Methods Camp

UT Austin, Department of Government

Andrés Cruz and Matt Martin

2023-08-10

Table of contents

Class schedule

Date	Time	Location
Thurs, Aug. 10	9:00 AM - 4:00 PM	RLP 1.302D
Fri, Aug. 11	9:00 AM - 4:00 PM	RLP 1.302E
Sat, Aug. 12	No class	-
Sun, Aug. 13	No class	-
Mon, Aug. 14	9:00 AM - 4:00 PM	RLP 1.302E
Tues, Aug. 15	9:00 AM - 4:00 PM	RLP 1.302E
Weds, Aug. 16	9:00 AM - 4:00 PM	RLP 1.302E

On class days, we will have a lunch break from 12:00-1:00 PM. We'll also take short breaks periodically during the morning and afternoon sessions as needed.

Description

Welcome to Introduction to Methods for Political Science, aka "Methods Camp"! In the past our incoming students have told us their math skills are rusty and they would like to be better prepared for UT's methods courses. Methods Camp is designed to give everyone a chance to brush up on some skills in preparation for the Stats I and Formal Theory I courses. The other goal of Methods Camp is to allow you to get to know your cohort. We hope that struggling with matrix algebra and the dreaded chain rule will still prove to be a good bonding exercise.

As you can see from the above schedule, we'll be meeting on Thursday, August 10th and Friday, August 11th as well as from Monday, August 14th through Wednesday, August 16th. Classes at UT begin the start of the following week on Monday, August 22nd. Below is a tentaive schedule outlining what will be covered in the class, although we may rearrange things a bit if we find we're going too slowly or too quickly through any of the material.

Course outline

1 Thursday morning: R and RStudio

- Introductions
- RStudio (materials are on the website as zipped RStudio projects)
- Objects (vectors, matrices, data frames)
- Basic functions (`mean()`, `length()`, etc.)

2 Thursday afternoon: `tidyverse` basics I

- Packages: installation and loading (including the tidyverse)
- Data wrangling with dplyr (basic verbs, including the new `.by =` syntax)
- Data visualization basics with `ggplot2`
- Data loading (.csv, .rds, .dta, .xlsx)
- Quarto fundamentals

3 Friday morning: Matrices

- Matrices
- Systems of linear equations
- Matrix operations (multiplication, transpose, inverse, determinant).
- Solving systems of linear equations in matrix form (and why that's cool)
- Introduction to OLS

4 Friday afternoon: `tidyverse` basics II

- Data merging and pivoting (`*_join()`, `pivot_*()`)
- Value recoding (`if_else()`, `case_when()`)
- Missing values
- Data visualization extensions: facets, text annotations

5 Monday morning: Functions and loops

- Functions
- For-loops and `lapply()`
- Finding R help (help files, effective Googling, ChatGPT)

6 Monday afternoon: Calculus

- Limits (not sure how to teach this in an R-centric way yet, but there must be a way)
- Derivatives (symbolic, numerical, automatic)
- Integrals

7 Tuesday morning: Probability

- Concepts: probability, random variables, etc.
- PMF, PDF, CDF, etc.
- Distributions (binomial, normal; different functions in R and how to use them)
- Expectation and variance

8 Tuesday afternoon: Simulations

- Simulations (ideas, seed setting, etc.)
- Sampling
- Bootstrapping

9 Wednesday morning: Text analysis

- String manipulation with `stringr`
- Simple text analysis (counts, tf-idf, etc.) with `tidytext` and visualization

10 Wednesday afternoon: Wrap-up

- Project management fundamentals (RStudio projects, keeping raw data, etc.)
- Self-study resources and materials
- Other software (Overleaf, Zotero, etc.)
- Methods at UT

Contact info

If you have any questions during or outside of methods camp, you can contact us via email:

- Andrés Cruz: andres.cruz@utexas.edu
- Matt Martin: mjmartin@utexas.edu

If you are interested in learning more about our research, you can also check out our respective websites:

- Andrés Cruz via GitHub Pages
- Matt Martin via GitHub Pages

Or, follow us on Twitter (or should we say X…):

- Andrés Cruz: https://twitter.com/arcruz0
- Matt Martin: https://twitter.com/MattJ_Martin

Setup

Installing R and RStudio

R is a programming language optimized for statistics and data analysis. Most people use R from RStudio, a graphical user interface (GUI) that includes a file pane, a graphics pane, and other goodies. Both R and RStudio are open source, i.e., free as in beer and free as in freedom!

Your first steps should be to install R and RStudio, in that order (if you have installed these programs before, make sure that your versions are up-to-date—if they are not, follow the instructions below):

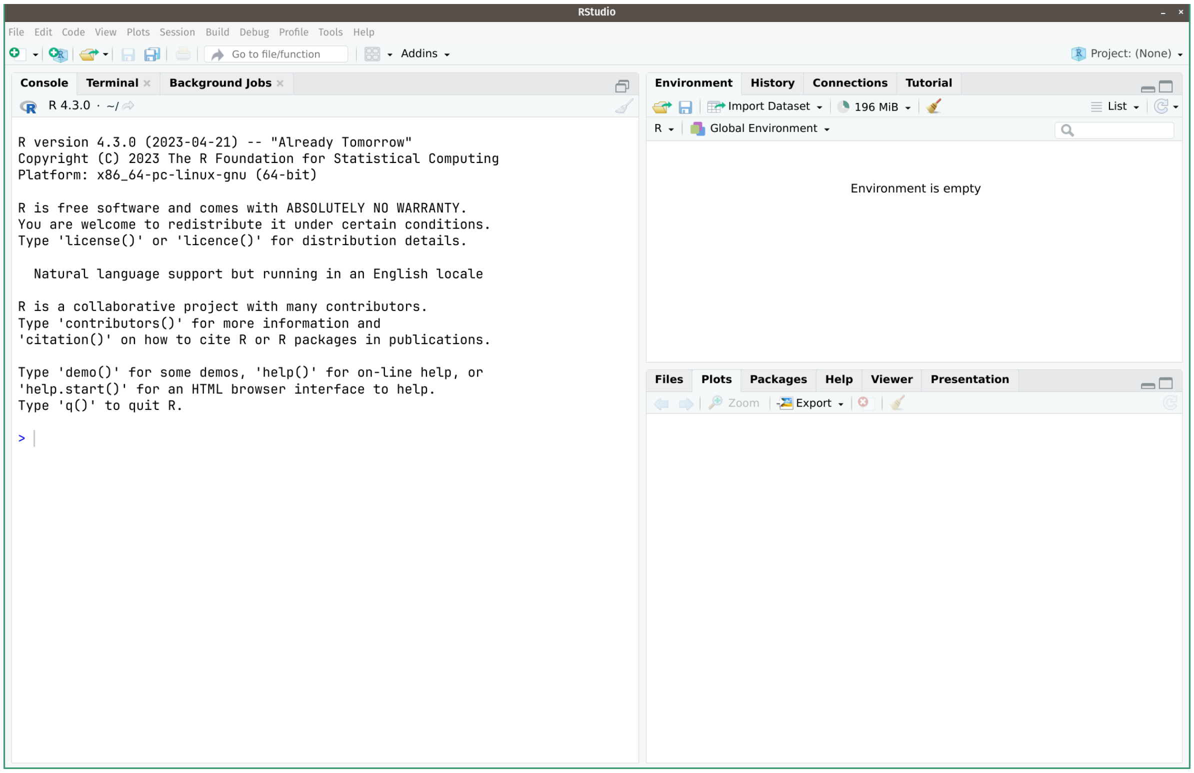
1. Download and install R from the official website, CRAN. Click on "Download R for <Windows/Mac>" and follow the instructions. If you have a Mac, make sure to select the version appropriate for your system (Apple Silicon for newer M1/M2 Macs and Intel for older Macs).
2. Download and install RStudio from the official website. Scroll down and select the installer for your operating system.

After these two steps, you can open RStudio in your system, as you would with any program. You should see something like this:

That's it for the installation! We also *strongly* recommend that you change a couple of RStudio's default settings.[1] You can change settings by clicking on `Tools > Global Options` in the menubar. Here are our recommendations:

- `General > Uncheck "Restore .RData into workspace at startup"`
- `General > Save workspace to .RData on Exit > Select "Never"`
- `Code > Check "Use native pipe operator"`
- `Tools > Global Options > Appearance` to change to a dark theme, if you want! Pros: better for night sessions, hacker vibes…

[1] The idea behind these settings (or at least the first two) is to force R to start from scratch with each new session. No lingering objects from previous coding sessions avoids misunderstandings and helps with reproducibility!



```
R version 4.3.0 (2023-04-21) -- "Already Tomorrow"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Environment is empty

Figure 1: How RStudio looks after a clean installation.

Setting up for Methods Camp

All materials for Methods Camp are both on this website and an RStudio project. An RStudio project is simply a folder where one keeps scripts, datasets, and other files needed for a data analysis project.

You can download our RStudio project here, as a .zip compressed file. On MacOS, the file will be uncompressed automatically. On Windows, you should do `Right click > Extract all`.

> ⚠ Warning
>
> Make sure to properly unzip the materials. Double-clicking the .zip file on most Windows systems *will not* unzip the folder—you must do `Right click > Extract all`.

You should now have a folder called `methodscamp/` on your computer. Navigate to the `methodscamp.Rproj` file within it and open it. RStudio should open the project right away. You should see `methodscamp` on the top-right of RStudio—this indicates that you are working in our RStudio project.

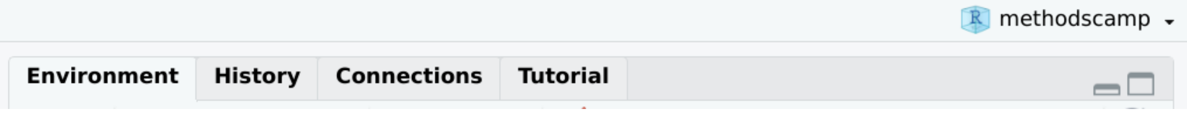


Figure 2: How the bottom-right corner of RStudio looks after opening our project.

That's all for setup! We can now start coding. After opening our RStudio project, we'll begin by opening the `01_r_intro.qmd` file from the "Files" panel, in the bottom-right portion of RStudio. This is a Quarto document,[2] which contains both code and explanations (you can also read in the next chapter of this website).

[2] Perhaps you have used R Markdown before. Quarto is the next iteration of R Markdown, and is both more flexible and more powerful!

1 Intro to R

In Quarto documents like this one, we can write comments by just using plain text. In contrast, code needs to be within *code blocks*, like the one below. To execute a code block, you can click on the little “Play” button or press `Cmd/Ctrl + Shift + Enter` when your keyboard is hovering the code block.

```r
2 + 2
```

```
[1] 4
```

That was our first R command, a simple math operation. Of course, we can also do more complex arithmetic:

```r
12345 ^ 2  / (200 + 25 - 6 * 2) # this is an inline comment, see the leading "#"
```

```
[1] 715488.4
```

In order to *create* a code block, you can press `Cmd/Ctrl + Alt + i` or click on the little green “+C” icon on top of the script.

> ℹ Exercise
>
> Create your own code block below and run a math operation.

1.1 Objects

A huge part of R is working with *objects*. Let’s see how they work:

```r
my_object <- 10 # opt/alt + minus sign will make the arrow
```

```r
my_object # to print the value of an object, just call its name
```

```
[1] 10
```

We can now use this object in our operations:

```
2 ^ my_object
```

```
[1] 1024
```

Or even create another object out of it:

```
my_object2 <- my_object * 2
```

```
my_object2
```

```
[1] 20
```

You can delete objects with the `rm()` function (for "remove"):

```
rm(my_object2)
```

1.2 Vectors and functions

Objects can be of different types. One of the most useful ones is the *vector*, which holds a series of values. To create one manually, we can use the `c()` function (for "combine"):

```
my_vector <- c(6, -11, my_object, 0, 20)
```

```
my_vector
```

```
[1]   6 -11  10   0  20
```

One can also define vectors by sequences:

```
3:10
```

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

We can use square brackets to retrieve parts of vectors:

```r
my_vector[4] # fourth element
```

```
[1] 0
```

```r
my_vector[1:2] # first two elements
```

```
[1]   6 -11
```

Let's check out some basic functions we can use with numbers and numeric vectors:

```r
sqrt(my_object) # squared root
```

```
[1] 3.162278
```

```r
log(my_object) # logarithm (natural by default)
```

```
[1] 2.302585
```

```r
abs(-5) # absolute value
```

```
[1] 5
```

```r
mean(my_vector)
```

```
[1] 5
```

```r
median(my_vector)
```

```
[1] 6
```

```r
sd(my_vector) # standard deviation
```

```
[1] 11.53256
```

```
sum(my_vector)
```

```
[1] 25
```

```
min(my_vector) # minimum value
```

```
[1] -11
```

```
max(my_vector) # maximum value
```

```
[1] 20
```

```
length(my_vector) # length (number of elements)
```

```
[1] 5
```

Notice that if we wanted to save any of these results for later, we would need to *assign* them:

```
my_mean <- mean(my_vector)
```

```
my_mean
```

```
[1] 5
```

These functions are quite simple: they take one object and do one operation. A lot of functions are a bit more complex—they take multiple objects or take options. For example, see the `sort()` function, which by default sorts a vector *increasingly*:

```
sort(my_vector)
```

```
[1] -11   0   6  10  20
```

If we instead want to sort our vector *decreasingly*, we can use the `decreasing = TRUE` argument (`T` also works as an abbreviation for `TRUE`).

```r
sort(my_vector, decreasing = TRUE)
```

```
[1]  20  10   6   0 -11
```

Tip

If you use the argument values in order, you can avoid writing the argument names (see below). This is sometimes useful, but can also lead to confusing code—use it with caution.

```r
sort(my_vector, T)
```

```
[1]  20  10   6   0 -11
```

A useful function to create vectors in sequence is `seq()`. Notice its arguments:

```r
seq(from = 30, to = 100, by = 5)
```

```
 [1]  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
```

To check the arguments of a function, you can examine its help file: look the function up on the "Help" panel on RStudio or use a command like the following: `?sort`.

Exercise

Examine the help file of the `log()` function. How can we compute the the base-10 logarithm of `my_object`? Your code:

Other than numeric vectors, character vectors are also useful:

```r
my_character_vector <- c("Apple", "Orange", "Watermelon", "Banana")
```

```r
my_character_vector[3]
```

```
[1] "Watermelon"
```

```r
nchar(my_character_vector) # count number of characters
```

```
[1]  5  6 10  6
```

1.3 Data frames and lists

Another useful object type is the *data frame*. Data frames can store multiple vectors in a tabular format. We can manually create one with the `data.frame()` function:

```r
my_data_frame <- data.frame(fruit = my_character_vector,
                            calories_per_100g = c(52, 47, 30, 89),
                            water_per_100g = c(85.6, 86.8, 91.4, 74.9))
```

```r
my_data_frame
```

```
       fruit calories_per_100g water_per_100g
1      Apple                52           85.6
2     Orange                47           86.8
3 Watermelon                30           91.4
4     Banana                89           74.9
```

Now we have a little 4x3 data frame of fruits with their calorie counts and water composition. We gathered the nutritional information from the USDA (2019).

We can use the `data_frame$column` construct to access the vectors within the data frame:

```r
mean(my_data_frame$calories_per_100g)
```

```
[1] 54.5
```

> Exercise
>
> Obtain the maximum value of water content per 100g in the data. Your code:

Some useful commands to learn attributes of our data frame:

```r
dim(my_data_frame)
```

```
[1] 4 3
```

```r
nrow(my_data_frame)
```

```
[1] 4
```

```r
names(my_data_frame) # column names
```

```
[1] "fruit"             "calories_per_100g" "water_per_100g"
```

We will learn much more about data frames in our next module on data analysis.

After talking about vectors and data frames, the last object type that we will cover is the *list*. Lists are super flexible objects that can contain just about anything:

```r
my_list <- list(my_object, my_vector, my_data_frame)
```

```r
my_list
```

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

[[2]]
[1]   6 -11  10   0  20

[[3]]
       fruit calories_per_100g water_per_100g
1      Apple                52           85.6
2     Orange                47           86.8
3 Watermelon                30           91.4
4     Banana                89           74.9
```

To retrieve the elements of a list, we need to use double square brackets:

```r
my_list[[1]]
```

```
[1] 10
```

Lists are sometimes useful due to their flexibility, but are much less common in routine data analysis compared to vectors or data frames.

1.4 Packages

The R community has developed thousands of *packages*, which are specialized collections of functions, datasets, and other resources. To install one, you should use the `install.packages()` command. Below we will install the `tidyverse` package, a suite for data analysis that we will use in the next modules. You just need to install packages once, and then they will be available system-wide.

```r
install.packages("tidyverse") # this can take a couple of minutes
```

If you want to use an installed package in your script, you must load it with the `library()` function. Some packages, as shown below, will print descriptive messages once loaded.

```r
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.2     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.2     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

⚠ Warning

Remember that `install.packages("package")` needs to be executed just once, while `library(package)` needs to be in each script in which you plan to use the package. In general, never include `install.packages("package")` as part of your scripts or Quarto documents!

2 Tidy data analysis I

The tidyverse is a suite of packages that streamline data analysis in R. After installing the tidyverse with install.packages("tidyverse") (see the previous module), you can load it with:

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.2     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.2     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

Tip

Upon loading, the tidyverse prints a message like the one above. Notice that multiple packages (the constituent elements of the "suite") are actually loaded. For instance, dplyr and tidyr help with data wrangling and transformation, while ggplot2 allows us to draw plots. In most cases, one just loads the tidyverse and forgets about these details, as the constituent packages work together nicely.

Throughout this module, we will use tidyverse functions to load, wrangle, and visualize real data.

2.1 Loading data

Throughout this module we will work with a dataset of senators during the Trump presidency, which was adapted from FiveThirtyEight (2021).

We have stored the dataset in .csv format under the `data/` subfolder. Loading it into R is simple (notice that we need to assign it to an object):

```r
trump_scores <- read_csv("data/trump_scores_538.csv")
```

```
Rows: 122 Columns: 8
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (4): bioguide, last_name, state, party
dbl (4): num_votes, agree, agree_pred, margin_trump

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.
```

```r
trump_scores
```

```
# A tibble: 122 x 8
   bioguide last_name  state party num_votes agree agree_pred margin_trump
   <chr>    <chr>      <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 A000360  Alexander  TN    R           118 0.890      0.856       26.0
 2 B000575  Blunt      MO    R           128 0.906      0.787       18.6
 3 B000944  Brown      OH    D           128 0.258      0.642        8.13
 4 B001135  Burr       NC    R           121 0.893      0.560        3.66
 5 B001230  Baldwin    WI    D           128 0.227      0.510        0.764
 6 B001236  Boozman    AR    R           129 0.915      0.851       26.9
 7 B001243  Blackburn  TN    R           131 0.885      0.889       26.0
 8 B001261  Barrasso   WY    R           129 0.891      0.895       46.3
 9 B001267  Bennet     CO    D           121 0.273      0.417       -4.91
10 B001277  Blumenthal CT    D           128 0.203      0.294      -13.6
# i 112 more rows
```

Let's review the dataset's columns:

- `bioguide`: A unique ID for each politician, from the Congress Bioguide.
- `last_name`
- `state`
- `party`
- `num_votes`: Number of votes for which data was available.
- `agree`: Proportion (0-1) of votes in which the senator voted in agreement with Trump.
- `agree_pred`: Predicted proportion of vote agreement, calculated using Trump's margin (see next variable).

- `margin_trump`: Margin of victory (percentage points) of Trump in the senator's state.

We can inspect our data by using the interface above. An alternative is to run the command `View(trump_scores)` or click on the object in RStudio's environment panel (in the top-right section).

Do you have any questions about the data?

By the way, the `tidyverse` works amazingly with *tidy data*. If you can get your data to this format (and we will see ways to do this), your life will be much easier:

2.2 Wrangling data with `dplyr`

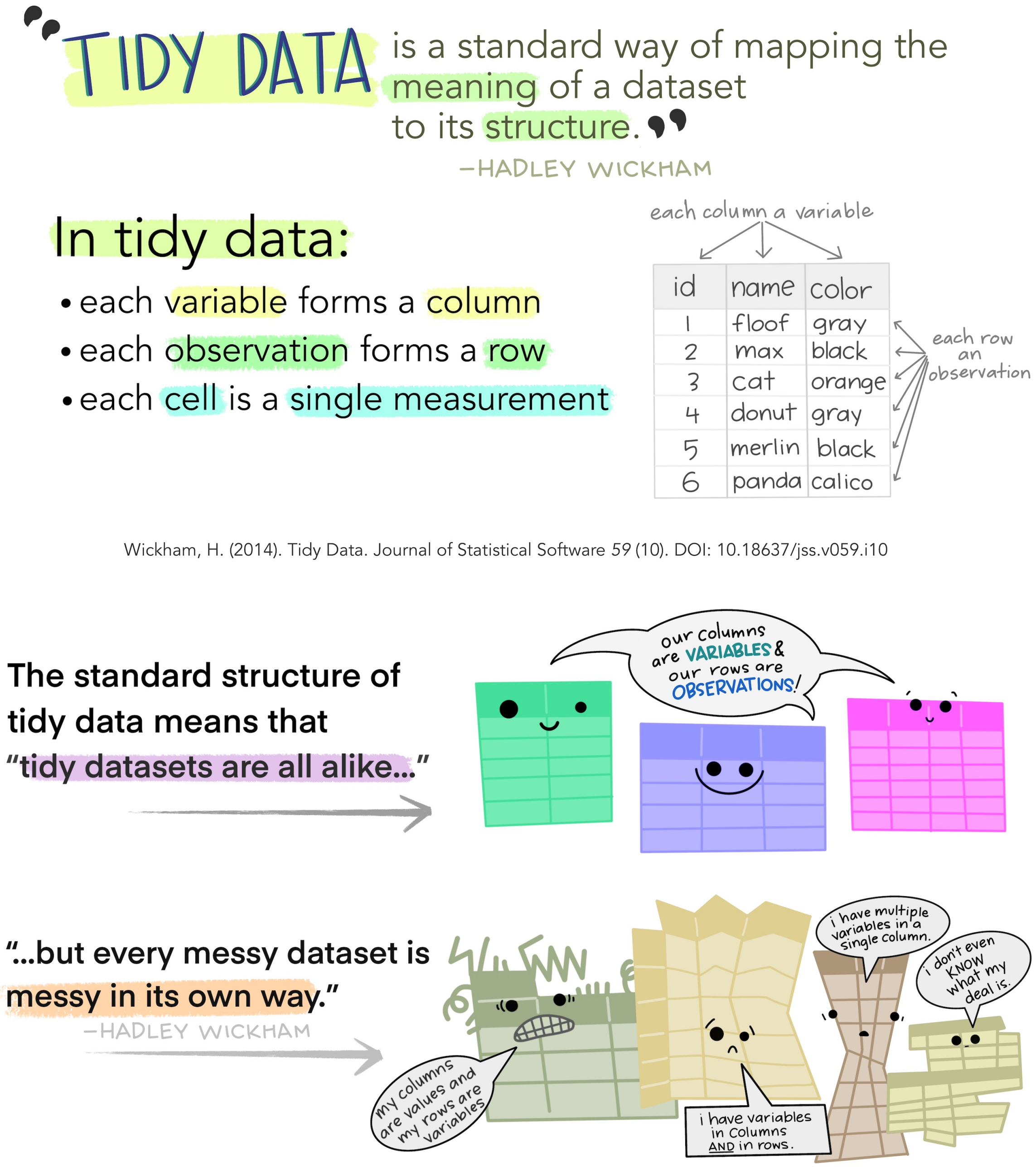
We often need to modify data to conduct our analyses, e.g., creating columns, filtering rows, etc. In the `tidyverse`, these operations are conducted with multiple *verbs*, which we will review now.

2.2.1 Selecting columns

We can select specific columns in our dataset with the `select()` function. All `dplyr` wrangling verbs take a data frame as their first argument—in this case, the columns we want to select are the other arguments.

```
select(trump_scores, last_name, party)
```

```
# A tibble: 122 x 2
   last_name  party
   <chr>      <chr>
 1 Alexander  R
 2 Blunt      R
 3 Brown      D
 4 Burr       R
 5 Baldwin    D
 6 Boozman    R
 7 Blackburn  R
 8 Barrasso   R
 9 Bennet     D
10 Blumenthal D
# i 112 more rows
```


"TIDY DATA is a standard way of mapping the meaning of a dataset to its structure."
—HADLEY WICKHAM
In tidy data:
• each variable forms a column
• each observation forms a row
• each cell is a single measurement
each column a variable
each row an observation

id	name	color
1	floof	gray
2	max	black
3	cat	orange
4	donut	gray
5	merlin	black
6	panda	calico

Wickham, H. (2014). Tidy Data. Journal of Statistical Software *59* (10). DOI: 10.18637/jss.v059.i10

The standard structure of tidy data means that "tidy datasets are all alike..."
our columns are VARIABLES & our rows are OBSERVATIONS!
"...but every messy dataset is messy in its own way."
—HADLEY WICKHAM
my columns are values and my rows are variables
i have variables in columns AND in rows.
i have multiple variables in a single column.
i don't even KNOW what my deal is.

Figure 2.1: Source: Illustrations from the Openscapes blog *Tidy Data for reproducibility, efficiency, and collaboration* by Julia Lowndes and Allison Horst.

This is a good moment to talk about "pipes." Notice how the code below produces the same output as the one above, but with a slightly different syntax. Pipes (`|>`) "kick" the object on the left of the pipe to the first argument of the function on the right. One can read pipes as "then," so the code below can be read as "take `trump_scores`, then select the columns `last_name` and `party`." Pipes are very useful to *chain multiple operations*, as we will see in a moment.

