suggest then you might have to adapt your project structure as it develops and grows. I think it's helpful to consider yourself as a tailor when making these changes. I'm providing you with a one size fits all design, that's great for lots of projects but perfect for none of them. It's your job to alter and refine this design for each individual case.

One final caveat before we get started: companies and businesses usually have a house style for how to write and organize your code or projects. If that's the case, then follow the style guide that your business or company uses. The most important thing here is to be consistent at both an individual level and across the entire data science team. It's this consistency that reaps the benefits, not the particular structure.

Okay, so imagine now that you've been assigned an exciting new project and have created a single directory in which to house that project. Here we've, quite imaginatively, called that directory `exciting-new-project`. What do we populate this folder with?

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```



`exciting-new-project`

In the rest of this chapter, I'll define the house-style for organizing the root directory of your data science projects in this module.

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```



Within the project directory there will be some subdirectories. You can tell that these are folders in this file structure diagram because they have a forward-slash following their names. There will also be some files directly in the root directory. One of these is called `readme.md` and the another called either `makefile` or `make.r`. We will explore each of these files and directories in turn.

1.5.1 README.md

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```

The diagram shows the same file structure as above, but with `README.md` highlighted in pink. To the right of the diagram is a window titled "Exciting-new-project/README.md" showing the content of the file:

```
# My Exciting New Project

Here is a short description of what
the project is about.

- Aim 1
- Aim 2
- Aim 3

The code and analysis are structured
as follows:

## data/

This directory contains all raw and
derived data sets. Further information
can be found in
[metadata.txt](./data/metadata.txt)...
```

Let's begin with the readme file. This gives a brief introduction to your project and gives information on what the project aims to do. The readme file should describe how to get started using the project and how to contribute to its development.

The readme is written either in a plain text format so `readme.txt` or in markdown format `readme.md`. The benefit of using markdown is that it allows some light formatting such as sections headers and lists using plain text characters. Here you can see me doing that by using hashes to mark out first and second level headers and using bullet points for a unnumbered list. Whichever format you use, the readme file for your project is always stored in the root directory and is typically named in all uppercase letters.

The readme file should be the first thing that someone who's new to your project reads. By placing the readme in the root directory and capitalising the file name you are increase the visibility of this file and increase the chances of this actually happening.

An additional benefit to keeping the readme in the root directory of your project is that code hosting services like GitHub, GitLab or BitBucket will display the contents of that readme file next to the contents of your project, nicely formatting and displaying any markdown syntax that you use.

When writing the readme it can be useful to imagine that you're writing this for a new, junior team member. The readme file should let them get started with the project and make some simple contributions after reading only that file. It might also link out to more detailed project documentation that helps the new team member toward having a more advanced understanding of the project or making a more complex contribution.

1.5.2 Inside the README

Let's take a moment to cover in more detail what should be included within a readme file.

1.5.2.1 Project name and status

A readme we should include the name of the project, which should be self-explanatory (so nothing like my generic choice of `exciting-new-project`). The readme should also give the project status, which is just a couple of sentences to say whether your project is still under development, the version of the current release or, on the other end of the project life-cycle, if the project is being deprecated or closed.

Following this, we should also include a description of your project. This will state the purpose of your work and to provide, or link to, any additional context or references that visitors aren't assumed to be familiar with.

1.5.2.2 Code dependencies and examples

If your project involves code or depends on other packages then you should give some instruction on how to install those dependencies and run your code. This might just be text but it could also include things like screenshots, code snippets, gifs or a video of the whole process.

It's also a good practice to include some simple examples of how to use the code within your project and the expected results, so that new users can confirm that everything is working on their local instance. Keep the examples as simple and minimal as you can so that new users

For longer or more complicated examples that aren't necessary in this short introductory document you can add links to those in the readme and explain them in detail elsewhere.

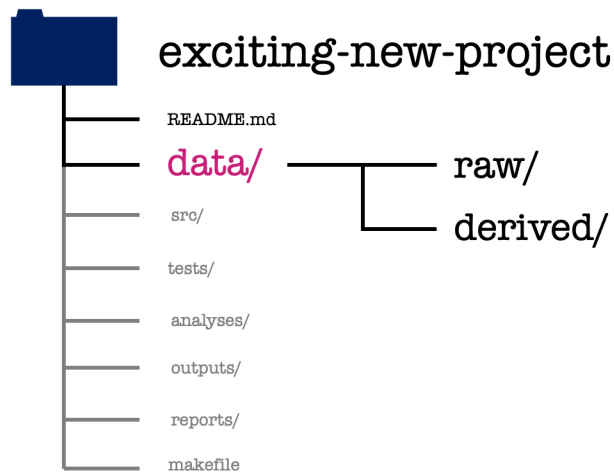
1.5.2.3 How to contribute

There should ideally be a short description of how people can report issues with the project and also how people can get started in resolving those issues or extend the project in some way.

That leads me on to one point that I've forgotten to list here. There there should be a section listing the authors of the work and the license in which under which it's distributed. This is to give credit to all the people who've contributed to your project and the license file then says how other people may use your work. The license declares how other may use your project and whether they have to give direct attribution to your work in any modifications that they use.

1.5.3 data

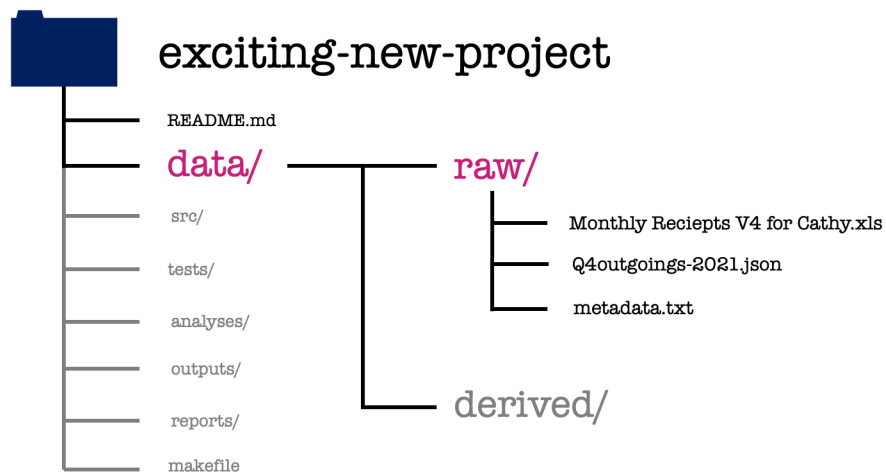
```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```



Moving back to our project structure, next we have the data directory.

The data directory will have two subdirectories one called `raw` and one called `derived`. All data that is not generate as part of your project is stored in the `raw` subdirectory. To ensure that a project is reproducible, data in the `raw` folder should never be edited or modified.

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```

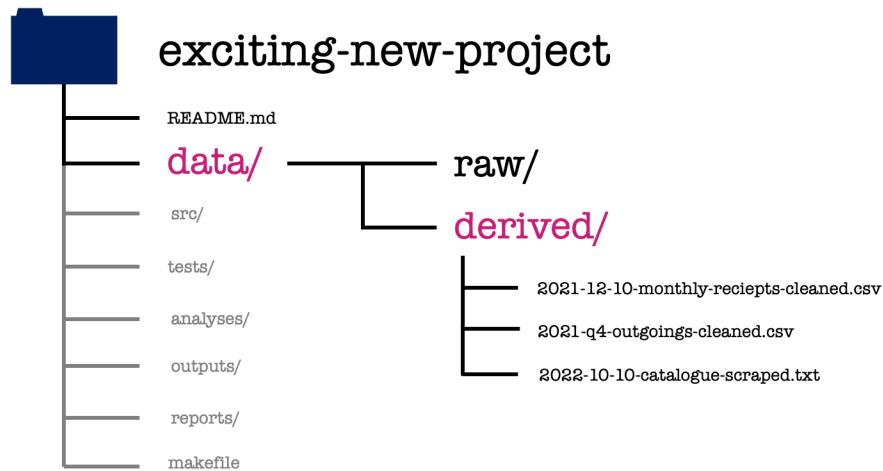


In this example we've got two different data types: an Excel spreadsheet and a JSON file. These files are exacty as we received them from our project stakeholder.

A plain text file, `metadata.txt` explains the contents and interpretation of each of the raw data sets. This metadata should include descriptions of all the measured variables, the units that are recorded in, the date the file was created or acquired, and the source from which it was obtained.

The raw data most likely is not going to be in a form that's amenable to analyzing straight away. To get the data into a more pleasant form to work with, we will need to do some data manipulation and cleaning. Any operations applied when cleaning the raw data should be well documented and the resulting cleaned files are saved within the `derived` data directory for use in our modelling or analysis.

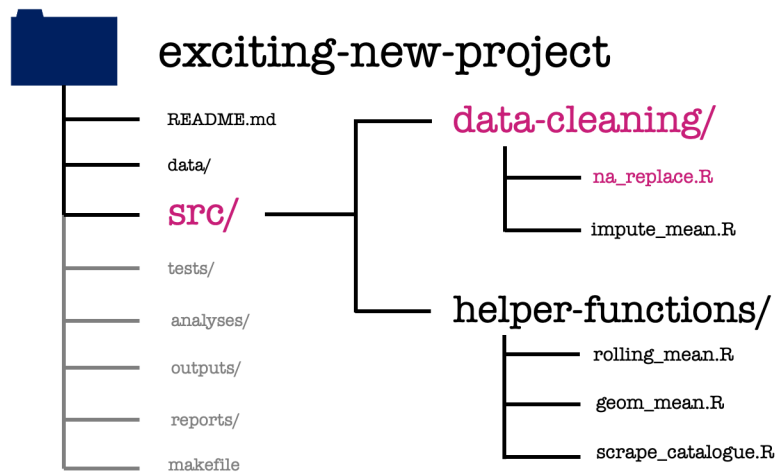
```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```



In our exciting new project, we can see the cleaned versions of the previous data sets that are ready for modelling. There is also a third file in this folder. This is data that we've acquired for ourselves through web scraping, using a script within the project.

1.5.4 src

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```



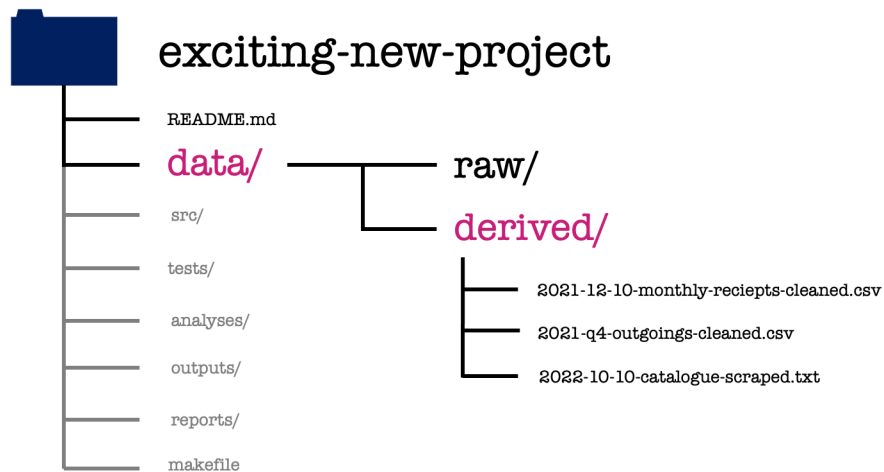
The `src` or source directory contains all the source code for your project. This will typically be the functions that you've written to make the analysis or modelling code more accessible.

In this project, we have saved each function in its own R script and used subdirectories to organise these by their use case. We have two functions used in data cleaning: the first replaces NA values with a user specified value, the second function replaces missing values by the mean of all non-missing values.

We also have three helper functions: the first two calculate the rolling mean and the geometric mean of a given vector, while the third is a function to scrape the web data we previously found in the derived data subdirectory.

1.5.5 tests

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire  
knitr::include_graphics(path = image_path)
```

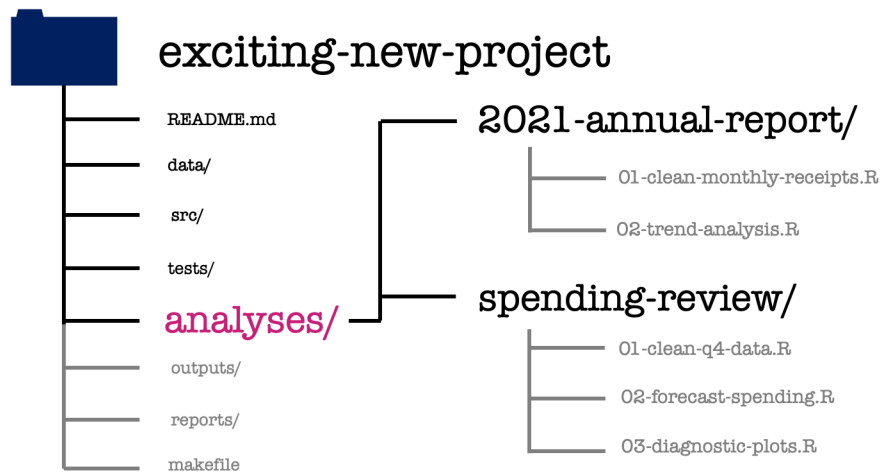


The structure of the `tests/` directory mirrors that of the source directory. Each function file has its own counterpart file of tests.

These test files provide example sets of inputs and the expected outputs for each function. The test files are used to check edge cases of a function and to assure yourself that you didn't break anything while you are fixing some small bug or adding new capabilities to that function.

1.5.6 analyses

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```



The analyses directory contains what you probably think of as the bulk of your data science work. It's going to have one subdirectory for each major analysis that's performed within your project and within each of these there might be a series of steps which we separate into separate scripts.

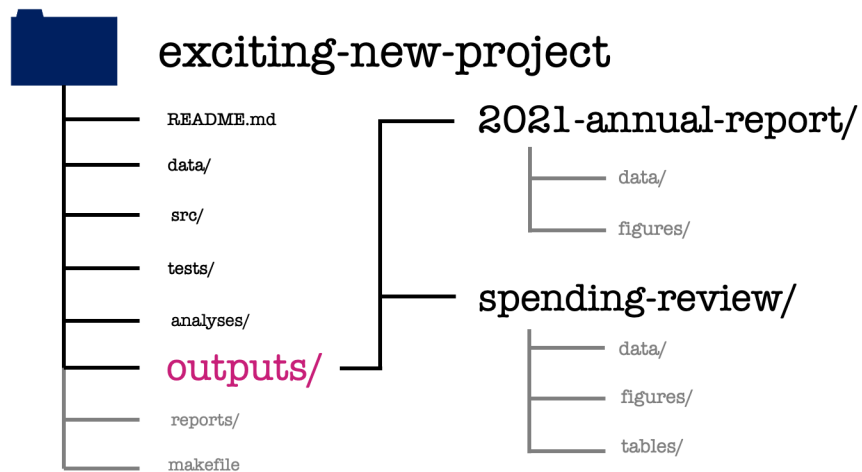
The activity performed at each step is made clear by the name of each script, as is the order in which we're going to perform these steps. Here we can see the scripts used for the 2021 annual report. First is a script used to take the raw monthly receipts and produce the *cleaned* version of the same data set that we saw earlier. This is followed by a trend analysis of this cleaned data set.

Similarly for the spending review we have a data cleaning step, followed by some forecast modelling and finally the production of some diagnostic plots to compare these forecasts.

1.5.7 outputs

```

image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
  
```



The outputs directory has again one subdirectory for each meta-analysis within the project. These are then further organized by the output type, whether that be some data, a figure, or a table.

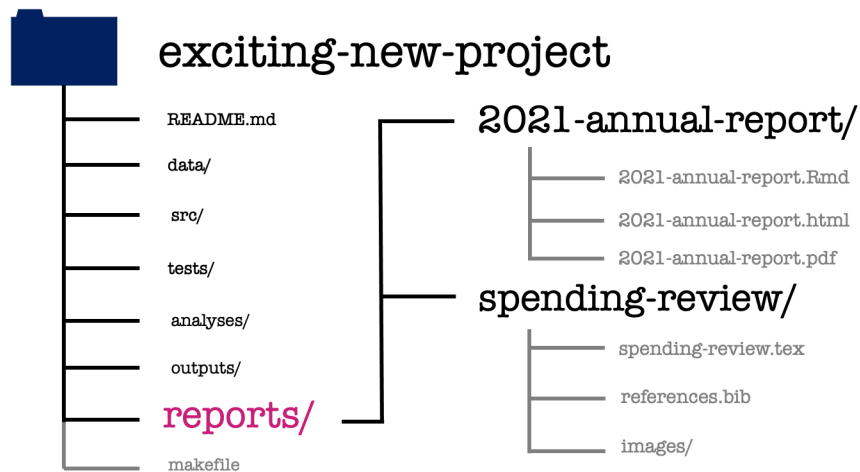
Depending on the nature of your project, you might want to use a modified subdirectory structure here. For example, if you're doing several numerical experiments then you might want to arrange your outputs by experiment, rather than by output type.

1.5.8 reports

```

image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)

```



The reports directory is then where everything comes together. This is where the written documents that form the final deliverables of your project are created. If these final documents are written in LaTeX or markdown, both the source and the compiled documents can be found within this directory.

When including content in this report, for example figures, I'd recommend against making copies of those figure files within the reports directory. If you do that, then you'll have to manually update the files every time you modify them. Instead you can use relative file paths to include these figures. Relative file paths specify how to get to the image, starting from your TeX document and moving up and down through the levels of your project directory.

For example when including a figure in the annual report you might use the following code to include a figure within the markdown source code.

```
![] (../../outputs/2021-annual-report/figures/2021-student-survey-histogram.png)
```

While in the spending review, the company logo might be included in the LaTeX source code as follows.

```
\includegraphics[width=0.2\textwidth]{images/logo.svg}
```

If you're not using markdown or LaTeX to write your reports, but instead use an online platform like [Overleaf](#) as a LaTeX editor or [Google Docs](#) to write collaboratively then add links to these in the reports directory by adding additional readme files. Make sure you set the read and write permissions for those links appropriately, too!

When using these online writing systems, you'll have to manually upload and update your plots and other outputs whenever you modify any of your earlier analysis. That's one of the drawbacks of these online tools that has to be traded off against their ease of use.

In our exciting new project, the annual report is written in a markdown format, which is compiled to both HTML and PDF. The spending review is written in LaTeX and we only have the source for it, we don't have the compiled pdf version of the document but should be able to create this for ourselves using the materials provided.

1.5.9 make file

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```



The final element of our template project structure is a **makefile**. We aren't going to cover how to read or write make files in this course. Instead, I'll give you a brief description of what one is and what it is supposed to do.

At a high level, the **makefile** is just another text file. What makes it special is what it contains. Similar to a shell or a bash script, a makefile contains code that can be run at the command line. This code will create or update each element of your project.

The **makefile** makes this easier by defining shorthand commands for the full lines of code that create each element of your project (running R scripts, compiling LaTeX documents and so on). The **makefile** also records the order in which these operations have to happen and which of these steps are dependent on one another. This means that if a single step part of your project is updated then any changes will be propagated through your entire project. This

is done in quite a clever way, so the only part of your projects that are re-run are those which need to be updated.

We're omitting the writing of makefiles from this course not because they're fiendishly difficult to write or read, but rather because they require a reasonable foundation in working at the command line to be understood. What I suggest you do instead throughout this course is to create your own `make.R` or `make.md` file. This file will define the intended running order and dependencies of your project and if it is an R file might also automate some parts of your analysis.

If you'd like to learn more about makefiles, some good resources are:

- [Monash Informatics Platform](#) (Motivates makefiles via compiling Rmd to multiple output formats)
- [Rob Hyndman blog post](#) (Old, but motivates use well)
- [Makefile tutorial](#) (Pretty and current but extensive and with a CS focus)

1.6 Wrapping up

Wrapping up then, that's everything for this chapter.

We have introduced a project structure that will serve you well as a baseline for the vast majority of projects in data science.

```
image_path <- "images/101-workflows-organising-your-work/directory-structure-drawings/dire
knitr::include_graphics(path = image_path)
```



In your own work, remember that the key here is standardisation. Working consistently across projects, a company or a group is more important than sticking rigidly to the particular structure that I have defined here.

There are two notable exceptions where you probably don't want to use this project structure. That's when you're building an app or you're building a package. These require specific organisation of the files within your project directory. We'll explore the project structure used for package development during the live session this week.

1.7 Session Information

```
pander::pander(sessionInfo())
```

R version 4.3.1 (2023-06-16)

Platform: x86_64-apple-darwin20 (64-bit)

locale: en_US.UTF-8|en_US.UTF-8|en_US.UTF-8|C|en_US.UTF-8|en_US.UTF-8

attached base packages: *stats*, *graphics*, *grDevices*, *utils*, *datasets*, *methods* and *base*

loaded via a namespace (and not attached): *compiler(v.4.3.1)*, *fastmap(v.1.1.1)*, *cli(v.3.6.2)*, *tools(v.4.3.1)*, *htmltools(v.0.5.7)*, *rstudioapi(v.0.15.0)*, *yaml(v.2.3.8)*, *Rcpp(v.1.0.11)*, *pander(v.0.6.5)*, *rmarkdown(v.2.25)*, *knitr(v.1.45)*, *jsonlite(v.1.8.8)*, *xfun(v.0.41)*, *digest(v.0.6.33)*, *rlang(v.1.1.2)*, *png(v.0.1-8)* and *evaluate(v.0.23)*

2 Naming Files

2.1 Introduction

“There are only two hard things in Computer Science: cache invalidation and naming things.”

Phil Karlton, Netscape Developer

When working on a data science project we can in principle name directories, files, functions and other objects whatever we like. In reality though, using an ad-hoc naming system is likely to cause confusion, mistakes and headaches. We obviously want to avoid all of those things in the spirit of being kind to our current colleges and also to our future selves.

Coming up with good names is a bit of an art form, it’s an activity that you get better at with practice. Another similarity to art is that the best naming systems don’t come from giving data scientists free reign over their naming system. The best approaches to naming things give you strong guidelines and boundaries within which to express your creativity and skill.

In this chapter we’ll explore what these boundaries and what we want them to achieve for us. The content of this chapter is based largely around a [talk of the same name](#) given by Jennifer Bryan and the [tidyverse style guide](#), which forms the basis of Google’s style guide for R programming.

The core principles we introduce hold across most high-level programming languages. Just like in natural languages, there are differences in how the principles are implemented between languages and between different groups of people who use each language. A fun aspect of this is that people often develop a code “accent” based on their “native” programming language.

If you’d like to check out how this guidance translates to Python, check out the [naming conventions section](#) of the PEP8 style guide.

2.2 Naming Files

We’ll begin by focusing in on what we call our files. That is, we’ll first focus on the part of the file name that comes before the dot. Later in the chapter, we’ll cycle back around to discuss file extensions.

2.2.1 What do we want from our file names?

To decide how we name our files, we should first consider what we want from our file naming system. There are three key properties that that we would like to satisfy.

1. Machine Readable
2. Human Readable
3. Order Friendly

The first desirable property is for file names to be easily readable by computers, the second is for the file names to be easily readable by humans and finally the file names should take advantage of the default ordering imposed on our files.

This set of current file names is sorely lacking across all of these properties:

```
abstract.docx
Effective Data Science's module guide 2022.docx
fig 12.png
Rplot7.png
1711.05189.pdf
HR Protocols 2015 FINAL (Nov 2015).pdf
```

We want to provide naming conventions to move us toward the better file names listed below.

```
2015-10-22_human-resources-protocols.pdf
2022_effective-data-science-module-guide.docx
2022_RSS-conference-abstract.docx
fig12_earthquake-timeseries.png
fig07_earthquake-location-map.png
ogata_1984_spacetime-clustering.pdf
```

Let's take a few minutes to examine what exactly we mean by each of these properties.

2.2.2 Machine Readable

What do we mean by machine readable file names?

- Easy to compute on by *deliberate use of delimiters*:
 - underscores_separate_metadata, hyphens-separate-words.
- Play nicely with *regular expressions* and *globbing*:

- avoid spaces, punctuation, accents, cases;
- `rm Rplot*.png`

Machine readable names are useful when:

- *managing files*: ordering, finding, moving, deleting;
- *extracting information* directly from file names,
- *working programmatically* with file names and regex.

2.2.2.1 Computing on File Names

When we are operating on a large number of files it is useful to be able to work with them programmatically.

One example of where this might be useful is when downloading and marking assessments. This requires me to download and unzip a large number of compressed folders, copying the pdf report from each unzipped folder into a single directory and then repeat this, moving all of the R scripts from each unzipped folder into a second directory. The marked scripts and code then need to be paired back up in folders named by student, and re-zipped ready to be returned.

This is *monotonously* dull and might work for ~50 students but would not scale to a class of ~5000. Working programmatically with files is the way to get this job done efficiently. This requires the file names to play nicely with the way that computers interpret file names, which they regard as a string of characters.

It is often helpful to have some meta-data included in the file name, for example the student's id number and the assessment title. We will use an underscore (`_`) to separate elements of meta-data within the file name and a hyphen (`-`) to separate sub-elements of meta-data, for example words within the assessment title.

```
001562679_assessment-1.tex  
001562679_assessment-1.pdf  
001562679_assessment-1.r
```

2.2.2.2 Regular expressions and Globbing

[Regular expressions](#) and [globbing](#) are two ideas from string manipulation that you may not have met, but which will inform our naming conventions.

Regular expressions allow you to search for strings (in our case file names) that match a particular pattern. Regular expressions can do really complicated searches but become gnarly

when you have to worry about special characters like spaces, punctuation, accents and cases, so these should be avoided in file names.

A special type of regular expression is called a glob. The most common use of globbing is the use of an asterisk (*) to replace any number of subsequent characters in a file name. Globbing becomes particularly powerful when you use a consistent structure to create your file names. We might, for example, delete all `png` images in our working directory that begin with `Rplot` using a single line of code at the command line: `rm Rplot*.png`.

Having machine readable file names is particularly useful when managing files, such as ordering, finding, moving or deleting them. Another common use case is when an analysis requires you to load a large number of individual data files. Machine readable file names are also useful in this setting for extracting meta-information from files without having to open them in memory. This is particularly useful when the files might be too large to load into memory, or you only want to load data from a certain year.

The final benefit we list here is the scalability, reduction in drudgery and lowered risk for human error when operating on a very large number of files.

2.2.3 Order Friendly

The next property we will focus on also links to how computers operate. We'd like our file names to exploit the default orderings used by computers. This means starting file names with character strings or metadata that allow us order our files in some meaningful way.

2.2.3.1 Running Order

One example of this is where there's some logical order in which your code should be executed, as in the example analysis below.

```
diagnositc-plots.R
download.R
runtime-comparison.R
...
model-evaluation.R
wrangle.R
```

Adding numbers to the start of these file names can make the intended ordering immediately obvious.

```
00_download.R
01_wrangle.R
02_model.R
...
09_model-evaluation.R
10_model-comparison-plots.R
```

Starting single digit numbers with a leading 0 is a very good idea here to prevent script 1 being sorted in with the tens, script 2 in with the twenties and so on. If you might have over 100 files, for example when saving the output from many simulations, use two or more zeros to maintain this nice ordering.

2.2.3.2 Date Order

A second example of orderable file names is when the file has a date associated with it. This might be a version of a report or the date on which some data were recorded, cleaned or updated.

```
2015-10-22_human-resources-protocols.pdf
...
...
2022-effective-data-science-module-guide.docx
```

When using dates, in file names or elsewhere, you should conform to the [ISO standard date format](#).

ISO 8601 sets an international standard format for dates: YYYY-MM-DD.

This format uses four numbers for the year, followed by two numbers for the month and two numbers of the day of the month. This structure mirrors a nested file structure moving from least to most specific. It also avoids confusion over the ordering of the date elements. Without using the ISO standard a date like 04-05-22 might be interpreted as the fourth of May 2022, the fifth of April 2022, or the twenty-second of May 2004.

2.2.4 Human Readable

The final property we would like our file names to have is human readability. This requires the names of our files to be meaningful, informative and easily read by real people.

The first two of these are handled by including appropriate metadata in the file name. The ease with which these are read by real people is determined by the length of the file name and by how that name is formatted.

There are lots of formatting options with fun names like `camelCase`, `PascalCase`, and `snake_case`.

```
easilyReadByRealPeople (camelCase)
EasilyReadByRealPeople (PascalCase)
easily_read_by_real_people (snake_case)
easily-read-by-real-people (skewer-case)
```

There is weak evidence to suggest that snake case and skewer case are most the readable. We'll use a mixture of these two, using snake case *between* metadata items and skewer case *within* them. This has a slight cost to legibility, in a trade-off against making computing on these file names easier.

The final aspect that you have control over is the length of the name. Having short, evocative and useful file names is not easy and is a skill in itself. For some hints and tips you might want to look into tips for writing [URL slugs](#). These are last part of a web address that are intended to improve accessibility by being immediately and intuitively meaningful to any user.

2.2.5 Naming Files - Summary

1. File names should be meaningful, informative and scripts end in `.r`
2. Stick to letters, numbers underscores (`_`) and hyphens (`-`).
3. Pay attention to capitalisation `file.r` \neq `File.r` on all operating systems.
4. Show order with left-padded numbers or ISO dates.

2.3 File Extensions and Where You Work

So far we have focused entirely on what comes before the dot, that is the file name. Equally if not more important is the file extension that comes after the dot.

```
example-script.r
example-script.py
```

```
project-writeup.doc
project-writeup.tex
```

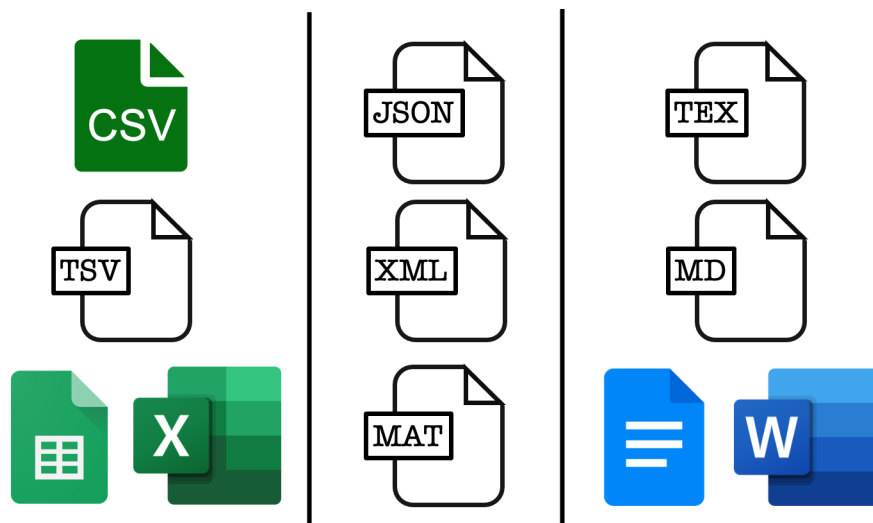
The file extension describes how information is stored in that file and determines the software that can use, view or run that file.

You likely already use file extensions to distinguish between code scripts, written documents, images, and notebook files. We'll now explore the benefits and drawbacks of various file types with respect to several important features.

2.3.1 Open Source vs Proprietary File Types

The first feature to consider is whether the file type is open source and can be used by anyone without charge or if specialist software must be paid for to interact with those files.

```
image_path <- "images/102-workflows-naming-files/file-types-image.png"  
knitr::include_graphics(path = image_path)
```



In the figure above, each column represents a different class of file, moving left to right we have example file types for tabular data, list-like data and text documents. File types closer to the top are open source while those lower down rely on proprietary software, which may or may not require payment.

To make sure that our work is accessible to as many people as possible we should favour the open source options like csv files over Google sheets or excel, JSON files over Matlab data files, and tex or markdown over a word or Google doc.

This usually has a benefit in terms of project longevity and scalability. This is because open source file types are often somewhat simpler in structure, making them more robust to changes over time and less memory intensive.

To see this, let's take a look inside some data files.

2.3.2 Inside Data Files

2.3.2.1 Inside a CSV file

CSV or comma separated value files are used to store tabular data.

In tabular data, each row of the data represents one record and each column represents a data value. A csv encodes this by having each record on a separate line and using commas to separate values with that record. You can see this by opening a csv file in a text editor such as notepad.

The raw data stores line breaks using `\n` and indicates new rows by `\r`. These backslashed indicate that these are *escape characters* with special meanings, and should not be literally interpreted as the letters n and r.

```
library(readr)
read_file(file = "images/102-workflows-naming-files/example.csv")
```

```
[1] "Name,Number\r\nA,1\r\nB,2\r\nC,3"
```

When viewed in a text editor, the example file would look something like this.

```
Name,Number
A,1
B,2
C,3
```

2.3.2.2 Inside a TSV file

TSV or tab separated value files are also used to store tabular data.

Like in a csv each record is given on a new line but in a tsv tabs rather than commas are used to separate values with each record. This can also be seen by opening a tsv file in a text editor such as notepad.

```
library(readr)
read_file(file = "images/102-workflows-naming-files/example.tsv")
```

```
[1] "Name\tNumber\r\nA\t1\r\nB\t2\r\nC\t3"
```

Name	Number
A	1
B	2
C	3

One thing to note is that tabs are a separate character and are not just multiple spaces. In plain text these can be impossible to tell apart, so most text editors have an option to display tabs differently from repeated spaces, though this is usually not enabled by default.

2.3.2.3 Inside an Excel file

When you open an excel file in a text editor, you will immediately see that this is not a human interpretable file format.

```
504b 0304 1400 0600 0800 0000 2100 62ee
9d68 5e01 0000 9004 0000 1300 0802 5b43
6f6e 7465 6e74 5f54 7970 6573 5d2e 786d
6c20 a204 0228 a000 0200 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
.....
0000 0000 0000 0000 ac92 4d4f c330 0c86
ef48 fc87 c8f7 d5dd 9010 424b 7741 48bb
2154 7e80 49dc 0fb5 8da3 241b ddbf 271c
1054 1a83 0347 7fbd 7efc cadb dd3c 8dea
.....
```

Each entry here is a four digit hexadecimal number and there are a lot more of them than we have entries in our small table.

This is because excel files can carry a lot of additional information that a csv or tsv are not able to, such as cell formatting or having multiple tables stored within a single file (called sheets by excel).

This means that excel files take up much more memory because they are carrying a lot more information than is strictly contained within the data itself.

2.3.2.4 Inside a JSON file

JSON, or Java Script Object Notation, files are an open source format for list-like data. Each record is represented by a collection of **key:value** pairs. In our example table each entry has two fields, one corresponding to the **Name** key and one corresponding to the **Number** key.

```
[{
  "Name": "A",
  "Number": "1"
}, {
  "Name": "B",
  "Number": "2"
}, {
  "Name": "C",
  "Number": "3"
}]
```

This list-like structure allows non-tabular data to be stored by using a property called nesting: the value taken by a key can be a single value, a vector of values or another list-like object.

This ability to create nested data structures has led to this data format being used widely in a range of applications that require data transfer.

2.3.2.5 Inside an XML file

XML files are another open source format for list-like data, where each record is represented by a collection of `key:value` pairs.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <row>
    <Name>A</Name>
    <Number>1</Number>
  </row>
  <row>
    <Name>B</Name>
    <Number>2</Number>
  </row>
  <row>
    <Name>C</Name>
    <Number>3</Number>
  </row>
</root>
```

The difference from a JSON file is mainly in how those records are formatted within the file. In a JSON file this is designed to look like objects in the Java Script programming language and in XML the formatting is done to look like HTML, the markup language used to write websites.

2.3.3 A Note on Notebooks

- There are two and a half notebook formats that you are likely to use: `.rmd`, `.ipynb` or alternatively `.qmd`.
- R markdown documents `.rmd` are plain text files, so are very human friendly.
- ***JuPyteR*** notebooks have multi-language support but are not so human friendly (JSON in disguise).
- Quarto documents offer the best of both worlds and more extensive language support. Not yet as established as a format.

In addition to the files you read and write, the files that you code in will largely determine your workflow.

There are three main options for the way that you code. You might typing your code directly at the command line, you might using a text editor or IDE to write scripts or you might use a notebook file that mixes code, text and output together in a single document.

We'll compare these methods of working soon, but first let's do a quick review of what notebooks are available to you and why you might want to use them.

As a data scientist, there are two and a half notebook formats that you're likely to have met before. The first two are Rmarkdown files for those working predominantly in R and interactive Python or JuPyteR notebooks for those working predominantly in Python. The final half format are quarto markdown documents, which are relatively new and extend the functionality of Rmarkdown files to provide multi-language support.

The main benefit of Rmarkdown documents is that they're plain text files, so they're very human friendly and work very well with version control software like git. ***JuPyteR*** notebooks have the benefit of supporting code written in Julia, Python or R, but are not so human friendly - under the hood these documents are JSON files that should not be edited directly (because a misplaced bracket will break them!).

Quarto documents offer the best of both worlds, with plain text formatting and even more extensive language support than jupyter notebooks. Quarto is a recent extension of Rmarkdown, which is rapidly becoming popular in the data science community. Quarto also allows you to create a wider range of documents, including websites, these course notes and the associated slides.

Each format has its benefits and drawbacks depending on the context in which they are used and all have some shared benefits and limitations by nature of them all being notebook documents.

2.3.4 File Extensions and Where You Code

Property	Notebook	Script	Command Line
reproducible	~		X
readable	~		~
self-documenting		X	X
in production	X		~
ordering / automation	~		~

The main benefit of notebook documents is that they are self-documenting: they can mix the documentation, code and report all into a single document. Notebooks also provide a level of interactivity when coding that is not possible when working directly at the command line or using a text editor to write scripts. This limitation is easily overcome by using an integrated development environment when scripting, rather than a plain text editor.

Writing code in `.r` files is not self-documenting but this separation of code, documentation and outputs has many other benefits. Firstly, the resulting scripts provide a reproducible and automatable workflow, unlike one-off lines of code being run at the command line. Secondly, using an IDE to write these provides you with syntax highlighting and code linting features to help you write readable and accurate code. Finally, the separation of code from documentation and output allows your work to be more easily or even directly put into production.

In this course we will generally advocate for a scripting-first approach to data science, though notebooks and command line work definitely have their place.

Notebooks are great as a teaching tool, for small reports and for rapid prototyping but they have strong limitations with being put into production. Conversely, coding directly at the command line is perfect for simple one-time tasks but it leaves no trace of your workflow and leads to an analysis that cannot be easily replicated in the future.

2.3.5 Summary

Finally, let's wrap things up by summarising what we have learned about naming files.

Before the dot we want to pick file names that machine readable, human friendly and play nicely with the default orderings provided to us.

Name files so that they are:

- Machine Readable,
- Human Readable,
- Order Friendly.

After the dot, we want to pick file types that are widely accessible, easily read by humans and allow for our entire analysis to be reproduced.

Use document types that are:

- Widely accessible,
- Easy to read and reproduce,
- Appropriate for the task at hand.

Above all we want to name our files and pick our file types to best match with the team we are working in and the task that is at hand.

2.4 Session Information

```
pander::pander(sessionInfo())
```

R version 4.3.1 (2023-06-16)

Platform: x86_64-apple-darwin20 (64-bit)

locale: en_US.UTF-8|en_US.UTF-8|en_US.UTF-8|C|en_US.UTF-8|en_US.UTF-8

attached base packages: *stats*, *graphics*, *grDevices*, *utils*, *datasets*, *methods* and *base*

other attached packages: *readr*(v.2.1.4)

loaded via a namespace (and not attached): *vctrs*(v.0.6.5), *cli*(v.3.6.2), *knitr*(v.1.45), *rlang*(v.1.1.2), *xfun*(v.0.41), *png*(v.0.1-8), *jsonlite*(v.1.8.8), *glue*(v.1.6.2), *htmltools*(v.0.5.7), *hms*(v.1.1.3), *fansi*(v.1.0.6), *rmarkdown*(v.2.25), *pander*(v.0.6.5), *evaluate*(v.0.23), *tibble*(v.3.2.1), *tzdb*(v.0.4.0), *fastmap*(v.1.1.1), *yaml*(v.2.3.8), *lifecycle*(v.1.0.4), *compiler*(v.4.3.1), *Rcpp*(v.1.0.11), *pkgconfig*(v.2.0.3), *rstudioapi*(v.0.15.0), *digest*(v.0.6.33), *R6*(v.2.5.1), *utf8*(v.1.2.4), *pillar*(v.1.9.0), *magrittr*(v.2.0.3) and *tools*(v.4.3.1)

3 Code

3.1 Introduction

We have already described how we might organise an effective data science project at the directory and file level. In this chapter we will delve one step deeper and consider how we can structure our work within those files.

In particular, we'll focus on code files here. We'll start by comparing the two main approaches to structuring our code, called functional programming and object oriented programming. We'll then consider conventions on how we should order code within our scripts and how we name the functions and objects we create.

To round up this chapter, we'll summarise the main points from the R style guide that we will be following in this course and highlight some useful packages for writing effective code.

3.2 Functional Programming

A functional programming style has two major properties:

- Object immutability,
- Complex programs are written using function composition.

The first point states that the original data or objects should never be modified or altered by the code that we write. We've met this idea before when making new, cleaner versions of our raw data while taking care to leave the original messy data intact. Object immutability extends this same idea to cover our code as well as our data.

The second point encapsulates that in functional programming complex problems are solved by decomposing them into a series of smaller problems. A separate, self-contained function is then written to solve each sub-problem. This means that each individual function is simple and easy to understand. This makes each of these small functions easy to test and to reuse in many places throughout our analysis. Code complexity is then built up by composing these functions in various ways.

It can be difficult to get into this way of thinking, but people with mathematical training often find it quite natural. This is because mathematicians have many years experience of working with function compositions in the abstract, mathematical sense.

$$y = g(x) = f_3 \circ f_2 \circ f_1(x).$$

3.2.1 The Pipe Operator

One issue with functional programming is that lots of nested functions means that there are also lots of nested brackets. These start to get tricky to keep track of when you have upwards of 3 functions being composed.

```
log(exp(cos(sin(pi))))
```

```
[1] 1
```

This reading difficulty is only exacerbated if your functions have additional arguments on top of the original inputs.

The pipe operator `%>%` from the `{magrittr}` package helps with this issue. It works exactly like function composition: it takes the whatever is on the left (whether that is an existing object or the output of a function) and passes it to the following function call as the first argument of that function.

```
library(magrittr)
pi %>%
  sin() %>%
  cos() %>%
  exp() %>%
  log()
```

```
[1] 1
```

```
iris %>%
  head(n = 3)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa

The pipe operator is often referred to as “syntactic sugar”. This is because it doesn’t add anything to your code in itself, but rather it makes your code *so* much more palatable to read.

In R versions 4.1 and greater, there’s a built-in version of this pipe operator, `|>`. This is written using the vertical bar symbol followed by a greater than sign. You can usually find this vertical bar above backslash on the keyboard. (Just to cause confusion, the vertical bar symbol is also called the pipe symbol and performs a similar operation in other programming languages.)

```
library(magrittr)
pi |>
  sin() |>
  cos() |>
  exp() |>
  log()
```

```
[1] 1
```

```
iris |>
  head(n = 3)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa

The base R pipe usually behaves in the same way as the pipe from `magrittr`, but there are a few cases where they differ. For reasons of back-compatibility and consistency we’ll stick to the `{magrittr}` pipe in this course. (For an outline of some of the differences between these two pipes, check out this [blog post](#) by Isabella Velasquez)

3.2.2 When not to pipe

Pipes are designed to put focus on the the actions you are performing rather than the object that you are performing those operations on. This means that there are two cases where you should almost certainly not use a pipe.

The first of these is when you need to manipulate more than one object at a time. Using secondary objects as reference points (but leaving them unchanged) is of course perfectly fine, but pipes should be used when applying a sequence of steps to create a new, modified version of one primary object.

Secondly, just because you *can* chain together many actions into a single pipeline that doesn't mean you necessarily should do so. Very long sequences of piped operations are easier to read than nested functions but they still burden the reader with the same cognitive load on their short-term memory. Be kind to them and create meaningful intermediate objects with informative names. This will help the reader to more easily understand the logic within your code.

3.3 Object Oriented Programming

The main alternative to functional programming is object oriented programming.

TL;DR

- OOP solves complex problems by using lots of simple objects
- R has 3 OOP systems: S3, S4 and R6.
- Objects belong to a class, have methods and fields.
- Example: agent based simulation of beehive.

3.3.1 OOP Philosophy

In functional programming, we solve complicated problems by using lots of simple functions. In object oriented programming we solve complicated problems using lots of simple objects. Which of these programming approaches is best will depend on the particular type of problem that you are trying to solve.

Functional programming is excellent for most types of data science work. Object oriented comes into its own when your problem has many small components interacting with one another. This makes it great for things like designing agent-based simulations, which I'll come back to in a moment.

In R there are three different systems for doing object oriented programming (called S3, S4, and R6), so things can get a bit complicated. We won't go into detail about them here, but I'll give you an overview of the main ideas.

This approach to programming might be useful for you in the future, for example if you want to extend base R functions to work with new types of input, and to have user-friendly displays. In that case [Advanced R](#) by Hadley Wickham is an excellent reference text.

