Working with datasets

Francis L. Yuen

2019-12-09

Learning goals:

1. Analyzing and plotting data
2. Reading data in a reproducible fashion
3. Tidy data and an introduction to tidyverse

Analyzing and plotting data

Working with datasets is the main goal of using R (at least for us). To practice working with a dataset, we're going to use a sample dataset built into R. The dataset is called 'mtcars'.

- Open a new R script
- Create a new object in your environment called 'data' by assigning 'mtcars' to it (remember the <- operator?)
- Inspect 'data'
- load the "tidyverse" package for use later

```
data <- mtcars
library(tidyverse)
```

An **observation** (or obs.) is how many *rows* are in your dataset, and a **variable** is how many *columns*. R is especially good at working with **tidydata**, but we will get into what that means in a bit. Let's first play around with this dataset by using some useful functions. Before we do that, we need to figure out what the variable names mean since we're not the ones that named it. Fortunately, 'mtcars' is well documented, so we can type in `?mtcars` to inspect it. The first thing you should do when you load in a new dataset is to examine its `structure` using the `str()` function:

```
str(data)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
```

```
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

This will tell you how each variable is defined. If you are familiar with your dataset, you should use this chance to double check that the category is correct for each variable (e.g. if a variable is a group, it should be a `factor` and not `numeric`; more on this later).

Next, let's get a sense of what we're working with:

```
summary(data)
```

```
##       mpg             cyl
##  Min.   :10.40   Min.   :4.000
##  1st Qu.:15.43   1st Qu.:4.000
##  Median :19.20   Median :6.000
##  Mean   :20.09   Mean   :6.188
##  3rd Qu.:22.80   3rd Qu.:8.000
##  Max.   :33.90   Max.   :8.000
##       disp             hp
##  Min.   : 71.1   Min.   : 52.0
##  1st Qu.:120.8   1st Qu.: 96.5
##  Median :196.3   Median :123.0
##  Mean   :230.7   Mean   :146.7
##  3rd Qu.:326.0   3rd Qu.:180.0
##  Max.   :472.0   Max.   :335.0
##       drat             wt
##  Min.   :2.760   Min.   :1.513
##  1st Qu.:3.080   1st Qu.:2.581
##  Median :3.695   Median :3.325
##  Mean   :3.597   Mean   :3.217
##  3rd Qu.:3.920   3rd Qu.:3.610
##  Max.   :4.930   Max.   :5.424
##       qsec             vs
##  Min.   :14.50   Min.   :0.0000
##  1st Qu.:16.89   1st Qu.:0.0000
##  Median :17.71   Median :0.0000
##  Mean   :17.85   Mean   :0.4375
##  3rd Qu.:18.90   3rd Qu.:1.0000
##  Max.   :22.90   Max.   :1.0000
##        am              gear
##  Min.   :0.0000   Min.   :3.000
##  1st Qu.:0.0000   1st Qu.:3.000
##  Median :0.0000   Median :4.000
```

```
##  Mean   :0.4062   Mean   :3.688
##  3rd Qu.:1.0000   3rd Qu.:4.000
##  Max.   :1.0000   Max.   :5.000
##       carb
##  Min.   :1.000
##  1st Qu.:2.000
##  Median :2.000
##  Mean   :2.812
##  3rd Qu.:4.000
##  Max.   :8.000
```

`summary()` gives you some very basic descriptives. These can sometimes be useful to check for outliers, but doesn't tell us anything particularly interesting. Let's say we want to see if the number of forward gears (gear) is related to horsepower (hp). Since the number of forward gears is our independent variable, We will ask R to organize the dataset by gear:

```
data <- group_by(.data = data, gear)
```

`group_by` takes quite a few arguments (check `?group_by` if you're interested!), the most important ones being the .data argument where you specify which dataset you want it to work with, and *one or more* variable arguments it should group by.

Look at 'data' again. Doesn't seem like anything happened, right? All `group_by()` does is add an invisible 'marker' to all the observations depending on their value in the 'gear' column. Now, let's check the dataset again, this time using `summarize()`. We will store our results as a new object 'mean_hp':