```
trump_scores |>
  select(last_name, party)
```

```
# A tibble: 122 x 2
   last_name  party
   <chr>      <chr>
 1 Alexander  R
 2 Blunt      R
 3 Brown      D
 4 Burr       R
 5 Baldwin    D
 6 Boozman    R
 7 Blackburn  R
 8 Barrasso   R
 9 Bennet     D
10 Blumenthal D
# i 112 more rows
```

> Tip
>
> You can insert a pipe with the `Cmd/Ctrl + Shift + M` shortcut. If you have not changed the default RStudio settings, an "old" pipe (`%>%`) might appear. While most of the functionality is the same, the `|>` "new" pipes are more readable. You can change this RStudio option in `Tools > Global Options > Code > Use native pipe operator`. Make sure to check the other suggested settings in our Setup module!

Going back to selecting columns, you can select ranges:

```
trump_scores |>
  select(bioguide:party)
```

```
# A tibble: 122 x 4
   bioguide last_name  state party
   <chr>    <chr>      <chr> <chr>
```

```
 1 A000360  Alexander  TN    R
 2 B000575  Blunt      MO    R
 3 B000944  Brown      OH    D
 4 B001135  Burr       NC    R
 5 B001230  Baldwin    WI    D
 6 B001236  Boozman    AR    R
 7 B001243  Blackburn  TN    R
 8 B001261  Barrasso   WY    R
 9 B001267  Bennet     CO    D
10 B001277  Blumenthal CT    D
# i 112 more rows
```

You can also **de**select columns using a minus sign:

```
trump_scores |>
  select(-last_name)
```

```
# A tibble: 122 x 7
   bioguide state party num_votes agree agree_pred margin_trump
   <chr>    <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 A000360  TN    R           118 0.890      0.856       26.0
 2 B000575  MO    R           128 0.906      0.787       18.6
 3 B000944  OH    D           128 0.258      0.642        8.13
 4 B001135  NC    R           121 0.893      0.560        3.66
 5 B001230  WI    D           128 0.227      0.510        0.764
 6 B001236  AR    R           129 0.915      0.851       26.9
 7 B001243  TN    R           131 0.885      0.889       26.0
 8 B001261  WY    R           129 0.891      0.895       46.3
 9 B001267  CO    D           121 0.273      0.417       -4.91
10 B001277  CT    D           128 0.203      0.294      -13.6
# i 112 more rows
```

And use a few helper functions, like `matches()`:

```
trump_scores |>
  select(last_name, matches("agree"))
```

```
# A tibble: 122 x 3
   last_name  agree agree_pred
   <chr>      <dbl>      <dbl>
```

```
 1 Alexander  0.890      0.856
 2 Blunt      0.906      0.787
 3 Brown      0.258      0.642
 4 Burr       0.893      0.560
 5 Baldwin    0.227      0.510
 6 Boozman    0.915      0.851
 7 Blackburn  0.885      0.889
 8 Barrasso   0.891      0.895
 9 Bennet     0.273      0.417
10 Blumenthal 0.203      0.294
# i 112 more rows
```

Or `everything()`, which we usually use to reorder columns:

```r
trump_scores |>
  select(last_name, everything())
```

```
# A tibble: 122 x 8
   last_name  bioguide state party num_votes agree agree_pred margin_trump
   <chr>      <chr>    <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 Alexander  A000360  TN    R           118 0.890      0.856       26.0
 2 Blunt      B000575  MO    R           128 0.906      0.787       18.6
 3 Brown      B000944  OH    D           128 0.258      0.642        8.13
 4 Burr       B001135  NC    R           121 0.893      0.560        3.66
 5 Baldwin    B001230  WI    D           128 0.227      0.510        0.764
 6 Boozman    B001236  AR    R           129 0.915      0.851       26.9
 7 Blackburn  B001243  TN    R           131 0.885      0.889       26.0
 8 Barrasso   B001261  WY    R           129 0.891      0.895       46.3
 9 Bennet     B001267  CO    D           121 0.273      0.417       -4.91
10 Blumenthal B001277  CT    D           128 0.203      0.294      -13.6
# i 112 more rows
```

Tip

Notice that all these commands have not edited our existent objects—they have just printed the requested outputs to the screen. In order to modify objects, you need to use the assignment operator (`<-`). For example:

```r
trump_scores_reduced <- trump_scores |>
  select(last_name, matches("agree"))
```

```
trump_scores_reduced
```

```
# A tibble: 122 x 3
   last_name  agree agree_pred
   <chr>      <dbl>      <dbl>
 1 Alexander  0.890      0.856
 2 Blunt      0.906      0.787
 3 Brown      0.258      0.642
 4 Burr       0.893      0.560
 5 Baldwin    0.227      0.510
 6 Boozman    0.915      0.851
 7 Blackburn  0.885      0.889
 8 Barrasso   0.891      0.895
 9 Bennet     0.273      0.417
10 Blumenthal 0.203      0.294
# i 112 more rows
```

ℹ Exercise

Select the variables `last_name`, `party`, `num_votes`, and `agree` from the data frame. Your code:

2.2.2 Renaming columns

We can use the `rename()` function to rename columns, with the syntax `new_name = old_name`. For example:

```
trump_scores |>
  rename(prop_agree = agree, prop_agree_pred = agree_pred)
```

```
# A tibble: 122 x 8
   bioguide last_name  state party num_votes prop_agree prop_agree_pred
   <chr>    <chr>      <chr> <chr>     <dbl>      <dbl>           <dbl>
 1 A000360  Alexander  TN    R           118      0.890           0.856
 2 B000575  Blunt      MO    R           128      0.906           0.787
 3 B000944  Brown      OH    D           128      0.258           0.642
 4 B001135  Burr       NC    R           121      0.893           0.560
 5 B001230  Baldwin    WI    D           128      0.227           0.510
 6 B001236  Boozman    AR    R           129      0.915           0.851
 7 B001243  Blackburn  TN    R            131       0.885            0.889
 8 B001261  Barrasso   WY    R            129       0.891            0.895
 9 B001267  Bennet     CO    D            121       0.273            0.417
10 B001277  Blumenthal CT    D            128       0.203            0.294
# i 112 more rows
# i 1 more variable: margin_trump <dbl>
```

This is a good occasion to show how pipes allow us to chain operations. How do we read the following code out loud? (Remember that pipes are read as "then").

```
trump_scores |>
  select(last_name, matches("agree")) |>
  rename(prop_agree = agree, prop_agree_pred = agree_pred)
```

```
# A tibble: 122 x 3
   last_name  prop_agree prop_agree_pred
   <chr>           <dbl>           <dbl>
 1 Alexander       0.890           0.856
 2 Blunt           0.906           0.787
 3 Brown           0.258           0.642
 4 Burr            0.893           0.560
 5 Baldwin         0.227           0.510
 6 Boozman         0.915           0.851
 7 Blackburn       0.885           0.889
 8 Barrasso        0.891           0.895
 9 Bennet          0.273           0.417
10 Blumenthal      0.203           0.294
# i 112 more rows
```

2.2.3 Creating columns

It is common to want to create columns, based on existing ones. We can use `mutate()` to do so. For example, we could want our main variables of interest in terms of percentages instead of proportions:

```
trump_scores |>
  select(last_name, agree, agree_pred) |> # select just for clarity
  mutate(pct_agree = 100 * agree,
         pct_agree_pred = 100 * agree_pred)
```

```
# A tibble: 122 x 5
   last_name  agree agree_pred pct_agree pct_agree_pred
   <chr>      <dbl>      <dbl>     <dbl>          <dbl>
 1 Alexander  0.890      0.856      89.0           85.6
 2 Blunt      0.906      0.787      90.6           78.7
 3 Brown      0.258      0.642      25.8           64.2
 4 Burr       0.893      0.560      89.3           56.0
 5 Baldwin    0.227      0.510      22.7           51.0
 6 Boozman    0.915      0.851      91.5           85.1
 7 Blackburn  0.885      0.889      88.5           88.9
 8 Barrasso   0.891      0.895      89.1           89.5
 9 Bennet     0.273      0.417      27.3           41.7
10 Blumenthal 0.203      0.294      20.3           29.4
# i 112 more rows
```

We can also use multiple columns for creating a new one. For example, let's retrieve the total *number* of votes in which the senator agreed with Trump:

```
trump_scores |>
  select(last_name, num_votes, agree) |> # select just for clarity
  mutate(num_votes_agree = num_votes * agree)
```

```
# A tibble: 122 x 4
   last_name  num_votes agree num_votes_agree
   <chr>          <dbl> <dbl>           <dbl>
 1 Alexander        118 0.890           105
 2 Blunt            128 0.906           116
 3 Brown            128 0.258            33
 4 Burr             121 0.893           108
 5 Baldwin          128 0.227            29
 6 Boozman          129 0.915           118
 7 Blackburn        131 0.885           116
 8 Barrasso         129 0.891           115
 9 Bennet           121 0.273            33.0
10 Blumenthal       128 0.203            26
# i 112 more rows
```

2.2.4 Filtering rows

Another common operation is to filter rows based on logical conditions. We can do so with the `filter()` function. For example, we can filter to only get Democrats:

```
trump_scores |>
  filter(party == "D")
```

```
# A tibble: 55 x 8
   bioguide last_name  state party num_votes agree agree_pred margin_trump
   <chr>    <chr>      <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 B000944  Brown      OH    D           128 0.258      0.642        8.13 
 2 B001230  Baldwin    WI    D           128 0.227      0.510        0.764
 3 B001267  Bennet     CO    D           121 0.273      0.417       -4.91 
 4 B001277  Blumenthal CT    D           128 0.203      0.294      -13.6  
 5 B001288  Booker     NJ    D           119 0.160      0.290      -14.1  
 6 C000127  Cantwell   WA    D           128 0.242      0.276      -15.5  
 7 C000141  Cardin     MD    D           128 0.25       0.209      -26.4  
 8 C000174  Carper     DE    D           129 0.295      0.318      -11.4  
 9 C001070  Casey      PA    D           129 0.287      0.508        0.724
10 C001088  Coons      DE    D           128 0.289      0.319      -11.4  
# i 45 more rows
```

Notice that `==` here is a *logical operator*, read as "is equal to." So our full chain of operations says the following: take `trump_scores`, then filter it to get rows where party is equal to "D".

There are other logical operators:

Logical operator	Meaning
`==`	"is equal to"
`!=`	"is not equal to"
`>`	"is greater than"
`<`	"is less than"
`>=`	"is greater than or equal to"
`<=`	"is less than or equal to"
`%in%`	"is contained in"
`&`	"and" (intersection)
`|`	"or" (union)

Let's see a couple of other examples.

```
trump_scores |>
  filter(agree > 0.5)
```

```
# A tibble: 69 x 8
   bioguide last_name state party num_votes agree agree_pred margin_trump
   <chr>    <chr>     <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 A000360  Alexander TN    R           118 0.890      0.856        26.0
 2 B000575  Blunt     MO    R           128 0.906      0.787        18.6
 3 B001135  Burr      NC    R           121 0.893      0.560         3.66
 4 B001236  Boozman   AR    R           129 0.915      0.851        26.9
 5 B001243  Blackburn TN    R           131 0.885      0.889        26.0
 6 B001261  Barrasso  WY    R           129 0.891      0.895        46.3
 7 B001310  Braun     IN    R            44 0.909      0.713        19.2
 8 C000567  Cochran   MS    R            68 0.971      0.830        17.8
 9 C000880  Crapo     ID    R           125 0.904      0.870        31.8
10 C001035  Collins   ME    R           129 0.651      0.441        -2.96
# i 59 more rows
```

```r
trump_scores |>
  filter(state %in% c("CA", "TX"))
```

```
# A tibble: 4 x 8
  bioguide last_name state party num_votes agree agree_pred margin_trump
  <chr>    <chr>     <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
1 C001056  Cornyn    TX    R           129 0.922      0.659         9.00
2 C001098  Cruz      TX    R           126 0.921      0.663         9.00
3 F000062  Feinstein CA    D           128 0.242      0.201       -30.1
4 H001075  Harris    CA    D           116 0.164      0.209       -30.1
```

```r
trump_scores |>
  filter(state == "WV" & party == "D")
```

```
# A tibble: 1 x 8
  bioguide last_name state party num_votes agree agree_pred margin_trump
  <chr>    <chr>     <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
1 M001183  Manchin   WV    D           129 0.504      0.893         42.2
```

ℹ Exercise

1. Add a new column to the data frame, called `diff_agree`, which subtracts `agree` and `agree_pred`. How would you create `abs_diff_agree`, defined as the absolute value of `diff_agree`? Your code:

2. Filter the data frame to only get senators for which we have information on fewer than (or equal to) five votes. Your code:
3. Filter the data frame to only get Democrats who agreed with Trump in at least 30% of votes. Your code:

2.2.5 Ordering rows

The `arrange()` function allows us to order rows according to values. For example, let's order based on the `agree` variable:

```
trump_scores |>
  arrange(agree)
```

```
# A tibble: 122 x 8
   bioguide last_name    state party num_votes agree agree_pred margin_trump
   <chr>    <chr>        <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 H000273  Hickenlooper CO    D             2 0         0.0302        -4.91
 2 H000601  Hagerty      TN    R             2 0         0.115         26.0
 3 L000570  Luján        NM    D           186 0.124     0.243         -8.21
 4 G000555  Gillibrand   NY    D           121 0.124     0.242        -22.5
 5 M001176  Merkley      OR    D           129 0.155     0.323        -11.0
 6 W000817  Warren       MA    D           116 0.155     0.216        -27.2
 7 B001288  Booker       NJ    D           119 0.160     0.290        -14.1
 8 S000033  Sanders      VT    D           112 0.161     0.221        -26.4
 9 H001075  Harris       CA    D           116 0.164     0.209        -30.1
10 M000133  Markey       MA    D           127 0.165     0.213        -27.2
# i 112 more rows
```

Maybe we only want senators with more than a few data points. Remember that we can chain operations:

```
trump_scores |>
  filter(num_votes >= 10) |>
  arrange(agree)
```

```
# A tibble: 115 x 8
   bioguide last_name  state party num_votes agree agree_pred margin_trump
   <chr>    <chr>      <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 L000570  Luján      NM    D           186 0.124      0.243        -8.21
 2 G000555  Gillibrand NY    D             121 0.124      0.242        -22.5 
 3 M001176  Merkley    OR    D             129 0.155      0.323        -11.0 
 4 W000817  Warren     MA    D             116 0.155      0.216        -27.2 
 5 B001288  Booker     NJ    D             119 0.160      0.290        -14.1 
 6 S000033  Sanders    VT    D             112 0.161      0.221        -26.4 
 7 H001075  Harris     CA    D             116 0.164      0.209        -30.1 
 8 M000133  Markey     MA    D             127 0.165      0.213        -27.2 
 9 W000779  Wyden      OR    D             129 0.186      0.323        -11.0 
10 B001277  Blumenthal CT    D             128 0.203      0.294        -13.6 
# i 105 more rows
```

By default, `arrange()` uses increasing order (like `sort()`). To use decreasing order, add a minus sign:

```
trump_scores |>
  filter(num_votes >= 10) |>
  arrange(-agree)
```

```
# A tibble: 115 x 8
   bioguide last_name state party num_votes agree agree_pred margin_trump
   <chr>    <chr>     <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 M001198  Marshall  KS    R           183 0.973      0.933        20.6 
 2 C000567  Cochran   MS    R            68 0.971      0.830        17.8 
 3 H000338  Hatch     UT    R            84 0.964      0.825        18.1 
 4 M001197  McSally   AZ    R           136 0.949      0.562         3.55
 5 P000612  Perdue    GA    R           119 0.941      0.606         5.16
 6 C001096  Cramer    ND    R           135 0.941      0.908        35.7 
 7 R000307  Roberts   KS    R           127 0.937      0.818        20.6 
 8 C001056  Cornyn    TX    R           129 0.922      0.659         9.00
 9 H001061  Hoeven    ND    R           129 0.922      0.883        35.7 
10 C001047  Capito    WV    R           127 0.921      0.896        42.2 
# i 105 more rows
```

You can also order rows by more than one variable. What this does is to order by the first variable, and resolve any ties by ordering by the second variable (and so forth if you have more than two ordering variables). For example, let's first order our data frame by party, and then within party order by agreement with Trump:

```
trump_scores |>
  filter(num_votes >= 10) |>
  arrange(party, agree)
```

```
# A tibble: 115 x 8
   bioguide last_name  state party num_votes agree agree_pred margin_trump
   <chr>    <chr>      <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 L000570  Luján      NM    D           186 0.124      0.243        -8.21
 2 G000555  Gillibrand NY    D           121 0.124      0.242       -22.5 
 3 M001176  Merkley    OR    D           129 0.155      0.323       -11.0 
 4 W000817  Warren     MA    D           116 0.155      0.216       -27.2 
 5 B001288  Booker     NJ    D           119 0.160      0.290       -14.1 
 6 S000033  Sanders    VT    D           112 0.161      0.221       -26.4 
 7 H001075  Harris     CA    D           116 0.164      0.209       -30.1 
 8 M000133  Markey     MA    D           127 0.165      0.213       -27.2 
 9 W000779  Wyden      OR    D           129 0.186      0.323       -11.0 
10 B001277  Blumenthal CT    D           128 0.203      0.294       -13.6 
# i 105 more rows
```

Exercise

Arrange the data by `diff_pred`, the difference between agreement and predicted agreement with Trump. (You should have code on how to create this variable from the last exercise). Your code:

2.2.6 Summarizing data

`dplyr` makes summarizing data a breeze using the `summarize()` function:

```
trump_scores |>
  summarize(mean_agree = mean(agree),
            mean_agree_pred = mean(agree_pred))
```

```
# A tibble: 1 x 2
  mean_agree mean_agree_pred
       <dbl>           <dbl>
1      0.592           0.572
```

To make summaries, we can use any function that takes a vector and returns one value. Another example:

```
trump_scores |>
  filter(num_votes >= 5) |> # to filter out senators with few data points
  summarize(max_agree = max(agree),
            min_agree = min(agree))
```

```
# A tibble: 1 x 2
  max_agree min_agree
      <dbl>     <dbl>
1         1     0.124
```

Grouped summaries allow us to disaggregate summaries according to other variables (usually categorical):

```
trump_scores |>
  filter(num_votes >= 5) |> # to filter out senators with few data points
  summarize(mean_agree = mean(agree),
            max_agree = max(agree),
            min_agree = min(agree),
            .by = party) # to group by party
```

```
# A tibble: 2 x 4
  party mean_agree max_agree min_agree
  <chr>      <dbl>     <dbl>     <dbl>
1 R          0.876     1         0.651
2 D          0.272     0.548     0.124
```

> **Exercise**
>
> Obtain the maximum absolute difference in agreement with Trump (the `abs_diff_agree` variable from before) for each party.

2.2.7 Overview

Function	Purpose
`select()`	Select columns
`rename()`	Rename columns
`mutate()`	Creating columns
`filter()`	Filtering rows
`arrange()`	Ordering rows
`summarize()`	Summarizing data
`summarize(…, .by = )`	Summarizing data (by groups)

Function	Purpose

2.3 Visualizing data with ggplot2

ggplot2 is the package in charge of data visualization in the tidyverse. It is extremely flexible and allows us to draw bar plots, box plots, histograms, scatter plots, and many other types of plots (see examples at R Charts).

Throughout this module we will use a subset of our data frame, which only includes senators with more than a few data points:

```r
trump_scores_ss <- trump_scores |>
  filter(num_votes >= 10)
```

The ggplot2 syntax provides a unifying interface (the “grammar of graphics” or “gg”) for drawing all different types of plots. One draws plots by adding different “layers,” and the core code always includes the following:

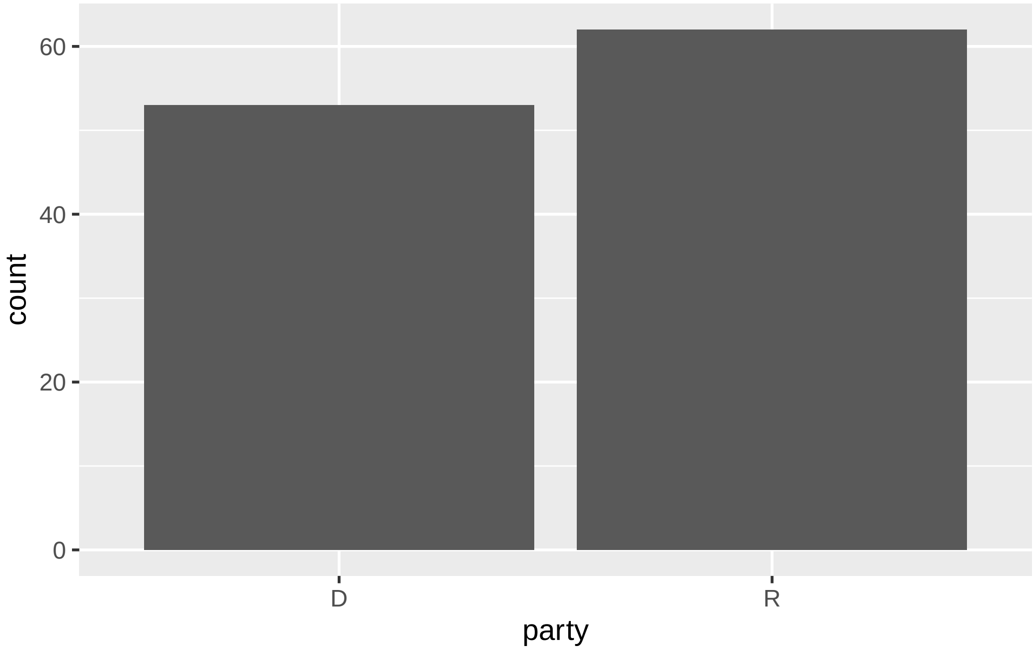
- A ggplot() command with a data = argument specifying a data frame and a mapping = aes() argument specifying “aesthetic mappings,” i.e., how we want to use the columns in the data frame in the plot (for example, in the x-axis, as color, etc.).
- “geoms,” such as geom_bar() or geom_point(), specifying what to draw on the plot.

So *all* ggplot2 commands will have at least three elements: data, aesthetic mappings, and geoms.

2.3.1 Univariate plots: categorical

Let’s see an example of a bar plot with a categorical variable:

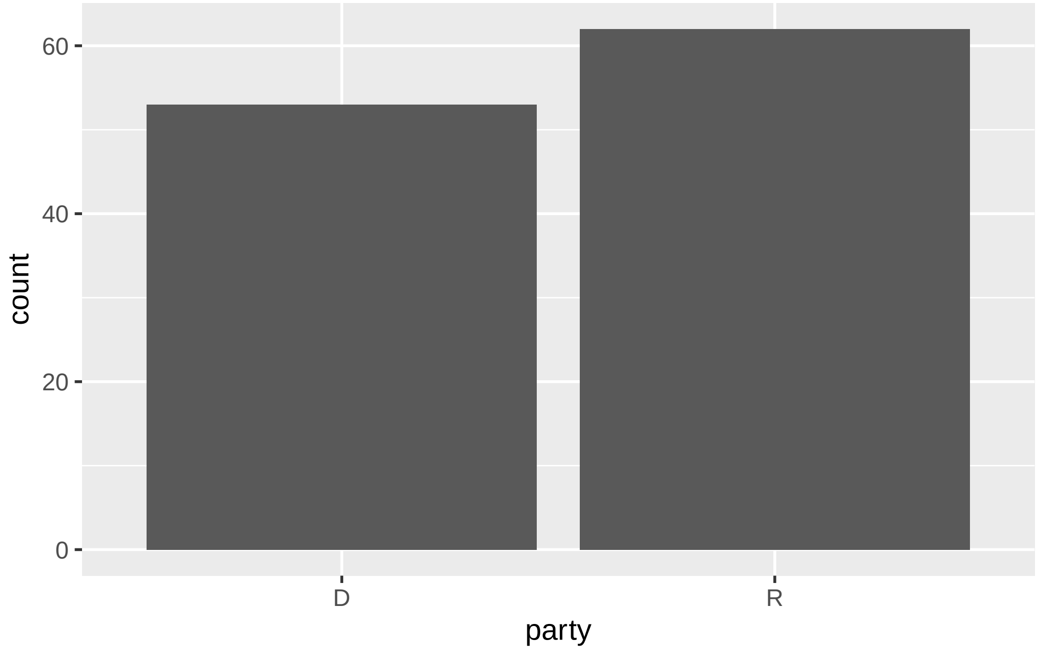
```r
ggplot(data = trump_scores_ss, mapping = aes(x = party)) +
  geom_bar()
```



Tip

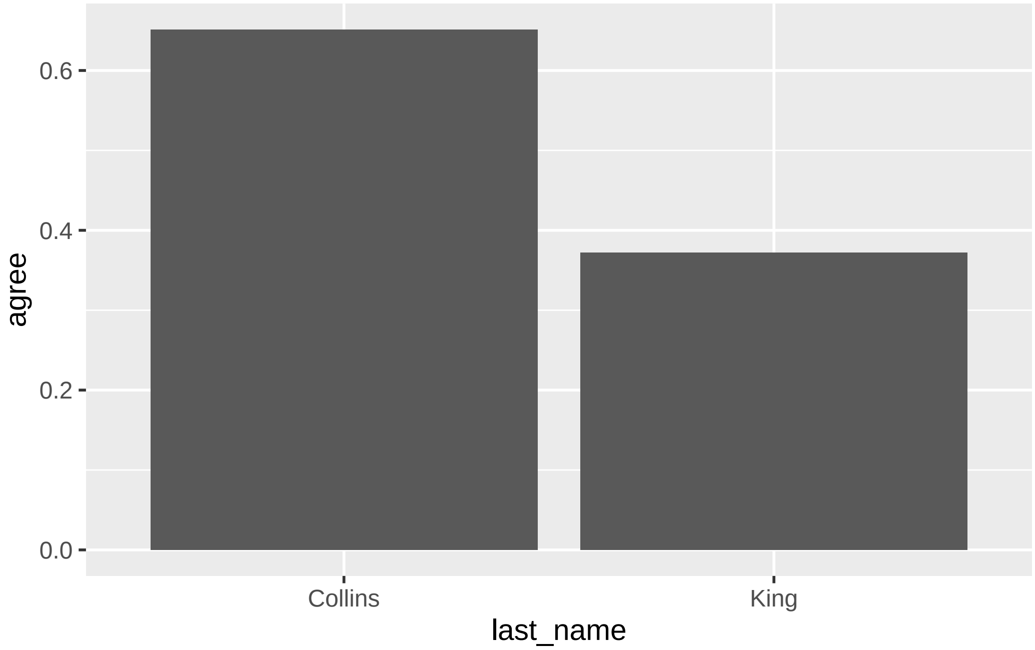
As with any other function, we can drop the argument names if we specify the argument values in order. This is common in ggplot2 code:

```
ggplot(trump_scores_ss, aes(x = party)) +
  geom_bar()
```



Notice how `geom_bar()` automatically computes the number of observations in each category for us. Sometimes we want to use numbers in our data frame as part of a bar plot. Here we can use the `geom_col()` geom specifying both `x` and `y` aesthetic mappings, in which is sometimes called a "column plot:"

```
ggplot(trump_scores_ss |> filter(state == "ME"),
       aes(x = last_name, y = agree)) +
  geom_col()
```



ℹ Exercise

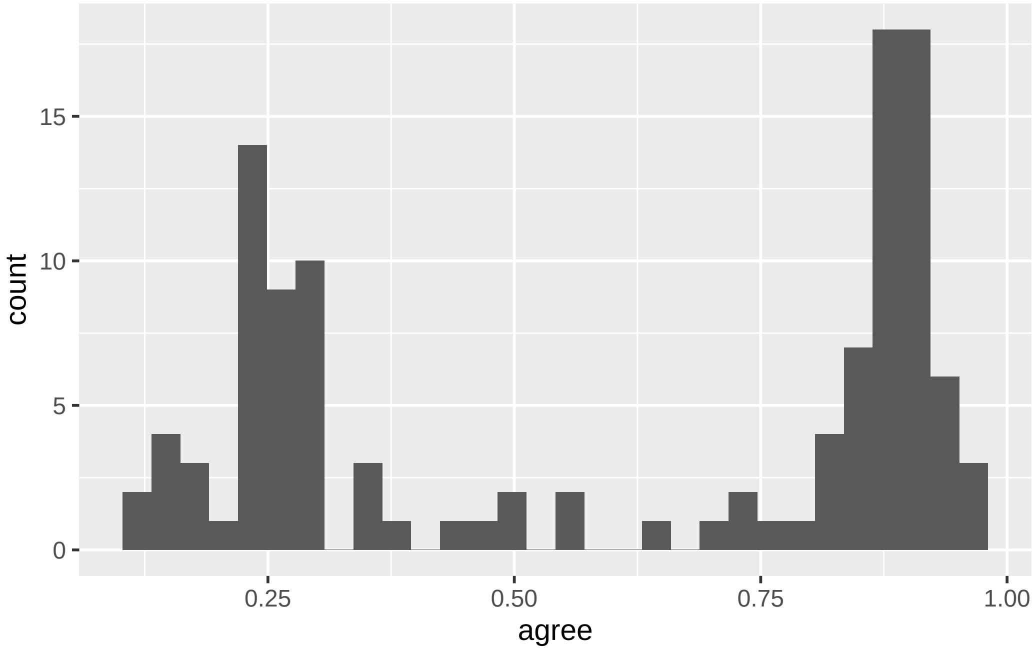
Draw a column plot with the agreement with Trump of Bernie Sanders and Ted Cruz. What happens if you use `last_name` as the `y` aesthetic mapping and `agree` in the `x` aesthetic mapping? Your code:

2.3.2 Univariate plots: numerical

We can draw a histogram with `geom_histogram()`:

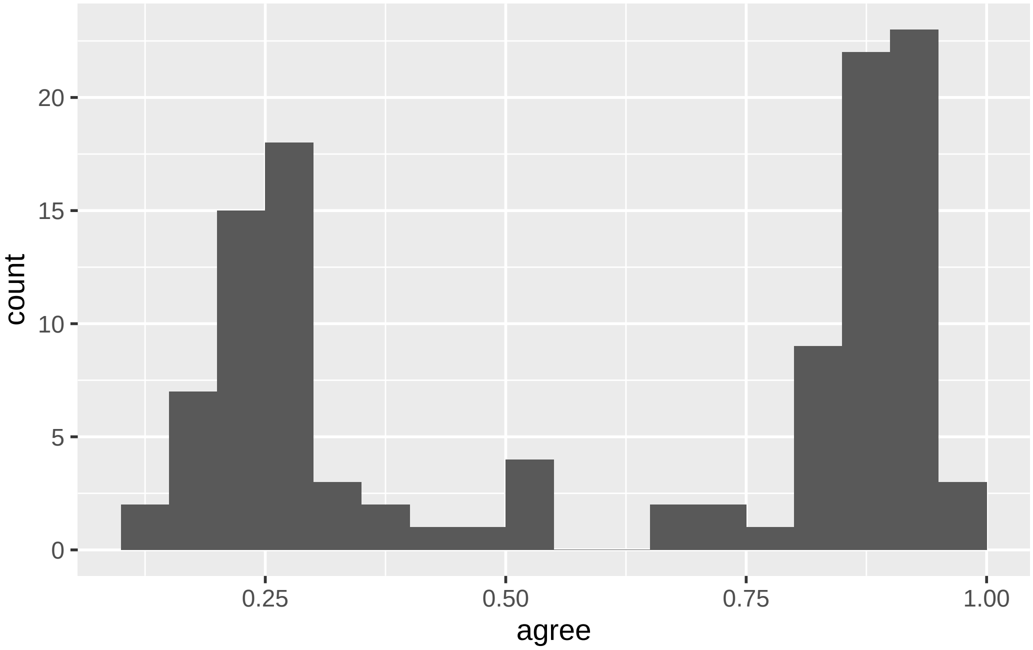
```
ggplot(trump_scores_ss, aes(x = agree)) +
  geom_histogram()
```

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



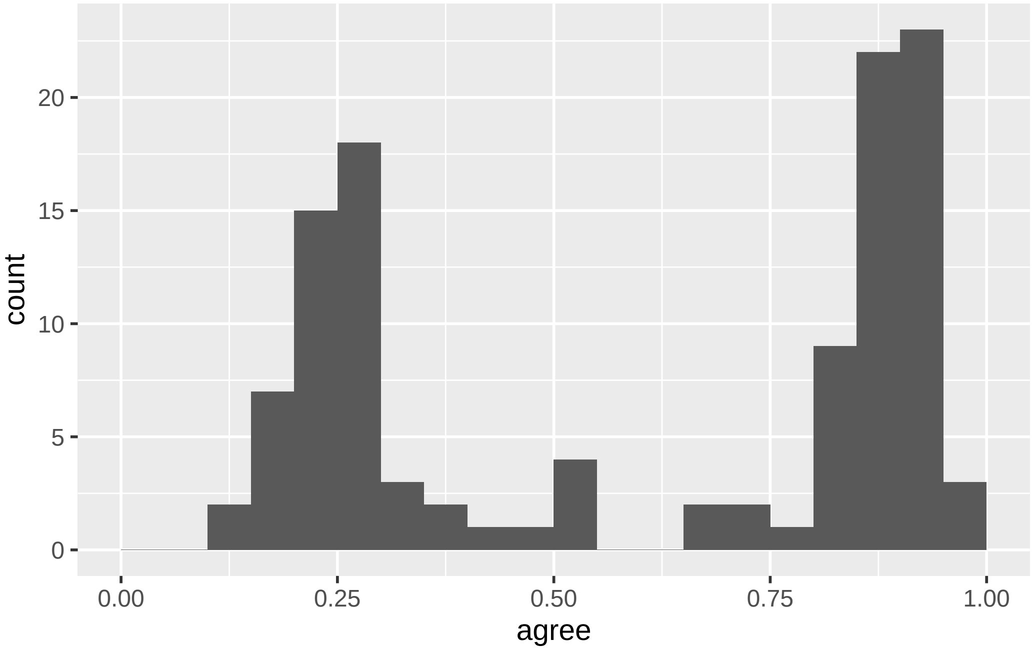
Notice the warning message above. It's telling us that, by default, `geom_histogram()` will draw 30 bins. Sometimes we want to modify this behavior. The following code has some common options for `geom_histogram()` and their explanations:

```
ggplot(trump_scores_ss, aes(x = agree)) +
  geom_histogram(binwidth = 0.05,    # draw bins every 0.05 jumps in x
                 boundary = 0,       # don't shift bins to integers
                 closed   = "left") # close bins on the left
```



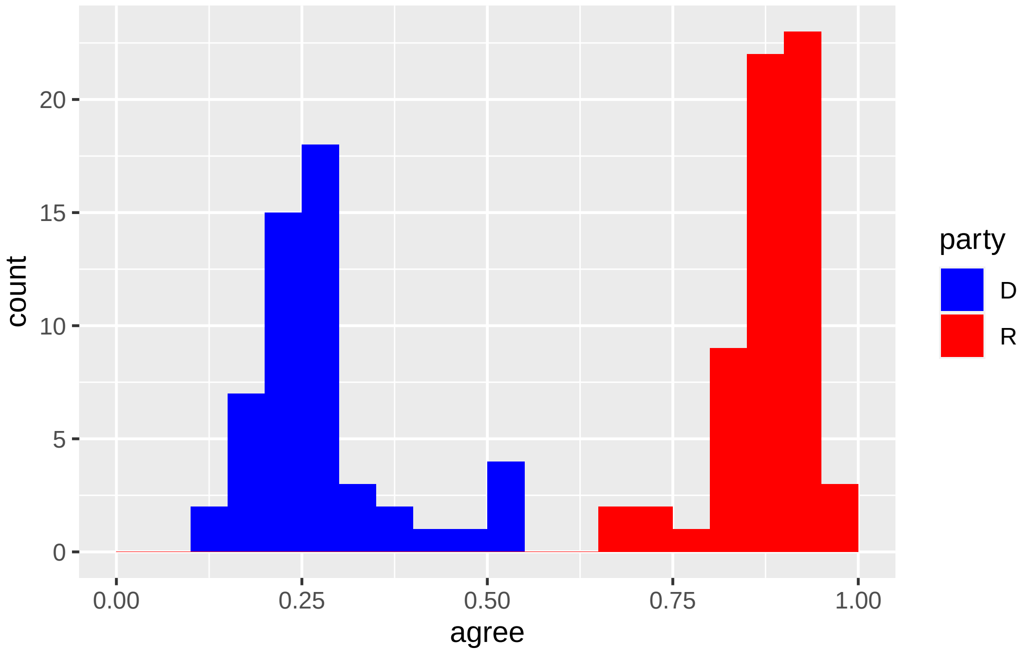
Sometimes we want to manually alter a scale. This is accomplished with the `scale_*()` family of `ggplot2` functions. Here we use the `scale_x_continuous()` function to make the x-axis go from 0 to 1:

```
ggplot(trump_scores_ss, aes(x = agree)) +
  geom_histogram(binwidth = 0.05, boundary = 0, closed   = "left") +
  scale_x_continuous(limits = c(0, 1))
```



Adding the `fill` aesthetic mapping to a histogram will divide it according to a categorical variable. This is actually a bivariate plot!

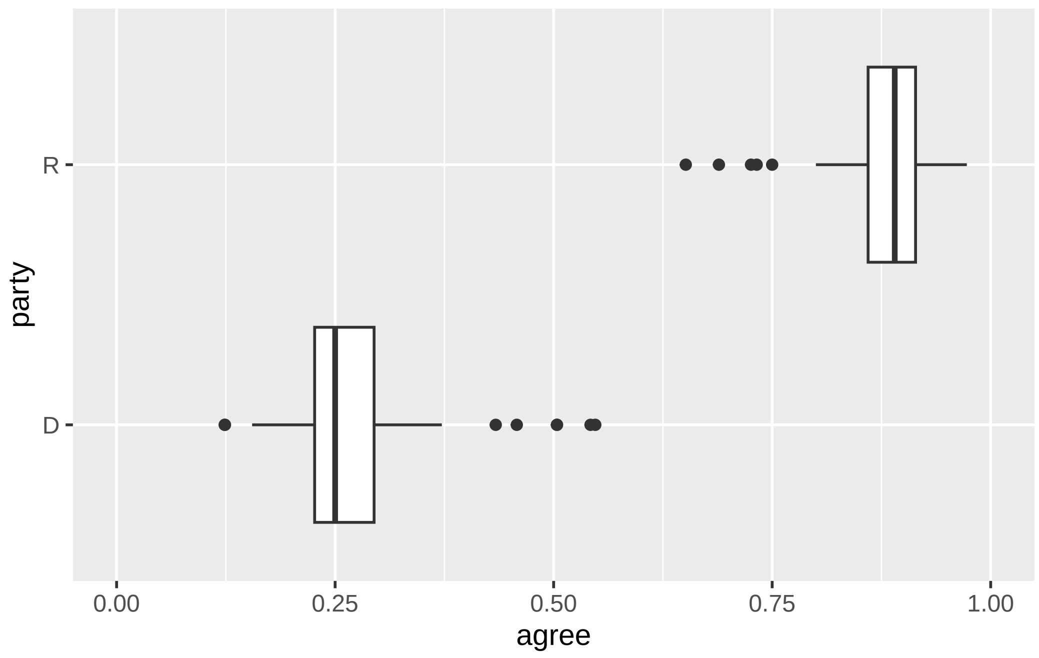
```
ggplot(trump_scores_ss, aes(x = agree, fill = party)) +
  geom_histogram(binwidth = 0.05, boundary = 0, closed   = "left") +
  scale_x_continuous(limits = c(0, 1)) +
  # change default colors:
  scale_fill_manual(values = c("D" = "blue", "R" = "red"))
```



2.3.3 Bivariate plots

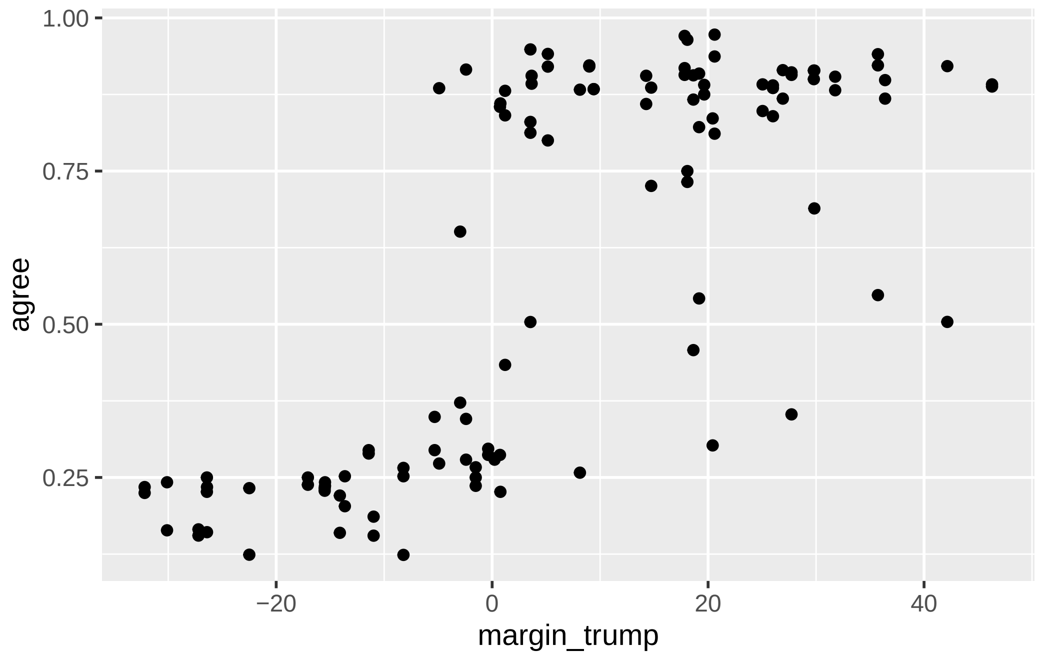
Another common bivariate plot for categorical and numerical variables is the grouped box plot:

```
ggplot(trump_scores_ss, aes(x = agree, y = party)) +
  geom_boxplot() +
  scale_x_continuous(limits = c(0, 1)) # same change as before
```



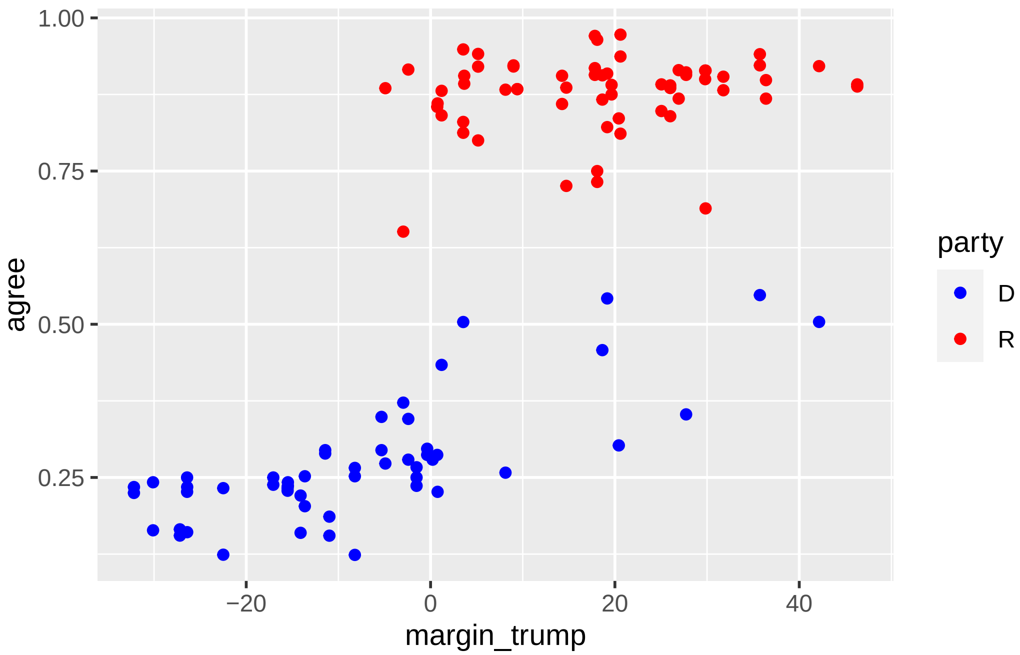
For bivariate plots of numerical variables, scatter plots are made with `geom_point()`:

```
ggplot(trump_scores_ss, aes(x = margin_trump, y = agree)) +
  geom_point()
```



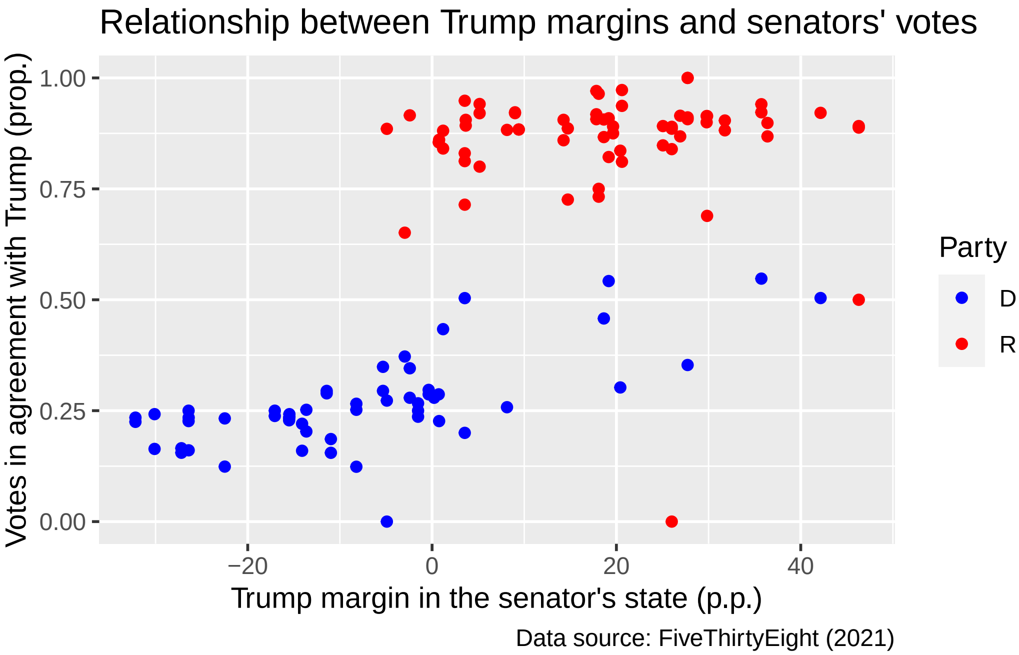
We can add the `color` aesthetic mapping to add a third variable:

```
ggplot(trump_scores_ss, aes(x = margin_trump, y = agree, color = party)) +
  geom_point() +
  scale_color_manual(values = c("D" = "blue", "R" = "red"))
```



Let's finish our plot with the `labs()` function, which allows us to add labels to our aesthetic mappings, as well as titles and notes:

```r
ggplot(trump_scores, aes(x = margin_trump, y = agree, color = party)) +
  geom_point() +
  scale_color_manual(values = c("D" = "blue", "R" = "red")) +
  labs(x = "Trump margin in the senator's state (p.p.)",
       y = "Votes in agreement with Trump (prop.)",
       color = "Party",
       title = "Relationship between Trump margins and senators' votes",
       caption = "Data source: FiveThirtyEight (2021)")
```



Data source: FiveThirtyEight (2021)

We will review a few more customization options, including text labels and facets, in a subsequent module.

3 Matrices

Matrices are rectangular collections of numbers. In this module we will introduce them and review some basic operators, to then introduce a sneak peek of why matrices are useful (and cool).

3.1 Introduction

3.1.1 Scalars

One number (for example, 12) is referred to as a scalar.

$$a = 12$$

3.1.2 Vectors

We can put several scalars together to make a vector. Here is an example:

$$\vec{b} = \begin{bmatrix} 12 \\ 14 \\ 15 \end{bmatrix}$$

Since this is a column of numbers, we cleverly refer to it as a *column vector*.

Here is another example of a vector, this time represented as a *row vector*:

$$\vec{c} = \begin{bmatrix} 12 & 14 & 15 \end{bmatrix}$$

Column vectors are possibly more common and useful, but we sometimes write things down using row vectors to

Vectors are fairly easy to construct in R. As we saw before, we can use the `c()` function to combine elements:

```
c(5, 25, -2, 1)
```

```
[1]  5 25 -2  1
```

> ⚠ Warning
>
> Remember that the code above does not *create* any objects. To do so, you'd need to use the assignment operator (`<-`):
>
> ```
> vector_example <- c(5, 25, -2, 1)
> vector_example
> ```
>
> ```
> [1]  5 25 -2  1
> ```

Or we can also create vectors from sequences with the `:` operator or the `seq()` function:

```
10:20
```

```
 [1] 10 11 12 13 14 15 16 17 18 19 20
```

```
seq(from = 3, to = 27, by = 3)
```

```
[1]  3  6  9 12 15 18 21 24 27
```

3.2 Operators

3.2.1 Summation

The summation operator $\sum$ (i.e., the uppercase Sigma letter) lets us perform an operation on a sequence of numbers, which is often but not always a vector.

$$\vec{d} = \begin{bmatrix} 12 & 7 & -2 & 3 & -1 \end{bmatrix}$$

We can then calculate the sum of the first three elements of the vector, which is expressed as follows:

$$\sum_{i=1}^{3} d_i$$

Then we do the following math:

$$12 + 7 + (-2) = 17$$

It is also common to use n in the superscript to indicate that we want to sum all elements:

$$\sum_{i=1}^{n} d_i = 12 + 7 + (-2) + 3 + (-1) = 19$$

We can perform these operations using the `sum()` function in R:

```
vector_d <- c(12, 7, -2, 3, -1)
```

```
sum(vector_d[1:3])
```

```
[1] 17
```

```
sum(vector_d)
```

```
[1] 19
```

3.2.2 Product

The product operator $\prod$ (i.e., the uppercase Pi letter) can also perform operations over a sequence of elements in a vector. Recall our previous vector:

$$\vec{d} = \begin{bmatrix} 12 & 7 & -2 & 3 & 1 \end{bmatrix}$$

We might want to calculate the product of all its elements, which is expressed as follows:

$$\prod_{i=1}^{n} d_i = 12 \cdot 7 \cdot (-2) \cdot 3 \cdot (-1) = 504$$

In R, we can compute products using the `prod()` function:

```
prod(vector_d)
```

```
[1] 504
```

Exercise

Get the product of the first three elements of vector d. Write the notation by hand and use R to obtain the number.

3.3 Matrices

3.3.1 Basics

We can append vectors together to form a matrix:

$$A = \begin{bmatrix} 12 & 14 & 15 \\ 115 & 22 & 127 \\ 193 & 29 & 219 \end{bmatrix}$$

The number of rows and columns of a matrix constitute the *dimensions* of the matrix. The first number is the number of rows ("r") and the second number is the number of columns ("c") in the matrix.

Important

Find a way to remember "r x c" *permanently*. The order of the dimensions never changes.

Matrix A above, for example, is a $3x3$ matrix. Sometimes we'd refer to it as A_{3x3}.

Tip

It is common to use capital letters (sometimes **bold-faced**) to represent matrices. In contrast, vectors are usually represented with either bold lowercase letters or lowercase letters with an arrow on top (e.g., $\vec{v}$).

Constructing matrices in R

There are different ways to create matrices in R. One of the simplest is via `rbind()` or `cbind()`, which paste vectors together (either by **r**ows or by **c**olumns):

```
# Create some vectors
vector1 <- 1:4
vector2 <- 5:8
vector3 <- 9:12
```

```
vector4 <- 13:16
```

```
# Using rbind(), each vector will be a row
rbind_mat <- rbind(vector1, vector2, vector3, vector4)
rbind_mat
```

```
        [,1] [,2] [,3] [,4]
vector1    1    2    3    4
vector2    5    6    7    8
vector3    9   10   11   12
vector4   13   14   15   16
```

```
# Using cbind(), each vector will be a column
cbind_mat <- cbind(vector1, vector2, vector3, vector4)
cbind_mat
```

```
     vector1 vector2 vector3 vector4
[1,]       1       5       9      13
[2,]       2       6      10      14
[3,]       3       7      11      15
[4,]       4       8      12      16
```

An alternative is to use to properly named `matrix()` function. The basic syntax is `matrix(data, nrow, ncol, byrow)`:

- `data` is the input vector which becomes the data elements of the matrix.
- `nrow` is the number of rows to be created.
- `ncol` is the number of columns to be created.
- `byrow` is a logical clue. If `TRUE` then the input vector elements are arranged by row. By default (`FALSE`), elements are arranged by column.

Let's see some examples:

```
# Elements are arranged sequentially by row.
M <- matrix(c(1:12), nrow = 4, byrow = T)
M
```

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

```
[3,]    7    8    9
[4,]   10   11   12
```

```
# Elements are arranged sequentially by column (byrow = F by default).
N <- matrix(c(1:12), nrow = 4)
N
```

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

3.3.2 Structure

How do we refer to specific elements of the matrix? For example, matrix A is an $m \times n$ matrix where $m = n = 3$. This is sometimes called a *square matrix*.

More generally, matrix B is an $m \times n$ matrix where the elements look like this:

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2n} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ b_{m1} & b_{m2} & b_{m3} & \dots & b_{mn} \end{bmatrix}$$

Thus b_{23} refers to the second unit down and third across. More generally, we refer to row indices as i and to column indices as j.

In R, we can access a matrix's elements using square brackets:

```
# In matrix N, access the element at 1st row and 3rd column.
N[1,3]
```

```
[1] 9
```

```
# In matrix N, access the element at 4th row and 2nd column.
N[4,2]
```

```
[1] 8
```

Tip

When trying to identify a specific element, the first subscript is the element's row and the second subscript is the element's column (*always* in that order).

3.4 Matrix operations

3.4.1 Addition and subtraction

- Addition and subtraction are straightforward operations.
- Matrices must have *exactly* the same dimensions for both of these operations.
- We add or subtract each element with the corresponding element from the other matrix.
- This is expressed as follows:

$$A \pm B = C$$

$$c_{ij} = a_{ij} \pm b_{ij} \ \forall i, j$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \pm \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$
$$=$$
$$\begin{bmatrix} a_{11} \pm b_{11} & a_{12} \pm b_{12} & a_{13} \pm b_{13} \\ a_{21} \pm b_{21} & a_{22} \pm b_{22} & a_{23} \pm b_{23} \\ a_{31} \pm b_{31} & a_{32} \pm b_{32} & a_{33} \pm b_{33} \end{bmatrix}$$

Addition and subtraction in R

We start by creating two 2x3 matrices:

```r
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
matrix1
```

```
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
```

```
matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
matrix2
```

```
     [,1] [,2] [,3]
[1,]    5    0    3
[2,]    2    9    4
```

We can simply use the `+` and `-` operators for addition and substraction:

```
matrix1 + matrix2
```

```
     [,1] [,2] [,3]
[1,]    8   -1    5
[2,]   11   13   10
```

```
matrix1 - matrix2
```

```
     [,1] [,2] [,3]
[1,]   -2   -1   -1
[2,]    7   -5    2
```

Exercise

(Use code for one of these and do the other one by hand!)
1) Calculate $A + B$

$$A = \begin{bmatrix} 1 & 0 \\ -2 & -1 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 1 \\ 2 & -1 \end{bmatrix}$$

2) Calculate $A - B$

$$A = \begin{bmatrix} 6 & -2 & 8 & 12 \\ 4 & 42 & 8 & -6 \end{bmatrix}$$

$$B = \begin{bmatrix} 18 & 42 & 3 & 7 \\ 0 & -42 & 15 & 4 \end{bmatrix}$$

3.4.2 Scalar multiplication

Scalar multiplication is very intuitive. As we know, a scalar is a single number. We multiply each value in the matrix by the scalar to perform this operation.

Formally, this is expressed as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$cA = \begin{bmatrix} ca_{11} & ca_{12} & ca_{13} \\ ca_{21} & ca_{22} & ca_{23} \\ ca_{31} & ca_{32} & ca_{33} \end{bmatrix}$$

In R, all we need to do is take an established matrix and multiply it by some scalar:

```
# matrix1 from our previous example
matrix1
```

```
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
```

```
matrix1 * 3
```

```
     [,1] [,2] [,3]
[1,]    9   -3    6
[2,]   27   12   18
```

Exercise

Calculate $2 \times A$ and $-3 \times B$. Again, do one by hand and the other one using R.

$$A = \begin{bmatrix} 1 & 4 & 8 \\ 0 & -1 & 3 \end{bmatrix}$$

$$B = \begin{bmatrix} -15 & 1 & 5 \\ 2 & -42 & 0 \\ 7 & 1 & 6 \end{bmatrix}$$

3.4.3 Matrix multiplication

- Multiplying matrices is slightly trickier than multiplying scalars.
- Two matrices must be *conformable* for them to be multiplied together. This means that the number of columns in the first matrix equals the number of rows in the second.
- When multiplying $A \times B$, if A is $m \times n$, B must have n rows.

! Important

The conformability requirement *never* changes. Before multiplying anything, check to make sure the matrices are indeed conformable.

- The resulting matrix will have the same number of rows as the first matrix and the number of columns in the second. For example, if A is $i \times k$ and B is $k \times j$, then $A \times B$ will be $i \times j$.

Which of the following can we multiply? What will be the dimensions of the resulting matrix?

$$B = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 1 \end{bmatrix} M = \begin{bmatrix} 1 & 0 & 2 \\ 1 & 2 & 4 \\ 2 & 3 & 2 \end{bmatrix} L = \begin{bmatrix} 6 & 5 & -1 \\ 1 & 4 & 3 \end{bmatrix}$$

Why can't we multiply in the opposite order?

⚠ Warning

When multiplying matrices, *order matters*. Even if multiplication is possible in both directions, in general $AB \neq BA$.

Multiplication steps

- Multiply each row by each column, summing up each pair of multiplied terms.

Tip

This is sometimes to referred to as the "dot product," where we multiply matching members, then sum up.

- The element in position ij is the sum of the products of elements in the ith row of the first matrix (A) and the corresponding elements in the jth column of the second matrix (B).

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Example

Suppose a company manufactures two kinds of furniture: chairs and sofas.

- A chair costs \$100 for wood, \$270 for cloth, and \$130 for feathers.
- Each sofa costs \$150 for wood, \$420 for cloth, and \$195 for feathers.

	Chair	Sofa
Wood	100	150
Cloth	270	420
Feathers	130	195

The same information about unit cost (C) can be presented as a matrix.

$$C = \begin{bmatrix} 100 & 150 \\ 270 & 420 \\ 130 & 195 \end{bmatrix}$$

Note that each of the three rows of this 3 x 2 matrix represents a material (wood, cloth, or feathers), and each of the two columns represents a product (chair or coach). The elements are the unit cost (in USD).

Now, suppose that the company will produce 45 chairs and 30 sofas this month. This production quantity can be represented in the following table, and also as a 2 x 1 matrix (Q):

Product	Quantity
Chair	45
Sofa	30

$$Q = \begin{bmatrix} 45 \\ 30 \end{bmatrix}$$

What will be the company's total cost? The "total expenditure" is equal to the "unit cost" times the "production quantity" (the number of units).

The total expenditure (E) for each material this month is calculated by multiplying these two matrices.

$$E = CQ = \begin{bmatrix} 100 & 150 \\ 270 & 420 \\ 130 & 195 \end{bmatrix} \begin{bmatrix} 45 \\ 30 \end{bmatrix} = \begin{bmatrix} (100)(45) + (150)(30) \\ (270)(45) + (420)(30) \\ (130)(45) + (195)(30) \end{bmatrix} = \begin{bmatrix} 9,000 \\ 24,750 \\ 11,700 \end{bmatrix}$$

Multiplying the 3x2 Cost matrix (C) times the 2x1 Quantity matrix (Q) yields the 3x1 Expenditure matrix (E).

As a result of this matrix multiplication, we determine that this month the company will incur expenditures of:

- \$9,000 for wood
- \$24,750 for cloth
- \$11,700 for feathers.

Matrix multiplication in R

Before attempting matrix multiplication, we must make sure the matrices are conformable (as we do for our manual calculations).

Then we can multiply our matrices together using the `%*%` operator.

```r
C <- matrix(c(100, 270, 130, 150, 420, 195), nrow = 3)
C
```

```
     [,1] [,2]
[1,]  100  150
[2,]  270  420
[3,]  130  195
```

```r
Q <- matrix(c(45, 30), nrow = 2)
Q
```

```
     [,1]
[1,]   45
[2,]   30
```

```r
C %*% Q
```

```
      [,1]
[1,]  9000
[2,] 24750
[3,] 11700
```

> ⚠ Warning
>
> If you have a missing value or `NA` in one of the matrices you are trying to multiply (something we will discuss in further detail in the next module), you will have `NA`s in your resulting matrix.

3.4.4 Properties of operations

- Addition and subtraction:
  - Associative: $(A \pm B) \pm C = A \pm (B \pm C)$
  - Communicative: $A \pm B = B \pm A$

- Multiplication:
  - $AB \neq BA$
  - $A(BC) = (AB)C$
  - $A(B + C) = AB + AC$
  - $(A + B)C = AC + BC$

3.5 Special matrices

Square matrix

- In a square matrix, the number of rows equals the number of columns ($m = n$):
- The *diagonal* of a matrix is a set of numbers consisting of the elements on the line from the upper-left-hand to the lower-right-hand corner of the matrix. Diagonals are particularly useful in square matrices.
- The *trace* of a matrix, denoted as $tr(A)$, is the sum of the diagonal elements of the matrix.

Diagonal matrix:

- In a diagonal matrix, all of the elements of the matrix that are not on the diagonal are equal to zero.

Scalar matrix:

- A scalar matrix is a diagonal matrix where the diagonal elements are all equal to each other. In other words, we're really only concerned with one scalar (or element) held in the diagonal.

Identity matrix:

- The identity matrix is a scalar matrix with all of the diagonal elements equal to one.
- Remember that, as with all diagonal matrices, the off-diagonal elements are equal to zero.
- The capital letter I is reserved for the identity matrix. For convenience, a 3x3 identity matrix can be denoted as I_3.

3.6 Transpose

The transpose is the original matrix with the rows and the columns interchanged.

The notation is either J' ("J prime") or J^T ("J transpose").

$$J = \begin{bmatrix} 4 & 5 \\ 3 & 0 \\ 7 & -2 \end{bmatrix}$$

$$J' = J^T = \begin{bmatrix} 4 & 3 & 7 \\ 5 & 0 & -2 \end{bmatrix}$$

In R, we use `t()` to get the transpose.

```
J <- matrix(c(4, 3, 7, 5, 0, -2), ncol = 2)
J
```

```
     [,1] [,2]
[1,]    4    5
[2,]    3    0
[3,]    7   -2
```

```
t(J)
```

```
     [,1] [,2] [,3]
[1,]    4    3    7
[2,]    5    0   -2
```

3.7 Inverse

- Just like a number has a reciprocal, a matrix has an inverse.
- When we multiply a matrix by its inverse we get the identity matrix (which is like "1" for matrices).

$$A \times A^{-1} = I$$

- The inverse of A is A^{-1} only when:

$$AA^{-1} = A^{-1}A = I$$

- Sometimes there is no inverse at all.

> **Note**
>
> For now, don't worry about calculating the inverse of a matrix manually. This is the type of task we use R for.

- In R, we use the `solve()` function to calculate the inverse of a matrix:

```r
A <- matrix(c(3, 2, 5, 2, 3, 2, 5, 2, 4), ncol = 3)
A
```

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

```r
solve(A)
```

```
            [,1]        [,2]       [,3]
[1,] -0.29629630 -0.07407407  0.4074074
[2,] -0.07407407  0.48148148 -0.1481481
[3,]  0.40740741 -0.14814815 -0.1851852
```

3.8 Linear systems and matrices

- A system of equations can be represented by an *augmented matrix*.
- System of equations:

$$3x + 6y = 12$$

$$5x + 10y = 25$$

- In an augmented matrix, each row represents one equation in the system and each column represents a variable or the constant terms.

$$\begin{bmatrix} 3 & 6 & 12 \\ 5 & 10 & 25 \end{bmatrix}$$

3.9 OLS and matrices

- We can use the logic above to calculate estimates for our ordinary least squares (OLS) models.
- OLS is a linear regression technique used to find the best-fitting line for a set of data points (observations) by minimizing the residuals (the differences between the observed and predicted values).
- We minimize the *sum of the squared errors.*

3.9.1 Dependent variable

- Suppose, for example, we have a sample consisting of n observations.
- The dependent variable is denoted as an $n \times 1$ column vector.

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

3.9.2 Independent variables

- Suppose there are k independent variables and a constant term, meaning $k+1$ columns and n rows.
- We can represent these variables as an $n \times (k+1)$ matrix, expressed as follows:

$$X = \begin{bmatrix} 1 & x_{11} & ... & x_{1k} \\ 1 & x_{21} & ... & x_{2k} \\ \vdots & \vdots & ... & \vdots \\ 1 & x_{n1} & ... & x_{nk} \end{bmatrix}$$

- x_{ij} is the i-th observation of the j-th independent variable.

3.9.3 Linear regression model

- Let's say we have 173 observations (n = 173) and 2 IVs (k = 3).
- This can be expressed as the following linear equation:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon$$

- In matrix form, we have:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{21} \\ 1 & x_{21} & x_{22} \\ \vdots & \vdots & \vdots \\ 1 & x_{1173} & x_{2173} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_{173} \end{bmatrix}$$

- All 173 equations can be represented by:

$$y = X\beta + \epsilon$$

3.9.4 Estimates

- Without getting too much into the mechanics, we can calculate our coefficient estimates with matrix algebra using the following equation:

$$\hat{\beta} = (X'X)^{-1}X'Y$$

- Read aloud, we say "X prime X inverse, X prime Y".
- The little hat on our beta ($\hat{\beta}$) signifies that these are estimates, that is our OLS estimators.
- Remember, the OLS method is to choose $\hat{\beta}$ such that the sum of squared residuals ("SSR") is minimized.

3.9.4.1 Example in R

- We will load the `mtcars` data set (our favorite) for this example, which contains data about many different car models.

```r
cars_df <- mtcars
```

- Now, we want to estimate the association between `hp` (horsepower) and `wt` (weight), our independent variables, and `mpg` (miles per gallon), our dependent variable.

- First, we transform our dependent variable into a matrix, using the `as.matrix` function and specifying the column of the `mtcars` data set to create a column vector of our observed values for the DV.

```
Y <- as.matrix(cars_df[,1])
Y
```

```
      [,1]
 [1,] 21.0
 [2,] 21.0
 [3,] 22.8
 [4,] 21.4
 [5,] 18.7
 [6,] 18.1
 [7,] 14.3
 [8,] 24.4
 [9,] 22.8
[10,] 19.2
[11,] 17.8
[12,] 16.4
[13,] 17.3
[14,] 15.2
[15,] 10.4
[16,] 10.4
[17,] 14.7
[18,] 32.4
[19,] 30.4
[20,] 33.9
[21,] 21.5
[22,] 15.5
[23,] 15.2
[24,] 13.3
[25,] 19.2
[26,] 27.3
[27,] 26.0
[28,] 30.4
[29,] 15.8
[30,] 19.7
[31,] 15.0
[32,] 21.4
```

- Next, we do the same thing for our independent variables of interest, and our constant.

```
# create two separate matrices for IVs
X1 <- as.matrix(cars_df[,4])
X2 <- as.matrix(cars_df[,6])

# create constant column

# bind them altogether into one matrix
constant <-  rep(1,nrow(X1))
X <- cbind(constant,X1,X2)
X
```

```
      constant
 [1,]        1 110 2.620
 [2,]        1 110 2.875
 [3,]        1  93 2.320
 [4,]        1 110 3.215
 [5,]        1 175 3.440
 [6,]        1 105 3.460
 [7,]        1 245 3.570
 [8,]        1  62 3.190
 [9,]        1  95 3.150
[10,]        1 123 3.440
[11,]        1 123 3.440
[12,]        1 180 4.070
[13,]        1 180 3.730
[14,]        1 180 3.780
[15,]        1 205 5.250
[16,]        1 215 5.424
[17,]        1 230 5.345
[18,]        1  66 2.200
[19,]        1  52 1.615
[20,]        1  65 1.835
[21,]        1  97 2.465
[22,]        1 150 3.520
[23,]        1 150 3.435
[24,]        1 245 3.840
[25,]        1 175 3.845
[26,]        1  66 1.935
[27,]        1  91 2.140
[28,]        1 113 1.513
[29,]        1 264 3.170
[30,]        1 175 2.770
```

```
[31,]        1 335 3.570
[32,]        1 109 2.780
```

- Next, we calculate $X'X$, $X'Y$, and $(X'X)^{-1}$.

Don't forget to use `%*%` for matrix multiplication!

```
# X prime X
XpX <- t(X)%*%X

# X prime X inverse
XpXinv <- solve(XpX)

# X prime Y
XpY <- t(X)%*%Y

# beta coefficient estimates
bhat <- XpXinv %*% XpY
bhat
```

```
                [,1]
constant 37.22727012
         -0.03177295
         -3.87783074
```

4 Tidy data analysis II

In this session, we'll cover a few more advanced topics related to data wrangling. Again we'll use the `tidyverse`:

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.2     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.2     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

4.1 Loading data in different formats.

In this module we will use cross-national data from the Quality of Government (QoG) project (Dahlberg et al., 2023).

Notice how in the `data/` folder we have multiple versions of the same dataset (a subset of the QOG basic dataset): .csv (comma-separated values), .rds (R), .xlsx (Excel), .dta (Stata), and .sav (SPSS).

4.1.1 CSV and R data files

We can use the `read_csv()` and `read_rds()` functions from the `tidyverse`[1] to read the .csv and .rds (R) data files:

[1] Technically, the `read_csv()` and `read_rds()` functions come from `readr`, one of the `tidyverse` constituent packages.

```r
qog_csv <- read_csv("data/sample_qog_bas_ts_jan23.csv")
```

```
Rows: 1085 Columns: 8
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (4): cname, ccodealp, region, ht_colonial
dbl (4): year, wdi_pop, vdem_polyarchy, vdem_corr

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.
```

```r
qog_rds <- read_rds("data/sample_qog_bas_ts_jan23.rds")
```

For reading files from other software (Excel, Stata, or SPSS), we need to load additional packages. Luckily, they are automatically installed when one installs the `tidyverse`.

4.1.2 Excel data files

For Excel files (.xls or .xlsx files), the `readxl` package has a handy `read_excel()` function.

```r
library(readxl)
qog_excel <- read_excel("data/sample_qog_bas_ts_jan23.xlsx")
```

> Tip
>
> Useful arguments of the `read_excel()` function include `sheet =`, which reads particular sheets (specified via their positions or sheet names), and `range =`, which extracts a particular cell range (e.g., ‘A5:E25‘).

4.1.3 Stata and SPSS data files

To load files from Stata (.dta) or SPSS (.spss), one needs the `haven` package and its properly-named `read_stata()` and `read_spss()` functions:

```r
library(haven)
qog_stata <- read_stata("data/sample_qog_bas_ts_jan23.dta")
qog_spss <- read_spss("data/sample_qog_bas_ts_jan23.sav")
```

Tip

Datasets from Stata and SPSS can have additional properties, like variable labels and special types of missing values. To learn more about this, check out the “Labelled data” chapter from Danny Smith’s *Survey Research Datasets and R* (2020).

4.1.4 Our data for this session

We will rename one of our objects to qog:

```
qog <- qog_csv
qog
```

```
# A tibble: 1,085 x 8
   cname      ccodealp  year region wdi_pop vdem_polyarchy vdem_corr ht_colonial
   <chr>      <chr>    <dbl> <chr>    <dbl>          <dbl>     <dbl> <chr>
 1 Antigua a~ ATG       1990 Carib~   63328             NA        NA British
 2 Antigua a~ ATG       1991 Carib~   63634             NA        NA British
 3 Antigua a~ ATG       1992 Carib~   64659             NA        NA British
 4 Antigua a~ ATG       1993 Carib~   65834             NA        NA British
 5 Antigua a~ ATG       1994 Carib~   67072             NA        NA British
 6 Antigua a~ ATG       1995 Carib~   68398             NA        NA British
 7 Antigua a~ ATG       1996 Carib~   69798             NA        NA British
 8 Antigua a~ ATG       1997 Carib~   71218             NA        NA British
 9 Antigua a~ ATG       1998 Carib~   72572             NA        NA British
10 Antigua a~ ATG       1999 Carib~   73821             NA        NA British
# i 1,075 more rows
```

This dataset is a small sample of QOG, which contains data for countries in the Americas from 1990 to 2020. The observational unit is thus country-year. You can access the full codebook online. The variables are as follows:

Variable	Description
cname	Country name
ccodealp	Country code (ISO-3 character convention)
year	Year
region	Region (following legacy WDI convention). Added to QOG by us.
wdi_pop	Total population, from the World Development Indicators
vdem_polyarchy	V-Dem’s polyarchy index (electoral democracy)

Variable	Description
`vdem_corr`	V-Dem's corruption index
`ht_colonial`	Former colonial ruler

4.2 Recoding variables

Take a look at the `ht_colonial` variable. We can do a simple tabulation with `count()`:

```r
qog |>
  count(ht_colonial)
```

```
# A tibble: 6 x 2
  ht_colonial         n
  <chr>           <int>
1 British           372
2 Dutch              31
3 French             31
4 Never colonized    62
5 Portuguese         31
6 Spanish           558
```

We might want to recode this variable. For instance, we could create a *dummy/binary* variable for whether the country was a British colony. We can do this with `if_else()`, which works with logical conditions:

```r
qog |>
  # the arguments are condition, true (what to do if true), false
  mutate(d_britishcol = if_else(ht_colonial == "British", 1, 0)) |>
  count(d_britishcol)
```

```
# A tibble: 2 x 2
  d_britishcol     n
         <dbl> <int>
1            0   713
2            1   372
```

Instead of a numeric classification (0 and 1), we could use characters:

```r
qog |>
  mutate(cat_britishcol = if_else(ht_colonial == "British", "British", "Other")) |>
  count(cat_britishcol)
```

```
# A tibble: 2 x 2
  cat_britishcol     n
  <chr>          <int>
1 British          372
2 Other            713
```

`if_else()` is great for binary recoding. But sometimes we want to create more than two categories. We can use `case_when()`:

```r
qog |>
  # syntax is condition ~ value
  mutate(cat_col = case_when(
    ht_colonial == "British" ~ "British",
    ht_colonial == "Spanish" ~ "Spanish",
    .default = "Other" # what to do in all other cases
  )) |>
  count(cat_col)
```

