R Techniques for Reproducible Research Day 2

Tom Robinson

April 2019

Recap

In yesterday's session we:

- Motivated why we should we care about reproducibility
- How coding clarity is integral to reproducible research
- Covered the basics of coding style that boost the clarity of your code

In today's session we will:

- Focus on the substance of our code – writing efficient code that makes both our intentions clear and easy to reproduce
- Go through the basics of setting up reproducible workflows

Motivation for writing efficient code

"It doesn't matter how I code, so long as the computer can understand it!"

- R is sufficiently flexible that there are often many different ways to get the same result
- Not all methods are created equal
- Different ways of doing the same thing may increase or diminish the reproducibility of your code!

Example: returning a list of names based on conditions

```
# Base-R manipulation of a dataframe:
living_male_names1 <- data[data$sex == "Male" &
    data$dth_flag == 0,"full_name"]

# Another base-R alternative (altering a vector):
living_male_names2 <- data$full_name[data$sex == "Male" &
    data$dth_flag == 0]

# And yet another base-R alternative (altering a vector):
living_male_names3 <- data[data$sex == "Male" &
    data$dth_flag == 0,]$full_name

# Or a tidyverse solution:
living_males_names4 <- data %>%
    filter(sex == "Male",
    dth_flag == 0) %>%
    select(full_name)
```

Which method is best?

When we talk about writing code efficiently, we are talking about multiple things:

Which method is best?

When we talk about writing code efficiently, we are talking about multiple things:

- How clear it is to both yourself and other readers [**code clarity**]

Which method is best?

When we talk about writing code efficiently, we are talking about multiple things:

- How clear it is to both yourself and other readers [**code clarity**]
- How succinctly your code solves a specific problem [**code minimisation**]

Which method is best?

When we talk about writing code efficiently, we are talking about multiple things:

- How clear it is to both yourself and other readers [**code clarity**]
- How succinctly your code solves a specific problem [**code minimisation**]
- How quickly it runs/ how computationally intensive the process is [**run-time optimisation**]

Which method is best?

When we talk about writing code efficiently, we are talking about multiple things:

- How clear it is to both yourself and other readers [**code clarity**]
- How succinctly your code solves a specific problem [**code minimisation**]
- How quickly it runs/ how computationally intensive the process is [**run-time optimisation**]
- The time it takes to write the code [**coding-time optimisation**]

Which method is best?

When we talk about writing code efficiently, we are talking about multiple things:

- How clear it is to both yourself and other readers [**code clarity**]
- How succinctly your code solves a specific problem [**code minimisation**]
- How quickly it runs/ how computationally intensive the process is [**run-time optimisation**]
- The time it takes to write the code [**coding-time optimisation**]

These factors are not necessarily mutually reinforcing:

- The most succinct code may not be the most clear
- The most optimised code (using parallel-processing, minimised memory usage, efficient looping etc.) may take days to write/test/refine...

Efficiency and reproducibility

How do these different types of efficiency impact reproducibility:

- We've discussed the utility of clear code yesterday!
- Succinct code is also (typically) easier to follow (and to debug)
- Overly-intensive code that takes hours to run can limit others' ability to reproduce findings.
  - Are there optimisations I can make to speed up my code?
  - Does my code really need to be this computationally demanding?
- Code that takes years to write is unlikely to get reproduced because it'll never be released!

Two rules-of-thumb for efficient coding

1. Use packages like **tidyverse** to make your code more efficient as you write it
   - More intuitive syntax
   - More readable (vertically biased)
   - Effective (powerful data manipulations with single commands)
2. Optimize your existing code: “code first, optimise later”

The **tidyverse** suite

The **tidyverse** suite

- Developed chiefly by Hadley Wickham, and funded by RStudio
- The aim is to make R more amenable for “data-science”
- A series of packages focused on different aspects of working with data:
  - readr: Reading/writing data
  - tidyr: Making data ‘tidy’
  - dplyr: Manipulating data
  - ggplot2: Visualising data
  - purrr: Working with functions and vectors

Tibbles

- Tibbles are *almost* identical to data.frames
- But they have more consistent and tidy features (a subset of a tibble is always a tibble)
- A good blog post on the differences: `https://www.jumpingrivers.com/blog/the-trouble-with-tibbles/`

"Simple things like checking the dimensions of your data or converting strings to factors are small jobs. Small jobs that take time. With tibbles they take no time. Tibbles force you to look at your data earlier; confront the problems earlier. Ultimately leading to cleaner code."

