Lab 2

Updated Mond, October 17th at 11:00a

Submit via CCLE by Friday, October 21st, 10pm

Directions: Create an R Markdown document to complete the tasks below. Include all necessary lines of code and explain your work using complete sentences. Both the code you write and the outputs from `R` should be included in the compiled/knitted HTML document. Submit both the `.Rmd` and `.html` files to CCLE. Name the files `#########-lab02.Rmd` and `#########-lab02.html` where the `#########` are replaced by your Bruin ID.

1 Reading and cleaning data:

1.1 Reading in Large data files

As the number of observations and/or variables increases, the time needed for `R` to read in data files grows. Luckily, `R` programmers have developed a few tricks to cut down the number of computations (and thus cut down on the time) required to load in larger data sets.

The `flights.csv` data file contains information about 336,776 flights that departed from the New York City area. This data set also contains 18 different variables.

Perform the following operations using the console or in an `R` script (that is, don't include the code you run below in your `.Rmd` document:

1. Install and then load the `readr` package. Read-in the `flights.csv` data using both the `read.csv()` function, from `base R`, and the `read_csv()` function, from `readr`. Name one data frame `flights_base` and name the other `flights_readr`.
2. Install and then load the `microbenchmarks` package. This package is helpful for measuring the time required to perform computations in `R`. Run the following code chunk (BUT first change `file = "REPLACE"` to the location of the `flights.csv` data on your computer):

NOTE: The `microbenchmark()` function works by running the different read csv functions a specified number of `times` each. It then outputs some summary statistics for the number of seconds each function took to run (we use the argument `unit = "s"` to specify the time-units).

NOTE: Reading in data can be computationally heavy (meaning it can take your computer a while to finish the operation). The code below might take a few minutes to finish running on your computer.

```
microbenchmark(
  read.csv(file = "REPLACE"),
  times = 10, unit = "s")
microbenchmark(
  read_csv(file = "REPLACE", progress = FALSE),
  times = 10, unit = "s")
```

Based on the steps you've performed so far, answer the following questions in your R Markdown document:

1. The `readr` functions decide the class of a variable by looking at the first 1,000 observations. The `base` functions decide the class of a variable by looking at ALL of the observations. Based on the `microbenchmark` outputs from the provided code chunk, report what the average number of seconds your computer needed to load in the `flights.csv` data using `read.csv()` and `read_csv()`. Write a few sentences explaining why you think one function read in the data faster than the other.

2. The `read.csv()` and `read_csv()` functions read in data slightly differently. `View()` the two data sets and, by looking at the values of the `origin` and `time_hour` variables, write down the class YOU would assign to each variable. Compare what you thought to the `class()` of the `origin` and `time_hour` variables in `flights_base` and `flights_readr`? How do the two functions differ in how they decide variable classes and how are they different from what YOU personally thought each class would be.

NOTE: A `POSIX` type variable is understood by your computer to be a variable dealing with dates and times. Meaning your computer will understand that dates from January come before February or that times earlier on a particular day come before later times on the same day.

James says: This question is really meant to expose you to a couple ideas:

1. Computers sometimes need time to finish computations (i.e. not everything is accomplished instantly).
2. Many `R` programmers prefer to have categorical variables be classified as `"characters"` instead of `"factors"`. This is the reason why categorical variables have different classes in `read_csv()` than they do with `read.csv()`.
3. Computers use special formats/classes when dealing with dates and times. We'll learn how to perform more operation with these `POSIX` type objects later in the course.

1.2 Cleaning data with `base` methods

1. Use `read.csv()` to read in the `planes.csv` data file (Don't specify any additional arguments except the file you're reading). Assign this data the name `planes`.
2. Change the `names` of the variables from all UPPERcase to all lowercase. *Hint*: This can be done manually, as you saw in the first lab OR can be done with the help of the `tolower()` function.
3. Use `R`'s bracket notation (i.e. `data[, ]`, `data[]` or `data$variable[]`) to change the `"-"` values to `NA` for the variables: `tailnum`, `manufacturer`, `model` and `speed`.
4. The `tailnum` variable is considered to be a `factor` when read using `read.csv()`. Based on the number of observations in the `planes` data and the number of levels for the `tailnum` variable, why is classifying `tailnum` as a factor (i.e. categorical variable) incorrect? Write how you determined the number of levels for the `tailnum` variable.
5. We can change the class of a variable using functions like: `as.character()`, `as.factor()`, `as.numeric()`, etc. We can do this using code like:

```
data$variable <- as.whatever(data$variable)
```

6. Change the class of the `tailnum` variable to `character` (i.e. permanently change the class in the data frame).
7. The `speed` variable should be a `numeric` class variable. Use `R`'s bracket notation to first create a numeric vector called `plane_speed` that only includes values that are NOT `NA`.
8. Use the `as.numeric()` function to have `R` convert `plane_speed` from a `factor` to `numeric` (BUT don't assign the values with `<-`). What do you notice about these `numeric` values compared to what they were when they were factors.
9. Convert `plane_speed` to characters. Then, convert the characters to numeric. How does the output compare to the output from the previous step?
10. Use `R`'s bracket notation to create a new data frame called `planes_new` that only includes the variables `tailnum`, `manufacturer` and `model`.

1.3 Cleaning data with `tidyverse` methods

Install and load the `dplyr` package to accomplish the following tasks:

1. Use `read_csv()` to read in the `airports.csv` data file.
2. Use the `select()` function to create a new data frame called `airports_new` that includes the variables: `FAA`, `LAT` and `LON`.