```
# A tibble: 3 x 2
  cat_col     n
  <chr>   <int>
1 British   372
2 Other     155
3 Spanish   558
```

The `.default =` argument in `case_when()` can also be used to leave the variable as-is for non-specified cases. For example, let's combine Portuguese and Spanish colonies:

```r
qog |>
  # syntax is condition ~ value
  mutate(cat_col = case_when(
    ht_colonial %in% c("Spanish", "Portuguese") ~ "Spanish/Portuguese",
    .default = ht_colonial # what to do in all other cases
  )) |>
  count(cat_col)
```

```
# A tibble: 5 x 2
  cat_col                n
  <chr>              <int>
1 British              372
2 Dutch                 31
3 French                31
4 Never colonized       62
5 Spanish/Portuguese   589
```

Exercise

1. Create a dummy variable, `d_large_pop`, for whether the country-year has a population of more than 1 million. Then compute its mean. Your code:
2. Which countries are recorded as "Never colonized"? Change their values to other reasonable codings and compute a tabulation with `count()`. Your code:

4.3 Missing values

Missing values are commonplace in real datasets. In R, missing values are a special type of value in vectors, denoted as `NA`.

Warning

The special value `NA` is different from the character value "NA". For example, notice that a numeric vector can have `NA`s, while it obviously cannot hold the character value "NA":

```
c(5, 4.6, NA, 8)
```

```
[1] 5.0 4.6  NA 8.0
```

A quick way to check for missing values in small datasets is with the `summary()` function:

```
summary(qog)
```

```
    cname             ccodealp                year          region         
 Length:1085        Length:1085        Min.   :1990   Length:1085       
 Class  :character  Class  :character  1st Qu.:1997   Class  :character  
 Mode   :character  Mode   :character  Median :2005   Mode   :character  
                                       Mean   :2005                      
                                       3rd Qu.:2013
                                       Max.   :2020

    wdi_pop          vdem_polyarchy     vdem_corr      ht_colonial
 Min.   :    40542   Min.   :0.0710   Min.   :0.0260   Length:1085
 1st Qu.:   389131   1st Qu.:0.5570   1st Qu.:0.1890   Class :character
 Median :  5687744   Median :0.7030   Median :0.5550   Mode  :character
 Mean   : 25004057   Mean   :0.6569   Mean   :0.4922
 3rd Qu.: 16195902   3rd Qu.:0.8030   3rd Qu.:0.7540
 Max.   :331501080   Max.   :0.9160   Max.   :0.9630
                     NA's   :248      NA's   :248
```

Notice that we have missingness in the `vdem_polyarchy` and `vdem_corr` variables. We might want to filter the dataset to see which observations are in this situation:

```
qog |>
  filter(vdem_polyarchy == NA | vdem_corr == NA)
```

```
# A tibble: 0 x 8
# i 8 variables: cname <chr>, ccodealp <chr>, year <dbl>, region <chr>,
#   wdi_pop <dbl>, vdem_polyarchy <dbl>, vdem_corr <dbl>, ht_colonial <chr>
```

But the code above doesn't work! To refer to missing values in logical conditions, we cannot use `== NA`. Instead, we need to use the `is.na()` function:

```
qog |>
  filter(is.na(vdem_polyarchy) | is.na(vdem_corr))
```

```
# A tibble: 248 x 8
   cname      ccodealp  year region wdi_pop vdem_polyarchy vdem_corr ht_colonial
   <chr>      <chr>    <dbl> <chr>    <dbl>          <dbl>     <dbl> <chr>
 1 Antigua a~ ATG       1990 Carib~   63328             NA        NA British
 2 Antigua a~ ATG       1991 Carib~   63634             NA        NA British
 3 Antigua a~ ATG       1992 Carib~   64659             NA        NA British
 4 Antigua a~ ATG       1993 Carib~   65834             NA        NA British
 5 Antigua a~ ATG       1994 Carib~   67072             NA        NA British
 6 Antigua a~ ATG       1995 Carib~   68398             NA        NA British
 7 Antigua a~ ATG       1996 Carib~   69798             NA        NA British
 8 Antigua a~ ATG       1997 Carib~   71218             NA        NA British
 9 Antigua a~ ATG       1998 Carib~   72572             NA        NA British
```

```
10 Antigua a~ ATG       1999 Carib~   73821              NA        NA British
# i 238 more rows
```

Notice that, in most R functions, missing values are "contagious." This means that any missing value will contaminate the operation and carry over to the results. For example:

```
qog |>
  summarize(mean_vdem_polyarchy = mean(vdem_polyarchy))
```

```
# A tibble: 1 x 1
  mean_vdem_polyarchy
                <dbl>
1                  NA
```

Sometimes we'd like to perform our operations even in the presence of missing values, simply excluding them. Most basic R functions have an `na.rm =` argument to do this:

```
qog |>
  summarize(mean_vdem_polyarchy = mean(vdem_polyarchy, na.rm = T))
```

```
# A tibble: 1 x 1
  mean_vdem_polyarchy
                <dbl>
1               0.657
```

> ℹ Exercise
>
> Calculate the median value of the corruption variable for each region (i.e., perform a grouped summary). Your code:

4.4 Pivoting data

We will now load another time-series cross-sectional dataset, but in a slightly different format. It's adapted from the World Bank's World Development Indicators (WDI) (2023) and records gross domestic product at purchasing power parity (GDP PPP).

```
gdp <- read_excel("data/wdi_gdp_ppp.xlsx")
```

```
gdp
```

```
# A tibble: 266 x 35
   country_name        country_code   `1990`   `1991`   `1992`   `1993`   `1994`
   <chr>               <chr>           <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
 1 Aruba               ABW           2.03e 9  2.19e 9  2.32e 9  2.48e 9  2.69e 9
 2 Africa Eastern and~ AFE           9.41e11  9.42e11  9.23e11  9.19e11  9.35e11
 3 Afghanistan         AFG          NA       NA       NA       NA       NA
 4 Africa Western and~ AFW           5.76e11  5.84e11  5.98e11  5.92e11  5.91e11
 5 Angola              AGO           6.85e10  6.92e10  6.52e10  4.95e10  5.02e10
 6 Albania             ALB           1.59e10  1.14e10  1.06e10  1.16e10  1.26e10
 7 Andorra             AND          NA       NA       NA       NA       NA
 8 Arab World          ARB           2.19e12  2.25e12  2.35e12  2.41e12  2.48e12
 9 United Arab Emirat~ ARE           2.01e11  2.03e11  2.10e11  2.12e11  2.27e11
10 Argentina           ARG           4.61e11  5.04e11  5.43e11  5.88e11  6.22e11
# i 256 more rows
# i 28 more variables: `1995` <dbl>, `1996` <dbl>, `1997` <dbl>, `1998` <dbl>,
#   `1999` <dbl>, `2000` <dbl>, `2001` <dbl>, `2002` <dbl>, `2003` <dbl>,
#   `2004` <dbl>, `2005` <dbl>, `2006` <dbl>, `2007` <dbl>, `2008` <dbl>,
#   `2009` <dbl>, `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>,
#   `2014` <dbl>, `2015` <dbl>, `2016` <dbl>, `2017` <dbl>, `2018` <dbl>,
#   `2019` <dbl>, `2020` <dbl>, `2021` <dbl>, `2022` <dbl>
```

Note how the information is recorded differently. Here columns are not variables, but years. We call datasets like this one **wide**, in contrast to the **long** datasets we have seen before. In general, R and the `tidyverse` work much nicer with long datasets. Luckily, the `tidyr` package of the `tidyverse` makes it easy to convert datasets between these two formats.

We will use the `pivot_longer()` function:

```
gdp_long <- gdp |>
  pivot_longer(cols = -c(country_name, country_code), # cols to not pivot
               names_to = "year", # how to name the column with names
               values_to = "wdi_gdp_ppp",  # how to name the column with values
               names_transform = as.integer) # make sure that years are numeric
gdp_long
```

```
# A tibble: 8,778 x 4
   country_name country_code  year wdi_gdp_ppp
   <chr>        <chr>        <int>       <dbl>
 1 Aruba        ABW           1990 2025472682.
```

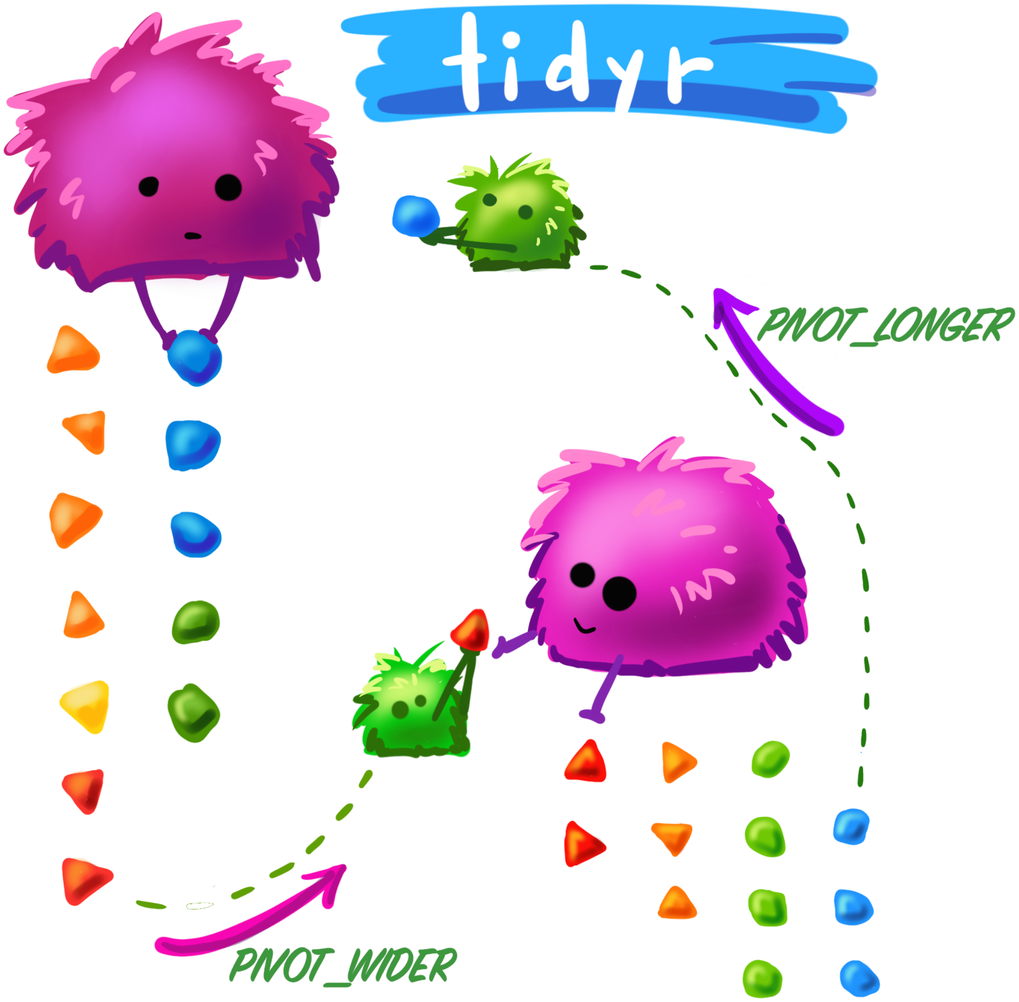


Figure 4.1: Source: Illustration by Allison Horst, adapted by Peter Higgins.

```
 2 Aruba        ABW           1991 2186758474.
 3 Aruba        ABW           1992 2315391348.
 4 Aruba        ABW           1993 2484593045.
 5 Aruba        ABW           1994 2688426606.
 6 Aruba        ABW           1995 2756904694.
 7 Aruba        ABW           1996 2789595753.
 8 Aruba        ABW           1997 2986175079.
 9 Aruba        ABW           1998 3045659222.
10 Aruba        ABW           1999 3083365758.
# i 8,768 more rows
```

Done! This is a much friendlier format to work with. For example, we can now do summaries:

```
gdp_long |>
  summarize(mean_gdp_ppp = mean(wdi_gdp_ppp, na.rm = T), .by = country_name)
```

```
# A tibble: 266 x 2
   country_name                mean_gdp_ppp
   <chr>                              <dbl>
 1 Aruba                            3.38e 9
 2 Africa Eastern and Southern      1.61e12
 3 Afghanistan                      5.56e10
 4 Africa Western and Central       1.15e12
 5 Angola                           1.38e11
 6 Albania                          2.56e10
 7 Andorra                        NaN
 8 Arab World                       4.22e12
 9 United Arab Emirates             4.29e11
10 Argentina                        8.06e11
# i 256 more rows
```

ℹ Exercise

Convert back `gdp_long` to a wide format using `pivot_wider()`. Check out the help file using `?pivot_wider`. Your code:

4.5 Merging datasets

It is extremely common to want to integrate data from multiple sources. Combining information from two datasets is called *merging* or *joining*.

To do this, we need ID variables in common between the two data sets. Using our QOG and WDI datasets, these variables will be country code (which in this case is shared between the two datasets) and year.

Tip

Standardized unit codes (like country codes) are extremely useful when merging data. It's harder than expected for a computer to realize that "Bolivia (Plurinational State of)" and "Bolivia" refer to the same unit. By default, these units will not be matched.[2]

Okay, now to the merging. Imagine we want to add information about GDP to our QOG main dataset. To do so, we can use the `left_join()` function, from the `tidyverse`'s `dplyr` package:

```
qog_plus <- left_join(qog, # left data frame, which serves as a "base"
                      gdp_long, # right data frame, from which to draw new columns
                      by = c("ccodealp" = "country_code", # can define name equivalencies!
                             "year"))
```

```
qog_plus |>
  # select variables for clarity
  select(cname, ccodealp, year, wdi_pop, wdi_gdp_ppp)
```

```
# A tibble: 1,085 x 5
   cname               ccodealp  year wdi_pop wdi_gdp_ppp
   <chr>               <chr>    <dbl>   <dbl>       <dbl>
 1 Antigua and Barbuda ATG       1990   63328  966660878.
 2 Antigua and Barbuda ATG       1991   63634  987701012.
 3 Antigua and Barbuda ATG       1992   64659  999143284.
 4 Antigua and Barbuda ATG       1993   65834 1051896837.
 5 Antigua and Barbuda ATG       1994   67072 1122128908.
 6 Antigua and Barbuda ATG       1995   68398 1073208718.
 7 Antigua and Barbuda ATG       1996   69798 1144088355.
 8 Antigua and Barbuda ATG       1997   71218 1206688391.
 9 Antigua and Barbuda ATG       1998   72572 1263778328.
10 Antigua and Barbuda ATG       1999   73821 1310634399.
# i 1,075 more rows
```

[2] There are R packages to deal with these complications. `fuzzyjoin` matches units by their approximate distance, using some clever algorithms. `countrycode` allows one to standardize country names and country codes across different conventions.

Tip

Most of the time, you'll want to do a `left_join()`, which is great for adding new information to a "base" dataset, without dropping information from the latter. In limited situations, other types of joins can be helpful. To learn more about them, you can read Jenny Bryan's excelent tutorial on `dplyr` joins.

Exercise

There is a dataset on country's CO2 emissions, again from the World Bank (2023), in "data/wdi_co2.csv". Load the dataset into R and add a new variable with its information, `wdi_co2`, to our `qog_plus` data frame. Finally, compute the average values of CO2 emissines *per capita*, by country. Tip: this exercise requires you to do many steps—plan ahead before you start coding! Your code:

5 Functions and loops

5.1 Basics

5.1.1 What is a function?

Informally, a function is anything that takes input(s) and gives one defined output. There are always three main parts:

- The input (x values, or each value in the domain)
- The relationship of interest
- The output (y values, or a unique value in the range)

> Note
>
> “$f(x) = ...$ is the classic notation for writing a function, but we can also use”$y = ...$“. This is because y is”a function of” x, so $y = f(x)$.

Let’s take a look at an example and break down the structure:

$$f(x) = 3x + 4$$

- x is the *input* (some value) that the function takes.
- For any x, we multiply by three and add 4, which is the *relationship*.
- Finally, $f(x)$ or y is the unique result, or the *output*.

The most common name to give a function is, predictably, “f”, but we can have other names such as “g” or “h”. The choice is yours.

> Important
>
> When reading out loud, we say “[name of function] of x equals [relationship]. For example, $f(x) = x^2$ is referred to as”f of x equals x squared.”

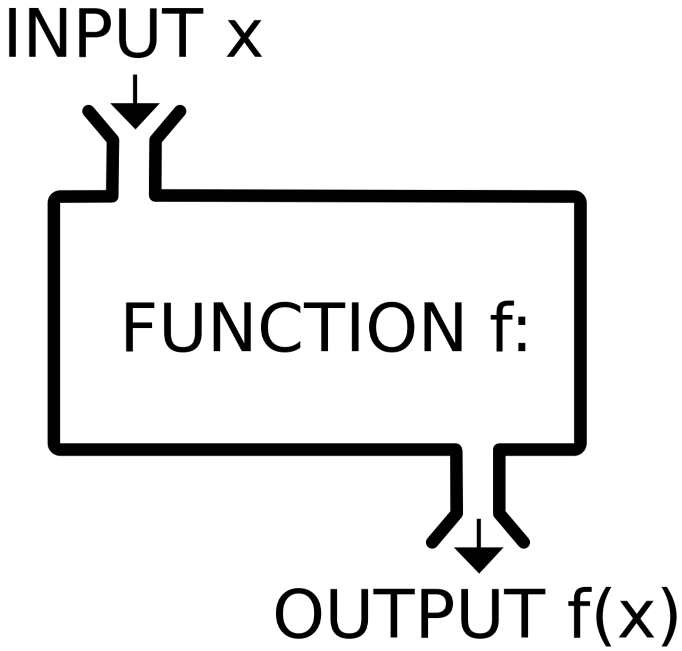


Figure 5.1: Function machine. Source: Bill Bailey on Wikimedia Commons.

5.1.2 Vertical line test

Exercise

When graphed, vertical lines cannot touch functions at more than one point. Why? Which of the following represent functions?

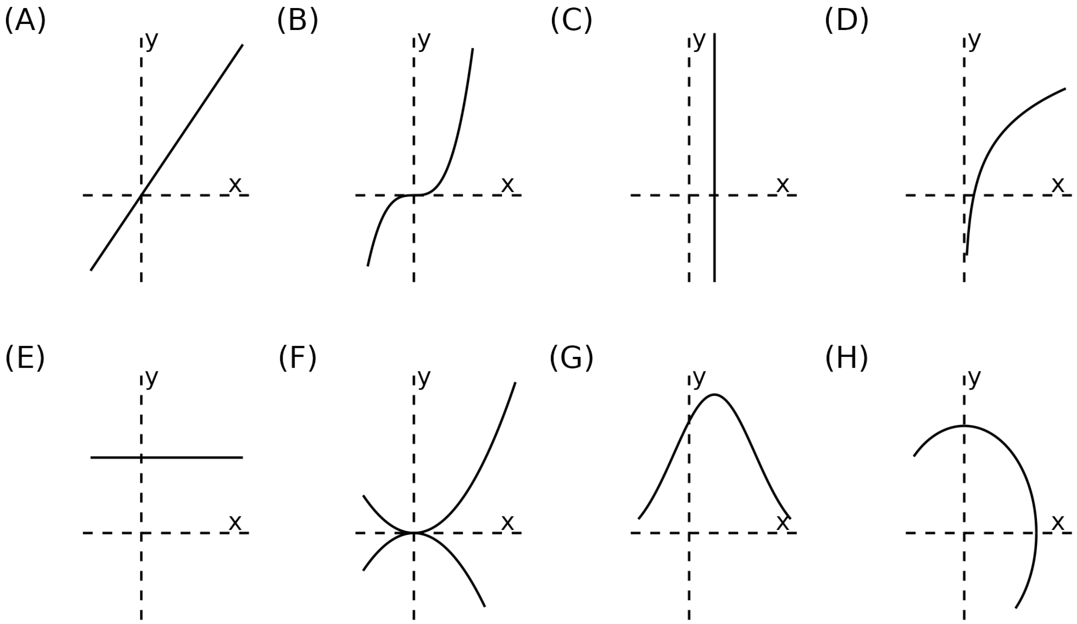


Figure 5.2: Vertical line test: examples.

5.2 Functions in R

Often we need to create our own functions in R. To build them: we use the keyword `function` alongside the following syntax: `function_name <- function(argumentnames){ operation }`

- `function_name`: the name of the function, that will be stored as an object in the R environment. Make the name concise and memorable!
- `function(argumentnames)`: the inputs of the function.
- `{ operation }`: a set of commands that are run in a predefined order every time we call the function.

For example, we can create a function that multiplies a number by 2:

```r
mult_by_two <- function(x){x * 2}
```

```r
mult_by_two(x = 5) # we can also omit the argument name (x =)
```

```
[1] 10
```

If the function body works for vectors, our custom function will do too:

```r
mult_by_two(1:10)
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

We can also automate more complicated tasks such as calculating the area of a circle from its radius:

```r
calc_circle_area <- function(r){
    pi * r ^ 2
}
calc_circle_area(r = 3)
```

```
[1] 28.27433
```

> ℹ Exercise
>
> Create a function that calculates the area of a circle *from its diameter*. So `your_function(d = 6)` should yield the same result as the example above. Your code:

Functions can take more than one argument/input. In a silly example, let's generalize our first function:

```r
mult_by <- function(x, mult){x * mult}
```

```r
mult_by(x = 1:5, mult = 10)
```

```
[1] 10 20 30 40 50
```

```
mult_by(1:5, mult = 10)
```

```
[1] 10 20 30 40 50
```

```
mult_by(1:5, 10)
```

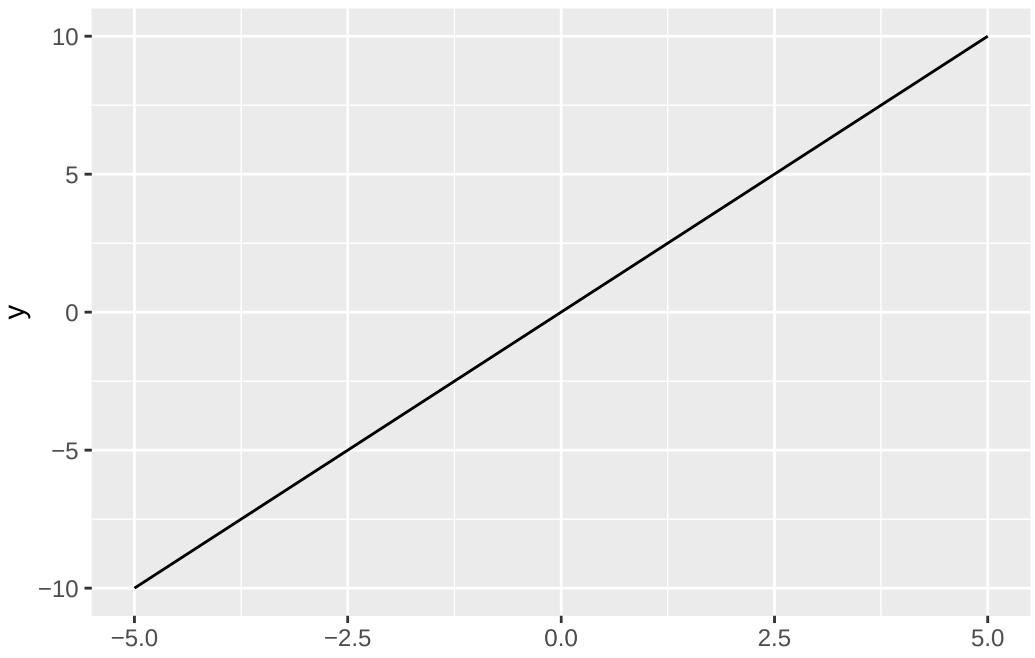
```
[1] 10 20 30 40 50
```

To graph a function, we'll use our friend `ggplot2` and `stat_function()`:

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.2     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.2     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

```
ggplot() +
  stat_function(fun = mult_by_two,
                xlim = c(-5, 5)) # domain over which we will plot the function
```



User-defined functions have endless possibilities! We encourage you to get creative and try to automate new tasks when possible, especially if they are repetitive.

