Tiffany Timbers, Trevor Campbell and Melissa Lee

Data Science: A First Introduction

Dedication to be completed ...

Contents

List of Tables

List of Figures

Preface

This is an open source textbook aimed at introducing undergraduate students to data science. It was originally written for the University of British Columbia's DSCI 100 - Introduction to Data Science[1] course. In this book, we define data science as the study and development of reproducible, auditable processes to obtain value (i.e., insight) from data.

Why read this book

This book will provide readers unfamiliar with data science an introduction to the field, advice on best practices for performing data analysis, as well as practical skills to get up and going doing their own data analyses.

Structure of the book

The book is structured so that learners spend the first four chapters learning how to use the R programming language and Jupyter notebooks to load, wrangle/clean, and visualize data, while answering descriptive and exploratory data analysis questions. The remaining chapters illustrate how to solve four common problems in data science, which are useful for answering predictive and inferential data analysis questions.

Software information and conventions

I used the **knitr** package (Xie, 2015) and the **bookdown** package (Xie, 2020) to compile my book. My R session information is shown below:

[1] https://ubc-dsci.github.io/dsci-100/

```
xfun::session_info()
```

```
## R version 4.0.3 (2020-10-10)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Catalina 10.15.7, RStudio 1.4.953
##
## Locale: en_CA.UTF-8 / en_CA.UTF-8 / en_CA.UTF-8 / C / en_CA.UTF-
8 / en_CA.UTF-8
##
## Package version:
##   askpass_1.1          assertthat_0.2.1
##   backports_1.1.10     base64enc_0.1.3
##   BH_1.72.0.3          bit_4.0.4
##   bit64_4.0.5          blob_1.2.1
##   bookdown_0.21        brew_1.0.6
##   broom_0.7.2          callr_3.5.1
##   cancensus_0.3.2      canlang_0.0.1
##   caret_6.0-86         cellranger_1.1.0
##   class_7.3-17         cli_2.2.0
##   clipr_0.7.1          codetools_0.2-16
##   colorspace_1.4-1     commonmark_1.7
##   compiler_4.0.3       covr_3.5.1
##   cpp11_0.2.3          crayon_1.3.4
##   crosstalk_1.1.0.1    curl_4.3
##   data.table_1.13.2    DBI_1.1.0
##   dbplyr_1.4.4         desc_1.2.0
##   devtools_2.3.2       dials_0.0.9
##   DiceDesign_1.8-1     digest_0.6.26
##   dplyr_1.0.2          DT_0.16
##   ellipsis_0.3.1       evaluate_0.14
##   fansi_0.4.1          farver_2.0.3
##   forcats_0.5.0        foreach_1.5.1
##   fs_1.5.0             furrr_0.2.1
##   future_1.21.0        gdtools_0.2.2
##   generics_0.1.0       GGally_2.0.0
##   ggplot2_3.3.2        gh_1.1.0
##   git2r_0.27.1         globals_0.14.0
##   glue_1.4.2           gower_0.2.2
##   GPfit_1.0-8          graphics_4.0.3
##   grDevices_4.0.3      grid_4.0.3
##   gridExtra_2.3        gtable_0.3.0
##   hardhat_0.1.5        haven_2.3.1
##   highr_0.8            hms_0.5.3
##   htmltools_0.5.0      htmlwidgets_1.5.2
```

```
##   httr_1.4.2           igraph_1.2.6
##   infer_0.5.3          ini_0.3.1
##   ipred_0.9-9          isoband_0.2.2
##   iterators_1.0.13     jsonlite_1.7.1
##   KernSmooth_2.23.17   kknn_1.3.1
##   knitr_1.30           labeling_0.4.2
##   later_1.1.0.1        lattice_0.20-41
##   lava_1.6.8.1         lazyeval_0.2.2
##   lhs_1.1.1            lifecycle_0.2.0
##   listenv_0.8.0        lubridate_1.7.9
##   magick_2.5.2         magrittr_2.0.1
##   markdown_1.1         MASS_7.3-53
##   Matrix_1.2-18        memoise_1.1.0
##   methods_4.0.3        mgcv_1.8-33
##   mime_0.9             modeldata_0.1.0
##   ModelMetrics_1.2.2.2 modelr_0.1.8
##   munsell_0.5.0        nlme_3.1-149
##   nnet_7.3-14          numDeriv_2016.8.1.1
##   openssl_1.4.3        parallel_4.0.3
##   parallelly_1.22.0    parsnip_0.1.4
##   pillar_1.4.6         pkgbuild_1.1.0
##   pkgconfig_2.0.3      pkgload_1.1.0
##   plogr_0.2.0          plyr_1.8.6
##   png_0.1-7            praise_1.0.0
##   prettyunits_1.1.1    pROC_1.16.2
##   processx_3.4.4       prodlim_2019.11.13
##   progress_1.2.2       promises_1.1.1
##   ps_1.4.0             purrr_0.3.4
##   R6_2.4.1             rcmdcheck_1.3.3
##   RColorBrewer_1.1-2   Rcpp_1.0.5
##   readr_1.4.0          readxl_1.3.1
##   recipes_0.1.15       rematch_1.0.1
##   rematch2_2.1.2       remotes_2.2.0
##   reprex_0.3.0         reshape_0.8.8
##   reshape2_1.4.4       rex_1.2.0
##   rlang_0.4.8          rmarkdown_2.5
##   roxygen2_7.1.1       rpart_4.1-15
##   rprojroot_2.0.2      rsample_0.0.8
##   RSQLite_2.2.1        rstudioapi_0.13
##   rversions_2.0.2      rvest_0.3.6
##   scales_1.1.1         selectr_0.4-2
##   sessioninfo_1.1.1    slider_0.1.5
##   splines_4.0.3        SQUAREM_2020.5
##   stats_4.0.3          stats4_4.0.3
##   stringi_1.5.3        stringr_1.4.0
```

```
##   survival_3.2-7       svglite_1.2.3.2
##   sys_3.4              systemfonts_0.3.2
##   testthat_2.3.2       tibble_3.0.4
##   tidymodels_0.1.2     tidyr_1.1.2
##   tidyselect_1.1.0     tidyverse_1.3.0
##   timeDate_3043.102    tinytex_0.26
##   tools_4.0.3          tune_0.1.2
##   usethis_1.6.3        utf8_1.1.4
##   utils_4.0.3          vctrs_0.3.4
##   viridisLite_0.3.0    warp_0.2.0
##   whisker_0.4          withr_2.3.0
##   workflows_0.2.1      xfun_0.18
##   xml2_1.3.2           xopen_1.0.0
##   yaml_2.2.1           yardstick_0.0.7
```

Package names are in bold text (e.g., **rmarkdown**), and inline code and filenames are formatted in a typewriter font (e.g., `knitr::knit('foo.Rmd')`). Function names are followed by parentheses (e.g., `bookdown::render_book()`).

Acknowledgments

A lot of people helped me when I was writing the book.

The DSCI 100 teaching assistant team and Matías Salibián-Barrera

About the Authors

Tiffany Timbers is an Assistant Professor of Teaching in the Department of Statistics and an Co-Director for the Master of Data Science program (Vancouver Option) at the University of British Columbia. In these roles she teaches and develops curriculum around the responsible application of Data Science to solve real-world problems. One of her favourite courses she teaches is a graduate course on collaborative software development, which focuses on teaching how to create R and Python packages using modern tools and workflows.

Trevor Campbell is an Assistant Professor of Statistics at the University of British Columbia. His research focuses on automated, scalable Bayesian inference algorithms, Bayesian nonparametrics, streaming data, and Bayesian theory. He was previously a postdoctoral associate advised by Tamara Broderick in the Computer Science and Artificial Intelligence Laboratory (CSAIL) and Institute for Data, Systems, and Society (IDSS) at MIT, a Ph.D. candidate under Jonathan How in the Laboratory for Information and Decision Systems (LIDS) at MIT, and before that he was in the Engineering Science program at the University of Toronto.

Melissa Lee is an Assistant Professor of Teaching in the Department of Statistics at the University of British Columbia. She enjoys teaching introductory statistics and data science courses. She is dedicated to improving statistics and data science education for instructors and learners alike through developing content grounded in evidence-based teaching practices.

1

Introduction

This is an open source textbook aimed at introducing the field of data science. In this book, we define data science as the study and development of reproducible, auditable processes to obtain value (i.e., insight) from data.

The book is structured so that learners spend the first four chapters learning how to use the R programming language and Jupyter notebooks to load, wrangle/clean, and visualize data, while answering descriptive and exploratory data analysis questions. The remaining chapters illustrate how to solve four common problems in data science, which are useful for answering predictive and inferential data analysis questions:

1. Predicting a class/category for a new observation/measurement (e.g., cancerous or benign tumour)
2. Predicting a value for a new observation/measurement (e.g., 10 km race time for 20 year old females with a BMI of 25).
3. Finding previously unknown/unlabelled subgroups in your data (e.g., products commonly bought together on Amazon)
4. Estimating an average or a proportion from a representative sample (group of people or units) and using that estimate to generalize to the broader population (e.g., the proportion of undergraduate students that own an iphone)

For each of these problems, we map them to the type of data analysis question being asked and discuss what kinds of data are needed to answer such questions (Leek and Peng, 2015; Peng and Matsui, 2015). More advanced (e.g., causal or mechanistic) data analysis questions are beyond the scope of this text.

Types of data analysis questions

Question type	Description	Example
Descriptive	A question which asks about summarized characteristics of a data set without interpretation (i.e., report a fact).	How many people live in each province or territory in Canada?
Exploratory	A question asks if there are patterns, trends, or relationships within a single data set. Often used to propose hypotheses for future study.	Does political party voting change with indicators of wealth in a set of data collected on 2,000 people living in Canada?
Inferential	A question that looks for patterns, trends, or relationships in a single data set **and** also asks for quantification of how applicable these findings are to the wider population.	Does political party voting change with indicators of wealth for all people living in Canada?
Predictive	A question that asks about predicting measurements or labels for individuals (people or things). The focus is on what things predict some outcome, but not what causes the outcome.	What political party will someone vote for in the next Canadian election?
Causal	A question that asks about whether changing one factor will lead to a change in another factor, on average, in the wider population.	Does wealth lead to voting for a certain political party in Canadian elections?

Question type	Description	Example
Mechanistic	A question that asks about the underlying mechanism of the observed patterns, trends, or relationship (i.e., how does it happen?)	How does wealth lead to voting for a certain political party in Canadian elections?

Source: What is the question?[1] by Jeffery T. Leek, Roger D. Peng & The Art of Data Science[2] by Roger Peng & Elizabeth Matsui

1.1 Chapter learning objectives

By the end of the chapter, students will be able to:

- use a Jupyter notebook to execute provided R code
- edit code and markdown cells in a Jupyter notebook
- create new code and markdown cells in a Jupyter notebook
- load the `tidyverse` package into R
- create new variables and objects in R using the assignment symbol
- use the help and documentation tools in R
- match the names of the following functions from the `tidyverse` package to their documentation descriptions:
  - `read_csv`
  - `select`
  - `filter`
  - `mutate`
  - `ggplot`
  - `aes`

1.2 Jupyter notebooks

Jupyter notebooks are documents that contain a mix of computer code (and its output) and formattable text. Given that they are able to combine these two in a single document—code is not separate from the output or written

[1] https://science.sciencemag.org/content/347/6228/1314
[2] https://leanpub.com/artofdatascience

report—notebooks are one of the leading tools to create *reproducible data analyses*. A reproducible data analysis is one where you can reliably and easily recreate the same results when analyzing the same data. Although this sounds like something that should always be true of any data analysis, in reality this is not often the case; one needs to make a conscious effort to perform data analysis in a reproducible manner.

The name Jupyter came from combining the names of the three programming language that it was initially targeted for (Julia, Python, and R), and now many other languages can be used with Jupyter notebooks.

A notebook looks like this:

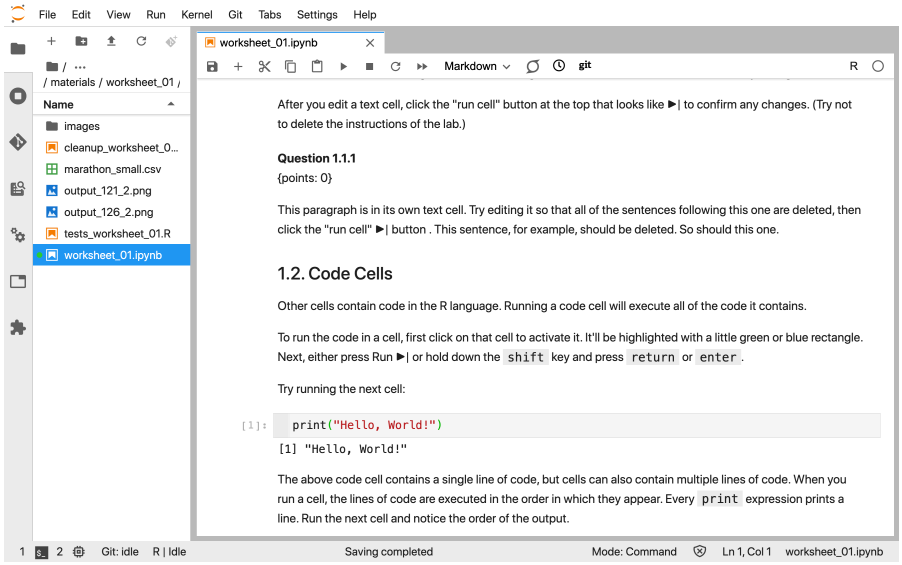


FIGURE 1.1: Jupyter Notebook

We have included a short demo video here to help you get started and to introduce you to R and Jupyter. However, the best way to learn how to write and run code and formattable text in a Jupyter notebook is to do it yourself! Here is a worksheet[3] that provides a step-by-step guide through the basics.

[3] https://github.com/UBC-DSCI/dsci-100-assets/blob/master/2019-fall/materials/worksheet_01/worksheet_01.ipynb

1.3 Loading a spreadsheet-like dataset

Often, the first thing we need to do in data analysis is to load a dataset into R. When we bring spreadsheet-like (think Microsoft Excel tables) data, generally shaped like a rectangle, into R it is represented as what we call a *data frame* object. It is very similar to a spreadsheet where the rows are the collected observations and the columns are the variables.

Spreadsheet

	A	B	C	D	E	F
1	category	language	mother_tongue	most_at_home	most_at_work	lang_known
2	Aboriginal languages	Aboriginal languages, n.o.s.	590	235	30	665
3	Non-Official & Non-Aboriginal languages	Afrikaans	10260	4785	85	23415
4	Non-Official & Non-Aboriginal languages	Afro-Asiatic languages, n.i.e.	1150	445	10	2775
5	Non-Official & Non-Aboriginal languages	Akan (Twi)	13460	5985	25	22150
6	Non-Official & Non-Aboriginal languages	Albanian	26895	13135	345	31930
7	Aboriginal languages	Algonquian languages, n.i.e.	45	10	0	120
8	Aboriginal languages	Algonquin	1260	370	40	2480
9	Non-Official & Non-Aboriginal languages	American Sign Language	2685	3020	1145	21930
10	Non-Official & Non-Aboriginal languages	Amharic	22465	12785	200	33670
11	Non-Official & Non-Aboriginal languages	Arabic	419890	223535	5585	629055

Data frame in R

```
# A tibble: 214 x 6
   category                          language                      mother_tongue most_at_home most_at_work lang_known
   <chr>                             <chr>                                 <dbl>        <dbl>        <dbl>      <dbl>
 1 Aboriginal languages              Aboriginal languages, n.o.s.            590          235           30        665
 2 Non-Official & Non-Aboriginal languages Afrikaans                       10260         4785           85      23415
 3 Non-Official & Non-Aboriginal languages Afro-Asiatic languages, n.i.e.   1150          445           10       2775
 4 Non-Official & Non-Aboriginal languages Akan (Twi)                      13460         5985           25      22150
 5 Non-Official & Non-Aboriginal languages Albanian                        26895        13135          345      31930
 6 Aboriginal languages              Algonquian languages, n.i.e.            45           10            0        120
 7 Aboriginal languages              Algonquin                             1260          370           40       2480
 8 Non-Official & Non-Aboriginal languages American Sign Language          2685         3020         1145      21930
 9 Non-Official & Non-Aboriginal languages Amharic                        22465        12785          200      33670
10 Non-Official & Non-Aboriginal languages Arabic                        419890       223535         5585     629055
# … with 204 more rows
```

FIGURE 1.2: A spreadsheet versus a data frame in R

The first kind of data we will learn how to load into R (as a data frame) is the spreadsheet-like *comma-separated values* format (`.csv` for short). These files have names ending in `.csv`, and can be opened open and saved from common spreadsheet programs like Microsoft Excel and Google Sheets. For example, a `.csv` file named `can_lang.csv` is included with the code for this book[4]. This file— originally from {canlang} R data package[5]—has language data collected in the 2016 Canadian census (Canada, 2016). If we were to open this data in a plain text editor, we would see each row on its own line, and each entry in the table separated by a comma:

```
category,language,mother_tongue,most_at_home,most_at_work,lang_known
```

[4] https://github.com/UBC-DSCI/introduction-to-datascience/blob/master/data/can_lang.csv

[5] https://ttimbers.github.io/canlang/

```
Aboriginal languages,"Aboriginal languages, n.o.s.",590,235,30,665
Non-Official & Non-Aboriginal languages,Afrikaans,10260,4785,85,23415
Non-Official & Non-Aboriginal languages,"Afro-
Asiatic languages, n.i.e.",1150,445,10,2775
Non-Official & Non-Aboriginal languages,Akan (Twi),13460,5985,25,22150
Non-Official & Non-Aboriginal languages,Albanian,26895,13135,345,31930
Aboriginal languages,"Algonquian languages, n.i.e.",45,10,0,120
Aboriginal languages,Algonquin,1260,370,40,2480
Non-Official & Non-Aboriginal languages,American Sign Language,2685,3020,1145,21930
Non-Official & Non-Aboriginal languages,Amharic,22465,12785,200,33670
```

To load this data into R, and then to do anything else with it afterwards, we will need to use something called a *function.* A function is a special word in R that takes in instructions (we call these *arguments*) and does something. The function we will use to read a `.csv` file into R is called `read_csv`.

In its most basic use-case, `read_csv` expects that the data file:

- has column names (or *headers*),
- uses a comma (`,`) to separate the columns, and
- does not have row names.

Below you'll see the code used to load the data into R using the `read_csv` function. But there is one extra step we need to do first. Since `read_csv` is not included in the base installation of R, to be able to use it we have to load it from somewhere else: a collection of useful functions known as a *package*. The `read_csv` function in particular is in the `tidyverse` package (more on this later), which we load using the `library` function.

Next, we call the `read_csv` function and pass it a single argument: the name of the file, `"can_lang.csv"`. We have to put quotes around filenames and other letters and words that we use in our code to distinguish it from the special words that make up R programming language. This is the only argument we need to provide for this file, because our file satifies everthing else the `read_csv` function expects in the default use-case (which we just discussed). Later in the course, we'll learn more about how to deal with more complicated files where the default arguments are not appropriate. For example, files that use spaces or tabs to separate the columns, or with no column names.

```
library(tidyverse)
read_csv("data/can_lang.csv")
```

```
## # A tibble: 214 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~           590          235
##  2 Non-Off~ Afrikaa~         10260         4785
```

```
##  3 Non-Off~ Afro-As~          1150          445
##  4 Non-Off~ Akan (T~         13460         5985
##  5 Non-Off~ Albanian         26895        13135
##  6 Aborigi~ Algonqu~            45           10
##  7 Aborigi~ Algonqu~          1260          370
##  8 Non-Off~ America~          2685         3020
##  9 Non-Off~ Amharic          22465        12785
## 10 Non-Off~ Arabic          419890       223535
## # ... with 204 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

Above you can also see something neat that Jupyter does to help us understand our code: it colours text depending on its meaning in R. For example, you'll note that functions get bold green text, while letters and words surrounded by quotations like filenames get blue text.

In case you want to know more (optional): We use the `read_csv` function from the `tidyverse` instead of the base R function `read.csv` because it's faster and it creates a nicer variant of the base R data frame called a *tibble*. This has several benefits that we'll discuss in further detail later in the course.

1.4 Assigning value to a data frame

When we loaded the language data collected in the 2016 Canadian census in R above using `read_csv`, we did not give this data frame a name, so it was just printed to the screen and we cannot do anything else with it. That isn't very useful; what we would like to do is give a name to the data frame that `read_csv` outputs so that we can use it later for analysis and visualization.

To assign name to something in R, there are two possible ways—using either the assignment symbol (`<-`) or the equals symbol (`=`). From a style perspective, the assignment symbol is preferred and is what we will use in this course. When we name something in R using the assignment symbol, `<-`, we do not need to surround it with quotes like the filename. This is because we are formally telling R about this word and giving it a value. Only characters and words that act as values need to be surrounded by quotes.

Let's now use the assignment symbol to give the name `can_lang` to the lan-

guage data collected in the 2016 Canadian census data frame that we get from `read_csv`.

```
can_lang <- read_csv("data/can_lang.csv")
```

Wait a minute! Nothing happened this time! Or at least it looks like that. But actually, something did happen: the data was read in and now has the name `can_lang` associated with it. And we can use that name to access the data frame and do things with it. First we will type the name of the data frame to print it to the screen.

```
can_lang
```

```
## # A tibble: 214 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~           590          235
##  2 Non-Off~ Afrikaa~         10260         4785
##  3 Non-Off~ Afro-As~          1150          445
##  4 Non-Off~ Akan (T~         13460         5985
##  5 Non-Off~ Albanian         26895        13135
##  6 Aborigi~ Algonqu~            45           10
##  7 Aborigi~ Algonqu~          1260          370
##  8 Non-Off~ America~          2685         3020
##  9 Non-Off~ Amharic          22465        12785
## 10 Non-Off~ Arabic          419890       223535
## # ... with 204 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

1.5 Creating subsets of data frames with `select` & `filter`

Now, we are going to learn how to obtain subsets of data from a data frame in R using two other `tidyverse` functions: `select` and `filter`. The `select` function allows you to create a subset of the columns of a data frame, while the `filter` function allows you to obtain a subset of the rows with specific values.

Before we start using `select` and `filter`, let's take a look at the language data collected in the 2016 Canadian census again to familiarize ourselves with it. We will do this by printing the data we loaded earlier in the chapter to the screen.

```
can_lang
```

```
## # A tibble: 214 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~           590          235
##  2 Non-Off~ Afrikaa~         10260         4785
##  3 Non-Off~ Afro-As~          1150          445
##  4 Non-Off~ Akan (T~         13460         5985
##  5 Non-Off~ Albanian         26895        13135
##  6 Aborigi~ Algonqu~            45           10
##  7 Aborigi~ Algonqu~          1260          370
##  8 Non-Off~ America~          2685         3020
##  9 Non-Off~ Amharic          22465        12785
## 10 Non-Off~ Arabic          419890       223535
## # ... with 204 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

In this data frame there are 214 rows rows (corresponding to the 214 languages recorded on the 2016 Canadian census) and 6 columns:

1. `category`: Higher level language category (describing whether the language is an Official Canadian language, an Aboriginal language, or a Non-Official and Non-Aboriginal language).
2. `language`: Language queried about on the Canadian Census.
3. `mother_tongue`: Total count of Canadians from the Census who reported the language as their mother tongue. Mother tongue is generally defined as the language someone was exposed to since birth.
4. `most_at_home`: Total count of Canadians from the Census who reported the language as spoken most often at home.
5. `most_at_work`: Total count of Canadians from the Census who reported the language as used most often at work for the population.
6. `lang_known`: Total count of Canadians from the Census who reported knowledge of language for the population in private households.

Now let's use `select` to extract the language column from this data frame. To do this, we need to provide the `select` function with two arguments. The first argument is the name of the data frame object, which in this example is `can_lang`. The second argument is the column name that we want to select, here `language`. After passing these two arguments, the `select` function returns a single column (the `language` column that we asked for) as a data frame.

```
language_column <- select(can_lang, language)
language_column
```

```
## # A tibble: 214 x 1
##    language
##    <chr>
##  1 Aboriginal languages, n.o.s.
##  2 Afrikaans
##  3 Afro-Asiatic languages, n.i.e.
##  4 Akan (Twi)
##  5 Albanian
##  6 Algonquian languages, n.i.e.
##  7 Algonquin
##  8 American Sign Language
##  9 Amharic
## 10 Arabic
## # ... with 204 more rows
```

1.5.1 Using `select` to extract multiple columns

We can also use `select` to obtain a subset of the data frame with multiple columns. Again, the first argument is the name of the data frame. Then we list all the columns we want as arguments separated by commas. Here we create a subset of three columns: language, mother tongue, and language spoken most often at home.

```
three_columns <- select(can_lang, language, mother_tongue, most_at_home)
three_columns
```

```
## # A tibble: 214 x 3
##    language                  mother_tongue most_at_home
##    <chr>                             <dbl>        <dbl>
##  1 Aboriginal languages, n.~           590          235
##  2 Afrikaans                         10260         4785
##  3 Afro-Asiatic languages, ~          1150          445
##  4 Akan (Twi)                        13460         5985
##  5 Albanian                          26895        13135
##  6 Algonquian languages, n.~            45           10
##  7 Algonquin                          1260          370
##  8 American Sign Language             2685         3020
##  9 Amharic                           22465        12785
## 10 Arabic                           419890       223535
## # ... with 204 more rows
```

1.5.2 Using `select` to extract a range of columns

We can also use `select` to obtain a subset of the data frame constructed from a range of columns. To do this we use the colon (`:`) operator to denote the

range. For example, to get all the columns in the data frame from `language` to `most_at_home` we pass `language:most_at_home` as the second argument to the `select` function.

```
column_range <- select(can_lang, language:most_at_home)
column_range
```

```
## # A tibble: 214 x 3
##    language                  mother_tongue most_at_home
##    <chr>                             <dbl>        <dbl>
##  1 Aboriginal languages, n.~           590          235
##  2 Afrikaans                         10260         4785
##  3 Afro-Asiatic languages, ~          1150          445
##  4 Akan (Twi)                        13460         5985
##  5 Albanian                          26895        13135
##  6 Algonquian languages, n.~            45           10
##  7 Algonquin                          1260          370
##  8 American Sign Language             2685         3020
##  9 Amharic                           22465        12785
## 10 Arabic                           419890       223535
## # ... with 204 more rows
```

1.5.3 Using `filter` to extract a single row

We can use the `filter` function to obtain the subset of rows with desired values from a data frame. Again, our first argument is the name of the data frame object, `can_lang`. The second argument is a logical statement to use when filtering the rows. Here, for example, we'll say that we are interested in rows where the language is Mandarin. To make this comparison, we use the *equivalency operator* `==` to compare the values of the `language` column with the value `"Mandarin"`. Similar to when we loaded the data file and put quotes around the filename, here we need to put quotes around `"Mandarin"` to tell R that this is a character value and not one of the special words that make up R programming language, nor one of the names we have given to data frames in the code we have already written.

With these arguments, `filter` returns a data frame that has all the columns of the input data frame but only the rows we asked for in our logical filter statement.

```
mandarin <- filter(can_lang, language == "Mandarin")
mandarin
```

```
## # A tibble: 1 x 6
##   category language mother_tongue most_at_home
```

```
##   <chr>    <chr>            <dbl>        <dbl>
## 1 Non-Off~ Mandarin        592040       462890
## # ... with 2 more variables: most_at_work <dbl>,
## #   lang_known <dbl>
```

1.5.4 Using `filter` to extract rows with values above a threshold

If we are interested in finding information about the languages who have a higher number of people who primarily speak it at home compared to Mandarin—which is reported to have 462890 people speaking it as the primary language they speak in their home—then we can create a filter to obtain rows where the value of `most_at_home` is greater than 462890. In this case, we see that `filter` returns a data frame with 2 rows; this indicates that there are two languages that are spoken more often at home compared to Mandarin.

```
spoke_often_at_home <- filter(can_lang, most_at_home > 462890)
spoke_often_at_home
```

```
## # A tibble: 2 x 6
##   category language mother_tongue most_at_home
##   <chr>    <chr>            <dbl>        <dbl>
## 1 Officia~ English       19460850     22162865
## 2 Officia~ French         7166700      6943800
## # ... with 2 more variables: most_at_work <dbl>,
## #   lang_known <dbl>
```

1.6 Exploring data with visualizations

Creating effective data visualizations is an essential piece to any data analysis. For the remainder of Chapter 1, we will learn how to use functions from the `tidyverse` to make visualizations that let us explore relationships in data. In particular, we'll develop a visualization of the language data collected in the 2016 Canadian census we've been working with that will help us understand two potential relationships in the data: first, the relationship between the number of people who speak a language as their mother tongue and the number of people who speak that language as their primary spoken language at home, and second, whether there is a pattern in the strength of this relationship in the higher level language categories (Official languages, Aboriginal languages, or non-official and non-Aboriginal languages). This is an example of an exploratory data analysis question: we are looking for relationships and

patterns within the data set we have, but are not trying to generalize what we find beyond this data set.

1.6.1 Using `ggplot` to create a scatter plot

Taking another look at our data set below, we can immediately see that the three columns (or variables) we are interested in visualizing—mother tongue, language spoken most at home, and higher level language category—are all in separate columns. In addition, there is a single row (or observation) for each language. The data are therefore in what we call a *tidy data* format. Tidy data is particularly important concept and will be a major focus in the remainder of this course: many of the functions from `tidyverse` require tidy data, including the `ggplot` function that we will use shortly for our visualization. We will formally introduce this concept in chapter 3.

```
can_lang
```

```
## # A tibble: 214 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~           590          235
##  2 Non-Off~ Afrikaa~         10260         4785
##  3 Non-Off~ Afro-As~          1150          445
##  4 Non-Off~ Akan (T~         13460         5985
##  5 Non-Off~ Albanian         26895        13135
##  6 Aborigi~ Algonqu~            45           10
##  7 Aborigi~ Algonqu~          1260          370
##  8 Non-Off~ America~          2685         3020
##  9 Non-Off~ Amharic          22465        12785
## 10 Non-Off~ Arabic          419890       223535
## # ... with 204 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

We will begin with a scatter plot of the `mother_tongue` and `most_at_home` columns from our data frame. To create a scatter plot of these two variables using the `ggplot` function, we do the following:

1. call the `ggplot` function
2. provide the name of the data frame as the first argument
3. call the aesthetic function, `aes`, to specify which column will correspond to the x-axis and which will correspond to the y-axis
4. add a `+` symbol at the end of the `ggplot` call to add a layer to the plot
5. call the `geom_point` function to tell R that we want to represent the data points as dots/points to create a scatter plot.

```
ggplot(can_lang, aes(x = most_at_home, y = mother_tongue)) +
  geom_point()
```

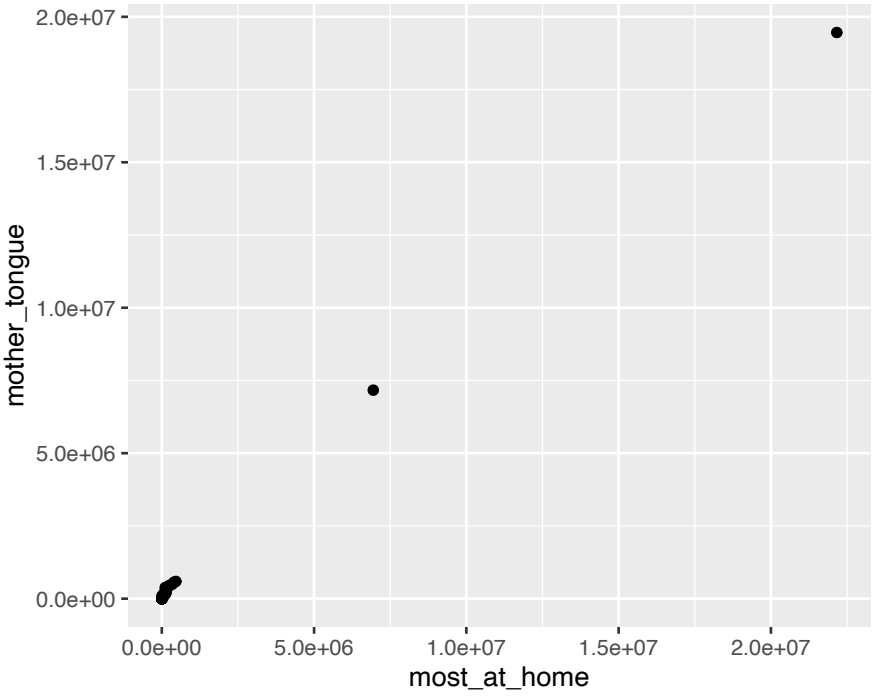


FIGURE 1.3: Scatter plot of number of Canadians reporting a language as their mother tongue vs the primary language at home

---

In case you have used R before and are curious: There are a small number of situations in which you can have a single R expression span multiple lines. Here, the + symbol at the end of the first line tells R that the expression isn't done yet and to continue reading on the next line. While not strictly necessary, this sort of pattern will appear a lot when using `ggplot` as it keeps things more readable.

1.6.2 Formatting ggplot objects

It is motivating and exciting that we have already been able to visualize our data to help answer our question, but we are not done yet! There is more we can (and should) do to improve the interpretability of the data visualization that we created. For example, by default, R uses the column names as the axis labels, however, usually these column names do not have enough information about the variable in the column. We really should replace this default with a more informative label. For the example above, the column name `mother_tongue` is used as the label for the y-axis, but most people will not know what that is. And even if they did, they will not know how we are measuring mother tongue, nor which group of people the measurements were taken. An axis label that read "Mother tongue (number of Canadian residents)" would be much more informative.

Adding additional layers to our visualizations that we create in `ggplot` is one common and easy way to improve and refine our data visualizations. New layers are added to `ggplot` objects using the `+` symbol. For example, we can use the `xlab` and `ylab` functions to add layers where we specify meaningful and informative labels for the x and y axes. Again, since we are specifying words (e.g. `"Mother tongue (number of Canadian residents)"`) as arguments to `xlab` and `ylab`, we surround them with double quotes. There are many more layers we can add to format the plot further, and we will explore these in later chapters.

```
ggplot(can_lang, aes(x = most_at_home, y = mother_tongue)) +
  geom_point() +
  xlab("Language spoken most at home (number of Canadian residents)") +
  ylab("Mother tongue (number of Canadian residents)")
```

Most of the data points from the 214 observations in this data set are bunched up in the lower left-handside of this visualization. This is because many many more people in Canada speak the two official languages (English and French). Thus to answer our question, we will need to adjust the scale of the x and y axes so that they are on a log scale. We can again add additional layers to the plot object using the `+` symbol to do this:

```
ggplot(can_lang, aes(x = most_at_home, y = mother_tongue)) +
  geom_point() +
  xlab("Language spoken most at home (number of Canadian residents)") +
  ylab("Mother tongue (number of Canadian residents)") +
  scale_x_log10(labels = scales::comma) +
  scale_y_log10(labels = scales::comma)
```

From this visualization we see that for the 214 languages in this data set, as

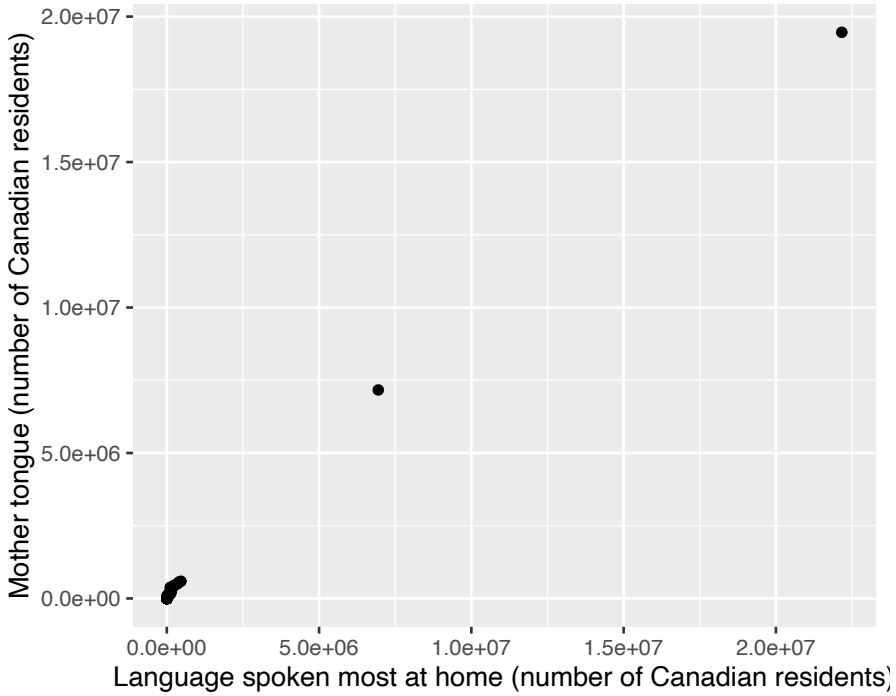


FIGURE 1.4: Scatter plot of number of Canadians reporting a language as their mother tongue vs the primary language at home with x and y labels

the number of people who have a language as their mother tongue increases, so does the number of people who speak that language at home. When we see two variables do this, we call this a *positive relationship*. Because the points are fairly close together, we can say that the relationship is strong. Because drawing a straight line through these points would fit the pattern we observe quite well, we say that it's linear.

Learning how to describe data visualizations is a very useful skill. We will provide descriptions for you in this course (as we did above) until we get to Chapter 4, which focuses on data visualization. Then, we will explicitly teach you how to do this yourself, and how to not over-state or over-interpret the results from a visualization.

1.6.3 Changing the units

What does it mean that 19,460,850 people reported that their mother tongue was English in the 2016 Canadian census? To really understand this number, we need context. In particular, how many people were in Canada when this

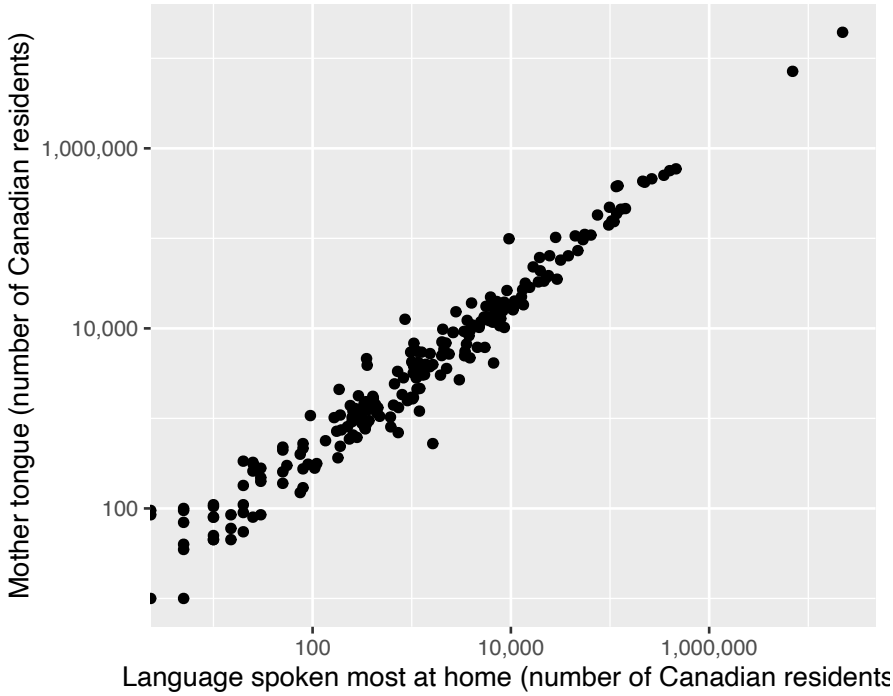


FIGURE 1.5: Scatter plot of number of Canadians reporting a language as their mother tongue vs the primary language at home with log adjusted x and y axes

data was collected? From the 2016 Canadian census profile, we can see that the population was reported to be 35,151,728 people. The count of the number of people who report that English is their mother tongue is much more meaningful when we report it in this context. We can even go a step further and transform this count to a relative frequency, or proportion, so that we can represent this as a single meaningful number in our data visualizations. We can do this by dividing the number of people reporting a given language as their mother tongue by the number of people who live in Canada. For example, the proportion of people who reported that their mother tongue was English in the 2016 Canadian census was 0.55.

We can use the `mutate` function in R to do this for all of the languages in the 2016 Canadian census data set. `mutate` is useful for creating new columns in a data frame, as well as transforming existing columns. It's general syntax is: `mutate(dataframe, column_to_create/transform = value/how_to_transform)`. Below we use mutate to calculate the proportion of people reporting a given

language as their mother tongue for all the languages in the `can_lang` data set:

```
can_lang <- mutate(can_lang, mother_tongue = mother_tongue / 35151728)
can_lang
```

```
## # A tibble: 214 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~    0.0000168           235
##  2 Non-Off~ Afrikaa~    0.000292           4785
##  3 Non-Off~ Afro-As~    0.0000327           445
##  4 Non-Off~ Akan (T~    0.000383           5985
##  5 Non-Off~ Albanian    0.000765          13135
##  6 Aborigi~ Algonqu~    0.00000128           10
##  7 Aborigi~ Algonqu~    0.0000358           370
##  8 Non-Off~ America~    0.0000764          3020
##  9 Non-Off~ Amharic     0.000639          12785
## 10 Non-Off~ Arabic      0.0119           223535
## # ... with 204 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

Let's also do this for the counts of the number of people who report that they speak a given language most often at home:

```
can_lang <- mutate(can_lang, most_at_home = most_at_home / 35151728)
can_lang
```

```
## # A tibble: 214 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~    0.0000168   0.00000669
##  2 Non-Off~ Afrikaa~    0.000292    0.000136
##  3 Non-Off~ Afro-As~    0.0000327   0.0000127
##  4 Non-Off~ Akan (T~    0.000383    0.000170
##  5 Non-Off~ Albanian    0.000765    0.000374
##  6 Aborigi~ Algonqu~    0.00000128  0.000000284
##  7 Aborigi~ Algonqu~    0.0000358   0.0000105
##  8 Non-Off~ America~    0.0000764   0.0000859
##  9 Non-Off~ Amharic     0.000639    0.000364
## 10 Non-Off~ Arabic      0.0119      0.00636
## # ... with 204 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

Finally, let's visualize the data now that we have represented it as proportions (and change our axis labels to reflect this change in units!):

```
ggplot(can_lang, aes(x = most_at_home, y = mother_tongue)) +
  geom_point() +
  xlab("Language spoken most at home (proportion of Canadian residents)") +
  ylab("Mother tongue (proportion of Canadian residents)") +
  scale_x_log10(labels = scales::comma) +
  scale_y_log10(labels = scales::comma)
```

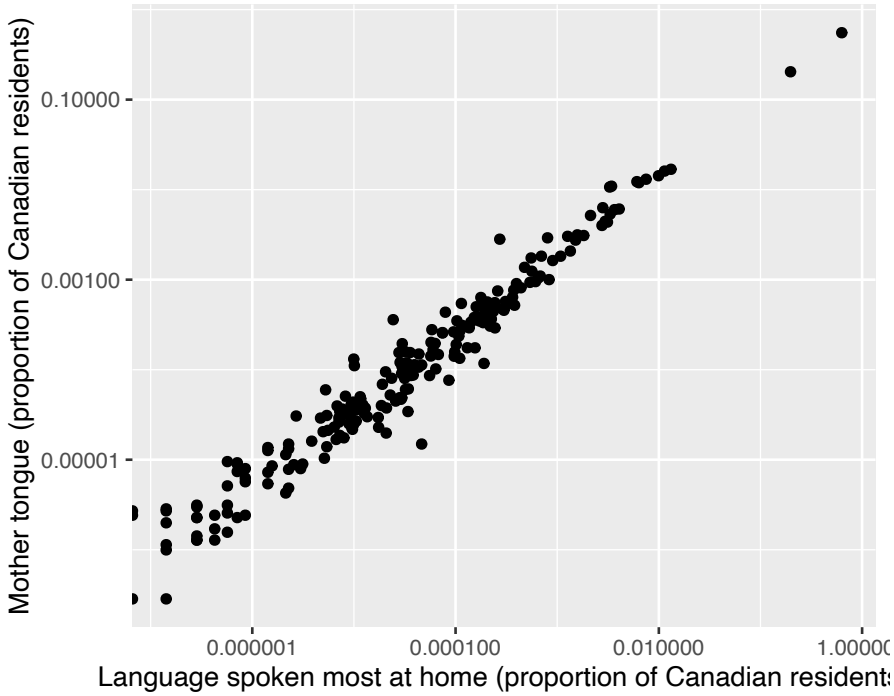


FIGURE 1.6: Scatter plot of proportion of Canadians reporting a language as their mother tongue vs the primary language at home

From the visualization above, we can now clearly see that not just a lot, but that the majority of Canadians reported English as their mother tongue and as the language they speak most often at home. Changing the units to include this context increases our understanding and allows us to interpret the numbers in our data set better.

1.6.4 Coloring points by group

Now we'll move onto the second part of our exploratory data analysis question: when considering the relationship between the number of people who have a language as their mother tongue and the number of people who speak that language at home, is there a pattern in the strength of this relationship in the higher-level language categories (Official languages, Aboriginal languages, or non-official and non-Aboriginal languages)? One common way to explore this is to colour the data points on the scatter plot we have already created by group/category. For example, given that we have the higher level language category for each language recorded in the 2016 Canadian census, we can colour the points in our previous scatter plot to represent each language's higher-level language category.

To do this, we modify our scatter plot code above. Specifically, we will add an argument to the `aes` function, specifying that the points should be coloured by the `category` column:

```
ggplot(can_lang, aes(x = most_at_home, y = mother_tongue, color = category)) +
  geom_point() +
  xlab("Language spoken most at home (proportion of Canadian residents)") +
  ylab("Mother tongue (proportion of Canadian residents)") +
  scale_x_log10(labels = scales::comma) +
  scale_y_log10(labels = scales::comma)
```

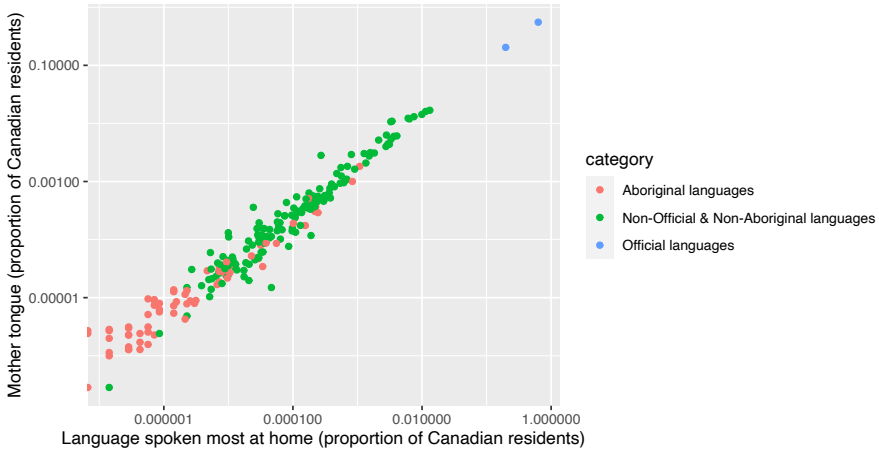


FIGURE 1.7: Scatter plot of proportion of Canadians reporting a language as their mother tongue vs the primary language at home coloured by language category

What do we see when considering the second part of our exploratory question?

Do we see a difference in the pattern of the relationship between the number of people who speak a language as their mother tongue and the number of people who speak a language as their primary spoken language at home between higher-level language categories? Probably not!

For each higher-level language category there appears to be a positive relationship between the number of people who speak a language as their mother tongue and the number of people who speak a language as their primary spoken language at home. This relationship looks similar, regardless of the category.

Does this mean that this relationship is positive for all languages in the world? Can we use this data visualization on its own to predict how many people have a given language as their mother tongue if we know how many people speak it as their primary language at home?

The answer to both these questions is "no." However, with this exploratory data analysis, we can create new hypotheses, ideas, and questions (like the ones at the beginning of this paragraph). Answering those questions would likely involve gathering additional data and doing more complex analyses, which we will see more of later in this course.

1.6.5 Putting it all together

Below, we put everything from this chapter together in one code chunk. We have added a few more layers to make the data visualization even more effective. Specifically we used have improved the visualizations accessibility by choosing colours that are easier to distinguish and also mapped category to shape, we handled the problem of overlapping data points by making them slightly transparent, and we changed the background from grey to white to improve the contrast. This demonstrates the power of R: in relatively few lines of code, we are able to create an entire data science workflow with a highly effective data visualization.

```
library(tidyverse)

can_lang <- read_csv("data/can_lang.csv")

can_lang <- mutate(can_lang, mother_tongue = mother_tongue / 35151728)
can_lang <- mutate(can_lang, most_at_home = most_at_home / 35151728)

ggplot(can_lang, aes(
  x = most_at_home,
  y = mother_tongue,
  colour = category,
  shape = category
```

```
)) + # map categories to different shapes
  geom_point(alpha = 0.6) + # set the transparency of the points
  scale_color_manual(values = c("deepskyblue2", "firebrick1", "black")) + # choose point colours
  xlab("Language spoken most at home (proportion of Canadian residents)") +
  ylab("Mother tongue (proportion of Canadian residents)") +
  scale_x_log10(labels = scales::comma) +
  scale_y_log10(labels = scales::comma) +
  theme_bw() # use a theme to have a white background
```

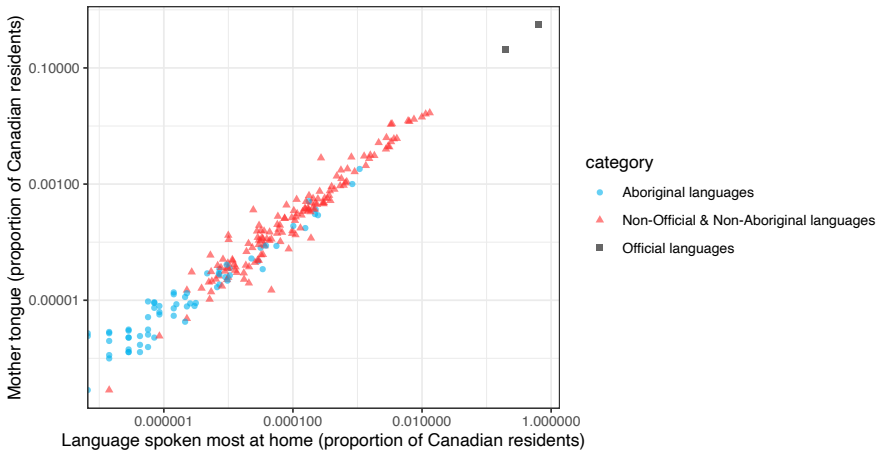


FIGURE 1.8: Putting it all together: Scatter plot of proportion of Canadians reporting a language as their mother tongue vs the primary language at home coloured by language category

2

Reading in data locally and from the web

2.1 Overview

In this chapter, you'll learn to read spreadsheet-like data of various formats into R from your local device and the web. "Reading" (or "loading") is the process of converting data (stored as plain text, a database, HTML, etc.) into an object (e.g., a data frame) that R can easily access and manipulate. Thus reading data is the gateway to any data analysis; you won't be able to analyze data unless you've loaded it first. And because there are many ways to store data, there are similarly many ways to read data into R. The more time you spend upfront matching the data reading method to the type of data you have, the less time you will have to devote to re-formatting, cleaning and wrangling your data (the second step to all data analyses). It's like making sure your shoelaces are tied well before going for a run so that you don't trip later on!

2.2 Chapter learning objectives

By the end of the chapter, students will be able to:

- define the following:
  - absolute file path
  - relative file path
  - url
- read data into R using a relative path and a url
- compare and contrast the following functions:
  - `read_csv`
  - `read_tsv`
  - `read_csv2`
  - `read_delim`
  - `read_excel`

- match the following `tidyverse` `read_*` function arguments to their descriptions:
  - `file`
  - `delim`
  - `col_names`
  - `skip`
- choose the appropriate `tidyverse` `read_*` function and function arguments to load a given plain text tabular data set into R
- use `readxl` package's `read_excel` function and arguments to load a sheet from an excel file into R
- connect to a database using the `DBI` package's `dbConnect` function
- list the tables in a database using the `DBI` package's `dbListTables` function
- create a reference to a database table that is queriable using the `tbl` from the `dbplyr` package
- retrieve data from a database query and bring it into R using the `collect` function from the `dbplyr` package
- use `write_csv` to save a data frame to a `.csv` file
- (*optional*) scrape data from the web
  - read/scrape data from an internet URL using the rvest `html_nodes` and `html_text` functions
  - compare downloading tabular data from a plain text file (e.g. `.csv`) from the web versus scraping data from a `.html` file

2.3 Absolute and relative file paths

When you load a data set into R, you first need to tell R where those files live. The file could live on your computer (*local*) or somewhere on the internet (*remote*). In this section, we will discuss the case where the file lives on your computer.

The place where the file lives on your computer is called the "path". You can think of the path as directions to the file. There are two kinds of paths: relative paths and absolute paths. A relative path is where the file is with respect to where you currently are on the computer (e.g., where the Jupyter notebook file that you're working in is). On the other hand, an absolute path is where the file is in respect to the base (or root) folder of the computer's filesystem.

Suppose our computer's filesystem looks like the picture below, and we are

working in the Jupyter notebook titled `worksheetk_02.ipynb`. If we want to read in the `.csv` file named `happiness_report.csv` into our Jupyter notebook using R, we could do this using either a relative or an absolute path. We show both choices below.

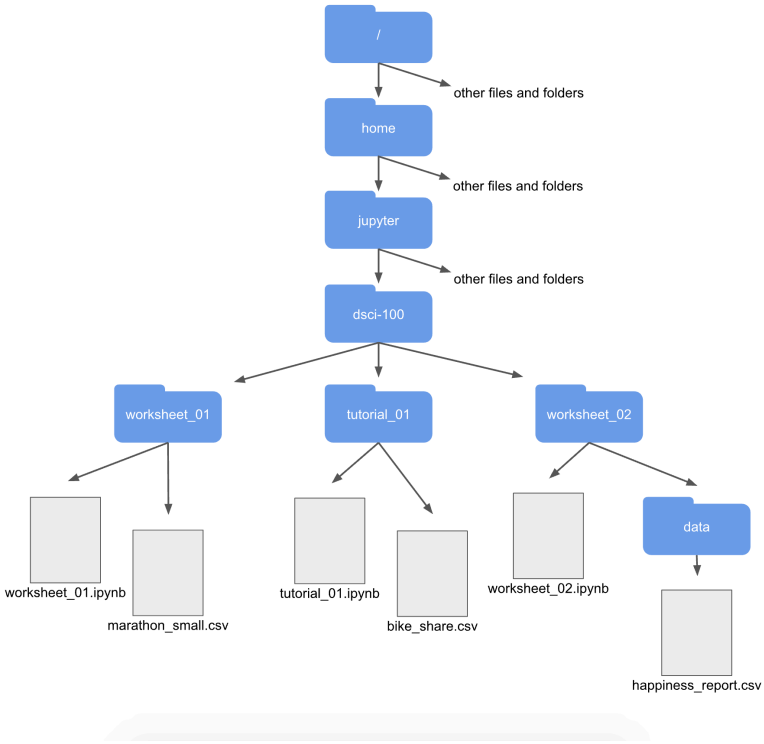


FIGURE 2.1: Example file system

Reading `happiness_report.csv` using a relative path:

```
happiness_data <- read_csv("data/happiness_report.csv")
```

Reading `happiness_report.csv` using an absolute path:

```
happiness_data <- read_csv("/home/jupyter/dsci-100/worksheet_02/data/happiness_report.csv")
```

So which one should you use? Generally speaking, to ensure your code can be run on a different computer, you should use relative paths. An added bonus is that it's also less typing! This is because the absolute path of a file (the names of folders between the computer's root `/` and the file) isn't usually the same across different computers. For example, suppose Fatima and Jayden are

working on a project together on the `happiness_report.csv` data. Fatima's file is stored at

`/home/Fatima/project/data/happiness_report.csv`,

while Jayden's is stored at

`/home/Jayden/project/data/happiness_report.csv`.

Even though Fatima and Jayden stored their files in the same place on their computers (in their home folders), the absolute paths are different due to their different usernames. If Jayden has code that loads the `happiness_report.csv` data using an absolute path, the code won't work on Fatima's computer. But the relative path from inside the `project` folder (`data/happiness_report.csv`) is the same on both computers; any code that uses relative paths will work on both!

See this video for another explanation:

Source: Udacity course "Linux Command Line Basics"[1]

2.4 Reading tabular data from a plain text file into R

Now we will learn more about reading tabular data from a plain text file into R, as well as how to write tabular data to a file. Last chapter, we learned about using the `tidyverse` `read_csv` function when reading files that match that function's expected defaults (column names are present, and commas are used as the delimiter/separator between columns). In this section, we will learn how to read files that do not satisfy the default expectations of `read_csv`.

Before we jump into the cases where the data aren't in the expected default format for `tidyverse` and `read_csv`, let's revisit the more straightforward case where the defaults hold, and the only argument we need to give to the function is the path to the file, `data/can_lang.csv`. The `can_lang` data set contains language data from the 2016 Canadian census. We put `data/` before the file's name when we are loading the data set because this data set is located in a sub-folder, named `data`, relative to where we are running our R code.

Here is what the file would look like in a plain text editor:

```
category,language,mother_tongue,most_at_home,most_at_work,lang_known
Aboriginal languages,"Aboriginal languages, n.o.s.",590,235,30,665
Non-Official & Non-Aboriginal languages,Afrikaans,10260,4785,85,23415
Non-Official & Non-Aboriginal languages,"Afro-
Asiatic languages, n.i.e.",1150,445,10,2775
```

[1] https://www.udacity.com/course/linux-command-line-basics--ud595

```
Non-Official & Non-Aboriginal languages,Akan (Twi),13460,5985,25,22150
Non-Official & Non-Aboriginal languages,Albanian,26895,13135,345,31930
Aboriginal languages,"Algonquian languages, n.i.e.",45,10,0,120
Aboriginal languages,Algonquin,1260,370,40,2480
Non-Official & Non-Aboriginal languages,American Sign Language,2685,3020,1145,21930
Non-Official & Non-Aboriginal languages,Amharic,22465,12785,200,33670
```

And here is a review of how we can use `read_csv` to load it into R. First we load the `tidyverse` package to gain access to useful functions for reading the data.

```
library(tidyverse)
```

Note: it is normal and expected that a message is printed out after loading the `tidyverse` and some packages. Generally, this message let's you know if functions from the different packages were loaded share the same name (which is confusing to R), and if so, which one you can access using just it's name (and which one you need to refer the package name and the function name to refer to it, this is called masking). Additionally, the `tidyverse` is a special R package - it is a meta-package that bundles together several related and commonly used packages. Because of this it lists the packages it does the job of loading. In future when we load this package in this book we will silence these messages to help with readability of the book.

Next we use `read_csv` to load the data into R, and in that call we specify the relative path to the file.

```
canlang_data <- read_csv("data/can_lang.csv")
```

```
##
## -- Column specification --------------------------------
## cols(
##   category = col_character(),
##   language = col_character(),
##   mother_tongue = col_double(),
##   most_at_home = col_double(),
##   most_at_work = col_double(),
```

```
##   lang_known = col_double()
## )
```

Note: it is also normal and expected that a message is printed out after using the `read_csv` and related functions. This message functions to let you know the data types of each of the columns that R inferred while reading the data into R. In future when we use this and related functions to load data in this book we will silence these messages to help with readability of the book.

```
canlang_data
```

```
## # A tibble: 214 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~           590          235
##  2 Non-Off~ Afrikaa~         10260         4785
##  3 Non-Off~ Afro-As~          1150          445
##  4 Non-Off~ Akan (T~         13460         5985
##  5 Non-Off~ Albanian         26895        13135
##  6 Aborigi~ Algonqu~            45           10
##  7 Aborigi~ Algonqu~          1260          370
##  8 Non-Off~ America~          2685         3020
##  9 Non-Off~ Amharic          22465        12785
## 10 Non-Off~ Arabic          419890       223535
## # ... with 204 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

2.4.1 Skipping rows when reading in data

Often times information about how data was collected, or other relevant information, is included at the top of the data file. This information is usually written in sentence and paragraph form, with no delimiter because it is not organized into columns. An example of this is shown below. This information gives the data scientist useful context and information about the data, however, it is not well formatted or intended to be read into a data frame cell along with the tabular data that follows later in the file.

```
Data source: https://ttimbers.github.io/canlang/
Data originally published in: Statistics Canada Census of Population 2016.
```

```
Reproduced and distributed on an as is basis with the permission of Statistics Canada.
category,language,mother_tongue,most_at_home,most_at_work,lang_known
Aboriginal languages,"Aboriginal languages, n.o.s.",590,235,30,665
Non-Official & Non-Aboriginal languages,Afrikaans,10260,4785,85,23415
Non-Official & Non-Aboriginal languages,"Afro-
Asiatic languages, n.i.e.",1150,445,10,2775
Non-Official & Non-Aboriginal languages,Akan (Twi),13460,5985,25,22150
Non-Official & Non-Aboriginal languages,Albanian,26895,13135,345,31930
Aboriginal languages,"Algonquian languages, n.i.e.",45,10,0,120
Aboriginal languages,Algonquin,1260,370,40,2480
Non-Official & Non-Aboriginal languages,American Sign Language,2685,3020,1145,21930
Non-Official & Non-Aboriginal languages,Amharic,22465,12785,200,33670
```

With this extra information being present at the top of the file, using `read_csv` as we did previously does not allow us to correctly load the data into R. In the case of this file we end up only reading in one column of the data set:

```
canlang_data <- read_csv("data/can_lang_meta-data.csv")
```

Note: In contrast to the normal and expected messages above, this time R printed out a warning for us indicating that there might be a problem with how our data is being read in.

```
canlang_data
```

```
## # A tibble: 217 x 1
##    `Data source: https://ttimbers.github.io/canlang/`
##    <chr>
##  1 Data originally published in: Statistics Canada Cen~
##  2 Reproduced and distributed on an as is basis with t~
##  3 category
##  4 Aboriginal languages
##  5 Non-Official & Non-Aboriginal languages
##  6 Non-Official & Non-Aboriginal languages
##  7 Non-Official & Non-Aboriginal languages
##  8 Non-Official & Non-Aboriginal languages
##  9 Aboriginal languages
## 10 Aboriginal languages
## # ... with 207 more rows
```

To successfully read data like this into R, the `skip` argument can be useful to tell R how many lines to skip before it should start reading in the data. In the example above, we would set this value to 3:

```
canlang_data <- read_csv("data/can_lang_meta-data.csv", skip = 3)
canlang_data
```

```
## # A tibble: 214 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~           590          235
##  2 Non-Off~ Afrikaa~         10260         4785
##  3 Non-Off~ Afro-As~          1150          445
##  4 Non-Off~ Akan (T~         13460         5985
##  5 Non-Off~ Albanian         26895        13135
##  6 Aborigi~ Algonqu~            45           10
##  7 Aborigi~ Algonqu~          1260          370
##  8 Non-Off~ America~          2685         3020
##  9 Non-Off~ Amharic          22465        12785
## 10 Non-Off~ Arabic          419890       223535
## # ... with 204 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

2.4.2 `read_delim` as a more flexible method to get tabular data into R

When our tabular data comes in a different format, we can use the `read_delim` function instead. For example, a different version of this same data set has no column names and uses tabs as the delimiter instead of commas.

Here is how the file would look in a plain text editor:

```
Aboriginal languages    Aboriginal languages, n.o.s.    590 235 30  665
Non-Official & Non-Aboriginal languages Afrikaans  10260  4785   85  23415
Non-Official       &       Non-Aboriginal       languages       Afro-
Asiatic languages, n.i.e.  1150    445 10  2775
Non-Official & Non-Aboriginal languages Akan (Twi)  13460  5985   25  22150
Non-Official & Non-Aboriginal languages Albanian   26895  13135  345 31930
Aboriginal languages    Algonquian languages, n.i.e.    45  10  0   120
Aboriginal languages    Algonquin   1260    370 40  2480
Non-Official & Non-Aboriginal languages American Sign Language  2685   3020   1145   21930
Non-Official & Non-Aboriginal languages Amharic 22465  12785  200 33670
Non-Official & Non-Aboriginal languages Arabic  419890  223535  5585   629055
```

To get this into R using the `read_delim()` function, we specify the first argument as the path to the file (as done with `read_csv`), and then provide

values to the `delim` argument (here a tab, which we represent by `"\t"`) and the `col_names` argument (here we specify that there are no column names to assign, and give it the value of `FALSE`). Both `read_csv()` and `read_delim()` have a `col_names` argument and the default is `TRUE`.

```
canlang_data <- read_delim("data/can_lang.tsv", delim = "\t", col_names = FALSE)
canlang_data
```

```
## # A tibble: 214 x 6
##    X1              X2              X3     X4    X5     X6
##    <chr>           <chr>        <dbl>  <dbl> <dbl>  <dbl>
##  1 Aboriginal la~ Aborigina~    590    235    30    665
##  2 Non-Official ~ Afrikaans   10260   4785    85  23415
##  3 Non-Official ~ Afro-Asia~   1150    445    10   2775
##  4 Non-Official ~ Akan (Twi)  13460   5985    25  22150
##  5 Non-Official ~ Albanian    26895  13135   345  31930
##  6 Aboriginal la~ Algonquia~     45     10     0    120
##  7 Aboriginal la~ Algonquin    1260    370    40   2480
##  8 Non-Official ~ American ~   2685   3020  1145  21930
##  9 Non-Official ~ Amharic     22465  12785   200  33670
## 10 Non-Official ~ Arabic     419890 223535  5585 629055
## # ... with 204 more rows
```

Data frames in R need to have column names, thus if you read data into R as a data frame without column names then R assigns column names for them. If you used the `read_*` functions to read the data into R, then R gives each column a name of X1, X2, …, XN, where N is the number of columns in the data set.

