R Refresher: Notebooks, Notation and Visualization

UC Berkeley Social 273M: Computational Social Science, Part B

Spring 2021

Contents

Learning Objectives **2**

Basic R Commands **2**

Importing and Manipulating Data **5**

A note on data.table vs data.frame and dplyr **9**

Generating Random Numbers **9**

ggplot **11**

R Markdown **14**

The Header 14

Basics 14

Making PDFs using R Markdown **16**

How to export from RStudio Cloud and save onto your own computer **16**

Latex symbols and useful math equations for Causal Inference **17**

Review of Notation **18**

Observed Data - Review 18

Variance and Covariance - Review 18

Notation Example Problems **19**

Notation Exercise 1 19

Notation Exercise 2 19

Learning Objectives

1. Basic R commands and useful functions in tidyverse
2. Import and manipulate data
3. A note on data.table vs data.frame and dplyr
4. Introduce functions to generate random numbers
5. R markdown
6. Latex symbols and math equations for Casual Inference
7. Review of Notation

Basic R Commands

R shares many similarities to Python. Both are object oriented programming languages, and thus the same conventions around variable assignment, functions, for loops etc. will be the same. There are differences in terms of syntax, but if you're comfortable with Python, making the switch over to R will be relatively painless. Both languages complement each other well - Python is well suited to text analysis, computer vision, and software development while R has more options for state-of-the-art causal inference methods. These differences mostly come down to the availability of open source libraries that specialize in these tasks, rather than inherent difference between the languages themselves. Both are increasingly well suited to machine learning generally through the sklearn library in Python and the tidymodels library in R.

Let's start with variable assignment. Here you'll notice that this looks basically the same as what we did with Python, but in R it's more standard to use the carrot "<-" operator to do variable assignment instead of "=", but both work. We can also check booleans exactly the same way:

```
#Assignment using <- or =
x <- 1
x
```

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

```
y = 2
y
```

```
## [1] 2
```

```
#Check equality using ==
x == y
```

```
## [1] FALSE
```

```
x != y
```

```
## [1] TRUE
```

In Python, we mainly worked organized data in dataframes, lists, and dictionaries, and data was usually something like an integer, float, or string. R has similar concepts, but introduces a few new data types that are common as well. In particular, R introduces the notion of a `vector` which is a special data type that contains elements of the same type. We can concatenate a vector by using the `c()` function.

```r
pets_vector <- c('dog', 'cat', 'parrot')

pets_list <- list('dog', 'cat', 'parrot')

num_vector <- c(1, 2, 3)

mixed_data <- c(1, 'dog', 2, 'cat')

dog <- 'dog'

class(pets_vector)
```

```
## [1] "character"
```

```r
class(pets_list)
```

```
## [1] "list"
```

```r
class(num_vector)
```

```
## [1] "numeric"
```

```r
class(mixed_data)
```

```
## [1] "character"
```

```r
class(dog)
```

```
## [1] "character"
```

Notice how when we checked the class of both `pets_vector` and `dog` it returned a character, but a list when we explicitly called list. We got “numeric” with our vector of numbers, but then “character” again when we mixed strings and numbers - R converted these all to characters! Vectors have important consequences for computation because many methods in R are vectorized - meaning they operate on all items in a vector simultaneously. This makes vectorized methods a powerful alternative to for loops. Before we see how this works, let’s see how we construct for loops:

```r
for (pet in pets_vector) {
  print(pet)
}
```

```
## [1] "dog"
## [1] "cat"
## [1] "parrot"
```

The syntax is a little different from Python but otherwise looks and feels pretty similar! Now let’s check out the difference between vectorized methods and for loops by finding the log of numbers 1 to 1,000,000.

```r
numbers <- 1:1000000

print(system.time(numbers_log_vec <- log(numbers)))
```

```
##    user  system elapsed
##    0.06    0.01    0.08
```

```r
numbers_for_loop <- list()

print(system.time(for (number in numbers) {
  numbers_for_loop[number] <- log(number)
}))
```

```
##    user  system elapsed
##    3.34    0.28    4.00
```

This is a huge difference! This particular example is a bit straightforward and the difference between a fraction of a second and three seconds doesn't seem like a big deal - but this can make a huge difference with big data!

We'll introduce two more concepts here to show you the power of vectorization. First, we'll define a function that takes a number, subtracts 1, and then returns the result. Again, notice the slight differences in syntax between R and Python. Now what if we wanted to "apply" this function to every element in our vector - or say every row in a dataframe? We can use an `apply` method (in this case `sapply` which returns a vector) to apply the function to every element in the vector. Again notice how quickly this goes!

```r
minus_one <- function(num) {
  nums_minus_one <- num - 1
  return(nums_minus_one)
}

minus_one_list <- list()
system.time(for (num in numbers) {
  minus_one_list[num] <- minus_one(num)
})
```

```
##    user  system elapsed
##    5.69    0.12    6.22
```

```r
system.time(apply_minus_one <- sapply(numbers, minus_one))
```

```
##    user  system elapsed
##    1.89    0.11    2.17
```

```r
apply_minus_one[1]
```

```
## [1] 0
```

These examples seem straightforward, but vectorization is a powerful concept that makes R a versatile tool for computational social science. Mastering the use of vectors, functions, and apply methods can dramatically speed up your computations.

Importing and Manipulating Data

Now let's move to data frames. These work basically the same way they do in Python - we organize data in rows and columns and we can have numbers, characters, and even other dataframes/lists/vectors as values. Here we will import the "strength.csv" data. The dataset contains information on subjects (id) who had one of three exercise treatments (tx). Their strength rating (y) is tracked over time (7 weeks).