Tip

Functions in R can also take non-numeric inputs. For example:

```
say_my_name <- function(my_name){paste("My name is", my_name)}
```

```
say_my_name("Inigo Montoya")
```

```
[1] "My name is Inigo Montoya"
```

While w

5.3 Common types of functions

5.3.1 Linear functions

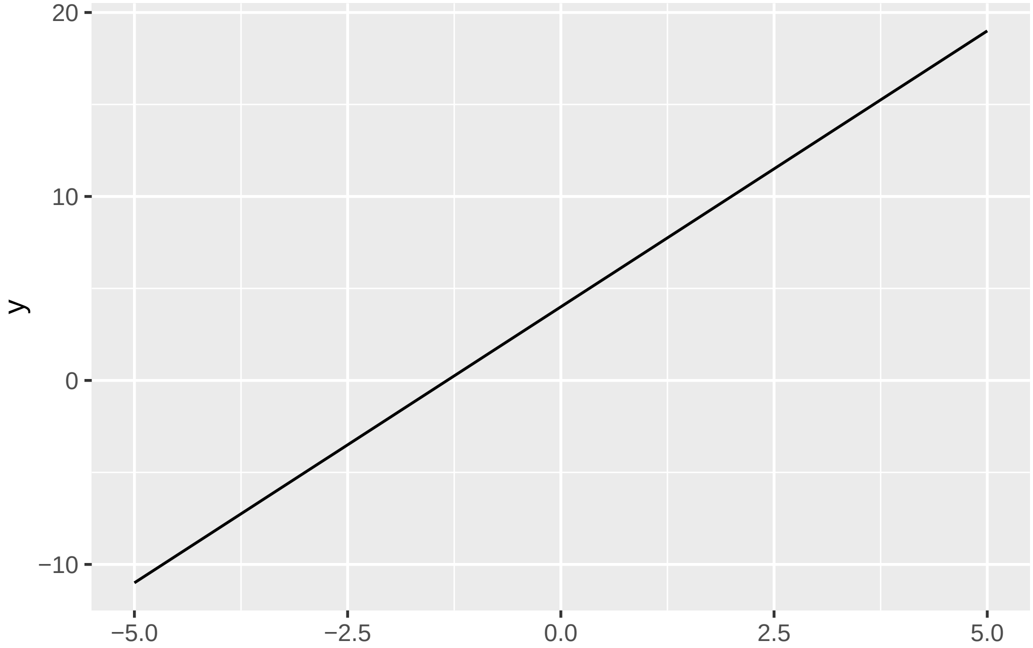
$$y = mx + b$$

Linear functions are those whose graph is a straight line (in two dimensions).

- m is the slope, or the rate of change (common interpretation: for every one unit increase in x, y increases m units).
- b is the y intercept, or the constant term (the value of y when $x = 0$).

Below is a graph of the function $y = 3x + 4$:

```
ggplot() +
  stat_function(fun = function(x){3 * x + 4}, # we don't need to create an object
                xlim = c(-5, 5))
```



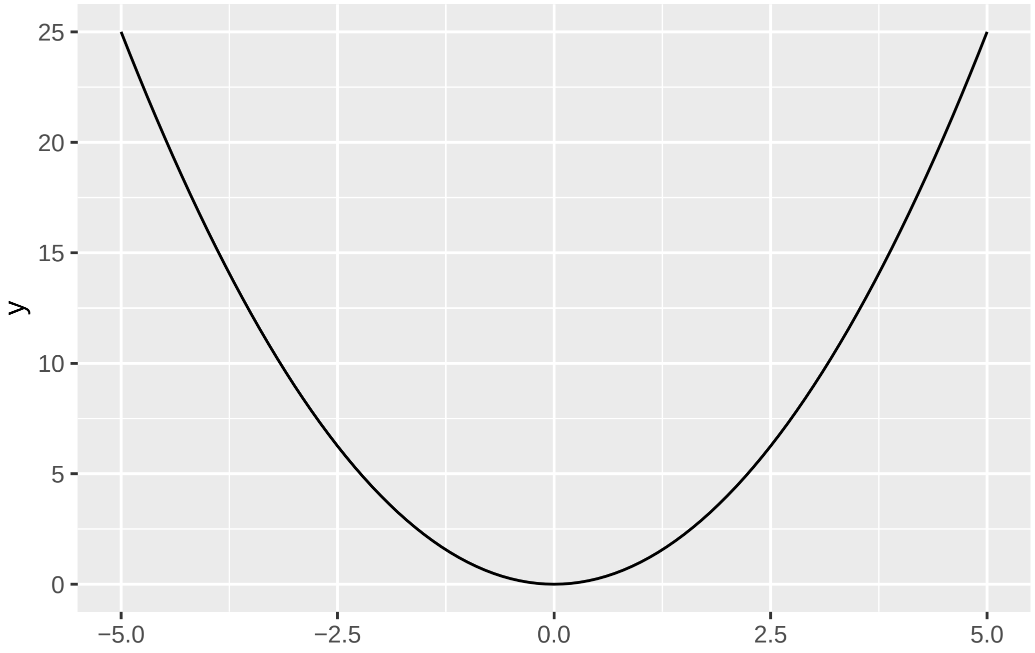
5.3.2 Quadratic functions

$$y = ax^2 + bx + c$$

- These lines have one curve.
- a, b, and c don't have well-defined meanings here.
- If a is negative, the function opens downward; if a is positive,it opens upward.
- Note that x^2 always returns positive values.

- Below is a graph of the function $y = x^2$:

```
ggplot() +
  stat_function(fun = function(x){x ^ 2},
                xlim = c(-5, 5))
```



Exercise

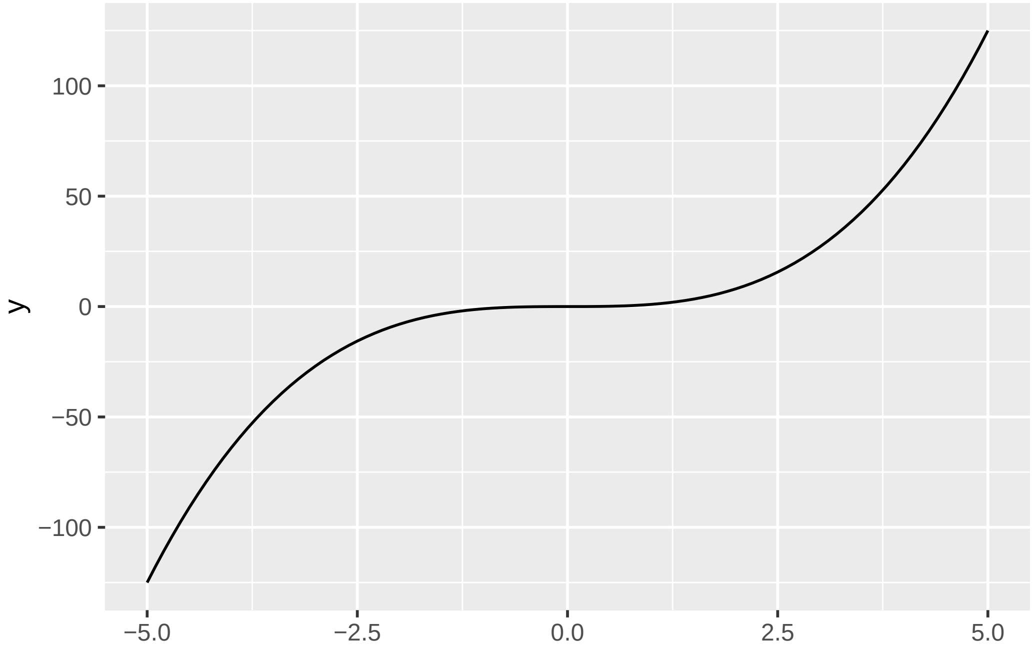
Social scientists commonly use linear or quadratic functions as theoretical simplifications of social phenomena. Can you give any examples?

5.3.3 Cubic functions

$$y = ax^3 + bx^2 + cx + d$$

- These lines (generally) have two curves (inflection points).
- a, b, c, and d don't have well-defined meanings here.
- Below is a graph of the function $y = x^3$:

```r
ggplot() +
  stat_function(fun = function(x){x ^ 3},
                xlim = c(-5, 5))
```

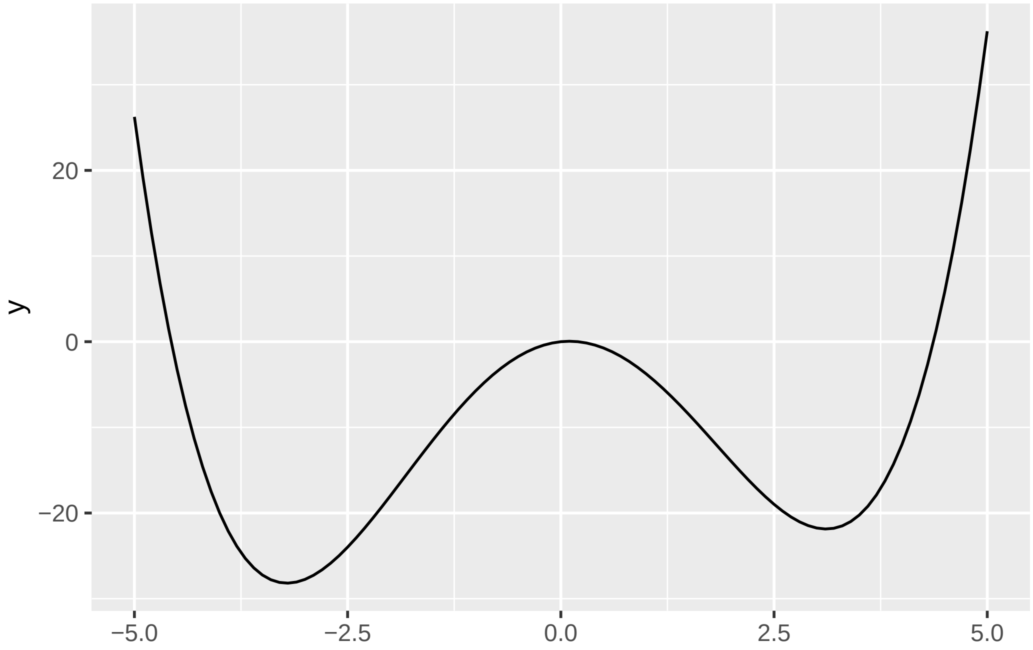


5.3.4 Polynomial functions

$$y = ax^n + bx^{n-1} + ... + c$$

- These functions have (maximum) $n - 1$ changes in direction (turning points).
- They also have (maximum) n x-intercepts. - They can be made arbitrarily precise.
- These functions have (maximum) $n - 1$ changes in direction (turning points).
- They also have (maximum) n x-intercepts.
- They can be made arbitrarily precise.
- Below is a graph of the function $y = \frac{1}{4}x^4 - 5x^2 + x$.

```r
ggplot() +
  stat_function(fun = function(x){1/4 * x ^ 4 - 5 * x ^ 2 + x},
                xlim = c(-5, 5))
```

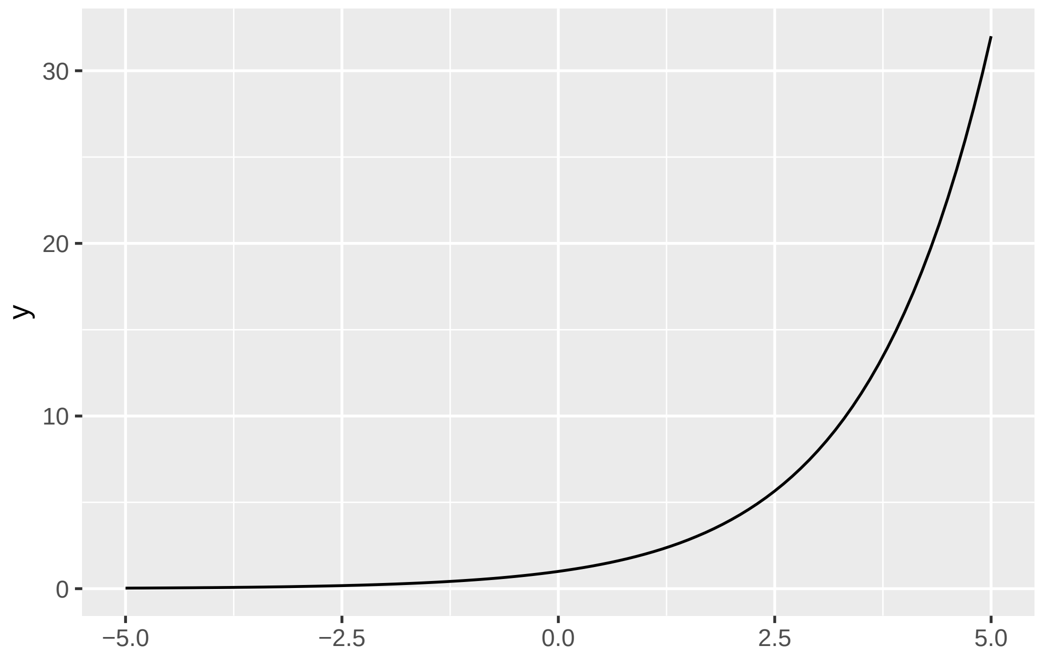


5.3.5 Exponential functions

$$y = ab^x$$

- Here our input (x), is the exponent.
- Below is a graph of the function $y = 2^x$:

```
ggplot() +
  stat_function(fun = function(x){2 ^ x},
                xlim = c(-5, 5))
```



5.4 Logarithms and exponents

5.4.1 Logarithms

- Logarithms are basically the opposite (inverse) of exponents.
- They ask how many times you must raise the base to get x.
- $log_a(b) = x$ is asking "a raised to what power x gives b?"
- $\log_3(81) = 4$ because $3^4 = 81$

> ⚠ Warning
>
> Logarithms can be *undefined*. The base cannot be 0, 1, or negative.

5.4.2 Relationships

If,

$$log_a x = b$$

then,

$$a^{log_a x} = a^b$$

and

$$x = a^b$$

5.4.3 Basic rules

$$\frac{\log_x n}{\log_x m} = \log_m n$$

$$\log_x(ab) = \log_x a + \log_x b$$

$$\log_x\left(\frac{a}{b}\right) = \log_x a - \log_x b$$

$$\log_x a^b = b \log_x a$$

$$\log_x 1 = 0$$

$$log_x x = 1$$

$$m^{\log_m(a)} = a$$

5.4.4 Natural logarithms

- We most often use natural logarithms for our purposes.
- This means $log_e(x)$, which is usually written as $ln(x)$.

> ❗ Important
>
> $e \approx 2.7183$.

- $ln(x)$ and its exponent opposite, e^x, have nice properties when we perform calculus.

5.4.5 Illustration of e

- Imagine you invest $1 in a bank and receive 100% interest for one year, and the bank pays you back once a year:
$$(1+1)^1 = 2$$
- When it pays you twice a year with compound interest:
$$(1+1/2)^2 = 2.25$$
- If it pays you three times a year:
$$(1+1/3)^3 = 2.37...$$
- What will happen when the bank pays you once a month? Once a day?
$$(1+\frac{1}{n})^n$$
- However, there is limit to what you can get.
$$\lim_{n\to\infty}(1+\frac{1}{n})^n = 2.7183... = e$$
- For any interest rate k and number of times the bank pays you t:
$$\lim_{n\to\infty}(1+\frac{k}{n})^{nt} = e^{kt}$$
- e is important for defining *exponential growth*. Since $ln(e^x) = x$, the natural logarithm helps us turn exponential functions into linear ones.

ℹ Exercise

Solve the problems below, simplifying as much as you can.

$$log_{10}(1000)$$

$$log_2(\frac{8}{32})$$

$$10^{log_{10}(300)}$$

$$ln(1)$$

$$ln(e^2)$$

$$ln(5e)$$

5.4.6 Logarithms in R

By default, R's `log()` function computes natural logarithms:

```r
log(100)
```

```
[1] 4.60517
```

We can change this behavior with the `base =` argument:

```r
log(100, base = 10)
```

```
[1] 2
```

5.5 Composite functions (functions of functions)

- Functions can take other functions as inputs, e.g., $f(g(x))$.
- This means that the outside function takes the output of the inside function as its input.
- Say we have the exterior function f(x)=x^2 and the interior function g(x)=$x - 3$.
- Then if we want f(g(x)), we would subtract 3 from any input, and then square the result.
- We write this $(x - 3)^2$, not $x^2 - 3$.

6 Calculus

- Calculus is about dealing with infinitesimal values.
- We are going to focus on two big ideas:
  - Derivatives
  - Integrals

6.1 Derviatives

- "Derivative" is just a fancy term for slope.
- Slope is the rate of change $\frac{\delta y}{\delta x}$ or $\frac{dy}{dx}$.
- Specifically, the derivative is the *instantaneous* rate of change.
- We need slope for our statistics, which are all about fitting lines.
- We also need slope for taking maxima and minima.
- The equation for a line is $y = mx + b$. What is its slope?

6.1.1 Calculating derivatives

- Slope is rise over run, which is $\dfrac{f(x + \Delta x) - f(x)}{\Delta x}$
- To see why, consider the slope of a line connecting two points:

$$m = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

- We can define $x_2 = x_1 + \Delta x$ (or equivalently $\Delta x = x_2 - x_1$)

$$m = \frac{f(x_1 + \Delta x) - f(x_1)}{\Delta x}$$

- As we've seen, for a curve, we need to be infinitely close for our line's defining points, yielding

$$\lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- This gives us this instantaneous slope (rate of change) of a function at every point on its domain. The above equation is the definition of the derivative.

6.1.2 Notation

- $\frac{d}{dx}f(x)$ is read "The derivative of f of x with respect to x."
  - You can also say "The instantaneous rate of change in f of x with respect to x."
- Lagrange's prime notation: $f'(x)$ (read: "f prime x") is the derivative of $f(x)$.
- If $y = f(x)$, $\frac{dy}{dx}$ is "The derivative of y with respect to x".
  - The variable with respect to which we're differentiating is the one that appears in the denominator.

⚠ Warning

Do not try to cancel out the d's, no matter how tempting it is.

Examples

- What is $\dfrac{d(x^2)}{dx}$?
  - x^2
  - $2x^{2-1}$ -$2x$
- What is $\frac{d(4x^3)}{dx}$?
  - $4x^3$
  - $4 * 3x^{3-1}$
  - $12x^2$

Practice

- Take the derivative of each of the following:

$$x^3$$

$$3x^2$$

$$60x^{11}$$

$$x$$

$$\frac{4}{x^2}$$

$$9\sqrt{x}$$

$$6x^{5/2}$$

$$11,596,232$$

Practice

- Evaluate the derivatives at $x = 2$ and $x = -1$

$$x^3$$

$$3x^2$$

$$60x^{11}$$

$$x$$

$$\frac{4}{x^2}$$

$$9\sqrt{x}$$

$$6x^{5/2}$$

$$11,596,232$$

Practice

- Take the derivative of each of the following:

$$x^3$$

$$3x^2$$

$$60x^{11}$$

$$x$$

$$\frac{4}{x^2}$$

$$9\sqrt{x}$$

$$6x^{5/2}$$

$$11,596,232$$

- Evaluate the derivatives at $x = 2$ and $x = -1$

$$x^3$$

$$3x^2$$

$$60x^{11}$$

$$x$$

$$\frac{4}{x^2}$$

$$9\sqrt{x}$$

$$6x^{5/2}$$

$$11,596,232$$

6.1.3 Special functions

- A few functions have particular rules:

$$\frac{d(ln(x))}{dx} = \frac{1}{x}$$

$$\frac{d(log_b(x))}{dx} = \frac{1}{x * ln(b)}$$

$$\frac{d(e^x)}{dx} = e^x$$

$$\frac{d(a^x)}{dx} = a^x ln(a)$$

$$\frac{dy}{dx}c = 0$$

$$\frac{d(x^x)}{dx} = x^x(1 + ln(x))$$

6.1.4 Derivatives with addition and substraction

- This is perhaps the rasiest rule to remember:

$$\frac{d(f(x) \pm g(x))}{dx} = f'(x) \pm g'(x)$$

Practice

- Take the derivative of each of the following:

$$x^2 + x + 5$$

$$x^4 - 4x^3 + 5x^2 + 8x - 6$$

$$3x^5 - 6x^2$$

$$5x^2 + 8\sqrt{x} - \frac{1}{x}$$

$$ln(x) + 5e^x - 4x^3$$

6.2 Advanced rules

6.2.1 Product rule

- This rule is more complicated:

$$\frac{d(f(x) \times g(x))}{dx} = f'(x)g(x) + g'(x)f(x)$$

- Example:

$$\begin{gathered} 2x \times 3x \\ f(x) = 2x \\ g(x) = 3x \\ f'(x) = 2 \\ g'(x) = 3 \\ f'(x)g(x) + g'(x)f(x) = 2 \times 3x + 3 \times 2x \\ \frac{d(2x \times 3x)}{dx} = 6x + 6x = 12x \end{gathered}$$

Practice

- Take the derivative of each of these:

$$\begin{gathered} x^3 * x \\ e^x * x^2 \\ ln(x) * x^{-3} \end{gathered}$$

Reminder!

$$\frac{d(f(x) * g(x))}{dx} = f'(x)g(x) + g'(x)f(x)$$

6.2.2 Quotient rule

$$\frac{d\frac{f(x)}{g(x)}}{dx} = \frac{f'(x)g(x) - g'(x)f(x)}{[g(x)]^2}$$

- If you're having trouble with this, just apply the product rule to the following:

$$\frac{d[f(x) * g^{-1}(x)]}{dx}$$

Reminder!

$$\frac{d\frac{f(x)}{g(x)}}{dx} = \frac{f'(x)g(x) - g'(x)f(x)}{[g(x)]^2}$$

6.2.3 Chain rule

$$\frac{d[f(g(x))]}{dx} = f'(g(x)) * g'(x)$$

- Let's take the derivative of a function of a function:

$$\frac{d[ln(x^2)]}{dx}$$

$$f(x) = ln(x)$$

$$g(x) = x^2$$

$$f'(x) = \frac{1}{x}$$

$$g'(x) = 2x$$

$$\frac{1}{x^2} * 2x = \frac{2}{x}$$

Practice

- Take the derivative of each of the following:

$$(3x^4 - 8)^2$$

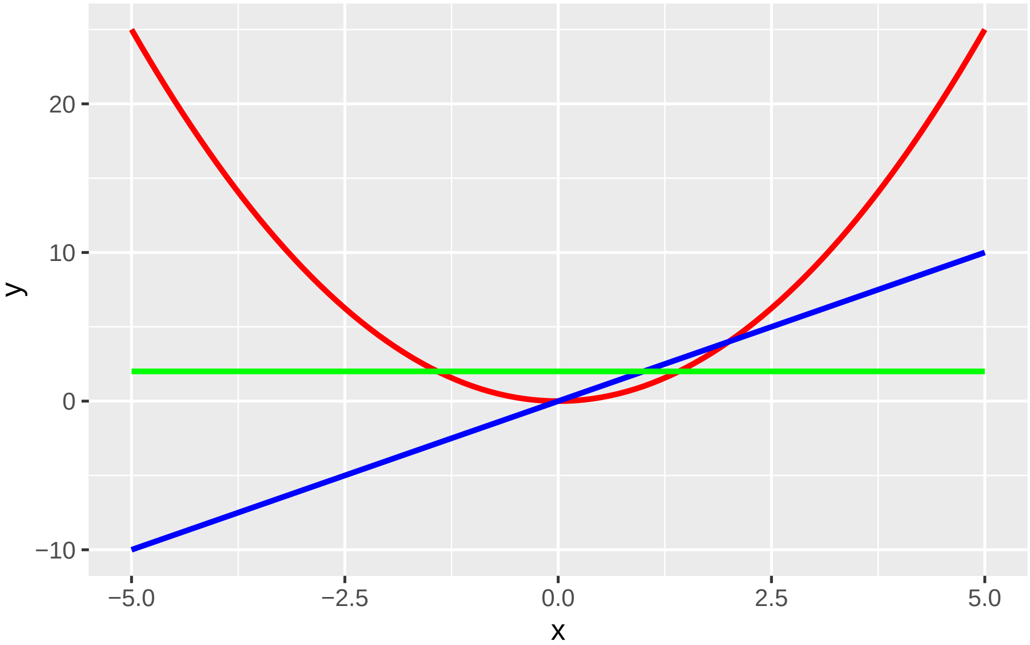
$$e^{x^2}$$

Reminder!

$$\frac{d(f(g(x))}{dx} = f'(g(x)) * g'(x)$$

6.2.4 Second derivative

- Same process as taking single derivative, except input for second derivative is output from first.
- Second derivative tells us whether the slope of a function is increasing, decreasing, or staying the same at any point x on the function's domain.
- Example: driving a car.
  - $f(x)$ = distance traveled at time x
  - $f'(x)$ = speed at time x
  - $f''(x)$ = acceleration at time x
- Let's graph $f(x) = x^2$, $f'(x)$, and $f''(x)$.



$$\frac{d^2(x^4)}{dx^2} = f''(x^4)$$

- First, we take the first derivative:

$$f'(x^4) = 4x^3$$

- Then we use that output to take the second derivative:

$$f''(x^4) = f'(4x^3) = 12x^2$$

Practice

Take the second derivative of the following functions:

$$x^5$$

$$6x^2$$

$$4ln(x)$$

$$3x$$

$$4x^{3/2}$$

6.3 Differentiable and Continuous Functions

- Informally: A function is continuous at a point if its graph has no holes or breaks at that point
- Formally: A function is continuous at a point a if:

$$\lim_{x \to a} f(x) = f(a)$$

- Continuity requires 3 conditions to hold:
  - $f(a)$ is defined (a is in the domain of f)
  - $\lim_{x \to a} f(x)$ exists
  - $\lim_{x \to a} f(x) = f(a)$ (the value of f equals the limit of f at a)
- Differentiable:
  - If $f'(x)$ exists, f is differentiable at x.
  - If f is differentiable at every point of an open interval I, f is differentiable on I.
  - Graph must have a (non-vertical) tangent line at each point, be relatively smooth, and not contain any breaks, bends, or cusps.
- If a function is differentiable at a point, it is also continuous at that point.
- If a function is continuous at a point, it is *not* necessarily differentiable at that point.

