2 The Tidyverse and Reproducible R Workflows

2.1. Introduction

This chapter serves two goals. First, I will introduce you to the tidyverse, a modern way of doing R. Second, I will introduce you to reproducible research practices. As part of this, I will talk about ways efficient workflows for analysis projects. A disclaimer is necessary: This chapter is very technical. If you feel overwhelmed, don't worry, as the concepts discussed here will come up again and again throughout the book.

The tidyverse is a modern way of doing R that is structured around a set of packages created by Hadley Wickham and colleagues. The idea is to facilitate interactive data analysis via functions that are more intuitive and 'tidier' than some of the corresponding base R functions. Let's begin by installing and loading in the `tidyverse` package (Wickham, 2017).

```
install.packages('tidyverse')

library(tidyverse)
```

The tidyverse package is actually just a handy way of installing and loading lots of packages at the same time; namely, all those packages that are part of the tidyverse.[1] I will walk you through some, but not all, of the core members and some of their most useful functions. In this chapter, you will learn about `tibble` (Müller & Wickham, 2018), `readr` (Wickham, Hester, & François, 2017), `dplyr` (Wickham, François, Henry, & Müller, 2018), `magrittr` (Milton Bache & Wickham, 2014), and `ggplot2` (Wickham, 2016).

1 You could also load the individual packages from the tidyverse separately. These are the tidyverse packages discussed in this chapter:

```
library(tibble)
library(readr)
library(dplyr)
library(magrittr)
library(ggplot2)
```

2.2. tibble and readr

Tibbles from the tibble package (Müller & Wickham, 2018) are a modern take on data frames. They are like base R data frames, but better. Specifically, they confer the following four advantages:

- For text, tibbles default to character vectors rather than factor vectors, which is useful because character vectors are easier to manipulate.
- When typing the name of a tibble into the console, only the first ten rows are displayed, saving you a lot of `head()` function calls.
- Tibbles additionally display row and column numbers, saving you a lot of `nrow()` and `ncol()` function calls.
- Finally, displaying a tibble also reveals how each column in a tibble is coded (character vector, numeric vector, etc.).

All of these appear to be minor cosmetic adjustments. Together, however, these small changes end up saving you lots of time (and typing!). To see tibbles in action, let's load in a data frame and convert it to a tibble with `as_tibble()`:

```
# Load data:

nettle <- read.csv('nettle_1999_climate.csv')

# Convert data frame to tibble:

nettle <- as_tibble(nettle)
```

Type the name of the tibble:

```
nettle
```

```
# A tibble: 74 x 5
   Country    Population  Area   MGS Langs
   <fct>           <dbl> <dbl> <dbl> <int>
 1 Algeria          4.41 6.38   6.6     18
 2 Angola           4.01 6.1    6.22    42
 3 Australia        4.24 6.89   6      234
 4 Bangladesh       5.07 5.16   7.4     37
 5 Benin            3.69 5.05   7.14    52
 6 Bolivia          3.88 6.04   6.92    38
 7 Botswana         3.13 5.76   4.6     27
 8 Brazil           5.19 6.93   9.71   209
 9 Burkina Faso     3.97 5.44   5.17    75
10 CAR              3.5  5.79   8.08    94
# ... with 64 more rows
```

Notice that, besides printing out only ten rows, the number of rows (74) and columns (5) is stated, as well as information about vector classes. The `<dbl>` stands for 'double', which is computer-science-speak for a particular type of numeric vector—just treat doubles as numeric vectors. The `<int>` stands for ‘integer’; `<fct>` stands for ‘factor’. But wait, didn’t I just tell you that tibbles default to character vectors? Why is the `Country` column coded as a factor? The culprit here is the base R function `read.csv()`, which automatically interprets any text column as factor. So, before the data frame was converted into a tibble, the character-to-factor conversion has already happened.

To avoid this and save yourself the conversion step, use `read_csv()` from the `readr` package (Wickham et al., 2017) (notice the underscore in the function name).

```
nettle <- read_csv('nettle_1999_climate.csv')
```

```
Parsed with column specification:
cols(
  Country = col_character(),
  Population = col_double(),
  Area = col_double(),
  MGS = col_double(),
  Langs = col_integer()
)
```

The `read_csv()` function tells you how columns have been ‘parsed’, that is, how particular columns from the file were converted to particular vector types. In addition, `read_csv()` creates tibbles by default. Let’s verify this:

```
nettle
```

```
# A tibble: 74 x 5
   Country    Population Area    MGS Langs
   <chr>           <dbl> <dbl> <dbl> <int>
 1 Algeria          4.41 6.38   6.6     18
 2 Angola           4.01 6.1    6.22    42
 3 Australia        4.24 6.89   6      234
 4 Bangladesh       5.07 5.16   7.4     37
 5 Benin            3.69 5.05   7.14    52
 6 Bolivia          3.88 6.04   6.92    38
 7 Botswana         3.13 5.76   4.6     27
 8 Brazil           5.19 6.93   9.71   209
 9 Burkina Faso     3.97 5.44   5.17    75
10 CAR              3.5  5.79   8.08    94
# ... with 64 more rows
```

Notice how the `Country` column is now coded as a character vector, rather than as a factor vector.

In addition, `read_csv()` runs faster than `read.csv()`, and it provides a progress bar for large datasets. For files that are not comma-separated files (.csv), use `read_delim()`, for which the `delim` argument specifies the type of separator. The following command loads the tab-delimited ‘example_file.txt’:

```
x <- read_delim('example_file.txt', delim = '\t')
```

```
Parsed with column specification:
cols(
  amanda = col_integer(),
  jeannette = col_integer(),
  gerardo = col_integer()
)
```

```
x
```

```
# A tibble: 2 x 3
  amanda jeannette gerardo
   <int>     <int>   <int>
1      3         1       2
2      4         5       6
```

2.3. dplyr

The `dplyr` package (Wickham et al., 2018) is the tidyverse's workhorse for changing tibbles. The `filter()` function filters rows. For example, the following command reduces the `nettle` tibble to only those rows with countries that have more than 500 languages.

```
filter(nettle, Langs > 500)
```

```
# A tibble: 2 x 5
  Country           Population  Area   MGS Langs
  <chr>                  <dbl> <dbl> <dbl> <int>
1 Indonesia               5.27  6.28  10.7   701
2 Papua New Guinea        3.58  5.67  10.9   862
```

Alternatively, you may be interested in the data for a specific country, such as Nepal:

```
filter(nettle, Country == 'Nepal')
```

```
# A tibble: 1 x 5
  Country Population  Area   MGS Langs
  <chr>        <dbl> <dbl> <dbl> <int>
1 Nepal         4.29  5.15  6.39   102
```

So, the `filter()` function takes the input tibble as its first argument. The second argument is a logical statement that you use to put conditions on the tibble, thus restricting the data to a subset of rows.