- In reality, you will probably forget you are even working with tibbles

Piping

- In base-R you have two strategies for manipulating your data i.e. transforming one thing into another:
    - New line, new variable
    - Nested functions
- Neither is particularly good for reproducibility
- Piping (from the magrittr package, but also bundled in tidyverse) allows you to 'pipe' your data through a series of functions
- It's efficient, quick to code, and readable!

Piping

The piping operator is "$\% > \%$"

- $\% > \%$ carries forward whatever value is to the left of the operator (a tibble, a vector, a string etc.) and passes it to whatever is on the right
- It then saves the result to the **original** variable

```
data <- read_csv("data/got_data_final.csv") %>%
        summary()
```

- The rest of the **tidyverse** suite is designed to work with piping, and automatically feeds in the left-hand side
- We can chain together multiple pipes to keep manipulating the same object without creating new variables each time

Piping extras

1. Piping works with non-**tidyverse** function calls too:
    - Use '.' (period) to stand in for whatever you are piping through
    - You can use '.' just like any variable
    - If '.' is a dataframe: .$VAR_NAME is a column variable, and
    - .[ROW_CONDITION, COL_CONDITION] is a (subsetted) dataframe
2. There are other piping operators if you load the **magrittr** package directly
    - Use "$\%T > \%$" to assign the object on the left of the operator, but run the code on the right nonetheless
    - This operator *can* be useful when visualizing results, although it is not the most useful if you use **ggplot2**
    - Hopefully at some point **tidyverse** will implement $\%T > \%$ in a nice way!

Filter & Select

Another really awkward part of base-R is subsetting data.frames:

```
data_subset <- data[ROW CONDITIONS, COLUMN CONDITIONS]
```

- The logic behind this syntax is great, but with anything but the most basic condition it becomes unreadable
- **dplyr** from **tidyverse** makes this much easier and readable:
  - To select columns, use “select()”
  - To filter rows, use “filter()”
- Both work with piping, and both use standard evaluation i.e. you can drop the quote marks around variable names

```
data <- read_csv("data/got_data_final.csv") %>%
        select(name, allegiance, gender, dth_flg) %>%
        filter(allegiance == "Stark")
```

Modifying variables: renaming and mutating

But what about if we want to modify our variables or create new variables using our existing data? Easy!

- "rename()" does exactly what it says on the tin
  $\% > \% rename(NEW\_NAME = OLD\_NAME, ...)$
- "mutate()" allows you to alter existing variables, or create new ones!
- You can rename multiple variables, or mutate new/existing ones, in the same command by separating each one with commas
- most **dplyr** commands like rename and mutate also have conditional versions: "mutate_if()", "rename_if()" that allow you to selectively and consistently manipulate specific subsets of your data

Grouping and Summarising

- With *filter*, *select*, *rename* and *mutate* you can do almost all single-level data manipulation you need
- But sometimes we want to aggregate our data (e.g. from individuals to groups) and generate new summary values
- Most basically, **tidyverse** handles this through two commands: "group_by()" and "summarise()"

group_by()

- group_by(VARIABLES) separates your data into a series of tibbles subsetted by the given categorical variable(s)
- Used on its own, the results are invisible! You won't see any change to your original data
- But "underneath the hood" your existing tibble becomes segmented by the categories you choose
- Once you have grouped your data, you can then use "summarise()"

summarise()

- Once you've grouped your data, you can generate new aggregate-level data using "summarise()"
- It is like "mutate()" but operates at the group level and collapses individual rows of data as a result
- We can use typical functions like "N()", "mean()", and "sum()" to summarise our data by group

```
data <- read_csv("data/got_data_cleaned.csv") %>%
    select(full_name, allegiance_last,
           sex,dth_flag, prominence) %>%
    filter(dth_flag == 0) %>%
    group_by(allegiance_last, sex) %>%
    summarise(surviving = n(),
              avg_prominence = mean(prominence, na.rm=TRUE))
```

Brief pause: the benefits of functions

Functions are ubiquitous in R:

- library(...), read_csv(...), mean(...) etc. are all functions
- But we can also create custom functions that boost the readability and efficiency of our code

Benefits:

