Data Science Workshop Session 2

SCRIPTS Data and Methodology Center

Therese Anders Allison Koh

January 17, 2020

Data wrangling with `dplyr`

Introduction

Data cleaning and reshaping is one of the tasks that we end up spending the most time on. Today, we will be introducing the `dplyr` library. Together with `stringr` (string operations using regular expressions) and `tidyr` these packages offer functionality for virtually any data cleaning and reshaping task in `R`.

For an overview of the most common functions inside `dplyr`, please refer to the RStudio data wrangling cheat sheet https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf.

Functions in dplyr

dplyr does not accept tables or vectors, just data frames (similar to ggplot2)! dplyr uses a strategy called "Split - Apply - Combine". Some of the key functions include:

- select(): Subset columns.
- filter(): Subset rows.
- arrange(): Reorders rows.
- mutate(): Add columns to existing data.
- summarise(): Summarizing data set.

First, lets dowload the package and call it using the library() function.

```
# install.packages("dplyr")
library(dplyr)
```

Today, we will be working with a data set from the hflights package. The data set contains all flights from the Houston IAH and HOU airports in 2011. Install the package hflights, load it into the library, extract the data frame into a new object called raw

Using `select()` and introducing the Piping Operator %>%

Using the so-called **piping operator** will make the `R` code faster and more legible, because we are not saving every output in a separate data frame, but passing it on to a new function. First, let's use only a subsample of variables in the data frame, specifically the year of the flight, the airline, as well as the origin airport, the destination, and the distance between the airports.

Notice a couple of things in the code below:

- We can assign the output to a new data set.
- We use the piping operator to connect commands and create a single flow of operations.
- We can use the select function to rename variables.
- Instead of typing each variable, we can select sequences of variables.
- Note that the `everything()` command inside `select()` will select all variables.

```
data <-  raw %>%
  dplyr::select(Month,
```

Introducting `filter()`

There are a number of key operations when manipulating observations (rows).

- `x < y`
- `x <= y`
- `x == y`
- `x != y`
- `x >= y`
- `x > y`
- `x %in% c(a,b,c)` is `TRUE` if `x` is in the vector `c(a, b, c)`.

Suppose, we wanted to filter all the flights that have their destination in the greater Los Angeles area, specifically Los Angeles (LAX), Ontario (ONT), John Wayne (SNA), Bob Hope (BUR), and Long Beach (LGB) airports.

```
airports <- c("LAX", "ONT", "SNA", "BUR", "LGB")
la_flights <- data %>%
  filter(Dest %in% airports)
```

Helper functions

dplyr has a number of helper functions–and that is where the magic lies. These can be used with either select() or filter(). Here are some useful functions:

- starts_with()
- ends_with()
- contains()
- matches(): Every name that matches "X", which can be a regular expression (we will talk about regular expressions in session 5).
- one_of(): Every name that appears in x, which should be a character vector.

For example, let's create a data frame with all variables that contain the word "Time".

```
testframe <- raw %>%
  select(contains("Time"))
head(testframe)
```

Introducting `mutate()`

Currently, we have two taxi time variables in our data set: `TaxiIn` and `TaxiOut`. I care about total taxi time, and want to add the two together. Also, people hate sitting in planes while it is not in the air. To see how much time is spent taxiing versus flying, we create a variable which measures the proportion of taxi time of total time of flight.

```
la_flights <- la_flights %>%
  mutate(TaxiTotal = TaxiIn + TaxiOut,
         TaxiProp = TaxiTotal/Time)
```

Suppose, I only wanted to fly on weekends. Therefore, I create another variable that codes whether the flight is on a weekend or not and filter the data by this variable. Here, we introduce the `ifelse()` function that is tremendously helpful in data wrangling exercises. The syntax of `ifelse()` is as follows:

`ifelse(condition, if TRUE this, if FALSE this).`

```
la_flights <- la_flights %>%
```

Introducting summarise() and arrange()

One of the most powerful dplyr features is the summarise() function, especially in combination with group_by().

First, in a simple example, lets compute the average flight time from Houston to Los Angeles by each day of the week. Also, I want to know what the maximum total taxi time is for each day of the weak. Note, that because there are missing values, we need to tell R what to do with them.

```
weekday_time <- la_flights %>%
  group_by(DayOfWeek) %>%
  summarise(AverageTime = mean(Time, na.rm = T),
            MaxTaxiTotal = max(TaxiTotal, na.rm = T))
```

For legibility, lets reorder the output in ascending order using arrange(), with MaxTaxiTotal as a tie breaker.

```
weekday_time <- la_flights %>%
  group_by(DayOfWeek) %>%
  summarise(AverageTime = mean(Time, na.rm = T),
```

Putting it all together: The power of piping

In this example, I am starting all the way from the original `hflights` data set to demonstrate the power of the piping operator.

```r
carriers_new <-  raw %>%
  select(Month,
         DayOfWeek,
         Airline = UniqueCarrier,
         Time = ActualElapsedTime,
         Origin:TaxiOut) %>%
  filter(Dest %in% c("LAX", "ONT", "SNA", "BUR", "LGB")) %
  mutate(TaxiTotal = TaxiIn + TaxiOut,
         TaxiProp = TaxiTotal/Time,
         Weekend = ifelse(DayOfWeek %in% c(1,7), 1, 0)) %>
  filter(Weekend == 1) %>%
  group_by(Airline) %>%
  summarise(NoFlights = n()) %>%
  arrange(desc(NoFlights))
 head(carriers_new)
```

Saving files

Finally, lets save the output of our code. This time, we would like to save it in the .dta format. Remember to load the `foreign()` library before saving.

```r
library(foreign)
write.dta(carriers_new, "carriers_houston_la.dta")
```