The `select()` function is used to select columns. Just list all the columns you want to select, separated by commas. Notice that the original column order does not need to be obeyed, which means that `select()` can also be used to reorder tibbles.

```
select(nettle, Langs, Country)
```

```
# A tibble: 74 x 2
   Langs Country
   <int> <chr>
 1    18 Algeria
 2    42 Angola
 3   234 Australia
 4    37 Bangladesh
 5    52 Benin
 6    38 Bolivia
 7    27 Botswana
 8   209 Brazil
 9    75 Burkina Faso
10    94 CAR
# ... with 64 more rows
```

Using the minus sign in front of a column name excludes that column.

```
select(nettle, -Country)
```

```
# A tibble: 74 x 4
 Population  Area   MGS Langs
      <dbl> <dbl> <dbl> <int>
 1     4.41  6.38  6.6     18
 2     4.01  6.1   6.22    42
 3     4.24  6.89  6      234
 4     5.07  5.16  7.4     37
 5     3.69  5.05  7.14    52
 6     3.88  6.04  6.92    38
 7     3.13  5.76  4.6     27
 8     5.19  6.93  9.71   209
 9     3.97  5.44  5.17    75
10     3.5   5.79  8.08    94
# ... with 64 more rows
```

Use the colon operator to select consecutive columns, such as all the columns from `Area` to `Langs`.

```
select(nettle, Area:Langs)
```

```
# A tibble: 74 x 3
    Area   MGS Langs
   <dbl> <dbl> <int>
 1  6.38  6.6     18
 2  6.1   6.22    42
 3  6.89  6      234
 4  5.16  7.4      37
 5  5.05  7.14     52
 6  6.04  6.92     38
 7  5.76  4.6      27
 8  6.93  9.71    209
 9  5.44  5.17     75
10  5.79  8.08     94
# ... with 64 more rows
```

To summarize the two `dplyr` functions introduced so far: `filter()` is used to filter rows; `select()` is used to select columns.

The `rename()` function can be used to change the name of existing columns. Each argument is structured as follows: ‘New column name equals old column name.’ For example, the following code shortens the name of the `Population` column to `Pop`.

```
nettle <- rename(nettle, Pop = Population)

nettle
```

```
# A tibble: 74 x 5
   Country        Pop  Area  MGS   Langs
   <chr>        <dbl> <dbl> <dbl> <int>
 1 Algeria       4.41  6.38  6.6     18
 2 Angola        4.01  6.1   6.22    42
 3 Australia     4.24  6.89  6      234
 4 Bangladesh    5.07  5.16  7.4     37
 5 Benin         3.69  5.05  7.14    52
 6 Bolivia       3.88  6.04  6.92    38
 7 Botswana      3.13  5.76  4.6     27
 8 Brazil        5.19  6.93  9.71   209
 9 Burkina Faso 3.97  5.44  5.17    75
10 CAR           3.5   5.79  8.08    94
# ... with 64 more rows
```

The `mutate()` function can be used to change the content of a tibble. For example, the following command creates a new column `Lang100`, which is specified to be the `Langs` column divided by 100.

```
nettle <- mutate(nettle, Lang100 = Langs / 100)

nettle
```

```
# A tibble: 74 x 6
   Country    Population  Area   MGS Langs Lang100
   chr>            <dbl> <dbl> <dbl> <int>   <dbl>
 1 Algeria          4.41  6.38  6.6     18    0.18
 2 Angola           4.01  6.1   6.22    42    0.42
 3 Australia       4.24  6.89  6      234    2.34
 4 Bangladesh      5.07  5.16  7.4     37    0.37
 5 Benin           3.69  5.05  7.14    52    0.52
 6 Bolivia         3.88  6.04  6.92    38    0.38
 7 Botswana        3.13  5.76  4.6     27    0.27
 8 Brazil          5.19  6.93  9.71   209    2.09
 9 Burkina Faso    3.97  5.44  5.17    75    0.75
10 CAR             3.5   5.79  8.08    94    0.94
# ... with 64 more rows
```

Finally, `arrange()` can be used to order a tibble in ascending or descending order. Let's use this function to look at the countries with the largest and the smallest number of languages.

```
arrange(nettle, Langs)  # ascending
```

```
# A tibble: 74 x 6
   Country       Population  Area   MGS Langs Lang100
   <chr>              <dbl> <dbl> <dbl> <int>   <dbl>
 1 Cuba                4.03  5.04  7.46     1    0.01
 2 Madagascar          4.06  5.77  7.33     4    0.04
 3 Yemen               4.09  5.72  0        6    0.06
 4 Nicaragua           3.6   5.11  8.13     7    0.07
 5 Sri Lanka           4.24  4.82  9.59     7    0.07
 6 Mauritania          3.31  6.01  0.75     8    0.08
 7 Oman                3.19  5.33  0        8    0.08
 8 Saudi Arabia        4.17  6.33  0.4      8    0.08
 9 Honduras            3.72  5.05  8.54     9    0.09
10 UAE                 3.21  4.92  0.83     9    0.09
# ... with 64 more rows
```

```
arrange(nettle, desc(Langs))  # descending
```

```
# A tibble: 74 x 6
  Country          Population  Area   MGS Langs Lang100
  <chr>                 <dbl> <dbl> <dbl> <int>   <dbl>
 1 Papua New Guinea      3.58  5.67  10.9   862    8.62
 2 Indonesia             5.27  6.28  10.7   701    7.01
 3 Nigeria               5.05  5.97  7      427    4.27
 4 India                 5.93  6.52  5.32   405    4.05
 5 Cameroon              4.09  5.68  9.17   275    2.75
 6 Mexico                4.94  6.29  5.84   243    2.43
 7 Australia             4.24  6.89  6      234    2.34
 8 Zaire                 4.56  6.37  9.44   219    2.19
 9 Brazil                5.19  6.93  9.71   209    2.09
10 Philippines           4.8   5.48 10.3    168    1.68
# ... with 64 more rows
```

2.4. ggplot2

The ggplot2 package (Wickham, 2016) is many people's favorite package for plotting. The logic of ggplot2 takes some time to get used to, but once it clicks you'll be able to produce beautiful plots very quickly.

Let's use ggplot2 to graphically explore the relation between climate and linguistic diversity. Nettle (1999) discusses the intriguing idea that linguistic diversity is correlated with climate factors. The proposal is that countries with lower ecological risk have more different languages than countries with higher ecological risk. A subsistence farmer in the highlands of Papua New Guinea lives in a really fertile environment where crops can be grown almost the entire year, which means that there is little reason to travel. When speakers stay local and only speak with their immediate neighbors, their languages can accumulate differences over time which would otherwise be levelled through contact.