Before we start looking at the data though, we'll need to deal with some issues with how R handles working directories. While Jupyter recognizes your current working directory as the one where your .ipynb notebook lives, R does not do this by default. One way to make this work is to go to Session -> Set Working Directory -> To Source File Location. This is a bit cumbersome and not always reproducible though. Luckily, the `here` library should be able to help us out:

```
#install.packages('here')
here::i_am('R Refresher.Rmd')
```

```
## here() starts at C:/Users/Anike/Documents/Computational-Social-Science-Training-Program/Causal Infer
```

```
library(here)
```

```
## Warning: package 'here' was built under R version 4.0.3
```

```
setwd(here())
```

Now that we have our working directory set, let's read in our dataframe. In addition to the head() function, investigate the output of the names() and dim() functions.

```
#Tidyverse Import
library(readr)
df <- read_csv("../../data/strength.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   tx = col_double(),
##   y = col_double(),
##   time = col_double()
## )
```

```
head(df)
```

```
## # A tibble: 6 x 4
##      id    tx     y  time
##   <dbl> <dbl> <dbl> <dbl>
## 1     1     1    85     1
## 2     1     1    85     2
## 3     1     1    86     3
## 4     1     1    85     4
## 5     1     1    87     5
## 6     1     1    86     6
```

```r
names(df)
```

```
## [1] "id"   "tx"   "y"    "time"
```

```r
dim(df)
```

```
## [1] 399   4
```

We can also manipulate data using functions from the tidyverse package. For example we can add a column using the mutate() function. We can select (or deselect) a subset of columns using select(). Note that in the example of the select function below there is a negative sign preceding y_2, meaning we are selecting everything except y_2.

```r
#Add a column that creates a new variable y_2 as y times 2 using tidyverse tools
#install.packages("dplyr")
#library(tidyverse)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
df = mutate(df, y_2 = y*2)
head(df)
```

```
## # A tibble: 6 x 5
##      id    tx     y  time   y_2
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1     1    85     1   170
## 2     1     1    85     2   170
## 3     1     1    86     3   172
## 4     1     1    85     4   170
## 5     1     1    87     5   174
## 6     1     1    86     6   172
```

```r
#Note this is equivalent to the base R command: df$y_2 = (df$y)*2

#Select or drop columns using the select() function
df = select(df, -y_2)
head(df)
```

```
## # A tibble: 6 x 4
##      id    tx     y  time
##   <dbl> <dbl> <dbl> <dbl>
## 1     1     1    85     1
## 2     1     1    85     2
## 3     1     1    86     3
## 4     1     1    85     4
## 5     1     1    87     5
## 6     1     1    86     6
```

tidyverse also provides the pipe operator. This operator (%>%) can be used to pass the output from one function to another, such as when you are performing a series of operations. This is probably the most powerful and intuitive functionality provided by the tidyverse - basically the output of the last step becomes the input to the next step.

```
#piping
tidy_df <- df %>%
  rename(treatment = tx) %>%
  mutate(rescale_y = y * 1000)
head(tidy_df)
```

```
## # A tibble: 6 x 5
##      id treatment     y  time rescale_y
##   <dbl>     <dbl> <dbl> <dbl>     <dbl>
## 1     1         1    85     1     85000
## 2     1         1    85     2     85000
## 3     1         1    86     3     86000
## 4     1         1    85     4     85000
## 5     1         1    87     5     87000
## 6     1         1    86     6     86000
```

The filter() function is used to select a subset of rows.

```
#filtering
df_id1 <- filter(df, id == 1)
dim(df_id1)
```

```
## [1] 7 4
```

```
#Note: to achieve the same result in base R we could do:
#df_id1 <- df[df$id == 1, ]

#We can also filter by more than one condition
filter(df, id == 1, time == 1)
```

```
## # A tibble: 1 x 4
##      id    tx     y  time
##   <dbl> <dbl> <dbl> <dbl>
## 1     1     1    85     1
```

```r
#Or using an OR statement
filter(df, time == 1 | time == 7)
```

```
## # A tibble: 114 x 4
##       id    tx     y  time
##    <dbl> <dbl> <dbl> <dbl>
##  1     1     1    85     1
##  2     1     1    NA     7
##  3     2     1    80     1
##  4     2     1    NA     7
##  5     3     1    78     1
##  6     3     1    77     7
##  7     4     1    84     1
##  8     4     1    85     7
##  9     5     1    80     1
## 10     5     1    80     7
## # ... with 104 more rows
```

We can also summarize data within groups using the group_by() and summarize() functions.

```r
#summarizing
tidy_df %>%
  group_by(time) %>%
  summarize(mean_strength = mean(y, na.rm=TRUE),
            standard_deviation_y = sd(y, na.rm=TRUE))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 7 x 3
##    time mean_strength standard_deviation_y
##   <dbl>         <dbl>                <dbl>
## 1     1          80.2                 2.98
## 2     2          80.8                 3.13
## 3     3          80.9                 3.36
## 4     4          81.4                 3.20
## 5     5          81.2                 3.60
## 6     6          81.0                 3.42
## 7     7          80.9                 3.65
```

Challenge 1:

Find the average strength at the last time point in each group (hint, use filter, group_by and summarize)

```r
tidy_df %>%
  filter(time == 7) %>%
  group_by(treatment) %>%
  summarize(mean_strength = mean(y, na.rm=TRUE))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 2
##   treatment mean_strength
##       <dbl>         <dbl>
## 1         1          79.3
## 2         2          81.1
## 3         3          82.5
```