- Modular - we can redeploy a single function multiple times
- Tidy - we don't have to copy and paste big blocks of code (and we can store functions elsewhere!)
- Easier to debug - the finite number of inputs and ouputs makes it easier to trace problems

Data cleaning function

```
data_clean <- function(filename) {
###########################################################
# Input: filename as string
#
# Output: cleaned dataframe, saves new clean .csv
###########################################################
  # ...
  ## Load in data and rename vars
  data <- read_csv(filename) %>% ... %>%
    mutate(sex = ifelse(sex == 1, "Male",
                        ifelse(sex == 2, "Female",NA)),...) %>%
    # Get rid of empty columns (caused by excel file)
    select(-grep("X",names(.)))

  write_csv(data, "data/got_data_cleaned.csv")
  return(data)
}
```

Nesting

Sometimes we want to perform tests or calculate models for various sub-groups of our main dataset rather than just generating summary statistics etc.

- Like "summarise()", we can use group_by() to to break the dataset up into subsetted tibbles
- Then we call "nest()" to create a new tibble that contains, as a column, the subsetted tibble for each group
- Unlike "summarise()", we have preserved all the underlying data, by nesting the subsets as separate tibbles within a tibble!
- The beauty of this approach is that we can then use **purrr**'s map command[1] to run functions, regression models, and other useful procedures in a super-efficient way!

[1] Bundled in **tidyverse**

Mapping

- So we've nested our data into a compact tibble of tibbles!
- Suppose we also have some function that we want to compute, like a basic "lm(...)" model
- "map()" runs that function on every element of a column in our tibble i.e. the column that contains the individual, subsetted tibbles!
- The output (a model object, for example) is stored in a new column
- This procedure is like a cleaner, more obvious alternative to apply (and its relatives sapply, lapply etc.)
- We can also modify functions to generate tidier output (i.e. convert model objects into plain old tibbles!)
    - Use the package **broom**, and its function *tidy(...)*

Un-nesting

Now we have a tibble of tibbles and model objects! How can we see what's going on?

- Since a tibble is just like a data.frame, we can inspect its elements using [...] and [[...]] notation
- But we can also "unnest()" our tibble to create a 'long' tibble that expands the output of our map() call
- I can then plot/make a table of the results, or pass the output onto another function/procedure.

Is nesting/mapping useful?

Is this really that useful?

Is nesting/mapping useful?

Is this really that useful?

- For running a small number of regressions - probably not (though it is still tidier in my opinion)

Is nesting/mapping useful?

Is this really that useful?

- For running a small number of regressions - probably not (though it is still tidier in my opinion)
- But as datasets become bigger, and we want to run 10, 20, or even 100 regressions...

Is nesting/mapping useful?

Is this really that useful?

- For running a small number of regressions - probably not (though it is still tidier in my opinion)
- But as datasets become bigger, and we want to run 10, 20, or even 100 regressions...
- ...the readability of our code will decline with:
  1. The increased number of coding lines to run 100 separate models
  2. The proliferation of model objects

Is nesting/mapping useful?

Is this really that useful?

- For running a small number of regressions - probably not (though it is still tidier in my opinion)
- But as datasets become bigger, and we want to run 10, 20, or even 100 regressions...
- ...the readability of our code will decline with:
    1. The increased number of coding lines to run 100 separate models
    2. The proliferation of model objects
- The benefit of the nested approach is that it produces exactly the same objects (we can still access individual models etc.) but in a more efficient, readable and tidy way!

Recap

So far, the **tidyverse** universe has helped us to:

- Read in data into a consistent and useful object type (the trusty tibble!)
- Subset our data using transparent and accessible syntax
- Generate summary statistics using group_by and summarise
- And nest our data to easily and efficiently map functions to subsets of our data

Let's finish off the **tidyverse** part of this course by visualising the results.

ggplot-ing

“ggplot2()” is now the standard for plotting graphs in R

- The “grammar of graphics” approach (data + coordinates + geoms) is reasonably intuitive, and much more flexible than base-R plots
- The key thing to remember are:
  - You layer various geoms etc. on top of each other
  - You use aes(...) to set parameters that refer to variables in the main data; aes ‘inherits’ that data and uses standard evaluation
- It’s fully integrated into the tidyverse!