Nettle (1999) measured ecological risk by virtue of a country's 'mean growing season' (listed in the MGS column), which specifies how many months per year one can grow crops.

Let's plot the number of languages (Langs) against the mean growing season (MGS). Type in the following command and observe the result, which is shown in Figure 2.1 (left plot)—a detailed explanation will follow.

```
ggplot(nettle) +
  geom_point(mapping = aes(x = MGS, y = Langs))
```

Like most other functions from the tidyverse, the ggplot() function takes a tibble as its first argument. However, the rest of how this function works takes some time to wrap your head around. In particular, you have to think about your plot in a different way, as a layered object, where the data is the substrate, with different visual representations (shapes, colors, etc.) layered on top.

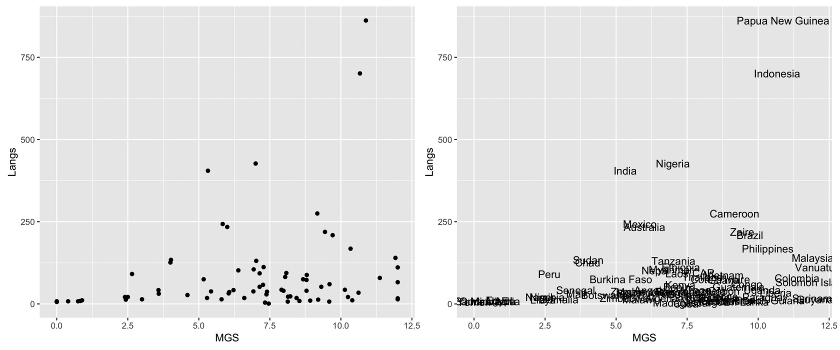

Figure 2.1. Left: scatterplot of the number of languages per mean growing season; right: the same scatterplot but with text; each data point is represented by the respective country name

In this case, the data are `MGS` and `Langs` taken from the `nettle` tibble. However, the same data could be displayed in any number of different ways. How exactly the data is visualized is specified by what is called a 'geom', a geometric object. Each of the plots that you commonly encounter in research papers (histograms, scatterplots, bar plots, etc.) has their own geom, that is, their own basic shape. In the case of scatter-plots, for example, the data is mapped to points in a 2D plane (as seen in Figure 2.1). In the case of bar plots (see below), the data is mapped to the height of the bars. As you proceed in this book, you will learn about other geoms, such as `geom_boxplot` (the data is mapped to boxes) or `geom_text` (the data is mapped to text).

So, the geom indicates the primary shape which is used to visually represent the data from a tibble. The code above adds a point geom to the plot via the `geom_point()` function. For plotting points in a two-dimensional plane, one needs both *x* and *y* values. The function `aes()` specifies the 'aesthetic mappings', which characterize which aspect of the data is mapped onto which aspect of the geom. In this case, `MGS` is mapped onto the *x*-values, and `Langs` is mapped onto the *y*-values. Think about it this way: the geometric objects are flying on top of the data substrate, but they need to know which specific aspects of the data to draw from, and this is specified by the aesthetic mappings.

Now, I recognize that the logic of the `ggplot2` package may appear confusing at this stage. I was definitely confused when I encountered `ggplot2` for the first time. As you proceed and type in more different plotting commands, you will slowly get the hang of things.

Let's learn a bit more about the aesthetic mappings. It turns out that you can either specify them inside a geom, or inside the main `ggplot()` function call. The latter allows multiple geoms to draw from the same mappings, which will become useful later. Notice, furthermore, that I omitted the argument name '`mapping`'.

```
ggplot(nettle, aes(x = MGS, y = Langs)) +
  geom_point()
```

Finally, let's create a plot where instead of points, the country names are displayed at their respective *x*, *y* coordinates. Let's see what happens when `geom_point()` is replaced with `geom_text()`.

```
ggplot(nettle, aes(x = MGS, y = Langs)) +
  geom_text()
```

```
Error: geom_text requires the following missing aesthet-
ics: label
```

The problem here is that `geom_text()` needs an additional aesthetic mapping. In particular, it needs to know which column is mapped to the actual text (`label`) shown in the plot, which is given in the `Country` column.

```
ggplot(nettle, aes(x = MGS, y = Langs, label = Country)) +
  geom_text()
```

The result is shown in the right plot of Figure 2.1. To save a plot, use `ggsave()` after a `ggplot2` command. For example, the following command saves the text plot into the file `nettle.png` with the width:height ratio of 8:6 (measurement units are given in inches by default).

```
ggsave('nettle.png', width = 8, height = 6)
```

To create the two-plot arrangement displayed in Figure 2.1, use the `gridExtra` package (Auguie, 2017).

```
# Create plots and save them in plot1 and plot2:

plot1 <- ggplot(nettle) +
  geom_point(mapping = aes(x = MGS, y = Langs))

plot2 <- ggplot(nettle,
         aes(x = MGS, y = Langs, label = Country)) +
  geom_text()

# Plot double plot:

library(gridExtra)
grid.arrange(plot1, plot2, ncol = 2)
```

The respective plots are saved into two objects, `plot1` and `plot2`, which are then used as arguments of the `grid.arrange()` function. The additional argument `ncol = 2` specifies a plot arrangement with two columns.

2.5. Piping with `magrittr`

A final component of the tidyverse relevant to us is the 'pipe', which is represented by the symbol sequence '`%>%`'. This functionality is unlocked by the tidyverse package `magrittr` (Milton Bache & Wickham, 2014).

Imagine a conveyor belt where the output of one function serves as the input to another function. The following code chunk exemplifies such a pipeline. The tibble `nettle` is first piped to the `filter()` function, which reduces the tibble to only those countries where one can grow crops for more than eight months of the year (`MGS > 8`). The filtered tibble is then piped to `ggplot()`, which results in a truncated version of Figure 2.1.

```
# Plotting pipeline with %>%:

nettle %>%
  filter(MGS > 8) %>%
  ggplot(aes(x = MGS, y = Langs, label = Country)) +
     geom_text()
```

Notice that the tibble containing the data only had to be mentioned once at the beginning of the pipeline, which saves a lot of typing. The true benefits of pipelines will be felt more strongly later on in this book.

2.6. A More Extensive Example: Iconicity and the Senses

This section guides you through the first steps of an analysis that was published in Winter, Perlman, Perry, and Lupyan (2017). This study investigated iconicity, the resemblance between a sign's form and its meaning. For example, the words *squealing*, *banging*, and *beeping* resemble the sounds they represent (also known as onomatopeia, a specific form of iconicity). It has been proposed that sound concepts are more expressible via iconic means than concepts related to the other senses, such as sight, touch, smell, or taste. This may be because auditory ideas are easier to express via imitation in an auditory medium, speech.