A note on data.table vs data.frame and dplyr

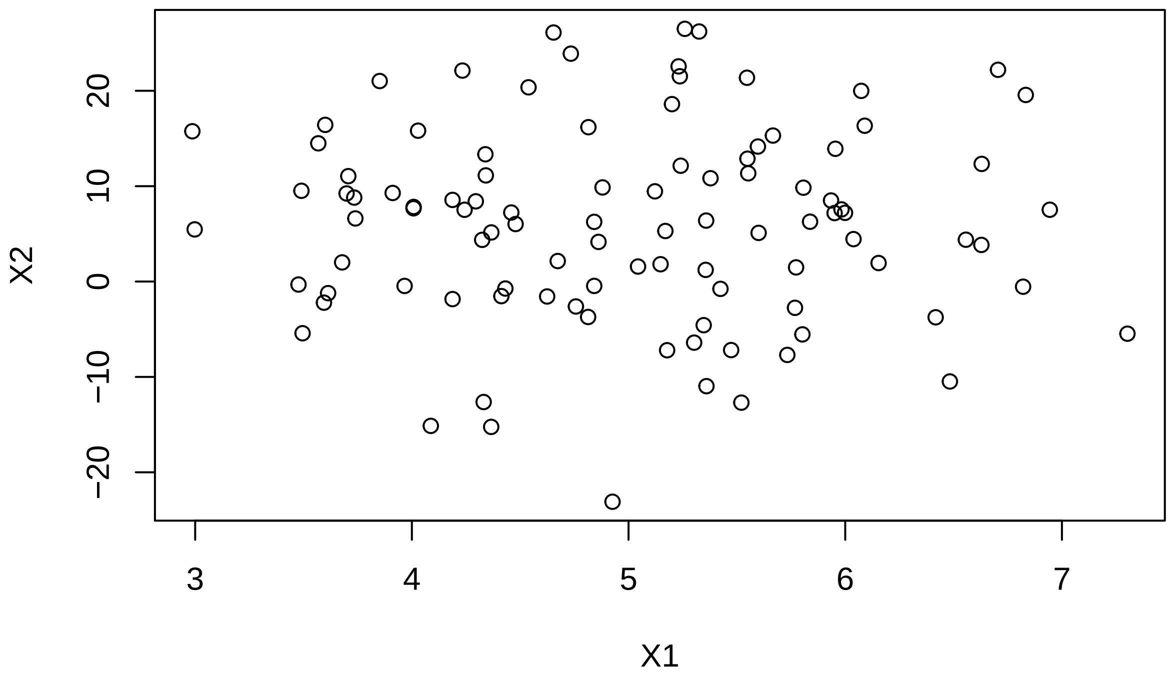
The *data.table* package is a package that is useful for manipulating large datasets. You can store data as a *data.table* object, and manipulate it using syntax similar to base R syntax, and using additional functions including *subset*, and read/write functions like *fread* and *fwrite*. The *data.table* objects are more-or-less enhanced *data.frame* objects.

dplyr is a package in the *tidyverse* collection of packages that contains the functions I emphasized in labs 1-3, such as *filter*, *select*, and *rename*. The functions in *dplyr* are really useful for operating on data stored in *data.frame* objects.

Generating Random Numbers

Sometimes we want to generate random numbers from probability distributions. This will be useful later when we assess the performance of estimators on a sample from a known data-generating process.

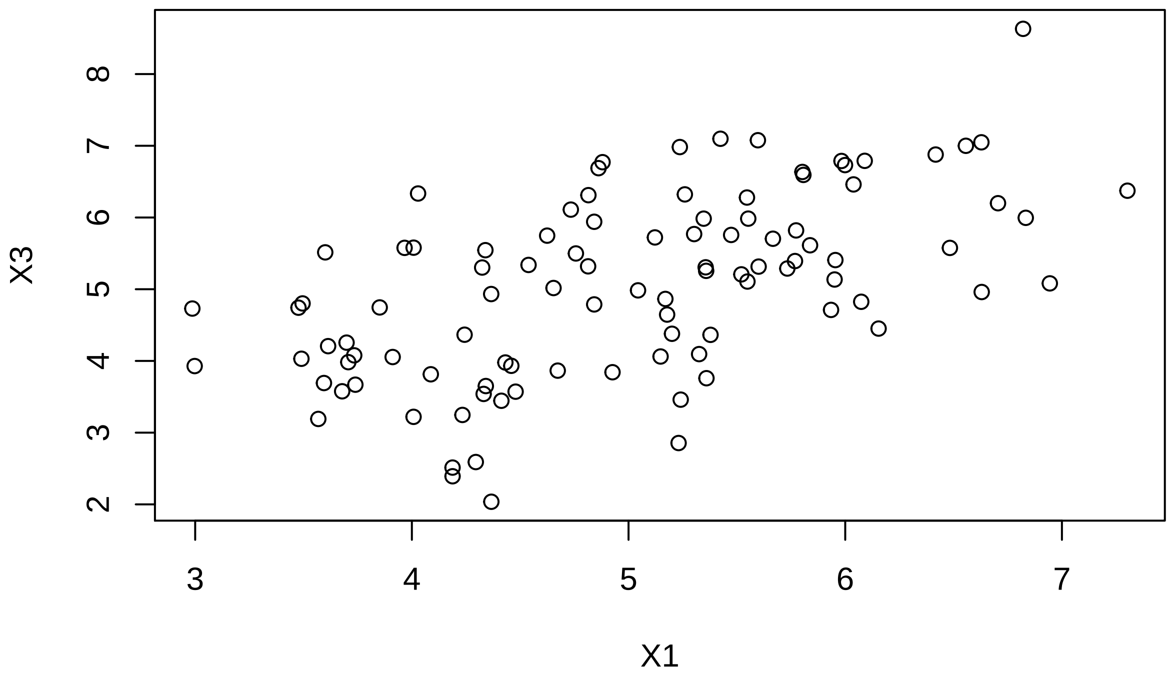
```
library("tidyverse")
#X1 and X2 are independently generated normal random variables
X1 <- rnorm(n = 100, mean = 5, sd = 1)
X2 <- rnorm(n = 100, mean = 5, sd = 10)
#And they don't appear to be related
plot(x = X1, y = X2)
```



```
cor(X1, X2)
```

```
## [1] -0.008582714
```

```
#Here X3 depends on X1
X3 <- rnorm(n = 100, mean = X1, sd = 1)
#And from a visual inspection they appear to be associated
plot(x = X1, y = X3)
```



```
cor(X1, X3)
```

```
## [1] 0.5692717
```

```
#We can put these three random variables together in a data frame
rand_nums = tibble(X1, X2, X3) #Similar to data.frame() function
head(rand_nums)
```

```
## # A tibble: 6 x 3
##      X1     X2    X3
##   <dbl>  <dbl> <dbl>
## 1  5.33 26.2    4.10
## 2  4.84 -0.458  4.79
## 3  3.91  9.29   4.05
## 4  4.34 11.1    3.65
## 5  6.82 -0.541  8.63
## 6  4.65 26.1    5.02
```