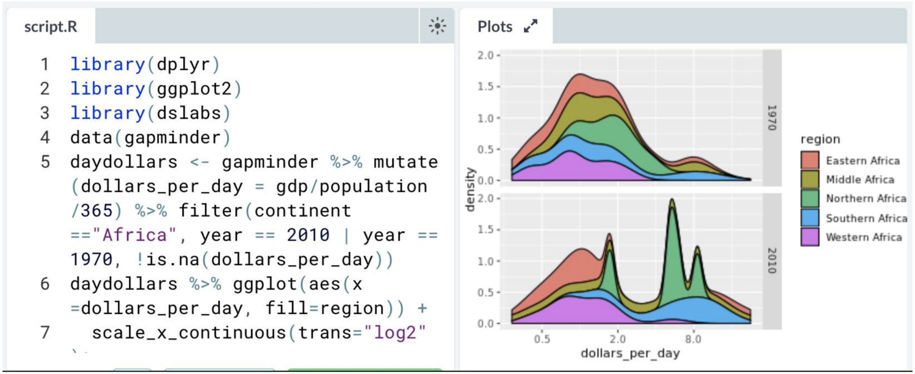
ggplot-ing

```
prominence_survival <- ... %>%
    unnest() %>%
    ggplot(aes(x = , y = , fill = )) +
        geom_point() +
        theme_minimal() +
        ggsave("figures/prom_survival.pdf",
                device = ".pdf",
                width = 6,
                height = 8)
```

- With the piping in place, the variable here gets assigned the ggplot object, not the code!
- Hard-code the image save, and save the image to a separate folder

Warning: **tidyverse** doesn't solve every problem!

It's perfectly possible to use **tidyverse** and still have code that is hard to read/understand!



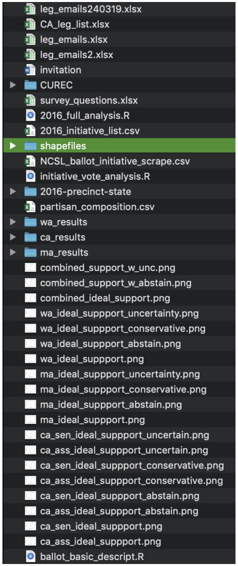
```
library(dplyr)
library(ggplot2)
library(dslabs)
data(gapminder)
daydollars <- gapminder %>% mutate(dollars_per_day = gdp/population/365) %>% filter(continent=="Africa", year == 2010 | year == 1970, !is.na(dollars_per_day))
daydollars %>% ggplot(aes(x=dollars_per_day, fill=region)) +
  scale_x_continuous(trans="log2"
```

Workflow basics

- Raw data should remain raw
- Separate your code into separate scripts
- Establish a good file structure for your project
- Set up versioning ASAP - add, commit, push, repeat

File structure

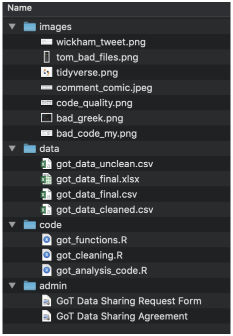
There are bad file structures:



```
leg_emails240319.xlsx
CA_leg_list.xlsx
leg_emails.xlsx
leg_emails2.xlsx
invitation
CUREC
survey_questions.xlsx
2016_full_analysis.R
2016_initiative_list.csv
shapefiles
NCSL_ballot_initiative_scrape.csv
initiative_vote_analysis.R
2016-precinct-state
partisan_composition.csv
wa_results
ca_results
ma_results
combined_support_w_unc.png
combined_support_w_abstain.png
combined_ideal_support.png
wa_ideal_suppport_uncertainty.png
wa_ideal_suppport_conservative.png
wa_ideal_suppport_abstain.png
wa_ideal_suppport.png
ma_ideal_suppport_uncertainty.png
ma_ideal_suppport_conservative.png
ma_ideal_suppport_abstain.png
ma_ideal_suppport.png
ca_sen_ideal_suppport_uncertain.png
ca_ass_ideal_suppport_uncertain.png
ca_sen_ideal_suppport_conservative.png
ca_ass_ideal_suppport_conservative.png
ca_sen_ideal_suppport_abstain.png
ca_ass_ideal_suppport_abstain.png
ca_sen_ideal_suppport.png
ca_ass_ideal_suppport.png
ballot_basic_descript.R
```

File structure

And there are better file structures:



Name
- images
  - wickham_tweet.png
  - tom_bad_files.png
  - tidyverse.png
  - comment_comic.jpeg
  - code_quality.png
  - bad_greek.png
  - bad_code_my.png
- data
  - got_data_unclean.csv
  - got_data_final.xlsx
  - got_data_final.csv
  - got_data_cleaned.csv
- code
  - got_functions.R
  - got_cleaning.R
  - got_analysis_code.R
- admin
  - GoT Data Sharing Request Form
  - GoT Data Sharing Agreement

File structure

In general, try to separate your files into types:

- Data file(s)
- Code
- Output tables and figures
- LaTeXfiles

How you set up your file structure will depend on how large the project is, who your contributors are etc.

- There's often no point having a folder just for one file
- Your file structure influences your project beyond simply your R code
- A well-organised project means you could feasibly execute data cleaning through to re-compiling your LaTeXpaper without ever leaving the terminal!

Handling file structures within R

The key things within R to remember are:

- setwd("...") is always bad!
  - Almost always bespoke to the user
  - Encourages bad file structure habits
  - Force yourself to set the working directory manually (through the console or via the session dropdown)
- Use relative file directories within your code: read_csv("/data/...") and ggsave("/figures/...")
- Packages like 'here' can switch between subdirectories quite effectively, but it's unnecessary if you are strict about maintaining your directories.
- The aim is that your main directory is like a briefcase - you can carry it around and give it to others

Versioning: the basics

Worst case scenario:

- Single-copy of your code
- You overwrite 1000 lines accidentally
- Then RStudio crashes/undo refuses to work/your laptop runs out of battery
- You might have lost hours/days/months of work

Versioning: the basics

Worst case scenario:

- Single-copy of your code
- You overwrite 1000 lines accidentally
- Then RStudio crashes/undo refuses to work/your laptop runs out of battery
- You might have lost hours/days/months of work

Better-but-still-awful scenario:

- You have a fully-working set of tools used by multiple people
- You change a series of steps in a function
- Save, upload, forget about it
- Email from a collaborator the next day:
  "Tom, I keep getting this error... Presentation's in an hour."

Versioning: the basics

- Versioning is the process by which we save indexed *copies* of our work:
  - If we do it regularly, it prevents us from losing 100(0)'s of lines of code
  - We can quickly revert to previous, *working* versions if something goes wrong!
- In its most basic form, versioning involves saving new copies of your files:
  - "got_analysis_010419.R"
  - "got_analysis_020419.R"
  - "got_analysis_020419_pm.R"
  - "got_analysis_020419_pm2.R"
- But this can become tedious, untidy, (and annoying for collaborators)!

Versioning: Git and GitHub

An alternative is to use a more sophisticated version control system like Git:

- Free
- Open-source
- Industry-standard version control, integrated into lots of software like RStudio

Versioning: Git and GitHub

An alternative is to use a more sophisticated version control system like Git:

- Free
- Open-source
- Industry-standard version control, integrated into lots of software like RStudio
- An offline solution (it sits hidden in your directory, recording *changes* to your files)
- `https://git-scm.com/`
- If your computer dies, so too does your local versioning!

GitHub

GitHub uses Git but adds online functionality:

- Backs up your data remotely
- Is great for collaborating and sharing projects
- Promotes open and reproducible research
- Easy to follow instructions for setting up a local Git repository and linking it to GitHub
- Academic students can get free premium accounts (letting you make projects private/by invitation only)
- `https://education.github.com/`

Using Terminal

- Terminal is the gateway to controlling your Unix-machine via the command line
- Windows has a command line tool, too
- We can do things really quickly using very short prompts
- When we open Terminal, it will take us to a starting directory
- We can then navigate through to other directories on our system
- And from those directories, call specific commands

Terminal commands

- How do I change directory: **cd**
- All paths are relative
- To go up a level: **cd ..**
- To go down a level: **cd** FOLDER_NAME
- We can string these together: **cd** ../FOLDER1/FOLDER1_1
- **ls** (short for list) prints out all the files within your current directory