To test this idea, we used sensory modality ratings from Lynott and Connell (2009), paired with our own set of iconicity ratings (Perry, Perlman, Winter, Massaro, & Lupyan, 2017; Winter et al., 2017). Let's load in the respective datasets and have a look at them.

```
icon <- read_csv('perry_winter_2017_iconicity.csv')
mod <- read_csv('lynott_connell_2009_modality.csv')
```

Let's check the content of both files, starting with the `icon` tibble. Depending on how wide your console is, more or fewer columns will be shown. Also, some numbers may be displayed differently. For example, the raw frequency 1041179 of the article *a* could be displayed in the abbreviated form 1.04e6 (this notation will be explained in more detail in Chapter 11, fn. 1).

```
icon
```

```
# A tibble: 3,001 x 8
   Word     POS         SER CorteseImag  Conc  Syst   Freq
   <chr>    <chr>     <dbl>       <dbl> <dbl> <dbl>  <int>
 1 a        Grammati… NA             NA  1.46    NA 1.04e6
 2 abide    Verb      NA             NA  1.68    NA 1.38e2
 3 able     Adjective  1.73          NA  2.38    NA 8.15e3
 4 about    Grammati…  1.2           NA  1.77    NA 1.85e5
 5 above    Grammati…  2.91          NA  3.33    NA 2.49e3
 6 abrasive Adjective NA             NA  3.03    NA 2.30e1
 7 absorbe… Adjective NA             NA  3.1     NA 8.00e0
 8 academy  Noun      NA             NA  4.29    NA 6.33e2
 9 accident Noun      NA             NA  3.26    NA 4.15e3
10 accordi… Noun      NA             NA  4.86    NA 6.70e1
# ... with 2,991 more rows, and 1 more variable:
#   Iconicity <dbl>
```

The only three relevant columns for now are `Word`, `POS`, and `Iconicity`. Let's reduce the tibble to these columns using `select()`.

```
icon <- select(icon, Word, POS, Iconicity)

icon
```

```
# A tibble: 3,001 x 3
   Word       POS          Iconicity
   <chr>      <chr>            <dbl>
 1 a          Grammatical      0.462
 2 abide      Verb             0.25
 3 able       Adjective        0.467
 4 about      Grammatical     -0.1
 5 above      Grammatical      1.06
 6 abrasive   Adjective        1.31
 7 absorbent  Adjective        0.923
 8 academy    Noun             0.692
 9 accident   Noun             1.36
10 accordion  Noun            -0.455
# ... with 2,991 more rows
```

The dataset contains 3,001 words and their iconicity ratings. The `POS` column contains part-of-speech tags which were generated based on the SUBTLEX subtitle corpus of English (Brysbaert, New, & Keuleers, 2012). What about the content of the `Iconicity` column? In our rating task, we asked participants to rate words on a scale from −5 ('the word sounds like the opposite of what it means') to +5 ('the word sounds exactly like what it means'). The iconicity value of each word is an average of the ratings of multiple native speakers. Let's have a look at the range of this variable.

```
range(icon$Iconicity)
```

```
[1] -2.800000 4.466667
```

So, the lowest iconicity score is −2.8, the largest is +4.5 (rounded). This perhaps suggests that the iconicity ratings are skewed towards the positive end of the scale. To get a more complete picture, let's draw a histogram (depicted in Figure 2.2). Execute the following `ggplot2` code snippet—an explanation will follow.

```
ggplot(icon, aes(x = Iconicity)) +
  geom_histogram(fill = 'peachpuff3') +
  geom_vline(aes(xintercept = 0), linetype = 2) +
  theme_minimal()
```

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

The warning message on binwidth can safely be ignored in this case. The code pipes the `icon` tibble to `ggplot()`. To draw a histogram, you just need only one aesthetic, namely, an aesthetic that maps the data to the relevant *x*-values. The `fill`

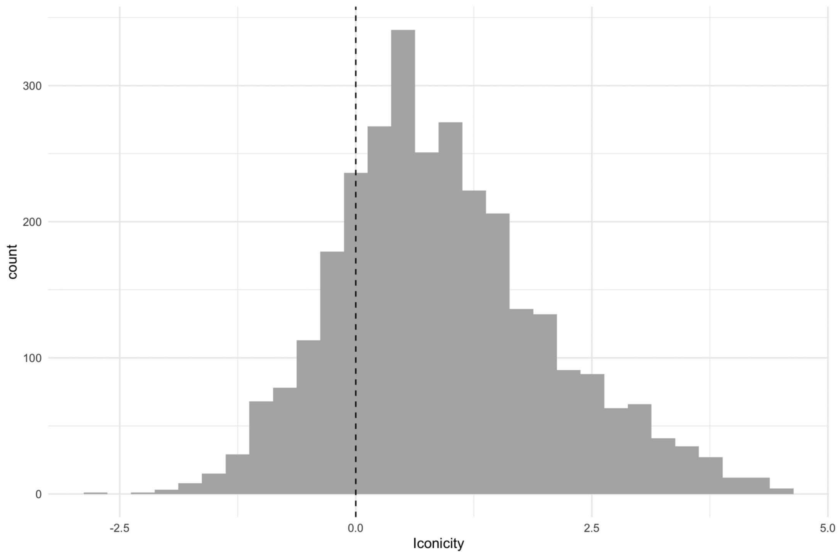

Figure 2.2. Histogram of iconicity values from Winter et al. (2017) and Perry et al. (2017)

of the histogram is specified to have the color `'peachpuff3'`. `geom_vline()` plots a vertical line at 0, which is specified by the `xintercept`. The optional argument `linetype = 2` makes the line dashed. Finally, adding `theme_minimal()` to the plot makes everything look nice. Themes are high-level plotting commands that change various aspects of the entire plot, such as the background color (notice the gray tiles have become white) and the grid lines. To see the effects of themes, explore what happens if you rerun the plot with `theme_linedraw()` or `theme_light()` instead. Themes save you a lot of typing, because you can avoid having to set all graphical parameters individually.

Next, let's check the `mod` tibble, which contains the modality norms from Lynott and Connell (2009).

```
mod
```

```
# A tibble: 423 x 9
   PropertyBritish Word      DominantModality Sight Touch
   <chr>           <chr>     <chr>            <dbl> <dbl>
 1 abrasive        abrasive  Haptic            2.89 3.68
 2 absorbent       absorbent Visual            4.14 3.14
 3 aching          aching    Haptic            2.05 3.67
 4 acidic          acidic    Gustatory         2.19 1.14
 5 acrid           acrid     Olfactory         1.12 0.625
 6 adhesive        adhesive  Haptic            3.67 4.33
 7 alcoholic       alcoholic Gustatory         2.85 0.35
 8 alive          alive    Visual           4.57 3.81
 9 amber          amber    Visual           4.48 0.524
10 angular        angular  Visual           4.29 4.10
# ... with 413 more rows, and 4 more variables:
#   Sound <dbl>, Taste <dbl>, Smell <dbl>,
#   ModalityExclusivity <dbl>
```

One feature of tibbles is that they only show as many columns as fit into your console. To display all columns, type the following (the extended output is not shown in the book):

```
mod %>% print(width = Inf)  # output not shown in book
```

The `width` argument allows you to control how many columns are displayed (thus, expanding or shrinking the 'width' of a tibble). By setting the width to the special value `Inf` (infinity), you display as many columns as there are in your tibble.