In OOP, each object belongs to a class and has a set of methods associated with it. The class defines what an object *is* and methods describe what that object can *do*. On top of that, each object has class-specific attributes or data fields. These fields are shared by all objects in a class but the values that they take give information about that specific object.

3.3.2 OOP Example

This is all sounding very abstract. Let's consider writing some object oriented code to simulate a beehive. Each object will be a bee, and each bee is an instance of one of three bee classes: it might be a **queen**, a **worker** or a **drone**. Different bee classes have different methods associated with them, which describe what the bee can do, for example all bees would have 6 methods that let them move **up**, **down**, **left**, **right**, **forward** and **backward** within the hive. An additional **reproduce** method might only be defined for queen bees and a **pollinate** method might only be defined for workers. Each instance of a bee has then has its own fields, which give data about that specific bee. All bees have **x**, **y** and **z** coordinate fields that give the bee's current location within the hive. These fields can then be altered by using the movement methods defined above. The queen class might have an additional field, counting its number of offspring and the workers might have an additional field for how much pollen they are carrying.

As the simulation progresses, methods are applied to each object altering their fields and potentially creating or destroying objects. This is very different from the preservation mindset of functional programming, but hopefully you can see that it is a very natural approach to many types of problem.

3.4 Structuring R Script Headers

TL;DR

- Start script with a comment of 1-2 sentences explaining what it > does.
- `setwd()` and `rm(1s())` are the devil's work.
- "Session" > "Restart R" or Keyboard shortcut: `ctrl/cmd + shift > + 0`
- Polite to gather all `library()` and `source()` calls.
- Rude to mess with other people's set up using `install.packages()`.
- Portable scripts use paths relative to the root directory of the project.

First things first, let's discuss what should be at the top of your R scripts.

It is almost always a good idea to start your file with a few commented out sentences describing the purpose of the script and, if you work in a large team, perhaps who contact with any questions about this script. (There is more on comments coming up soon, don't worry!)

It is also good practise to move all `library()` and `source()` calls to the top of your script. These indicate the packages and helper function that are dependencies of your script; it's useful to know what you need to have installed before trying to run any code.

That segues nicely to the next point, which is never to hard code package installations. It is extremely bad practise and very rude to do so because then your script might alter another person's R installation. If you don't know already, this is precisely the difference between

an `install.packages()` and `library()` call: `install.packages()` will download the code for that package to the users computer, while `library()` takes that downloaded code and makes it available in the current R session. To avoid messing with anyone's R installation, you should always type `install.package()` commands directly in the console and then place the corresponding `library()` calls within your scripts.

Next, it is likely that you, or someone close to you, will commit the felony of starting every script by setting the working directory and clearing R's global environment. This is *very* bad practice, it's indicative of a workflow that's not project based and it's problematic for at least two reasons. Firstly, the path you set will likely not work on anyone else's computer. Secondly, clearing the environment like this may *look* like it gets you back to fresh, new R session but all of your previously loaded packages will still be loaded and lurking in the background.

Instead, to achieve your original aim of starting a new R session, go to the menu and select the "Session" drop down then select "Restart R". Alternatively, you can use keyboard shortcuts to do the same. This is "ctrl + shift + 0" on Windows and "cmd + shift + 0" on a mac. The fact that a keyboard shortcut exists for this should quite strongly hint that, in a reproducible and project oriented workflow, you should be restarting R quite often in an average working day. This is the scripting equivalent of "clear all output and rerun all" in a notebook.

Finally, let's circle back to the point I made earlier about setting the working directory. This is fragile because you are likely giving file paths that are specific to your computer, your operating system and your file organisation system. The chances of someone else having all of these the same are practically zero.

3.5 Portable File paths with {here}

```
# Bad - breaks if project moved
source("zaks-mbp/Desktop/exciting-new-project/src/helper_functions/rolling_mean.R")

# Better - breaks if Windows (older)
source("../..src/helper_functions/rolling_mean.R")

# Best - but use here:here() to check root directory correctly identified
source(here::here("src","helper_functions","rolling_mean.R"))

# For more info on the {here} package:
vignette("here")
```

To fix the problem of person- and computer-specific file paths you can have two options.

The first is to use *relative* file paths. In this you assume that each R script is being run in its current location and my moving up and down through the levels of your project directory you point to the file that you need.

This is good in that it solves the problem of paths breaking because you move the project to a different location on your own laptop. However, it does not fully solve the portability problem because you might move your file to a different location *within* the same project. It also does not solve the problem that windows uses MacOS and linux use forward slashes in file paths with widows uses backslashes.

To resolve these final two issues I recommend using the `here()` function from the `{here}` package. This package looks for a `.Rproj` or `.git` file to identify the root directory of your project. It then creates file paths relative to the root of your project that are suitable for the operating system on which the code is being run. It really is quite marvellous.

For more information on how to use the here package, explore its chapter in [R - What They Forgot, R for Data Science](#) or this [project oriented workflow blog post](#).

3.6 Code Body

We will moving on now from the head of the code to the body.

Having well named and organised code will do most of the work of helping people to read and understand your code. Comments and sectioning do the rest of the work.

Note, this section is designed as an introduction to the [tidyverse style guide](#) and not as a replacement to it.

3.6.1 Comments

Comments may be either short in-line comments at the end of a line or full lines dedicated to comments. To create either type of comment in R, simply type hash followed by one space. The rest of that line will not be evaluated and will function as a comment. If multi-line comments are needed simply start multiple lines with a hash and a space.

Comments can also be used to add structure to your code, buy using commented lines of hyphens and equal signs to chunk your files into minor and major sections.

Markdown-like section titles can be added to these section and subsection headers. Many IDEs, such as RStudio, will interpret these as a table of contents for you, so that you can more easily navigate your code.

```

# This is an example script showing good use of comments and sectioning

library(here)
source(here("src","helper_functions","rolling_mean.R"))

#=====
# Major Section on Comments ----
#=====

#-----
## Minor Section on inline comments ----
#-----
x <- 1:10 # this is an inline comment

#-----
## Minor Section on full line comments ----
#-----
rolling_mean(x)
# This is an full line comment
# Use 80 characters max for readability

```

3.6.2 Objects are Nouns

- Object names should use only lowercase letters, numbers, and `_`.
- Use underscores (`_`) to separate words within a name. (`snake_case`)
- Use nouns, preferring singular over plural names.

```

# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1

```

When creating and naming objects a strong guideline is that objects should be named using short but meaningful nouns. Names should not include any special characters and should use underscores to separate words within the object name.

This is similar to our file naming guide, but note that hyphens can't be used in object names because this conflicts with the subtraction operator.

When naming objects, as far as possible use singular nouns. The main reason for this is that the plurisation rules in English are complex and will eventually trip up either you or a user of your code.

3.6.3 Functions are Verbs

- Function names should use only lower-case letters, numbers, and `_`.
- Use underscores (`_`) to separate words within a name. (`snake_case`)
- Suggest imperative mood, as in a recipe.
- Break long functions over multiple lines, using 4 rather than the usual 2 spaces for indent.

```
# Good
add_row()
permute()

# Bad
row_adder()
permutation()

long_function_name <- function(
  a = "a long argument",
  b = "another argument",
  c = "another long argument") {
  # As usual code is indented by two spaces.
}
```

The guidelines for naming functions are broadly similar, with the advice that functions should be verbs rather than nouns.

Functions should be named in the imperative mood, like in a recipe. This is again for consistency; having function names in a range of moods and tenses leads to coding nightmares.

As with object names you should aim to give your functions and their arguments short, evocative names. For functions with many arguments or a long name, you might not be able to fit the function definition on a single line. In this case you can should place each argument on its own double indented line and the function body on a single indented line.

3.6.4 Casing Consistently

As we have mentioned already, we have many options for separating words within names:

- CamelCase
- pascalCase
- snakecase
- underscore_separated
- hyphen-separated
- point.separated

For people used to working in Python it is tempting to use point separation in function names, in the spirit of methods from object oriented programming. Indeed, some base R functions even use this convention.

However, the reason that we advise against it is because it is already used for methods in some of R's inbuilt OOP functionality. We will use underscore separation in our work.

3.6.5 Style Guide Summary

1. Use comments to structure your code
2. Objects = Nouns
3. Functions = Verbs
4. Use snake case and consistent grammar

3.7 Further Tips for Friendly Coding

In addition to naming conventions the style guide gives lots of other guidance on writing code in a way that is kind to future readers of that code.

I'm not going to go repeat all of that guidance here, but the motivation for all of these can be boiled down into the following points.

- Write your code to be easily understood by humans.
- Use informative names, typing is cheap.

```
# Bad
for (i in dmt) {
  print(i)
}

# Good
```

```
for (temperature in daily_max_temperature) {  
    print(temperature)  
}
```

- Divide your work into logical stages, human memory is expensive.

When writing your code, keep that future reader in mind. This means using names that are informative and reasonably short, it also means adding white space, comments and formatting to aid comprehension. Adding this sort of structure to your code also helps to reduce the cognitive burden that you are placing on the human reading your code.

Informative names are more important than short names. This is particularly true when using flow controls, which are things like for loops and while loops. Which of these for loops would you like to encounter when approaching a deadline or urgently fixing a bug? Almost surely the second one, where context is immediately clear.

A computer doesn't care if you call a variable by only a single letter, by a random key smash (like `aksnbioawb`) or by an informative name. A computer also doesn't care if you include no white space your code - the script will still run. However, doing these things are friendly practices that can help yourself when debugging and your co-workers when collaborating.

3.8 Reduce, Reuse, Recycle

In this final section, we'll look at how you can make your workflow more efficient by reducing the amount of code you write, as well as reusing and recycling code that you've already written.

3.8.1 DRY Coding

This idea of making your workflow more efficient by reducing, reusing and recycling your code is summarised by the DRY acronym: don't repeat yourself.

This can be boiled down to three main points:

- if you do something twice in a single script, then write a function to do that thing;
- if you want to use your function elsewhere *within* your project, then save it in a separate script;
- If you want to use your function *across* projects, then add it to a package.

Of course, like with scoping projects in the first place, this requires some level of clairvoyance: you have to be able to look into the future and see whether you'll use a function in another script or project. This is difficult, bordering on impossible. So in practice, this is done retrospectively - you find a second script or project that needs a function then pull it out its own separate file or include it in a package.

As a rule of thumb, if you are having to consider whether or not to make the function more widely available then you should do it. It takes much less effort to do this work now, while it's fresh in your mind, than to have to re-familiarise yourself with the code in several years time.

Let's now look at how to implement those sub-bullet points: "when you write a function, document it" and "when you write a function, test it".

3.8.2 Rememer how to use your own code

When you come to use a function written by somebody else, you likely have to refer to their documentation to teach or to remind yourself of things like what the expected inputs are and how exactly the method is implemented.

When writing your own functions you should create documentation that fills the same need. Even if the function is just for personal use, over time you'll forget exactly how it works.

When you write a function, document it.

But what should that documentation contain?

- Inputs
- Outputs
- Example use cases
- Author (if not obvious or working in a team)

Your documentation should describe the inputs and outputs of your function, some simple example uses. If you are working in a large team, the documentation should also indicate who wrote the function and who's responsible for maintaining it over time.

3.8.3 {roxygen2} for documentation

In the same way that we used the {here} package to simplify our file path problems, we'll use the {roxygen2} package to simplify our testing workflow.

The {roxygen2} package gives us an easily insert-able temple for documenting our functions. This means we don't have to waste our time and energy typing out and remembering boilerplate code. It also puts our documentation in a format that allows us to get hints and

auto-completion for our own functions, just like the functions we use from packages that are written by other people.

To use Roxygen, you only need to install it once - it doesn't need to be loaded with a library call at the top of your script. After you've done this, and with your cursor inside a function definition, you can then insert skeleton code to document that function in one of two ways: you can either use the Rstudio menu or the keyboard short cut for your operating system.

1. `install.packages("roxygen2")`
2. With cursor inside function: Code > Insert Roxygen Skeleton
3. Keyboard shortcut: *cmd + option + shift + r* or *crtl + option + shift + r*
4. Fill out relevant fields

3.8.4 An {roxygen2} example

Below, we've got an example of an Roxygen skeleton to document a function that calculates the geometric mean of a vector. Here, the hash followed by an apostrophe is a special type of comment. It indicates that this is function documentation rather than just a regular comment.

```
#' Title
#'
#' @param x
#' @param remove_NA
#'
#' @return
#' @export
#'
#' @examples
geometric_mean <- function(x, remove_NA = FALSE){
  # Function body goes here
}
```

We'll fill in all of the fields in this skeleton apart from export, which we'll remove. If we put this function in a R package, then the export field makes it available to users of that package. Since this is just a standalone function we won't need the export field, though keeping it wouldn't actually cause us any problems either.

```
#' Calculate the geometric mean of a numeric vector
#'
#' @param x numeric vector
#' @param remove_NA logical scalar, indicating whether NA values should be stripped before
```

```

#'
#' @return the geometric mean of the values in `x`, a numeric scalar value.
#'
#' @examples
#' geometric_mean(x = 1:10)
#' geometric_mean(x = c(1:10, NA), remove_NA = TRUE)
#'
geometric_mean <- function(x, remove_NA = FALSE){
  # Function body goes here
}

```

Once we have filled in the skeleton documentation it might look something like this. We have described what the function does, what the expected inputs are and what the user can expect as an output. We've also given a few simple examples of how the function can be used.

For more on Roxygen, see the [package documentation](#) or the chapter of R packages on [function documentation](#).

3.8.5 Checking Your Code

If you write a function, test it.

Testing code has two main purposes:

- To warn or prevent user misuse (e.g. strange inputs),
- To catch edge cases.

On top of explaining how our functions *should* work, we really ought to check that they *do* work. This is the job of unit testing.

Whenever you write a function you should test that it works as you intended it to. Additionally, you should test that your function is robust to being misused by the user. Depending on the context, this might be accidental or malicious misuse. Finally, you should check that the function behaves properly for strange, but still valid, inputs. These are known as edge cases.

Testing can be a bit of a brutal process, you've just created a beautiful function and now your job is to do your best to break it!

3.8.6 An Informal Testing Workflow

1. Write a function
2. Experiment with the function in the console, try to break it
3. Fix the break and repeat.

Problems: Time consuming and not reproducible.

An informal approach to testing your code might be to first write a function and then play around with it in the console to check that it behaves well when you give it obvious inputs, edge cases and deliberately wrong inputs. Each time you manage to break the function, you edit it to fix the problem and then start the process all over again.

This *is* testing the code, but only informally. There's no record of how you have tried to break your code already. The problem with this approach is that when you return to this code to add a new feature, you'll probably have forgotten at least one of the informal tests you ran the first time around. This goes against our efforts towards reproducibility and automation. It also makes it very easy to break code that used to work just fine.

3.8.7 A Formal Testing Workflow

We can formalise this testing workflow by writing our tests in their own R script and saving them for future reference. Remember from the first lecture that these should be saved in the `tests/` directory, the structure of which should mirror that of the `src/` directory for your project. All of the tests for one function should live in a single file, which is named after that function.

One way of writing these tests is to use lots of if statements. The `{testthat}` can do some of that syntactic heavy lifting for us. It has lots of helpful functions to test that the output of your function is what you expect.

```
geometric_mean <- function(x , remove_NA = FALSE){prod(x)^(1/length(x))}

testthat::expect_equal(
  object = geometric_mean(x = c(1, NA), remove_NA = FALSE),
  expected = NA)

# Error: geometric_mean(x = c(1, NA), remove_NA = FALSE) not equal to NA.
# Types not compatible: double is not logical
```

In this example, we have an error because our function returns a logical NA rather than a double NA. Yes, R really does have different types of NA for different types of missing data, it usually just handles these nicely in the background for you.

This subtle difference is probably not something that you would have spotted on your own, until it caused you trouble much further down the line. This rigorous approach is one of the benefits of using the `{testthat}` functions.

To fix this test we change out expected output to `NA_real_`.

```
testthat::expect_equal(  
  object = geometric_mean(x = c(1,NA), remove_NA = FALSE),  
  expected = NA_real_)
```

We'll revisit the `{testthat}` package in the live session this week, when we will learn how to use it to test functions within our own packages.

3.9 Summary

- Functional and Object Oriented Programming
- Structuring your scripts
- Styling your code
- Reduce, reuse, recycle
- Documenting and testing

Let's wrap up by summarising what we have learned in this chapter.

We started out with a discussion on the differences between functional and object oriented programming. While R is capable of both, data science work tends to have more of a functional flavour to it.

We've then described how to structure your scripts and style your code to make it as human-friendly and easy to debug as possible.

Finally, we discussed how to write DRY code that is well documented and tested.

3.10 Session Information

```
pander::pander(sessionInfo())
```

R version 4.3.1 (2023-06-16)

Platform: x86_64-apple-darwin20 (64-bit)

locale: en_US.UTF-8|en_US.UTF-8|en_US.UTF-8|C|en_US.UTF-8|en_US.UTF-8

attached base packages: *stats*, *graphics*, *grDevices*, *utils*, *datasets*, *methods* and *base*

other attached packages: *magrittr*(v.2.0.3)

loaded via a namespace (and not attached): *crayon*(v.1.5.2), *vctrs*(v.0.6.5), *cli*(v.3.6.2), *knitr*(v.1.45), *rlang*(v.1.1.2), *xfun*(v.0.41), *stringi*(v.1.8.3), *purrr*(v.1.0.2), *pkgload*(v.1.3.3), *generics*(v.0.1.3), *assertthat*(v.0.2.1), *jsonlite*(v.1.8.8), *glue*(v.1.6.2), *rprojroot*(v.2.0.4),

htmltools(v.0.5.7), *brio(v.1.1.4)*, *rmarkdown(v.2.25)*, *pander(v.0.6.5)*, *emo(v.0.0.0.9000)*,
evaluate(v.0.23), *fastmap(v.1.1.1)*, *yaml(v.2.3.8)*, *lifecycle(v.1.0.4)*, *stringr(v.1.5.1)*, *com-*
piler(v.4.3.1), *Rcpp(v.1.0.11)*, *testthat(v.3.2.1)*, *timechange(v.0.2.0)*, *rstudioapi(v.0.15.0)*,
digest(v.0.6.33), *R6(v.2.5.1)*, *tools(v.4.3.1)*, *lubridate(v.1.9.3)* and *desc(v.1.4.3)*

Workflows Checklist

Videos / Chapters

- [Organising your work](#) (30 min) [slides]
- [Naming Files](#) (20 min) [slides]
- [Organising your code](#) (27 min) [slides]

Reading

Use the [workflows section](#) of the reading list to support and guide your exploration of this week's materials. Note that these texts are divided into core reading, reference materials and materials of interest.

Tasks

Short:

- When downloading papers from a journal webpage or ArXiv, they often have unhelpful file names like `2310.12711.pdf`. Come up with your own naming convention that will keep your downloaded papers organised and easy to search. Write a short paragraph explaining and justifying your naming convention for someone else.
- When downloading data from websites you can encounter similiary uninformative names. Use the [US Geological Survey website](#) to download a csv file of all earthquakes in the conterminous US exceeding magnitude 3.0 during 2023. Note the default name of this file and suggest your own, improved name for this file.
- Consider simulating a fire evacuation of the Huxley building taking an object oriented approach. List the classes of object might you define and what methods and fields would give to each of these. (There is no expectation to code this simulation)

Core:

- Find 3 data science projects on Github and explore how they organise their work. Write a post on the EdStem forum that links to all three, and in a couple of paragraphs describe the content and structure of one project.
- Create your own project directory (or directories) for this course and its assignments.
- Write two of your own R functions. The first should calculate the geometric mean of a numeric vector. The second should calculate the rolling arithmetic mean of a numeric vector.

Bonus:

- Re-factor an old project to match the project organisation and coding guides for this course. This might be a small research project, class notes or a collection of homework assignments. Use an R-based project if possible. If you only have python projects, then either translate these to R or apply the [PEP8](#) style guide. Take care to select a suitably sized project so that this is a meaningful exercise but does not take more than a few hours.
- If you are able to do so, host your re-factored project publicly and share it with the rest of the class on the EdStem Discussion forum.

Live Session

In the live session we will begin with a discussion of this week's tasks. We will then create a minimal R package to organise and test the functions you have written.

- Please come to the live session prepared to discuss the following points:
 - Did you make the assignment projects as subdirectories or as their stand alone projects? Why?
 - What were some terms that you had not met before during the readings? How did you find their meanings?
 - What did you have to consider when writing your rolling mean function?
- If you would like to follow along in the live session then you should prepare by:
 - Installing/Updating [R](#), [RStudio](#) and [Quarto](#).
 - Follow the [Happy git with R](#) instructions to link RStudio and Git and Github (§1-14).
 - Write a `username/README.md` introducing yourself on your GitHub profile. (See e.g.: [StatsRhian](#), [nrennie](#))

Part II

Acquiring and Sharing Data

Introduction

i Note

Effective Data Science is still a work-in-progress. This chapter is largely complete and just needs final proof reading.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

Data can be difficult to acquire and gnarly when you get it.

The raw material that you work with as a data scientist is, unsurprisingly, data. In this part of the course we will focus on the different ways in which data can be stored, distributed and obtained.

Being able to obtain and read a dataset is often a surprisingly large hurdle in getting a new data science project off the ground. The skill of being able to source and read data from many locations is usually sanitised during a statistics programme: you're given a ready-to-go, cleaned CSV file and all focus is placed on modelling. This week aims to remedy that by equipping you with the skills to acquire and manage your own data.

We will begin this week by explore different file types. This dictates what type of information you can store, who can access that information and how they read that it into R. We will then turn our attention to the case when data are not given to you directly. We will learn how to obtain data from a raw webpage and how to request data that via a service known as an API.

4 Tabular Data

! Important

Effective Data Science is still a work-in-progress. This chapter is undergoing heavy restructuring and may be confusing or incomplete.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

4.1 Loading Tabular Data

Recall that simpler, open source formats improve accessibility and reproducibility. We will begin by reading in three open data formats for tabular data.

- `random-data.csv`
- `random-data.tsv`
- `random-data.txt`

Each of these data sets contains 26 observations of 4 variables:

- `id`, a Roman letter identifier;
- `gaussian`, standard normal random variates;
- `gamma`, `gamma(1,1)` random variates;
- `uniform`, `uniform(0,1)` random variates.

4.1.1 Base R

```
random_df <- read.csv(file = 'random-data.csv')
print(random_df)
#>   id  gaussian      gamma  uniform
#> 1  a -1.20706575 0.98899970 0.22484576
#> 2  b  0.27742924 0.03813386 0.08498474
#> 3  c  1.08444118 1.09462335 0.63729826
```

```

#> 4 d -2.34569770 1.49301101 0.43101637
#> 5 e 0.42912469 5.40361248 0.07271609
#> 6 f 0.50605589 1.72386539 0.80240202
#> 7 g -0.57473996 1.95357133 0.32527830
#> 8 h -0.54663186 0.07807803 0.75728904
#> 9 i -0.56445200 0.21198194 0.58427152
#> 10 j -0.89003783 0.20803673 0.70883941
#> 11 k -0.47719270 2.08607862 0.42697577
#> 12 l -0.99838644 0.49463708 0.34357270
#> 13 m -0.77625389 0.77171305 0.75911999
#> 14 n 0.06445882 0.37216648 0.42403021
#> 15 o 0.95949406 1.88207991 0.56088725
#> 16 p -0.11028549 0.76622568 0.11613577
#> 17 q -0.51100951 0.50488585 0.30302180
#> 18 r -0.91119542 0.22979791 0.47880269
#> 19 s -0.83717168 0.75637275 0.34483055
#> 20 t 2.41583518 0.62435969 0.60071414
#> 21 u 0.13408822 0.64638373 0.07608332
#> 22 v -0.49068590 0.11247545 0.95599261
#> 23 w -0.44054787 0.11924307 0.02220682
#> 24 x 0.45958944 4.91805535 0.84171063
#> 25 y -0.69372025 0.60282666 0.63244245
#> 26 z -1.44820491 0.64446571 0.31009417

```

Output is a `data.frame` object. (List of vectors with some nice methods)

4.1.2 {readr}

```

random_tbl <- readr::read_csv(file = 'random-data.csv')
#> Rows: 26 Columns: 4
#> -- Column specification -----
#> Delimiter: ","
#> chr (1): id
#> dbl (3): gaussian, gamma, uniform
#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
print(random_tbl)
#> # A tibble: 26 x 4
#>   id      gaussian  gamma uniform
#>   <chr>    <dbl> <dbl> <dbl>

```

```

#> 1 a      -1.21  0.989  0.225
#> 2 b       0.277 0.0381 0.0850
#> 3 c       1.08  1.09   0.637
#> 4 d      -2.35  1.49   0.431
#> 5 e       0.429 5.40   0.0727
#> 6 f       0.506 1.72   0.802
#> # i 20 more rows

```

Output is a `tibble` object. (List of vectors with some nicer methods)

4.1.2.1 Benefits of `readr::read_csv()`

1. Increased speed (approx. 10x) and progress bar.
2. Strings are not coerced to factors. No more `stringsAsFactors = FALSE`
3. No row names and nice column names.
4. Reproducibility bonus: does not depend on operating system.

4.1.3 WTF: Tibbles

4.1.3.1 Printing

- Default to first 10 rows and as many columns as will comfortably fit on your screen.
- Can adjust this behaviour in the print call:

```

# print first three rows and all columns
print(random_tbl, n = 3, width = Inf)
#> # A tibble: 26 x 4
#>   id      gaussian  gamma uniform
#>   <chr>    <dbl> <dbl>   <dbl>
#> 1 a      -1.21  0.989   0.225
#> 2 b       0.277 0.0381  0.0850
#> 3 c       1.08  1.09    0.637
#> # i 23 more rows

```

Bonus: Colour formatting in IDE and each column tells you it's type.

4.1.3.2 Subsetting

Subsetting tibbles will always return another tibble.

```
# Row Subsetting
random_tbl[1, ] # returns tibble
random_df[1, ]  # returns data.frame

# Column Subsetting
random_tbl[ , 1] # returns tibble
random_df[ , 1]  # returns vector

# Combined Subsetting
random_tbl[1, 1] # returns 1x1 tibble
random_df[1, 1]  # returns single value
```

This helps to avoid edge cases associated with working on data frames.

4.1.4 Other {readr} functions

See {readr} [documentation](#), there are lots of useful additional arguments that can help you when reading messy data.

Functions for reading and writing other types of tabular data work analogously.

4.1.4.1 Reading Tabular Data

```
library(readr)
read_tsv("random-data.tsv")
read_delim("random-data.txt", delim = " ")
```

4.1.4.2 Writing Tabular Data

```
write_csv(random_tbl, "random-data-2.csv")
write_tsv(random_tbl, "random-data-2.tsv")
write_delim(random_tbl, "random-data-2.tsv", delim = " ")
```

4.1.5 Need for Speed

Some times you have to load *lots of large data sets*, in which case a 10x speed-up might not be sufficient.

If each data set still fits inside RAM, then check out `data.table::fread()` which is optimised for speed. (Alternatives exist for optimal memory usage and data too large for working memory, but not covered here.)

Note: While it can be much faster, the resulting `data.table` object lacks the consistency properties of a tibble so be sure to check for edge cases, where the returned value is not what you might expect.

4.2 Tidy Data

4.2.1 Wide vs. Tall Data

4.2.1.1 Wide Data

- First column has unique entries
- Easier for humans to read and compute on
- Harder for machines to compute on

4.2.1.2 Tall Data

- First column has repeating entries
- Harder for humans to read and compute on
- Easier for machines to compute on

4.2.1.3 Examples

Example 1 (Wide)

Person	Age	Weight	Height
Bob	32	168	180
Alice	24	150	175
Steve	64	144	165

Example 1 (Tall)

Person	Variable	Value
Bob	Age	32
Bob	Weight	168
Bob	Height	180
Alice	Age	24
Alice	Weight	150
Alice	Height	175
Steve	Age	64
Steve	Weight	144
Steve	Height	165

[Source: Wikipedia - Wide and narrow data]

Example 2 (Wide)

Team	Points	Assists	Rebounds
A	88	12	22
B	91	17	28
C	99	24	30
D	94	28	31

Example 2 (Tall)

Team	Variable	Value
A	Points	88
A	Assists	12
A	Rebounds	22
B	Points	91
B	Assists	17
B	Rebounds	28
C	Points	99
C	Assists	24
C	Rebounds	30
D	Points	94
D	Assists	28
D	Rebounds	31

[Source: Statology - Long vs wide data]

4.2.1.4 Pivoting Wider and Longer

- Error control at input and analysis is format-dependent.
- Switching between long and wide formats useful to control errors.
- Easy with the {tidyr} package functions

```
tidyr::pivot_longer()  
tidyr::pivot_wider()
```

4.2.2 Tidy What?

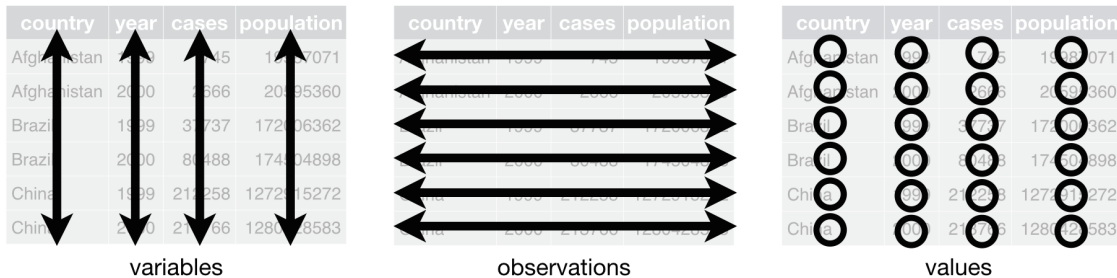


Figure 4.1: [Image: R4DS - Chapter 12]

Tidy Data is an opinionated way to store tabular data.

Image Source: Chapter 12 of R for Data Science.

- Each column corresponds to a exactly one measured variable
- Each row corresponds to exactly one observational unit
- Each cell contains exactly one value.

Benefits of tidy data

- *Consistent data format*: Reduces cognitive load and allows specialised tools (functions) to efficiently work with tabular data.
- *Vectorisation*: Keeping variables as columns allows for very efficient data manipulation. (this goes back to data frames and tibbles being lists of vectors)

4.2.3 Example - Tidy Longer

Consider trying to plot these data as time series. The `year` variable is trapped in the column names!

```
#> # A tibble: 3 x 3
#>   country    `1999` `2000`
#>   <chr>      <dbl> <dbl>
#> 1 Afghanistan    745   2666
#> 2 Brazil        37737  80488
#> 3 China         212258 213766
```

To tidy this data, we need to `pivot_longer()`. We will turn the column names into a new `year` variable and retaining cell contents as a new variable called `cases`.

```
library(magrittr)

countries %>%
  tidyr::pivot_longer(cols = c(`1999`, `2000`), names_to = "year", values_to = "cases")
#> # A tibble: 6 x 3
#>   country    year    cases
#>   <chr>      <chr> <dbl>
#> 1 Afghanistan 1999     745
#> 2 Afghanistan 2000    2666
#> 3 Brazil      1999   37737
#> 4 Brazil      2000   80488
#> 5 China       1999  212258
#> 6 China       2000  213766
```

Much better!

4.2.4 Example - Tidy Wider

There are other times where we might have to widen our data to tidy it.

This example is not tidy. Why not?

Team	Variable	Value
A	Points	88
A	Assists	12
A	Rebounds	22

Team	Variable	Value
B	Points	91
B	Assists	17
B	Rebounds	28
C	Points	99
C	Assists	24
C	Rebounds	30
D	Points	94
D	Assists	28
D	Rebounds	31

The observational unit here is a team. However, each variable should be stored in a separate column, with cells containing their values.

To tidy this data we first generate it as a tibble. We use the `tribble()` function, which allows us to create a tibble row-wise rather than column-wise.

We can then tidy it by creating new columns for each value of the current `Variable` column and taking the values for these from the current `Value` column.

```

tournament %>%
  tidyr::pivot_wider(
    id_cols = "Team",
    names_from = "Variable",
    values_from = "Value")
#> # A tibble: 4 x 4
#>   Team Points Assists Rebounds
#>   <chr> <dbl> <dbl> <dbl>
#> 1 A      88     12     22
#> 2 B      91     17     28
#> 3 C      99     24     30
#> 4 D      94     28     31

```

4.2.5 Other helpful functions

The `pivot_*()` family of functions resolve issues with rows (too many observations per row or rows per observation).

There are similar helper functions to solve column issues:

- Multiple variables per column: `tidyr::separate()`,
- Multiple columns per variable: `tidyr::unite()`.

4.2.6 Missing Data

In tidy data, every cell contains a value. Including cells with missing values.

- Missing values are coded as `NA` (generic) or a type-specific `NA`, such as `NA_character_`.
- The `{readr}` family of `read_*()` function have good defaults and helpful `na` argument.
- Explicitly code `NA` values when collecting data, avoid ambiguity: ” “, -999 or worst of all 0.
- More on missing values in EDA videos...

4.3 Wrapping Up

1. Reading in tabular data by a range of methods
2. Introduced the `tibble` and tidy data (+ tidy not always best)
3. Tools for tidying messy tabular data

4.4 Session Information

R version 4.3.1 (2023-06-16)

Platform: x86_64-apple-darwin20 (64-bit)

locale: en_US.UTF-8|en_US.UTF-8|en_US.UTF-8|C|en_US.UTF-8|en_US.UTF-8

attached base packages: *stats*, *graphics*, *grDevices*, *utils*, *datasets*, *methods* and *base*

other attached packages: *magrittr*(v.2.0.3)

loaded via a namespace (and not attached): *crayon*(v.1.5.2), *vctrs*(v.0.6.5), *cli*(v.3.6.2), *knitr*(v.1.45), *rlang*(v.1.1.2), *xfun*(v.0.41), *purrr*(v.1.0.2), *generics*(v.0.1.3), *jsonlite*(v.1.8.8), *glue*(v.1.6.2), *bit*(v.4.0.5), *htmltools*(v.0.5.7), *hms*(v.1.1.3), *fansi*(v.1.0.6), *rmarkdown*(v.2.25), *pander*(v.0.6.5), *evaluate*(v.0.23), *tibble*(v.3.2.1), *tzdb*(v.0.4.0), *fastmap*(v.1.1.1), *yaml*(v.2.3.8), *lifecycle*(v.1.0.4), *compiler*(v.4.3.1), *dplyr*(v.1.1.4), *Rcpp*(v.1.0.11), *pkgconfig*(v.2.0.3), *tidyr*(v.1.3.0), *rstudioapi*(v.0.15.0), *digest*(v.0.6.33), *R6*(v.2.5.1), *tidyselect*(v.1.2.0), *readr*(v.2.1.4), *utf8*(v.1.2.4), *parallel*(v.4.3.1), *vroom*(v.1.6.5), *pillar*(v.1.9.0), *withr*(v.2.5.2), *tools*(v.4.3.1) and *bit64*(v.4.0.5)

5 Web Scraping

i Note

Effective Data Science is still a work-in-progress. This chapter is largely complete and just needs final proof reading.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

5.1 Scraping webpage data using `{rvest}`

You can't always rely on tidy, tabular data to land on your desk. Sometimes you are going to have to go out and gather data for yourself.

I'm not suggesting you will need to do this manually, but you will likely need to get data from the internet that's been made publicly or privately available to you.

This might be information from a webpage that you gather yourself, or data shared with you by a collaborator using an API.

In this chapter we will cover the basics of scraping webpages, following the [vignette](#) for the `{rvest}` package.

5.2 What is a webpage?

Before we can even hope to get data from a webpage, we first need to understand *what* a webpage is.

Webpages are written in a similar way to LaTeX: the content and styling of webpages are handled separately and are coded using plain text files.

In fact, websites go one step further than LaTeX. The content and styling of websites are written in different files and in different languages. HTML (HyperText Markup Language) is used to write the content and then CSS (Cascading Style Sheets) are used to control the appearance of that content when it's displayed to the user.

5.3 HTML

A basic HTML page with no styling applied might look something like this:

```
<html>
<head>
  <title>Page title</title>
</head>
<body>
  <h1 id='first'>A level 1 heading</h1>
  <p>Hello World!</p>
  <p>Here is some plain text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100'>
</body>
```

5.3.1 HTML elements

Just like XML data files, HTML has a hierarchical structure. This structure is crafted using HTML elements. Each HTML element is made up of of a start tag, optional attributes, an end tag.

We can see each of these in the first level header, where `<h1>` is the opening tag, `id='first'` is an additional attribute and `</h1>` is the closing tag. Everything between the opening and closing tag are the contents of that element. There are also some special elements that consist of only a single tag and its optional attributes. An example of this is the `` tag.

Since `<` and `>` are used for start and end tags, you can't write them directly in an HTML document. Instead, you have to use escape characters. This sounds fancy, but it's just an alternative way to write characters that serve some special function within a language.

You can write greater than `>`; and less than as `<`;. You might notice that those escapes use an ampersand (`&`). This means that if you want a literal ampersand on your webpage, you have to escape too using `&`;

There are a wide range of possible HTML tags and escapes. We'll cover the most common tags in this lecture and you don't need to worry about escapes too much because `{rvest}` will automatically handle them for you.

5.3.2 Important HTML Elements

In all, there are in excess of 100 HTML elements. The most important ones for you to know about are:

- The `<html>` element, that must enclose every HTML page. The `<html>` element must have two child elements within it. The `<head>` element contains metadata about the document, like the page title that is shown in the browser tab and the CSS style sheet that should be applied. The `<body>` element then contains all of the content that you see in the browser.
- Block elements are used to give structure to the page. These are elements like headings, sub-headings and so on from `<h1>` all the way down to `<h6>`. This category also contains paragraph elements `<p>`, ordered lists `` unordered lists ``.
- Finally, inline tags like `` for bold, `<i>` for italics, and `<a>` for hyperlinks are used to format text inside block elements.

When you come across a tag that you've never seen before, you can find out what it does with just a little bit of googling. A good resource here is the [MDN Web Docs](#) which are produced by Mozilla, the company that makes the Firefox web browser. The [W3schools website](#) is another great resource for web development and coding resources more generally.

5.4 HTML Attributes

We've seen one example of a header with an additional attribute. More generally, all tags can have named attributes. These attributes are contained within the opening tag and look something like:

```
<tag attribute1='value1' attribute2='value2'>element contents</tag>
```

Two of the most important attributes are `id` and `class`. These attributes are used in conjunction with the CSS file to control the visual appearance of the page. These are often very useful to identify the elements that you are interested in when scraping data off a page.

5.5 CSS Selectors

The Cascading Style Sheet is used to describe how your HTML content will be displayed. To do this, CSS has it's own system for selecting elements of a webpage, called CSS selectors.

CSS selectors define patterns for locating the HTML elements that a particular style should be applied to. A happy side-effect of this is that they can sometimes be very useful for scraping, because they provide a concise way of describing which elements you want to extract.

CSS Selectors can work on the level of an element type, a class, or a tag and these can be used in a nested (or *cascading*) way.

- The `p` selector will select all paragraph `<p>` elements.
- The `.title` selector will select all elements with class `"title"`.
- The `p.special` selector will select all `<p>` elements with class `"special"`.
- The `#title` selector will select the element with the id attribute `"title"`.

When you want to select a single element id attributes are particularly useful because that *must* be unique within a html document. Unfortunately, this is only helpful if the developer added an id attribute to the element(s) you want to scrape!

If you want to learn more CSS selectors I recommend starting with the fun [CSS diner tutorial](#) to build a base of knowledge and then using the [W3schools resources](#) as a reference to explore more webpages in the wild.

5.6 Which Attributes and Selectors Do You Need?

To scrape data from a webpage, you first have to identify the tag and attribute combinations that you are interested in gathering.

To find your elements of interest, you have three options. These go from hardest to easiest but also from most to least robust.

- right click + “inspect page source” (F12)
- right click + “inspect”
- Rvest [Selector Gadget](#) (very useful but fallible)

Inspecting the source of some familiar websites can be a useful way to get your head around these concepts. Beware though that sophisticated webpages can be quite intimidating. A good place to start is with simpler, static websites such as personal websites, rather than the dynamic webpages of online retailers or social media platforms.

5.7 Reading HTML with {rvest}

With `{rvest}`, reading a html page can be as simple as loading in tabular data.

```
html <- rvest::read_html("https://www.zakvarty.com/professional/teaching.html")
```

The `class` of the resulting object is an `xml_document`. This type of object is from the low-level package `{xml2}`, which allows you to read xml files into R.

```
class(html)
#> [1] "xml_document" "xml_node"
```

We can see that this object is split into several components: first is some metadata on the type of document we have scraped, followed by the head and then the body of that html document.

```
html
#> {html_document}
#> <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
#> [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF ...
#> [2] <body class="nav-fixed">\n\n<div id="quarto-search-results"></div>\n ...
```

We have several possible approaches to extracting information from this document.

5.8 Extracting HTML elements

In `{rvest}` you can extract a single element with `html_element()`, or all matching elements with `html_elements()`. Both functions take a document object and one or more CSS selectors as inputs.

```
library(rvest)

html %>% html_elements("h1")
#> {xml_nodeset (1)}
#> [1] <h1>Teaching</h1>
html %>% html_elements("h2")
#> {xml_nodeset (2)}
#> [1] <h2 id="toc-title">On this page</h2>
#> [2] <h2 class="anchored" data-anchor-id="course-history">Course History</h2>
html %>% html_elements("p")
#> {xml_nodeset (7)}
#> [1] <p>I am fortunate to have had the opportunity to teach in a variety of ...
#> [2] <p>Developing and teaching a number of modules in statistics, data sci ...
#> [3] <p>Supervising undergraduate, postgraduate and doctoral research proje ...
#> [4] <p>Adapting and leading short courses on scientific writing and commun ...
#> [5] <p>Running workshops and computer labs for undergraduate and postgradu ...
#> [6] <p>Speaking at univerisity open days and providing one-to-one tuition ...
#> [7] <p>I am an associate fellow of the Higher Education Academy, which you ...
```

You can also combine and nest these selectors. For example you might want to extract all links that are within paragraphs *and* all second level headers.

```
html %>% html_elements("p a,h2")
#> {xml_nodeset (3)}
#> [1] <h2 id="toc-title">On this page</h2>
#> [2] <a href="https://www.advance-he.ac.uk/fellowship/associate-fellowship" ...
#> [3] <h2 class="anchored" data-anchor-id="course-history">Course History</h2>
```

5.9 Extracting Data From HTML Elements

Now that we've got the elements we care about extracted from the complete document. But how do we get the data we need out of those elements?

You'll usually get the data from either the contents of the HTML element or else from one of its attributes. If you're really lucky, the data you need will already be formatted for you as a HTML table or list.

5.9.1 Extracting text

The functions `rvest::html_text()` and `rvest::html_text2()` can be used to extract the plain text contents of an HTML element.

```
html %>%
  html_elements("#teaching li") %>%
  html_text2()
#> [1] "Developing and teaching a number of modules in statistics, data science and data e
#> [2] "Supervising undergraduate, postgraduate and doctoral research projects."
#> [3] "Adapting and leading short courses on scientific writing and communication."
#> [4] "Running workshops and computer labs for undergraduate and postgraduate modules."
#> [5] "Speaking at univerisity open days and providing one-to-one tuition to high school
```

The difference between `html_text()` and `html_text2()` is in how they handle whitespace. In HTML whitespace and line breaks have very little influence over how the code is interpreted by the computer (this is similar to R but very different from Python). `html_text()` will extract the text as it is in the raw html, while `html_text2()` will do its best to extract the text in a way that gives you something similar to what you'd see in the browser.

5.9.2 Extracting Attributes

Attributes are also used to record information that you might like to collect. For example, the destination of links are stored in the `href` attribute and the source of images is stored in the `src` attribute.

As an example of this, consider trying to extract the twitter link from the icon in the page footer. This is quite tricky to locate in the html source, so I used the [Selector Gadget](#) to help find the correct combination of elements.

```
html %>% html_element(".compact:nth-child(1) .nav-link")
#> {html_node}
#> <a class="nav-link" href="https://www.twitter.com/zakvarty">
#> [1] <i class="bi bi-twitter" role="img">\n</i>
```

To extract the `href` attribute from the scraped element, we use the `rvest::html_attr()` function.

```
html %>%
  html_elements(".compact:nth-child(1) .nav-link") %>%
  html_attr("href")
#> [1] "https://www.twitter.com/zakvarty"
```

Note: `rvest::html_attr()` will always return a character string (or list of character strings). If you are extracting an attribute that describes a quantity, such as the width of an image, you'll need to convert this from a string to your required data type. For example, if the width is measured in pixels you might use `as.integer()`.

5.9.3 Extracting tables

HTML tables are composed in a similar, nested manner to LaTeX tables.

There are four main elements to know about that make up an HTML table:

- `<table>`,
- `<tr>` (table row),
- `<th>` (table heading),
- `<td>` (table data).