2.4.3 Reading tabular data directly from a URL

We can also use `read_csv()` or `read_delim()` (and related functions) to read in tabular data directly from a url that contains tabular data. In this case, we provide the url to `read_csv()` as the path to the file instead of a path to a local file on our computer. Similar to when we specify a path on our local computer, here we need to surround the url by quotes. All other arguments that we use are the same as when using these functions with a local file on our computer.

```
canlang_data <- read_csv("https://raw.githubusercontent.com/UBC-DSCI/introduction-to-datascience/m
canlang_data
```

```
## # A tibble: 214 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
```

```
##  1 Aborigi~ Aborigi~          590          235
##  2 Non-Off~ Afrikaa~        10260         4785
##  3 Non-Off~ Afro-As~         1150          445
##  4 Non-Off~ Akan (T~        13460         5985
##  5 Non-Off~ Albanian        26895        13135
##  6 Aborigi~ Algonqu~           45           10
##  7 Aborigi~ Algonqu~         1260          370
##  8 Non-Off~ America~         2685         3020
##  9 Non-Off~ Amharic         22465        12785
## 10 Non-Off~ Arabic         419890       223535
## # ... with 204 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

2.4.4 Previewing a data file before reading it into R

In all the examples above, we gave you previews of the data file before we read it into R. This is essential so that you can see whether or not there are column names, what the delimiters are, and if there are lines you need to skip. You should do this yourself when trying to read in data files. In Jupyter, you preview data as a plain text file by right-clicking on the file's name in the Jupyter home menu and selecting "Open with" and then selecting "Editor".

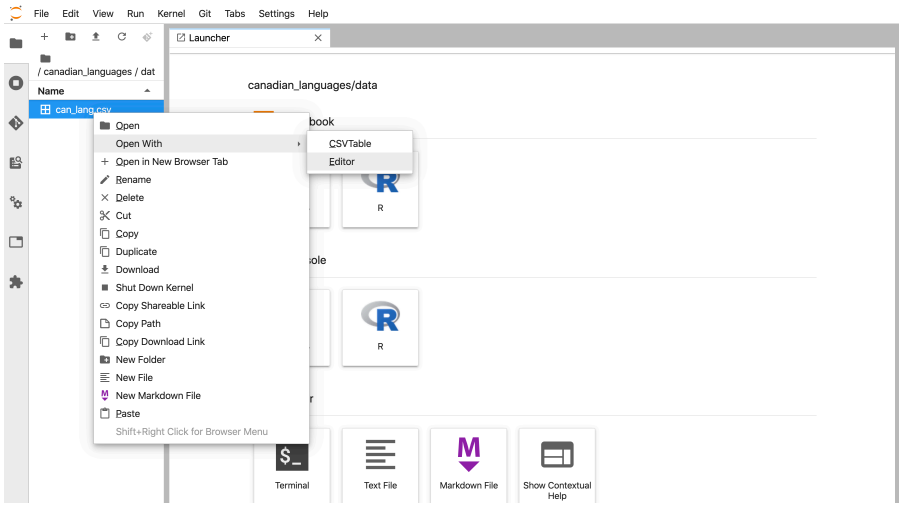


FIGURE 2.2: Opening data files with an editor in Jupyter

If you do not specify to open the data file with an editor, then Jupyter will render a nice table for you and you will not be able to see the column delimiters, and therefore you will not know which function to use, nor which arguments to use and values to specify for them.

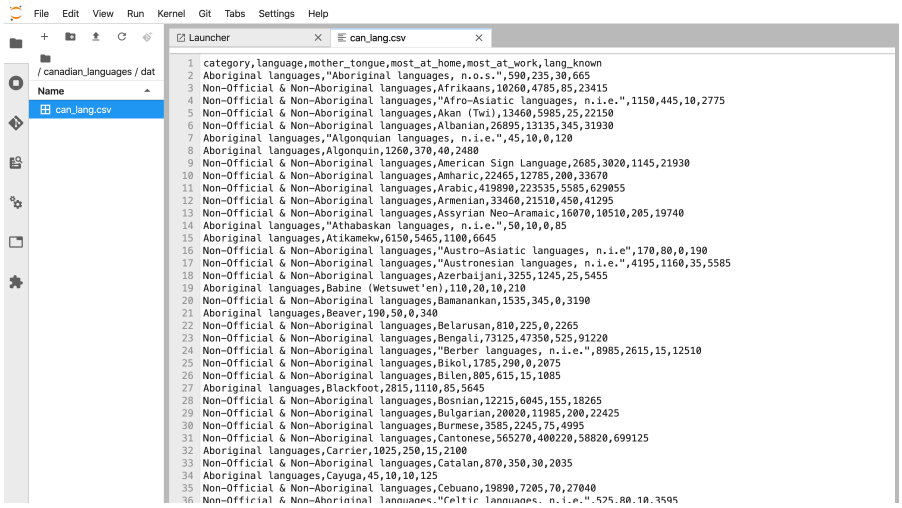


FIGURE 2.3: A data file as viewed in an editor in Jupyter

This is also demonstrated in the video below:

2.5 Reading data from an Microsoft Excel file

There are many other ways to store tabular data sets beyond plain text files, and similarly, many ways to load those data sets into R. For example, it is very common to encounter, and need to load into R, data stored as a Microsoft Excel spreadsheet (with the filename extension `.xlsx`). To be able to do this, a key thing to know is that even though `.csv` and `.xlsx` files look almost identical when loaded into Excel, the data themselves are stored completely differently. While `.csv` files are plain text files, where the characters you see when you open the file in a text editor are exactly the data they represent, this is not the case for `.xlsx` files. Take a look at what a `.xlsx` file would look like in a text editor:

```
,?'O
    _rels/.rels???J1??>E?{7?
<?V????w8?'J???'QrJ???Tf?d??d?o?wZ'???@>?4'?|??hlIo??F
t                                                    8f??3wn
????t??u"/
         %~Ed2??<?w??
                    ?Pd(??J-?E???7?'t(?-GZ?????y???c~N?g[^_r?4
                                                        yG?O
```

```
                                                    ?K??G?RPX?<??,?'O[Content_Types].xml???n?0E%?J
                                                                                    ]TUEe??O??c[???????6q
X?a??4VT?,D?Jq
               D
           ?????u?]??;??L?.8AhfNv}?hHF*??Jr?Q?%?g?U??CtX"8x>?.|????5j?/$???JE?c??~??4iw?????E;?+?S
                                      ?0???k????A?u?U?]??{#?:;/<?g?Cd????M+?=???Z?O??R+??u?P?X KV@??M
                                                  ??J?{???3j?h'??(q??U4J
                                                         ??=i?I'?b??[v?!??{gk?
                                                                          F2????v5yj??"J???,?d???J???C
?`$?4t?K?.;?%c?J??G<?H????
                                    X????z???6?????~q??X??????q^>??tH???*?D???M?g
??D?????????d?:g).?3.??j?P?F?'Oxl/_rels/workbook.xml.rels??Ak1??J?{7???R?^J?kk@Hf7??I?L???E]A?Þ?{a
                             O
?YY??4?L??S??k?252j??
??V ?C?g?C]???????
?
???E??TENyf6%
         ?Y????|??:%???}^ N?Q?N'????)??F?\??P?G??,?'O'xl/printerSettings/printerSettings1.bin?Wmn?
                                                                                  ??Sp>?G???q?#
                                                                                          ?I??5R'???q????(?L
??m??8F?5<  L`??`?A??2{dp??9R#?>7??Xu???/?X??HI?|?
                                                  ??r)???\?VA8?2dFfq???I]]o
5`????6A ?
```

This type of file representation allows Excel files to store additional things that you cannot store in a `.csv` file, such as fonts, text formatting, graphics, multiple sheets and more. And despite looking odd in a plain text editor, we can read Excel spreadsheets into R using the `readxl` package developed specifically for this purpose.

```
library(readxl)
canlang_data <- read_excel("data/can_lang.xlsx")
canlang_data
```

```
## # A tibble: 214 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~           590          235
##  2 Non-Off~ Afrikaa~         10260         4785
##  3 Non-Off~ Afro-As~          1150          445
##  4 Non-Off~ Akan (T~         13460         5985
##  5 Non-Off~ Albanian         26895        13135
##  6 Aborigi~ Algonqu~            45           10
##  7 Aborigi~ Algonqu~          1260          370
##  8 Non-Off~ America~          2685         3020
```

```
##  9 Non-Off~ Amharic          22465        12785
## 10 Non-Off~ Arabic          419890       223535
## # ... with 204 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

If the `.xlsx` file has multiple sheets, you have to use the `sheet` argument to specify the sheet number or name. You can also specify cell ranges using the `range` argument. This functionality is useful when a single sheet contains multiple tables (a sad thing that happens to many Excel spreadsheets).

As with plain text files, you should always explore the data file before importing it into R. Exploring the data beforehand helps you decide which arguments you need to to load the data into R successfully. If you do not have the Excel program on your computer, you can use other programs to preview the file. Examples include Google Sheets and Libre Office.

2.6 Reading data from a database

Another very common form of data storage is the relational database. There are many relational database management systems, such as SQLite[2], MySQL[3], PostgreSQL[4], Oracle[5], and many more. These different relational database management systems each have their own advantages and limitations. Almost all employ SQL (*structured query language*) to pull data from the database. Thankfully, you don't need to know SQL to analyze data from a database; several packages have been written that allows R to connect to relational databases and use the R programming language as the front end (what the user types in) to pull data from them. These different relational database management systems have their own advantages, limitations, and excels in particular scenarios. In this book, we will give examples of how to do this using R with SQLite and PostgreSQL databases.

2.6.1 Connecting to a database

2.6.1.1 Reading data from a SQLite database

SQLite is probably the simplest relational database that one can use in combination with R. SQLite databases are self-contained and usually stored and accessed locally on one computer. Data is usually stored in a file with a `.db`

[2] https://www.sqlite.org/index.html
[3] https://www.mysql.com/
[4] https://www.postgresql.org/
[5] https://www.oracle.com/ca-en/index.html

extension. Similar to Excel files, these are not plain text files and cannot be read in a plain text editor.

The first thing you need to do to read data into R from a database is to connect to the database. We do that using the `dbConnect` function from the `DBI` (database interface) package. This does not read in the data, but simply tells R where the database is and opens up a communication channel.

```
library(DBI)
con_lang_data <- dbConnect(RSQLite::SQLite(), "data/can_lang.db")
```

Often relational databases have many tables, and their power comes from the useful ways they can be joined. Thus anytime you want to access data from a relational database, you need to know the table names. You can get the names of all the tables in the database using the `dbListTables` function:

```
tables <- dbListTables(con_lang_data)
tables
```

```
## [1] "lang"
```

We only get one table name returned from calling `dbListTables`, which tells us that there is only one table in this database. To reference a table in the database to do things like select columns and filter rows, we use the `tbl` function from the `dbplyr` package. The package `dbplyr` allows us to work with data stored in databases as if they were local data frames, which is useful because we can do a lot with big datasets without actually having to bring these vast amounts of data into your computer!

```
library(dbplyr)
lang_db <- tbl(con_lang_data, "lang")
lang_db
```

```
## # Source:   table<lang> [?? x 6]
## # Database: sqlite 3.33.0
##   #       [/Users/tiffanytimbers/Documents/UBC-DSCI/introduction-to-
datascience/data/can_lang.db]
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~           590          235
##  2 Non-Off~ Afrikaa~         10260         4785
##  3 Non-Off~ Afro-As~          1150          445
##  4 Non-Off~ Akan (T~         13460         5985
##  5 Non-Off~ Albanian         26895        13135
##  6 Aborigi~ Algonqu~            45           10
```

```
##  7 Aborigi~ Algonqu~          1260          370
##  8 Non-Off~ America~          2685         3020
##  9 Non-Off~ Amharic          22465        12785
## 10 Non-Off~ Arabic          419890       223535
## # ... with more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

Although it looks like we just got a data frame from the database, we didn't! It's a *reference*, showing us data that is still in the SQLite database (note the first two lines of the output). It does this because databases are often more efficient at selecting, filtering and joining large data sets than R. And typically, the database will not even be stored on your computer, but rather a more powerful machine somewhere on the web. So R is lazy and waits to bring this data into memory until you explicitly tell it to do so using the `collect` function from the `dbplyr` package.

Here we will filter for only rows in the Aboriginal languages category according to the 2016 Canada Census, and then use `collect` to finally bring this data into R as a data frame.

```
aboriginal_lang_db <- filter(lang_db, category == "Aboriginal languages")
aboriginal_lang_db
```

```
## # Source:   lazy query [?? x 6]
## # Database: sqlite 3.33.0
##   #       [/Users/tiffanytimbers/Documents/UBC-DSCI/introduction-to-
datascience/data/can_lang.db]
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~           590          235
##  2 Aborigi~ Algonqu~            45           10
##  3 Aborigi~ Algonqu~          1260          370
##  4 Aborigi~ Athabas~            50           10
##  5 Aborigi~ Atikame~          6150         5465
##  6 Aborigi~ Babine ~           110           20
##  7 Aborigi~ Beaver             190           50
##  8 Aborigi~ Blackfo~          2815         1110
##  9 Aborigi~ Carrier           1025          250
## 10 Aborigi~ Cayuga              45           10
## # ... with more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

```
aboriginal_lang_data <- collect(aboriginal_lang_db)
aboriginal_lang_data
```

```
## # A tibble: 67 x 6
##    category language mother_tongue most_at_home
##    <chr>    <chr>            <dbl>        <dbl>
##  1 Aborigi~ Aborigi~           590          235
##  2 Aborigi~ Algonqu~            45           10
##  3 Aborigi~ Algonqu~          1260          370
##  4 Aborigi~ Athabas~            50           10
##  5 Aborigi~ Atikame~          6150         5465
##  6 Aborigi~ Babine ~           110           20
##  7 Aborigi~ Beaver             190           50
##  8 Aborigi~ Blackfo~          2815         1110
##  9 Aborigi~ Carrier           1025          250
## 10 Aborigi~ Cayuga              45           10
## # ... with 57 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

Why bother to use the `collect` function? The data looks pretty similar in both outputs shown above. And `dbplyr` provides lots of functions similar to `filter` that you can use to directly feed the database reference (what `tbl` gives you) into downstream analysis functions (e.g., `ggplot2` for data visualization and `lm` for linear regression modeling). However, this does not work in *every* case; look what happens when we try to use `nrow` to count rows in a data frame:

```
nrow(aboriginal_lang_db)
```

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

or `tail` to preview the last 6 rows of a data frame:

```
tail(aboriginal_lang_db)
```

```
## Error: tail() is not supported by sql sources
```

Additionally, some operations will not work to extract columns or single values from the reference given by the `tbl` function. Thus, once you have finished your data wrangling of the `tbl` database reference object, it is advisable to bring it into your local machine's memory using `collect` as a data frame.

> Warning: Usually, databases are very big! Reading the object into your local machine may give an error or take a lot of time to run so be careful if you plan to do this!

2.6.1.2 Reading data from a PostgreSQL database

PostgreSQL (also called Postgres) is a very popular and open-source option for relational database software. Unlike SQLite, PostgreSQL uses a client–server database engine, as it was designed to be used and accessed on a network. This means that you have to provide more information to R when connecting to Postgres databases. The additional information that you need to include when you call the `dbConnect` function is listed below:

- `dbname` - the name of the database (a single PostgreSQL instance can host more than one database)
- `host` - the URL pointing to where the database is located
- `port` - the communication endpoint between R and the PostgreSQL database (this is typically 5432 for PostgreSQL)
- `user` - the username for accessing the database
- `password` - the password for accessing the database

Additionally, we must use the `RPostgres` package instead of `RSQLite` in the `dbConnect` function call. Below we demonstrate how to connect to a version of the `can_mov_db` database, which contains information about Canadian movies (*note - this is a synthetic, or artificial, database*).

```
library(RPostgres)
can_mov_db_con <- dbConnect(RPostgres::Postgres(), dbname = "can_mov_db",
                            host = "r7k3-mds1.stat.ubc.ca", port = 5432,
                          user = "user0001", password = '#################')
```

2.6.2 Interacting with a database

After opening the connection, everything looks and behaves almost identically to when we were using an SQLite database in R. For example, we can again use `dbListTables` to find out what tables are in the `can_mov_db` database:

```
dbListTables(can_mov_db_con)
```

```
 [1] "themes"        "medium"        "titles"    "title_aliases"     "forms"
 [6] "episodes"      "names"     "names_occupations" "occupation"     "ratings"
```

We see that there are 10 tables in this database. Let's first look at the `"ratings"` table to find the lowest rating that exists in the `can_mov_db` database:

```
ratings_db <- tbl(can_mov_db_con, "ratings")
ratings_db
```

```
# Source:   table<ratings> [?? x 3]
# Database: postgres [user0001@r7k3-mds1.stat.ubc.ca:5432/can_mov_db]
   title              average_rating num_votes
   <chr>                       <dbl>     <int>
 1 The Grand Seduction          6.6       150
```

```
 2 Rhymes for Young Ghouls   6.3      1685
 3 Mommy                     7.5      1060
 4 Incendies                 6.1      1101
 5 Bon Cop, Bad Cop          7.0       894
 6 Goon                      5.5      1111
 7 Monsieur Lazhar           5.6       610
 8 What if                   5.3      1401
 9 The Barbarian Invations   5.8        99
10 Away from Her             6.9      2311
# … with more rows
```

To find the lowest rating that exists in the data base, we first need to extract the `average_rating` column using `select`:

```
avg_rating_db <- select(ratings_db, average_rating)
avg_rating_db
```

```
# Source:   lazy query [?? x 1]
# Database: postgres [user0001@r7k3-mds1.stat.ubc.ca:5432/can_mov_db]
   average_rating
            <dbl>
 1            6.6
 2            6.3
 3            7.5
 4            6.1
 5            7.0
 6            5.5
 7            5.6
 8            5.3
 9            5.8
10            6.9
# … with more rows
```

Next we use `min` to find the minimum rating in that column:

```
min(avg_rating_db)
```

```
Error in min(avg_rating_db) : invalid 'type' (list) of argument
```

Instead of the minimum, we get an error! This is another example of when we need to use the `collect` function to bring the data into R for further computation:

```
avg_rating_data <- collect(avg_rating_db)
min(avg_rating_data)
```

```
[1] 1
```

We see the lowest rating given to a movie is 1, indicating that it must have been a really bad movie…

Why should we bother with databases at all?

Opening a database stored in a `.db` file involved a lot more effort than just opening a `.csv`, `.tsv`, or any of the other plain text or Excel formats. It was a bit of a pain to use a database in that setting since we had to use `dbplyr` to translate `tidyverse`-like commands (`filter`, `select`, `head`, etc.) into SQL commands that the database understands. Not all `tidyverse` commands can currently be translated with SQLite databases. For example, we can compute a mean with an SQLite database but can't easily compute a median. So you might be wondering why should we use databases at all?

Databases are beneficial in a large-scale setting:

- they enable storing large data sets across multiple computers with automatic redundancy and backups
- they allow multiple users to access them simultaneously and remotely without conflicts and errors
- they provide mechanisms for ensuring data integrity and validating input
- they provide security to keep data safe For example, there are billions of Google searches conducted daily[6]. Can you imagine if Google stored all of the data from those queries in a single `.csv file`!? Chaos would ensue!

2.7 Writing data from R to a `.csv` file

At the middle and end of a data analysis, we often want to write a data frame that has changed (either through filtering, selecting, mutating or summarizing) to a file to share it with others or use it for another step in the analysis. The most straightforward way to do this is to use the `write_csv` function from the `tidyverse` package. The default arguments for this file are to use a comma (`,`) as the delimiter and include column names. Below we demonstrate creating a new version of the Canadian languages data set without the official languages category according to the Canadian 2016 Census, and then writing this to a `.csv` file:

```
no_official_lang_data <- filter(can_lang, category != "Official languages")
write_csv(no_official_lang_data, "data/no_official_languages.csv")
```

[6] https://www.internetlivestats.com/google-search-statistics/

2.8 Scraping data off the web using R

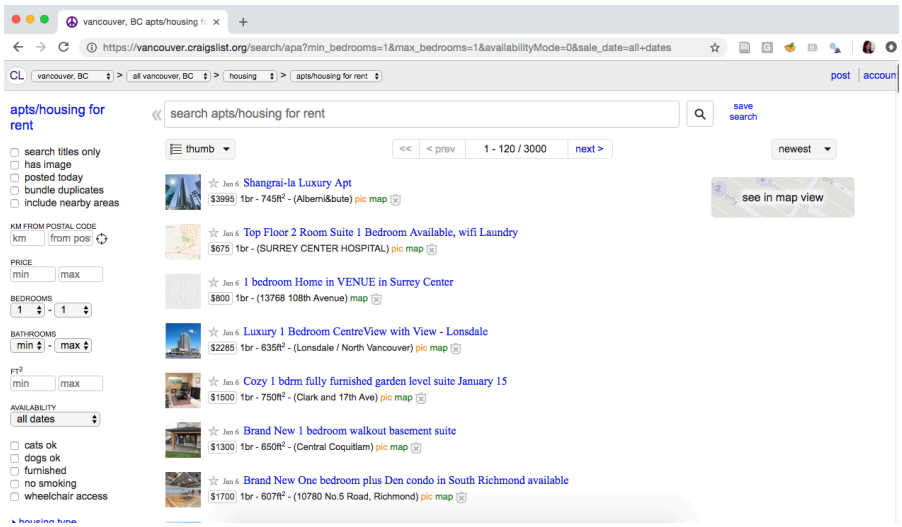
In the first part of this chapter, we learned how to read in data from plain text files that are usually "rectangular" in shape using the `tidyverse` `read_*` functions. Sadly, not all data comes in this simple format, but we can happily use many other tools to read in more messy/wild data formats. One common place people often want/need to read in data from is websites. Such data exists in a non-rectangular format. One quick and easy solution to get this data is to copy and paste it. However, this becomes painstakingly long and boring when there is a lot of data that needs gathering. And any time you start doing a lot of copying and pasting, you will likely introduce errors.

The formal name for gathering non-rectangular data from the web and transforming it into a more useful format for data analysis is **web scraping**. There are two different ways to do web scraping: 1) screen scraping (similar to copying and pasting from a website, but done in a programmatic way to minimize errors and maximize efficiency) and 2) web APIs (**a**pplication **p**rogramming **i**nterface) (a website that provides a programatic way of returning the data as JSON or XML files via http requests). In this course, we will explore the first method, screen scraping using R's `rvest` package[7].

2.8.1 HTML and CSS selectors

Before we jump into scraping, let's set up some motivation and learn a little bit about what the "source code" of a website looks like. Say we are interested in knowing the average rental price (per square footage) of the most recently available one-bedroom apartments in Vancouver from `https://vancouver.craigslist.org`. When we visit the Vancouver Craigslist website and search for one-bedroom apartments, this is what we are shown:

[7] `https://github.com/hadley/rvest`



From that page, it's pretty easy for our human eyes to find the apartment price and square footage. But how can we do this programmatically, so we don't have to copy and paste all these numbers? Well, we have to deal with the webpage source code, which we show a snippet of below (and link to the entire source code here[8]):

```
        <span class="result-meta">
                <span class="result-price">$800</span>

                <span class="housing">
                    1br -
                </span>

                <span class="result-hood"> (13768 108th Avenue)</span>

                <span class="result-tags">
                  <span class="maptag" data-pid="6786042973">map</span>
                </span>

                <span class="banish icon icon-trash" role="button">
              <span class="screen-reader-text">hide this posting</span>
                </span>

          <span class="unbanish icon icon-trash red" role="button" aria-
hidden="true"></span>
            <a href="#" class="restore-link">
                <span class="restore-narrow-text">restore</span>
```

[8] img/website_source.txt

```
            <span class="restore-wide-text">restore this posting</span>
            </a>

        </span>
    </p>
</li>
         <li class="result-row" data-pid="6788463837">

            <a  href="https://vancouver.craigslist.org/nvn/apa/d/north-
vancouver-luxury-1-bedroom/6788463837.html"             class="result-
image gallery" data-ids="1:00U0U_lLWbuS4jBYN,1:00T0T_9JYt6togdOB,1:00r0r_hlMkwxKqoeq,1:00n0n_2U8Stp
                <span class="result-price">$2285</span>
        </a>

    <p class="result-info">
        <span class="icon icon-star" role="button">
            <span class="screen-reader-text">favorite this post</span>
        </span>

                        <time  class="result-date"  datetime="2019-01-
06 12:06" title="Sun 06 Jan 12:06:01 PM">Jan  6</time>


            <a  href="https://vancouver.craigslist.org/nvn/apa/d/north-
vancouver-luxury-1-bedroom/6788463837.html"                       data-
id="6788463837" class="result-title hdrlnk">Luxury 1 Bedroom CentreView with View -
Lonsdale</a>
```

This is not easy for our human eyeballs to read! However, it is easy for us to use programmatic tools to extract the data we need by specifying which HTML tags (things inside < and > in the code above). For example, if we look in the code above and search for lines with a price, we can also look at the tags that are near that price and see if there's a common "word" we can use that is near the price but doesn't exist on other lines that have the information we are not interested in:

```
<span class="result-price">$800</span>
```

and

```
<span class="result-price">$2285</span>
```

What we can see is there is a special "word" here, "result-price", which appears only on the lines with prices and not on the other lines (that have information we are not interested in). This special word and the context in which is is used (learned from the other words inside the HTML tag) can be combined

to create something called a CSS selector. The CSS selector can then be used by R's `rvest` package to select the information we want (here price) from the website source code.

Now, many websites are quite large and complex, and so then is their website source code. And as you saw above, it is not easy to read and pick out the special words we want with our human eyeballs. So to make this easier, we will use the SelectorGadget tool. It is an open source tool that simplifies generating and finding CSS selectors. We recommend you use the Chrome web browser to use this tool, and install the selector gadget tool from the Chrome Web Store[9]. Here is a short video on how to install and use the SelectorGadget tool to get a CSS selector for use in web scraping:

From installing and using the selectorgadget as shown in the video above, we get the two CSS selectors `.housing` and `.result-price` that we can use to scrape information about the square footage and the rental price, respectively. The selector gadget returns them to us as a comma separated list (here `.housing , .result-price`), which is exactly the format we need to provide to R if we are using more than one CSS selector.

2.8.2 Are you allowed to scrape that website?

BEFORE scraping data from the web, you should always check whether or not you are **ALLOWED** to scrape it! There are two documents that are important for this: the robots.txt file and reading the website's Terms of Service document. The website's Terms of Service document is probably the more important of the two, and so you should look there first. What happens when we look at Craigslist's Terms of Service document? Well we read this:

"You agree not to copy/collect CL content via robots, spiders, scripts, scrapers, crawlers, or any automated or manual equivalent (e.g., by hand)."

source: https://www.craigslist.org/about/terms.of.use

Want to learn more about the legalities of web scraping and crawling? Read this interesting blog post titled "Web Scraping and Crawling Are Perfectly Legal, Right?" by Benoit Bernard[10] (this is optional, not required reading).

---

[9] https://chrome.google.com/webstore/detail/selectorgadget/mhjhnkcfbdhnjickkkdbjoemdmbfginb

[10] https://benbernardblog.com/web-scraping-and-crawling-are-perfectly-legal-right/

So what to do now? Well, we shouldn't scrape Craigslist! Let's instead scrape some data on the population of Canadian cities from Wikipedia (who's Terms of Service document[11] does not explicilty say do not scrape). In this video below we demonstrate using the selectorgadget tool to get CSS Selectors from Wikipedia's Canada[12] page to scrape a table that contains city names and their populations from the 2016 Canadian Census:

2.8.3 Using `rvest`

Now that we have our CSS selectors we can use `rvest` R package to scrape our desired data from the website. First we start by loading the `rvest` package:

```
library(rvest)
```

Next, we tell R what page we want to scrape by providing the webpage's URL in quotations to the function `read_html`:

```
page <- read_html("https://en.wikipedia.org/wiki/Canada")
```

Then we send the page object to the `html_nodes` function. We also provide that function with the CSS selectors we obtained from the selectorgadget tool. These should be surrounded by quotations. The `html_nodes` function select nodes from the HTML document using CSS selectors. Nodes are the HTML tag pairs as well as the content between the tags. For our CSS selector `td:nth-child(5)` and example node that would be selected would be: `<td style="text-align:left;background:#f0f0f0;"><a href="/wiki/London,_Ontario" title="London, Ontario">London</a></td>`

We will use `head()` here to limit the print output of these vectors to 6 lines.

```
population_nodes <- html_nodes(page, "td:nth-child(5) , td:nth-child(7) , .infobox:nth-child(122) 
head(population_nodes)
```

```
## {xml_nodeset (6)}
## [1] <td style="text-align:right;">5,928,040</td>
## [2] <td style="text-align:left;background:#f0f0f0;"> ...
## [3] <td style="text-align:right;">494,069\n</td>
## [4] <td style="text-align:right;">4,098,927</td>
## [5] <td style="text-align:left;background:#f0f0f0;"> ...
## [6] <td style="text-align:right;">406,074\n</td>
```

Next we extract the meaningful data from the HTML nodes using the

[11] https://foundation.wikimedia.org/wiki/Terms_of_Use/en
[12] https://en.wikipedia.org/wiki/Canada

`html_text` function. For our example, this functions only required argument is the an html_nodes object, which we named `rent_nodes`. In the case of this example node: `<td style="text-align:left;background:#f0f0f0;"><a href="/wiki/London,_Ontario" title="London, Ontario">London</a></td>`, the `html_text` function would return `London`.

```
population_text <- html_text(population_nodes)
head(population_text)
```

```
## [1] "5,928,040"              "London"
## [3] "494,069\n"              "4,098,927"
## [5] "St. Catharines-Niagara" "406,074\n"
```

Are we done? Not quite... If you look at the data closely you see that the data is not in an optimal format for data analysis. Both the city names and population are encoded as characters in a single vector instead of being in a data frame with one character column for city and one numeric column for population (think of how you would organize the data in a spreadsheet). Additionally, the populations contain commas (not useful for programmatically dealing with numbers), and some even contain a line break character at the end (`\n`). Next chapter we will learn more about data wrangling using R so that we can easily clean up this data with a few lines of code.

2.9 Additional resources

- Data import chapter[13] from R for Data Science[14] by Garrett Grolemund & Hadley Wickham

[13] https://r4ds.had.co.nz/data-import.html
[14] https://r4ds.had.co.nz/

3

Cleaning and wrangling data

3.1 Overview

This chapter will be centered around tools for cleaning and wrangling data that move data from its raw format into a format that is suitable for data analysis. They will be presented in the context of a real world data science application, providing more practice working through a whole case study.

3.2 Chapter learning objectives

By the end of the chapter, students will be able to:

- define the term "tidy data"
- discuss the advantages and disadvantages from storing data in a tidy data format
- recall and use the following tidyverse functions and operators for their intended data wrangling tasks:
  - `select`
  - `filter`
  - `mutate`
  - `%>%`
  - `%in%`
  - `pivot_longer`
  - `pivot_wider`
  - `separate`
  - `summarize`
  - `group_by`
  - `map`

3.3 Vectors and Data frames

At this point, we know how to load data into R from various file formats. Once loaded into R, most of the tools we have learned about for reading data into R represent the data as a data frame. So now we will spend some time learning more about data frames in R so that we have a better understanding of how we can use and manipulate these objects.

3.3.1 What is a data frame?

Let's first start by defining what a data frame is exactly. From a data perspective, it is a rectangle where the rows are the observations:

region	year	population
Toronto	2016	2235145
Vancouver	2016	1027613
Montreal	2016	1823281
Calgary	2016	544870
Ottawa	2016	571146
Winnipeg	2016	321484
Hamilton	2016	306034
Edmonton	2016	537634
Halifax	2016	187478
London	2016	220452
Victoria	2016	172559
St. John's	2016	92353
Saskatoon	2016	124766

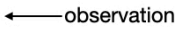


FIGURE 3.1: Rows are observations in a data frame

and the columns are the variables:

From a computer programming perspective, in R, a data frame is a special subtype of a list object whose elements (columns) are *vectors*. For example, the data frame below has 3 elements that are vectors whose names are `state`, `year` and `population`.

3.3.2 What is a vector?

In R, vectors are objects that can contain 1 or more elements. The vector elements are ordered, and they must all be of the same type. Common example

region	year	population
Toronto	2016	2235145
Vancouver	2016	1027613
Montreal	2016	1823281
Calgary	2016	544870
Ottawa	2016	571146
Winnipeg	2016	321484
Hamilton	2016	306034
Edmonton	2016	537634
Halifax	2016	187478
London	2016	220452
Victoria	2016	172559
St. John's	2016	92353
Saskatoon	2016	124766

FIGURE 3.2: Columns are variables in a data frame

region	year	population
Toronto	2016	2235145
Vancouver	2016	1027613
Montreal	2016	1823281
Calgary	2016	544870
Ottawa	2016	571146
Winnipeg	2016	321484
Hamilton	2016	306034
Edmonton	2016	537634
Halifax	2016	187478
London	2016	220452
Victoria	2016	172559
St. John's	2016	92353
Saskatoon	2016	124766

FIGURE 3.3: Data frame with 3 vectors

types of vectors are character (e.g., letter or words), numeric (whole numbers and fractions) and logical (e.g., `TRUE` or `FALSE`). In the vector shown below, the elements are of numeric type:

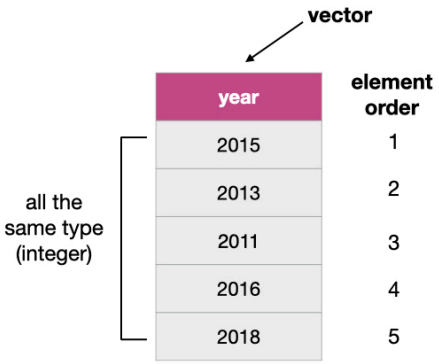


FIGURE 3.4: Example of a numeric type vector

3.3.3 How are vectors different from a list?

Lists are also objects in R that have multiple elements. Vectors and lists differ by the requirement of element type consistency. All elements within a single vector must be of the same type (e.g., all elements are numbers), whereas elements within a single list can be of different types (e.g., characters, numbers, logicals and even other lists can be elements in the same list).

3.3.4 What does this have to do with data frames?

As mentioned earlier, a data frame is really a special type of list where the elements can only be vectors. Representing data with such an object enables us to easily work with our data in a rectangular/spreadsheet-like manner, and to have columns/vectors of different characteristics associated/linked in one object. This is similar to a table in a spreadsheet or a database.

The functions from the `tidyverse` package that we are using often give us a special class of data frame, called a tibble. Tibbles have some additional features and benefits over the built-in data frame object. These include ability to add grouping (and other useful) attributes, as well as more predictable type preservation when subsetting. Because tibble is just a data frame with some added features, we will collectively refer to both built-in R data frames and tibbles as data frames in this book.

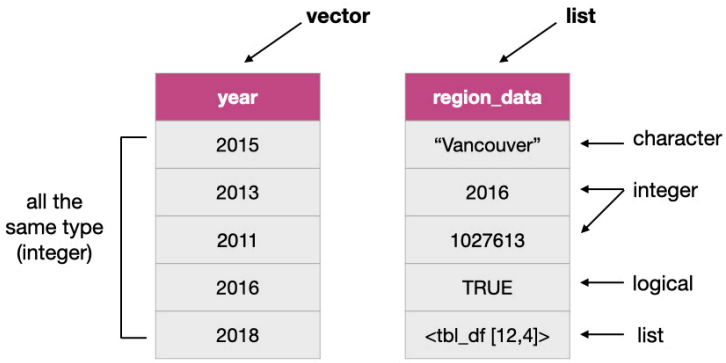
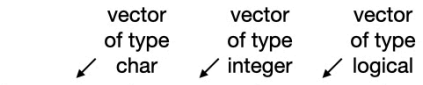


FIGURE 3.5: A vector versus a list



region	year	voted
Toronto	2016	TRUE
Vancouver	2016	TRUE
Montreal	2016	TRUE
Calgary	2016	TRUE
Ottawa	2016	TRUE
Winnipeg	2016	TRUE
Hamilton	2016	TRUE
Edmonton	2016	TRUE
Halifax	2016	TRUE
London	2016	TRUE
Victoria	2016	TRUE
St. John's	2016	TRUE
Saskatoon	2016	TRUE

FIGURE 3.6: Data frame and vector types

You can use the function `class` on a data object to assess whether a data frame is a built in R data frame or a tibble. If the data object is a date frame `class` will return `"data.frame"`, whereas if the data object is a tibble it will return `"tbl_df" "tbl" "data.frame"`. You can easily convert built in R data frames to tibbles using the `tidyverse` `as_tibble` function.

3.4 Tidy Data

There are many ways a spreadsheet-like data set can be organized. This chapter will focus on the *tidy data* format of organization, and how to make your raw (and likely messy) data tidy. We want to tidy our data because a variety of tools we would like to use in R are designed to work most effectively (and efficiently) with tidy data.

3.4.1 What is tidy data?

Tidy data satisfy the following three criteria (Wickham et al., 2014):

- each row is a single observation,
- each column is a single variable, and
- each value is a single cell (i.e., its row and column position in the data frame is not shared with another value)

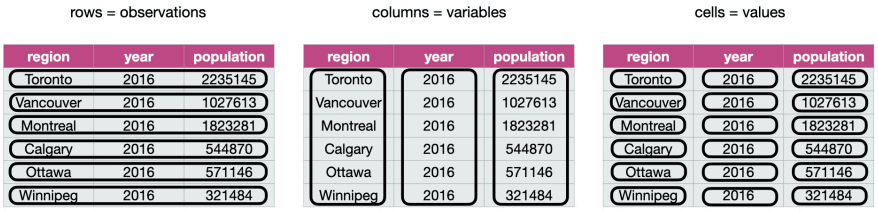


FIGURE 3.7: Tidy data

Definitions to know:

observation - all of the quantities or a qualities we collect from a given entity/object

variable - any characteristic, number, or quantity that can be measured or collected

value - a single collected quantity or a quality from a given entity/object

---

3.4.2 Why is tidy data important in R?

First, one of R's most popular plotting tool sets, the `ggplot2` package (which is one of the packages that the `tidyverse` package loads), expects the data to be in a tidy format. Second, most statistical analysis functions also expect data in a tidy format. Given that both of these tasks are central in almost all data analysis projects, it is well worth spending the time to get your data into a tidy format upfront. Luckily there are many well-designed `tidyverse` data cleaning/wrangling tools to help you easily tidy your data. Let's explore them now!

3.4.3 Going from wide to long (or tidy!) using `pivot_longer`

One common thing that often has to be done to get data into a tidy format is to combine columns that are really part of the same variable but currently stored in separate columns. Data is often stored in a wider, not tidy, format because this format is often more intuitive for human readability and understanding, and humans create data sets. We can use the function `pivot_longer`, which combines columns, thus making the data frame longer and narrower.

To learn how to use `pivot_longer`, we will work with a data set called `region_lang`[1], containing data retrieved from the 2016 Canadian census. For each census metropolitan area, this data set includes counts of how many Canadians cited each language as their mother tongue, the language spoken most often at home/work and which language they know.

We will use `read_csv` to import a subset of the `region_lang` data called `region_lang_top5_cities_wide.csv`, which contains only the counts of how many Canadians cited each language as their mother tongue for five major Canadian cities (Toronto, Montreal, Vancouver, Calgary and Edmonton). Our data set is stored in an untidy format, as shown below:

[1] https://ttimbers.github.io/canlang/

```
library(tidyverse)
lang_wide <- read_csv("data/region_lang_top5_cities_wide.csv")
lang_wide
```

```
## # A tibble: 214 x 7
##    category language Toronto Montréal Vancouver Calgary
##    <chr>    <chr>      <dbl>    <dbl>     <dbl>   <dbl>
##  1 Aborigi~ Aborigi~      80       30        70      20
##  2 Non-Off~ Afrikaa~     985       90      1435     960
##  3 Non-Off~ Afro-As~     360      240        45      45
##  4 Non-Off~ Akan (T~    8485     1015       400     705
##  5 Non-Off~ Albanian   13260     2450      1090    1365
##  6 Aborigi~ Algonqu~       5        5         0       0
##  7 Aborigi~ Algonqu~       5       30         5       5
##  8 Non-Off~ America~     470       50       265     100
##  9 Non-Off~ Amharic     7460      665      1140    4075
## 10 Non-Off~ Arabic     85175   151955     14320   18965
## # ... with 204 more rows, and 1 more variable:
## #   Edmonton <dbl>
```

What is wrong with our untidy format above? From a data analysis perspective, this format is not ideal because, in this format, the outcome of the variable *region* (Toronto, Montreal, Vancouver, Calgary and Edmonton) is stored as column names. Thus it is not easily accessible for the data analysis functions we will want to apply to our data set. Additionally, the values of the *mother tongue* variable are spread across multiple columns. This will prevent us from doing any desired visualization or statistical tasks until we somehow combine them into one column. For instance, suppose we want to know which languages had the highest number of Canadians reporting it as their mother tongue among all five regions? This question would be very difficult to answer with the data in its current format. It would be much easier to answer if we tidy our data first.

To accomplish this data transformation, we will use the `tidyverse` function `pivot_longer`. To use `pivot_longer` we need to specify the:

1. `data`: the data set
2. `cols` : the names of the columns that we want to combine
3. `names_to`: the name of a new column that will be created, whose values will come from the *names of the columns* that we want to combine
4. `values_to`: the name of a new column that will be created, whose values will come from the *values of the columns* we want to combine

For the above example, we use `pivot_longer` to combine the Toronto, Montreal, Vancouver, Calgary and Edmonton columns into a single column called `region`, and create a column called `mother_tongue` that contains the count of how many Canadians report each language as their mother tongue for each metropolitan area. We use a colon `:` between Toronto and Edmonton tells R to select all the columns in between Toronto and Edmonton:

```
lang_mother_tidy <- pivot_longer(lang_wide,
  cols = Toronto:Edmonton,
  names_to = "region",
  values_to = "mother_tongue"
)
lang_mother_tidy
```

```
## # A tibble: 1,070 x 4
##    category          language      region mother_tongue
##    <chr>             <chr>         <chr>          <dbl>
##  1 Aboriginal langu~ Aboriginal l~ Toron~            80
##  2 Aboriginal langu~ Aboriginal l~ Montr~            30
##  3 Aboriginal langu~ Aboriginal l~ Vanco~            70
##  4 Aboriginal langu~ Aboriginal l~ Calga~            20
##  5 Aboriginal langu~ Aboriginal l~ Edmon~            25
##  6 Non-Official & N~ Afrikaans     Toron~           985
##  7 Non-Official & N~ Afrikaans     Montr~            90
##  8 Non-Official & N~ Afrikaans     Vanco~          1435
##  9 Non-Official & N~ Afrikaans     Calga~           960
## 10 Non-Official & N~ Afrikaans     Edmon~           575
## # ... with 1,060 more rows
```

> **Splitting code across lines**: In the code above, the call to the `pivot_longer` function is split across several lines. This is allowed and encouraged when programming in R when your code line gets too long to read clearly. When doing this, it is important to end the line with a comma `,` so that R knows the function should continue to the next line.*

The data above is now tidy because all 3 criteria for tidy data have now been met:

1. All the variables (`category`, `language`, `region` and `mother_tongue`) are now their own columns in the data frame.
2. Each observation, i.e., each `category`, `language`, `region`, and count of Canadians where that language is the mother tongue, are in a single row.
3. Each value is a single cell, i.e., its row, column position in the data frame is not shared with another value.

3.4.4 Going from long to wide using `pivot_wider`

Suppose we have observations spread across multiple rows rather than in a single row. To tidy this data, we can use the function `pivot_wider`, which generally increases the number of columns (widens) and decreases the number of rows in a data set.

The data set `region_lang_top5_cities_long.csv` contains the number of Canadians reporting the primary language at home and work for five major cities (Toronto, Montreal, Vancouver, Calgary and Edmonton).

```
lang_long <- read_csv("data/region_lang_top5_cities_long.csv")
lang_long
```

```
## # A tibble: 2,140 x 5
##    region   category      language        type    count
##    <chr>    <chr>         <chr>           <chr>   <dbl>
##  1 Montréal Aboriginal l~ Aboriginal lan~ most_a~    15
##  2 Montréal Aboriginal l~ Aboriginal lan~ most_a~     0
##  3 Toronto  Aboriginal l~ Aboriginal lan~ most_a~    50
##  4 Toronto  Aboriginal l~ Aboriginal lan~ most_a~     0
##  5 Calgary  Aboriginal l~ Aboriginal lan~ most_a~     5
##  6 Calgary  Aboriginal l~ Aboriginal lan~ most_a~     0
##  7 Edmonton Aboriginal l~ Aboriginal lan~ most_a~    10
##  8 Edmonton Aboriginal l~ Aboriginal lan~ most_a~     0
##  9 Vancouv~ Aboriginal l~ Aboriginal lan~ most_a~    15
## 10 Vancouv~ Aboriginal l~ Aboriginal lan~ most_a~     0
## # ... with 2,130 more rows
```

What is wrong with this format above? In this example, each observation should be a language in a region. However, in the messy data set above, each observation is split across multiple two rows - one where the count for `most_at_home` is recorded and one where the count for `most_at_work` is recorded. Suppose we wanted to visualize the relationship between the number of Canadians reporting their primary language at home and at work. It would be difficult to do that with the data in its current format. To fix this, we will use `pivot_wider`, and we need to specify the:

1. `data`: the data set
2. `names_from`: the name of a the column from which to take the variable names
3. `values_from`: the name of the column from which to take the values

```
lang_home_tidy <- pivot_wider(lang_long,
  names_from = type,
  values_from = count
)
lang_home_tidy
```

```
## # A tibble: 1,070 x 5
##    region  category  language most_at_home most_at_work
##    <chr>   <chr>     <chr>           <dbl>        <dbl>
##  1 Montré~ Aborigin~ Aborigi~           15            0
##  2 Toronto Aborigin~ Aborigi~           50            0
##  3 Calgary Aborigin~ Aborigi~            5            0
##  4 Edmont~ Aborigin~ Aborigi~           10            0
##  5 Vancou~ Aborigin~ Aborigi~           15            0
##  6 Montré~ Non-Offi~ Afrikaa~           10            0
##  7 Toronto Non-Offi~ Afrikaa~          265            0
##  8 Calgary Non-Offi~ Afrikaa~          505           15
##  9 Edmont~ Non-Offi~ Afrikaa~          300            0
## 10 Vancou~ Non-Offi~ Afrikaa~          520           10
## # ... with 1,060 more rows
```

The data above is now tidy! We can go through the three criteria again to check that this data is a tidy data set.

1. All the variables are their own columns in the data frame, i.e., `most_at_home`, and `most_at_work` have been separated into their own columns in the data frame.
2. Each observation, i.e., each `category`, `language`, `region`, `most_at_home` and `most_at_work`, are in a single row.
3. Each value is a single cell, i.e., its row, column position in the data frame is not shared with another value.

You might notice that we have the same number of columns in our tidy data set as we did in our messy one. Therefore `pivot_wider` didn't really "widen" our data as the name suggests. However, if we had more than two categories in the original `type` column, then we would see the data set "widen."

3.4.5 Using `separate` to deal with multiple delimiters

Data are also not considered tidy when multiple values are stored in the same cell, as discussed above. In addition to the previous untidiness we addressed in the earlier versions of this data set, the one we show below is even messier:

the `Toronto`, `Montreal`, `Vancouver`, `Calgary` and `Edmonton` columns contain the number of Canadians reporting their primary language at home and work in one column separated by the delimiter "/". The column names are the values of a variable, AND each value does not have its own cell! To make this messy data tidy, we'll have to fix both of these issues.

```
lang_messy <- read_csv("data/region_lang_top5_cities_messy.csv")
lang_messy
```

```
## # A tibble: 214 x 7
##    category language Toronto Montréal Vancouver Calgary
##    <chr>    <chr>    <chr>   <chr>    <chr>     <chr>
##  1 Aborigi~ Aborigi~ 50/0    15/0     15/0      5/0
##  2 Non-Off~ Afrikaa~ 265/0   10/0     520/10    505/15
##  3 Non-Off~ Afro-As~ 185/10  65/0     10/0      15/0
##  4 Non-Off~ Akan (T~ 4045/20 440/0    125/10    330/0
##  5 Non-Off~ Albanian 6380/2~ 1445/20  530/10    620/25
##  6 Aborigi~ Algonqu~ 5/0     0/0      0/0       0/0
##  7 Aborigi~ Algonqu~ 0/0     10/0     0/0       0/0
##  8 Non-Off~ America~ 720/245 70/0     300/140   85/25
##  9 Non-Off~ Amharic  3820/55 315/0    540/10    2730/50
## 10 Non-Off~ Arabic   45025/~ 72980/1~ 8680/275  11010/~
## # ... with 204 more rows, and 1 more variable:
## #   Edmonton <chr>
```

First we'll use `pivot_longer` to create two columns, `region` and `value`, similar to what we did previously:

```
lang_messy_longer <- pivot_longer(lang_messy,
  cols = Toronto:Edmonton,
  names_to = "region",
  values_to = "value"
)
lang_messy_longer
```

```
## # A tibble: 1,070 x 4
##    category              language         region  value
##    <chr>                 <chr>            <chr>   <chr>
##  1 Aboriginal languages  Aboriginal lang~ Toronto 50/0
##  2 Aboriginal languages  Aboriginal lang~ Montré~ 15/0
```

```
##  3 Aboriginal languages  Aboriginal lang~ Vancou~ 15/0
##  4 Aboriginal languages  Aboriginal lang~ Calgary 5/0
##  5 Aboriginal languages  Aboriginal lang~ Edmont~ 10/0
##  6 Non-Official & Non-A~ Afrikaans        Toronto 265/0
##  7 Non-Official & Non-A~ Afrikaans        Montré~ 10/0
##  8 Non-Official & Non-A~ Afrikaans        Vancou~ 520/~
##  9 Non-Official & Non-A~ Afrikaans        Calgary 505/~
## 10 Non-Official & Non-A~ Afrikaans        Edmont~ 300/0
## # ... with 1,060 more rows
```

Then we'll use `separate` to split the `value` column into two columns, one that contains only the counts of Canadians that speak each language most at home, and one that contains the counts for most at work for each region. To use `separate` we need to specify the:

1. `data`: the data set
2. `col`: the name of a the column we need to split
3. `into`: a character vector of the new column names we would like to put the split data into
4. `sep`: the separator on which to split

```
lang_no_delimiter <- separate(lang_messy_longer,
  col = value,
  into = c("most_at_home", "most_at_work"),
  sep = "/"
)
lang_no_delimiter
```

```
## # A tibble: 1,070 x 5
##    category   language region most_at_home most_at_work
##    <chr>      <chr>    <chr>  <chr>        <chr>
##  1 Aborigina~ Aborigi~ Toron~ 50           0
##  2 Aborigina~ Aborigi~ Montr~ 15           0
##  3 Aborigina~ Aborigi~ Vanco~ 15           0
##  4 Aborigina~ Aborigi~ Calga~ 5            0
##  5 Aborigina~ Aborigi~ Edmon~ 10           0
##  6 Non-Offic~ Afrikaa~ Toron~ 265          0
##  7 Non-Offic~ Afrikaa~ Montr~ 10           0
##  8 Non-Offic~ Afrikaa~ Vanco~ 520          10
##  9 Non-Offic~ Afrikaa~ Calga~ 505          15
## 10 Non-Offic~ Afrikaa~ Edmon~ 300          0
## # ... with 1,060 more rows
```

You might notice in the table above the word `<chr>` appears beneath each

of the column names. The word under the column name indicates the data type of each column. Here all of our variables are "character" data types. Recall, a character data type is a letter or a number. In the previous example, `most_at_home` and `most_at_work` were `<dbl>` (double) (you can verify this by looking at the tables in the previous sections), which is a numeric data type. This change is due to the delimiter "/" when we read in this messy data set. R read the columns in as character types, and it stayed that way after we separated the columns.

Here it makes sense for `region`, `category`, and `language` to be stored as a character type. However, if we want to apply any functions that treat the `most_at_home` and `most_at_work` columns as a number (e.g. finding the maximum of the column), it won't be possible to do if the variable is stored as a `character`. R has a variety of data types, but here we will use the function `mutate` to convert these two columns to an "numeric" data type. `mutate` is a function that will allow us to create a new variable in our data set. We specify the data set in the first argument, and in the proceeding arguments, we specify the function we want to apply (`as.numeric`) to which columns (`most_at_home`, `most_at_work`). Then we give the mutated variable a new name. Here we are naming the columns the same names ("most_at_home", "most_at_work"), but you can call these mutated variables anything you'd like.

```
lang_no_delimiter <- mutate(lang_no_delimiter,
  most_at_home = as.numeric(most_at_home),
  most_at_work = as.numeric(most_at_work)
)
lang_no_delimiter
```

```
## # A tibble: 1,070 x 5
##    category   language region most_at_home most_at_work
##    <chr>      <chr>    <chr>         <dbl>        <dbl>
##  1 Aborigina~ Aborigi~ Toron~           50            0
##  2 Aborigina~ Aborigi~ Montr~           15            0
##  3 Aborigina~ Aborigi~ Vanco~           15            0
##  4 Aborigina~ Aborigi~ Calga~            5            0
##  5 Aborigina~ Aborigi~ Edmon~           10            0
##  6 Non-Offic~ Afrikaa~ Toron~          265            0
##  7 Non-Offic~ Afrikaa~ Montr~           10            0
##  8 Non-Offic~ Afrikaa~ Vanco~          520           10
##  9 Non-Offic~ Afrikaa~ Calga~          505           15
## 10 Non-Offic~ Afrikaa~ Edmon~          300            0
## # ... with 1,060 more rows
```

Now we see `<dbl>` appears under our columns, `most_at_home` and `most_at_work`, indicating they are double data types (which is one of the sub-types of numeric)!

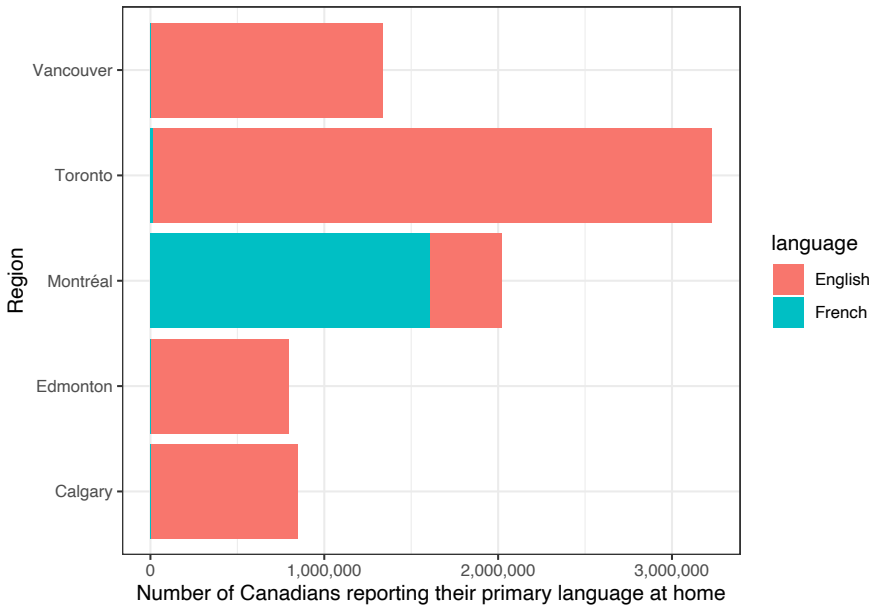
Is this data now tidy? Well, if we recall the three criteria for tidy data:

- each row is a single observation,
- each column is a single variable, and
- each value is a single cell.

We can see that this data now satisfies all three criteria, making it easier to analyze. For example, we could visualize how many people speak each of Canada's two official languages (English and French) as their primary language at home in these 5 regions. To do this, we first need to filter the data set for the rows that list the category as "Official languages", and then we can again use `ggplot` to create our data visualization. Here we create a bar chart to represent the counts for each region, and colour the counts by language.

```
official_langs <- filter(lang_no_delimiter, category == "Official languages")

ggplot(official_langs, aes(x = region, y = most_at_work, fill = language)) +
  geom_bar(stat = "identity") +
  scale_color_manual(values = c("deepskyblue2", "firebrick1")) +
  xlab("Region") +
  scale_y_continuous(
    name = "Number of Canadians reporting their primary language at home",
    labels = scales::comma
  ) +
  coord_flip() + # making bars horizontal
  theme_bw()
```



From this visualization, we can see that in Calgary, Edmonton, Toronto and Vancouver, English was reported as the most common primary language used at home compared to French. However, in Montreal, French was reported as the most common primary language used at home over English.

3.4.6 Notes on defining tidy data

Is there only one shape for tidy data for a given data set? Not necessarily! It depends on the statistical question you are asking and what the variables are for that question. For tidy data, each variable should be its own column. So, just as it's essential to match your statistical question with the appropriate data analysis tool (classification, clustering, hypothesis testing, etc.). It's important to match your statistical question with the appropriate variables and ensure they are represented as individual columns to make the data tidy.

3.5 Combining functions using the pipe operator, `%>%`:

In R, we often have to call multiple functions in a sequence to process a data frame. The basic ways of doing this can become quickly unreadable if there are many steps. For example, suppose we need to perform three operations on a data frame `data`:

1) add a new column `new_col` that is double another `old_col`
2) filter for rows where another column, `other_col`, is more than 5, and
3) select only the new column `new_col` for those rows.

One way of doing is to just write multiple lines of code, storing temporary objects as you go:

```
output_1 <- mutate(data, new_col = old_col * 2)
output_2 <- filter(output_1, other_col > 5)
output <- select(output_2, new_col)
```

This is difficult to understand for multiple reasons. The reader may be tricked into thinking the named `output_1` and `output_2` objects are important for some reason, while they are just temporary intermediate computations. Further, the reader has to look through and find where `output_1` and `output_2` are used in each subsequent line.

Another option for doing this would be to *compose* the functions:

```
output <- select(filter(mutate(data, new_col = old_col * 2), other_col > 5), new_col)
```

Code like this can also be difficult to understand. Functions compose (reading from left to right) in the *opposite order* in which they are computed by R (above, `mutate` happens first, then `filter`, then `select`). It is also just a really long line of code to read in one go.

The *pipe operator* `%>%` solves this problem, resulting in cleaner and easier-to-follow code. The below accomplishes the same thing as the previous two code blocks:

```
output <- data %>%
            mutate(new_col = old_col * 2) %>%
            filter(other_col > 5) %>%
            select(new_col)
```

You can think of the pipe as a physical pipe. It takes the output from the function on the left-hand side of the pipe, and passes it as the first argument to the function on the right-hand side of the pipe. Note here that we have again split the code across multiple lines for readability; R is fine with this, since it knows that a line ending in a pipe `%>%` is continued on the next line. Similarly, you see that after the first pipe, the remaining lines are indented until the end of the pipeline. This is not required for the R code to work, but again is used to aid in improving code readability.

Next, let's learn about the details of using the pipe, and look at some examples of how to use it in data analysis.

3.5.1 Using `%>%` to combine `filter` and `select`

Let's work with our tidy `lang_home_tidy` data set from above, which contains the number of Canadians reporting their primary language at home and work for five major cities (Toronto, Montreal, Vancouver, Calgary and Edmonton):

```
lang_home_tidy
```

```
## # A tibble: 1,070 x 5
##    region  category  language most_at_home most_at_work
##    <chr>   <chr>     <chr>           <dbl>        <dbl>
##  1 Montré~ Aborigin~ Aborigi~           15            0
##  2 Toronto Aborigin~ Aborigi~           50            0
##  3 Calgary Aborigin~ Aborigi~            5            0
##  4 Edmont~ Aborigin~ Aborigi~           10            0
##  5 Vancou~ Aborigin~ Aborigi~           15            0
##  6 Montré~ Non-Offi~ Afrikaa~           10            0
##  7 Toronto Non-Offi~ Afrikaa~          265            0
##  8 Calgary Non-Offi~ Afrikaa~          505           15
##  9 Edmont~ Non-Offi~ Afrikaa~          300            0
## 10 Vancou~ Non-Offi~ Afrikaa~          520           10
## # ... with 1,060 more rows
```

Suppose we want to create a subset of the data with only the languages and counts of each language spoken most at home for the city of Vancouver. To do this, we can use the functions `filter` and `select`. First, we use `filter` to create a data frame called `van_data` that contains only values for Vancouver. We then use `select` on this data frame to keep only the variables we want:

```
van_data <- filter(lang_home_tidy, region == "Vancouver")
van_data
```

```
## # A tibble: 214 x 5
##    region  category  language most_at_home most_at_work
##    <chr>   <chr>     <chr>           <dbl>        <dbl>
##  1 Vancou~ Aborigin~ Aborigi~           15            0
##  2 Vancou~ Non-Offi~ Afrikaa~          520           10
##  3 Vancou~ Non-Offi~ Afro-As~           10            0
##  4 Vancou~ Non-Offi~ Akan (T~          125           10
##  5 Vancou~ Non-Offi~ Albanian          530           10
##  6 Vancou~ Aborigin~ Algonqu~            0            0
##  7 Vancou~ Aborigin~ Algonqu~            0            0
##  8 Vancou~ Non-Offi~ America~          300          140
##  9 Vancou~ Non-Offi~ Amharic           540           10
## 10 Vancou~ Non-Offi~ Arabic           8680          275
```

```
## # ... with 204 more rows
```

```
van_data_selected <- select(van_data, language, most_at_home)
van_data_selected
```

```
## # A tibble: 214 x 2
##    language                       most_at_home
##    <chr>                                 <dbl>
##  1 Aboriginal languages, n.o.s.             15
##  2 Afrikaans                               520
##  3 Afro-Asiatic languages, n.i.e.           10
##  4 Akan (Twi)                              125
##  5 Albanian                                530
##  6 Algonquian languages, n.i.e.              0
##  7 Algonquin                                 0
##  8 American Sign Language                  300
##  9 Amharic                                 540
## 10 Arabic                                 8680
## # ... with 204 more rows
```

Although this is valid code, there is a more readable approach we could take by using the pipe, `%>%`. With the pipe, we do not need to create an intermediate object to store the output from `filter`. Instead we can directly send the output of `filter` to the input of `select`:

```
van_data_selected <- filter(lang_home_tidy, region == "Vancouver") %>%
  select(language, most_at_home)
van_data_selected
```

```
## # A tibble: 214 x 2
##    language                       most_at_home
##    <chr>                                 <dbl>
##  1 Aboriginal languages, n.o.s.             15
##  2 Afrikaans                               520
##  3 Afro-Asiatic languages, n.i.e.           10
##  4 Akan (Twi)                              125
##  5 Albanian                                530
##  6 Algonquian languages, n.i.e.              0
##  7 Algonquin                                 0
##  8 American Sign Language                  300
##  9 Amharic                                 540
## 10 Arabic                                 8680
## # ... with 204 more rows
```

But wait - why does our `select` function call look different in these two ex-

amples? When you use the pipe, the output of the function on the left is automatically provided as the first argument for the function on the right, and thus you do not specify that argument in that function call. In the code above, the first argument of `select` is the data frame we are `select`-ing from, which is provided by the output of `filter`.

As you can see, both of these approaches give us the same output, but the second approach is more clear and readable.

3.5.2 Using `%>%` with more than two functions

The `%>%` can be used with any function in R. Additionally, we can pipe together more than two functions. For example, we can pipe together three functions to order the rows by counts of the language most spoken at home for only the counts that are more than 10,000 and only include the region, language and count of Canadians reporting their primary language at home in our table.

To order the by counts of the language most spoken at home we will use another `tidyverse` function, `arrange`. This function takes column names as input and orders the rows in the data frame in ascending order based on the values in the columns. Here we use only one column for sorting (`most_at_home`), but more than one can also be used. To do this, list additional columns separated by commas. The order they are listed in indicates the order in which they will be used for sorting. This is much like how an English dictionary sorts words: first by the first letter, then by the second letter, and so on. *Note: If you want to sort in reverse order, you can pair a function called* `desc` *with* `arrange` *(e.g.,* `arrange(desc(column_name))`*).*

```
large_region_lang <- filter(lang_home_tidy, most_at_home > 10000) %>%
  select(region, language, most_at_home) %>%
  arrange(most_at_home)
large_region_lang
```

```
## # A tibble: 67 x 3
##    region    language most_at_home
##    <chr>     <chr>           <dbl>
##  1 Edmonton  Arabic          10590
##  2 Montréal  Tamil           10670
##  3 Vancouver Russian         10795
##  4 Edmonton  Spanish         10880
##  5 Edmonton  French          10950
##  6 Calgary   Arabic          11010
##  7 Calgary   Urdu            11060
##  8 Vancouver Hindi           11235
##  9 Montréal  Armenian        11835
## 10 Toronto   Romanian        12200
```

```
## # ... with 57 more rows
```

Note: You might also have noticed that we split the function calls across lines after the pipe, similar as to when we did this earlier in the chapter for long function calls. Again this is allowed and recommeded, especially when the piped function calls would create a long line of code. Doing this makes your code more readable. When you do this it is important to end each line with the pipe operator `%>%` to tell R that your code is continuing onto the next line.

3.6 Iterating over data with `group_by` + `summarize`

3.6.1 Calculating summary statistics:

As a part of many data analyses, we need to calculate a summary value for the data (a summary statistic). A useful `dplyr` function for doing this is `summarize`. Examples of summary statistics we might want to calculate are the number of observations, the average/mean value for a column, the minimum value for a column, etc. Below we show how to use the `summarize` function to calculate the minimum and maximum number of Canadians reporting a particular language as their primary language at home:

```
lang_summary <- summarize(lang_home_tidy,
  min_most_at_home = min(most_at_home),
  most_most_at_home = max(most_at_home)
)
lang_summary
```

```
## # A tibble: 1 x 2
##   min_most_at_home most_most_at_home
##              <dbl>             <dbl>
## 1                0           3836770
```

From this we see that there are some languages in the data set the no one speaks as their primary language at home, as well as that the most commonly spoken primary language at home is spoken by 3,836,770 people.

3.6.2 Calculating group summary statistics:

A common pairing with `summarize` is `group_by`. Pairing these functions together can let you summarize values for subgroups within a data set. For example, here, we can use `group_by` to group the regions and then calculate the minimum and maximum number of Canadians reporting the language as the primary language at home for each of the groups.

The `group_by` function takes at least two arguments. The first is the data frame that will be grouped, and the second and onwards are columns to use in the grouping. Here we use only one column for grouping (`region`), but more than one can also be used. To do this, list additional columns separated by commas.

```
lang_summary_by_region <- group_by(lang_home_tidy, region) %>%
  summarize(
    min_most_at_home = min(most_at_home),
    max_most_at_home = max(most_at_home)
  )
lang_summary_by_region
```

```
## # A tibble: 5 x 3
##   region    min_most_at_home max_most_at_home
##   <chr>                <dbl>            <dbl>
## 1 Calgary                  0          1065070
## 2 Edmonton                 0          1050410
## 3 Montréal                 0          2669195
## 4 Toronto                  0          3836770
## 5 Vancouver                0          1622735
```

3.6.3 Additional reading on the `dplyr` functions

As we breifly mentioned earlier in a note, the `tidyverse` is actually a *meta R package*: it installs and loads a collection of R packages that all follow the tidy data philosophy we discussed above. One of the `tidyverse` packages is `dplyr` - a data wrangling workhorse. You have already met six of the dplyr function (`select`, `filter`, `mutate`, `arrange`, `summarize`, and `group_by`). To learn more about those six and meet a few more useful functions, we recommend you checkout this chapter[2] of the Stat 545 book.

[2] http://stat545.com/block010_dplyr-end-single-table.html#where-were-we

3.7 Using purrr's map* functions to iterate

Where should you turn when you discover the next step in your data wrangling/cleaning process requires you to apply a function to each column in a data frame? For example, if you wanted to know the maximum value of each column in a data frame? Well, you could use `summarize` as discussed above. However, this becomes inconvenient when you have many columns, as `summarize` requires you to type out a column name and a data transformation for each summary statistic you want to calculate.

In cases like this, where you want to apply the same data transformation to all columns, it is more efficient to use `purrr`'s `map` function to apply it to each column. For example, let's find the maximum value of each column of the complete `region_lang` data frame by using `map` with the `max` function. First, let's peak at the data to familiarize ourselves with it:

```
region_lang <- read_csv("data/region_lang.csv")
region_lang
```

```
## # A tibble: 7,490 x 7
##    region category language mother_tongue most_at_home
##    <chr>  <chr>    <chr>            <dbl>        <dbl>
##  1 St. J~ Aborigi~ Aborigi~             5            0
##  2 Halif~ Aborigi~ Aborigi~             5            0
##  3 Monct~ Aborigi~ Aborigi~             0            0
##  4 Saint~ Aborigi~ Aborigi~             0            0
##  5 Sague~ Aborigi~ Aborigi~             5            5
##  6 Québec Aborigi~ Aborigi~             0            5
##  7 Sherb~ Aborigi~ Aborigi~             0            0
##  8 Trois~ Aborigi~ Aborigi~             0            0
##  9 Montr~ Aborigi~ Aborigi~            30           15
## 10 Kings~ Aborigi~ Aborigi~             0            0
## # ... with 7,480 more rows, and 2 more variables:
## #   most_at_work <dbl>, lang_known <dbl>
```

Next, we will select only the numeric columns of the data frame:

```
region_lang_numeric <- region_lang %>%
  select(mother_tongue:lang_known)
region_lang_numeric
```

```
## # A tibble: 7,490 x 4
##    mother_tongue most_at_home most_at_work lang_known
```

```
##            <dbl>        <dbl>        <dbl>      <dbl>
##  1             5            0            0          0
##  2             5            0            0          0
##  3             0            0            0          0
##  4             0            0            0          0
##  5             5            5            0          0
##  6             0            5            0         20
##  7             0            0            0          0
##  8             0            0            0          0
##  9            30           15            0         10
## 10             0            0            0          0
## # ... with 7,480 more rows
```

Next, we use `map` to apply the `max` function to each column. `map` takes two arguments, an object (a vector, data frame or list) that you want to apply the function to, and the function that you would like to apply. Here our arguments will be `region_lang_numeric` and `max`:

```
max_of_columns <- map(region_lang_numeric, max)
max_of_columns
```

```
## $mother_tongue
## [1] 3061820
##
## $most_at_home
## [1] 3836770
##
## $most_at_work
## [1] 3218725
##
## $lang_known
## [1] 5600480
```

Note: `purrr` is part of the tidyverse, and so like the `dplyr` and `ggplot` functions, once we call `library(tidyverse)` we do not need to load the `purrr` package separately.

Our output looks a bit weird… we passed in a data frame, but our output doesn't look like a data frame. As it so happens, it is *not* a data frame, but rather a plain vanilla list:

```
typeof(max_of_columns)
```

```
## [1] "list"
```

So what do we do? Should we convert this to a data frame? We could, but a simpler alternative is to just use a different `map_⋆` function from the `purrr` package. There are quite a few to choose from, they all work similarly, and their name reflects the type of output you want from the mapping operation:

`map` function	Output
`map()`	list
`map_lgl()`	logical vector
`map_int()`	integer vector
`map_dbl()`	double vector
`map_chr()`	character vector
`map_df()`	data frame

Let's get the columns' maximums again, but this time use the `map_df` function to return the output as a data frame:

```
max_of_columns <- map_df(region_lang_numeric, max)
max_of_columns
```

```
## # A tibble: 1 x 4
##   mother_tongue most_at_home most_at_work lang_known
##           <dbl>        <dbl>        <dbl>      <dbl>
## 1       3061820      3836770      3218725    5600480
```

Which `map_⋆` function you choose depends on what you want to do with the output; you don't always have to pick `map_df`!

What if you need to add other arguments to the functions you want to map? For example, what if there were NA values in our columns that we wanted to know the maximum of?