6.3.1 When is f not differentiable?

When does $f'(x)$ not exist?

- When the function is discontinuous at that point.
  - Jump or break in the graph.
- There are different slopes approaching the point from the left and from the right.
  - Corner point
- When the graph of the function has a vertical tangent line at that point.
  - Cusp
  - Vertical inflection point

6.4 Extrema and optimization

Optimization lets us find the minimum or maximum value a function takes.

- Formal theory
  - Utility maximization, continuous choices
- Ordinary Least Squares (OLS)
  - Focuses on *minimizing* the squared errors between observed data and values predicted by a regression.
- Maximum Likelihood Estimation (MLE)
  - Focuses on *maximizing* a likelihood function, given observed values.

6.4.1 Extrema

- Informally, a maximum is just the highest value a function takes, and a minimum is the lowest value.
- Easy to identify extrema (maxima or minima) intuitively by looking at a graph of the function.
  - Maxima are high points ("peaks")
  - Minima are low points ("valleys")
- Extrema can be *local* or *global*.

6.4.2 Identifying extrema

- The derivative of a function gives the rate of change.
- When the derivative is zero (or fails to exist), the function has *usually* reached a (local) maximum or minimum.
- At a maximum, the function must be increasing before the point and decreasing after it.
- At a minimum, the function must be decreasing before the point and increasing after it.
- We'll start by identifying points where this is the case ("critical points" or "stationary points").

> ℹ Note
>
> A point where $f'(x) = 0$ or $f'(x)$ does not exist is called a *critical point* (or *stationary point*). Local extrema occur at critical points, but not all critical points are extrema. For instance, sometimes the graph is changing between concave and convex ("inflection points"). Sometimes the function is not differentiable at that point for other reasons.

- We can find the local maxima and/or minima of a function by taking the derivative, setting it equal to zero, and solving for x (or whatever).

$$f'(x) = 0$$

- This gives us the first-order condition (FOC).

6.4.3 Minimum or maximum?

BUT we don't know if we've found a maximum or minimum, or even if we've found an extremum or just an inflection point.

6.4.4 Second derivatives

- The second derivative gives us the rate of change of the rate of change of the original function. It tells us whether the slope is getting larger or smaller.

$$f(x) = x^2$$
$$f'(x) = 2x$$
$$f''(x) = 2$$

Second Derivative Test - Start by identifying $f''(x)$

- Substitute in the stationary points (x^*) identified from the FOC.
  - $f''(x^*) > 0$ we have a local minimum
  - $f''(x^*) < 0$ we have a local maximum
  - $f''(x^*) = 0$ we (may) have an inflection point - need to calculate higher-order derivatives (don't worry about this now)

Collectively these give use the Second-Order Condition (SOC).

6.4.5 Local vs. Global Extrema

- To find the minimum/maximum on some interval, compare the local min/max to the value of the function at the interval's endpoints.
- To find the global minimum/maximum, check the function's limits as it approaches $+\infty$ and $-\infty$.
- Extreme value theorem: if a real-valued function f is continuous on the closed interval [a,b], then f must attain a (global) maximum and a (global) minimum.

6.5 Partial derivatives

- We can take the derivative with respect to different variables.
- For a function $fy = (x, z) = xz$, we might want to know how the function changes with x:

$$\frac{\partial}{\partial_x} f(x, y) = \frac{\partial_y}{\partial_x} = \partial_x f$$

- We treat all other variables as constants and take derivative with respect to the variable of interest (here x).

How do we take a partial derivative?

Treat all other variables as *constants* and take *derivative* with respect to the variable of interest.

From our earlier example:

$$y = f(x, z) = xz$$

$$\frac{\partial_y}{\partial_x} = ?$$

$$y = f(x, z) = xz$$

$$\frac{\partial_y}{\partial_x} = z$$

Why? Because the partial derivative of xz with respect to x treats z as a constant.

What is $\frac{\partial_y}{\partial_z}$?

6.5.1 Application

- $\frac{\partial(x^2y+xy^2-x)}{\partial x}$
- We apply the addition rule to take the derivative of each term with respect to x.
- $\frac{\partial(x^2y)}{\partial x}+\frac{\partial(xy^2)}{\partial x}+\frac{\partial(-x)}{\partial x}$
- $2xy+y^2-1$

- $\frac{\partial(x^2y+xy^2-x)}{\partial y}$
- We apply the addition rule to take the derivative of each term with respect to y
- $\frac{\partial(x^2y)}{\partial y}+\frac{\partial(xy^2)}{\partial y}+\frac{\partial(-x)}{\partial y}$
- x^2+2xy

Practice

Take the partial derivative with respect to x and to y of the following functions. What would the notation for each look like?

$$3xy-x$$
$$ln(xy)$$
$$x^3+y^3+x^4y^4$$
$$e^{xy}$$

6.6 Integrals

6.6.1 Area under a curve

- Often we want to find the area under a curve.
- Sometimes finding the area is easy. What's the area under the curve between $x=-1$ and $x=1$ for this function?

$$f(x)=\begin{cases}\frac{1}{3} & \text{for } x\in[0,3]\\ 0 & \text{otherwise}\end{cases}$$

> “Hint
>
> We can draw this and look at the graph. Remember:
>
> $$area = \ell * w$$

- Normally, finding the area under a curve is much harder. But this is basically the question behind integration.

6.6.2 Integrals as summation

- We are already familiar with summation notation.

$$\sum_{i=1}^{n} i$$

- This only works when we have discrete values to add.
  - When we need to add continuously, we have to use something else. Specifically, integrals.

6.6.3 Definite integrals

- Let's say we have a function

  $$y = x^2$$

  And we want to find the area under the curve from $x = 0$ to $x = 1$.
- To find the area we're interested in here, we can use the definite integral.
- Generally speaking, the notation looks like this:

$$\int_{x=a}^{b} f(x), dx$$

- Here a is the lower limit of integration, b is the upper limit of integration, our function $f(x)$ is our integrand, and x is our variable of integration.

- For our question, we're looking for the following:

$$\int_{x=0}^{1} f(x) dx$$

- This will give us a real number denoting the area under the curve of our function ($y = x^2$) between $x = 0$ and $x = 1$.
- If f is continuous on $[a, b]$ or bounded on $[a, b]$ with a finite number of discontinuities, then f is integrable on $[a, b]$.

6.6.4 Indefinite integrals

- The *indefinite* integral, also known as the *anti-derivative*, $F(x)$ is the inverse of the function $f'(x)$.

$$F(x) = \int f(x)\ dx$$

- This means if you take the derivative of $F(x)$, you wind up back at $f(x)$.

$$F' = f \text{ or } \frac{dF(x)}{dx} = f(x)$$

- This process is called anti-differentiation, or indefinite integration.
    - While the definite integral gives us a real number (the total area under a curve), the indefinite integral gives us a function.
- We need the concept of indefinite integrals to help us solve definite integrals.

6.6.5 Solving definite integrals

- The easiest way to calculate definite integrals, known as the "fundamental theorem of calculus," is shown below:

$$\int_a^b f(x)\ dx = F(b) - F(a) = F(x)\bigg|_a^b$$

- First we determine the antiderivative (indefinite integral) of $f(x)$ (and represent it $F(x)$), substitute the upper limit first and then the lower limit one by one, and subtract the results in order.

6.6.6 Constants

Note

C in the following slides is the called the "constant of integration." We need to add it when we define *all* antiderivatives (integrals) of a function because the anti-derivative "undoes" the derivative.

Remember that the derivative of any constant is zero. So if we find an integral $F(x)$ whose derivative is $f(x)$, adding (or subtracting) any constant will give us another integral $F(x) + C$ whose derivative is *also* $f(x)$.

6.6.7 Rules of integration

$$\int_a^a f(x)\ dx = 0$$

$$\int_a^b f(x)\ dx = -\int_b^a f(x)dx$$

$$\int a\ dx = ax + C \text{ where } a \text{ is a constant}$$

$$\int af(x)dx = a\int f(x)\ dx \text{ where } a \text{ is a constant}$$

6.6.8 More rules

$$\int (f(x) + g(x))\ dx = \int f(x)dx + \int g(x)dx$$

$$\int x^n dx = \frac{x^{n+1}}{n+1} + C \qquad \forall n \neq -1$$

$$\int x^{-1} dx = \ln|x| + C$$

6.6.9 Solving the problem

Remember our function $y = x^2$ and our goal of finding the area under the curve from $x = 0$ to $x = 1$.

- Find the indefinite integral, $F(x)$
  - $\displaystyle\int x^2 \, dx$
  - $\dfrac{x^3}{3} + C$

! Important

We use the “power rule” of integration, which is the following:

$$\int x^n dx = \frac{x^{n+1}}{n+1} + C \qquad \forall n \neq -1$$

- Evaluate at our lowest and highest points, $F(0)$ and $F(1)$.
  - $F(0) = 0$
  - $F(1) = \dfrac{1}{3}$
  - Technically $0 + C$ and $\dfrac{1}{3} + C$, but the C’s will fall out in the next step
- Calculate $F(1) - F(0)$

$$\frac{1}{3} - 0 = \frac{1}{3}$$

Practice — indefinite integrals

$$\int x^2 \, dx$$

$$\int 3x^2 \, dx$$

$$\int x \, dx$$

$$\int 3x^2 + 2x - 7 \, dx$$

$$\int \frac{2}{x}\ dx$$

Practice — definite integrals

$$\int_1^7 x^2\ dx$$

$$\int_1^{10} 3x^2\ dx$$

$$\int_7^7 x\ dx$$

$$\int_1^5 3x^2 + 2x - 7\ dx$$

$$\int_1^e \frac{2}{x}\ dx$$

7 Probability

7.1 What is probability?

- Frequency with which an event occurs.
  - Typically:

$$Pr(A) = P(A) = \pi(A) = \frac{\text{Number of ways an event can occur}}{\text{Total number of possible outcomes}}$$

- Probability predicts real-world events using theoretical quantities.
  - Formally, it assigns a likelihood of occurrence to each event in sample space
  - We use the probability space triplet (Ω, S, P), which are the sample space, event space, and probability mapping, respectively.
- We can consider probability as a function that maps $\Omega \to \mathbb{R}$.
- We can conceive it in terms of relative frequency or subjective belief.

7.2 Kolmogorov's axioms

- $Pr(S_i) \in \mathbb{R},\ \ 1 \geq Pr(S_i) \geq 0 \qquad \forall S_i \in S$
  - Where S is the event space, S_i are events.
  - Probabilities must be non-negative.
- $Pr(\Omega) = 1$
  - Where Ω is the sample space.
  - Something has to happen.
  - Probabilities sum/integrate to 1.
- $Pr\left(\bigcup_{i=1}^{\infty} S_i\right) = \sum_{i=1}^{\infty} Pr(S_i) \iff Pr(S_i \cap S_j) = 0 \ \ \forall i \neq j$
  - The probability of disjoint (mutually exclusive) sets is equal to the sum of their individual probabilities.

7.3 Some definitions

- **Random variable**: a variable whose value is determined by the outcome of a random process.
  - Sometimes also called a *stochastic variable*.
  - May be discrete or continuous.
- **Distribution** (of a random variable): the set of values the variable might take.
  - Probability mass function / probability density function defines the probability with which each value occurs.
  - Always sums / integrates to 1.
- **Realization** (of a random variable): a particular value taken by the variable.

- **Population**: the entire set of objects (people, cases, etc.) in which we are interested.
  - Often denoted N.
- **Sample**: a subset of the population we can observe, from which we try to make generalizations about the population.
  - Often denoted n.
- **Frequency distribution**: a count of how often a variable takes each of its possible values.
  - The number of members of a sample that take each value of a variable.
- **Independent random variables**: two variables are statistically independent if the value of one does not affect the value of the other.
  - Formally, $Pr(A \cap B) = Pr(A)Pr(B)$

7.4 Discrete probability

- A sample space in which there are a (finite or infinite) countable number of outcomes
- Each realization of random process has a discrete probability of occurring.
  - $f(X = x_i) = P(X = x_i)$ is the probability the variable takes the value x_i.

7.4.1 Probability Mass Function (PMF)

Probability of each occurrence encoded in probability mass function (PMF)

- $0 \leq f(x_i) \leq 1$
  - Probability of any value occurring must be between 0 and 1.
- $\sum_x f(x_i) = 1$
  - Probabilities of all values must sum to 1.

7.4.2 Discrete distribution

- What's the probability that we'll roll a 3 on one die roll:

$$Pr(y = 3) = \frac{1}{6}$$

- If one roll of the die is an "experiment."
- We can think of a 3 as a "success."
- $Y \sim Bernoulli\left(\frac{1}{6}\right)$
- Fair coins are $\sim Bernoulli(.5)$, for example.
- More generally, $Bernoulli(\pi)$.
  - π represents the probability of success.

- Drawing a specific card from a deck:

$$Pr(y = \text{ace of spades}) = \frac{1}{52}$$

- Drawing any card with a specific value from a deck:

$$Pr(y = ace) = \frac{4}{52}$$

- Getting a specific value on two dice rolls:

$$Pr(y = 8) = \frac{5}{36}$$

- We can express the probability mass function in tabular format or in a graph.

7.5 Continuous probability

- What happens when our outcome is continuous?
- There are an infinite number of outcomes.
- This makes the denominator of our fraction difficult to work with.
- The probability of the whole space must equal 1.
- Even if all events are equally likely, $\frac{1}{\infty} = 0$

7.5.1 Basics

- The domain may not span -∞ to ∞.
  - Even space between 0 and 1 is infinite.
- The domain is defined as the area under the probability density function.
- Two common examples are the uniform and bell curves.

7.5.2 Probability Density Function (PDF)

- Similar to PMF from before, but for continuous variables.
- Gives the probability a value falls within a particular interval
  - $P[a \leq X \leq b] = \int_a^b f(x)\,dx$
  - Total area under the curve is 1.
  - $P(a < X < b)$ is the area under the curve between a and b (where $b > a$).

7.6 Cumulative Density Function (CDF)

7.6.1 Discrete

- Cumulatve density function is probability X will take a value of x or lower.
- PDF is written $f(x)$, and CDF is written $F'(x)$.

$$F_X(x) = Pr(X \leq x)$$

- For discrete CDFs, that means summing up over all values.
- What is the probability of rolling a 6 or lower with two dice? $F(6) = ?$

7.6.2 Continuous

- We can't sum probabilities for continuous distributions (remember the 0 problem).
- Solution: integration

$$F_Y(y) = \int_{-\infty}^{y} f(y)dy$$

- Examples of uniform distribution.

7.7 Statistics

7.7.1 Introduction

- While probability allows us to make predictions about events using distributions, statistics uses events to make estimates about distributions and variables.
- It is the process of learning from data.
- A statistic is a summary of data, capturing some theoretically-relevant quantity.
- Broad categories of numerical and categorical.

7.7.2 Univariate statistics

- These measure a single variable.
- Readily expressed in graphical form.
- Common examples:
  - Central tendency (mean, median, and mode)
  - Variance

7.7.3 Examples of univariate statistics

- The mean ($\bar{x}$) is calculated by summing the data, then dividing by the number of observations:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- The median is found by ordering the observations from highest to lowest and finding the one in the middle.

- The mode is the most common number.

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 6 & 7 & 8 & 9 \end{bmatrix}$$

- What are the mean, median, and mode of x?

7.7.4 Measures of central tendency

- Mean balances values on either side.
- Median balances observations on either side.
- Mode finds the most typical observation.
- Which is the best? Like most of what you'll learn in statistics, it depends.

7.7.5 Deviations from central tendency

- Consider two data sets:

$$x = \begin{bmatrix} 1 & 1.5 & 2 & 2.5 & 5.5 & 8.5 & 9 & 9.5 & 10 \end{bmatrix}$$

$$y = \begin{bmatrix} 4.5 & 4.8 & 5 & 5.3 & 5.5 & 5.7 & 6 & 6.2 & 6.5 \end{bmatrix}$$

- What is the mean of each?
- What is the median of each?
- Are they similar distributions?

7.7.6 Variance

- We use variance to measure the spread of a single variable.
- Formally defined as the squared deviation from the mean (μ).
- For discrete random variables, it is written $Var(x) = \sigma^2 = \dfrac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$
- For continuous random variables, it is written $Var(x) = \sigma^2 = \int (x - \mu)^2 f(x)\ dx$

7.7.7 Standard deviation

- Sometimes variance doesn't make sense, either mathematically or conceptually.
  - Not always clear how to interpret "squared deviation from the mean."
- Instead, will frequently see standard deviation, which is square root of variance.
- It is written σ.

7.8 Bivariate statistics

7.8.1 Covariance

- While measures of central tendency and variance/standard deviation provide useful summaries of a single variable, they don't provide insights into relationships between variables.
- For that, we need bivariate statistics.
- Most common and straightforward is covariance.

- Colloquially, can think of covariance as measure of linear deviation from mean.
- When values from one variable are above their mean, are values from the other above or below their mean?
- Put another way, if I told you the value of x was high, would you expect values of y to be high or low?
- Formally, it is written as:

$$cov(X, Y) = E(X - E(X))(Y - E(Y)) = E(XY) - E(X)E(Y)$$

- It is important to note that the magnitude is meaningless; only the direction is interpretable.

7.8.2 Correlation

- Correlation is a normalized measure of covariance
- It is calculated as:

$$\rho_{X,Y} = \frac{cov(X, Y)}{\sigma_X \sigma_Y}$$

- It varies between -1 and 1.
- What is correlation of two independent variables?

7.9 Regression

7.9.1 Ordinary least squares

- Ordinary least squares regression (OLS) is probably the most widely-used model in political science.
- It is all about drawing a line through data.

- This allows us to evaluate the relationship (the association) between x on y.
- The dependent variable, y, must be continuous, generally speaking.
- The main question is which line to draw.

Line and equation ($\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$) on board

7.9.2 Residuals

- In basically any set of data, no line can pass through every point (observation).
- We will always have make some error in predicting values.
- The error between the line and some point is referred to as the residual.
- If we refer to our predicted value as $\hat{y}$, then we can calculate the residual for each observation with the following equation:

$$e_i = y_i - \hat{y}_i$$

7.9.3 Finding the right line

- OLS determines the "best" line by minimizing the sum of squared residuals.
- Plug in all the values for the slope amd intercept and calculate the sum of squared residuals for these infinity combinations.
- That is a lot of work.
- The best solution turns out to be calculus.
- We want to minimize the sum of squared residuals with respect to our β's.

8 Simulations

9 Text analysis

9.1 Strings

- In R, a piece of text is represented as a sequence of characters (letters, numbers, and symbols).
- A string is a sequence of characters, which is used for storing text.
  - For example, “methods” is a string that includes characters: m, e, t, h, o, d, s.
- Creating strings is very straightforward in RStudio. We assign character values to a variable, being sure to enclose the character values (the text) in double or single quotation marks.
  - We can create strings of single words, or whole sentences if we so wish.

```
string1 <- "camp"
string1
```

```
[1] "camp"
```

```
string2 <- "I love methods camps."
string2
```

```
[1] "I love methods camps."
```

- We can also create a vector of strings.

```
string3 <- c("I", "love", "methods", "camp", ".")
string3
```

```
[1] "I"       "love"    "methods" "camp"    "."
```

9.2 String manipulation

- Often, strings, and more broadly text, contain information that we want to extract for the purpose of our research.
    - For example, perhaps we wanted to count the number of times a certain country was mentioned during the U.S. President's annual State of the Union Address.
- For tasks such as these, we can use regular expressions (also known as 'regex'), which search for one or more specified pattern of characters.
    - These patterns can be exact matches, or more general.

```
test <- "test"
```

- Regular expressions can be used to:
    - Extract information from text.
    - Parse text.
    - Clean/replace strings.

> **Note**
>
> Fortunately, the syntax for regular expressions is relatively stable across all programming languages (e.g., Java, Python, R).

9.3 Stringr

```
library(stringr)
```

- `stringr` comes with the `tidyverse` and provides functions for both (a) basic string manipulations and (b) regular expression operations. Some basic functions are listed below:

Function	Description
`str_c()`	string concatenation
`str_length()`	number of characters
`str_sub()`	extracts substrings
`str_dup()`	duplicates characters
`str_trim()`	removes leading and trailing whitespace

Function	Description
str_pad()	pads a string
str_wrap()	wraps a string paragraph
str_trim()	trims a string

9.3.1 Basic string manipulation

- Let's try some examples of basic string manipulation using `stringr`:

```r
my_string <- "I know people who have seen the Barbie movie 2, 3, even 4 times!"
my_string
```

```
[1] "I know people who have seen the Barbie movie 2, 3, even 4 times!"
```

- One common thing we want to do with strings is lowercase them:

```r
lower_string <- tolower(my_string)
lower_string
```

```
[1] "i know people who have seen the barbie movie 2, 3, even 4 times!"
```

- We can also combine (concatenate) strings using the `str_c` command:

```r
my_string2 <- "I wonder if they have seen Oppenheimer, too."
cat_string <- str_c(my_string, my_string2, sep = " ")
cat_string
```

```
[1] "I know people who have seen the Barbie movie 2, 3, even 4 times! I wonder if they have
```

- We can also split up strings on a particular character sequence.
  - ! denotes where split occurs and deletes the "!" The double bracket instructs to grab the first part of the split string.

```r
my_string_vector <- str_split(cat_string, "!")[[1]]
my_string_vector
```

```
[1] "I know people who have seen the Barbie movie 2, 3, even 4 times"
[2] " I wonder if they have seen Oppenheimer, too."
```

- We can also find which strings in a vector contain a particular character or sequence of characters.
  - The `grep` (Globally search for Regular Expression and Print) command will return any instance that (partially) matches the provided pattern.
  - Closely related to the `grep()` function is the `grepl()` function, which returns a logical for whether a string contains a character or sequence of characters.

```
grep("Barbie",
     cat_string,
     value = FALSE,
     ignore.case = TRUE)
```

```
[1] 1
```

```
# To search for some special characters (e.g., "!"), you need to "escape" it
grep("\\!", cat_string, value = TRUE)
```

```
[1] "I know people who have seen the Barbie movie 2, 3, even 4 times! I wonder if they have s
```

```
grepl("\\!", cat_string)
```

```
[1] TRUE
```

- The `str_replace_all` function can be used to replace all instances of a given string, with an alternative string.

```
str_replace_all(cat_string, "e","_")
```

```
[1] "I know p_opl_ who hav_ s__n th_ Barbi_ movi_ 2, 3, _v_n 4 tim_s! I wond_r if th_y hav_ s
```

- We can also pull out all sub-strings matching a given string argument.
  - This becomes especially useful when we generalize the patterns of interest.

```
str_extract_all(cat_string, "have")
```

```
[[1]]
[1] "have" "have"
```

```
str_extract_all(cat_string,"[0-9]+")[[1]]
```

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

```
#   The square brackets define a set of possibilities.
#   The "0-9" says the possibilities are any digit from 0 to 9.
#   The "+" means "one or more of the just-named thing"

str_extract_all(cat_string,"\\d+")[[1]] #   Instead of 0-9, we can just say "\\d" for digi
```

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

```
str_extract_all(cat_string,"[a-zA-Z]+")[[1]] # letters
```

```
 [1] "I"           "know"        "people"      "who"         "have"
 [6] "seen"        "the"         "Barbie"      "movie"       "even"
[11] "times"       "I"           "wonder"      "if"          "they"
[16] "have"        "seen"        "Oppenheimer" "too"
```

```
str_extract_all(cat_string,"\\w+")[[1]] # "word" characters
```

```
 [1] "I"           "know"        "people"      "who"         "have"
 [6] "seen"        "the"         "Barbie"      "movie"       "2"
[11] "3"           "even"        "4"           "times"       "I"
[16] "wonder"      "if"          "they"        "have"        "seen"
[21] "Oppenheimer" "too"
```

9.4 Simple text analysis

- We can use the tidytext package to conduct some basic text analysis using tidy data principles.
- As Wickham 2014 reminds us, tidy data has a specific structure:
  - Each variable is a column.
  - Each observation is a row.
  - Each type of observational unit is a table.