Here's our simple example data, formatted as an HTML table:

```
html_2 <- minimal_html("
  <table>
```

```

<tr>
  <th>Name</th>
  <th>Number</th>
</tr>
<tr>
  <td>A</td>
  <td>1</td>
</tr>
<tr>
  <td>B</td>
  <td>2</td>
</tr>
<tr>
  <td>C</td>
  <td>3</td>
</tr>
</table>
")

```

Since tables are a common way to store data, `{rvest}` includes a useful function `html_table()` that converts directly from an HTML table into a tibble.

```

html_2 %>%
  html_element("table") %>%
  html_table()
#> # A tibble: 3 x 2
#>   Name Number
#>   <chr>  <int>
#> 1 A      1
#> 2 B      2
#> 3 C      3

```

Applying this to our real scraped data we can easily extract the table of taught courses.

```

html %>%
  html_element("table") %>%
  html_table()
#> # A tibble: 31 x 3
#>   Year      Course                                Role
#>   <chr>    <chr>                                <chr>
#> 1 "2022-23" Data Science                                Lecturer
#> 2 ""      Ethics in Data Science I, II and III            Lecturer

```

```
#> 3 ""          Data Ethics for Digital Chemistry      Lecturer
#> 4 ""          Y1 research projects: point process models Lecturer
#> 5 "2021-22"  Supervised Learning                  Lecturer
#> 6 ""          Ethics in Data Science I             Lecturer
#> # i 25 more rows
```

5.10 Tip for Building Tibbles

When scraping data from a webpage, your end-goal is typically going to be constructing a `data.frame` or a tibble.

If you are following our description of tidy data, you'll want each row to correspond some repeated unit on the HTML page. In this case, you should

1. Use `html_elements()` to select the elements that contain each observation unit;
2. Use `html_element()` to extract the variables from each of those observations.

Taking this approach guarantees that you'll get the same number of values for each variable, because `html_element()` always returns the same number of outputs as inputs. This is vital when you have missing data - when not every observation unit has a value for every variable of interest.

As an example, consider this extract of text about the [starwars dataset](#).

```
starwars_html <- minimal_html("
  <ul>
    <li><b>C-3P0</b> is a <i>droid</i> that weighs <span class='weight'>167 kg</span></li>
    <li><b>R2-D2</b> is a <i>droid</i> that weighs <span class='weight'>96 kg</span></li>
    <li><b>Yoda</b> weighs <span class='weight'>66 kg</span></li>
    <li><b>R4-P17</b> is a <i>droid</i></li>
  </ul>
")
```

This is an unordered list where each list item corresponds to one observational unit (one character from the starwars universe). The name of the character is given in bold, the character species is specified in italics and the weight of the character is denoted by the `.weight` class. However, some characters have only a subset of these variables defined: for example Yoda has no species entry.

If we try to extract each element directly, our vectors of variable values are of different lengths. We don't know where the missing values should be, so we can't line them back up to make a tibble.


```

starwars_html %>% html_elements("b") %>% html_text2()
#> [1] "C-3P0" "R2-D2" "Yoda" "R4-P17"
starwars_html %>% html_elements("i") %>% html_text2()
#> [1] "droid" "droid" "droid"
starwars_html %>% html_elements(".weight") %>% html_text2()
#> [1] "167 kg" "96 kg" "66 kg"

```

What we should do instead is start by extracting all of the list item elements using `html_elements()`. Once we have done this, we can then use `html_element()` to extract each variable for all characters. This will pad with NAs, so that we can collate them into a tibble.

```

starwars_characters <- starwars_html %>% html_elements("li")

starwars_characters %>% html_element("b") %>% html_text2()
#> [1] "C-3P0" "R2-D2" "Yoda" "R4-P17"
starwars_characters %>% html_element("i") %>% html_text2()
#> [1] "droid" "droid" NA "droid"
starwars_characters %>% html_element(".weight") %>% html_text2()
#> [1] "167 kg" "96 kg" "66 kg" NA

tibble::tibble(
  name = starwars_characters %>% html_element("b") %>% html_text2(),
  species = starwars_characters %>% html_element("i") %>% html_text2(),
  weight = starwars_characters %>% html_element(".weight") %>% html_text2()
)
#> # A tibble: 4 x 3
#>   name species weight
#>   <chr> <chr> <chr>
#> 1 C-3P0 droid 167 kg
#> 2 R2-D2 droid 96 kg
#> 3 Yoda <NA> 66 kg
#> 4 R4-P17 droid <NA>

```

5.11 Session Information

R version 4.3.1 (2023-06-16)

Platform: x86_64-apple-darwin20 (64-bit)

locale: en_US.UTF-8|en_US.UTF-8|en_US.UTF-8|C|en_US.UTF-8|en_US.UTF-8

attached base packages: *stats, graphics, grDevices, utils, datasets, methods* and *base*

other attached packages: *rvest(v.1.0.3)*

loaded via a namespace (and not attached): *vctrs(v.0.6.5), httr(v.1.4.7), cli(v.3.6.2), knitr(v.1.45), rlang(v.1.1.2), xfun(v.0.41), stringi(v.1.8.3), jsonlite(v.1.8.8), glue(v.1.6.2), selectr(v.0.4-2), htmltools(v.0.5.7), fansi(v.1.0.6), rmarkdown(v.2.25), pander(v.0.6.5), evaluate(v.0.23), tibble(v.3.2.1), fastmap(v.1.1.1), yaml(v.2.3.8), lifecycle(v.1.0.4), stringr(v.1.5.1), compiler(v.4.3.1), Rcpp(v.1.0.11), pkgconfig(v.2.0.3), rstudioapi(v.0.15.0), digest(v.0.6.33), R6(v.2.5.1), utf8(v.1.2.4), pillar(v.1.9.0), curl(v.5.2.0), magrittr(v.2.0.3), tools(v.4.3.1) and xml2(v.1.3.6)*

6 APIs

i Note

Effective Data Science is still a work-in-progress. This chapter is largely complete and just needs final proof reading.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

6.1 Acquiring Data Via an API

We've already established that you can't always rely on tidy, tabular data to land on your desk.

Sometimes you are going to have to go out and gather data for yourself. We have already seen how to scrape information directly from the HTML source of a webpage. But surely there has to be an easier way. Thankfully, there often is!

In this chapter we will cover the basics of obtaining data via an API. This material draws together the [Introduction to APIs](#) book by Brian Cooksey and the [DIY web data](#) section of STAT545 at the University of British Columbia.

6.2 Why do I need to know about APIs?

An API, or application programming interface, is a set of rules that allows different software applications to communicate with each other.

As a data scientist, you will often need to access data that is stored on remote servers or in cloud-based services. APIs provide a convenient way for data scientists to programmatically retrieve this data, without having to manually download data sets or and process them locally on their own computer.

This has multiple benefits including automation and standardisation of data sharing.

- **Automation:** It is much faster for a machine to process a data request than a human. Having a machine handling data requests also scales much better as either the number or the complexity of data requests grows. Additionally, there is a lower risk of introducing human error. For example, a human might accidentally share the wrong data, which can have serious legal repercussions.
- **Standardisation:** Having a machine process data requests requires the format of these requests and the associated responses to be standardised. This allows data sharing and retrieval to become a reproducible and programmatic aspect of our work.

6.3 What is an API?

So then, if APIs are so great, what exactly are they?

In human-to-human communication, the set of rules governing acceptable behaviour is known as etiquette. Depending on when or where you live, social etiquette can be rather strict. The rules for computer-to-computer communication take this to a whole new level, because with machines there can be no room left for interpretation.

The set of rules governing interactions between computers or programmes is known as a **protocol**.

APIs provide a standard protocol for different programs to interact with one another. This makes it easier for developers to build complex systems by leveraging the functionality of existing services and platforms. The benefits of working in a standardised and modular way apply equally well to sharing data as they do to writing code or organising files.

There are two sides to communication and when *machines* communicate these are known as the **server** and the **client**.

Servers can seem intimidating, because unlike your laptop or mobile phone they don't have their own input and output devices; they have no keyboard, no monitor, and no a mouse. Despite this, servers are just regular computers that are designed to store data and run programmes. Servers don't have their own input or output devices because they are intended to be used *remotely*, via another computer. There is no need for a screen or a mouse if the user is miles away. Nothing scary going on here!

People often find clients much less intimidating - they are simply any other computer or application that might contact the sever.

6.4 HTTP

This leads us one step further down the rabbit-hole. An API is a protocol that defines the rules of how applications communicate with one another. But how does this communication happen?

HTTP (Hypertext Transfer Protocol) is the dominant mode of communication on the World Wide Web. You can see the secure version of HTTP, HTTPS, at the start of most web addresses up at the top of your browser. For example:

```
https://www.zakvarty.com/blog
```

HTTP is the foundation of data communication on the web and is used to transfer files (such as text, images, and videos) between web servers and clients.

To understand HTTP communications, I find it helpful to imagine the client and the server as being a customer and a waiter at a restaurant. The client makes some request to the server, which then tries to comply before giving a response. The server might respond to confirm that the request was completed successfully. Alternatively, the server might respond with an error message, which is (hopefully) informative about why the request could not be completed.

This request-response model is the basis for HTTP, the communication system used by the majority of APIs.

6.5 HTTP Requests

An HTTP request consists of:

- Uniform Resource Locator (URL) [unique identifier for a thing]
- Method [tells server the type of action requested by client]
- Headers [meta-information about request, e.g. device type]
- Body [Data the client wants to send to the server]

6.5.1 URL

The URL in a HTTP request specifies where that request is going to be made, for example `http://example.com`.

6.5.2 Method

The action that the client wants to take is indicated by a set of well-defined methods or HTTP verbs. The most common HTTP verbs are GET, POST, PUT, PATCH, and DELETE.

The GET verb is used to retrieve a resource from the server, such as a web page or an image. The POST verb is used to send data to the server, such as when submitting a form or uploading a file. The PUT verb is used to replace a resource on the server with a new one, while the PATCH verb is used to update a resource on the server without replacing it entirely. Finally, the DELETE verb is used to delete a resource from the server.

In addition to these common HTTP verbs, there are also several less frequently used verbs. These are used for specialized purposes, such as requesting only the headers of a resource, or testing the connectivity between the client and the server.

6.5.3 Header

The request headers contain meta-information about the request. This is where information about the device type would be included within the request.

6.5.4 Body

Finally, the body of the request contains the data that the client is providing to the server.

6.6 HTTP Responses

When the server receives a request it will attempt to fulfil it and then send a response back to the client.

A response has a similar structure to a request apart from:

- responses **do not have** a URL,
- responses **do not have** a method,
- responses **have** a status code.

6.6.1 Status Codes

The status code is a 3 digit number, each of which has a specific meaning. Some common error codes that you might (already have) come across are:

- 200: Success,
- 404: Page not found (all 400s are errors),
- 503: Page down.

In a data science context, a successful response will return the requested data within the data field. This will most likely be given in JSON or XML format.

6.7 Authentication

Now that we know *how* applications communicate, you might ask how we can control who has access to the API and what types of request they can make. This can be done by the server setting appropriate permissions for each client. But then how does the server verify that the client is really who is claims to be?

Authentication is a way to ensure that only authorized clients are able to access an API. This is typically done by the server requiring each client to provide some secret information that uniquely identifies them, whenever they make requests to the API. This information allows the API server to validate the authenticity this user before it authorises the request.

6.7.1 Basic Authentication

There are various ways to implement API authentication.

Basic authentication involves each legitimate client having a username and password. An encrypted version of these is included in the **Authorization** header of the HTTP request. If the hear matches with the server's records then the request is processed. If not, then a special status code (401) is returned to the client.

Basic authentication is dangerous because it does not put any restrictions on what a client can do once they are authorised. Additional, individualised restrictions can be added by using an alternative authentication scheme.

6.7.2 API Key Authentication

An API key is long, random string of letters and numbers that is assigned to each authorised user. An API key is distinct from the user’s password and keys are typically issued by the service that provides an API. Using keys rather than basic authentication allows the API provider to track and limit the usage of their API.

For example, a provider may issue a unique API key to each developer or organization that wants to use the API. The provider can then limit access to certain data. They could also limit the number of requests that each key can make in a given time period or prevent access to certain administrative functions, like changing passwords or deleting accounts.

Unlike Basic Authentication, there is no standard way of a client sharing a key with the server. Depending on the API this might be in the `Authorization` field of the header, at the end of the URL (`http://example.com?api_key=my_secret_key`), or within the body of the data.

6.8 API wrappers

We’ve learned a lot about how the internet works. Fortunately, a lot of the time we won’t have to worry about all of that new information other than for debugging purposes.

In the best case scenario, a very kind developer has written a “wrapper” function for the API. These wrappers are functions in R that will construct the HTTP request for you. If you are particularly lucky, the API wrapper will also format the response for you, converting it from XML or JSON back into an R object that is ready for immediate use.

6.9 {geonames} wrapper

[rOpenSci](#) has a curated list of many wrappers for accessing scientific data using R. We will focus on the [GeoNames API](#), which gives open access to a geographical database. To access this data, we will use wrapper functions provided by the `{geonames}` package.

The aim here is to illustrate the important steps of getting started with a new API.

6.9.1 Set up

Before we can get any data from the GeoNames API, we first need to do a little bit of set up.

1. Install and load `{geonames}` from CRAN


```
#install.packages("geonames")
library(geonames)
```

2. Create a [user account](#) for the GeoNames API

[GeoNames Home](#) | [Postal Codes](#) | [Download / Webservice](#) | [About](#) search

GeoNames user account

Login

Username

Password

remember me on this computer

[\[I forgot my password\]](#)

or create a new user account


Username
username may only include characters, digits and underscore

Email

Confirm Email

Password

Confirm password

info@geonames.org 

[GeoNames Home](#) • [Postal Codes](#) • [Download / Webservice](#) • [Forum](#) • [Blog](#) • [Sitemap](#)

3. Activate the account (see activation email)
4. Enable the free web services for your GeoNames account by logging in at this [link](#).
5. Tell R your credentials for GeoNames.

Hello example_username

Welcome to GeoNames. We have created an account for you with the username 'example_username'. Please use the following link to activate your account :

https://www.geonames.org/activate/bg3ibKhX/example_username/

In case of questions or problems don't hesitate to get in touch with us at info@geonames.org.

Here some links you could find useful:

GeoNames User Manual : <http://www.geonames.org/manual.html>

GeoNames Blog : <https://geonames.wordpress.com>

GeoNames Forum : <https://forum.geonames.org>

GeoNames Mailinglist : <https://groups.google.com/group/geonames>

Your GeoNames team

Warning

We could use the following code to tell R our credentials, **but we absolutely should not**.

```
options(geonamesUsername = "example_username")
```

This would save our username as an environment variable, but it *also* puts our API credentials directly into the script. If we share the script with others (internally, externally or publicly) we would be sharing our credentials too. Not good!

6.9.2 Keep it Secret, Keep it Safe

The solution to this problem is to add our credentials as environment variables in our `.Rprofile` rather than in this script. The `.Rprofile` is an R script that is run at the start of every session. It can be created and edited directly, but can also be created and edited from within R.

To make/open your `.Rprofile` use the `edit_r_profile()` function from the `{usethis}` package.

```
library(usethis)
usethis::edit_r_profile()
```

Within this file, add `options(geonamesUsername="example_username")` on a new line, remembering to replace `example_username` with your own GeoNames username.

The final step is to **check this this file ends with a blank line**, save it and restart R. Then we are all set to start using `{geonames}`.

This set up procedure is indicative of most API wrappers, but of course the details will vary between each API. This is why good documentation is important!

If you are using `{renv}` to track the versions of R and the packages you are using take care. For `{renv}` to be effective you have to place your project level `.Rprofile` under version control - this means you might accidentally share your API credentials.

A workaround for this problem is to create an R file that you `source()` from within the project level `.Rprofile` but is included in `.gitignore`, so that your credentials remain secret.

6.9.3 Using `{geonames}`

GeoNames has a whole host of [different geo-datasets](#) that you can explore. As a first example, let's get all of the geo-tagged wikipedia articles that are within 1km of Imperial College London.

```
imperial_coords <- list(lat = 51.49876, lon = -0.1749)
search_radius_km <- 1

imperial_neighbours <- geonames::GNfindNearbyWikipedia(
  lat = imperial_coords$lat,
  lng = imperial_coords$lon,
  radius = search_radius_km,
  lang = "en",           # english language articles
  maxRows = 500         # maximum number of results to return
)
```

Looking at the structure of `imperial_neighbours` we can see that it is a data frame with one row per geo-tagged wikipedia article.

```
str(imperial_neighbours)
#> 'data.frame':   204 obs. of  13 variables:
#> $ summary      : chr  "The Department of Mechanical Engineering is responsible for tea
#> $ elevation     : chr  "20" "18" "19" "24" ...
#> $ feature       : chr  "edu" "edu" "landmark" "edu" ...
#> $ lng           : chr  "-0.1746" "-0.1748" "-0.17425" "-0.1757" ...
#> $ distance      : chr  "0.0335" "0.0494" "0.0508" "0.0558" ...
#> $ rank          : chr  "81" "91" "90" "96" ...
#> $ lang          : chr  "en" "en" "en" "en" ...
#> $ title         : chr  "Department of Mechanical Engineering, Imperial College London" "
```

```
#> $ lat          : chr "51.498524" "51.4992" "51.49897222222222" "51.4987" ...
#> $ wikipediaUrl: chr "en.wikipedia.org/wiki/Department_of_Mechanical_Engineering%2C_Im
#> $ countryCode : chr NA "AE" NA "GB" ...
#> $ thumbnailImg: chr NA NA NA NA ...
#> $ geoNameId   : chr NA NA NA NA ...
```

To confirm we have the correct location we can inspect the title of the first five neighbours.

```
imperial_neighbours$title[1:5]
#> [1] "Department of Mechanical Engineering, Imperial College London"
#> [2] "Imperial College Business School"
#> [3] "Exhibition Road"
#> [4] "Imperial College School of Medicine"
#> [5] "Department of Civil and Environmental Engineering, Imperial College London"
```

Nothing too surprising here, mainly departments of the college and Exhibition Road, which runs along one side of the campus. These sorts of check are important - I initially forgot the minus in the longitude and was getting results in East London!

6.10 What if there is no wrapper?

If there is not a wrapper function, we can still access APIs fairly easily using the `{httr}` package.

We will look at an example using [OMDb](#), which is an open source version of [IMDb](#), to get information about the movie Mean Girls.

To use the OMDb API you will once again need to [request a free API key](#), follow a verification link and add your API key to your `.Rprofile`.

```
# Add this to .Rprofile, pasting in your own API key
options(OMDB_API_Key = "PASTE YOUR KEY HERE")
```

You can then restart R and safely access your API key from within your R session.

```
# Load your API key into the current R session
ombd_api_key <- getOption("OMDB_API_Key")
```

Using the documentation for the API, requests have URLs of the following form, where terms in angular brackets should be replaced by you.

```
http://www.omdbapi.com/?t=<TITLE>&y=<YEAR>&plot=<LENGTH>&r=<FORMAT>&apikey=<API_KEY>
```

With a little bit of effort, we can write a function that composes this type of request URL for us. We will use the `{glue}` package to help us join strings together.

Running the example we get:

```
mean_girls_request <- omdb_url(  
  title = "mean+girls",  
  year = "2004",  
  plot = "short",  
  format = "json",  
  api_key = getOption("OMDB_API_Key"))
```

We can then use the `{httr}` package to construct our request and store the response we get.

```
#> [1] 200
```

Thankfully it was a success! If you get a 401 error code here, check that you have clicked the activation link for your API key.

The full structure of the response is quite complicated, but we can easily extract the requested data using `content()`

```
#> $Title  
#> [1] "Mean Girls"  
#>  
#> $Year  
#> [1] "2004"  
#>  
#> $Rated  
#> [1] "PG-13"  
#>  
#> $Released  
#> [1] "30 Apr 2004"  
#>  
#> $Runtime  
#> [1] "97 min"  
#>  
#> $Genre  
#> [1] "Comedy"  
#>  
#> $Director  
#> [1] "Mark Waters"  
#>
```

```

#> $Writer
#> [1] "Rosalind Wiseman, Tina Fey"
#>
#> $Actors
#> [1] "Lindsay Lohan, Jonathan Bennett, Rachel McAdams"
#>
#> $Plot
#> [1] "Cady Heron is a hit with The Plastics, the A-list girl clique at her new school, unt
#>
#> $Language
#> [1] "English, German, Vietnamese, Swahili"
#>
#> $Country
#> [1] "United States, Canada"
#>
#> $Awards
#> [1] "7 wins & 25 nominations"
#>
#> $Poster
#> [1] "https://m.media-amazon.com/images/M/MV5BMjE1MDQ4MjI1OV5BMl5BanBnXkFtZTcwNzcwODAzMw@@
#>
#> $Ratings
#> $Ratings[[1]]
#> $Ratings[[1]]$Source
#> [1] "Internet Movie Database"
#>
#> $Ratings[[1]]$Value
#> [1] "7.1/10"
#>
#>
#> $Ratings[[2]]
#> $Ratings[[2]]$Source
#> [1] "Rotten Tomatoes"
#>
#> $Ratings[[2]]$Value
#> [1] "84%"
#>
#>
#> $Ratings[[3]]
#> $Ratings[[3]]$Source
#> [1] "Metacritic"
#>
#> $Ratings[[3]]$Value

```

```
#> [1] "66/100"
#>
#>
#>
#> $Metascore
#> [1] "66"
#>
#> $imdbRating
#> [1] "7.1"
#>
#> $imdbVotes
#> [1] "425,799"
#>
#> $imdbID
#> [1] "tt0377092"
#>
#> $Type
#> [1] "movie"
#>
#> $DVD
#> [1] "01 Aug 2013"
#>
#> $BoxOffice
#> [1] "$86,058,055"
#>
#> $Production
#> [1] "N/A"
#>
#> $Website
#> [1] "N/A"
#>
#> $Response
#> [1] "True"
```

6.11 Wrapping up

We have learned a bit more about how the internet works, the benefits of using an API to share data and how to request data from Open APIs.

When obtaining data from the internet it's vital that you keep your credentials safe, and that don't do more work than is needed.

- Keep your API keys out of your code. Store them in your `.Rprofile` (and make sure this is not under version control!)
- Scraping is always a last resort. Is there an API already?
- Writing your own code to access an API can be more painful than necessary.
- Don't repeat other people, if a suitable wrapper exists then use it.

6.12 Session Information

R version 4.3.1 (2023-06-16)

Platform: x86_64-apple-darwin20 (64-bit)

locale: en_US.UTF-8|en_US.UTF-8|en_US.UTF-8|C|en_US.UTF-8|en_US.UTF-8

attached base packages: *stats*, *graphics*, *grDevices*, *utils*, *datasets*, *methods* and *base*

other attached packages: *geonames*(v.0.999)

loaded via a namespace (and not attached): *digest*(v.0.6.33), *R6*(v.2.5.1), *fastmap*(v.1.1.1), *xfun*(v.0.41), *glue*(v.1.6.2), *knitr*(v.1.45), *htmltools*(v.0.5.7), *rmarkdown*(v.2.25), *cli*(v.3.6.2), *pander*(v.0.6.5), *compiler*(v.4.3.1), *httr*(v.1.4.7), *rstudioapi*(v.0.15.0), *tools*(v.4.3.1), *curl*(v.5.2.0), *evaluate*(v.0.23), *Rcpp*(v.1.0.11), *yaml*(v.2.3.8), *rlang*(v.1.1.2) and *jsonlite*(v.1.8.8)

Checklist

i Note

Effective Data Science is still a work-in-progress. This chapter is largely complete and just needs final proof reading.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

Videos / Chapters

- [Tabular Data](#) (27 min) [\[slides\]](#)
- [Web Scraping](#) (22 min) [\[slides\]](#)
- [APIs](#) (25 min) [\[slides\]](#)

Reading

Use the [Acquiring and Sharing Data section](#) of the reading list to support and guide your exploration of this week's topics. Note that these texts are divided into core reading, reference materials and materials of interest.

Tasks

Core:

- Revisit the Projects that you explored on Github last week. This time look for any data or documentation files.
 - Are there any file types that are new to you?
 - If so, are there packages or helper function that would let you read this data into R?
 - Why might you not find many data files on Github?

- Play [CSS Diner](#) to familiarise yourself with some CSS selectors.
- Identify 3 APIs that give access to data on topics that interest you. Write a post on the discussion forum describing the APIs and use one of them to load some data into R.
- Scraping Book Reviews:
 - Visit the Amazon page for R for Data Science. Write code to scrape the percentage of customers giving each “star” rating (5 , ..., 1).
 - Turn your code into a function that will return a tibble of the form:

product	n_review	percent_5_star	percent_4_star	percent_3_star	percent_2_star	percent_1_star	url
example_name	1000	20	20	20	20	20	www.example.com

- Generalise your function to work for other Amazon products, where the function takes as input a vector of product names and an associated vector of URLs.
- Use your function to compare the reviews of the following three books: [R for Data Science](#), [R packages](#) and [ggplot2](#).

Bonus:

- Add this function to the R package you made last week, remembering to add tests and documentation.

Live Session

In the live session we will begin with a discussion of this week’s tasks. We will then work through some examples of how to read data from non-standard sources.

Please come to the live session prepared to discuss the following points:

- Roger Peng states that files can be imported and exported using `readRDS()` and `saveRDS()` for fast and space efficient data storage. What is the downside to doing so?
- What data types have you come across (that we have not discussed already) and in what context are they used?
- What do you have to give greater consideration to when scraping data than when using an API?

Part III

Data Exploration and Visualisation

Introduction

Now that we have read our raw data into R we can start getting our data science project moving and being to see some initial returns on the time and effort that we have invested so far.

In this section we will explore how to wrangle, explore and visualise the data that forms the basis of our projects. These skills are often overlooked by folks coming into data science as being “soft skills” compared to modelling. However, I would argue that this is not the case because each of these tasks requires its own specialist knowledge and tools.

Additionally, these task make up the majority of data scientist’s work and are often where we can add the most value to an organisation. At this stage in a project we turn useless, messy data into a form that can be used; we derive initial insights from this data while making minimal assumptions; and we communicate all of this in an accurate and engaging way, to drive decision making both within and outwith the organisation.

7 Data Wrangling

i Note

Effective Data Science is still a work-in-progress. This chapter is largely complete and just needs final proof reading.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

7.1 What is Data Wrangling?

Okay, so you've got some data. That's a great start!

You might have had it handed to you by a collaborator, [requested it via an API](#) or [scraped it from the raw html of a webpage](#). In the worst case scenario, you're an *actual* scientist (not just a *data* one) and you spent the last several months of your life painstakingly measuring flower petals or car parts. Now we really want to do something useful with that data.

We've seen already how you can load the data into R and pivot between wider and longer formats, but that probably isn't enough to satisfy your curiosity. You want to be able to view your data, manipulate and subset it, create new variables from existing ones and cross-reference your dataset with others. All of these are things possible in R and are known under various collective names including data manipulation, data munging and data wrangling.

I've decided to use the term data wrangling here. That's because data manipulation sounds boring as heck and data munging is both unpleasant to say and makes me imagine we are squelching through some sort of information swamp.

In what follows, I'll give a fly-by tour of tools for data wrangling in R, showing some examples along the way. I'll focus on some of the most common and useful operations and link out to some more extensive guides for wrangling your data in R, that you can refer back to as you need them.



7.2 Example Data Sets

To demonstrate some standard skills we will use two datasets. The `mtcars` data comes built into any R installation. The second data set we will look at is the `penguins` data from `{palmerpenguins}`.

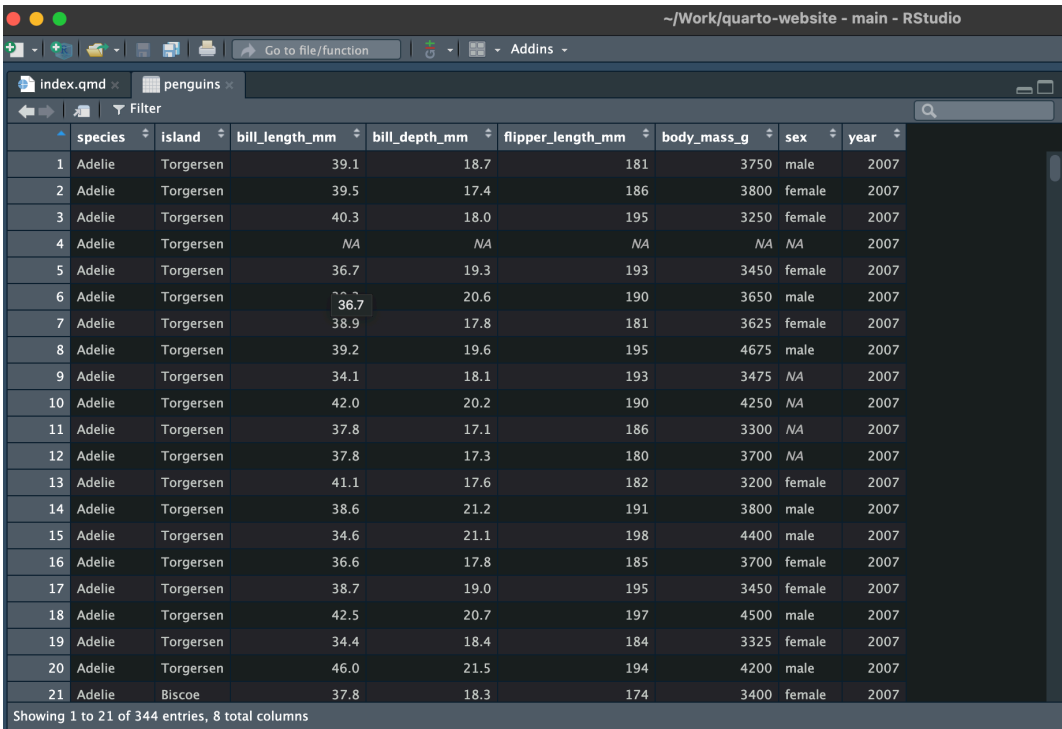
```
library(palmerpenguins)
penguins <- palmerpenguins::penguins
cars <- datasets::mtcars
```

7.3 Viewing Your Data

7.3.1 View()

The `View()` function can be used to create a spreadsheet-like view of your data. In RStudio this will open as a new tab.

`View()` will work for any “matrix-like” R object, such as a tibble, data frame, vector or matrix. Note the capital letter - the function is called `View()`, not `view()`.



	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
1	Adelle	Torgersen	39.1	18.7	181	3750	male	2007
2	Adelle	Torgersen	39.5	17.4	186	3800	female	2007
3	Adelle	Torgersen	40.3	18.0	195	3250	female	2007
4	Adelle	Torgersen	NA	NA	NA	NA	NA	2007
5	Adelle	Torgersen	36.7	19.3	193	3450	female	2007
6	Adelle	Torgersen	36.7	20.6	190	3650	male	2007
7	Adelle	Torgersen	38.9	17.8	181	3625	female	2007
8	Adelle	Torgersen	39.2	19.6	195	4675	male	2007
9	Adelle	Torgersen	34.1	18.1	193	3475	NA	2007
10	Adelle	Torgersen	42.0	20.2	190	4250	NA	2007
11	Adelle	Torgersen	37.8	17.1	186	3300	NA	2007
12	Adelle	Torgersen	37.8	17.3	180	3700	NA	2007
13	Adelle	Torgersen	41.1	17.6	182	3200	female	2007
14	Adelle	Torgersen	38.6	21.2	191	3800	male	2007
15	Adelle	Torgersen	34.6	21.1	198	4400	male	2007
16	Adelle	Torgersen	36.6	17.8	185	3700	female	2007
17	Adelle	Torgersen	38.7	19.0	195	3450	female	2007
18	Adelle	Torgersen	42.5	20.7	197	4500	male	2007
19	Adelle	Torgersen	34.4	18.4	184	3325	female	2007
20	Adelle	Torgersen	46.0	21.5	194	4200	male	2007
21	Adelle	Bischoe	37.8	18.3	174	3400	female	2007

Showing 1 to 21 of 344 entries, 8 total columns

7.3.2 head()

For large data sets, you might not want (or be able to) view it all at once. You can then use `head()` to view the first few rows. The integer argument `n` specifies the number of rows you would like to return.

```
head(x = penguins, n = 3)
#> # A tibble: 3 x 8
#>   species island  bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
#>   <fct>  <fct>          <dbl>          <dbl>          <int>          <int>
#> 1 Adelie  Torgers~           39.1           18.7            181            3750
#> 2 Adelie  Torgers~           39.5           17.4            186            3800
#> 3 Adelie  Torgers~           40.3            18             195            3250
#> # i 2 more variables: sex <fct>, year <int>
```

7.3.3 str()

An alternative way to view a data set that is large or has a complicated format is to examine its structure using `str()`. This is a useful way to inspect list-like objects with a nested structure.

```
str(penguins)
#> tibble [344 x 8] (S3: tbl_df/tbl/data.frame)
#> $ species      : Factor w/ 3 levels "Adelie","Chinstrap",...: 1 1 1 1 1 1 1 1 1 1 1 .
#> $ island       : Factor w/ 3 levels "Biscoe","Dream",...: 3 3 3 3 3 3 3 3 3 3 3 ...
#> $ bill_length_mm : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
#> $ bill_depth_mm : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
#> $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
#> $ body_mass_g    : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
#> $ sex           : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
#> $ year          : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

7.3.4 names()

If you just want to access the variable names you can do so with the `names()` function from base R.

```
names(penguins)
#> [1] "species"          "island"           "bill_length_mm"
```



```
#> [4] "bill_depth_mm"      "flipper_length_mm" "body_mass_g"
#> [7] "sex"                "year"
```

Similarly, you can explicitly access the row and column names of a data frame or tibble using `colnames()` or `rownames()`.

```
colnames(cars)
#> [1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear"
#> [11] "carb"
```

```
rownames(cars)
#> [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
#> [4] "Hornet 4 Drive"     "Hornet Sportabout"  "Valiant"
#> [7] "Duster 360"        "Merc 240D"          "Merc 230"
#> [10] "Merc 280"          "Merc 280C"          "Merc 450SE"
#> [13] "Merc 450SL"        "Merc 450SLC"        "Cadillac Fleetwood"
#> [16] "Lincoln Continental" "Chrysler Imperial"  "Fiat 128"
#> [19] "Honda Civic"       "Toyota Corolla"     "Toyota Corona"
#> [22] "Dodge Challenger" "AMC Javelin"        "Camaro Z28"
#> [25] "Pontiac Firebird"  "Fiat X1-9"          "Porsche 914-2"
#> [28] "Lotus Europa"     "Ford Pantera L"     "Ferrari Dino"
#> [31] "Maserati Bora"    "Volvo 142E"
```

In the `cars` data, the car model are stored as the row names. This doesn't really jive with our idea of tidy data - we'll see how to fix that shortly.

7.4 Renaming Variables

7.4.1 `colnames()`

The function `colnames()` can be used to set, as well as to retrieve, column names.

```
cars_renamed <- cars
colnames(cars_renamed)[1] <- "miles_per_gallon"
colnames(cars_renamed)
#> [1] "miles_per_gallon" "cyl"           "disp"
#> [4] "hp"              "drat"         "wt"
#> [7] "qsec"           "vs"          "am"
#> [10] "gear"          "carb"
```

7.4.2 `dplyr::rename()`

We can also use functions from `{dplyr}` to rename columns. Let's alter the second column name.

```
library(dplyr)
cars_renamed <- rename(.data = cars_renamed, cylinders = cyl)
colnames(cars_renamed)
#> [1] "miles_per_gallon" "cylinders"      "disp"
#> [4] "hp"              "drat"          "wt"
#> [7] "qsec"           "vs"            "am"
#> [10] "gear"          "carb"
```

This could be done as part of a pipe if we were making many alterations.

```
cars_renamed <- cars_renamed %>%
  rename(displacement = disp) %>%
  rename(horse_power = hp) %>%
  rename(rear_axel_ratio = drat)

colnames(cars_renamed)
#> [1] "miles_per_gallon" "cylinders"      "displacement"
#> [4] "horse_power"     "rear_axel_ratio" "wt"
#> [7] "qsec"           "vs"            "am"
#> [10] "gear"          "carb"
```

When using the `{dplyr}` function `rename()`, you have to remember the format `new_name = old_name`. This matches the format used to create a data frame or tibble, but is the opposite order to the python function of the same name and often catches people out.

In the section on [creating new variables](#), we will see an alternative way of doing this by copying the column and then deleting the original.

7.5 Subsetting

7.5.1 Base R

In base R you can extract rows, columns and combinations thereof using index notation.

```
# First row
penguins[1, ]
```

```

#> # A tibble: 1 x 8
#>   species island  bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
#>   <fct>   <fct>          <dbl>         <dbl>           <int>       <int>
#> 1 Adelie  Torgers~          39.1           18.7             181         3750
#> # i 2 more variables: sex <fct>, year <int>

# First Column
penguins[ , 1]
#> # A tibble: 344 x 1
#>   species
#>   <fct>
#> 1 Adelie
#> 2 Adelie
#> 3 Adelie
#> 4 Adelie
#> 5 Adelie
#> 6 Adelie
#> # i 338 more rows

# Rows 2-3 of columns 1, 2 and 4
penguins[2:3, c(1, 2, 4)]
#> # A tibble: 2 x 3
#>   species island  bill_depth_mm
#>   <fct>   <fct>          <dbl>
#> 1 Adelie  Torgersen          17.4
#> 2 Adelie  Torgersen           18

```

Using negative indexing you can remove rows or columns

```

# Drop all but first row
penguins[-(2:344), ]
#> # A tibble: 1 x 8
#>   species island  bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
#>   <fct>   <fct>          <dbl>         <dbl>           <int>       <int>
#> 1 Adelie  Torgers~          39.1           18.7             181         3750
#> # i 2 more variables: sex <fct>, year <int>

# Drop all but first column
penguins[ , -(2:8)]
#> # A tibble: 344 x 1
#>   species

```

```

#> <fct>
#> 1 Adelie
#> 2 Adelie
#> 3 Adelie
#> 4 Adelie
#> 5 Adelie
#> 6 Adelie
#> # i 338 more rows

```

You can also select rows or columns by their names. This can be done using the bracket syntax ([]) or the dollar syntax (\$).

```

penguins[ ,"species"]
#> # A tibble: 344 x 1
#>   species
#>   <fct>
#> 1 Adelie
#> 2 Adelie
#> 3 Adelie
#> 4 Adelie
#> 5 Adelie
#> 6 Adelie
#> # i 338 more rows
penguins$species
#>   [1] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>   [8] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [15] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [22] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [29] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [36] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [43] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [50] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [57] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [64] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [71] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [78] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [85] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [92] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#>  [99] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#> [106] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#> [113] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie
#> [120] Adelie   Adelie   Adelie   Adelie   Adelie   Adelie   Adelie

```

```

#> [127] Adelie      Adelie      Adelie      Adelie      Adelie      Adelie      Adelie
#> [134] Adelie      Adelie      Adelie      Adelie      Adelie      Adelie      Adelie
#> [141] Adelie      Adelie      Adelie      Adelie      Adelie      Adelie      Adelie
#> [148] Adelie      Adelie      Adelie      Adelie      Adelie      Adelie      Gentoo
#> [155] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [162] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [169] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [176] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [183] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [190] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [197] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [204] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [211] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [218] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [225] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [232] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [239] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [246] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [253] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [260] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [267] Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo      Gentoo
#> [274] Gentoo      Gentoo      Gentoo      Chinstrap  Chinstrap  Chinstrap  Chinstrap
#> [281] Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap
#> [288] Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap
#> [295] Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap
#> [302] Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap
#> [309] Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap
#> [316] Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap
#> [323] Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap
#> [330] Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap
#> [337] Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap  Chinstrap
#> [344] Chinstrap
#> Levels: Adelie Chinstrap Gentoo

```

Since `penguins` is a tibble, these return different types of object. Subsetting a tibble with bracket syntax will return a tibble but extracting a column using the dollar syntax returns a vector of values.

7.5.2 `filter()` and `select()`

`{dplyr}` has two functions for subsetting, `filter()` subsets by rows and `select()` subsets by column.

In both functions you list what you would like to retain. Filter and select calls can be piped together to subset based on row and column values.

```
penguins %>%
  select(species, island, body_mass_g)
#> # A tibble: 344 x 3
#>   species island    body_mass_g
#>   <fct>   <fct>         <int>
#> 1 Adelie  Torgersen     3750
#> 2 Adelie  Torgersen     3800
#> 3 Adelie  Torgersen     3250
#> 4 Adelie  Torgersen      NA
#> 5 Adelie  Torgersen     3450
#> 6 Adelie  Torgersen     3650
#> # i 338 more rows
```

```
penguins %>%
  select(species, island, body_mass_g) %>%
  filter(body_mass_g > 6000)
#> # A tibble: 2 x 3
#>   species island body_mass_g
#>   <fct>   <fct>         <int>
#> 1 Gentoo  Biscoe          6300
#> 2 Gentoo  Biscoe          6050
```

Subsetting rows can be inverted by negating the `filter()` statement

```
penguins %>%
  select(species, island, body_mass_g) %>%
  filter(!(body_mass_g > 6000))
#> # A tibble: 340 x 3
#>   species island    body_mass_g
#>   <fct>   <fct>         <int>
#> 1 Adelie  Torgersen     3750
#> 2 Adelie  Torgersen     3800
#> 3 Adelie  Torgersen     3250
#> 4 Adelie  Torgersen     3450
#> 5 Adelie  Torgersen     3650
#> 6 Adelie  Torgersen     3625
#> # i 334 more rows
```

and dropping columns can be done by selecting all columns except the one(s) you want to drop.

```

penguins %>%
  select(species, island, body_mass_g) %>%
  filter(!(body_mass_g > 6000)) %>%
  select(!c(species, island))
#> # A tibble: 340 x 1
#>   body_mass_g
#>   <int>
#> 1         3750
#> 2         3800
#> 3         3250
#> 4         3450
#> 5         3650
#> 6         3625
#> # i 334 more rows

```

7.6 Creating New Variables

7.6.1 Base R

We can create new variables in base R by assigning a vector of the correct length to a new column name.

```
cars_renamed$weight <- cars_renamed$wt
```

If we then drop the original column from the data frame, this gives us an alternative way of renaming columns.

```

cars_renamed <- cars_renamed[ , -which(names(cars_renamed) == "wt")]
head(cars_renamed, n = 5)
#>   miles_per_gallon cylinders displacement horse_power
#> Mazda RX4          21.0         6           160          110
#> Mazda RX4 Wag     21.0         6           160          110
#> Datsun 710         22.8         4           108           93
#> Hornet 4 Drive     21.4         6           258          110
#> Hornet Sportabout 18.7         8           360          175
#>   rear_axel_ratio  qsec vs am gear carb weight
#> Mazda RX4         3.90 16.46 0 1  4   4  2.620
#> Mazda RX4 Wag     3.90 17.02 0 1  4   4  2.875
#> Datsun 710         3.85 18.61 1 1  4   1  2.320
#> Hornet 4 Drive     3.08 19.44 1 0  3   1  3.215

```

```
#> Hornet Sportabout          3.15 17.02  0  0   3   2  3.440
```

One thing to be aware of is that this operation does not preserve column ordering.

Generally speaking, code that relies on columns being in a specific order is fragile - it breaks easily. If possible, you should try to write your code in another way that's robust to column reordering. I've done that here when removing the `wt` column by looking up the column index as part of my code, rather than assuming it will always be the fourth column.

7.6.2 `dplyr::mutate()`

The function from `{dplyr}` to create new columns is `mutate()`. Let's create another column that has the car's weight in kilogrammes rather than tonnes.

```
cars_renamed <- cars_renamed %>%
  mutate(weight_kg = weight * 1000)

cars_renamed %>%
  select(miles_per_gallon, cylinders, displacement, weight, weight_kg) %>%
  head(n = 5)

#>           miles_per_gallon cylinders displacement weight weight_kg
#> Mazda RX4                21.0         6           160  2.620  2620
#> Mazda RX4 Wag            21.0         6           160  2.875  2875
#> Datsun 710                22.8         4           108  2.320  2320
#> Hornet 4 Drive            21.4         6           258  3.215  3215
#> Hornet Sportabout        18.7         8           360  3.440  3440
```

You can also create new columns that are functions of multiple other columns.

```
cars_renamed <- cars_renamed %>%
  mutate(cylinder_adjusted_mpg = miles_per_gallon / cylinders)
```

7.6.3 `rownames_to_column()`

One useful example of adding an additional row to a data frame is to convert its row names to a column of the data frame.

```
cars %>%
  mutate(model = rownames(cars_renamed)) %>%
  select(mpg, cyl, model) %>%
```



```

head(n = 5)
#>           mpg cyl      model
#> Mazda RX4      21.0   6      Mazda RX4
#> Mazda RX4 Wag  21.0   6      Mazda RX4 Wag
#> Datsun 710     22.8   4      Datsun 710
#> Hornet 4 Drive  21.4   6      Hornet 4 Drive
#> Hornet Sportabout 18.7   8      Hornet Sportabout

```

There's a neat function called `rownames_to_column()` in the `{tibble}` package. This will add the row names as the first column and remove the row names all in one step.