3. Use the `rename()` function to change the all UPPERcase variable names to all lowercase.

2 Joining data together

Shortly, we will have data in 3 different data frames: `planes_new`, `airports_new` and (in a moment) `flights`. Taking data from separate files and combining them together is a very common occurrence in statistics/machine learning/data science because different information is stored in different databases. In this exercise, we'll keep our `flights` information the same and supplement it with the additional information found in `planes_new` and `airports_new`. To accomplish this, we'll use the `left_join()` function from `dplyr`.

The first two arguments of `left_join()` will be two different data frames. The idea is to keep the first data frame (the one on the *left* when the two data frames are specified in the function) exactly as it is except that we'll add the information from the data frame on the right. To match up the data correctly, we need to specify a third argument to `left_join()` called `by`. The `by` argument will tell `left_join()` which variable(s) to use for matching.

It's important to note that, when we're performing a `left_join()`, the variable (often categories aka factor levels) we're using to match with must be a one-to-one mapping. This means that a specific factor level for an observation in the *left* data frame should only occur once in the data frame on the *right*. If there's more than one of the same factor level on the *right* then `R` can't decide which one to match to.

This is a hard concept to understand at first. But I'm hoping after you perfrom the `left_join()` below that it will become clearer to you.

1. Use the `read_csv()` function to read in the `flights.csv` data. Assign the data the name `flights`.
2. In a code chunk, output the dimensions of the `flights` data along with its variable names.
3. Notice that `flights` doesn't contain any information about the manufacturer or model of the airplane. Notice also that this information is found in `planes_new`.
4. Find a variable the both `flights` and `planes_new` have in common (Ideally, the variable will be an *ID* variable in `planes_new`. That is, a variable that uniquely IDs a specific manufacturer and model).
5. Use the `left_join()` function to create a new data frame called `flights_new` where:
    1. The `flights` data is used for the `x` argument.
    2. The `planes_new` data is used for the `y` argument.
    3. The variable the two data sets have in common is used for the `by` argument.
6. In a code chunk, output the dimensions of the `flights_new` data along with its variable names. If your join was successful, you should have the same number of observations as `flights` but two additional variables.
7. Look at the variables found in `flights_new` and the variables found in `airports_new`. There's two variables, one from `flights_new` and one from `airports_new` that contain the same categories but are named two different things.
8. Use another `left_join()` to combine the `flights_new` data with the `airports_new` data BUT this time specify the `by` argument as demonstrated below to match the differently named variables:

```
by = c("name_from_1st_data_frame" = "name_from_2nd_data_frame")
```

9. Assign the data from (8) to `flights_new` and use `rename()` to change the names of the two latest variables to: `origin_lat` and `origin_lon`. Then print the names and dimensions of `flights_new` again.
10. Perform a similar operation as you did in steps (8) and (9) but this time for the `dest` variable. Print the names and dimensions of your `flights_new` data one last time. You should have 24 variables if you've done everthing correctly thus far.
11. The formula to calculate distances between two points (on a Cartesian plane) is the following:

$$\sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}.$$

With your `flights_new` data frame, use the `mutate()` function to create a new variable called `distance`. Create this variable by plugging in `origin_lat`, `origin_lon`, `dest_lat` and `dest_lon` into the formula shown above.

NOTE: Calculating distances between places on Earth using this formula is misleading because it doesn't account for the curvature of the Earth. It's also strange because the distances are in terms of latitudes and longitudes and not kilometers or miles. I realize this, and I hope you'll forgive me for having you compute bizarre numbers. Later in the course (after we learn *loops* and *functions*), we can come back and use the haversine formula to compute the distances properly.

3 Combining data manipulations and plots

James says: In my experience, combining data manipulations with plots is an extremely powerful tool to wield. Combining these operations makes EDA faster as well more informative. It also helps keep things organized because you don't have to create so many data frames for different subsets.

Using `dplyr`, *pipes* and `ggplot2` is my go-to workflow for accomplishing much of the work I do. In this problem, I'll try to demonstrate why I like this workflow so much:

1. Load the `knitr` and `ggplot2` packages and fill in the blanks (`___`) below to create a table for the top 5 most popular airplane manufacturers:

```
flights_new %>%
  group_by(___) %>%
  summarize(planes_used = n()) %>%
  top_n(___) %>%
  arrange(desc(planes_used)) %>%
  kable()
```

2. Explain how each line of code from above transforms our `flights_new` data into the resulting table.
3. This table should reveal that `"AIRBUS"` and `"AIRBUS INDUSTRIE"` are considered to be two different manufacturers when they should be considered one. Fill in the blanks below to combine the two together:

```
___$___[___$___ == "___"] <- "AIRBUS"
```

3. Recreate the table in step (1), but this time only ouput the top 3 manufacturers.
4. Fill in the blanks below to create a faceted histogram of the distances traveled by each of the top 3 manufacturer's gets as they leave NYC:

```
flights_new %>%
  filter(manufacturer %in% c("___", "___", "___")) %>%
  ggplot() +
    geom_histogram(aes(x = ___), binwidth = 5) +
    facet_wrap(~___, ncol = ___)
```

5. Explain how each line of code from above transforms our `flights_new` data into the resulting plot
6. Based on the plot, how are the different manufacturers similar/different? (If your plot seems "squished" in the compiled R Markdown document, include code chunk options `fig.height` and `fig.width` to change the appearence in the document)
7. Using similar ideas as those shown above, create a heatmap (using `geom_bin2d()`) where the x-axis is the three origin cities and the y-axis is the top 5 destination cities for just Boeing aircraft.
8. Based on your heatmap, which city do more Boeing aircraft land at when the take-off from JFK airport? What other inferences can you make based on the plot?