You may also want to display all rows:

```
mod %>% print(n = Inf)  # output not shown
```

The argument for rows is called `n` in line with the statistical convention to use the letter '*N*' to represent the number of data points.

For the present purposes, only the `Word` and `DominantModality` columns are relevant to us. The columns labeled `Sight`, `Touch`, `Sound`, `Taste`, and `Smell` contain the respective ratings for each sensory modality (on a scale from 0 to 5). Just as was the case with the iconicity rating study, these are averages over the ratings from several different native speakers. The content of the `DominantModality` column is determined by these ratings: for example, the first word, *abrasive*, is categorized as 'haptic' because its touch rating (3.68) is higher than the ratings for any of the other sensory modalities. Again, let's reduce the number of columns with `select()`.[2]

```
mod <- select(mod, Word, DominantModality:Smell)

mod
```

```
# A tibble: 423 x 7
   Word      Dominant     Sight Touch Sound  Taste  Smell
             Modality
   <chr>     <chr>        <dbl> <dbl>  <dbl>  <dbl>  <dbl>
 1 abrasive  Haptic        2.89  3.68  1.68  0.579  0.579
 2 absorbent Visual        4.14  3.14  0.714 0.476  0.476
 3 aching    Haptic        2.05  3.67  0.667 0.0476 0.0952
 4 acidic    Gustatory     2.19  1.14  0.476 4.19   2.90
 5 acrid     Olfactory     1.12  0.625 0.375 3      3.5
```

2 It's usually a good idea to spend considerable time getting the data in shape. The more time you spend preparing your data, the less trouble you will have later.

```
 6 adhesive  Haptic       3.67  4.33  1.19  0.905  1.76
 7 alcoholic Gustatory    2.85  0.35  0.75  4.35   4.3
 8 alive     Visual       4.57  3.81  4.10  1.57   2.38
 9 amber     Visual       4.48  0.524 0.143 0.571  0.857
10 angular   Visual       4.29  4.10  0.25  0.0476 0.0476
# ... with 413 more rows
```

To save yourself some typing further down the line, use `rename()` to shorten the name of the `DominantModality` column.

```
mod <- rename(mod, Modality = DominantModality)
```

Of course, you *could* have just changed the name in the spreadsheet before loading it into R. However, many times you work with large data files that are generated by some piece of software (such as E-Prime for psycholinguistic experiments, or Qualtrics for web surveys), in which case you want to leave the raw data untouched. It is usually a good idea to perform as much of the data manipulation as possible from within R. Even something as simple as renaming a column is a manipulation of the data, and it should be recorded in your scripts for the sake of reproducibility (more on this topic later).

When engaging with new datasets, it's usually a good idea to spend considerable time familiarizing yourself with their contents. For bigger datasets, it makes sense to check some random rows with the `dplyr` function `sample_n()`.

```
sample_n(mod, 4)  # shows 4 random rows, use repeatedly
```

```
# A tibble: 4 x 7
  Word        Modality Sight Touch Sound  Taste Smell
  <chr>       <chr>    <dbl> <dbl> <dbl>  <dbl> <dbl>
1 thumping    Auditory 2.62  2.52  3.90   0.143 0.190
2 transparent Visual   4.81  0.619 0.25   0.143 0.143
3 empty       Visual   4.75  3.6   1.65   0.25  0.15
4 spicy       Gustato… 1.67  0.429 0.333  5     4.24
```

To assess whether the sensory modalities differ in iconicity, it is necessary to merge the two tibbles. The `left_join()` function call below takes two tibbles as argument, 'joining' the second tibble ('to the right') into the first tibble ('to the left').

```
both <- left_join(icon, mod)
```

```
Joining, by = "Word"
```

The `left_join()` function is smart and sees that both tibbles contain a `Word` column, which is then used for matching the respective data. If the identifiers for matching have different names, you can either rename the columns so that they match, or you can use the `by` argument (check the `?left_join()` help file).

Next, let's filter the dataset so that it only includes adjectives, verbs, and nouns (there are very few adverbs and grammatical words, anyway). For this, the very useful

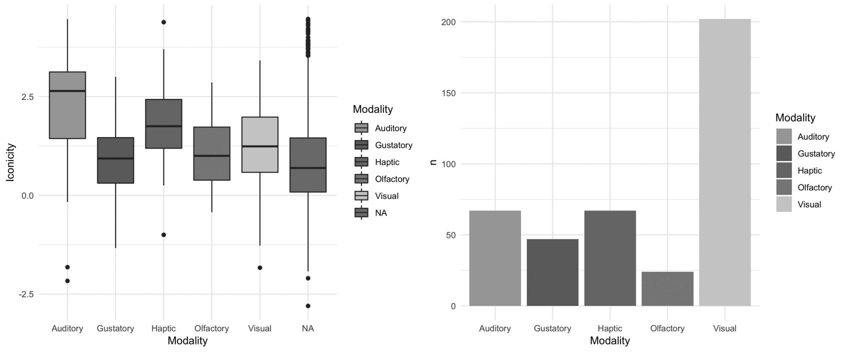

Figure 2.3. Left: Boxplot of iconicity ratings by sensory modality; right: Bar plot of word counts showing the over-representation of visual words in English

`%in%` function comes in handy. The function needs two vectors to compare, and it then checks whether the elements of the second vector are matched in the first vector.[3]

```
both <- filter(both,
         POS %in% c('Adjective', 'Verb', 'Noun'))
```

To put this command into plain English, one might paraphrase it as follows: 'Of the `both` tibble, filter only those rows for which the content of the `POS` column is in the set of adjectives, verbs, and nouns.'