```

cars %>%
  tibble::rownames_to_column(var = "model") %>%
  head(n = 5)
#>           model mpg cyl disp  hp drat   wt  qsec vs am gear carb
#> 1      Mazda RX4 21.0   6  160 110 3.90 2.620 16.46 0  1    4    4
#> 2      Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02 0  1    4    4
#> 3      Datsun 710 22.8   4  108  93 3.85 2.320 18.61 1  1    4    1
#> 4      Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44 1  0    3    1
#> 5      Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02 0  0    3    2

```

7.6.4 `rowids_to_column()`

Another function from `{tibble}` adds the row id of each observation as a new column. This is often useful when ordering or combining tables.

```

cars %>%
  tibble::rowid_to_column(var = "row_id") %>%
  head(n = 5)
#>   row_id mpg cyl disp  hp drat   wt  qsec vs am gear carb
#> 1     1 21.0   6  160 110 3.90 2.620 16.46 0  1    4    4
#> 2     2 21.0   6  160 110 3.90 2.875 17.02 0  1    4    4
#> 3     3 22.8   4  108  93 3.85 2.320 18.61 1  1    4    1
#> 4     4 21.4   6  258 110 3.08 3.215 19.44 1  0    3    1
#> 5     5 18.7   8  360 175 3.15 3.440 17.02 0  0    3    2

```

7.7 Summaries

The `summarise()` function allows you to collapse a data frame into a single row, which using a summary statistic of your choosing.

We can calculate the average bill length of all penguins in a single `summarise()` function call.

```
summarise(penguins, average_bill_length_mm = mean(bill_length_mm))
#> # A tibble: 1 x 1
#>   average_bill_length_mm
#>   <dbl>
#> 1 NA
```

Since we have missing values, we might instead want to calculate the mean of the recorded values.

```
summarise(penguins, average_bill_length_mm = mean(bill_length_mm, na.rm = TRUE))
#> # A tibble: 1 x 1
#>   average_bill_length_mm
#>   <dbl>
#> 1 43.9
```

We can also use `summarise()` to gather multiple summaries in a single data frame.

```
bill_length_mm_summary <- penguins %>%
  summarise(
    mean = mean(bill_length_mm, na.rm = TRUE),
    median = median(bill_length_mm, na.rm = TRUE),
    min = min(bill_length_mm, na.rm = TRUE),
    q_0 = min(bill_length_mm, na.rm = TRUE),
    q_1 = quantile(bill_length_mm, prob = 0.25, na.rm = TRUE),
    q_2 = median(bill_length_mm, na.rm = TRUE),
    q_3 = quantile(bill_length_mm, prob = 0.75, na.rm = TRUE),
    q_4 = max(bill_length_mm, na.rm = TRUE))

bill_length_mm_summary
#> # A tibble: 1 x 8
#>   mean median  min  q_0  q_1  q_2  q_3  q_4
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 43.9  44.4 59.6 32.1 39.2 44.4 48.5 59.6
```

In all, this isn't overly exciting. You might rightly wonder why you'd want to use `summarise()` when we could just use the simpler base R calls directly.

One benefit of `summarise()` is that it provides certainty that the object returned will be of a certain class (a tibble) no matter what summary function is used. However, `summarise()` really comes into its own when you want to apply these summaries to distinct subgroups of the data.

7.8 Grouped Operations

The real benefit of `summarise()` comes from its combination with `group_by()`. This allows to you calculate the same summary statistics for each level of a factor with only one additional line of code.

Here we're re-calculating the same set of summary statistics we just found for all penguins, but for each individual species.

```
penguins %>%
  group_by(species) %>%
  summarise(
    mean = mean(bill_length_mm, na.rm = TRUE),
    median = median(bill_length_mm, na.rm = TRUE),
    min = min(bill_length_mm, na.rm = TRUE),
    q_0 = min(bill_length_mm, na.rm = TRUE),
    q_1 = quantile(bill_length_mm, prob = 0.25, na.rm = TRUE),
    q_2 = median(bill_length_mm, na.rm = TRUE),
    q_3 = quantile(bill_length_mm, prob = 0.75, na.rm = TRUE),
    q_4 = max(bill_length_mm, na.rm = TRUE))
#> # A tibble: 3 x 9
#>   species    mean median   min  q_0  q_1  q_2  q_3  q_4
#>   <fct>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 Adelie    38.8  38.8  46    32.1  36.8  38.8  36.8  46
#> 2 Chinstrap 48.8  49.6  58    40.9  46.3  49.6  46.3  58
#> 3 Gentoo   47.5  47.3  59.6  40.9  45.3  47.3  45.3  59.6
```

You can group by multiple factors to calculate summaries for each distinct combination of levels within your data set. Here we group by combinations of species and the island to which they belong.

```
penguin_summary_stats <- penguins %>%
  group_by(species, island) %>%
  summarise(
    mean = mean(bill_length_mm, na.rm = TRUE),
    median = median(bill_length_mm, na.rm = TRUE),
```

```

min = max(bill_length_mm, na.rm = TRUE),
q_0 = min(bill_length_mm, na.rm = TRUE),
q_1 = quantile(bill_length_mm, prob = 0.25, na.rm = TRUE),
q_2 = median(bill_length_mm, na.rm = TRUE),
q_3 = quantile(bill_length_mm, prob = 0.25, na.rm = TRUE),
q_4 = max(bill_length_mm, na.rm = TRUE))
#> `summarise()` has grouped output by 'species'. You can override using the
#> `.groups` argument.

```

```

penguin_summary_stats
#> # A tibble: 5 x 10
#> # Groups:   species [3]
#>   species  island    mean median  min   q_0   q_1   q_2   q_3   q_4
#>   <fct>    <fct>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 Adelie   Biscoe    39.0  38.7  45.6  34.5  37.7  38.7  37.7  45.6
#> 2 Adelie   Dream     38.5  38.6  44.1  32.1  36.8  38.6  36.8  44.1
#> 3 Adelie   Torgersen 39.0  38.9  46    33.5  36.7  38.9  36.7  46
#> 4 Chinstrap Dream     48.8  49.6  58    40.9  46.3  49.6  46.3  58
#> 5 Gentoo   Biscoe    47.5  47.3  59.6  40.9  45.3  47.3  45.3  59.6

```

7.8.1 Ungrouping

By default, each call to `summarise()` will undo one level of grouping. This means that our previous result was still grouped by species. We can see this in the tibble output above or by examining the structure of the returned data frame. This tells us that this is an S3 object of class `grouped_df`, which inherits its properties from a `tbl_df`, `tbl`, and `data.frame` objects.

```

class(penguin_summary_stats)
#> [1] "grouped_df" "tbl_df"      "tbl"        "data.frame"

```

Since we have grouped by two variables, `{dplyr}` expects us to use two `summary()` function calls before it will return a data frame (or tibble) that is not grouped. One way to satisfy this is to apply a second summary at the species level of grouping.

```

penguin_summary_stats %>%
  summarise_all(mean, na.rm = TRUE)
#> # A tibble: 3 x 10
#>   species  island    mean median  min   q_0   q_1   q_2   q_3   q_4
#>   <fct>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 Adelie   NA    38.8  38.7  45.2  33.4  37.0  38.7  37.0  45.2

```

```
#> 2 Chinstrap      NA 48.8 49.6 58 40.9 46.3 49.6 46.3 58
#> 3 Gentoo         NA 47.5 47.3 59.6 40.9 45.3 47.3 45.3 59.6
```

However, we won't always want to do apply another summary. In that case, we can undo the grouping using `ungroup()`. Remembering to ungroup is a common mistake and cause of confusion when working with multiple-group summaries.

```
ungroup(penguin_summary_stats)
#> # A tibble: 5 x 10
#>   species island      mean median   min  q_0  q_1  q_2  q_3  q_4
#>   <fct>   <fct>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 Adelie  Biscoe     39.0  38.7 45.6 34.5 37.7 38.7 37.7 45.6
#> 2 Adelie  Dream      38.5  38.6 44.1 32.1 36.8 38.6 36.8 44.1
#> 3 Adelie  Torgersen  39.0  38.9 46   33.5 36.7 38.9 36.7 46
#> 4 Chinstrap Dream     48.8  49.6 58   40.9 46.3 49.6 46.3 58
#> 5 Gentoo  Biscoe     47.5  47.3 59.6 40.9 45.3 47.3 45.3 59.6
```

There's an alternative method to achieve the same thing in a single step when using `{dplyr}` versions 1.0.0 and above. This is to set the `.by` parameter of the `summarise()` function call, which determines the grouping that should be applied to the original data frame, just for this one operation.

```
#> # A tibble: 3 x 2
#>   island      mean_bill_length_mm
#>   <fct>           <dbl>
#> 1 Torgersen         39.0
#> 2 Biscoe            45.3
#> 3 Dream             44.2
```

The `.by` argument applies to a single operation. This means that the result of the `summarise()` call will always be an ungrouped tibble, regardless of the number of grouping columns.

```
#> # A tibble: 5 x 3
#>   island species      mean_bill_length_mm
#>   <fct>   <fct>           <dbl>
#> 1 Torgersen Adelie         39.0
#> 2 Biscoe   Adelie         39.0
#> 3 Dream   Adelie         38.5
#> 4 Biscoe   Gentoo         47.5
#> 5 Dream   Chinstrap      48.8
```

See `?dplyr_by` for more information on using the `.by` argument with `summarise()` and other `{dplyr}` verbs.

7.9 Reordering Factors

R stored factors as integer values, which it then maps to a set of labels. Only factor levels that appear in your data will be assigned a coded integer value and the mapping between factor levels and integers will depend on the order that the labels appear in your data.

This can be annoying, particularly when your factor levels relate to properties that aren't numerical but do have an inherent ordering to them. In the example below, we have the t-shirt size of twelve people.

```
tshirts <- tibble::tibble(
  id = 1:12,
  size = as.factor(c("L", NA, "M", "S", "XS", "M", "XXL", "L", "XS", "M", "L", "S"))
)

levels(tshirts$size)
#> [1] "L" "M" "S" "XS" "XXL"
```

Irritatingly, the sizes aren't in order and extra large isn't included because it's not included in this particular sample. This leads to awkward looking summary tables and plots.

```
tshirts %>% group_by(size) %>% summarise(count = n())
#> # A tibble: 6 x 2
#>   size count
#>   <fct> <int>
#> 1 L      3
#> 2 M      3
#> 3 S      2
#> 4 XS     2
#> 5 XXL    1
#> 6 <NA>   1
```

We can fix this by creating a new variable with the factors explicitly coded in the correct order. We also need to specify that we should not drop empty groups as part of `group_by()`.

```
tidy_tshirt_levels <- c("XS", "S", "M", "L", "XL", "XXL", NA)

tshirts %>%
  mutate(size_tidy = factor(size, levels = tidy_tshirt_levels)) %>%
  group_by(size_tidy, .drop = FALSE) %>%
  summarise(count = n())
#> # A tibble: 7 x 2
```

```
#>   size_tidy count
#>   <fct>     <int>
#> 1 XS         2
#> 2 S          2
#> 3 M          3
#> 4 L          3
#> 5 XL         0
#> 6 XXL        1
#> # i 1 more row
```

7.10 Be Aware: Factors

As we have seen a little already, categorical variables can cause issues when wrangling and presenting data in R. All of these problems are solvable using base R techniques but the `{forcats}` package provides tools for the most common of these problems. This includes functions for changing the order of factor levels or the values with which they are associated.

Some examples functions from the package include:

- `fct_reorder()`: Reordering a factor by another variable.
- `fct_infreq()`: Reordering a factor by the frequency of values.
- `fct_relevel()`: Changing the order of a factor by hand.
- `fct_lump()`: Collapsing the least/most frequent values of a factor into “other”.

Examples of each of these can be found in the [forcats vignette](#) or the [factors chapter](#) of R for data science.

7.11 Be Aware: Strings

Working with and analysing text data is a skill unto itself. However, it is useful to be able to do some basic manipulation of character strings programmatically.

Because R was developed as a statistical programming language, it is well suited to the computational and modelling aspects of working with text data but the base R string manipulation functions can be a bit unwieldy at times.

The `{stringr}` package aims to combat this by providing useful helper functions for a range of text management problems. Even when not analysing text data these can be useful, such as when removing prefixes from many column names.

Suppose we wanted to keep only the text following an underscore in these column names. We could do that by using a regular expression to extract lower-case or upper-case letters which follow an underscore.

```
head(poorly_named_df)
#> # A tibble: 6 x 11
#>   observation_id  V1_A    V2_B    V3_C    V4_D    V5_E    V6_F    V7_G    V8_H
#>       <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1         1 -1.33  -0.147 -0.321  0.557  0.281  0.630  0.763 -0.167
#> 2         2  1.60   0.377  0.675 -0.934 -0.334 -0.661  2.33  -1.12
#> 3         3  0.904 -0.537  0.558  0.508 -1.65  0.0267 -1.49  0.197
#> 4         4  0.0468  0.738 -0.708 -0.662  0.241  0.737  -1.66  0.921
#> 5         5  0.846  0.0883  0.686 -1.37  -0.684 -1.63   0.659 -0.461
#> 6         6 -0.499 -3.10   1.19   0.839  0.118 -0.396 -0.737  1.18
#> # i 2 more variables: V9_I <dbl>, V10_J <dbl>
```

```
stringr::str_extract(names(poorly_named_df), pattern = "(?<=_)([a-zA-Z]+)")
#> [1] "id" "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

Alternatively, can avoid using [regular expressions](#). We can use `stringr::str_split()` to divide each column name at the underscore and keep only the second part of each string.

```
# split column names at underscores and inspect structure of resulting object
split_strings <- stringr::str_split(names(poorly_named_df), pattern = "_")
str(split_strings)
#> List of 11
#> $ : chr [1:2] "observation" "id"
#> $ : chr [1:2] "V1" "A"
#> $ : chr [1:2] "V2" "B"
#> $ : chr [1:2] "V3" "C"
#> $ : chr [1:2] "V4" "D"
#> $ : chr [1:2] "V5" "E"
#> $ : chr [1:2] "V6" "F"
#> $ : chr [1:2] "V7" "G"
#> $ : chr [1:2] "V8" "H"
#> $ : chr [1:2] "V9" "I"
#> $ : chr [1:2] "V10" "J"

# keep only the second element of each character vector in the list
purrr::map_chr(split_strings, function(x){x[2]})
#> [1] "id" "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```


Unless you plan to work extensively with text data, I would recommend that you look up such string manipulations as you need them. The [strings](#) section of R for Data Science is a useful starting point.

7.12 Be Aware: Date-Times

Remember [all the fuss](#) we made about storing dates in the ISO standard format? That was because dates and times are complicated enough to work on without adding extra ambiguity.

YYYY – MM – DD

Dates, times and time intervals have to reconcile two factors: the physical orbit of the Earth around the Sun and the social and geopolitical mechanisms that determine how we measure and record the passing of time. This makes the history of date and time records fascinating but also make working with this type of data complicated.

Moving from larger to smaller time spans: leap years alter the number of days in a year, months are of variable length (with February's length changing from year to year). If your data are measured in a place that uses daylight saving, then one day a year will be 23 hours long and another will be 25 hours long. To make things worse, the dates and the hour at which the clocks change are not uniform across countries, which might span multiple time zones and those time-zone boundaries can shift over time.

Even at the level of minutes and seconds we aren't safe - since the Earth's orbit is gradually slowing down and a leap second is added approximately every 21 months. Nor are things any better when looking at longer time scales or across cultures, where we might have to account for different calendars: months are added removed and altered over time, other calendar systems still take different approaches to measuring time and using different units and origin points.

With all of these issues you have to be very careful when working with date and time data. Functions to help you with this can be found in the `{lubridate}` package, with examples in the [dates and times](#) chapter of R for data science.

7.13 Be Aware: Relational Data

When the data you need are stored across two or more data frames you need to be able to cross-reference those and match up values for observational unit. This sort of data is known as relational data, and is used extensively in data science.

The variables you use to match observational units across data frames are known as *keys*. The primary key belongs to the first table and the foreign key belongs to the secondary table. There are various ways to join these data frames, depending on if you want to retain.

7.13.0.1 Join types

You might want to keep only observational units that have key variables values in both data frames, this is known as an inner join.

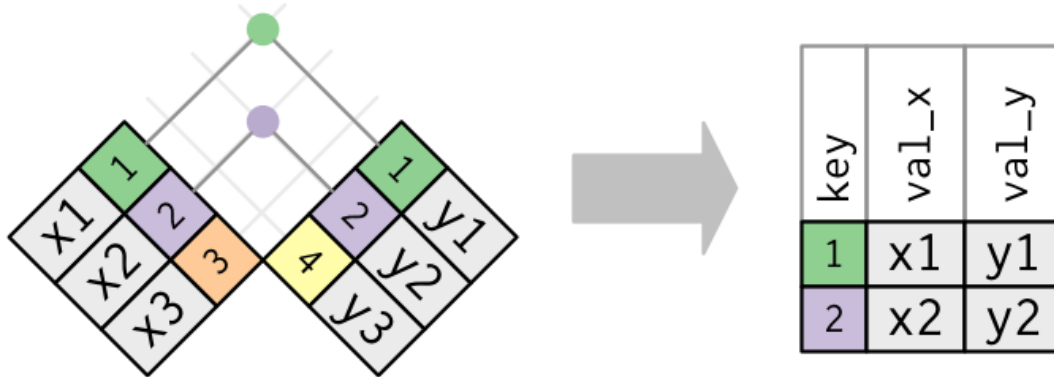


Figure 7.1: Inner join diagram. Source: R for Data Science

You might instead want to keep all units from the primary table but pad with NAs where there is not a corresponding foreign key in the second table. This results in an **(outer) left-join**.

Conversely, you might keep all units from the second table but pad with NAs where there is not a corresponding foreign key in the primary table. This is imaginatively named an **(outer) right-join**.

In the **(outer) full join**, all observational units from either table are retained and all missing values are padded with NAs.

Things get more complicated when keys don't uniquely identify observational units in either one or both of the tables. I'd recommend you start exploring these ideas with the [relational data](#) chapter of R for Data Science.

7.13.0.2 Why and where to learn more

Working with relational data is essential to getting any data science up and running out in the wilds of reality. This is because businesses and companies don't store all of their data in a huge single csv file. For one this isn't very efficient, because most cells would be empty. Secondly, it's not a very secure approach, since you can't grant partial access to the data. That's why information is usually stored in many data frames (more generically known as tables) within one or more databases.

These data silos are created, maintained, accessed and destroyed using a relational data base management system. These management systems use code to manage and access the stored

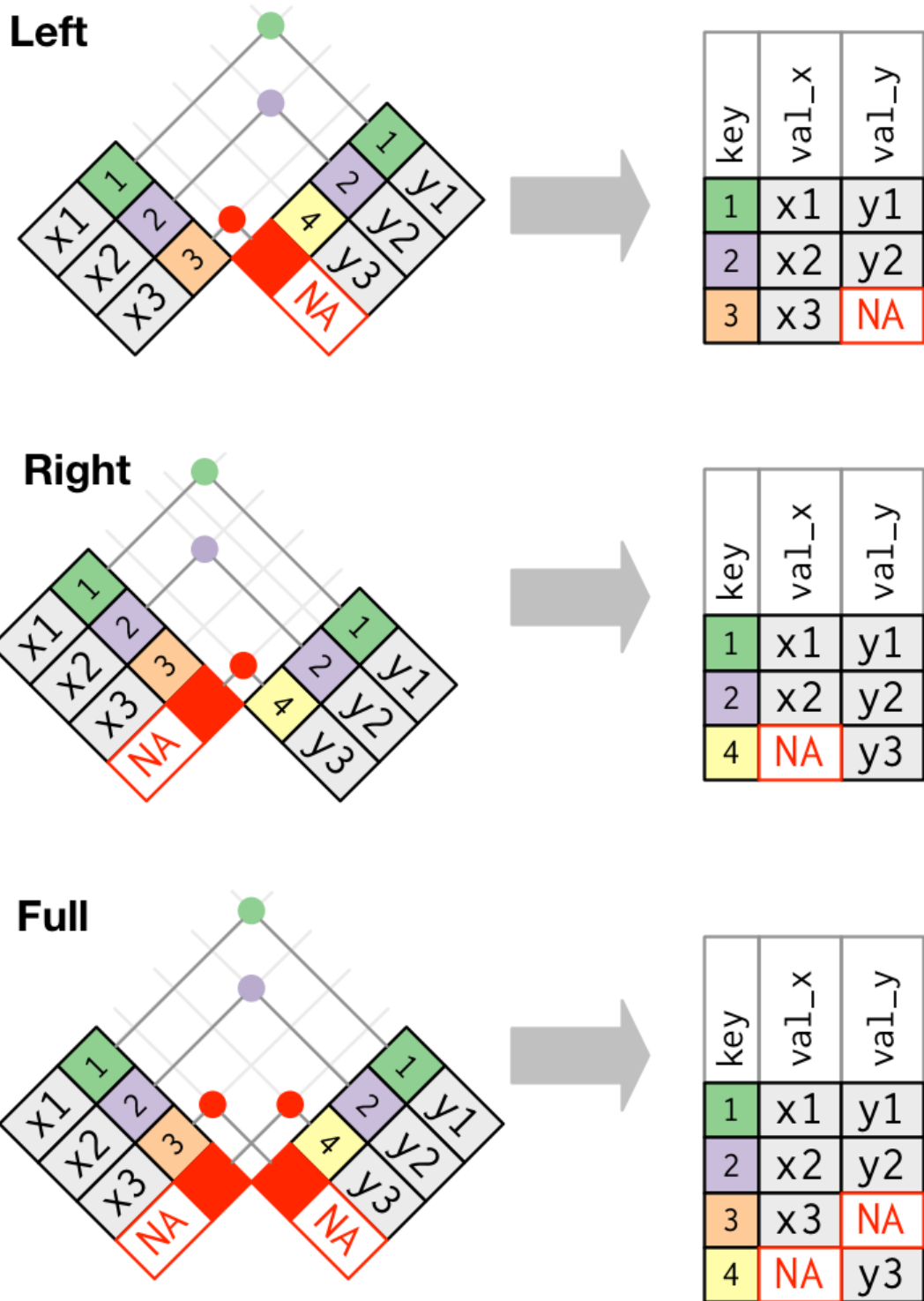


Figure 7.2: Diagram for left, right and outer joins. Source: R for Data Science

data, just like we have seen in the dplyr commands above. You might well have heard of the SQL programming language (and its many variants), which is a popular language for data base management and is the inspiration for the dplyr package and verbs.

If you'd like to learn more then there are many excellent introductory SQL books and courses, I'd recommend picking one that focuses on data analysis or data science unless you really want to dig into efficient storage and querying of databases.

7.14 Wrapping up

We have:

- Learned how to wrangle tabular data in R with {dplyr}
- Met the idea of relational data and {dplyr}'s relationship to SQL
- Become aware of some tricky data types and packages that can help.

7.15 Session Information

R version 4.3.1 (2023-06-16)

Platform: x86_64-apple-darwin20 (64-bit)

locale: en_US.UTF-8|en_US.UTF-8|en_US.UTF-8|C|en_US.UTF-8|en_US.UTF-8

attached base packages: *stats*, *graphics*, *grDevices*, *utils*, *datasets*, *methods* and *base*

other attached packages: *dplyr(v.1.1.4)* and *palmerpenguins(v.0.1.1)*

loaded via a namespace (and not attached): *vctrs(v.0.6.5)*, *cli(v.3.6.2)*, *knitr(v.1.45)*, *rlang(v.1.1.2)*, *xfun(v.0.41)*, *stringi(v.1.8.3)*, *purrr(v.1.0.2)*, *generics(v.0.1.3)*, *jsonlite(v.1.8.8)*, *glue(v.1.6.2)*, *htmltools(v.0.5.7)*, *fansi(v.1.0.6)*, *rmarkdown(v.2.25)*, *pander(v.0.6.5)*, *evaluate(v.0.23)*, *tibble(v.3.2.1)*, *fastmap(v.1.1.1)*, *yaml(v.2.3.8)*, *lifecycle(v.1.0.4)*, *stringr(v.1.5.1)*, *compiler(v.4.3.1)*, *Rcpp(v.1.0.11)*, *pkgconfig(v.2.0.3)*, *rstudioapi(v.0.15.0)*, *digest(v.0.6.33)*, *R6(v.2.5.1)*, *tidyselect(v.1.2.0)*, *utf8(v.1.2.4)*, *pillar(v.1.9.0)*, *magrittr(v.2.0.3)*, *tools(v.4.3.1)* and *withr(v.2.5.2)*

8 Exploratory Data Analysis

8.1 Introduction

Exploratory data analysis is an essential stage in any data science project. It allows you to become familiar with the data you are working with while also to identify potential strategies for progressing the project and flagging any areas of concern.

In this chapter we will look at three different perspectives on exploratory data analysis: its purpose for you as a data scientist, its purpose for the broader team working on the project and finally its purpose for the project itself.

8.2 Get to know your data

Let's first focus on an exploratory data analysis from our own point of view, as data scientists.

Exploratory data analysis (or EDA) is a process of examining a data set to understand its overall structure, contents, and the relationships between the variables it contains. EDA is an iterative process that's often done before building a model or making other data-driven decisions within a data science project.

Exploratory Data Analysis: quick and simple excerpts, summaries and plots to better understand a data set.

One key aspect of EDA is generating quick and simple summaries and plots of the data. These plots and summary statistics can help to quickly understand the distribution of and relationships between the recorded variables. Additionally, during an exploratory analysis you will familiarise yourself with the structure of the data you're working with and how that data was collected.

```
library("ggplot2")
library("GGally")

colours <- colorspace::adjust_transparency(
  col = PrettyCols::prettycols(name = "Fun", n = 3),
  alpha = 0.6)
```

```

ggpairs(iris,mapping = aes(col = Species)) +
  scale_colour_manual(values = colours) +
  scale_fill_manual(values = colours) +
  theme_minimal() +
  theme(axis.text = element_text(size = 6))

```

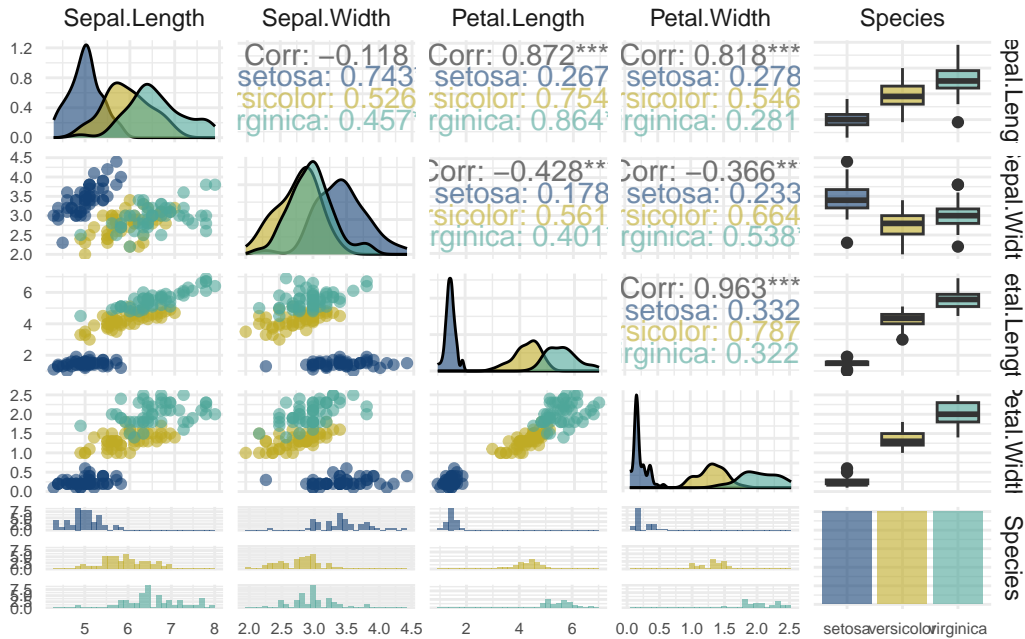


Figure 8.1: Investigating marginal and pairwise relationships in the Iris dataset.

Since EDA is an initial and iterative process, it's rare that any component of the analysis will be put into production. Instead, the goal is to get a general understanding of the data that can inform the next steps of the analysis.

In terms of workflow, this means that using one or more notebooks is often an effective way of organising your work during an exploratory analysis. This allows for rapid iteration and experimentation, while also providing a level of reproducibility and documentation. Since notebooks allow you to combine code, plots, tables and text in a single document, this makes it easy to share your initial findings with stakeholders and project managers.

8.3 Start a conversation

An effective EDA sets a precedent for open communication with the stakeholder and project manager.

We've seen the benefits of an EDA for you as a data scientist, but this isn't the only perspective.

One key benefit of an EDA is that it can kick-start your communication with subject matter experts and project managers. You can build rapport and trust early in a project's life cycle by sharing your preliminary findings with these stakeholders. This can lead to a deeper understanding of both the available data and the problem being addressed for everyone involved. If done well, it also starts to build trust in your work before you even begin the modelling stage of a project.

8.3.1 Communicating with specialists

Sharing an exploratory analysis will inevitably require a time investment. The graphics, tables, and summaries you produce need to be presented to a higher standard and explained in a way that is clear to a non-specialist. However, this time investment will often pay dividends because of the additional contextual knowledge that the domain-expert can provide. They have a deep understanding of the business or technical domain surrounding the problem. This can provide important insights that aren't in the data itself, but which are vital to the project's success.

As an example, these stakeholder conversations often reveal important features in the data generating or measurement process that should be accounted for when modelling. These details are usually left out of the data documentation because they would be immediately obvious to any specialist in that field.

8.3.2 Communicating with project manager

An EDA can sometimes allow us to identify cases where the strength of signal within the available data is clearly insufficient to answer the question of interest. By clearly communicating this to the project manager, the project can be postponed while different, better quality or simply more data are collected. It's important to note that this data collection is not trivial and can have a high cost in terms of both time and capital. It might be that collecting the data needed to answer a question will cost more than we're likely to gain from knowing that answer. Whether the project is postponed or cancelled, this constitutes a successful outcome for the project; the aim is to produce insight or profit - not to fit models for their own sake.

8.4 Scope Your Project

EDA is an initial assessment of whether the available data measure the correct values, in sufficient quality and quantity, to answer a particular question.

A third view on EDA is as an initial assessment of whether the available data measure the correct values, with sufficient quality and quantity, to answer a particular question. In order for EDA to be successful, it's important to take a few key steps.

First, it's important to formulate a specific question of interest or line of investigation and agree on it with the stakeholder. By having a clear question in mind, it will be easier to focus the analysis and identify whether the data at hand can answer it.

Next, it's important to make a record (if one doesn't already exist) of how the data were collected, by whom it was collected, what each recorded variable represents and the units in which they are recorded. This meta-data is often known as a data sheet or data dictionary. Having this information in written form is crucial when adding a new collaborator to a project, so that they can understand the data generating and measurement processes, and are aware of the quality and accuracy of the recorded values. The [Open Science Foundation](#) give a concise description of the meta-data you might list in a data dictionary.

8.5 Investigate Your Data

EDA is an opportunity to quantify data completeness and investigate the possibility of informative missingness.

In addition, it's essential to investigate and document the structure, precision, completeness and quantity of data available. This includes assessing the degree of measurement noise or misclassification in the data, looking for clear linear or non-linear dependencies between any of the variables, and identifying if any data are missing or if there's any structure to this missingness. Other data features to be aware of are the presence of any censoring or whether some values tend to be missing together.

Furthermore, a more advanced EDA might include a simulation study to estimate the amount of data needed to detect the smallest meaningful effect. This is more in-depth than a typical EDA but if you suspect that the signals within your data are weak relative to measurement noise, can help to demonstrate the limitations of the current line of enquiry with the information that is currently available.

8.6 What is not EDA?

It's important to understand that an exploratory data analysis is not the same thing as modelling. In particular is *not* the construction of your baseline model, which is sometimes called initial data analysis. Though it might inform the choice of baseline model, EDA is usually not model based. In an EDA, simple plots and summaries are used to identify patterns in the data that inform how you approach the rest of the project.

Some degree of statistical rigour can be added to EDA through the use of non-parametric techniques; methods like rolling averages, smoothing or partitioning can to help identify trends or patterns while making minimal assumptions about the data generating process.

```
library(tidyverse)
library(ismev)
library(cowplot)

data("dowjones", package = "ismev")

theme_dow_jones <- theme_minimal() +
  theme(
    axis.text = element_text(size = 10),
    axis.title = element_text(size = 9, face = "bold", hjust = 0.98, vjust = 2),
    plot.title = element_text(size = 15, hjust = 0.02),
    plot.subtitle = element_text(size = 12, hjust = 0.02)
  )

p1 <- dowjones %>%
  mutate(index_change = Index - lag(Index)) %>%
  filter(!is.na(index_change)) %>%
  ggplot(aes(x = Date, y = index_change)) +
  geom_point(col = rgb(0,0,0,0.2), size = 2) +
  geom_smooth(col = "darkorange", lwd = 1.5) +
  geom_vline(xintercept = as.Date("1998-06-01")) +
  ggtitle("Daily change in Dow Jones Index", "with smoothed estimate of mean change") +
  ylab("") +
  theme_dow_jones

p2 <- dowjones %>%
  mutate(index_change = Index - lag(Index)) %>%
  filter(!is.na(index_change)) %>%
  ggplot(aes(x = Date, y = index_change)) +
  geom_point(col = rgb(0, 0, 0, 0.2), size = 2) +
  geom_smooth(
    col = "darkorange",
    fill = colorspace::adjust_transparency("darkorange", 0.05),
    lwd = 1.5) +
  geom_vline(xintercept = as.Date("1998-06-01")) +
  ylab("") +
  coord_cartesian(ylim = c(-2, 12)) +
  theme_dow_jones
```

```
plot_grid(p1,p2)
```

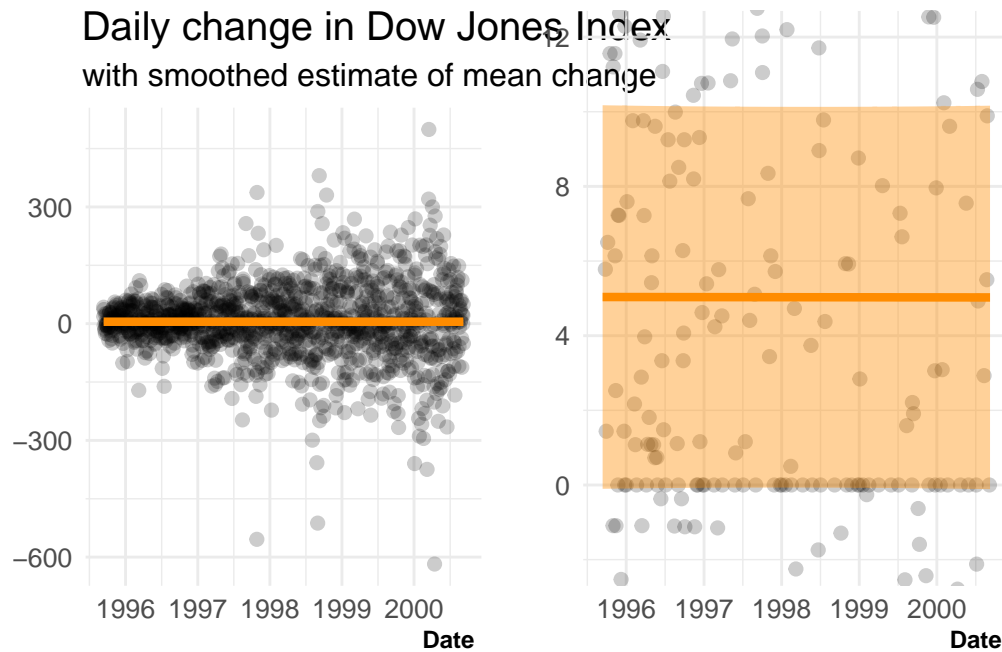


Figure 8.2: Daily change in Dow Jones Index with smoothed estimate of mean and 95% confidence interval.

```
knitr::kable(  
  x = dowjones %>%  
  mutate(index_change = Index - lag(Index)) %>%  
  mutate(after_june_98 = Date > as.Date("1998-06-01")) %>%  
  group_by(after_june_98) %>%  
  summarise(  
    mean = mean(index_change, na.rm = TRUE),  
    sd = sd(index_change, na.rm = TRUE)),  
  caption = "Mean and standard deviation of daily change in Dow Jones Index, before and af  
  )
```

Table 8.1: Mean and standard deviation of daily change in Dow Jones Index, before and after 1st of June 1998.

after_june_98	mean	sd
FALSE	5.916798	65.19093
TRUE	3.972929	119.56067

after_june_98	mean	sd
---------------	------	----

Though the assumptions in an EDA are often minimal it can help to make them explicit. For example, in this plot a moving averages is shown with a confidence band, but the construction of this band makes the implicit assumption that, at least locally, our observations have the same distribution and so are exchangeable.

Finally, EDA is not a prescriptive process. While I have given a lot of suggestions on what you might usually want to consider, there is no correct way to go about an EDA because it is so heavily dependent on the particular dataset, its interpretation and the task you want to achieve with it. This is one of the parts of data science that make it a craft that you hone with experience, rather than an algorithmic process. When you work in a particular area for a long time you develop domain-specific knowledge of common data quirks, which may or may not translate to other applications.

Now that we have a better idea of what is and what is not EDA, let's talk about the issue that an EDA tries to resolve and the other issues that it generates.

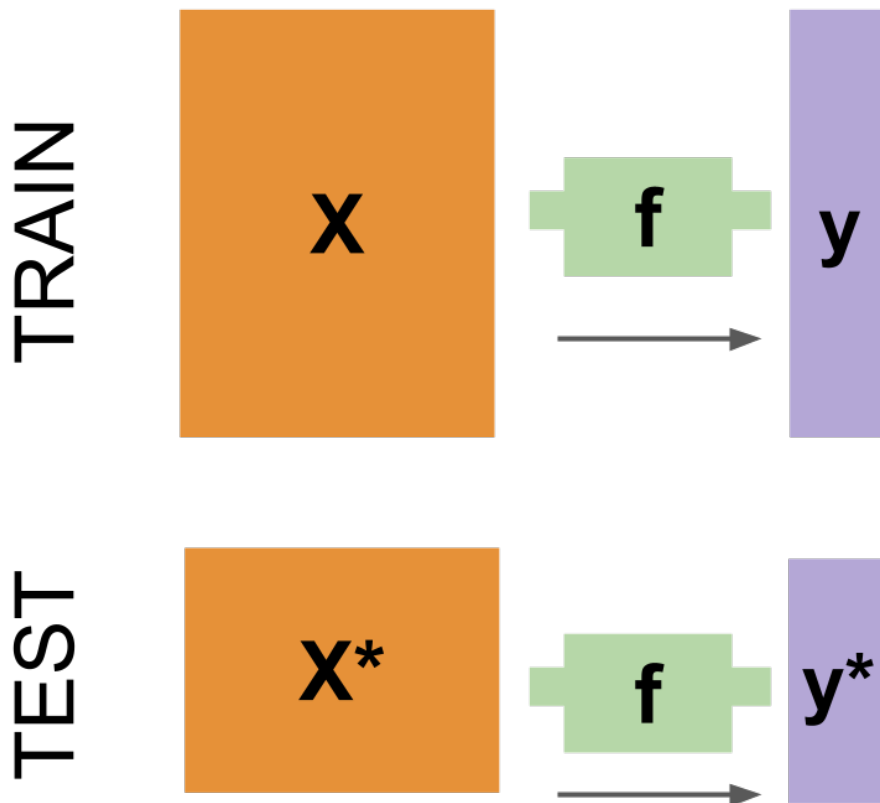
8.7 Issue: Forking Paths

In any data science project you have to make a sequence of very many decisions, each with many potential options. It is almost impossible to anticipate all of the decisions you will have to make ahead of time and even more difficult do decide how to proceed with each decision *a priori*.



Focusing in on only one small part of the data science process, we might consider picking a null or baseline model that we will then try and improve on. Should that null model make constant predictions, incorporate a simple linear trend or is something more flexible obviously needed? Do you have the option to try all of these or are you working under time constraints? Is there any domain knowledge that rules some of these null models out on contextual grounds?

An EDA lets you narrow down your options by looking at your data and helps you to decide what might be reasonable modelling approaches.



The problem that sneaks in here is data leakage. Formally this is where training data is included in test set but this sort of information leak can happen informally too. Usually this is because you've seen the data you're trying to model or predict and then selected your modelling approach based on that information.

Standard, frequentist statistical methods for estimation and testing assume no “peeking” of this type has occurred. If we use these methods without acknowledging that we have already observed our data then we will artificially inflate the significance of our findings. For example, we might be comparing two models: the first of which makes constant predictions with regard to a predictor, while the second includes a linear trend. We will of course use a statistical test to confirm that what we are seeing is unlikely by chance. However, we must be aware this test was only performed because we had previously examined at the data and noticed what looked to be a trend.

Similar issues arise in Bayesian approaches, particularly when constructing or eliciting prior distributions for our model parameters. One nice thing that we can do in the Bayesian setting is to simulate data from the prior predictive distribution and then get an expert to check that these datasets seem reasonable. However, it is often the case this expert is also the

person who collected the data we will soon be modelling. It's very difficult for them to ignore what they have seen, which leads to similar, subtle leakage problems. (For further discussion see the relevant sections of [Gelman et al. \(2020\)](#) or [Betancourt \(2020\)](#))

8.8 Correction Methods

There are various methods or corrections that we can apply during our testing and estimation procedures to ensure that our error rates or confidence intervals account for our previous “peeking” during EDA.

Examples of these corrections have been developed across many fields of statistics. In medical statistics we have approaches like the Bonferroni correction, to account for carrying out multiple hypothesis tests. In the change-point literature there are techniques for estimating a change location given that a change has been detected somewhere in a time series. While in the extreme value literature there are methods to estimate the required level of protection against rare events, given that the analysis was triggered by the current protections having been compromised.

