Social Sciences Intro to Statistics

Week 2.1 Basics of R

Week 2: Learning goal - Apply basic Dplyr functions in R and produce graphs of continuous and categorical variables.

Introduction

Lecture overview:

- Practice dplyr functions
- Pipes %>%

Libraries we will use today

"Load" the package we will use today (output omitted)

- **you must run this code chunk**

```
library(tidyverse)
```

Pipes

What are "pipes", %>%

Pipes are a means of performing multiple steps in a single line of code. This helps us combine multiple operations together in a way that is concise. Pipes becomes very useful when we have many different operations in order to focus on new areas of the data.

- When writing code, the pipe symbol is `%>%`
- The pipe operator `%>%` is created by the **magrittr** package, which is not part of base R

- However, the magrittr package is automatically loaded when you load the tidyverse package

```
?magrittr::`%>%`
```

What are "pipes", %>%

pipe syntax: `LHS %>% RHS`

- `LHS` (refers to "left hand side" of the pipe) is an object or function
- `RHS` (refers to "right hand side" of the pipe) is a function

How pipes work:

- Object created by `LHS` becomes the first argument of the function (`RHS`) to the right of the `%>%` pipe symbol
- Basic code flow: `object %>% function1 %>% function2 %>% function3`

- Output of `some_function1` becomes the input (the first argument) of the function `some_function2` to the right of the `%>%` pipe symbol

Example of using pipes to calculate mean value of atomic vector

```
1:10 # an atomic vector
#>  [1]  1  2  3  4  5  6  7  8  9 10
mean(1:10) # calculate mean without pipes
#> [1] 5.5
1:10 %>% mean() # calculate mean with pipes
#> [1] 5.5
```

- no pipe: (1) write function; (2) data object `1:10` is 1st argument of `mean()`
- pipe: (1) write data object; (2) "pipe" (verb) object as 1st argument of `mean()`

```
rm(list = ls()) # remove all objects in current environment

getwd()
#> [1] "/Users/bellelee/Downloads/SSS Lectures/Lectures/Week 2 Lectures"
netflix_data <- read_csv("https://raw.githubusercontent.com/bcl96/Social-Sciences-Stats/main/
```

What are "pipes", %>%

Intuitive mnemonic device for understanding pipes

- whenever you see a pipe `%>%` think of the words "**and then...**"

Example: isolate all the dramas [output omitted]

- in words: start with object `netflix_data` **and then** filter productions under the main genre of drama

```
netflix_data %>% filter(MAIN_GENRE == "drama")
```

below code in words:

- start with `netflix_data` **and then** select a few vars **and then** filter **and then** sort **and then** investigate structure of object

```
netflix_data %>% select(TITLE, MAIN_GENRE, RELEASE_YEAR, SCORE) %>%
  filter(MAIN_GENRE == "drama", RELEASE_YEAR == "2017") %>%
  arrange(desc(SCORE)) %>% str()
#> tibble [9 x 4] (S3: tbl_df/tbl/data.frame)
#>  $ TITLE       : chr [1:9] "Anne with an E" "Innocent" "GLOW" "Tabula Rasa" ...
#>  $ MAIN_GENRE  : chr [1:9] "drama" "drama" "drama" "drama" ...
#>  $ RELEASE_YEAR: num [1:9] 2017 2017 2017 2017 2017 ...
#>  $ SCORE       : num [1:9] 8.7 8.4 8 8 7.8 7.8 7.7 7.6 7.5
```

More intuition on the pipe operator, %>%

Example: apply "structure" function `str()` to `netflix_data` with and without pipes

```
str(netflix_data) # without pipe
netflix_data %>% str() # with pipe
```

I use the `str()` when I add new `%>%`; shows what kind of object being piped in

- task: select a few vars from `netflix_data`; isolate dramas in 2017; sort descending by score (output omitted)

```
netflix_data %>% select(TITLE, MAIN_GENRE, RELEASE_YEAR, SCORE) %>% str()

netflix_data %>% select(TITLE, MAIN_GENRE, RELEASE_YEAR, SCORE) %>%
  filter(MAIN_GENRE == "drama", RELEASE_YEAR == "2017") %>% str()

netflix_data %>% select(TITLE, MAIN_GENRE, RELEASE_YEAR, SCORE) %>%
  filter(MAIN_GENRE == "drama", RELEASE_YEAR == "2017") %>%
  arrange(desc(SCORE)) %>% str()
```

Compare data tasks, with and without pipes

Task: Using object `netflix_data` print data for "drama" productions (`MAIN_GENRE == "drama"`)

```
# without pipes
filter(netflix_data, MAIN_GENRE == "drama")

# with pipes
netflix_data %>% filter(MAIN_GENRE == "drama")
```

Comparing the two approaches:

- "without pipes", object `netflix_data` is the first argument `filter()` function
- In "pipes" approach, you don't specify object `wwlist` as first argument in `filter()`
    - Why? Because `%>%` "pipes" the object to the left of the `%>%` operator into the function to the right of the `%>%` operator

Compare data tasks, with and without pipes

Task: Using object `netflix_data`, print data for "drama" productions for selected variables

```
#Without pipes
select(filter(netflix_data, MAIN_GENRE == "drama"), TITLE, RELEASE_YEAR, SCORE)
#With pipes
netflix_data %>% filter(MAIN_GENRE == "drama") %>% select(TITLE, RELEASE_YEAR, SCORE)
```

