R: An Introduction

Background

Our time in class will be brief relative to the many objectives we've set out to accomplish. Practicing R outside of class is imperative, which you can do either on your own machine, by visiting the computer lab or by using one of many online sites that you can run R. I don't recommend using internet sites (e.g., RStudio Cloud) to use R on the regular because they are slow, unreliable and tend to inexplicably erase projects. But they do allow us to communicate remotely, which will likely become necessary as you work on your research projects. So while I advise against using R on the internet for out of class work, I think you should at least create an RStudio Cloud account. As its name suggests, it's RStudio (the application we use in class to run R code), but it's located on the cloud. You can access the site at https://rstudio.cloud/.

You'll write your R code on R script so that you can save your work. This course builds on past material, so you'll want to keep a folder where you save all your work.

We'll frequently be using notes like the present ones to learn R. Your job is to create an R script, follow along by running the code, and also adding your own code. Any code executed below, whether it be a simple command that returns the mean of some numbers or generates a plot, should be included in your script.

Using Comments in R

When you run R code, R wants to interpret everything you command it to run. You'll frequently find yourself needing to run code you've ran in the past, but can't remember exactly what the code means or what it will do when executed. Using comments to remind yourself (and others viewing your script) what the code means and does is, perhaps, the most helpful and underappreciated way to get the hang of R. The best advice I can give you at this early stage is the following: **Comment the hell out of every script you make**. Be sure to make your comments as descriptive as possible and don't be afraid to write lengthy comments; better to have too much description than not enough.

To insert comments into your script use the `#` symbol at the beginning of your comment. For example, run the following lines of code:

```
# six + six
6 + 6
```

```
## [1] 12
```

Notice how R simply ignored the line beginning with the `#`; that line is for your understanding and not for R. I cannot understate the importance of inserting lots of comments in your script, especially when you're doing something complicated or counterintuitive. Trust me, you'll thank yourself later in the semester when you're working on your research projects and you remember you need some code from, like, week 2 but don't know what the commands look like. Your comments will save you a lot of trouble.

Variables and Assignment

Most of your R code will require you to store it for later use. To do so you need to assign the object[1] a name. To assign an object a name use the `<-` symbol. For example, run the following code:

[1] By way of example, the Consumer Price Index in the FRED database is named CPIAUCSL. Not a very descriptive name. Simplifying its name to `cpi` in R will keep you from constantly asking yourself "What is CPIAUCSL?".

```
multiply <- 5*6
```

Notice that, by assigning `5*6` a name, R didn't return the number 30 (the produce of 5 and 6). Here is what R understood from the above command: The name of `5*6` is `multiply`. To get R to return the result, you need to refer to it by its name, like this:

```
multiply
```

```
## [1] 30
```

We'll assign names to most of the objects we'll use in this class. You should always use descriptive names for these objects to help you remember what they are. For example, if you have data on unemployment rates you might name it `ur`.

Functions

Functions allow us to manipulate objects. Like mathematical functions, R functions require arguments and these arguments are always enclosed by parenthesis `()`. One of the most basic and frequently used function is the concatenate function, written as `c()`. You'll use it to create a vector (an ordered sequence). For example, create a vector called `x` with some randomly chosen numbers:

```
x <- c(3, 6, -2, 0, -27)
```

What's the mean of `x`? Use the `mean()` function to find out:

```
mean(x)
```

```
## [1] -4
```

Most of the functions we'll encounter have multiple arguments that then need to separated by commas. Further, functions often have default values. We can specify values by putting the name of the argument followed by an equal sign. Here is an example of a function that allows us to view the first n rows of a dataframe. The default value is six, but we can change it by using the `n =` argument, as shown below.

```
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
head(mtcars, n = 3)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

Packages

Later in the semester we'll create our own functions. Fortunately, though, most of the functions we'll need are already built into base R or else can be accessed through packages. An example of a function built into base R is the `head()` function we just used. But, there are many functions that are not built into base R, and so we'll need to install and load packages.

Packages are collections of functions, data and code that are created and made available by other R users. To load a package into R use the `library()` function, like this: `library(name_of_package)`. If we haven't already downloaded the package, you first need to install it using the `install.packages()` function: `install.packages("name_of_package")`. (Note the quotes around the name of the package you're installing!) Once you've installed a package you only need to use the `library()` function to load the package into R.

We'll use the **tidyverse** package every day this semester, so before each session run the command:

```
# Load the tidyverse package into R
library(tidyverse)
```

It's a good habit to put `library(tidyverse)` at the top of every R script you're working with.

To see which packages you've already loaded into R, click the Global Environment tab in the Environment pane (upper-left pane). If see the package you want to use listed under the Global Environment tab, then you don't need to load it into R; it's ready to use.

One package we'll use later on is the **car** package. To get some practice, let's install and load **car** as explained above:

```
install.packages("car") # Install the car package
library(car) # Load the car package into R
```

Dataframes

The term *dataframe* refers to datasets, and a dataframe is essentially a dynamic Excel spreadsheet. Dataframes consist of rows (observations) and columns (variables). Some dataframes are built into base R and so we don't need to install packages to access them. We have already seen an example of this with the `mtcars` dataframe.

A typical work flow may start with us importing data and saving it as a variable, like this:

```
df <- mtcars # Rename mtcars as df
```

It wasn't necessary to assign the `mtcars` dataframe a new name, but it's frequently convenient to do so. The reason why, and one we'll encounter multiple times this semester, is that when we download data from websites (e.g., the Federal Reserve's FRED database), their given names are awkward alphanumeric strings and non-descriptive of the dataset.[2]

The first thing you should do after importing a new dataset is to check whether the data are in proper order and in the proper format. To view the data use the `View()` function:

```
# View data frame df
View(df)
```

The `View()` function gives you access to the entire dataframe; it's like looking at an Excel spreadsheet, but it's only for viewing purposes and not for manipulating data.

In addition to viewing your data to verify it looks like it should, you also need to verify the data are in the proper format. That is, you need to check to see if R interpreted numbers as numbers, dates of the year as dates of the year, and so on. To verify the data types of your data, use the `glimpse()` function from the **tidyverse** package as follows:

```
# Verify data types of dataframe df
glimpse(df)
```

We'll have much more to say about data types and dataframes later on.

[2] By way of example, the Consumer Price Index in the FRED database is named CPIAUCSL. Not a very descriptive name. Simplifying its name to `cpi` in R will keep you from constantly asking yourself "What is CPIAUCSL?".

Identifying Columns in a Dataframe

Often we need to access to a single column of a dataframe. For example, we may want to use the `mean()` function to get the mean of a column of numbers. Working with a particular column requires that you *extract* it from its dataframe. To extract a column use the $ symbol as in the following example:

```
# Take the mean mpg
mean(df$mpg)
```

```
## [1] 20.09062
```

Notice what's happening: We want to find the mean of the `mpg` column, but we need to first specify which dataframe (here `df`) the `mpg` column is located. So when we want to work with a specific column from a dataframe:

1. Specify the dataframe that column belongs to.
2. Use the `$` symbol to tell R we want extract a column from this dataframe.
3. Specify the column you want from the dataframe.

Thus, the argument inside the last command (`df$mpg`) means: *Extract the column* `mpg` *from the dataframe* `df`.

Graphing with `ggplot()`

The gold standard for graphing and for data visualization is an R package called **ggplot2** and its corresponding function `ggplot()`. Typical plots made with the `ggplot()` function have the same basic syntax:

```
# Generic syntax for a plot
ggplot(data = name_of_dataframe) +
  geom_***(mapping = aes(x = column_name, y = column_name))
```

To create a plot:

1. Begin by calling the `ggplot()` function.
2. The name of your dataframe is the first argument of the `ggplot()` function.
3. After specifying your dataframe you need to decide how you want the data to be displayed. Do you want you plot to be a scatterplot or a histogram or a bar chart? That's what the `geom_***()` function is for. The `***` is just a placeholder for which type of geometry you decide on (e.g., `geom_point()` or `geom_histogram()` or `geom_bar()`.)
4. The final step is to determined the variable(s) you want to appear on your plot. That's what the `aes()` function is for, which is short for "aesthetics." Inside `aes()` indicate which variable you want displayed on the x-axis and, if necessary, which you want on the y-axis. These variables need to be column names of the dataframe you specified in Step 2.

Let's make plots using the data in the **car** package that we've already installed and loaded. Next, load **ggplot2**, which should be preinstalled so all we need to do is load the package:

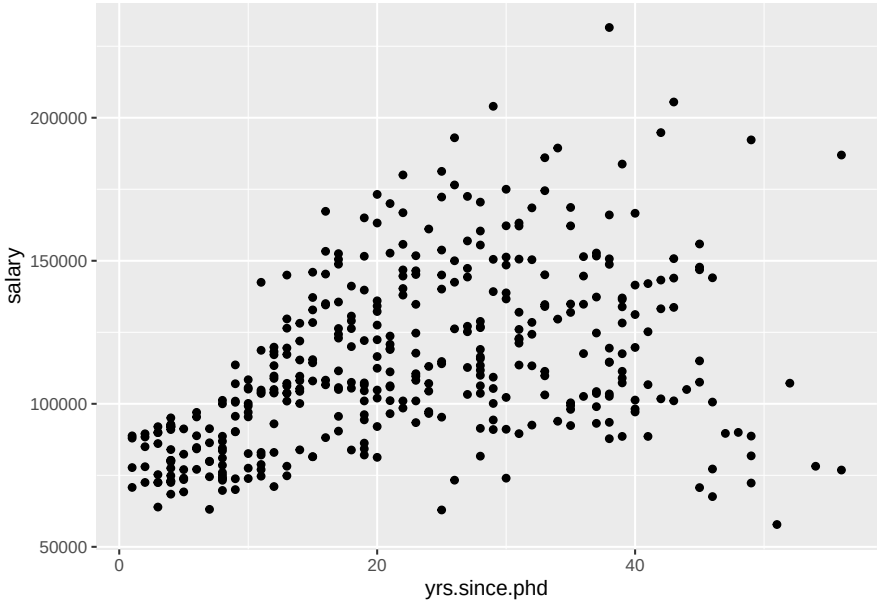
```
# Load the ggplot2 package
library(ggplot2)
```

```
## Registered S3 methods overwritten by 'tibble':
##   method     from
##   format.tbl pillar
##   print.tbl  pillar
```

Now, we'll use a dataset in the **car** package called `Salaries`. To see what kind of data we're dealing with run the `View()` and Let's create a scatter plot of the years since professors received their PhD versus their salaries:

```
## Loading required package: carData
```

```
# Create a scatterplot
ggplot(data = Salaries, aes(x = yrs.since.phd, y = salary)) + geom_point()
```



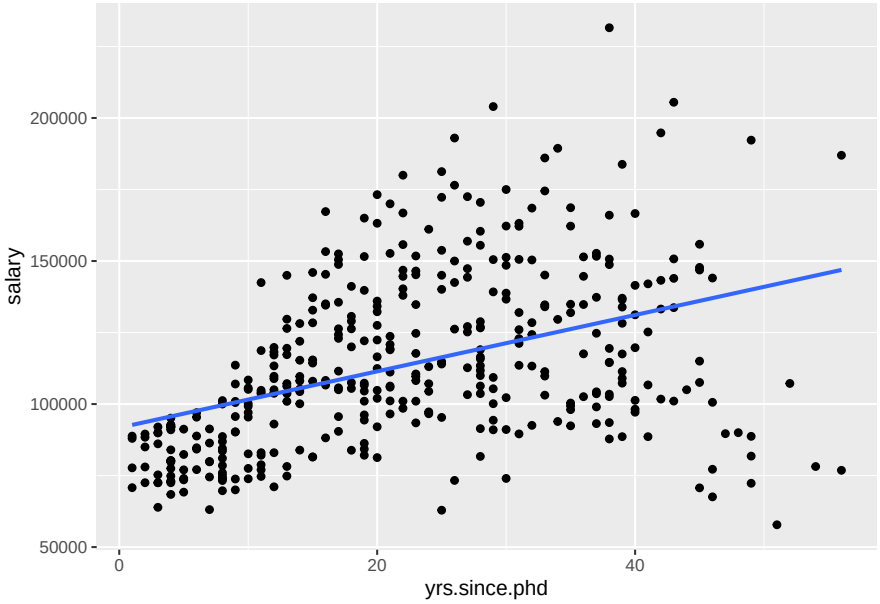
Notice that, when specifying the x and y variables, we didn't use the dataframe$column syntax. That's because we already told R which dataframe we want to use, `Salaries`, and so R then knows which columns make up the dataframe.

From personal experience I understand that **ggplot2** has a large learning curve. There are two primary reasons for using it: (1) there's broad consensus that it's the best graphical package available and (2) the graphs are easy to customize.

Let's consider some of the customization options of **ggplot2**. First, let's add a regression line to represent how a professor's salary tends to change the greater the number of years has elapsed since receiving their PhD. To do so, we add the `geom_smooth()` function to the previous code with two arguments, as shown below:

```
# Add a regression line to the scatterplot
ggplot(data = Salaries, aes(x = yrs.since.phd, y = salary)) + geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```

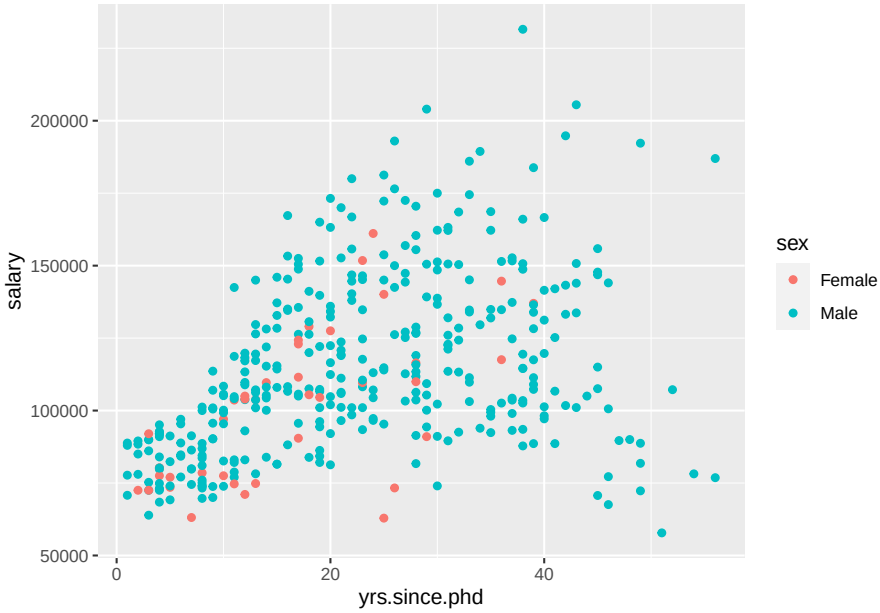
```
## `geom_smooth()` using formula 'y ~ x'
```



Judging from the regression line, salaries tend to be lower for professors who received their PhD more recently.

The `Salaries` dataframe also includes a column called `sex` that indicates the gender of each professor in the dataset. Let's return to a simple scatterplot without the regression line, but now let's distinguish between female and male professors. We do this by including a third argument in the `aes()` called `color`:

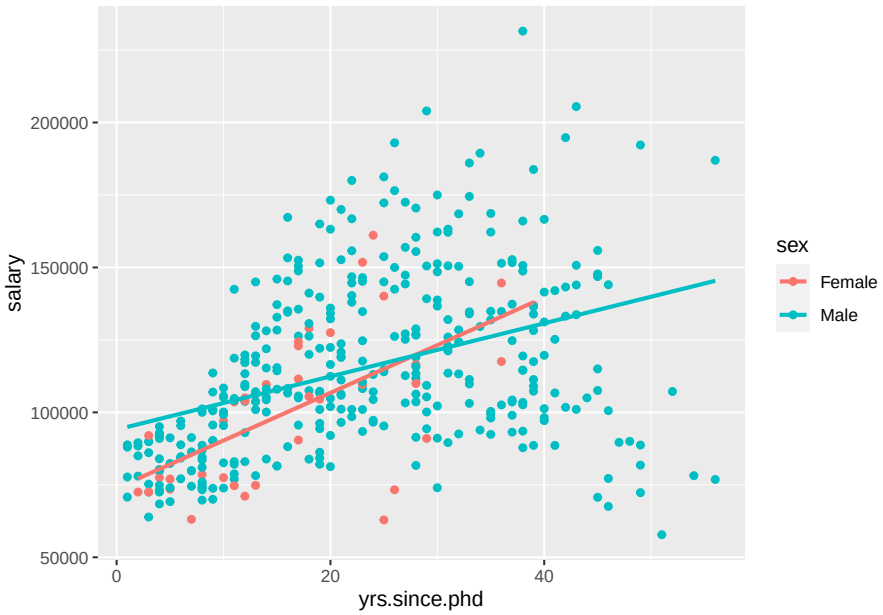
```
ggplot(data=Salaries, aes(x=yrs.since.phd, y=salary, color=sex)) + geom_point()
```



Finally, let's add a regression line as before to this new scatterplot:

```
ggplot(data = Salaries, aes(x = yrs.since.phd, y = salary, color = sex)) + geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Note that by adding the third variable, `sex`, we now have two regression lines: One indicating the average change in salaries for females and a seperate one for males.