```
# Simulate data
date <- seq.Date(
  from = as.Date("2018-03-01"),
  to = as.Date("2022-05-03"),
  by = "day")

sim_size <- length(date)

set.seed(1234)
surge_height <- rnorm(n = sim_size) + c(rep(-3, sim_size - 1), 3)
surface_height <- rnorm(n = sim_size) + rep(x = c(-5,-3), times = c(1234, sim_size - 1234))

waves <- tibble::tibble(date, surge_height, surface_height)

# Extreme Value Plot
p3 <- waves %>%
  mutate(is_extreme = surge_height > 1) %>%
  ggplot(aes(x = date, y = surge_height)) +
  geom_point(aes(color = is_extreme), show.legend = FALSE) +
  scale_color_manual(values = c("FALSE" = "grey40", "TRUE" = "darkorange")) +
  ylab("Wave surge height (m)") +
  theme_minimal() +
  theme(
```

```

axis.text = element_text(size = 10),
axis.title = element_text(size = 9, face = "bold", hjust = 0.98, vjust = 2),
plot.title = element_text(size = 15, hjust = 0.02),
plot.subtitle = element_text(size = 12, hjust = 0.02)
)

# Changepoint Plot
p4 <- waves %>%
  ggplot(aes(x = date, y = surface_height)) +
  geom_point(color = "grey40") +
  geom_vline(xintercept = as.Date("2021-07-17"),
            colour = "darkorange",
            linewidth = 1.2) +
  ylab("Sea surface height (m)") +
  theme_minimal() +
  theme(
    axis.text = element_text(size = 10),
    axis.title = element_text(size = 9, face = "bold", hjust = 0.98, vjust = 2),
    plot.title = element_text(size = 15, hjust = 0.02),
    plot.subtitle = element_text(size = 12, hjust = 0.02)
  )

plot_grid(p3,p4,nrow = 2, ncol = 1)

```

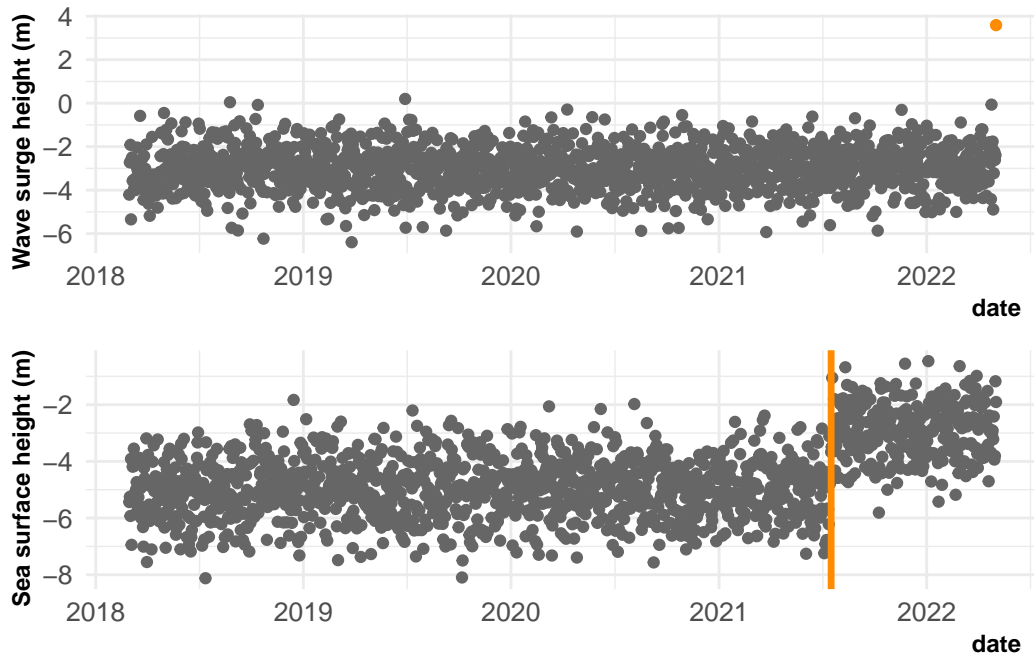


Figure 8.3: Example sea-height datasets where an analysis has been triggered by an extreme value (above) or a visually identified change in mean (below).

All of these corrections require us to make assumptions about the nature of the peeking. They are either very specific about the process that has occurred, or else are very pessimistic about how much information has been leaked. Developing such corrections to account for EDA isn't really possible, given its adaptive and non-prescriptive nature.

In addition to being either highly specific or pessimistic, these corrections can also be hard to derive and complicated to implement. This is why in settings where the power of tests or level of estimation is critically important, the entire analysis is pre-registered. In clinical trials, for example, every step of the analysis is specified before any data are collected. In data science this rigorous approach is rarely taken.

As statistically trained data scientists, it is important for us to remain humble about our potential findings and to suggest follow up studies to confirm the presence of any relationships we do find.

8.9 Learning More

In this chapter we have acknowledged that exploratory analyses are an important part of the data science workflow; this is true not only for us as data scientists, but also for the other people who are involved with our projects. We've also seen that an exploratory analysis can

help to guide the progression of our projects, but that in doing so we must take care to prevent and acknowledge the risk of data leakage.

If you want to explore this topic further, it can be quite challenging. Examples of good, exploratory data analyses can be difficult to come by because, unlike the papers and technical reports that they precede, exploratory analyses are not often made publicly available. Additionally, they are often kept out of public repositories because they are not as “polished” as the rest of the project. Personally, I think this is a shame and the culture on this is slowly changing.

For now, your best approach to learning about what makes a good exploratory analysis is to do lots of your own and to talk to your colleagues about their approaches. There are lots of list-articles out there claiming to give you a comprehensive list of steps for any exploratory analysis. These can be good for inspiration, but I strongly advise you against falling into the trap of treating these as prescriptive or foolproof.

Despite the name of the chapter, [Roger Peng’s EDA check-list](#) gives an excellent worked example of an exploratory analysis in R. In discussion article “[Exploratory Data Analysis for Complex Models](#)”, Andrew Gelman makes a more abstract discussion of both exploratory analyses (which happen before modelling) and confirmatory analyses (which happen afterwards).

9 Data Visualisation

9.1 Introduction

9.1.1 More than a pretty picture

Data visualisation is an integral part of your work as a data scientist.



Figure 9.1: Book jacket showing the effect of global temperatures rising. A series of vertical stripes represent the annual global average temperature, progressing from blue on the left to red on the right.

You'll use visualisations to rapidly explore new data sets, to understand their structure and to establish which types of model might be suitable for the task at hand. Visualisation is also vital during model evaluation and when check the validity of the assumptions on which that model is based. These are relatively technical uses of visualisation, but graphics have a much broader role within your work as an effective data scientist.

When well designed, plots, tables and animations can tell compelling stories that were once trapped within your data. They can also intuitively communicate the strength of evidence for your findings and draw attention to the most salient parts your argument.

Data visualisation is an amalgamation of science, statistics, graphic design and storytelling. It's multi-disciplinary nature means that we have to draw on all of our skills to ensure success. While there are certainly many ways to go wrong when visualising data, there are many more ways to get it right.

This chapter won't be a a step-by-step tutorial of how to visualise any specific type of data. Nor will it be a line-up of visualisations gone wrong. Instead, I hope to pose some questions that'll get you thinking critically about exactly what you want from each graphic that you produce.

There are (at least) five things that you should think about when producing any sort of data visualisation. We will consider each of these in turn.

9.2 Your Tools

9.2.1 Picking the right tool for the job

When you think of data visualisation, you might immediately think of impressive animations or complex, interactive dashboards that allow users to explore relationships within the data for themselves.

Such tools are no doubt impressive but they are by no means necessary for an effective data visualisation. In many cases there is no technology is needed at all. The history of data visualisation vastly pre-dates that of computers and some of the most effective visualisations remain analogue creations.

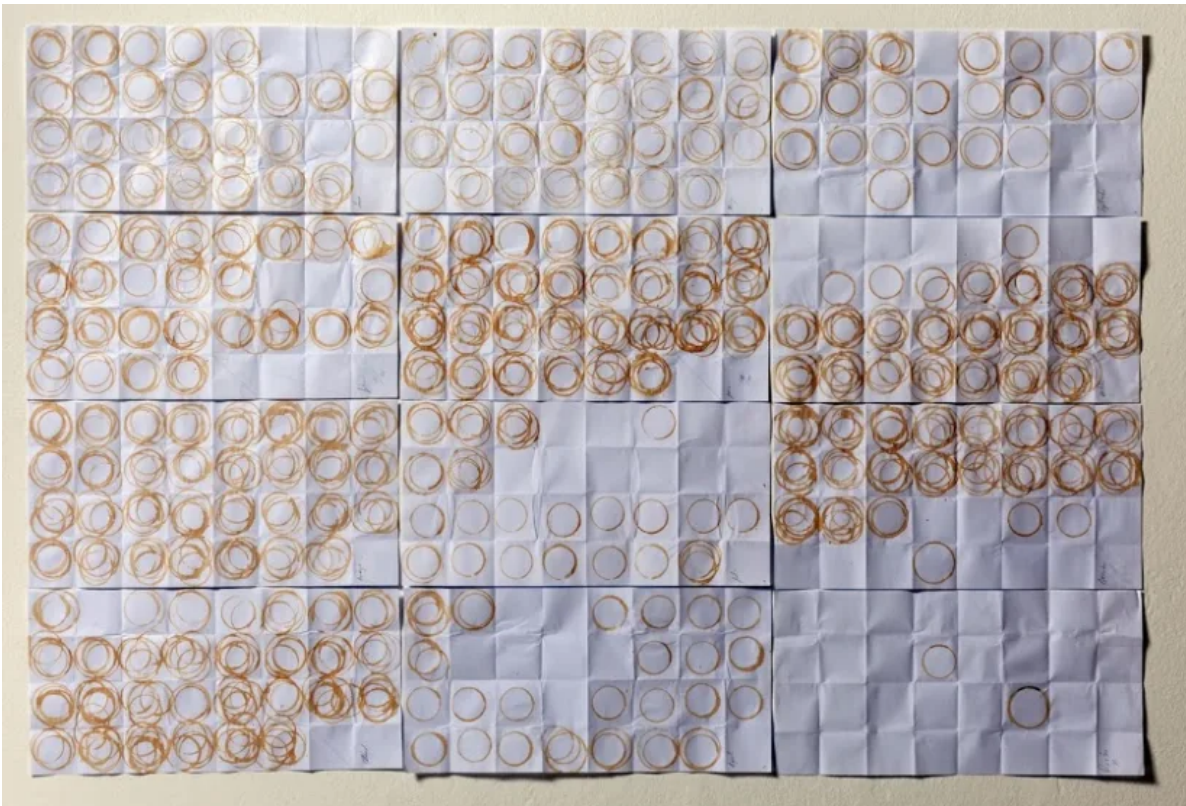


Figure 9.2: Coffee consumption, visualised. Jaime Serra Palou.

This visualisation of a year's coffee consumption is an ideal example. Displaying the number of cups of coffee in a bar chart or line graph would have been a more accurate way to collect and display this data, but that wouldn't have the same resonance or impact and it certainly wouldn't have been as memorable.

9.2.2 Analogue or Digital

9.2.2.1 Analogue Data Viz

Here we have another example of an analogue data visualisation that is created as part of data collection. Each member of the department is invited to place a Lego brick on a grid to indicate how much caffeine they have consumed and how much sleep they have had. The beauty of using Lego bricks here is that they are stackable and so create a bar plot over two dimensions.

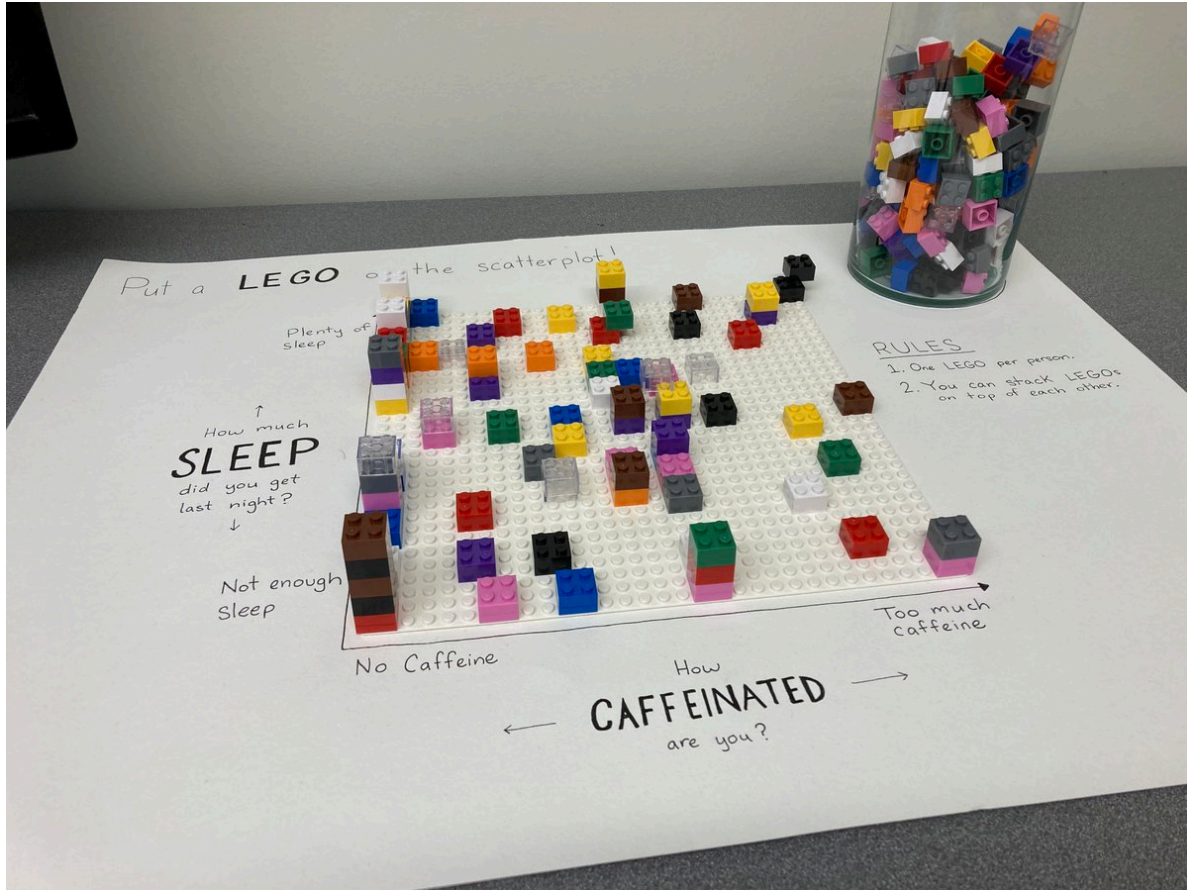


Figure 9.3: Caffeination vs sleep, shown in lego. Elsie Lee-Robbins

A third example can be found next to the tills in many supermarkets. Each customer is given a token as they pay for their goods. They can then drop this token into one of three large perspex containers as they leave the shop, each representing a different charity. At the end of the month £10,000 is split between the charities in proportion to the number of tokens. Because the containers are made from a transparent material you can see how the tokens are distributed, giving a visualisation of the level of support for each of the charities.

There are many other way of constructing a physical, analogue visualisation of your data and this doesn't need to be done as part of the data collection process. The simplest and perhaps most obvious most obvious is to create a plot of tabular data using a pen and paper.

9.2.2.2 Digital Data Viz

When it comes to digital tools for data visualisation you have a plethora of options. The most similar to pen-and-paper plotting is to draw your visualisations using a Photoshop, or an open source equivalent like Inkscape. The benefit here is that if you misplace a line or dot you can correct this small error without having to start all over again.

There are then more data-focused tools that have point-and-click interfaces. These are things like Excel's chart tools, or specialist visualisation software like Tableau. These are great because they scale with the quantity of data, so that you can plot larger amounts of raw data values that you wouldn't have the time or patience to do by hand - whether that's in an analogue or digital format.

Analogue and point-and-click approaches to visualisation have the shared limitation of not being reproducible, at least not without extensive documentation explaining how the graphic was created.

Using code to create your visualisations can resolve this reproducibility issue, and includes visualisation as a part of your larger, reproducible workflow for data science. Scripted visualisations also scale easily to large data sets and are easy to alter if any changes are required. The downside here is that there is a relatively steep learning curve to creating such visualisations, which is exactly what point-and-click methods are trying to avoid.

No matter how you produce your visualisations, the time spent on developing your skills in that medium is what buys you the ability to control and customise your creations. This upfront time investment will also often make you faster at producing future graphics in that medium.

Whenever you approach a new visualisation problem, you should pick your tools and medium judiciously. You have to balance your immediate needs for speed, accuracy and reproducibility against your current skill level and improving those skills in the medium to long term. Unfortunately, the only way to make good visualisations is to make lots of bad ones and even more mediocre ones first.

9.2.3 ggplot2

If your aim is to produce a wide range of high quality data visualisation using R, then the `{ggplot2}` package is one of the most versatile and well documented tools available to you.

The g's at the start of the package name stand for grammar of graphics. This is an opinionated, abstract approach to constructing data visualisations programmatically, by building them up slowly and adding additional plot elements one layer at a time.

A Layered Grammar of Graphics

Hadley WICKHAM

A grammar of graphics is a tool that enables us to concisely describe the components of a graphic. Such a grammar allows us to move beyond named graphics (e.g., the “scatterplot”) and gain insight into the deep structure that underlies statistical graphics. This article builds on Wilkinson, Anand, and Grossman (2005), describing extensions and refinements developed while building an open source implementation of the grammar of graphics for R, `ggplot2`.

The topics in this article include an introduction to the grammar by working through the process of creating a plot, and discussing the components that we need. The grammar is then presented formally and compared to Wilkinson's grammar, highlighting the hierarchy of defaults, and the implications of embedding a graphical grammar into a programming language. The power of the grammar is illustrated with a selection of examples that explore different components and their interactions, in more detail. The article concludes by discussing some perceptual issues, and thinking about how we can build on the grammar to learn how to create graphical “poems.”

Supplemental materials are available online.

Key Words: Grammar of graphics; Statistical graphics.

This idea of a “grammar of Graphics” was originally introduced by Leland Wilkinson. The paper shown by Hadley Wickham, and the associated `{ggplot2}` package popularised this approach within the R community. Like many of the tidyverse collection of packages that we have met already, `{ggplot2}` provides simple but specialised modular functions that can be composed to create complex visualisations.

If you'd like to learn how to use `{ggplot2}`, I wouldn't recommend starting with the paper nor would I recommend trying to get started with the docs alone. Instead, I would suggest you work through an introductory tutorial (e.g. [evolution of a ggplot](#) or [beautiful plotting in R](#)), or one of the resources linked within the [package documentation](#).

Once you have a grasp of the basic principles `{ggplot2}` (geometries, aesthetics, themes) the best way to improve is to make your own plots, using reference texts and other people's work as a guide. A great source of inspiration here is the [Tidy Tuesday](#) data visualisation challenge. You can [search for the challenge on Github](#) to inspect both the plots made by other people and the code that was used to make them.

9.3 Your Medium

9.3.1 Where is your visualisation going?

The second aspect that I recommend you think about before starting a data visualisation is where that graphic is going to be used. The intended location for your visualisation will influence both the composition of your graphic and also the amount of effort that you dedicate to it.



Analysis



Journalism

Presentations



Reports

For example, consider making an exploratory plot at the start of a project to improve your own understanding of the data and its structure. In this case you don't need to spend hours worrying about refining axis labels, colour schemes or which file format to save your work in.

If you are developing a figure to share in a daily stand-up meeting with your team then you should take a little more care to ensure that your work can be clearly understood by others. For example you might spend some time to ensure that the legend and axis labels are both large and sufficiently informative.

Further refinement will be required if you plan to use your visualisation in external presentation. Is the message of the visualisation immediately clear? Will the graphic still be clear when displayed in a boardroom or conference hall, or will it pixellate? Finally, how long will the audience have to interpret the visualisation while you are speaking? Even if slide decks are made available, very few audience members will actually refer to them before or after the presentation.

The opposing consideration has to be made when preparing a visualisation for a report or scientific paper. In print, plots and tables can be very small - particularly within two-column or grid layouts. You then have to be wary about the legibility of your smallest text (think values on axes) to ensure that your visualisation can be clearly understood, whether the document is being read zoomed-in on a computer screen or printed out in black and white.

9.3.2 File Types

To ensure that your graphics are suitable for the intended medium it is helpful to know a little bit about image file types.

There are two dominant types of image file: vector graphics and bitmap graphics.

Bitmap graphics store images as a grid of little squares and each of these pixels takes a single, solid colour. If you make a bitmap image large enough, either by zooming in or by using a really big screen, then these individual pixels become visible. Usually this isn't going to be your intention, so you need to ensure that the resolution of your graphic (its dimensions counted in pixels) is sufficiently large.

Vector graphics create images using continuous paths and fill the areas that these enclose with flat colour. These vector images can be enlarged as much as you like without the image quality becoming compromised. This is great for simple designs like logos, which have to be clear when used on both letterhead and billboards.

However, these vector graphics are more memory intensive than bitmap images, particularly when there are many distinct colours or objects within the image. This can be a particular problem in data science, for example when creating a scatter plot with many thousands of data points.



Figure 9.4: Exaggerated example of a pixellated image. Picture shows a stag standing in the countryside with a resolution of approximated 40 x 40 pixels.

It can often be useful to save both a bitmap and vector version of your most important graphics. This way you can use bitmap when you need small files that load quickly (like when loading a webpage) and vector graphics when you need your visualisation to stay sharp when enlarged (like when creating a poster or giving a presentation in an auditorium).

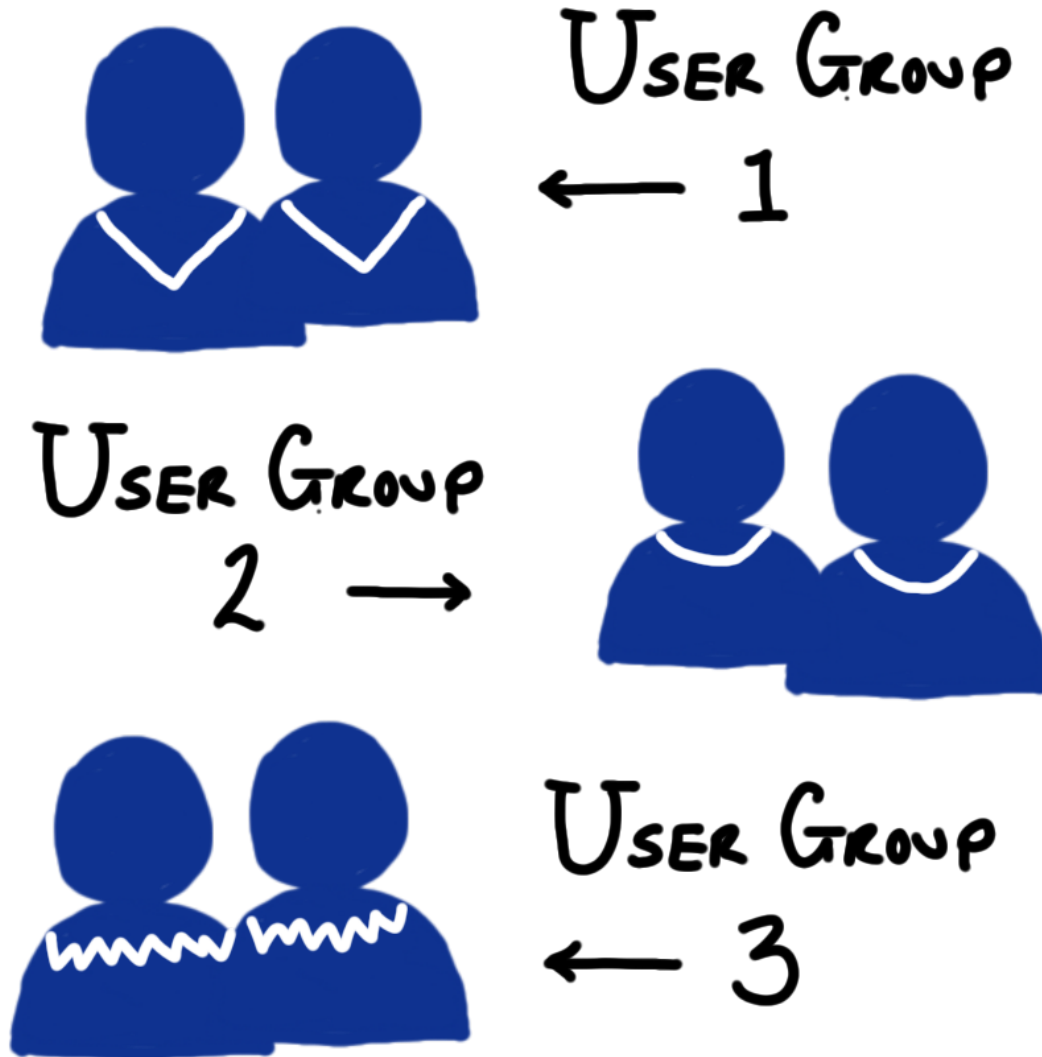
9.4 Your Audience

9.4.1 Know Your Audience

Data visualisations are a tool for communicating information. To make this communication as effective as possible, you have to target your delivery to the intended audience.

- Who is the intended audience for your visualisation?
- What knowledge do they bring with them?
- What assumptions and biases do they hold?

Creating *personas* for distinct user groups can be a helpful way to answer these questions, particularly when the user population are heterogeneous.



To know *how* to target your delivery to a particular audience, you first have to identify exactly who that is.

To make a compelling data visualisation you have to have some idea of the background knowledge that the viewer brings. Are they a specialist in statistics and data science or does their expertise lie in the area of application? Are the findings that you're presenting going to come as a surprise to the viewer or will they validate their pre-existing knowledge?

It's worth considering these prior beliefs and how strongly they are held when constructing your visualisation. Take the time to consider how this existing knowledge could alter or influence

the viewer's interpretation of what you're showing to them. If your conclusions go against their experience or knowledge then you need to design your visualisation to be as clear and persuasive as possible. On the flip-side, you might be presenting information on a topic that the viewer is ambivalent about, is actively bored by or is wilfully trying to ignore. In that case, you can take special care to compose engaging visualisations to capture and hold the attention of your audience.

9.4.2 Preattentive Attributes

When crafting a visualisation we want to require as little work as possible from the viewer. We can do this by using pre-attentive attributes such as colour, shape size and position to encode our data values.

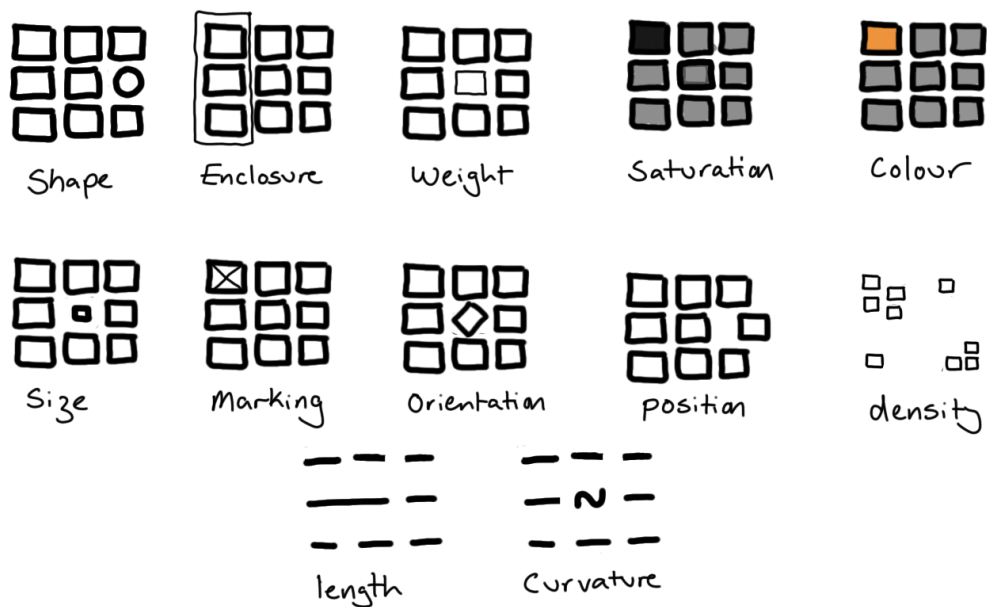


Figure 9.5: Examples of preattentive attributes

These preattentive attributes are properties of lines and shapes that provide immediate visual contrast without requiring active thought from the viewer. As we will see, care needs to be taken here to ensure that we don't mislead the viewer with how we use these attributes.

9.4.3 Example: First Impressions Count

This figure presents a bar chart of the mean height of males in several countries, but has swapped out the bars for human outlines. While the visualisation has an attractive, minimal

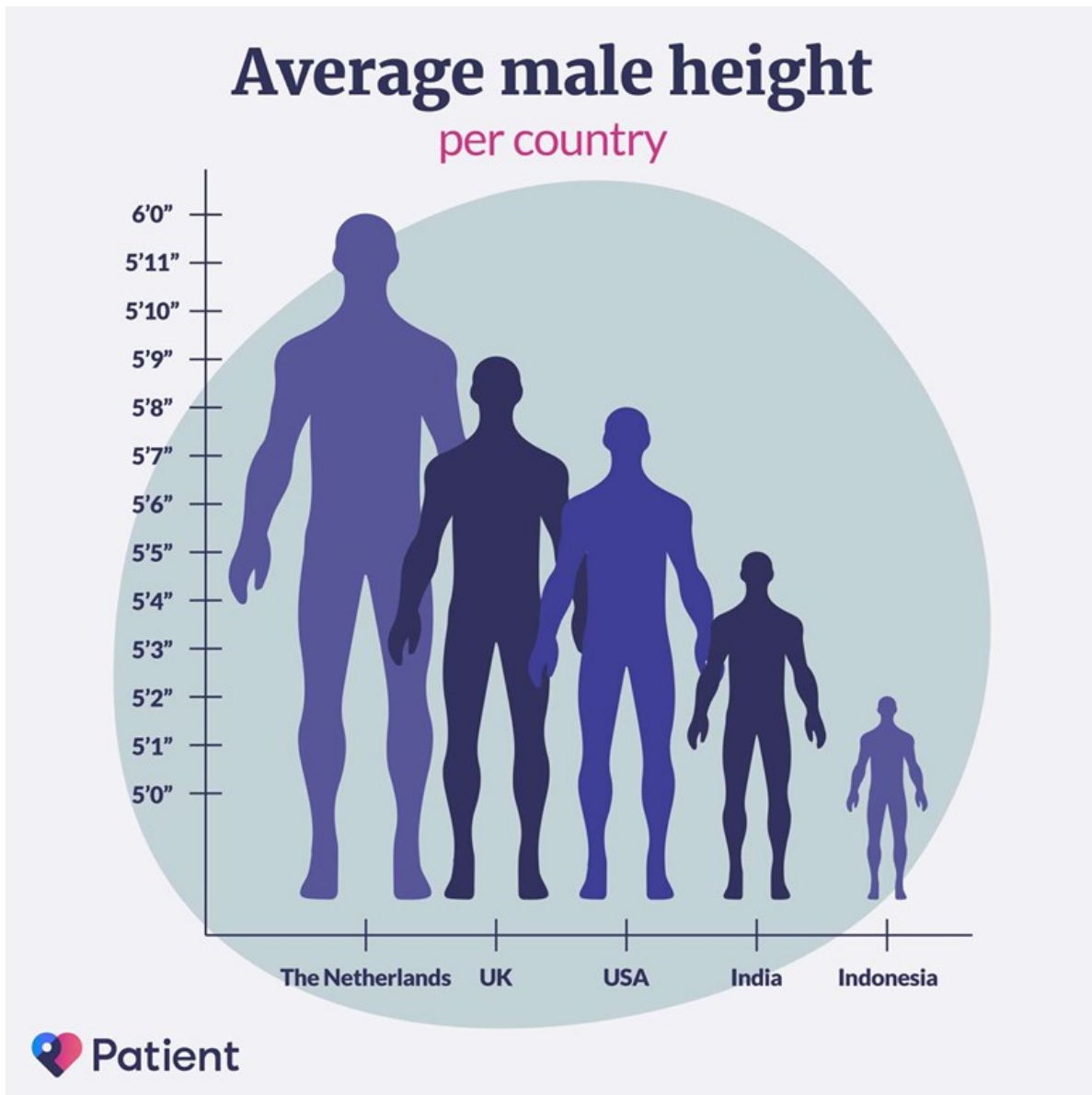


Figure 9.6: A questionably designed barchart of average male heights in the Netherlands, United Kingdom, USA Inida and Indonesia.

design and a pleasant colour scheme, it doesn't do a good job of immediately conveying the relevant information to the viewer.

The three main issues with this plot arise from swapping the bars of this plot for male silhouettes and are caused by the difference in how humans perceive lengths and areas. Typically, we make immediate pre-attentive comparisons based on area but can draw more accurate, considered comparisons when comparing lengths.

By replacing bars with human outlines this plot breaks the proportionality of length and area that is inherent in a bar plot. This causes dissonance between immediate and considered interpretation of this plot. An additional issue is that the silhouettes overlap, creating a forced perspective that makes it seem like some the outlines are further back and therefore even larger if this forced perspective is taken into account.

These three issues are important to consider when constructing your own visualisations. Are you showing all the values that the data could take, or focusing on a smaller interval to provide better contrast? If you are using the size of a circle to represent a value, are you changing the diameter or area in proportion to the data value? Finally, if you are making a plot that appears three-dimensional, have you done so on purpose? If so, could one of those dimensions be better represented by an attribute other than position?

9.4.4 Visual Perception

When reducing the dimensionality of your plot you may wish to represent a data value using colour rather than position. When deciding on how to use colour, you should keep your audience in mind.

Is your aim to two or more categories? In that case, you'll need to select your finite set of colours and ensure that these can be distinguished.

Are you representing a data value that is continuous or has an associated ordering? Then you will again have to select your palette to provide sufficient contrast to all viewers of your work.

If you are representing a measurement that has a *reference value* (for example 0 for temperature in centigrade) then a diverging colour palette can be used to represent data that are above or below this reference point. This requires some cultural understanding of how the colours will be interpreted, for example you are likely to cause confusion if you an encoding of red for cold and blue for hot.

For colour scales without such a reference point then a gradient in a single colour is likely the best option. In either case, it is important to check that a unit change in data value represents a consistent change in colour across all values. This is not the case for the rainbow palette here (which is neither a single gradient nor a diverging colour scale).



Figure 9.7: Some default colour scales in R

To improve accessibility of your designs, I would recommend one of the many on-line tools to simulate colour vision deficiency or using a pre-made palette where this has been considered for you. A good, low-tech rule of thumb is to design your visualisations so that they're still easily understood when printed in greyscale. This can mean picking appropriate colours or additionally varying the point shape, line width or line types used.

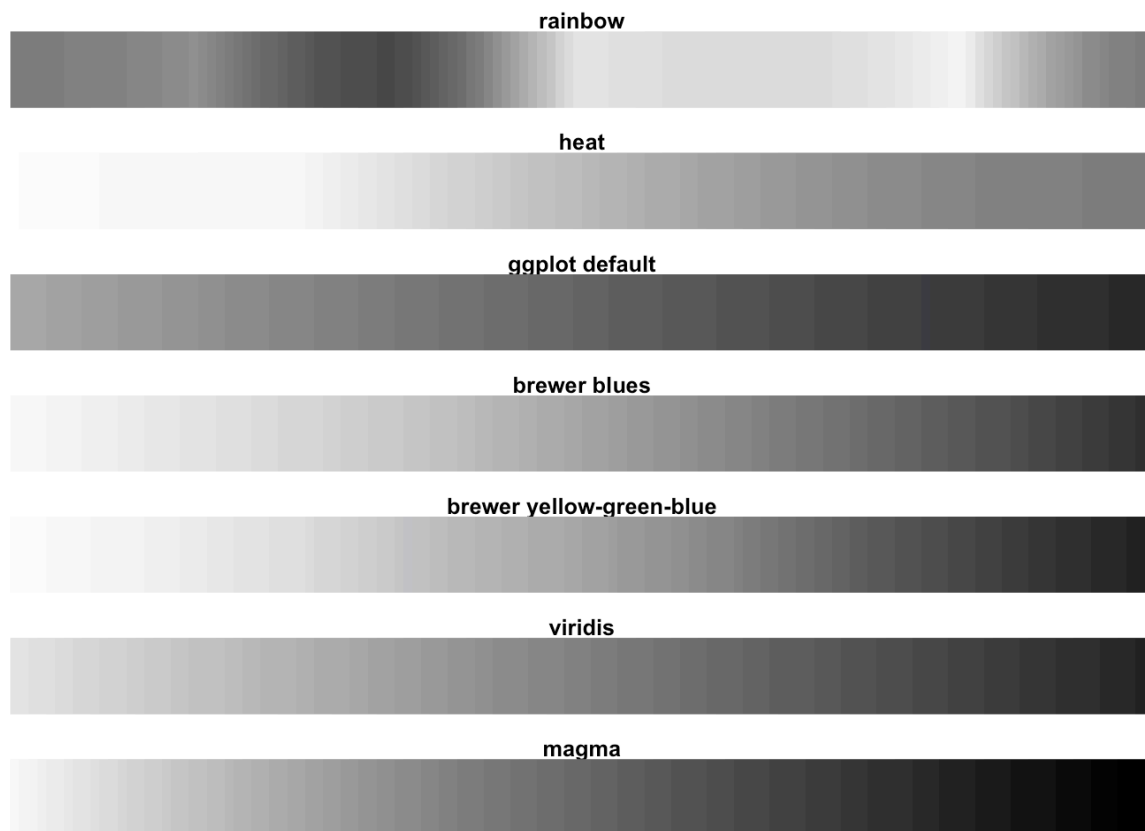


Figure 9.8: Desaturated colour scales in R

For a practical guide on setting colours see this [chapter](#) of exploratory data analysis by Roger Peng.

9.4.5 Alt-text, Titles and Captions

Captions describe a figure or table so that it may be identified in a list of figures and to add additional detail (where appropriate).

Alternative text describes the content of an image for a person who cannot view it. ([Guide to writing alt-text](#))

Titles give additional context or identify key findings. Active titles are preferable.

When visualisations are included in a report, article or website, they are often accompanied by three pieces of text. The title, the caption and the alt-text all help the audience to understand a visualisation but each serves a distinct purpose.

9.4.5.1 Captions

A caption is short description of a visualisation. Captions usually displayed directly above or below the figure or table that they describe. These captions serve two purposes: in a report, the caption can be used to look up the visualisation from a list of figures or tables. The second purpose of a caption is to add additional detail that you don't want to add to the plot directly. For example caption might be "Time series of GDP in the United States of America, 2017-2022. Lines show daily (solid), monthly (dashed) and five-year (dotted) mean values."

9.4.5.2 Alt-text

Alt text or alternative text is used to describe the content of an image to a person who can't view it. This text is helpful for people with a visual impairment, particularly those who use a screen reader. Screen reading software reads digital text out loud but can't interpret images. Such software replaces the image with the provided alternative text. Alt text is also valuable in cases when the image can't be found or loaded, for example because of an incorrect file path or a slow internet connection, because it will be displayed in place of the image.

The purpose of alt-text is different from a caption. It's designed as a replacement for the image, not just a shorthand or to provide additional information. If there is an important trend or conclusion to be drawn from the visualisation (that is not already mentioned in the main text) this should be identified in the alt-text. This sort of interpretation is a key aspect of alt-text that shouldn't be included in a caption.

9.4.5.3 Titles

Titles give additional context that is not conveyed by the axis labels or chart annotations. Alternatively the title can be used like a newspaper headline to deliver the key findings of the visualisation. One example of this might be when looking at a visualisation that is composed of many smaller plots, each showing the GDP for a US state over the last five years. Title for each smaller plot would identify the state it is describing, while the overall title might be something like "All US states have increased GDP in the period 2017-2022".

If you are including this type of interpretive title, make sure that the same interpretation is clear in the alt-text.

9.5 Your Story

The fourth aspect of a successful data visualisation is that it must tell a story. This story doesn't need to be a multi-generational novel or even a captivating novella. If a picture speaks a thousand words, really what you're aiming for is the visual equivalent of an engaging anecdote.

Your visualisation should be something that grabs viewers' attention and through its composition or content alters their knowledge or view of the world in some way.

Telling effective stories requires planning. How you construct your narrative depends on what effect you want to have on your audience. I'd encourage you to think like a data journalist and go about your work with its intended effect clear in your mind. Is your purpose to inform them of a fact, to persuade them to use a different methodology or entertain them by presenting dull or commonplace data in a fresh and engaging way?

In reality your goal will be some mixture of these, at an interior point of this triangle. Clearly identifying this point will help you to present your visual story in a way that works towards your aims, rather than against them.

On the point of presentation, it is important to realise that there is no neutral way to present information. In creating a visualisation you're choosing which aspects of the data to emphasise, what gets summarised and what is not presented at all. This is how you construct a plot that tells a clear and coherent story. However, there is more than one story that you could tell from a single dataset.

As an example of this, let's consider a time-series showing the price of two stocks and in particular the choice of scale on the y-axis. Suppose the two stocks have values fluctuating around \$100 per share. Choosing a scale that goes from \$90 to \$120 would emphasise the differences between the two stocks. Setting the lower limit to \$0 would instead emphasise that these variations are small relative to the overall value of the stocks. Both are valid approaches but tell different stories. Be clear and be open about which of these you are telling and why you have chosen that over the alternative.

```
# Load required libraries
library(ggplot2)
library(dplyr)
library(magrittr)
library(cowplot)

# Set simulation parameters
n_out <- 365
initial_value <- 100
```

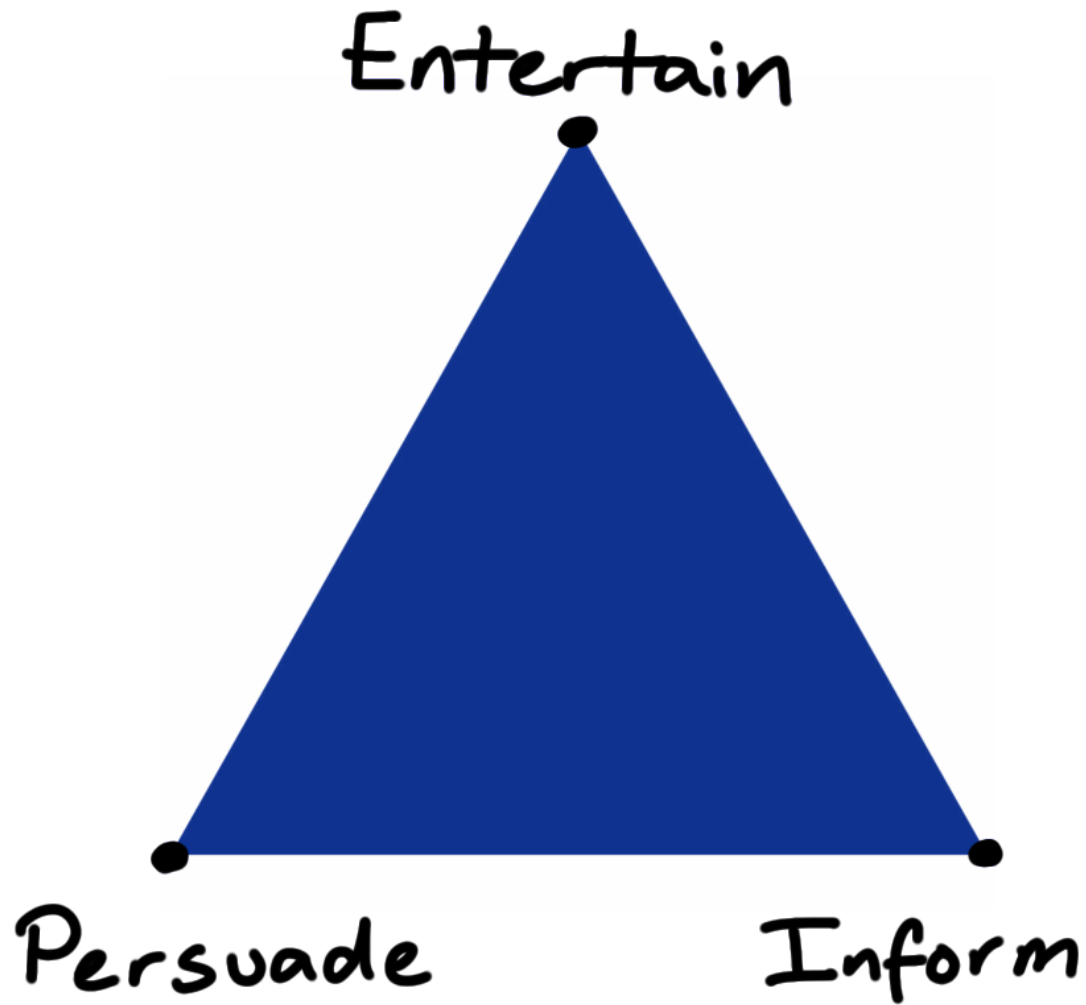


Figure 9.9: Illustration of a blue triangle with the words entertain, persuade and inform written at each corner.