```
region_with_nas <- read_csv("data/region_lang_with_nas.csv")
region_with_nas
```

```
## # A tibble: 7,491 x 4
##    mother_tongue most_at_home most_at_work lang_known
##            <dbl>        <dbl>        <dbl>      <dbl>
##  1             5            5           NA         NA
##  2             5            0            0          0
```

```
##  3             5            0            0          0
##  4             0            0            0          0
##  5             0            0            0          0
##  6             5            5            0          0
##  7             0            0            0          0
##  8             0            0            0          0
##  9             0            0            0          0
## 10             0            0            0          0
## # ... with 7,481 more rows
```

```
map_df(region_with_nas, max)
```

```
## # A tibble: 1 x 4
##   mother_tongue most_at_home most_at_work lang_known
##           <dbl>        <dbl>        <dbl>      <dbl>
## 1       3061820      3836770           NA         NA
```

Notice `map_df()` returns `NA` for the `most_at_work` and `lang_known` variables since those columns contained NAs in the data frame. Thus, we also need to add the argument `na.rm = TRUE` to the `max` function so that we get a more useful value than `NA` returned (remember that is what happens with many of the built-in R statistical functions when NA's are present…). What we need to do in that case is add these additional arguments to the end of our call to to `map` and they will be passed to the function that we are mapping. An example of this is shown below:

```
map_df(region_with_nas, max, na.rm = TRUE)
```

```
## # A tibble: 1 x 4
##   mother_tongue most_at_home most_at_work lang_known
##           <dbl>        <dbl>        <dbl>      <dbl>
## 1       3061820      3836770      3218725    5600480
```

Now `map_df()` returns the maximum count for each column ignoring the NAs in the data set!

The `map_*` functions are generally quite useful for solving problems involving iteration/repetition. Additionally, their use is not limited to columns of a data frame; `map_*` functions can be used to apply functions to elements of a vector or list, and even to lists of data frames, or nested data frames.

3.8 Additional resources

Grolemund & Wickham's R for Data Science[3] has a number of useful sections that provide additional information:

- Data transformation[4]
- Tidy data[5]
- The `map_*` functions[6]

[3] https://r4ds.had.co.nz/
[4] https://r4ds.had.co.nz/transform.html
[5] https://r4ds.had.co.nz/tidy-data.html
[6] https://r4ds.had.co.nz/iteration.html#the-map-functions

4

Effective data visualization

4.1 Overview

This chapter will introduce concepts and tools relating to data visualization beyond what we have seen and practiced so far. We will focus on guiding principles for effective data visualization and explaining visualizations independent of any particular tool or programming language. In the process, we will cover some specifics of creating visualizations (scatter plots, bar charts, line graphs, and histograms) for data using R. There are external references that contain a wealth of additional information on the topic of data visualization:

- Professor Claus Wilke's Fundamentals of Data Visualization[1] has more details on general principles of effective visualizations
- Grolemund & Wickham's R for Data Science[2] chapter on creating visualizations using `ggplot2`[3] has a more in-depth introduction into the syntax and grammar of plotting with `ggplot2` specifically
- the ggplot2 reference[4] has a useful list of useful `ggplot2` functions

4.2 Chapter learning objectives

- Describe when to use the following kinds of visualizations to answer specific questions using a dataset:
  - scatter plots
  - line plots
  - bar plots
  - histogram plots
- Given a data set and a question, select from the above plot types and use R to create a visualization that best answers the question

[1] https://serialmentor.com/dataviz/
[2] https://r4ds.had.co.nz/
[3] https://r4ds.had.co.nz/data-visualisation.html
[4] https://ggplot2.tidyverse.org/reference/

- Given a visualization and a question, evaluate the effectiveness of the visualization and suggest improvements to better answer the question
- Referring to the visualization, communicate the conclusions in nontechnical terms
- Identify rules of thumb for creating effective visualizations
- Define the three key aspects of ggplot objects:
  - aesthetic mappings
  - geometric objects
  - scales
- Use the `ggplot2` package in R to create and refine the above visualizations using:
  - geometric objects: `geom_point`, `geom_line`, `geom_histogram`, `geom_bar`, `geom_vline`, `geom_hline`
  - scales: `scale_x_continuous`, `scale_y_continuous`
  - aesthetic mappings: `x`, `y`, `fill`, `colour`, `shape`
  - labelling: `xlab`, `ylab`, `labs`
  - font control and legend positioning: `theme`
  - flipping axes: `coord_flip`
  - subplots: `facet_grid`
- Describe the difference in raster and vector output formats
- Use `ggsave` to save visualizations in `.png` and `.svg` format

4.3 Choosing the visualization

4.3.0.1 *Ask a question, and answer it*

The purpose of a visualization is to answer a question about a data set of interest. So naturally, the first thing to do **before** creating a visualization is to formulate the question about the data that you are trying to answer. A good visualization will clearly answer your question without distraction; a *great* visualization will suggest even what the question was itself without additional explanation. Imagine your visualization as part of a poster presentation for a project; even if you aren't standing at the poster explaining things, an effective visualization will be able to convey your message to the audience.

Recall the different data analysis questions[5] from the first chapter of this book. With the visualizations we will cover in this chapter, we will be able to answer *only descriptive and exploratory* questions. Be careful not to try to answer any *predictive, inferential, causal or mechanistic* questions, as we have not learned the tools necessary to do that properly just yet.

As with most coding tasks, it is totally fine (and quite common) to make

[5] index.html#chapter-learning-objectives

mistakes and iterate a few times before you find the right visualization for your data and question. There are many different kinds of plotting graphics[6] available to use. For the kinds we will introduce in this course, the general rules of thumb are:

- **line plots** visualize trends with respect to an independent, ordered quantity (e.g. time)
- **histograms** visualize the distribution of one quantitative variable (i.e., all its possible values and how often they occur)
- **scatter plots** visualize the distribution / relationship of two quantitative variables
- **bar plots** visualize comparisons of amounts

All types of visualization have their (mis)uses, but three kinds are usually hard to understand or are easily replaced with an oft-better alternative. In particular, you should avoid **pie charts**; it is generally better to use bars, as it is easier to compare bar heights than pie slice sizes. You should also not use **3-D visualizations**, as they are typically hard to understand when converted to a static 2-D image format. Finally, do not use tables to make numerical comparisons; humans are much better at quickly processing visual information than text and math. Bar plots are again typically a better alternative.

4.4 Refining the visualization

4.4.0.1 *Convey the message, minimize noise*

Just being able to make a visualization in R with `ggplot2` (or any other tool for that matter) doesn't mean that it effectively communicates your message to others. Once you have selected a broad type of visualization to use, you will have to refine it to suit your particular need. Some rules of thumb for doing this are listed below. They generally fall into two classes: you want to *make your visualization convey your message*, and you want to *reduce visual noise* as much as possible. Humans have limited cognitive ability to process information; both of these types of refinement aim to reduce the mental load on your audience when viewing your visualization, making it easier for them to understand and remember your message quickly.

Convey the message

- Make sure the visualization answers the question you have asked most simply and plainly as possible.
- Use legends and labels so that your visualization is understandable without reading the surrounding text.

[6] https://serialmentor.com/dataviz/directory-of-visualizations.html

- Ensure the text, symbols, lines, etc., on your visualization are big enough to be easily read.
- Ensure the data are clearly visible; don't hide the shape/distribution of the data behind other objects (e.g. a bar).
- Make sure to use colour schemes that are understandable by those with colourblindness (a surprisingly large fraction of the overall population). For example, colorbrewer.org[7] and the `RColorBrewer` R package provide the ability to pick such colour schemes, and you can check your visualizations after you have created them by uploading to online tools such as the colour blindness simulator[8].
- Redundancy can be helpful; sometimes conveying the same message in multiple ways reinforces it for the audience.

Minimize noise

- Use colours sparingly. Too many different colours can be distracting, create false patterns, and detract from the message.
- Be wary of overplotting. If your plot has too many dots or lines and starts to look like a mess, you need to do something different.
- Only make the plot area (where the dots, lines, bars are) as big as needed. Simple plots can be made small.
- Don't adjust the axes to zoom in on small differences. If the difference is small, show that it's small!

4.5 Creating visualizations with `ggplot2`

4.5.0.1 *Build the visualization iteratively*

This section will cover examples of how to choose and refine a visualization given a data set and a question that you want to answer, and then how to create the visualization in R using `ggplot2`. To use the `ggplot2` package, we need to load the `tidyverse` metapackage.

```
library(tidyverse)
```

[7] https://colorbrewer2.org
[8] https://www.color-blindness.com/coblis-color-blindness-simulator/

4.5.1 The Mauna Loa CO2 data set

The Mauna Loa CO2 data set[9], curated by Dr. Pieter Tans, NOAA/GML[10] and Dr. Ralph Keeling, Scripps Institution of Oceanography[11] records the atmospheric concentration of carbon dioxide (CO2, in parts per million) at the Mauna Loa research station in Hawaii from 1959 onwards.

Question: Does the concentration of atmospheric CO2 change over time, and are there any interesting patterns to note?

```
# mauna loa carbon dioxide data
co2_df <- read_csv("data/mauna_loa.csv") %>%
  filter(ppm > 0, date_decimal < 2000)
co2_df
```

```
## # A tibble: 495 x 4
##     year month date_decimal   ppm
##    <dbl> <dbl>        <dbl> <dbl>
##  1  1958     3        1958.  316.
##  2  1958     4        1958.  317.
##  3  1958     5        1958.  318.
##  4  1958     7        1959.  316.
##  5  1958     8        1959.  315.
##  6  1958     9        1959.  313.
##  7  1958    11        1959.  313.
##  8  1958    12        1959.  315.
##  9  1959     1        1959.  316.
## 10  1959     2        1959.  316.
## # ... with 485 more rows
```

Since we are investigating a relationship between two variables (CO2 concentration and date), a scatter plot is a good place to start. Scatter plots show the data as individual points with `x` (horizontal axis) and `y` (vertical axis) coordinates. Here, we will use the decimal date as the `x` coordinate and CO2 concentration as the `y` coordinate. When using the `ggplot2` package, we create the plot object with the `ggplot` function; there are a few basic aspects of a plot that we need to specify:

- the *data*: the name of the data frame object that we would like to visualize
  - here, we specify the `co2_df` data frame
- the *aesthetic mapping*: tells `ggplot` how the columns in the data frame map to properties of the visualization
  - to create an aesthetic mapping, we use the `aes` function

[9] https://www.esrl.noaa.gov/gmd/ccgg/trends/data.html
[10] https://www.esrl.noaa.gov/gmd/staff/Pieter.Tans/
[11] https://scrippsco2.ucsd.edu/

  - here, we set the plot `x` axis to the `date_decimal` variable, and the plot `y` axis to the `ppm` variable
- the *geometric object*: specifies how the mapped data should be displayed
  - to create a geometric object, we use a `geom_*` function (see the ggplot reference[12] for a list of geometric objects)
  - here, we use the `geom_point` function to visualize our data as a scatterplot

We could pass many other possible arguments to the aesthetic mapping and geometric object to change how the plot looks. For the purposes of quickly testing things out to see what they look like, though, we can just go with the default settings:

```
co2_scatter <- ggplot(co2_df, aes(x = date_decimal, y = ppm)) +
  geom_point()
co2_scatter
```

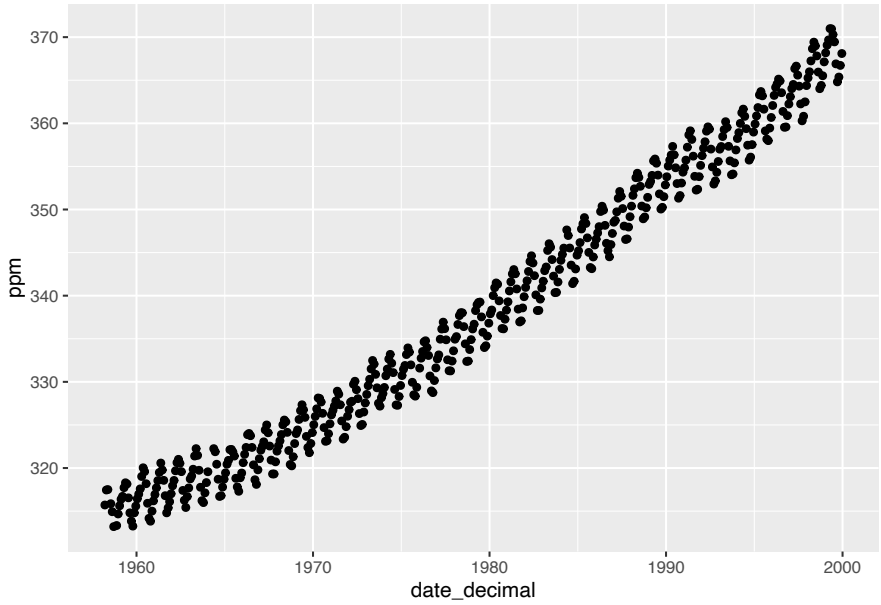


FIGURE 4.1: Scatter plot of atmospheric concentration of CO2 over time

Certainly, the visualization shows a clear upward trend in the atmospheric concentration of CO2 over time. This plot answers the first part of our question in the affirmative, but that appears to be the only conclusion one can make from the scatter visualization. However, since time is an ordered quantity,

[12] https://ggplot2.tidyverse.org/reference/

we can try using a line plot instead using the `geom_line` function. Line plots require that their `x` coordinate orders the data, and connect the sequence of `x` and `y` coordinates with line segments. Let's again try this with just the default arguments:

```
co2_line <- ggplot(co2_df, aes(x = date_decimal, y = ppm)) +
  geom_line()
co2_line
```

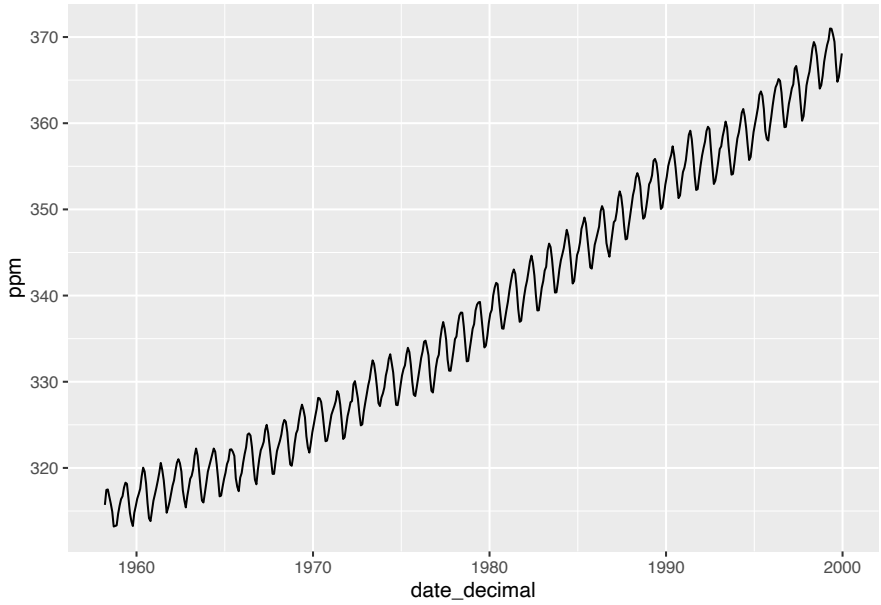


FIGURE 4.2: Line plot of atmospheric concentration of CO2 over time

Aha! There *is* another interesting phenomenon in the data: in addition to increasing over time, the concentration seems to oscillate as well. Given the visualization as it is now, it is still hard to tell how fast the oscillation is, but nevertheless, the line seems to be a better choice for answering the question than the scatter plot was. The comparison between these two visualizations illustrates a common issue with scatter plots: often, the points are shown too close together or even on top of one another, muddling information that would otherwise be clear (*overplotting*).

Now that we have settled on the rough details of the visualization, it is time to refine things. This plot is fairly straightforward, and there is not much visual noise to remove. But there are a few things we must do to improve clarity, such as adding informative axis labels and making the font a more readable

size. To add axis labels, we use the `xlab` and `ylab` functions. To change the font size, we use the `theme` function with the `text` argument:

```
co2_line <- ggplot(co2_df, aes(x = date_decimal, y = ppm)) +
  geom_line() +
  xlab("Year") +
  ylab("Atmospheric CO2 (ppm)") +
  theme(text = element_text(size = 18))
co2_line
```

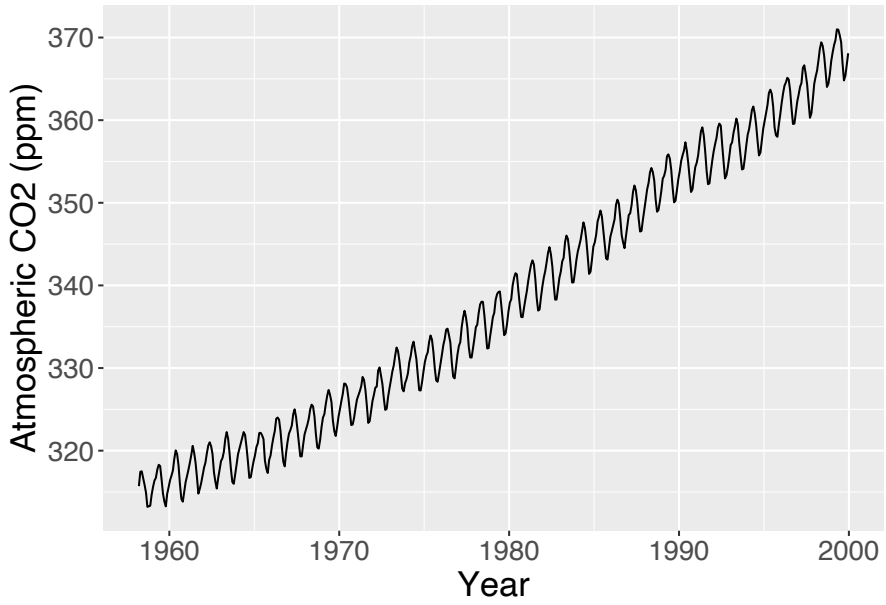


FIGURE 4.3: Line plot of atmospheric concentration of CO2 over time with clearer axes and labels

Finally, let's see if we can better understand the oscillation by changing the visualization slightly. Note that it is totally fine to use a small number of visualizations to answer different aspects of the question you are trying to answer. We will accomplish this by using *scales*, another important feature of `ggplot2` that easily transforms the different variables and set limits. We scale the horizontal axis using the `scale_x_continuous` function, and the vertical axis with the `scale_y_continuous` function. We can transform the axis by passing the `trans` argument, and set limits by passing the `limits` argument. In particular, here, we will use the `scale_x_continuous` function with the `limits` argument to zoom in on just five years of data (say, 1990-1995):

```
co2_line <- ggplot(co2_df, aes(x = date_decimal, y = ppm)) +
  geom_line() +
  xlab("Year") +
  ylab("Atmospheric CO2 (ppm)") +
  scale_x_continuous(limits = c(1990, 1995)) +
  theme(text = element_text(size = 18))
co2_line
```

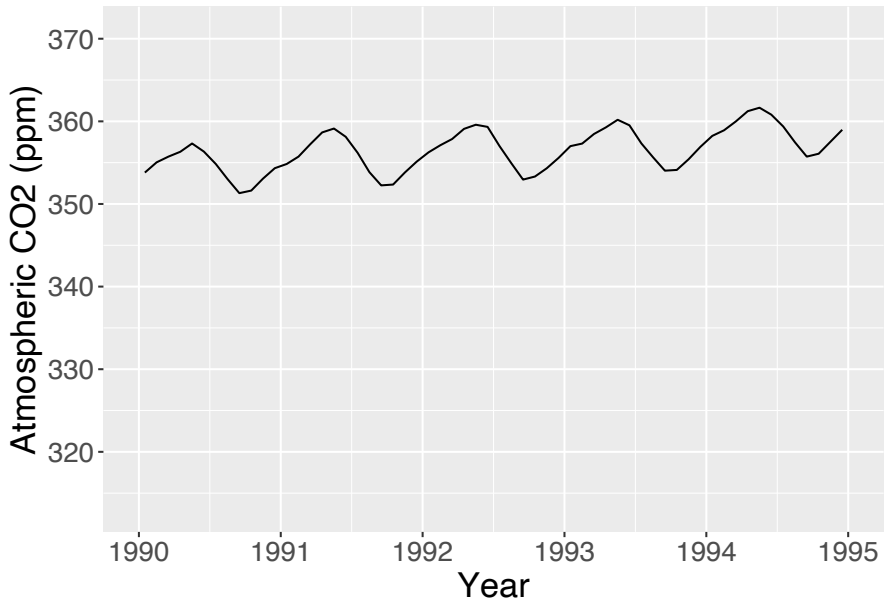


FIGURE 4.4: Line plot of atmospheric concentration of CO2 from 1990 to 1995 only

Interesting! It seems that each year, the atmospheric CO2 increases until it reaches its peak somewhere around April, decreases until around late September, and finally increases again until the end of the year. In Hawaii, there are two seasons: summer from May through October, and winter from November through April. Therefore, the oscillating pattern in CO2 matches up fairly closely with the two seasons.

4.5.2 The island landmass data set

The `islands.csv` data set contains a list of Earth's landmasses as well as their area (in thousands of square miles).

Question: Are the continents (North / South America, Africa, Europe, Asia,

Australia, Antarctica) Earth's seven largest landmasses? If so, what are the next few largest landmasses after those?

```
# islands data
islands_df <- read_csv("data/islands.csv")
islands_df
```

```
## # A tibble: 48 x 2
##    landmass      size
##    <chr>        <dbl>
##  1 Africa       11506
##  2 Antarctica    5500
##  3 Asia         16988
##  4 Australia     2968
##  5 Axel Heiberg    16
##  6 Baffin         184
##  7 Banks           23
##  8 Borneo         280
##  9 Britain         84
## 10 Celebes         73
## # ... with 38 more rows
```

Here, we have a list of Earth's landmasses, and are trying to compare their sizes. The right type of visualization to answer this question is a bar plot, specified by the `geom_bar` function in `ggplot2`. However, by default, `geom_bar` sets the heights of bars to the number of times a value appears in a data frame (its *count*); here, we want to plot exactly the values in the data frame, i.e., the landmass sizes. So we have to pass the `stat = "identity"` argument to `geom_bar`:

```
islands_bar <- ggplot(islands_df, aes(x = landmass, y = size)) +
  geom_bar(stat = "identity")
islands_bar
```

Alright, not bad! This plot is definitely the right kind of visualization, as we can clearly see and compare sizes of landmasses. The major issues are that the smaller landmasses' sizes are hard to distinguish, and the names of the landmasses are obscuring each other as they have been squished into too little space. But remember that the question we asked was only about the largest landmasses; let's make the plot a little bit clearer by keeping only the largest 12 landmasses. We do this using the `top_n` function. Then to help us make sure the labels have enough space, we'll use horizontal bars instead of vertical ones. We do this using the `coord_flip` function, which swaps the `x` and `y` coordinate axes:

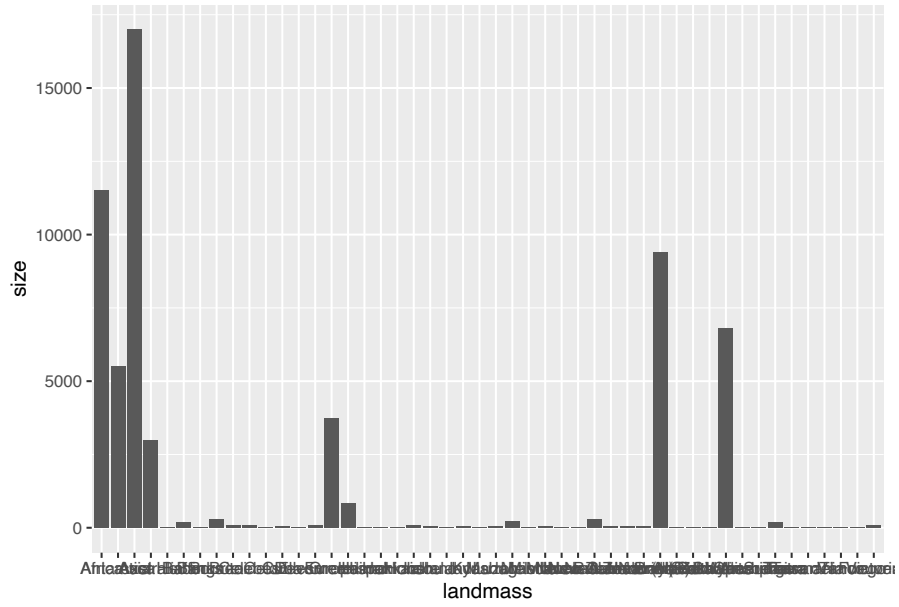


FIGURE 4.5: Bar plot of all Earth's landmasses' size with squished labels

```
islands_top12 <- top_n(islands_df, 12, size)
islands_bar <- ggplot(islands_top12, aes(x = landmass, y = size)) +
  geom_bar(stat = "identity") +
  coord_flip()
islands_bar
```

This plot is definitely clearer now, and allows us to answer our question ("are the top 7 largest landmasses continents?") in the affirmative. But the question could be made clearer from the plot by organizing the bars not by alphabetical order but by size, and to colour them based on whether they are a continent. To do this, we use `mutate` to add a column to the data regarding whether or not the landmass is a continent:

```
islands_top12 <- top_n(islands_df, 12, size)
continents <- c("Africa", "Antarctica", "Asia", "Australia", "Europe", "North America", "South Ame
islands_ct <- mutate(islands_top12, is_continent = ifelse(landmass %in% continents, "Continent", "
islands_ct
```

```
## # A tibble: 12 x 3
##    landmass       size is_continent
##    <chr>         <dbl> <chr>
```

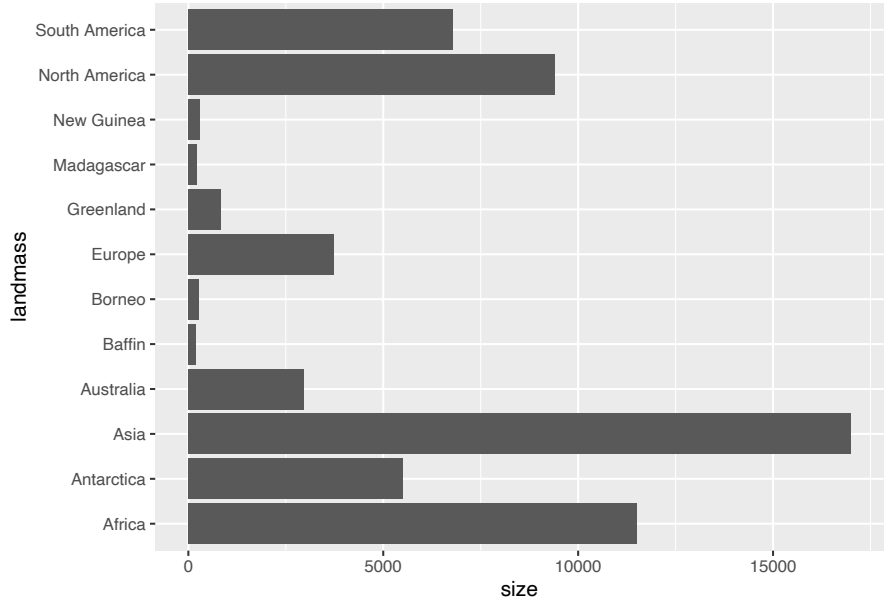


FIGURE 4.6: Bar plot of size for Earth's largest 12 landmasses

```
##  1 Africa        11506 Continent
##  2 Antarctica     5500 Continent
##  3 Asia          16988 Continent
##  4 Australia      2968 Continent
##  5 Baffin          184 Other
##  6 Borneo          280 Other
##  7 Europe         3745 Continent
##  8 Greenland       840 Other
##  9 Madagascar      227 Other
## 10 New Guinea      306 Other
## 11 North America  9390 Continent
## 12 South America  6795 Continent
```

In order to colour the bars, we add the `fill` argument to the aesthetic mapping. Then we use the `reorder` function in the aesthetic mapping to organize the landmasses by their `size` variable. Finally, we use the `labs` and `theme` functions to add labels, change the font size, and position the legend:

```
islands_bar <- ggplot(islands_ct, aes(x = reorder(landmass, size), y = size, fill = is_continent))
  geom_bar(stat = "identity") +
  labs(x = "Landmass", y = "Size (1000 square mi)", fill = "Type") +
  coord_flip() +
```

```
  theme(text = element_text(size = 18), legend.position = c(0.75, 0.45))
islands_bar
```

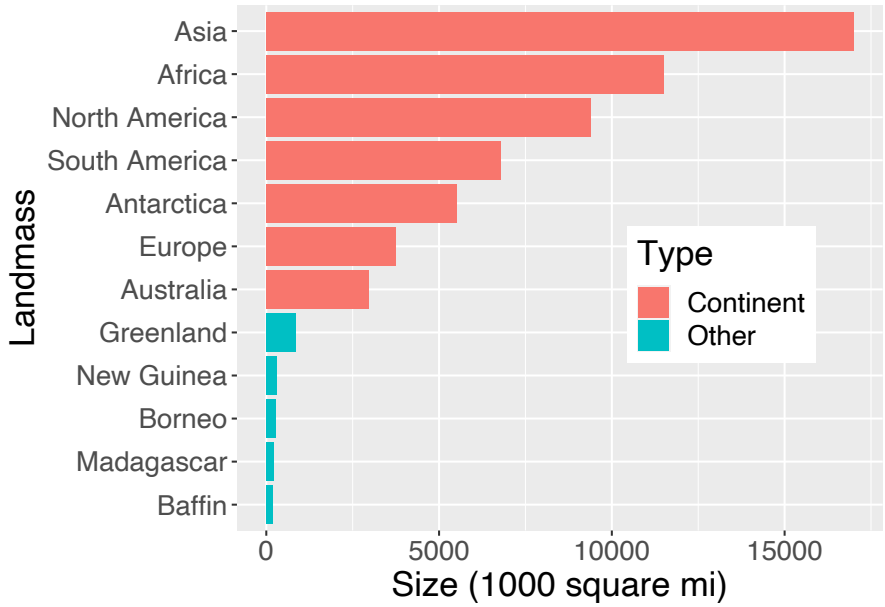


FIGURE 4.7: Bar plot of size for Earth's largest 12 landmasses coloured by whether its a continent with clearer axes and labels

This is now a very effective visualization for answering our original questions. Landmasses are organized by their size, and continents are coloured differently than other landmasses, making it quite clear that continents are the largest seven landmasses.

4.5.3 The Old Faithful eruption/waiting time data set

The `faithful` data set contains measurements of the waiting time between eruptions and the subsequent eruption duration (in minutes). The `faithful` data set is available in base R under the name `faithful` so it does not need to be loaded.

Question: Is there a relationship between the waiting time before an eruption to the duration of the eruption?

```
# old faithful eruption time / wait time data
faithful
```

```
## # A tibble: 272 x 2
##    eruptions waiting
##        <dbl>   <dbl>
##  1      3.6       79
##  2      1.8       54
##  3      3.33      74
##  4      2.28      62
##  5      4.53      85
##  6      2.88      55
##  7      4.7       88
##  8      3.6       85
##  9      1.95      51
## 10      4.35      85
## # ... with 262 more rows
```

Here again, we investigate the relationship between two quantitative variables (waiting time and eruption time). But if you look at the output of the data frame, you'll notice that neither of the columns are ordered. So, in this case, let's start again with a scatter plot:

```
faithful_scatter <- ggplot(faithful, aes(x = waiting, y = eruptions)) +
  geom_point()
faithful_scatter
```

We can see that the data tend to fall into two groups: one with short waiting and eruption times, and one with long waiting and eruption times. Note that in this case, there is no overplotting: the points are generally nicely visually separated, and the pattern they form is clear. In order to refine the visualization, we need only to add axis labels and make the font more readable:

```
faithful_scatter <- ggplot(faithful, aes(x = waiting, y = eruptions)) +
  geom_point() +
  labs(x = "Waiting Time (mins)", y = "Eruption Duration (mins)") +
  theme(text = element_text(size = 18))
faithful_scatter
```

4.5.4 The Michelson speed of light data set

The `morley` data set contains measurements of the speed of light (in kilometres per second with 299,000 subtracted) from the year 1879 for five experiments, each with 20 consecutive runs. This data set is available in base R under the name `morley` so it does not need to be loaded.

Question: Given what we know now about the speed of light (299,792.458 kilometres per second), how accurate were each of the experiments?

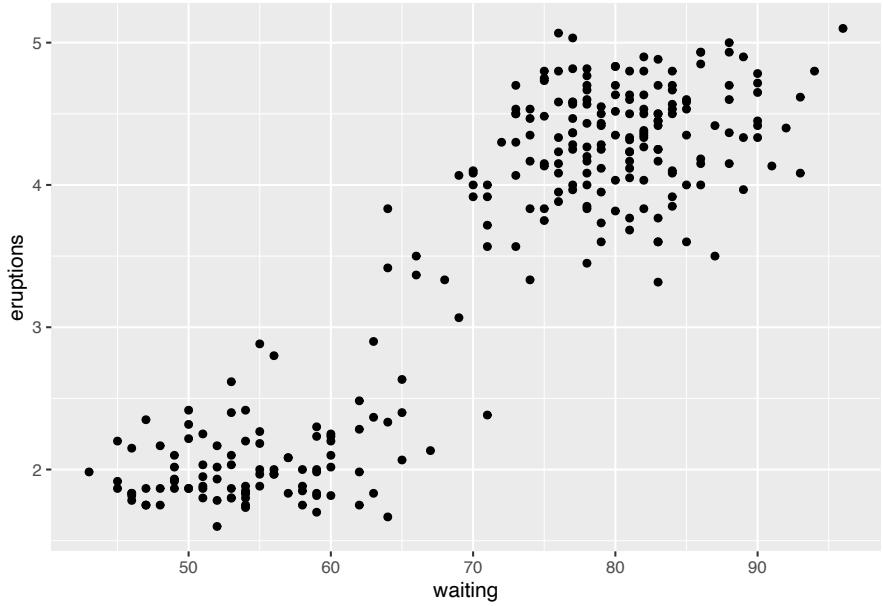


FIGURE 4.8: Scatter plot of waiting time and eruption time

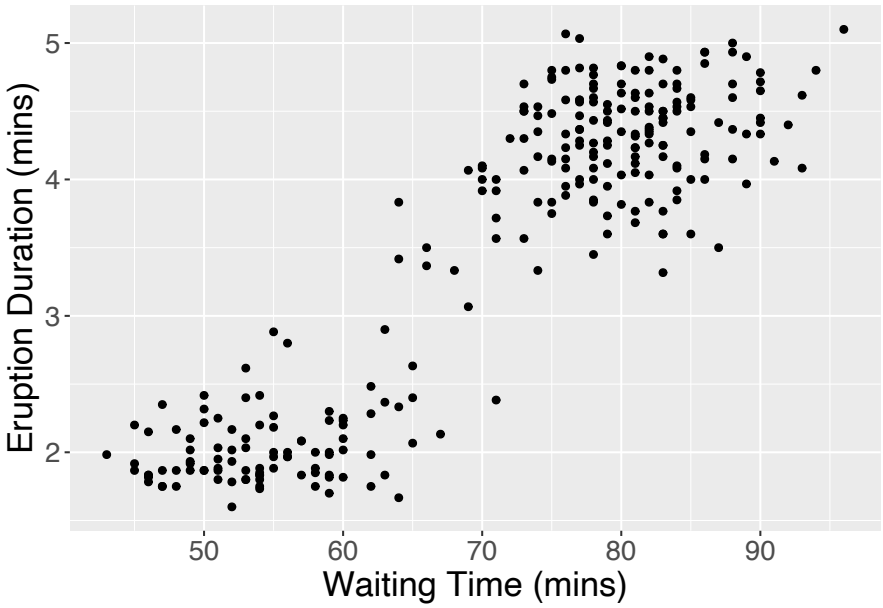


FIGURE 4.9: Scatter plot of waiting time and eruption time with clearer axes and labels

```
# michelson morley experimental data
morley
```

```
## # A tibble: 100 x 3
##     Expt   Run Speed
##    <int> <int> <int>
##  1     1     1   850
##  2     1     2   740
##  3     1     3   900
##  4     1     4  1070
##  5     1     5   930
##  6     1     6   850
##  7     1     7   950
##  8     1     8   980
##  9     1     9   980
## 10     1    10   880
## # ... with 90 more rows
```

In this experimental data, Michelson was trying to measure just a single quantitative number (the speed of light). The data set contains many measurements of this single quantity. To tell how accurate the experiments were, we need to visualize the distribution of the measurements (i.e., all their possible values and how often each occurs). We can do this using a *histogram*. A histogram helps us visualize how a particular variable is distributed in a data set by separating the data into bins, and then using vertical bars to show how many data points fell in each bin. To create a histogram in `ggplot2` we will use the `geom_histogram` geometric object, setting the `x` axis to the `Speed` measurement variable; and as we did before, let's use the default arguments just to see how things look:

```
morley_hist <- ggplot(morley, aes(x = Speed)) +
  geom_histogram()
morley_hist
```

This is a great start. However, we cannot tell how accurate the measurements are using this visualization unless we can see what the true value is. In order to visualize the true speed of light, we will add a vertical line with the `geom_vline` function, setting the `xintercept` argument to the true value. There is a similar function, `geom_hline`, that is used for plotting horizontal lines. Note that *vertical lines* are used to denote quantities on the *horizontal axis*, while *horizontal lines* are used to denote quantities on the *vertical axis*.

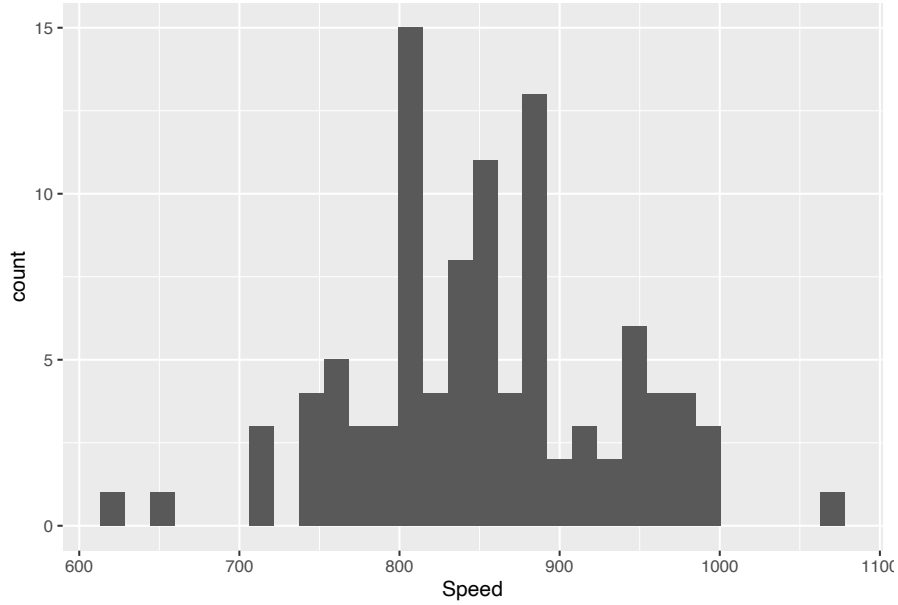


FIGURE 4.10: Histogram of Michelson's speed of light data

```
morley_hist <- ggplot(morley, aes(x = Speed)) +
  geom_histogram() +
  geom_vline(xintercept = 792.458, linetype = "dashed", size = 1.0)
morley_hist
```

We also still cannot tell which experiments (denoted in the `Expt` column) led to which measurements; perhaps some experiments were more accurate than others. To fully answer our question, we need to separate the measurements from each other visually. We can try to do this using a *coloured* histogram, where counts from different experiments are stacked on top of each other in different colours. We create a histogram coloured by the `Expt` variable by adding it to the `fill` aesthetic mapping. We make sure the different colours can be seen (despite them all sitting on top of each other) by setting the `alpha` argument in `geom_histogram` to `0.5` to make the bars slightly translucent:

```
morley_hist <- ggplot(morley, aes(x = Speed, fill = factor(Expt))) +
  geom_histogram(position = "identity", alpha = 0.5) +
  geom_vline(xintercept = 792.458, linetype = "dashed", size = 1.0)
morley_hist
```

Unfortunately, the attempt to separate out the experiment number visually

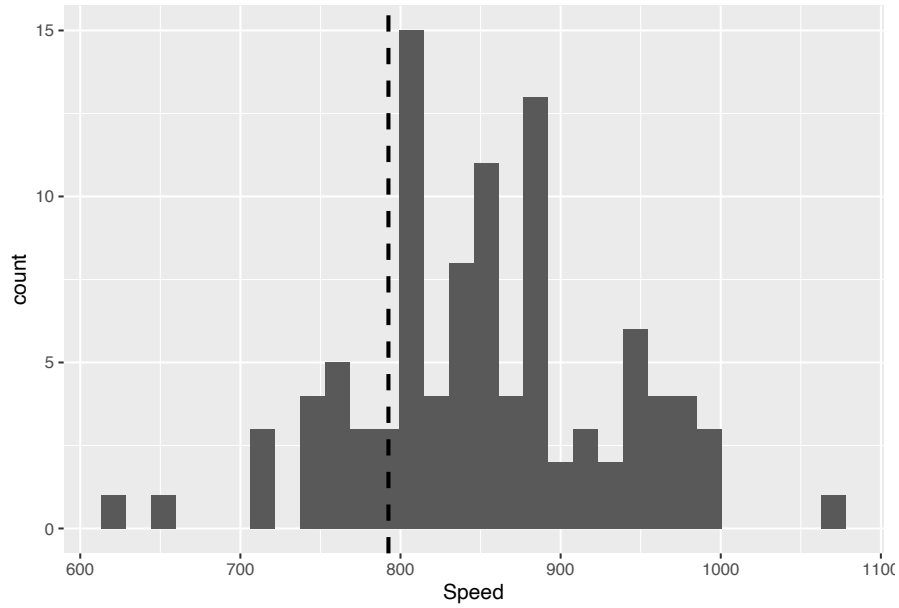


FIGURE 4.11: Histogram of Michelson's speed of light data with vertical line indicating true speed of light

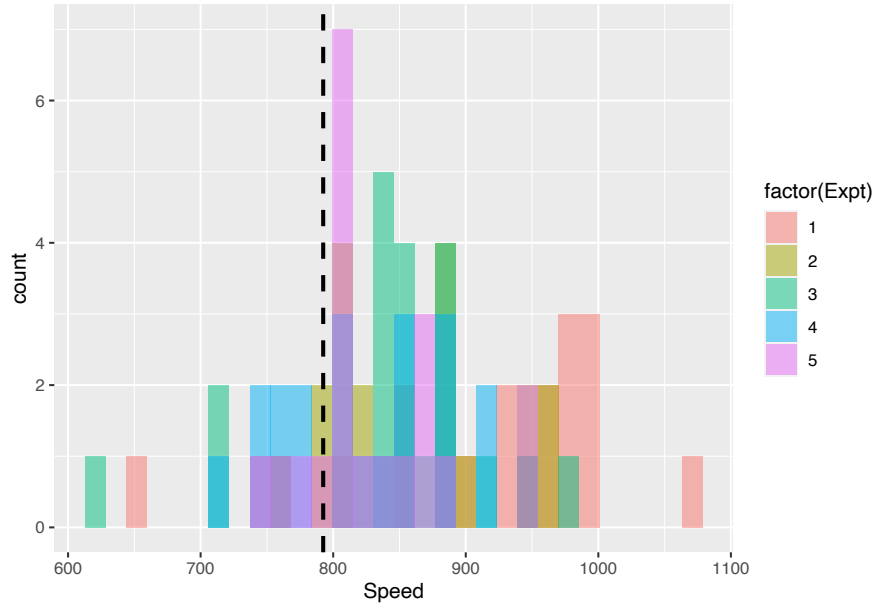


FIGURE 4.12: Histogram of Michelson's speed of light data coloured by experiment

has created a bit of a mess. All of the colours are blending together, and although it is possible to derive *some* insight from this (e.g., experiments 1 and 3 had some of the most incorrect measurements), it isn't the clearest way to convey our message and answer the question. Let's try a different strategy of creating multiple separate histograms on top of one another.

In order to create a plot in `ggplot2` that has multiple subplots arranged in a grid, we use the `facet_grid` function. The argument to `facet_grid` specifies the variable(s) used to split the plot into subplots. It has the syntax `vertical_variable ~ horizontal_variable`, where `veritcal_variable` is used to split the plot vertically, `horizontal_variable` is used to split horizontally, and `.` is used if there should be no split along that axis. In our case, we only want to split vertically along the `Expt` variable, so we use `Expt ~ .` as the argument to `facet_grid`.

```
morley_hist <- ggplot(morley, aes(x = Speed, fill = factor(Expt))) +
  geom_histogram(position = "identity") +
  facet_grid(Expt ~ .) +
  geom_vline(xintercept = 792.458, linetype = "dashed", size = 1.0)
morley_hist
```

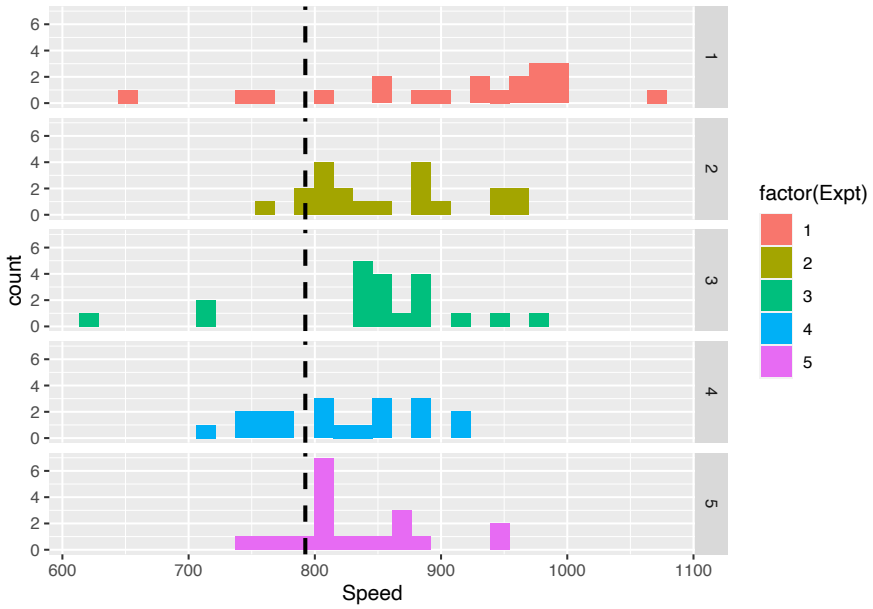


FIGURE 4.13: Histogram of Michelson's speed of light data split vertically by experiment

The visualization now makes it quite clear how accurate the different exper-

iments were with respect to one another. There are two finishing touches to make this visualization even clearer. First and foremost, we need to add informative axis labels using the `labs` function, and increase the font size to make it readable using the `theme` function. Second, and perhaps more subtly, even though it is easy to compare the experiments on this plot to one another, it is hard to get a sense for just how accurate all the experiments were overall. For example, how accurate is the value 800 on the plot, relative to the true speed of light? To answer this question, we'll use the `mutate` function to transform our data into a relative measure of accuracy rather than absolute measurements:

```
morley_rel <- mutate(morley, relative_accuracy = 100 * ((299000 + Speed) - 299792.458) / (299792.4
morley_hist <- ggplot(morley_rel, aes(x = relative_accuracy, fill = factor(Expt))) +
  geom_histogram(position = "identity") +
  facet_grid(Expt ~ .) +
  geom_vline(xintercept = 0, linetype = "dashed", size = 1.0) +
  labs(x = "Relative Accuracy (%)", y = "# Measurements", fill = "Experiment ID") +
  theme(text = element_text(size = 18))
morley_hist
```

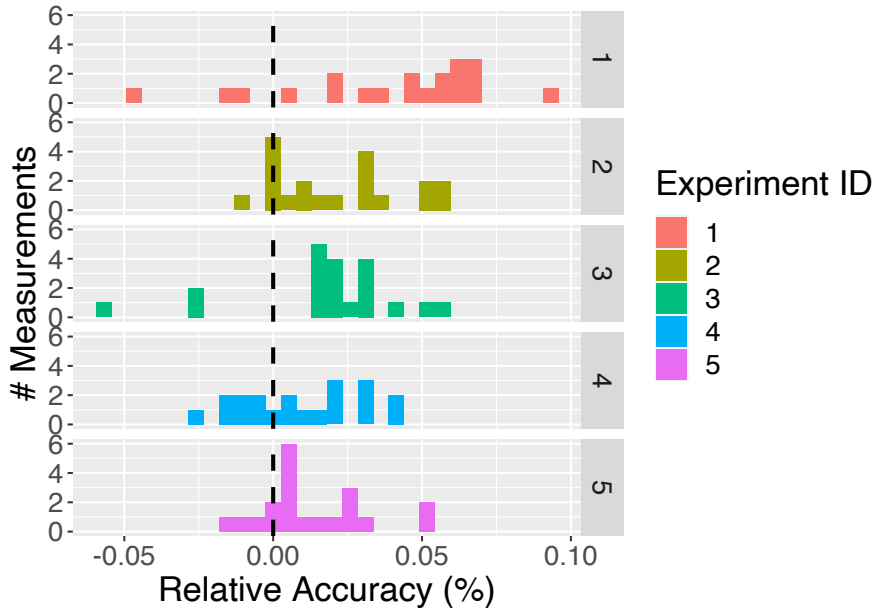


FIGURE 4.14: Histogram of relative accuracy split vertically by experiment with clearer axes and labels

Wow, impressive! These measurements of the speed of light from 1879 had

errors around *0.05%* of the true speed. This shows you that even though experiments 2 and 5 were perhaps the most accurate, all of the experiments did quite an admirable job given the technology available at the time.

4.6 Explaining the visualization

4.6.0.1 *Tell a story*

Typically, your visualization will not be shown entirely on its own, but rather it will be part of a larger presentation. Further, visualizations can provide supporting information for any aspect of a presentation, from opening to conclusion. For example, you could use an exploratory visualization in the opening of the presentation to motivate your choice of a more detailed data analysis / model, a visualization of the results of your analysis to show what your analysis has uncovered, or even one at the end of a presentation to help suggest directions for future work.

Regardless of where it appears, a good way to discuss your visualization is as a story:

1) Establish the setting and scope, and motivate why you did what you did.
2) Pose the question that your visualization answers. Justify why the question is important to answer.
3) Answer the question using your visualization. Make sure you describe *all* aspects of the visualization (including describing the axes). But you can emphasize different aspects based on what is important to answer your question:
    - **trends (lines):** Does a line describe the trend well? If so, the trend is *linear*, and if not, the trend is *nonlinear*. Is the trend increasing, decreasing, or neither? Is there a periodic oscillation (wiggle) in the trend? Is the trend noisy (does the line "jump around" a lot) or smooth?
    - **distributions (scatters, histograms):** How spread out are the data? Where are they centered, roughly? Are there any obvious "clusters" or "subgroups", which would be visible as multiple bumps in the histogram?
    - **distributions of two variables (scatters):** is there a clear / strong relationship between the variables (points fall in a distinct pattern), a weak one (points fall in a pattern but there is some noise), or no discernible relationship (the data are too noisy to make any conclusion)?

- **amounts (bars):** How large are the bars relative to one another? Are there patterns in different groups of bars?

4) Summarize your findings, and use them to motivate whatever you will discuss next.

Below are two examples of how one might take these four steps in describing the example visualizations that appeared earlier in this chapter. Each of the steps is denoted by its numeral in parentheses, e.g. (3).

Mauna Loa Atmospheric CO2 Measurements: (1) Many current forms of energy generation and conversion—from automotive engines to natural gas power plants—rely on burning fossil fuels and produce greenhouse gases, typically primarily carbon dioxide (CO2), as a byproduct. Too much of these gases in the Earth's atmosphere will cause it to trap more heat from the sun, leading to global warming. (2) In order to assess how quickly the atmospheric concentration of CO2 is increasing over time, we (3) used a data set from the Mauna Loa observatory from Hawaii, consisting of CO2 measurements from 1959 to the present. We plotted the measured concentration of CO2 (on the vertical axis) over time (on the horizontal axis). From this plot, you can see a clear, increasing, and generally linear trend over time. There is also a periodic oscillation that occurs once per year and aligns with Hawaii's seasons, with an amplitude that is small relative to the growth in the overall trend. This shows that atmospheric CO2 is clearly increasing over time, and (4) it is perhaps worth investigating more into the causes.

Michelson Light Speed Experiments: (1) Our modern understanding of the physics of light has advanced significantly from the late 1800s when Michelson and Morley's experiments first demonstrated that it had a finite speed. We now know based on modern experiments that it moves at roughly 299792.458 kilometres per second. (2) But how accurately were we first able to measure this fundamental physical constant, and did certain experiments produce more accurate results than others? (3) To better understand this we plotted data from 5 experiments by Michelson in 1879, each with 20 trials, as histograms stacked on top of one another. The horizontal axis shows the accuracy of the measurements relative to the true speed of light as we know it today, expressed as a percentage. From this visualization, you can see that most results had relative errors of at most 0.05%. You can also see that experiments 1 and 3 had measurements that were the farthest from the true value, and experiment 5 tended to provide the most consistently accurate result. (4) It would be worth further investigating the differences between these experiments to see why they produced different results.

4.7 Saving the visualization

4.7.0.1 *Choose the right output format for your needs*

Just as there are many ways to store data sets, there are many ways to store visualizations and images. Which one you choose can depend on several factors, such as file size/type limitations (e.g., if you are submitting your visualization as part of a conference paper or to a poster printing shop) and where it will be displayed (e.g., online, in a paper, on a poster, on a billboard, in talk slides). Generally speaking, images come in two flavours: *bitmap* (or *raster*) formats and *vector* (or *scalable graphics*) formats.

Bitmap / Raster images are represented as a 2-D grid of square pixels, each with its own colour. Raster images are often *compressed* before storing so they take up less space. A compressed format is *lossy* if the image cannot be perfectly recreated when loading and displaying, with the hope that the change is not noticeable. *Lossless* formats, on the other hand, allow a perfect display of the original image.

- *Common file types:*
  - JPEG[13] (`.jpg`, `.jpeg`): lossy, usually used for photographs
  - PNG[14] (`.png`): lossless, usually used for plots / line drawings
  - BMP[15] (`.bmp`): lossless, raw image data, no compression (rarely used)
  - TIFF[16] (`.tif`, `.tiff`): typically lossless, no compression, used mostly in graphic arts, publishing
- *Open-source software:* GIMP[17]

Vector / Scalable Graphics images are represented as a collection of mathematical objects (lines, surfaces, shapes, curves). When the computer displays the image, it redraws all of the elements using their mathematical formulas.

- *Common file types:*
  - SVG[18] (`.svg`): general-purpose use
  - EPS[19] (`.eps`), general-purpose use (rarely used)
- *Open-source software:* Inkscape[20]

Raster and vector images have opposing advantages and disadvantages. A raster image of a fixed width / height takes the same amount of space and time to load regardless of what the image shows (caveat: the compression

[13] https://en.wikipedia.org/wiki/JPEG
[14] https://en.wikipedia.org/wiki/Portable_Network_Graphics
[15] https://en.wikipedia.org/wiki/BMP_file_format
[16] https://en.wikipedia.org/wiki/TIFF
[17] https://www.gimp.org/
[18] https://en.wikipedia.org/wiki/Scalable_Vector_Graphics
[19] https://en.wikipedia.org/wiki/Encapsulated_PostScript
[20] https://inkscape.org/

algorithms may shrink the image more or run faster for certain images). A vector image takes space and time to load corresponding to how complex the image is, since the computer has to draw all the elements each time it is displayed. For example, if you have a scatter plot with 1 million points stored as an SVG file, it may take your computer some time to open the image. On the other hand, you can zoom into / scale up vector graphics as much as you like without the image looking bad, while raster images eventually start to look "pixellated."

PDF files: The portable document format PDF[21] (`.pdf`) is commonly used to store *both* raster and vector graphics formats. If you try to open a PDF and it's taking a long time to load, it may be because there is a complicated vector graphics image that your computer is rendering.

Let's investigate how different image file formats behave with a scatter plot of the Old Faithful data set[22]:

```
library(svglite) # we need this to save SVG files
faithful_plot <- ggplot(data = faithful, aes(x = waiting, y = eruptions)) +
  geom_point()

faithful_plot
```

```
ggsave("faithful_plot.png", faithful_plot)
ggsave("faithful_plot.jpg", faithful_plot)
ggsave("faithful_plot.bmp", faithful_plot)
ggsave("faithful_plot.tiff", faithful_plot)
ggsave("faithful_plot.svg", faithful_plot)

print(paste("PNG filesize: ", file.info("faithful_plot.png")["size"] / 1000000, "MB"))
## [1] "PNG filesize:  0.178079 MB"
print(paste("JPG filesize: ", file.info("faithful_plot.jpg")["size"] / 1000000, "MB"))
## [1] "JPG filesize:  0.19125 MB"
print(paste("BMP filesize: ", file.info("faithful_plot.bmp")["size"] / 1000000, "MB"))
```

[21] https://en.wikipedia.org/wiki/PDF
[22] https://www.stat.cmu.edu/~larry/all-of-statistics/=data/faithful.dat

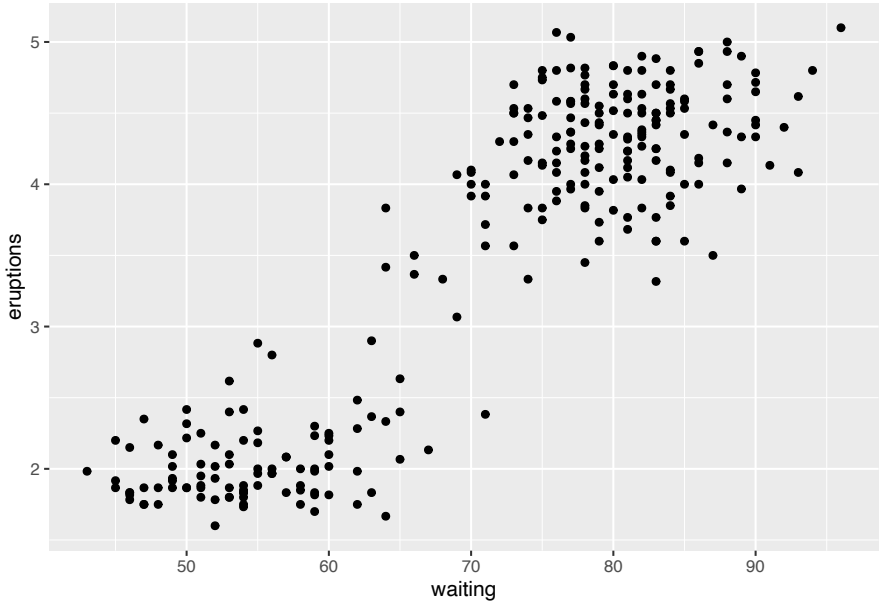


FIGURE 4.15: Scatter plot of waiting time and eruption time