Our main research question is whether iconicity ratings differ by modality. One option is to visualize this relationship with a boxplot, which will be explained in more detail in Chapter 3. A boxplot shows distributions as a function of categorical variable on the *x*-axis, as shown in Figure 2.3 (left plot). The following command maps `Modality` to the *x*-axis and `Iconicity` to the *y*-axis. Finally, an additional aesthetic maps the `Modality` categories onto the `fill` argument, which assigns each sensory modality a unique color.

```
ggplot(both,
    aes(x = Modality, y = Iconicity, fill = Modality)) +
  geom_boxplot() + theme_minimal()
```

The boxplot shows that the bulk of 'auditory' words (sound-related) have higher iconicity ratings than words for the other senses. 'Haptic' words (touch-related) also

3 To understand the set function `%in%`, check out what the following commands do:
```
c('A', 'B', 'C') %in% c('A', 'A', 'C', 'C')
```
… and then the reverse:
```
c('A', 'A', 'C', 'C') %in% c('A', 'B', 'C')
```
Basically, the `%in%` function is necessary when you want to use '==', but there are two things on the right-hand side of the equation. For the time being, it's OK to think of `%in%` as a generalized '=='.

have high iconicity ratings, and so on (for a discussion of these results, see Winter et al., 2017).

The right-most box displays the distribution of iconicity ratings for those words that couldn't be matched to a sensory modality because the word wasn't rated in Lynott and Connell's (2009) study. Let's exclude these missing values using `filter()`.[4] The `is.na()` function inside the following `filter()` command returns `TRUE` for any data point that is missing, and `FALSE` for any data point that is complete. The exclamation sign '`!`' inverts these logical values so that you only get the complete cases, that is, those cases that are *not* NA.[5] The code below also uses piping. The `both` tibble is piped to `filter()`, the output of which is then piped to `ggplot()`.

```
both %>% filter(!is.na(Modality)) %>%
  ggplot(aes(x = Modality, y = Iconicity,
             fill = Modality)) +
  geom_boxplot() + theme_minimal()
```

Let's explore another aspect of this data, which is discussed in more detail in Lievers and Winter (2017) and Winter, Perlman, and Majid (2018). In particular, it has been proposed that some sensory modalities are easier to talk about than others (Levinson & Majid, 2014), and that the English language makes more visual words available to its speakers than words for the other sensory modalities (see also Majid & Burenhult, 2014). To investigate this feature of the English language, have a look at the counts of words per sensory modality with `count()`.

```
both %>% count(Modality)
```

```
# A tibble: 6 x 2
  Modality      n
  <chr>     <int>
1 Auditory     67
2 Gustatory    47
3 Haptic       67
4 Olfactory    24
5 Visual      202
6 <NA>       2389
```

Ignoring the NAs for the time being, this tibble shows that there are overall more visual words. This was already noted by Lynott and Connell (2009).

Let's make a bar plot of these counts. As before, the `filter()` function is used to exclude NAs. The geom for bar plots is `geom_bar()`. You need to specify the additional argument `stat = 'identity'`. This is because the `geom_bar()` function

4 Generally, you should be concerned about missing values. In particular, you always need to ask *why* certain values are missing.

5 Alternatively, you could use the function `complete.cases()`, which returns `TRUE` for complete cases and `FALSE` for missing values. In that case, you wouldn't have to use the negation operation '`!`'. That said, I prefer `!is.na()` because it takes less time to type than `complete.cases()` ...

likes to perform its own statistical computations by default (such as counting). With the argument `stat = 'identity'`, you instruct the function to treat the data as is. Figure 2.4 (right plot) shows the resulting bar plot.

```
both %>% count(Modality) %>%
  filter(!is.na(Modality)) %>%
  ggplot(aes(x = Modality, y = n, fill = Modality)) +
  geom_bar(stat = 'identity') + theme_minimal()
```

To exemplify why you had to specify `stat = 'identity'` in the last code chunk, have a look at the following code, which reproduces Figure 2.4 in a different way.

```
both %>% filter(!is.na(Modality)) %>%
  ggplot(aes(Modality, fill = Modality)) +
  geom_bar(stat = 'count') + theme_minimal()
```

In this case, the `geom_bar()` function does the counting. This pipeline is a bit more concise since you don't have to create an intervening table of counts.

2.7. R Markdown

Chapter 1 introduced you to .R script files. Now you will learn about R markdown files, which have the extension .Rmd. R markdown files have more expressive power than regular R scripts, and they facilitate the reproducibility of your code. Basically, an R markdown file allows you to write plain text alongside R code. Moreover, you can 'knit' your analysis easily into a html file, which is done via the `knitr` package (Xie, 2015, 2018). The resulting html file contains your text, R code, and output. It is extremely useful to have all of this in one place. For example, when you write up your results, it'll be much easier to read off the numbers from a knitted markdown report, rather than having to constantly re-create the output in the console, which is prone to error. Moreover, the report can easily be shared with others. This is useful when collaborating—if your collaborators also know R, they can not only see the results, but they'll also know how you achieved these results. R markdown is a standard format for sharing your analysis on publicly accessible repositories, such as GitHub and the Open Science Framework.

To create an R markdown file in RStudio, click 'File', 'New File', and then 'R Markdown …' For now, simply leave the template as it is. See what happens if you press the 'knit' button (you will be asked to save the file first). This will create an html report of your analysis that shows the code, as well as the markdown content.

Let me guide you through some R markdown functionality. First, you can write plain text the same way you would do in a text editor. Use this text to describe your analysis. In addition, there are code chunks, which always begin with three `'''` (backward ticks, or the grave accent symbol). The R code goes inside the chunk. Any R code outside of code chunks won't be executed when knitting the file.

```
'''{r}
# R code goes in here
'''
```

You can specify additional options for each code chunk. For example, the code chunk below will print results, but not messages (such as warning messages, or package loading messages), and it will also 'cache' the results of running the code chunk. Caching means that the result of all computations conducted in this code chunk will be saved to memory files outside of the markdown script. This way, the computations don't have to be re-executed the next time you knit the markdown file. The argument `cache = TRUE` is useful when a particular code chunk takes a lot of time to run.

```
'''{r message = FALSE, cache = TRUE}
# R code goes in here
'''
```

The following code chunk, named `myplot`, does not print the R code (`echo = FALSE`). The additional code chunk options specify the width and height of the plot that will be displayed in the knitted html file.

```
'''{r echo = FALSE, fig.width = 8, fig.height = 6}
# plotting commands go in here
'''
```

2.8. Folder Structure for Analysis Projects

One big advantage of R markdowns is that when opening up an .Rmd file in RStudio, the working directory is always set to the location of the file. This facilitates reproducibility because somebody does not have to change the working directory to fit the folder structure of their own machine. It's generally frowned upon to use `setwd()` in a script for this reason, since any use of `setwd()` is specific to one's machine. Instead, wouldn't it be much easier if the user who wants to reproduce your analysis just needs to download the entire project folder, with no fiddling of `setwd()` required? R markdown scripts make this possible.

In general, it is good to structure your project in a consistent manner around folders. The R markdown scripts then operate *relatively* from within that folder structure, not *absolutely* specific to the folder structure on your machine. At a bare minimum, there should be a data folder, a scripts folder, and a figures folder (see Figure 2.4).