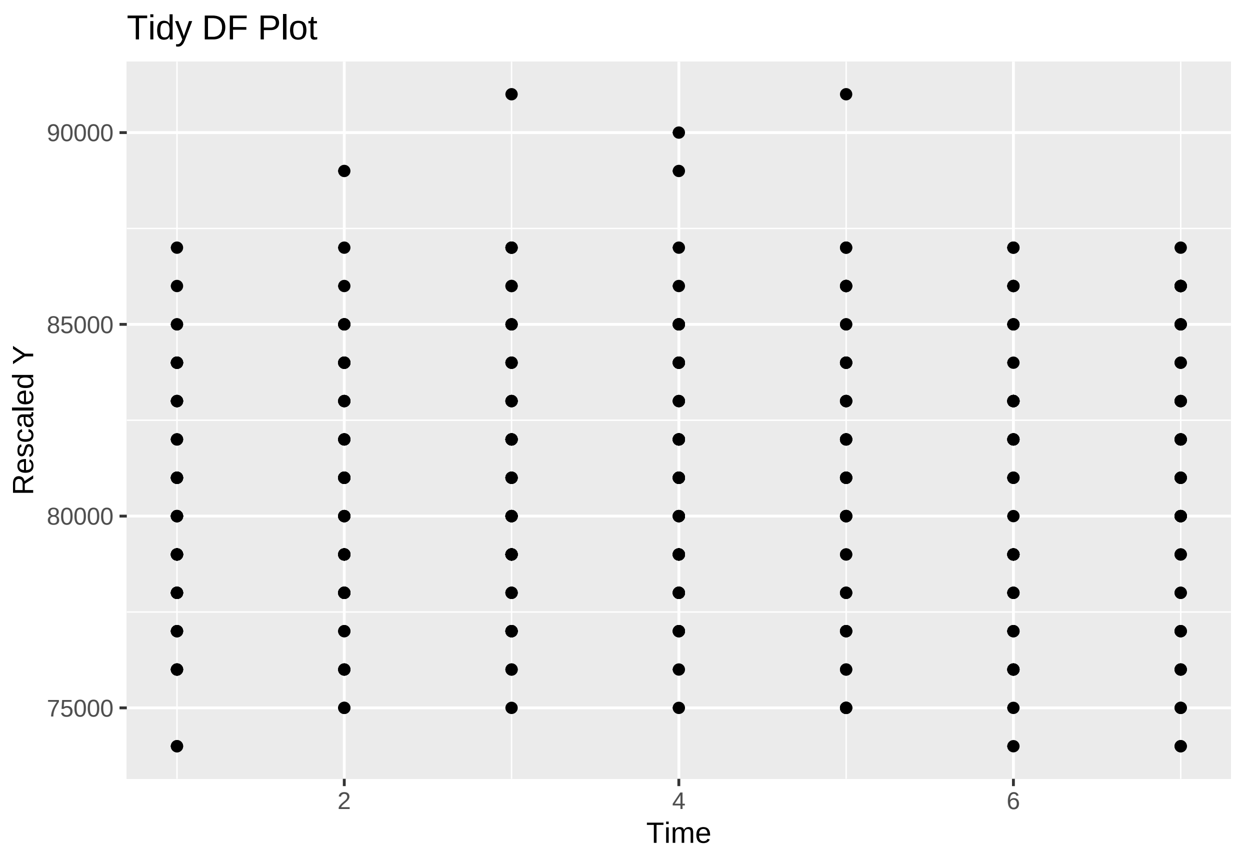
```
#We could now work with these randomly generated numbers in this data frame, e.g., to fit a model or es
```

ggplot

Aside from dplyr and tidyr, another pillar of the tidyverse is ggplot. R has base plotting functions that are good for visualizing data quickly (as we saw above), but ggplot offers a ot more customization and aesthetically pleasing options.

```r
tidy_df %>%
  ggplot() +
  geom_point(aes(x = time, y = rescale_y)) +
  ggtitle("Tidy DF Plot") +
  xlab("Time") +
  ylab("Rescaled Y")
```