```
## [1] "BMP filesize:  10.522254 MB"
print(paste("TIFF filesize: ", file.info("faithful_plot.tiff")["size"] / 1000000, "MB"))
## [1] "TIFF filesize:  10.52511 MB"
print(paste("SVG filesize: ", file.info("faithful_plot.svg")["size"] / 1000000, "MB"))
## [1] "SVG filesize:  0.046062 MB"
```

Wow, that's quite a difference! Notice that for such a simple plot with few graphical elements (points), the vector graphics format (SVG) is over 100 times smaller than the uncompressed raster images (BMP, TIFF). Also, note that the JPG format is twice as large as the PNG format since the JPG compression algorithm is designed for natural images (not plots). Below, we also show what the images look like when we zoom in to a rectangle with only 3 data points. You can see why vector graphics formats are so useful: because they're just based on mathematical formulas, vector graphics can be scaled up to arbitrary sizes. This makes them great for presentation media of all sizes, from papers to posters to billboards.

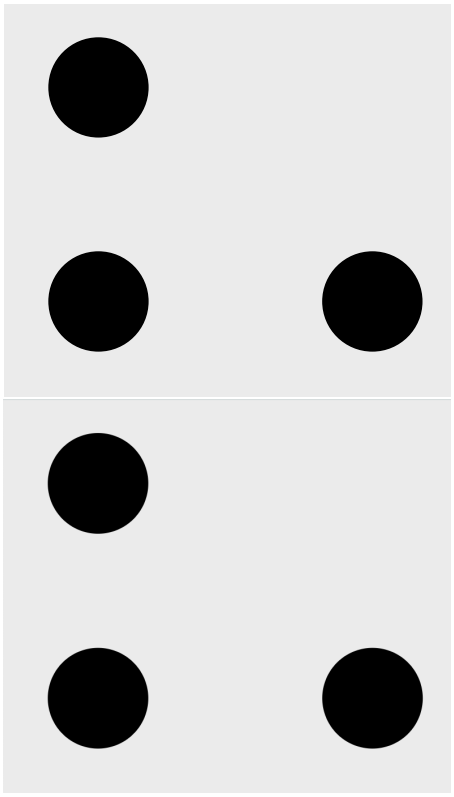


FIGURE 4.16: Zoomed in ‘faithful‘, raster (PNG, left) and vector (SVG, right) formats

5

Collaboration with version control

5.1 Overview

This chapter will introduce the concept of using version control systems to track changes to a project over its lifespan, to share and edit code in a collaborative team, and to distribute the finished project to its intended audience. This chapter also demonstrates how to implement these ideas effectively in practice using Git[1], GitHub[2], and JupyterHub.

5.2 Chapter learning objectives

By the end of the chapter, students will be able to:

- Describe what version control is and why data analysis projects can benefit from it
- Create a remote version control repository on GitHub
- Move changes to files from GitHub to JupyterHub, and from JupyterHub to GitHub
- Give collaborators access to the repository
- Resolve conflicting edits made by multiple collaborators
- Communicate with collaborators using issues
- Use best practices when collaborating on a project with others

5.3 What is version control, and why should I use it?

Data analysis projects often require iteration and revision to move from an initial idea to a finished product that is ready for the intended audience. Without

[1] https://git-scm.com
[2] https://github.com

deliberate and conscious effort towards tracking changes made to the analysis, projects tend to become messy, with mystery results files that cannot be reproduced, temporary files with snippets of ideas, mind-boggling filenames like `document_final_draft_v5_final.txt`, large blocks of commented code "saved for later," and more. Additionally, the iterative nature of data analysis projects makes it important to be able to examine earlier versions of code and writing. Finally, data analyses are typically completed by a team of people rather than a single person. This means that files need to be shared across multiple computers, and multiple people often end up editing the project simultaneously. In such a situation, determining who has the latest version of the project—and how to resolve conflicting edits—can be a real challenge.

Version control helps solve these challenges by tracking changes to the files in the analysis (code, writing, data, etc) over the lifespan of the project, including when the changes were made and who made them. This provides the means both to view earlier versions of the project and to revert changes. Version control also facilitates collaboration via tools to share edits with others and resolve conflicting edits. But even if you're working on a project alone, you should still use version control. It helps you keep track of what you've done, when you did it, and what you're planning to do next!

> *You mostly collaborate with yourself, and me-from-two-months-ago never responds to email.*
>
> –Mark T. Holder

In order to version control a project, you generally need two things: a *version control system* and a *repository hosting service*. The version control system is the software that is responsible for tracking changes, sharing changes you make with others, obtaining changes others have made, and resolving conflicting edits. The repository hosting service is responsible for storing a copy of the version controlled project online (a *repository*), where you and your collaborators can access it remotely, discuss issues and bugs, and distribute your final product. For both of these items, there is a wide variety of choices; some of the more popular ones are:

- **Version control systems:**
  - Git[3]
  - Mercurial[4]

[3] https://git-scm.com
[4] https://mercurial-scm.org

  - Subversion[5]
- **Repository hosting services:**
  - GitHub[6]
  - GitLab[7]
  - BitBucket[8]

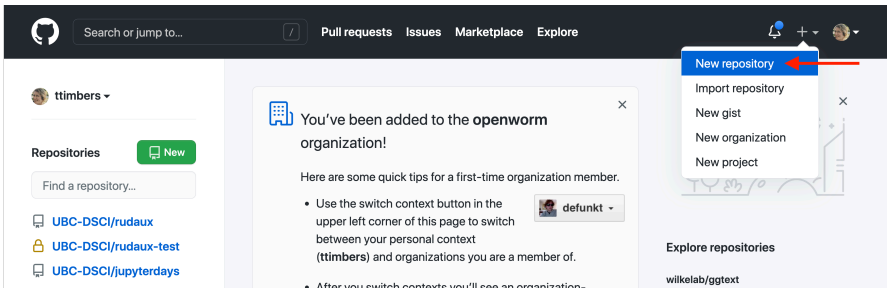
In this textbook we'll use Git for version control, and GitHub for repository hosting, because both are currently the most widely-used platforms.

Note: technically you don't *have to* use a repository hosting service. You can, for example, use Git to version control a project that is stored only in a folder on your computer. But using a repository hosting service provides a few big benefits, including managing collaborator access permissions, tools to discuss and track bugs, and the ability to have external collaborators contribute work, not to mention the safety of having your work backed up in the cloud. Since most repository hosting services now offer free accounts, there are not many situations in which you wouldn't want to use one for your project.

5.4 Creating a space for your project online

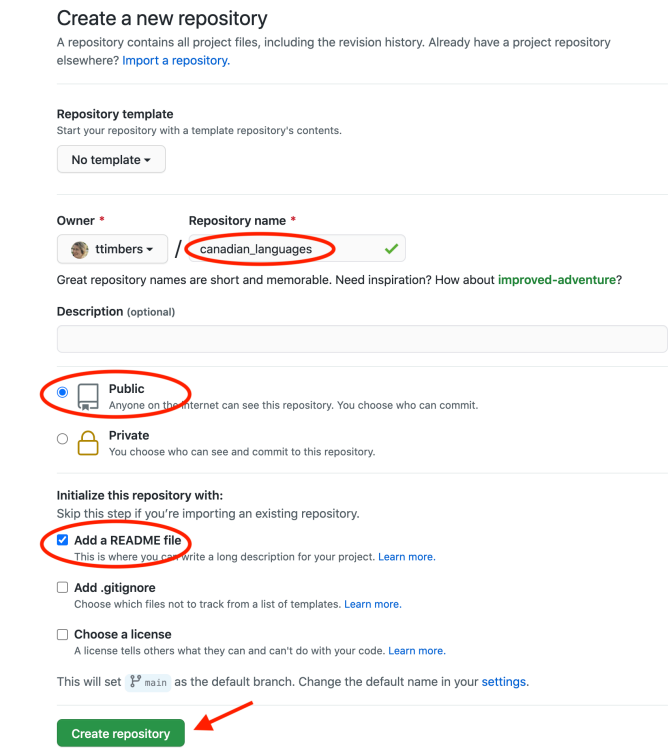
Before you can create repositories, you will need a GitHub account; you can sign up for a free account at `https://github.com/`. Once you have logged into your account, you can create a new repository to host your project by clicking on the "+" icon in the upper right hand corner, and then on "New Repository" as shown below:

[5] `https://subversion.apache.org`
[6] `https://github.com`
[7] `https://gitlab.com`
[8] `https://bitbucket.org`

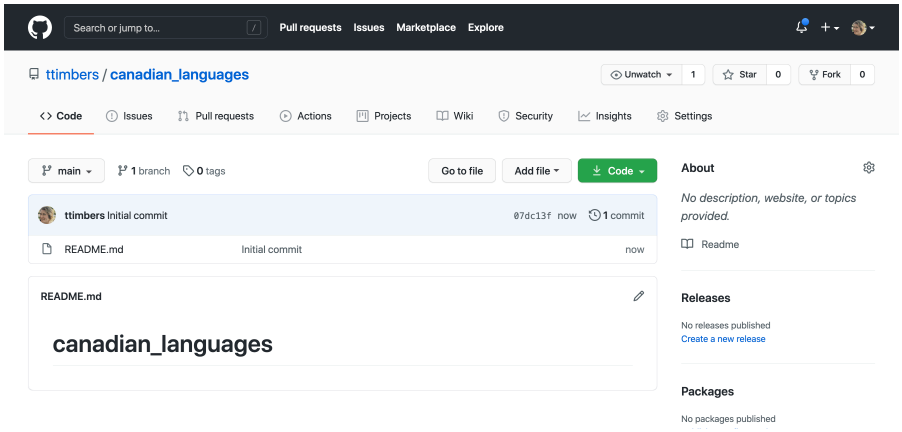


On the next page, do the following:

1. Enter the name for your project repository. In the example below, we use `canadian_languages`. Most repositories follow this naming convention, which involves lowercase letter words separated by either underscores or hyphens.
2. Choose an option for the privacy of your repository
    1. If you select "Public", your repository may be *viewed* by anyone, but only you and collaborators you designate will be able to *modify* it.
    2. If you select "Private", only you and your collaborators can *view* or *modify* it.
3. Select "Add a README file." This creates a template `README.md` file in your repository's root folder.
4. When you are happy with your repository name and configuration, click on the green "Create Repository" button.



Now you should have a repository that looks something like this:

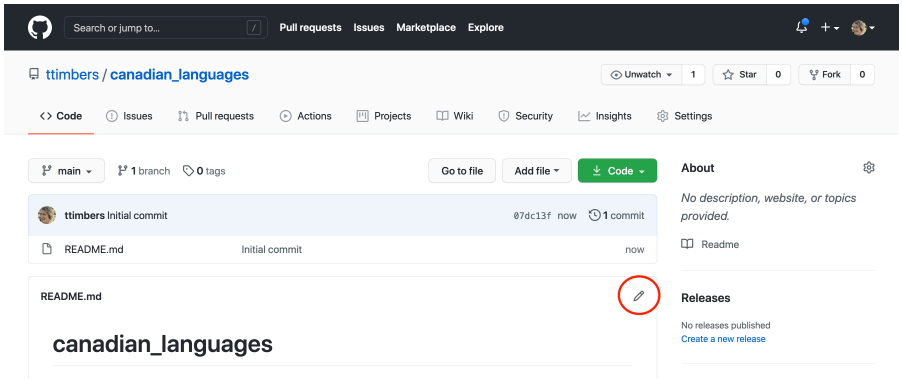


5.5 Creating and editing files on GitHub

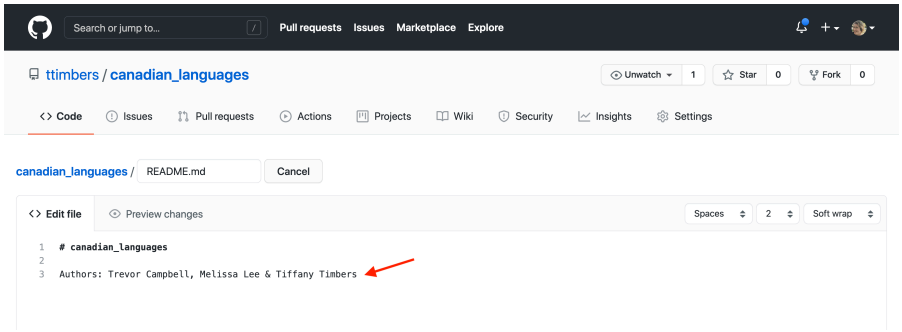
There are several ways to use the GitHub interface to add files to your repository and to edit them. Below we cover how to use the pen tool to edit existing files, and how to use the "Add file" drop down to create a new file or upload files from your computer. These techniques are useful for handling simple plaintext files, for example, the `README.md` file that is already present in the repository.

5.5.1 The pen tool

The pen tool can be used to edit existing plaintext files. Click on the pen tool:

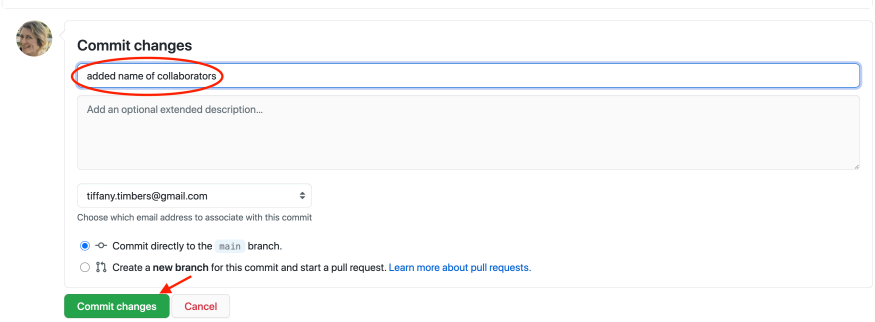


Use the text box to make your changes:



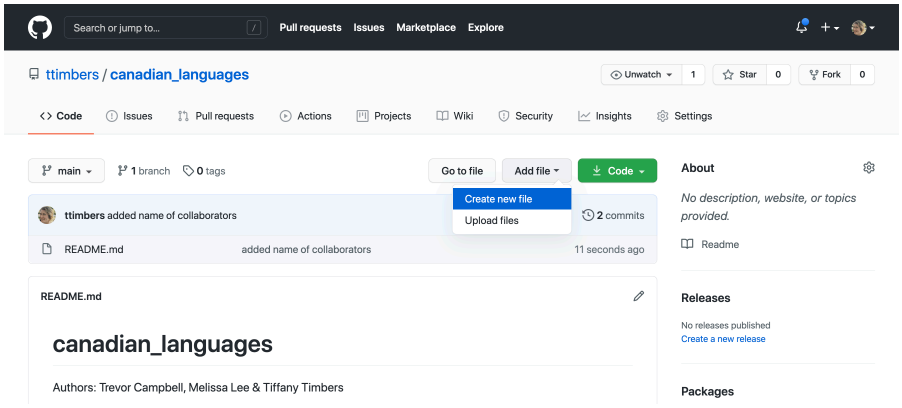
Finally, *commit* your changes. When you *commit a file* in a repository, the version control system takes a snapshot of what the file looks like. As you continue working on the project, over time you will possibly make many commits to a single file; this generates a useful version history for that file. On GitHub,

if you click the green "Commit changes" button, it will save the file and then make a commit. Do this now:



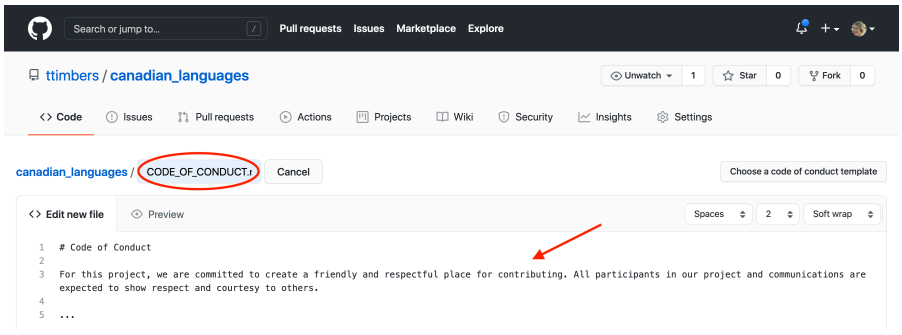
5.5.2 The "Add file" menu

The "Add file" menu can be used to create new plaintext files and upload files from your computer. To create a new plaintext file, click the "Add file" drop down menu and select the "Create new file" option:

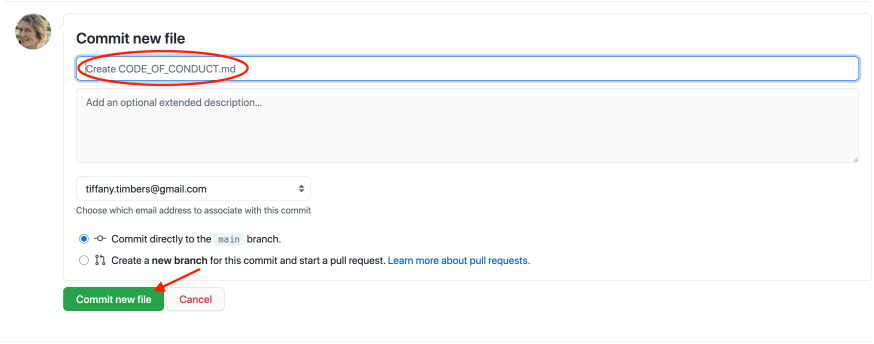


A page will open with a small text box for the file name to be entered, and a larger text box where the desired file content text can be entered. Note the two tabs, "Edit new file" and "Preview". Toggling between them lets you enter and edit text and view what the text will look like when rendered, respectively. Note that GitHub understands and renders `.md` files using a markdown syntax[9] very similar to Jupyter notebooks, so the "Preview" tab is especially helpful for checking markdown code correctness.

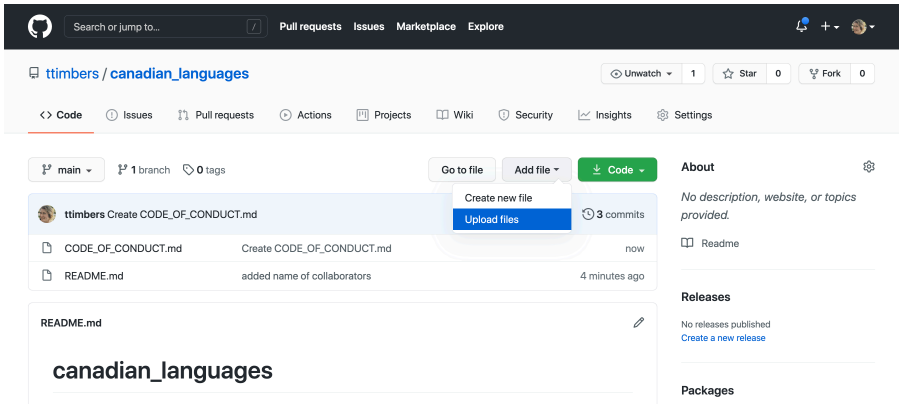
[9] https://guides.github.com/pdfs/markdown-cheatsheet-online.pdf



Save and commit your changes by click the green "Commit changes" button at the bottom of the page.

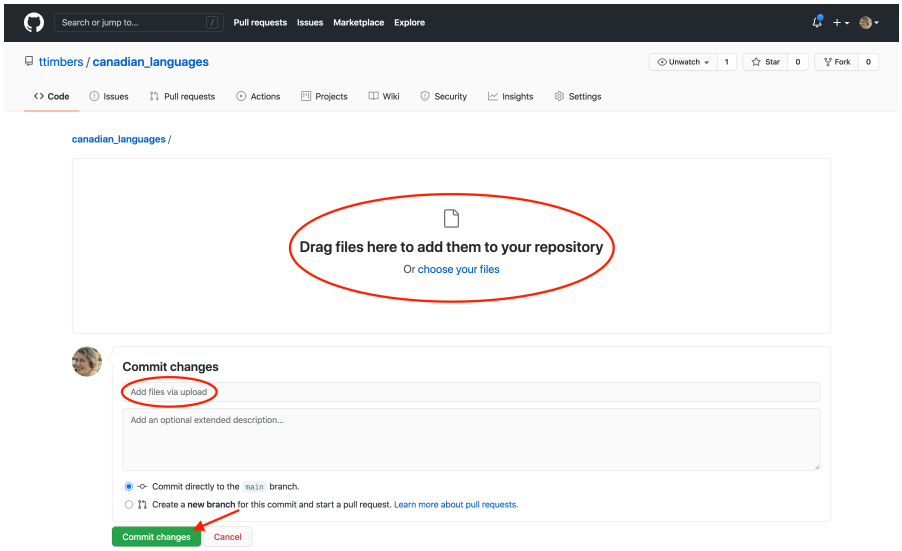


You can also upload files that you have created on your local machine by using the "Add file" drop down menu and selecting "Upload files":



To select the files from your local computer to upload, you can either drag and drop them into the grey box area shown below, or click the "choose your files" link to access a file browser dialog. Once the files you want to upload

have been selected, click the green "Commit changes" button at the bottom of the page.

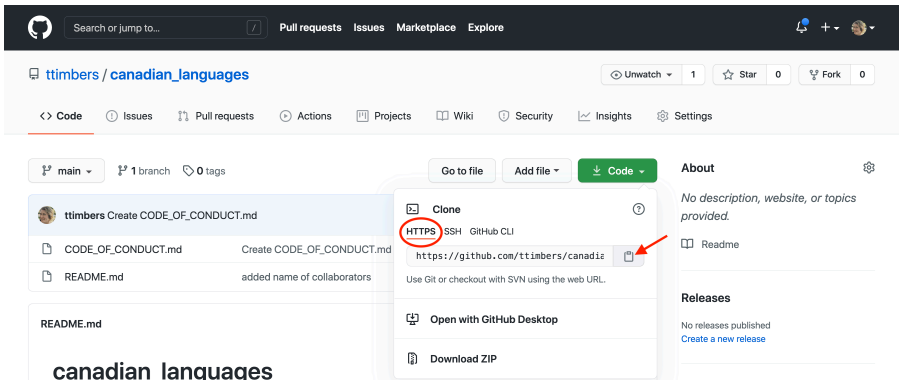


Note that Git and GitHub are designed to track changes in individual files. *Do not* upload your whole project in an archive file (e.g. `.zip`), because then Git can only keep track of changes to the entire `.zip` file—that wouldn't be very useful if you're trying to see the history of changes to a single code file in your project!

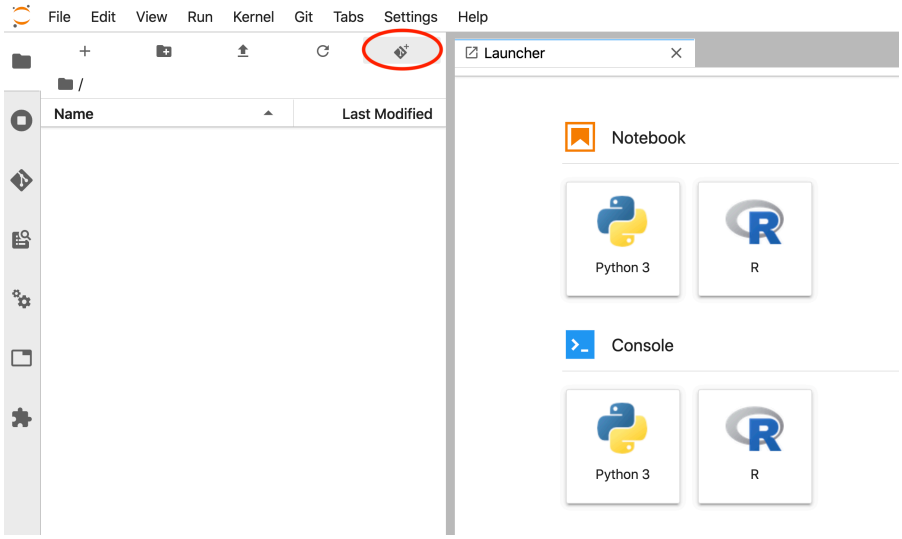
5.6 Cloning your repository on JupyterHub

Although there are several ways to create and edit files on GitHub, they are not quite powerful enough for efficiently creating and editing complex files, or files that need to be executed to assess whether they work (e.g., files containing code). Thus, it is useful to be able to connect the project repository that was created on GitHub to a coding environment. This can be done on your local computer, or using a JupyterHub; below we show how to do this using a JupyterHub.

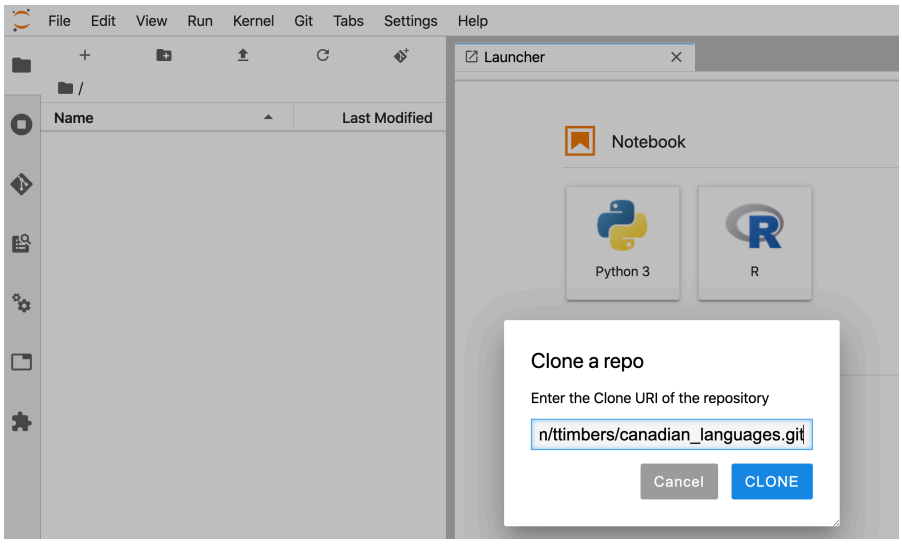
We need to *clone* our project's Git repository to our JupyterHub—i.e., make a copy that knows where it was obtained from so that it knows where send/receive new committed edits. In order to do this, first copy the url from the HTTPS tab of the Code drop down menu on GitHub:



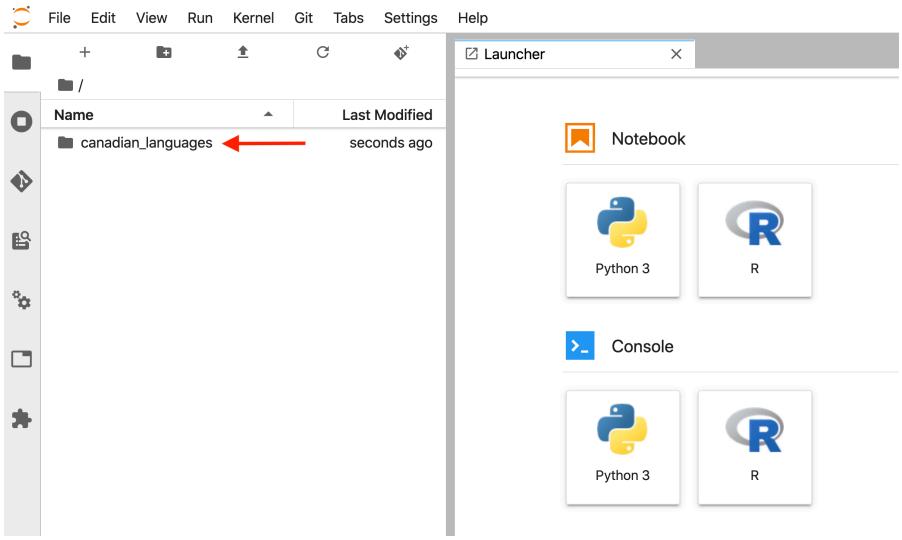
Then open JupyterHub, and click the Git+ icon on the file browser tab:



Paste the url of the GitHub project repository you created and click the blue "CLONE" button:



On the file browser tab, you will now see a folder for your project's repository (and inside it will be all the files that existed on GitHub):



5.7 Working in a cloned repository on JupyterHub

Now that you have cloned your repository on JupyterHub, you can get to work editing, creating, and deleting files. Once you reach a point that you

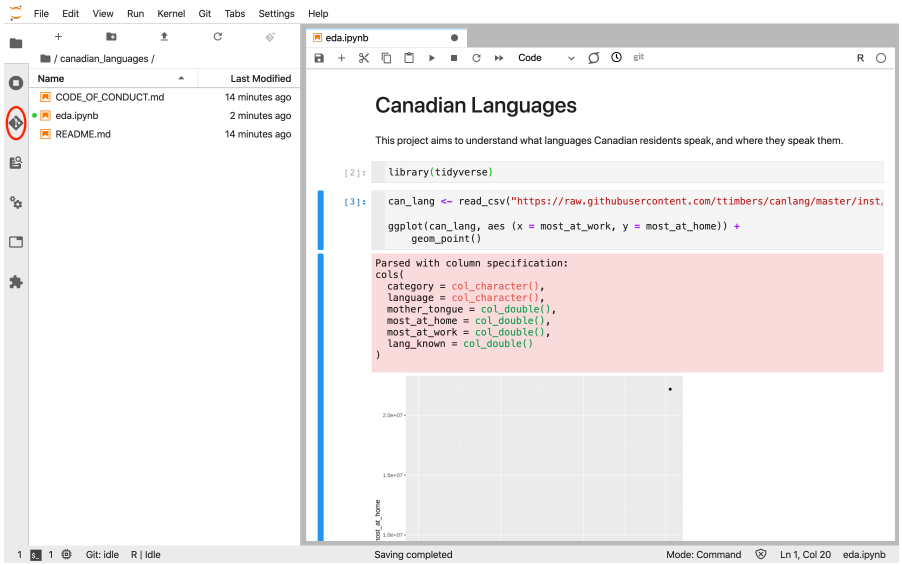
want Git to keep a record of the current version, you need to *commit* (i.e., snapshot) your changes. Then once you have made commits that you want to share with your collaborators, you need to *push* (i.e., send) those commits back to GitHub. Again, we can use the JupyterLab Git extension tool to do all of this. In particular, your workflow on JupyterHub should look like this:

1. You edit, create, and delete files in your cloned repository on JupyterHub.
2. Once you want a record of the current version, you specify which files to "add" to Git's *staging area*. You can think of files in the staging area as those modified files for which you want a snapshot.
3. You commit those flagged files to your repository, and include a helpful *commit message* to tell your collaborators about the changes you made. **Note:** here you are *only* committing to your *cloned repository* stored on JupyterHub. The repository on GitHub has not changed, and your collaborators cannot see your work yet.
4. Go back to step 1. and keep working!
5. When you want to store your commits (that only exist in your cloned repository right now) on the cloud where they can be shared with your collaborators, you *push* them back to the hosted repository on GitHub.

Below we walk through how to use the Jupyter Git extension tool to do each of the steps outlined above.

5.7.1 Specifying files to commit

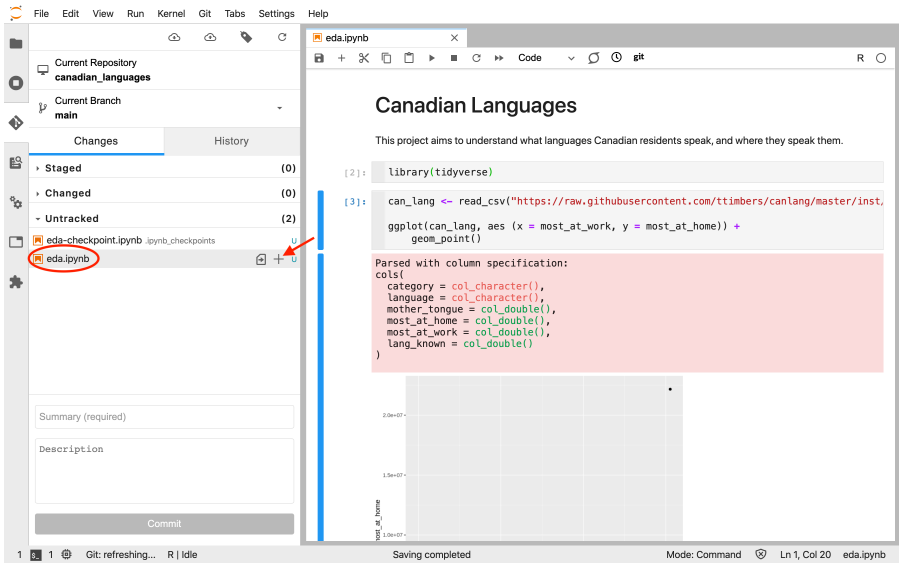
Below we created and saved a new file (named `eda.ipynb`) that we would like to send back to the project repository on GitHub. To "add" this modified file to the staging area (*i.e.*, flag that this is a file whose changes we would like to commit), we click the Jupyter Git extension icon on the far left-hand side of JupyterLab:



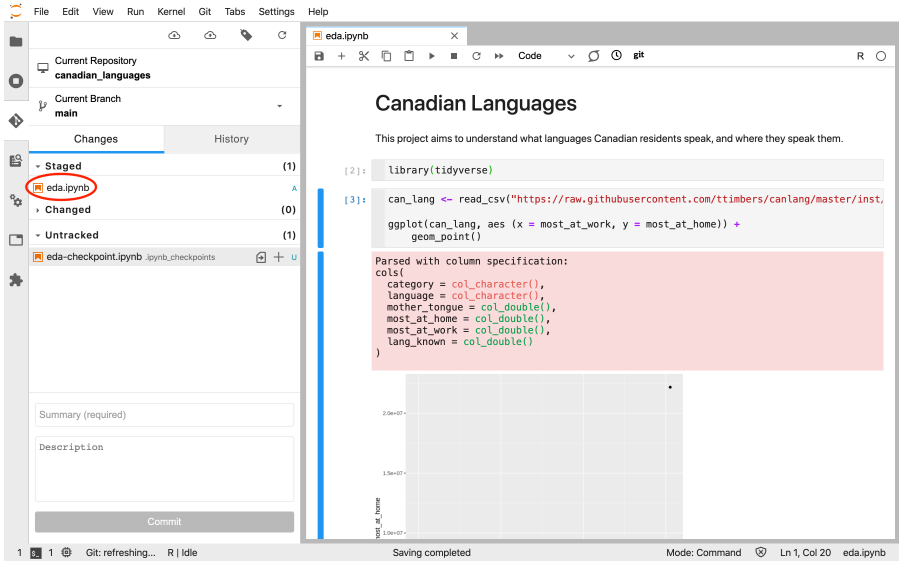
This opens the Jupyter Git graphical user interface pane, and then we click the plus sign beside the file that we want to "add".

Note: because this is the first change for this file that we want to add, it falls under the "Untracked" heading. However, next time we edit this file and want to add the changes we made, we will find it under the "Changed" heading.

Note: do not add the `eda-checkpoint.ipynb` file (sometimes called `.ipynb_checkpoints`). This file is automatically created by Jupyter when you work on `eda.ipynb`. You generally do not add auto-generated files to Git repositories, only add the files you directly create and edit.

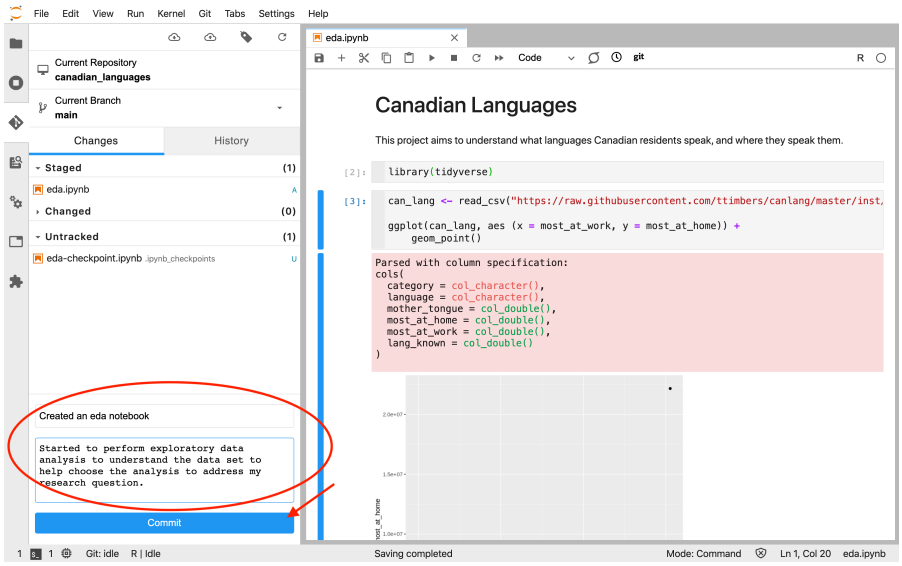


This moves the file from the "Untracked" heading to the "Staged" heading, flagging this file so that Git knows we want a snapshot of its current state as a commit. Now we are ready to "commit" the changes. Make sure to include a (clear and helpful!) message about what was changed so that your collaborators (and future you) know what happened in this commit.



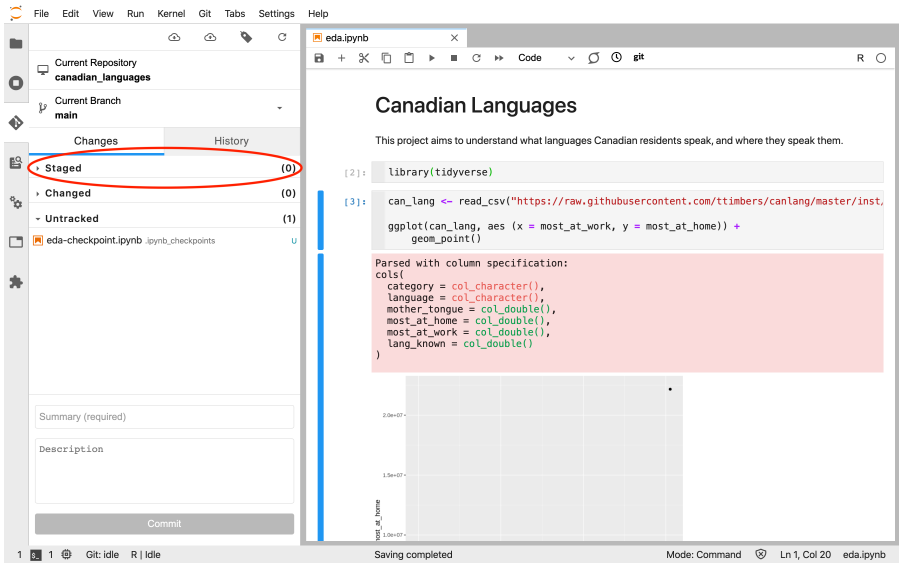
5.7.2 Making the commit

To snapshot the changes with an associated commit message, we put the message in the text box at the bottom of the Git pane and click on the blue "Commit" button. It is highly recommended to write useful and meaningful messages about what was changed. These commit messages, and the datetime stamp for a given commit, are the primary means to navigate through the project's histry in the event that we need to view or retrieve a past version of a file, or revert our project to an earlier state.



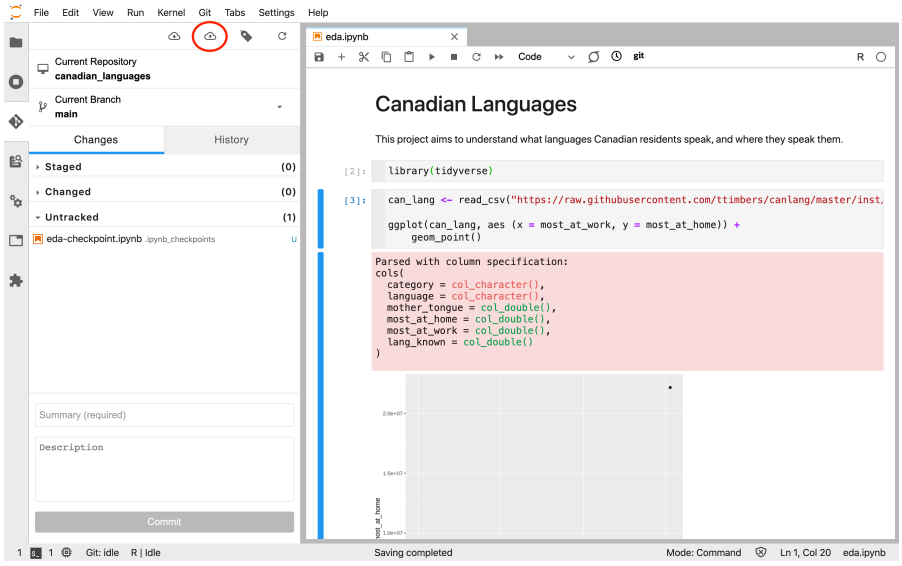
When you click the "Commit" button for the first time, you will be prompted to enter your name and email. This only needs to be done once for each machine you use Git on.

After "commiting" the file(s), you will see there there are 0 "Staged" files and we are now ready to push our changes (and the attached commit message) to our project repository on GitHub:

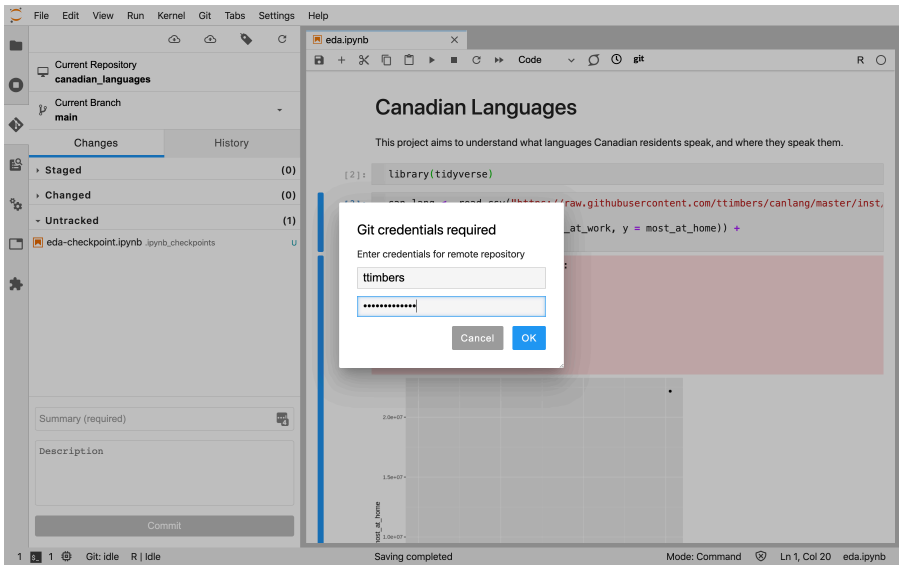


5.7.3 Pushing the commits to GitHub

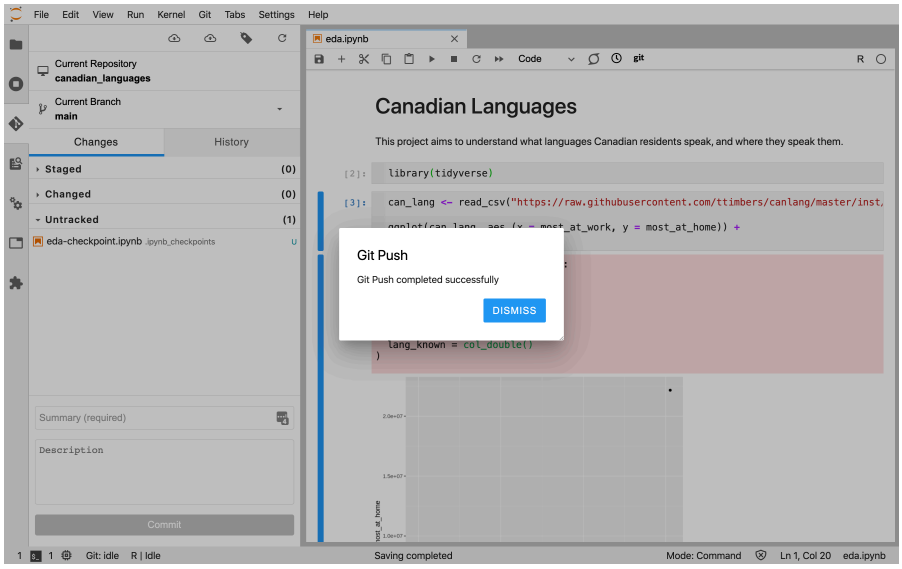
To send the committed changes back to the project repository on GitHub, we need to *push* them. To do this we click on the cloud icon with the up arrow on the Jupyter Git tab:



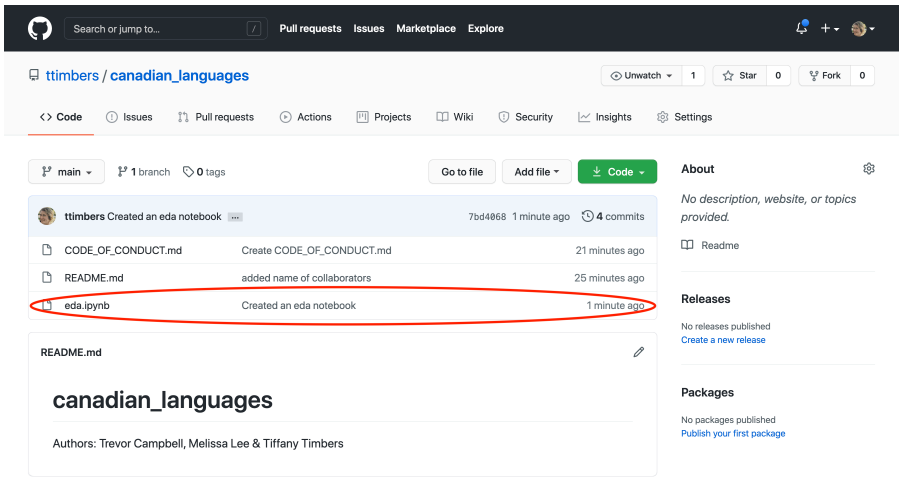
We will then be prompted to enter our GitHub username and password, and click the blue "OK" button:



If the files were successfully pushed to our project repository on GitHub we will be given the success message shown below. Click "Dismiss" to continue working in Jupyter.



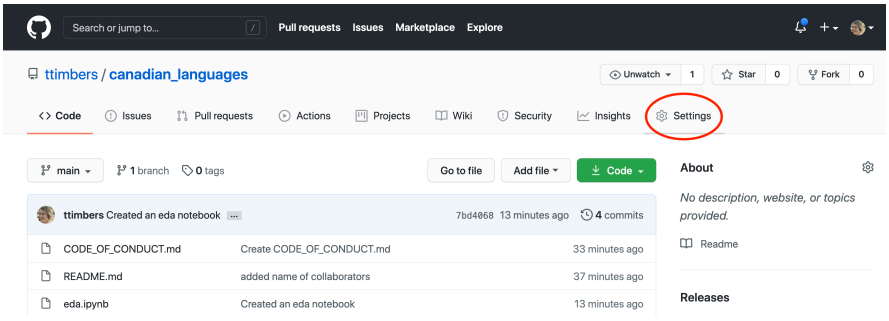
You will see that the changes now exist there!



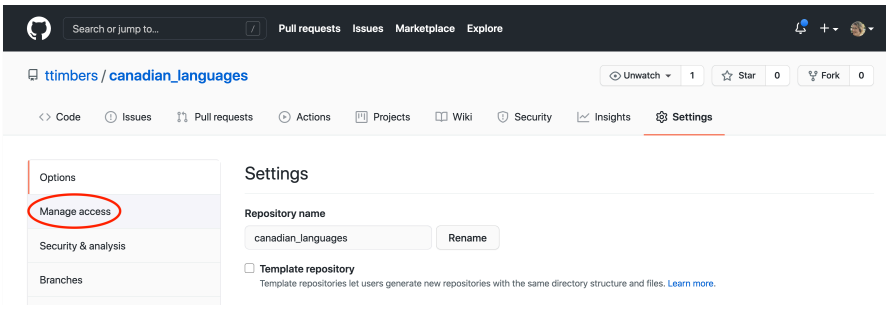
5.8 Collaboration

5.8.1 Giving collaborators access to your project

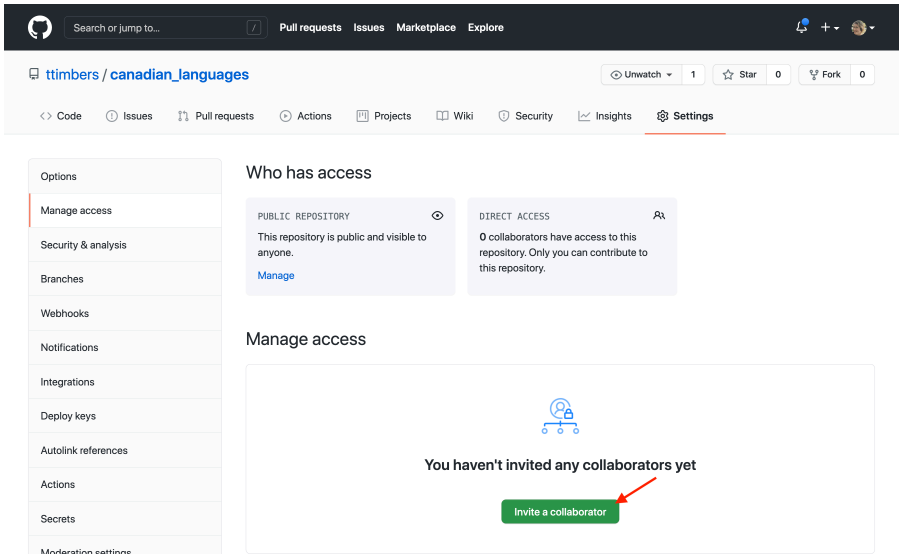
As mentioned earlier, GitHub allows you to control who has access to your project. The default of both public and private projects are that only the person who created the GitHub repository has permissions to create, edit and delete files (*write access*). To give your collaborators write access to the projects, navigate to the "Settings" tab:



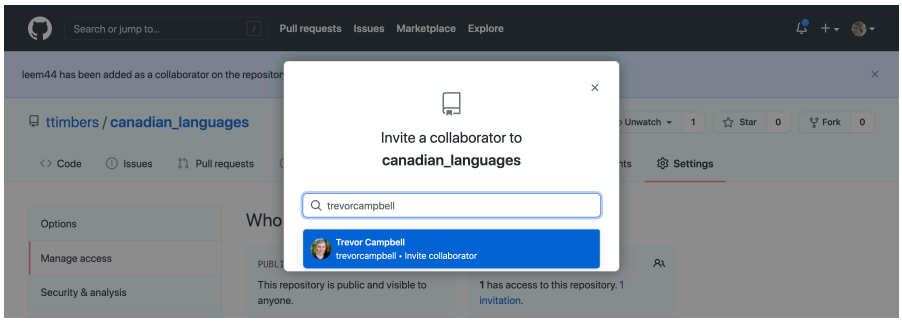
Then click "Manage access":



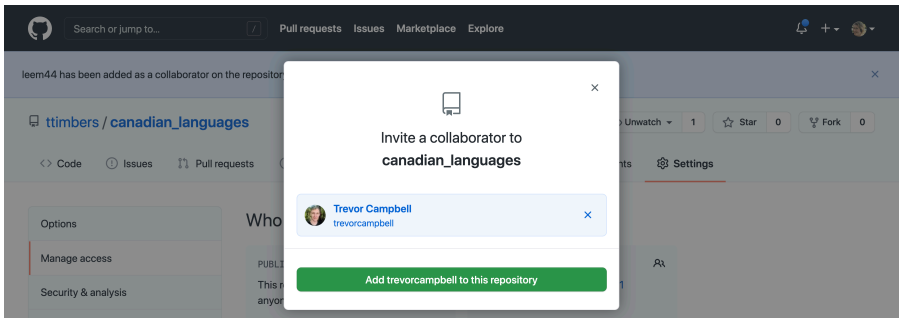
Click the green "Invite a collaborator" button:



Type in the collaborator's GitHub username and select their name when it appears:



Finally, click the green "Add to this repository" button:

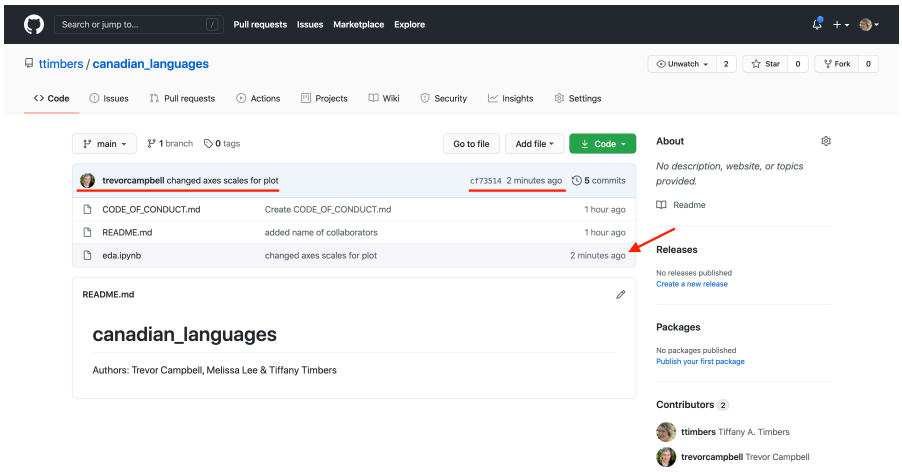


After this you should see your newly added collaborator listed under the "Manage access" tab. They should receive an email invitation to join the GitHub repository as a collaborator. They need to accept this invitation to enable write access.

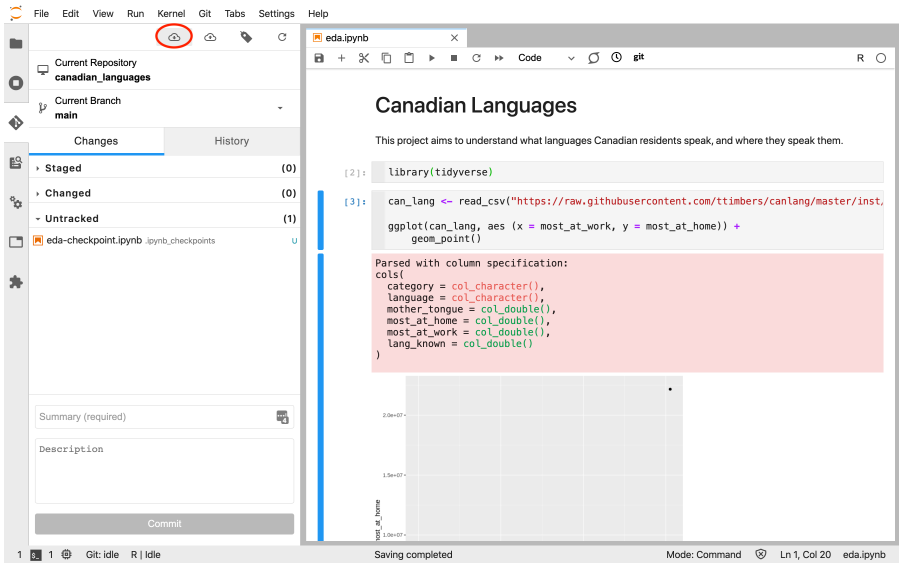
5.8.2 Pulling changes from GitHub

If your collaborators send their own commits to the GitHub repository, you will need to *pull* those changes to your own cloned copy on JupyterHub before you're allowed to push any more changes yourself. By pulling their changes, you sync your local repository to what is present on GitHub.

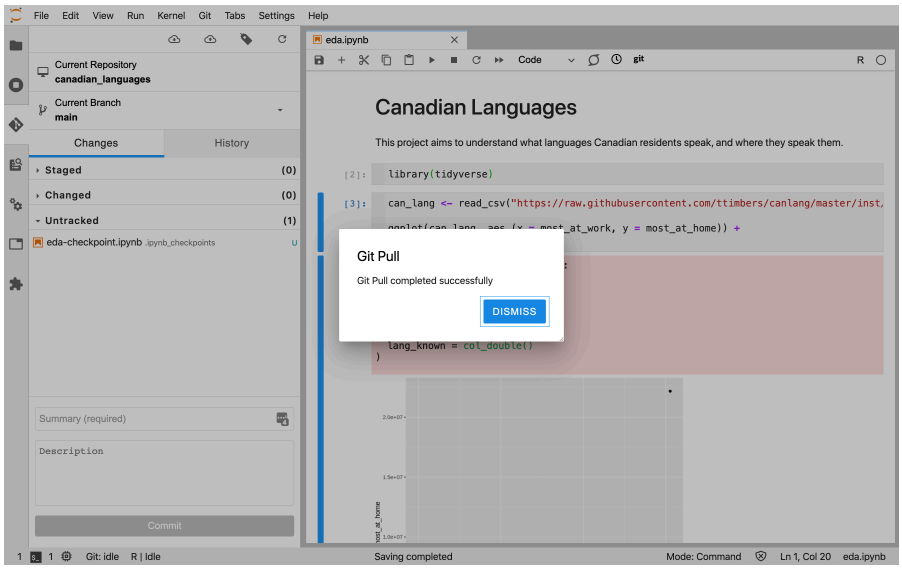
Note: you can still work on your own cloned repository and commit changes even if collaborators have pushed changes to the GitHub repository. It is only when you try to *push* your changes back to GitHub that Git will make sure nobody else has pushed any work in the meantime.



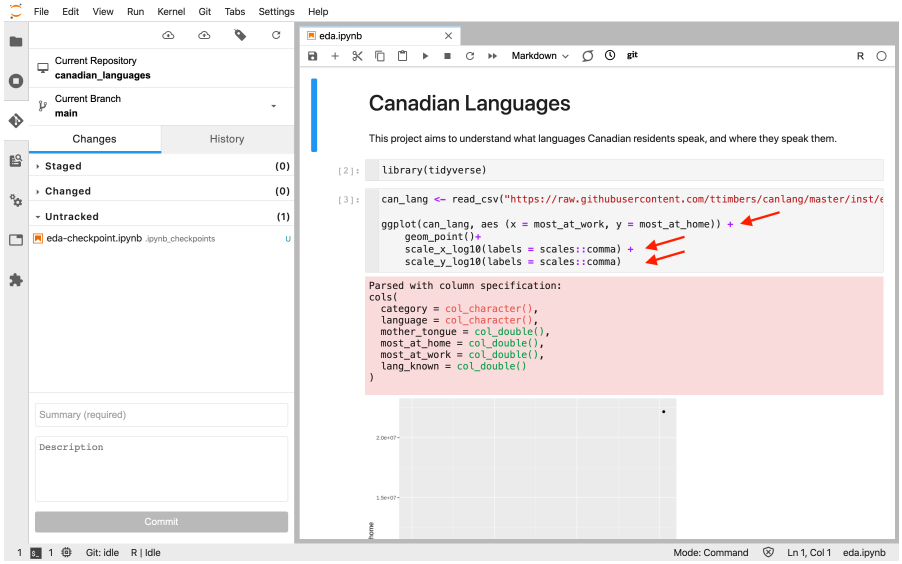
You can do this using the Jupyter Git tab by clicking on the cloud icon with the down arrow:



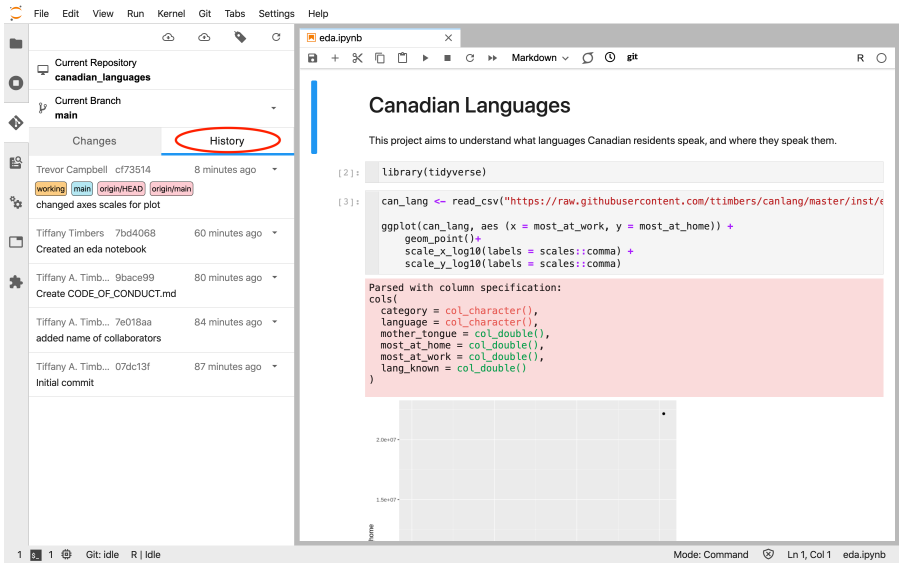
Once the files are successfully pulled from GitHub, you need to click "Dismiss" to keep working:



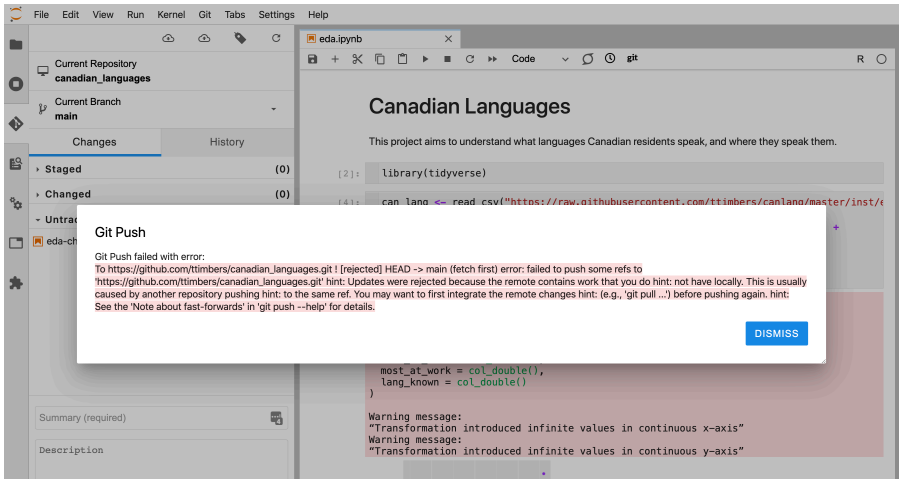
And then when you open (or refresh) the files whose changes you just pulled, you should be able to see them:



It can be very useful to review the history of the changes to your project. You can do this directly on the JupyterHub by clicking "History" in the Git tab.

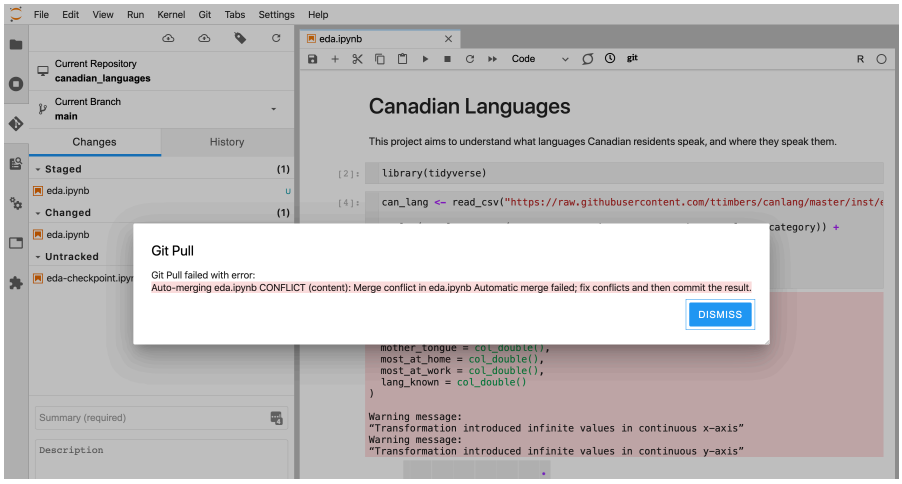


It is good practice to pull any changes at the start of *every* work session before you start working on your local copy. If you do not do this, and your collaborators have pushed some changes to the project to GitHub, then you will be unable to push your changes to GitHub until you pull. This situation can be recognized by this error message:



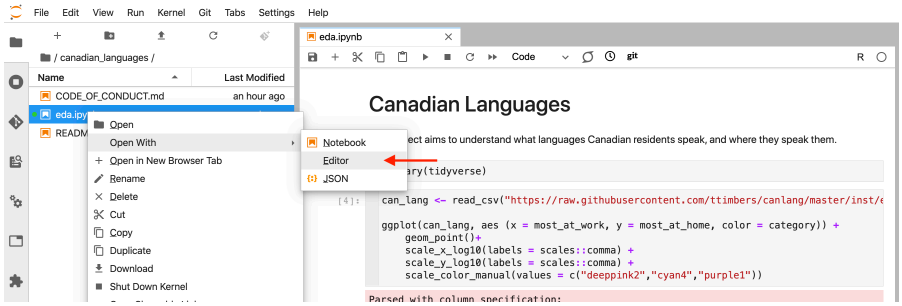
Usually, getting out of this situation is not too troublesome. First you need to pull the changes that exist on GitHub that you do not yet have on your machine. Usually when this happens, Git can automatically merge the changes for you, even if you and your collaborators were working on different parts of the same file!

If however, you and your collaborators made changes to the same line of the same file, Git will not be able to automatically merge the changes—it will not know whether to keep your version of the line(s), your collaborators version of the line(s), or some blend of the two. When this happens, Git will tell you that you have a merge conflict and that it needs human intervention (you!), and which file(s) this occurs in.

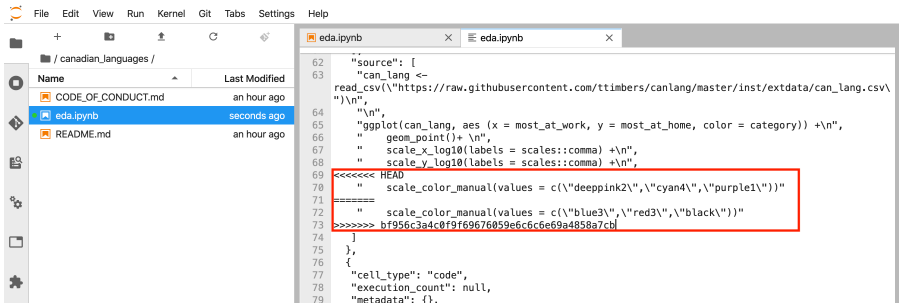


5.8.3 Handling merge conflicts

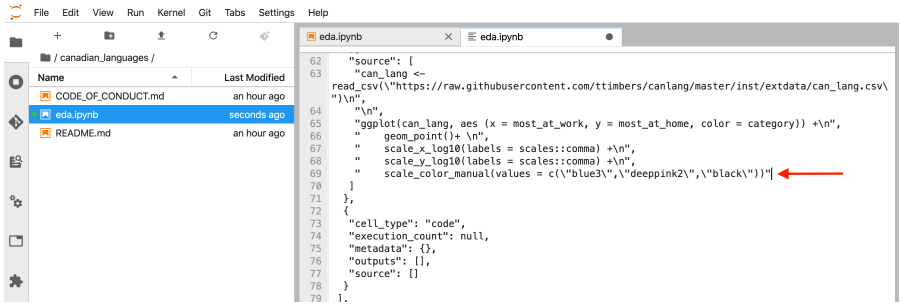
To fix the merge conflict we need to open the file that had the merge conflict in a plain text editor and look for special marks that Git puts in the file to tell you where the merge conflict occurred.



The beginning of the merge conflict is preceded by `<<<<<<< HEAD` and the end of the merge conflict is marked by `>>>>>>>`. Between these markings, Git also inserts a separator (`=======`). The version of the change before the separator is your change, and the version that follows the separator was the change that existed on GitHub.



Once you have decided which version of the change (or what combination!) to keep, you need to use the plain text editor to remove the special marks that Git added.



The file must be saved, added to the staging area, and then committed before you will be able to push your changes to GitHub.

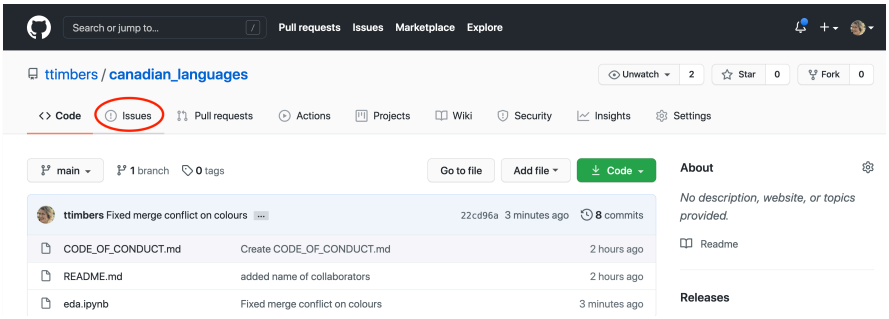
5.8.4 Communicating using GitHub issues

When working on a project in a team, you don't just want a historical record of who changed what file and when in the project—you also want a record of decisions that were made, ideas that were floated, problems that were identified and addressed, and all other communication surrounding the project. Email and messaging apps are both very popular for general communication, but are not designed for project-specific communication: they both generally do not have facilities for organizing conversations by project subtopics, searching for conversations related to particular bugs or software versions, etc.

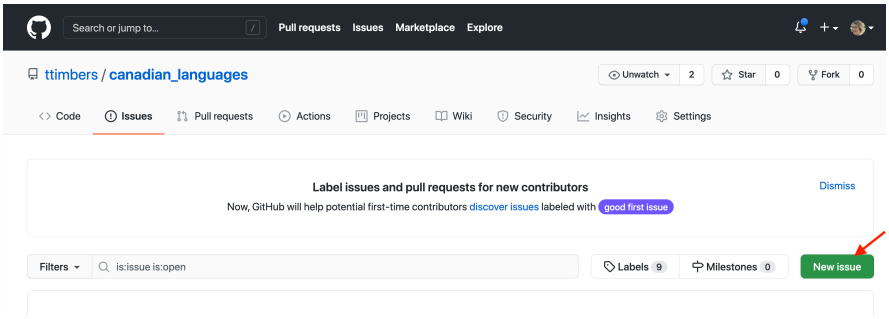
GitHub *issues* are an alternative written communication medium to email and messaging apps, and were designed specifically to facilitate project-specific communication. Issues are *opened* from the "Issues" tab on the project's GitHub page, and they persist there even after the conversation is over and the issue is *closed* (in contrast to email, issues are not usually deleted). One issue thread is usually created per topic, and they are easily searchable using GitHub's search tools. All issues are accessible to all project collaborators, so no one is left out of the conversation. Finally, issues can be setup so that

team members get email notifications when a new issue is created or a new post is made in an issue thread. Replying to issues from email is also possible. Given all of these advantages, we highly recommend the use of issues for project-related communication.

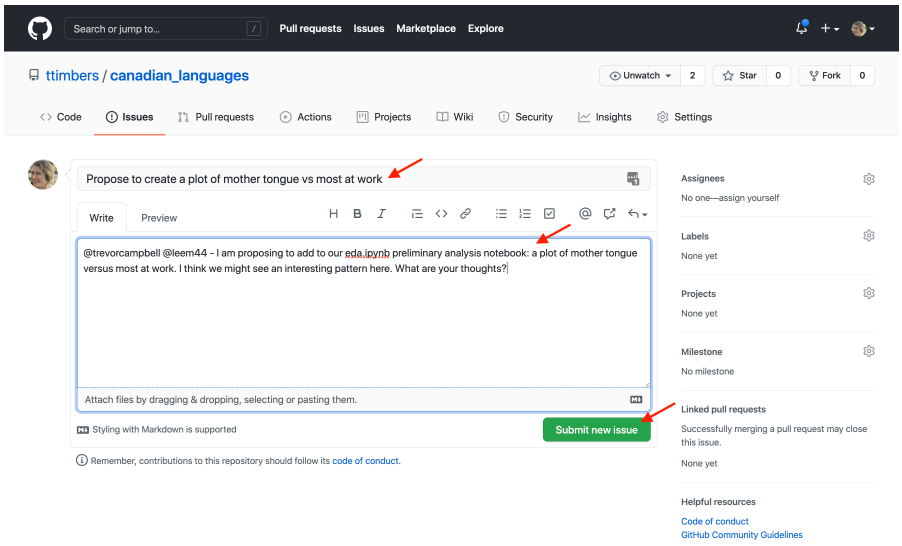
To open a GitHub issue, first click on the "Issues" tab:



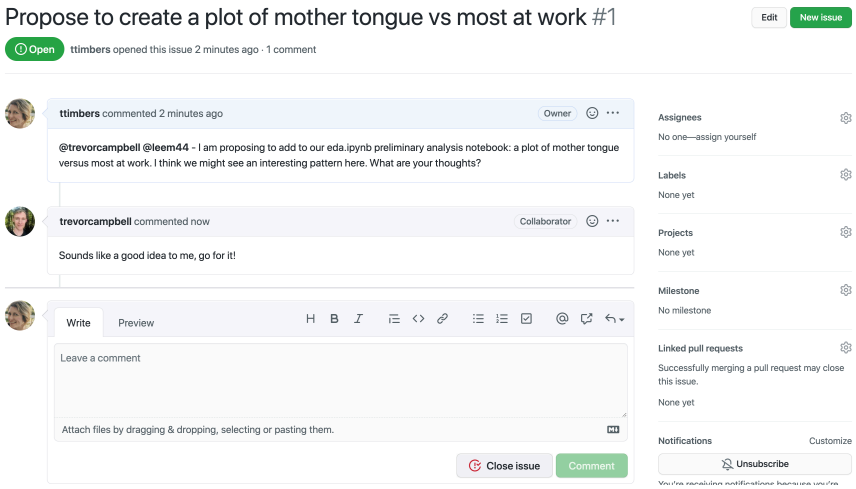
Next click the "New issue" button:



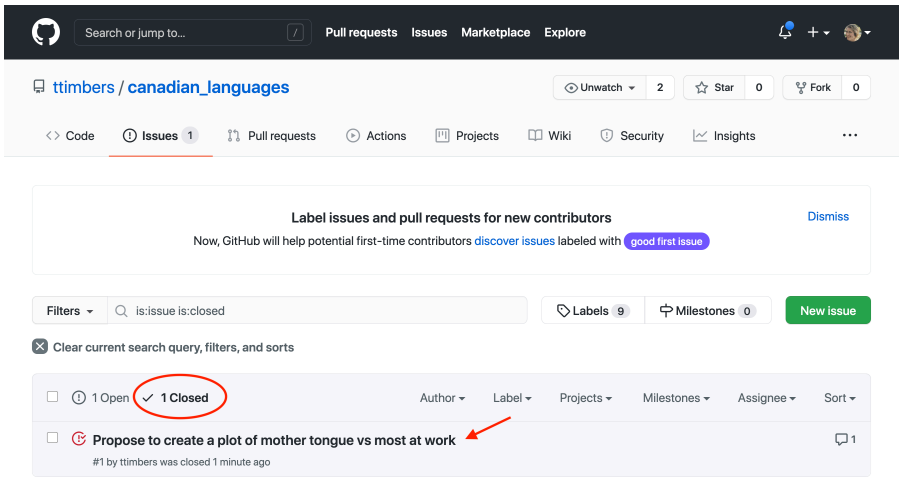
Add an issue title (which acts like an email subject line), and then put the body of the message in the larger text box. Finally click "Submit new issue" to post the issue to share with others:



You can reply to an issue that someone opened by adding your written response to the large text box and clicking comment:



When a conversation is resolved, you can click “Close issue”. The closed issue can be later viewed be clicking the “Closed” header link in the “Issue” tab:



5.9 Additional resources

Now that you've picked up the basics of version control with Git and GitHub, you can expand your knowledge using one of the many tutorials available online:

5.9.1 Best practices and workflows

- Good enough practices in scientific computing[10] (2014) by Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt & Tracy K. Teal
- Configuration Management for Large-Scale Scientific Computing at the UK Met Office[11] (2008) by David Mathews, Greg Wilson & Steve Easterbrook

5.9.2 Technical references

- GitHub's guides website[12]
- GitHub's YouTube channel[13]
- BitBucket's tutorials[14]

[10] https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510#sec014

[11] https://ieeexplore.ieee.org/iel5/5992/4653193/04653206.pdf?casa_token=6gY_5D7UbE4AAAAA:Z7BBJHplUfZEq31EfOpF_iP1qUQJwn170LdU8-ZUTtCu1N-ZpXYMV6yZHqIcSuRHLNVSkjM

[12] https://guides.github.com/

[13] https://www.youtube.com/githubguides

[14] https://www.atlassian.com/git/tutorials/what-is-version-control

6

Classification I: training & predicting

6.1 Overview

Up until this point, we have focused solely on descriptive and exploratory questions about data. This chapter and the next together serve as our first foray into answering *predictive* questions about data. In particular, we will focus on the problem of *classification*, i.e., using one or more quantitative variables to predict the value of a third, categorical variable. This chapter will cover the basics of classification, how to preprocess data to make it suitable for use in a classifier, and how to use our observed data to make predictions. The next will focus on how to evaluate how accurate the predictions from our classifier are, as well as how to improve our classifier (where possible) to maximize its accuracy.

6.2 Chapter learning objectives

- Recognize situations where a classifier would be appropriate for making predictions
- Describe what a training data set is and how it is used in classification
- Interpret the output of a classifier
- Compute, by hand, the straight-line (Euclidean) distance between points on a graph when there are two explanatory variables/predictors
- Explain the K-nearest neighbour classification algorithm
- Perform K-nearest neighbour classification in R using `tidymodels`

- Explain why one should center, scale, and balance data in predictive modelling
- Preprocess data to center, scale, and balance a dataset using a `recipe`
- Combine preprocessing and model training using a Tidymodels `workflow`

6.3 The classification problem

In many situations, we want to make predictions based on the current situation as well as past experiences. For instance, a doctor may want to diagnose a patient as either diseased or healthy based on their symptoms and the doctor's past experience with patients; an email provider might want to tag a given email as "spam" or "not spam" depending on past email text data; or an online store may want to predict whether an order is fraudulent or not.

These tasks are all examples of **classification**, i.e., predicting a categorical class (sometimes called a *label*) for an observation given its other quantitative variables (sometimes called *features*). Generally, a classifier assigns an observation (e.g. a new patient) to a class (e.g. diseased or healthy) on the basis of how similar it is to other observations for which we know the class (e.g. previous patients with known diseases and symptoms). These observations with known classes that we use as a basis for prediction are called a **training set**. We call them a "training set" because we use these observations to train, or teach, our classifier so that we can use it to make predictions on new data that we have not seen previously.

There are many possible classification algorithms that we could use to predict a categorical class/label for an observation. In addition, there are many variations on the basic classification problem, e.g., binary classification where only two classes are involved (e.g. disease or healthy patient), or multiclass classification, which involves assigning an object to one of several classes (e.g., private, public, or not for-profit organization). Here we will focus on the simple, widely used **K-nearest neighbours** algorithm for the binary classification problem. Other examples you may encounter in future courses include decision trees, support vector machines (SVMs), logistic regression, and neural networks.

6.4 Exploring a labelled data set

In this chapter and the next, we will study a data set of digitized breast cancer image features[1], created by Dr. William H. Wolberg, W. Nick Street, and Olvi L. Mangasarian at the University of Wisconsin, Madison. Each row in the data set represents an image of a tumour sample, including the diagnosis (benign or malignant) and several other measurements (e.g., nucleus texture, perimeter, area, etc.). Diagnosis for each image was conducted by physicians.

[1] http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

As with all data analyses, we first need to formulate a precise question that we want to answer. Here, the question is *predictive*: can we use the tumour image measurements available to us to predict whether a future tumour image (with unknown diagnosis) shows a benign or malignant tumour? Answering this question is important because traditional, non-data-driven methods for tumour diagnosis are quite subjective and dependent upon how skilled and experienced the diagnosing physician is. Furthermore, benign tumours are not normally dangerous; the cells stay in the same place and the tumour stops growing before it gets very large. By contrast, in malignant tumours, the cells invade the surrounding tissue and spread into nearby organs where they can cause serious damage (learn more about cancer here[2]). Thus, it is important to quickly and accurately diagnose the tumour type to guide patient treatment.

Loading the data

Our first step is to load, wrangle, and explore the data using visualizations in order to better understand the data we are working with. We start by loading the necessary packages for our analysis. Below you'll see (in addition to the usual `tidyverse`) a new package: `forcats`. The `forcats` package enables us to easily manipulate factors in R; factors are a special categorical type of variable in R that are often used for class label data.

