Assignment 2B Instructions

Steven Spiegel

2023-08-26

PDAT 610G: Module 2b Assignment

Introduction

Please go through the lab available after question #5 and answer the following questions.

1. Edit the command below so that you're specifying either male *or* first-year. Are more or fewer individuals selected compared to the command below? Give a brief explanation of why that's the case.

```
FreshOrMales <-KimData %>%
  filter(Semester < 2 & Gender == "M")
```

2. A person's "Body Mass Index" is calculated by taking mass divided by height squared. If you're measuring in metric (kg and m), you're done. If you're measuring in pounds and inches (as we're doing here), you then have to multiply by 703. In other words, BMI = 703*(Weight/Height^2). Use the `mutate` command twice to first create the `BMI` variable, and then create a variable `Obese` whose value is TRUE for anyone whose BMI is greater than or equal to 30. Print the first six observations using the function head().
3. We've heard of the "freshman 15," the weight that many college students gain after their first year of all-you-can-eat dorm food. Use `group_by` and `summarize` to group students by `Year` and calculate mean BMI for each group. Does BMI seem to be higher for students who have been here more years?
4. Write the code to calculate the percentage of obese people in each year. Does this percentage show any clear trend?
5. Write two R commands using `dplyr`. One should calculate the average size of men and women's feet using all data points. The second should calculate the average size of men's and women's feet after removing the person with the giant feet. (Think about how you'll remove that person using a `dplyr` command.) How much of a difference in average shoe size did removing the outlier make?

Submission Method

- For best results, copy-and-paste all of the text from this document into a blank RMarkdown file. Create an R Markdown document containing both your code and any written responses or explanations.
- Make sure to number your answers so that I can easily see which questions you were answering
- The code for any question part should be contained in a single code block.
- When you're done, knit your code to a PDF file, and submit that file through Blackboard.

Assignment

The Story…

Dr. Hyun-Joo Kim has given a short survey to her introductory STAT classes for the past few years, and she then uses that data throughout the semester. The dataset found here has the information across six semesters (from about ten years ago) that she has used the survey. Her student assistant types the information from the paper sheets in by hand. Over time, this leads to dirty data. There was an especially big change in recording between rows 195 and 196, when she added additional questions about cell phone usage.

You can find the data set on Blackboard, along with this assignment. You might want to look at it in a spreadsheet before we begin, so you understand what we'll be doing with the data.

```
#Before you start, save this RMarkdown file into your project folder.
# Load the tidyverse package with dplyr commands.
library(tidyverse)
# Load the data file.
# First save the data Clean-KimData.csv from Blackboard in your own computer. Use the "import dataset"
#the code should look something like
#Clean_KimData <- read_csv(".../Clean-KimData.csv") ... is the directory where you #save the data

#Immediately, you will want to save your raw dataset into your local folder
write_csv(Clean_KimData, "Clean_KimData.csv")

# Finally, create a new copy of the data to keep the "clean" version clean.
KimData <- Clean_KimData
```

Data Transformations with `dplyr`

In this section, we'll talk about the five(six) `dplyr` verb that you want to know: 1. `filter`: Pick individuals by their values. 2. `arrange`: Reorder the rows. 3. `select`: Pick variables by their names. 4. `mutate`: Create new variables as functions of existing variables. 5. `group_by` and `summarize` : These two go together a. `group_by`: Mark data points as falling in different groups according to some variable. b. `summarise` or `summarize`: Collapse many values down to a single summary.

#For more Information The cheatsheet from RStudio has a list of helpful commands to include in mutate functions: [https://www.rstudio.com/resources/cheatsheets/] So does that free textbook. [http://r4ds.had.co.nz/transform.html#mutate-funs]

#Brief Aside: Pipes A pipe is a way to connect multiple lines of code. It basically means, "take the result of this line down to the next line." dplyr and several other tidyverse packages use a unique pipe that has no other meaning. %>% No, really, that's what it looks like. Yes, that's weird. But, you have to admit that you aren't going to use %>% for anything else.

Filter

Basic Syntax

The `filter` command selects certain observations. So, if we just want those who identified themselves as women or as only children in the dataset:

```
filter(KimData, Gender == "F")  # you need the quotes for characters or factors.
filter(KimData, Siblings == 0)  # you don't need the "quotes" for numbers
```

The "pipe" syntax is a great way to combine more than one transformation, and it's really the preferred format for these commands. For clarity, we typically put each piped command on its own line.

```
KimData %>%
  filter(Gender == "F")  #you need the quotes for characters or factors.
KimData %>%
  filter(Siblings == 0)  #you don't need the "quotes" for numbers
```

When you use one of these `dplyr` commands, the result is not, by default, stored in memory. If you want to save the results, you need to explicitly assign the results of the transformation to a variable. For example:

```
Boyz <- KimData %>%
  filter(Gender == "M")
View(Boyz)
```

Using View( ) causes a popup window which is not included in the actual knitted output document (the .pdf, .html or .doc file). We almost always want our output in our knitted document. So, if you want something in your output, use head( ) or summary( ) instead. Also notice that view has a capital "V."