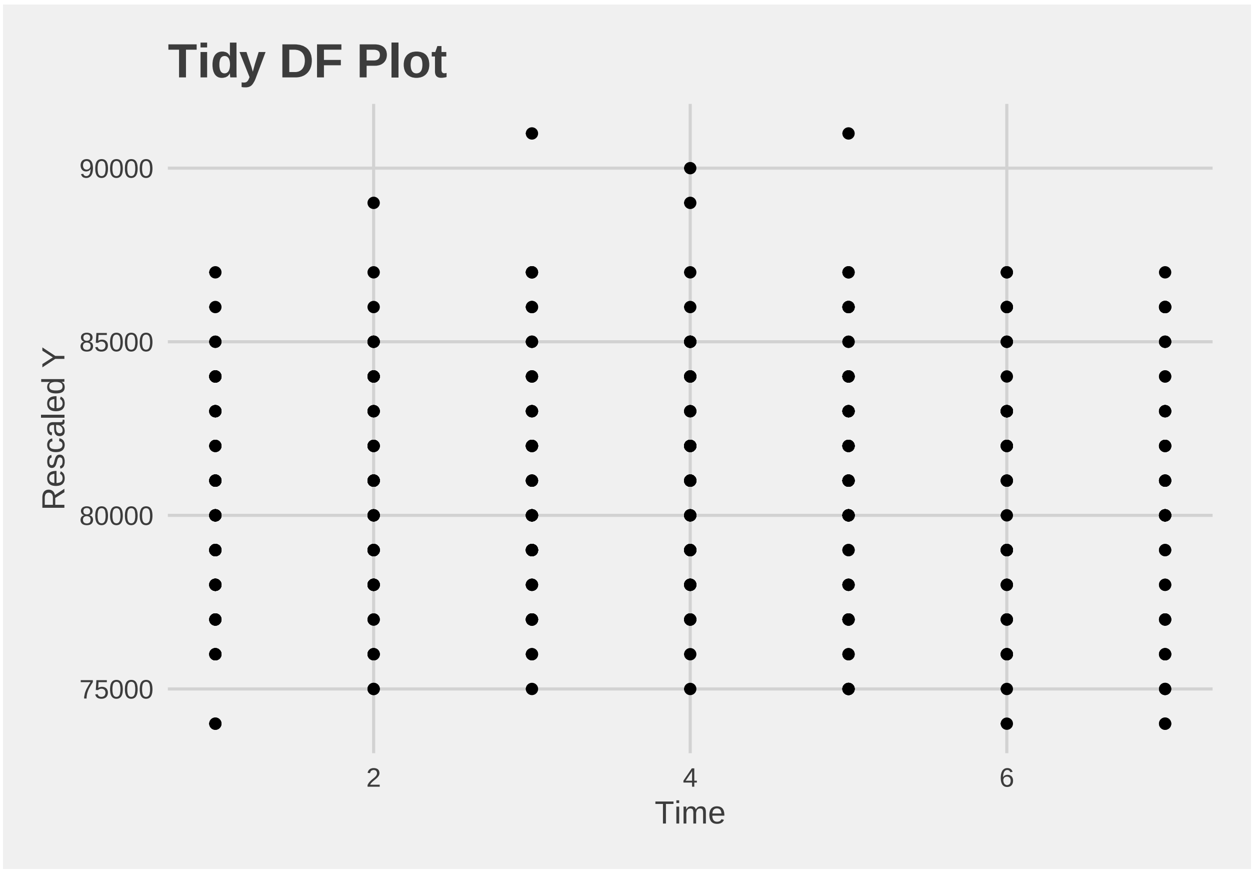
```
## Warning: Removed 29 rows containing missing values (geom_point).
```



I also like the `ggthemes` library that lets you style your ggplot. FOr example, you might use the fivethirtyeight theme:

```r
tidy_df %>%
  ggplot() +
  geom_point(aes(x = time, y = rescale_y)) +
  ggthemes::theme_fivethirtyeight() +
  theme(axis.title = element_text()) +
  ggtitle("Tidy DF Plot") +
  xlab("Time") +
  ylab("Rescaled Y")
```

```
## Warning: Removed 29 rows containing missing values (geom_point).
```