```
library(tidyverse)
library(forcats)
```

In this case, the file containing the breast cancer data set is a simple `.csv` file with headers. We'll use the `read_csv` function with no additional arguments, and then inspect its contents:

```
cancer <- read_csv("data/wdbc.csv")
cancer
```

```
## # A tibble: 569 x 12
##        ID Class Radius Texture Perimeter   Area
##     <dbl> <chr>  <dbl>   <dbl>     <dbl>  <dbl>
##  1 8.42e5 M      1.10   -2.07     1.27    0.984
##  2 8.43e5 M      1.83   -0.353    1.68    1.91
##  3 8.43e7 M      1.58    0.456    1.57    1.56
##  4 8.43e7 M     -0.768   0.254   -0.592  -0.764
##  5 8.44e7 M      1.75   -1.15     1.78    1.82
##  6 8.44e5 M     -0.476  -0.835   -0.387  -0.505
##  7 8.44e5 M      1.17    0.161    1.14    1.09
##  8 8.45e7 M     -0.118   0.358   -0.0728 -0.219
```

[2] https://www.worldwidecancerresearch.org/who-we-are/cancer-basics/

```
##  9 8.45e5 M     -0.320   0.588   -0.184  -0.384
## 10 8.45e7 M     -0.473   1.10    -0.329  -0.509
## # ... with 559 more rows, and 6 more variables:
## #   Smoothness <dbl>, Compactness <dbl>,
## #   Concavity <dbl>, Concave_Points <dbl>,
## #   Symmetry <dbl>, Fractal_Dimension <dbl>
```

Variable descriptions

Breast tumours can be diagnosed by performing a *biopsy*, a process where tissue is removed from the body and examined for the presence of disease. Traditionally these procedures were quite invasive; modern methods such as fine needle asipiration, used to collect the present data set, extract only a small amount of tissue and are less invasive. Based on a digital image of each breast tissue sample collected for this data set, 10 different variables were measured for each cell nucleus in the image (3-12 below), and then the mean for each variable across the nuclei was recorded. As part of the data preparation, these values have been *scaled*; we will discuss what this means and why we do it later in this chapter. Each image additionally was given a unique ID and a diagnosis for malignance by a physician. Therefore, the total set of variables per image in this data set are:

1. ID number
2. Class: the diagnosis of **M**alignant or **B**enign
3. Radius: the mean of distances from center to points on the perimeter
4. Texture: the standard deviation of gray-scale values
5. Perimeter: the length of the surrounding contour
6. Area: the area inside the contour
7. Smoothness: the local variation in radius lengths
8. Compactness: the ratio of squared perimeter and area
9. Concavity: severity of concave portions of the contour
10. Concave Points: the number of concave portions of the contour
11. Symmetry
12. Fractal Dimension

Below we use `glimpse` to preview the data frame. This function can make it easier to inspect the data when we have a lot of columns:

```
glimpse(cancer)
```

```
## Rows: 569
## Columns: 12
## $ ID                <dbl> 842302, 842517, 84300903...
## $ Class             <chr> "M", "M", "M", "M", "M",...
## $ Radius            <dbl> 1.0961, 1.8282, 1.5785, ...
## $ Texture           <dbl> -2.0715, -0.3533, 0.4558...
```

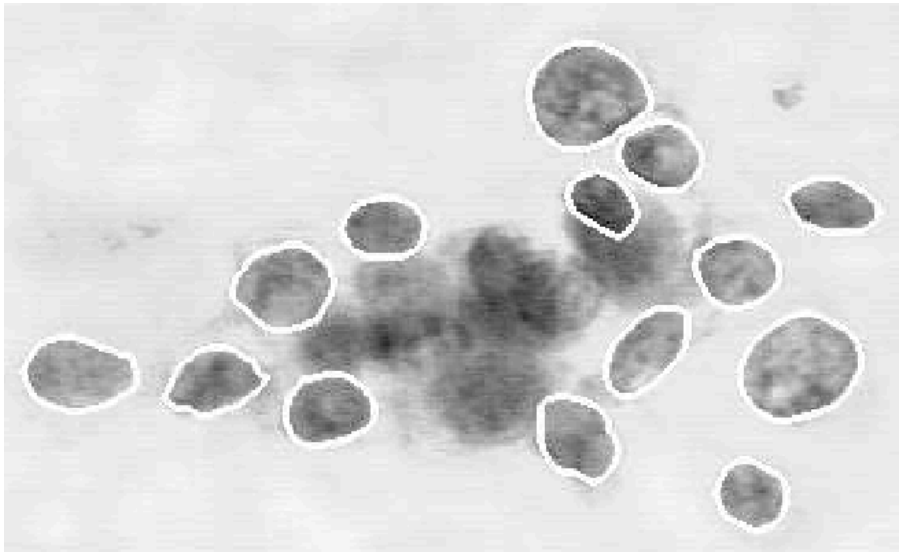


FIGURE 6.1: A malignant breast fine needle aspiration image. [Source](https://pubsonline.informs.org/doi/abs/10.1287/opre.43.4.570)

```
## $ Perimeter         <dbl> 1.26882, 1.68447, 1.5651...
## $ Area              <dbl> 0.98351, 1.90703, 1.5575...
## $ Smoothness        <dbl> 1.56709, -0.82624, 0.941...
## $ Compactness       <dbl> 3.28063, -0.48664, 1.052...
## $ Concavity         <dbl> 2.65054, -0.02382, 1.362...
## $ Concave_Points    <dbl> 2.5302, 0.5477, 2.0354, ...
## $ Symmetry          <dbl> 2.215566, 0.001391, 0.93...
## $ Fractal_Dimension <dbl> 2.2538, -0.8679, -0.3977...
```

We can see from the summary of the data above that `Class` is of type character (denoted by `<chr>`). Since we are going to be working with `Class` as a categorical statistical variable, we will convert it to factor using the function `as_factor`.

```
cancer <- cancer %>%
  mutate(Class = as_factor(Class))
glimpse(cancer)
```

```
## Rows: 569
## Columns: 12
## $ ID                <dbl> 842302, 842517, 84300903...
## $ Class             <fct> M, M, M, M, M, M, M, M, ...
## $ Radius            <dbl> 1.0961, 1.8282, 1.5785, ...
```

```
## $ Texture           <dbl> -2.0715, -0.3533, 0.4558...
## $ Perimeter         <dbl> 1.26882, 1.68447, 1.5651...
## $ Area              <dbl> 0.98351, 1.90703, 1.5575...
## $ Smoothness        <dbl> 1.56709, -0.82624, 0.941...
## $ Compactness       <dbl> 3.28063, -0.48664, 1.052...
## $ Concavity         <dbl> 2.65054, -0.02382, 1.362...
## $ Concave_Points    <dbl> 2.5302, 0.5477, 2.0354, ...
## $ Symmetry          <dbl> 2.215566, 0.001391, 0.93...
## $ Fractal_Dimension <dbl> 2.2538, -0.8679, -0.3977...
```

Factors have what are called "levels", which you can think of as categories. We can ask for the levels from the `Class` column by using the `levels` function. This function should return the name of each category in that column. Given that we only have 2 different values in our `Class` column (B and M), we only expect to get two names back. Note that the `levels` function requires a *vector* argument, while the `select` function outputs a *data frame*; so we use the `pull` function, which converts a single column of a data frame into a vector.

```
cancer %>%
  select(Class) %>%
  pull() %>% # turns a data frame into a vector
  levels()
```

```
## [1] "M" "B"
```

Exploring the data

Before we start doing any modelling, let's explore our data set. Below we use the `group_by` + `summarize` code pattern we used before to see that we have 357 (63%) benign and 212 (37%) malignant tumour observations.

```
num_obs <- nrow(cancer)
cancer %>%
  group_by(Class) %>%
  summarize(
    n = n(),
    percentage = n() / num_obs * 100
  )
```

```
## # A tibble: 2 x 3
##   Class     n percentage
##   <fct> <int>      <dbl>
## 1 M       212       37.3
## 2 B       357       62.7
```

Next, let's draw a scatter plot to visualize the relationship between the perime-

ter and concavity variables. Rather than use `ggplot's` default palette, we define our own here (`cbPalette`) and pass it as the `values` argument to the `scale_color_manual` function. We also make the category labels ("B" and "M") more readable by changing them to "Benign" and "Malignant" using the `labels` argument.

```
# colour palette
cbPalette <- c("#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC79A7", "#9999

perim_concav <- cancer %>%
  ggplot(aes(x = Perimeter, y = Concavity, color = Class)) +
  geom_point(alpha = 0.5) +
  labs(color = "Diagnosis") +
  scale_color_manual(labels = c("Malignant", "Benign"), values = cbPalette)
perim_concav
```

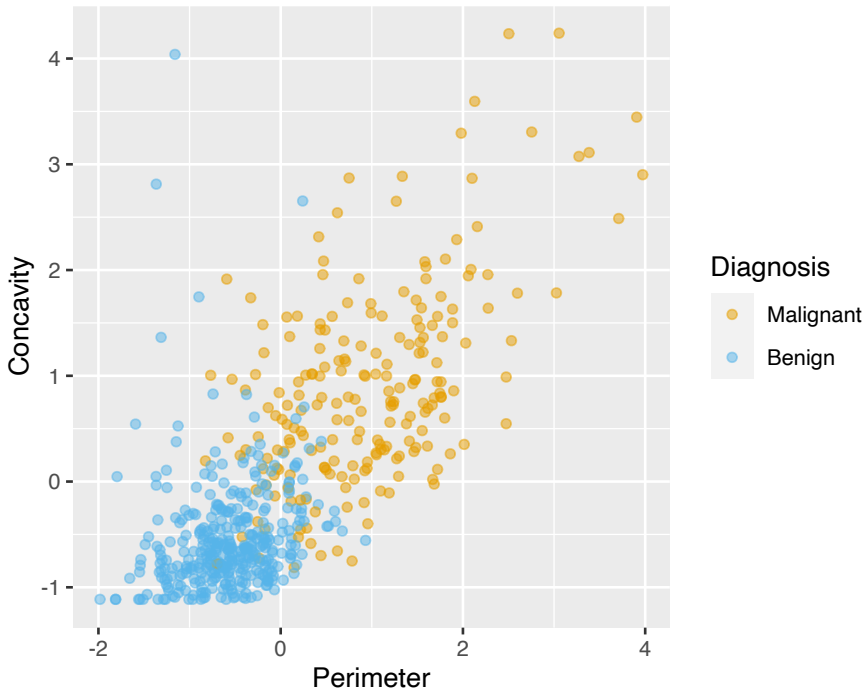


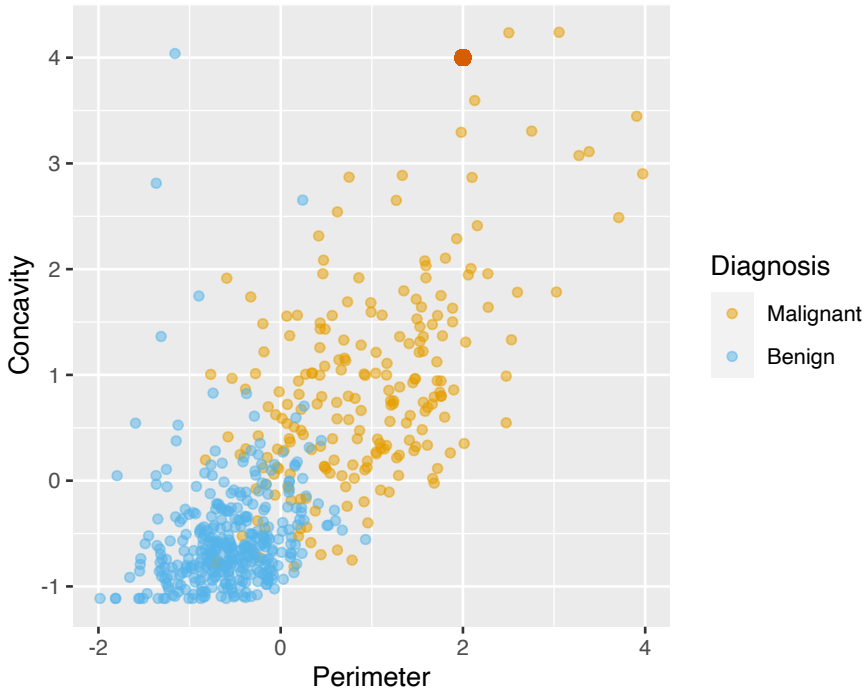
FIGURE 6.2: Scatterplot of concavity versus perimeter coloured by diagnosis label

In this visualization, we can see that malignant observations typically fall in the the upper right-hand corner of the plot area. By contrast, benign obser-

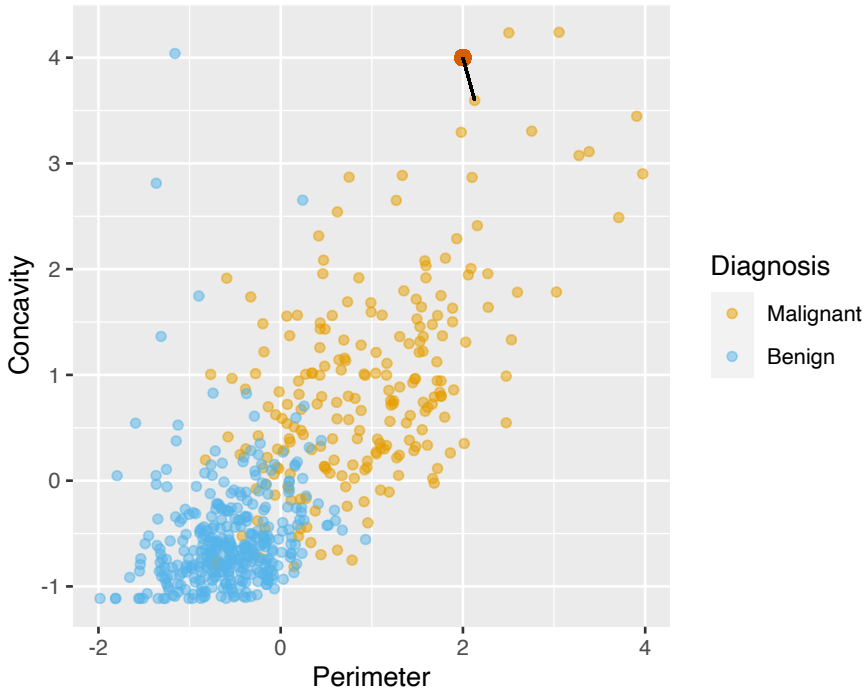
vations typically fall in lower left-hand corner of the plot. Suppose we obtain a new observation not in the current data set that has all the variables measured *except* the label (i.e., an image without the physician's diagnosis for the tumour class). We could compute the perimeter and concavity values, resulting in values of, say, 1 and 1. Could we use this information to classify that observation as benign or malignant? What about a new observation with perimeter value of -1 and concavity value of -0.5? What about 0 and 1? It seems like the *prediction of an unobserved label* might be possible, based on our visualization. In order to actually do this computationally in practice, we will need a classification algorithm; here we will use the K-nearest neighbour classification algorithm.

6.5 Classification with K-nearest neighbours

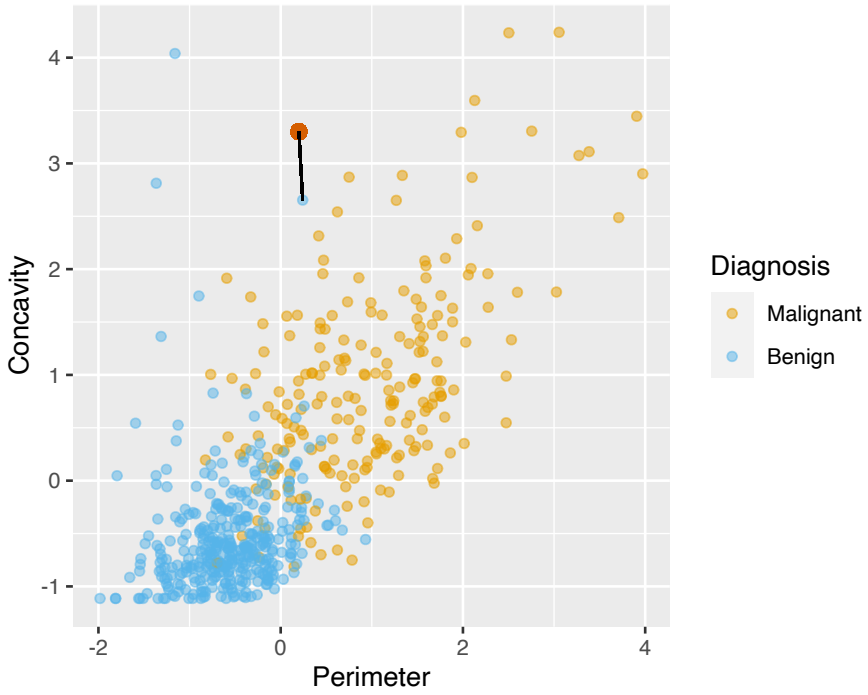
To predict the label of a new observation, i.e., classify it as either benign or malignant, the K-nearest neighbour classifier generally finds the K "nearest" or "most similar" observations in our training set, and then uses their diagnoses to make a prediction for the new observation's diagnosis. To illustrate this concept, we will walk through an example. Suppose we have a new observation, with perimeter of 2 and concavity of 4 (labelled in red on the scatterplot), whose diagnosis "Class" is unknown.



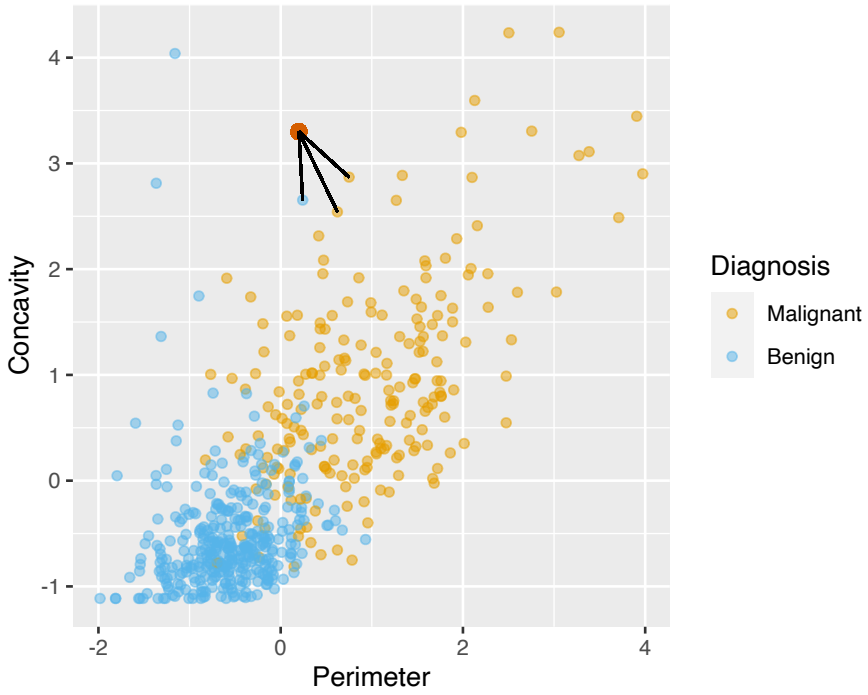
We see that the nearest point to this new observation is **malignant** and located at the coordinates (2.1, 3.6). The idea here is that if a point is close to another in the scatterplot, then the perimeter and concavity values are similar, and so we may expect that they would have the same diagnosis.



Suppose we have another new observation with perimeter 0.2 and concavity of 3.3. Looking at the scatterplot below, how would you classify this red observation? The nearest neighbour to this new point is a **benign** observation at (0.2, 2.7). Does this seem like the right prediction to make? Probably not, if you consider the other nearby points…



So instead of just using the one nearest neighbour, we can consider several neighbouring points, say $K = 3$, that are closest to the new red observation to predict its diagnosis class. Among those 3 closest points, we use the *majority class* as our prediction for the new observation. In this case, we see that the diagnoses of 2 of the 3 nearest neighbours to our new observation are malignant. Therefore we take majority vote and classify our new red observation as malignant.



Here we chose the $K = 3$ nearest observations, but there is nothing special about $K = 3$. We could have used $K = 4, 5$ or more (though we may want to choose an odd number to avoid ties). We will discuss more about choosing K in the next chapter.

Distance between points

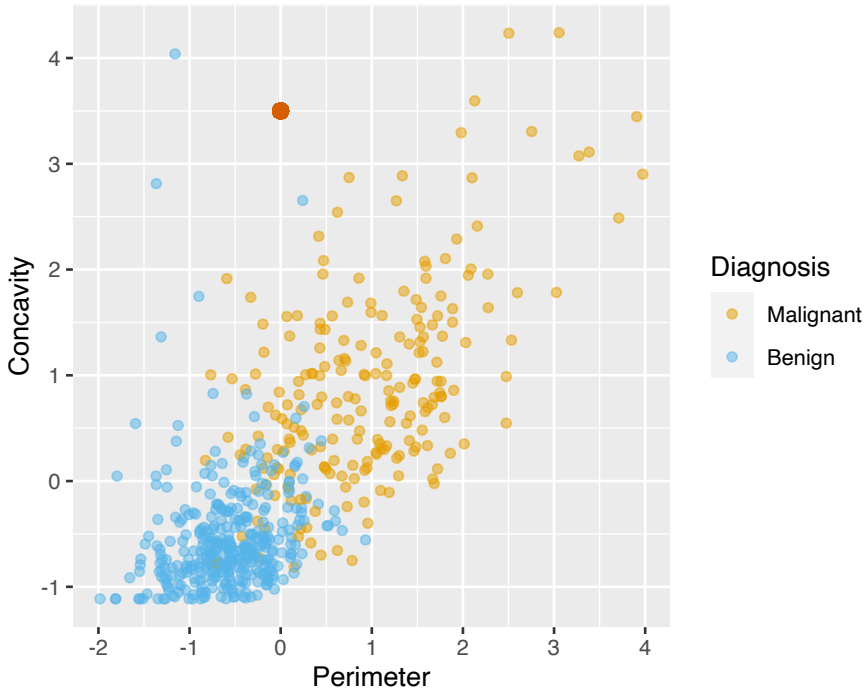
How do we decide which points are the K "nearest" to our new observation? We can compute the distance between any pair of points using the following formula:

$$\text{Distance} = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

This formula – sometimes called the *Euclidean distance* – is simply the straight line distance between two points on the x-y plane with coordinates (x_a, y_a) and (x_b, y_b).

Suppose we want to classify a new observation with perimeter of 0 and con-

cavity of 3.5. Let's calculate the distances between our new point and each of the observations in the training set to find the $K = 5$ observations in the training data that are nearest to our new point.

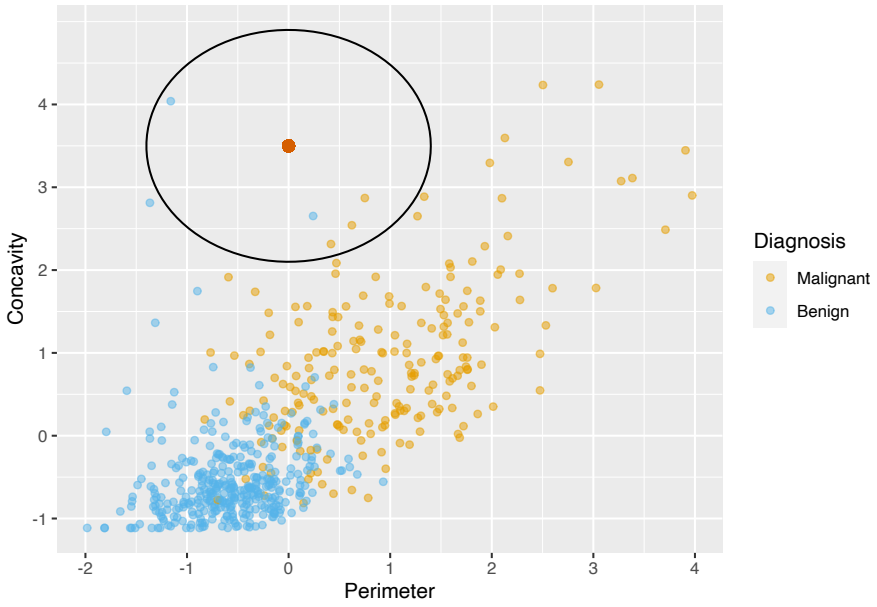



```
new_obs_Perimeter <- 0
new_obs_Concavity <- 3.5
cancer %>%
  select(ID, Perimeter, Concavity, Class) %>%
  mutate(dist_from_new = sqrt((Perimeter - new_obs_Perimeter)^2 + (Concavity - new_obs_Concavity)^
  arrange(dist_from_new) %>%
  slice(1:5) # subset the first 5 rows
```

```
## # A tibble: 5 x 5
##        ID Perimeter Concavity Class dist_from_new
##     <dbl>     <dbl>     <dbl> <fct>         <dbl>
## 1   86409     0.241      2.65 B             0.881
## 2  887181     0.750      2.87 M             0.980
## 3  899667     0.623      2.54 M             1.14
## 4  907914     0.417      2.31 M             1.26
## 5 8710441    -1.16       4.04 B             1.28
```

From this, we see that 3 of the 5 nearest neighbours to our new observation

are malignant so classify our new observation as malignant. We circle those 5 in the plot below:



It can be difficult sometimes to read code as math, so here we mathematically show the calculation of distance for each of the 5 closest points.

Perimeter	Concavity	Distance	Class
0.24	2.65	$\sqrt{0-0.241)^2 + (3.5-2.65)^2} = 0.88$	B
0.75	2.87	$\sqrt{(0-0.750)^2 + (3.5-2.87)^2} = 0.98$	M
0.62	2.54	$\sqrt{(0-0.623)^2 + (3.5-2.54)^2} = 1.14$	M
0.42	2.31	$\sqrt{(0-0.417)^2 + (3.5-2.31)^2} = 1.26$	M
-1.16	4.04	$\sqrt{(0-(-1.16))^2 + (3.5-4.04)^2} = 1.28$	B

More than two explanatory variables

Although the above description is directed toward two explanatory variables / predictors, exactly the same K-nearest neighbour algorithm applies when you have a higher number of explanatory variables (i.e., a higher-dimensional predictor space). Each explanatory variable/predictor can give us new information to help create our classifier. The only difference is the formula for the distance between points. In particular, let's say we have m predictor variables for two observations u and v, i.e., $u = (u_1, u_2, \dots, u_m)$ and $v = (v_1, v_2, \dots, v_m)$. Before, we added up the squared difference between each of our (two) vari-

ables, and then took the square root; now we will do the same, except for *all* of our m variables. In other words, the distance formula becomes

$$Distance = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + \cdots + (u_m - v_m)^2}$$

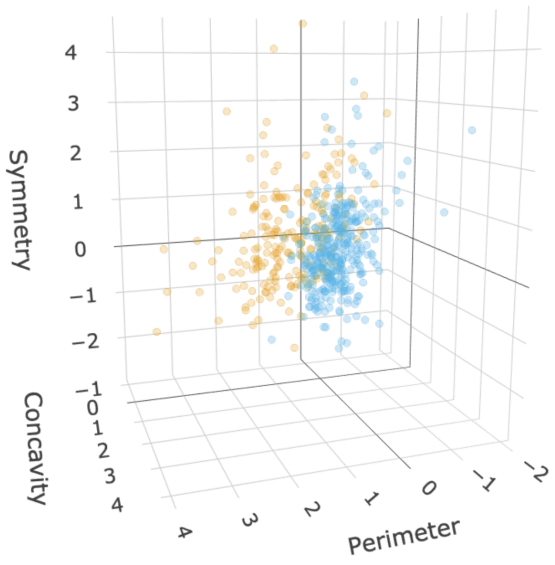


FIGURE 6.3: 3D scatterplot of symmetry, concavity and perimeter

Click and drag the plot above to rotate it, and scroll to zoom. Note that in general we recommend against using 3D visualizations; here we show the data in 3D only to illustrate what "higher dimensions" look like for learning purposes.

Summary

In order to classify a new observation using a K-nearest neighbour classifier, we have to:

1. Compute the distance between the new observation and each observation in the training set
2. Sort the data table in ascending order according to the distances
3. Choose the top K rows of the sorted table
4. Classify the new observation based on a majority vote of the neighbour classes

6.6 K-nearest neighbours with `tidymodels`

Coding the K-nearest neighbour algorithm in R ourselves would get complicated if we might have to predict the label/class for multiple new observations, or when there are multiple classes and more than two variables. Thankfully, in R, the K-nearest neighbour algorithm is implemented in the `parsnip` package included in the `tidymodels` package collection[3], along with many other models[4] that you will encounter in this and future classes. The `tidymodels` collection provides tools to help make and use models, such as classifiers. Using the packages in this collection will help keep our code simple, readable and accurate; the less we have to code ourselves, the fewer mistakes we are likely to make. We start off by loading `tidymodels`:

```
library(tidymodels)
```

Let's again suppose we have a new observation with perimeter 0 and concavity 3.5, but its diagnosis is unknown (as in our example above). Suppose we want to use the perimeter and concavity explanatory variables/predictors to predict the diagnosis class of this observation. Let's pick out our 2 desired predictor variables and class label and store it as a new dataset named `cancer_train`:

```
cancer_train <- cancer %>%
  select(Class, Perimeter, Concavity)
cancer_train
```

```
## # A tibble: 569 x 3
##    Class Perimeter Concavity
##    <fct>     <dbl>     <dbl>
##  1 M         1.27      2.65
##  2 M         1.68     -0.0238
##  3 M         1.57      1.36
##  4 M        -0.592     1.91
##  5 M         1.78      1.37
##  6 M        -0.387     0.866
##  7 M         1.14      0.300
##  8 M        -0.0728    0.0610
##  9 M        -0.184     1.22
## 10 M        -0.329     1.74
## # ... with 559 more rows
```

[3] https://www.tidymodels.org/
[4] https://www.tidymodels.org/find/parsnip/

Next, we create a *model specification* for K-nearest neighbours classification by calling the `nearest_neighbor` function, specifying that we want to use $K = 5$ neighbours (we will discuss how to choose K in the next chapter) and the straight-line distance (`weight_func = "rectangular"`). The `weight_func` argument controls how neighbours vote when classifying a new observation; by setting it to `"rectangular"`, each of the K nearest neighbours gets exactly 1 vote as described above. Other choices, which weight each neighbour's vote differently, can be found on the tidymodels website[5]. We specify the particular computational engine (in this case, the `kknn` engine) for training the model with the `set_engine` function. Finally we specify that this is a classification problem with the `set_mode` function.

```
knn_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = 5) %>%
  set_engine("kknn") %>%
  set_mode("classification")
knn_spec
```

```
## K-Nearest Neighbor Model Specification (classification)
##
## Main Arguments:
##   neighbors = 5
##   weight_func = rectangular
##
## Computational engine: kknn
```

In order to fit the model on the breast cancer data, we need to pass the model specification and the dataset to the `fit` function. We also need to specify what variables to use as predictors and what variable to use as the target. Below, the `Class ~ .` argument specifies that `Class` is the target variable (the one we want to predict), and `.` (everything *except* `Class`) is to be used as the predictor.

```
knn_fit <- knn_spec %>%
  fit(Class ~ ., data = cancer_train)
knn_fit
```

```
## parsnip model object
##
## Fit time:  14ms
##
## Call:
## kknn::train.kknn(formula = Class ~ ., data = data, ks = min_rows(5,    data, 5), kernel = ~"rectangul
##
```

[5] https://parsnip.tidymodels.org/reference/nearest_neighbor.html

```
## Type of response variable: nominal
## Minimal misclassification: 0.07557
## Best kernel: rectangular
## Best k: 5
```

Here you can see the final trained model summary. It confirms that the computational engine used to train the model was `kknn::train.kknn`. It also shows the fraction of errors made by the nearest neighbour model, but we will ignore this for now and discuss it in more detail in the next chapter. Finally it shows (somewhat confusingly) that the "best" weight function was "rectangular" and "best" setting of K was 5; but since we specified these earlier, R is just repeating those settings to us here. In the next chapter, we will actually let R tune the model for us.

Finally, we make the prediction on the new observation by calling the `predict` function, passing the fit object we just created. As above when we ran the K-nearest neighbours classification algorithm manually, the `knn_fit` object classifies the new observation as malignant ("M"). Note that the `predict` function outputs a data frame with a single variable named `.pred_class`.

```
new_obs <- tibble(Perimeter = 0, Concavity = 3.5)
predict(knn_fit, new_obs)
```

```
## # A tibble: 1 x 1
##   .pred_class
##   <fct>
## 1 M
```

6.7 Data preprocessing with `tidymodels`

6.7.1 Centering and scaling

When using K-nearest neighbour classification, the *scale* of each variable (i.e., its size and range of values) matters. Since the classifier predicts classes by identifying observations that are nearest to it, any variables that have a large scale will have a much larger effect than variables with a small scale. But just because a variable has a large scale *doesn't mean* that it is more important for making accurate predictions. For example, suppose you have a data set with two attributes, salary (in dollars) and years of education, and you want to predict the corresponding type of job. When we compute the neighbour distances, a difference of \$1000 is huge compared to a difference of 10 years of education. But for our conceptual understanding and answering of the prob-

lem, it's the opposite; 10 years of education is huge compared to a difference of $1000 in yearly salary!

In many other predictive models, the *center* of each variable (e.g., its mean) matters as well. For example, if we had a data set with a temperature variable measured in degrees Kelvin, and the same data set with temperature measured in degrees Celcius, the two variables would differ by a constant shift of 273 (even though they contain exactly the same information). Likewise in our hypothetical job classification example, we would likely see that the center of the salary variable is in the tens of thousands, while the center of the years of education variable is in the single digits. Although this doesn't affect the K-nearest neighbour classification algorithm, this large shift can change the outcome of using many other predictive models.

Standardization: when all variables in a data set have a mean (center) of 0 and a standard deviation (scale) of 1, we say that the data have been *standardized*.

To illustrate the effect that standardization can have on the K-nearest neighbour algorithm, we will read in the original, unscaled Wisconsin breast cancer data set; we have been using a standardized version of the data set up until now. To keep things simple, we will just use the `Area`, `Smoothness`, and `Class` variables:

```
unscaled_cancer <- read_csv("data/unscaled_wdbc.csv") %>%
  mutate(Class = as_factor(Class)) %>%
  select(Class, Area, Smoothness)
unscaled_cancer
```

```
## # A tibble: 569 x 3
##    Class  Area Smoothness
##    <fct> <dbl>      <dbl>
##  1 M     1001      0.118
##  2 M     1326      0.0847
##  3 M     1203      0.110
##  4 M      386.     0.142
##  5 M     1297      0.100
##  6 M      477.     0.128
##  7 M     1040      0.0946
##  8 M      578.     0.119
##  9 M      520.     0.127
## 10 M      476.     0.119
## # ... with 559 more rows
```

Looking at the unscaled / uncentered data above, you can see that the difference between the values for area measurements are much larger than those for smoothness, and the mean appears to be much larger too. Will this affect

predictions? In order to find out, we will create a scatter plot of these two predictors (coloured by diagnosis) for both the unstandardized data we just loaded, and the standardized version of that same data.

In the `tidymodels` framework, all data preprocessing happens using a `recipe`[6]. Here we will initialize a recipe for the `unscaled_cancer` data above, specifying that the `Class` variable is the target, and all other variables are predictors:

```
uc_recipe <- recipe(Class ~ ., data = unscaled_cancer)
print(uc_recipe)
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          2
```

So far, there is not much in the recipe; just a statement about the number of targets and predictors. Let's add scaling (`step_scale`) and centering (`step_center`) steps for all of the predictors so that they each have a mean of 0 and standard deviation of 1. The `prep` function finalizes the recipe by using the data (here, `unscaled_cancer`) to compute anything necessary to run the recipe (in this case, the column means and standard deviations):

```
uc_recipe <- uc_recipe %>%
  step_scale(all_predictors()) %>%
  step_center(all_predictors()) %>%
  prep()
uc_recipe
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          2
##
## Training data contained 569 data points and no missing data.
##
## Operations:
```

[6] https://tidymodels.github.io/recipes/reference/index.html

```
##
## Scaling for Area, Smoothness [trained]
## Centering for Area, Smoothness [trained]
```

You can now see that the recipe includes a scaling and centering step for all predictor variables. Note that when you add a step to a recipe, you must specify what columns to apply the step to. Here we used the `all_predictors()` function to specify that each step should be applied to all predictor variables. However, there are a number of different arguments one could use here, as well as naming particular columns with the same syntax as the `select` function. For example:

- `all_nominal()` and `all_numeric()`: specify all categorical or all numeric variables
- `all_predictors()` and `all_outcomes()`: specify all predictor or all target variables
- `Area, Smoothness`: specify both the `Area` and `Smoothness` variable
- `-Class`: specify everything except the `Class` variable

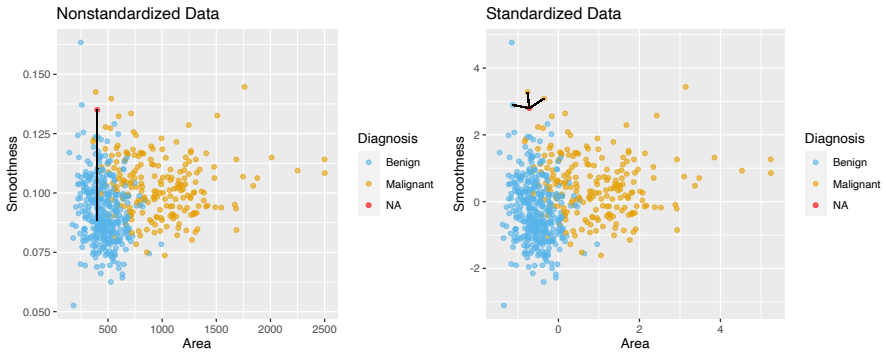
You can find a full set of all the steps and variable selection functions[7] on the recipes home page. We finally use the `bake` function to apply the recipe.

```
scaled_cancer <- bake(uc_recipe, unscaled_cancer)
scaled_cancer
```

```
## # A tibble: 569 x 3
##      Area Smoothness Class
##     <dbl>      <dbl> <fct>
##  1  0.984      1.57  M
##  2  1.91      -0.826 M
##  3  1.56       0.941 M
##  4 -0.764      3.28  M
##  5  1.82       0.280 M
##  6 -0.505      2.24  M
##  7  1.09      -0.123 M
##  8 -0.219      1.60  M
##  9 -0.384      2.20  M
## 10 -0.509      1.58  M
## # ... with 559 more rows
```

Now let's generate the two scatter plots, one for `unscaled_cancer` and one for `scaled_cancer`, and show them side-by-side. Each has the same new observation annotated with its $K = 3$ nearest neighbours:

[7] https://tidymodels.github.io/recipes/reference/index.html



In the plot for the nonstandardized original data, you can see some odd choices for the three nearest neighbours. In particular, the "neighbours" are visually well within the cloud of benign observations, and the neighbours are all nearly vertically aligned with the new observation (which is why it looks like there is only one black line on this plot). Here the computation of nearest neighbours is dominated by the much larger-scale area variable. On the right, the plot for standardized data shows a much more intuitively reasonable selection of nearest neighbours. Thus, standardizing the data can change things in an important way when we are using predictive algorithms. As a rule of thumb, standardizing your data should be a part of the preprocessing you do before any predictive modelling / analysis.

6.7.2 Balancing

Another potential issue in a data set for a classifier is *class imbalance*, i.e., when one label is much more common than another. Since classifiers like the K-nearest neighbour algorithm use the labels of nearby points to predict the label of a new point, if there are many more data points with one label overall, the algorithm is more likely to pick that label in general (even if the "pattern" of data suggests otherwise). Class imbalance is actually quite a common and important problem: from rare disease diagnosis to malicious email detection, there are many cases in which the "important" class to identify (presence of disease, malicious email) is much rarer than the "unimportant" class (no disease, normal email).

To better illustrate the problem, let's revisit the breast cancer data; except now we will remove many of the observations of malignant tumours, simulating what the data would look like if the cancer was rare. We will do this by picking only 3 observations randomly from the malignant group, and keeping all of the benign observations.

```
set.seed(3)
rare_cancer <- bind_rows(
```

```
  filter(cancer, Class == "B"),
  cancer %>% filter(Class == "M") %>% sample_n(3)
) %>%
  select(Class, Perimeter, Concavity)

rare_plot <- rare_cancer %>%
  ggplot(aes(x = Perimeter, y = Concavity, color = Class)) +
  geom_point(alpha = 0.5) +
  labs(color = "Diagnosis") +
  scale_color_manual(labels = c("Malignant", "Benign"), values = cbPalette)
rare_plot
```

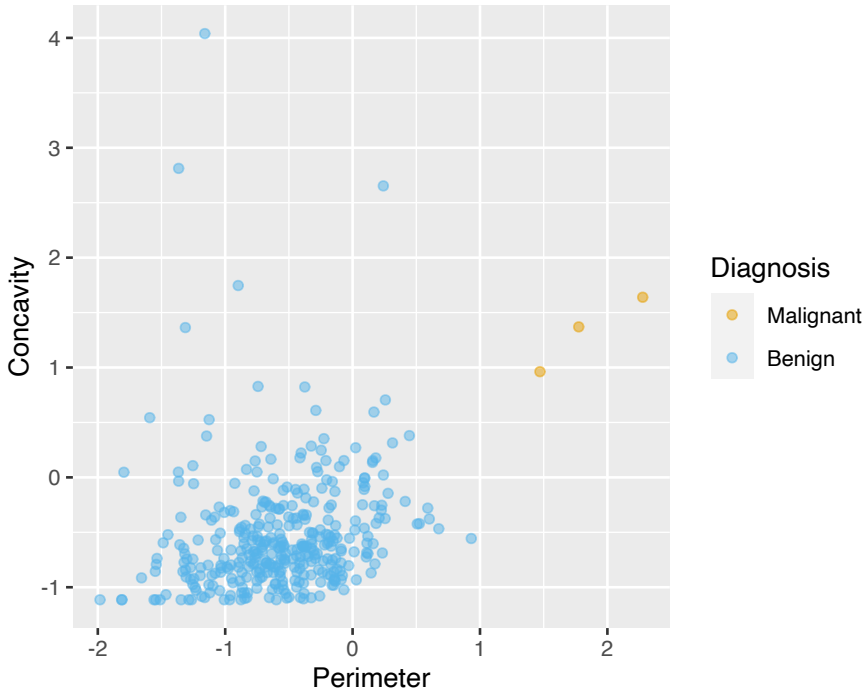
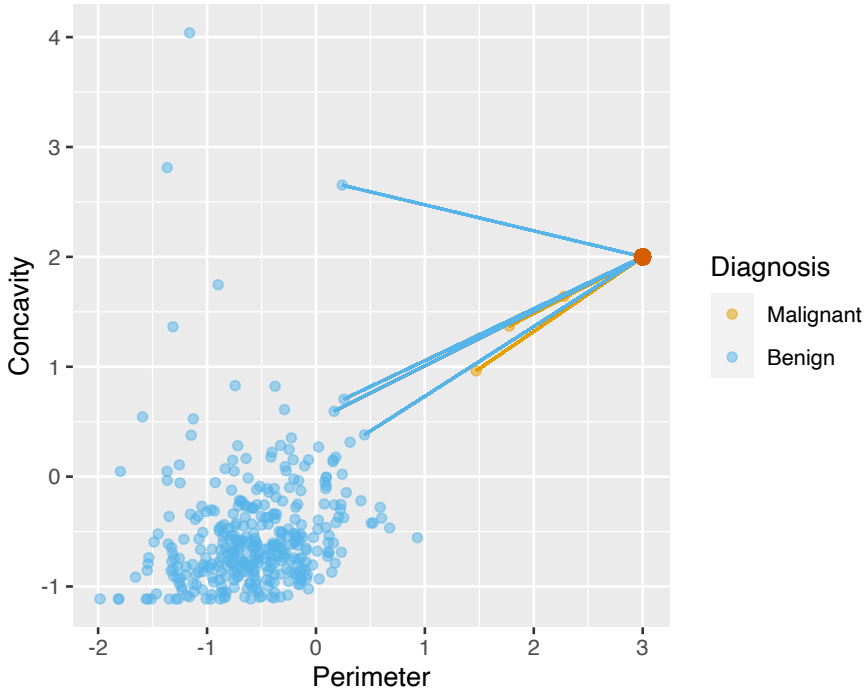


FIGURE 6.4: Imbalanced data

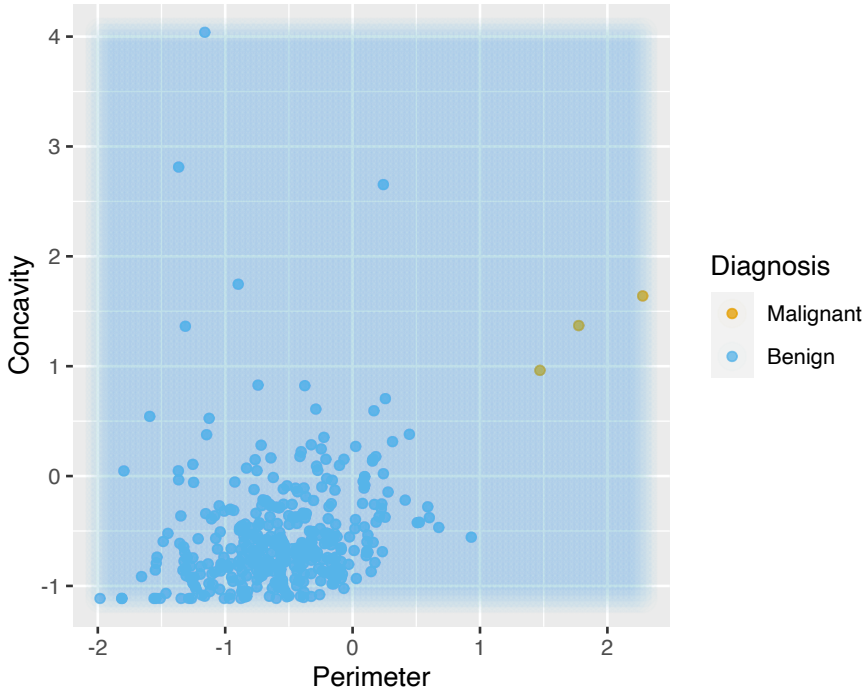
Note: You will see in the code above that we use the `set.seed` function. This is because we are using `sample_n` to artificially pick only 3 of the malignant tumour observations, which uses random sampling to choose which rows will be in the training

set. In order to make the code reproducible, we use `set.seed` to specify where the random number generator starts for this process, which then guarantees the same result, i.e., the same choice of 3 observations, each time the code is run. In general, when your code involves random numbers, if you want *the same result* each time, you should use `set.seed`; if you want a *different result* each time, you should not.

Suppose we now decided to use $K = 7$ in K-nearest neighbour classification. With only 3 observations of malignant tumours, the classifier will *always predict that the tumour is benign, no matter what its concavity and perimeter are!* This is because in a majority vote of 7 observations, at most 3 will be malignant (we only have 3 total malignant observations), so at least 4 must be benign, and the benign vote will always win. For example, look what happens for a new tumour observation that is quite close to two that were tagged as malignant:



And if we set the background colour of each area of the plot to the decision the K-nearest neighbour classifier would make, we can see that the decision is always "benign," corresponding to the blue colour:



Despite the simplicity of the problem, solving it in a statistically sound manner is actually fairly nuanced, and a careful treatment would require a lot more detail and mathematics than we will cover in this textbook. For the present purposes, it will suffice to rebalance the data by *oversampling* the rare class. In other words, we will replicate rare observations multiple times in our data set to give them more voting power in the K-nearest neighbour algorithm. In order to do this, we will add an oversampling step to the earlier `uc_recipe` recipe with the `step_upsample` function. We show below how to do this, and also use the `group_by + summarize` pattern we've seen before to see that our classes are now balanced:

```
ups_recipe <- recipe(Class ~ ., data = rare_cancer) %>%
  step_upsample(Class, over_ratio = 1, skip = FALSE) %>%
  prep()
ups_recipe
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
```

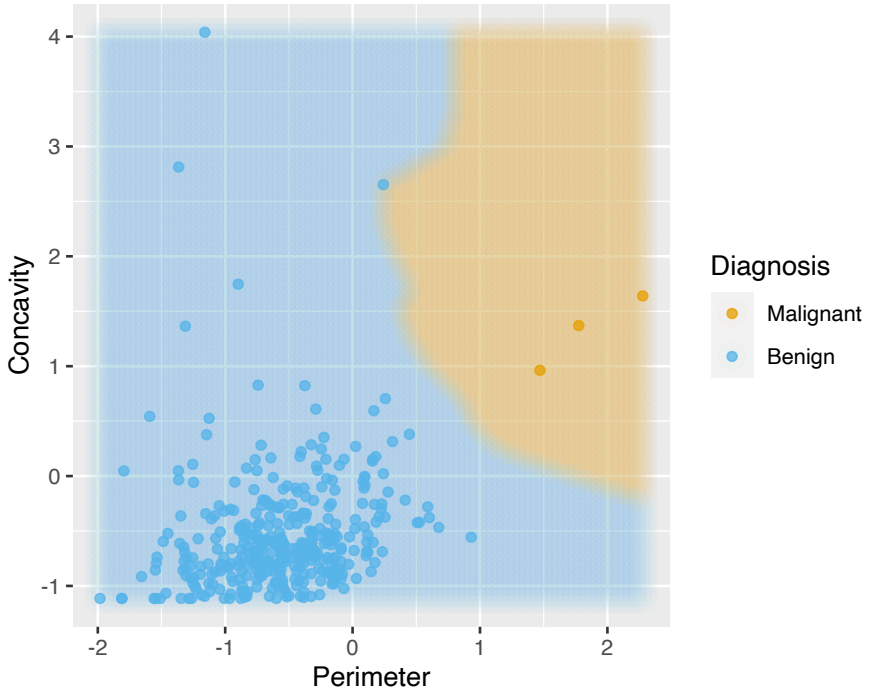
```
##  predictor          2
##
## Training data contained 360 data points and no missing data.
##
## Operations:
##
## Up-sampling based on Class [trained]
```

```
upsampled_cancer <- bake(ups_recipe, rare_cancer)

upsampled_cancer %>%
  group_by(Class) %>%
  summarize(n = n())
```

```
## # A tibble: 2 x 2
##   Class     n
##   <fct> <int>
## 1 M       357
## 2 B       357
```

Now suppose we train our K-nearest neighbour classifier with $K = 7$ on this *balanced* data. Setting the background colour of each area of our scatter plot to the decision the K-nearest neighbour classifier would make, we can see that the decision is more reasonable; when the points are close to those labelled malignant, the classifier predicts a malignant tumour, and vice versa when they are closer to the benign tumour observations:



6.8 Putting it together in a `workflow`

The `tidymodels` package collection also provides the `workflow`, a simple way to chain together multiple data analysis steps without a lot of otherwise necessary code for intermediate steps. To illustrate the whole pipeline, let's start from scratch with the `unscaled_wdbc.csv` data. First we will load the data, create a model, and specify a recipe for how the data should be preprocessed:

```
# load the unscaled cancer data and make sure the target Class variable is a factor
unscaled_cancer <- read_csv("data/unscaled_wdbc.csv") %>%
  mutate(Class = as_factor(Class))

# create the KNN model
knn_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = 7) %>%
  set_engine("kknn") %>%
  set_mode("classification")

# create the centering / scaling recipe
```

```
uc_recipe <- recipe(Class ~ Area + Smoothness, data = unscaled_cancer) %>%
  step_scale(all_predictors()) %>%
  step_center(all_predictors())
```

Note that each of these steps is exactly the same as earlier, except for one major difference: we did not use the `select` function to extract the relevant variables from the data frame, and instead simply specified the relevant variables to use via the formula `Class ~ Area + Smoothness` (instead of `Class ~ .`) in the recipe. You will also notice that we did not call `prep()` on the recipe; this is unnecssary when it is placed in a workflow.

We will now place these steps in a `workflow` using the `add_recipe` and `add_model` functions, and finally we will use the `fit` function to run the whole workflow on the `unscaled_cancer` data. Note another difference from earlier here: we do not include a formula in the `fit` function. This is again because we included the formula in the recipe, so there is no need to respecify it:

```
knn_fit <- workflow() %>%
  add_recipe(uc_recipe) %>%
  add_model(knn_spec) %>%
  fit(data = unscaled_cancer)
knn_fit
```

```
## == Workflow [trained] ==================================
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## -- Preprocessor ----------------------------------------
## 2 Recipe Steps
##
## * step_scale()
## * step_center()
##
## -- Model -----------------------------------------------
##
## Call:
## kknn::train.kknn(formula = ..y ~ ., data = data, ks = min_rows(7,    data, 5), kernel = ~"rectangular
##
## Type of response variable: nominal
## Minimal misclassification: 0.1125
## Best kernel: rectangular
## Best k: 7
```

As before, the fit object lists the function that trains the model as well as the "best" settings for the number of neighbours and weight function (for now,

these are just the values we chose manually when we created `knn_spec` above). But now the fit object also includes information about the overall workflow, including the centering and scaling preprocessing steps.

Let's visualize the predictions that this trained K-nearest neighbour model will make on new observations. Below you will see how to make the coloured prediction map plots from earlier in this chapter. The basic idea is to create a grid of synthetic new observations using the `expand.grid` function, predict the label of each, and visualize the predictions with a coloured scatter having a very high transparency (low `alpha` value) and large point radius. We include the code here as a learning challenge; see if you can figure out what each line is doing!

```
# create the grid of area/smoothness vals, and arrange in a data frame
are_grid <- seq(min(unscaled_cancer$Area), max(unscaled_cancer$Area), length.out = 100)
smo_grid <- seq(min(unscaled_cancer$Smoothness), max(unscaled_cancer$Smoothness), length.out = 100
asgrid <- as_tibble(expand.grid(Area = are_grid, Smoothness = smo_grid))

# use the fit workflow to make predictions at the grid points
knnPredGrid <- predict(knn_fit, asgrid)

# bind the predictions as a new column with the grid points
prediction_table <- bind_cols(knnPredGrid, asgrid) %>% rename(Class = .pred_class)

# plot:
# 1. the coloured scatter of the original data
# 2. the faded coloured scatter for the grid points
wkflw_plot <-
  ggplot() +
  geom_point(data = unscaled_cancer, mapping = aes(x = Area, y = Smoothness, color = Class), alpha
  geom_point(data = prediction_table, mapping = aes(x = Area, y = Smoothness, color = Class), alph
  labs(color = "Diagnosis") +
  scale_color_manual(labels = c("Malignant", "Benign"), values = cbPalette)
```

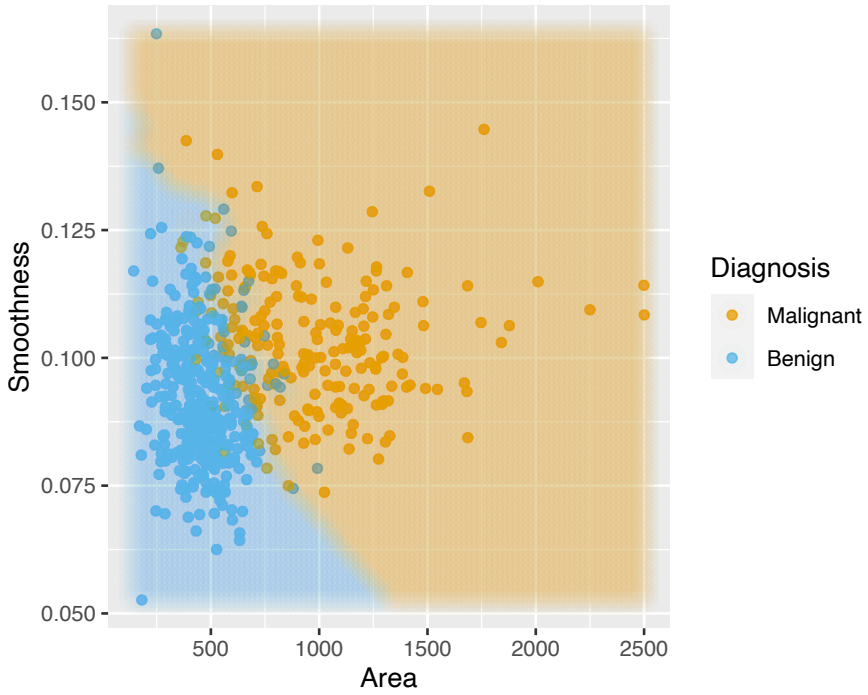


FIGURE 6.5: Scatterplot of smoothness versus area where background colour indicates the decision of the classifier

7

Classification II: evaluation & tuning

7.1 Overview

This chapter continues the introduction to predictive modelling through classification. While the previous chapter covered training and data preprocessing, this chapter focuses on how to split data, how to evaluate prediction accuracy, and how to choose model parameters to maximize performance.

7.2 Chapter learning objectives

By the end of the chapter, students will be able to:

- Describe what training, validation, and test data sets are and how they are used in classification
- Split data into training, validation, and test data sets
- Evaluate classification accuracy in R using a validation data set and appropriate metrics
- Execute cross-validation in R to choose the number of neighbours in a K-nearest neighbour classifier
- Describe advantages and disadvantages of the K-nearest neighbour classification algorithm

7.3 Evaluating accuracy

Sometimes our classifier might make the wrong prediction. A classifier does not need to be right 100% of the time to be useful, though we don't want the classifier to make too many wrong predictions. How do we measure how "good" our classifier is? Let's revisit the breast cancer images example[1] and think

[1] http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

about how our classifier will be used in practice. A biopsy will be performed on a *new* patient's tumour, the resulting image will be analyzed, and the classifier will be asked to decide whether the tumour is benign or malignant. The key word here is *new*: our classifier is "good" if it provides accurate predictions on data *not seen during training*. But then how can we evaluate our classifier without having to visit the hospital to collect more tumour images?

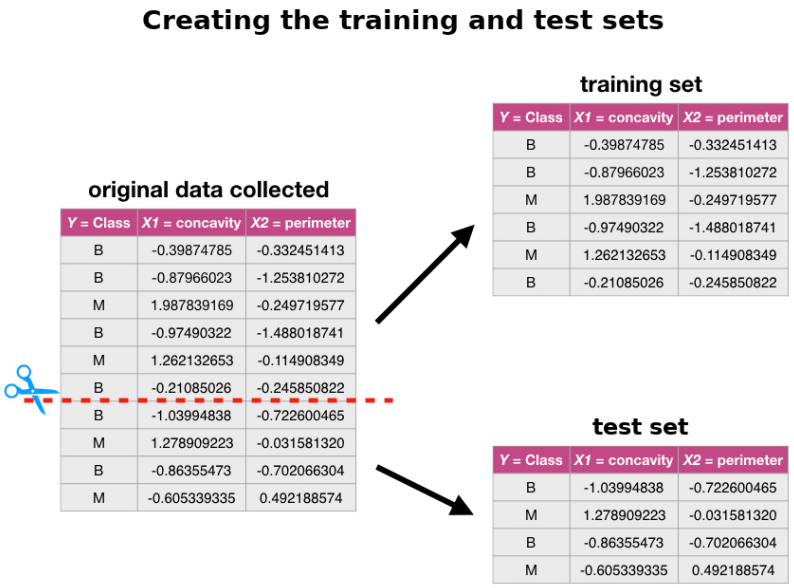
The trick is to split up the data set into a **training set** and **test set**, and only show the classifier the **training set** when building the classifier. Then to evaluate the accuracy of the classifier, we can use it to predict the labels (which we know) in the **test set**. If our predictions match the true labels for the observations in the **test set** very well, then we have some confidence that our classifier might also do a good job of predicting the class labels for new observations that we do not have the class labels for.

> Note: if there were a golden rule of machine learning, it might be this: *you cannot use the test data to build the model!* If you do, the model gets to "see" the test data in advance, making it look more accurate than it really is. Imagine how bad it would be to overestimate your classifier's accuracy when predicting whether a patient's tumour is malignant or benign!

How exactly can we assess how well our predictions match the true labels for the observations in the test set? One way we can do this is to calculate the **prediction accuracy**. This is the fraction of examples for which the classifier made the correct prediction. To calculate this we divide the number of correct predictions by the number of predictions made. Other measures for how well our classifier performed include *precision* and *recall*; these will not be discussed here, but you will encounter them in other more advanced courses on this topic. This process is illustrated below:

In R, we can use the `tidymodels` library collection not only to perform K-nearest neighbour classification, but also to assess how well our classification worked. Let's start by loading the necessary libraries, reading in the breast cancer data from the previous chapter, and making a quick scatter plot visualization of tumour cell concavity versus smoothness coloured by diagnosis.

```
# load packages
library(tidyverse)
library(tidymodels)
```



Y = Class	X1 = concavity	X2 = perimeter
B	-0.39874785	-0.332451413
B	-0.87966023	-1.253810272
M	1.987839169	-0.249719577
B	-0.97490322	-1.488018741
M	1.262132653	-0.114908349
B	-0.21085026	-0.245850822
B	-1.03994838	-0.722600465
M	1.278909223	-0.031581320
B	-0.86355473	-0.702066304
M	-0.605339335	0.492188574

FIGURE 7.1: Splitting the data into training and testing sets

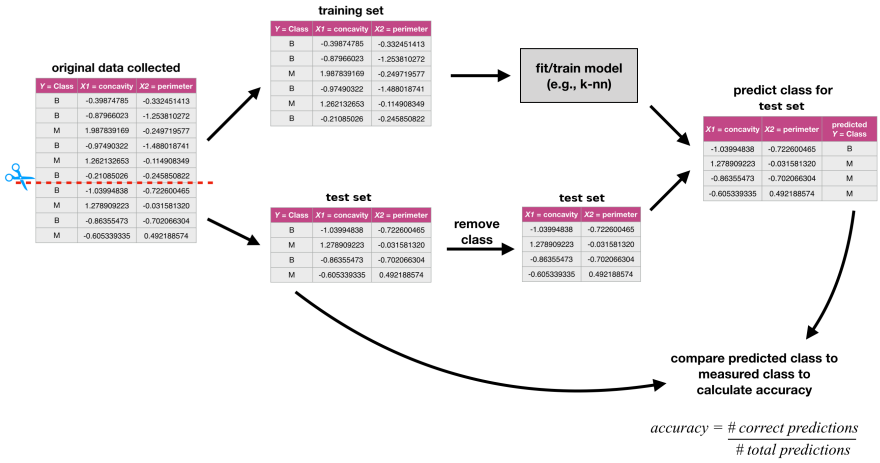


FIGURE 7.2: Process for splitting the data and finding the prediction accuracy

```
# load data
cancer <- read_csv("data/unscaled_wdbc.csv") %>%
  mutate(Class = as_factor(Class)) # convert the character Class variable to the factor datatype

# colour palette
cbPalette <- c("#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC79A7", "#9999

# create scatter plot of tumour cell concavity versus smoothness,
# labelling the points be diagnosis class
perim_concav <- cancer %>%
  ggplot(aes(x = Smoothness, y = Concavity, color = Class)) +
  geom_point(alpha = 0.5) +
  labs(color = "Diagnosis") +
  scale_color_manual(labels = c("Malignant", "Benign"), values = cbPalette)

perim_concav
```

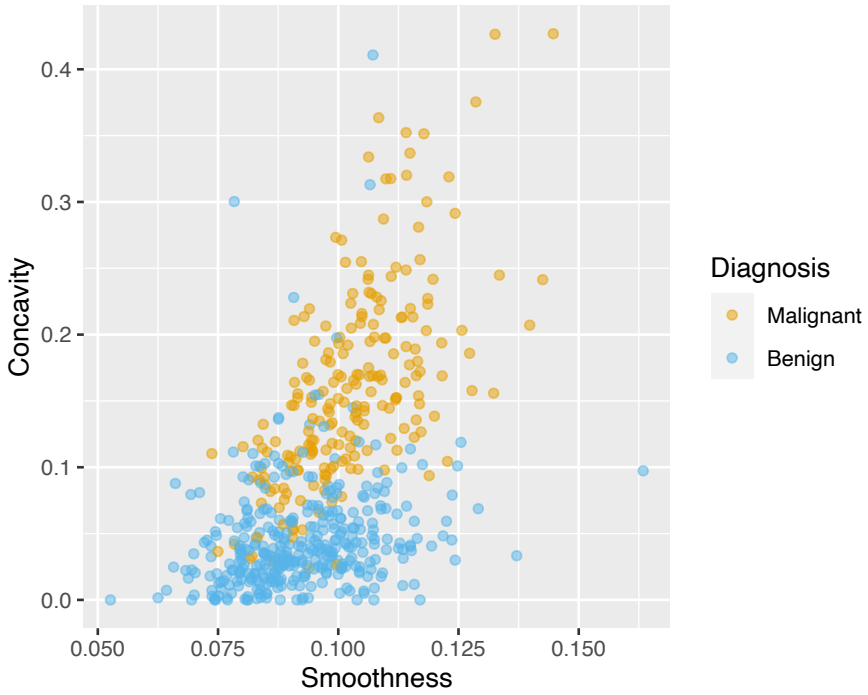


FIGURE 7.3: Scatterplot of tumour cell concavity versus smoothness coloured by diagnosis label

1. Create the train / test split

Once we have decided on a predictive question to answer and done some preliminary exploration, the very next thing to do is to split the data into the training and test sets. Typically, the training set is between 50 - 100% of the data, while the test set is the remaining 0 - 50%; the intuition is that you want to trade off between training an accurate model (by using a larger training data set) and getting an accurate evaluation of its performance (by using a larger test data set). Here, we will use 75% of the data for training, and 25% for testing. To do this we will use the `initial_split` function, specifying that `prop = 0.75` and the target variable is `Class`:

```
set.seed(1)
cancer_split <- initial_split(cancer, prop = 0.75, strata = Class)
cancer_train <- training(cancer_split)
cancer_test <- testing(cancer_split)
```

> Note: You will see in the code above that we use the `set.seed` function again, as discussed in the previous chapter. In this case it is because `initial_split` uses random sampling to choose which rows will be in the training set. Since we want our code to be reproducible and generate the same train/test split each time it is run, we use `set.seed`.