Comparing the two approaches:

- In the "without pipes" approach, code is written "inside out"

  - The first step in the task – identifying the object – is the innermost part of code
  - The last step in task – selecting variables to print – is the outermost part of code

- In "pipes" approach the left-to-right order of code matches how we think about the task
  - First, we start with an object ***and then*** (%>%) we use `filter()` to isolate first-gen students ***and then*** (%>%) we select which variables to print

`str()` helpful to understand object piped in from one function to another

```
#object that was "piped" into `select()` from `filter()`
netflix_data %>% filter(MAIN_GENRE == "drama") %>% str()

#object that was created after `select()` function
netflix_data %>% filter(MAIN_GENRE == "drama") %>% select(TITLE, RELEASE_YEAR, SCORE) %>% st
```

Aside: `count()` function

`count()` function from `dplyr` package counts the number of obs by group

Syntax [see help file for full syntax]

- `count(x,...)`

Arguments [see help file for full arguments]

- `x`: an object, often a data frame
- `...`: variables to group by

Examples of using `count()`

- Without vars in `...` argument, counts number of obs in object

```
count(netflix_data)
netflix_data %>% count()
netflix_data %>% count() %>% str()
```

- With vars in `...` argument, counts number of obs per variable value
  - This is the best way to create frequency table, better than `table()`
  - note: by default, `count()` always shows `NAs` [this is good!]

```
count(netflix_data, SCORE)
netflix_data %>% count(SCORE)
netflix_data %>% count(SCORE) %>% str()
```

pipe operators and new lines

Often want to insert line breaks to make long line of code more readable

- When inserting line breaks, **pipe operator `%>%` should be the last thing before a line break, not the first thing after a line break**

This works

```
netflix_data %>% filter(MAIN_GENRE == "drama") %>%
  select(TITLE, MAIN_GENRE, RELEASE_YEAR, SCORE) %>%
  count(SCORE)
```

This works too

```
netflix_data %>% filter(MAIN_GENRE == "drama",
                    RELEASE_YEAR != "2017") %>%
  select(TITLE, MAIN_GENRE, RELEASE_YEAR, SCORE) %>%
  count(SCORE)
```

This doesn't work

```
netflix_data %>% filter(MAIN_GENRE == "drama")
  %>% select(TITLE, MAIN_GENRE, RELEASE_YEAR, SCORE)
  %>% count(SCORE)
```

The power of pipes

You might be thinking, "what's the big deal?"

TasK:

- in one line of code, modify `netflix_data` and create bar chart that counts number of productions by genre, separately for production outside of U.S. vs. inside of U.S.

```
netflix_data %>%
  mutate( # create out-of-U.S. indicator
    out_us = as_factor(if_else(MAIN_PRODUCTION != "US", "outside-of-us", "inside-of-us"))
  ) %>%
  group_by(out_us) %>% # group_by "outside-of-us" vs. "inside-of-us"
  count(MAIN_GENRE) %>%  # count of number of productions by genre
  ggplot(aes(x=MAIN_GENRE, y=n)) +  # plot
  ylab("number of productions") + xlab("genre") +
  geom_col() +  coord_flip() + facet_wrap(~ out_us)
```

The power of pipes

TasK:

- in one line of code, modify `netflix_data` and create bar chart of median income (in zip-code) of prospects purchased by race/ethnicity, separately for in-state vs. out-of-state

```
netflix_data %>%
  mutate(out_us = as_factor(if_else(MAIN_PRODUCTION != "US", "outside-of-us", "inside-of-us"
  ggplot(aes(x = MAIN_GENRE, y = SCORE * NUMBER_OF_VOTES, fill = out_us)) +
  geom_bar(stat = "identity", position = "dodge") +
  ylab("Total Votes") + xlab("Genre") +
  coord_flip() + facet_wrap(~ out_us)
```

The power of pipes

Example R script from Ben Skinner, which creates analysis data for Skinner (2018)

- Link to R script

Other relevant links

- Link to Github repository for Skinner (2018)
- Link to published paper
- Link to Skinner's Github page
  - A lot of cool stuff here
- Link to Skinner's personal website
  - A lot of cool stuff here

Which objects and functions are pipeable

Which objects and functions are "pipeable" (i.e., work with pipes)

- function is pipeable if it takes a data object as first argument and returns an object of same type
- In general, doesn't seem to be any limit on which kinds of objects are pipeable (could be atomic vector, list, data frame)

```
# applying pipes to atomic vectors
1:10 %>% mean
#> [1] 5.5
1:10 %>% mean %>% str()
#>  num 5.5
```

But some pipeable functions restrict which kinds of data objects they accept

- In particular, the `dplyr` functions (e.g., `filter`, `arrange`, etc.) expect the first argument to be a data frame.
- `dplyr` functions won't even accept a list as first argument, even though data frames are a particular class of list

```
netflix_data %>% filter(MAIN_GENRE == "drama") %>% str()

as.data.frame(netflix_data)  %>% str()
as.data.frame(netflix_data) %>% filter(MAIN_GENRE == "drama") %>% str()

as.list(netflix_data)  %>% str()
```