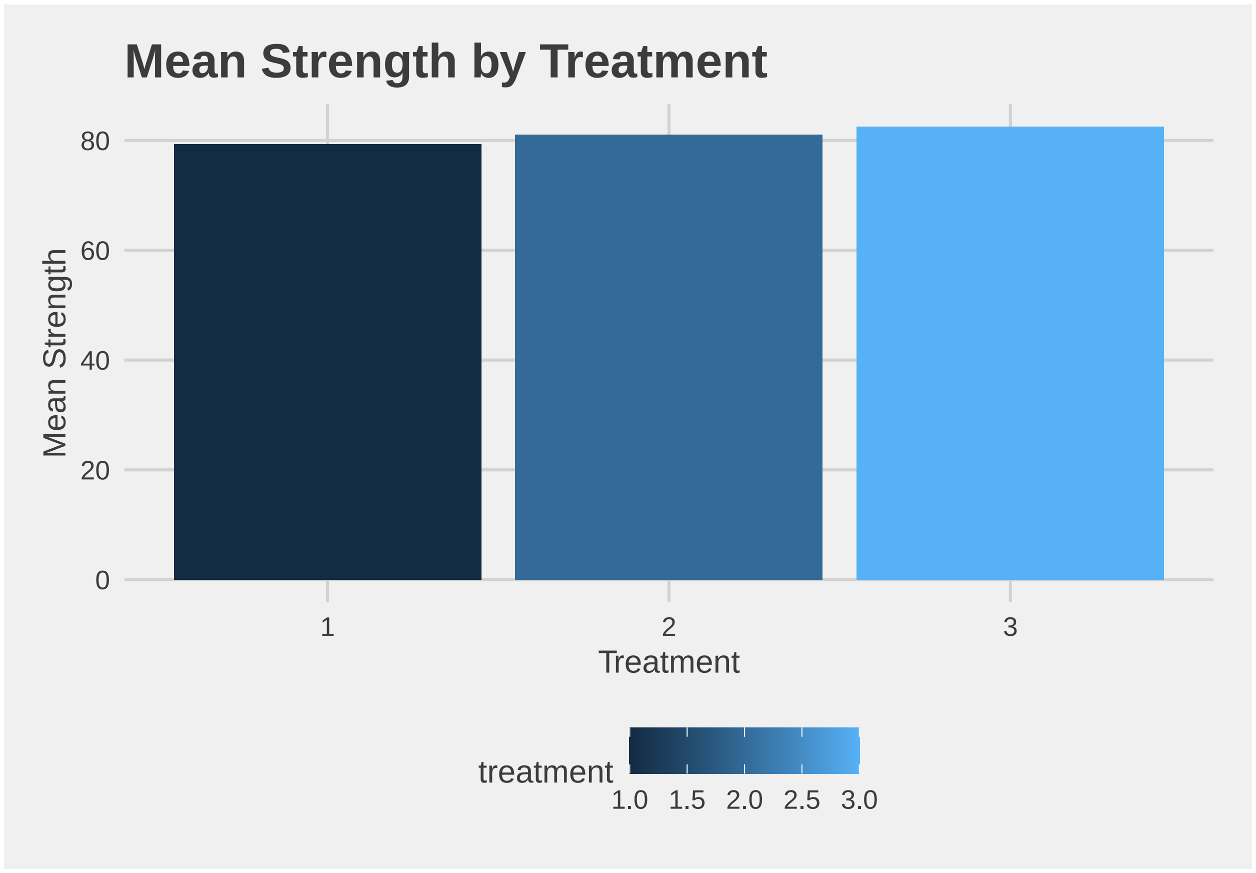


Challenge 2:

Try making some of your own plots! Plot the mean mean_strength by treatment based on the dataframe we created earlier in challenge 1, and then another plot of your choice! Hint: You may want to look up the documentation for a method like geom_bar() to plot this one.

```
tidy_df %>%
  filter(time == 7) %>%
  group_by(treatment) %>%
  summarize(mean_strength = mean(y, na.rm=TRUE)) %>%
  ggplot() +
  geom_bar(aes(x = treatment, y = mean_strength, fill = treatment),
           stat = 'identity') +
  ggthemes::theme_fivethirtyeight() +
  theme(axis.title = element_text()) +
  ggtitle("Mean Strength by Treatment") +
  xlab("Treatment") +
  ylab("Mean Strength")
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```



R Markdown

The Header

The first lines of the .Rmd file include information about the file that will be used to render a title. There are extensions to include things like page numbers, headers, footers, bibliographies, and variations on the table of contents.

Basics

In this white space you can write in English; the compiler will not interpret things in white space as `R` code.

```
# Anything in gray space is interpreted as R code
#(which is why these words are commented out using the # symbol).
# Use ```{r} to start gray space and ``` to end it.
# The gray space is called a code chunk
2+7
```

```
## [1] 9
```

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

You can also make lists

- One
- Two
- Three

And write equations in LaTeX:

$Y = \alpha + \beta_1 X + \gamma_1 D$

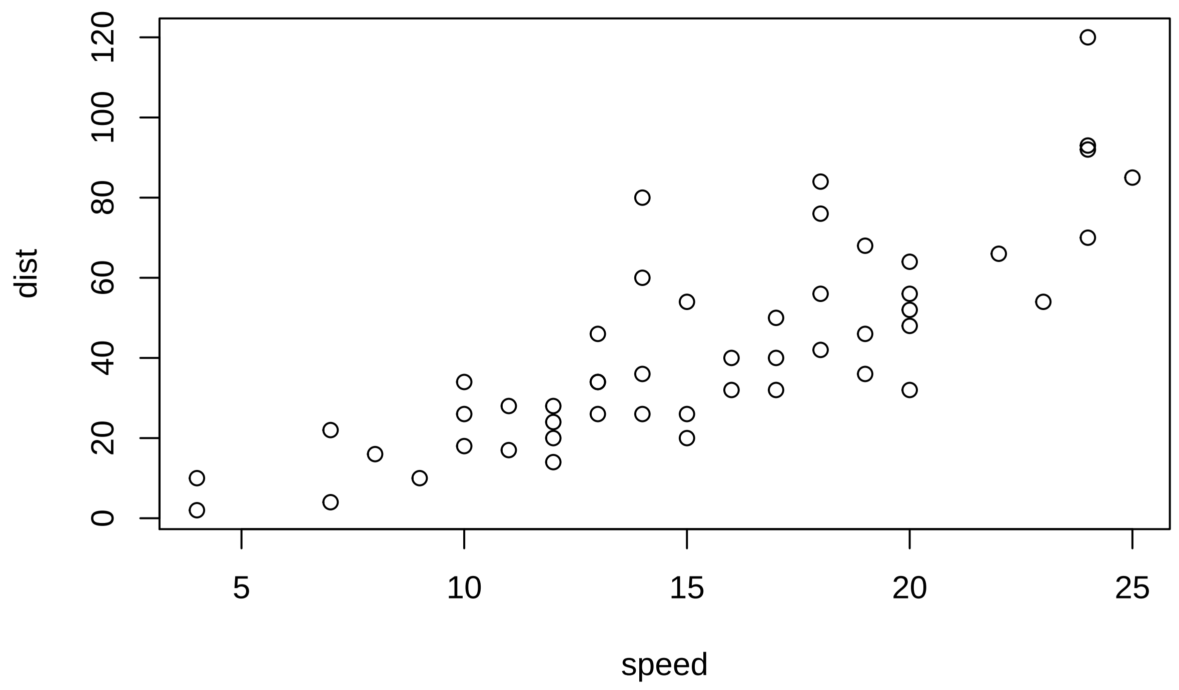
Take a look at this RMD cheatsheet for other useful formatting information:

https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf

This is an R Markdown Notebook. When you execute code within the notebook, the results appear beneath the code.

Try executing this chunk by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Cmd+Shift+Enter*.