Let's say you are working within the 'analysis.Rmd' file in the 'scripts' folder, aiming to load the 'mydata.csv' file from the 'data' folder. The following command would achieve this:

```
mydf <- read_csv('../data/mydata.csv')
```

The '`..`' instructs the markdown file to jump one folder up in the hierarchy of folders, which is the overarching 'project' folder in this case. From there, the '`/data/`'

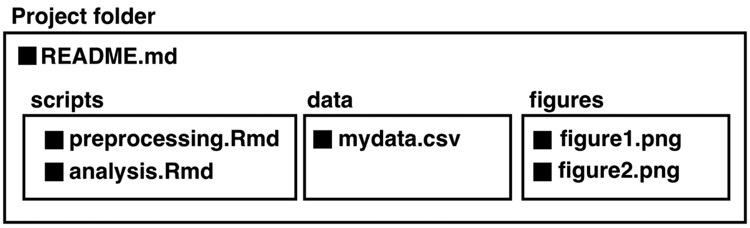

Figure 2.4. Folder structure for a data analysis project; black squares represent data files

bit instructs the markdown script to look for the 'data' folder. The logic is similar for saving a ggplot from within a script in the 'scripts' to the 'figures' folder:

```
ggsave('../figures/figure1.png')
```

2.9. Readme Files and More Markdown

The main project folder of any data analysis project should always contain a readme file describing the overall structure of the analysis. For this, markdown files (.md) are useful, as these files are often interpreted by data sharing repositories such as GitHub or the Open Science Framework (OSF). You can create markdown files with any text editor, such as the built-in 'Notes' on Macs or 'Notepad' on PCs.

Let's talk about some markdown features, which can also be used in R markdown files (.Rmd): Single hashtags '#' or double hashtags '##' display the corresponding text as a major or minor title on GitHub.

```
# My major title
## My minor title
```

Text enclosed by two stars '**' will be displayed in bold; text enclosed by one star '*' will be displayed in italics.

```
**bold text**
*italic text*
```

Lines beginning with hyphens will be displayed as bullet points.

```
- bullet point 1
- bullet point 2
```

Here's an example of what the beginning of a README.md file for a project could look like:

```
# Title of my statistical analysis

-  **Study design & data collection:** My friend
-  **Statistical Analysis:** Bodo Winter
- **Date:** 24/12/18

## Libraries required for this analysis:

- tidyverse
- lme4

## Script files contained in this analysis:

- preprocessing.Rmd : Getting the data into shape
- analysis.Rmd : Linear mixed effects model analysis
```

Ideally, there's also a 'codebook' for every dataset which details the content of every column. In particular, it is useful if there is a full description of which values are contained within each column, such as, '`dialect: categorical variable, assumes one of the three values Brummie, Geordie, Scouse`'.

It is important to use the features introduced in this section in R markdown files to give your analysis structure. For example, you can use '`#`' and '`##`' to highlight major and minor sections of your analysis.[6] As will be explained in the next section, code cleanliness is more than just cosmetics—it is intimately tied to the reproducibility of an analysis.

2.10. Open and Reproducible Research

Scientific progress is cumulative—it builds on past achievements. However, cumulative progress is only possible if results are both 'replicable' and 'reproducible'. What's the difference between replicability and reproducibility? In short: replicability characterizes the ability to replicate a study, that is, being able to conduct the same study again (with new data). Reproducibility characterizes the more basic requirement of being able to reproduce a researcher's analysis of a given dataset on one's own machine.

A study is replicable if another researcher can read the methods section of a paper and has enough information to replicate the study with new participants. Recently, researchers have found that many famous results failed to replicate, which has led to the 'replication crisis' (Open Science Collaboration, 2015; Nieuwland et al., 2018). Linguistics doesn't have a replication crisis yet, but it's looming around the corner. There already are important linguistic results that failed to replicate, such as the idea that there is a bilingual advantage in certain cognitive processing tasks (Paap & Greenberg, 2013; de Bruin, Bak, & Della Sella, 2015). Other linguistic findings that failed to replicate involve sentence processing (Nieuwland et al., 2018; Stack, James, & Watson, 2018) and embodied language understanding (Papesh, 2015).

There are many reasons why a study may not replicate. One reason, however, has to do with a lack of reproducibility of existing research. For any study, it should be possible for other researchers to obtain the same results if they were given the same data. The problem is that statistical analysis—as will be pointed out repeatedly throughout this book—is a strikingly subjective process (this may surprise you). For example,

6 Note that inside a code chunk, the hashtag '`#`' is interpreted as a comment. Outside of a code chunk, it is interpreted as a title.

even expert analysts will come to different conclusions when given the same dataset (Silberzahn et al., 2018). There are myriads of decisions to make in an analysis, what some people call "researcher degrees of freedom" (Simmons, Nelson, & Simonsohn, 2011), and what others call "the garden of forking paths" (Gelman & Loken, 2014). Without the ability to trace what a researcher did in their analysis, these choices remain hidden. Reproducibility requires us to lay all of this open.

A fully reproducible study gives you the data and the code. When you execute the code on your machine, you will get exactly the same results. However, reproducibility is a gradable notion, with some studies being more reproducible than others. There are several things that increase a study's reproducibility:

- The minimal requirement for reproducible research is that the data and scripts are shared ('open data, open code'). The scientific community, including the linguistic community, needs to rethink what it means to 'publish' a result. The following mindset may be helpful to induce change and to assure replicability and reproducibility in the long run: without publishing the data and code, a publication is *incomplete*. The scientific process hasn't finished unless the data and code are provided as well.
- The choice of software influences reproducibility. R is more reproducible than SPSS, STATA, SAS, Matlab, or other proprietary software, because it's free, open-source, and platform-independent. Whenever possible, use software that is accessible to everybody.
- More thoroughly documented code is more reproducible, as other researchers will have an easier time tracing your steps.
- R markdown scripts facilitate reproducibility because they make extensive documentation easier via the ability to incorporate plain text, and because they allow avoiding `setwd()` (see discussion above). In addition, the ability to knit a final report of your analysis allows researchers (including yourself) to more quickly see the results *together* with the code that has produced the results.
- Use publicly accessible platforms such as OSF and GitHub[7] rather than a journal's online supplementary materials. It's not in the publisher's interest to store your data for eternity, and it's well known that publishers' websites are subject to change, which leads to link rot.

When people first hear about the idea of open reproducible research, they may be worried. There are a few common objections that are worth addressing at this stage (see also Houtkoop, Chambers, Macleod, Bishop, Nichols, & Wagenmakers, 2018).