```
glimpse(cancer_train)
```

```
## Rows: 427
## Columns: 12
## $ ID                <dbl> 842302, 842517, 84300903...
## $ Class             <fct> M, M, M, M, M, M, M, M, ...
## $ Radius            <dbl> 17.990, 20.570, 19.690, ...
## $ Texture           <dbl> 10.38, 17.77, 21.25, 20....
## $ Perimeter         <dbl> 122.80, 132.90, 130.00, ...
## $ Area              <dbl> 1001.0, 1326.0, 1203.0, ...
## $ Smoothness        <dbl> 0.11840, 0.08474, 0.1096...
## $ Compactness       <dbl> 0.27760, 0.07864, 0.1599...
## $ Concavity         <dbl> 0.30010, 0.08690, 0.1974...
## $ Concave_Points    <dbl> 0.14710, 0.07017, 0.1279...
## $ Symmetry          <dbl> 0.2419, 0.1812, 0.2069, ...
```

```
## $ Fractal_Dimension <dbl> 0.07871, 0.05667, 0.0599...
```

```
glimpse(cancer_test)
```

```
## Rows: 142
## Columns: 12
## $ ID                <dbl> 844981, 84799002, 848406...
## $ Class             <fct> M, M, M, M, B, M, M, M, ...
## $ Radius            <dbl> 13.000, 14.540, 14.680, ...
## $ Texture           <dbl> 21.82, 27.54, 20.13, 22....
## $ Perimeter         <dbl> 87.50, 96.73, 94.74, 130...
## $ Area              <dbl> 519.8, 658.8, 684.5, 126...
## $ Smoothness        <dbl> 0.12730, 0.11390, 0.0986...
## $ Compactness       <dbl> 0.19320, 0.15950, 0.0720...
## $ Concavity         <dbl> 0.18590, 0.16390, 0.0739...
## $ Concave_Points    <dbl> 0.093530, 0.073640, 0.05...
## $ Symmetry          <dbl> 0.2350, 0.2303, 0.1586, ...
## $ Fractal_Dimension <dbl> 0.07389, 0.07077, 0.0592...
```

We can see from `glimpse` in the code above that the training set contains 427 observations, while the test set contains 142 observations. This corresponds to a train / test split of 75% / 25%, as desired.

2. Pre-process the data

As we mentioned last chapter, K-NN is sensitive to the scale of the predictors, and so we should perform some preprocessing to standardize them. An additional consideration we need to take when doing this is that we should create the standardization preprocessor using **only the training data**. This ensures that our test data does not influence any aspect of our model training. Once we have created the standardization preprocessor, we can then apply it separately to both the training and test data sets.

Fortunately, the `recipe` framework from `tidymodels` makes it simple to handle this properly. Below we construct and prepare the recipe using only the training data (due to `data = cancer_train` in the first line).

```
cancer_recipe <- recipe(Class ~ Smoothness + Concavity, data = cancer_train) %>%
  step_scale(all_predictors()) %>%
  step_center(all_predictors())
```

3. Train the classifier

Now that we have split our original data set into training and test sets, we can create our K-nearest neighbour classifier with only the training set using the technique we learned in the previous chapter. For now, we will just choose

the number K of neighbours to be 3, and use concavity and smoothness as the predictors.

```
set.seed(1)
knn_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = 3) %>%
  set_engine("kknn") %>%
  set_mode("classification")

knn_fit <- workflow() %>%
  add_recipe(cancer_recipe) %>%
  add_model(knn_spec) %>%
  fit(data = cancer_train)

knn_fit
```

```
## == Workflow [trained] ==================================
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## -- Preprocessor ----------------------------------------
## 2 Recipe Steps
##
## * step_scale()
## * step_center()
##
## -- Model -----------------------------------------------
##
## Call:
## kknn::train.kknn(formula = ..y ~ ., data = data, ks = min_rows(3,    data, 5), kernel = ~"rectangular
##
## Type of response variable: nominal
## Minimal misclassification: 0.1265
## Best kernel: rectangular
## Best k: 3
```

Note: Here again you see the `set.seed` function. In the K-nearest neighbour algorithm, there is a tie for the majority neighbour class, the winner is randomly selected. Although there is no chance of a tie when K is odd (here $K = 3$), it is possible that the code may be changed in the future to have an even value of K. Thus, to prevent potential issues with reproducibility, we

have set the seed. Note that in your own code, you only have to set the seed once at the beginning of your analysis.

---

4. Predict the labels in the test set

Now that we have a K-nearest neighbour classifier object, we can use it to predict the class labels for our test set. We use the `bind_cols` to add the column of predictions to the original test data, creating the `cancer_test_predictions` data frame. The `Class` variable contains the true diagnoses, while the `.pred_class` contains the predicted diagnoses from the model.

```
cancer_test_predictions <- predict(knn_fit, cancer_test) %>%
  bind_cols(cancer_test)
cancer_test_predictions
```

```
## # A tibble: 142 x 13
##    .pred_class     ID Class Radius Texture Perimeter
##    <fct>        <dbl> <fct>  <dbl>   <dbl>     <dbl>
##  1 M           8.45e5 M       13      21.8      87.5
##  2 M           8.48e7 M       14.5    27.5      96.7
##  3 B           8.48e5 M       14.7    20.1      94.7
##  4 M           8.49e5 M       19.8    22.2     130
##  5 B           8.51e6 B       13.5    14.4      87.5
##  6 M           8.51e6 M       15.3    14.3     102.
##  7 M           8.53e5 M       18.6    25.1     125.
##  8 M           8.54e5 M       19.3    26.5     128.
##  9 B           8.55e5 M       13.4    21.6      86.2
## 10 M           8.56e5 M       13.3    20.3      87.3
## # ... with 132 more rows, and 7 more variables:
## #   Area <dbl>, Smoothness <dbl>, Compactness <dbl>,
## #   Concavity <dbl>, Concave_Points <dbl>,
## #   Symmetry <dbl>, Fractal_Dimension <dbl>
```

5. Compute the accuracy

Finally we can assess our classifier's accuracy. To do this we use the `metrics` function from `tidymodels` to get the statistics about the quality of our model, specifying the `truth` and `estimate` arguments:

```
cancer_test_predictions %>%
  metrics(truth = Class, estimate = .pred_class)
```

```
## # A tibble: 2 x 3
```

```
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.880
## 2 kap      binary         0.741
```

This shows that the accuracy of the classifier on the test data was 88%. We can also look at the *confusion matrix* for the classifier, which shows the table of predicted labels and correct labels, using the `conf_mat` function:

```
cancer_test_predictions %>%
  conf_mat(truth = Class, estimate = .pred_class)
```

```
##           Truth
## Prediction  M  B
##          M 43  7
##          B 10 82
```

This says that the classifier labelled 43+82 = 125 observations correctly, 10 observations as benign when they were truly malignant, and 7 observations as malignant when they were truly benign.

7.4 Tuning the classifier

The vast majority of predictive models in statistics and machine learning have *parameters* that you have to pick. For example, in the K-nearest neighbour classification algorithm we have been using in the past two chapters, we have had to pick the number of neighbours K for the class vote. Is it possible to make this selection, i.e., *tune* the model, in a principled way? Ideally what we want is to somehow maximize the performance of our classifier on data *it hasn't seen yet*. So we will play the same trick we did before when evaluating our classifier: we'll split our **overall training data set** further into two subsets, called the **training set** and **validation set**. We will use the newly-named **training set** for building the classifier, and the **validation set** for evaluating it! Then we will try different values of the parameter K and pick the one that yields the highest accuracy.

> **Remember:** *don't touch the test set during the tuning process. Tuning is a part of model training!*

7.4.1 Cross-validation

There is an important detail to mention about the process of tuning: we can, if we want to, split our overall training data up in multiple different ways, train and evaluate a classifier for each split, and then choose the parameter based on ***all*** of the different results. If we just split our overall training data *once*, our best parameter choice will depend strongly on whatever data was lucky enough to end up in the validation set. Perhaps using multiple different train / validation splits, we'll get a better estimate of accuracy, which will lead to a better choice of the number of neighbours K for the overall set of training data.

> **Note:** you might be wondering why we can't we use the multiple splits to test our final classifier after tuning is done. This is simply because at the end of the day, we will produce a single classifier using our overall training data. If we do multiple train / test splits, we will end up with multiple classifiers, each with their own accuracy evaluated on different test data.

Let's investigate this idea in R! In particular, we will use different seed values in the `set.seed` function to generate five different train / validation splits of our overall training data, train five different K-nearest neighbour models, and evaluate their accuracy.

```
accuracies <- c()
for (i in 1:5) {
  set.seed(i) # makes the random selection of rows reproducible

  # create the 25/75 split of the training data into training and validation
  cancer_split <- initial_split(cancer_train, prop = 0.75, strata = Class)
  cancer_subtrain <- training(cancer_split)
  cancer_validation <- testing(cancer_split)

  # recreate the standardization recipe from before (since it must be based on the training data)
  cancer_recipe <- recipe(Class ~ Smoothness + Concavity, data = cancer_subtrain) %>%
    step_scale(all_predictors()) %>%
    step_center(all_predictors())

  # fit the knn model (we can reuse the old knn_spec model from before)
  knn_fit <- workflow() %>%
```

```
    add_recipe(cancer_recipe) %>%
    add_model(knn_spec) %>%
    fit(data = cancer_subtrain)

  # get predictions on the validation data
  validation_predicted <- predict(knn_fit, cancer_validation) %>%
    bind_cols(cancer_validation)

  # compute the accuracy
  acc <- validation_predicted %>%
    metrics(truth = Class, estimate = .pred_class) %>%
    filter(.metric == "accuracy") %>%
    select(.estimate) %>%
    pull()
  accuracies <- append(accuracies, acc)
}
accuracies
```

```
## [1] 0.9151 0.8679 0.8491 0.8962 0.9151
```

With five different shuffles of the data, we get five different values for accuracy. None of these is necessarily "more correct" than any other; they're just five estimates of the true, underlying accuracy of our classifier built using our overall training data. We can combine the estimates by taking their average (here 0.8887) to try to get a single assessment of our classifier's accuracy; this has the effect of reducing the influence of any one (un)lucky validation set on the estimate.

In practice, we don't use random splits, but rather use a more structured splitting procedure so that each observation in the data set is used in a validation set only a single time. The name for this strategy is called **cross-validation**. In **cross-validation**, we split our **overall training data** into C evenly-sized chunks, and then iteratively use 1 chunk as the **validation set** and combine the remaining $C-1$ chunks as the **training set**:

In the picture above, $C = 5$ different chunks of the data set are used, resulting in 5 different choices for the **validation set**; we call this *5-fold* cross-validation. To do 5-fold cross-validation in R with `tidymodels`, we use another function: `vfold_cv`. This function splits our training data into `v` folds automatically:

```
cancer_vfold <- vfold_cv(cancer_train, v = 5, strata = Class)
cancer_vfold
```

```
## #  5-fold cross-validation using stratification
```

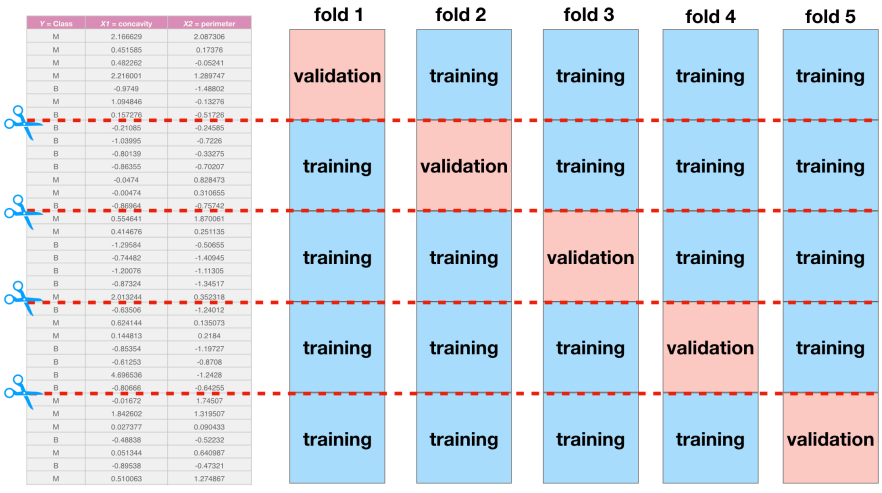


FIGURE 7.4: 5-fold cross validation

```
## # A tibble: 5 x 2
##   splits           id
##   <list>           <chr>
## 1 <split [341/86]> Fold1
## 2 <split [341/86]> Fold2
## 3 <split [341/86]> Fold3
## 4 <split [342/85]> Fold4
## 5 <split [343/84]> Fold5
```

Then, when we create our data analysis workflow, we use the `fit_resamples` function instead of the `fit` function for training. This runs cross-validation on each train/validation split.

Note: we set the seed when we call `train` not only because of the potential for ties, but also because we are doing cross-validation. Cross-validation uses a random process to select how to partition the training data.

```
set.seed(1)

# recreate the standardization recipe from before (since it must be based on the training data)
cancer_recipe <- recipe(Class ~ Smoothness + Concavity, data = cancer_train) %>%
  step_scale(all_predictors()) %>%
  step_center(all_predictors())

# fit the knn model (we can reuse the old knn_spec model from before)
knn_fit <- workflow() %>%
  add_recipe(cancer_recipe) %>%
  add_model(knn_spec) %>%
  fit_resamples(resamples = cancer_vfold)

knn_fit
```

```
## # Resampling results
## # 5-fold cross-validation using stratification
## # A tibble: 5 x 4
##   splits          id    .metrics         .notes
##   <list>          <chr> <list>           <list>
## 1 <split [341/86~ Fold1 <tibble [2 x 4]> <tibble [0 x ~
## 2 <split [341/86~ Fold2 <tibble [2 x 4]> <tibble [0 x ~
## 3 <split [341/86~ Fold3 <tibble [2 x 4]> <tibble [0 x ~
## 4 <split [342/85~ Fold4 <tibble [2 x 4]> <tibble [0 x ~
## 5 <split [343/84~ Fold5 <tibble [2 x 4]> <tibble [0 x ~
```

The `collect_metrics` function is used to aggregate the mean and *standard error* of the classifier's validation accuracy across the folds. The standard error is a measure of how uncertain we are in the mean value. A detailed treatment of this is beyond the scope of this chapter; but roughly, if your estimated mean (that the `collect_metrics` function gives you) is 0.88 and standard error is 0.02, you can expect the *true* average accuracy of the classifier to be somewhere roughly between 0.86 and 0.90 (although it may fall outside this range).

```
knn_fit %>% collect_metrics()
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.883     5  0.0189 Preprocessor~
## 2 roc_auc  binary     0.923     5  0.0104 Preprocessor~
```

We can choose any number of folds, and typically the more we use the better our accuracy estimate will be (lower standard error). However, we are limited by computational power: the more folds we choose, the more computation it

takes, and hence the more time it takes to run the analysis. So when you do cross-validation, you need to consider the size of the data, and the speed of the algorithm (e.g., K-nearest neighbour) and the speed of your computer. In practice, this is a trial and error process, but typically C is chosen to be either 5 or 10. Here we show how the standard error decreases when we use 10-fold cross validation rather than 5-fold:

```
cancer_vfold <- vfold_cv(cancer_train, v = 10, strata = Class)

workflow() %>%
  add_recipe(cancer_recipe) %>%
  add_model(knn_spec) %>%
  fit_resamples(resamples = cancer_vfold) %>%
  collect_metrics()
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.873    10  0.0184 Preprocessor~
## 2 roc_auc  binary     0.922    10  0.0147 Preprocessor~
```

7.4.2 Parameter value selection

Using 5- and 10-fold cross-validation, we have estimated that the prediction accuracy of our classifier is somewhere around 88%. Whether 88% is good or not depends entirely on the downstream application of the data analysis. In the present situation, we are trying to predict a tumour diagnosis, with expensive, damaging chemo/radiation therapy or patient death as potential consequences of misprediction. Hence, we'd like to do better than 88% for this application.

In order to improve our classifier, we have one choice of parameter: the number of neighbours, K. Since cross-validation helps us evaluate the accuracy of our classifier, we can use cross-validation to calculate an accuracy for each value of K in a reasonable range, and then pick the value of K that gives us the best accuracy. The `tidymodels` package collection provides a very simple syntax for tuning models: each parameter in the model to be tuned should be specified as `tune()` in the model specification rather than given a particular value.

```
knn_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = tune()) %>%
  set_engine("kknn") %>%
  set_mode("classification")
```

Then instead of using `fit` or `fit_resamples`, we will use the `tune_grid` function to fit the model for each value in a range of parameter values. Here the `grid`

= 10 argument specifies that the tuning should try 10 values of the number of neighbours K when tuning. We set the seed prior to tuning to ensure results are reproducible:

```
set.seed(1)
knn_results <- workflow() %>%
  add_recipe(cancer_recipe) %>%
  add_model(knn_spec) %>%
  tune_grid(resamples = cancer_vfold, grid = 10) %>%
  collect_metrics()
knn_results
```

```
## # A tibble: 20 x 7
##    neighbors .metric .estimator  mean     n std_err
##        <int> <chr>   <chr>      <dbl> <int>   <dbl>
##  1         2 accura~ binary     0.848    10  0.0146
##  2         2 roc_auc binary     0.897    10  0.0149
##  3         3 accura~ binary     0.873    10  0.0184
##  4         3 roc_auc binary     0.922    10  0.0147
##  5         5 accura~ binary     0.880    10  0.0164
##  6         5 roc_auc binary     0.928    10  0.0136
##  7         6 accura~ binary     0.880    10  0.0164
##  8         6 roc_auc binary     0.932    10  0.0144
##  9         7 accura~ binary     0.889    10  0.0167
## 10         7 roc_auc binary     0.934    10  0.0135
## 11         9 accura~ binary     0.878    10  0.0154
## 12         9 roc_auc binary     0.940    10  0.0125
## 13        10 accura~ binary     0.878    10  0.0154
## 14        10 roc_auc binary     0.943    10  0.0123
## 15        12 accura~ binary     0.882    10  0.0154
## 16        12 roc_auc binary     0.941    10  0.0124
## 17        13 accura~ binary     0.875    10  0.0136
## 18        13 roc_auc binary     0.942    10  0.0125
## 19        15 accura~ binary     0.873    10  0.0132
## 20        15 roc_auc binary     0.949    10  0.0127
## # ... with 1 more variable: .config <chr>
```

We can select the best value of the number of neighbours (i.e., the one that results in the highest classifier accuracy estimate) by plotting the accuracy versus K:

```
accuracies <- knn_results %>%
  filter(.metric == "accuracy")
```

```
accuracy_vs_k <- ggplot(accuracies, aes(x = neighbors, y = mean)) +
  geom_point() +
  geom_line() +
  labs(x = "Neighbors", y = "Accuracy Estimate")
accuracy_vs_k
```

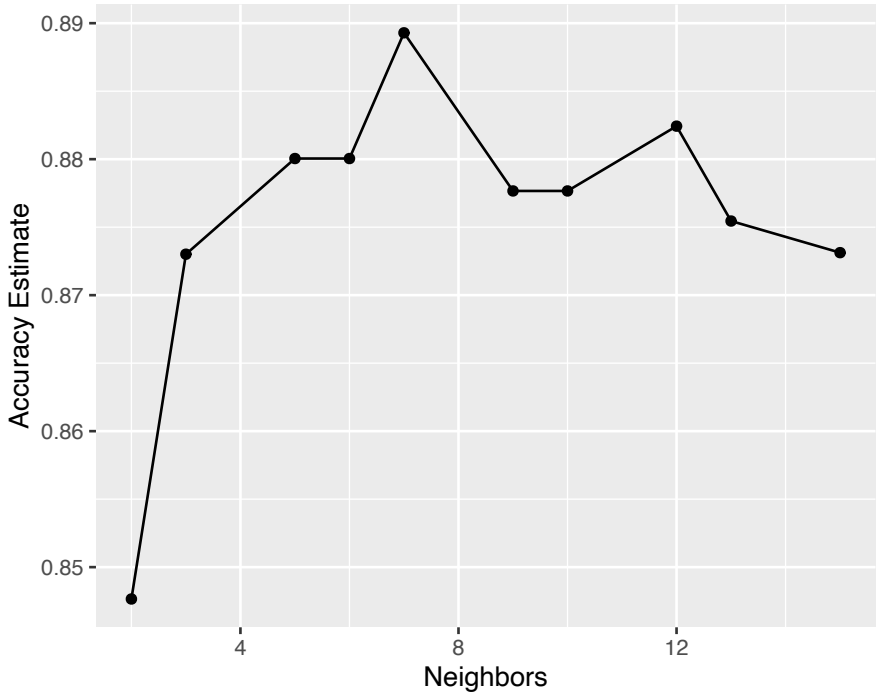


FIGURE 7.5: Plot of accuracy estimate versus number of neighbours

This visualization suggests that $K = 7$ provides the highest accuracy. But as you can see, there is no exact or perfect answer here; any selection between $K = 3$ and 13 would be reasonably justified, as all of these differ in classifier accuracy by less than 1%. Remember: the values you see on this plot are *estimates* of the true accuracy of our classifier. Although the $K = 7$ value is higher than the others on this plot, that doesn't mean the classifier is actually more accurate with this parameter value! Generally, when selecting K (and other parameters for other predictive models), we are looking for a value where:

- we get roughly optimal accuracy, so that our model will likely be accurate
- changing the value to a nearby one (e.g. from $K = 7$ to 6 or 8) doesn't

decrease accuracy too much, so that our choice is reliable in the presence of uncertainty
- the cost of training the model is not prohibitive (e.g., in our situation, if K is too large, predicting becomes expensive!)

7.4.3 Under/overfitting

To build a bit more intuition, what happens if we keep increasing the number of neighbours K? In fact, the accuracy actually starts to decrease! Rather than setting `grid = 10` and letting `tidymodels` decide what values of K to try, let's specify the values explicitly by creating a data frame with a `neighbors` variable. Take a look as the plot below as we vary K from 1 to almost the number of observations in the data set:

```
set.seed(1)
k_lots <- tibble(neighbors = seq(from = 1, to = 385, by = 10))
knn_results <- workflow() %>%
  add_recipe(cancer_recipe) %>%
  add_model(knn_spec) %>%
  tune_grid(resamples = cancer_vfold, grid = k_lots) %>%
  collect_metrics()

accuracies <- knn_results %>%
  filter(.metric == "accuracy")

accuracy_vs_k_lots <- ggplot(accuracies, aes(x = neighbors, y = mean)) +
  geom_point() +
  geom_line() +
  labs(x = "Neighbors", y = "Accuracy Estimate")
accuracy_vs_k_lots
```

Underfitting: What is actually happening to our classifier that causes this? As we increase the number of neighbours, more and more of the training observations (and those that are farther and farther away from the point) get a "say" in what the class of a new observation is. This causes a sort of "averaging effect" to take place, making the boundary between where our classifier would predict a tumour to be malignant versus benign to smooth out and become *simpler.* If you take this to the extreme, setting K to the total training data set size, then the classifier will always predict the same label regardless of what the new observation looks like. In general, if the model *isn't influenced enough* by the training data, it is said to **underfit** the data.

Overfitting: In contrast, when we decrease the number of neighbours, each individual data point has a stronger and stronger vote regarding nearby points. Since the data themselves are noisy, this causes a more "jagged" boundary cor-

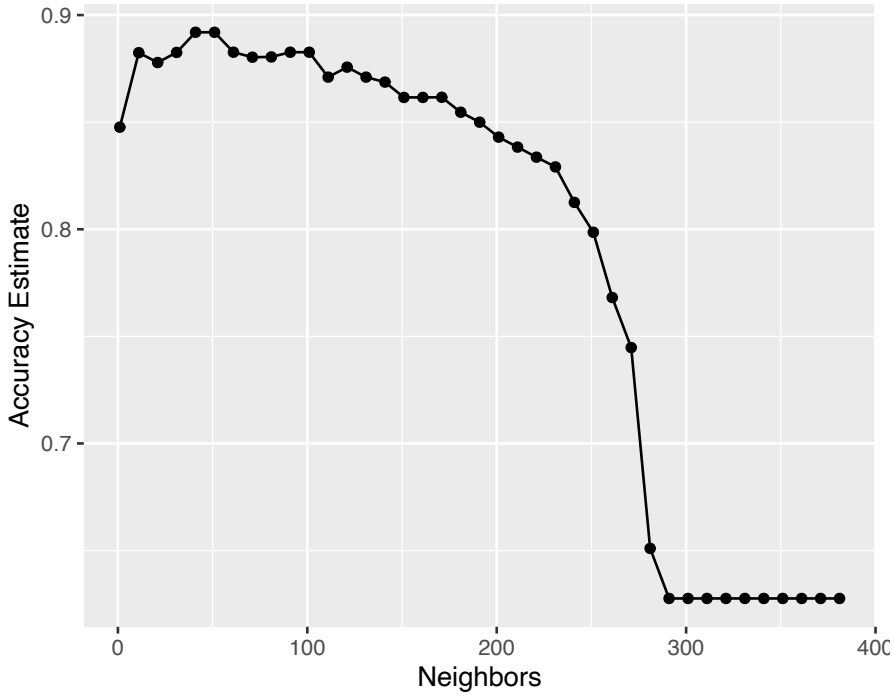
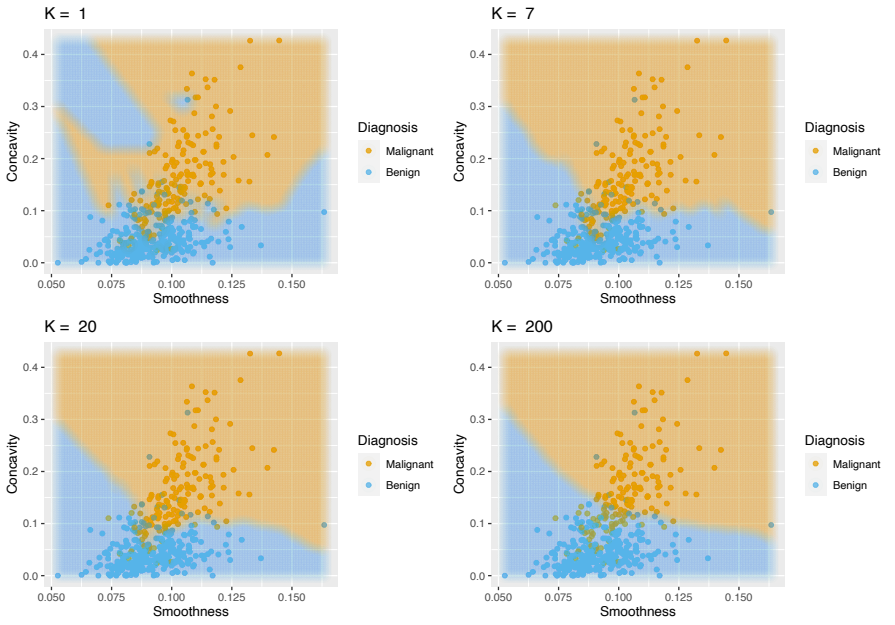


FIGURE 7.6: Plot of accuracy estimate versus number of neighbours for many K values

responding to a *less simple* model. If you take this case to the extreme, setting $K = 1$, then the classifier is essentially just matching each new observation to its closest neighbour in the training data set. This is just as problematic as the large K case, because the classifier becomes unreliable on new data: if we had a different training set, the predictions would be completely different. In general, if the model *is influenced too much* by the training data, it is said to **overfit** the data.

You can see this effect in the plots below as we vary the number of neighbours K in (1, 7, 20, 200):



7.5 Splitting data

Shuffling: When we split the data into train, test, and validation sets, we make the assumption that there is no order to our originally collected data set. However, if we think that there might be some order to the original data set, then we can randomly shuffle the data before splitting it. The `tidymodels` function `initial_split` and `vfold_cv` functions do this for us.

Stratification: If the data are imbalanced, we also need to be extra careful about splitting the data to ensure that enough of each class ends up in each of the train, validation, and test partitions. The `strata` argument in the `initial_split` and `vfold_cv` functions handles this for us too.

7.6 Summary

Classification algorithms use one or more quantitative variables to predict the value of a third, categorical variable. The K-nearest neighbour algorithm in particular does this by first finding the K points in the training data nearest to the new observation, and then returning the majority class vote from

those training observations. We can evaluate a classifier by splitting the data randomly into a training and test data set, using the training set to build the classifier, and using the test set to estimate its accuracy. To tune the classifier (e.g., select the K in K-nearest neighbours), we maximize accuracy estimates from cross-validation.

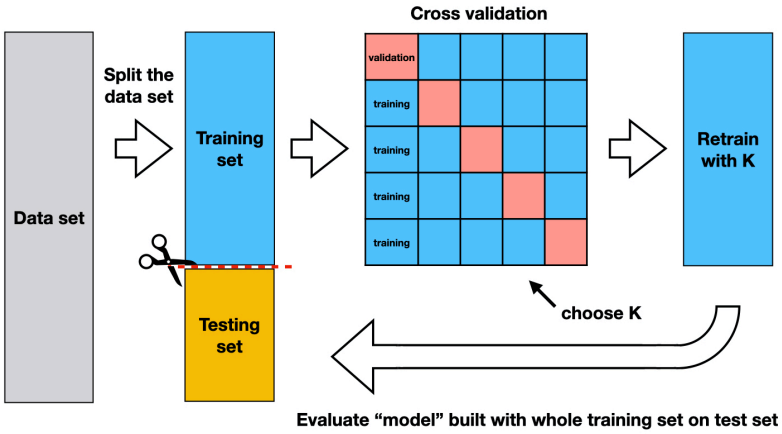


FIGURE 7.7: Overview of K-nn classification

The overall workflow for performing K-nearest neighbour classification using `tidymodels` is as follows:

1. Use the `initial_split` function to split the data into a training and test set. Set the `strata` argument to the target variable. Put the test set aside for now.
2. Use the `vfold_cv` function to split up the training data for cross validation.
3. Create a `recipe` that specifies the target and predictor variables, as well as preprocessing steps for all variables. Pass the training data as the `data` argument of the recipe.
4. Create a `nearest_neighbors` model specification, with `neighbors = tune()`.
5. Add the recipe and model specification to a `workflow()`, and use the `tune_grid` function on the train/validation splits to estimate the classifier accuracy for a range of K values.
6. Pick a value of K that yields a high accuracy estimate that doesn't change much if you change K to a nearby value.
7. Make a new model specification for the best parameter value, and retrain the classifier using the `fit` function.

8. Evaluate the estimated accuracy of the classifier on the test set using the `predict` function.

Strengths:

1. Simple and easy to understand
2. No assumptions about what the data must look like
3. Works easily for binary (two-class) and multi-class (> 2 classes) classification problems

Weaknesses:

1. As data gets bigger and bigger, K-nearest neighbour gets slower and slower, quite quickly
2. Does not perform well with a large number of predictors
3. Does not perform well when classes are imbalanced (when many more observations are in one of the classes compared to the others)

8

Regression I: K-nearest neighbours

8.1 Overview

This chapter will provide an introduction to regression through K-nearest neighbours (K-NN) in a predictive context, focusing primarily on the case where there is a single predictor and single response variable of interest. The chapter concludes with an example of K-nearest neighbours regression with multiple predictors.

8.2 Chapter learning objectives

By the end of the chapter, students will be able to:

- Recognize situations where a simple regression analysis would be appropriate for making predictions.
- Explain the K-nearest neighbour (K-NN) regression algorithm and describe how it differs from K-NN classification.
- Interpret the output of a K-NN regression.
- In a dataset with two or more variables, perform K-nearest neighbour regression in R using a `tidymodels` workflow
- Execute cross-validation in R to choose the number of neighbours.
- Evaluate K-NN regression prediction accuracy in R using a test data set and an appropriate metric (*e.g.*, root means square prediction error).
- In the context of K-NN regression, compare and contrast goodness of fit and prediction properties (namely RMSE vs RMSPE).
- Describe advantages and disadvantages of the K-nearest neighbour regression approach.

8.3 Regression

Regression, like classification, is a predictive problem setting where we want to use past information to predict future observations. But in the case of regression, the goal is to predict numerical values instead of class labels. For example, we could try to use the number of hours a person spends on exercise each week to predict whether they would qualify for the annual Boston marathon (*classification*) or to predict their race time itself (*regression*). As another example, we could try to use the size of a house to predict whether it sold for more than $500,000 (*classification*) or to predict its sale price itself (*regression*). We will use K-nearest neighbours to explore this question in the rest of this chapter, using a real estate data set from Sacremento, California.

8.4 Sacremento real estate example

Let's start by loading the packages we need and doing some preliminary exploratory analysis. The Sacramento real estate data set we will study in this chapter was originally reported in the Sacramento Bee[1], but we have provided it with this repository as a stable source for the data.

```
library(tidyverse)
library(tidymodels)
library(gridExtra)

sacramento <- read_csv("data/sacramento.csv")
sacramento
```

```
## # A tibble: 932 x 9
##    city    zip    beds baths  sqft type  price latitude
##    <chr>   <chr> <dbl> <dbl> <dbl> <chr> <dbl>    <dbl>
##  1 SACRAM~ z958~     2     1   836 Resi~ 59222     38.6
##  2 SACRAM~ z958~     3     1  1167 Resi~ 68212     38.5
##  3 SACRAM~ z958~     2     1   796 Resi~ 68880     38.6
##  4 SACRAM~ z958~     2     1   852 Resi~ 69307     38.6
##  5 SACRAM~ z958~     2     1   797 Resi~ 81900     38.5
##  6 SACRAM~ z958~     3     1  1122 Condo 89921     38.7
##  7 SACRAM~ z958~     3     2  1104 Resi~ 90895     38.7
##  8 SACRAM~ z958~     3     1  1177 Resi~ 91002     38.5
```

[1] https://support.spatialkey.com/spatialkey-sample-csv-data/

```
##  9 RANCHO~ z956~     2     2   941 Condo 94905     38.6
## 10 RIO_LI~ z956~     3     2  1146 Resi~ 98937     38.7
## # ... with 922 more rows, and 1 more variable:
## #   longitude <dbl>
```

The purpose of this exercise is to understand whether we can we use house size to predict house sale price in the Sacramento, CA area. The columns in this data that we are interested in are `sqft` (house size, in livable square feet) and `price` (house price, in US dollars (USD). The first step is to visualize the data as a scatter plot where we place the predictor/explanatory variable (house size) on the x-axis, and we place the target/response variable that we want to predict (price) on the y-axis:

```
eda <- ggplot(sacramento, aes(x = sqft, y = price)) +
  geom_point(alpha = 0.4) +
  xlab("House size (square footage)") +
  ylab("Price (USD)") +
  scale_y_continuous(labels = dollar_format())
eda
```

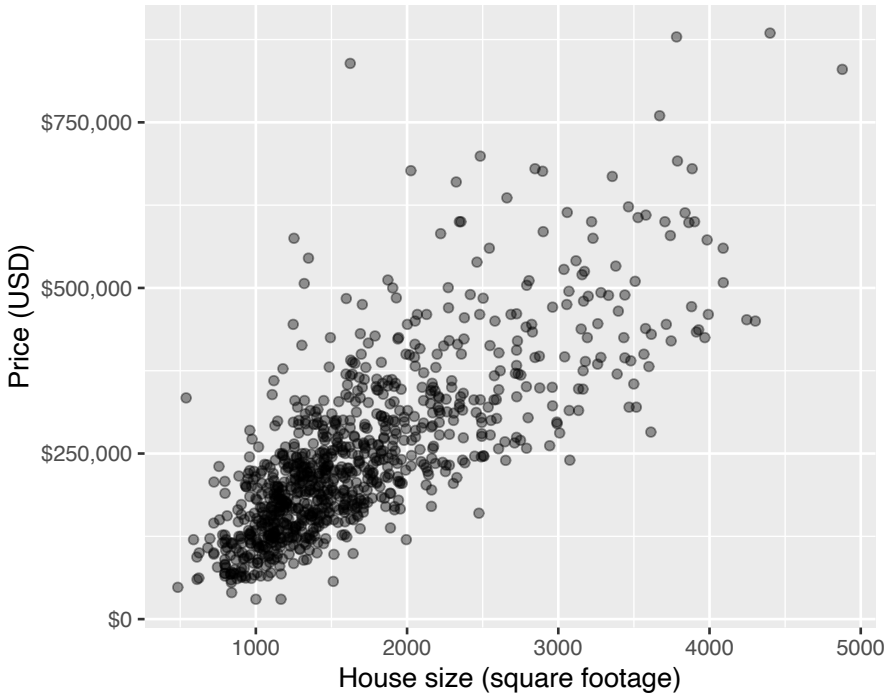


FIGURE 8.1: Scatter plot of price (USD) versus house size (square footage)

Based on the visualization above, we can see that in Sacramento, CA, as the size of a house increases, so does its sale price. Thus, we can reason that we may be able to use the size of a not-yet-sold house (for which we don't know the sale price) to predict its final sale price.

8.5 K-nearest neighbours regression

Much like in the case of classification, we can use a K-nearest neighbours-based approach in regression to make predictions. Let's take a small sample of the data above and walk through how K-nearest neighbours (knn) works in a regression context before we dive in to creating our model and assessing how well it predicts house price. This subsample is taken to allow us to illustrate the mechanics of K-NN regression with a few data points; later in this chapter we will use all the data.

To take a small random sample of size 30, we'll use the function `sample_n`. This function takes two arguments:

1. `tbl` (a data frame-like object to sample from)
2. `size` (the number of observations/rows to be randomly selected/sampled)

```
set.seed(1234)
small_sacramento <- sample_n(sacramento, size = 30)
```

Next let's say we come across a 2,000 square-foot house in Sacramento we are interested in purchasing, with an advertised list price of $350,000. Should we offer to pay the asking price for this house, or is it overpriced and we should offer less? Absent any other information, we can get a sense for a good answer to this question by using the data we have to predict the sale price given the sale prices we have already observed. But in the plot below, we have no observations of a house of size *exactly* 2000 square feet. How can we predict the price?

```
small_plot <- ggplot(small_sacramento, aes(x = sqft, y = price)) +
  geom_point() +
  xlab("House size (square footage)") +
  ylab("Price (USD)") +
  scale_y_continuous(labels = dollar_format()) +
  geom_vline(xintercept = 2000, linetype = "dotted")
small_plot
```

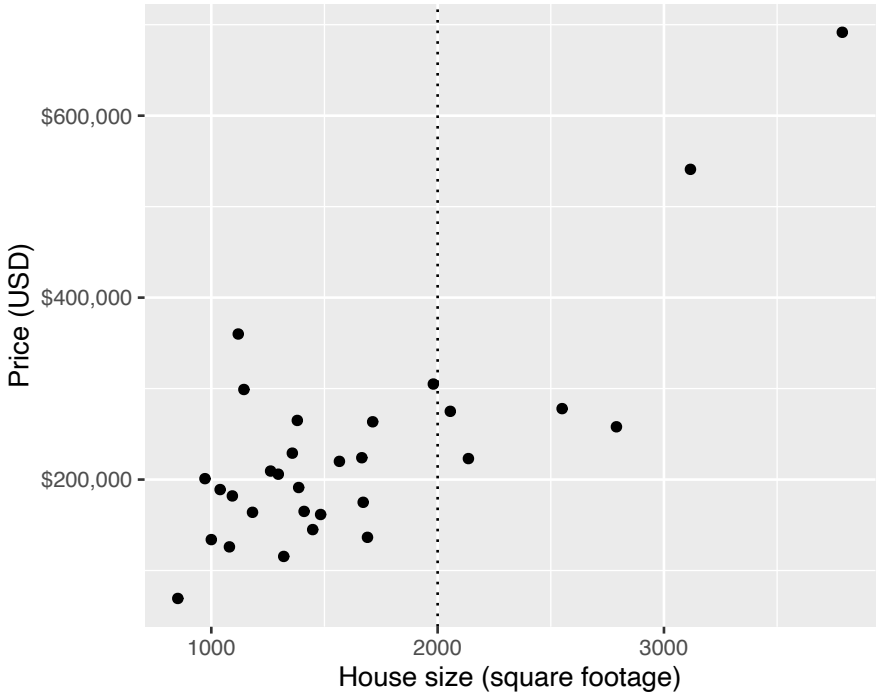


FIGURE 8.2: Scatter plot of price (USD) versus house size (square footage) with vertical line indicating 2000 square feet on x-axis

We will employ the same intuition from the classification chapter, and use the neighbouring points to the new point of interest to suggest/predict what its price should be. For the example above, we find and label the 5 nearest neighbours to our observation of a house that is 2000 square feet:

```
nearest_neighbours <- small_sacramento %>%
  mutate(diff = abs(2000 - sqft)) %>%
  arrange(diff) %>%
  slice(1:5) #subset the first 5 rows
nearest_neighbours
```

```
## # A tibble: 5 x 10
##   city  zip    beds baths  sqft type   price latitude
##   <chr> <chr> <dbl> <dbl> <dbl> <chr>  <dbl>    <dbl>
## 1 GOLD~ z956~     3     2  1981 Resi~ 305000     38.6
## 2 ELK_~ z957~     4     2  2056 Resi~ 275000     38.4
## 3 ELK_~ z956~     5     3  2136 Resi~ 223058     38.4
## 4 RANC~ z957~     4     2  1713 Resi~ 263500     38.6
```

```
## 5 RIO_~ z956~     2     2  1690 Resi~ 136500     38.7
## # ... with 2 more variables: longitude <dbl>,
## #   diff <dbl>
```

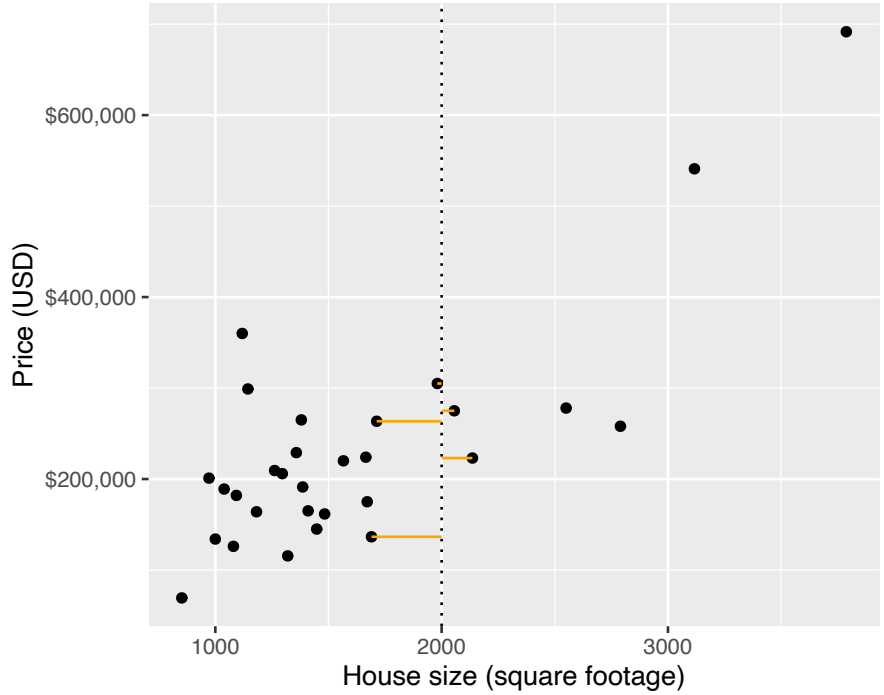


FIGURE 8.3: Scatter plot of price (USD) versus house size (square footage) with lines to 5 nearest neighbours

Now that we have the 5 nearest neighbours (in terms of house size) to our new 2,000 square-foot house of interest, we can use their values to predict a selling price for the new home. Specifically, we can take the mean (or average) of these 5 values as our predicted value.

```
prediction <- nearest_neighbours %>%
  summarise(predicted = mean(price))
prediction
```

```
## # A tibble: 1 x 1
##   predicted
##       <dbl>
## 1   240612.
```

Our predicted price is $240612 (shown as a red point above), which is much

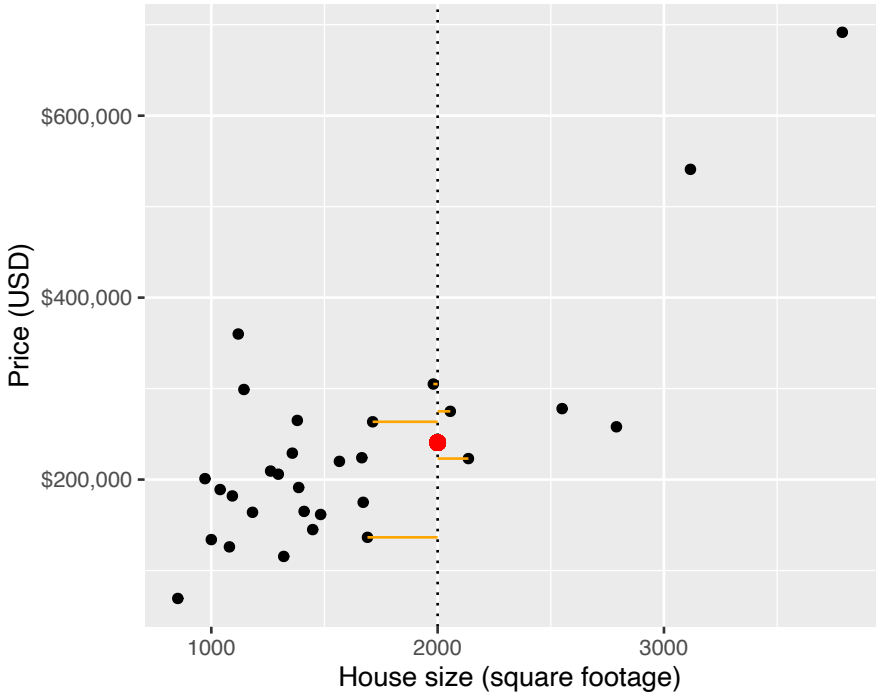


FIGURE 8.4: Scatter plot of price (USD) versus house size (square footage) with predicted price for a 2000 square-foot house based on 5 nearest neighbours represented as a red dot

less than $350,000; perhaps we might want to offer less than the list price at which the house is advertised. But this is only the very beginning of the story. We still have all the same unanswered questions here with K-NN regression that we had with K-NN classification: which K do we choose, and is our model any good at making predictions? In the next few sections, we will address these questions in the context of K-NN regression.

8.6 Training, evaluating, and tuning the model

As usual, we must start by putting some test data away in a lock box that we will come back to only after we choose our final model. Let's take care of that now. Note that for the remainder of the chapter we'll be working with the entire Sacramento data set, as opposed to the smaller sample of 30 points above.

```
set.seed(1234)
sacramento_split <- initial_split(sacramento, prop = 0.6, strata = price)
sacramento_train <- training(sacramento_split)
sacramento_test <- testing(sacramento_split)
```

Next, we'll use cross-validation to choose K. In K-NN classification, we used accuracy to see how well our predictions matched the true labels. Here in the context of K-NN regression we will use root mean square prediction error (RMSPE) instead. The mathematical formula for calculating RMSPE is:

$$\text{RMSPE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Where:

- n is the number of observations
- y_i is the observed value for the i^{th} observation
- $\hat{y}_i$ is the forcasted/predicted value for the i^{th} observation

A key feature of the formula for RMPSE is the squared difference between the observed target/response variable value, y, and the prediction target/response variable value, $\hat{y}_i$, for each observation (from 1 to i). If the predictions are very close to the true values, then RMSPE will be small. If, on the other-hand, the predictions are very different to the true values, then RMSPE will be quite large. When we use cross validation, we will choose the K that gives us the smallest RMSPE.

RMSPE versus RMSE When using many code packages (`tidymodels` included), the evaluation output we will get to assess the prediction quality of our K-NN regression models is labelled "RMSE", or "root mean squared error". Why is this so, and why not just RMSPE? In statistics, we try to be very precise with our language to indicate whether we are calculating the prediction error on the training data (*in-sample* prediction) versus on the testing data (*out-of-sample* prediction). When predicting and evaluating prediction quality on the training data, we say RMSE. By contrast, when predicting and evaluating prediction quality on the testing or validation data, we say RMSPE. The equation for calculating RMSE and RMSPE is exactly the same; all that changes is whether the ys are training or testing data. But many people just use RMSE for both, and rely on

context to denote which data the root mean squared error is being calculated on.

Now that we know how we can assess how well our model predicts a numerical value, let's use R to perform cross-validation and to choose the optimal K. First, we will create a model specification for K-nearest neighbours regression, as well as a recipe for preprocessing our data. Note that we use `set_mode("regression")` now in the model specification to denote a regression problem, as opposed to the classification problems from the previous chapters. Note also that we include standardization in our preprocessing to build good habits, but since we only have one predictor it is technically not necessary; there is no risk of comparing two predictors of different scales.

```
sacr_recipe <- recipe(price ~ sqft, data = sacramento_train) %>%
  step_scale(all_predictors()) %>%
  step_center(all_predictors())

sacr_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = tune()) %>%
  set_engine("kknn") %>%
  set_mode("regression")

sacr_vfold <- vfold_cv(sacramento_train, v = 5, strata = price)

sacr_wkflw <- workflow() %>%
  add_recipe(sacr_recipe) %>%
  add_model(sacr_spec)
sacr_wkflw
```

```
## == Workflow ============================================
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## -- Preprocessor ----------------------------------------
## 2 Recipe Steps
##
## * step_scale()
## * step_center()
##
## -- Model -----------------------------------------------
## K-Nearest Neighbor Model Specification (regression)
##
## Main Arguments:
```

```
##   neighbors = tune()
##   weight_func = rectangular
##
## Computational engine: kknn
```

The major difference you can see in the above workflow compared to previous chapters is that we are running regression rather than classification. The fact that we use `set_mode("regression")` essentially tells `tidymodels` that we need to use different metrics (RMSPE, not accuracy) for tuning and evaluation. You can see this in the following code, which tunes the model and returns the RMSPE for each number of neighbours.

```
gridvals <- tibble(neighbors = seq(1, 200))

sacr_results <- sacr_wkflw %>%
  tune_grid(resamples = sacr_vfold, grid = gridvals) %>%
  collect_metrics()

# show all the results
sacr_results
```

```
## # A tibble: 400 x 7
##    neighbors .metric .estimator    mean     n std_err
##        <int> <chr>   <chr>        <dbl> <int>   <dbl>
##  1         1 rmse    standard   1.17e+5     5 7.19e+3
##  2         1 rsq     standard   3.72e-1     5 4.63e-2
##  3         2 rmse    standard   1.01e+5     5 5.87e+3
##  4         2 rsq     standard   4.53e-1     5 3.86e-2
##  5         3 rmse    standard   9.67e+4     5 4.21e+3
##  6         3 rsq     standard   4.83e-1     5 2.77e-2
##  7         4 rmse    standard   9.39e+4     5 3.50e+3
##  8         4 rsq     standard   5.03e-1     5 2.98e-2
##  9         5 rmse    standard   8.96e+4     5 3.77e+3
## 10         5 rsq     standard   5.43e-1     5 3.03e-2
## # ... with 390 more rows, and 1 more variable:
## #   .config <chr>
```

We take the *minimum* RMSPE to find the best setting for the number of neighbours:

```
# show only the row of minimum RMSPE
sacr_min <- sacr_results %>%
  filter(.metric == "rmse") %>%
  filter(mean == min(mean))
sacr_min
```

```
## # A tibble: 1 x 7
##   neighbors .metric .estimator   mean     n std_err
##       <int> <chr>   <chr>       <dbl> <int>   <dbl>
## 1        14 rmse    standard   84356.     5   4050.
## # ... with 1 more variable: .config <chr>
```

Here we can see that the smallest RMSPE occurs when $K = 14$.

8.7 Underfitting and overfitting

Similar to the setting of classification, by setting the number of neighbours to be too small or too large, we cause the RMSPE to increase:

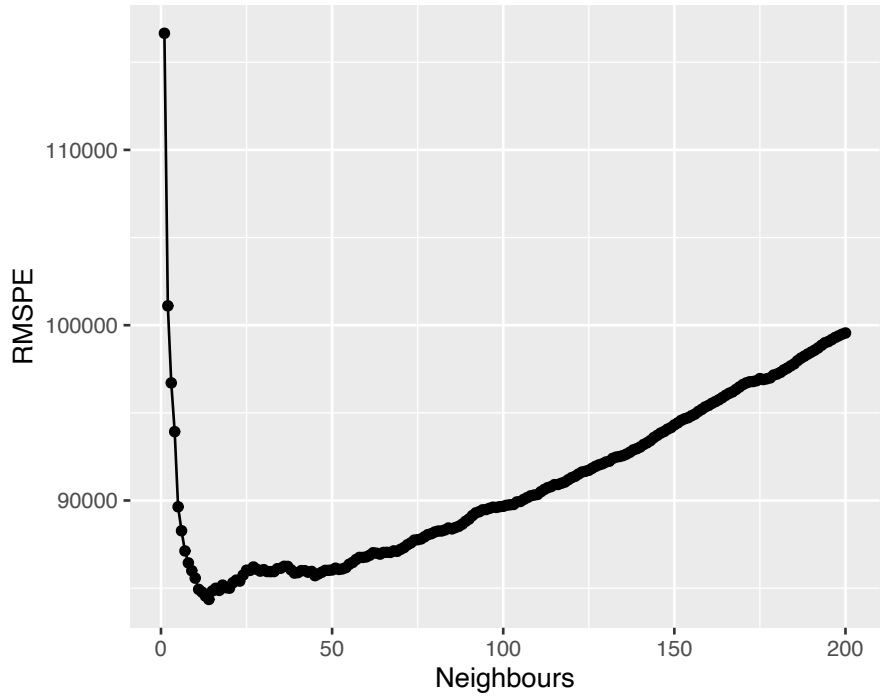


FIGURE 8.5: Effect of the number of neighbours on the RMSPE

What is happening here? To visualize the effect of different settings of K on the regression model, we will plot the predicted values for house price from our K-NN regression models for 6 different values for K. For each model, we predict a price for every possible home size across the range of home sizes

we observed in the data set (here 500 to 4250 square feet) and we plot the predicted prices as a blue line:

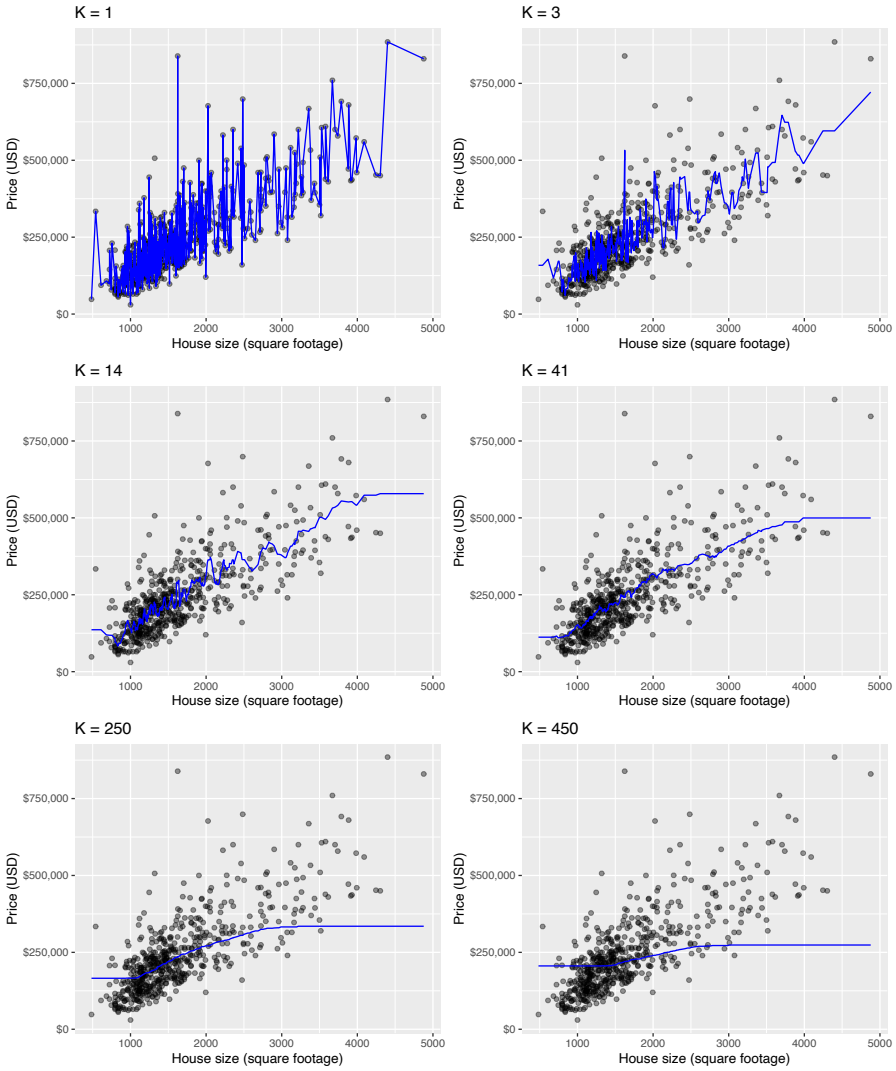


FIGURE 8.6: Predicted values for house price (represented as a blue line) from K-NN regression models for six different values for K

Based on the plots above, we see that when $K = 1$, the blue line runs perfectly through almost all of our training observations. This happens because our predicted values for a given region depend on just a single observation. A model like this has high variance and low bias (intuitively, it provides unreliable predictions). It has high variance because the flexible blue line follows

the training observations very closely, and if we were to change any one of the training observation data points we would change the flexible blue line quite a lot. This means that the blue line matches the data we happen to have in this training data set, however, if we were to collect another training data set from the Sacramento real estate market it likely wouldn't match those observations as well. Another term that we use to collectively describe this phenomenon is *overfitting*.

What about the plot where K is quite large, say $K = 450$, for example? When $K = 450$ for this data set, the blue line is extremely smooth, and almost flat. This happens because our predicted values for a given x value (here home size), depend on many many (450) neighbouring observations. A model like this has low variance and high bias (intuitively, it provides very reliable, but generally very inaccurate predictions). It has low variance because the smooth, inflexible blue line does not follow the training observations very closely, and if we were to change any one of the training observation data points it really wouldn't affect the shape of the smooth blue line at all. This means that although the blue line matches does not match the data we happen to have in this particular training data set perfectly, if we were to collect another training data set from the Sacramento real estate market it likely would match those observations equally as well as it matches those in this training data set. Another term that we use to collectively describe this kind of model is *underfitting*.

Ideally, what we want is neither of the two situations discussed above. Instead, we would like a model with low variance (so that it will transfer/generalize well to other data sets, and isn't too dependent on the observations that happen to be in the training set) **and** low bias (where the model does not completely ignore our training data). If we explore the other values for K, in particular $K = 14$ (as suggested by cross-validation), we can see it has a lower bias than our model with a very high K (e.g., 450), and thus the model/predicted values better match the actual observed values than the high K model. Additionally, it has lower variance than our model with a very low K (e.g., 1) and thus it should better transer/generalize to other data sets compared to the low K model. All of this is similar to how the choice of K affects K-NN classification (discussed in the previous chapter).

8.8 Evaluating on the test set

To assess how well our model might do at predicting on unseen data, we will assess its RMSPE on the test data. To do this, we will first re-train our K-NN regression model on the entire training data set, using $K = 14$ neighbours. Then we will use `predict` to make predictions on the test data, and use the `metrics` function again to compute the summary of regression

quality. Because we specify that we are performing regression in `set_mode`, the `metrics` function knows to output a quality summary related to regression, and not, say, classification.

```
set.seed(1234)
kmin <- sacr_min %>% pull(neighbors)
sacr_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = kmin) %>%
  set_engine("kknn") %>%
  set_mode("regression")

sacr_fit <- workflow() %>%
  add_recipe(sacr_recipe) %>%
  add_model(sacr_spec) %>%
  fit(data = sacramento_train)

sacr_summary <- sacr_fit %>%
  predict(sacramento_test) %>%
  bind_cols(sacramento_test) %>%
  metrics(truth = price, estimate = .pred)

sacr_summary
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard   87737.
## 2 rsq     standard       0.546
## 3 mae     standard   65400.
```

Our final model's test error as assessed by RMSPE is 87737. But what does this RMSPE score mean? When we calculated test set prediction accuracy in K-NN classification, the highest possible value was 1 and the lowest possible value was 0. If we got a value close to 1, our model was "good;" and otherwise, the model was "not good." What about RMSPE? Unfortunately there is no default scale for RMSPE. Instead, it is measured in the units of the target/response variable, and so it is a bit hard to interpret. For now, let's consider this approach to thinking about RMSPE from our testing data set: as long as its not significantly worse than the cross-validation RMSPE of our best model, then we can say that we're not doing too much worse on the test data than we did on the training data. So the model appears to be generalizing well to a new data set it has never seen before. In future courses on statistical/machine learning, you will learn more about how to interpret RMSPE from testing data and other ways to assess models.

Finally, what does our model look like when we predict across all possible house sizes we might encounter in the Sacramento area? We plotted it above

where we explored how k affects K-NN regression, but we show it again now, along with the code that generated it:

```
set.seed(1234)
sacr_preds <- sacr_fit %>%
  predict(sacramento_train) %>%
  bind_cols(sacramento_train)

plot_final <- ggplot(sacr_preds, aes(x = sqft, y = price)) +
  geom_point(alpha = 0.4) +
  xlab("House size (square footage)") +
  ylab("Price (USD)") +
  scale_y_continuous(labels = dollar_format()) +
  geom_line(data = sacr_preds, aes(x = sqft, y = .pred), color = "blue") +
  ggtitle(paste0("K = ", kmin))

plot_final
```

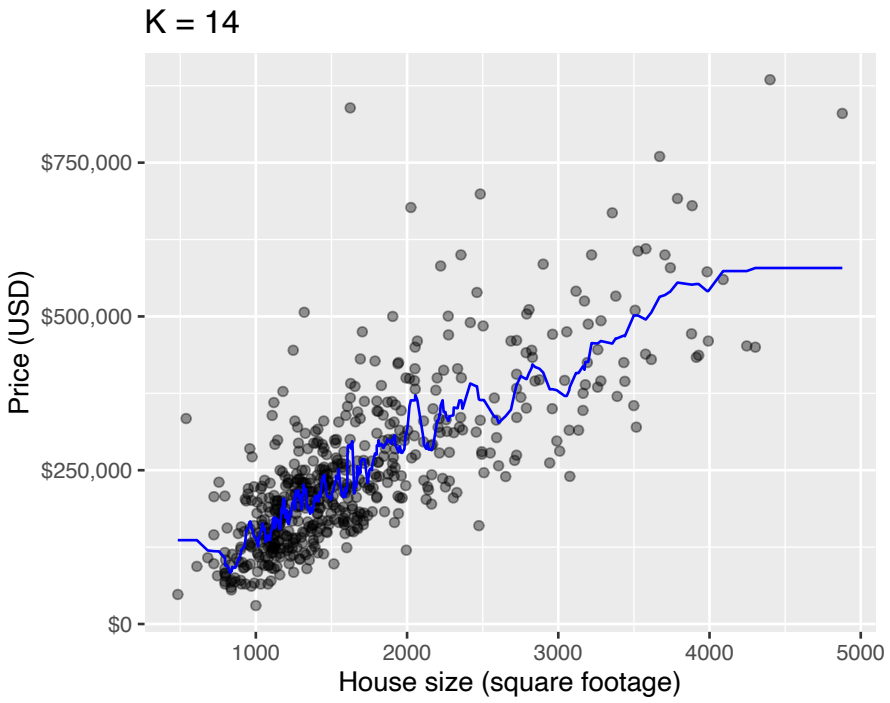


FIGURE 8.7: Predicted values for house price (represented as a blue line) for K-NN regression model with $K = 14$

8.9 Strengths and limitations of K-NN regression

As with K-NN classification (or any prediction algorithm for that manner), K-NN regression has both strengths and weaknesses. Some are listed here:

Strengths of K-NN regression

1. Simple and easy to understand
2. No assumptions about what the data must look like
3. Works well with non-linear relationships (i.e., if the relationship is not a straight line)

Limitations of K-NN regression

1. As data gets bigger and bigger, K-NN gets slower and slower, quite quickly 2. Does not perform well with a large number of predictors unless the size of the training set is exponentially larger
2. Does not predict well beyond the range of values input in your training data

8.10 Multivariate K-NN regression

As in K-NN classification, in K-NN regression we can have multiple predictors. When we have multiple predictors in K-NN regression, we have the same concern regarding the scale of the predictors. This is because once again, predictions are made by identifying the K observations that are nearest to the new point we want to predict, and any variables that are on a large scale will have a much larger effect than variables on a small scale. Since the `recipe` we built above scales and centers all predictor variables, this is handled for us.

We will now demonstrate a multivariate K-NN regression analysis of the Sacramento real estate data using `tidymodels`. This time we will use house size (measured in square feet) as well as number of bedrooms as our predictors, and continue to use house sale price as our outcome/target variable that we are trying to predict.

It is always a good practice to do exploratory data analysis, such as visualizing the data, before we start modeling the data. Thus the first thing we will do is use ggpairs (from the `GGally` package) to plot all the variables we are interested in using in our analyses:

```
library(GGally)
plot_pairs <- sacramento %>%
  select(price, sqft, beds) %>%
  ggpairs()
plot_pairs
```

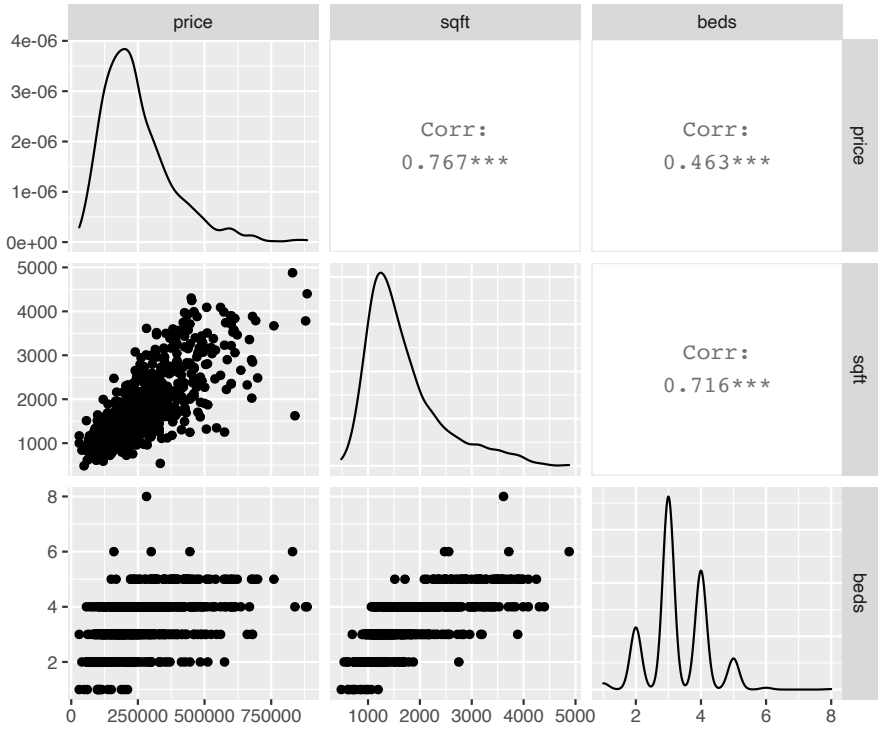


FIGURE 8.8: Scatterplots of each pair of variables (price, house size, and the number of bedrooms) are displayed in the figure's bottom left corner. Correlation coefficients are shown in the top right corner. The distributions of price, house size and number of bedrooms are each shown along the diagonal.

From this we can see that generally, as both house size and number of bedrooms increase, so does price. Does adding the number of bedrooms to our model improve our ability to predict house price? To answer that question, we will have to come up with the test error for a K-NN regression model using house size and number of bedrooms, and then we can compare it to the test error for the model we previously came up with that only used house size to see if it is smaller (decreased test error indicates increased prediction quality). Let's do that now!

First we'll build a new model specification and recipe for the analysis. Note that we use the formula `price ~ sqft + beds` to denote that we have two predictors, and set `neighbors = tune()` to tell `tidymodels` to tune the number of neighbours for us.

```
sacr_recipe <- recipe(price ~ sqft + beds, data = sacramento_train) %>%
  step_scale(all_predictors()) %>%
  step_center(all_predictors())

sacr_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = tune()) %>%
  set_engine("kknn") %>%
  set_mode("regression")
```

Next, we'll use 5-fold cross-validation to choose the number of neighbours via the minimum RMSPE:

```
gridvals <- tibble(neighbors = seq(1, 200))
sacr_k <- workflow() %>%
  add_recipe(sacr_recipe) %>%
  add_model(sacr_spec) %>%
  tune_grid(sacr_vfold, grid = gridvals) %>%
  collect_metrics() %>%
  filter(.metric == "rmse") %>%
  filter(mean == min(mean)) %>%
  pull(neighbors)
sacr_k
```

```
## [1] 14
```

Here we see that the smallest RMSPE occurs when $K = 14$.

Now that we have chosen K, we need to re-train the model on the entire training data set, and after that we can use that model to predict on the test data to get our test error.