```
# Note that the "cars" dataset is already loaded into R
# and only has two columns, one for speed and one for distance (dist).
plot(cars)
```



Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by pressing *Cmd+Option+I*.

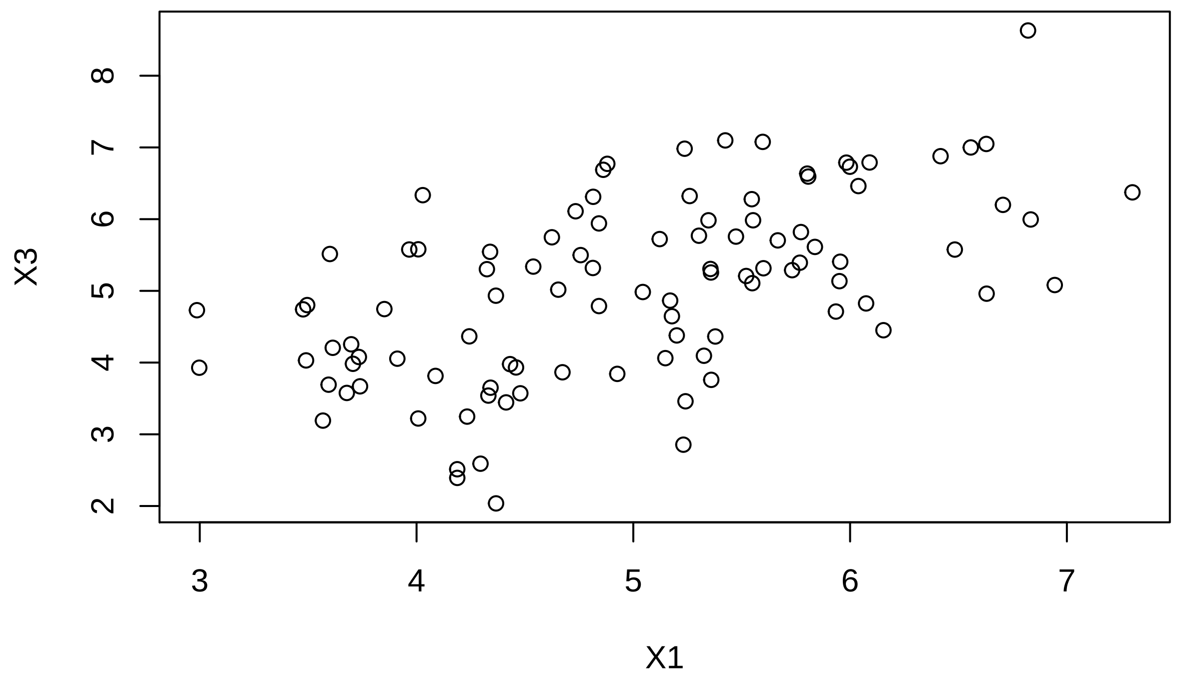
When you save the notebook, an HTML file containing the code and output will be saved alongside it (click the *Preview* button or press *Cmd+Shift+K* to preview the HTML file).

The preview shows you a rendered HTML copy of the contents of the editor. Consequently, unlike *Knit*, *Preview* does not run any R code chunks. Instead, the output of the chunk when it was last run in the editor is displayed.

Making PDFs using R Markdown

We can also create PDF files from .Rmd files by changing the output option in the header from "html_notebook" to "pdf_document".

We can suppress code, output, warnings, messages, or errors using chunk options, e.g. echo=FALSE will hide this code chunk.



Set a chunk to not evaluate using eval=FALSE.