```
mean_hp <- summarize(.data = data, mean = mean(hp))
mean_hp
```

```
## # A tibble: 3 x 2
##    gear  mean
##   <dbl> <dbl>
## 1     3 176.
## 2     4  89.5
## 3     5 196.
```

`summarize()` is a cool function that takes one dataset, and performs any number of additional functions to that dataset. What's really useful about `summarize()` is when you use it in tandem with `group_by` because any function you apply with `summarize()` will operate using the group markers. Let's see what else we can do:

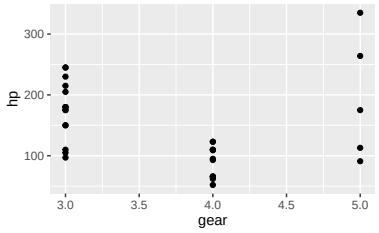
```
mean_hp <- summarize(data, mean = mean(hp), sd = sd(hp),
    n = length(hp), SEM = sd/sqrt(n))
mean_hp
```

```
## # A tibble: 3 x 5
##    gear  mean    sd     n   SEM
##   <dbl> <dbl> <dbl> <int> <dbl>
## 1     3 176.   47.7    15 12.3
## 2     4  89.5  25.9    12  7.47
## 3     5 196.  103.      5 46.0
```

Notice how we can nest several functions that are dependent on each other (i.e. in order to calculate the standard error of the mean, we needed to first calculate the standard deviation and count the number of observations in each group). These need to be in the correct order as summarize always evaluates from top to bottom.

Now that we have all this info, it's time to make some pretty graphs. The most common graphing tool in R is `ggplot()`. We will first start with a basic point plot using the original data set:

```
ggplot(data, aes(x = gear, y = hp)) + geom_point()
```



Interesting, looks like R doesn't know that the gear variable is not a continuous variable, so it plotted the x-axis as if gear can be a non-integer. To fix this, we will modify the definition of the gear column and turn it into a `factor` column instead of `numeric`. Then, we will check that we succeeded by using `str()`:

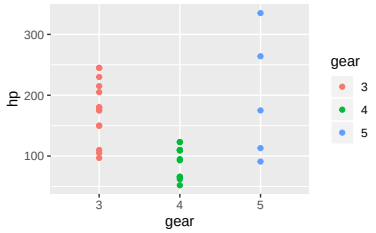
```
data$gear <- as.factor(data$gear)
str(data)
```

```
## Classes 'grouped_df', 'tbl_df', 'tbl' and 'data.frame':  32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: Factor w/ 3 levels "3","4","5": 2 2 2 1 1 1 1 2 2 2 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
##  - attr(*, "groups")=Classes 'tbl_df', 'tbl' and 'data.frame':   3 obs. of  2 variables:
##   ..$ gear : num  3 4 5
##   ..$ .rows:List of 3
```

```
##   .. ..$ : int  4 5 6 7 12 13 14 15 16 17 ...
##   .. ..$ : int  1 2 3 8 9 10 11 18 19 20 ...
##   .. ..$ : int  27 28 29 30 31
##   ..- attr(*, ".drop")= logi TRUE
```

Now that the gear variable is a factor, let's plot it again, this time with some color:

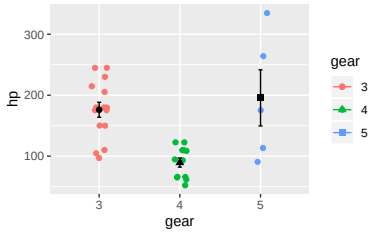
```
ggplot(data, aes(x = gear, y = hp, color = gear)) +
    geom_point()
```



The best thing about `ggplot()` is that you can customize it by adding more 'layers' on top of each other. Let's add the means for each gear group as well as their corresponding standard errors we calculated earlier as error bars (we need to specify gear as a factor for `mean_hp` first):