```
sacr_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = sacr_k) %>%
  set_engine("kknn") %>%
  set_mode("regression")

knn_mult_fit <- workflow() %>%
  add_recipe(sacr_recipe) %>%
  add_model(sacr_spec) %>%
  fit(data = sacramento_train)
```

```
knn_mult_preds <- knn_mult_fit %>%
  predict(sacramento_test) %>%
  bind_cols(sacramento_test)

knn_mult_mets <- metrics(knn_mult_preds, truth = price, estimate = .pred)
knn_mult_mets
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard   85152.
## 2 rsq     standard       0.572
## 3 mae     standard   63575.
```

This time when we performed K-NN regression on the same data set, but also included number of bedrooms as a predictor we obtained a RMSPE test error of 85152. This compares to a RMSPE test error of 87737 when we used only house size as the single predictor. Thus in this case, we did not improve the model by a large amount by adding this additional predictor.

We can also visualize the model's predictions overlaid on top of the data. This time the predictions will be a surface in 3-D space, instead of a line in 2-D space, as we have 2 predictors instead of 1.

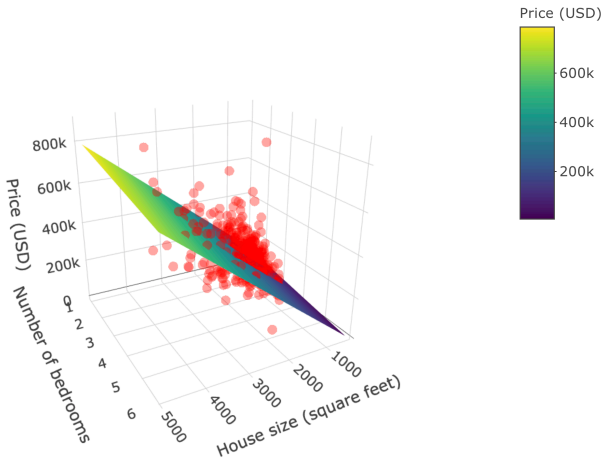


FIGURE 8.9: K-NN regression model's predictions represented as a surface in 3-D space overlaid on top of the data using three predictors.

We can see that the predictions in this case, where we have 2 predictors, form a surface instead of a line. Because the newly added predictor, number

of bedrooms, is correlated with price (USD) (meaning as price changes, so does number of bedrooms) and not totally determined by house size (our other predictor), we get additional and useful information for making our predictions. For example, in this model we would predict that the cost of a house with a size of 2,500 square feet generally increases slightly as the number of bedrooms increases. Without having the additional predictor of number of bedrooms, we would predict the same price for these two houses.

9

Regression II: linear regression

9.1 Overview

This chapter provides an introduction to linear regression models in a predictive context, focusing primarily on the case where there is a single predictor and single response variable of interest, as well as comparison to K-nearest neighbours methods. The chapter concludes with a discussion of linear regression with multiple predictors.

9.2 Chapter learning objectives

By the end of the chapter, students will be able to:

- Perform linear regression in R using `tidymodels` and evaluate it on a test dataset.
- Compare and contrast predictions obtained from K-nearest neighbour regression to those obtained using simple ordinary least squares regression from the same dataset.
- In R, overlay regression lines from `geom_smooth` on a single plot.

9.3 Simple linear regression

K-NN is not the only type of regression; another quite useful, and arguably the most common, type of regression is called simple linear regression. Simple linear regression is similar to K-NN regression in that the target/response variable is quantitative. However, one way it varies quite differently is how the training data is used to predict a value for a new observation. Instead of looking at the K-nearest neighbours and averaging over their values for a prediction, in simple linear regression all the training data points are used to

create a straight line of best fit, and then the line is used to "look up" the predicted value.

Note: for simple linear regression there is only one response variable and only one predictor. Later in this chapter we introduce the more general linear regression case where more than one predictor can be used.

For example, let's revisit the smaller version of the Sacramento housing data set. Recall that we have come across a new 2,000-square foot house we are interested in purchasing with an advertised list price of $350,000. Should we offer the list price, or is that over/undervalued?

To answer this question using simple linear regression, we use the data we have to draw the straight line of best fit through our existing data points:

The equation for the straight line is:

$$\text{house price} = \beta_0 + \beta_1 \cdot (\text{house size}),$$

where

- β_0 is the vertical intercept of the line (the value where the line cuts the vertical axis)
- β_1 is the slope of the line

Therefore using the data to find the line of best fit is equivalent to finding coefficients β_0 and β_1 that *parametrize* (correspond to) the line of best fit. Once we have the coefficients, we can use the equation above to evaluate the predicted price given the value we have for the predictor/explanatory variable—here 2,000 square feet.

```
## [1] 287962
```

By using simple linear regression on this small data set to predict the sale price for a 2,000 square foot house, we get a predicted value of $287962. But wait a minute…how exactly does simple linear regression choose the line of best fit? Many different lines could be drawn through the data points. We show some examples below:

Simple linear regression chooses the straight line of best fit by choosing the line that minimzes the **average** vertical distance between itself and each of the observed data points. From the lines shown above, that is the blue line. What exactly do we mean by the vertical distance between the predicted

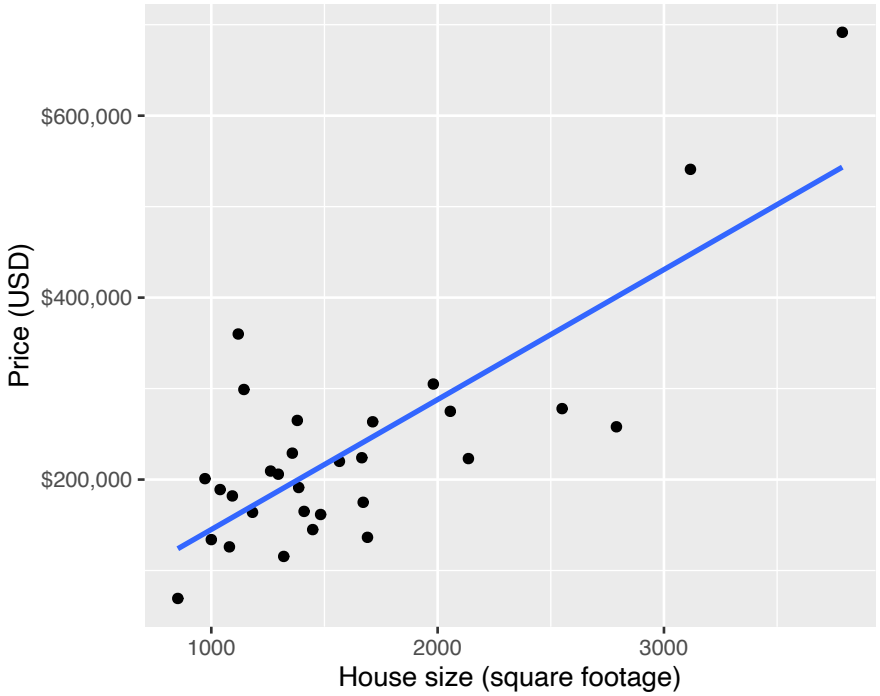


FIGURE 9.1: Scatter plot of price (USD) versus house size (square footage) with line of best fit for subset of the Sacramento housing data set

values (which fall along the line of best fit) and the observed data points? We illustrate these distances in the plot below with a red line:

To assess the predictive accuracy of a simple linear regression model, we use RMSPE—the same measure of predictive performance we used with K-NN regression.

9.4 Linear regression in R

We can perform simple linear regression in R using `tidymodels` in a very similar manner to how we performed K-NN regression. To do this, instead of creating a `nearest_neighbor` model specification with the `kknn` engine, we use a `linear_reg` model specification with the `lm` engine. Another difference is that we do not need to choose K in the context of linear regression, and so we do not need to perform cross validation. Below we illustrate how we can use the usual `tidymodels` workflow to predict house sale price given house size

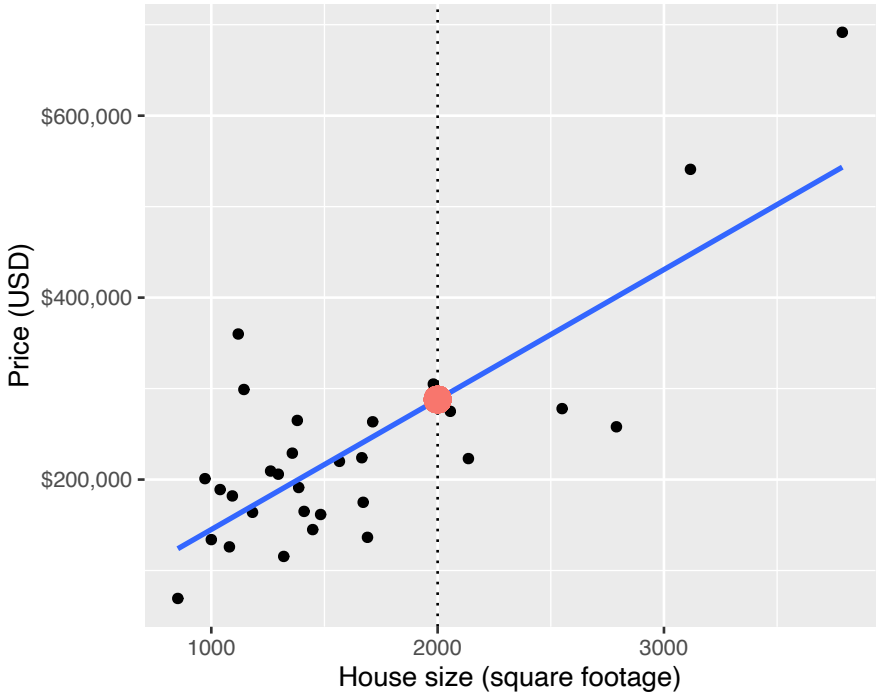


FIGURE 9.2: Scatter plot of price (USD) versus house size (square footage) with line of best fit and predicted price for a 2000 square foot home represented as a red dot

using a simple linear regression approach using the full Sacramento real estate data set.

An additional difference that you will notice below is that we do not standardize (i.e., scale and center) our predictors. In K-nearest neighbours models, recall that the model fit changes depending on whether we standardize first or not. In linear regression, standardization does not affect the fit (it *does* affect the coefficients in the equation, though!). So you can standardize if you want—it won't hurt anything—but if you leave the predictors in their original form, the best fit coefficients are usually easier to interpret afterward.

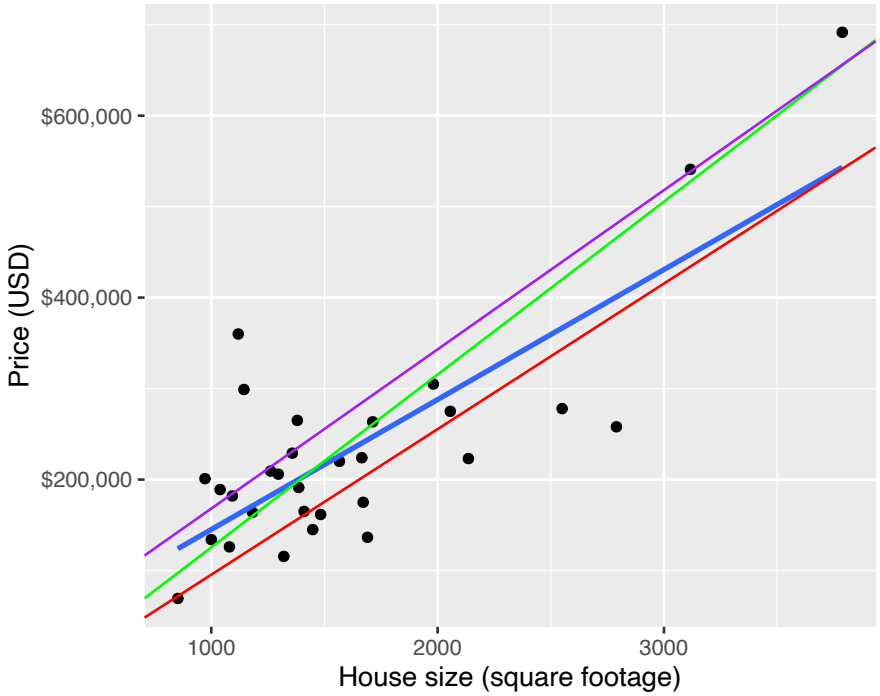


FIGURE 9.3: Scatter plot of price (USD) versus house size (square footage) with many possible lines that could be drawn through the data points

As usual, we start by putting some test data away in a lock box that we can come back to after we choose our final model. Let's take care of that now.

```
set.seed(1234)
sacramento_split <- initial_split(sacramento, prop = 0.6, strata = price)
sacramento_train <- training(sacramento_split)
sacramento_test <- testing(sacramento_split)
```

Now that we have our training data, we will create the model specification and recipe, and fit our simple linear regression model:

```
lm_spec <- linear_reg() %>%
  set_engine("lm") %>%
  set_mode("regression")

lm_recipe <- recipe(price ~ sqft, data = sacramento_train)
```

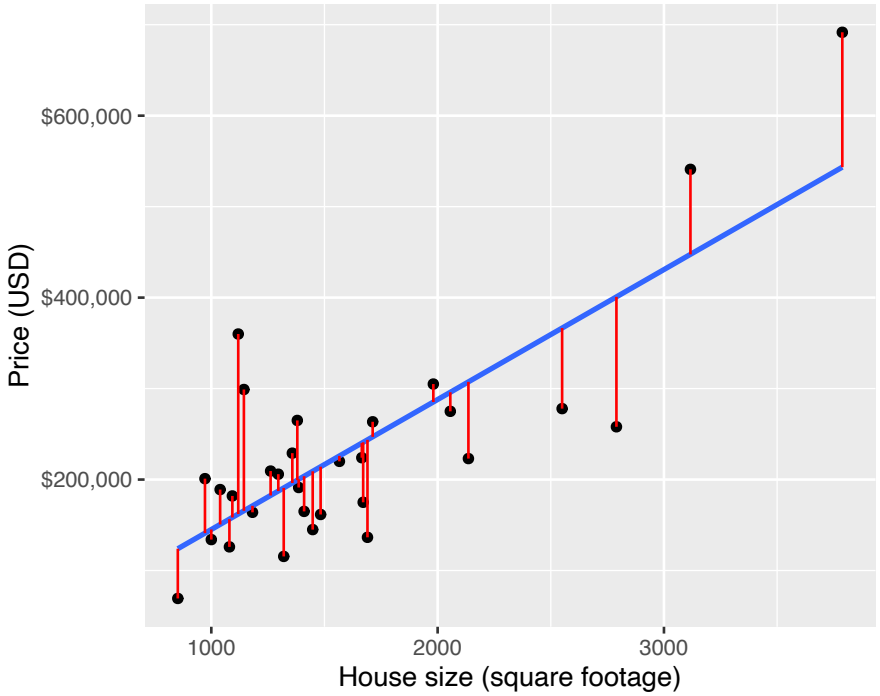


FIGURE 9.4: Scatter plot of price (USD) versus house size (square footage) with the vertical distances between the predicted values and the observed data points

```
lm_fit <- workflow() %>%
  add_recipe(lm_recipe) %>%
  add_model(lm_spec) %>%
  fit(data = sacramento_train)
lm_fit
```

```
## == Workflow [trained] ==================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------
## 0 Recipe Steps
##
## -- Model -----------------------------------------------
##
## Call:
```

```
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
## (Intercept)         sqft
##       15059          138
```

Our coefficients are (intercept) $\beta_0 = 15059$ and (slope) $\beta_1 = 138$. This means that the equation of the line of best fit is

$$\text{house price} = 15059 + 138 \cdot (\text{house size}),$$

and that the model predicts that houses start at \$15059 for 0 square feet, and that every extra square foot increases the cost of the house by \$138. Finally, we predict on the test data set to assess how well our model does:

```
lm_test_results <- lm_fit %>%
  predict(sacramento_test) %>%
  bind_cols(sacramento_test) %>%
  metrics(truth = price, estimate = .pred)
lm_test_results
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard   85161.
## 2 rsq     standard       0.572
## 3 mae     standard   62608.
```

Our final model's test error as assessed by RMSPE is 85161. Remember that this is in units of the target/response variable, and here that is US Dollars (USD). Does this mean our model is "good" at predicting house sale price based off of the predictor of home size? Again answering this is tricky to answer and requires to use domain knowledge and think about the application you are using the prediction for.

To visualize the simple linear regression model, we can plot the predicted house price across all possible house sizes we might encounter superimposed on a scatter plot of the original housing price data. There is a plotting function in the `tidyverse`, `geom_smooth`, that allows us to do this easily by adding a layer on our plot with the simple linear regression predicted line of best fit. The default for this adds a plausible range to this line that we are not interested in at this point, so to avoid plotting it, we provide the argument `se = FALSE` in our call to `geom_smooth`.

```
lm_plot_final <- ggplot(sacramento_train, aes(x = sqft, y = price)) +
  geom_point(alpha = 0.4) +
  xlab("House size (square footage)") +
```

```
  ylab("Price (USD)") +
  scale_y_continuous(labels = dollar_format()) +
  geom_smooth(method = "lm", se = FALSE)
lm_plot_final
```

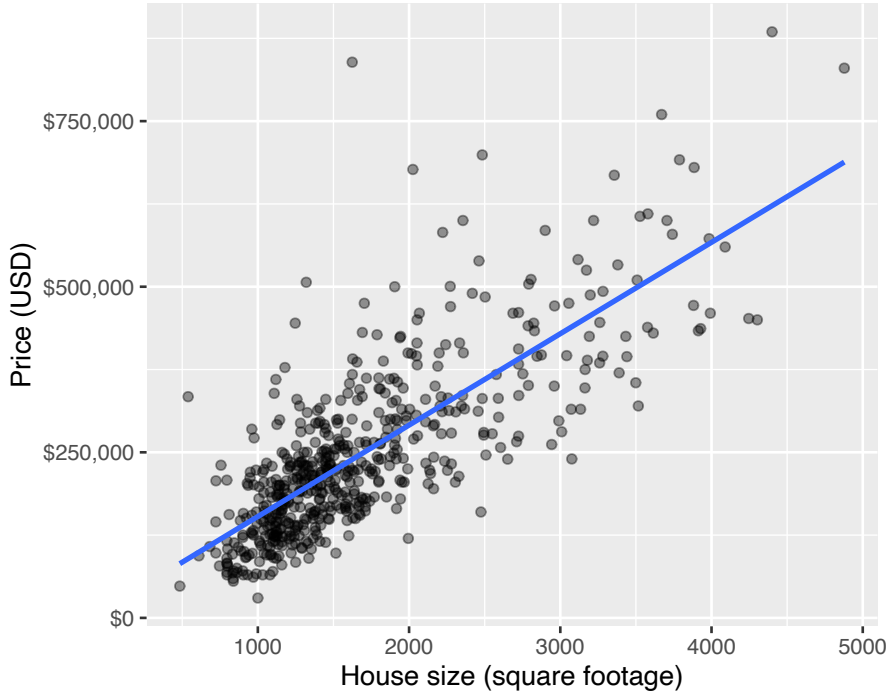


FIGURE 9.5: Scatter plot of price (USD) versus house size (square footage) with line of best fit for complete Sacramento housing data set

We can extract the coefficients from our model by accessing the fit object that is output by the `fit` function; we first have to extract it from the workflow using the `pull_workflow_fit` function, and then apply the `tidy` function to convert the result into a data frame:

```
coeffs <- tidy(pull_workflow_fit(lm_fit))
coeffs
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic   p.value
##   <chr>          <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)   15059.   8745.        1.72 8.56e-  2
```

```
## 2 sqft            138.      4.77     28.9  3.13e-113
```

9.5 Comparing simple linear and K-NN regression

Now that we have a general understanding of both simple linear and K-NN regression, we can start to compare and contrast these methods as well as the predictions made by them. To start, let's look at the visualization of the simple linear regression model predictions for the Sacramento real estate data (predicting price from house size) and the "best" K-NN regression model obtained from the same problem:

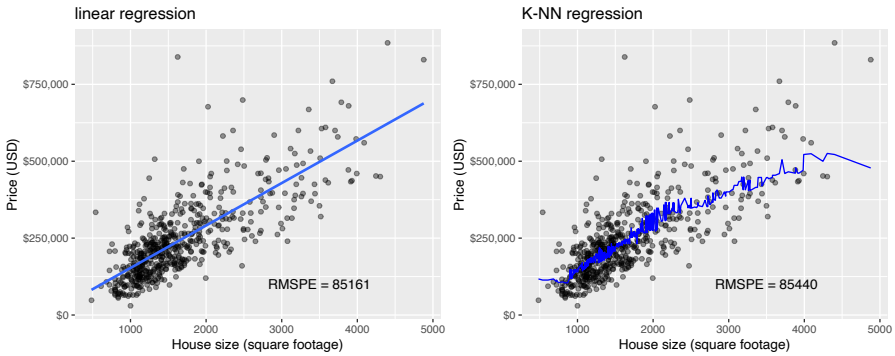


FIGURE 9.6: Comparison of simple linear regression and K-NN regression

What differences do we observe from the visualization above? One obvious difference is the shape of the blue lines. In simple linear regression we are restricted to a straight line, whereas in K-NN regression our line is much more flexible and can be quite wiggly. But there is a major interpretability advantage in limiting the model to a straight line. A straight line can be defined by two numbers, the vertical intercept and the slope. The intercept tells us what the prediction is when all of the predictors are equal to 0; and the slope tells us what unit increase in the target/response variable we predict given a unit increase in the predictor/explanatory variable. K-NN regression, as simple as it is to implement and understand, has no such interpretability from its wiggly line.

There can however also be a disadvantage to using a simple linear regression model in some cases, particularly when the relationship between the target and the predictor is not linear, but instead some other shape (e.g. curved or oscillating). In these cases the prediction model from a simple linear regression will underfit (have high bias), meaning that model/predicted values does not match the actual observed values very well. Such a model would probably have

a quite high RMSE when assessing model goodness of fit on the training data and a quite high RMPSE when assessing model prediction quality on a test data set. On such a data set, K-NN regression may fare better. Additionally, there are other types of regression you can learn about in future courses that may do even better at predicting with such data.

How do these two models compare on the Sacramento house prices data set? On the visualizations above we also printed the RMPSE as calculated from predicting on the test data set that was not used to train/fit the models. The RMPSE for the simple linear regression model is slightly lower than the RMPSE for the K-NN regression model. Considering that the simple linear regression model is also more interpretable, if we were comparing these in practice we would likely choose to use the simple linear regression model.

Finally, note that the K-NN regression model becomes "flat" at the left and right boundaries of the data, while the linear model predicts a constant slope. Predicting outside the range of the observed data is known as *extrapolation*; K-NN and linear models behave quite differently when extrapolating. Depending on the application, the flat or constant slope trend may make more sense. For example, if our housing data were slightly different, the linear model may have actually predicted a *negative* price for a small houses (if the intercept β_0 was negative), which obviously does not match reality. On the other hand, the trend of increasing house size corresponding to increasing house price probably continues for large houses, so the "flat" extrapolation of K-NN likely does not match reality.

9.6 Multivariate linear regression

As in K-NN classification and K-NN regression, we can move beyond the simple case of one response variable and only one predictor and perform multivariate linear regression where we can have multiple predictors. In this case we fit a plane to the data, as opposed to a straight line.

To do this, we follow a very similar approach to what we did for K-NN regression; but recall that we do not need to use cross-validation to choose any parameters, nor do we need to standardize (i.e., center and scale) the data for linear regression. We demonstrate how to do this below using the Sacramento real estate data with both house size (measured in square feet) as well as number of bedrooms as our predictors, and continue to use house sale price as our outcome/target variable that we are trying to predict. We will start by changing the formula in the recipe to include both the `sqft` and `beds` variables as predictors:

```
lm_recipe <- recipe(price ~ sqft + beds, data = sacramento_train)
```

Now we can build our workflow and fit the model:

```
lm_fit <- workflow() %>%
  add_recipe(lm_recipe) %>%
  add_model(lm_spec) %>%
  fit(data = sacramento_train)
lm_fit
```

```
## == Workflow [trained] ================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------
## 0 Recipe Steps
##
## -- Model -----------------------------------------------
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
## (Intercept)         sqft         beds
##       52690          155       -20209
```

And finally, we predict on the test data set to assess how well our model does:

```
lm_mult_test_results <- lm_fit %>%
  predict(sacramento_test) %>%
  bind_cols(sacramento_test) %>%
  metrics(truth = price, estimate = .pred)
lm_mult_test_results
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard   82835.
## 2 rsq     standard       0.596
## 3 mae     standard   61008.
```

In the case of two predictors, our linear regression creates a *plane* of best fit, shown below:

We see that the predictions from linear regression with two predictors form

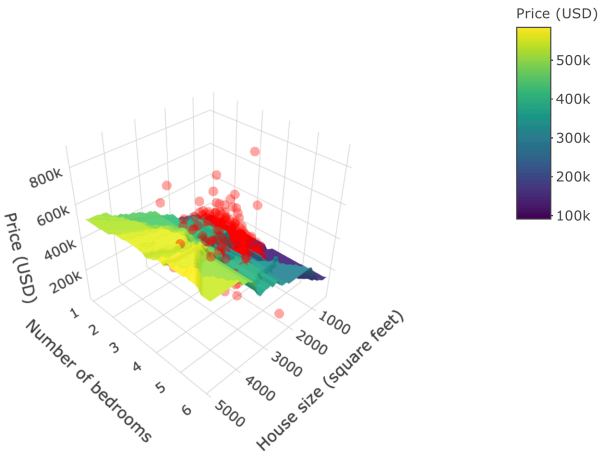


FIGURE 9.7: Simple linear regression model's predictions represented as a plane overlaid on top of the data using three predictors (price, house size, and the number of bedrooms)

a flat plane. This is the hallmark of linear regression, and differs from the wiggly, flexible surface we get from other methods such as K-NN regression. As discussed this can be advantageous in one aspect, which is that for each predictor, we can get slopes/intercept from linear regression, and thus describe the plane mathematically. We can extract those slope values from our model object as shown below:

```
coeffs <- tidy(pull_workflow_fit(lm_fit))
coeffs
```

```
## # A tibble: 3 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)   52690.  13745.        3.83 1.41e- 4
## 2 sqft            155.      6.72     23.0  4.46e-83
## 3 beds          -20209.   5734.       -3.52 4.59e- 4
```

And then use those slopes to write a mathematical equation to describe the prediction plane:

$$\text{house price} = \beta_0 + \beta_1 \cdot (\text{house size}) + \beta_2 \cdot (\text{number of bedrooms}),$$

where:

- β_0 is the vertical intercept of the hyperplane (the value where it cuts the vertical axis)

- β_1 is the slope for the first predictor (house size)
- β_2 is the slope for the second predictor (number of bedrooms)

Finally, we can fill in the values for β_0, β_1 and β_2 from the model output above to create the equation of the plane of best fit to the data:

$$\text{house price} = 52690 + 155 \cdot (\text{house size}) - 20209 \cdot (\text{number of bedrooms})$$

This model is more interpretable than the multivariate K-NN regression model; we can write a mathematical equation that explains how each predictor is affecting the predictions. But as always, we should look at the test error and ask whether linear regression is doing a better job of predicting compared to K-NN regression in this multivariate regression case. To do that we can use this linear regression model to predict on the test data to get our test error.

```
lm_mult_test_results
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard   82835.
## 2 rsq     standard       0.596
## 3 mae     standard   61008.
```

We get that the RMSPE for the multivariate linear regression model of 82835. This prediction error is less than the prediction error for the multivariate K-NN regression model, indicating that we should likely choose linear regression for predictions of house price on this data set. But we should also ask if this more complex model is doing a better job of predicting compared to our simple linear regression model with only a single predictor (house size). Revisiting last section, we see that our RMSPE for our simple linear regression model with only a single predictor was 85161, which is slightly more than that of our more complex model. Our model with two predictors provided a slightly better fit on test data than our model with just one.

But should we always end up choosing a model with more predictors than fewer? The answer is no; you never know what model will be the best until you go through the process of comparing their performance on held-out test data. Exploratory data analysis can give you some hints, but until you look at the prediction errors to compare the models you don't really know. Additionally, here we compare test errors purely for the purposes of teaching. In practice, when you want to compare several regression models with differing numbers of predictor variables, you should use cross-validation on the training set only; in this case choosing the model is part of tuning, so you cannot use the test data. There are several well known and more advanced methods to

do this that are beyond the scope of this course, and they include backward or forward selection, and L1 or L2 regularization (also known as Lasso and ridge regression, respectively).

9.7 The other side of regression

So far in this textbook we have used regression only in the context of prediction. However, regression is also a powerful method to understand and/or describe the relationship between a quantitative response variable and one or more explanatory variables. Extending the case we have been working with in this chapter (where we are interested in house price as the outcome/response variable), we might also be interested in describing the individual effects of house size and the number of bedrooms on house price, quantifying how big each of these effects are, and assessing how accurately we can estimate each of these effects. This side of regression is the topic of many follow-on statistics courses and beyond the scope of this course.

9.8 Additional resources

- Pages 59-71 of Introduction to Statistical Learning[1] with Applications in R by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani
- Pages 104 - 109 of An Introduction to Statistical Learning with Applications in R[2] by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani
- The `caret` Package[3]
- Chapters 6 - 11 of Modern Dive[4] Statistical Inference via Data Science by Chester Ismay and Albert Y. Kim

[1] http://www-bcf.usc.edu/~gareth/ISL/ISLR%20Seventh%20Printing.pdf
[2] https://www-bcf.usc.edu/~gareth/ISL/ISLR%20Seventh%20Printing.pdf
[3] https://topepo.github.io/caret/index.html
[4] https://moderndive.com/

10

Clustering

10.1 Overview

As part of exploratory data analysis, it is often helpful to see if there are meaningful subgroups (or *clusters*) in the data; this grouping can be used for many purposes, such as generating new questions or improving predictive analyses. This chapter provides an introduction to clustering using the *K-means* algorithm, including techniques to choose the number of clusters.

10.2 Chapter learning objectives

By the end of the chapter, students will be able to:

- Describe a case where clustering is appropriate, and what insight it might extract from the data.
- Explain the K-means clustering algorithm.
- Interpret the output of a K-means analysis.
- Identify when it is necessary to scale variables before clustering, and do this using R.
- Perform K-means clustering in R using `kmeans`.
- Use the elbow method to choose the number of clusters for K-means.
- Visualize the output of K-means clustering in R using coloured scatter plots.
- Describe the advantages, limitations and assumptions of the K-means clustering algorithm.

10.3 Clustering

Clustering is a data analysis task involving separating a data set into subgroups of related data. For example, we might use clustering to separate a data set of documents into groups that correspond to topics, a data set of

human genetic information into groups that correspond to ancestral subpopulations, or a data set of online customers into groups that correspond to purchasing behaviours. Once the data are separated we can, for example, use the subgroups to generate new questions about the data and follow up with a predictive modelling exercise. In this course, clustering will be used only for exploratory analysis, i.e., uncovering patterns in the data that we have.

Note that clustering is a fundamentally different kind of task than classification or regression. Most notably, both classification and regression are *supervised tasks* where there is a *predictive target* (a class label or value), and we have examples of past data with labels/values that help us predict those of future data. By contrast, clustering is an *unsupervised task*, as we are trying to understand and examine the structure of data without any labels to help us. This approach has both advantages and disadvantages. Clustering requires no additional annotation or input on the data; for example, it would be nearly impossible to annotate all the articles on Wikipedia with human-made topic labels, but we can still cluster the articles without this information to automatically find groupings corresponding to topics.

However, because there is no predictive target, it is not as easy to evaluate the "quality" of a clustering. With classification, we are able to use a test data set to assess prediction performance. In clustering, there is not a single good choice for evaluation. In this book, we will use visualization to ascertain the quality of a clustering, and leave rigorous evaluation for more advanced courses.

There are also so-called *semisupervised* tasks, where only some of the data come with labels / annotations, but the vast majority don't. The goal is to try to uncover underlying structure in the data that allows one to guess the missing labels. This sort of task is very useful, for example, when one has an unlabelled data set that is too large to manually label, but one is willing to provide a few informative example labels as a "seed" to guess the labels for all the data.

An illustrative example

Here we will present an illustrative example using a simulated data set. Suppose we have data with two variables measuring customer loyalty and satisfaction, and we want to learn whether there are distinct "types" of customer. Understanding this might help us come up with better products or promotions to improve our business in a data-driven way.

```
marketing_data
```

```
## # A tibble: 19 x 2
##    loyalty  csat
##      <dbl> <dbl>
##  1     7    1
##  2     7.5  1
##  3     8    2
##  4     7    2
##  5     8    3
##  6     1.5  1.75
##  7     1    3
##  8     0.5  4
##  9     2    4
## 10     7    6
## 11     6    6
## 12     7    7
## 13     6    7
## 14     5    7
## 15     9.5  8
## 16     7    8
## 17     8.3  9
## 18     4    8
## 19     2    3
```

Based on this visualization, we might suspect there are a few subtypes of customer, selected from combinations of high/low satisfaction and high/low loyalty. How do we find this grouping automatically, and how do we pick the number of subtypes to look for? The way to rigorously separate the data into groups is to use a clustering algorithm. In this chapter, we will focus on the *K-means* algorithm, a widely-used and often very effective clustering method, combined with the *elbow method* for selecting the number of clusters. This procedure will separate the data into the following groups denoted by colour:

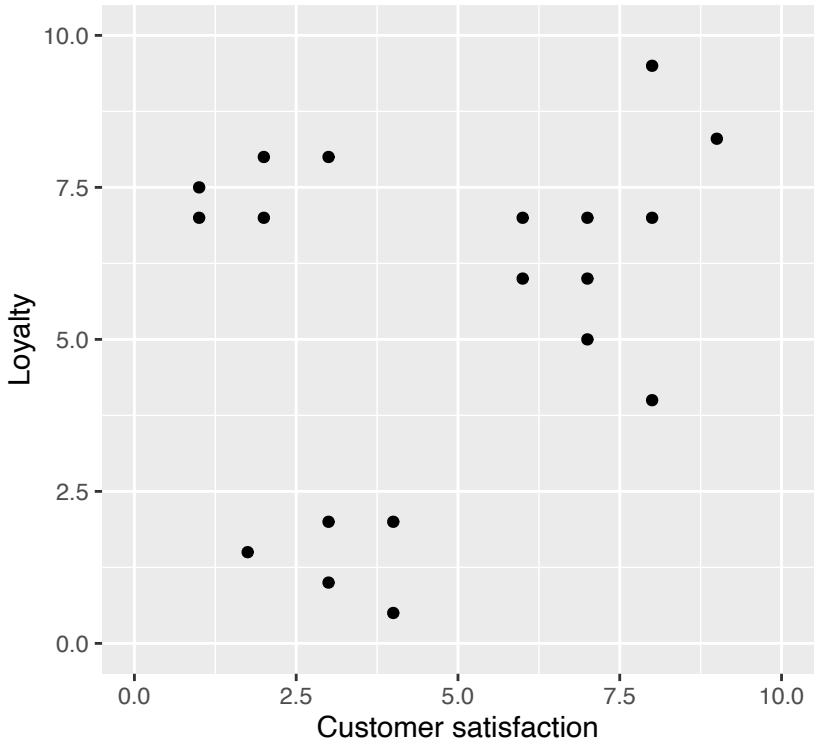
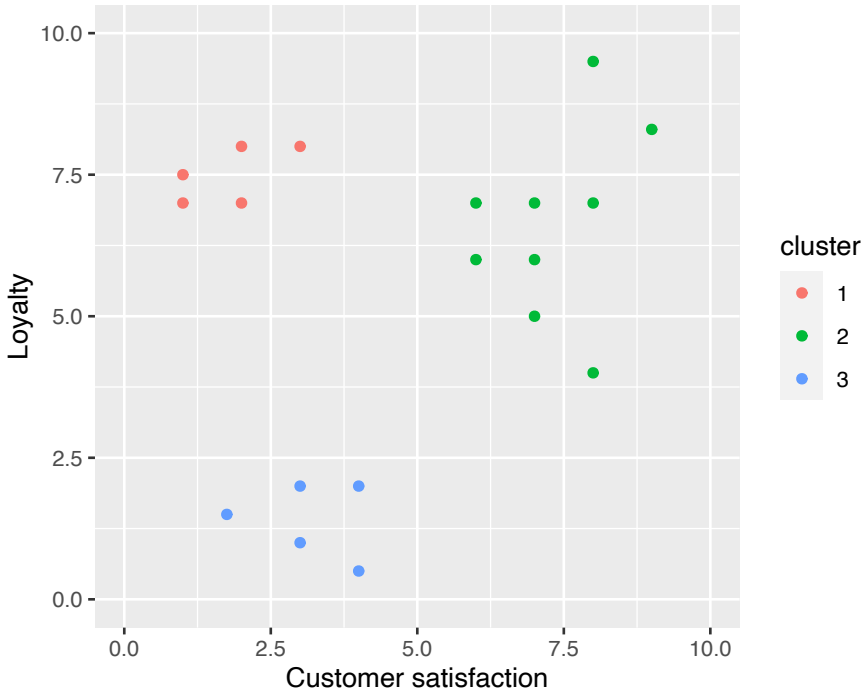


FIGURE 10.1: Simulated data of customer loyalty versus satisfaction. Example derived from Fripp (2020).



What are the labels for these groups? Unfortunately, we don't have any. K-means, like almost all clustering algorithms, just outputs meaningless "cluster labels" that are typically whole numbers: 1, 2, 3, etc. But in a simple case like this, where we can easily visualize the clusters on a scatter plot, we can give human-made labels to the groups using their positions on the plot:

- low loyalty and low satisfaction (blue cluster),
- high loyalty and low satisfaction (pink cluster),
- and high loyalty and high satisfaction (green cluster).

Once we have made these determinations, we can use them to inform our future business decisions, or to ask further questions about our data. For example, here we might notice based on our clustering that there aren't any customers with high satisfaction but low loyalty, and generate new analyses or business strategies based on this information.

10.4 K-means

10.4.1 Measuring cluster quality

The K-means algorithm is a procedure that groups data into K clusters. It starts with an initial clustering of the data, and then iteratively improves it by making adjustments to the assignment of data to clusters until it cannot improve any further. But how do we measure the "quality" of a clustering, and what does it mean to improve it? In K-means clustering, we measure the quality of a cluster by its *within-cluster sum-of-squared-distances* (WSSD). Computing this involves two steps. First, we find the cluster centers by computing the mean of each variable over data points in the cluster. For example, suppose we have a cluster containing 3 observations, and we are using two variables, x and y, to cluster the data. Then we would compute the x and y variables, μ_x and μ_y, of the cluster center via

$$\mu_x = \frac{1}{3}(x_1 + x_2 + x_3) \quad \mu_y = \frac{1}{3}(y_1 + y_2 + y_3).$$

In the first cluster from the customer satisfaction/loyalty example, there are 5 data points. These are shown with their cluster center (`csat = 1.8` and `loyalty = 7.5`) highlighted below.

The second step in computing the WSSD is to add up the squared distance between each point in the cluster and the cluster center. We use the straight-line / Euclidean distance formula that we learned about in the classification chapter. In the 3-observation cluster example above, we would compute the WSSD S^2 via

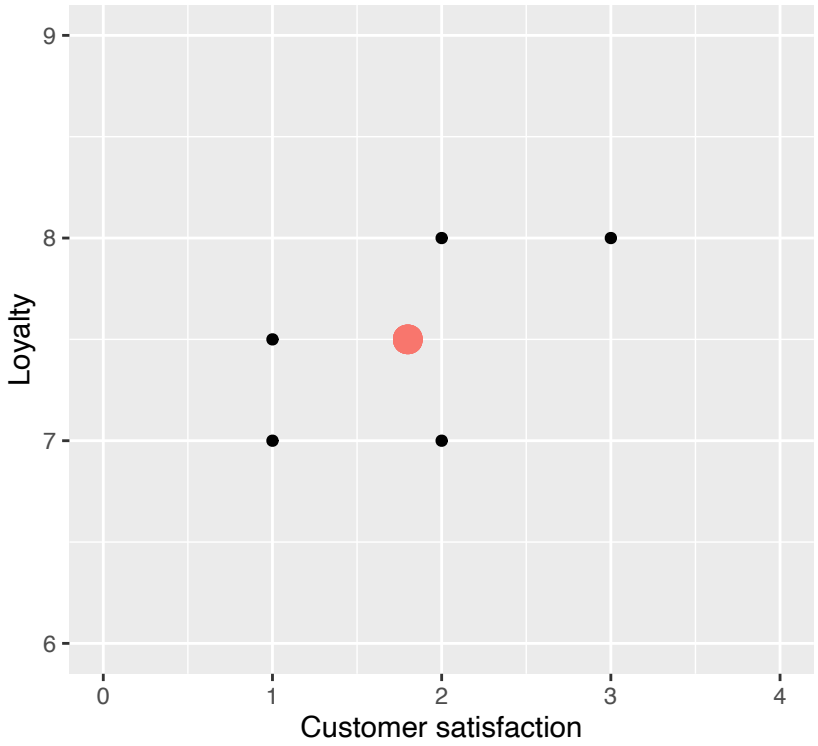


FIGURE 10.2: Cluster 1 from the toy example, with center highlighted.

$$S^2 = \big((x_1 - \mu_x)^2 + (y_1 - \mu_y)^2\big) + \big((x_2 - \mu_x)^2 + (y_2 - \mu_y)^2\big) + \big((x_3 - \mu_x)^2 + (y_3 - \mu_y)^2\big)\,.$$

These distances are denoted by lines for the first cluster of the customer satisfaction/loyalty data example below.

The larger the value of S^2, the more spread-out the cluster is, since large S^2 means that points are far away from the cluster center. Note, however, that "large" is relative to *both* the scale of the variables for clustering *and* the number of points in the cluster; a cluster where points are very close to the center might still have a large S^2 if there are many data points in the cluster.

10.4.2 The clustering algorithm

The K-means algorithm is quite simple. We begin by picking K, and uniformly randomly assigning data to the K clusters. Then K-means consists of two major steps that attempt to minimize the sum of WSSDs over all the clusters, i.e. the *total WSSD*:

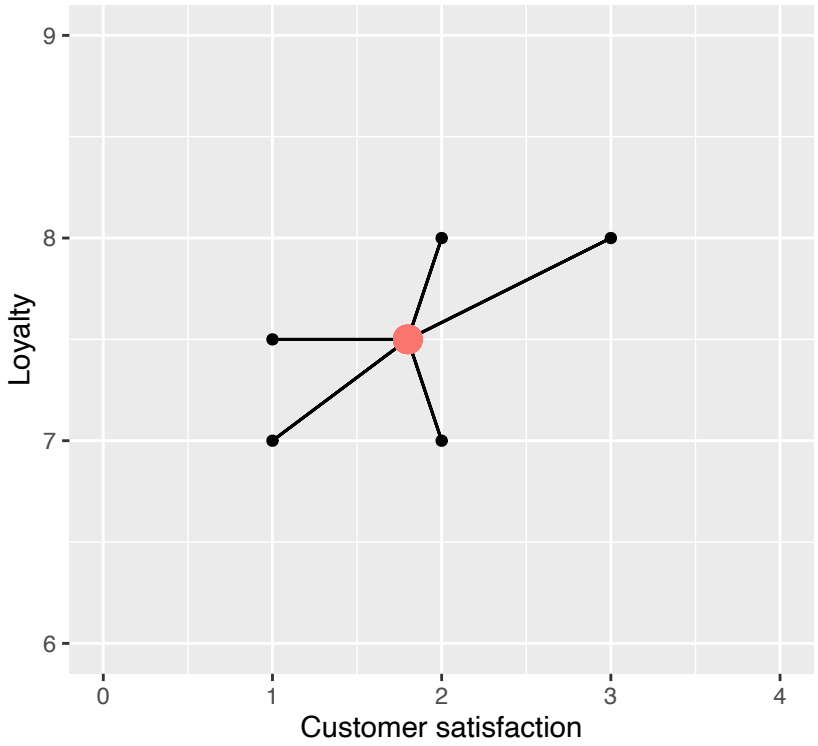
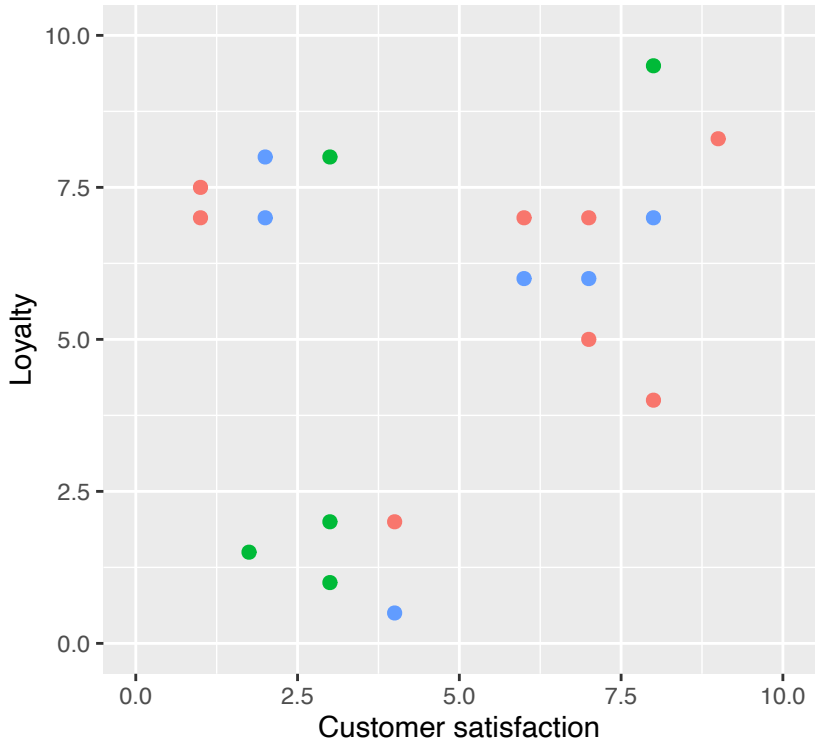


FIGURE 10.3: Cluster 1 from the toy example, with distances to the center highlighted.

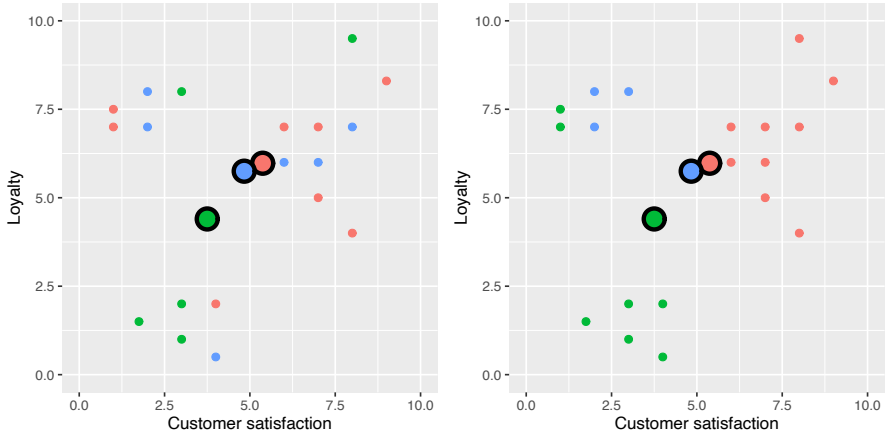
1. **Center update:** Compute the center of each cluster.
2. **Label update:** Reassign each data point to the cluster with the nearest center.

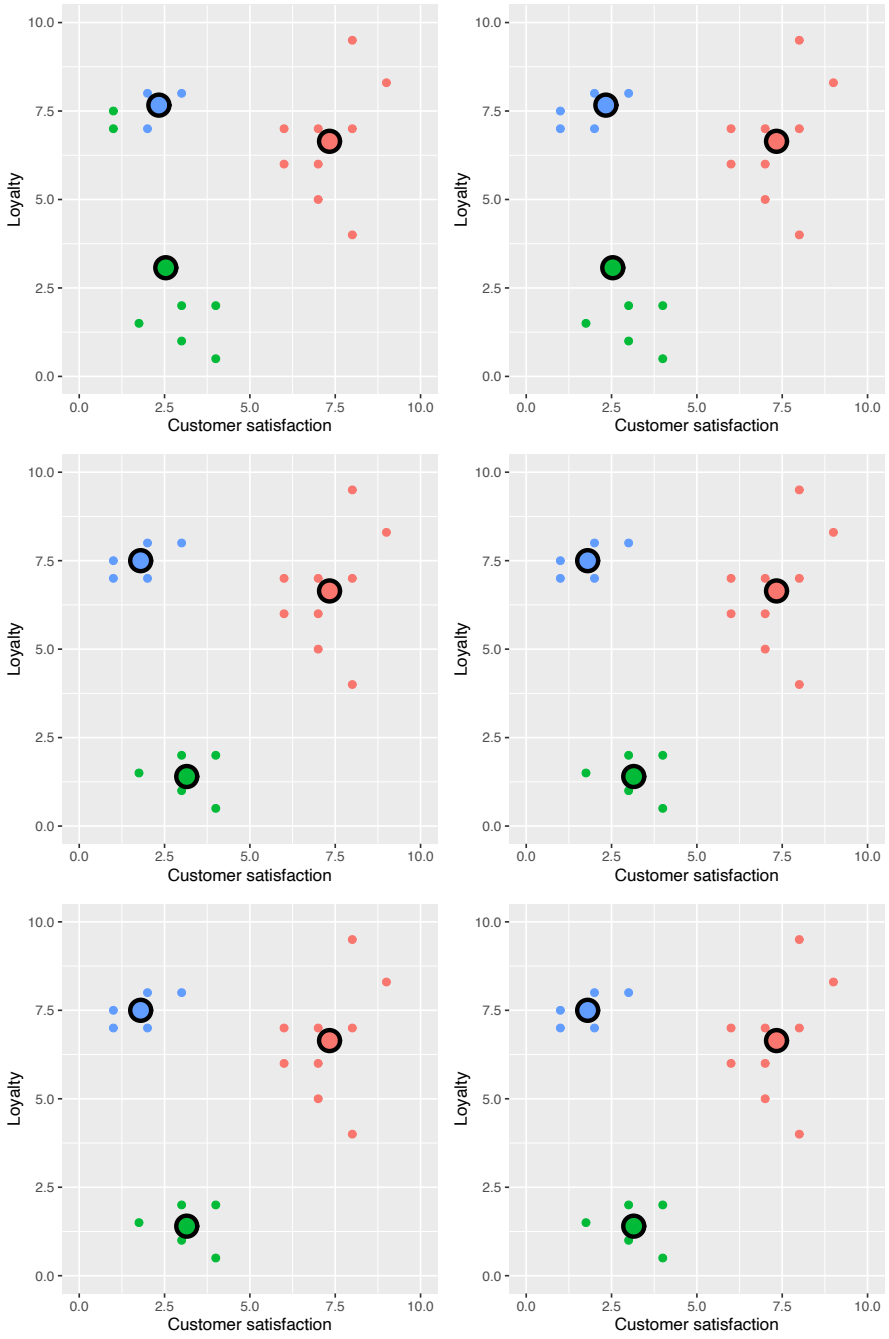
These two steps are repeated until the cluster assignments no longer change. For example, in the customer data example from earlier, our initialization might look like this:



And the first four iterations of K-means would look like (each row corresponds to an iteration, where the left column depicts the center update, and the right column depicts the reassignment of data to clusters):

Center Update **Label Update**





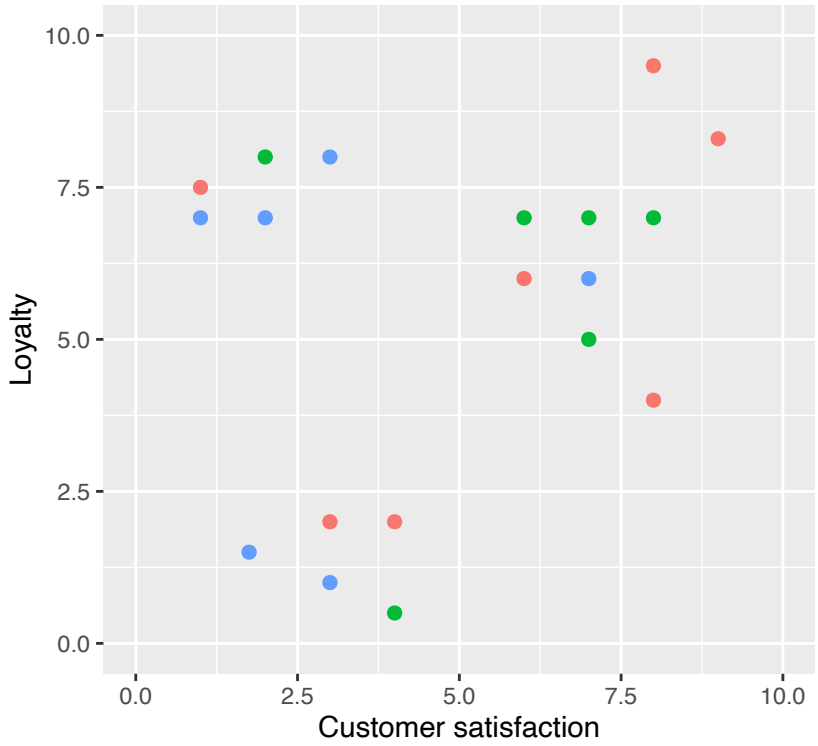
Note that at this point we can terminate the algorithm, since none of the

assignments changed in the fourth iteration; both the centers and labels will remain the same from this point onward.

Is K-means *guaranteed* to stop at some point, or could it iterate forever? As it turns out, the answer is thankfully that K-means is guaranteed to stop after *some* number of iterations. For the interested reader, the logic for this has three steps: (1) both the label update and the center update decrease total WSSD in each iteration, (2) the total WSSD is always greater than or equal to 0, and (3) there are only a finite number of possible ways to assign the data to clusters. So at some point, the total WSSD must stop decreasing, which means none of the assignments are changing and the algorithm terminates.

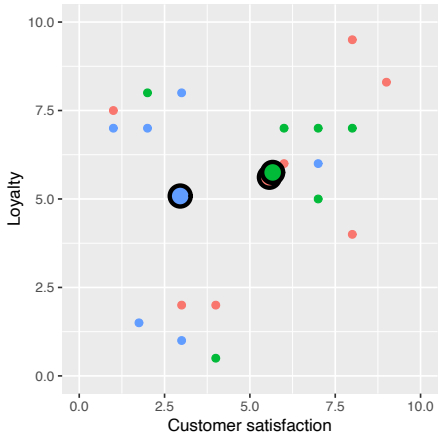
10.4.3 Random restarts

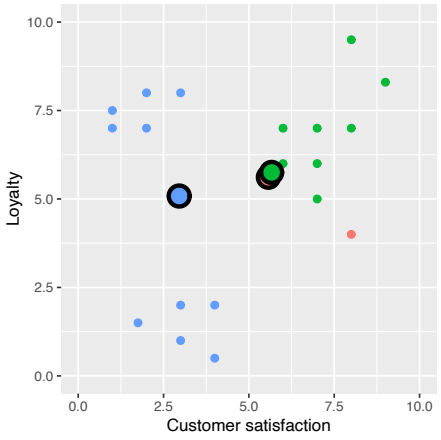
K-means, unlike the classification and regression models we studied in previous chapters, can get "stuck" in a bad solution. For example, if we were unlucky and initialized K-means with the following labels:

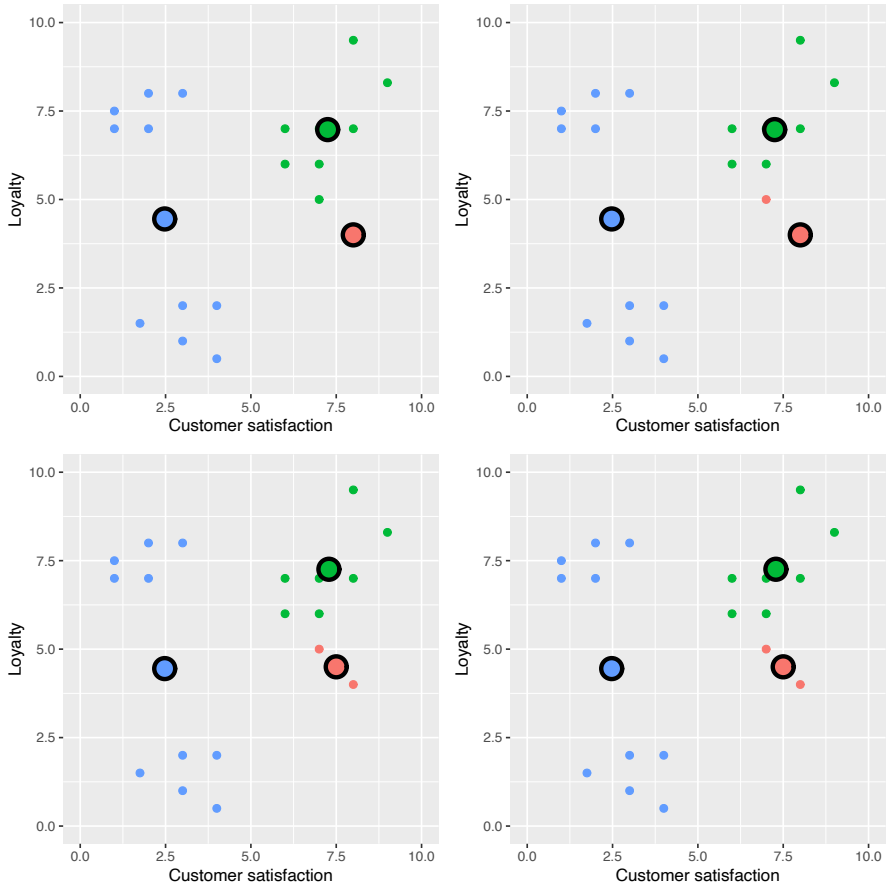


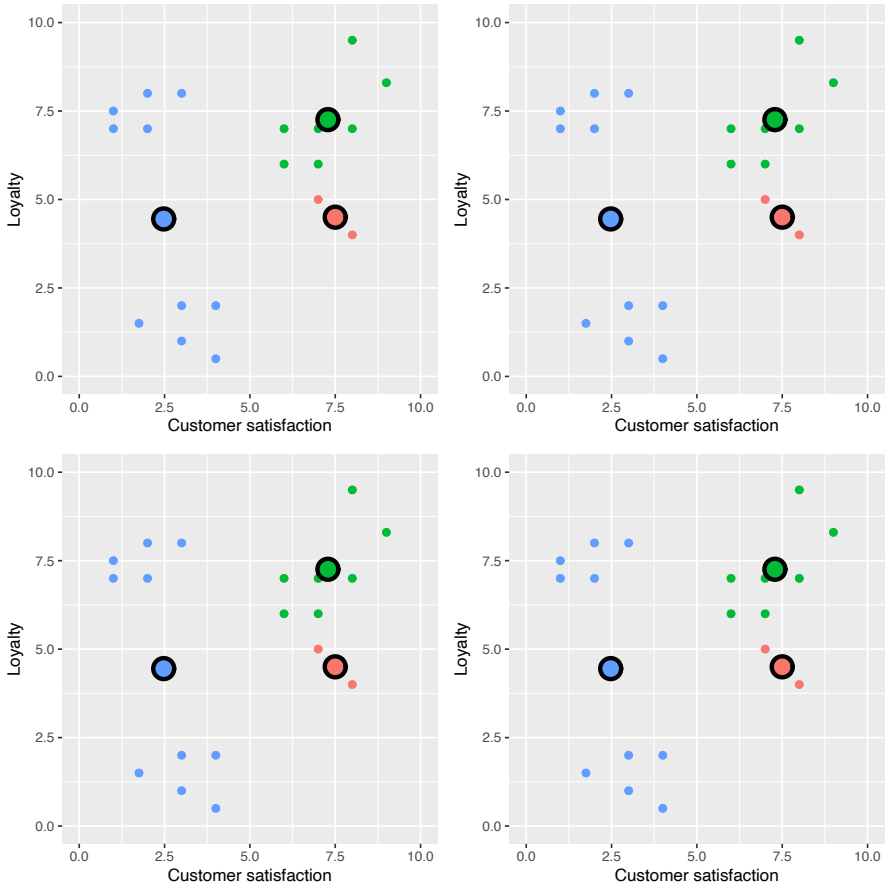
Then the iterations of K-means would look like:

Center Update **Label Update**






Loyalty
Customer satisfaction
0.0
2.5
5.0
7.5
10.0



This looks like a relatively bad clustering of the data, but K-means cannot improve it. To solve this problem when clustering data using K-means, we should randomly re-initialize the labels a few times, run K-means for each initialization, and pick the clustering that has the lowest final total WSSD.

10.4.4 Choosing K

In order to cluster data using K-means, we also have to pick the number of clusters, K. But unlike in classification, we have no data labels and cannot perform cross-validation with some measure of model prediction error. Further, if K is chosen too small, then multiple clusters get grouped together; if K is too large, then clusters get subdivided. In both cases, we will potentially miss interesting structure in the data. For example, take a look below at the K-means clustering of our customer satisfaction and loyalty data for a number of clusters ranging from 1 to 9.

If we set K less than 3, then the clustering merges separate groups of data;

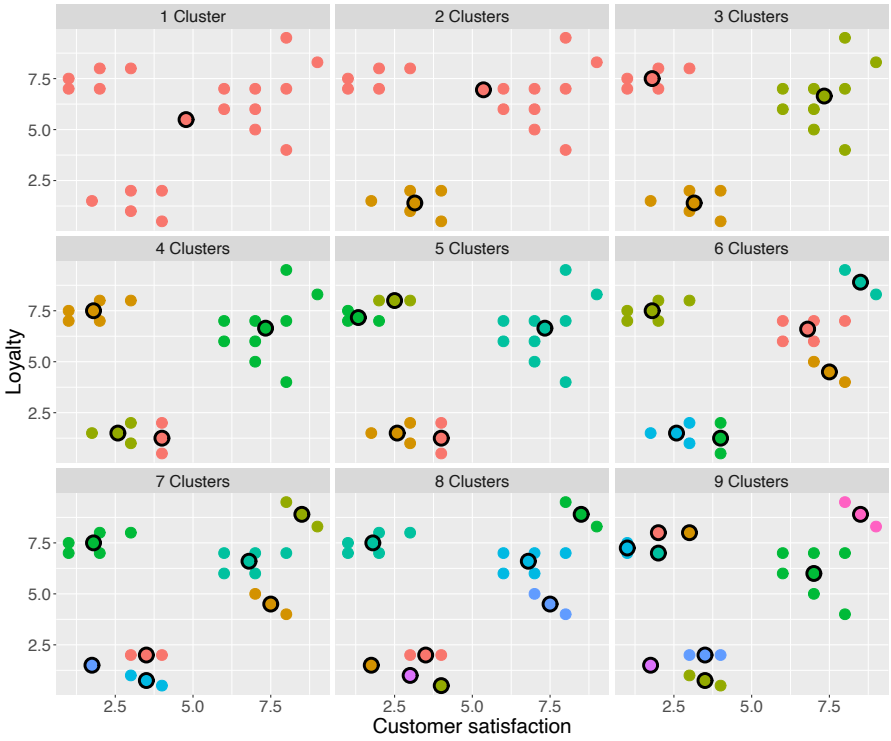
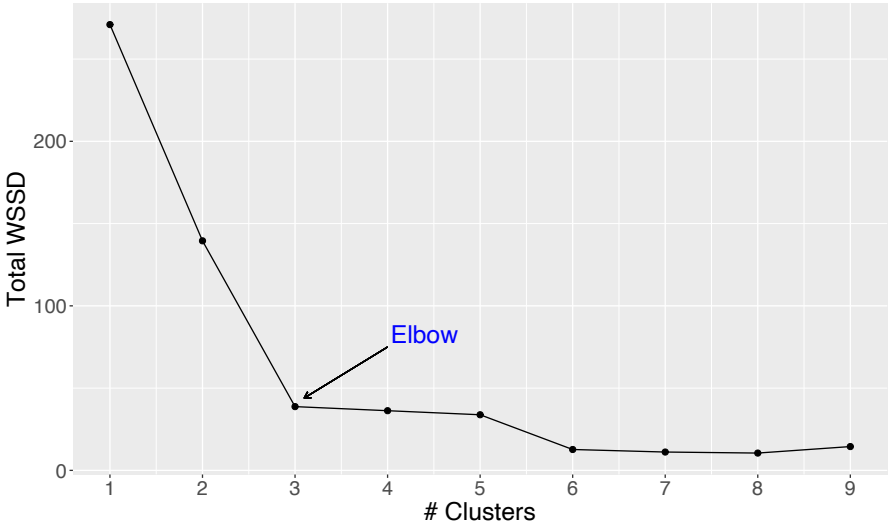


FIGURE 10.4: Clustering of the customer data for # clusters ranging from 1 to 9.

this causes a large total WSSD, since the cluster center (denoted by an "x") is not close to any of the data in the cluster. On the other hand, if we set K greater than 3, the clustering subdivides subgroups of data; this does indeed still decrease the total WSSD, but by only a *diminishing amount*. If we plot the total WSSD versus the number of clusters, we see that the decrease in total WSSD levels off (or forms an "elbow shape") when we reach roughly the right number of clusters.



10.5 Data pre-processing for K-means

Similar to K-nearest neighbours classification and regression, K-means clustering uses straight-line distance to decide which points are similar to each other. Therefore, the *scale* of each of the variables in the data will influence which cluster data points end up being assigned to. Variables that have a large scale will have a much larger effect on deciding cluster assignment than variables with a small scale. To address this problem, we typically standardize our data before clustering, which ensures that each variable has a mean of 0 and standard deviation of 1. The `scale` function in R can be used to do this. We show an example of how to use this function below using the data in this chapter:

```
scaled_data <- map_df(unscaled_data, scale)
scaled_data
```

```
## # A tibble: 19 x 2
##    loyalty[,1] csat[,1]
##          <dbl>    <dbl>
##  1       0.541   -1.40
##  2       0.720   -1.40
##  3       0.899   -1.03
##  4       0.541   -1.03
##  5       0.899   -0.659
```

```
##  6      -1.43    -1.12
##  7      -1.61    -0.659
##  8      -1.79    -0.288
##  9      -1.25    -0.288
## 10       0.541    0.454
## 11       0.183    0.454
## 12       0.541    0.826
## 13       0.183    0.826
## 14      -0.175    0.826
## 15       1.44     1.20
## 16       0.541    1.20
## 17       1.01     1.57
## 18      -0.533    1.20
## 19      -1.25    -0.659
```

10.6 K-means in R

To peform K-means clustering in R, we use the `kmeans` function. It takes at least two arguments: the data frame containing the data you wish to cluster, and K, the number of clusters (here we choose K = 3). Note that since the K-means algorithm uses a random initialization of assignments, we need to set the random seed to make the clustering reproducible.

```
set.seed(1234)
marketing_clust <- kmeans(marketing_data, centers = 3)
marketing_clust
```

```
## K-means clustering with 3 clusters of sizes 9, 5, 5
##
## Cluster means:
##   loyalty  csat label
## 1   6.644 7.333 1.778
## 2   7.500 1.800 3.000
## 3   1.400 3.150 3.000
##
## Clustering vector:
##  [1] 2 2 2 2 2 3 3 3 3 1 1 1 1 1 1 1 1 1 3
##
## Within cluster sum of squares by cluster:
## [1] 31.36  3.80  5.15
##  (between_SS / total_SS =  85.6 %)
##
```

```
## Available components:
##
## [1] "cluster"      "centers"      "totss"
## [4] "withinss"     "tot.withinss" "betweenss"
## [7] "size"         "iter"         "ifault"
```

As you can see above, the clustering object returned by `kmeans` has a lot of information that can be used to visualize the clusters, pick K, and evaluate the total WSSD. To obtain this information in a tidy format, we will call in help from the `broom` package. Let's start by visualizing the clustering as a coloured scatter plot. To do that we use the `augment` function, which takes in the model and the original data frame, and returns a data frame with the data and the cluster assignments for each point:

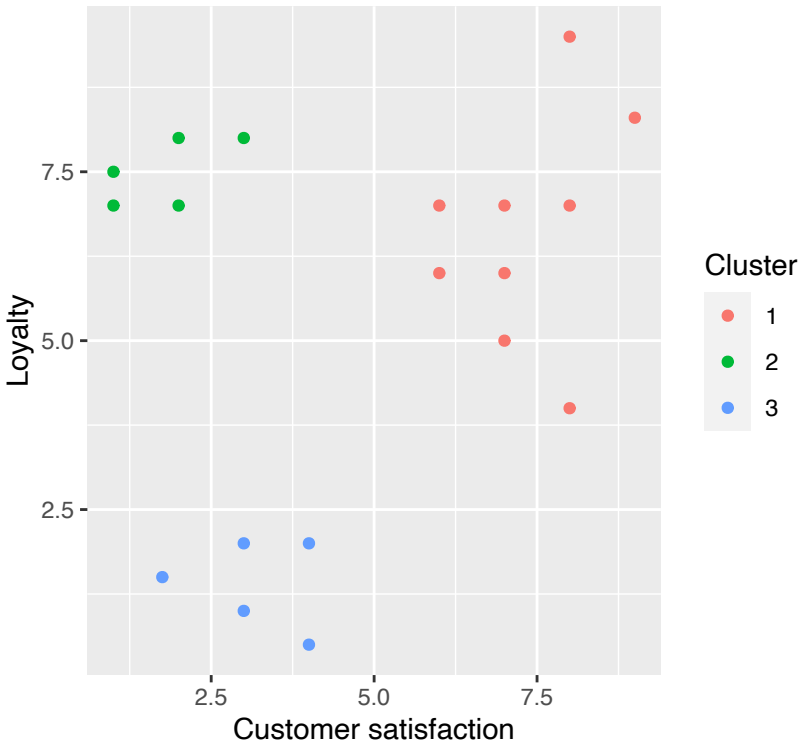
```
clustered_data <- augment(marketing_clust, marketing_data)
clustered_data
```

```
## # A tibble: 19 x 4
##    loyalty  csat label .cluster
##      <dbl> <dbl> <chr> <fct>
##  1     7    1    3     2
##  2     7.5  1    3     2
##  3     8    2    3     2
##  4     7    2    3     2
##  5     8    3    3     2
##  6     1.5  1.75 3     3
##  7     1    3    3     3
##  8     0.5  4    3     3
##  9     2    4    3     3
## 10     7    6    2     1
## 11     6    6    2     1
## 12     7    7    2     1
## 13     6    7    2     1
## 14     5    7    1     1
## 15     9.5  8    2     1
## 16     7    8    2     1
## 17     8.3  9    2     1
## 18     4    8    1     1
## 19     2    3    3     3
```

Now that we have this information in a tidy data frame, we can make a visualization of the cluster assignments for each point:

```
cluster_plot <- ggplot(clustered_data,
                       aes(x = csat, y = loyalty, colour = .cluster),
```

```
                      size=2) +
  geom_point() +
  labs(x = 'Customer satisfaction', y = 'Loyalty', colour = 'Cluster')
cluster_plot
```



As mentioned above, we also need to select K by finding where the "elbow" occurs in the plot of total WSSD versus number of clusters. We can obtain the total WSSD (`tot.withinss`) from our clustering using `broom`'s `glance` function. For example:

```
glance(marketing_clust)
```

```
## # A tibble: 1 x 4
##   totss tot.withinss betweenss  iter
##   <dbl>        <dbl>     <dbl> <int>
## 1  280.         40.3      239.     2
```

To calculate the total WSSD for a variety of Ks, we will create a data frame

with a column named `k` with rows containing each value of K we want to run K-means with (here, 1 to 9).