```r
plot(x = X1, y = X3)
```

There are similar options to suppress messages, warnings, and errors, which may be useful for homework.

How to export from RStudio Cloud and save onto your own computer

If you're working on RStudio Cloud and would like to download your files:

1. Check the file you'd like to export and the click More > Export in the File viewer. It will download to your computer's downloads folder.
2. You may want to Export slides as a PDF or MS Word document. To do that, you first need to change "slidy_presentation" to "pdf_document" or "word_document" in the file header (line 5 of the file, under output:).
3. Word documents then automatically download when you Knit them. PDF documents can be exported from the File viewer by following step 1.

Latex symbols and useful math equations for Causal Inference

Insert inline math symbols between dollar signs, e.g. Y_{ij} or $P(Y = 1)$.

We can also insert chunks of math spanning multiple lines between double dollar signs:

$$\mathbb{E}[Y|X = x]\mathbb{E}[Y|X]$$

Use two dollar signs to insert sections of LaTeX (there are other options to insert these chunks as well). Use underscore (_) for subscript, carrot (^) for superscript, and a backslash to initiate special characters (greek letters, curly braces, etc). When the subscript or superscript is more than 1 character it is necessary to surround the content with curly braces.

For example, a simple MSM:

$$E_{U,X}(Y_a) = m(a|\beta)$$

You can also align equations within the dollar signs using the "aligned" environment and an ampersand, and double backslashes for line breaks.

$$\begin{aligned} E_{U,X}(Y_a) &= m(a|\beta) \\ E_{U,X}(Y_a) &= \beta_0 + \beta_1 a \end{aligned}$$

We can think about an MSM as a projection of a true causal relationship onto some curve. Some important LaTeX tools for this are mathop, brackets, summation, and mathcal (for script A).

$$\beta(P_{U,X}|m) \equiv \mathop{argmin}_{\beta} E_{U,X}[\sum_{a \in \mathcal{A}} (Y_a - m(a|\beta))^2]$$

We can also include text within math equations. If we have many endogenous variables in W we might want to write something like:

$$\begin{aligned} W &= (\text{age}, \text{sex}, \text{risk factors}) \\ A &= (\text{treatment}) \\ Y &= (\text{death within 5 years}) \end{aligned}$$

Sometimes we need to render equations using set notation, such as $\in$, or using curly brackets.

$$\begin{aligned} a &\in \mathcal{A} \\ \mathcal{A} &= \{0, 1\} \end{aligned}$$

We can use the mathcal operator to render symbols for the Causal Model $\mathcal{M}^F$ or the Causal Model augmented with additional assumptions required for identifiability $\mathcal{M}^{F*}$.

To denote that the observed data is drawn according to the observed data distribution P_0:

$$O = (W, A, Y) \sim P_0$$

When we get to the estimation part of the course we may need to use a hat above symbol to indicate it is an estimate. This is similar to using "X bar" to represent the sample mean.

$$\bar{X}$$

$$\hat{\Psi}$$

Target Causal Parameters, such as the Average Treatment Effect often denoted using Ψ^F, while statistical parameters are denoted Ψ. The connection between these will be made during the "Idenfitication" part of the course.

$$\Psi^F(P_{U,X}) = E_{U,X}[Y_1 - Y_0]$$

To write that two random variables are independent or conditionally independent, use the perp operator, as in: $X \perp Z$ to indicate that the random variable is indepent of Z.

Review of Notation

Observed Data - Review

Suppose we have m independent individuals indexed by $i = 1...m$, each with observations $j = 1...n_i$. For each individual we measure 3 covariates.

$$O_i = (Y_{ij}, X_{ij1}, X_{ij2}, X_{ij3}, T_{ij}) \ j = 1, ..., n_i$$
$$O_i \sim^{iid} P_O$$

Alternatively, we can collect the covariate information at each timepoint into a row vector $\mathbf{X}_{ij} = (X_{ij1}, X_{ij2}, X_{ij3}, T_{ij})$. Combining across $j = 1, ...n_i$ gives a matrix..

$$O_i = (Y_{ij}, \mathbf{X}_{ij}) \ j = 1, ..., n_i$$
$$O_i \sim^{iid} P_O$$

Variance and Covariance - Review

For a random variable $Y \sim P_Y$, the variance is defined as:

$$Var(Y) = E[(Y - E[Y])^2] = E[Y^2] - (E[Y])^2$$

And for two random variables $Y \sim P_Y$ and $X \sim P_X$, the covariance is defined as:

$$Cov(X, Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$$

Note that this implies $Cov(Y, Y) = Var(Y)$.

For a random vector $Z = (Z_1, Z_2, Z_3)^T$, the Covariance Matrix is defined as:

$$\begin{bmatrix} Var(Z_1) & Cov(Z_1, Z_2) & Cov(Z_1, Z_3) \\ Cov(Z_2, Z_1) & Var(Z_2, Z_2) & Cov(Z_2, Z_3) \\ Cov(Z_3, Z_1) & Cov(Z_3, Z_2) & Var(Z_3) \end{bmatrix}$$

Notation Example Problems

Notation Exercise 1

We have $m = 50$ units of observation, $n_i = 12$ observations over time per unit, with the same timepoints used for each unit. At each time point one outcome and two covariates are measured. And there is one fixed covariate (i.e., constant over time) for each unit.

Question: Describe this data structure using the notation.

Solution:

$$O_i = (Y_{ij}, X_{ij1}, X_{ij2}, Z_i, T_{ij}) \ j = 1, ..., 12$$

We could also express this as a matrix. Note that Z_i could be denoted X_{ij3} in conjunction other notation indicating that it is constant over time. (Notation isn't perfect).

Notation Exercise 2

Suppose observed data consist of repeated over time (3 different times) observations of independent subjects, where at each time, 2 observations (measurements) are taken of an outcome and covariates, including the time of observation.

We could express this using i for individuals, j for time points, and k for measurements within a timepoint.

$$\begin{aligned} O_i &= (Y_{ijk}, \mathbf{X}_{ijk}, T_{ijk}) \ j = 1, ..., 3 \ k \in (1, 2) \\ \mathbf{X}_{ijk} &\in \mathbb{R}_{2n_i \times p} \end{aligned}$$

Alternatively, we can avoid a separate index for k and have $j = 1, ..., 6$, where the index j corresponds to the six measurements (3 time points, 2 measurements at each timepoint). The importance and use of this distinction could depend on whether we care about the order of the two observations at each timepoint.

$$\begin{aligned} O_i &= (Y_{ij}, \mathbf{X}_{ij}, T_{ij}) \ j = 1, ..., 6 \\ \mathbf{X}_{ij} &\in \mathbb{R}_{n_i \times p} \end{aligned}$$

Provide Notation for the Following Additional Variance-Covariance structures for the outcome conditional on covariates:

1. All measurements are uncorrelated. All individuals have the same variance.

Solutions:

1. We can use a single number σ^2 to describe the variance of an individual outcome. Covariance between different observations is 0. (In other words, to covariance between any pair of different outcome measurements (conditional on the covariates) is zero).

$$\begin{aligned} Var(Y_{ijk}|\mathbf{X}_{ijk}, T_{ijk}) &= \sigma^2 \\ Cov(e_{ijk}, e_{abc}) &= 0 \ (i \neq a \ or \ j \neq b \ or \ k \neq c) \end{aligned}$$

Alternatively,

$$\begin{aligned} Var(Y_{ij}|\mathbf{X}_{ij}, T_{ij}) &= \sigma^2 \\ Cov(e_{ij}, e_{ab}) &= 0 \ (i \neq a \ or \ j \neq b) \end{aligned}$$