More on "Relational Operators"

How do you select individuals of interest? Maybe you want to select individuals because a variable exactly matches some value. That's what the double equals sign, `==` means. But there are other comparisons you might want to do. What they all have in common is that they're expressions that end up giving you a `TRUE` or `FALSE` value. [Note: `TRUE` and `FALSE` are reserved values in R that stand for the logical results of these kinds of computations.They sometimes act like 0 and 1, but they are treated differently than a number, sometimes.]

- Equal to: `==` (more properly "Logically Equal to")
- Less than: `<`
- Less than or equal to: `<=`
- Greater than: `>`
- Greater than or equal to: `>=`
- Not equal to: `!=` (That's an exclamation point)

As an example, you might want to pick only students who have been at Truman at least 4 semesters:

```
UpperClass <- KimData %>%
  filter(Semester >= 4)
View(UpperClass)
```

You can also combine multiple comparisons using the "and" and "or" connectors.

- Or: `A | B` True if *either* of A or B is true.
- And: `A & B` True if *both* of A and B are true.

The example below shows code that finds all male students who are also first-year students.

```
FreshMales <- KimData %>%
  filter(Semester < 2 & Gender == "M")
```

Now is a good time to answer Question 1

There are also special logical commands that detect specific characteristics of your data. One you should know about is `is.na`. This command checks to see if the value of a variable is NA (i.e., a missing value).

The code below returns all individuals who did not give their number of siblings. This kind of blank is annoying, since we don't know for sure if the responder skipped this survey question or meant to say they were an only child. Since there's another question about being an only child, we could fill in some of the blanks.

```
NASibs <- KimData %>%
  filter(is.na(Siblings))
```

Arrange

The `arrange` command sorts your data into a certain order. So, if we want the data in order those with the fewest number of semesters to the greatest number of semesters, we could run

```
KimData %>%
  arrange(Semester)
```

On the other hand, if we wanted to order by *decreasing* number of semesters, the `desc()` command could be put around `Semester`.

```
KimData %>%
  arrange(desc(Semester))
```

Select

The `select` command selects only certain variables. This is especially helpful for giant datasets, so you can make something smaller to work with. Let's make a new data set with only the variables that describe something "physical" about each student.

Note that `Shoe Size` needs the funny back quotes (like the triple-backquotes used for RMarkdown chunks) around it because it has a space in the name. The quotes will appear when you TAB-complete the name as long as you've already run a command that loads the data set in. One reason some people like read.csv( ) over read_csv is that read_csv keeps variable names like their work in the original file, but read.csv replaces all the spaces with periods, so it would be imported as Shoe.Size instead of `Shoe Size`. Of course, keeping it authentic to the original is also a good thing.

```
KimDataPhysical <- KimData %>%
  select(Semester, Gender, `Shoe Size`, Height, Weight, Handed)
head(KimDataPhysical)
```

If you want to have "all but" a certain number of variables, give a list with minus signs in front. If we decided we didn't want the `Handed` variable, for example, we could get rid of it:

```
KimDataPhysicalNoHands <- KimDataPhysical %>% select(-Handed)
head(KimDataNoHands)
```

We could also select particular columns by their column number. This can be problematic or annoying, but may also be easier for datasets that do not have variable names included. Remember that the c() command makes a vector of numbers. So, this will select all of the numerical variables, but I did it the hard way, by going through and counting by hand which variables those are.

```
KimNumData <- select(KimData, c(1,3,5:7, 12, 13:22))
View(KimNumData)
```

Rename is a variation of select that simply changes the name of a variable. These two commands do almost the same thing.

```
KimDataPhys2 <- rename(KimDataPhysical, Shoe.Size = `Shoe Size`)
KimDataPhys3 <- select(KimDataPhysical, Shoe.Size = `Shoe Size`, everything())
```

You can see how the name changes in the environment window on the right. It is possible, but annoying, to rename variables directly with the select command. You can see in KimDataPhys3 that it weirdly moved Shoe.Size to the first column. the everything( ) command, as you might guess, literally keeps everything. Renaming variables can be especially nice when the original dataset has really long or confusing names, but it makes it harder to connect back to the original later.

Mutate

The `mutate` command creates a new variable by applying a function to an existing variable or variables. This is especially helpful for data cleaning, or changing units or data types.