```

# Simulate data
set.seed(007)

price <- rep(NA_real_, n_out)
price[1] <- initial_value

for (i in 2:n_out) {
  a <- rnorm(n = 1, mean = 1, sd = 0.01)
  b <- rnorm(n = 1, mean = 0, sd = 0.05)
  price[i] <- a * price[i-1] + b
}

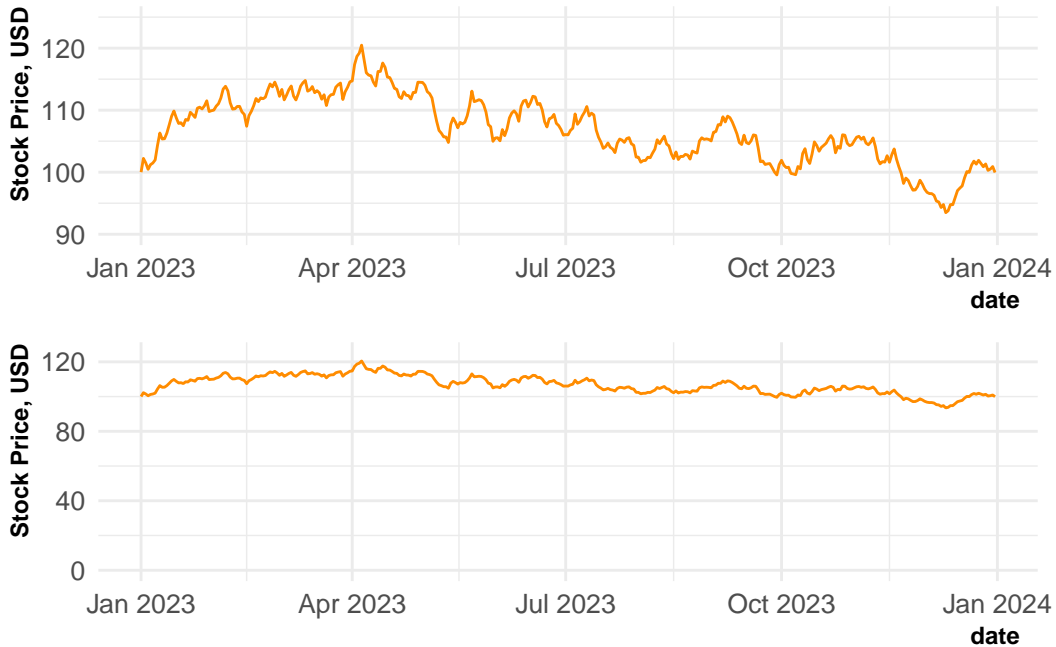
stock <- tibble(
  date = seq(from = as.Date("2023-01-01"), length.out = n_out, by = "day"),
  price = price
)

# Construct plot
p <- stock %>%
  ggplot(aes(x = date, y = price)) +
  geom_line(color = "darkorange") +
  ylab("Stock Price, USD") +
  theme_minimal() +
  theme(
    axis.text = element_text(size = 10),
    axis.title = element_text(size = 9, face = "bold", hjust = 0.98, vjust = 2),
    plot.title = element_text(size = 15, hjust = 0.02),
    plot.subtitle = element_text(size = 12, hjust = 0.02)
  )

p1 <- p + ylim(90, 125)
p2 <- p + ylim(0, 125)

plot_grid(p1, p2, nrow = 2, ncol = 1)

```



A final cross-over from data journalism is that your visualisations will be competing for your viewers attention. You have to compete against everything else that is going on in their lives. Establish a clear “hook” within your visualisation to attract your viewer’s attention and immediately deliver the core message. This might be done with a contrasting trend-line or an intriguing title. Lead their attention first to the key message and then the supporting evidence.

9.6 Your Guidelines

9.6.1 Standardise and Document

The final consideration when creating visualisations is to reduce the number of considerations that you have to make in the future. This is done by thinking carefully about each of the decisions that you make and writing guidelines so that you make these choices consistently.

The choices that go into making an effective data visualisation are important and deserve careful consideration. However, this consideration comes at a cost. To the employer this is the literal, financial cost of paying for your time. More broadly this is the opportunity cost of all the other things that you could have been doing instead.

To be efficient in our visualisation design, we should extend our DRY coding principles the design processes. Make choices carefully and document your decisions to externalise the cognitive work required of you in the future.

Many companies aware of these financial and opportunity costs and provide style guides for visualisations in a similar manner to a coding or writing style guide. This not only externalises and formalises many decisions, but it also leads to a more uniform style across visualisations and the data scientists producing them. This leads to a unified, house-style for graphic design and a visual brand that is easily identifiable. This is beneficial for large companies or personal projects alike.

9.6.2 Example Style Guides

I'd highly recommend exploring some visualisation guides to get an idea of how these are constructed and how you might develop your own.

Unsurprisingly some of the best guides come from media outlets and government agencies. These businesses are used to writing style guides for text to create and maintain a distinctive style across all of their writers.

- [BBC](#)
 - [Infographics Guidelines](#)
 - [R Cookbook](#)
 - [{bbplot}](#)
- [The Economist](#)
- [The Office for National Statistics](#)
- [Eurostat](#)
- [Urban Institute](#)
- [The Pudding](#) (learning resources)

The level of detail and technicality varies wildly between these examples. For instance, the BBC do not provide strong guidelines on the details of the final visualisation but do provide a lot of technical tools and advice on how to construct those in a consistent way across the corporation. They've even gone so far as to write their own theme for ggplot and to publish this as an R package!

9.7 Wrapping Up

Think about your *tools*.

Think about your *medium*.

Think about your *audience*.

Think about your *story*.

Think about your *guidelines*.

Data visualisation might seem like a soft skill in comparison to data acquisition, wrangling or modelling. However, it is often effective visualisations that have the greatest real world impact.

It is regularly the highly effective figures within reports and presentations that determine which projects are funded or renewed. Similarly, visualisations in press releases can determine whether the result of your study are trusted, correctly interpreted, and remembered by the wider public.

When constructing visualisations it is important to consider whether there are existing guidelines that provide helpful constraints to your work. From there, determine the story that you wish to tell and exactly who it is that you are telling that story to. Once this is decided you can select the medium and the tools that you use to craft your visualisation so that you have the greatest chance of achieving your intended effect.

Checklist

Videos / Chapters

- [Data Wrangling](#) (20 min) [\[slides\]](#)
- [Data Exploration](#) (25 min) [\[slides\]](#)
- [Data Visualisation](#) (27 min) [\[slides\]](#)

Reading

Use the [Data Exploration and Visualisation](#) section of the reading list to support and guide your exploration of this week's topics. Note that these texts are divided into core reading, reference materials and materials of interest.

Activities

Core:

- [NormConf](#) is a conference dedicated to the unglamorous but essential aspects of working in the data sciences. The conference talks from December 2022 are available as a [Youtube Playlist](#). Find a talk that interests you and watch it, then post a short summary to EdStem, describing what you learned from the talk and one thing that you still do not understand.
- Work through this ggplot2 tutorial for [beautiful plotting in R](#) by Cédric Scherer, recreating the examples for yourself.
- Using your `rolling_mean()` function as inspiration, write a `rolling_sd()` function that calculates the rolling standard deviation of a numeric vector.
 - Extend your `rolling_sd()` function to optionally return approximate point-wise confidence bands for your rolling standard deviations. These should be ± 2 standard errors by default and may be computed using analytical or re-sampling methods.

- Create a visualisation using your extended `rolling_sd()` function to assess whether the variability in the daily change in Dow Jones Index is changing over time. [\[data\]](#)

Bonus:

- Add your `rolling_sd()` function to your R package, adding documentation and tests.
- During an exploratory analysis, we often need to assess the validity of an assumed distribution based on a sample of data. Write your own versions of `qqnorm()` and `qqplot()`, which add point-wise tolerance intervals to assess whether deviation from the line $y = x$ are larger than expected.
- Add your own versions of `qqnorm()` and `qqplot()` to your R package, along with documentation and tests.

Live Session

In the live session we will begin with a discussion of this week's tasks. We will then break into small groups for two data visualisation exercises.

(Note: For one of these exercises, it would be helpful to bring a small selection of coloured pens or pencils, if you have access to some. If not, please don't worry - inventive use of black, blue and shading are perfectly acceptable alternatives!)

Please come to the live session prepared to discuss the following points:

- Which NormConf video did you watch and what did you learn from it?
- Other than `{ggplot2}`, what else have you used to create data visualisations? What are their relative strengths and weaknesses?
- How did you implement your `rolling_sd()` function and what conclusions did you draw when applying it to the Dow Jones data?

Part IV

Preparing for Production

Part V

Introduction

! Important

Effective Data Science is still a work-in-progress. This chapter is currently a dumping ground for ideas, and we don't recommend reading it.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

10 Reproducibility

i Note

Effective Data Science is still a work-in-progress. This chapter is largely complete and just needs final proof reading.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

10.1 The Data Scientific Method

In what we have covered so far we have been very much focused on the first aspect of data science: the data. When we come to consider about whether our work can be reproduced or our results can be replicated, this shifts our focus to the second other, the science.

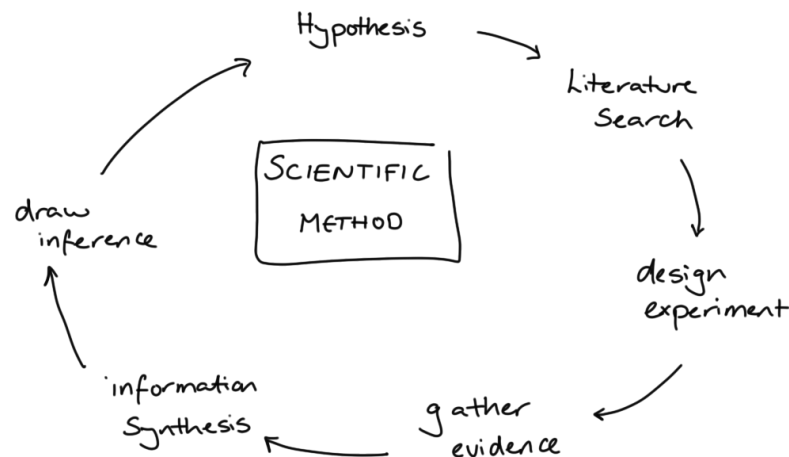


Figure 10.1: Cycle of scientific enquiry.

As data scientists, we like to think that we are applying the scientific method in our work.

We start with a question we want to answer or a problem we want to solve. This is followed by a search of the existing literature: is this a well-known problem that lots of people have solved before? If it is, fantastic, we can learn from their efforts. If not, then we proceed to gather our own evidence and combine this with whatever existing knowledge we could scrape together. Finally, we draw conclusions from this synthesised information.

When doing so we acknowledge that the conclusions we reach are not the truth, just our current best approximation of it. We have a usefully simplified model of the messy reality that we can share with the world. We will happily update our model as we become aware of new evidence, whether that new information supports or contradicts our current way of thinking.

That sounds excellent and is, in an ideal world, how both science and data science would progress. However, just like our models this is a simplified (and in this case idealised) description of what really happens.

10.2 Issue: Multiple, Dependent Tests

- Projects are usually not a single hypothesis test
- Sequence of dependent decisions
- e.g. Model development
- Can fool ourselves by looking lots of times or ignoring sequential and dependent structure.

The aims of a data science project are rarely framed as a clear unambiguous hypothesis, for which we will design a study and perform a single statistical analysis. Apart from in special cases, like A/B testing, we have a much more general aim for our data science projects.

The garden of forking paths: Why multiple comparisons can be a problem, even when there is no “fishing expedition” or “p-hacking” and the research hypothesis was posited ahead of time*

Andrew Gelman[†] and Eric Loken[‡]

14 Nov 2013

“I thought of a labyrinth of labyrinths, of one sinuous spreading labyrinth that would encompass the past and the future . . . I felt myself to be, for an unknown period of time, an abstract perceiver of the world.” — Borges (1941)

Abstract

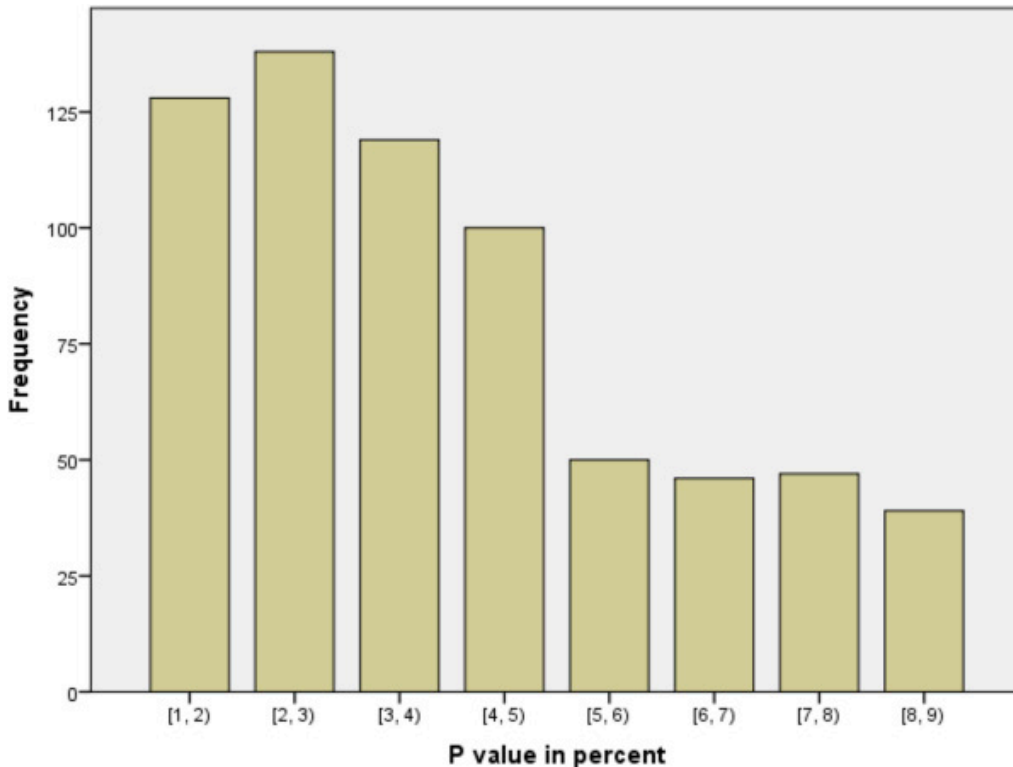
Researcher degrees of freedom can lead to a multiple comparisons problem, even in settings where researchers perform only a single analysis on their data. The problem is there can be a large number of *potential* comparisons when the details of data analysis are highly contingent on data, without the researcher having to perform any conscious procedure of fishing or examining multiple p-values. We discuss in the context of several examples of published papers where data-analysis decisions were theoretically-motivated based on previous literature, but where the details of data selection and analysis were not pre-specified and, as a result, were contingent on data.

{Abstract

text of Gelman and Loken (2013).}

We might want to construct a model for a given phenomenon and we’ll try many variants of that model along the way. By taking a more relaxed approach to data analysis, data scientists can run the risk of finding spurious relationships within our data that don’t hold more generally. If you look for a relationship between the quantity you are trying to predict and enough samples of random noise will almost surely find a significant relationship, even though there is no true link.

10.3 Issues: p -hacking and Publication Bias



{Bar chart of p -values in medical publications, showing a large drop between 4-5 percent and 5-6 percent.}

Okay, so our methods of investigation as data scientists might not be completely sound, but that should be balanced by the results of other studies that exist, right? Well, the second worrisome aspect of this process is that we can't always trust the published literature to be a fair representation of what people have tried in the past.

Studies that don't provide strong evidence against the null hypothesis rarely make it into publications, reports or the news. This is largely because of the way that scientific enquiry is rewarded in the the academy, business and the media. Funding and attention tend to go to studies with new findings, rather than those which aim to confirm or strengthen the findings of existing studies.

This systemic problem incentives scientists to 'massage' numbers to obtain a p -value less than 0.05 so that a result can be reported as statistically significant. This process is known as p -hacking and can occur through deliberate malpractice but more often it's a result of scientists generally not receiving adequate training in statistics. As statistically trained data scientists we know that a declaration of significance is no indication of a meaningful effect size and that the conventional significance level of 5% is entirely arbitrary. However, we need to be aware

that this is not the case across science and that even we aren't immune to the societal and systematic influences that favour the publication of novel results over confirmatory ones.

These influences also lead to a much more subtle problem than direct p -hacking. Consider a model with an obvious but unnecessary additional property to add: a example here might be adding an unnecessary term to a regression. Because this extension is such low hanging fruit, many scientists independently design experiments to test it out. Most of these experiments provide insufficient evidence against the null hypothesis and don't get developed into published papers or technical reports, because they just support the status-quo. However, after enough people have attempted this some scientist will get "lucky" and find a significant (and potentially relevant) benefit. Of all the experiments that were done, this is the only one that makes it onto the public record.

All of these studies being left in the proverbial desk drawer induces publication bias in the scientific literature. When we come to assess the state of existing knowledge, we are unable to properly assess the importance of findings, because we lack the context of all those null results that went unreported.

This same process means that the results of many more scientific studies than we would expect cannot be recreated. This is what is known as the scientific replication crisis.

10.4 Reproducibility

Reproducibility: given the original raw data and code, can you get all of the results again?

- Reproducible \neq Correct
- "Code available on request" is the new "Data available on request"
- Reproducible data analysis requires effort, time and skill.

This idea of reproducibility requires us to be able to recover the exact same numerical summaries as the original investigator. In particular this means we should be able to reproduce the exact same point estimates and measures of uncertainty that they did, which ensures we'll draw the same conclusions as that original investigator.

When putting our work into production there are several reasons why we might require it to be reproducible. The first is logistical: production code needs to be robust and efficient - this often means your code will be re-factored, rewritten or translated into another language. If your results are not reproducible then there is no way to verify that this has been done correctly. Secondly, if a problem is identified in your work (say a customer raises a complaint that their loan application was incorrectly rejected) you need to be able to accurately recreate that instance to diagnose if there is a problem and exactly what caused it.

Note that just because findings are reproducible, that doesn't by any means imply that they're correct. We could have a very well documented but flawed analysis that is entirely reproducible but is also completely unsuitable or just plain wrong.

In our data science projects, we have already taken several steps that greatly improve the reproducibility of our work. Although we scripted, tested and documented our work to improve the management of our project, these decisions improve the scientific quality of our work too. This puts us in a strong position relative to the scientific literature as a whole.

At a point now where it is almost standard to publish data along with papers, but for a long time this was not the case and data if data were available at all, this was only by request. We are now in a similar situation when it comes to code. It's still far from standard for the analysis code to be required and put up to detailed scrutiny as part of the peer-review process.

With a little more context this isn't so unreasonable. Across many scientific disciplines, code-based approaches to analysis is not standard; statistical software with a graphical user interface is used instead. The idea here is to allow scientists to analyse their own data by providing tools through a combination of menus and buttons. However, these interfaces often leave no record of how the data were manipulated and the software itself can be highly specialised or proprietary. This combination means that even when full datasets are provided, it is often impossible for others to reproduce the original analysis.

None of this is meant to scold or disparage scientists who use this type of software to allow them to perform statistical analyses. You're well aware of how much time and effort it takes to learn how to use and implement statistical methods correctly. This is time that other scientists invest in learning their subject, so that they can get to the point of doing research in the first place. This is one of the wonders of data science: the ability to work in multi-disciplinary teams where individual members are specialised in different areas.

This is where we need to pause and check ourselves, because the same fate can easily befall us as data scientists. Yes, it takes time to learn the skills and practices to ensure reproducibility, but it also takes time to implement them and the time of an expert data practitioner doesn't come cheap. If you wait until the end of a project before you make it reproducible then you'll usually be too late - time or money will have run out.

10.5 Replicability

Replicable: if the experiment were repeated by an independent investigator, you would get slightly different data but would the substantive conclusions be the same?

- In the specific sense, this is the core worry for a statistician!
- Also used more generally: are results stable to perturbations in population / study design / modelling / analysis?

- Only real test is to try it. Control risk with shadow and parallel deployment. Statisticians are well aware that if we were to repeat an experiment we would get slightly different data. This would lead to slightly different estimates and slightly different results.

Ultimately, this is the core problem that statisticians get paid to worry about: will those changes be small enough that the substantive conclusions are not impacted? Yes, point estimates will vary slightly but do your conclusions about the existence, direction or magnitude of an effect still hold? Alternatively, if you are estimating a relationship between two variables, is the same functional form chosen as the most suitable?

In a general scientific context, replication takes a more broad meaning and asks whether the key properties of your results could be replicated by another person. In the context of getting your work put into production, we are concerned about whether your the results of your findings will also hold when applied to future instances that might differ from those you have seen already.

If you come to data science from a statistical background then you are well accustomed to these sorts of considerations. Whenever you perform a hypothesis test or compare two models, you take steps to make sure the comparison is not only valid for this particular sample, but that is also true out-of-sample. This is the whole reason data scientists make training and test sets in the first place, as an approximate test for this sort of generalisation. We do any of these things we are asking of ourselves: will this good performance replicate if we had different inputs?

Of course train-test split, bootstrap resampling or asymptotic arguments can only ever approximate the ways in which our sample differs from the production population, to which our models will be applied. The only way to truly assess the out-of-sample performance of our models or generalisability of our findings is to put that work into production.

This opens us up to risk: what if our findings don't generalise and our new model is actually much worse than the current one? It's not possible to both deploy our new model and also avoid this risk entirely. However, we can take some steps to mitigate our level of exposure.

10.5.1 Shadow deployment

In the most risk-adverse setting we might implement a shadow deployment of our new model. In this case, the current model is still used for all decision making but our candidate model is also run in the background so that we can see how it might behave in the wild. This is good in that we can identify any points of catastrophic failure for the new model, but is also expensive to run and can give us only limited information.

Suppose, for example, our model is a recommender system on a retail website. A shadow deployment will let us check that the new system functions correctly and we can gather data on what products are recommended to each customer and investigate how these differ from those recommended by the current system. A shadow deployment cannot in this case tell us

what the customer would have done had they been shown those products instead. This means that a shadow deployment doesn't allow us to investigate whether the new system leads to equivalent or greater revenue than the current system.

10.5.2 Parallel deployment

Parallel deployment or A/B tests have both the current and the proposed new models running at the same time. This allows us to truly test whether our findings generalise to the production population while controlling our level of risk exposure by setting the proportion of times each model is used. The more instances we assign to the new model the faster we will learn about its performance but this also increases our risk exposure.

10.6 Reproduction and Replication in Statistical Data Science

10.6.1 Monte Carlo Methods

In data science we rely a lot on the use of stochastic methods. These are often used to increase the chance of our findings being replicated by another person or in production. However, they also make it more difficult to ensure that our exact results can be reproduced, whether by another person or our future selves.

Monte Carlo methods are any modelling, estimation or approximation technique that leverages randomness in some way.

We have seen examples of this to improve the probability of successful replication. The most obvious example of this is the random allocation of data in a Train/Test split for model selection.

Another example focused on improving replication is the use of bootstrap resampling to approximate the sampling distribution of a test statistic. This might be a parametric bootstrap, where alternative datasets are generated by sampling values from the fitted model of our observed data. Alternatively, a non-parametric bootstrap would generate these alternative datasets by sampling from the original data with replacement.

Monte Carlo methods can also be used to express uncertainties more generally, or to approximate difficult integrals. These are both common applications of Monte Carlo methods in Bayesian modelling, where (unless our models are particularly simple) the posterior or posterior-predictive distribution has to be approximated by a collection of values sampled from that distribution.

Each time we run any of these analyses we'll get slightly different outcomes. For our work to be replicable we need to quantify this level of variation. For example, if we had a different sample from the posterior distribution, how much would the estimated posterior mean change

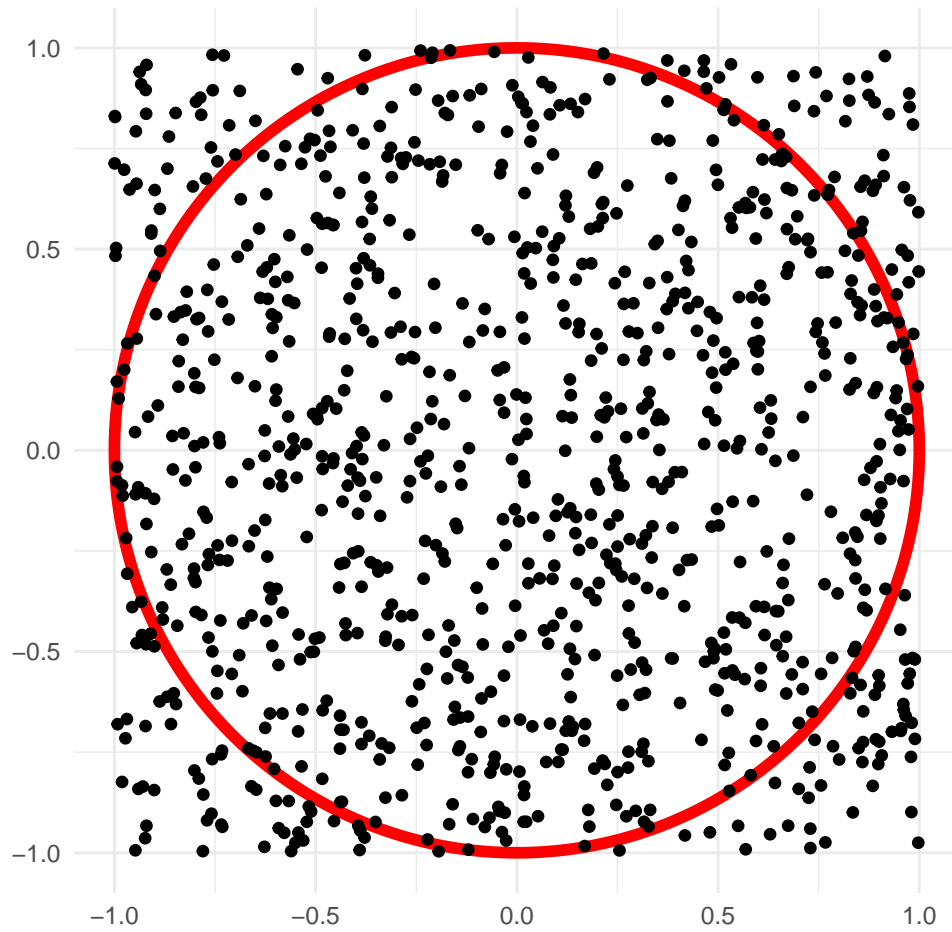


Figure 10.2: Monte Carlo approximation of π

by? Sometimes, as in this case, we can appeal to the law of large numbers to help us out. If we take more samples, the variation between realisations will shrink. We can then collect enough samples so our estimated mean is stable across realisations, up to our desired number of significant figures.

To make our results reproducible we will have to ensure that we can reproduce the remaining, unstable digits for any particular realisation. We can do this by setting the seed of our random number generator, an idea we will return to shortly.

10.6.2 Optimisation

Optimisation is the second aspect of data science that can be very difficult to ensure is reproducible and replicable.

Is the optimum you find stable over:

- runs of the procedure?
- starting points?
- step size / learning rate?
- realisations of the data?

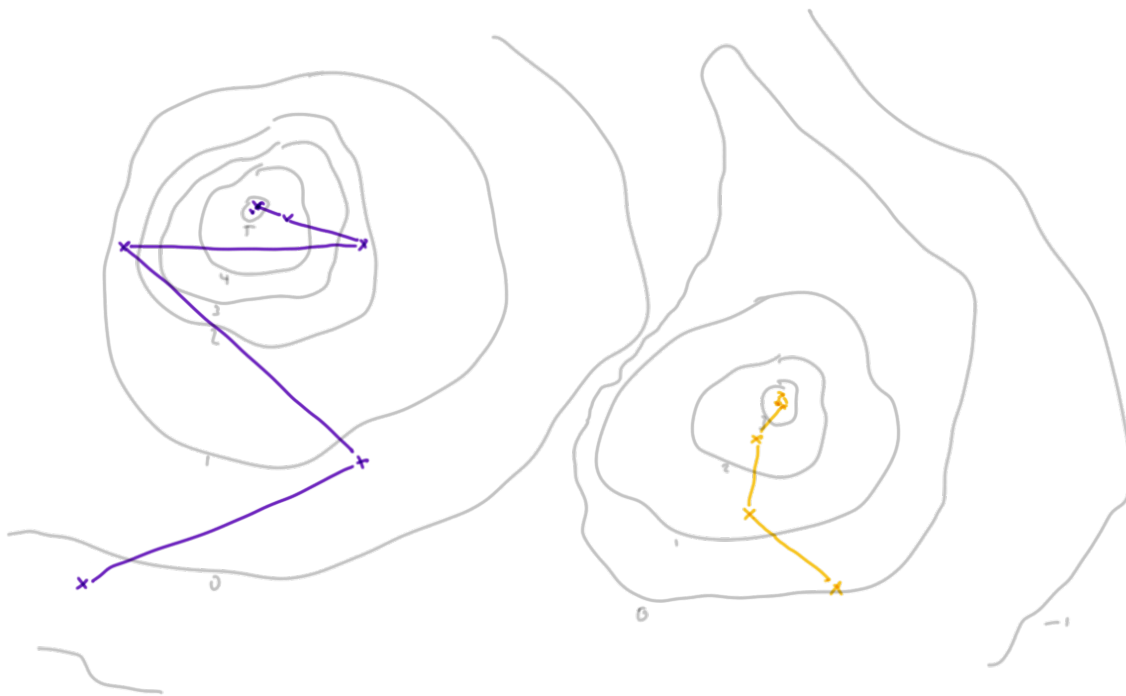


Figure 10.3: A poorly drawn contour plot. Local modes make this optimisation unstable to the choice of starting point.

If optimisation routines are used in parameter estimation, we have to ensure that the results they find for a particular data set are reproducible for a given configuration. This might be a given data set, initial set of starting parameters and step size, the learning rate (which controls how that step size changes) and the maximum number of iterations to perform.

We additionally need to be concerned about replication here. If we had chosen a different starting point would the optimisation still converge, and would it converge to the same mode? Our original method may have found a local optimum but how can we confirm that this is a global optimum?

If the optimisation fails to converge, can you reproduce that case to diagnose the problem? This can be particularly tricky when the optimisation routine itself uses Monte Carlo methods, such as stochastic gradient descent or simulated annealing.

10.6.3 (Pseudo-)Random Numbers

Sometimes we have stochastic elements to our work that we can't use brute force to eliminate. Perhaps this is beyond our computational abilities or else the number of realisations is an important, fixed aspect of our study.

Fortunately, in computing it's not common to have truly random numbers. Instead, what we usually have is a complex but deterministic function that generates a sequence of numbers that are statistically indistinguishable from a series of independent random variates.

The next term in this sequence of pseudo-random numbers is generated based on value the current one. This means that, by setting the starting point, we can always get the same sequence of pseudo-random variates each time we run the code. In R we set this starting point using `set.seed()`

This is especially useful for simulations that involve random variables, as it allows us to recreate the same results exactly. This not only makes it possible for others to recreate our results but it can also make it much easier to test and debug our own code.

```
# different values
rnorm(n = 4)
#> [1] -1.2053334  0.3014667 -1.5391452  0.6353707
rnorm(n = 4)
#> [1]  0.7029518 -1.9058829  0.9389214 -0.2244921
```

```
# the same value
set.seed(1234)
rnorm(n = 4)
#> [1] -1.2070657  0.2774292  1.0844412 -2.3456977

set.seed(1234)
rnorm(n = 4)
#> [1] -1.2070657  0.2774292  1.0844412 -2.3456977
```

10.7 Beware

When running code sequentially and interactively, setting the seed is about all you will need to solve reproducibility problems. However, I'd advise you to take great care when combining this with some of the methods we'll see for speeding up your code. In some cases, the strategies to optimize your code performance can interfere with the generation of random numbers and lead to unintended results.

When writing code that's executed in parallel across multiple cores or processors, you have to carefully consider whether or not to give each the same seed value. The correct decision here is context specific and depends on the interpretation of the random variates you will be generating. If you are making a comparison between iterations it might be important the the random aspects are kept as similar as possible, while if you are paralleling only for speed gains this might not be important at all.

Finally, it's important to be wary of the quality of pseudo-random number generation and the interfacing R with other programming languages. R was developed as a *statistical* programming language and most other languages are not as statistically focused. Different languages may use different algorithms for generating pseudo-random numbers, and the quality of the generated numbers can vary. It's important to make sure that seeds are appropriately passed between languages to ensure that the correct sequence of random numbers is generated.

10.8 Wrapping Up

To get our work put into production it should be both reproducible and replicable.

- *Reproducible*: can recreate the same results from the same code and data
- *Replicable*: core results remain valid when using different data

While randomness is a key part of most data science workflows but can lead to reproducibility nightmares. We can manage these by appealing to the stability of averages in large samples and by explicitly setting the sequence of pseudo-random numbers that we generate using `set.seed()`.

Finally, we should take special care when we have to combine efficiency with replicable workflows.

With these ideas you can now use good data to do good science.

11 Explainability

i Note

Effective Data Science is still a work-in-progress. This chapter is largely complete and just needs final proof reading.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

11.1 What are we explaining and to whom?

There are many reasons you might need to explain the behaviour of your model before it can be put into production. As an example, we can consider a credit scoring system that determines whether or not customers should be given a line of credit.

- Regulatory or legal requirements to describe how your model works (e.g. ban on “black-box” modelling).
- Understanding how your model works to improve it.
- Explaining to individual loan decisions to customers.

In each of these cases, what exactly do we mean by an explanation? It’s likely not the same thing in each example.

- Data scientists we might be interested to know exactly what types of mapping between covariates and responses can be represented by the neural network architecture underlying the credit scoring system.
- Stakeholders within the company or regulators are likely indifferent to this and are more concerned about understanding the general behaviour of the model across large numbers of loan applications.
- Finally, individual customers might have some investment in the overall behaviour of the scoring model but would also like to know what actions they can take to increase their chance of securing a loan.

Between each of these examples, the level of technical detail differs but more importantly the fundamental nature of the explanations are different.

11.2 Explaining a Decision Tree

With some models giving an explanation is relatively straightforward. Decision trees are perhaps the easiest model to explain because they mimic human decision making and can be represented like flow-charts that make sequential, linear partitions of the predictor space.

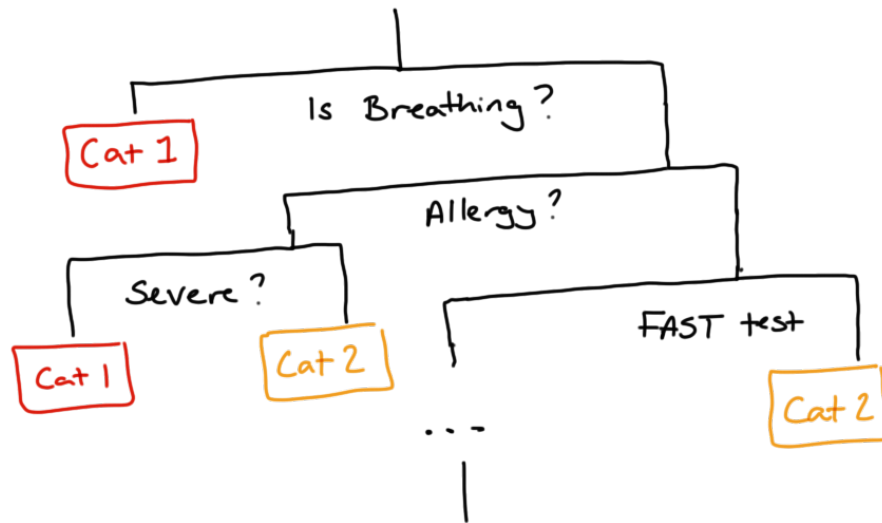


Figure 11.1: An example of a decision tree, optimised to correctly identify category 1 ambulance calls in as few questions as possible.

These models use the same sort of logic that is used for medical triage when you call an ambulance, to determine the urgency of the call. The binary decisions used in this type of triage are optimised to identify critical calls as soon as possible, but this is just one form of loss function we could use. We might instead pick these partitions to get the most accurate overall classification of calls to urgency categories. This might not be an appropriate loss function for ambulance calls but might be when deciding which loan applicants to grant credit to.

The issue is that these decision trees are limited in the relationships they can represent (linear relationships approximated by step function) and are sensitive to small changes in the training data. To overcome these deficiencies we can use a bootstrap aggregation or a random forest model to make predictions based on a collection of these trees. This leads to models that are more stable and flexible but also removes any chance of a simple and human-friendly explanation.

11.3 Explaining Regression Models

Another model that is relatively straightforward to interpret is a linear regression. We can interpret this model using the estimated regression coefficients, which describe how the predicted outcome changes with a unit change in each covariate while the values of all other covariates are held constant.

This is a global and a conditional explanation.

It is **global** because the effect of increasing a covariate by one unit is the same no matter what the starting value of that covariate. The explanation is the same in all parts of the covariate space.

The explanation is **conditional** because it assumes that all other values are held constant. This can lead to some odd behaviour in our explanations, they are dependent on what other terms are included (or left out of) our model.

This can be contrasted against non-linear regression, where covariate effects are still interpreted conditional on the value of other covariates but the size or direction of that effect might vary depending on the value of the covariate.

Here we have an example where a unit increase in the covariate is associated with a large change in the model response at low values of the covariate, but a much smaller change at large values of the covariate.

11.4 Example: Cherrywood regression

As an example of this we can look at the height, girth and volume of some cherry trees.

If we are wanting to use a lathe to produce pretty, cherry wood ornaments we might be interested in understanding how the girth of the trees varies with their height and total volume. Using a linear model, we see that both have a positive linear association with girth.

```
lm(Girth ~ 1 + Height, data = trees)
#>
```

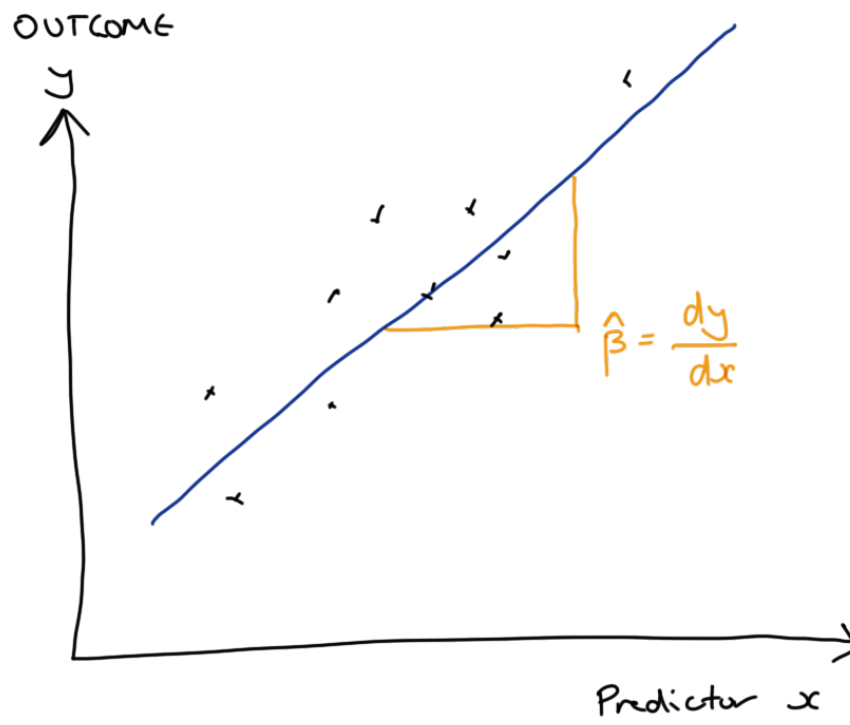


Figure 11.2: Linear models have global, conditional explanations, provided by the estimated regression coefficients.

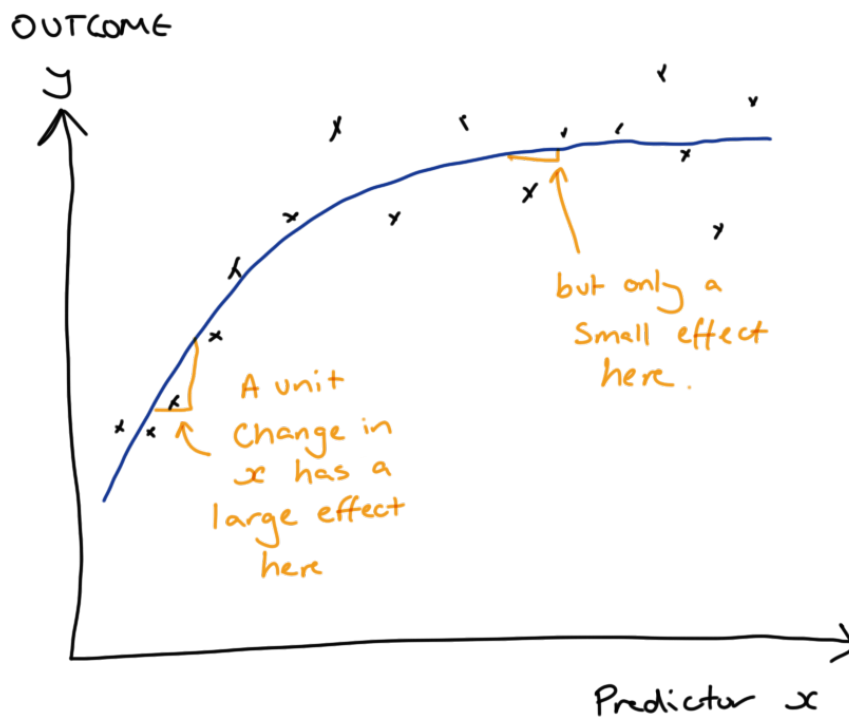


Figure 11.3: Non-linear models have local, conditional explanations, provided by the estimated regression coefficients.

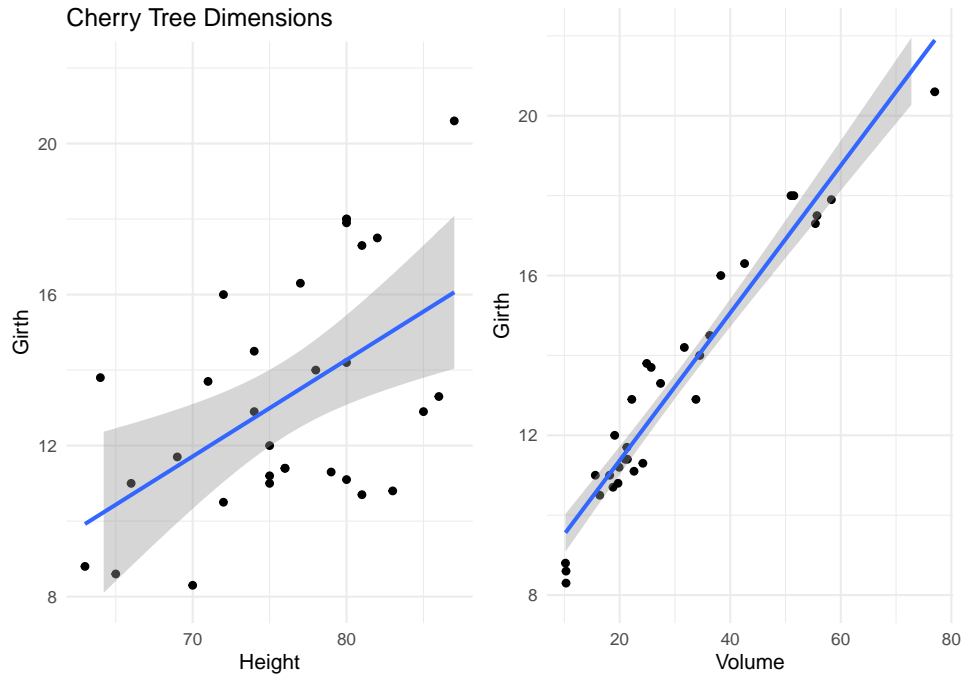


Figure 11.4: Cherry tree girth can be well modelled as a linear function of either tree height or harvestable volume.

```
#> Call:
#> lm(formula = Girth ~ 1 + Height, data = trees)
#>
#> Coefficients:
#> (Intercept)      Height
#>    -6.1884      0.2557
```

```
lm(Girth ~ 1 + Volume, data = trees)
#>
#> Call:
#> lm(formula = Girth ~ 1 + Volume, data = trees)
#>
#> Coefficients:
#> (Intercept)      Volume
#>     7.6779     0.1846
```

However, when we include both terms in our model, our interpretation changes dramatically.