- We can thus define the format as a table with one-token-per-row.
  - A token is a unit of text (e.g., word) that we use for analysis. Tokenization is the process of turning text into tokens.
- As Silge and Robinson (2017) remind us, it is important to contrast this structure with the alternative ways that text is often structured and stored in text analysis:
  - String: Text can be stored as strings, i.e., character vectors. Text data is often first read into memory in this form.
  - Corpus: These objects usually contain raw strings annotated with metadata and details.
  - Document-term matrix: This sparse matrix describe a collection (i.e., a corpus) of documents with one row for each document and one column for each term. The value in the matrix is typically word count or tf-idf (term frequency-inverse document frequency).
- Let's try an example. To create a tidy text dataset, we need to first put some text into a data frame.
  - We print out each line as a "tibble," which has a convenient print method that does not convert strings to factors or use row names.

```
library(dplyr)

Barbie <- c("I'm a Barbie girl in the Barbie world",
            "Life in plastic, it's fantastic",
            "You can brush my hair, undress me everywhere",
            "Imagination, life is your creation")
Barbie
```

```
[1] "I'm a Barbie girl in the Barbie world"
[2] "Life in plastic, it's fantastic"
[3] "You can brush my hair, undress me everywhere"
[4] "Imagination, life is your creation"
```

```
Barbie_df <- tibble(line = 1:4, text = Barbie)
Barbie_df
```

```
# A tibble: 4 x 2
   line text
  <int> <chr>
```

```
1      1 I'm a Barbie girl in the Barbie world
2      2 Life in plastic, it's fantastic
3      3 You can brush my hair, undress me everywhere
4      4 Imagination, life is your creation
```

- We then break the text into individual tokens (tokenization) using tidytext's `unnest_tokens()` function.
  - The two basic arguments for the `unnest_tokens()` function are column names. We have the output column, `word`, created by unnesting the text, and we have the input column, `text`, where the text being unnested comes from.

```
install.packages("tidytext")
```

```
library(tidytext)

Barbie_df %>%
  unnest_tokens(word, text)
```

```
# A tibble: 26 x 2
    line word
   <int> <chr>
 1     1 i'm
 2     1 a
 3     1 barbie
 4     1 girl
 5     1 in
 6     1 the
 7     1 barbie
 8     1 world
 9     2 life
10     2 in
# i 16 more rows
```

9.4.1 Counts

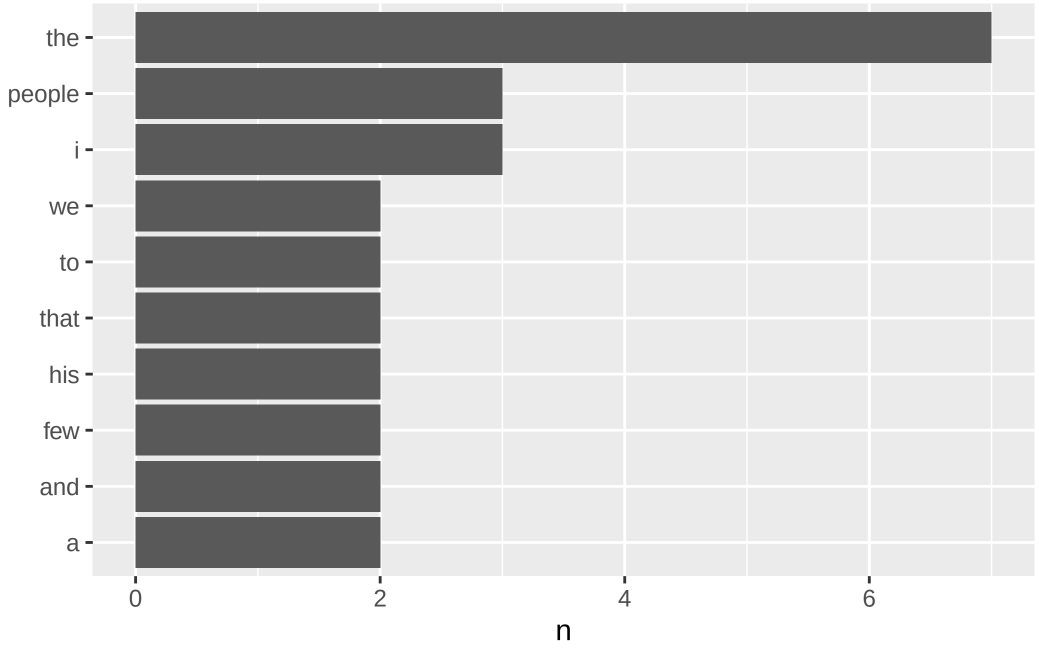
- Once we have our tidy structure, we can then perform very simple tasks such as finding the most common words in our text as a whole. Let's instead work with a short passage from a famous interview with J. Robert Oppenheimer.
  - We can use the `count()` function from the `dplyr` package with ease here.

```
Oppenheimer <- c("We knew the world would not be the same.",
                 "A few people laughed, a few people cried, most people were silent.",
                 "I remembered the line from the Hindu scripture, the Bhagavad-Gita.",
                 "Vishnu is trying to persuade the Prince that he should do his duty and t
                 takes on his multi-armed form and says, "Now, I am become Death, the dest
                 worlds."",
                 "I suppose we all thought that one way or another.")

Opp_df <- tibble(line = 1:5, text = Oppenheimer)

Opp_tok <- unnest_tokens(Opp_df, word, text)

Opp_tok %>%
  count(word, sort = TRUE)
```

```
# A tibble: 59 x 2
   word       n
   <chr>  <int>
 1 the        7
 2 i          3
 3 people     3
 4 a          2
 5 and        2
 6 few        2
 7 his        2
 8 that       2
 9 to         2
10 we         2
# i 49 more rows
```

- Our word counts are stored in a tidy data frame, which allows us to pipe these data directly to the ggplot2 package and create a simple visualization of the most common words in the short excerpt.

```
library(ggplot2)
Opp_tok %>%
  count(word, sort = TRUE) %>%
  filter(n > 1) %>%
  mutate(word = reorder(word, n)) %>%
  ggplot(aes(n, word)) +
  geom_col() +
  labs(y = NULL)
```



9.4.2 tf-idf

- Another way to quantify what a document is about is to calculate a term's *inverse document frequency* (idf), which decreases the weight for commonly used words and increases the weight for words that are not used as frequently in a corpus.
- If we multiply together the term frequency (tf) with the idf, we can calculate the tf-idf, the frequency of a term adjusted for how infrequently it is used.
  - The tf-idf statistic measures how important a word is to document that is part of a corpus.
- We are going to take a look at the published novels of Jane Austen, an example from Silge and Robinson (2017).
  - Let's start by calculating the term frequency.

```
library(janeaustenr)

book_words <- austen_books() %>%
```

```r
  unnest_tokens(word, text) %>%
  count(book, word, sort = TRUE)

total_words <- book_words %>%
  group_by(book) %>%
  summarize(total = sum(n))

book_words <- left_join(book_words, total_words)

book_words
```

```
# A tibble: 40,379 x 4
   book              word      n  total
   <fct>             <chr> <int>  <int>
 1 Mansfield Park    the    6206 160460
 2 Mansfield Park    to     5475 160460
 3 Mansfield Park    and    5438 160460
 4 Emma              to     5239 160996
 5 Emma              the    5201 160996
 6 Emma              and    4896 160996
 7 Mansfield Park    of     4778 160460
 8 Pride & Prejudice the    4331 122204
 9 Emma              of     4291 160996
10 Pride & Prejudice to     4162 122204
# i 40,369 more rows
```

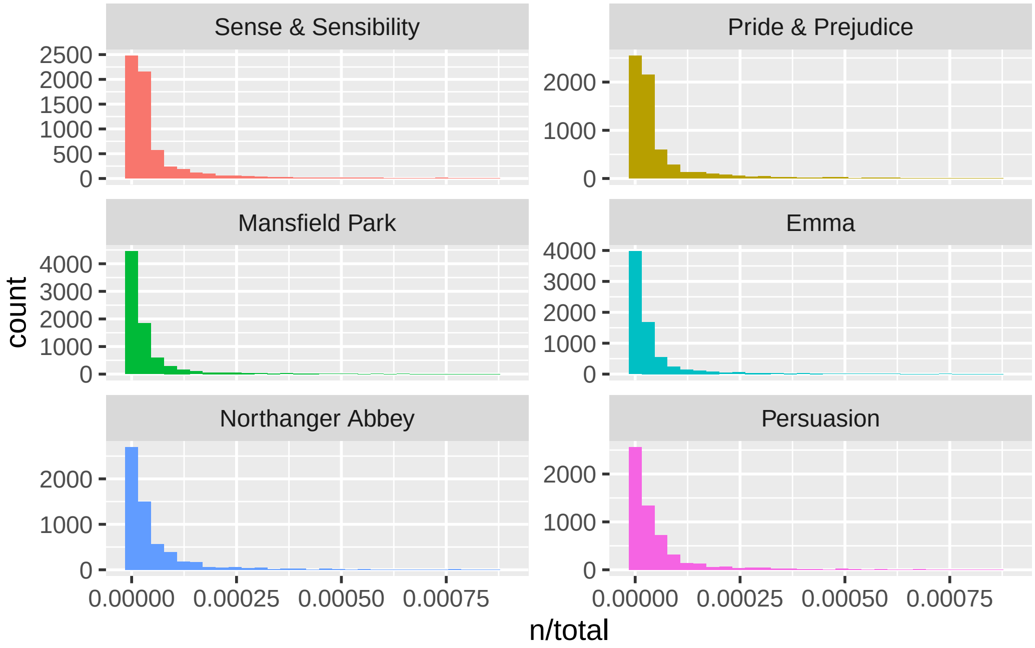
- We can then take these data and visualize them for each of the books in the dataset.

```r
ggplot(book_words, aes(n/total, fill = book)) +
  geom_histogram(show.legend = FALSE) +
  xlim(NA, 0.0009) +
  facet_wrap(~book, ncol = 2, scales = "free_y")
```

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

```
Warning: Removed 896 rows containing non-finite values (`stat_bin()`).
```

```
Warning: Removed 6 rows containing missing values (`geom_bar()`).
```



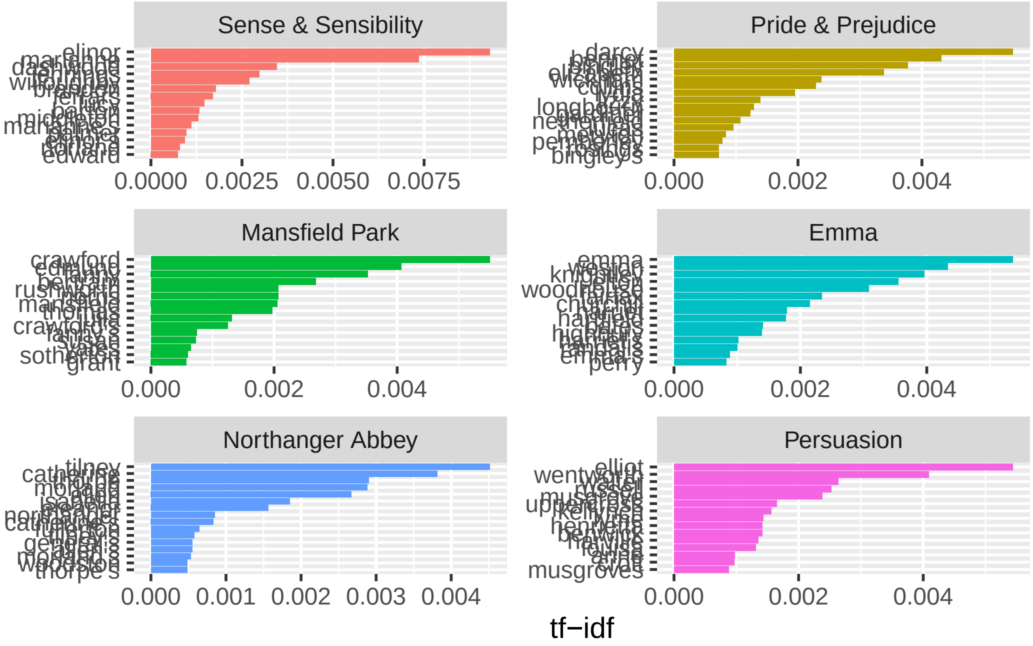
- The `bind_tf_idf()` function in the `tidytext` package then takes a dataset as input with one row per token (term) per document, calculating the tf-idf statistics. Let's look at terms with high scores.
  - Below we see all proper nouns, mostly names of characters. None of them occur across all of Jane Austen's novels, which is why they are important, defining terms for each of the texts.

```
book_tf_idf <- book_words %>%
  bind_tf_idf(word, book, n)

book_tf_idf %>%
  select(-total) %>%
  arrange(desc(tf_idf))
```

```
# A tibble: 40,379 x 6
   book                word          n      tf   idf  tf_idf
   <fct>               <chr>     <int>   <dbl> <dbl>   <dbl>
 1 Sense & Sensibility elinor      623 0.00519  1.79 0.00931
 2 Sense & Sensibility marianne    492 0.00410  1.79 0.00735
 3 Mansfield Park      crawford    493 0.00307  1.79 0.00551
 4 Pride & Prejudice   darcy       373 0.00305  1.79 0.00547
 5 Persuasion          elliot      254 0.00304  1.79 0.00544
 6 Emma                  emma        786 0.00488  1.10 0.00536
 7 Northanger Abbey      tilney      196 0.00252  1.79 0.00452
 8 Emma                  weston      389 0.00242  1.79 0.00433
 9 Pride & Prejudice     bennet      294 0.00241  1.79 0.00431
10 Persuasion            wentworth   191 0.00228  1.79 0.00409
# i 40,369 more rows
```

- Let's end with a visualization for the high tf-idf words in each of Jane Austen's novels.
  - These results highlight that what distinguishes one novel from another within the collection of her works (the corpus) are the proper nouns, mainly the names of people and places. These are the terms that are "important" for defining the character of each document.

```
library(forcats)

book_tf_idf %>%
  group_by(book) %>%
  slice_max(tf_idf, n = 15) %>%
  ungroup() %>%
  ggplot(aes(tf_idf, fct_reorder(word, tf_idf), fill = book)) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~book, ncol = 2, scales = "free") +
  labs(x = "tf-idf", y = NULL)
```

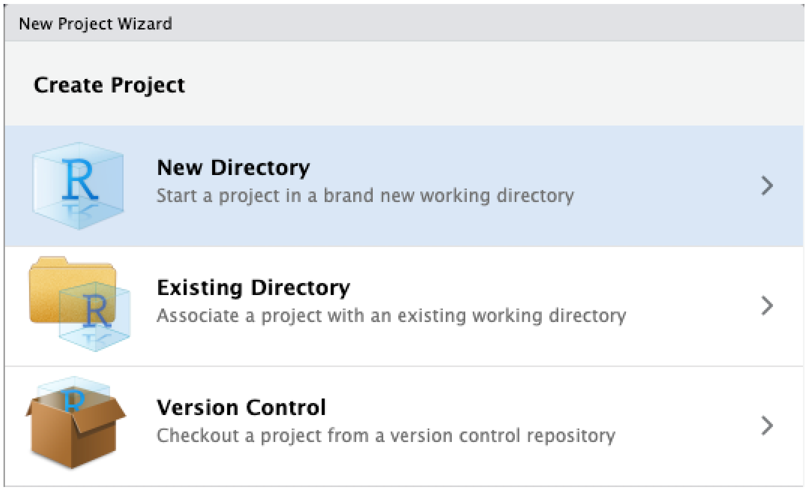

Sense & Sensibility
elinor
marianne
dashwood
jennings
willoughby
brandon
ferrars
lucy
barton
middleton
marianne's
palmer
elinor's
norland
edward
0.0000
0.0025
0.0050
0.0075
Pride & Prejudice
darcy
bennet
bingley
elizabeth
wickham
collins
lydia
longbourn
gardiner
netherfield
meryton
pemberley
rosings
bingleys
0.000
0.002
0.004
Mansfield Park
crawford
edmund
fanny
bertram
rushworth
norris
mansfield
thomas
julia
crawford's
fanny's
susan
yates
sotherton
grant
0.000
0.002
0.004
Emma
emma
weston
knightley
elton
woodhouse
fairfax
churchill
harriet
hartfield
bates
highbury
harriet's
randalls
emma's
perry
0.000
0.002
0.004
Northanger Abbey
tilney
catherine
thorpe
morland
allen
isabella
eleanor
northanger
catherine's
fullerton
tilney's
general's
allen's
morland's
woodston
thorpe's
0.000
0.001
0.002
0.003
0.004
Persuasion
elliot
wentworth
walter
russell
musgrove
uppercross
kellynch
lyme
henrietta
benwick
harville
louisa
anne
elliot's
musgroves
0.000
0.002
0.004
tf−idf

10 Wrap up

10.1 Project management

10.1.1 RStudio projects

- RStudio projects are an excellent way to keep all the files associated with a project (data, R scripts, results, figures, etc.) in one place on your computer.
- This is one of the best ways to improve your workflow in RStudio, allowing you to:
  - Create a project for each paper or data analysis project.
  - Store data files in one place.
  - Save, edit, and run scripts.
  - Keep outputs such as plots and cleaned data.
- To create a new project file, click `File > New Project`, then:


New Project Wizard

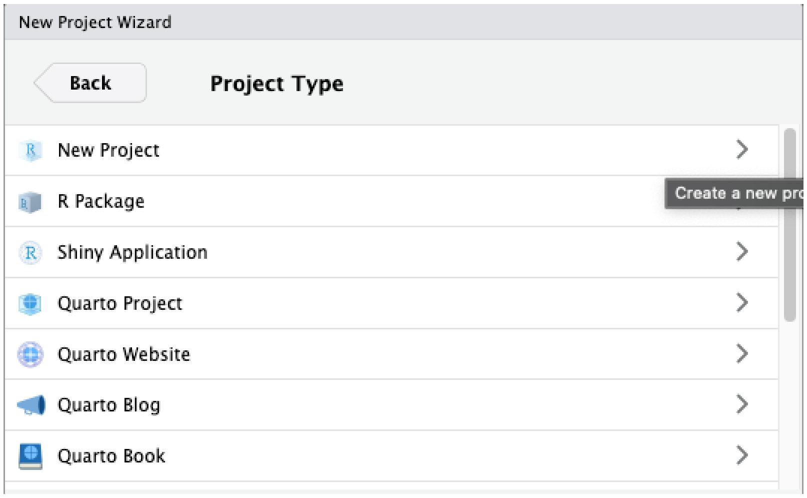
Create Project

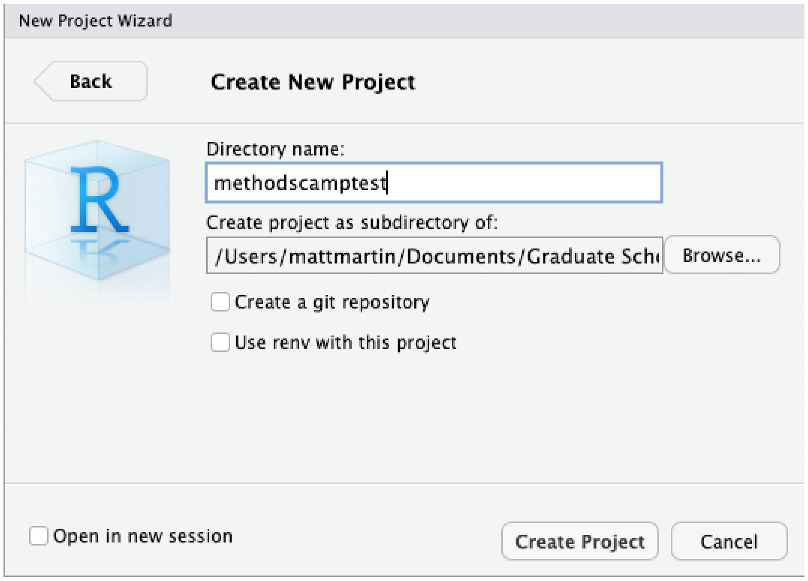
New Directory
Start a project in a brand new working directory

Existing Directory
Associate a project with an existing working directory

Version Control
Checkout a project from a version control repository

- Call your project some version of "methodscamptest" and choose carefully where you wish to store the project on your machine.





Directory name:
methodscamptest

Create project as subdirectory of:
/Users/mattmartin/Documents/Graduate Sch

□ Create a git repository

□ Use renv with this project

□ Open in new session

⚠ Warning

If you don’t store your project (and your other files, too!) somewhere reasonable, it will be hard to find it in the future! We recommend creating a clear organizational scheme for yourself early on.

10.1.1.1 Using RStudio projects

When using an RStudio project, you should see its name in the top-right corner of RStudio, next to a light blue icon. You can check with R the folder in which your project operates:

```r
getwd()
```

- Now, as an example, let’s run the following commands in the script editor and save the files into the project directory.

```r
library(tidyverse)

my_plot <- ggplot(mtcars, aes(wt, mpg)) +
  geom_point()

ggsave(plot = my_plot,
       filename = "plot_mtcars.pdf")

write_csv(mtcars, "mtcars.csv")
```

- Quit RStudio and check out the folder associated with the project.
- You should see the PDF file for the plot, the .csv file for the data, and the `.Rproj` file for the project itself.
- Double-click the `.Rproj` file to reopen the project and pick up where you left off! Everything you need should be ready to go.

10.2 Other resources

10.2.1 Overleaf

- Overleaf is a collaborative cloud-based LaTeX editor designed for writing, editing, and publishing documents.
  - LaTeX is a software used for typesetting technical documents. It is used widely in our discipline for the preparation for manuscripts to journals and other publishing venues.
- UT Austin actually provides free access to Overleaf Professional to all graduate students using your UT email.
- Overleaf Professional upgrades include:
  - Real-time collaboration
  - Real-time track changes and visible collaborator cursor(s)
  - Real-time PDF preview of your document while editing and writing
  - Full history view of your documents
  - Two-way sync with Dropbox and GitHub
  - Reference manager sync and advanced reference search.
  - UT Austin resource portal, including UT Austin templates, FAQs, and resource links

> ! Important
>
> LaTeX is actually the markup language that powers this website! If you are curious about general syntax and commands, you can access our repository at any time to get a closer look.

10.2.2 Zotero

- Zotero is an open-source reference manager used to store, manage, and cite bibliographic references, such as books and articles.
- When it is time to write, you can insert your sources directly into your paper as in-text citations via a word processor plugin, which generates a bibliography in your style of choice.
    - This can save a lot of time, especially when you have to change citation styles for submission to another journal.
- You can download the software for free here.
    - You can also find a guide on how to install it here.

> ℹ Note
>
> Zotero is one of many other reference managers out there. Alternatives include Mendeley and EndNote, among others. You should choose whatever option best suits your needs.

10.2.2.1 Benefits of Zotero

- If you have not yet chosen a reference manager or are considering switching, below are some advantages of Zotero:
  - Works as a standalone desktop software with plugins for Chrome, Safari, and Firefox
  - Full compatibility with Google docs
  - Free plugin for Word and LibreOffice included
  - Includes most popular citation styles with more styles available on the Zotero Style Repository
  - Drag and drop PDF files into the library, extracting metadata such as authors, year, etc.
  - Allows advanced searches of all content in your library using full-text PDF indexing
  - Use cloud storage (optional) and sync libraries across devices
  - Create unlimited private or public groups and collaborate by sharing files and citations
  - 300MB of free cloud storage and 2GB of storage for $20 USD/year (equal to $1.67 per month)
- Here is a comprehensive guide to unlocking all of Zotero's potential.

10.3 Methods at UT

10.3.1 Required methods courses

- Scope and Methods of Political Science
  - Statistics I (Statistics/linear regression)
- Statistics II (Linear regression and more)
- Statistics III (Maximum likelihood estimation)
  - Only required if your major field is methods

10.3.2 Other methods courses

- **Statistics/econometrics:**
  - Bayesian Statistics
  - Causal Inference
  - Math Methods for Political Analysis
  - Time Series and Panel Data

  - Panel and Multilevel Analysis

10.3.3 More courses

- **Formal Theory**
  - Intro to Formal Political Analysis
  - Formal Political Analysis II
  - Formal Theories of International Relations
- **Everything else**
  - Conceptualization and Measurement
  - Experimental Methods in Political Science
  - Qualitative Methods
  - Network Analysis
  - Seminar in Field Experiments

10.3.4 Other departments at UT

You can also take courses through the Economics, Mathematics, or Statistics (Statistics and Data Science) departments.

- M.S. in Statistics

Software and Topic Short Courses - R, Python, Stata, etc.

- More info here.

10.3.5 Other resources

Summer programs at UT:

- Short courses in statistics
  - Department sometimes offers scholarships to cover part of the cost.

Summer programs outside UT:

- ICPSR (Inter-university Consortium for Political and Social Research)
  - Ann Arbor, Michigan
- EITM (Empirical Implications of Theoretical Models)

  - Houston and other locations (Michigan, Duke, Berkeley, Emory)
- IQMR (Institute for Qualitative and Multi-Method Research)
  - Syracuse, NY

References

Arel-Bundock, Vincent, Nils Enevoldsen, and CJ Yetman. 2018. "Countrycode: An r Package to Convert Country Names and Country Codes." *Journal of Open Source Software* 3 (28): 848. https://doi.org/10.21105/joss.00848.

Bank, World. 2023. "World Bank Open Data." https://data.worldbank.org/.

Dahlberg, Stefan, Aksen Sundström, Sören Holmberg, Bo Rothstein, Natalia Alvarado Pachon, Cem Mert Dalli, and Yente Meijers. 2023. "The Quality of Government Basic Dataset, Version Jan23." University of Gothenburg: The Quality of Government Institute. https://www.gu.se/en/quality-government doi:10.18157/qogbasjan23.

FiveThirtyEight. 2021. "Tracking Congress In The Age Of Trump [Dataset]." https://projects.fivethirtyeight.com/congress-trump-score/.

Robinson, David. 2020. *Fuzzyjoin: Join Tables Together on Inexact Matching*. https://github.com/dgrtwo/fuzzyjoin.

Smith, Danny. 2020. *Survey Research Datasets and R*. https://socialresearchcentre.github.io/r_survey_datasets/.

U. S. Department of Agriculture [USDA], Agricultural Research Service. 2019. "Department of Agriculture Agricultural Research Service." https://fdc.nal.usda.gov/.