Suppose we wanted to know the number of *years* that a student had been at Truman, rather than the number of semesters. Using `mutate`, we could create the `Year` variable by dividing Semester by 2, then rounding up with the `round` command.

```
KimDataPhysical <- KimDataPhysical %>%
  mutate(Year = round(Semester/2))
```

```
head(KimDataPhysical)
```

The `mutate` command can also be helpful when you want to create a "logical" variable that's TRUE when a certain condition is true and FALSE otherwise. For example, maybe we want to label all seniors who have had at least 7 prior semesters:

```
KimDataPhysical <- KimDataPhysical %>%
  mutate(Senior = Semester >= 7)
head(KimDataPhysical)
```

mutate can get tricky pretty quickly. Maybe we want to convert Gender into a factor, and turn the missing one into an NA.

```
KimDataP4<- mutate(KimDataPhysical, Gender=as.factor(sub("other", NA, Gender)))
```

Now is a good time to answer Question 2

Grouping and Summarizing

The `group_by` command doesn't do much by itself. It merely tells R that some of the individuals in your data set are grouped together according to the values of one or more of the categorical variables. Here's what grouping by Gender looks like. Can you see the slight difference in output between the regular data set and the grouped version?

```
KimDataPhysical
KimDataPhysical <- KimDataPhysical %>%
  group_by(Gender)
KimDataPhysical
```

But, combined with `summarize`, the two commands become a "magical machine" (Dr. Alberts' words) that creates summary tables and pivot tables. The `summarize` commands collapses groups down to one or more descriptive statistics that are calculated from each group. It has the following form: `summarize(new.summary.variable = function(old.variables))`

where you replace each of the variables with actual variable names and `function` with an actual function. You can create multiple summary variables in the same command.

```
KimDataPhysical %>% group_by(Gender) %>%
  summarize(MHeight = mean(Height, na.rm=TRUE), MWeight = mean(Weight, na.rm=TRUE))
```

That these commands NEED that `na.rm=TRUE` option added. Otherwise, the missing values (NA) from people who didn't enter their height and weight would keep R from calculating a mean. Here's what you get without 'na.rm':

```
KimDataPhysical %>% group_by(Gender) %>%
  summarize(MHeight = mean(Height), MWeight = mean(Weight))
```

Now is a good time to answer Question 3 at the bottom of the assignment ### Calculating Counts and Percentages

The `n()` command without any arguments inside the parentheses will count the *number* of individuals within a group.

Another specific use of grouping and summarizing is in calculating *percentages*, or *proportions*. Here's the code to calculate the percentage of each gender that are senior students:

```
KimDataPhysical %>% group_by(Gender) %>%
  summarize(n = n(), Senior.Pct = mean(Senior==TRUE, na.rm=TRUE))
```

This code works because a comparison function like 'Senior == TRUE' creates a list of TRUE and FALSE values for each individual. If you try to do calculations with these TRUE and FALSE values, R converts then to 1's and 0's, where 1 stands for being a senior. So, the mean of that list of 0's and 1's becomes (Number TRUE)/(Total Observations), which is a percent. Pretty nifty!

Multiple Groups

You can create multiple groups and summary statistics. For example, you could break the sample down by both `Gender` and `Year` just by listing both of these variables in the `group_by` command. it might get complicated to display both groups at once, but sometimes, it's just what you need.

Now is a good time to answer Question 4 ##Return of pipes We can stack these verbs together with pipes. Remember that with a pipe %>%, each line takes what sent from the previous line. A pipe chain is a simple script that can do very non-simple things.

```
tall_Kim <- KimData %>%              #This line just declares the dataset
  group_by(Gender,Semester) %>%      #what shall we group by?
  summarize(count = n(),                 #this counts up how many in each thing
            aveht=mean(Height, na.rm=TRUE)) %>%     #this finds the average height
  filter(count > 2) %>%                  #this gets rid of low-n categories
  arrange(aveht)                     #this sorts them from shortest to tallest
tall_Kim                         #See what you made?
```

This script groups individuals by Gender and Semester, counts the number in each cell, then computes the average height. It eliminates low-n cells, then sorts it smallest to largest. This example is weird, but this skill is very handy. Also notice that this script is long and wordy, but "literate" and easy to understand.

Data Cleaning

Commands in the `dplyr` package are often useful for data cleaning.

When you find an outlier, you shouldn't just throw it out. However, if you investigate, can't find a good reason for it, can't find the correct value for it, and it's clearly some sort of mistake, then likely removing the entire individual is a reasonable thing to do. Let's see the effect of removing the individual with the "giant" feet.

Now is a good time to answer Question 5, the final question in this assignment