```
lm(Girth ~ 1 + Height + Volume, data = trees)
#>
#> Call:
#> lm(formula = Girth ~ 1 + Height + Volume, data = trees)
#>
#> Coefficients:
#> (Intercept)      Height      Volume
#>  10.81637      -0.04548      0.19518
```

Height is no longer positively associated with girth. This is because the size, direction and significance of our estimated effects is conditional on what other terms are included in the model. For a *fixed volume* of wood, a taller tree necessarily has to have a smaller girth.

Techniques such as [SHAP](#) try to quantify the importance of a predictor by averaging over all combinations of predictors that might be included within the model. You can read more about such techniques in [Interpretable Machine Learning](#) by Christoph Molnar.

11.5 Simpson's Paradox

This effect is related to **Simpson's Paradox**, where a trend appears in several groups of data but disappears or reverses when the groups are combined.

This regularly arises in fields like epidemiology, where population level trends are assumed to apply at the level of individuals or small groups, where this is known as the ecological fallacy.

Actually, Simpson's paradox is a terrible name, because it isn't actually a paradox at all. It's not surprising that we have two different answers to two different questions, the supposed contradiction only arises when we fail to distinguish between those questions.

What I hope to have highlighted here is that for some of the simplest models we might use as data scientists, explanations are very much possible but must be made with care and attention to detail - correctly interpreting these models in context can be far from straightforward.

11.6 What hope do we have?

At this stage, you might be asking yourself what hope we have of explaining more complex models like random forests or neural networks, given how difficult it is to explain even the simple models we might take as our benchmark. You'd be right to worry about this and it is important to remain humble in what we can and cannot know about these complex systems that we are building.

All hope isn't lost though - we still have a few tricks up our sleeve!

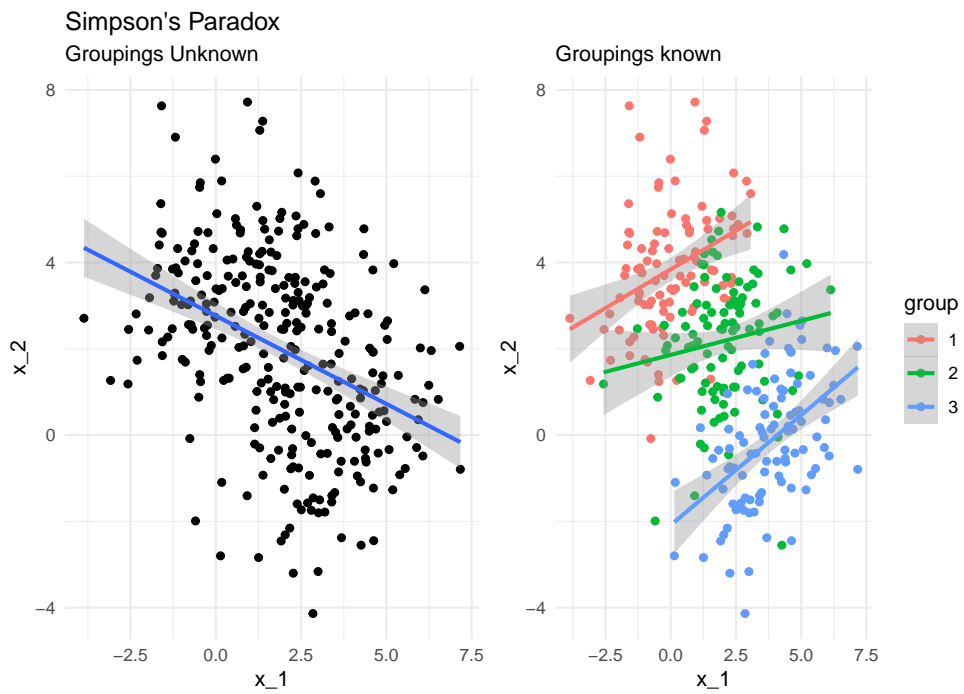


Figure 11.5: A simulated example in which a covariate has a negative trend at the population level but a positive trend within each sub-group.

11.6.1 Permutation Testing

Suppose we're asked by our product manager to determine which predictors or covariates are usually the most important in our model when determining credit scores and loan outcomes.

One way to do this would be to remove each covariate from the model and investigate how that changes the predictions made by our model. However, we've seen already that removing a predictor can substantively change some models.

So instead, we could answer this question by using permutation methods.

If we take the observed values of a covariate, say income, and randomly allocate these among all our training examples then this will destroy any association between income and loan outcome. This allows us to remove the information provided by income but without altering the overall structure of our model. We can then refit our model to this modified data set and investigate how rearranging the covariate alters our predictions. If there is a large performance drop, then the covariate is playing an important role within our model.

There are many variations on this sort of permutation test. They can be simple but powerful tools for understanding the behaviour of all sorts of model.

11.6.2 Meta-modelling

Meta-models are, as the name suggests, models of models and these can be effective methods of providing localised explanations for complex models.

The idea here is to look at a small region of the covariate space that is covered by a complex and highly flexible model, such as a neural network. We can't easily give a global explanation for this complex model's behaviour - it is just too complicated.

However, we can interrogate the model's behaviour within a small region and construct a simplified version of the model (a meta-model) that lets us explain the model within that small region. Often this meta-model is chosen as a linear model.

This is partly for convenience and familiarity but also has a theoretical backing: if our complex model is sufficiently smooth then we can appeal to [Taylor's Theorem](#) to say that in a small enough region the mapping can be well approximated by a linear function.

This sort of local model explanation is particularly useful where we want to explain individual predictions or decisions made by our model: for example why a single loan applicant was not granted a loan. By inspecting the coefficients of this local surrogate model we can identify which covariates were the most influential in the decision and suggest how the applicant could increase their chance of success by summarising those covariates that were both influential and are within the ability of the applicant to change. This is exactly the approach taken by the [LIME](#) methodology developed by Ribeiro et al.

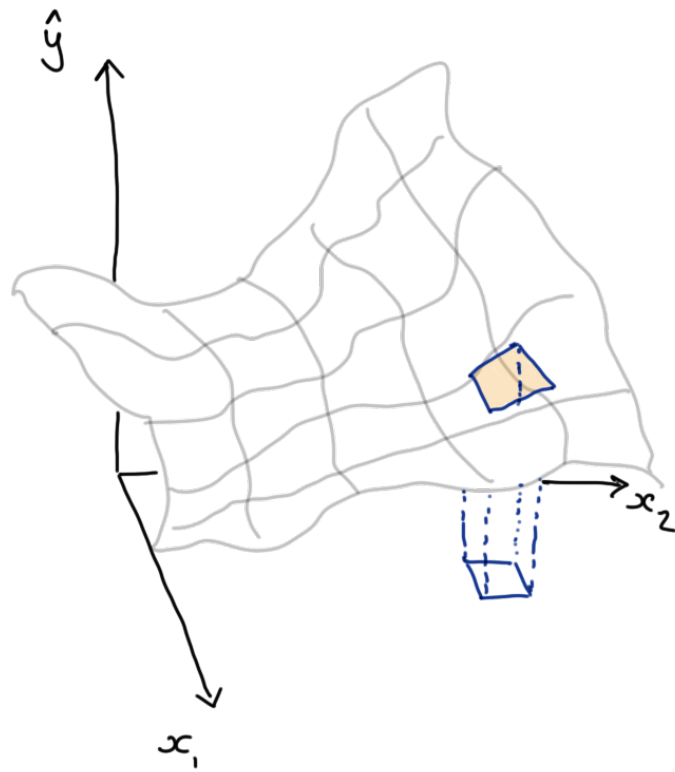


Figure 11.6: A local linear approximation in two dimensions

11.6.3 Aggregating Meta-models

Using local or conditional explanations of our model's behaviour can be useful in some circumstances but they don't give a broader understanding of what is going on overall. What if we want to know how a covariate influences *all* outcomes not a particular one? What if we care about the covariate's expected effect over all loan applicants, or the *distribution* of effects over all applicants?

This is where local and conditional explanations are particularly nice. By making these explanations at many points we can aggregate these explanations to understand the global and marginal behaviour of our models.

To aggregate our conditional effects into a marginal effect, or a local effect into a global effect we must integrate these over the joint distribution of all covariates. If this sounds a bit scary to you, then you are right. Integration is hard enough at the best of times without adding in the fact that we don't know the joint distribution of the covariates we are using as predictors.

Don't worry though, we can take the easy way out and do both of these things approximately. We can approximate the joint distribution of the covariates by the empirical distribution that we observe in our sample, and then our nasty integrals simplify to averages over the measurement units in our data (the loan applicants in our case).

If we construct a local, conditional model for each loan applicant in our data set, we can approximate the marginal effect of each covariate by averaging the conditional effects we obtain for each loan applicant.

This gives us a global understanding of how each covariate influences the response of our model. It does this over all possible values of the other covariates and appropriately weights these according to their frequency within the sample (and also within the population, if our sample is representative).

11.7 Wrapping Up

We've seen that as our models become more flexible they also become more difficult to explain, whether that is to ourselves, and subject expert or a user of our model.

That's not to say that simple models are always easy to explain or interpret, simple models can be deceptively tricky to communicate accurately.

Finally, we looked at a couple of techniques for explaining more complex models.

We can use permutation tests measure feature importance in our models: shuffling the predictor values breaks any relationship to our response, and we can measure how much this degrades our model performance. A big dip implies that feature was doing a lot of explaining.

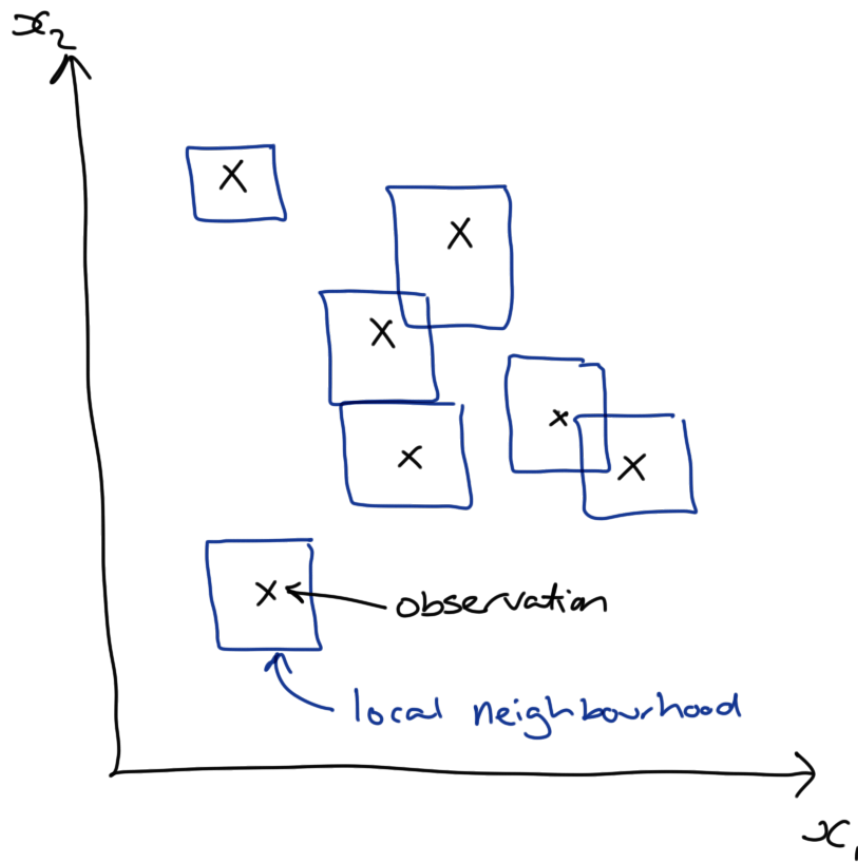


Figure 11.7: Local approximations around each observation can be combined to understand global model behaviour.

We can also look at the local behaviour of models by making surrogate or meta models, that are interpretable, and aggregate these to understand the model globally.

Effective explanations are essential if you want your model to be used in production and to feed into real decisions decision making. This requires some level of skill with rhetoric to tailor your explanation so that it is clear to the person who requested it. But this isn't a soft skill, it also requires a surprising amount of computational and mathematical skill to extract such explanations from complex modern models.

12 Scalability

! Important

Effective Data Science is still a work-in-progress. This chapter is currently a dumping ground for ideas, and we don't recommend reading it.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

12.1 Scalability and Production

When put into production code gets used more and on more data. We will likely have to consider scalability of our methods in terms of

- Computation time
- Memory requirements

When doing so we have to balance a trade-off between development costs and usage costs.

12.1.1 Example: Bayesian Inference

- MCMC originally takes ~24 hours
- Identifying and amending bottlenecks in code reduced this to ~24 minutes.

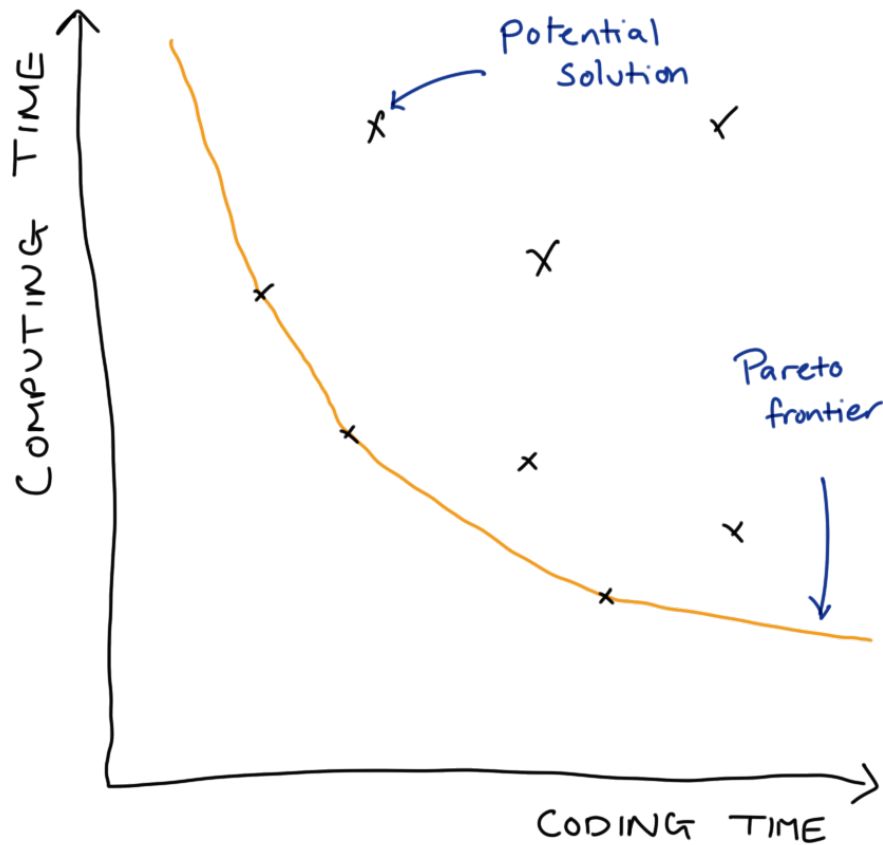
Is this actually better? That will depend on a number of factors, including:

- human hours invested
- frequency of use
- safe / stable / general / readable
- trade for scalability

12.1.2 Knowing when to worry

Sub-optimal optimisation can be worse than doing nothing

... programmers have spent far too much time worrying about efficiency in *the wrong places* and at *the wrong times*; premature optimisation is the root of all evil (or at least most of it) in programming. - Donald Knuth



12.1.3 Our Focus

Writing code that scales well in terms of computation time or memory used is a huge topic. In this section we restrict our aims to:

- Basic profiling to find bottlenecks.
- Strategies for writing scalable (R) code.

- Signpost advanced methods & further reading.

12.2 Basics of Code Profiling

12.2.1 R as a stopwatch

The simplest way to profile your code is to time how long it takes to run. There are three common ways to do this.

Firstly, you could record the time before your code starts executing, the time it completes and look at the difference of those.

```
t_start <- Sys.time()
Sys.sleep(0.5) # YOUR CODE
t_end <- Sys.time()

t_end - t_start
#> Time difference of 0.5057299 secs
```

The `system.time` function provides a shorthand for this if your code runs sequentially and extends the functionality to work for parallel code too.

```
system.time(
  Sys.sleep(0.5)
)
#>   user  system elapsed
#> 0.000  0.000  0.506
```

The `{tictoc}` package has similar features, but also allows you to add intermediate timers to more understand which parts of your code are taking the most time to run.

```
library(tictoc)

tic()
Sys.sleep(0.5) # YOUR CODE
toc()
#> 0.505 sec elapsed
```

With `{tictoc}` we can get fancy

```

tic("total")
tic("first, easy part")
Sys.sleep(0.5)
toc(log = TRUE)
#> first, easy part: 0.509 sec elapsed
tic("second, hard part")
Sys.sleep(3)
toc(log = TRUE)
#> second, hard part: 3.01 sec elapsed
toc()
#> total: 3.527 sec elapsed

```

If your code is already very fast (but will be run *very* many times, so further efficiency gains are required) then the methods may fail because they do not sample the state of the code at a high enough frequency. In those cases you might want to explore the `{microbenchmark}` package.

12.3 Profiling Your Code

To diagnose scaling issues you have to understand what your code is doing.

- Stop the code at time τ and examine the *call-stack*.
 - The current function being evaluated, the function that called that, the function that called that, ..., top level function.
- Do this a lot and you can measure (estimate) the proportion of working memory (RAM) uses over time and the time spent evaluating each function.

```

library(profvis)
library(bench)

```

12.3.1 Profiling: Toy Example

Suppose we have the following code in a file called `prof-vis-example.R`.

```

h <- function() {
  profvis::pause(1)
}

```

```

g <- function() {
  profvis::pause(1)
  h()
}

f <- function() {
  profvis::pause(1)
  g()
  profvis::pause(1)
  h()
}

```

Then the call stack for `f()` would look something like this.

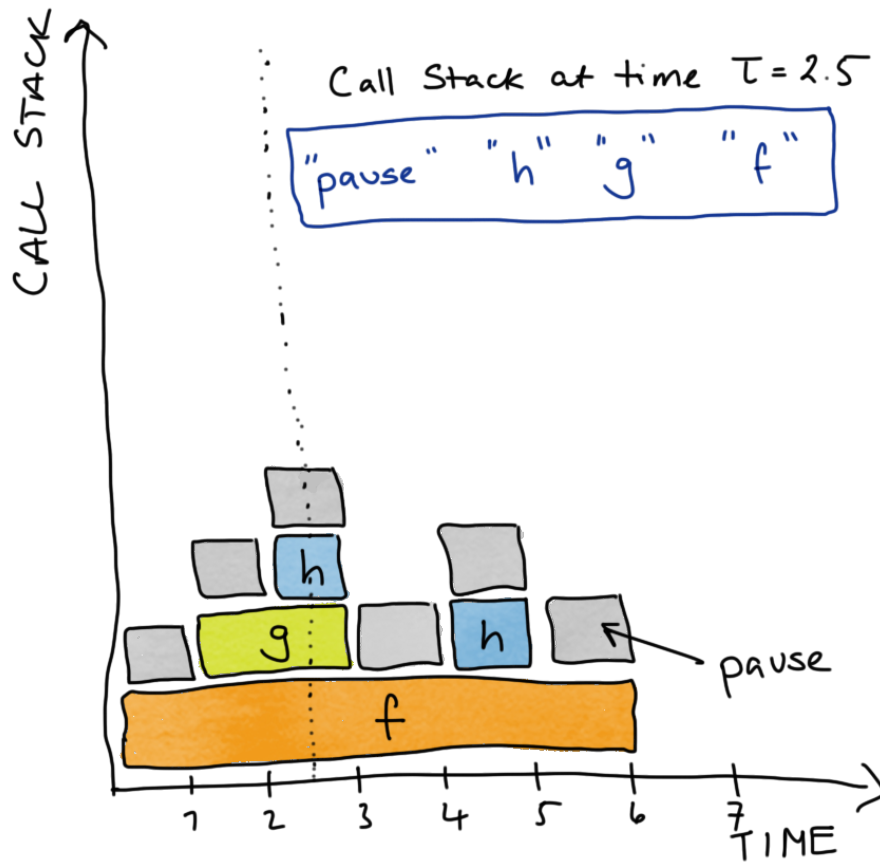


Figure 12.1: Callstack for `f()`

We can examine the true call stack using the `profvis()` function from the `{profvis}` package. By saving the code in a separate file and sourcing it into our session, this function will also give us line-by-line information about the time and memory demands of our code.

```
source("prof-vis-example.R")
profvis::profvis(f())
```

In both the upper histogram and the lower flame plot we can see that the majority of time is being spent in `pause()` and `h()`. What we have to be careful of here is that the upper plot shows the total amount of time in each function call, so `h()` appears to take longer than `g()`, but this is because it is called more often in the code snippet we are profiling.

12.4 Notes on Time Profiling

We will get slightly different results each time you run the function

- Changes to internal state of computer
- Usually not a big deal, mainly effects fastest parts of code
- Be careful with stochastic simulations
- Use `set.seed()` to make a fair comparison over many runs.

12.4.1 Source code and compiled functions

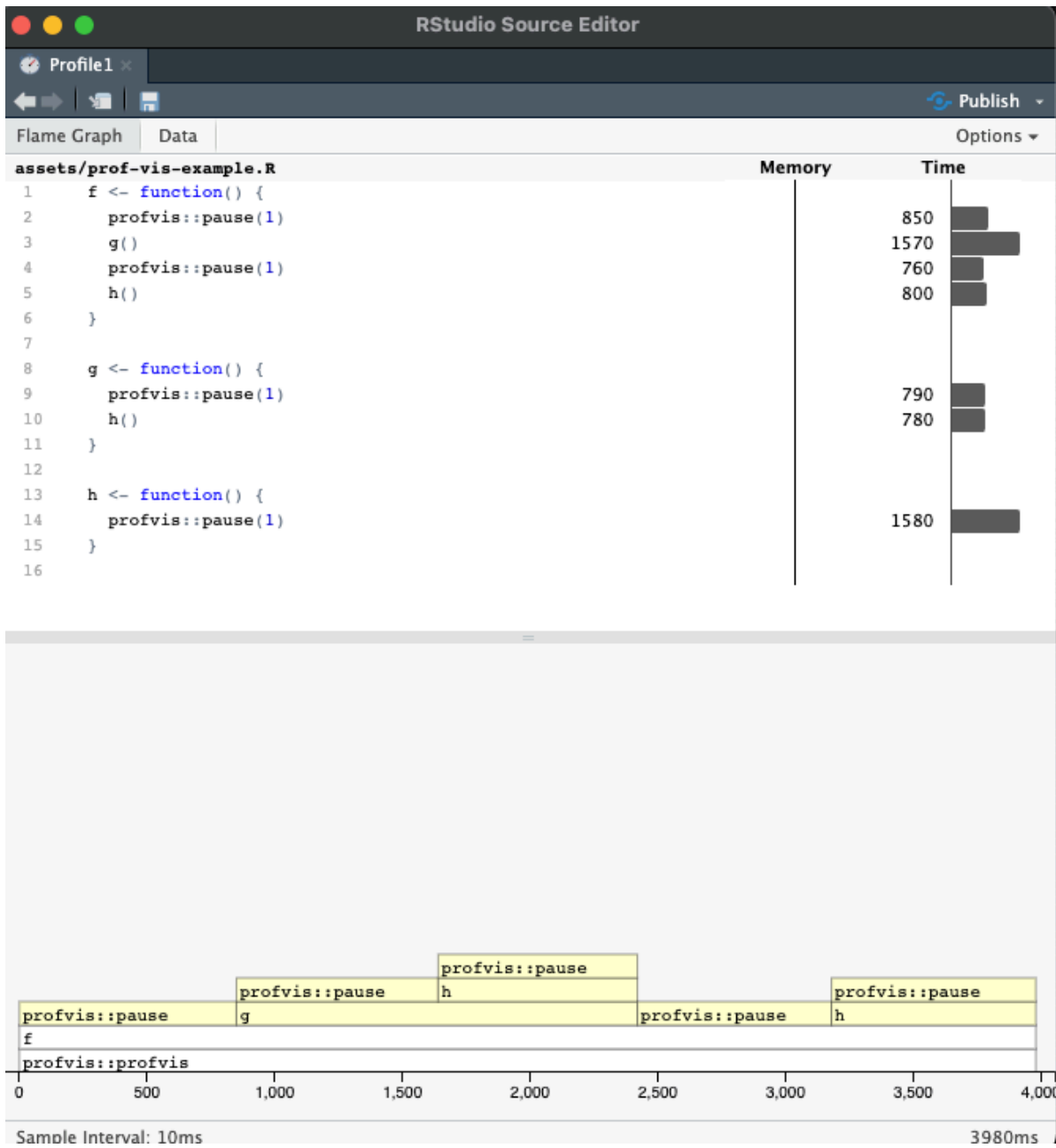
If you write a function you can see the source of that function by calling it's name

```
pad_with_NAs <- function(x, n_left, n_right){
  c(rep(NA, n_left), x, rep(NA, n_right))
}
```

```
pad_with_NAs
#> function(x, n_left, n_right){
#>   c(rep(NA, n_left), x, rep(NA, n_right))
#> }
```

This is equally true for functions within packages.

```
eds::pad_with_NAs
#> function (x, n_left, n_right)
#> {
#>   stopifnot(n_left >= 0)
```



```

#>   stopifnot(n_right >= 0)
#>   stopifnot(class(x) %in% c("character", "complex", "integer",
#>     "logical", "numeric", "factor"))
#>   c(rep(NA, n_left), x, rep(NA, n_right))
#> }
#> <bytecode: 0x7fc5682894d8>
#> <environment: namespace:eds>

```

Some functions use *compiled code* that is written in another language. This is the case for dplyr's `arrange()`, which calls some compiled C++ code.

```

dplyr::arrange
#> function (.data, ..., .by_group = FALSE)
#> {
#>   UseMethod("arrange")
#> }
#> <bytecode: 0x7fc56a25f208>
#> <environment: namespace:dplyr>

```

It is also true for many functions from base R, for which there is (for obvious reason) no R source code.

```

mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x7fc562b97a80>
#> <environment: namespace:base>

```

These compiled functions have no R source code, and the profiling methods we have used here don't extend into compiled code. See [{jointprof}](#) if you really need this profiling functionality.

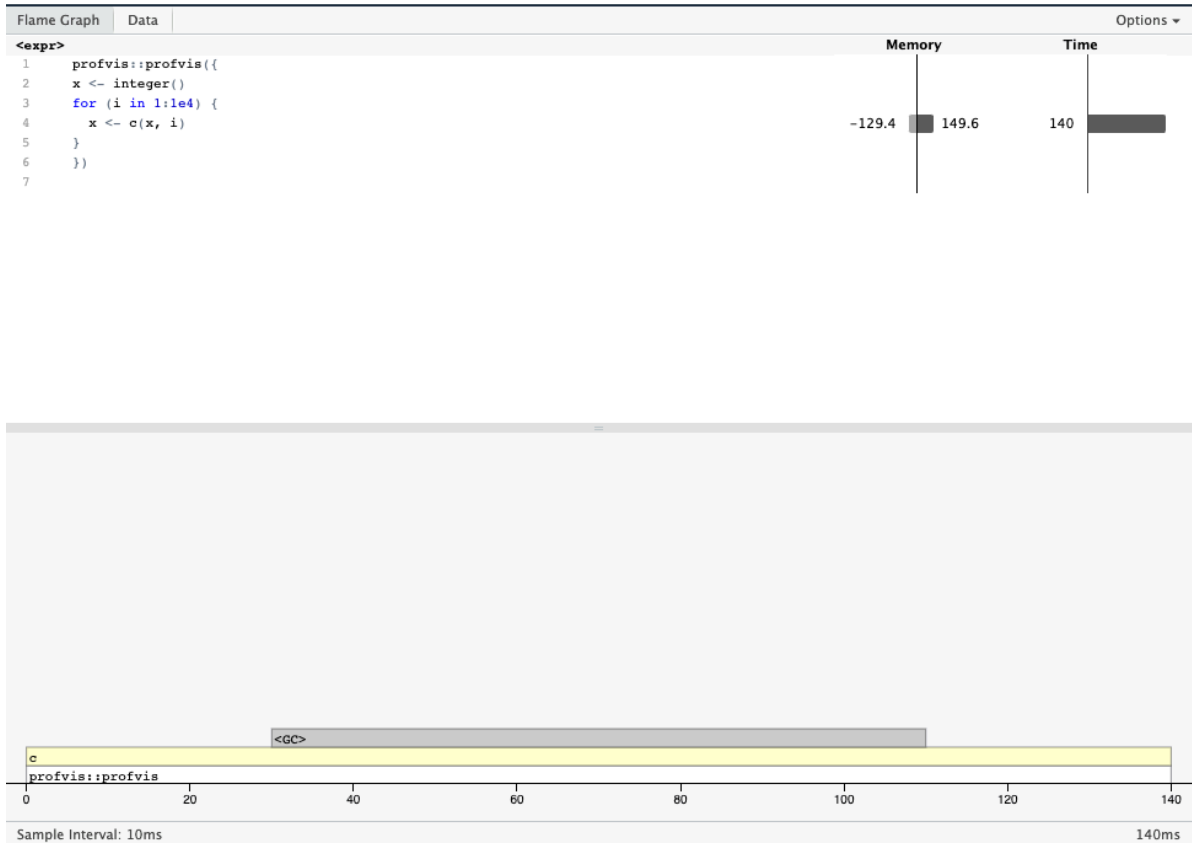
12.5 Memory Profiling

`profvis()` can similarly measure the memory usage of your code.

```

x <- integer()
for (i in 1:1e4) {
  x <- c(x, i)
}

```



- Copy-on-modify behaviour makes growing objects slow.
- Pre-allocate storage where possible.
- Strategies and structures, see [R inferno](#) and [Efficient R](#).

12.6 Tips to work at scale

TL;DR: pick your object types carefully, vectorise your code and as a last resort implement your code in a faster language.

12.6.1 Vectorise

Two bits of code do the same task, but the second is much faster, because it involves fewer function calls.

```
x <- 1:10
y <- 11:20
z <- rep(NA, length(x))

for (i in seq_along(x)) {
  z[i] <- x[i] * y[i]
}
```

```
x <- 1:10
y <- 11:20
z <- x * y
```

Where possible write and use functions to take advantage of vectorised inputs. E.g.

```
rnorm(n = 100, mean = 1:10, sd = rep(1, 10))
```

Be careful of recycling!

12.6.2 Linear Algebra

```
X <- diag(x = c(2, 0.5))
y <- matrix(data = c(1, 1), ncol = 1)

X %*% y
```



```
#>      [,1]
#> [1,]  2.0
#> [2,]  0.5
```

More on vectorising: [Noam Ross Blog Post](#)

12.7 For loops in disguise

12.7.1 The apply family

Functional programming equivalent of a for loop. [`apply()`, `mapply()`, `lapply()`, ...]

Apply a function to each element of a list-like object.

```
A <- matrix(data = 1:12, nrow = 3, ncol = 4)
A
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    4    7   10
#> [2,]    2    5    8   11
#> [3,]    3    6    9   12

# MARGIN = 1 => rows, MARGIN = 2 => columns
apply(X = A, MARGIN = 1, FUN = sum)
#> [1] 22 26 30
```

This generalises functions from `{matrixStats}`, where for some special operations we can do all to the necessary calculation in C++.

```
rowSums(A)
#> [1] 22 26 30
```

12.7.2 `{purrr}`

Iterate over a single object with `map()`.

```
mu <- c(-10, 0, 10)
purrr::map(.x = mu, .f = rnorm, n = 5)
#> [[1]]
#> [1] -10.591484 -9.060351 -9.492216 -10.642034 -10.405463
#>
```

```

#> [[2]]
#> [1] -1.3510401  1.4797410 -0.2580480  0.4378293  1.5759402
#>
#> [[3]]
#> [1] 11.344607  9.237907 10.439590 10.357743  8.956596

```

Iterate over multiple objects `map2()` and `pmap()`.

```

mu <- c(-10, 0, 10)
sigma <- c(0, 0.1, 0)
purrr::map2(.x = mu, .y = sigma, .f = rnorm, n = 5)
#> [[1]]
#> [1] -10 -10 -10 -10 -10
#>
#> [[2]]
#> [1] 0.03008142 -0.22460345 0.09546957 0.02397148 0.02343102
#>
#> [[3]]
#> [1] 10 10 10 10 10

```

```

mu <- c(-10, 0, 10)
sigma <- c(0, 0.1, 0)

purrr::pmap(
  .f = rnorm,
  n = 5,
  .l = list(
    mean = mu,
    sd = sigma))
#> [[1]]
#> [1] -10 -10 -10 -10 -10
#>
#> [[2]]
#> [1] 0.02217263 0.07850441 0.02629728 0.03945660 -0.08541733
#>
#> [[3]]
#> [1] 10 10 10 10 10

```

For more details and variants see Advanced R [chapters 9-11](#) on functional programming.

12.8 Easy parallelisation with furrr

- {parallel} and {futures} allow parallel coding over multiple cores.
- Powerful, but steep learning curve.
- {furrr} makes this very easy, just add `future_` to purrr verbs.