```
mean_hp$gear <- as.factor(mean_hp$gear)
```

```
ggplot(data = data, aes(x = gear, y = hp, color = gear)) +
    geom_point(position = position_jitter(width = 0.1)) +
    geom_errorbar(data = mean_hp, aes(x = gear,
        y = mean, color = NULL, ymin = mean -
            SEM, ymax = mean + SEM), width = 0.05) +
    geom_point(data = mean_hp, aes(x = gear, y = mean,
        color = NULL, shape = gear), size = 2)
```



Much better!

Exercise 1

Look at 'data' again. Come up with a new hypothesis and make a new plot illustrating the relationship between the variables you chose.

Note: The easiest ones would be to have a categorical independent variable and a continuous dependent variable. If you want a challenge and to do something fancier, feel free to choose continuous variables for both!

Reproducible workflow

Creating an Rproject

Now that we know what to do with data, we need to know how to load in our own datasets that are not a part of R. This means we need to start thinking about how to make our entire workflow reproducible, i.e. how to make it so that someone else can download our entire project and be able to run everything smoothly. To do that, we will create an **RProject**. Go to File -> New Project, and choose New Directory then New Project. Click 'Browse' to go to where you want to save your project, then for directory name, put "Workshop_2"

without the quotation marks. This will create a folder where you can store all files related to your study.

It's generally a good idea to have subfolders for different types of content. For example, you might want a folder just for data, and a folder just for Rscripts.

There is no rule for how to set up your project, you can tailor it to your personal preference as long as it's organized.

To start, navigate to "Workshop_2" and create two new folders: "data" and "scripts". Save the Rscript you have been working on in the "scripts" folder and give it a name (suggestion: mtcars_practice).

Next, we will start a cleaning script. Raw datasets are often really messy and difficult to work with; luckily, R has a lot of neat tools that can help us manipulate the data. Another option would be to manually edit the data in Excel, which can sometimes be easier if the dataset is not huge. Most of the time it will be easier and faster with the help of the 'janitor' package.

Hint: install the package using the function `install.packages("janitor")`

Create a new R script and save it in the "scripts" folder. Call this "cleaning_script". Start the script by writing a few lines that loads in three packages: "tidyverse", "here", and "janitor":

Reading raw data into R

Next, download **sample_data_1** and save it in, or move it to, the "data" folder.

This is a pseudo-randomly generated dataset and is not from a real study, but for the purposes of practicing cleaning it will be fine.

Now, we are going to use the `read_csv()` function from the 'tidyverse' package to load in the data:

```
data <- read_csv("sample_data_1.csv")
```

Oops! If you try to run this on its own, R knows to look for a file called sample_data_1.csv, but it doesn't know where to start! Traditionally, the way to help R find the data is to set your working directory to the correct folder. However, people have different places where they store things on their computer, so this is *not* a reproducible method. This is where the 'here' package kicks in:

Note: Even if people download your entire project, they would still have to set the working directory manually. The goal is to prevent any additional step that a researcher would have to take to reproduce your results.

```
data <- read_csv(here("data", "sample_data_1.csv"))
```

`here()` tags an additional argument that tells R which folder to look in. It always starts in the directory where the code is located (in our case, the "scripts" folder), which means that when someone else downloads your Rproject, it doesn't matter where they store the actual project because `here()` will never leave the parent directory.

Exercise 2

Suppose you want to create a subfolder in your "data" folder so you can differentiate between raw data and cleaned data. Make a folder in "data" called "raw_data", and move sample_data_1.csv in there. Now modify your read_csv line of code to help R find your file.

Cleaning the data

Inspect your data. It doesn't look too bad, but the column names are inconsistent: some have capital letters, some are separated by hyphens and some by spaces. It will save us a lot of time down the road if we clean these up right now:

```
data <- clean_names(data)
```

Inspect the data again. What changed? Think of at least two reasons why this is easier to work with than the previous one.

Tidy data and tidyverse

What is tidy data?

In the first section, it was really easy for us to use `group_by()`, `summarize()`, and `ggplot()`. Unfortunately, not all raw datasets will be 'ready' for use like 'mtcars'. In fact, most datasets will NOT be nicely organized that way. For R to have maximal efficiency, datasets need to be manipulated into a special format called **tidy data**. Tidy data has three main requirements:

1. Each column represents *one* **variable**
2. Each row represents *one* **observation**
3. Each cell contains only *one* **value**

Exercise 3
Take a look at our sample data. Is it tidy? Why or why not?

Tidying your dataset

Now we are going to try and turn our sample data into the tidy format. The most useful function for this step is usually `pivot_longer()` from the 'tidyverse' package, which takes data in the 'wide' format (a lot of columns, usually NOT tidy) and turn it into the 'long' format (a lot of rows, which is characteristic of tidy data). Since each row should be an observation, each row should have the looking time of only **one trial** instead of 10. Also, 'trial1' is *not* a variable. What we need to do is to take the values from trial1 to trial 10, and generate 10 rows of data for each baby with the trial number as the actual variable (which can range from 1 to 10 since there are 10 total trials). We're going to store this as a new object 'data_long' so that we can compare the two afterwards:

```
data_long <- pivot_longer(data = data, cols = c(trial1:trial10),
    names_to = "trial_number", names_prefix = "trial",
    values_to = "looking_time")
```

pivot_longer takes quite a few arguments. Use ?pivot_longer() to see the full description and see if you can decipher what each argument is doing

Take a look at data_long. It is now tidy and ready to be analyzed.

Data analysis and an intro to tidyverse

Exercise 4

Using the tools you learned earlier, check if there is a difference between the mean looking time of babies who saw the yellow character as the mean guy versus those who saw the blue as the mean guy.

Turns out for this study, we only want to compare the looking time of the first 4 trials. We can use the `filter()` function to set a criteria and pick out only the **rows** in which we are interested:

```
data_1to4 <- filter(.data = data_long, trial_number <
    5)
```

We can also do the same with **columns** if there are too many columns that we don't care about. For example, the 'age' column is redundant, so we can get rid of it with the `select()` function:

```
data_no_age <- select(.data = data_long, -age)
```

Here, the advantage of having column names with no spaces really shows: you can simply type in the column name age_month instead of having to wrap it with '`and type`age months'

You can also specify multiple columns at the same time using the following format:

```
data_no_age <- select(.data = data_long, -c(age,
    age_month, age_day))
```

We can also easily add new columns by using the `mutate()` function. Let's say we noticed that our coding software sytematically subtracted 2 seconds from everyone's looking time. To correct for that, let's add a new column with the corrected looking time:

```
data_correct <- mutate(.data = data_long, corrected_looking_time = looking_time +
    2)
```

Now you have most of the tools you need to do simple data wrangling and analysis!

Exercise 5

On your Rscript, write a series of code that does the following in sequence:

1. Load all the necessary packages
2. Read in sample_data_1.csv

Steps 1 - 4 should already be completed

3. Clean the column names
4. Convert to tidy format
5. Create a new column that adds 3.4 seconds to everyone's looking time
6. Filter out trials where corrected looking time exceeds 30 seconds
7. Create a column that log transforms everyone's looking time
8. Compare the log transformed looking time of babies who chose with vs against by calculating the mean, sd, number of observations, and SEM. Store the results as 'lt_by_choice'

Hint: log(x) is the math forumla for log transformation where x is the number to be transformed

Hint: SEM = sd/sqrt(n)

```
lt_by_choice
```

```
## # A tibble: 2 x 5
##   hypothesis  mean    sd     n    SEM
##   <chr>      <dbl> <dbl> <int>  <dbl>
## 1 Against     2.76 0.504    88 0.0538
## 2 With        2.72 0.506   184 0.0373
```

This is the expected result for lt_by_choice. Did you get the same thing?

Take a look at your answer for Exercise 5. How many lines did you have to execute? Is it 'readable'? If you left this script for a couple of weeks and come back later, will you remember what each line does?

It's often really complicated (and let's be real, quite annoying) to have several different steps until you reach the desired outcome. The more complicated the code is, the less likely you will understand what 'past you' was doing, and even less likely a stranger will know your thought process. In the next workshop, we will discuss documentation as well as introducing the 'pipe' function, which is a neat tool that allows you to streamline your codes.