Versioning: Git procedures

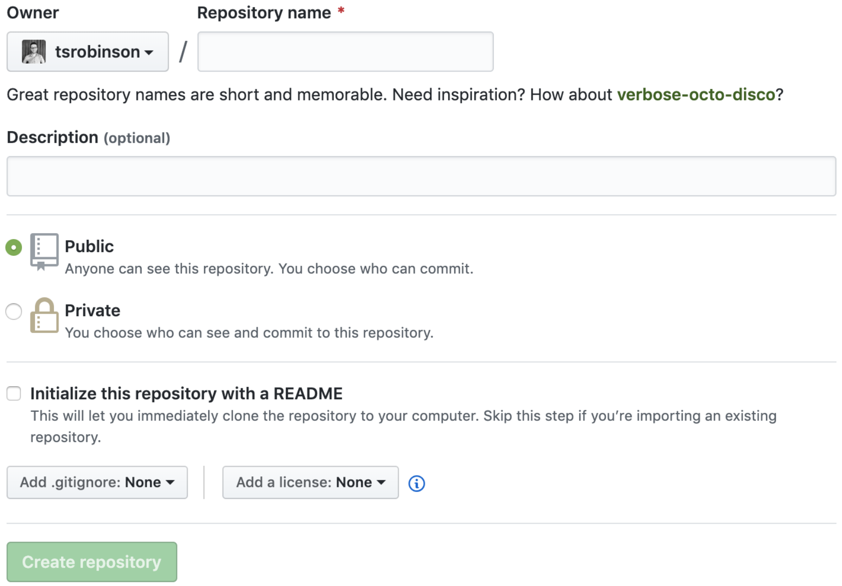
Using Git(Hub) is easy once it's set up:

1. Make changes to your files
2. Open your Terminal (or Command Line for Windows users) and navigate to your project directory
3. If this is your first commit, then we need to set up the remote call (following GitHub instructions)
4. Stage your changes to GitHub:
   git add -A
5. Commit your staged changes:
   git commit -m "Description of changes"
6. At this point your local version control is done!
7. Push your changes to GitHub to store remotely:
   git push -u origin master

Versioning: Git and RStudio

- Using the command line is a bit tedious (though it has a certain retro appeal)
- You can do version control using GitHub directly through RStudio
- Instructions to do so are here: `https://happygitwithr.com/rstudio-git-github.html`
- In my opinion, there are drawbacks to this approach:
  - It relies on setting up Rproject files which can be a bit untidy
  - Terminal is foolproof once you get used to it

Create new GitHub repository


Owner

Repository name *

tsrobinson /

Great repository names are short and memorable. Need inspiration? How about verbose-octo-disco?

Description (optional)

[✓ Public]
Anyone can see this repository. You choose who can commit.

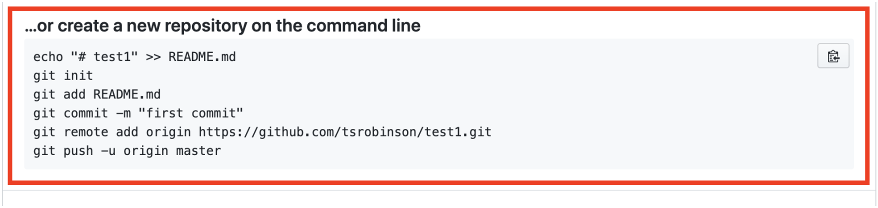
Private
You choose who can see and commit to this repository.

Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None

Add a license: None

Initialise your repository locally



...or create a new repository on the command line

```
echo "# test1" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/tsrobinson/test1.git
git push -u origin master
```

Final thoughts

- Consistency and clarity when coding are key for reproducibility

Final thoughts

- Consistency and clarity when coding are key for reproducibility
- **tidyverse**, tidy code, tidy mind!

Final thoughts

- Consistency and clarity when coding are key for reproducibility
- **tidyverse**, tidy code, tidy mind!
- Many of the key principles in this course translate to other coding languages and types of project

Final thoughts

- Consistency and clarity when coding are key for reproducibility
- **tidyverse**, tidy code, tidy mind!
- Many of the key principles in this course translate to other coding languages and types of project
- Reproducible research has never been easier to do and there are so many great resources out there

Final thoughts

- All the code as well as the slides (not the GoT data, sorry!) are available on my GitHub: `tsrobinson.github.com/r_for_reproducible_research/`
- I'm always keen to hear your feedback, or if you have any other questions!
- Email: `thomas.robinson@politics.ox.ac.uk`
- Website: `https://ts-robinson.com`