```
mu <- c(-10, 0, 10)
furrr::future_map(
  .x = mu,
  .f = rnorm,
  .options = furrr::furrr_options(seed = TRUE),
  n = 5)
#> [[1]]
#> [1] -8.716024 -10.148955 -8.125526 -9.211252 -9.474119
#>
#> [[2]]
#> [1] -0.6436627 -0.5135344 1.2984521 0.9901563 -0.8190061
#>
#> [[3]]
#> [1] 9.608681 7.975333 9.062700 8.481350 9.700701
```

This is, of course excessive for this small example!

One thing to be aware of is that we need to be very careful handling random number generation in relation to parallelisation. There are many options for how you might want to set this up, see [R-bloggers](#) for more details.

12.9 Sometimes R doesn't cut it



RCPP: An API for running C++ code in R. Useful when you need:

- loops to be run in order

- lots of function calls (e.g. deep recursion)
- optimised data structures

Rewriting R code in C++ and other low-level programming languages is beyond our scope, but good to know exists. Starting point: Advanced R [Chapter 25](#).

12.10 Wrapping up

Summary

1. Pick you battles wisely
2. Target your energy with profiling
3. Scale loops with vectors
4. Scale loops in parallel processing
5. Scale in another language

Help!

- Articles and blog links
- The R inferno ([Circles 2-4](#))
- Advanced R ([Chapters 23-25](#)),
- Efficient R ([Chapter 7](#)).

Checklist

! Important

Effective Data Science is still a work-in-progress. This chapter is currently a dumping ground for ideas, and we don't recommend reading it.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

Videos / Chapters

- [Reproducibility](#) (26 min) [[slides](#)]
- [Explainability](#) (16 min) [[slides](#)]
- [Scalability](#) (30 min) [[slides](#)]

Reading

Use the [Preparing for Production](#) section of the reading list to support and guide your exploration of this week's topics. Note that these texts are divided into core reading, reference materials and materials of interest.

Activities

This week has fewer activities, since you will be working on the first assessment.

Core

- Read the LIME paper, which we will discuss during the live session.
- Work through the [understanding LIME R tutorial](#)
- Use code profiling tools to assess the performance of your `rolling_mean()` and `rolling_sd()` functions. Identify any efficiencies that can be made.

Bonus:

- Write two functions to simulate a [homogeneous Poisson process](#) with intensity $\lambda > 0$ on the interval $(t_1, t_2) \subset \mathbb{R}$. The first should use the exponential distribution of inter-event times to simulate events in sequence. The second should use the Poisson distribution of the total event count to first simulate the number of events and then randomly allocate locations over the interval. Evaluate and compare the reproducibility and scalability of each implementation.

Live Session

In the live session we will begin with a discussion of this week's tasks. We will then break into small groups for a reading group style discussion of the LIME paper that was set as reading for this week.

Part VI

Data Science Ethics

Part VII
Introduction

When carrying out a data science project, ethical concerns can arise at many stages.

This week we focus on the two most common, privacy and fairness. We explore case studies where these ethical principles have been violated in the past and investigate how technical measures and professional guidelines can help to prevent such events from occurring again in the future.

13 Privacy

13.1 Privacy and Data Science

Data privacy is all about keeping people's personal information confidential and secure. It's the idea that people have control over their personal data, and that companies and organizations are responsible for protecting it. Personal data can include things like names, addresses, phone numbers, email addresses, medical information, and even online activity.

What is personal data?

- Name, national insurance number, passport number
- Contact details: address, phone number, email address
- Medical history
- Online activity, GPS data, finger print or face-ID,

Should not be collected, analysed or distributed without consent.

Under the concept of data privacy, individuals have the right to know what information is being collected about them, how it's being used, who it's being shared with, and to have the ability to control that information. Companies and organizations, on the other hand, have a responsibility to keep this information secure and to use it only for the purpose it was intended.

It's important to remember that with the growing use of machine learning and data science methods, personal information is being collected, stored, and shared more than ever before. This makes data privacy a critical issue to our work as data scientists. By ensuring that personal information is handled responsibly and with respect for people's privacy, we can build trust and confidence in the digital world. In this section of the course I hope to introduce you to some key ideas around data privacy and use some case studies to demonstrate that privacy is not an easy thing to ensure.

13.2 Privacy as a Human Right

13.2.1 Article 12 of the Universal Declaration of Human Rights

No one shall be subjected to arbitrary interference with his privacy, family, home or correspondence, nor to attacks upon his honour and reputation. Everyone has the right to the protection of the law against such interference or attacks. - [UN General Assembly](#), 1948.

The idea of a right to privacy is not a new one. Article 12 of the Universal Declaration of Human Rights, written in 1948, states that everyone has the right to privacy in their personal and family life, home and correspondence (this includes communication via the post, telephone, and email).

This means that everyone has the right to keep their personal information, private and protected from being disclosed to others without their consent. This right to privacy is essential for protecting an individual's autonomy, dignity, and freedom.

The Universal Declaration of Human Rights it is often considered as a benchmark for human rights and many countries have incorporated its principles into their own laws and regulations. This means that in many countries the right to privacy is legally protected and people have the right to take action if their privacy is violated.

This means we have to take particular care in our work as data scientists when handling any information personal information, whether that is at the individual level or in aggregate.

13.3 Data Privacy and the European Union

13.3.1 General Data Protection Regulation (2018)

‘Consent’ of the data subject means any freely given, specific, informed and unambiguous indication of the data subject's wishes by which he or she, by a statement or by a clear affirmative action, signifies agreement to the processing of personal data relating to him or her; - [GDPR Article 4](#)

A more recent set of regulations relating to the use of personal data is the General Data Protection Regulation (GDPR). This is a comprehensive data privacy regulation that went into effect on May 2018 within the European Union. The purpose of the GDPR is to give individuals more control over their personal information and how it's used, and to unify data protection laws across the EU.

The GDPR is an extensive legal document that lays down strict rules for how companies and organizations must handle the personal data of EU citizens, *regardless of where that data is stored or processed*.

Some key provisions of the GDPR include the requirement to obtain explicit, informed and active consent from individuals before the collection and processing of their personal data. This is precisely why banner notifications about cookies on websites became ubiquitous,

GDPR also establishes the right for individuals to request access to, or deletion of, their personal data. Furthermore, it states that in the event of a data breach (where unauthorised access or use of personal data occurs) the data holder must inform the affected individuals and relevant authorities within 72 hours.

13.4 Privacy: Key Terms

Measuring privacy within a dataset is a complex task that involves assessing the degree to which personal information is protected from unauthorized access, use, or disclosure. There are many ways of measuring and increasing the degree of privacy within a data set. To understand these, and the literature on data privacy, it helps to be familiar with a few key terms.

Pseudonimisation: processing data so that it does not relate to an identifiable person.

A data entry is pseudonymised when it has been processed in a way that it does not relate to an identifiable person. The key word here is identifiable. Replacing your name with your CID would be considered a form of pseudonymisation, because based on that information alone you cannot be identified.

Re-identification: relating a pseudonymised data entry to an identifiable person.

Re-identification is the act of relating a pseudonymised data entry to an identifiable person. Re-identification makes something about that person known that wasn't known beforehand, perhaps by using processing techniques or cross-referencing against external information. In the previous grading scenario, re-identification occurs after marking so that your grades can be returned to you.

Anonymisation: A pseudonimisation method that precludes re-identification.

Anonymisation is a word that in casual usage is often conflated with our previous definition of pseudonymisation. In the technical sense, anonymisation is any form of pseudonymisation that precludes the possibility of re-identification.

We don't want to anonymise test scores, or we would not be able to map them back to individual students. However, if grades were to be published online then we would want to ensure that this is only done in an anonymised format.

13.5 Measuring Privacy

13.5.1 Pseudo-identifiers and k -anonymity

Pseudo-identifiers: Attributes that can also be observed in public data. For example, someone's name, job title, zip code, or email.

For the set of quasi-identifiers A_1, \dots, A_p , a table is *k-anonymous* if each possible value assignment to these variables (a_1, \dots, a_n) is observed for either 0 or at least k individuals.

k -anonymity is a technique that ensures each combination of attributes within a table, is shared by at least k records within that dataset. This ensures that each individual's data can't be distinguished from at least $k - 1$ other individuals.

This provides some rudimentary level of privacy, where k corresponds to the size of smallest equivalence class of pseudo-identifiers within the data. Therefore larger values of k correspond to greater levels of privacy.

To see this more concretely, let's take a look at an example.

13.5.2 k -anonymity example

In this example we have a dataset recording visits to a sexual health clinic. We wish to establish the k -anonymity of this data set, where the diagnosed condition should be kept private. To do help with this, the drug use status has been removed for all patients and only partial information is available about their postcode and age.

	Post Code	Age	Drug Use	Condition
1	OX1****	<20	*	Herpes
2	OX1****	<20	*	Herpes
3	OX2****	>=30	*	Chlamydia
4	OX2****	>=30	*	Herpes
5	OX1****	<20	*	Gonorrhoea
6	OX2****	>=30	*	Gonorrhoea
7	OX1****	<20	*	Gonorrhoea
8	LA1****	2*	*	Chlamydia
9	LA1****	2*	*	Chlamydia

	Post Code	Age	Drug Use	Condition
10	OX2****	>=30	*	Gonorrhoea
11	LA1****	2*	*	Chlamydia
12	LA1****	2*	*	Chlamydia

By grouping observations by each distinct combination of pseudo-identifiers we can establish the equivalence classes within the data.

	Post Code	Age	Drug Use	Condition	<i>Equivalence Class</i>
1	OX1****	<20	*	Herpes	1
2	OX1****	<20	*	Herpes	1
3	OX2****	>=30	*	Chlamydia	2
4	OX2****	>=30	*	Herpes	2
5	OX1****	<20	*	Gonorrhoea	1
6	OX2****	>=30	*	Gonorrhoea	2
7	OX1****	<20	*	Gonorrhoea	1
8	LA1****	2*	*	Chlamydia	3
9	LA1****	2*	*	Chlamydia	3
10	OX2****	>=30	*	Gonorrhoea	2
11	LA1****	2*	*	Chlamydia	3
12	LA1****	2*	*	Chlamydia	3

Here we have three distinct equivalence classes, each with four observations. Therefore the smallest equivalence class is also of size four and this data set is 4-anonymous.

While we can easily identify the equivalence classes in this small dataset, doing so in large datasets is a non-trivial task.

13.6 Improving Privacy

There are three main ways that you can improve the level of privacy within your data, and we have seen examples of two of these already.

Redaction may be applied to individual or to an attribute, leading to a whole row or column being censored. This is quite an extreme approach: it can lead to a large amount of information being removed from the data set. However, sometimes redacting a full row is necessary; for example when that row contains identifying information like a person's name or national insurance number. An additional concern when redacting *rows* from your data is that it will artificially alter the distribution of your sample, making it unrepresentative of the population values.

Aggregation or coarsening is a second approach where the level of anonymity can be increased by binning continuous variables into discrete ranges or by combining categories within a variable that already takes discrete values. The idea here is to reduce the number of equivalence classes within the quasi-identifiers so that the level of k -anonymity is increased.

A similar approach is to corrupt or obfuscate the observed data by **adding noise to the observations**, or permuting some portion of them. The aim is to retain overall patterns but ensure individual recorded values no longer correspond to an individual in the raw data set. The difficulty here is in setting the type and amount of noise to be added to the data to grant sufficient privacy without removing all information from the dataset.

This trade-off between information loss and privacy is a common theme throughout all of these methods.

13.7 Breaking k -anonymity

k -anonymity ensures that there are at least $k-1$ other people with your particular combination of pseudo-identifiers. What it does not do is ensure that there is any variation within a particular group. The dataset on sexual health we just saw was 4-anonymous, but if we know a person how attended the clinic was from a Lancashire (LA) postcode (and in their 20s) then we know for certain that they have Chlamydia. An alternative privacy measure called l -diversity tries to address this issue.

	Post Code	Age	Drug Use	Condition
8	LA1****	2*	*	<i>Chlamydia</i>
9	LA1****	2*	*	<i>Chlamydia</i>
11	LA1****	2*	*	<i>Chlamydia</i>
12	LA1****	2*	*	<i>Chlamydia</i>

A second problem with k -anonymity is that this type of privacy measure is focused entirely on the data available within this dataset. It does not take into account data that might be available elsewhere or might become publicly available in the future. An **external data-linkage attack** can cross-reference this table against other information to reduce the size of equivalence classes and reveal personal information.

13.8 Cautionary tales

13.8.1 Massachusetts Medical Data

Medical research is often slow because it is very difficult to share medical records while maintaining patients' privacy. In the 1990s a government agency in Massachusetts wanted to improve this by releasing a dataset summarising the hospital visits made by all state employees. They were understandably quite careful about this, making this information available only to academic researchers and redacted all information like names, addresses and security numbers. They did include the the patient's date of birth, zip code, and sex - this information was deemed sufficiently general while allowing difference in healthcare provision to be investigated.

Latanya Sweeney is now a pre-eminent researcher in the field of data privacy. In the 1990s she was studying for a PhD at MIT and wanted to demonstrate the potential risks of de-anonymising this sort of data. To demonstrate her point she chose to focus on the public records of Massachusetts' governor, William Weld. For a small fee, Sweeney was able to obtain the voter registration records for the area in which the governor lived. By cross-referencing the two datasets Sweeney was able to uniquely identify the governors medical history and send them to him in the post. This was particularly embarrassing for Weld. since he had previously given a statement reassuring the public that this data release would not compromise the privacy of public servants.

This interaction between Sweeney and the Governor of Massachusetts was significant because it highlighted the potential privacy risks associated with the release of publicly available information. It demonstrated that even when data is stripped of names and other identifying information, it can still be possible to re-identify individuals and potentially access sensitive information. The problem here only grows with the dimension of the dataset - the more characteristics that are measured the greater the chance of one person having a unique combination of those.

13.8.2 Netflix Competition

In 2006 Netflix announced a public competition to improve the performance of their recommender system with a prize of 1 million USD to the first person or group to improve its performance by at least 10%. For this competition over 100 million ratings from 480,000 users were made public. Each entry contained a pseudonymised user ID, a film id, the rating out of 5 stars and the date that rating was given.

User ID	Film ID	Rating	Date
000001	548782	5	2001-01-01
000001	549325	1	2001-01-01

User ID	Film ID	Rating	Date
...

Although the Netflix team had pseudonymised the data (and taken other steps like adding noise to the observations), two researchers at the University of Texas were able to successfully re-identify a 96% of individuals within the data. They did this by cross reference the competition dataset against reviews on the openly available internet movie database (IMDb), working on the supposition that users would rate films on both services at approximately the same time - the 96% figure uses a margin of 3 days.

The researchers went further, showing that if we were to alter the data to achieve even a modest 2-anonymity then almost all of the useful information would be removed from competition data set.

This example should show how difficult it can be to ensure individual privacy in the face of unknown external data sources. It might seem like a trivial example compared to medical records but the media that you consume, and in particular how you rate that media, can reveal you religious beliefs, your political stance or your sexual orientation. These are protected characteristics that you might not want to broadcast freely.

It might not be important if you, or even the average Netflix user, if that information becomes public. What is important is whether any user would find this privacy breach objectionable and potentially come to harm because of it.

13.9 Wrapping Up

- Privacy is a fundamental concern.
- Privacy is hard to measure and hard to ensure.
- Also a model issue, since models are trained on data.
- No universal answers, but an exciting area of ongoing research.

Although we have only scratched the surface of privacy in data science, we will have to wrap this video up here.

We have seen that privacy should be a fundamental concern when working with any from of human-related data. This is chiefly because we aren't in a position to determine what types of privacy breach might significantly and negatively impact the lives of the people we hold data about.

We have seen through one example metric that privacy can be both difficult to measure and even more difficult to preserve. This is not only an issue when releasing data into the world, but also when publishing models trained on this data. Approaches analogous to those used by



Figure 13.1: Latanya Sweeney Speaking in New York, 2017. Image CC-4.0 from [Parker Higgins](#).



Figure 13.2: Netflix logo 2001-2014. Public domain image.

Latanya Sweeney's can be used by bad-actors to identify with high precision whether a given individual was included within the data that was used to train a published model.

There are no universal answers to the question of privacy in data science, this is what makes it an exciting area of ongoing research. It is also for this reason that a lot of Sweeney's work was published only after overcoming great resistance: it exposed systematic vulnerabilities for which there are currently few reliable solutions.

14 Fairness

14.1 Fairness and the Data Revolution



Before the 1990s, large datasets were typically only collected to understand huge, complex systems. These systems might be the weather, public infrastructure (e.g. hospitals, roads or train networks), the stock market or even populations of people.

Collecting high quality data on these systems was immensely expensive but paid dividends by allowing us to describe the expected behaviour of these systems at an aggregate level. Using this sort of information, we can't make journeys or healthcare better at an individual level but we *can* make changes to try and make these experiences better on average.

Things changed with the widespread adoption of the internet in the mid-1990s and the subsequent surge in data collection, sharing and processing. Suddenly, we as individuals shifted from being just one part of these huge processes to being a complex process worth of modelling all on our own.

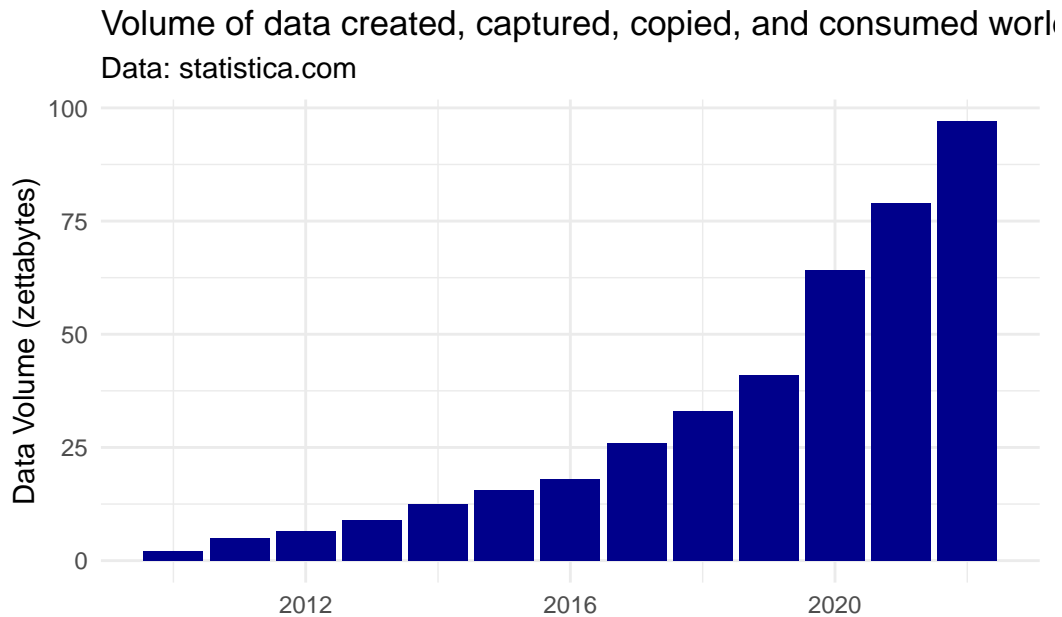


Figure 14.1: Volume of data created, captured, copied, and consumed worldwide from 2010 to 2022.

It was at this point that focus shifted toward making individual, personalised predictions for specific people, based on the vast amounts of data that we generate as we go about our daily lives.

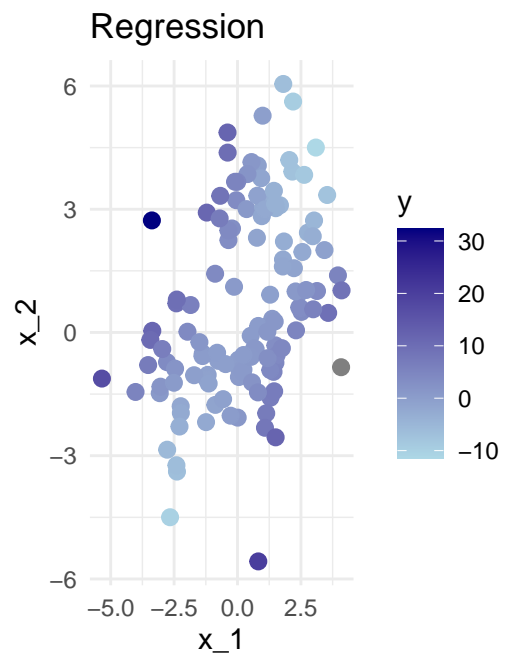
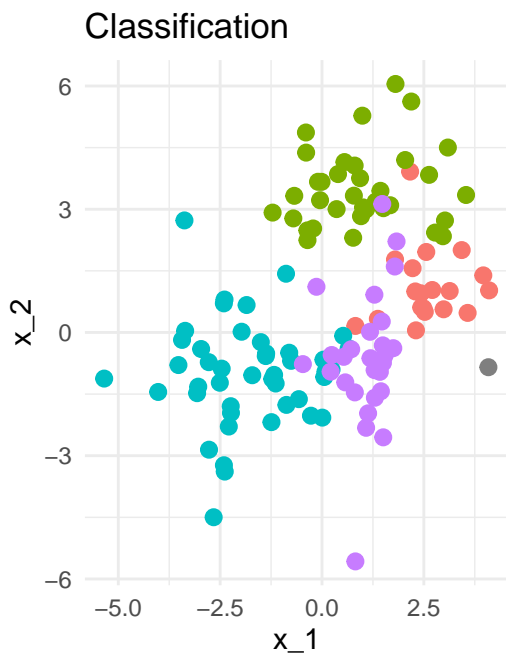
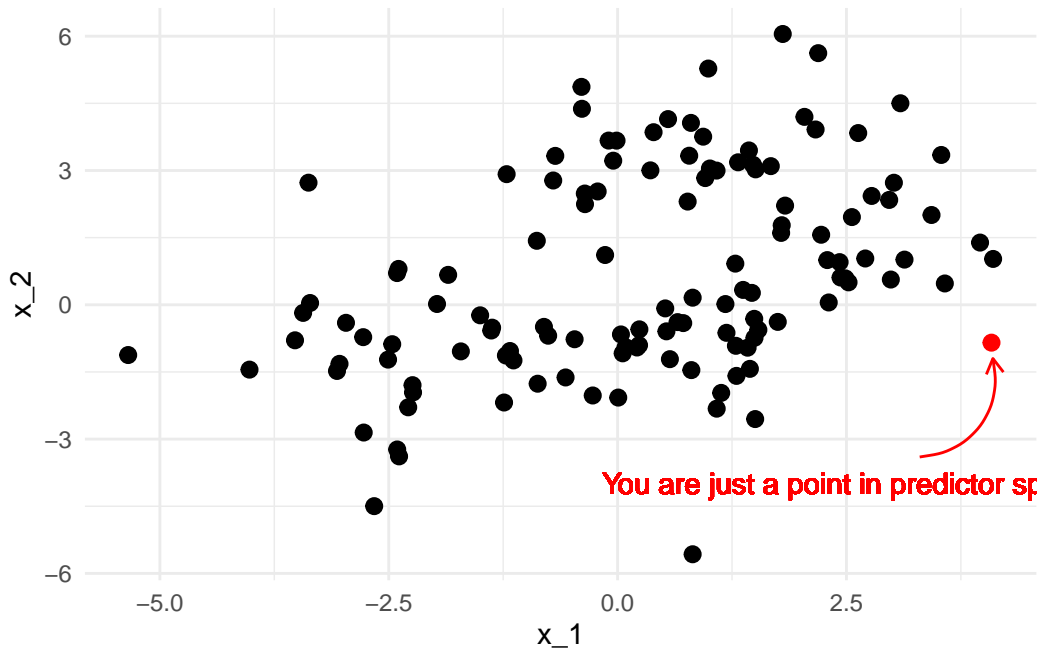
This shift from aggregate to individual behaviour creates the opportunity not only for these predictions to systematically harm groups of people, as they always could, but also to acutely harm individuals.

14.2 You are Your Data

The blunt truth is that, as far as a data science model is concerned, you are nothing more than a point in a high-dimensional predictor space.

The model might use your location in that space to group you with other points that are in some sense “nearby”. Alternatively, the model might estimate some information about you that it currently doesn’t have, based on what it knows about those surrounding points. These other points also represent unique humans with rich and fascinating lives - but the model doesn’t care about that, it is just there to group some points or predict some values.

The idea of fairness comes into data science when we begin to ask ourselves which predictors we should provide the model with when carrying out these tasks. We aren’t asking this from a



model selection stand-point. We are asking what are morally permissible predictors, not what leads to a significant improvement in model fit.

14.3 Forbidden Predictors

The argument about what features of a human being can be used to make decisions about them started well before the 1990s. The most contentious of these arguments centre around the inclusion of characteristics that are either immutable or not easily changed. Some of these characteristics including race, gender, age or religion receive legal protections. These protected attributes are often forbidden to be used in important decisions, such as whether a bank loan is accepted.

This natural lead us to ask what classifies as an important decision?

Protected Characteristics under the [Equality Act \(2010\)](#)

- age
- gender reassignment
- marriage / civil partnership
- pregnancy / parental leave
- disability

- race including colour, nationality, ethnic or national origin
- religion or belief
- sex
- sexual orientation

We also need to be careful if these protected attributes actually have strong predictive power and would improve our predictions (potentially to the benefit of the groups that are being protected by these regulations). Just because a protected attribute isn't used directly within a model that doesn't mean the model will not discriminate according to that attribute.

If we have multiple predictors within a model, then withholding a protected attribute does not make the model ignorant of that attribute. If you have access to someone's browsing history, where they live and some of their recent purchases you can probably make a fairly accurate profile of that person, including many of these supposedly protected attributes. In the same way, a model can use or combine attributes that are not legally protected to create a new variable that acts as an accurate proxy for the protected characteristic.

And why wouldn't our model do this? When using a standard loss function we have literally asked it to get the best possible predictive performance. If a protected attribute has predictive power then the model is likely to approximate it using the predictors that are available to it.

Before we see how to handle this concern, let's step back and consider how we can quantify and measure fairness in a model.

14.4 Measuring Fairness

Converting the concept of fairness into a mathematical statement is a very difficult task. This is partly because moving from natural language to precise formalism is hard, but it's also because the term fairness means different things to different people in different contexts. Because of this, there are many complementary definitions of fairness that all try to capture some intuitive notion of a fair model. However, these measures all capture different facets of this complicated concept. Despite this, these measures vary to such an extent they can't all be satisfied simultaneously.

I'll introduce four such measures shortly, focusing in on the case of binary outcomes where a "positive" response of 1 corresponds to an event that would be considered favourably when taken in context. For example this might be a loan that will be successfully repaid or that a person released on bail will not re-offend.

- Binary outcome $Y \in \{0, 1\}$.

We'll consider the simple case where a binary prediction is made in each instance, and where we want our predictions to be fair across the k distinct levels of some protected attribute A .

- Binary Prediction $\hat{Y} \in \{0, 1\}$.
- Protected attribute A takes values in $\mathcal{A} = \{a_1, \dots, a_k\}$.

14.4.1 Demographic Parity

The first, and potentially most obvious fairness definition is that of demographic parity. Here a model is deemed fair if, across all subgroups of the protected attribute, the probability of predicting a successful outcome is equal.

The probability of predicting a 'positive' outcome is the same for all groups.

$$\mathbb{P}(\hat{Y} = 1|A = a_i) = \mathbb{P}(\hat{Y} = 1|A = a_j), \text{ for all } i, j \in \mathcal{A}.$$

An obvious shortcoming demographic parity is that it does not allow us to account for the fact that a positive outcome might not be equally likely in each of these subgroups. In this way demographic parity is analogous to treating people equally, rather than equitably.

14.4.2 Equal Opportunity

Equality of opportunity addresses this shortcoming by conditioning on a truly positive outcome. Equality of opportunity states that of those who are “worthy” of a loan (in some sense), all subgroups of the protected characteristic should be treated equally.

Among those who have a true ‘positive’ outcome, the probability of predicting a ‘positive’ outcome is the same for all groups.

$$\mathbb{P}(\hat{Y} = 1|A = a_i, Y = 1) = \mathbb{P}(\hat{Y} = 1|A = a_j, Y = 1), \text{ for all } i, j \in \mathcal{A}.$$

14.4.3 Equal Odds

Of course, you have encountered two-way tables, type-I and type-II errors. Equally important as granting loans to people who will repay them is to deny loans to those who cannot afford them.

A model satisfying the equal odds condition can identify true positives and false negatives equally well across all sub-groups of the protected characteristic.

Among those who have a true ‘positive’ outcome, the probability of predicting a ‘positive’ outcome is the same for all groups.

AND

Among those who have a true ‘negative’ outcome, the probability of predicting a ‘negative’ outcome is the same for all groups.

$$\mathbb{P}(\hat{Y} = y|A = a_i, Y = y) = \mathbb{P}(\hat{Y} = y|A = a_j, Y = y), \text{ for all } y \in \{0, 1\} \text{ and } i, j \in \mathcal{A}.$$

14.4.4 Predictive Parity

All of the measures we have considered so far consider the probability of a prediction given the true credit-worthiness of an applicant. Predictive Parity reverses the order of conditioning (as compared to equal opportunity).

It ensures that among those predicted to have a successful outcome, the probability of a truly successful outcome should be the same for all subgroups of the protected characteristic. This ensures that, in our financial example, the bank is spreading its risk exposure equally across all subgroups; each subgroup should have an approximately equal proportion of approved loans being successfully repaid.

The probability of a true ‘positive’ outcome for people who were predicted a ‘positive’ outcome is equal across groups.

$$\mathbb{P}(Y = 1|\hat{Y} = 1, A = a_i) = \mathbb{P}(Y_1 = 1|\hat{Y} = 1, A = a_j) \text{ for all } i, j \in \mathcal{A}.$$

We can play devil’s advocate here and say that this might not be appropriate if there is a genuine difference in the probability of successful repayment between groups.

14.5 Metric Madness

Even with this very simple binary classification problem that there are many ways we can interpret the term fairness. Which, if any, of these will be appropriate is going to be highly context dependent.

An issue with many of these metrics, including some of those introduced, is that they require knowledge of the true outcome. This means that these metrics can only be evaluated retrospectively: if we knew this information to begin with then we wouldn’t need a model to decide who get a loan. On top of this, it means that we only ever get information about the loans that are granted - we don’t have access to the counterfactual outcome of whether a loan that was not granted would have been repaid.

An additional problem is that evaluating these fairness metrics requires us to know which protected sub-group each individual belongs to. This is clearly a problem: to evaluate the fairness of our loan applications we need to know sensitive information about the applicants, who would - very reasonably - be unwilling to provide that information because it legally cannot be used to inform the decision making process. For this reason, an independent third-party is often required to assess fairness by collating data from the applicants and the bank.

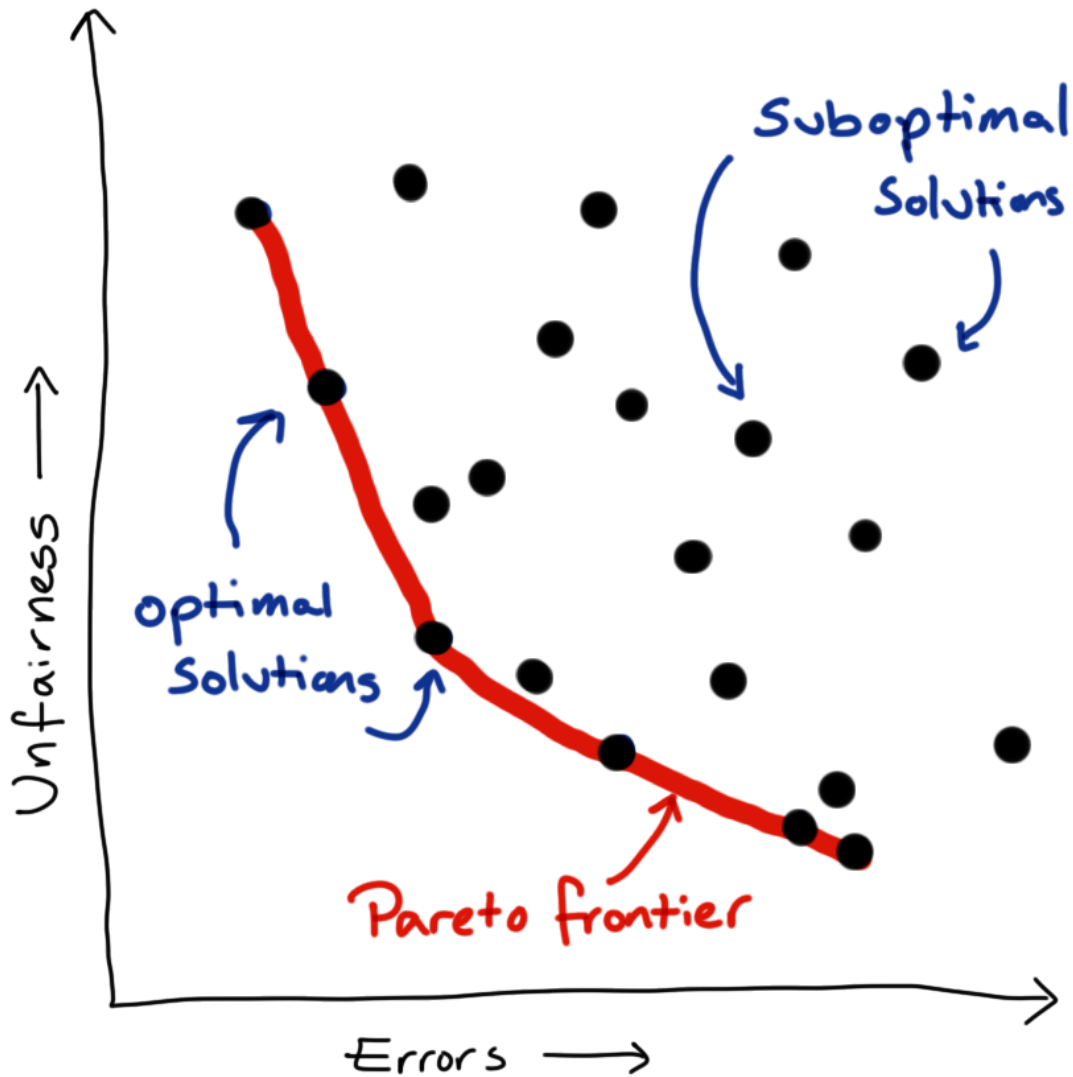
A third complication here is that these definitions deal in strict equalities. In any given sample, these are almost surely not going to be satisfied even if the metric is truly satisfied. A formal statistical test should be used to assess whether these differences are consistent with a truly fair model, however the more common approach is for regulators to set some acceptable tolerance on the discrepancy in metric values between sub-groups.

Finally, it is worth noting that all of these problems arise for a simple binary classifier but most models are far more complicated than this. Even working with these conditional probability statements requires careful attention, but things get much trickier when the response or sensitive attribute are continuous valued or when other, non-sensitive predictors are also included in the model.

14.6 Modelling Fairly

14.6.1 Fairness Aware Loss Functions

Now we have some methods to detect and quantify the fairness of our models, how do we incorporate that into the model fitting process?



We now have multiple objectives: to predict our outcome as accurately as possible while also treating people as fairly as possible (by which ever fairness metric or combination of metrics

we care to consider). Unfortunately, these things are generally in competition with each other. There is no one best model but rather a family of best models, from which we have to pick a single model to implement.

Can resolve this issue by linearisation, create our loss function as a linear weighted sum of the two component objectives. This simplifies the problem mathematically, but actually just shift the problem rather than resolving it. Up to scaling constant, each combination of weights corresponds to a unique point on the Pareto frontier, so we have just translated our problem from picking a point on the frontier to picking a pair of weights.

To do actually resolve this issue we need to define our relative preference between fairness and predictive power. When I say “our preference”, what I actually mean that of the company or organisation for whom we are doing an analysis - not our own personal view. Eliciting this preference and communicating the idea of an optimal frontier can be tricky. One solution is to present a small set of possible models, which represent a range of preferences between the two competing objectives, and ask the stakeholder to choose between these.

14.6.2 Other Approaches

- Re-weighting or resampling to better represent minority groups.
- Forgetting factor to reduce impact of bias in historical data.
- Meta-modelling to intervene in harmful feedback loops.

Whenever we treating all predictions equally and our loss function optimises purely for predictive performance, good predictions for minority groups will never be prioritised. One strategy to correct for this is to either re-weighting or re-sample each observation so that minority groups are given greater importance within the loss function.

A lot of the problems of fairness that we see are because our models are replicating what happens or used to happen in reality. In some cases, this is being better addressed now and a model can be made more fair by down-weighting training data as it ages. This allows our model to more quickly adapt to changes in the system it is trying to represent.

In other cases the use of historically biased data to train models that are put into production has lead to a feedback loop that makes more recent data even more unjust. One example of this, we can consider racial disparity in the interest rates offered on mortgages. Suppose that one racial group of applicants was in the past slightly more likely to default on loans, perhaps due to historical pay inequity. This means that models would likely suggest higher interest loans to this group, in an attempt to offset the bank’s exposure to the risk of non-repayment.

This not only reduces the number of loans that will be granted to that racial group but it *also* makes the loans that are granted more difficult to repay and more likely to be defaulted on. This in turn leads to another increase in the offered interest rate, driving down the number of loans approved and pushing up the chance of non-repayment even further.

The decisions made using this model are impacting its future training data and creating a harmful and self-reinforcing feedback loop. Historical down weighting will do nothing to address this sort of issue, which requires active intervention.

A meta-modelling approach is possible type of intervention. Here post-hoc methods used to estimate the biases within a fitted model and these estimates are used to explicitly correct for historical biases, *before* the model is used to make predictions or decisions.

14.7 Wrapping Up

That's a good point for us to wrap up this introduction to fairness in data science.

We have seen that optimising for predictive accuracy alone can lead to unjust models. We also raised concerns about protected characteristics being included in models, whether that is directly as a predictor or via a collection of other predictors that well approximate them.

We have seen that there are a multitude of measures to assess the fairness of our models. We can combine these with standard metrics for goodness-of-fit to create custom loss functions which represent our preference between fairness and predictive performance.

As with privacy, there are no universal answers when it comes to measuring and implementing fair data science methodology. This is still a relatively new and rapidly evolving field of data science.

15 Codes of Conduct

15.1 Data Science: Miracle Cure and Sexiest Job

It has been more than 10 years since it was proclaimed that data science was the sexiest job of the century. Current turbulence in the technology sector may have you questioning the veracity of that claim, but it still rings true if we take a less myopic view of where data science is used.

Data science and machine learning are being used more extensively than ever across fields including medicine, healthcare and sociology. Data science is also applied to understand our environment, study ecological systems and inform our strategies for their preservation. In addition to this, data science is used widely across the private sector, where it informs business strategies and financial services, alongside the public sector where it influences governance and policy development.

15.2 What could go wrong?

While data science methods can be wonderful, they are not infallible. As the number of use cases increases, we should also expect that the number of misuses or high-profile failures to increase too.

These two news articles highlight just how badly this can all go.

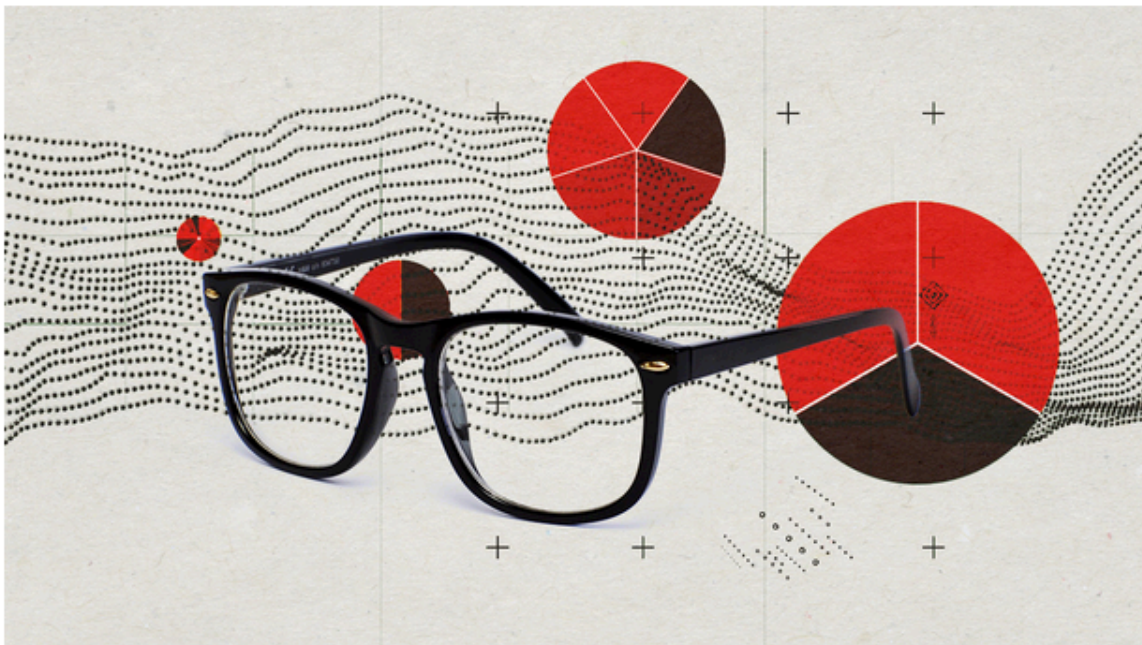
The first article shows how a combination of non-representative training data and lack of consideration on the part of developers led to a facial recognition system that was much less accurate for people with darker skin. This is absolutely unacceptable, particularly given the widespread use of similar systems in important security applications such as biometric ID for online banking or automatic passport gates.

A second example shows the risk of catastrophic failures in self-driving cars. We know that misclassification and missed edge cases are inevitable and that in the case of self-driving cars these errors can cause serious and even fatal accidents. We might then ask ourselves if these errors are acceptable if they lead to fewer accidents or fatalities than would be caused by human drivers.

Is Data Scientist Still the Sexiest Job of the 21st Century?

by Thomas H. Davenport and DJ Patil

July 15, 2022



HBR Staff/StudioM1/Moritz Otto/Getty Images

Figure 15.1: Title of Harvard Business Review article: Is data scientist still the sexiest job of the 21st Century?

Facial recognition fails on race, government study says

20 December 2019



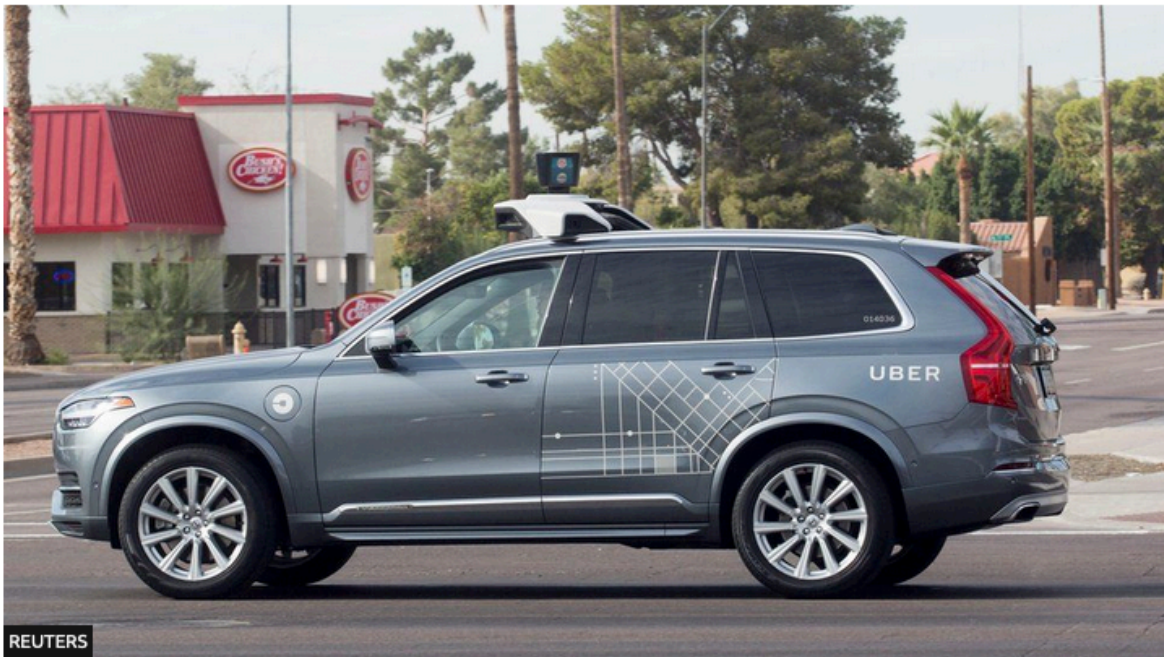
Facial recognition tools are increasingly being used by police forces

A US government study suggests facial recognition algorithms are far less accurate at identifying African-American and Asian faces compared to Caucasian faces.

Figure 15.2: BBC news article: facial recognition fails on race, government study says.

Uber's self-driving operator charged over fatal crash

🕒 16 September 2020



| The self-driving Volvo hit a pedestrian at 39mph, despite the presence of a safety driver

The back-up driver of an Uber self-driving car that killed a pedestrian has been charged with negligent homicide.

Figure 15.3: BBC news article: Uber's self-driving operator charged over fatal crash.

Another important point to establish is where liability falls in such cases: should it be with the operator of the self-driving vehicle, the vendor who sold it or the data scientist who wrote the script controlling the car's actions? If this behaviour was determined by training data for test-drives, should that driver share in the liability.

These are all important questions that we haven't necessarily thought to ask before deploying data science solutions into the world.

15.3 That's not *my* type of data science ...

You might be thinking at this doesn't effect you because you don't work in image analysis or on self-driving cars, but related issues come up in more standard applications of data science.

Consider a retailer implementing targeted promotions. They might combine previous shopping habits and recent product searches to target their offers.

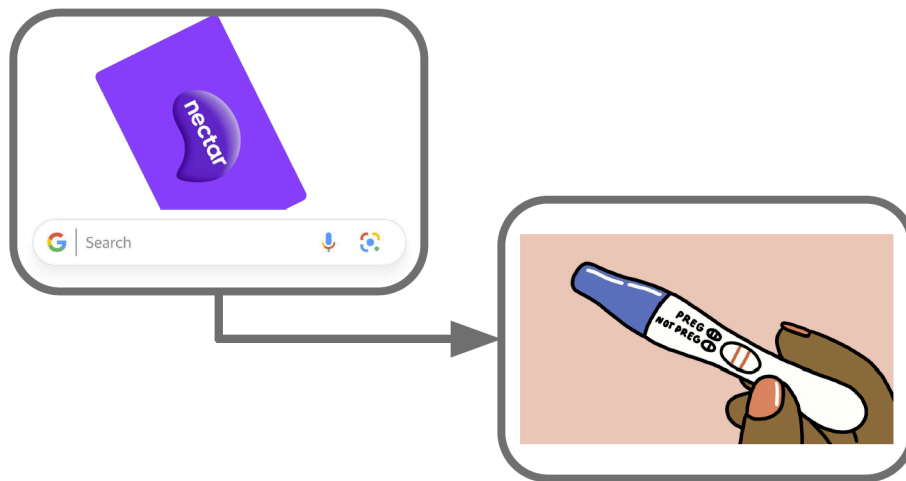


Figure 15.4: Predicting pregnancy from search and purchase history.

Around 7 months ago, one customer stopped regularly buying contraceptives and alcohol and started buying supplements for vitamin D and folic acid. Based on similar behaviour by lots of other customers, a recommender model might now expect a good way to increase sales would be to send that customer offers for nappies, baby food and talcum powder.

This could be seriously upsetting if that customer also happened to be experiencing fertility issues or had been pregnant but had an abortion or miscarriage. It is important that human-overrides are included in such systems like this so that the customer could contact the retailer and prevent this sort of advertising from continuing to happen for months or years to come.

15.4 Technological Adoption Relies on Public Trust



The collapsed 35W bridge in Minneapolis seen on August 2, 2007.
Morry Gash/AP

Figure 15.5: Thirteen people were killed and 145 injured during the Interstate 35W bridge collapse in 2007.

Data science is a new discipline, which means that some of the issues we have mentioned are new and happening for the first time. However, many of these issues are not unique to data science and that means we can learn from other disciplines that have already invested a lot of time and energy into those problems.

Across many other professions it's standard to follow a professional code of practice or to have a code of ethics that applies to everyone working in that space. Usually, the professions that have this in place are also those who are legally liable in some way for the outcomes and consequences of their work.

The most obvious example of this might be doctors and other medical professionals, but this holds equally true in many jobs from law to engineering, where practitioner's work can impact the freedom or safety of other people. When thinking critically about the practical and ethical implications of our work as data scientists, we shouldn't start from scratch but should learn as much as possible from these other fields.

One profession in particular that we can learn from is medicine. Around the world, doctors

agree to uphold the privacy of their patients by maintaining medical confidentiality. They also agree to only act in what they believe to be the best interests of the patient and to do them no harm. These are both admirable principles that can be tracked directly onto our work in data science.

Google's original internal code encapsulated a more extreme version of this non-maleficence principle, starting with the phrase 'Don't be evil' (though this was later rephrased to 'You can make money without being evil').

15.5 A Hippocratic Oath for Data Scientists



Figure 15.6: In *Weapons of Math Destruction* (2016), Cathy O'Neil was among the first authors to make a call for a Hippocratic oath for data scientists

We've seen examples of both the broad and the acute harms that can befall individuals because of data science. With these in mind, it seems reasonable to expect that data scientists should have to be aware of the negative impacts of their work and be required to mitigate these wherever possible.

This is the argument made by Cathy O'Neil in her book *Weapons of Math Destruction*: data scientists, and those working with mathematical models more generally, need an equivalent of the Hippocratic oath that's used in medicine. To understand why this is necessary, we have to understand why it is *not* the case already.

Doing the right thing, in data science as in life, is neither obvious nor easy.

In the most naive sense, negative consequences can come about as a result of a lack of understanding or because of unanticipated consequences. This lack of understanding might relate to the model itself (which might not be explainable) or about the setting in which the model is being applied (which might require domain specific expertise that the data scientist lacks).

Alternatively, this ignorance can be about the groups of people who are most at risk of harm from data science models. Data science methods tend to model expected behaviour and echo biases from within the training data. This means that minority groups or groups that have been historically disadvantaged are most at risk of further harm. This same minority status or historical disadvantage means that these same groups have low representation within data science teams. This increases the chance that data science teams are ignorant of or ignore the breadth and depth of the damage they might be causing.

A second way that data science can lead to harm is when incentives are not properly aligned. As an example we consider a credit scoring application where a data scientist is trying to include fairness measures in addition to optimising predictive accuracy for loan repayments. If business incentives are purely based around immediate increases in profit, then this will likely be very difficult to get put into production. There is a misalignment between the priorities of the business and the ethics of the data scientist.

As we have seen, some errors are inevitable and sometimes the best we can do is to balance an inherent trade-off between conflicting priorities or different sources of harm. In these cases it is vitally important that we highlight this trade-off explicitly and offer decision makers a range of solutions that they may choose according to their own priorities.

Doing the right thing is neither obvious or easy:

- Lack of understanding,
- Unanticipated consequences
- Incentive structures,
- Inherent trade-offs.

15.6 Codes of Conduct

Membership of a professional body can offer some of the ethical benefits of a Hippocratic oath, because these memberships often require agreement to abide by a code of conduct.

This provides individual data scientists with accountability for their actions in a broader sense than through their direct accountability to their employer. This can be helpful as a motivator to at a high quality and in line with ethical guidelines.

It can also be useful as a form of external support, to convince organisations who employ data scientists that ethical considerations are an important aspect of our work. There is also the opportunity for peer-to-peer support, so that best practices can be established and improved in a unified way across industries, rather than in isolation within individual organisations. This might be through informal networking and knowledge sharing or through formal training organised by that professional body.

Data science is still a developing field and there currently is no professional body specifically dedicated to data science. Instead, data scientists might join one or more related professional bodies, such as the [Royal Statistical Society](#), the [Operational Research Society](#), or the [Society for Industrial and Applied Mathematics](#).

15.7 Wrapping Up

Data Science is still maturing, both as a field of study and as a career path. In combination with the widespread adoption of data science in recent years, this means that as a discipline we're now navigating many new modes of failure and ethical issues. However, not all of these issues are entirely novel. There is an awful lot that we can learn, borrow or adapt from more established fields such as medicine, engineering or physics.

It is important that we remain alert to the dangers of our work: both the liability it opens us up to and the harm we might be causing to others, potentially without being aware of either of these things. By joining professional bodies and upholding their codes of conduct we can push back against bad practices and reward good ones.

Checklist

Videos / Chapters

- [Privacy](#) (19 min) [\[slides\]](#)
- [Fairness](#) (20 min) [\[slides\]](#)
- [Codes of Conduct](#) (14 min) [\[slides\]](#)

Reading

Use the [Data Science Ethics](#) section of the reading list to support and guide your exploration of this week's topics. Note that these texts are divided into core reading, reference materials and materials of interest.

Activities

This week has fewer activities, so that you may look over the second assessment before the end of the course.

Core:

- Read [Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification](#) by Joy Buolamwini and Timnit Gebru (2018). Proceedings of the 1st Conference on Fairness, Accountability and Transparency.
- Find an example case study or method relating to ethical data science that has not been covered in the lectures. Share what you find by writing a short summary of the case study or method on the discussion forum.
- Skim over the Professional Guidelines listed in the reference materials for this week, in preparation for the live session.

Bonus

- Complete the [Fairness worksheet](#).

- Complete the [Privacy worksheet](#).

Live Session

In the live session this week we will begin with a few minutes of Q & A about the assessments. We will then break into groups to discuss and compare several sets of professional guidelines on ethical data science.

Finally, if time allows we will round up the session with an activity on randomised response survey designs or data anonymisation.

A Reading List

i Note

Effective Data Science is still a work-in-progress. This chapter is largely complete and just needs final proof reading.

If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

This reading list is organised by topic, according to each week of the course. These are split into several categories.

- **Core Materials:** These form a core part of the course activities.
- **Reference Materials:** These will be used extensively in the course, but should be seen as helpful guides, rather than required reading from cover to cover.
- **Materials of Interest:** These will not form a core part of the course, but will give you a deeper understanding or interesting perspective on the weekly topic. There might be some fun other stuff in here too.

A.1 Effective Data Science Workflows

Core Materials

- The [Tidyverse R Style Guide](#) by Hadley Wickham.
- [Wilson, et al \(2017\)](#). Good Enough Practices in Scientific Computing. PLOS Computational Biology.

Reference Materials

- [R For Data Science Chapters 2, 6 and 8](#) by Hadley Wickham and Garrett Golemund. Chapters covering R workflow basics, a scripting and project based workflow.
- [Documentation](#) for the {here} package

- [R Packages Book](#) (Second Edition) by Hadley Wickham and Jenny Bryan.

Materials of Interest

- [STAT545, Part 1](#) by Jennifer Bryan and The STAT 545 TAs
- [What they forgot to teach you about R, Chapters 2-4](#) by Jennifer Bryan and Jim Hester.
- [Broman et al \(2017\)](#). Recommendations to Funding Agencies for Supporting Reproducible Research. American Statistical Association.
- [Advanced R](#) by Hadley Wickham Section introductions on [functional](#) and [object oriented](#) approaches to programming.
- [Atlassian Article](#) on Agile Project Management
- [The Pragmatic Programmer, 20th Anniversary Edition Edition](#) by David Thomas and Andrew Hunt. The section on [DRY coding](#) and a few others are freely available.
- [Efficient R programming](#) by Colin Gillespie and Robin Lovelace. Chapter 5 considers [Efficient Input/Output](#) is relevant to this week. Chapter 4 on [Efficient Workflows](#) links nicely with last week's topics.
- [Towards A Principled Bayesian Workflow](#) by Michael Betancourt.
- [Happy Git and GitHub for the useR](#) by Jennifer Bryan
- [Make Tutorial](#) by the Monash Informatics Platform.
- [Makefiles for R and LaTeX projects](#) blog post by Rob Hyndman
- [Makefile tutorial](#) by Chase Lambert

A.2 Acquiring and Sharing Data

Core Materials

- [R for Data Science Chapters 9 - 12](#) by Hadley Wickham. These chapters introduce tibbles as a data structure, how to import data into R and how to wrangle that data into tidy format.

- [Efficient R programming](#) by Colin Gillespie and Robin Lovelace. Chapter 5 considers [Efficient Input/Output](#) is relevant to this week.
- [Wickham \(2014\)](#). Tidy Data. Journal of Statistical Software. The paper that brought tidy data to the mainstream.

Reference Materials

- The `{readr}` [documentation](#)
- The `{data.table}` [documentation](#) and [vignette](#)
- The `{rvest}` [documentation](#)
- The `{tidyr}` [documentation](#)
- MDN Web Docs on [HTML](#) and [CSS](#)

Materials of Interest

- [Introduction to APIs](#) by Brian Cooksey
- [R for Data Science \(Second Edition\)](#) Chapters within the [Import](#) section.

This covers importing data from spreadsheets, databases, using Apache Arrow and importing hierarchical data as well as web scraping.

A.3 Data Exploration and Visualisation

Core Materials

- [Exploratory Data Analysis with R](#) by Roger Peng.

Chapters 3 and 4 are core reading, respectively introducing [data frame manipulation with {dplyr}](#) and an example [workflow for exploratory data analysis](#). Other chapters may be useful as references.

- [Flexible Imputation of Missing Data](#) by Stef van Buuren. [Sections 1.1-1.4](#) give a thorough introduction to missing data problems.

Referene Materials

- A ggplot2 Tutorial for Beautiful Plotting in R <https://www.cedricscherer.com/2019/08/05/a-ggplot2-tutorial-for-beautiful-plotting-in-r/> by Cédric Scherer.
- The `{dplyr}` [documentation](#)
- [RStudio Data Transformation Cheat Sheet](#)
- [R for Data Science \(First Edition\)](#) Chapters on [Data Transformations](#), [Exploratory Data Analysis](#) and [Relational Data](#).
- Equivalent sections in R for Data Science [Second Edition](#)

Materials of Interest

- [Wickham, H. \(2010\)](#). A Layered Grammar of Graphics. *Journal of Computational and Graphical Statistics*.
- [Better Data Visualisations](#) by Jonathan Schwabish
- [Data Visualization: A Practical Introduction](#) by Kieran Healy

A.4 Preparing for Production

Core Materials

- [The Ethical Algorithm](#) M Kearns and A Roth (Chapter 4)
- [Ribeiro et al \(2016\)](#). “Why Should I Trust You?”: Explaining the Predictions of Any Classifier.

Reference Materials

- The [Docker Curriculum](#) by Prakhar Srivastav.
- [LIME package documentation](#) on CRAN.
- [Interpretable Machine Learning: A Guide for Making Black Box Models Explainable](#) by Christoph Molnar.
- Documentation for `apply()`, `map()` and `pmap()`
- [Advanced R \(Second Edition\)](#) by Hadley Wickham. [Chapter 23](#) on measuring performance and [Chapter 24](#) on improving performance.

Materials of Interest

- [The ASA Statement on \$p\$ -values: Context, Process and Purpose](#)
- [The Garden of Forking Paths: Why multiple comparisons can be a problem, even when there is no “Fishing expedition” or “p-hacking” and the research hypothesis was posited ahead of time.](#) A Gelman and E Iken (2013)
- [Understanding LIME tutorial](#) by T Pedersen and M Benesty.
- [Advanced R \(Second Edition\)](#) by Hadley Wickham. [Chapter 25](#) on writing R code in C++.

A.5 Data Science Ethics

Core Materials

- [The Ethical Algorithm](#) M Kearns and A Roth. Chapters 1 and 2 on Algorithmic Privacy and Algorithmic Fairness.
- [Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification](#) by Joy Buolamwini and Timnit Gebru (2018). Proceedings of the 1st Conference on Fairness, Accountability and Transparency.
- [Robust De-anonymization of Large Sparse Datasets](#) by Arvind Narayanan and Vitaly Shmatikov (2008). IEEE Symposium on Security and Privacy.

Reference Materials

- [Fairness and machine learning Limitations and Opportunities](#) by Solon Barocas, Moritz Hardt and Arvind Narayanan.
- Professional Guidelines on Data Ethics from:
 - [The American Mathematical Society](#)
 - [The European Union](#)
 - [UK Government](#)
 - [Royal Statistical Society](#)
 - [Dutch Government](#)

Materials of Interest

- [Algorithmic Fairness](#) (2020). Pre-print of review paper by Dana Pessach and Erez Shmueli.

References