```
marketing_clust_ks <- tibble(k = 1:9)
marketing_clust_ks
```

```
## # A tibble: 9 x 1
##       k
##   <int>
## 1     1
## 2     2
## 3     3
## 4     4
## 5     5
## 6     6
## 7     7
## 8     8
## 9     9
```

Then we use `map` to apply the `kmeans` function to each K. However, we need to use `map` a little bit differently than we have before. This is because we need to iterate over `k`, which is the *second argument* to the `kmeans` function. In the past, we have used `map` only to iterate over values of the *first argument* of a function. Since that is the default, we could simply write `map(data_frame, function_name)`. This won't work here; we need to provide our data frame as the first argument to the `kmeans` function.

The solution is to create something called an *anonymous function*. An anonymous function is a function that has no name, unlike other functions you've seen so far (`kmeans`, `select`, etc). To do this we will write our map statement like this:

```
map(marketing_clust_ks, function(k) kmeans(marketing_data, k))
```

The anonymous function in the above call is `function(k) kmeans(marketing_data, k)`. This function takes a single argument (`k`) and evaluates `kmeans(marketing_data, k)`. Since `k` is the *first* (and only) argument to the function, we can use `map` just like we did before! The rest of the call above does just that – it passes each row of `marketing_clust_ks` to our anonymous function.

Below, we execute this `map` call inside of a `mutate` call on the `marketing_clust_ks` data frame and get a list column that contains a K-means clustering object for each value of K we had:

```
marketing_clust_ks <- tibble(k = 1:9) %>%
  mutate(marketing_clusts = map(k, function(ks) kmeans(marketing_data, ks)))
marketing_clust_ks
```

```
## # A tibble: 9 x 2
##       k marketing_clusts
##   <int> <list>
## 1     1 <kmeans>
## 2     2 <kmeans>
## 3     3 <kmeans>
## 4     4 <kmeans>
## 5     5 <kmeans>
## 6     6 <kmeans>
## 7     7 <kmeans>
## 8     8 <kmeans>
## 9     9 <kmeans>
```

Next, we use `map` again to apply `glance` to each of the K-means clustering objects to get the clustering statistics (including WSSD). The output of glance is a data frame, and so we get another list column. This results in a complex data frame with 3 columns, one for K, one for the K-means clustering objects, and one for the clustering statistics:

```
marketing_clust_ks <- tibble(k = 1:9) %>%
  mutate(marketing_clusts = map(k, ~kmeans(marketing_data, .x)),
         glanced = map(marketing_clusts, glance))
marketing_clust_ks
```

```
## # A tibble: 9 x 3
##       k marketing_clusts glanced
##   <int> <list>           <list>
## 1     1 <kmeans>         <tibble [1 x 4]>
## 2     2 <kmeans>         <tibble [1 x 4]>
## 3     3 <kmeans>         <tibble [1 x 4]>
## 4     4 <kmeans>         <tibble [1 x 4]>
## 5     5 <kmeans>         <tibble [1 x 4]>
## 6     6 <kmeans>         <tibble [1 x 4]>
## 7     7 <kmeans>         <tibble [1 x 4]>
## 8     8 <kmeans>         <tibble [1 x 4]>
## 9     9 <kmeans>         <tibble [1 x 4]>
```

Finally we extract the total WSSD from the `glanced` column. Given that each item in this column is a data frame, we will need to use the `unnest` function to unpack the data frames into simpler column data types.

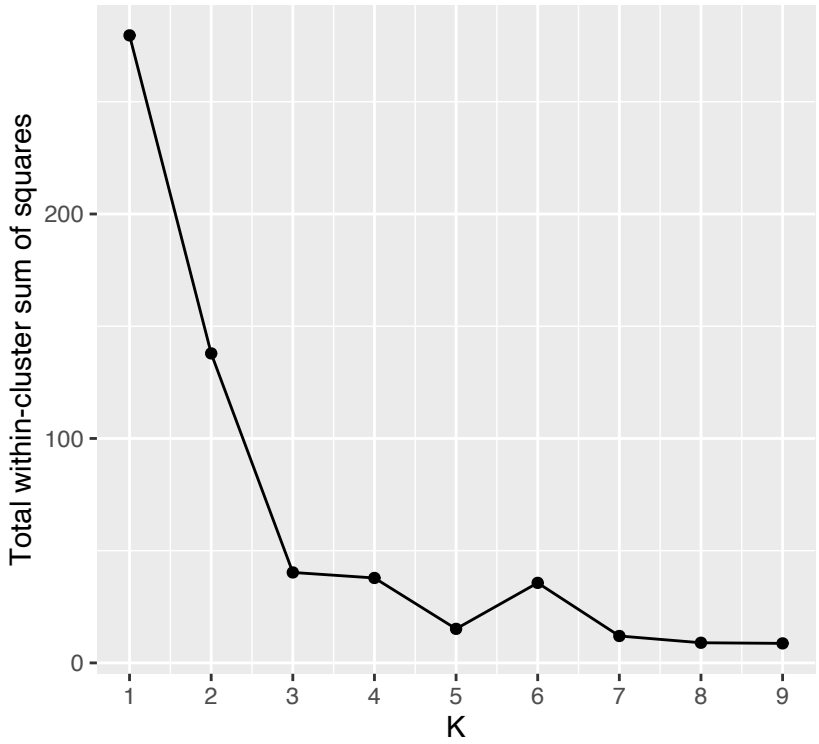
```
clustering_statistics <- marketing_clust_ks %>%
  unnest(glanced)

clustering_statistics
```

```
## # A tibble: 9 x 6
##       k marketing_clusts totss tot.withinss betweenss
##   <int> <list>           <dbl>        <dbl>     <dbl>
## 1     1 <kmeans>          280.       280.   -5.68e-14
## 2     2 <kmeans>          280.       138.    1.42e+ 2
## 3     3 <kmeans>          280.        40.3   2.39e+ 2
## 4     4 <kmeans>          280.        37.8   2.42e+ 2
## 5     5 <kmeans>          280.        15.2   2.64e+ 2
## 6     6 <kmeans>          280.        35.7   2.44e+ 2
## 7     7 <kmeans>          280.        12.0   2.68e+ 2
## 8     8 <kmeans>          280.         8.98  2.71e+ 2
## 9     9 <kmeans>          280.         8.70  2.71e+ 2
## # ... with 1 more variable: iter <int>
```

Now that we have `tot.withinss` and `k` as columns in a data frame, we can make a line plot and search for the "elbow" to find which value of K to use.

```
elbow_plot <- ggplot(clustering_statistics, aes(x = k, y = tot.withinss)) +
  geom_point() +
  geom_line() +
  xlab("K") +
  ylab("Total within-cluster sum of squares")+
  scale_x_continuous(breaks = 1:9)
elbow_plot
```



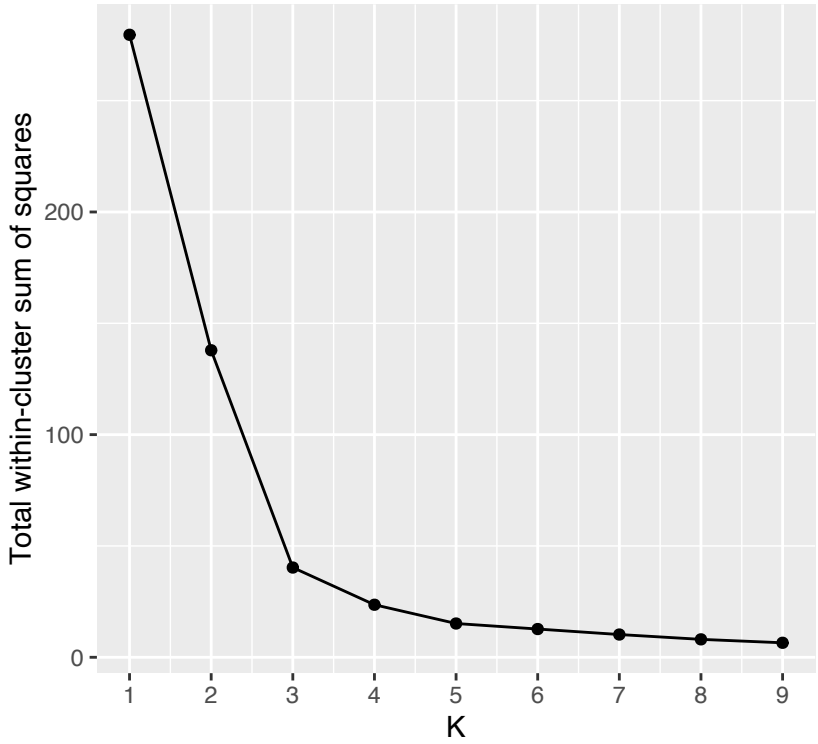
It looks like 3 clusters is the right choice for this data. But why is there a "bump" in the total WSSD plot here? Shouldn't total WSSD always decrease as we add more clusters? Technically yes, but remember: K-means can get "stuck" in a bad solution. Unfortunately, for K = 6 we had an unlucky initialization and found a bad clustering! We can help prevent finding a bad clustering by trying a few different random initializations via the `nstart` argument (here we use 10 restarts).

```
marketing_clust_ks <- tibble(k = 1:9) %>%
  mutate(marketing_clusts = map(k, ~kmeans(marketing_data, nstart = 10, .x)),
         glanced = map(marketing_clusts, glance))

clustering_statistics <- marketing_clust_ks %>%
  unnest(glanced)

elbow_plot <- ggplot(clustering_statistics, aes(x = k, y = tot.withinss)) +
  geom_point() +
  geom_line() +
  xlab("K") +
```

```
  ylab("Total within-cluster sum of squares")+
  scale_x_continuous(breaks = 1:9)
elbow_plot
```



10.7 Additional resources

For more about clustering and K-means, refer to pages 385-390 and 404-405 of Introduction to Statistical Learning with Applications in R[1] (2013). We have also linked a helpful companion video below:

[1] http://www-bcf.usc.edu/~gareth/ISL/ISLR%20Seventh%20Printing.pdf

11

Introduction to Statistical Inference

11.1 Overview

A typical data analysis task in practice is to draw conclusions about some unknown aspect of a population of interest based on observed data sampled from that population; we typically do not get data on the *entire* population. Data analysis questions regarding how summaries, patterns, trends, or relationships in a data set extend to the wider population are called *inferential questions*. This chapter will start with the fundamental ideas of sampling from populations and then introduce two common techniques in statistical inference: *point estimation* and *interval estimation*.

11.2 Chapter learning objectives

By the end of the chapter, students will be able to:

- Describe real-world examples of questions that can be answered with the statistical inference.
- Define common population parameters (e.g. mean, proportion, standard deviation) that are often estimated using sampled data, and estimate these from a sample.
- Define the following statistical sampling terms (population, sample, population parameter, point estimate, sampling distribution).
- Explain the difference between a population parameter and sample point estimate.
- Use R to draw random samples from a finite population.
- Use R to create a sampling distribution from a finite population.
- Describe how sample size influences the sampling distribution.
- Define bootstrapping.
- Use R to create a bootstrap distribution to approximate a sampling distribution.
- Contrast the bootstrap and sampling distributions.

11.3 Why do we need sampling?

Statistical inference can help us decide how quantities we observe in a subset of data relate to the same quantities in the broader population. Suppose a retailer is considering selling iPhone accessories, and they want to estimate how big the market might be. Additionally, they want to strategize how they can market their products on North American college and university campuses. This retailer might use statistical inference to answer the question:

What proportion of all undergraduate students in North America own an iPhone?

In the above question, we are interested in making a conclusion about *all* undergraduate students in North America; this is our **population**. In general, the population is the complete collection of individuals or cases we are interested in studying. Further, in the above question, we are interested in computing a quantity—the proportion of iPhone owners—based on the entire population. This is our **population parameter**. In general, a population parameter is a numerical characteristic of the entire population. To compute this number in the example above, we would need to ask every single undergraduate in North America whether or not they own an iPhone. In practice, directly computing population parameters is often time-consuming and costly, and sometimes impossible.

A more practical approach would be to collect measurements for a **sample**: a subset of individuals collected from the population. We can then compute a **sample estimate**—a numerical characteristic of the sample—that estimates the population parameter. For example, suppose we randomly selected 100 undergraduate students across North America (the sample) and computed the proportion of those students who own an iPhone (the sample estimate). In that case, we might suspect that that proportion is a reasonable estimate of the proportion of students who own an iPhone in the entire population.

Note that proportions are not the *only* kind of population parameter we might be interested in. Suppose an undergraduate student studying at the University of British Columbia in Vancouver, British Columbia, is looking for an apartment to rent. They need to create a budget, so they want to know something about studio apartment rental prices in Vancouver, BC. This student might use statistical inference to tackle the question:

What is the average price-per-month of studio apartment rentals in Vancouver, Canada?

The population consists of all studio apartment rentals in Vancouver, and the population parameter is the *average price-per-month*. Here we used the average as a measure of center to describe the "typical value" of studio apartment

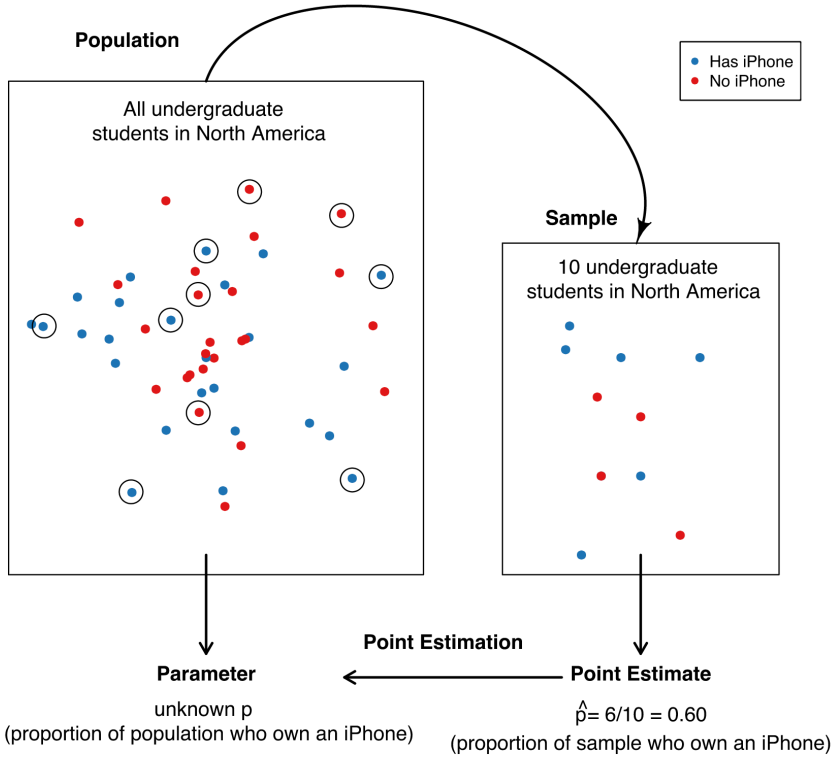


FIGURE 11.1: Population versus sample

rental prices. But even within this one example, we could also be interested in many other population parameters. For instance, we know that not every studio apartment rental in Vancouver will have the same price-per-month. The student might be interested in how much monthly prices vary and want to find a measure of the rentals' spread (or variability), such as the standard deviation. We might be interested in the fraction of studio apartment rentals that cost more than $1000 per month. And the list of population parameters we might want to calculate goes on. The question we want to answer will help us determine the parameter we want to estimate. If we were somehow able to observe the whole population of studio apartment rental offerings in Vancouver, we could compute each of these numbers exactly; therefore, these are all population parameters. There are many kinds of observations and population parameters that you will run into in practice, but in this chapter, we will focus on two settings:

1. Using categorical observations to estimate the proportion of each category
2. Using quantitative observations to estimate the average (or mean)

11.4 Sampling distributions

11.4.1 Sampling distributions for proportions

Let's start with an illustrative (and tasty!) example. Timbits are bite-sized doughnuts sold at Tim Hortons, a popular Canadian-based fast-food restaurant chain founded in Hamilton, Ontario, Canada.

FIGURE 11.2: A box of Timbits

Suppose we wanted to estimate the true proportion of chocolate doughnuts at Tim Hortons restaurants. Now, of course, we (the authors!) do not have access to the true population. So for this chapter, we created a fictitious box of 10,000 Timbits with two flavours—old-fashioned and chocolate—as our population, and use this to illustrate inferential concepts. Below we have a `tibble()` called `virtual_box` with a Timbit ID and flavour as our columns. We have also loaded our necessary packages: `tidyverse` and the `infer` package, which we will need to perform sampling later in the chapter.

```
library(tidyverse)
library(infer)
virtual_box
```

```
## # A tibble: 10,000 x 2
##    timbit_id flavour
##        <dbl> <fct>
##  1         1 chocolate
##  2         2 chocolate
##  3         3 chocolate
```

```
##  4         4 chocolate
##  5         5 old fashioned
##  6         6 old fashioned
##  7         7 chocolate
##  8         8 chocolate
##  9         9 old fashioned
## 10        10 chocolate
## # ... with 9,990 more rows
```

From our simulated box, we can see that the proportion of chocolate Timbits is 0.63. This value, 0.63, is the *population parameter*. Note that this parameter value is usually unknown in real data analysis problems.

```
virtual_box %>%
  group_by(flavour) %>%
  summarize(
    n = n(),
    proportion = n() / 10000
  )
```

```
## # A tibble: 2 x 3
##   flavour           n proportion
##   <fct>         <int>      <dbl>
## 1 old fashioned  3705      0.370
## 2 chocolate      6295      0.630
```

What would happen if we were to buy a box of 40 randomly-selected Timbits and count the number of chocolate Timbits (*i.e.,* take a random sample of size 40 from our Timbits population)? Let's use R to simulate this using our `virtual_box` population. We can do this using the `rep_sample_n` function from the `infer` package. The arguments of `rep_sample_n` are (1) the data frame (or tibble) to sample from, and (2) the size of the sample to take.

```
set.seed(1)
samples_1 <- rep_sample_n(tbl = virtual_box, size = 40)
choc_sample_1 <- summarize(samples_1,
  n = sum(flavour == "chocolate"),
  prop = sum(flavour == "chocolate") / 40
)
choc_sample_1
```

```
## # A tibble: 1 x 3
##   replicate     n  prop
##       <int> <int> <dbl>
## 1         1    29 0.725
```

Here we see that the proportion of chocolate Timbits in this random sample is 0.72. This value is our estimate — our best guess of our population parameter using this sample. Given that it is a single value that we are estimating, we often refer to it as a **point estimate**.

Now imagine we took another random sample of 40 Timbits from the population. Do you think we would get the same proportion? Let's try sampling from the population again and see what happens.

```
set.seed(2)
samples_2 <- rep_sample_n(virtual_box, size = 40)
choc_sample_2 <- summarize(samples_2,
  n = sum(flavour == "chocolate"),
  prop = sum(flavour == "chocolate") / 40
)
choc_sample_2
```

```
## # A tibble: 1 x 3
##   replicate     n  prop
##       <int> <int> <dbl>
## 1         1    27 0.675
```

Notice that we get a different value for our estimate this time. The proportion of chocolate Timbits in this sample is 0.68. If we were to do this again, another random sample could also give a different result. Estimates vary from sample to sample due to **sampling variability**.

But just how much should we expect the estimates of our random samples to vary? In order to understand this, we will simulate taking more samples of size 40 from our population of Timbits, and calculate the proportion of chocolate Timbits in each sample. We can then visualize the distribution of sample proportions we calculate. The distribution of the estimate for all possible samples of a given size (which we commonly refer to as n) from a population is called a **sampling distribution**. The sampling distribution will help us see how much we would expect our sample proportions from this population to vary for samples of size 40. Below we again use the `rep_sample_n` to take samples of size 40 from our population of Timbits, but we set the `reps` argument to specify the number of samples to take, here 15,000. We will use the function `head()` to see the first few rows and `tail()` to see the last few rows of our `samples` data frame.

```
samples <- rep_sample_n(virtual_box, size = 40, reps = 15000)
head(samples)
```

```
## # A tibble: 6 x 3
## # Groups:   replicate [1]
```

```
##   replicate timbit_id flavour
##       <int>     <dbl> <fct>
## 1         1      9054 chocolate
## 2         1      4322 old fashioned
## 3         1      1685 chocolate
## 4         1      3958 chocolate
## 5         1      2765 old fashioned
## 6         1       358 old fashioned
```

```
tail(samples)
```

```
## # A tibble: 6 x 3
## # Groups:   replicate [1]
##   replicate timbit_id flavour
##       <int>     <dbl> <fct>
## 1     15000      4633 old fashioned
## 2     15000       552 chocolate
## 3     15000      7998 old fashioned
## 4     15000      8649 chocolate
## 5     15000      2974 chocolate
## 6     15000      7811 old fashioned
```

Notice the column `replicate` is indicating the replicate, or sample, with which each Timbit belongs. Since we took 15,000 samples of size 40, there are 15,000 replicates. Now that we have taken 15,000 samples, to create a sampling distribution of sample proportions for samples of size 40, we need to calculate the proportion of chocolate Timbits for each sample, $\hat{p}_{\text{chocolate}}$:

```
sample_estimates <- samples %>%
  group_by(replicate) %>%
  summarise(sample_proportion = sum(flavour == "chocolate") / 40)
head(sample_estimates)
```

```
## # A tibble: 6 x 2
##   replicate sample_proportion
##       <int>             <dbl>
## 1         1             0.625
## 2         2             0.675
## 3         3             0.7
## 4         4             0.675
## 5         5             0.45
## 6         6             0.425
```

```
tail(sample_estimates)
```

```
## # A tibble: 6 x 2
##   replicate sample_proportion
##       <int>             <dbl>
## 1     14995             0.575
## 2     14996             0.6
## 3     14997             0.45
## 4     14998             0.675
## 5     14999             0.7
## 6     15000             0.525
```

Now that we have calculated the proportion of chocolate Timbits for each sample, $\hat{p}_{\text{chocolate}}$, we can visualize the sampling distribution of sample proportions for samples of size 40:

```
sampling_distribution <- ggplot(sample_estimates, aes(x = sample_proportion)) +
  geom_histogram(fill = "dodgerblue3", color = "lightgrey", bins = 12) +
  xlab("Sample proportions")
sampling_distribution
```

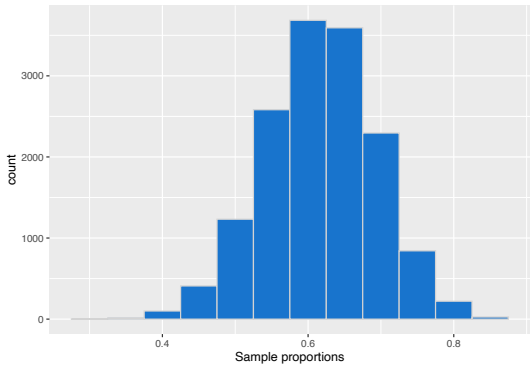


FIGURE 11.3: Sampling distribution of the sample proportion for sample size 40

The sampling distribution appears to be bell-shaped with one peak. It is centered around 0.6 and the sample proportions range from about 0.3 to about 0.9. In fact, we can calculate the mean of the sample proportions.

```
sample_estimates %>%
  summarise(mean = mean(sample_proportion))
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1 0.629
```

We notice that the sample proportions are centred around the population proportion value, 0.63! In general, the mean of the distribution of $\hat{p}$ should be equal to p, which is good because that means the sample proportion is neither an overestimate nor an underestimate of the population proportion.

So what can we learn from this sampling distribution? This distribution tells us what we might expect from proportions from samples of size 40 when our population proportion is 0.63. In practice, we usually don't know the proportion of our population, but if we can use what we know about the sampling distribution, we can use it to make inferences about our population when we only have a single sample.

11.4.2 Sampling distributions for means

In the previous section, our variable of interest—Timbit flavour—was *categorical*, and the population parameter of interest was the proportion of chocolate Timbits. As mentioned in the introduction to this chapter, there are many choices of population parameter for each type of observed variable. What if we wanted to infer something about a population of *quantitative* variables instead? For instance, a traveller visiting Vancouver, BC may wish to know about the prices of staying somewhere using Airbnb, an online marketplace for arranging places to stay. Particularly, they might be interested in estimating the population mean price per night of Airbnb listings in Vancouver, BC. This section will study the case where we are interested in the population mean of a quantitative variable.

We will look at an example using data from Inside Airbnb[1]. The data set contains Airbnb listings for Vancouver, Canada, in September 2020. Let's imagine (for learning purposes) that our data set represents the population of all Airbnb rental listings in Vancouver, and we are interested in the population mean price per night. Our data contains an ID number, neighbourhood, type of room, the number of people the rental accommodates, number of bathrooms, bedrooms, beds, and the price per night.

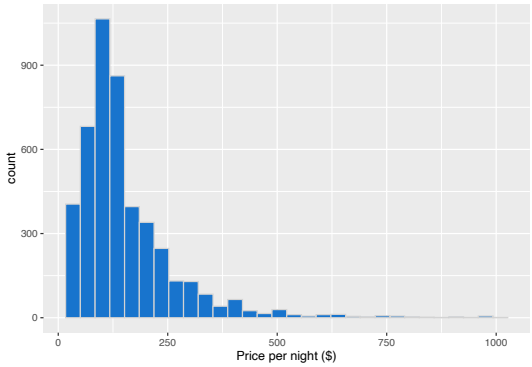
```
## # A tibble: 6 x 8
##      id neighbourhood room_type accommodates bathrooms
##   <int> <chr>         <chr>            <dbl> <chr>
## 1     1 Downtown      Entire h~            5 2 baths
## 2     2 Downtown Eas~ Entire h~            4 2 baths
## 3     3 West End      Entire h~            2 1 bath
```

[1] http://insideAirbnb.com/

```
## 4     4 Kensington-C~ Entire h~            2 1 bath
## 5     5 Kensington-C~ Entire h~            4 1 bath
## 6     6 Hastings-Sun~ Entire h~            4 1 bath
## # ... with 3 more variables: bedrooms <dbl>,
## #   beds <dbl>, price <dbl>
```

We can visualize the population distribution of the price per night with a histogram.

```
population_distribution <- ggplot(airbnb, aes(x = price)) +
  geom_histogram(fill = "dodgerblue3", color = "lightgrey") +
  xlab("Price per night ($)")
population_distribution
```



\begin{figure} \caption{Population distribution of price per night ($) for all Airbnb listings in Vancouver, Canada} \end{figure} We see that the distribution has one peak and is skewed—most of the listings are less than $250 per night, but a small proportion of listings cost more than that, creating a long tail on the histogram's right side.

We can also calculate the population mean, the average price per night for all the Airbnb listings.

```
population_parameters <- airbnb %>%
  summarize(pop_mean = mean(price))
population_parameters
```

```
## # A tibble: 1 x 1
##   pop_mean
##      <dbl>
## 1     155.
```

The price per night of all Airbnb rentals in Vancouver, BC is $154.51, on

average. This value is our population parameter since we are calculating it using the population data.

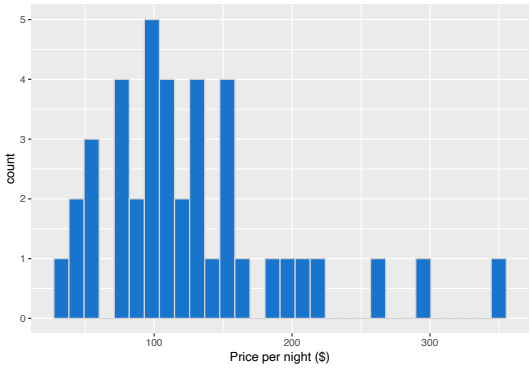
Suppose that we did not have access to the population data, yet we still wanted to estimate the mean price per night. We could answer this question by taking a random sample of as many Airbnb listings as we had time to, let's say we could do this for 40 listings. What would such a sample look like?

Let's take advantage of the fact that we do have access to the population data and simulate taking one random sample of 40 listings in R, again using `rep_sample_n`. After doing this we create a histogram to visualize the distribution of observations in the sample, and calculate the mean of our sample. This number is a **point estimate** for the mean of the full population.

```
sample_1 <- airbnb %>%
  rep_sample_n(40)
head(sample_1)
```

```
## # A tibble: 6 x 9
## # Groups:   replicate [1]
##   replicate    id neighbourhood room_type accommodates
##       <int> <int> <chr>         <chr>             <dbl>
## 1         1  2317 Kensington-C~ Entire h~            2
## 2         1  1028 Riley Park    Entire h~            6
## 3         1  2487 South Cambie  Entire h~            2
## 4         1  2644 Downtown      Entire h~            2
## 5         1  3059 Downtown Eas~ Entire h~            9
## 6         1  2507 Fairview      Entire h~            4
## # ... with 4 more variables: bathrooms <chr>,
## #   bedrooms <dbl>, beds <dbl>, price <dbl>
```

```
sample_distribution <- ggplot(sample_1, aes(price)) +
  geom_histogram(fill = "dodgerblue3", color = "lightgrey") +
  xlab("Price per night ($)")
sample_distribution
```



\begin{figure} \caption{Distribution of price per night ($) for sample of 40 Airbnb listings} \end{figure}

```
estimates <- sample_1 %>%
  summarize(sample_mean = mean(price))
estimates
```

```
## # A tibble: 1 x 2
##   replicate sample_mean
##       <int>       <dbl>
## 1         1        127.
```

Recall that the population mean was $154.51. We see that our point estimate for the mean is $127.35. So our estimate was actually quite close to the population parameter: the mean was about 17.6% off. Note that in practice, we usually cannot compute the accuracy of the estimate, since we do not have access to the population parameter; if we did, we wouldn't need to estimate it!

Also recall from the previous section that the point estimate can vary; if we took another random sample from the population, then the value of our estimate may change. So then did we just get lucky with our point estimate above? How much does our estimate vary across different samples of size 40 in this example? Again, since we have access to the population, we can take many samples and plot the **sampling distribution** of sample means for samples of size 40 to get a sense for this variation. In this case, we'll use 15,000 samples of size 40.

```
samples <- rep_sample_n(airbnb, size = 40, reps = 15000)
head(samples)
```

```
## # A tibble: 6 x 9
## # Groups:   replicate [1]
##   replicate    id neighbourhood room_type accommodates
```

```
##       <int> <int> <chr>         <chr>            <dbl>
## 1         1   896 Downtown      Entire h~            6
## 2         1  2723 Kensington-C~ Entire h~            4
## 3         1  3130 Fairview      Entire h~            3
## 4         1  2103 Kitsilano     Entire h~            2
## 5         1  4242 Downtown      Entire h~            2
## 6         1   610 Grandview-Wo~ Entire h~            6
## # ... with 4 more variables: bathrooms <chr>,
## #   bedrooms <dbl>, beds <dbl>, price <dbl>
```

```
sample_estimates <- samples %>%
  group_by(replicate) %>%
  summarise(sample_mean = mean(price))
head(sample_estimates)
```

```
## # A tibble: 6 x 2
##   replicate sample_mean
##       <int>       <dbl>
## 1         1        142.
## 2         2        129.
## 3         3        114.
## 4         4        186.
## 5         5        137.
## 6         6        178.
```

```
sampling_distribution_40 <- ggplot(sample_estimates, aes(x = sample_mean)) +
  geom_histogram(fill = "dodgerblue3", color = "lightgrey") +
  xlab("Sample mean price per night ($)")
sampling_distribution_40
```

Here we see that the sampling distribution of the mean has one peak and is bell-shaped. Most of the estimates are between about $140 and $170; but there are a good fraction of cases outside this range (i.e., where the point estimate was not close to the population parameter). So it does indeed look like we were quite lucky when we estimated the population mean with only 17.6% error. Let's visualize the population distribution, distribution of the sample, and the sampling distribution on one plot to compare them.

Given that there is quite a bit of variation in the sampling distribution of the sample mean—i.e., the point estimate that we obtain is not very reliable—is there any way to improve the estimate? One way to improve a point estimate is to take a *larger* sample. To illustrate what effect this has, we will take many samples of size 20, 50, 100, and 500, and plot the sampling distribution of the sample mean below.

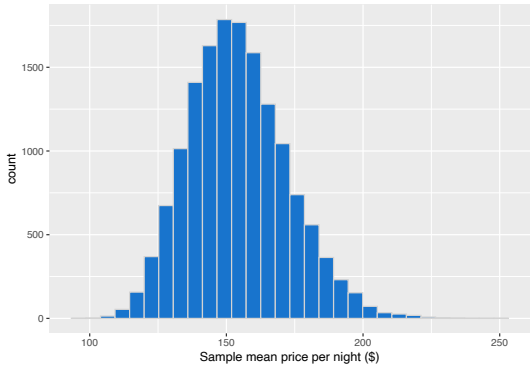


FIGURE 11.4: Sampling distribution of the sample means for sample size of 40

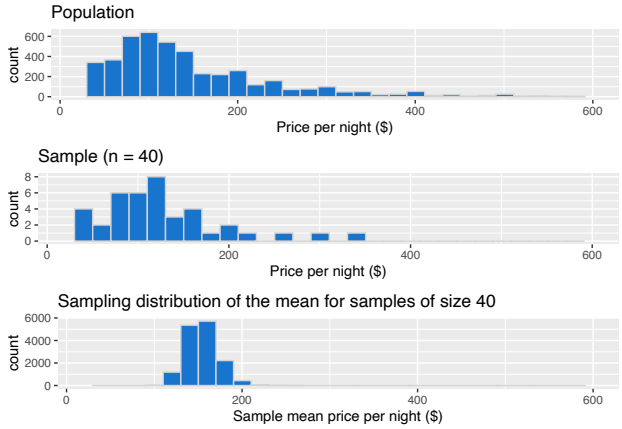


FIGURE 11.5: Comparision of population distribution, sample distribution and sampling distribution

Based on the visualization, two points about the sample mean become clear. First, the mean of the sample mean (across samples) is equal to the population mean. Second, increasing the size of the sample decreases the spread (i.e., the variability) in the sample mean point estimate of the population mean. Therefore, a larger sample size results in a more reliable point estimate of the population parameter.

11.4.3 Summary

1. A *point estimate* is a single value computed using a sample from a population (e.g. a mean or proportion)

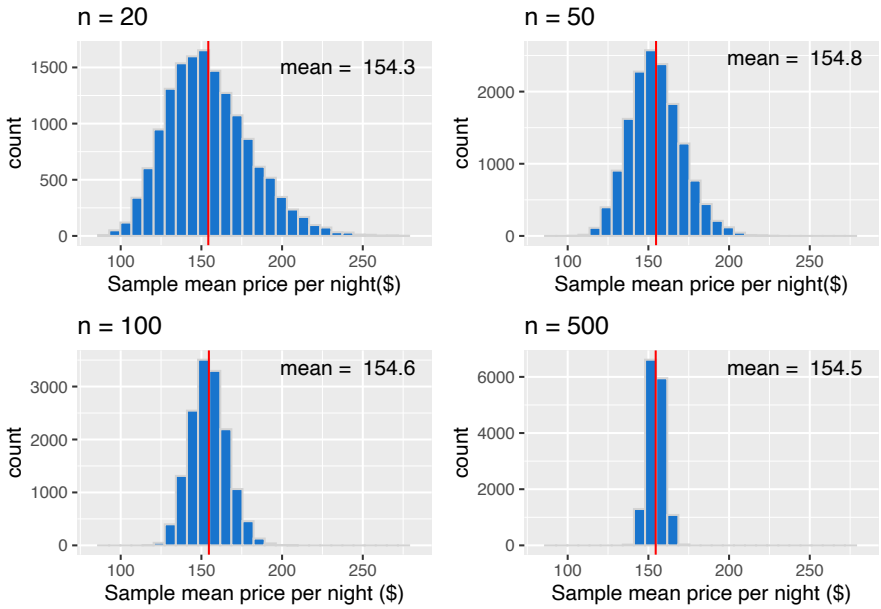


FIGURE 11.6: Comparision of sampling distributions

2. The *sampling distribution* of an estimate is the distribution of the estimate for all possible samples of a fixed size from the same population.
3. The sample means and proportions calculated from samples are centered around the population mean and proportion, respectively.
4. The spread of the sampling distribution is related to the sample size. As the sample size increases, the spread of the sampling distribution decreases.
5. The shape of the sampling distribution is usually bell-shaped with one peak and centred at the population mean or proportion.

Why all this emphasis on sampling distributions?

Usually, we don't have access to the population data, so we cannot construct the sampling distribution as we did in this section. As we saw, our sample estimate's value will likely not equal the population parameter value exactly. We saw from the sampling distribution just how much our estimates can vary. So reporting a single *point estimate* for the population parameter alone may not be enough. Using simulations, we can see patterns of the sample estimate's sampling distribution would look like for a sample of a given size. We can use these patterns to approximate the sampling distribution when we only have one sample, which is the realistic case. If we can "predict" what the sampling distribution would look like for a sample, we could construct a range of values

we think the population parameter's value might lie. We can use our single sample and its properties that influence sampling distributions, such as the spread and sample size, to approximate the sampling distribution as best as we can. There are several methods to do this; however, in this book, we will use the bootstrap method to do this, as we will see in the next section.

11.5 Bootstrapping

11.5.1 Overview

We saw in the previous section that we could compute a **point estimate** of a population parameter using a sample of observations from the population. And since we had access to the population, we could evaluate how accurate the estimate was, and even get a sense for how much the estimate would vary for different samples from the population. But in real data analysis settings, we usually have *just one sample* from our population, and do not have access to the population itself. So how do we get a sense for how variable our point estimate is when we only have one sample to work with? In this section, we will discuss **interval estimation** and construct **confidence intervals** using just a single sample from a population. A confidence interval is a range of plausible values for our population parameter.

Here is the key idea. First, if you take a big enough sample, it *looks like* the population. Notice the histograms' shapes for samples of different sizes taken from the population in the picture below. We see that for a large enough sample, the sample's distribution looks like that of the population.

In the previous section, we took many samples of the same size *from our population* to get a sense for the variability of a sample estimate. But if our sample is big enough that it looks like our population, we can pretend that our sample *is* the population, and take more samples (with replacement) of the same size from it instead! This very clever technique is called **the bootstrap**. Note that by taking many samples from our single, observed sample, we do not obtain the true sampling distribution, but rather an approximation that we call **the bootstrap distribution**.

> Note that we need to sample *with* replacement when using the bootstrap. Otherwise, if we had a sample of size n, and obtained a sample from it of size n *without* replacement, it would just return our original sample.

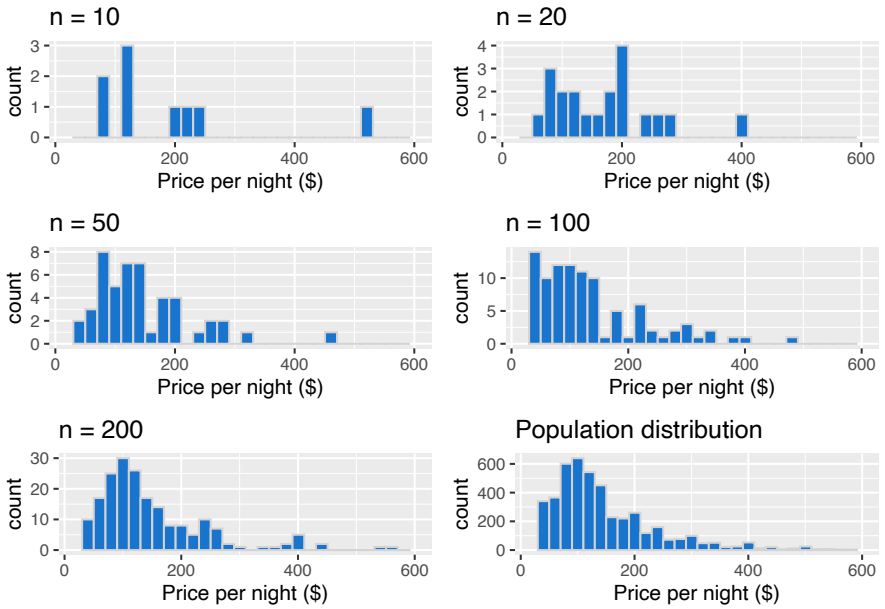


FIGURE 11.7: Comparision of samples of different sizes from the population

---

This section will explore how to create a bootstrap distribution from a single sample using R. For a sample of size n, the process we will go through is as follows:

1. Randomly select an observation from the original sample, which was drawn from the population
2. Record the observation's value
3. Replace that observation
4. Repeat steps 1 - 3 (sampling *with* replacement) until you have n observations, which form a bootstrap sample
5. Calculate the bootstrap point estimate (e.g., mean, median, proportion, slope, etc.) of the n observations in your bootstrap sample
6. Repeat steps (1) - (5) many times to create a distribution of point estimates (the bootstrap distribution)
7. Calculate the plausible range of values around our observed point estimate

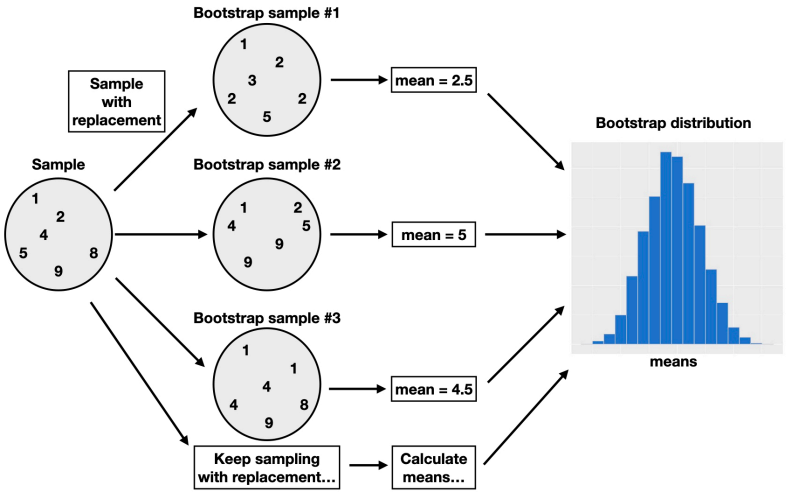


FIGURE 11.8: Overview of the bootstrap process

11.5.2 Bootstrapping in R

Let's continue working with our Airbnb data. Once again, let's say we are interested in estimating the population mean price per night of all Airbnb listings in Vancouver, Canada using a single sample we collected of size 40.

To simulate doing this in R, we will use `rep_sample_n` to take a random sample from from our population. In real life we wouldn't do this step in R, we would instead simply load the data into R, that we, or our collaborators collected.

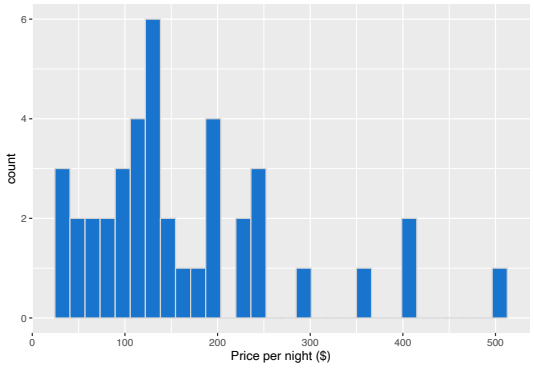
After we have our sample, we will visualize it's distribution and calculate our point estimate, the sample mean.

```
one_sample <- airbnb %>%
  rep_sample_n(40) %>%
  ungroup() %>% # ungroup the data frame
  select(price) # drop the replicate column
head(one_sample)
```

```
## # A tibble: 6 x 1
##   price
##   <dbl>
## 1   250
## 2   106
## 3   150
## 4   357
## 5    50
```

```
## 6   110
```

```
one_sample_dist <- ggplot(one_sample, aes(price)) +
  geom_histogram(fill = "dodgerblue3", color = "lightgrey") +
  xlab("Price per night ($)")
one_sample_dist
```



\begin{figure} \caption{Histogram of price per night ($) for one sample of size 40} \end{figure}

```
one_sample_estimates <- one_sample %>%
  summarise(sample_mean = mean(price))
one_sample_estimates
```

```
## # A tibble: 1 x 1
##   sample_mean
##         <dbl>
## 1        166.
```

The sample distribution is skewed with a few observations out to the right. The mean of the sample is $165.62. Remember, in practice, we usually only have one sample from the population. So this sample and estimate are the only data we can work with.

We now perform steps (1) - (5) listed above to generate a single bootstrap sample in R using the sample we just took, and calculate the bootstrap estimate for that sample. We will use the `rep_sample_n` function as we did when we were creating our sampling distribution. Since we want to sample with replacement, we change the argument for `replace` from its default value of `FALSE` to `TRUE`.

```
boot1 <- one_sample %>%
  rep_sample_n(size = 40, replace = TRUE, reps = 1)
head(boot1)
```

```
## # A tibble: 6 x 2
## # Groups:   replicate [1]
##   replicate price
##       <int> <dbl>
## 1         1  201
## 2         1  199
## 3         1  127.
## 4         1   85
## 5         1  169
## 6         1   60
```

```
boot1_dist <- ggplot(boot1, aes(price)) +
  geom_histogram(fill = "dodgerblue3", color = "lightgrey") +
  xlab("Price per night ($)")

boot1_dist
```

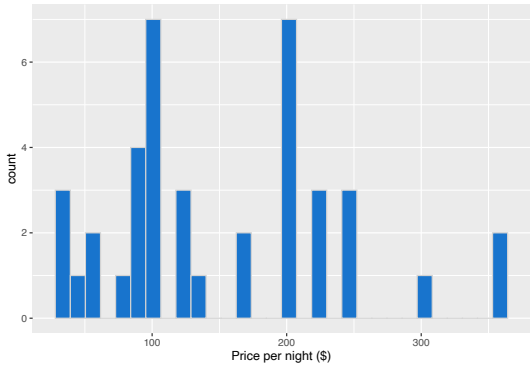


FIGURE 11.9: Bootstrap distribution

```
summarise(boot1, mean = mean(price))
```

```
## # A tibble: 1 x 2
##   replicate  mean
##       <int> <dbl>
## 1         1  152.
```

Notice that our bootstrap distribution has a similar shape to the original sample distribution. Though the shapes of the distributions are similar, they are not identical. You'll also notice that the original sample mean and the bootstrap sample mean differ. How might that happen? Remember that we are sampling with replacement from the original sample, so we don't end up

with the same sample values again. We are trying to mimic drawing another sample from the population without actually having to do that.

Let's now take 15,000 bootstrap samples from the original sample we drew from the population (`one_sample`) using `rep_sample_n` and calculate the means for each of those replicates. Recall that this assumes that `one_sample` *looks like* our original population; but since we do not have access to the population itself, this is often the best we can do.

```
boot15000 <- one_sample %>%
  rep_sample_n(size = 40, replace = TRUE, reps = 15000)
head(boot15000)
```

```
## # A tibble: 6 x 2
## # Groups:   replicate [1]
##   replicate price
##       <int> <dbl>
## 1         1   200
## 2         1   176
## 3         1   105
## 4         1   105
## 5         1   105
## 6         1   132
```

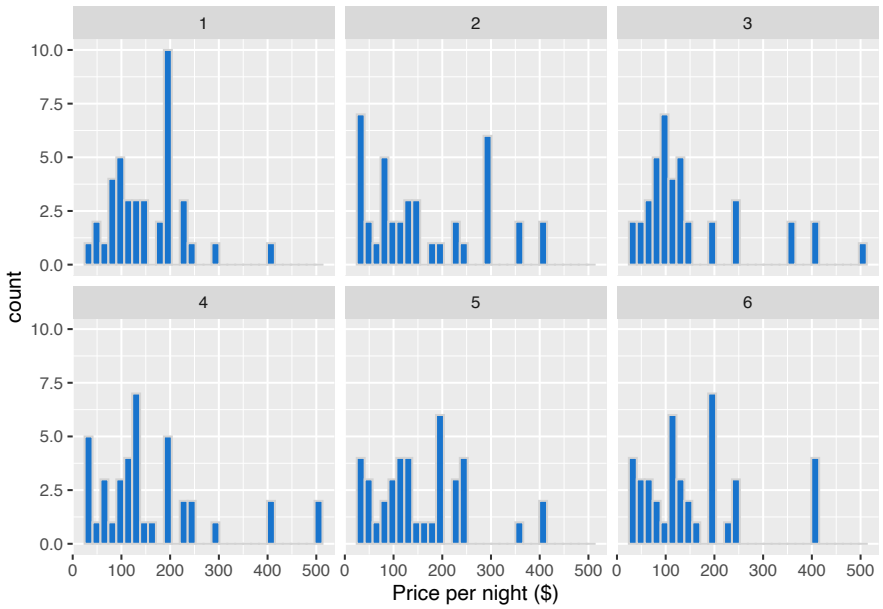
```
tail(boot15000)
```

```
## # A tibble: 6 x 2
## # Groups:   replicate [1]
##   replicate price
##       <int> <dbl>
## 1     15000   357
## 2     15000    49
## 3     15000   115
## 4     15000   169
## 5     15000   145
## 6     15000   357
```

Let's take a look at histograms of the first six replicates of our bootstrap samples.

```
six_bootstrap_samples <- boot15000 %>%
  filter(replicate <= 6)
ggplot(six_bootstrap_samples, aes(price)) +
  geom_histogram(fill = "dodgerblue3", color = "lightgrey") +
```

```
  xlab("Price per night ($)") +
  facet_wrap(~replicate)
```



We see in the graph above how the bootstrap samples differ. We can also calculate the sample mean for each of these six replicates.

```
six_bootstrap_samples %>%
  group_by(replicate) %>%
  summarize(mean = mean(price))
```

```
## # A tibble: 6 x 2
##   replicate  mean
##       <int> <dbl>
## 1         1  154.
## 2         2  162.
## 3         3  151.
## 4         4  163.
## 5         5  158.
## 6         6  156.
```

We can see that the bootstrap sample distributions and the sample means are different. This is because we are sampling with replacement. We will now calculate point estimates for our 15,000 bootstrap samples and generate a bootstrap distribution of our point estimates. The bootstrap distribution sug-

gests how we might expect our point estimate to behave if we took another sample.

```
boot15000_means <- boot15000 %>%
  group_by(replicate) %>%
  summarize(mean = mean(price))
head(boot15000_means)
```

```
## # A tibble: 6 x 2
##   replicate  mean
##       <int> <dbl>
## 1         1  154.
## 2         2  162.
## 3         3  151.
## 4         4  163.
## 5         5  158.
## 6         6  156.
```

```
tail(boot15000_means)
```

```
## # A tibble: 6 x 2
##   replicate  mean
##       <int> <dbl>
## 1     14995  155.
## 2     14996  148.
## 3     14997  139.
## 4     14998  156.
## 5     14999  158.
## 6     15000  176.
```

```
boot_est_dist <- ggplot(boot15000_means, aes(x = mean)) +
  geom_histogram(fill = "dodgerblue3", color = "lightgrey") +
  xlab("Sample mean price per night ($)")
```

Let's compare our bootstrap distribution with the true sampling distribution (taking many samples from the population).

There are two essential points that we can take away from these plots. First, the shape and spread of the true sampling distribution and the bootstrap distribution are similar; the bootstrap distribution lets us get a sense of the point estimate's variability. The second important point is that the means of these two distributions are different. The sampling distribution is centred at $154.51, the population mean value. However, the bootstrap distribution is centred at the original sample's mean price per night, $165.56. Because we

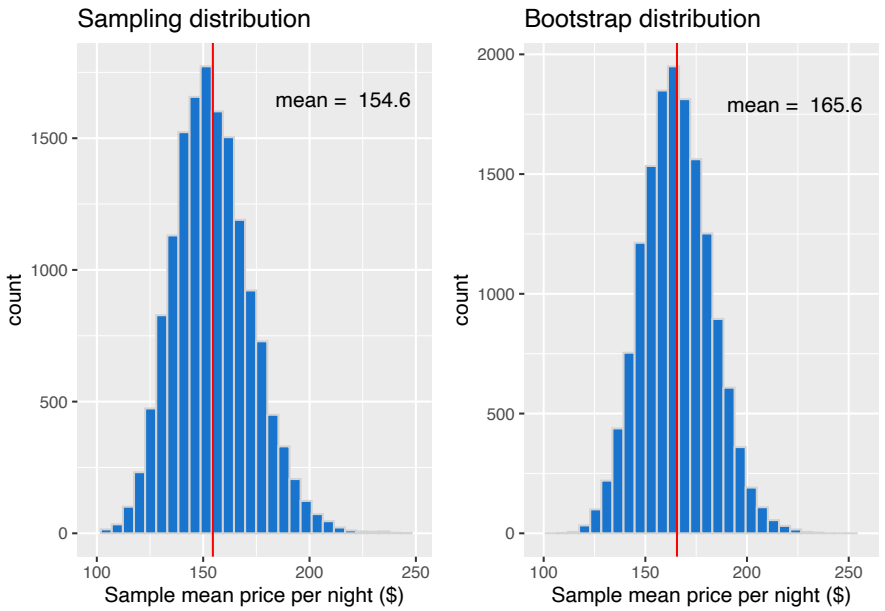


FIGURE 11.10: Comparison of distribution of the bootstrap sample means and sampling distribution

are resampling from the original sample repeatedly, we see that the bootstrap distribution is centred at the original sample's mean value (unlike the sampling distribution of the sample mean, which is centred at the population parameter value).

The idea here is that we can use this distribution of bootstrap sample means to approximate the sampling distribution of the sample means when we only have one sample. Since the bootstrap distribution pretty well approximates the sampling distribution spread, we can use the bootstrap spread to help us develop a plausible range for our population parameter along with our estimate!

11.5.3 Using the bootstrap to calculate a plausible range

Now that we have constructed our bootstrap distribution let's use it to create an approximate bootstrap confidence interval, a range of plausible values for the population mean. We will build a 95% percentile bootstrap confidence interval and find the range of values that cover the middle 95% of the bootstrap distribution. A 95% confidence interval means that if we were to repeat the sampling process and calculate 95% confidence intervals each time and repeat this process many times, then 95% of the intervals would capture the popu-

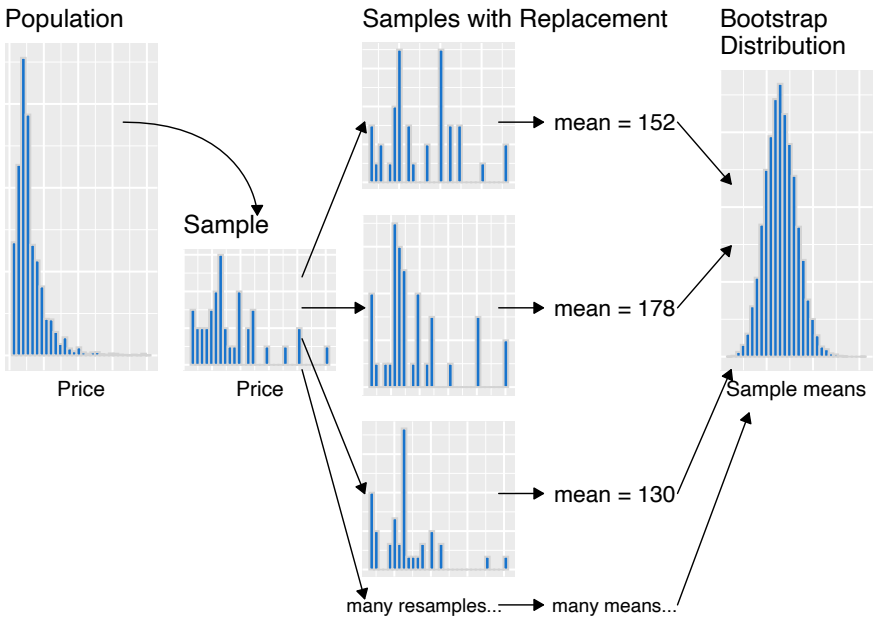


FIGURE 11.11: Summary of bootstrapping process

lation parameter's value. Note that there's nothing particularly special about 95%, we could have used other confidence levels, such as 90% or 99%. There is a balance between our level of confidence and precision. A higher confidence level corresponds to a wider range of the interval, and a lower confidence level corresponds to a narrower range. Therefore the level we choose is based on what chance we are willing to take of being wrong based on the implications of being wrong for our application. In general, we choose confidence levels to be comfortable with our level of uncertainty, but not so strict that the interval is unhelpful. For instance, if our decision impacts human life and the implications of being wrong are deadly, we may want to be very confident and choose a higher confidence level.

To calculate our 95% percentile bootstrap confidence interval, we will do the following:

1. Arrange the observations in the bootstrap distribution in ascending order
2. Find the value such that 2.5% of observations fall below it (the 2.5% percentile). Use that value as the lower bound of the interval
3. Find the value such that 97.5% of observations fall below it (the 97.5% percentile). Use that value as the upper bound of the interval

To do this in R, we can use the `quantile()` function:

```
bounds <- boot15000_means %>%
  select(mean) %>%
  pull() %>%
  quantile(c(0.025, 0.975))
bounds
```

```
##  2.5% 97.5%
## 134.1 200.3
```

Our interval, $134.08 to $200.28, captures the middle 95% of the sample mean prices in the bootstrap distribution. We can visualize the interval on our distribution in the picture below.

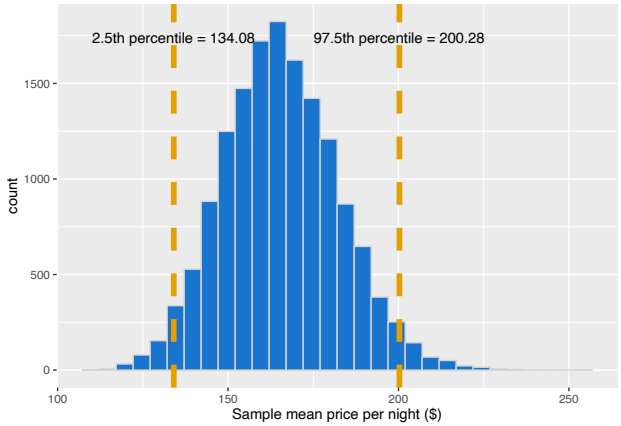


FIGURE 11.12: Distribution of the bootstrap sample means with percentile lower and upper bounds

To finish our estimation of the population parameter, we would report the point estimate and our confidence interval's lower and upper bounds. Here the sample mean price-per-night of 40 Airbnb listings was $127.35, and we are 95% "confident" that the true population mean price-per-night for all Airbnb listings in Vancouver is between $(134.08, 200.28).

Notice that our interval does indeed contain the true population mean value, $154.51! However, in practice, we would not know whether our interval captured the population parameter or not because we usually only have a single sample, not the entire population. However, this is the best we can do when we only have one sample!

This chapter is only the beginning of the journey into statistical inference. We can extend the concepts learned here to do much more than report point estimates and confidence intervals, such as hypothesis testing for differences

between populations, tests for associations between variables, and so much more! We have just scratched the surface of statistical inference; however, the material presented here will serve as the foundation for more advanced statistical techniques you may learn about in the future!

11.6 Additional resources

For more about statistical inference and bootstrapping, refer to

- Chapters 7 - 8 of Modern Dive: Statistical Inference via Data Science[2] by Chester Ismay and Albert Y. Kim
- Chapters 4 - 7 of OpenIntro Statistics - Fourth Edition[3] by David M. Diez, Christopher D. Barr and Mine Cetinkaya-Rundel

[2] https://moderndive.com/
[3] https://www.openintro.org/

12

Moving to your own machine

12.1 Overview

Throughout this book, we have assumed that you are working on a web-based platform (e.g., JupyterHub) that already has Jupyter, R, a number of R packages, and Git set up and ready to use. In this chapter, you'll learn how to install all of that software on your own computer in case you don't have a preconfigured JupyterHub available to you.

12.2 Chapter learning objectives

By the end of the chapter, students will be able to:

- install Git and the miniconda Python distribution
- install and launch a local instance of JupyterLab with the R kernel
- download files from a JupyterHub for later local use

12.3 Installing software on your own computer

In this section we will provide instructions for installing the software required by this book on our own computer. Given that installation instructions can vary widely based on the computer setup we have created instructions for multiple operating systems. In particular, the installation instructions below have been verified to work on a computer that:

- runs one of the following operating systems: MacOS 10.15.X (Catalina); Ubuntu 20.04; Windows 10, version 2004.
- can connect to networks via a wireless connection
- uses a 64-bit CPU
- uses English as the default language

For macOS users only: Apple recently changed the default shell in the terminal to Zsh. However, the programs we need work better with the Bash shell. Thus, we recomend you change the default shell to Bash by opening the terminal (how to video[1]) and typing:

```
chsh -s /bin/bash
```

You will have to quit all instances of open terminals and then restart the terminal for this to take effect.

12.3.1 Git

As shown in the version control chapter, Git is a very useful tool for version controlling your projects, as well as sharing your work with others.

Windows: To install Git on Windows go to `https://git-scm.com/download/win` and download the windows version of git. Once the download has finished, run the installer and accept the default configuration for all pages.

MacOS: To install Git on Mac OS open the terminal and type the following command:

```
xcode-select --install
```

Ubuntu: To install Git on Ubuntu open the terminal and type the following commands:

```
sudo apt update
sudo apt install git
```

12.3.2 Miniconda

To run Jupyter notebooks on our computers we will need to install a program similar to the one we used as our web-based platform. One such program is JupyterLab. But JupyterLab relies on Python; we can install this via the miniconda Python package distribution[2].

Windows: To install miniconda on Windows, download the Python 3.8 64-bit version from here[3]. Once the download has finished, run the installer and accept the default configuration for all pages. After installation, you can open the Anaconda Prompt by opening the Start Menu and searching for the program called "Anaconda Prompt (miniconda3)". When this opens you will see a prompt similar to `(base) C:\Users\your_name`.

MacOS: To install miniconda on MacOS, download the Python 3.8 64-bit

[1] https://youtu.be/5AJbWEWwnbY
[2] https://docs.conda.io/en/latest/miniconda.html
[3] https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.exe

version from here[4]. After the download has finished, run the installer and accept the default configuration for all pages.

Ubuntu: To install miniconda on Ubuntu, we first download the Python 3.8 64-bit version from here[5]. After the download has finished, open the terminal and execute the following commands:

```
bash path/to/Miniconda3-latest-Linux-x86_64.sh
```

Note: most often this file is downloaded to the Downloads directory, and thus the command will look like this:

```
bash Downloads/Miniconda3-latest-Linux-x86_64.sh
```

The instructions for the installation will then appear:

(1) Press Enter.
(2) Once the licence agreement shows, you can press space scroll down, or press `q` to skip reading it.
(3) Type `yes` and press enter to accept the licence agreement.
(4) Press enter to accept the default installation location.
(5) Type `yes` and press enter to instruct the installer to run `conda init`, which makes `conda` available from the terminal/shell.

12.3.3 JupyterLab

With miniconda set up, we can now install JupyterLab and the Jupyter Git extension. Type the following into the Anaconda Prompt (Windows) or the terminal (MacOS and Ubuntu) and press enter:

```
conda install -c conda-forge -y jupyterlab
conda install -y nodejs=10.*
pip install --upgrade jupyterlab-git
jupyter lab build
```

To test that your JupyterLab installation is functional, you can type `jupyter lab` into the Anaconda Prompt (Windows) or terminal (MacOS and Ubuntu) and press enter. This should open a new tab in your default browser with the JupyterLab interface. To exit out of JupyterLab you can click `File ->`

[4] https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.pkg
[5] https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh

`Shutdown`, or go to the terminal from which you launched JupyterLab, hold `Ctrl`, and press `c` twice.

12.3.4 R and the IRkernel

To have R available to you in JupyterLab, you will need to install the R programming language and the IRkernel. To install these, type the following into the Anaconda Prompt (Windows) or terminal (MacOS and Ubuntu):

```
conda install -c conda-forge -y r-base
conda install -c conda-forge -y r-irkernel
```

To improve the experience of using R in JupyterLab, we will add an extension that allows us to setup keyboard shortcuts for inserting text. By default, this extension creates shortcuts for inserting two of the most common R operators: `<-` and `%>%`. Type the following in the Anaconda Prompt (Windows) or terminal (MacOS and Ubuntu) and press enter:

```
jupyter labextension install @techrah/text-shortcuts
jupyter lab build
```

12.3.5 R packages

To install the packages used in this book, type the following in the Anaconda Prompt (Windows) or terminal (MacOS and Ubuntu) and press enter:

```
conda install -c conda-forge -y \
  r-cowplot \
  r-ggally \
  r-gridextra \
  r-infer \
  r-kknn \
  r-rodbc \
  r-rpostgres \
  r-rsqlite \
  r-testthat \
  r-tidymodels \
  r-tinytex \
  unixodbc
```

12.3.6 LaTeX

To be able to render `.ipynb` files to `.pdf` you need to install a LaTeX distribution. These can be quite large, so we will opt to use `tinytex`, a light-weight cross-platform, portable, and easy-to-maintain LaTeX distribution based on TeX Live. To install it open JupyterLab by typing `jupyter lab` in the Anaconda Prompt (Windows) or terminal (MacOS and Ubuntu) and press enter.

Then from JupyterLab open an R console and type the commands listed below and press Shift + enter to install `tinytex`:

```
tinytex::install_tinytex()
tinytex::tlmgr_install(c("eurosym",
                         "adjustbox",
                         "caption",
                         "collectbox",
                         "enumitem",
                         "environ",
                         "fp",
                         "jknapltx",
                         "ms",
                         "oberdiek",
                         "parskip",
                         "pgf",
                         "rsfs",
                         "tcolorbox",
                         "titling",
                         "trimspaces",
                         "ucs",
                         "ulem",
                         "upquote"))
```

12.4 Moving files to your computer

In the course that uses this textbook, students work on a web-based platform (a JupyterHub) to do their course work. This section is to help students save their work from this platform at the end of the course.

First in JupyterHub, open a terminal by clicking "terminal" in the Launcher tab. Next, type the following in the terminal to create a compressed `.zip` archive for the course work you are interested in downloading:

```
zip -r course_folder.zip your_course_folder
```

After the compressing process is complete, right-click on `course_folder.zip` in the JupyterHub file browser and click "Download". You should be able to use your computer's software to unzip the compressed folder by double-clicking on it.

Bibliography

Canada, S. (2016). Census of population. reproduced and distributed on an "as is" basis with the permission of statistics canada.

Fripp, G. (2020). Using cluster analysis for market segmentation.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An introduction to statistical learning*, volume 112. Springer.

Leek, J. T. and Peng, R. D. (2015). What is the question? *Science*, 347(6228):1314–1315.

Matthews, D., Wilson, G., and Easterbrook, S. (2008). Configuration management for large-scale scientific computing at the uk met office. *Computing in Science & Engineering*, 10(6):56–64.

Peng, R. D. and Matsui, E. (2015). The art of data science. *A Guide for Anyone Who Works with Data. Skybrude Consulting, LLC*.

Wickham, H. et al. (2014). Tidy data. *Journal of Statistical Software*, 59(10):1–23.

Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H., Huff, K. D., Mitchell, I. M., Plumbley, M. D., et al. (2014). Best practices for scientific computing. *PLoS Biol*, 12(1):e1001745.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2020). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.21.

Index