- *I don't want to my results or ideas stolen (I don't want to get 'scooped').* This is a very common worry. Paradoxically, perhaps, making everything open from the get-go actually makes it *less* likely to get scooped. Publishing on a publicly

7 GitHub and OSF play well together. For my analysis projects, I use an OSF repository as the overarching repo, linking to a corresponding GitHub repo which stores the data and code. OSF will become more important in years to come, as more and more researchers are 'pre-registering' their studies (see Nosek & Lakens, 2014).

accessible repository allows you to claim precedence, which makes scooping more difficult.[8]

- *I fear retaliation or loss of reputation if somebody finds an error, or provides an alternative analysis with different conclusions*. This is a very common fear, and I think it's safe to say that most scientists have worried about this at some point in their career. However, this objection gets it the wrong way around. You are more likely to be criticized if somebody finds out that something is wrong and you have *not* shared your data and code. When making your materials openly accessible, people are less likely to ascribe deliberate wrongdoing to you. Keeping your data behind locked doors is only going to make you look more suspicious. You and your research will appear more trustworthy if you share everything right away.
- *I cannot actually share the data for ethical reasons*. Certain datasets are impossible to share for very good reasons. However, it's usually the case that the final steps of a data analysis can be shared, such as the summary data that was used to construct a statistical model. In general, it is your responsibility that the data is made anonymous so that it can be appropriately shared without ethical concerns.
- *I fear that companies may use my data for wrongdoing*. It is good to be concerned about big-data-harvesting that is done with bad intentions, especially when it involves the recognition of identities. That said, almost all of the data dealt with in linguistics is anonymous or, if not, it can easily be anonymized. Moreover, the data of most studies is often of little use to companies. In particular, most experimental studies are quite constrained to very particular, theoretically involved hypotheses that cannot easily be commercialized. Things are different if you are dealing with non-anonymized data, or such data sources as social media. However, in general, compared to the overwhelming benefits of sharing data and code in the face of the replication crisis, corporate misuse is a very minor concern. If you are truly worried about corporate data pirates, consider licensing your data accordingly.
- *I feel embarrassed about my code*. This is a very understandable concern—don't worry, as you're not alone! The first code that I put up online a few years ago looks absolutely horrible to me right now. People understand that everybody's coding practice is different, and that different researchers have different levels of statistical sophistication. Nobody will look at your code and laugh at it. Instead, they will appreciate the willingness to share your materials, even if it doesn't look snazzy.
- *Sharing data and code is not a common practice in my field*. My response to this is: not yet! Things are clearly heading this way, and more and more journals are implementing policies for data sharing—many already have! Journals such as *PLOS One* and the Royal Society journals are trendsetters in this respect. In addition: if it's not yet a common practice in your field, why not make it a common practice? You can lead the change.

Putting scientific progress aside, if you wanted some purely cynical reasons for 'going open' and reproducible, you might want to know that studies with open data and open code have higher citation rates (Piwowar & Vision, 2013).

8 Moreover, let's face it: many linguists are working on such highly specific topics that getting scooped is of little concern. However, if you are working on something that many other people are currently working on as well, use a publicly accessible repository to be the first to get it out there!

Finally, since all kinds of disciplines (including the language sciences) are currently moving towards open and reproducible methods, it makes sense for you to adopt open and reproducible methods early on, so that you are not overwhelmed when this becomes a general requirement. Rather than being late adopters, let's try to be ahead of the curve and at the same time further progress in science!

2.11. Exercises

2.11.1. Review

Review the tidyverse style guide …

http://style.tidyverse.org/

. . . and the RStudio keyboard shortcut list:

https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts

Think about which shortcuts you want to adopt in your own practice (experiment!). After having incorporated a few shortcuts into your workflow, return to the shortcut list and think about which other shortcuts to adopt.

2.11.2. Create and Knit a Markdown File

Create a markdown file with the title 'Analysis of linguistic diversity'. In the first R code chunk, load in the tidyverse package and the Nettle (1999) dataset. Using the `sum()` function, compute the sum of languages across the entire dataset. Describe each step with a few short sentences of plain text outside of the code chunks. Then, knit the file to an html file and check the output.

2.11.3. Subsetting Data Frames with Tidyverse Function

This exercise uses the `nettle` data frame to explore different ways of indexing using `filter()` and `select()`. First, load in the `nettle` data:

```
nettle <- read.csv('nettle_1999_climate.csv')

head(nettle)  # display first 6 rows
```

```
    Country Population Area  MGS Langs
1  Algeria        4.41 6.38 6.60    18
2   Angola        4.01 6.10 6.22    42
3 Australia       4.24 6.89 6.00   234
4 Bangladesh      5.07 5.16 7.40    37
5   Benin         3.69 5.05 7.14    52
6  Bolivia        3.88 6.04 6.92    38
```

Next, attempt to understand what the following commands do. *Then* execute them in R and see whether the output matches your expectations.

```
filter(nettle, Country == 'Benin')

filter(nettle, Country %in% c('Benin', 'Zaire'))

select(nettle, Langs)

filter(nettle, Country == 'Benin') %>% select(Langs)

filter(nettle, Country == 'Benin') %>%
select(Population:MGS)

filter(nettle, Langs > 200)

filter(nettle, Langs > 200, Population < median(Population))
```

2.11.4. Extended Exercise: Creating a Pipeline

Execute the following code in R (you may omit the comments for now) and then read the explanation below.

```
# Reduce the nettle tibble to small countries:

smallcountries <- filter(nettle, Population < 4)

# Create categorical MGS variable:

nettle_MGS <- mutate(smallcountries,
     MGS_cat = ifelse(MGS < 6, 'dry', 'fertile'))

# Group tibble for later summarizing:

nettle_MGS_grouped <- group_by(nettle_MGS, MGS_cat)

# Compute language counts for categorical MGS variable:

summarize(nettle_MGS_grouped, LangSum = sum(Langs))
```

```
# A tibble: 2 x 2
  MGS_cat LangSum
  <chr>     <int>
1 dry         447
2 fertile    1717
```

This code reduces the `nettle` tibble to small countries (`Population < 4`). The resulting tibble, `smallcountries`, is changed using the `ifelse()` function.

In this case, the function splits the dataset into countries with high and low ecological risk, using six months as a threshold. The `ifelse()` function spits out '`dry`' when `MGS < 6` is `TRUE` and '`fertile`' when `MGS < 6` is `FALSE`. Then, the resulting tibble is grouped by this categorical ecological risk measure. As a result of the grouping, the subsequently executed `summarize()` function knows that summary statistics should be computed based on this grouping variable.

This code is quite cumbersome! In particular, there are many intervening tibbles (`smallcountries`, `nettle_MGS`, and `nettle_MGS_grouped`) that might not be used anywhere else in the analysis. For example, the grouping is only necessary so that the `summarize()` function knows what groups to perform summary statistics for. These tibbles are fairly dispensable. Can you condense all of these steps into a single pipeline where the `nettle` tibble is first piped to `filter()`, then to `mutate()`, then to `group_by()`, and finally to `summarize()`?